```
Getting Started with C++
------------------------

Here is your very first C++ program. It displays a message on the screen:

// Author: Ted Obuchowicz
// Jan. 7, 2001
// hello world program



#include <iostream>
#include <string>
using namespace std;

int main()
{

 cout << "Hello World" << endl;
 return 0;
}



We compile this program using the g++ compiler installed on our UNIX systems
as:

ted@flash Programs 5:05pm >g++ -o hello hello.C

To run this program, we simply type hello from the command prompt
(if your path is setup differently you may have to type ./hello)

ted@flash Programs 5:06pm >hello
Hello World
ted@flash Programs 5:07pm >


The program runs and displays the rather bland message and control returns
back to your operating system.

The first three lines of the program are COMMENTS. The compiler ignores anything
following the // (ignores everything following the // up to the end of the line)

// Author: Ted Obuchowicz
// Jan. 7, 2001
// hello world program


The next three lines:

#include <iostream>
#include <string>
using namespace std;

are known as COMPILER DIRECTIVES. They are instructions telling to compiler wher
e
to look for certain definitions of often used library routines such as the cout
(which is used to perform output ).
```

The remaining lines for the body of the main function:

```
int main()
{

 cout << "Hello World" << endl;
 return 0;
}
```

In C++, there can only be one function called main. The main function defines
the starting point of the program. When a program runs, main is the first functi
on
to begin execution. The

```
{


}
```

are used to delimit the beginning and the end of the function called main.
The int in the line

```
int main()
```

refers to the return value that function main will return upon completion.
The line

```
return 0
```

is used to return a value of 0 to the operating system when the program terminat
es.
Historically, a return value of 0 has been used to indicate success.

The line:

```
cout << "Hello World" << endl;
```

sends the string "Hello World' to the display device. THe << is called
the INSERTION OPERATOR. The first << inserts the string to the output stream
cout (which is connected to the display). The second insertion after sends the
MANIPULATOR endl to the output stream. The endl manipulator is used to print a
new line. This means any that any further output would commence on the next
line.

Here is another version of the program which makes use of a function to
print a message:

```
// Author: Ted Obuchowicz
// Jan. 7, 2001
// example program illustrating use of function calls
// some of the programs do not have the
// using namespace std; compiler directive
// but all your programs should have it since
// the new version of the g++ compiler requires it
// and these notes were written for an earlier version
```

```
// which did not require it...   Jan. 10, 2008

#include <iostream>
#include <string>

void print_message(char *s)
{
  // this is an example of a function
  // it receives a single argument which is a C++ string
  // and prints this argument to the screen

  cout << s << endl;

}

int main()
{
  print_message("Hello Coen 243");
  print_message("Goodbye Coen 243");
  return 0;
}
```

This program contains a definition of a function called
print_message. The function receives a single string argument
and sends it to the display device.

The main() program simply INVOKES this function two times in a row
to print two different messages.

It is necessary to define the function print_message FIRST, since the
main() function refers to it. If we were to switch the order of the
definitions around, the program would NOT COMPILE.

A bit more on comments

C++ has two forms of comments:

- the double slash style as in

```
// this is a comment

int i ; // everything appearing // after is a comment //
```

- the old C style comment /*   */ as in

```
int i ; /* stuff in between these special delimiters is treated as a comment */

/*  ********************************************** */
/*                                                 */
/*  can be used to build pretty comment boxes      */
/*  which grab the readers attention               */
/*                                                 */
/*  ********************************************** */
```

Comment pairs do not nest:

```
/*

float pi = 3.14 ; /* pi is used in math */
float x, r ;

x = pi * r* r ;

*/


The compiler will not recognize pi in the line x = pi * r* r
since its declaration has been commented out. Furthermore,
the last


*/

will cause a compile time error.

IDENTIFIERS
-----------

An identifier is a sequence of LETTERS, DIGITS, and the underscore (_) character
.
An identifier CANNOT begin with a digit.  C++ is case-sensitive which means uppe
r and
lower case letters are treated as being different.

Here are some legal C++ identifiers:

n
count
num_of_letters;
buff_size

Here are some ILLEGAL identifiers :

for  // cannot use a KEYWORD as an identifier
3q   // an identifier cannot start with a digit
.count // invalid character .

Don't use the _ as the first character of an identifier. Some compilers
use these to name special functions and it may cause problems during linking
of your program.

LITERALS
--------

Lierals are constant values which appear in a program. Every literal has an
associated type.  Literals are used to assign a given value to a given variable
as in

int x ;    // declare an integer variable called x

x = 5 ;

The 5 in the assignment statement x = 5 is an example of what is known
```

```
as an integer literal.

C++ has several rules which define how one expresses a literal:

Integer Literals
--------------

C++ allows for decimal, octal (base 8), and hexadecimal (base 16) whole
numbers:

24    // a decimal integer literal
030   // same decimal value expressed as an octal number
0x18  // 24 expressed as a hexadecimal number
0X18  // 24 expressed as a hexadecimal number


The following program will display the number 24 , 4 times:

// Author: Ted Obuchowicz
// Jan. 28, 2002
// example program illustrating use of
// some integer literal constants with different bases


#include <iostream>
#include <string>

using namespace std;



int main()
{

cout << 24 << " " << 030 << " " << 0x18 <<  "  " << 0X18 << endl;

return 0;
}

The program output is:

ted@flash Programs 6:45pm >literals
24 24 24  24
ted@flash Programs 6:45pm >


We can conclude from the above that:

*  prepending a 0 (zero) to an integer literal constant will cause it
   to be interpreted as an OCTAL NUMBER.

*  prepending either 0x or 0X to an integer literal will cause it
   to be interpreted as a HEXADECIMAL number.

Integer literal constants are treated as signed values of type int.
The modifiers long and unsigned may be used when specifyinh literal integer
constants by appending L (or l), and U (or u) respectively to the number:

5U, 5u
```

```
5L, 5l,

5Ul, 5uL

Floating-point integer literal constants
---------------------------------------

Floating point literals may be written in either common decimal
notation or SCIENTIFIC NOTATION with the exponent written using
e or E.

The default type is double precision, if desired single precision can be
indicated with either F or f, EXTENDED precisions is indicated with a L
or l:

Some examples of default double precision floating point literals:

2.
2.0
1.0E-3
3E1


The following is a single precision floating point number:

3.1415926F

Here is a extended precision number: 1.0L

Literal Character Constants
--------------------------

* Printable literal character constants are written by enclosing
  the character within single quotation marks:

'a'   // lower case a is ASCII 97
'A'   // upper case A is ASCII 65
'5'   // character 5 is ASCII 53

* Certain non-printable characters, single and double quotes, and
  the backslash characters can be represented using SPECIAL ESCAPE
  SEQUENCES to define these character literals. The escape sequence
  is enclose in single quotes:

'\n'  // character for printing a new line
'\t'  // character for printing a tab
'\''  // single quote literal
'\"'  // double quote literal
'\a'  // escape sequence to ring the bell


The following program, when run, will cause the computer speaker
to make a pleasant beeping sound:


// Author: Ted Obuchowicz
// Jan. 28, 2002
// example program illustrating use of
```

```
// escape sequence to ring the computer's bell


#include <iostream>
#include <string>

using namespace std;



int main()
{


cout << '\a' << endl;


return 0;
}


STRING LITERAL C// Author: Ted Obuchowicz
// Jan. 28, 2002
// example program illustrating use of
// escape sequence to ring the computer's bell


#include <iostream>
#include <string>

using namespace std;



int main()
{


cout << '\a' << endl;


return 0;
}


STRING LITERAL CONSTANTS
-----------------------

A string literal constant is composed of zero or more characters
ENCLOSED IN DOUBLE QUOTES. Non-printable characters are represented
by enclosing their escape sequences within the string.  The NULL
character '\0' is used to indicate the end of the string (as it
is stored in the computers main memory):

"5"  // the string consisitng of the two characters '5' and '\0'

"a\tb\n" // the string consisting of 5 bytes stored in memory:
```

```
---
'a'   some location in memory 1 byte wide
---
'\t'  next location
---
'b'   the location after that
---
'\n'  etc.
---
'\0'
---



When printed the above string would print:

a       b

(an a follwed by a number of blank spaces as specified by the horizontal
tab setting of your terminal, a b, follwed by a new line)

C++ BUILT-IN DATA TYPES
----------------------

The C++ language contains several native, or built-in data types which
are used to represent INTEGERS, floating point and character values.

The INTEGRAL built in data types are:

char  : this type can be used for individual characters or small integers.
        A char is stored as a single BYTE in  main memory

   (Here is a lyric from a Roger Waters song :

    "Memory is a stranger,
    History's for fools,
    The president is a fool
    Why do I have to keep reading these technical manuals"

I'll give a dollar to the first person who correctly identifies the song title
and the CD it comes from... reply by email to ted@ece.concordia.ca)

int : used to represent integer values. Typically stored in 2 bytes

short, long : some more integer types whose sizes depend upon the operating
              system and hardware

The INTEGRAL TYPES may either be SIGNED or UNSIGNED. The difference is the
interpretation of the left-most bit

example: unsigned char     00000000   (0)
                           00000001   (1)
                           00000010   (2)
                           ...
                           11111110
                           11111111   (255)

   In unsigned char, all 8 bits are used to represent the magnitude of the numbe
```

r.

```
example: signed char      00000000 (+0)
                          00000001 (+1)
                          ....
                          01111111 (+127)
                          10000000 (-128)
                          10000001 (-127)
                          ...
                          11111110 (-2)
                          11111111 (-1)
```

In the binary representation for an unsigned char, a 0 in the left-most bit postion designates a positive number, a 1 represents a negative number. You don't have to concern yourselves with the meaning of all those 1s and 0s in the numbers, that is how the number is stored inside the computers memory. All you have to know is that the range of singed chars is less than that of unsigned char because half of the available bit patterns are used to represent negative values and the remaining half are for the postive numbers.

FLOATING POINT NUMBERS

There are three data types used to represent floating point numbers in C++:

float
double
long double

Again, the differences are the amount of memory which is used to store a value of a particular type and the range of allowed values that can be assigned to a particular data type. Again, this is operating and hardware dependent.

Using the sizeof library function

There is a built-in function called sizeof() which is used to determine the number of bytes which a data type occupies in main memory. Its use is illustrated in the following program:

```
// Author: Ted Obuchowicz
// Jan. 7, 2001
// example program  illustrating use of #define and the sizeof function
// to determine amount of memory storage required to hold some basic C++
// data types



#define NL      <<"\n" // this is the escape sequence for a newline character
#define TAB     <<"\t" // this is the escape sequence for a tab
```

```
#include <iostream>

int main()
{
        cout << sizeof(char     )       << "\tis sizeof char    " NL;
        cout << sizeof(short    )       << "\tis sizeof short   " NL;
        cout << sizeof(int      )       << "\tis sizeof int     " NL;
        cout << sizeof(long     )       << "\tis sizeof long    " NL;

// now, let's determine the size of the "unsigned" versions of the above
// integer data types... the sizes should be the same as the "signed"
// versions

        cout << sizeof( unsigned char     )       << "\tis sizeof unsigned char
   " NL;
        cout << sizeof( unsigned short    )       << "\tis sizeof unsigned short
   " NL;
        cout << sizeof( unsigned int      )       << "\tis sizeof unsigned int
   " NL;
        cout << sizeof( unsigned long     )       << "\tis sizeof unsigned long
   " NL;

// find out the size of the available floating point data types

        cout << sizeof(float    )       << "\tis sizeof float   " NL;
        cout << sizeof(double   )       << "\tis sizeof double  " NL;
        cout << sizeof(long double)     << "\tis sizeof long double " NL;

// find out the size of a void pointer, which can point to any data type

        cout << sizeof(void *  )        << "\tis sizeof void *      " NL;
        return 0;
}
```

When we run this program on our UNIX systems we obtain:

```
ted@flash Programs 7:32pm >sizeof
1       is sizeof char
2       is sizeof short
4       is sizeof int
4       is sizeof long
1       is sizeof unsigned char
2       is sizeof unsigned short
4       is sizeof unsigned int
4       is sizeof unsigned long
4       is sizeof float
8       is sizeof double
16      is sizeof long double
4       is sizeof void *
ted@flash Programs 7:33pm >
```

Range of built-in data types

Every built-in data type has limits on the range of legal
values that may be assigned to a variable of a certain data

```
type.  The range is dependent upon the size of the data type.
On our UNIX system there is a special file called limits.h
(found in the directory /usr/include ) which give the various limits
of values which can be assigned.  The file limits.h is 5 pages
long, so I will not include it in here , the more curious of you are
welcome to print it out and examine it in detail .

These are the ranges for the integral data types:

unsigned char: 0 to 255
char  : -128 to 127

unsigned short: 0 to 65535
short : -32768 to  32767

unsigned int : 0 to 4294967295
int  : -2147483648 to 2147483647

unsigned long: 0 to 4294967295
long: -2147483648 to 2147483647

Range of floating point data types:
----------------------------------

On most computer systems, the float and double data types are implemented
using the IEEE 754 floating point representation. There is a 32 bit version
and a 64 bit version.  Floating point numbers are stored in a computer's
memory using a format of the type:

   sign   exponent mantissa

  the base is usually 2. The value of the magnitude of the number
   is given by
                   exponent
     0.mantissa x 2

   The value of the sign bit determines whether the number is positive or negati
ve.
   The exponent is stored as a integer (ie it can either be positive or negative
).

On our system, these are the values for the float and double data types:

1.175494351E-38F  /* min decimal value of a "float" */
3.402823466E+38F /* max decimal value of a "float" */

2.2250738585072014E-308 /* min decimal value of a */
                        /* "double" */

1.7976931348623157E+308 /* max decimal value of a */
                        /* "double" */

Note that the double data type has more significant digits of precision
as well as a larger range of possible exponent values. limits.h also
defines these:

#define DBL_DIG         15      /* digits of precision of a "double" */
#define FLT_DIG         6       /* digits of precision of a "float" */
```

```
CONSTANTS
---------

A constant is something whose value cannot change.  C++ uses the keyword
const to declare constants.

In the old days (before the definition of the ANSI-C standard), programmers
had to use to PREPROCESSOR DIRECTIVE #define to create a symblic named constant
as in:

// Author: Ted Obuchowicz
// Jan. 28 , 2002
// example program illustrating use of
// of old pre ANSI C #define methos of declaring a
// symbolic constant


#include <iostream>
#include <string>

#define pi 3.1415926

using namespace std;



int main()
{

float r = 2.0;
float area = pi * r * r;
cout << "area of circle with radius 2 is " << area << endl;

return 0;
}

The output is:

ted@flash Programs 8:08pm >constant
area of circle with radius 2 is 12.5664
ted@flash Programs 8:08pm >


Using this method, the preprocessor simply substituted the literal
3.1415926 wherever it saw the identifier pi in the program source code.

The disadvantages of using this type of approach are:

* No memory is allocated for the symbolic macro names defined in a #define
  directive.  Consequently, these named constants are OUTSIDE THE SCOPE OF
  debuggers, and no TYPE CHECKING is performed on the named identifiers.


C++ uses another method to declare constants:

// Author: Ted Obuchowicz
// Jan. 28 , 2002
// example program illustrating use of
```

```
// of  better C++ way of declaring a
// symbolic constant


#include <iostream>
#include <string>


using namespace std;



int main()
{

const float pi = 3.1415926;
float r = 2.0;
float area = pi * r * r;
cout << "area of circle with radius 2 is " << area << endl;

return 0;
}
```

In general, the methos C++ uses for constants is:

```
const   data_type   identifier  initialization;
```

Attempting to define a constant without giving it an initial
value will result in a compile time error:

```
const float pi ;
```

The use of the keyword const allows the creation of an object in main
memory whose CONTENTS CANNOT BE CHANGED.

The benefits of using this type of constant declaration are
that the constant becomes an actual program variable with a
specific data type. As such, it may be accessed by debugging
tools and type-checking may be performed.

REFERENCES

C++ allows for the declaration of reference variables.
Essentially, reference variables allow a variable to have
more than one name associated with it.

```
// Author: Ted Obuchowicz
// Jan. 28, 2002
// example program illustrating use of
// reference variables



#include <iostream>
#include <string>

using namespace std;
```

```
int main()
{

int keith ; // declare an integer variable called keith

int & good_guitar_player = keith ; // declare and initialize
                                   // a reference to keith

int & former_heroin_addict = keith ; // declare and initialize
                                     // another reference to keith

keith = 5;
cout << keith << " "  << good_guitar_player << " " << former_heroin_addict << en
dl;

former_heroin_addict = 7;

cout << keith << " "  << good_guitar_player << " " << former_heroin_addict << en
dl;

return 0;
}
```

The program output is:

```
ted@flash Programs 8:32pm >reference
5 5 5
7 7 7
ted@flash Programs 8:32pm >
```

In this program, we have declared an integer variable called keith and
two references to keith called good_guitar_player and former_heroin_addict.
There is one location in main memory which is used to hold this integer
value. The location in main memory is associate with the following three
names:

keith
good_guitar_player
former_heroin_addict

A reference must be initialized to the variable to which it is referring
to when it is being declared:

```
int x;
int & a ; // error must always intialize a reference to some variable
```

A reference can only refer to one thing; once it has been intialized
IT CANNOT BE LATER CHANGED TO REFER TO SOMETHING ELSE.

USER DEFINED DATA TYPES: ENUM AND STRUCT

C++ allows for a programmer to declare and use their own data

```
types.  The enum and struct constructs are used for this purpose:

Suppose we want to associate special meanings to certain
integer values. We can use symbolic integer constants such
as

const int  Mon = 1;
const int  Tue = 2;
const int  Wed = 3;
const int  Thur = 4;
const int  Fri = 5;
const int  Sat = 6;
const int  Sun = 7;


Once we have declared these days, we can do things like:

int day;

day = Mon;

if ( day == Sat)
   cout << "Yahooo it is the weekend time for windsurfing" << endl;

Of course, we could have done the same thing using:

int day;
day = 1;
if ( day == 6)
   cout << "Yahooo it is the weekend time for windsurfing" << endl;

The benefit of associating symbolic names with the integer literal
1,2,3,4,5,6,7 is one of EASE OF PROGRAM READABILITY (most of a programmer's
time is spent not in writing original code, but in READING and MAINTAINING
someone else's previously (and maybe poorly commented) code, so clarity
and ease of readability are important concepts and not only some ivory-tower,
pie-in-the-sky academic niceties)

C++ allows for a convenient method of associating symbolic names with
some integer constant values through the use of the enum type declaration:

// Author: Ted Obuchowicz
// Jan. 28, 2002
// example program illustrating use of
// eumerated type declaration


#include <iostream>
#include <string>

using namespace std;



int main()
{

enum days {Mon=1, Tue, Wed, Thur, Fri, Sat, Sun};
```

```
days today ;

today = Sat;
if (today == Sat)
   cout << "Go windsurfing" << endl;

return 0;
}
```

The program output (as expected) is:

```
ted@flash Programs 8:49pm >enum
Go windsurfing
ted@flash Programs 8:50pm >
```

Had we not done {Mon=1, Tue, Wed, Thur, Fri, Sat, Sun}
but simply written {Mon, Tue, Wed, Thur, Fri, Sat, Sun}
then the symbolic constant Mon would be associated with the
value 0, Tue with 1, etc.

Struct

C++ has a struct declaration which is useful to hold information
of different data types pertaining to a certain object or thing.
For example, a student has an integer ID and a grade point average
which is a floating point value:

```
struct student
{
 int ID;
 float gpa;
}

student student_x, student_y;

student_x.ID = 9999999;
student_x.gpa = 0.00 // this person is in serious trouble

student_y.ID = 1234567;
student_y.gpa = 4.30 // this person wins a medal at convocation
```

We say that ID and gpa are the two FIELDS of the struct student.
We declare two variables student_x and student_y to be of type student.

The dot (.) notation is used to access the individual fields of a struct
variable.

Structs may be NESTED, meaning that the field of a certain struct can be
a previouslly declared STRUCT as in the following example:

```
// Author: Ted Obuchowicz
// Jan. 16, 2002
// example program illustrating use of
// nested structs
```

```
#include <iostream>
#include <string>

using namespace std;



int main()
{

enum months {Jan = 1, Feb, Mar, Apr, May, Jun, July, Aug, Sept, Oct, Nov, Dec };

enum days_of_the_week {Sun, Mon, Tue, Wed, Thur, Fri, Sat };

struct time
{
  int hour;
  int minute;
  int second;
  bool pm ;
};

struct date
{
 int year;
 months month;
 int day_of_month;
 days_of_the_week day_of_week;
 time time_now;
};

date today, yesterday, tomorrow;

today.year = 2002;
today.month = Jan;
today.day_of_month = 16;
today.day_of_week = Wed;
today.time_now.hour = 7;
today.time_now.minute = 12;
today.time_now.second = 35;
today.time_now.pm = true;


cout << "todays date is " << today.day_of_week << " " << today.month << " " <<
today.day_of_month << " " <<  today.year << endl;
cout << "time now is    " << today.time_now.hour << " " << today.time_now.minute
 << " " <<
today.time_now.second ;

if (today.time_now.pm == true )
  cout << " pm" << endl;
else
  cout << " am" << endl;

return 0;
}
```

Here is another program which makes whimsical use of the enum
and struct concepts:

```
// author: Ted Obuchowicz
// Jan. 15, 2002
// example program illustrating use of structs



#include <iostream>

using namespace std;



int main()
{

enum guitar_players {keith_richards, dave_gilmour, eric_clapton, mark_knopfler, chet_atkins,
                     pete_townshend, jimmy_page, ron_wood, angus_young, malcolm_young };
enum bass_players   {bill_wyman, john_entwistle, roger_waters, roger_glover, john_paul_jones };
enum vocalists      {mick_jagger, roger_daltrey, madonna, robert_plant, syd_barrett };
enum drummers       {keith_moon, nick_mason, charlie_watts, john_bonham};

struct band
{
 guitar_players  lead_guitar;
 guitar_players  rhythm_guitar;
 bass_players    bassist;
 vocalists       singer;
 drummers        percussionist;
 bool            are_they_any_good;
 bool            drummer_dead;
 unsigned long int   annual_income;
};


band rolling_stones;
band pink_floyd;
band led_zeppelin, the_who;
rolling_stones.rhythm_guitar = keith_richards;
rolling_stones.lead_guitar   = ron_wood;
rolling_stones.bassist   = bill_wyman;
rolling_stones.singer        = mick_jagger;
rolling_stones.percussionist = charlie_watts;
rolling_stones.are_they_any_good = true;
rolling_stones.drummer_dead = false;
rolling_stones.annual_income = 20123456;

pink_floyd.rhythm_guitar = dave_gilmour;
pink_floyd.lead_guitar = dave_gilmour;
pink_floyd.bassist = roger_waters;
pink_floyd.singer = syd_barrett;
pink_floyd.percussionist = nick_mason;
```

```
pink_floyd.are_they_any_good = true;
pink_floyd.drummer_dead = false;
pink_floyd.annual_income = 50000000;

led_zeppelin.rhythm_guitar = jimmy_page;
led_zeppelin.lead_guitar = jimmy_page;
led_zeppelin.bassist = john_paul_jones;
led_zeppelin.singer = robert_plant;
led_zeppelin.percussionist = john_bonham;
led_zeppelin.are_they_any_good = true;
led_zeppelin.drummer_dead = true;
led_zeppelin.annual_income = 30000000;

the_who.rhythm_guitar = pete_townshend;
the_who.lead_guitar = pete_townshend;
the_who.bassist = john_entwistle;
the_who.singer = roger_daltrey;
the_who.percussionist = keith_moon;
the_who.are_they_any_good = true;
the_who.drummer_dead = true;
the_who.annual_income = 25000000;


cout << rolling_stones.rhythm_guitar << endl;
cout << rolling_stones.lead_guitar << endl;
cout << rolling_stones.bassist << endl;
cout << rolling_stones.singer << endl;
cout << rolling_stones.percussionist << endl;
cout << rolling_stones.are_they_any_good << endl;
cout << rolling_stones.drummer_dead << endl;
cout << rolling_stones.annual_income << endl;


return 0;

Note that the program output is:

0
7
0
0
2
1
0
20123456
```

```
Assignment operator in C++
--------------------------

The assignment operator in C++ is the = symbol.  The
```

```
assignment operator STORES a result into a variable.

The general form of the assignment operator is the
following:

variable = expression ;

expression can be a literal value, another variable, or any
syntactically correct expression in C++.  Several examples are:

int x = 0 ;  // declare an integer variable called x and intialize it to 0
x = 10;  // give a a new value of 10

a more complex example is:

float gross_salary;
float tax_rate;
float take_home_pay;
float give_to_goverment_to_pay_for_social_programs;

give_to_goverment_to_pay_for_social_programs = tax_rate * gross_salary;
take_home_pay = gross_salary - give_to_goverment_to_pay_for_social_programs;

Assignment Conversions
---------------------

In an assignment statement, the left and right hand operators should be of the
same TYPE.

int x;
float y;
double z;

x = 6;
y = 2.3;
z = 5.678;

In the above expressions , the targets of the three assignment statements match
the values which are being assigned to them.  ( 6 is an integer literal, 2.3 and
5.678 are valid float and double literal expressions respectively).

In the case where the value of an assignment does not match the target type, the
 compiler
will attempt to convert the  right hand value so that its type MATCHES that of t
he target.

Consider the following program:

#include <iostream>
#include <string>

using namespace std;



int main()
{
 int x ;  // declare an integer variable called x
```

```
 x = 4.56 ;  // assign a floating point literal value to an integer
 cout << "The value of x after conversion is " << x << endl;
 return 0;

}
```

When compiled on our UNIX compiler (g++). The following is produced:

```
assignment_conversion.C: In function ‘int main()’:
assignment_conversion.C:19: warning: assignment to ‘int’ from ‘double’
```

Line 19 is the line:

```
x = 4.56 ;  // assign a double literal value to an integer
```

The program produces the following output:

```
The value of x after conversion is 4
```

Since 4.56 is a double constant, the assignment operator first converts the 4.5
6
into its integer equivalent 4 (by truncating the fractional portion).

Assignment conversion can lead to some unexpected results when the value of the
right hand operand is larger than can fit into the left-hand target:

```
#include <iostream>
#include <string>

using namespace std;



int main()
{
 short mick;
 int  keith;

 keith =  234456;
 mick = keith;

 cout << "Keith = " << keith << endl;
 cout << "Mick  = " << mick  << endl;

 return 0;

}
```

This code compiles with no warnings or errors.
It produces the following output:

```
Keith = 234456
```

```
Mick  = -27688


If we give the following initial values:

 keith =  65536;
 mick = keith;

and recompile and re-run the program, the following output results:

Keith = 65536
Mick  = 0


The reason is the following.  The size of an int is 4 bytes and the size
of a short is 2 bytes.  In binary the value of 65536 is

00000000 00000001 00000000  00000000

this is stored in memory as:

--------
00000000    keith
--------
00000000
--------
00000000
--------
00000001


The conversion of the int to a short takes the first 2 bytes, so mick is assigne
d
the value of 0. Please note that the actual ordering of the bytes stored in memo
ry
dpeends upon the compiler and the underlying compute rhardware upon which the pr
ogram
is run.  The above example is only for illustrative purposes and may not corresp
ond the
the exact way the binary information is stored in a particular's computer memory
.

Assignment precedence and associativity
--------------------------------------

C++ allows for expressions of the following form:

int mick, keith, ron;
ron = 1;


mick = keith = ron + 2;


How do we interpret the above expression? The answer lies in the
precedence and associativity of the assignment (=) operator.
Notice that the operand ron is surrounded by two different operators
(the = operator and the + operator).  Since the PRECEDENCE of the  +
operator is higher than that of the = operator, the addition is performed
```

```
first:

mick = keith = (ron + 2);

Next, we note that keith is surrounded by the = operator on both sides.
The associativity of the assignment operator determines which one is to
be performed first. Unlike the arithmetic operators, the assignment operator
is RIGHT associative.  This means the the original expression is to be interpret
ed
as

mick = (keith = (ron + 2));



The final values of the variables are:

mick = 3 keith = 3 ron = 1

This interpretation of the associativity of the assignment operator
is quite natural, since if the = operator were to be left associative
we would arrive at the following non-sensical expression:

(mick = keith) = (ron + 2);

Compound Operators
-----------------

In programming it is very common to perform operations of the
sort:

i = i + 1;
j = j - 5;
k = k * 4;
l = l / 2;
m = m % 2;

The above 5 examples all share the following common feature: An arithmetic
operator is being performed on a variable and then the result is stored
back in the variable.

C++ allows for the following SHORTHAND NOTATION:

i += 1;
j -= 5;
k *= 4;
l /= 2;
m %= 2;


The two versions are equivalent as shown by this following trivial
C++ program:

// Author: Ted Obuchowicz
// Jan. 14, 2002
// example program illustrating use of
// compound operators
```

```
#include <iostream>
#include <string>

using namespace std;



int main()
{

 int i,j,k,l,m ;

// give some initial values to the 5 variables

 i = 0;
 j = 10;
 k = 2;
 l = 8;
 m = 7;

// modify them using the "long hand versions"

i = i + 1;
j = j - 5;
k = k * 4;
l = l / 2;
m = m % 2;

// display the new values

cout << "i = " << i << endl;
cout << "j = " << j << endl;
cout << "k = " << k << endl;
cout << "l = " << l << endl;
cout << "m = " << m << endl;

// assign back the initial values

i = 0;
j = 10;
k = 2;
l = 8;
m = 7;

// modify them using the shorthand compound operators

i += 1;
j -= 5;
k *= 4;
l /= 2;
m %= 2;


// display the new values

cout << "i = " << i << endl;
cout << "j = " << j << endl;
cout << "k = " << k << endl;
```

```
cout << "l = " << l << endl;
cout << "m = " << m << endl;


return 0;

}


Increment and Decrement Operators
----------------------------------

C++ has two special operators for incrementing and decrementing a variable.
The increment operator is ++ and the decrement operator is --.  Can you
guess the origin of the name C++ ( C++ is C incremented with extra features ).

For example;

i++ is the same as i += 1 which is the same as i = i + 1

j-- is the samwe as j -= 1 which is the same as j = j - 1.

Both the increment and decrement operators come in two variations:
PREFIX and POSTFIX (the two above examples are both postfix versions).

The following sample program illustrates the differences between the
prefix and postfix forms of the increment and decrement operators:

Incorporating new mail into inbox...
// Author: Ted Obuchowicz
// Jan. 14, 2002
// example program illustrating use of increment and decrement
// operators in both postifx and prefix variations



#include <iostream>
#include <string>

using namespace std;



int main()
{
 int i = 0;

 int k = 5;

 cout << " i is " << i++ << endl;
 cout << " i is " << i << endl;
 cout << " k is "  << --k << endl;
 cout << " k is "  <<  k << endl;

}


When run, the program produces the following output:
```

```
 i is 0
 i is 1
 k is 4
 k is 4
```

One can conclude that the POSTFIX version of the increment and
decrement operators first use the value , then modify it. The PREFIX
versions first modify the value and then return the modified value
were it can be subsequently used.

Here is another program which illustrates the differences in the
postfix and prefix notations:

```
// Author: Ted Obuchowicz
// Jan. 14, 2002
//  another example program illustrating use of increment and decrement
// operators in both postifx and prefix variations



#include <iostream>
#include <string>

using namespace std;



int main()
{
 int i = 1;
 int j = 4;
 int k ;
 int l ;
 int m = 4;

 k = i * j++;
 l = i * --m;

 cout << " i  = " << i << " k = " << k << " j = " << j  << endl;
 cout << " i  = " << i << " l = " << l << " m = " << m << endl;
 return 0;

}
```

The program output is:

```
 i  = 1 k = 4 j = 5
 i  = 1 l = 3 m = 3
```

Note that in the expression k = i * j++ that the current value
of j (which is 4) is first used in computing the value of
i*j and this is then assigned to k. After the assignment to k,
the value of j is then incremented to 4.

Similarly, in the expression l = i * --m, the value of m is first

decrmented from 4 to 3 and then the decremented value is used to
calculate the value of 1 * 3, which is assigned to the variable
l.

Increment and Decrement operators for float, double, and long double
----

Both versions of the increment and decrement operators can be applied
to the floating point data types: float, double, and long double.
They have the effect of adding or subtracting 1.0 from a floating
point variable:

```
// Author: Ted Obuchowicz
// Jan. 14, 2002
// example program illustrating use of increment and decrement
// operators  with floating point numbers



#include <iostream>
#include <string>

using namespace std;



int main()
{
 float i = 0.2;

 double k = 5.8;

 cout << " i is " << i++ << endl;
 cout << " i is " << i << endl;
 cout << " k is "  << --k << endl;
 cout << " k is "  <<  k << endl;

}
```

The program output is (as we expect) :

```
 i is 0.2
 i is 1.2
 k is 4.8
 k is 4.8
```

Incrementing a char variable:

The ++ and -- operators can be applied to char variables as well with the
expected results that the result is equal to the next (or previous) character
in the ASCII collating sequence.

Consider the following program:

```
// Author: Ted Obuchowicz
```

```
// Jan. 15, 2002
// example program illustrating use of
// incrementing and decrementing a character variablw



#include <iostream>
#include <string>

using namespace std;



int main()
{

char keith;

keith = 'a';
cout << "keith is " << keith << endl;
keith++;
cout << "keith incremented once is " << keith << endl;
keith++;
cout << "keith incremented yet another time is " << keith << endl;
return 0;
}
```

Incrementing keith once yields ’b’ as the result. Incrementing yet again yields ’c’. Here is the program output:

```
keith is a
keith incremented once is b
keith incremented yet another time is c
```

The cast operator:

Consider the task of calculating the average value of two integers. It seems simple enough, read in the two integers, calculate the value of their sum divided by two and assign this to another variable and output the result.

Here is a first attempt:

```
// Author: Ted Obuchowicz
// Jan. 14, 2002
// example program illustrating use of
// cast operator
// This program reads in two integer values and computes their average


#include <iostream>
#include <string>

using namespace std;
```

```
int main()
{

int i, j ; // numbers to be read in from keyboard


cout << "Input an integer number " ;
cin >> i;
cout << "Input a second integer number ";
cin >> j;

float k ;

k = (i+j) / 2;

cout << " The average value is " << k << endl;

return 0;
}
```

When executed, the output is:

```
Input an integer number 3
Input a second integer number 2
 The average value is 2
```

What went wrong, we know that the average of 3 and 2 is 2.5 not 2.
The answer lies in the method that is used to calaculate the
value of

```
k = (i+j) / 2 ;
```

In this expression, i, j, and the literal 2 are all integer values,
hence the result of the expression is an integer value. The expression
is truncated from 2.5 to 2. The assignment opeator then converts this integer
2 into a float 2. This is not the desired result, we wish that that the
arithmetic expression be somehow performed in floating point arithmetic.
One solution is to force the user to input the values as floating point
numbers, but this burdens the user and is not natural. A second solution
is to force the expression to be performed in floating point arithmetic
by making one of the operands a floating point expression :

```
// Author: Ted Obuchowicz
// Jan. 14, 2002
// example program illustrating use of
// cast operator
// This program reads in two integer values and computes their average


#include <iostream>
#include <string>

using namespace std;
```

```
int main()
{

int i, j ; // numbers to be read in from keyboard


cout << "Input an integer number " ;
cin >> i;
cout << "Input a second integer number ";
cin >> j;

float k ;

k = (i+j) / 2.0;  // now the division will keep the fractional part

cout << " The average value is " << k << endl;

return 0;
}
```

The result is now:

```
Input an integer number 3
Input a second integer number 2
 The average value is 2.5
```

Their is another solution which makes use of the cast operator:

```
// Author: Ted Obuchowicz
// Jan. 14, 2002
// example program illustrating use of
// cast operator
// This program reads in two integer values and computes their average


#include <iostream>
#include <string>

using namespace std;



int main()
{

int i, j ; // numbers to be read in from keyboard


cout << "Input an integer number " ;
cin >> i;
cout << "Input a second integer number ";
cin >> j;

float k ;
```

```
k = ( (float) i+j) / 2;  // now the division will keep the fractional part

cout << " The average value is " << k << endl;

return 0;
}
```

A typical interaction with the program produces:

```
Input an integer number 3
Input a second integer number 4
 The average value is 3.5
```

In the expression k = ((float) i+j) / 2 , the (float)
portion is called a CAST. It causes the compiler to convert
the integer 1 into its equivalent float representation. The
rules of arithmetic in C++ then cause the addition to be
performed in floating point arithmetic and thus the division
is also performed in floating point arithmetic yielding a
floating point result.

In C++, there are indeed "many ways to skin a cat and cast an
expression to a desired type" .

Control Structures

Up to now, all of the programs we have seen were examples
of what is known as STRAIGHTLINE programming. Program
execution commenced with the first statement and proceeded
statement by statement until the end of the program was
reached. Most non-trivial programs require some sort of
mechanism to alter the sequence of statement execution.
C++ has two conditional constructs and three looping constructs.
The conditional constructs determine which statements are to be executed
depending upon a certain condition, the looping contructs determine the number
of times a block of statements is to be executed.

The Boolean type

In the original version of the C programming language a
LOGICAL FALSE was represented by the integer value 0,
a LOGICAL TRUE was represented by any NON-ZERO integer. Thus
in C (and early versions of C++) the following were all legal
ways of representing a logical true:

```
1
5
678*3
```

Similarly, the following are legal ways of expressing a
logical FALSE

```
0
16 % 16

C++ now has the built-in BOOLEAN logical data type which is named
bool.  The bool data type can take on one of two possible symbolic
constants: true, false.

Relational Operators
--------------------

C++ has two type of relational operators: equality and ordering.

The equality operators are:

==   returns true if both of the operands are equal to each other
!=   returns true if the two operands are not equal to each other

Some examples of the use of the equality operators are:

int i = 1;
int j = 1;
int k = 2;

(i==j) ; // this will  return true
(i==k) ; // this will return false
(i!=k) ; // this will return true

We are not limited to comparing only integers for equality or inequality.
Any built-in  C++ data type can be used (when working with floating point values
 we
should be a bit more careful... more on this later).

char letter1, letter2, letter3;

letter1 = 'a';
letter2 = 'b';
letter3 = 'c';

letter1 == letter2 ;  // this will return false;
letter2 != letter3 ;  // this will return true;



What do you think the output of the following program which tests
two structs for equality will be ?

// Author: Ted Obuchowicz
// Jan. 16, 2002
// example program illustrating use of
// equality operators on struct data type



#include <iostream>
#include <string>

using namespace std;
```

```
int main()
{

struct a_simple_struct
{
 int a_number;
 char a_letter;
 bool a_boolean;
};

a_simple_struct  first_struct, second_struct, third_struct;


first_struct.a_number = 1;
first_struct.a_letter = 'a';
first_struct.a_boolean = true;

second_struct.a_number = 1;
second_struct.a_letter = 'a';
second_struct.a_boolean = true;

third_struct.a_number = 5;
third_struct.a_letter = 'a';
third_struct.a_boolean = true;

if (first_struct == second_struct)
  cout << "first and second structs are equal" << endl;
else
  cout << "first and second structs are not equal" << endl;



return 0;
}


The answer is nothing! The program DOES NOT EVEN COMPILE :

ted@flash Programs 8:25pm >g++ -o struct_equality struct_equality.C
struct_equality.C: In function `int main()':
struct_equality.C:40: no match for `main()::a_simple_struct & == main()::a_simpl
e_struct &'


The compiler is complaining about line 40 in the source code which is the
if statement:

if (first_struct == second_struct)


If you want to compare structs, then the comparison must be done on a field-by
field basis. For example, the above program can be fixed by simply replacing
the if statement with something along the lines of:

if ((first_struct.a_number == second_struct.a_number) &&
   (first_struct.a_letter == second_struct.a_letter) &&
   (first_struct.a_boolean == second_struct.a_boolean) )
  cout << "first and second structs are equal" << endl;
```

```
else
  cout << "first and second structs are not equal" << endl;


Ordering Operators
------------------

C++ uses the following 4 ordering operators to determine the
relative size of two operands:

<    less than
>    greater than
<=   less than or equal
>=   greater than or equal

Examples:

( 1 < 2 ) will return true
( 1 > 2)  will return false



Boolean Operators
-----------------

C++ has three Boolean operators: &&, ||, and !. Their
meaning is as follows:

&& : used to perform a logical AND of two boolean expressions:

    ex.  (i < 3 ) && (k < 56 )

|| : used to perform a logical OR of two expressions:

    ex. (i < 2 ) || ( k > 45)

! : used to perform a logical NEGATION

    ex. ! (i < 2)

SHORT-CIRCUIT EVALUATION
------------------------

When evaluation the value of a logical expression, C++ requires that
first the left operand be evaluated to yield a boolean value, then the
right operand be evaluated. Furthermore, if the value of the entire
logical expression may be determined from the left operand, then the right
hand operand should not be evaluated.  This is known as short-circuit evaluation
.

For example:

( i != 0) && ( (k/i) < 2 )

the ( i != 0) will be equal to false only in the case of i being equal
to 0,  since the boolean && operator requires both its operands to be
true in order for the entire to be expression to be true, it is not even
necessary to evaluate the right-hand side ( (k/i) < 2 ) in the case
of i being equal to 0.  By virtue of short-circuit evaluation, we can
```

guarantee that division by zero will NEVER occur. If C++ did not
provide short-circuit evaluation, then there would be a possibility that
a division by zero would be attempted while evaluation the right hand
side operand.

Comparing floating point numbers

Round-off errors due to inherent limitiations of how real numbers
are internally stored in a computer's memory can cause some
problems when comparing floating point numbers for equality.

For example, consider the following program which adds 0.0000001 ten
times to theoretically give the value 0.000001:

```
// Author: Ted Obuchowicz
// Jan. 16, 2002
// example program illustrating use of
// testing real numbers for equality


#include <iostream>
#include <string>
#include <iomanip>

using namespace std;



int main()
{

float number;

number = 0.0000001 + 0.0000001 + 0.0000001 + 0.0000001
      + 0.0000001 + 0.0000001 + 0.0000001 + 0.0000001 + 0.0000001 + 0.0000001;

if (number == 0.000001 )
   cout << " number is equal to 0.000001" << endl;
else
   cout << " number is not equal to 0.000001" << endl;


cout <<  setprecision(15)   << "number is equal to " << number << endl;

return 0;
}
```

The output may surprise you:

```
 number is not equal to 0.000001
number is equal to 9.99999997475243e-07
```

When testing real number for equality, it is a better practice to check that
the two numbers are close enough to one another. The following program makes
use of the fabs() function found in the <math> library to check that the
absolute value of number - 0.000001 is less than some prespecified value called
epsilon:

```
// Author: Ted Obuchowicz
// Jan. 16, 2002
// example program illustrating use of
// testing real numbers for equality


#include <iostream>
#include <string>
#include <iomanip>
#include <math.h>

using namespace std;



int main()
{

float number;
const float epsilon = 0.001;

number = 0.0000001 + 0.0000001 + 0.0000001 + 0.0000001
      + 0.0000001 + 0.0000001 + 0.0000001 + 0.0000001 + 0.0000001 + 0.0000001;

if  (  fabs( (number - 0.000001) <= epsilon  )  )
   cout << " number is equal to 0.000001" << endl;
else
   cout << " number is not equal to 0.000001" << endl;


cout <<  setprecision(15)   << "number is equal to " << number << endl;

return 0;
}
```

We use the #include <math.h> statment to tell the compiler where to look for
the definition of the fabs function. We would compile this program on the
ECE UNIX g++ compiler with the following command line:

```
g++ -o floating_point_equality2 floating_point_equality2.C -lm
```

The -lm option must appear at the end of the command line. It tells the
linker to link the math library with your object code.

This time the program output is what we expect:

```
 number is equal to 0.000001
number is equal to 9.99999997475243e-07
```

The IF STATEMENT

The previous programs have already examined the use of the
IF statement. The if statement comes in two forms:

```
Form 1:  if (expression)
             statement;
```

```
Form 2:  if (expression)
             statement;
         else
             another statement;

NOTE:  If it is required to perform more than one statement in a branch of
       the if statement, then the entire block of statements must be enclosed
       in { } such as:

       if (expression)
          {
            statement1;
            statement2;
            ...
            statementn;
           };

       or using Form 2:

        if (expression)
          {
            statement1;
            statement2;
            ...
            statementn;
          };
        else
          {
            statementx;
            statementy;
            ...
            statementz;
          };





Be careful to use the delimiting { } properly. Failure to
do so may cause strange program behaviour as is the
following program:

// Author: Ted Obuchowicz
// Jan. 16, 2002
// example program illustrating INCORRECT use of
// if statement



#include <iostream>
#include <string>

using namespace std;



int main()
```

```
{

int i ;

cin >> i;


if (i != 1)
     cout << " i is not equal to 1" << endl;
     cout << " end of program goodbye" << endl;



return 0;
}


When we run this program and enter 1 for the value of 1,
the output is:

 end of program goodbye


What went wrong? The expression ( i != 1) is false for i equal to 1
(since 1 is equal to 1). So why was the "end of program goodbye"
produced as output? .  What the program actually did was:

if (i != 1)
     cout << " i is not equal to 1" << endl;


cout << " end of program goodbye" << endl;


The second cout is actually independent of the first if since we did
not delimit the two statements within { }.

If we change the program to:

// Author: Ted Obuchowicz
// Jan. 16, 2002
// example program illustrating INCORRECT use of
// if statement



#include <iostream>
#include <string>

using namespace std;



int main()
{

int i ;

cin >> i;
```

```
if (i != 1)
    {
     cout << " i is not equal to 1" << endl;
     cout << " end of program goodbye" << endl;
    };  // the above messages will be printed ONLY FOR values of i not equal to
1
        // when i is equal to 1, no output will be produced



return 0;
}


A sample session with the running program is:

ted@flash Programs 10:08pm >if_statement2
1
ted@flash Programs 10:09pm >


ted@flash Programs 10:09pm >if_statement2
3456
 i is not equal to 1
 end of program goodbye
ted@flash Programs 10:09pm >






CAVEAT
-------

The following is a VERY COMMON MISTAKE THAT BEGINNER C and C++
programmers make:


// Author: Ted Obuchowicz
// Jan. 16, 2002
// example program illustrating use of
// common programming error with an if



#include <iostream>
#include <string>

using namespace std;



int main()
{

int balance_owed;
```

```
cin >> balance_owed;

if (balance_owed = 0)
  cout << "You owe me nothing" << endl;
else
  cout <<  "You owe me " << balance_owed << " dollars" << endl;


return 0;
}


This program when run will ALWAYS print out:

You owe me 0 dollars

irrespective of the value the user inputs for the integer variable
balance_owed!

WHY????

look very carefully at the if statement:

if (balance_owed = 0)
    ^^^^^^^^^^^^^^^^^

  cout << "You owe me nothing" << endl;
else
  cout <<  "You owe me " << balance_owed << " dollars" << endl;


the (balance_owed = 0) is an ASSIGNMENT of 0 to the variable balance_owed.
The result of the assignment is that variable balance_owed take son the value
0. The condition then reduces to:

if(0)
   cout << "You owe me nothing" << endl;
else
   cout <<  "You owe me " << balance_owed << " dollars" << endl;

since a 0 is false in C++, the condition is always false so the
else part is always executed with a value of 0 for the amount owed!!!

/*********************************************************************/
/*                                                                   */
/*  ALWAYS USE == FOR COMPARISON IN IF STATEMENTS!!!!!               */
/*                                                                   */
/*                                                                   */
/*********************************************************************/

The proper version of the program is:


// Author: Ted Obuchowicz
// Jan. 16, 2002
// example program illustrating use of
// correct use of if statement
```

```
#include <iostream>
#include <string>

using namespace std;



int main()
{

int balance_owed;

cin >> balance_owed;

if (balance_owed == 0)
  cout << "You owe me nothing" << endl;
else
  cout <<  "You owe me " << balance_owed << " dollars" << endl;


return 0;
}
```

Note that the only difference between the good and the bad versions
is the difference between = and == . Amazing what a single missing
= will do in a program! PAY YOUR ATTENTION INSIDE OF IF STATEMENTS
AND ALWAYS CHECK FOR EQUALITY WITH THE ==   !!!!!!!

The proper output is:

```
ted@flash Programs 10:47pm >good_if
0
You owe me nothing
ted@flash Programs 10:47pm >good_if
456
You owe me 456 dollars
```

Nested if-else-if statements:

Consider the following if construct:

```
char command;
cin >> command;

if (command == 'u')
    cout << "Move up command was received" << endl;
else
   if (command == 'd')
      cout << "Move down command was received" << endl;
   else
      if (command == 'l')
```

```
         cout << "Move left command was received" << endl;
      else
         if (command == 'r')
            cout << "Move right command was received" << endl;
         else
            cout << "Invalid command" << endl;

The indentation of the if statements is purely cosmetic. The compiler does not c
are about
nesting. We can rewrite the above if statement as:

if (command == 'u')
        cout << "Move up command was received" << endl;
else  if (command == 'd')
        cout << "Move down command was received" << endl;
else  if (command == 'l')
        cout << "Move left command was received" << endl;
else  if (command == 'r')
         cout << "Move right command was received" << endl;
else
         cout << "Invalid command" << endl;

Using this style improves program readability and avoids the problem of running
out
of space on the right hand side as the nesting level of the if statements increa
ses.


The switch statement
-------------------

C++ has a very convenient statement for multi-branch conditional testing as in t
he
above statement. It is called the switch statement.  The above  if can be rewrit
ten
as a switch statement.

// Author: Ted Obuchowicz
// Jan. 17, 2002
// example program illustrating use of
//  switch statement



#include <iostream>
#include <string>

using namespace std;



int main()
{

char command;
cin >> command;

switch (command)
{
```

```
 case 'u':   cout << "Move up command was received" << endl;
             break;
 case 'd':   cout << "Move down command was received" << endl;
             break;
 case 'l':   cout << "Move left command was received" << endl;
             break;
 case 'r':   cout << "Move right command was received" << endl;
             break;
 default :   cout << "Invalid command" << endl;
}

return 0;

}
```

When the switch statment is executed, the value of command is compared with
the specified cases. If a match is found, the statements corresponding to the
matching case are executed. If the value of command does not match any of the
specified cases, then the sequience of statements for the specified default case
is executed (if there is no default case specified, flow of control resumes with
 the
statement immediately following the switch).

The break statement

Note that in the above switch, there is a break statement associated with every
specified case. The break statement causes a change of flow of control to the
statement following the switch. The break can be interpreted as "the switch sta
tement
has completed its task and program execution should resume with the statement im
mediately
following the switch statement".

If a switch statement does not contain breaks for the specified cases, then the
action for the FIRST matching value is executed, FOLLOWED BY ANY REMAINING group
s
of action statements (for the remaining values even though they do not match).

For example,

```
// Author: Ted Obuchowicz
// Jan. 17, 2002
// example program illustrating use of
//  switch statement



#include <iostream>
#include <string>

using namespace std;



int main()
{
```

```
char command;
cin >> command;

switch (command)
{
 case 'u':   cout << "Move up command was received" << endl;
 case 'd':   cout << "Move down command was received" << endl;
 case 'l':   cout << "Move left command was received" << endl;
 case 'r':   cout << "Move right command was received" << endl;
 default :   cout << "Invalid command" << endl;
}

return 0;
return 0;

}
```

If the user supplies u for the value of the command, the program produces the following output:

```
ted@flash Programs 12:55pm >switch_no_break
u
Move up command was received
Move down command was received
Move left command was received
Move right command was received
Invalid command
```

If we supply d for the value of the input variable command, the following output is produced:

```
ted@brownsugar Programs 7:21pm >switch_no_break
d
Move down command was received
Move left command was received
Move right command was received
Invalid command
```

giving l as an input value results in:

```
ted@brownsugar Programs 7:21pm >switch_no_break
l
Move left command was received
Move right command was received
Invalid command
```

if we give r as the value :

```
ted@brownsugar Programs 7:25pm >switch_no_break
r
Move right command was received
Invalid command
```

```
Cases in a switch with no break
-------------------------------

In most programming problems, every 'action' statement should
have an associated break statement  as in

switch (condition)
{
 case value1 : action1;
               break;
 case value2 : action2;
               break;
 ...

 case valuen : actionn;
               break;

 default : default_action;
}

In some programming situations, using the so-called "fall through"
behaviour (some actions without an associated break) makes sense.
Consider once again the menu switch where we now consider the
upper and lower case letters as being valid input characters
(i.e. both 'u' and 'U' should be recognized as a legal move up
command, 'd' and 'D' as legal move down commands etc).  We can rewrite
our  previous program as:



#include <iostream>
#include <string>

using namespace std;



int main()
{

char command;
cin >> command;

switch (command)
{
 case 'u':  // fall through to the 'U' case
 case 'U':   cout << "Move up command was received" << endl;
             break;
 case 'd':  // fall through to the 'D' case
 case 'D':   cout << "Move down command was received" << endl;
             break;

 case 'l':  // fall through to the 'L' case
 case 'L':   cout << "Move left command was received" << endl;
             break;
 case 'r':  // fall through to the 'R' case
 case 'R':   cout << "Move right command was received" << endl;
             break;
 default :   cout << "Invalid command" << endl;
```

```
}

return 0;

}


It should be noted that the switch statement in the above example
can be rewritten more clearly as:

case 'u' : case 'U' : cout << "Move up command was received" << endl;
                      break;

case 'd' : case 'D':  cout << "Move down command was received" << endl;
                      break;

case 'l':  case 'L':  cout << "Move left command was received" << endl;
                      break;
case 'r':  case 'R':  cout << "Move right command was received" << endl;
                      break;
default :   cout << "Invalid command" << endl;
}
```

Using this style clearly indicates the code which is to be executed
for common cases.

To reiterate, the switch statement is different from similar statements in
languages such as Pascal in a very important way. In C and C++, when a program
jumps to a certain line in a switch statement (when it finds the first matching
value of the listed cases) , the program THEN SEQUENTIALLY EXECUTES all the
statements following that line unless you EXPLICITLY TELL OTHERWISE
(with the break) . Program execution DOES NOT AUTOMATICALLY stop at the next
case. To make execution stop at a particular line, you must use the break
statement. The break causes the program to continue exection from the next
statement following the switch.

One final note regarding the switch statement. Both the expression which
is being tested and the values of the cases should be of an INTEGRAL type.
This limitation means that a switch statement cannot be used to determine
the flow of program control based on the value of a floating-point variable.

LOOPING CONSTRUCTS

Suppose we wish to find the sum , and average value of a list
of 4 numbers. Our first (brute-force) program might look like:

```
// Author: Ted Obuchowicz
// Jan. 21, 2002
// example program illustrating use of
// brute force approach to finding the sum and average of 5 floating
// point values



#include <iostream>
```

```
#include <string>

using namespace std;



int main()
{

float sum;
float average;
float number_1;
float number_2;
float number_3;
float number_4;
float number_5;


cout << "Please enter first number ";
cin >> number_1;

cout << "Please enter second number ";
cin >> number_2;

cout << "Please enter third number ";
cin >> number_3;

cout << "Please enter fourth number ";
cin >> number_4;

cout << "Please enter fifth number ";
cin >> number_5;

sum = number_1 + number_2 + number_3 + number_4 + number_5;

average = sum / 5.0;

cout << "The sum of the 5 numbers is : " << sum << endl;
cout << "The average value is : " << average << endl;

return 0;

}
```

Suppose we had to find the sum and average of a list of 10 000 values?
Clearly, the above brute force approach is not very convenient to use.

We can use a while loop to control the number of times a portion of
program code is to be executed. The following is an ALGORITHM which
repeatedly gets the next input value and adds it to the sum of the
previous input values and then calculates the average:

Step 1: Set the value of the running sum to 0.0

Step 2: Set the value of the number of processed values to 0.

```
Step 3:  while (number of processed items < sixe of the list )
              Step 3a:   read in a value;
              Step 3b:   sum = sum + value;
              Step 3c;   add 1 to the number of items processed so far

Step 4:   calculate the average as sum / list size

Step 5:  display the values of the sum and the average


A complete C++ program which corresponds to the above pseudocode is:

// Author: Ted Obuchowicz
// Jan. 21, 2002
// example program illustrating use of
// while loop to find sum and average of 5 floating point values read in
// from keyboard



#include <iostream>
#include <string>

using namespace std;



int main()
{

float sum;
float average;

const int size_of_list= 5;
float number;
int number_of_items_processed;

// perform some initialization

sum = 0.0;
number_of_items_processed = 0;

while (number_of_items_processed < size_of_list)
{
  cout << "Please enter a number " << endl;
  cin >> number;
  sum = sum + number;  // update running sum
  number_of_items_processed++; // increment number of items processed so far
}

average = sum / size_of_list;

cout << "The sum is " << sum << endl;
cout << "The average is "  << average << endl;


return 0;
}
```

Improving the above program:

The above program is fine for lists whose sizes is known ahead of time.
What if we wanted to find the sum and average of an unknown number of
values????

We can modify the above program to test the number which is input to
see if it is a special "flag" (in computer programming a flag variable
is one which the programmer places "special meaning" to). For example,
we can reserve the number -9999 to indicate that the list has no more values.
The modified program is :

```
// Author: Ted Obuchowicz
// Jan. 21, 2002
// example program illustrating use of
// while loop to find sum and average of 5 floating point values read in
// from keyboard



#include <iostream>
#include <string>

using namespace std;



int main()
{

float sum;
float average;

const float flag = -9999.0;
float number;
int number_of_items_processed;

// perform some initialization

sum = 0.0;
number_of_items_processed = 0;

cout << "Please enter a number " << endl;
cin >> number;

while ( ! ( (number - flag) < 0.00001 ) )
{

  sum = sum + number;  // update running sum
  number_of_items_processed++; // increment number of items processed so far
  cout << "Please enter a number " << endl;
  cin >> number;

}



if (number_of_items_processed != 0)
```

```
{
  average = sum / number_of_items_processed;
  cout << "The sum is " << sum << endl;
  cout << "The average is "  << average << endl;
}
else
{
  cout << " No numbers were entered " << endl;
  cout << " Sum is " << sum << endl;
  cout << " Impossible to compute average... sorry " << endl;
}



return 0;
}


Some comments on the above program:
----------------------------------

1) Note that before we enter the while loop, we first read in a
   number... this is necessary so that the variable called number
   has an intial value when it is being tested inside the conditional
   expression of the while loop:

cout << "Please enter a number " << endl;
cin >> number;

while ( ! ( (number - flag) < 0.00001 ) )
             -----
               |
                -----> this must have some value before it can be tested

2) the test in the while loop does not test for equality with -9999.0,
   rather it tests that the difference between the number read in and
   the constant flag of -9999 is less than some small number
   ( actually we should really test for the absolute value of the
     difference between number and flag being less than some value...)

3) in the while loop , we read in a new value for number:

   cin >> number;

    this is necessary , since  we eventually want the while loop to
   terminate.

4) we test that some numbers have been actually entered to avoid a potential
   division by zero in our source code:

   if (number_of_items_processed != 0)
{
  average = sum / number_of_items_processed;
  cout << "The sum is " << sum << endl;
  cout << "The average is "  << average << endl;
}
else
{
  cout << " No numbers were entered " << endl;
```

```
  cout << " Sum is " << sum << endl;
  cout << " Impossible to compute average... sorry " << endl;
}
```

Alternative solution

While flags are useful programming tricks, they are not totally
without problems of their own. What if the flag represents a
valid possible number to be entered?   Fortunately, C++ has a mechanism
which overcomes the problems associated with flags.

We can test for the end of input in the following manner:

```
while ( cin >> number )
 {
   do some more processing;
 }
```

The value of an extraction operator is true provided the extraction
is successfull (i.e. a number is typed in). When the input stream has
been exhausted (tha tis there are no more values to read in) the value of
the extraction operation is false and the while loop will terminate.

The expression cin >> number is true if and only if a number has been
read in.

To indicate that there are no more values to be entered, the user
types in an escape sequence which is specific to the operating system.
In UNIX, the end-of-file escape sequence is the Control-D sequence
(press the Control kety then the D key).

The program is as follows:

```
// Author: Ted Obuchowicz
// Jan. 21, 2002
// example program illustrating use of
// while loop to find sum and average of
// from keyboard using control-d to indicate end of file



#include <iostream>
#include <string>

using namespace std;



int main()
{

float sum;
float average;
```

```
float number;
int number_of_items_processed;

// perform some initialization

sum = 0.0;
number_of_items_processed = 0;

cout << "Please enter a number " << endl;

while ( cin >> number )
{

  sum = sum + number;  // update running sum
  number_of_items_processed++; // increment number of items processed so far
  cout << "Please enter a number " << endl;

}



if (number_of_items_processed != 0)
{
  average = sum / number_of_items_processed;
  cout << "The sum is " << sum << endl;
  cout << "The average is "  << average << endl;
}
else
{
  cout << " No numbers were entered " << endl;
  cout << " Sum is " << sum << endl;
  cout << " Impossible to compute average... sorry " << endl;
}



return 0;
}
```

Here is another example of a while loop. The program uses the simple
"keep subracting one number from another until you can't subtract anymore
" algorithm to perform simple integer division which yields an integer
quotient and remainder:

```
// Author: Ted Obuchowicz
// Jan. 16, 2002
// example program illustrating use of
// while loop to perform simple integer division which
// yields an integer quotient and an integer remainder
// using the grade school division algorithm



#include <iostream>
#include <string>

using namespace std;
```

```
int main()
{

int dividend;
int divisor;

int quotient = 0;
int remainder;

// read in the two values

cin >> dividend >> divisor;

cout << "Dividend = " << dividend <<  " Divisor = " << divisor << endl;

// check for a zero divisor

if ( divisor == 0 )
   cout << "Cannot divide by zero you bonehead!!!!" << endl;
else
  {
    while (dividend >= divisor)
       {
         quotient++;
         dividend = dividend - divisor;
       };
    remainder = dividend;
    cout << "Quotient = " << quotient <<  " Remainder = " << remainder << endl;
   }



return 0;
}
```

The for loop

C++ has another looping construct which is frequently used.
We will illustrate its use with a simple program which
performs integer multiplication by repeated addition. The program is based
on the fact that a multiplication of the form:

product = a * b

is equivalent to adding a to itself a total of b times
(or adding b to itself a total of a times). In the early
times of computing (when there were no built-in multiply instructions,
multiplication was actually carried out in such a manner).

```
// Author: Ted Obuchowicz
// Jan. 21, 2002
// example program illustrating use of
// for loop in a multiplication by repeated addition program
// we will keep it simple and assume only positive operands
```

```
#include <iostream>
#include <string>

using namespace std;



int main()
{

int product = 0; // initialize to 0
int multiplicand;
int multiplier;

cout << "Enter the multiplicand " << endl;
cin >> multiplicand ;

cout << "Enter the multiplier " << endl;
cin >> multiplier;

for (int i = 0 ; i < multiplier ; i++ )
{
    product = product + multiplicand;
}

cout << multiplicand << " * " << multiplier << " = " << product << endl;


return 0;
}
```

The general form of a for loop is the following:

```
for ( initialization ; test_expression ; postexpression )
    action;
```

(again as in the case of an if, if more than one statement
constitutes the action portion, then they must be enclosed
in the { }) :

```
for ( initialization ; test_expression ; postexpression )
{
  action1;
  action2;
  ...
   actionn;
}
```

When a for loop is encountered in a program the following steps
are performed:

1) the initialization is performed first
2) next, the test_expression is evaluated, if the expression is true then
   the step(s) in the Action portion are executed and then postexpression

```
   is executed
3) the test_expression is reevaluated, if true, the action statements are
   again executed, if false the for loop TERMINATES and control continues
   with the next statement in the program.


Another example:
---------------

We can re-write our while loop program to use a for loop:

// Author: Ted Obuchowicz
// Jan. 21, 2002
// example program illustrating use of
// for loop to find sum and average of 5 floating point values read in
// from keyboard



#include <iostream>
#include <string>

using namespace std;



int main()
{

float sum;
float average;

const int size_of_list= 5;
float number;
int number_of_items_processed;

// perform some initialization

sum = 0.0;
number_of_items_processed = 0;

for (int i = 0; i < size_of_list ; i++)
{
  cout << "Please enter a number " << endl;
  cin >> number;
  sum = sum + number;  // update running sum
}

average = sum / size_of_list;

cout << "The sum is " << sum << endl;
cout << "The average is "  << average << endl;


return 0;
}

In a for loop , any of the
```

initialization
test_expression
postexpression

may be omitted. This may appear strange , but consider the following
program

```
// Author: Ted Obuchowicz
// Jan. 21, 2002
// example program illustrating use of
//  an INFINITE for loop



#include <iostream>
#include <string>

using namespace std;



int main()
{

for( ; ; )
{
 cout << "This is an infinite for loop " << endl;
}

return 0;  // actually the program will never return untill you kill it with
           // control-c

}
```

The following is also another variation of the above theme:

```
for( ; true  ; )
{
 cout << "This is an infinite for loop " << endl;
}
```

As another example of the versatility of the for loop let us first
rewrite our general purpose sum and average program:

```
// Author: Ted Obuchowicz
// Jan. 21, 2002
// example program illustrating use of
// for loop to find sum and average of
// from keyboard using control-d to indicate end of file



#include <iostream>
#include <string>

using namespace std;
```

```
int main()
{

float sum;
float average;

float number;

// perform some initialization

sum = 0.0;

cout << "Please enter a number " << endl;

for ( int number_of_items_processed = 0; cin >> number; number_of_items_processe
d++ )
{

  sum = sum + number;  // update running sum
  cout << "Please enter a number " << endl;

}



if (number_of_items_processed != 0)
{
  average = sum / number_of_items_processed;
  cout << "The sum is " << sum << endl;
  cout << "The average is "  << average << endl;
}
else
{
  cout << " No numbers were entered " << endl;
  cout << " Sum is " << sum << endl;
  cout << " Impossible to compute average... sorry " << endl;
}



return 0;
}


While at firsrt glance, this program appears both syntactically and
semantically correct, it DOES NOT EVEN COMPILE!!!! :

Here are the compilation error messages as reported by the
g++ compiler:

ted@brownsugar Programs 8:47pm >g++ -o for_loop_gen_purpose_sum for_loop_gen_pur
pose_sum.C
for_loop_gen_purpose_sum.C: In function ‘int main()’:
for_loop_gen_purpose_sum.C:40: name lookup of ‘number_of_items_processed’ change
d for new ANSI ‘for’ scoping
for_loop_gen_purpose_sum.C:30:   using obsolete binding at ‘number_of_items_proc
essed’
```

```
The problem is related to RULES OF SCOPE.  Basically, the problem is that
the loop index variable number_of_items_processed is known only within the
body of the for loop.  Outside the for loop, this index variable cannot
be accessed.

The solution is to remove this variable declaration from the for loop
initialization statement and put it as part of the main program:

// Author: Ted Obuchowicz
// Jan. 21, 2002
// example program illustrating use of
// for loop to find sum and average of
// from keyboard using control-d to indicate end of file



#include <iostream>
#include <string>

using namespace std;



int main()
{

float sum;
float average;

float number;

// perform some initialization
sum = 0.0;

cout << "Please enter a number " << endl;

int number_of_items_processed = 0;


for ( ; cin >> number; number_of_items_processed++ )
{

  sum = sum + number;  // update running sum
  cout << "Please enter a number " << endl;

}



if (number_of_items_processed != 0)   // this will not compile since scope rules
                                      // limit visibility of number_of_items_pro
cessed
                                      // to body of for loop
{
  average = sum / number_of_items_processed;
  cout << "The sum is " << sum << endl;
  cout << "The average is "  << average << endl;
```

```
}
else
{
  cout << " No numbers were entered " << endl;
  cout << " Sum is " << sum << endl;
  cout << " Impossible to compute average... sorry " << endl;
}



return 0;
}
```

Note that in the case that the test_expression evaluates to false
right away, the body of the for loop is never executed:

```
// Author: Ted Obuchowicz
// Jan. 21, 2002
// example program illustrating use of
// of a for loop whose action stateements are never executed



#include <iostream>
#include <string>

using namespace std;



int main()
{

for(int index = 0 ;  ( 1 > 1024); index++ )
{
 cout << "This will not be printed" << endl;
 cout << "Keith is better than Madonna " << endl;
}


return 0;
}
```

In the above program, the initialization of the loop index
is performed and the expression (1 > 1024) is evaluated,
since it evaluates to false, the body of the loop is never
entered and control continues with the next statement (which
is the end of the program....)

Nested loops:

```
Loops may be nested.  For example;


// Author: Ted Obuchowicz
// Jan. 7, 2002
// example program illustrating use of
// nested for loops



#include <iostream>
#include <string>

using namespace std;



int main()
{

for(int out = 1 ; out <= 5; out++)
   for (int in = 1 ; in <= 5 ; in++)
   {
    cout << "out = " << out << " in = " << in << "  " ;
    if (in % 5 == 0 )
      cout << endl;
   }

return 0;
}


The program output is:

out = 1 in = 1  out = 1 in = 2  out = 1 in = 3  out = 1 in = 4  out = 1 in = 5
out = 2 in = 1  out = 2 in = 2  out = 2 in = 3  out = 2 in = 4  out = 2 in = 5
out = 3 in = 1  out = 3 in = 2  out = 3 in = 3  out = 3 in = 4  out = 3 in = 5
out = 4 in = 1  out = 4 in = 2  out = 4 in = 3  out = 4 in = 4  out = 4 in = 5
out = 5 in = 1  out = 5 in = 2  out = 5 in = 3  out = 5 in = 4  out = 5 in = 5


In the above example, the statements in the body of the innermost for loop are
repeated for every value of out in the outer for loop:

First the value of out is set to 1 and then the inner loop is executed
with values of in ranging from 1 to 5, then the outer loop index out is incremen
ted
to 2 and again the inner loop repeats with values of in ranging from 1 to 5.
The process repeats until the outer loop terminated when out becomes equal to
6.


The break statement:
-------------------

A break statement can be used inside a loop construct to
prematurely terminate the loop. Program execution then
resumes with the statement which follows the looping statement.
The break can be used in any of the C++ looping constructs.
```

Here is an example of a for loop with a break statement which
forces an exit from the loop when the user enters the value
-999:

```
// Author: Ted Obuchowicz
// Jan. 7, 2002
// example program illustrating use of
// break statement inside a for loop to cause premature exit
// from the loop


#include <iostream>
#include <string>

using namespace std;



int main()
{

int exit_value = -999;  // suppose we wish to stop processing the loop if user
                        // inputs this value
int input_value;


for ( ; cin >> input_value ; )
    if ( input_value != exit_value)
       cout << "You entered " << input_value << endl;
    else
      {
        cout << "exit_value encountered in input stream... breaking from loop!"
<< endl;
        break;  // exit from the loop
       }


return 0;
}
```

The use of break statements inside of loops is somewhat recommended against
as it makes program behaviour difficult to understand. Sometimes it is
possible to rewrite the loop without using a break as in the following
possible to rewrite the loop without using a break as in the following
alternate version of the above program:

```
// Author: Ted Obuchowicz
// Jan. 23, 2002
// example program illustrating use of
// loop rewritten without a break

#include <iostream>
#include <string>

using namespace std;
```

```
int main()
{

int exit_value = -999;  // suppose we wish to stop processing the loop if user
                        // inputs this value
int input_value;


for ( ;  ( (cin >> input_value)  && (input_value != exit_value) ) ; )
       cout << "You entered " << input_value << endl;




return 0;
}
```

A BREAK STATEMENT MAY ONLY APPEAR IN A SWITCH STATEMENT OR IN
ANY OF THE LOOP STATEMENTS. IT MAY NOT APPEAR ANYWHERE ELSE!

The following program does not compile (since it contains
a break in an if statement which is not allowed in C++):

```
ted@brownsugar Programs 8:01pm >g++ -o illegal_break illegal_break.C
illegal_break.C: In function ‘int main()’:
illegal_break.C:30: break statement not within loop or switch
```

```
// Author: Ted Obuchowicz
// Jan. 23, 2002
// example program illustrating
// an illegal use of a break statement



#include <iostream>
#include <string>

using namespace std;



int main()
{

int i, j;

cin >> i >> j;

if ( i < 5 )
  {
    cout << "i less than 5" << endl;
    cout << "Have a nice day" << endl;
    if ( j < 10 )
      {
         cout << " j less than 10 " << endl;
         cout << " have a good evening " << endl;
```

```
         break; // this break is illegal
       }
     cout << " j is not less than 10 " << endl;
   }

return 0;
}


The continue statement:
-----------------------

The continue statement is used ONLY IN LOOPS.  When a continue statement
is encountered in the body of a loop, the REMAINING statements in the
body of the loop are SKIPPED OVER, and a new LOOP ITERATION commences.
Note that that while a continue statement causes the program to skip any
remaining body statements , it does not skip the loop test_expression.

In a for loop, the continue makes the program skip to the
postexpression part and then it tests the condition.

In a while loop, the continue causes the program to go directly
to the test expression part of the while (condition) .

Here is a program which uses a continue statement within a
for loop which reads 5 characters and counts the number
of 't' characters which have been entered:

// Author: Ted Obuchowicz
// Jan. 23, 2002
// example program illustrating use of
// a continue statement within a for loop



#include <iostream>
#include <string>

using namespace std;



int main()
{

char a_letter;
int how_many_ts = 0;

for(int i = 0; i < 5; i++)
{
  cout << "Please enter a letter " ;
  cin >> a_letter;
  if (  a_letter != 't' )
     continue;
  cout << "Ahhhh... the t was entered!" << endl;
  how_many_ts++;
}

cout << "The total number of t's entered was " << how_many_ts << endl;
```

```
return 0;
}

If the letter entered is not equal to 't', then the program
goes to the i++ part of the for (int i = 0; i < 5; i++) part
and evaluates if the condition i < 5 is true or not. Based on this
evaluation either the loop continues or terminates.  If the letter
entered is equal to 't', then the two statements following the continue
are executed and loop execution carries on in the normal manner (when it
gets to the } the i++ is executed and the condition is evaluated).


More examples of programs with loops
------------------------------------

Example 1: Computing the factorial of a postive integer using a for loop.

The factorial of an integer n (designated as n!)
 is defined as the product of the integers
1 through n:

n!    =      1 if n = 0
      =      n x (n-1) x ... x 1  if n >=1

We can use a for loop to compute the product of the numbers 1 through
n as in the following program:




// Author: Ted Obuchowicz
// Jan. 23, 2002
// example program illustrating use of
// a for loop to compute the factorial
// of a number n



#include <iostream>
#include <string>

using namespace std;



int main()
{

unsigned long int n;
unsigned long int  n_factorial = 1;

cout << "Please enter a positive integer " << endl;
cin >> n;

for (unsigned long int i = 1 ; i <= n ; ++i)
   n_factorial = n_factorial * i;
```

```
cout << n << "! is " <<  n_factorial << endl;

return 0;
}


This program exhibits interesting behaviour:

it reports the following answers:

10! is 3628800
11! is 39916800
12! is 479001600
13! is 1932053504

the last value is incorrect as the true value of 13! = 6 227 020 000.
This value exceeds the maximum value of an unsigned long int
(which on our UNIX workstations is 4 294 967 295) so an integer
OVERFLOW arises during the calculation of 13! and an incorrect answer
is given.

How do we fix the problem???

Simple.... change the type of the variable n_factorial from unsigned
long int to a double:

// Author: Ted Obuchowicz
// Jan. 23, 2002
// example program illustrating use of
// a for loop to compute the factorial
// of a number n



#include <iostream>
#include <string>
#include <iomanip>   // needed for the setprecision()

using namespace std;



int main()
{

unsigned long int n;
double  n_factorial = 1.0;

cout << "Please enter a positive integer " << endl;
cin >> n;

for (unsigned long int i = 1 ; i <= n ; ++i)
   n_factorial = n_factorial * i;

cout << setprecision(15)  << n << "! is " <<  n_factorial << endl;

return 0;
}
```

Now, it gives correct results (to a certain point):

```
10! is 3628800
11! is 39916800
12! is 479001600
13! is 6227020800
14! is 87178291200
69! is 1.71122452428141e+98  (on most pocket calculators this is nearing the lim
it)
70! is 1.19785716699699e+100
100! is 9.33262154439441e+157
170! is 7.25741561530799e+306
171! is Infinity
```

Interestingly enough, it reports 171! as Infinity... recall the
range of the double data type... it was someting like 1.75 e +308.
When we tried to calculate 171! a floating point overflow occured...
most computers use a standard called the IEEE 754 floating point
standard to represent real numbers... the standard has a method
of representing Infinity (when the number exceeds the representable
range of values)... similary the IEEE 754 standard has something
called NaN (Not a Number),,, you will see this displayed when you
try to do something stupid like taking the square root of
a negative number .

What do you think the output of the following program will
be?

```
/ Author: Ted Obuchowicz
// Jan. 23, 2002
// example program illustrating
//  an attempt to divide by zero


#include <iostream>
#include <string>

using namespace std;



int main()
{


cout << " 1 divided by 0 is " << 1 / 0 << endl;

return 0;
}
```

These are the compile time messages:

```
infinity.C: In function 'int main()':
infinity.C:18: warning: division by zero in '1 / 0'
```

Let's run the program:

```
ted@flash Programs 9:57pm >infinity
Floating exception
ted@flash Programs 9:57pm >
```

Example 2: A program to compute the first 15 Fibonacci Numbers:

The sequence of numbers:

1 1 2 3 5 8 13 21 , ...

where the first two numbers in the sequence are equal to 1, and every successive term is the sum of the previous two terms is known as the Fibonacci sequence.
It was developed by an Italian mathematician by the name of Leonardo Pisano in 1202. He stumbled upon his famous sequence while pondering the problem of determ
ining
the number of rabbits which a pair of fertile rabbits may produce in one year...

Here is a program which uses a for loop to display the first 15 Fibonacci numbe
rs:

```
// Author: Ted Obuchowicz
// Jan. 23, 2002
// example program illustrating use of
// for loop to display the first 15 Fibonacci numbers


#include <iostream>
#include <string>

using namespace std;



int main()
{

int previous = 1;
int current  = 1;
int sum;

cout << 1  << endl;
cout << 1  << endl;

for(int i = 3; i <= 15; i++)
   {
     sum = previous + current;
     cout << sum << endl;
     previous = current;
     current = sum;
    }
```

```
return 0;
}



The program output is:

1
1
2
3
5
8
13
21
34
55
89
144
233
377
610


The do-while loop
-----------------

C++ has another loop constuct called the do-while loop which is
used in situations where a certain action is to be performed
AT LEAST ONCE.  The general form is:


do
{
  statement 1;
  statement 2;
  ...
  last_statement_to_do;
} while (some_condition);


Here is a simple program which uses the do while statement:

// Author: Ted Obuchowicz
// Jan. 28, 2002
// example program illustrating use of
// do while construct with a guessing game which continues
// prompting the user for a number between 1 and 100 until
// the user guess the "right answer" (which is set at 58,
// Keith Richards' current age. The program also keeps track
// of the number of guesses until a corrdsct guess is entered


#include <iostream>
#include <string>

using namespace std;
```

```
int main()
{

const int Keiths_age = 58;
int num_of_guesses = 0;
int user_guess;

do
{
 cout << "Please enter a guess between 1 and 100" << endl;
 cin >> user_guess;
 num_of_guesses++;
} while (user_guess != 58);

cout << "Congratulations. It took you " << num_of_guesses << " to guess Keith’s
age." << endl;


return 0;
}
```

Some more examples of loops:

Here is a program which uses while loops to "decompose" an
integer value (which is less than 1 000 000) into its
components consisting of the number of hundreds of thousands,
number of tens of thousands, number of thousands, etc.

A typical interaction withe the program is:

```
Please  enter a number less than a million
567890
hundreds of thousands = 5
tens of thousands = 6
thousands = 7
hundreds = 8
tens = 9
ones = 0
```

Here is the program which uses a sequence of while loops:

```
// Author: Ted Obuchowicz
// Jan. 29, 2002
// example program illustrating use of



#include <iostream>
#include <string>

using namespace std;
```

```
int main()
{

int hundred_thousands_counter = 0;
int ten_thousands_counter = 0;
int thousands_counter = 0;
int hundreds_counter = 0;
int tens_counter = 0;
int ones = 0;

int number;
cout << "Please  enter a number less than a million" << endl;
cin >> number;

while (number >= 100000)
{
  hundred_thousands_counter++;
  number = number - 100000;
}

while (number >= 10000)
{
  ten_thousands_counter++;
  number = number - 10000;
}

while (number >= 1000)
{
 thousands_counter++;
 number = number - 1000;
}

while (number >= 100)
{
 hundreds_counter++;
 number = number - 100;
}

while (number >= 10)
{
 tens_counter++;
 number = number - 10;
}

ones = number;

cout << "hundreds of thousands = " << hundred_thousands_counter << endl;
cout << "tens of thousands = " <<  ten_thousands_counter << endl;
cout << "thousands = " <<  thousands_counter << endl;
cout << "hundreds = " <<  hundreds_counter << endl;
cout << "tens = " << tens_counter << endl;
cout << "ones = " << ones << endl;



return 0;
```

```
}


We start off with the first while loop which will be entered
if the number >= 100000, the loop is entered  and we keep
on subtracting 100000 from the number and increment a counter.
Eventually, the number becomes less than 100000 and we continue
with the next loop. This loop is responsible for subtracting
10000 from the resulting number and incrementing a certain other
counter. We do this for  1000, 100, 10 ... whatever is left over
is the number of 'ones' in the number.




Here is a more cryptic version of a similar program.
It simply displays line by line the number of hundreds of thousands,
number of thousands, etc that make up a certain number:

The output is:

Please  enter a number less than a million 345678
3
4
5
6
7
8


The actual program is:

// Author: Ted Obuchowicz
// Jan. 29, 2002
// example program illustrating use of




#include <iostream>
#include <string>
#include <math.h>

using namespace std;




int main()
{



int number;
cout << "Please  enter a number less than a million" ;
cin >> number;



for(int i = 6 ; i >= 1 ; i--)
{
   cout <<  (number %  (int)( pow(10, i) )) / ( (int) pow(10, i-1)) << endl ;
}
```

```
return 0;
}
```

Which version is easier to understand???

The pow(x,n) function returns the value of

x^n by the way. It is found in the library <math.h>

It makes use of the following facts about the mod operator:

```
 i = 1 ones =   345 % 10 = 5
                                                1
 i = 2 tens =   345 % 100 =  45 , then do 45/ 10   , where the division
                                   is performed as INTEGER DIVISION
                             (4.5, we truncate the 0.5 to obtain 4)

                                                     2
 i = 3 hundreds =  345 % 1000  = 345, then do 345 / 10
                                      to obtain 3 (again as interger division)
```

in each loop interation we are doing

```
                     i
          number % 10   ,
          then take this result and do

                      i-1
          result %  10
```

Difficult to understand from the code, but it was possible to code
the program using a single clever for loop.

Library Functions

In this section we turn our attention to using predefined library
functions. We have already seen a limited use of library functions
(recall the use of the fabs library function).

C++ contains many different libraries. In order to access a function
within a particular library is is neccessary to #include <library_name>
in your C++ program. Before discussing the different libraries available
in C++, we will give an example of the sqrt function which is found in the
cmath library (or the old C math.h library).

sqrt function

The two real roots of the general quadratic equation:

```
  2
```

```
ax  +  bx  +  c = 0 ,
                                            2
are given by the formula : x = -b +- sqrt(b  - 4ac)
                               --------------------
                                    2a

(we will omit complex roots for the time being...).

The following program reads in three doubles for the values of
a,b,c.  It checks that the value of

b - 4*a*c is greater than 0.  If this is true, then it computes the
two real roots by PASSING a value to the sqrt library function:


// Author: Ted Obuchowicz
// Jan. 28, 2002
// example program illustrating use of
// square root library function found in the cmath library


#include <iostream>
#include <string>
#include <cmath>  // needed for the definition of the sqrt function

using namespace std;



int main()
{

cout << "Enter the three coefficients of the equation: a, b, c " ;
double a,b,c;
cin >> a >> b >> c;
double radical;
radical = (b*b) - (4*a*c) ;

if ( (a != 0) && (radical > 0)  )
   {
     double first_root =  (-b + sqrt(radical)) / (2*a);
     double second_root = (-b - sqrt(radical)) / (2*a);
     cout << "The roots are " << first_root << "  and " <<  second_root << endl;
   }
else
   cout << "The equation does not have two real roots " << endl;

return 0;
}


When the program gets to the two lines:

double first_root =  (-b + sqrt(radical)) / (2*a);
double second_root = (-b - sqrt(radical)) / (2*a);

a sequence of events occurs:

1) the appearance of the expression sqrt(radical) in the two
```

```
   lines will cause a FUNCTION INVOCATION to occur.  We say that the
   variable radical is a FUNCTION ARGUMENT.  The sqrt function function
   receives the value of this variable, it then calculates the value of the
   square root of whatever value was passed to it.

2) When the function  has computed the calculation of the square root,
   it returns the value back to the program.

In programming terminology, we see that there is an "invocation" of
the square root function which "returns" a value back to the "calling
program".

In order to make use of library functions we must first know several
things concerning the function:

1) the name of the function
2) the number and type of  arguments that the function expects
3) the return type of the function.

All this information is contained in what is known as the
function prototype.  For example, the prototype for the
sqrt function is:

double sqrt(double) ;

The above prototype (which is found in the math.h library file, which is
located in /usr/include/math.h on our UNIX systems) reveals the following:

the function is called sqrt, it expects a single (formal) parameter
 of type double.
(the parameter list is the part enclosed in () after the function name).
The function returns a single value of type double.


The type of the actual parameter which is passed to a function should
MATCH the type of the  formal parameter given in the function prototype.
If there is mismatch between the types of the formal parameter and the
actual arguments passed to the function, the compiler will attempt to
convert the types if possible (for example, promoting an int to a double).
If the usual conversions cannot convert the actual argument into the type
specified by the formal parameter, the compiler gives up and report and
error.

Here are some examples of legal invocations of sqrt:

// Author: Ted Obuchowicz
// Jan. 28, 2002
// example program illustrating use of
//  legal square root  invocation calls


#include <iostream>
#include <string>
#include <cmath>  // needed for the definition of the sqrt function

using namespace std;
```

```
int main()
{


cout << sqrt(2) << endl; ; // the integer 2 will be promoted to a double and a d
ouble will be ret
urned
int four = 4;
cout << sqrt(four) << endl ; // legal
cout << sqrt(-2.0) << endl ; // believe it or not this will display Nan
return 0;
}
```

The program output is:

```
ted@flash Programs 9:03pm >legal_square_root
1.41421
2
NaN
ted@flash Programs 9:03pm >
```

One can even do something of the sort illustrated in the next
program:

```
// Author: Ted Obuchowicz
// Jan. 31, 2002
// example program illustrating use of
// square root library function found in the cmath library


#include <iostream>
#include <string>
#include <cmath>  // needed for the definition of the sqrt function

using namespace std;



int main()
{

double num ;
cin >> num;
double quadric_root = sqrt(sqrt(num));
cout << quadric_root << endl;

return 0;
}
```

Some illegal invocations (where the usual conversions are not able to
convert the actual argument into the type defined by the formal parameter)
are:

```
// Author: Ted Obuchowicz
// Jan. 28, 2002
```

```
// example program illustrating use of
//  illegal square root  invocation calls


#include <iostream>
#include <string>
#include <cmath>  // needed for the definition of the sqrt function

using namespace std;



int main()
{



 cout << sqrt(3.4, 5.6) << endl  ;


return 0;
}
```

The errors reported by the compiler are:

```
ted@flash Programs 9:09pm >g++ -o illegal_square_root illegal_square_root.C
illegal_square_root.C: In function 'int main()':
/usr/include/math.h:133: too many arguments to function 'double sqrt(double)'
illegal_square_root.C:20: at this point in file
```

Note that the compiler reports the line number in the include file
/usr/include/math.h which contains the prototype and it also indicates
the line number in the source code where the illegal invocation appears.

A function can return a single value, or it can return no value.
For functions which do not return a value, the return type in the
function prototype must be the built-in type called void:

```
void this_function_returns_no_value(int, int);
```

The above is a prototype of a function which takes two formal arguments
of type int and does not return a value. You may think it strange that
we would decleare and define functiosn with no return types, but they are useful
in displaying messages:

(the following program uses what is known as a user-defined function, these
will be explained in greater detail in a later section):

```
// Author: Ted Obuchowicz
// Jan. 7, 2002
```

```
// example program illustrating use of
// of a function with no return value


#include <iostream>
#include <string>

using namespace std;

// define the function first

void this_function_returns_no_value(int first, int second)
{
 cout << "first is " << first << endl;
 cout << "second is " << second << endl;
}


int main()
{

this_function_returns_no_value(1,2);
this_function_returns_no_value(3,4);
this_function_returns_no_value(34,7869);

return 0;
}
```

The output is:

```
ted@flash Programs 9:22pm >void_function
first is 1
second is 2
first is 3
second is 4
first is 34
second is 7869
ted@flash Programs 9:22pm >
```

Some Common C++ libraries

We will now examine some of the common C++ libraries and
the functions they contain. We first discuss various methods
of including libraries.

The standard C libraries are those with a .h suffix in their library
name. They would be included as :

```
#include <stdlib.h>
```

C++ libraries do not require the .h suffix and are included
by simply specifiying the name of the library within the <>
brackets as in:

```
#include <iostream>
```

```
Some C++ are derived from the standard C libbraries and have a
c as the prefix as in:

#include <cassert>
#include <cmath>

Again, for the derived libraries, there is no need for the .h suffix
in the library name.

The best source of information for access to standard libraries is your
compiler documentation and a language reference manual.

The stdlib library
------------------

This is a C-based library called "standard lib". It contains a collection
of miscellaneous functions and type definitions.  Some of the commonly
used functions found in stdlib are:


int abs(int) :  returns the absolute value of the integer argument passed

void exit(int) : causes the calling program to terminate with a return value
                 equal to the parameter passed to the function:

    exit(0); // terminate program with a return value of 0
    exit(1); // terminate program with a return value of 1



#include <iostream>
#include <string>
#include <stdlib.h>

using namespace std;



int main()
{

exit(1);
cout << "This will never be printed " << endl;


}


This program when compiled and run will do nothing and control
will be passed back to the operating system.

Here is an interesting experiment to perform on a UNIX system.
From a X-window, type the following from the UNIX prompt:

exit(0)

What happens to your window? It has disappeared.  exit is actually
a system call.  C was originallt developed to write the UNIX operating
```

```
system. There is a strong coupling between UNIX and the C-language.


int rand()  :  returns a (somewhat) random number between 0 and
               RAND_MAX , where RAND_MAX is an operating system
               defined value. (On our UNIX system RAND_MAX is defined in
               /usr/include/stdlib.h as #define RAND_MAX        32767 )
               The default seed value is set to 1.

void srand(unsigned int val) : Sets the seed value used by the rand function
                               to the value given in the formal paramter.



Example programs making use of rand() and srand():

// Author: Ted Obuchowicz
// Jan. 29, 2002
// example program illustrating use of



#include <iostream>
#include <string>
#include <stdlib.h>

using namespace std;



int main()
{

for(int i = 0 ; i <= 9 ; i++)
  cout << rand() << endl;  // print out 10 random numbers between 0 and 32767
return 0;
}


EVERY time that we run the above program it will produce the values:

16838
5758
10113
17515
31051
5627
23010
7419
16212
4086


If we wish to produce different sequences every time we call the rand() function
, we
must ensure that the starting seed value is different:

// Author: Ted Obuchowicz
// Jan. 29, 2002
```

```
// example program illustrating use of
// random seed function


#include <iostream>
#include <string>
#include <stdlib.h>

using namespace std;



int main()
{
int seed_value;
cout << "Please enter a seed value" ;
cin >> seed_value;
srand(seed_value);
for(int i = 0 ; i <= 9 ; i++)
  cout << rand() << endl;  // print out 10 random numbers between 0 and 32767
return 0;
}
```

The program’s output when we run it two times with different initial seed values

is:

```
ted@flash Programs 12:38pm >random_seed
Please enter a seed value 3
17747
7107
10365
8312
20622
15796
4173
15543
24392
28207
ted@flash Programs 12:38pm >random_seed
Please enter a seed value 890
11116
32040
23590
22778
15455
20008
22032
30327
15027
27003
```

We can modify the above program to display random numbers between 0 and 6
by simply changing the :

```
 cout << rand() % 7 << endl;  // print out 20 random numbers between 0 and 6
```

```
The output is now:


0
1
3
0
3
6
4
6
6
5
0
2
0
1
2
4
5
4
6
6
2

An "upside-down" histogram which plots the frequency
of occurrence of each number is:


0       1       2       3       4       5       6
--------------------------------------------------
*       *       *       *       *       *       *
*       *       *       *       *       *       *
*               *               *               *
*                                               *



You can see that even for a very small sample space of
20 values, the distribution is quite uniform. Each of
the 7 values between 0 and 6 is equally likely to occur.


The time library:
----------------

There is a C-based library which contains several types,
and functions which are used in manipulating program time.
This library contains a function called time() which returns
a value of type time_t, which is an integral type.  If one
invokes the function time() with  argument of 0, function time()
returns the current value of time. If we cast the return type to an
unsigned int , then this value can be used as the seed value to function
srand().  Thus, new a new pseudorandom sequence of numbers can be
generated everytime a program is run, without asking the user for
a seed value:

// Author: Ted Obuchowicz
// Feb. 4, 2002
// example program illustrating use of
```

```
// time() to return a value which is used
// to seed the rand function


#include <iostream>
#include <string>
#include <stdlib.h>   // needed for rand() and srand()
#include <time.h>     // needed for time()

using namespace std;



int main()
{


unsigned int seed_value;
seed_value = (unsigned int) time(0);
srand(seed_value);


for(int i = 0 ; i <= 9 ; i++)
  cout << rand() << endl;  // print out 10 random numbers between 0 and 32767
return 0;
}


Every time we now run the program, a different sequence of numbers
is produced:

ted@flash Programs 9:54pm >random_with_time
2897
23640
12522
17160
28867
3998
26014
10937
15523
23595

ted@flash Programs 9:54pm >random_with_time
25187
16026
14157
22876
26612
20942
18259
14589
3158
156
```

```
User-Defined Functions
-----------------------

In addition to library functions C++ allows the programmer to define their
own functions.  A fuinction prototype is used to give information regarding
the return type, the function name, and the formal parameters the function
receives. The general form of a function prototype is:


return_type   function_name(type argument1_name, type argument2, ...)


where:

return_type can be any of the following:

char
short
int long
float
double
enum
struct
class ( we will learn about this type later on)
int& ( actually can be a reference to any of the above types, not only
       an int)
double * ( a pointer to any of the above type, we will learn about
           pointer type later)
void (signifies the absence of a return type)

NEITHER AN ARRAY (we will learn about array types later)NOR A FUNCTION
CAN BE SPECIFIED AS A RETURN TYPE OF A FUNCTION.  IN THESE CASES, A POINTER
TO EITHER AN ARRAY OR A FUNCTION CAN BE SPECIFIED AS THE RETURN TYPE.

Example: A function which returns the maximum of the values passed as
         parameters.

// Author: Ted Obuchowicz
// Jan. 28, 2002
// example program illustrating use of
// a user-defined function max(int, int) which
// returns the max of two numbers


#include <iostream>
#include <string>

using namespace std;

// give the definition of the function called max
// a function definition also serves as a function prototype

int max(int num1, int num2)
{
   if (num1 >= num2)
      return num1;
   else
```

```
      return num2;
}



int main()
{

int num1, num2;
cout << "Please enter two integers" ;
while (cin >> num1 >> num2)
  {
    cout << " The maximum is " << max(num1,num2) << endl;
    cout << " Please enter two integers" ;
  }

return 0;
}
```

The program output is:

```
ted@flash Programs 6:27pm >max
Please enter two integers 2 56
 The maximum is 56
 Please enter two integers
```

Note that in the above example, we gave the complete definition
of the function called max in the program source code. In C++, the
definition of the function also serves as its prototype.

Example 2: Same program as above, except that we use a HEADER file
which includes the prototype for function max, and we define
the function max in a separate file:

We can write the following and save it in a file called
max_with_header.C (for example)

```
// Author: Ted Obuchowicz
// Jan. 28, 2002
// example program illustrating use of
// a user-defined function max(int, int) which
// returns the max of two numbers


#include <iostream>
#include <string>
#include "max.h"

using namespace std;





int main()
{

int num1, num2;
```

```
cout << "Please enter two integers" ;
while (cin >> num1 >> num2)
  {
    cout << " The maximum is " << max(num1,num2) << endl;
    cout << " Please enter two integers" ;
  }

return 0;
}
```

Note that this file does not include the prototype for function max.
However the line,

```
#include "max.h"
```

tells the compiler to include the contents of the file called max.h
(which is assumed to reside in the directory from which the compiler
is invoked, alternatively, a full UNIX pathname can be specified as
in #include /home/t/ted/Coen243/Programs/max.h

The contents of the file max.h is one line which gives the prototype:

```
int max(int, int);
```

Note, that in a prototype it is not necessary to include the
names associated with the parameters, it is only necessary to
specify the type).

We now have the give the C++ code which makes up our function max.
This will be saved in a third file called max_definition.C :

```
int max(int num1, int num2)
{
   if (num1 >= num2)
      return num1;
   else
      return num2;
}
```

Finally, we can compile the two files using the command line:

```
ted@flash Programs 6:36pm >g++ -o max_with_header max_with_header.C max_definiti
on.C
ted@flash Programs 6:37pm >
```

The program runs the same as before with expected results:

```
ted@flash Programs 6:37pm >max_with_header
Please enter two integers 345 90
 The maximum is 345
```

Call-by-value versus Call-by-reference

All of the function examples we have seen thus far as examples make use of

```
what is known as the CALL-BY-VALUE PARAMETER PASSING MECHANISM.  When
call-by-value is used to pass parameters to a function, the function does
not operate on the actual parameters that the calling program passes to it.
Rather, the function makes local copies of the passed parameters and works
with these copies.  The actual parameters which are passed to it are not
affected by the function.

Everytime a function is called with the call-by-value mechanism, memory
is allocated to hold the copies of the passed parameters as well as any
variables which the function may declare itself.  Upon a function return,
this memory space is given back to the underlying operating system.  This
means that value parameters and any local variables a function may declare
have value only during a certain function invocation.  They ARE NOT
PRESERVED ACROSS FUNCTION CALLS.  A special place in main memory called
the STACK is used to hold copies of call-by-value parameters and only local
variables used by a function.

The following example explains the above concepts.  Consider a trivial
function called inc which simply increments an integer parameter by
1 :


// Author: Ted Obuchowicz
// Feb.4, 2002
// example program illustrating
// call-by-value paramter mechanism
// and how it only operates on COPIES of an argument



#include <iostream>
#include <string>

using namespace std;


// define a function which receives a single int as a call-by-value
// parameter. The function simply adds one to it. The return type is
// void meaning the function does not return a value.


void inc(int a)
{
 a = a + 1; // increment the value of the passed parameter by 1
}


int main()
{

int a = 5;
cout << "The value of a before we call the function is " << a << endl;
inc(a) ;  // call the function and pass the value a = 5 to it
cout << "Hello, I'm back from the function, this is the value of a " << a << end
l;


return 0;
}
```

The program output may surprise some of you:

```
ted@flash Programs 6:57pm >increment
The value of a before we call the function is 5
Hello, I'm back from the function, this is the value of a 5
```

The value of a is 5 BEFORE AND AFTER THE FUNCTION !!!!!
How come?

The program will exhibit the same behaviour even if we change the
name of the formal parameter to soemthing other than a. For example,
we can call it keith instead:

```
// Author: Ted Obuchowicz
// Feb.4, 2002
// example program illustrating
// call-by-value paramter mechanism
// and how it only operates on COPIES of an argument



#include <iostream>
#include <string>

using namespace std;


// define a function which receives a single int as a call-by-value
// parameter. The function simply adds one to it. The return type is
// void meaning the function does not return a value.


void inc(int keith)
{
 keith = keith + 1; // increment the value of the passed parameter by 1
}


int main()
{

int a = 5;
cout << "The value of a before we call the function is " << a << endl;
inc(a) ;  // call the function and pass the value a = 5 to it
cout << "Hello, I'm back from the function, this is the value of a " << a << end
l;


return 0;
}
```

The output is the same as before, the variable a has value of 5 before
and after we invoke the function.

```
The reason for this is that when a function is called and parameters are
passed with the call-by-value mechanism, the function makes COPIES of the
passed parameters and operates on the copies and not on the original parameters.
A special place in main memory called the STACK is used to hold the copies of
any passed parameters as well as any local variables which the function may decl
are
(our example function inc did not declare any local variables).

We can draw a picture of the contents of main memory for our first example(the
one in which the name of the parameter to function inc was called a and the name
of the variable in the main program was also a) as:


 Portion Main Memory      (  called the STACK SPACE )

-------------

-------------
      5          a   ----> this space in main memory (stack space) is to hold
------------               the integer a defined in the main program

------------

-------------

------------

------------
 5, then 6     a    -----> this space is used by the function inc
-----------                to make a COPY of the passed value parameter
                           when the function inc performs the a = a+1
                           it is working on this copy of a


The stack space in main memory is that portion of main memory which is
used to hold automatic variables declared in a main program and functions
as well as copies of any value parameters passed to a function.  Every
function invocation results in some more space being allocated from the
stack to hold the functions value parameters and any of its automatic
variables.  Every function call involves allocation of a so-called
STACK FRAME.  When a function returns, its stack frame is de-allocated.
and program control returns to the calling function.

This is the reason why our program outputs 5 as the value for variable a
both before and after the call to function inc.  All that function inc did
was to modify the copy of a (found in the stack) from 5 to 6.  The variable
a found in the heap (corresponding to the integer variable called a in the
main program) was left unchanged.

Call-by-value is the default parameter passing mechanism in C++.




So, the natural question you may be asking to yourselves is
"How do we make a function work on the actual argument and
not a copy of it?" This is where the REFERENCE data type is helpful.
```

```
We can pass a REFERENCE to an integer, and not the actual value of the
integer. If we pass a reference, then the function manipulates the actual
variable and not a copy of it.

Here is the same example as above, except that the function is changed
to indicate that a reference is being passed.  This is known as the
CALL-BY-REFERENCE parameter passing mechanism.

// Author: Ted Obuchowicz
// Feb.4, 2002
// example program illustrating
// call-by-reference parameter mechanism
// and how it only operates on  the actual parameter itself
// (and not a copy)



#include <iostream>
#include <string>

using namespace std;


// define a function which receives a single int as a  reference
// parameter. The function simply adds one to it. The return type is
// void meaning the function does not return a value.


void inc(int& a)
{
 a = a + 1; // increment the value of the passed parameter by 1
}


int main()
{

int a = 5;
cout << "The value of a before we call the function is " << a << endl;
inc(a) ;  // call the function and pass the value a = 5 to it
cout << "Hello, I'm back from the function, this is the value of a " << a << end
l;


return 0;
}


The program nopw produces the 'expected' output of:

ted@flash Programs 7:06pm >increment_reference
The value of a before we call the function is 5
Hello, I'm back from the function, this is the value of a 6
ted@flash Programs 7:06pm >



Amazing what a single & makes in a program... almost as amazing that
6 strings, some wood, a few electromagnets can produce an amazing variety
```

of pleasant tones...

Here is another example of the differences between a call-by-value and
a call-by-reference. The following program makes use of a call-by-value
function called swap which is used to interchange two integer values:

```
// Author: Ted Obuchowicz
// Feb. 4, 2002
// example program illustrating use of
// another function which uses the call-by-value
// parameter passing mechanism



#include <iostream>
#include <string>

using namespace std;


void swap(int num1, int num2)
{
 int temp ; // a local variable to temporarily hold the value
 temp = num1;
 num1 = num2;
 num2 = temp;
}



int main()
{

int x, y;

x = 10;
y = 20;

cout << "x before the swap is " << x << endl; // will be 10
cout << "y before the swap is " << y << endl; // will be 20

// call the swap function and pass the VALUES of x and y to it

swap(x,y);

cout << " x after the swap is " << x << endl; // will still be 10
cout << " y after the swap is " << y << endl; // will still be 20



return 0;
}
```

Here is the program output:

```
ted@flash Programs 6:41pm >./swap
x before the swap is 10
```

```
y before the swap is 20
 x after the swap is 10
 y after the swap is 20
ted@flash Programs 6:42pm >


The program did not swap the two values of x and y, since function swap
only swapped COPIES of the passed parameters and not the actual arguments
which were passed. The following pictures explains what the situation in
main memory is at the point function swap is called with the two value parameter
s
x and y and function swap has executed the line temp = num1.  (temp is said to
be a LOCAL variable, it is only accessible within function swap, no other functi
on
including main can access it):

--------------

-------------
   10          = x
-------------
   20          = y
-------------

-------------

-------------
  10           = num1
-------------
  20           = num2
-------------
  10           = temp
------------


after function swap has executed the two lines:

num1 = num2;
num2 = temp;


the contents of the memory locations is:

--------------

-------------
   10          = x
-------------
   20          = y
-------------

-------------

-------------
  20           = num1
-------------
  10           = num2
-------------
  10           = temp
```

Note how the variables x and y are not modified by the function swap.

If we change the function to make use of reference parameters instead of value parameters then the contents of variables x and y are actually changed by the function swap:

```
// Author: Ted Obuchowicz
// Feb. 4, 2002
// example program illustrating use of
// another function which uses the call-by-reference
// parameter passing mechanism



#include <iostream>
#include <string>

using namespace std;


void swap(int& num1, int& num2)
{
 int temp ; // a local variable to temporarily hold the value
 temp = num1;
 num1 = num2;
 num2 = temp;
}



int main()
{

int x, y;

x = 10;
y = 20;

cout << "x before the swap is " << x << endl; // will be 10
cout << "y before the swap is " << y << endl; // will be 20

// call the swap function and pass the VALUES of x and y to it

swap(x,y);

cout << " x after the swap is " << x << endl; // will now be 20
cout << " y after the swap is " << y << endl; // will now be 10



return 0;
}
```

When we run this version of the program, the output is:

```
ted@flash Programs 6:51pm > swap_reference
x before the swap is 10
```

```
y before the swap is 20
 x after the swap is 20
 y after the swap is 10
ted@flash Programs 6:51pm >
```

Suppose we wanted to count the number of times that a certain function
has been executed in a main program. Here is out first (somewhat naive)
attempt:

```
// Author: Ted Obuchowicz
// Feb.4, 2002
// example program illustrating
// lifetime of function local variables



#include <iostream>
#include <string>

using namespace std;


// define a function which receives no parameters,
//   all it does is initialize a local
// variable and add one to it every time it is called
// it then returns this value to the calling program

int count_me_up(void)
{
int counter = 0;
counter++;
return counter;
}


int main()
{


for(int tattoo_you = 0 ; tattoo_you <= 10000; tattoo_you++)
 cout <<    count_me_up() << endl;



return 0;
}
```

When we run the program, we will see 10000 lines of:

```
ted@flash Programs 7:18pm >count | more
1
1
```

```
1


and so on...


Everytime function count_me_up() is invoked, space in the stack is
allocated to store the value of the functions local variable
counter (which is initialized to 0), this variable is incremented to
1 which is returned to the calling program. Once the function has
returned, the value of the local variable is destroyed.  When the
function is entered again the process of creation , returning the
value, and the subsequent destruction of the local variable is repeated.



We can use the modifier static when we are declaring the
local variable counter in function count_me_up().  The use
of the static modifier .  The static modifier tells the compiler that
the variable is to maintain its value ACROSS function invocations
(or in other words that the lifetime of the static variable is the entire
duration of the program).

The following program now counts the number of times the function
has been called:


// Author: Ted Obuchowicz
// Feb.4, 2002
// example program illustrating
// lifetime of function local variables



#include <iostream>
#include <string>

using namespace std;


// define a function which receives no parameters,
//   all it does is initialize a local
// variable and add one to it every time it is called
// it then returns this value to the calling program

int count_me_up(void)
{
static int counter = 0;   // use a static local variable inside the function
                          // hence the lifetime of the local variable counter
                          // is the entire program duration... meaning the
                          // local variable counter will be intialized once and
                          // its value will be remembered across function invoca
tions
counter++;
return counter;
}


int main()
```

```
{


for(int tattoo_you = 0 ; tattoo_you <= 10000; tattoo_you++)
    cout << count_me_up() << endl;



return 0;
}



The program output is:


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
etc.






The above problem of counting the number of times a function
has been called may be solved in another way which makes use of
a GLOBAL variable.


// Author: Ted Obuchowicz
// Feb.4, 2002
// example program illustrating
// of a GLOBAL variable


#include <iostream>
#include <string>
```

```
using namespace std;

// define a GLOBAL integer variable
// which is seen and can be accessed by all functions

int counter = 0;

// define a function which receives no parameters,
// all it does is add 1 to the global variable

void count_me_up(void)
{
counter++; // now this refers to the global variable
cout << "Value of counter is " << counter << endl;
nt main()
{


for(int tattoo_you = 0 ; tattoo_you <= 10000; tattoo_you++)
   count_me_up();



return 0;
}
```

The program out is the 10000 lines of output as before.
The use of global variables should be TREATED WITH CAUTION.
The overzealous use of GLOBAL variables can lead to problems
caused by function sideeffects. Sideeffects can cause unexpected
problems. Suppose unbeknownst to you , some other function is accessing
and modifying the value of the global variable? This can happen when
a team of programmers is working on coding the individual functions and
there is poor communication and documentation among the team members:

Here is a rather contrived example of function sideeffects:
A program contains a count_me_up and and count_me_down function:

```
// Author: Ted Obuchowicz
// Feb.4, 2002
// example program illustrating
// of a GLOBAL variable


#include <iostream>
#include <string>

using namespace std;

// define a GLOBAL integer variable
// which is seen and can be accessed by all functions

int counter = 0;

// define a function which receives no parameters,
//   all it does is initialize a local
```

```
// variable and add one to it every time it is called
// it then returns this value to the calling program


void count_me_up(void)
{
counter++; // now this refers to the global variable
cout << "Value of counter is " << counter << endl;
}


void count_me_down(void)
{
counter--;
cout << "Value of counter is " << counter << endl;
}


int main()
{


for(int tattoo_you = 0 ; tattoo_you <= 10000; tattoo_you++)
 {
   count_me_up();
   count_me_down();
 }



return 0;
}



Now the output alternates between 1 and 0 :


Value of counter is 1
Value of counter is 0
Value of counter is 1
Value of counter is 0
Value of counter is 1
Value of counter is 0
Value of counter is 1
Value of counter is 0
Value of counter is 1
Value of counter is 0
Value of counter is 1
Value of counter is 0
Value of counter is 1
Value of counter is 0
Value of counter is 1
Value of counter is 0
Value of counter is 1
Value of counter is 0
Value of counter is 1
Value of counter is 0
Value of counter is 1
```

```
Value of counter is 0
Value of counter is 1
Value of counter is 0


The extern keyword:
-------------------

Suppose we had written our above program in three separate files:
one for the main program, one for the function count_me_up, and one
more for the function count_me_down as in:


file 1 containing the main program:

// Author: Ted Obuchowicz
// Feb.4, 2002
// example program illustrating
// of a GLOBAL variable


#include <iostream>
#include <string>

using namespace std;

// define a GLOBAL integer variable
// which is seen and can be accessed by all functions

int counter = 0;


// declare the two prototypes since they are
// defined elsewhere

void count_me_up(void);
void count_me_down(void);

int main()
{


for(int tattoo_you = 0 ; tattoo_you <= 10000; tattoo_you++)
 {
   count_me_up();
   count_me_down();
 }



return 0;
}



file 2 containing the definition of the count_me_up function:

#include <iostream>
```

```
void count_me_up(void)
{

extern int counter; // the extern tells the compiler that this variable
// was declared "somewhere else"

counter++; // now this refers to the global variable
cout << "Value of counter is " << counter << endl;
}


file 3 containing the definition  of the count_me_down function:

#include <iostream>

void count_me_down(void)
{

extern int counter;

counter--; // now this refers to the global variable
cout << "Value of counter is " << counter << endl;
}


The two lines in the functions :

extern int counter;

are basically promises to the compiler telling it  :
"Look buddy, I've defined this variable somewhere else
so don't complain about it"

NOTE: IT WOULD BE AN ERROR TO DO SOMETHING LIKE:

void count_me_down(void)
{

extern int counter = 1; // THIS IS INVALID

counter--; // now this refers to the global variable
cout << "Value of counter is " << counter << endl;
}



SCOPE and STORAGE CLASSES
--------------------------

The following program does not compile:

// Author: Ted Obuchowicz
// Feb. 7, 2002
// example program illustrating use of
// illegal access to an out-of-scope object


#include <iostream>
#include <string>
```

```
using namespace std;


void out_of_scope(int mick, int keith)
{
 cout << "Hello from function out_of_scope" << endl;
 cout << mick << keith << endl;
 mick = 10;
 keith= 20;
 cout << mick << keith << endl;
 int ron = 500;
 cout << ron << endl;
}



int main()
{

int a = 1;
int b = 2;

out_of_scope(a,b);
cout << mick << keith << endl;
cout << ron << endl;

return 0;
}

These are the errors reported by the g++ compiler:

ted@flash Programs 7:48pm >g++ -o does_not_compile does_not_compile.C
does_not_compile.C: In function ‘int main()’:
does_not_compile.C:33: ‘mick’ undeclared (first use this function)
does_not_compile.C:33: (Each undeclared identifier is reported only once
does_not_compile.C:33: for each function it appears in.)
does_not_compile.C:33: ‘keith’ undeclared (first use this function)
does_not_compile.C:34: ‘ron’ undeclared (first use this function)


The compiler is complaining about  lines 33 and 34 in the program which
correspond to the following lines of code in function main():

int main()
{

int a = 1;
int b = 2;

out_of_scope(a,b);
cout << mick << keith << endl;  // this is line 33 of the program
cout << ron << endl;            // this is line 34 of the program

return 0;
}

The compiler is saying that the identifiers mick, keith and ron are undeclared.
These attempts to access these variables are illegal since mick, keith, and ron
```

```
are said to be out of scope for function main().

mick, keith and ron are said to be LOCAL to function out_of_scope().
This means that they are only ACCESSIBLE inside the function out_of_scope().
Anywhere else in the program they are not visible (accessible).

C++ has several rules concerning the SCOPE of objects in a program.
We will now examine these rules in detail.  We first have to define
what we mean by a BLOCK.

Definition: In C++, a BLOCK is simply a list of statements enclosed
            within the curly braces {}. For example

{
// this is the beginning of a block
int a = 1;
// this is the end of the block
}

If we follow this definition, a function body is a block.

Scope rule : A local object can only be used in the block and any other
             nested blocks of the block in which it has been defined in.

Here is a simple example of a block within a main function:

// Author: Ted Obuchowicz
// Feb 6, 2002
// example program illustrating
// scope of variables


#include <iostream>
#include <string>

using namespace std;



int main()
{

int mick = 5;
cout << "The value of mick in main is  " << mick <<
        ", mick's address is " << &mick << endl;

// define a block with another different mick integer in it
{
int mick = 28;
cout << "The value of mick in the inner block is " << mick <<
        ", mick's address is " << &mick << endl;
}

return 0;
}


The program output is:
```

```
The value of mick in main is  5, mick's address is 0xffbef60c
The value of mick in the inner block is 28, mick's address is 0xffbef608


You can see how two different main memory locations are used to store
the two micks .

As another example,here is a program which contains a block which contains a nes
ted
block:

// Author: Ted Obuchowicz
// Jan. 7, 2002
// example program illustrating use of
// scope and nested blocks


#include <iostream>
#include <string>

using namespace std;



int main()
{

int i = 0;
cout << " i = " << i << " address of i = " << &i << endl;

// define a new block with a new integer i
{
int i = 6; // inside this block, the other i with value 0 is HIDDEN
cout << " i = " << i << " address of i = " << &i << endl;
{
// define a new block nested inside
int j = 4;
cout << " j = " << j << " address of j = "  << &j << endl;
// in this nested block, the i with value 6 is still visible
cout <<  " i = " << i << " address of i = " << &i << endl;
}
}


// now in here whenever we refer to i , it is the i with value 0
// since it is back in main()'s scope.

cout << " i = " << i << " address of i = " << &i << endl;

return 0;
}


Here is the output:

ted@flash Programs 8:16pm >nested_blocks
 i = 0 address of i = 0xffbef5fc
 i = 6 address of i = 0xffbef5f8
 j = 4 address of j = 0xffbef5f4
```

```
 i = 6 address of i = 0xffbef5f8
 i = 0 address of i = 0xffbef5fc
ted@flash Programs 8:16pm >
```

There are actually two different variables called i in this program.
There is the i defined in main with value of 0 which is stored
in main memory at the hexadecimal address of 0xffbef5fc, then there
is the i defined in the block which has value 6 and is stored in location
0xffbef5f8. Note how this second i is also visible in the nested block.

GLOBAL SCOPE

Objects not declared in any statement block are said to be of GLOBAL
SCOPE. Objects of global scope are visible by all functions.

Here is a simple example:

```
// Author: Ted Obuchowicz
// Feb.6, 2002
// example program illustrating use of
// a global scope object


#include <iostream>
#include <string>

using namespace std;


int visible_everywhere = 55;

void f1(void)
{
 cout << "value of visible_everywhere = " << visible_everywhere <<
          " address = " << &visible_everywhere << endl;
}

void f2(void)
{
cout << "value of visible_everywhere = " << visible_everywhere <<
          " address = " << &visible_everywhere << endl;
}


int main()
{

cout << "value of visible_everywhere = " << visible_everywhere <<
          " address = " << &visible_everywhere << endl;
f1();
f2();

return 0;
}
```

The output is:

```
ted@flash Programs 8:31pm >global
value of visible_everywhere = 55 address = 0x3c1b0
value of visible_everywhere = 55 address = 0x3c1b0
value of visible_everywhere = 55 address = 0x3c1b0
ted@flash Programs 8:32pm >
```

Of course, if any of the functions redefine the object, then this redefinition overrides the global one within the scope of the redefining function.

For example, suppose function f1() declares its own local version of integer variable visible_everywhere with value of 4567:

```
// Author: Ted Obuchowicz
// Feb.6, 2002
// example program illustrating use of
// a global scope object and a function which
// redefines its own local version of it


#include <iostream>
#include <string>

using namespace std;


int visible_everywhere = 55;

void f1(void)
{
 int visible_everywhere = 4567;
 cout << "value of visible_everywhere = " << visible_everywhere <<
         " address = " << &visible_everywhere << endl;
}

void f2(void)
{
cout << "value of visible_everywhere = " << visible_everywhere <<
         " address = " << &visible_everywhere << endl;
}


int main()
{

cout << "value of visible_everywhere = " << visible_everywhere <<
         " address = " << &visible_everywhere << endl;
f1();
f2();

return 0;
}
```

The program output is now:

```
ted@flash Programs 8:35pm >global2
```

```
value of visible_everywhere = 55 address = 0x3c1b0
value of visible_everywhere = 4567 address = 0xffbef59c
value of visible_everywhere = 55 address = 0x3c1b0
You have new mail.
ted@flash Programs 8:36pm >
```

Note how there are now two different variables called visible_everywhere
stored in separate and distinct memory locations.

When we are inside function f1(), the local definition HIDES the global one.

What if function f1() wanted to access the global visible_everywhere variable
instead of its local version? C++ provides a mechanism to do so through the
use of the SCOPE RESOLUTION OPERATOR ::

Here is a different version of the previous program which makes use of the
scope resolution operator to resolve to the versions of the visible_everywhere
variable seen in function f1() :

```
// Author: Ted Obuchowicz
// Feb.6, 2002
// example program illustrating use of
// a global scope object and a function which
// redefines its own local version of it and
// makes use of the :: operator to refer to the global variable


#include <iostream>
#include <string>

using namespace std;


int visible_everywhere = 55;

void f1(void)
{
 int visible_everywhere = 4567;
 cout << "value of local visible_everywhere = " << visible_everywhere <<
         " address = " << &visible_everywhere << endl;

 cout << "value of the global visible_everywhere = " << ::visible_everywhere <<
          " address = " << & ::visible_everywhere << endl;
}

void f2(void)
{
cout << "value of visible_everywhere = " << visible_everywhere <<
         " address = " << &visible_everywhere << endl;

// note: f2 can also refer to the global visible_everywhere using thr
// ::visible_everywhere
}


int main()
```

```
{

cout << "value of visible_everywhere = " << visible_everywhere <<
         " address = " << &visible_everywhere << endl;
f1();
f2();

return 0;
}


This is the output:

ted@flash Programs 10:04pm >global3
value of visible_everywhere = 55 address = 0x3c278
value of local visible_everywhere = 4567 address = 0xffbef59c
value of the global visible_everywhere = 55 address = 0x3c278
value of visible_everywhere = 55 address = 0x3c278
```

Note that the scope resolution operator can also be applied to functions
and user-defined types. This is useful in class definitions. Classes
will be studied in detail in your COEN 244 course and you will once again
come across the scope resolution operator.

STORAGE CLASSES

The storage class of a variable refers to how the compiler allocates space
in main memory to hold the value of a variable. The storage class also affects
the 'lifetime' or 'persistance' of a variable.

We will now explain the more commonly used storage classes in C++.

```
automatic:  automatic variables are those declared inside a function
            definition, (recall that main() is also a function).  Memory
            space for automatic variables is allocated when program execution
            enters the function or block in which these variables are defined.
The
            memory is given back when the function returns to the calling progra
m or
            when the program leaves the defining block.  Function parameters are
 considered
            to be automatic variables.  Automatic variables are stored in a spec
ial portion
            of main memory called the STACK.
```

One may use the keyword auto to explicitly indicate that the storage class of
an automatic variable. Since the use of the auto keyword is only allowed to be
applied to variables that are already automatic, it is actually rarely ever used
by programmers:

```
int keith_richards()
{
 auto int guitar_player;
 // some more code for the function
```

```
}



register:  C++ allows the use of the register keyword when declaring automatic
           variables.  This is a 'request' to the compiler asking it to use a
           special location inside the computer Central Processing Unit (CPU)
           instead of the stack area in main memory to store a certain variable.
           The motivation behind this is that it takes less time to access a
           variable in a CPU register than it does to access a main memory locat
ion.
           For example, inside a for loop the value of the loop index variable
           is accessed in every loop iteration, we can ask the compiler to use a
           CPU register to hold the loop index by doing something like:

         for( register int loop_index = 0 ; loop_index <= 9 ; loop_index++)
         {
           // do something inside the loop
         }

Sometimes the compiler cannot honor the request for a register automatic
variable. For example, maybe there are no more available registers to be used.
Most modern compilers are smart enough to deduce on their own when an automatic
variable should be stored in a register so programmers rarely bother to use
the register keyword.




The reason why the loop index variable cannot be accessed outside of the
loop shold be very apparent now.  It only exists within the duration
of the for loop, once the loop terminates, the automatic variable which
was used to hold the loop index no longer exists.

static : variables of storage class static are in existence for the entire progr
am
         duration.  The compiler sets aside a fixed size of main memory (distinc
t
         from the stack area) to use for a program's static variables.  If a sta
tic
         variable is not explicitly given an initial value by the programmer, th
e compiler
         gives it an initial value of 0.  We have already seen the use of a stat
ic variable
         in our example program which used a function which declared a static va
riable to
         keep track of the number of times the function has been called.




DEFAULT ARGUMENTS
-----------------

Normally, one must invoke a function  with the exact number of arguments
as the function prototype specifies. If one tries to call a function with a fewe
r
or greater number of arguments, the compiler will report an error as in the
following (incorrect) program:
```

```
     1  // Author: Ted Obuchowicz
     2  // Feb. 11, 2002
     3  // example program illustrating use of
     4  // incorrect function call
     5
     6
     7  #include <iostream>
     8  #include <string>
     9
    10  using namespace std;
    11
    12
    13  int do_something(int x, int y, int z)
    14  {
    15    return ( x - y - z);
    16  }
    17
    18  int main()
    19  {
    20
    21  int a, b, c;
    22  a=b=c= 5;
    23  cout << do_something(a,b) << endl ; // will cause compile time error
    24  return 0;
    25
    26  }
    27
    28
```

```
(line numbers corresponding to the source code are included for your convenience
)
The compiler output is:


ted@flash Programs 6:27pm >g++ -o incorrect_number_arguments incorrect_number_ar
guments.C
incorrect_number_arguments.C: In function `int main()':
incorrect_number_arguments.C:14: too few arguments to function `int do_something
(int, int, int)'
incorrect_number_arguments.C:23: at this point in file

line 14 is the opening { of the function definition, line 23 is the line in the
program source code which contains the function invocation.

C++ allows a programmer to provide for DEFAULT values of parameters. If the user
does not supply any values in a function invocation, and if default values have
been provided, then the unspecified parameters will take on the specified defaul
t
values.

Here is the same program, where we specify a default value for the last paramete
r:

// Author: Ted Obuchowicz
// Feb. 11, 2002
// example program illustrating use of
// incorrect function call
```

```
#include <iostream>
#include <string>

using namespace std;


int do_something(int x, int y, int z=0) // if we don’t provide a third argument
{                                        // it will take on default value of 0
  return ( x - y - z);
}

int main()
{

int a, b, c;
a=b=c= 5;
cout << do_something(a,b) << endl ; // will print the value 0
return 0;

}
```

It is important to remember that when defining a function with default
parameters, the default parameters be defined AFTER any required parameters
as in:

```
void foo_bar(char x, float d, int i = 0, char y = ’z’ , bool yes_no = false)
{
 cout << " x = " << x << endl << " d = " << d << endl << " i = " << i
      << endl << " y = " << y << endl << " yes_no = " << yes_no << endl;
}
```

we can invoke this function as:

```
foo_bar(’a’, 4.56) // x takes on ’a’, d takes on 4.56, and  the
                   // remaining three take on the default values
```

Default values can be overridden by proving a value at function
invocation as in:

```
// Author: Ted Obuchowicz
// Feb. 11, 2002
// example program illustrating use of
// overriding default values


#include <iostream>
#include <string>

using namespace std;


void foo_bar(char x, float d, int i = 0, char y = ’z’ , bool yes_no = false)
{
 cout << " x = " << x << endl << " d = " << d << endl << " i = " << i
```

```
      << endl << " y = " << y << endl << " yes_no = " << yes_no << endl;
}


int main()
{

char first = 'c';
float second = 1.345;
int third = 55;
char fourth = 'h';
bool fifth = true;

foo_bar(first,second,third) ; // the LAST TWO parameters will take on default va
lues

return 0;

}

The output is:

ted@flash Programs 6:48pm >default_arguments2
 x = c
 d = 1.345
 i = 55
 y = z
 yes_no = 0

You cannot "skip over" a default parameter (using a comma for example)
and specify  overriding values for any remaining default values as in:

foo_bar(first,second,  , fourth, fifth)


This will result in a compile time error of the sort:

ted@flash Programs 6:54pm >g++ -o default_arguments3 default_arguments3.C
default_arguments3.C: In function `int main()':
default_arguments3.C:29: parse error before `,'

The general rule for listing default parameters in a function
and the rule for providing arguments at invocation time can
be summarized as:

to list default parameters in a function:

return_type function_name( type first_required_parm,  type second_required_parm,
 ...
                           type last_required_parm, type first_default, type
                           second_default, type third_default, ... last default)



For example:

void snafu(int x, char y, float f, int z = 0, char c = 'a', float f2 = 45.67)
{
....
```

```
// body of the function
}

legal invocations would be:

snafu(56, 'f', 1.23) ; // x takes on 56, y takes on 'f', , f takes on 1.23
                       // and the remaining take on default values

snafu(2, 'z', 345.67, 10) ; // x takes on 2, y takes on 'z', f takes on 345.67
                            // z is overridden with 10,  c and f2 assume default
s

snafu(2, 'z', 345.67, 10, 'c') // only the last one takes on the default value
```

FUNCTION OVERLOADING

C++ allows a programmer to overload functions. Overloading is a method,
whereby two or more functions perform different tasks but share a common
name. Overloading is commonly used when similar tasks need to be performed
on different data types. Rather than creating a uniquely named function
for every data type that the functions are to be invoked with, the programmer
can supply different definitions for one commonly named function. The definitio
ns
must vary in their parameter list. The compiler then determines which definitio
n
of the overloaded function is to be used by examining the argument list and find
ing
a definiton which BEST matches the passed parameters. The following example
first illustrates how we would handle similar tasks (on different data types)
without the use of overloading. Next, the program is rewritten to exploit the
availability of function name overloading in C++.

```
nclude <iostream>
#include <string>

using namespace std;

struct  some_struct
{
 int first;
 float second;
};



void swap_integers(int& a, int& b)
{
 int temp;
 temp = b;
 b = a;
 a = temp;
}
```

```
void swap_some_structs( some_struct& a, some_struct& b)
{
 some_struct temp;
 temp.first = b.first;
 temp.second = b. second ; // make a copy of the second struct

 b.first = a.first;
 b.second = a.second;

 a.first = temp.first;
 a.second = temp.second;
}


int main()
{

int mick, keith;
mick = 10;
keith = 20;

cout << "mick = " << mick << "keith = " << keith << endl;
swap_integers(mick, keith);
cout << "mick = " << mick << "keith = " << keith << endl;


some_struct struct1, struct2;
struct1.first =  5;
struct1.second = 5.5;

struct2.first =  6;
struct2.second = 6.6;

cout << struct1.first << " " << struct1.second << " " << struct2.first << " "
     << struct2.second << endl;

swap_some_structs(struct1, struct2);


cout << struct1.first << " " << struct1.second << " " << struct2.first << " "
     << struct2.second << endl;

return 0;
}
```

The output of the program is:

```
ted@brownsugar Programs 12:16pm >no_overloading
mick = 10keith = 20
mick = 20keith = 10
5 5.5 6 6.6
6 6.6 5 5.5
ted@brownsugar Programs 12:16pm >
```

If we overload the function swap with two different definitions (one for swapping two integers, the other for swapping two some_structs),

```
then  we obtain the following:

// Author: Ted Obuchowicz
// Feb  11, 2002
// example program illustrating use of
// non-overloaded functions with different names
// but they 'more or less' do the same things on different types of
// data


#include <iostream>
#include <string>

using namespace std;

struct  some_struct
{
 int first;
 float second;
};


// define a swap function which takes on integer arguments

void swap(int& a, int& b)
{
 int temp;
 temp = b;
 b = a;
 a = temp;
}

// overload the swap function to handle 'some_struct' arguments instead
// of integer arguments

void swap( some_struct& a, some_struct& b)
{
 some_struct temp;
 temp.first = b.first;
 temp.second = b. second ; // make a copy of the second struct

 b.first = a.first;
 b.second = a.second;

 a.first = temp.first;
 a.second = temp.second;
}


int main()
{

int mick, keith;
mick = 10;
keith = 20;

cout << "mick = " << mick << "keith = " << keith << endl;
swap(mick, keith); // compiler will choose the definition which has two integers
                   // in the argument list
```

```
cout << "mick = " << mick << "keith = " << keith << endl;


some_struct struct1, struct2;
struct1.first =  5;
struct1.second = 5.5;

struct2.first =  6;
struct2.second = 6.6;

cout << struct1.first << " " << struct1.second << " " << struct2.first << " "
     << struct2.second << endl;

swap(struct1, struct2); // compiler will choose the definition of swap
                        // which has the two some_struct as arguments


cout << struct1.first << " " << struct1.second << " " << struct2.first << " "
     << struct2.second << endl;

return 0;
}
```

The program output is:

```
ted@brownsugar Programs 12:16pm >overloading
mick = 10keith = 20
mick = 20keith = 10
5 5.5 6 6.6
6 6.6 5 5.5
```

Overloading improves program readability and makes writing programs easier since the programmer does not have to invent different names for functions which perform similar jobs.

Consider the following test program written by Joseph Philip:

```
 // A test program
 // file : t.C
 #include <iostream>
 int foo ( int a  ) {
    cout << "int foo called." << endl;
    return a ;
 }

 float foo ( int  a ) {

    cout << "other foo called." << endl;
    return --a;
 }

 int main (void ) {
```

```
    float s = foo ( 7) ;
   return 0;
 }

 This does not compile.  I get
  coen243]$ g++ ./t.C
 ./t.C: In function ‘float foo (int)’:
 ./t.C:12: new declaration ‘float foo (int)’
 ./t.C:3: ambiguates old declaration ‘int foo (int)’
  coen243]$
```

In Bjarne Stroustrup’s book , The C++ Programming Language 3rd ed. (Stroustrup was the developer of the C++ language) he states on page 151:

"Return types are not considered in overload resolution."
He explains the method the compiler uses to determine which overloaded definitio
n
of a function to use as follows:

"When a function f is called, the compiler must figure out which of the function
s
to with the name f is to be invoked. This is done by comparing the types of the actual arguments with the types of the formal argumments of all functions called f. The idea is to invoke the function that is the BEST match on the arguments an
d give
a compile-time error if no function is a best match. ... A series of criteria ar
e tried
in order:

[1] Exact match; that is match using no or only trivial conversion.
[2] Match using promotions; that is integral promotions (bool to int, char to int, short to int and their unsigned counterparts and float to double
[3] Match using standard conversions (for example int to double, etc)

Overloading relies on a relatively complicated set of rules, and occasionally a programmer will be surprised by which function will be called."

To reiterate, the process of determining which function to invoke when more than one has the same name is called FUNCTION OVERLOAD RESOLUTION. With overload
ed
functions, the compiler examines the list of actual arguments passed to the over
loaded
function and then chooses the function whose formal argument list BEST matches t
he
passed argument list. Sometimes the compiler is not able to resolve the functio
n
name overloading and will report an error:

```
#include <iostream>
#include <string>

using namespace std;



// define a swap function which takes on integer arguments
```

```
void swap(int& a, int& b)
{
 int temp;
 temp = b;
 b = a;
 a = temp;
}

// overload the swap function to handle double1 arguments instead
// of integer arguments

void swap( double& a, double& b)
{
 double temp;
 temp = b;
 b = a;
 a = temp;
}



int main()
{

int mick, keith;
mick = 10;
keith = 20;

cout << "mick = " << mick << "keith = " << keith << endl;
swap(mick, keith); // compiler will choose the definition which has two integers
                   // in the argument list
cout << "mick = " << mick << "keith = " << keith << endl;


double ron, charlie;
ron = 100.0;
charlie = 200.0;
swap(ron,charlie) ; // compiler will choose the swap with two double
                    // arguments in the parameter list

int bill = 2;
double stu = 3.14159;

swap(bill, stu) ; // compiler is UNABLE to resolve which overloaded definition
                  // to use

return 0;
}



The compiler reports the following errors:

bad_overload.C: In function ‘int main()’:
bad_overload.C:55: call of overloaded ‘swap (int &, double &)’ is ambiguous
bad_overload.C:13: candidates are: void swap(int &, int &) <near match>
bad_overload.C:24:                 void swap(double &, double &) <near match>
```

The compiler is referring to the swap(bill, stu). It is unable to find
a best match since there are two possible candidates. The compiler could
choose to convert the double variable called stu to an integer and invoke the
integer version of swap, or alternatively the compiler could choose to
promote the integer bill to its double version and invoke the swap which
uses two doubles as arguments. When there is ambiguity, the compiler reports
it as an error.

Function overload resolution is a complicated and tricky matter, 14 pages
are allocated to the subject in "The Annotated C++ Reference Manual".

RECURSION

Recursion is a very powerful concept in programming. It refers to
when function calls itself. Here is a very trivial but simple
example which illustrates the main concepts involved in recursive function
calls. It is taken from a very nice book on C++ written by Stephen Prata:

```
// Author:  S. Prata
// Feb. 11, 2002
// example program illustrating use of
// of a contrived example of function recursion
// taken from the book C++ Primer Plus, 3rd Edition
// by Stephen Prata, Sams Publishing, 1998, p. 281


#include <iostream>
#include <string>

using namespace std;


void countdown(int n)
{
 cout << "Counting down ... " << n << endl;
 if (n > 0 )
   countdown(n-1); // make a recursive call to function countdown
 cout << "Welcome back to version " << n << " of countdown. " << endl;
}


int main()
{

countdown(5);


return 0;
}
```

The program output is:

```
Counting down ... 5
Counting down ... 4
Counting down ... 3
```

```
Counting down ... 2
Counting down ... 1
Counting down ... 0
Welcome back to version 0 of countdown.
Welcome back to version 1 of countdown.
Welcome back to version 2 of countdown.
Welcome back to version 3 of countdown.
Welcome back to version 4 of countdown.
Welcome back to version 5 of countdown.


Let us trace the sequence of calls to function countdown starting
from the initial call from main().

Every call to function countdown() causes a new stack frame to be allocated
to the function.  The functions value arguments and any local arguments are
copied into this stack frame.  When a function either executes an explicit
return statement, or when it reaches its closing } , the function returns
to whoever called it.  Program control then resumes with the caller continuing
from where the caller left off.

Here is a pictorial representation of what the stack frame would look
like when main calls countdown(5):

 frame allocated for invocation of countdown(5)


----------------
n = 5              stack frame allocated for invocation of countdown(5)


---------------

when countdown(5) begins execution it prints its
cout << "Counting down ... " << n << endl;
message and then makes a recursive call to countdown passing value argument of 4
.
A new stack frame is allocated and the function countdown begins execution from
its first line of count.  The stack frame looks like:



----------------
n = 4              stack frame allocated for invocation of countdown(4)


----------------
n = 5              stack frame allocated for invocation of countdown(5)


---------------


countdown(4) prints its "counting down... 4" message and executes its
conditional if statement, which results in another recursive call to
counbtdown with parameter n=3.  A new stack frame is allocated:
```

```
----------------
n = 3             stack frame allocated for invocation of countdown(3)


----------------
n = 4             stack frame allocated for invocation of countdown(4)


----------------
n = 5             stack frame allocated for invocation of countdown(5)


---------------
```

Again, countdown(3) begins execution and prints its starting message and
then makes a call to countdown(2), a new stack frame is allocated and
countdown(2) begins execution:

```
----------------
n = 2             stack frame allocated for invocation of countdown(2)


----------------
n = 3             stack frame allocated for invocation of countdown(3)


----------------
n = 4             stack frame allocated for invocation of countdown(4)


----------------
n = 5             stack frame allocated for invocation of countdown(5)


---------------
```

countdown(2) prints its first message and then makes yet another call
to countdown with value n= 1. A new stack frame is allocated and the
function begins execution:

```
----------------
n = 1             stack frame allocated for invocation of countdown(1)


----------------
n = 2             stack frame allocated for invocation of countdown(2)


----------------
n = 3             stack frame allocated for invocation of countdown(3)


----------------
n = 4             stack frame allocated for invocation of countdown(4)
```

```
----------------
n = 5              stack frame allocated for invocation of countdown(5)


---------------

countdown(1) prints its counting down ... 1 message and then calls
countdown with n = 0:



----------------
n = 0              stack frame allocated for invocation of countdown(0)


----------------
n = 1              stack frame allocated for invocation of countdown(1)


----------------
n = 2              stack frame allocated for invocation of countdown(2)


----------------
n = 3              stack frame allocated for invocation of countdown(3)


----------------
n = 4              stack frame allocated for invocation of countdown(4)


----------------
n = 5              stack frame allocated for invocation of countdown(5)


---------------


countdown(0) prints its Counting down ... 0 message, since it's local value
of n (0) is not greater than 0, the if statement is false and countdown(0)
prints its Welcome back to version  0 message and then reaches its closing
} which causes a return (down the stack frame) to countdown(1).  When a function
return , its stack frame is DEALLOCATED. The stack frame after countdown(0) retu
rns
is:

----------------
n = 1              stack frame allocated for invocation of countdown(1)


----------------
n = 2              stack frame allocated for invocation of countdown(2)


----------------
n = 3              stack frame allocated for invocation of countdown(3)


----------------
```

```
n = 4              stack frame allocated for invocation of countdown(4)


----------------
n = 5              stack frame allocated for invocation of countdown(5)


---------------



Program execution now resumes with countdown(1) continuing from after
the point where it made a recusive call. It prints its
Welcome back to version 1 and then returns.  It’s stack frame is deallocated:

----------------
n = 2              stack frame allocated for invocation of countdown(2)


----------------
n = 3              stack frame allocated for invocation of countdown(3)


----------------
n = 4              stack frame allocated for invocation of countdown(4)


----------------
n = 5              stack frame allocated for invocation of countdown(5)


---------------

Control returns to function countdown(2).  It  uses its local value of n
in the current top of stack frames when it returns to the
"Welcome back to version " << n  portion of code, then it returns...
and its stack frame is deallocated:

----------------
n = 3              stack frame allocated for invocation of countdown(3)


----------------
n = 4              stack frame allocated for invocation of countdown(4)


----------------
n = 5              stack frame allocated for invocation of countdown(5)


---------------



Eventually, control returns to countdown(5), it prints its welcome
back to version 5 and control then returns to main() (since main
intially called countdown with n = 5).  At this point, the program
stops.


Here is a very crude ASCII text picture of the Call graph foor
```

```
countdown(5):

  main()
   | /\
  \/  |
Count (5)
   | /\
  \/ |
count(4)
  |  /\
  \/ |
count(3)
  |  /\
  \/  |
count(2)
  |  /\
  \/ |
count(1)
  |  /\
  \/  |
count(0)


There are 6 invocations of function count.


Problems which have recursive solutions generally have the two
following properties:

1) there exists a 'base case' (or terminating case) with a trivial solution,

2) in the other  cases, it is possible to consider the problem which is
   similar to the original problem but with simpler (smaller) arguments.


In general, we can express  recursive solutions along the form of:

if (some terminating condition is satisfied)
   return ;
else
   make a recursive call with "simpler" arguments;

We will now solve the factorial program using a recursive function:

// Author: Ted Obuchowicz
// Feb. 11, 2002
// example program illustrating use of
// recursion to compute factorial
// n ! = 1 if n = 0
//     = n * (n-1)! if n >= 1


#include <iostream>
#include <string>

using namespace std;


int factorial(int n)
```

```
{
 if (n == 0)
   return 1;
 else
   return n * factorial(n-1);
}

int main()
{

cout << "Enter a positive integer ";
int n;
cin >> n;
cout << n << " ! = " << factorial(n);
return 0;
}


The following was taken from:

http://www.cs.princeton.edu/~lworthin/126/precepts/recursion.html

You can trace through a recursive function be repeatedly 'pasting' in its code
        whenever it calls itself. Example (factorial function):

        int f (int n)
        {
            if (n == 0) return 1;
            return n * f(n - 1);
        }

        Let's trace through the function with the pasting technique for n=4:

        f(4)
        {
           if (4 == 0) return 1;   // n is 4 here
           return 4 * f(3)
                      {
                         if (3 == 0) return 1;  // we just 'pasted' the code her
e
                         return 3 * f(2)
                                    {
                                       if (2 == 0) return 1;  //n is 2 here
                                       return 2 * f(1)
                                                  {
                                                     if (1 == 0) return 1;
                                                     return 1 * f(0)
                                                                {
                                                                   if (0 == 0) r
eturn 1;

        Now what? Well, we've finally gotten to the base case. We don't need to
'paste'
        again, because we don't get to that line. We just return 1 here. (You ca
n see why
        base cases are so important - without them, we'd never finish 'pasting.'
) We
        haven't finished yet, though...
```

```
        f(0) just finished its execution. There’s no mystery about what happens
next.
        Whenever any function finishes (returns), its return value (if any) is r
eturned to
        the calling function and execution resumes there. Make sure you understa
nd this
        - it’s critical.

        So, the value 1 is returned. In the function f(1), we were executing the
 line
        "return 1 * f(0)" when f(0) was called. So, we continue: f(1) will retur
n 1 * 1 to
        its current function: f(2). Then, f(2) will return 2 * f(1), which is 2
* 1. Next, this
        value, 2, is returned to f(3), so f(3) finishes by executing the line: 3
 * f(2). The
        number returned by f(2) is 2. Thus, 3 * 2 is returned by f(3), and we’re
 done -
        with the correct result, 3.

Our next example uses recursion to calculate the n’th Fibonacci number
in the famed sequence:

// Author: Ted Obuchowicz
// Feb. 19, 2002
// example program illustrating use of
// recursion to find the n’th Fibonacci number

#include <iostream>
#include <string>

using namespace std;


int fibonacci(int n)
{
 if ( n <= 2)
   return 1;
 else
   return fibonacci(n-1) + fibonacci(n-2);
}



int main()
{

int number;
cout << "Enter the number of the sequence you want" ;
cin >> number;

cout << "The " << number << " th in the Fibonacci sequence is  " << fibonacci(nu
mber) << endl;

return 0;
}
```

```
THE TOWERS OF HANOI
-------------------

Here is the solution to the towers of hanoi.
Add explanation later.

// Author: Ted Obuchowicz
// Feb. 15, 2002
// example program illustrating use of
// recursion to solve the Towers of Hanoi problem



#include <iostream>
#include <string>

using namespace std;

unsigned long int counter = 0;

void hanoi_tower(int n, int from, int to, int temp)
{
   if ( n > 0)
   {
    counter++;
    hanoi_tower(n-1, from, temp, to   );
    cout << from << "  " << to <<  endl;
    hanoi_tower(n-1, temp, to, from  );
   }
}

int main()
{
int n;
cout << "enter number of disks " ;
cin >> n;

hanoi_tower(n, 1,3,2);
cout << "it took " << counter << " moves " << endl;


return 0;
}


The function call graph explains how the recursive solution for
the Twoers of Hanoi problem works.  The graphs below are missing
the -----> and <------   arrows, but it is left as an exercise to the
reader to fill them in and to understand the call graphs :

enter number of disks: 2



Hanoi(2, 1, 3, 2)    Hanoi(1, 1, 2, 3)     Hanoi(0, - , -, -)
                                           1 ==>  2   (move #1)
                                           Hanoi(0, - , -, -)
```

```
                     1 ==>  3  ( move #2)


                     Hanoi(1, 2, 3, 1)     Hanoi(0, - , -, -)
                                           2 ==>  3 ( move #3)
                                           Hanoi(0, - , -, -)




it took 3 moves





void hanoi_tower(int n, int from, int dest, int temp)
//   note here I use the term "dest" for the destination peg
//   instead of "to" since it gets confusing in class when
//   we say the "to" argument is "2" since the word "to" is pronounced
//   the same as the number "two" and in previous years this led (zeppelin)
//   to confusion

{
   if ( n > 0)
   {
    hanoi_tower(n-1, from, temp, dest   );
    cout << from << " ==>  " << dest <<  endl;
    hanoi_tower(n-1, temp, dest, from  );
   }
}


enter number of disks  : 3


  Hanoi(3, 1, 3, 2)  Hanoi(2, 1, 2, 3)    Hanoi(1, 1, 3, 2)   Hanoi(0,-,-,-)
                                                              1 ==>  3  (move 1)
                                                              Hanoi(0,-,-,-)
                                          1 ==>  2  (move 2)
                                          Hanoi(1, 3, 2, 1)   Hanoi(0,-,-,-)
                                                              3 ==>  2  (move 3)
                                                              Hanoi(0,-,-,-)



                     1 ==>  3  (move 4)    (this is the halfway point, when we g
et to here
                                            the "n-1" disks which were on top of
 the bottom
                                            "n" disk have been moved over to peg
 #2 by the
                                             above 3 moves . Note carefully that
 these 3 moves
                                             observe the rules of the puzzle : w
e can move only
                                              1 disk at a time, and at no time c
an we place a
                                               larger disk on top of a smaller d
isk).

                                               so now we simply move the bottom
```

```
"n"th disk from
                                                 peg 1 to peg 3 , and next we hav
e to recursively
                                                 move the "n-1" disks which are s
itting in a pile
                                                 in the middle peg #2 to the dest
ination peg #3
                                                 and we do this by making the "bo
ttom" recursive
                                                call to  Hanoi(2,   2,   3,   1)
                                                              n   from  dest  te
mp




                     Hanoi(2, 2, 3, 1)    Hanoi(1, 2, 1, 3)   Hanoi(0,-,-,-)
                                                              2 ==>  1  (move 5)
                                                              Hanoi(0,-,-,-)
                                          2 ==>  3  (move 6)
                                          Hanoi(1, 1, 3, 2)   Hanoi(0,-,-,-)
                                                              1 ==>  3  (move 7)
                                                              Hanoi(0,-,-,-)


it took 7 moves.






enter number of disks:  4

Hanoi(4, 1, 3, 2) Hanoi(3, 1, 2, 3)  Hanoi(2, 1, 3, 2) Hanoi(1, 1, 2, 3)  Hanoi(
0,
                                                                          1 ==>
2 #1
                                                                          Hanoi(
0,
                                                       1 ==> 3   #2
                                                       Hanoi(1, 2, 3, 1)  Hanoi(
0,
                                                                          2 ==>
3 #3
                                                                          Hanoi(
0,

                                     1 ==> 2  #4

                                     Hanoi(2, 3, 2, 1)  Hanoi(1, 3, 1, 2)  3 ==>
 1 #5
                                                        3 ==>  2  #6
                                                        Hanoi(1, 1, 2, 3)  1 ==>
 2 #7

                  1 ==> 3   #8

                  Hanoi(3, 2, 3, 1)  Hanoi(2, 2, 1, 3) Hanoi(1, 2, 3, 1)   2 ==>
```

```
 3 #9
                                                       2 ==> 1  #10
                                                       Hanoi(1, 3, 1, 2)   3 ==>
 1 #11

                                     2 ==> 3    #12

                                     Hanoi(2, 1, 3, 2) Hanoi(1, 1, 2, 3)    1 ==
> 2 #13
                                                       1 ==> 3  #14
                                                       Hanoi(1, 2, 3, 1)    2 ==
> 3 #15

Note: #1 indicates the move number

Note: Only the  first 4 calls to functions Hanoi(0, -,-,-) are
      indicated in the call graph .. done for brevity...
it took 15 moves.



DETERMINING IF A SQUARE MATRIX IS SYMMETRIC USING RECURSION:
---------------------------------------------------------

(you can skip over this part and come back to it later after
 arrays (notes6.txt) have been covered )

// Author: Ted Obuchowicz
// April 24, 2002
// example program illustrating use of
// recursion to determine if a matrix is symmetric or not
// developed by C. Taillefer on whiteboard
#include <iostream>
#include <string>

using namespace std;


int symmetric(int a[3][3], int size)
{
 int i = size-1;
 int j = size-1;

 if ( size <= 0)
   return 1 ;
 else
   for(int k = 0; k < size-1 ; k++)
   {
      // print out elements being compared for testing purposes only
      // clean up in final version
      cout <<  "Comparing elements a[" << i << "][" << j-k-1 <<
               "]" << " and [" << i-k-1 << "][" << j << "]" << endl;
     if ( a[i][j-k-1] != a[i-k-1][j] )
       return 0;
   }

 return (symmetric(a, size-1));
}
```

```
int bigger_symmetric(int a[5][5], int size)
{
 int i = size-1;
 int j = size-1;

 if ( size <= 0)
   return 1 ;
 else
   for(int k = 0; k < size-1 ; k++)
   {
      // print out elements being compared for testing purposes only
      // clean up in final version
      cout <<  "Comparing elements a[" << i << "][" << j-k-1 <<
               "]" << " and [" << i-k-1 << "][" << j << "]" << endl;
     if ( a[i][j-k-1] != a[i-k-1][j] )
        return 0;
   }

 return (bigger_symmetric(a, size-1));
}



int main()
{

int matrix[3][3] = { {0,1,3},{1,2,5},{3,5,6} };

cout << symmetric(matrix, 3 ) << endl;


int bigger_matrix[5][5] = { {0, 1, 2,  3,  4},
                            {1, 0, 6,  7,  8},
                            {2, 6, 0,  9,  10},
                            {3, 7, 9,  0,  11},
                            {4, 8, 10, 11, 0} };



cout <<  bigger_symmetric(bigger_matrix, 5) << endl;


return 0;
}



// This is the output
//
// it seems to work...
//
//Comparing elements a[2][1] and [1][2]
//Comparing elements a[2][0] and [0][2]
//Comparing elements a[1][0] and [0][1]
//1
//Comparing elements a[4][3] and [3][4]
//Comparing elements a[4][2] and [2][4]
//Comparing elements a[4][1] and [1][4]
```

```
//Comparing elements a[4][0] and [0][4]
//Comparing elements a[3][2] and [2][3]
//Comparing elements a[3][1] and [1][3]
//Comparing elements a[3][0] and [0][3]
//Comparing elements a[2][1] and [1][2]
//Comparing elements a[2][0] and [0][2]
//Comparing elements a[1][0] and [0][1]
//1
```

ARRAYS

Suppose we want to find the maximum value of 5 integer values
stored in variables called value0, value1, value2, value3, value4.
The following portion of code will achieve the desired result:

```
int maximum  = value0;
if value1 > maximum
    maximum = value1;
if value2 > maximum
    maximum = value2;
if value3 > maximum
    maximum = value3;
if value4 > maximum
    maximum = value4;
```

Note how 4 separate if statements were required and a total of
5 separate integer variables were needed to hold the values being
compared.

One can easily see how such an approach would not be practical
to find the maximum of 1000 integer values.

C++ allows for the array data type to simplify such situations in which
one is working with a set of similar data items. An array is simply
a collection of similar items referred to by a single common name in which
every individual item or element in the collection can be individually reference
d.

The following program defines and initializes an array of 10 integers:

```
// Author: Ted Obuchowicz
// Feb. 13, 2002
// example program illustrating use of
// an array


#include <iostream>
#include <string>

using namespace std;



int main()
{

// declare and initialize an array of 10 integers
```

```
int my_array[10] = {12, 234, 23, 1, -7, 55, 18, 67, 99, 100};

// use a for loop to sequentially access the individual
// elements of the array

for(int i = 0 ; i <=9; i++)
  cout << "my_array[" << i << "] = " << my_array[i] << endl;

return 0;
}
```

The program produces the following output:

```
ted@flash Programs 8:16pm >array
my_array[0] = 12
my_array[1] = 234
my_array[2] = 23
my_array[3] = 1
my_array[4] = -7
my_array[5] = 55
my_array[6] = 18
my_array[7] = 67
my_array[8] = 99
my_array[9] = 100
ted@flash Programs 8:16pm >
```

In general arrays are declared in the following manner:

```
base_type_of_elements  name_of_the_array[ size_expression ]
```

where

```
base_type_of_elements: is the type of each array element such as
                       char, int, float, double, etc...
                       (it is even possible to have an array of structs)

name_of_the_array : is an indentifier which specifies the name of the array

size_expression: is an  constant or a constant expression  which identifies
                 the SIZE of the array.
                 The expression must be capable of being evaluated to integer
                 value at COMPILE time.
```

Some examples of legal array declarations are:

```
float some_real_numbers[25] ; // an array of 25 floats
char  two_letters[2] ; // an array which can hold two chars
const int m = 5;
const int  = 4;
```

Some illegal array definitions are:

```
float some_real_numbers[25.5] ; // an array of 25 floats
const float m=4 ;
const float n=5 ;
int an_array[m*n];
```

```
The above are illegal since non-integer types are used to specify
the array sizes:

illegal_array_definitions.C:18: size of array ‘some_real_numbers’ has non-intege
r type
illegal_array_definitions.C:21: size of array ‘an_array’ has non-integer type

int an_array[m*n];



Let us explore this issue of the array size specifier being a CONSTANT integer
expression:

Here is a C program which attempts to use an integer variable y to specify
the size of an integer array called x:

#include <stdio.h>

int main()
{
 int y;
 int x[y];
return 0;

}


This program does NOT compile with the Sun C Compiler :

ted@townshend JUNK 1:06pm >/opt/SUNWspro/bin/cc array3.c
"array3.c", line 6: integral constant expression expected
cc: acomp failed for array3.c


The Sun compiler is very strict, it is complaining (vehemently) that
it is illegal to use an non integral constant expression to specify the
size of an array.

Let us now compile the program using g++:

ted@townshend JUNK 1:07pm >g++ array3.c
ted@townshend JUNK 1:08pm >


Oddly enough, the g++ compiler compiles the source code with no warnings and
no errors and it produces an executable program called a.out :

ted@townshend JUNK 1:08pm >ls -al a.out
-rwx------   1 ted      staff        6064 Oct 18 13:06 a.out


Let’s add a little more functionality to our program and try to compile it
with g++ and run it:

#include <iostream>

int main()
```

```
{
 int y;
 cout << "Enter the array size " << endl;

 cin >> y;

 int x[y];
 for(int i = 0 ; i < y ; i++)
   {
     cout << "Enter the value " ;
     cin >> x[i];
   }

for(int i = 0 ; i < y ; i++)
  cout << x[i] << endl;


}



ted@townshend JUNK 1:09pm >g++ -o array array.C
ted@townshend JUNK 1:10pm >
```

It compiles fine... it even runs fine:

```
ted@townshend JUNK 1:10pm >array
Enter the array size
4
Enter the value 1
Enter the value 2
Enter the value 3
Enter the value 4
1
2
3
4
```

Let's change the program slightly, so that we input the value of y AFTER,
we declare the array x:

```
#include <iostream>

int main()
{
 int y;
 cout << "Enter the array size " << endl;

#include <iostream>

int main()
{
 int y;

 int x[y];

cout << "Enter the array size " << endl;
```

```
 cin >> y;

 for(int i = 0 ; i < y ; i++)
   {
     cout << "Enter the value " ;
     cin >> x[i];
   }

for(int i = 0 ; i < y ; i++)
  cout << x[i] << endl;


}
```

This program compiles fine with the g++ compiler:

```
ted@townshend JUNK 1:11pm >g++ -o array2 array2.C
```

HOWEVER, during run time, a SEGMENTATION FAULT occurs:

```
ted@townshend JUNK 1:11pm >array2
Segmentation fault
```

It appears that the issue of the array size specifier being an integer constant is compiler-dependent. The Sun C compiler is the most strict, there may be some command-line options with g++ which may tell the compiler to be more strict about these issues....

Here they are:

```
ted@brownsugar Programs 5:19pm >g++ -Wall -pedantic -o dynamic_size_array dynami
c_size_array.C
dynamic_size_array.C: In function ‘int main()’:
dynamic_size_array.C:30: error: ISO C++ forbids variable-size array ‘array’
```

These small examples help to prove that one does not know what will occur with a program until you actually compile and run it (using DIFFERENT) compilers.

It would be interesting to see how the Microsoft Visual C++ compiler would handle these examples. Unfortunately, I do not have access to Microsoft Visual C++.

Paul Lambert, a student, has reported that Visual C++ reports:

```
1>.\Test.cpp(18) : error C2466: cannot allocate an array of constant size 0

1>.\Test.cpp(18) : error C2133: ’x’ : unknown size
```

```
In C++, array element number ALWAYS BEGINS with 0 for the first
array element, 1 for the second array element, 2 for the third
array element, etc.  The programmer has no choice in this numbering
sequence (unlike the Pascal and BASIC programming languages).

The elements of an array ARE STORED IN CONTIGUOUS MEMORY LOCATIONS.
This following program prints out the 10 array elements as well
as the address in main memory where each element is stored:


// Author: Ted Obuchowicz
// Feb. 13, 2002
// example program illustrating use of
// an array


#include <iostream>
#include <string>
#include <iomanip>

using namespace std;



int main()
{

// declare and initialize an array of 10 integers

int my_array[10] = {12, 234, 23, 1, -7, 55, 18, 67, 99, 100};

// use a for loop to sequentially access the individual
// elements of the array

for(int i = 0 ; i <=9; i++)
  cout <<   "my_array[" << i << "] = " << my_array[i] <<
          " stored at address " <<( (unsigned long)  &my_array[i]) << endl;

return 0;
}


Note the use of the & operator in front of the &my_array[i]. This
operator is used to return the main memory address which is used
to store the variable.  Nore how we cast this value into an unsigned
long integer to avoid having addresses given in their default hexadecimal
value.

The output is:

ted@flash Programs 8:52pm >array_address
my_array[0] = 12  stored at address 4290704856
my_array[1] = 234 stored at address 4290704860
my_array[2] = 23  stored at address 4290704864
my_array[3] = 1   stored at address 4290704868
my_array[4] = -7  stored at address 4290704872
my_array[5] = 55  stored at address 4290704876
```

```
my_array[6] = 18  stored at address 4290704880
my_array[7] = 67  stored at address 4290704884
my_array[8] = 99  stored at address 4290704888
my_array[9] = 100 stored at address 4290704892


Note how an integer occupies 4 successive bytes in main memory.

If we had not cast the address as an unsigned long integer, the
addresses wold be given in hexadecimal as:

my_array[0] = 12  stored at address 0xffbef5d8
my_array[1] = 234 stored at address 0xffbef5dc
my_array[2] = 23  stored at address 0xffbef5e0
my_array[3] = 1   stored at address 0xffbef5e4
my_array[4] = -7  stored at address 0xffbef5e8
my_array[5] = 55  stored at address 0xffbef5ec
my_array[6] = 18  stored at address 0xffbef5f0
my_array[7] = 67  stored at address 0xffbef5f4
my_array[8] = 99  stored at address 0xffbef5f8
my_array[9] = 100 stored at address 0xffbef5fc


ARRAY BOUNDS
------------

C++ does not prevent the programmer from exceeding the boundaries
of an array. This may cause strange program behavior leading to
incorrect results. For example:

// Author: Ted Obuchowicz
// Feb. 13, 2002
// example program illustrating
//  exceeding an arrays bounds


#include <iostream>
#include <string>

using namespace std;



int main()
{


// define and initializr an array of 14 characters

char my_array[14] = {'K','E','I', 'T', 'H', ' ', 'R','I','C','H','A','R','D','S'
 };

// use a for loop to sequentially access the individual
// elements of the array and intentionally go beyond
// the end of the arrat

for(int i = 0 ; i <=19; i++)
  cout << "my_array[" << i << "] = " << my_array[i] << endl;
```

```
return 0;
}

The output is:

ted@flash Programs 9:04pm >bounds
my_array[0] = K
my_array[1] = E
my_array[2] = I
my_array[3] = T
my_array[4] = H
my_array[5] =
my_array[6] = R
my_array[7] = I
my_array[8] = C
my_array[9] = H
my_array[10] = A
my_array[11] = R
my_array[12] = D
my_array[13] = S
my_array[14] = ö
my_array[15] =
my_array[16] =
my_array[17] =
my_array[18] =
my_array[19] =

Note how the values for my_array[14] through to my_array[19] are
simply attempt to interpret the bit patterns found in these main
memory locations as characters.


C++ even lets a programmer inadvertently supply a negative
value as an array index (and go to a memory location which
PRECEDES the first array element) :

// Author: Ted Obuchowicz
// Feb. 13, 2002
// example program illustrating use of
// an array


#include <iostream>
#include <string>
#include <iomanip>

using namespace std;



int main()
{

// declare and initialize an array of 10 integers

int my_array[10] = {12, 234, 23, 1, -7, 55, 18, 67, 99, 100};

// use a for loop to sequentially access the individual
// elements of the array
```

```
for(int i = -2 ; i <=9; i++)
  cout <<    "my_array[" << i << "] = " << my_array[i] <<
          " stored at address " <<(   &my_array[i]) << endl;

return 0;
}
```

The output is:

```
ted@flash Programs 7:10pm >array_address2
my_array[-2] = 0 stored at address 0xffbef5d0
my_array[-1] = -1 stored at address 0xffbef5d4
my_array[0] = 12 stored at address 0xffbef5d8
my_array[1] = 234 stored at address 0xffbef5dc
my_array[2] = 23 stored at address 0xffbef5e0
my_array[3] = 1 stored at address 0xffbef5e4
my_array[4] = -7 stored at address 0xffbef5e8
my_array[5] = 55 stored at address 0xffbef5ec
my_array[6] = 18 stored at address 0xffbef5f0
my_array[7] = 67 stored at address 0xffbef5f4
my_array[8] = 99 stored at address 0xffbef5f8
my_array[9] = 100 stored at address 0xffbef5fc
```

The program attempts to interpret the random bit
patterns found in the 8 preceding bytes from
address 0xffbef5d8 as integer values and prints
them out.

We can now rewrite our program which finds the maximum of
a list of items making use of array notation:

```
// Author: Ted Obuchowicz
// Feb. 13, 2002
// example program illustrating use of
// an array to store 10 numbers and
// find the maximum value


#include <iostream>
#include <string>

using namespace std;



int main()
{

// declare and initialize an array of 10 integers

int my_array[10] ;


for(int i = 0 ; i <=9; i++)
```

```
 {
  cout <<  "Enter an integer " ;
  cin >> my_array[i];
 }

int max = my_array[0];

for(int i = 1; i <= 9; i++)
   if ( my_array[i] > max)
       max = my_array[i];

cout << "Max is " << max << endl;

return 0;
}

The output is:

ted@flash Programs 9:19pm >max_array
Enter an integer 1
Enter an integer 2
Enter an integer 3
Enter an integer 4
Enter an integer 5
Enter an integer 6
Enter an integer 7
Enter an integer 8
Enter an integer 9
Enter an integer 10
Max is 10



ARRAY INITIALIZATION
--------------------

C++ supports various methods of specifying the initial values
of array elements.


// Author: Ted Obuchowicz
// Feb. 13, 2002
// example program illustrating
//  different methods of array initialization


#include <iostream>
#include <string>

using namespace std;



int main()
{

// declare and initialize an array of 10 integers
// by giving the values of all 10 elements in a
// comma spearated list enclosed in curly parentheses
```

```
int my_array[10] = {12, 234, 23, 1, -7, 55, 18, 67, 99, 100};

//  declare and initialize an array of 5 integers by
//  explicitly initializing the first element to 5,
//  the remaining unspecified elements will be set to 0
//   also by the compiler


 int array2[5] = {5};

// use a for loop to sequentially access the individual
// elements of the array

for(int i = 0 ; i <=9; i++)
  cout << "my_array[" << i << "] = " << my_array[i] << endl;


for(int i = 0 ; i <=4; i++)
  cout << "array2[" << i << "] = " << array2[i] << endl;


return 0;
}
```

The output is:

```
my_array[0] = 12
my_array[1] = 234
my_array[2] = 23
my_array[3] = 1
my_array[4] = -7
my_array[5] = 55
my_array[6] = 18
my_array[7] = 67
my_array[8] = 99
my_array[9] = 100
array2[0] = 5
array2[1] = 0
array2[2] = 0
array2[3] = 0
array2[4] = 0
```

ARRAY INITIALIZATION WITHOUT SPECIFYING AN EXPLICIT ARRAY SIZE

It is possible to initialize an array without specifying it size.
The compiler will determine the array size by counting the number
of expressions encountered in the intialization list.
A few examples illustrare this:

```
int numbers[] = {0,1,2,3,4,5,6,7,8,9} ;

char lower_case_vowels[] = {'a','e','i','o','u'}
```

The size of the numbers array will be set to 10 and the size

```
of the lower_case_vowels array will be set to 5.


CHARACTER ARRAY STRING INITIALIZATION
-------------------------------------

C++ allows for an alternate initialization of character
arrays  which uses a character string (enclosed in
double quotes) to specify the individual initial values
of the array elements:

For example:

char my_name[] = "Ted";

will create and intialize an array of FOUR elements:


my_name[0] = 'T'
my_name[1] = 'e'
my_name[2] = 'd'
my_name[3] = '\0'

The last element of the my_name array is initialized to the NULL CHARACTER
('\0').  The null character is used in C++ to indicate the end of a string.

Character arrays which have been intialized in such a fashion may
either be printed out element by element or by simply sending
the array directly to the cout stream with the << manipulator:

// Author: Ted Obuchowicz
// Feb. 13, 2002
// example program illustrating use of
//  character array strings


#include <iostream>
#include <string>

using namespace std;



int main()
{

char my_name[] = "Ted";

// print out the array element by element

for(int i = 0; i <= 3; i++)
  cout << my_name[i] ;  // null character will cause nothing to be printed.

cout << endl;

// alternate method of printing out character array strings

cout << my_name << endl; // will print out all the characters in the array
                         // up to but not including the null character
```

```
return 0;
}


The output is the same in both cases:

Ted
Ted


Here are some more examples of different ways of initializing
array to the same initial value and different ways of
printing out the identical results:

// Author: Ted Obuchowicz
// Feb. 13, 2002
// example program illustrating use of
//  character array strings


#include <iostream>
#include <string>

using namespace std;



int main()
{


//  four ways to intialize an array to the same values

char my_name1[] = "Ted";
char my_name2[4] = {'T','e','d','\0'};
char my_name3[] = {'T','e','d','\0'};
char my_name4[4] = "Ted";


// print out the array element by element

for(int i = 0; i <= 3; i++)
  cout << my_name1[i] ;  // null character will cause nothing to be printed.

cout << endl;


for(int i = 0; i <= 3; i++)
  cout << my_name2[i] ;  // null character will cause nothing to be printed.

cout << endl;

for(int i = 0; i <= 3; i++)
  cout << my_name3[i] ;  // null character will cause nothing to be printed.

cout << endl;

for(int i = 0; i <= 3; i++)
```

```
  cout << my_name4[i] ;  // null character will cause nothing to be printed.

cout << endl;


cout << endl;

cout << my_name1 << endl << my_name2 << endl << my_name3 << endl <<  my_name4 <<
 endl;

// can even print out a character string array by passing the ADDRESS of the fir
st
// element in the array to cout !!!

cout << &my_name1[0] << endl << &my_name2[0] << endl << &my_name3[0] << endl <<
&my_name4[0] << endl;



// this will print out the letter 'T'

cout << my_name1[0] << endl;

return 0;
}

The output is:

ted@flash Programs 10:14pm >string_array2
Ted
Ted
Ted
Ted

Ted
Ted
Ted
Ted
Ted
Ted
Ted
Ted
T



Reading in character arrays from the keyboard
---------------------------------------------

Character arrays may be read in directly from the keyboard using a
simple

cin >> name_of_array ;

statement .  Any other type of array ( array of ints, array of floats, array of
doubles)
cannot be read in  entirely in a similar fashion, they have to be entered elemen
t by element
within a loop as in  :
```

```
int number_array[5];

for(int i = 0 ; i < 5 : i++)
   cin >> number_array[i];
```

When a character array is readin directly from the keyboard, the end-of-string character ('\0' is added after the last character is entered by pressing the enter key of the keyboard).

Here is a simple program which illustrates these concepts. It declares a charac
ter
array of up to 80 chars which is used to store a name which the user enters from
 the
keyboard. Pressing the Enter key terminates the input (and stores the '\0' char
acter
into the array). The program then prints out the array in reverse element by f
irst
locating the array index which corresponds to where the '\0' is stored, then usi
ng
a for loop from this index value backwards to value 0 to print out each individu
al array
element:

```
// Author: Ted Obuchowicz
// file:  read_in_word.C

#include <iostream>
#include <string>

using namespace std;

int main()
{


char word[80]; // reserve up to 80 characters
cin >> word;   // read in the entire string from the keyboard directly

cout << word << endl;   // print it out .. this will print only up to the '\0'



// use a for loop to print out the ENTIRE 80 characters...
int index = 0;
for (    ; index < 80; index++)
{
   cout << "word[" << index <<"] = " <<  word[index] << endl;
}


// find where the '\0' is stored which indicated the end of the entered word

index = 0;
```

```
for(  ; index < 80; index++)
{
  if (word[index] == '\0' )
     break;
}

// print the string backwards

for (int i = index - 1 ; i >= 0 ; i--)
  cout << word[i] ;

cout << endl;

return 0;
}
```

The output produced by the program is:

```
ted@brownsugar Programs 9:56pm >read_in_word
abc    ( the user types in this word )
abc    ( produced by cout << word statment )
word[0] = a    (produced by the for loop which loops thru all 80 elements of the
 array)
word[1] = b
word[2] = c
word[3] =       ( this is what the '\0' produced when fed in to cin <<  )
word[4] = ÿ  ( these are non-ASCII characters which happened to be stored in som
e locations)
word[5] = #
word[6] = À
word[7] =
word[8] = ÿ
word[9] = $
word[10] =
word[11] = È
word[12] =
word[13] =
word[14] =
word[15] =
word[16] =
word[17] =
word[18] =
word[19] =
word[20] =
word[21] =
word[22] =
word[23] =
word[24] =
word[25] =
word[26] =
word[27] =
word[28] = ÿ
word[29] = ?
word[30] =
word[31] = Ä
```

```
word[32] =
word[33] =
word[34] =
word[35] =
word[36] =
word[37] =
word[38] =
word[39] =
word[40] =
word[41] =
word[42] =
word[43] =
word[44] =
word[45] =
word[46] =
word[47] =
word[48] =
word[49] =
word[50] =
word[51] =
word[52] =
word[53] =
word[54] =
word[55] =
word[56] = ÿ
word[57] = ¿
word[58] = ù
word[59] = °
word[60] =
word[61] =
word[62] =
word[63] = ð
word[64] = ÿ
word[65] = =
word[66] =
word[67] =
word[68] =
word[69] =
word[70] =
word[71] =
word[72] =
word[73] =
word[74] =
word[75] =
word[76] =
word[77] =
word[78] =
word[79] =
cba        ( this is produced by printing out the backwards word starting from o
ne position
           ( before the '\0' character )




PASSING ARRAYS TO FUNCTIONS
----------------------------
```

Consider the following program which passes an array of 5 elements to
to a function. The function simple sets the value of all the array
elements to 0 and returns back to the calling program:

```
// Author: Ted Obuchowicz
// Feb. 13, 2002
// example program illustrating
//  how arrays are passed to functions


#include <iostream>
#include <string>

using namespace std;


void how_are_arrays_passed(int param[], int size_of_array)
{
 for (int i = 0 ; i < size_of_array; i++)
    param[i] = 0; // set all the elements to 0
}


int main()
{

// declare and initialize an array of 5 integers

int array[5] = {100, 100, 100, 100, 100};

// use a for loop to sequentially access the individual
// elements of the array

cout << "Before function : " << endl;

for(int i = 0 ; i <=4; i++)
  cout << "array[" << i << "] = " << array[i] << endl;


how_are_arrays_passed(array,5);

cout << "After function : " << endl;

for(int i = 0 ; i <=4; i++)
  cout << "array[" << i << "] = " << array[i] << endl;



return 0;
}
```

The program output may be surprising to a few readers:

```
Before function :
array[0] = 100
array[1] = 100
array[2] = 100
array[3] = 100
array[4] = 100
```

```
After function :
array[0] = 0
array[1] = 0
array[2] = 0
array[3] = 0
array[4] = 0
```

AHHHH!!!! the values of the array elements in the main program were
actually changed by the function!!!

It would be inefficient to pass an array using thr call-bu value
mechanism (imagine copying a 1 000 000 element array into a
function's stack frame every time to function is invoked, this
would have dramatic effects on a program's run time and memory
requirements).

In C++ (and in good old C) ARRAYS ARE ALWAYS PASSED AS REFERENCE
PARAMETERS. NO & IS NEITHER NEEDED NOR EXPECTED IN THE ARRAY
PARAMETER DEFINITION.

A few astute readers may have noticed that in the definition of
the array parameter to the function no array size was specified.
C++ allows the size to be left out so that the function may work with array
arguments of different size. Of course, a second integer parameter is passed
to the function idnicating the actual size of the array which the function
will be working with during the course of its execution.

SEARCHING A NON-SORTED ARRAY (LINEAR SEARCH)

Suppose we wish to search an array (which contains elements in
non-sorted order) to see if it contains a particular element (the key).
If the element is in the array, we want to (first) index of
where it occurs, otherwise a message saying the element is not
contained in the array is to be displayed. We can perform
a LINEAR SEARCH on the array starting from the first array
element and comparing it with the key, we keep on comparing
successive array elements with the key until we either find
a match or reach the end of the array. This is known as a
LINEAR SEARCH of an array. It requires a number of
comparisons which is proportional to the size, n, of the array.
In computer science terminology, we say that the "order of
complexity of a linear search of an array of n elements is
O(n)" .

Here is an implementation of the linear search in C++:

```
// Author: Ted Obuchowicz
// Feb. 13, 2002
// example program illustrating use of
// linear search on an array to find a key


#include <iostream>
#include <string>
```

```
using namespace std;



int main()
{

// declare and initialize an array of 10 integers

int my_array[10] = {12, 234, 23, 1, -7, 55, 18, 67, 99, 100};

int key;
cout << "enter the key value to be searched for";
cin >> key;

int array_index; // define this outside the for loop
                 // since we want to know its value upon
                 // loop termination

bool found = false;

for( array_index = 0 ; array_index <=9; array_index++)
 {
     if (my_array[array_index] == key)
         {
           found = true;
           break;
          }
 }


if (found)
  cout << "Key found in position " << array_index <<  " " << my_array[array_inde
x] << endl;
else
  cout << "Key is not in the array." << endl;



return 0;
}
```

We can employ recursion to perform a linear search of an unordered array.
We simply start from the beginning of the array, if the first element
of the array matches the key, a message is displayed, if there is no match
we make a recursive call with the remaining elements of the array. Eventually,
all the elements will be compared one-by-one with the key resulting in either
a match or the key not being found :

```
// Author: Ted Obuchowicz
// Feb. 13, 2002
// example program illustrating use of
// linear search on an array to find a key


#include <iostream>
```

```
#include <string>

using namespace std;


void linear_search(int array[], int start , int size, int key)
{
  if (key == array[start])
     cout << "key found in array[" << start << "]" << endl;
  else
     if ( start == (size-1) )
       cout << "key not found." << endl;
     else
       linear_search(array, start+1, size, key);
}


int main()
{

// declare and initialize an array of 10 integers
int my_array[10] = {-56, 0, 5, -234, 1, 12, -3, 678, 34, 100 };

int key;
cout << "enter the key value to be searched for  " ;
cin >> key;

linear_search(my_array, 0, 10, key);

return 0;
}
```

SORTING AN ARRAY LINEARLY

Sorting a set of data into some numeric order (increasing order or decreasing order) is a commonly required task in data processing. Here is a C++ program which performs a LINEAR SORT on an array of 10 elements:

```
// Author: Ted Obuchowicz
// Feb. 13, 2002
// example program illustrating use of
// linear sort on an array to sort it into
// ascending order (smallest to largest value)



#include <iostream>
#include <string>

using namespace std;


void print_array(int some_array[], int size)
{
 for(int i = 0 ; i < size ; i++)
```

```
  cout << some_array[i] << endl;
}


int main()
{

// declare and initialize an array of 10 integers

int unsorted_array[10] = {12, 234, 23, 1, -7, 55, 18, 67, 99, 100};


cout << "The unsorted array is: " << endl;
print_array(unsorted_array, 10);

int sorted_array[10];

// now sort the array in place

for(int i = 0 ; i < 10 ; i++)
{
  int smallest = unsorted_array[i];
  for(int j = i+1 ; j <10; j++)
  {
    if (unsorted_array[j] < smallest)
       {
        smallest = unsorted_array[j]; // found a new smallest value in array
        int temp = unsorted_array[i]; // now swap the two array elements
        unsorted_array[j] = temp;
        unsorted_array[i] = smallest;
       }
  }
}


cout << "The sorted array is: " << endl;

print_array(unsorted_array, 10);

return 0;
}


The program output is:

The unsorted array is:
12
234
23
1
-7
55
18
67
99
100
The sorted array is:
-7
1
```

```
12
18
23
55
67
99
100
234
```

New and Improved sorting program

Here is a simpler version of the previous array sorting example program.
It does not make use of the "int smallest" variable which was introduced
during preliminary program debugging. It simply "swaps" the two array
elements if necessary within the for loop.

```
// linear sort on an array to sort it into
// ascending order (smallest to largest value)



#include <iostream>
#include <string>

using namespace std;

// a function to print out the elements of an array

void print_array(int some_array[], int size)
{
 for(int i = 0 ; i < size ; i++)
  cout << some_array[i] << endl;
}


int main()
{

// declare and initialize an array of 10 integers

int unsorted_array[10] = {12, 234, 23, 1, -7, 55, 18, 67, 99, 100};


cout << "The unsorted array is: " << endl;
print_array(unsorted_array, 10);


// now sort the array in place

for(int i = 0 ; i < 10 ; i++)
{
  for(int j = i+1 ; j <10; j++)
  {
```

```
    if (unsorted_array[j] < unsorted_array[i])
       {
        int temp = unsorted_array[i]; // now swap the two array elements
        unsorted_array[i] = unsorted_array[j];
        unsorted_array[j] = temp;
       }
  }
}


cout << "The sorted array is: " << endl;

print_array(unsorted_array, 10);

return 0;
}
```

This program can be further refined by making use of a "swap" function which
swaps the two array elements when we find (unsorted_array[j] < unsorted_array[i]
)
to be true within the nested loop as in:

```
#include <iostream>
#include <string>

using namespace std;


// a function to print out the elements of a 1-dimensional array

void print_array(int some_array[], int size)
{
 for(int i = 0 ; i < size ; i++)
  cout << some_array[i] << endl;
}


// a function which swaps two elements of the array
// note since arrays are always passed by REFERENCE
// the swap occurs on the actual array which is passed to the function

void swap_elements(int some_array[], int first, int second)
{
 int temp;
 temp = some_array[second];
 some_array[second] = some_array[first];
 some_array[first]  = temp;
}


int main()
```

```
{

// declare and initialize an array of 10 integers

int unsorted_array[10] = {12, 234, 23, 1, -7, 55, 18, 67, 99, 100};


cout << "The unsorted array is: " << endl;
print_array(unsorted_array, 10);


// now sort the array in place

for(int i = 0 ; i < 10 ; i++)
{
  for(int j = i+1 ; j <10; j++)
  {
    if (unsorted_array[j] < unsorted_array[i])
       {
        swap_elements(unsorted_array, i,j); //  can also do swap_elements(unsort
ed_array, j , i)
       }
  }
}


cout << "The sorted array is: " << endl;

print_array(unsorted_array, 10);

return 0;
}

Either version of the program (as the original, somewhat confusing version) prod
uces
the correct output:

The unsorted array is:
12
234
23
1
-7
55
18
67
99
100
The sorted array is:
-7
1
12
18
23
55
67
99
100
234
```

Note how the use of the swap_elements(unsorted_array, i,j) function call within
 the
nested loops improves program readability.

HIGHER DIMENSIONAL ARRAYS

C++ allows for 2-dimensional (and higher dimensions) arrays.
A 2-d array is commonly referred to as a MATRIX.

Here is a pictorial representation of a 2-d array consisting of
three rows and four columns:

```
       col 0   col 1   col 2  col 3
      ------------------------------
row 0 | 0       1       2      3
      |
row 1 | 1       2       3      4
      |
row 2 | 2       3       4      5
```

In C++, we would declare a 2-d array with the following declaration:

```
int array_2d[3][4] ;
```

The first number in [] refers to the number of rows, the second number
enclosed in [] specifies the number of columns. NOTE: IN C++, ARRAY
INDICES START FROM 0. The element found in row 0 and column 0 would
be referred to a array_2d[0][0].

Here is an example program which initializes and prints out the
values of a 3 by 4 matrix:

```
// author: Ted Obuchowicz
// March. 11, 2002
// example program illustrating use of



#include <iostream>
#include <string>

using namespace std;



int main()
{
```

```
//  declare a 2 dimensional array of integers

int array_2d[3][4] ;

for(int row = 0; row < 3; row++)
  for(int col = 0 ; col < 4 ; col++)
      array_2d[row][col] = row + col;

// print out the array formatted into rows and columns

for(int row = 0; row < 3; row++)
  for(int col = 0 ; col < 4 ; col++)
    {
      cout << array_2d[row][col] << " "  ;
     if ( col  == 3)
         cout << endl;
    }

return 0;
}


The program output is:

ted@dea Programs 12:50pm >two_d_array
0 1 2 3
1 2 3 4
2 3 4 5
ted@dea Programs 12:50pm >


How 2-d arrays are stored in memory :
------------------------------------



Main memory in a computer is viewed by  the compiler as a
linear sequence of bytes.  Hence, a two-dimensional array
must be mapped onto this linear array of storage locations by the compiler.
The compiler stores EACH ROW of a two-d array in successive memory
locations.

For example, our two-d array of the previous program would be stored
1-dimensional array in main memory as follows:





0    one_d_array[0]

1    one_d_array[1]

2    one_d_array[2]

3    one_d_array[3]

1    one_d_array[4]

2    one_d_array[5]
```

```
3    one_d_array[6]

4    one_d_array[7]

2    one_d_array[8]

3    one_d_array[9]

4    one_d_array[10]

5    one_d_array[11]


Such a mapping of the elements of a two-dimensional array onto a one-
dimensional array is called "ROW-MAJOR FORM" .

The compiler uses the following  address translation mechanism
when accessing two dimensional  arrays stored in row-major
form in main memory

two_d_array[i][j]  =  one_d_array[ i*(number of columns) + j ]

For example, in the above example, there are 4 columns in our two
dimensional array, thus


two_d_array[0][2] is found in one_d_array[0*4 + 2]  = one_d_array[2]

and

two_d_array[1][3] is found in one_d_array[1*4 + 3] = one_d_array[7]


It is important to note that the name of the array is associated with the
starting address of the first element in the array.  In reality, the
compiler simply translates [row][column] indices of a two-d array into
an offset from some starting address of where the elements of the array
are stored in.

Here is a program which prints out the 12 addresses which are used to
store our 3 x 4 matrix:

// author: Ted Obuchowicz
// April 17j, 2002
// example program illustrating
// how a two d array is stored in successive memory locations
// in main memory in row-major form



#include <iostream>
#include <string>

using namespace std;



int main()
{
```

```
//  declare a 2 dimensional array of integers

int array_2d[3][4] ;

for(int row = 0; row < 3; row++)
  for(int col = 0 ; col < 4 ; col++)
      array_2d[row][col] = row + col;

// print out the array formatted into rows and columns

for(int row = 0; row < 3; row++)
  for(int col = 0 ; col < 4 ; col++)
    {
      cout <<  "Element found at row = [" << row << "]" <<
      " and column = [" << col << "]" << " stored at address " <<
       (unsigned int) &array_2d[row][col] << endl;

    }

return 0;
}
```

The output produced by this program is:

```
Element found at row = [0] and column = [0] stored at address 4290704760
Element found at row = [0] and column = [1] stored at address 4290704764
Element found at row = [0] and column = [2] stored at address 4290704768
Element found at row = [0] and column = [3] stored at address 4290704772
Element found at row = [1] and column = [0] stored at address 4290704776
Element found at row = [1] and column = [1] stored at address 4290704780
Element found at row = [1] and column = [2] stored at address 4290704784
Element found at row = [1] and column = [3] stored at address 4290704788
Element found at row = [2] and column = [0] stored at address 4290704792
Element found at row = [2] and column = [1] stored at address 4290704796
Element found at row = [2] and column = [2] stored at address 4290704800
Element found at row = [2] and column = [3] stored at address 4290704804
```

Let us examine this output to dig a little deeper into the address translation mechanism employed by the compiler. We see the first element array_2d[0][0] is stored in memory location with address 4290704760 (we used the cast operator to convert the hexadecimal address returned by the & operator into an unsigned integer for readability purposes).

If we use our address translation formula:

```
array_2d[0][1] = array_2d[0*4 + 1] = array_2d[1].
```

The compiler treats the name of an array as the starting address in main memory where the first element of the array is stored. Hence, the label we call "array_2d" really represents address 4290704760. What the compiler really does during the address conversion process is:

```
array_2d[i][j]  =   starting address + sizeof(int)*((i*4)+j)
```

As an example,

```
array_2d[0][1] = 4290704760 + 4*(0*4 + 1)
               = 4290704760 + 4*(1)
               = 4290704764
```

Recall that the sizeof function is used to determine the number of
bytes that a given data type occupies. An int occupies 4 bytes of
main memory.

As a final example:

```
array_2d[2][2] = 4290704760 + 4*(2*4 + 2)
               = 4290704760 + 4*(10)
               = 4290704800
```

which is the correct address.

Passing Higher-Dimensional Arrays to Functions:

When passing a two-dimensional (and higher dimensionsal) array
to a function, C++ requires that the second (and higher) dimensions
be passed directly (within the []). The first dimension may be omitted
from the first set of [] . The reason for this is that the compiler
requires the size of the second and higher dimensions in order to perform
the address translation mechanism discussed previosly.

The following program fails to compile since we did not specify
the size of the second dimension directly to the function:

```
// author: Ted Obuchowicz
// March. 11, 2002
// example program illustrating use of



#include <iostream>
#include <string>

using namespace std;

const int column_size = 4;


// when passing multidimensional arrays to a function, the
// size of each dimension other than the first must be
// specified
// this will not compile, the following error will be reported by g++
// two_d_array_function_errors.C:20: declaration of ‘some_array’
//  as multidimensional array
// two_d_array_function_errors.C:20: must have bounds for all dimensions
// except the first

void Print_Matrix(int some_array[][], int row, int col)
{
for(int i = 0; i < row; i++)
  for(int j = 0 ; j < col ; j++)
```

```
    {
      cout << some_array[i][j] << " "  ;
     if ( j  == 3)
         cout << endl;
    }
}



int main()
{

//  declare a 2 dimensional array of integers

int array_2d[3][4] ;

for(int row = 0; row < 3; row++)
  for(int col = 0 ; col < 4 ; col++)
      array_2d[row][col] = row + col;

// print out the array formatted into rows and columns


 Print_Matrix(array_2d, 3, 4);

return 0;
}


The program generates the following compile time errors:

two_d_array_function_errors.C:23: declaration of ‘some_array’ as multidimensiona
l array
two_d_array_function_errors.C:23: must have bounds for all dimensions except the
 first
two_d_array_function_errors.C: In function ‘void Print_Matrix(int, int)’:
two_d_array_function_errors.C:27: ‘some_array’ undeclared (first use this functi
on)
two_d_array_function_errors.C:27: (Each undeclared identifier is reported only o
nce
two_d_array_function_errors.C:27: for each function it appears in.)
two_d_array_function_errors.C: In function ‘int main()’:
two_d_array_function_errors.C:49: passing ‘int (*)[4]’ to argument 1 of ‘Print_M
atrix(int, int)’ lacks a cast
two_d_array_function_errors.C:23: too many arguments to function ‘void Print_Mat
rix(int, int)’
two_d_array_function_errors.C:49: at this point in file


The solution is to pass the second dimension directly to the function.
The following program declares a global constant integer called
column_size (set to 4) , this constant is included in the second set
of [] when passign the array to the function:

// author: Ted Obuchowicz
// March. 11, 2002
// example program illustrating use of
```

```
#include <iostream>
#include <string>

using namespace std;

const int column_size = 4;

void Print_Matrix(int some_array[][column_size], int row, int col)
{
for(int i = 0; i < row; i++)
  for(int j = 0 ; j < col ; j++)
    {
      cout << some_array[i][j] << " "  ;
     if ( j  == 3)
         cout << endl;
    }
}



int main()
{

//  declare a 2 dimensional array of integers

int array_2d[3][4] ;

for(int row = 0; row < 3; row++)
  for(int col = 0 ; col < 4 ; col++)
      array_2d[row][col] = row + col;

// print out the array formatted into rows and columns


 Print_Matrix(array_2d, 3, 4);

return 0;
}
```

Note in the function prototype we pass the parameters:

```
void Print_Matrix(int some_array[][column_size], int row, int col)
```

some_array[][column_size],
row, and
col.

the parameters row and col are used to control two nested for loops which access the array elements using array index notation : array_2d[row][col].

It is necessary to pass the second dimension size directly to the function as in int some_array[][column_size] since the value of the second dimension is needed by the compiler to perform the address translation mechanism.

As a byproduct of this requirement of passing higher dimensions sizes
of an array to a function is that we would have to write different functions
for different sized multi-dimensional arrays. There is a way of getting
around this restriction, but it requires the use of pointers which is
our next topic of discussion.

POINTERS

C++ has a data type called a POINTER data type. The * symbol is used to define a pointer
data type. For example, suppose we have an integer variable called x, we would declare
it as follows:

```
int x ;
```

To declare a pointer to an integer would use use the following declaration:

```
int* some_pointer;
```

Pointer declarations are best understood if we 'read' them backwards as in

```
      int *  some_pointer;
            ------------
               1) first read this as "some_pointer" is a

          2) read the * symbol as "pointer to an"

      3) finally read the data type to which to pointer is pointing to as "integ
er"
```

If we put the above three ideas together, we would interpret the C++ declaration

int* some_pointer to mean:

"some_pointer is pointer to an integer"

Pointers to other data types are declared in a similar fashion, the following de
clare
pointers to a char, float, double respectively:

```
char* char_pointer;
float* float_pointer;
double* double_pointer;
```

A variable which is declared to be a pointer simply hold the ADDRESS of some obj
ect.
Once we have declared a pointer, we can give it a value by using the address-of
(&) operator as in:

```
int x;
int* some_pointer;
```

```
some_pointer = &x; // assign the address where x is stored to the pointer variab
le

Suppose memory location with address 1000 is used to store the integer variable
x and memory location with address 2000 is used to store the pointer variable
some_pointer.  The situation looks like:


             contents         address

             ----------
x               ?               1000
             ----------

             ----------

             ----------
some_pointer  1000            2000
             ----------



Once we have a pointer to a variable, we can access the variable using the point
er
derferencing operator (*) (C++ is a bit confusing in the sense that the same sym
bol
has more than one meaning depending on where it is used in a program).

Suppose we use the assignment operator to give a value of  4 to the integer vari
able
x:

x = 4;

The contents of memory now look like:

             contents         address

             ----------
x               4               1000
             ----------

             ----------

             ----------
some_pointer  1000            2000
             ----------



We can change the value stored in this x variable by  the following:

*some_pointer = 5 ;

This statement means "go to the memory location at address 1000 and store
the value 5 in this location".  The situation in memory now looks like:


             contents         address
```

```
              ----------
x                5               1000
              ----------

              ----------

              ----------
some_pointer  1000             2000
              ----------



Here is a small C++ program which illustrate the pointer concepts we have discus
sed so
far:

// Author: Ted Obuchowicz
// April 17, 2002
// example program illustrating use of
// pointers and pointer dereferencing


#include <iostream>
#include <string>

using namespace std;



int main()
{

int mick = 4 ;  // declare an integer variable called mick and assign it value 4
int keith_richards = 23 ; // declare an int  called keith_richards initialized t
o 23
float ron = 4.567;
double charlie = 3.333333333;

// now declare some pointers

int* mick_ptr = &mick;
int* keith_ptr = & keith_richards; // C++ doesn't care if you put a space betwee
n & and variable

cout << &keith_richards << endl;
float* ron_ptr = &ron;
double* charlie_ptr = &charlie;

// print out the addresses of the 4 variables and print out the values of their
pointers
// the pointers will contain the addresses of the respective variables

cout << "Variable mick stored in location " << &mick
     << " Value stored in mick_ptr = " << mick_ptr  << endl;


cout << "Variable keith_richards stored in location " << &keith_richards
     << " Value stored in keith_ptr = " << keith_ptr << endl;
```

```
cout << "Variable ron stored in location " << &ron
     << " Value stored in ron_ptr = " << ron_ptr  << endl;

cout << "Variable charlie stored in location " << &charlie
     << " Value stored in charlie_ptr = " << charlie_ptr  << endl;


//  now change the values stored in the 4 variables using pointer dereferencing

*mick_ptr = 345;
*keith_ptr = 222;
*ron_ptr = 2345.678;
*charlie_ptr = 5555.55555;

// print out the new values again using pointer dereferencing

cout << "mick = " << *mick_ptr << endl;
cout << "keith_richards = " << *keith_ptr << endl;
cout << "ron = " << *ron_ptr << endl;
cout << "charlie = " << *charlie_ptr << endl;


return 0;
}
```

The program output is:

```
0xffbeea20
Variable mick stored in location 0xffbeea24 Value stored in mick_ptr = 0xffbeea2
4
Variable keith_richards stored in location 0xffbeea20 Value stored in keith_ptr
= 0xffbeea20
Variable ron stored in location 0xffbeea1c Value stored in ron_ptr = 0xffbeea1c
Variable charlie stored in location 0xffbeea10 Value stored in charlie_ptr = 0xf
fbeea10
mick = 345
keith_richards = 222
ron = 2345.68
charlie = 5555.56
```

Some Dos and Don’ts for Pointers
----

1) incorrect assignment of pointers

C++ is a very strongly typed language, the language does not allow the
assignment of the address of a non-integer data type to an integer pointer
variable for example. Basically, if a pointer to a certain data type (int, char
,
float, double, etc) has been declared, the pointer may be only assigned the addr
ess
of a variable of the appropriate data type. In other words:

an int* may only receive the address of some int variable
a char* may only receive the address of some char variable
a float* may only receive the address of some float variable

etc.

The following program contains an error since we are attempting to assign
the address of a float to something which has been decalred to an integer
pointer:

```
// Author: Ted Obuchowicz
// April 17, 2002
// example program illustrating use of
// typical pointer errors


#include <iostream>
#include <string>

using namespace std;



int main()
{

int x;
float f;
int* pointer = &f; // bad , bad, bad,,, compiler will vomit on this line


return 0;
}
```

The compiler reports the following error:

```
pointer_errors.C:19: initialization to ‘int *’ from ‘float *’
```

2) Attempting to deference a non-initialized pointer

Attempting to deference a non-initialized pointer variable will give
undefined and unpredictable results. Merely declaring a pointer variable
does not give it an initial value. This is a common error which has kept
many a C++ programmer up at night trying to debug their program. Here is
a sample of this hard to trace error:

```
// Author: Ted Obuchowicz
// April 17, 2002
// example program illustrating use of



#include <iostream>
#include <string>

using namespace std;



int main()
{
```

```
int fender;
int* les_paul;  // the value of the pointer called les_paul is unitialized

cout << *les_paul << endl; // garbage will be produced as output

return 0;
}
```

This program compiles fine, but when run on a UNIX system produces the dreaded
Segmentation Fault run-time error message:

```
ted@flash Programs 8:18pm >unitialized_pointer
Segmentation fault
ted@flash Programs 8:19pm >
```

SO ALWAYS REMEMBER TO GIVE AN ADDRESS TO A POINTER VARIABLE USING THE
ADDRESS OF OPERATOR (&) APPLIED TO SOME PREVIOUSLY DECLARED VARIABLE.

VOID POINTERS

There is one exception to the rule about assigning the address of an int
to a int pointer, the address of a float to a float pointer, etc. C++ allows
the programmer to declare a VOID POINTER. A VOID POINTER may be assigned the ad
dress
of any type. For example;

```
void* pointer_to_anything ;
void* another_pointer_to_anything;
int an_integer;
float a_float;
```

the following assignments of addresses to void pointers are legal:

```
pointer_to_anything = &an_integer;
another_pointer_to_anything = &a_float;
```

Although void pointers may be assinged the address of any data type,
VOID POINTERS MAY NOT BE DEREFERENCED DIRECTLY!!!!!!!!!!!!!!!!!

The following program is incorrect since it attempts to deference
void pointers directly:

```
// Author: Ted Obuchowicz
// April 17, 2002
// example program illustrating INCORRECT  use of
// void pointers


#include <iostream>
#include <string>

using namespace std;
```

```
int main()
{

// declare some void pointers

void* pointer_to_anything ;
void* another_pointer_to_anything;

// declare some variables of type int and float
int an_integer = 5 ;
float a_float = 4.56;

// assign some addresses to the void pointers.. so far this is all legal C++

pointer_to_anything = &an_integer;
another_pointer_to_anything = &a_float;

// now we get into trouble by trying to directly dereference the void pointers

*pointer_to_anything = 67;  // compiler complains about this line
cout << an_integer << endl;



return 0;
}
```

The compiler output is:

```
ted@flash Programs 8:34pm >g++ -o void_pointers void_pointers.C
void_pointers.C: In function ‘int main()’:
void_pointers.C:33: invalid use of void expression
```

The reason why it is forbidden to dereference a void pointer directly
is that the compiler has no information as to what the void pointer is
pointing to..

Void pointers may be dereferenced PROVIDED THEY ARE CAST TO A POINTER OF
THE APPROPRIATE DATA TYPE BY USING THE CAST OPERATOR:

The following program is the correct version of the above program.
It applies the cast operator to the void pointers , then it dereferences
the "casted" pointers:

```
// Author: Ted Obuchowicz
// April 17, 2002
// example program illustrating CORRECT  use of
// void pointers by first casting the void pointer to
// an appropriate pointer


#include <iostream>
#include <string>

using namespace std;
```

```
int main()
{

// declare some void pointers

void* pointer_to_anything ;
void* another_pointer_to_anything;

// declare some variables of type int and float
int an_integer = 5 ;
float a_float = 4.56;

// assign some addresses to the void pointers.. so far this is all legal C++

pointer_to_anything = &an_integer;
another_pointer_to_anything = &a_float;

* (int*)pointer_to_anything = 67;  // cast this pointer to an integer pointer
cout << an_integer << endl;        // then apply the dereference operator

* (float*)another_pointer_to_anything = 7778.89;  // cast it to a float pointer
cout << a_float << endl;



return 0;
}
```

The program output is:

```
67
7778.89
```

POINTERS AND ARRAY NAMES

In C++, the name of an array is the same as the starting address of the first element in the array. We can use this fact to traverse the array using pointer notation :

```
// Author: Ted Obuchowicz
// Feb. 11, 2002
// example program illustrating use of



#include <iostream>
#include <string>

using namespace std;



int main()
{
```

```
int array[5] = {0,1,2,3,4};

int * ptr;

ptr = &array[0];  // same as doing ptr = array

for(int i = 0 ; i < 4; i++)
{
  cout <<  (*ptr) << ptr << endl;
  ptr = ptr + 1;
}

return 0;
}
```

The program output is:

```
ted@flash Programs 8:49pm >array_pointer
0  0xffbeea00
1  0xffbeea04
2  0xffbeea08
3  0xffbeea0c
ted@flash Programs 8:49pm >
```

In the above program an integer pointer called ptr is assigned the name of the array , and then a for loop is used to successivley one 1 to this starting address to print out the array contents and the address at which the contents are stored.

POINTER ARITHMETIC

The above program illustrates the concept of what is referred to as POINTER ARITHMETIC in C++. If we have an integer pointer (ptr), and we perform the assignment:

```
ptr = ptr + 1 ;
```

The actual value of ptr will be incremented by 4 and not by 1.
Look at the addresses contained in the pointer ptr in the above program at every loop iterations:

```
0xffbeea00   for the first time through the loop ptr = ptr + 0;
0xffbeea04   for the second time through the loop ptr = ptr + 1;
0xffbeea08   for the third time through the loop 0xffbeea08 = 0xffbeea04 + 1
etc.
```

When we write down something like:

```
pointer_variable = pointer_variable + 1
```

the compiler really translates this to

```
pointer_variable = pointer_variable + (1*sizeof(thing that pointer_variable is p
ointing to)
```

Since an integer occupies 4 bytes of storage (sizeof(int) = 4), when we add "1"

```
to an integer pointer we are really increasing the value of the pointer by 4. If
 we
add "1" to a character pointer, we are increasing the address by 1 since a chara
cter
occupies only 1 byte of storage.  This means that it is incorrect to add "1" to
a void
pointer since the compiler has no way of determining how much to add to the void
 pointer
since it does not know what the void pointer is pointing to:


// Author: Ted Obuchowicz
// April 17, 2002
// example program illustrating INCORRECT  use of
// void pointers


#include <iostream>
#include <string>

using namespace std;



int main()
{

// declare some void pointers

void* pointer_to_anything ;

int an_integer = 5 ;


pointer_to_anything = &an_integer;


// this is wrong!!!

pointer_to_anything = pointer_to_anything + 1;


return 0;
}



The compiler output is:

ted@flash Programs 9:03pm >g++ -o void_pointer_increment void_pointer_increment.
C
void_pointer_increment.C: In function ‘int main()’:
void_pointer_increment.C:29: ANSI C++ forbids using pointer of type ‘void *’ in
arithmetic
ted@flash Programs 9:03pm >
```

Here is another method of accessing the elements of the array using
only the array name, an offset, and pointer dereferencing:

```
// Author: Ted Obuchowicz
// April 17, 2002
// example program illustrating use of
// address offset notation


#include <iostream>
#include <string>

using namespace std;



int main()
{

int array[5] = {0,1,2,3,4};



for(int i = 0 ; i < 4; i++)
{
  cout <<  *(array + i) << "  " <<  array + i << endl;
}

return 0;
}
```

The output is (exactly the same as before):

```
ted@flash Programs 9:07pm >array_pointer2
0  0xffbeea00
1  0xffbeea04
2  0xffbeea08
3  0xffbeea0c
```

The concept being exploited in this program is that of pointer arithmetic
combined with the fact that an arrat name is the same as the starting address
of the first element of the array. So

```
array + 0 = address of array[0]
array + 1 = address of array[1]
array + 2 = address of array[2]
array + 3 = address of arrar[3]
```

IT IS IMPORTANT TO NOTE THAT WE CANNOT DO SOMETHING LIKE array = array + 1 !!!!!
IN C++, the name of an array is treated as a constant address, once the compiler
has allocated memory to hold an array, this starting address cannot be changed!!
!

The following program does not compile sionce it attempts to change the
address of an array:

```
#include <iostream>
#include <string>
```

```
using namespace std;



int main()
{

int array[5] = {0,1,2,3,4};



for(int i = 0 ; i < 4; i++)
{
  cout <<  *(array ) << "  " <<  array << endl;
  array = array + 1 ; // this is wrong... array is treated as a constant pointer

                      // so it cannot be changed
}

return 0;
}
```

The compiler reports the error:

array_pointer3.C:24: incompatible types in assignment of ‘int *’ to ‘int[5]’

CONSTANT POINTERS

C++ allows for the declaration of constant pointers. A constant pointer must be assigned an address when it is declared, and once it has been assigned this address it cannot be subsequently changed to another address:

```
int x;
int* const ptr  = &x;

4  3   2    1   (this is to help you read the pointer declaration)
```

The pointer declaration is to be read as:

1: ptr is a
2: constant
3: pointer
4: to an integer

It would be an error to attempt to assign the value of ptr after its declaration:

```
int x;
int* const ptr;
ptr = &x ; // wrong...
```

It would also be an error to attempt to reassign the value of th constant pointe
r:

```
int x;
int y;
int* const ptr = &x; // this is ok

ptr = &y ; // INCORRECT !! assignment of read-only variable 'ptr'



REFERENCES AND CONSTANT POINTERS
--------------------------------

The C++ compiler treats reference variables as CONSTANT POINTERS. The compiler
performs the dereferencing of these constant pointers for you automatically.
For example,


int some_int;
int& some_reference = &some_int;

some_int = 5 ;
cout << some_int << endl ; // produces 5 as output

some_reference = 10;
cout << some_int << endl;  // produces 10 as output

The compiler actually produces run -time code equivalent to the
following:

int some_int;
int* const some_reference = &some_int

some_int = 5 ;
cout << some_int << endl ; // produces 5 as output

*some_reference = 10; // compiler does the dereferencing of reference variables
                      // for you automatically
cout << some_int << endl;  // produces 10 as output



Pointers to constant data :

Consider the following declaration:

const int i = 5;
int const * ptr;
ptr = &i;

ptr is a pointer to an integer constant.  The compiler allows the integer
to be read (directly and indirectly through the pointer) but any attempt to
deference the pointer ptr yields a compile time error stating
"assignment of read-only location" .  This is the case in the following
example program:

// Author: Ted Obuchowicz
// April 17, 2002
// example program illustrating use of
// constant pointers
```

```
#include <iostream>
#include <string>

using namespace std;



int main()
{

const int i = 5;
int const * ptr;
ptr = &i;

// we may read the value of the integer constant through its pointer

cout << *ptr << endl;

//  but we are not allowed to dereference a pointer which is pointing to a const
ant

*ptr = 10;


return 0;
}
```

It is a good thing the compiler enforces this, were it not the case, it
would be possible to change the value of a constant indirectly though a pointer.

The address of a non-constant variable may be assigned to a pointer to a constan
t,
but the compiler will still not allow you to dereference the pointer:

```
// Author: Ted Obuchowicz
// April 17, 2002
// example program illustrating use of
// constant pointers


#include <iostream>
#include <string>

using namespace std;



int main()
{

int i = 5;
int const * ptr;
ptr = &i; // assign the address of a non-constant integer i to a pointer
          // which points to an integer constant

// we may read the value of the integer constant through its pointer
```

```
cout << *ptr << endl;

//  but we are not allowed to dereference a pointer which is pointing to a const
ant

*ptr = 10;


return 0;


Constant Pointers to Constant Data
----------------------------------

This is the most restrictive, neither the value of the pointer, nor the contents
that it is pointing to can be changed. A declaration of a constant pointer which
points to an integer constant is:


const int x = 5;
const int * const y = &x ;


A generic Print_Matrix function which uses pointers
---------------------------------------------------

We are now in a position to write a generic Print_Matrix function which
is capable of printing out any sized two-dimensional array . It makes use
of the address trnanslation mechanism, pointer arithmetic, and void pointers:

// Author: Ted Obuchowicz
// March. 20, 2002
// example program illustrating use of
// pointers to access the elements of a two-dimensional array


#include <iostream>
#include <string>

using namespace std;


void Print_Matrix(void* a , int rows, int columns, char data_type)
{
 for(int i = 0 ; i < rows ; i++)
   for(int j =0 ; j < columns ; j++)
   {
       switch (data_type)
       {
        case 'i' : cout << *( (int *)a + (i * columns) + j) << " " ;
                   break;
        case 'c' : cout << *( (char *)a + (i * columns) + j) << " " ;
                   break;
        case 'f' : cout << *( (float *)a + (i * columns) + j) << " " ;
                   break;
        default : cout << "Invalid data type" << endl;
                  return;
        }
```

```
        if ( j == columns - 1)
           cout << endl;
   }
   cout << endl;
}


int main()
{

int little_matrix[3][3] = { 1,2,3,4,5,6,7,8,9 };

int big_matrix[5][5] = { 10, 11, 12, 13, 14,
                         16, 17, 18, 19, 20,
                         22, 23, 24, 25, 26,
                         28, 29, 30, 31, 32,
                         34, 35, 36, 37, 38 };

char array_of_chars[2][2] = { 'a', 'b', 'c', 'd'};

float array_of_floats[4][4] = { 1.23, 2.34, 3.33, 4.44,
                                5.55, 6.66, 7.77, 8.88,
                                9.99, 10.10, 11.11, 12.12,
                                13.13, 14.14, 15.15, 16.16};


cout << "The little matrix is : " << endl;
Print_Matrix(&little_matrix[0][0], 3, 3, 'i');

cout << "The big_matrix is : " << endl;
Print_Matrix(&big_matrix[0][0], 5, 5, 'i');


cout << "The character array is : " << endl;
Print_Matrix(&array_of_chars[0][0], 2,2, 'c');

cout << "The float array is : " << endl;
Print_Matrix(&array_of_floats[0][0], 4,4, 'f');

return 0;
}



The program output is:

ed@flash Programs 9:47pm >generic_print_matrix_using_pointers
The little matrix is :
1 2 3
4 5 6
7 8 9

The big_matrix is :
10 11 12 13 14
16 17 18 19 20
22 23 24 25 26
28 29 30 31 32
34 35 36 37 38
```

```
The character array is :
a b
c d

The float array is :
1.23 2.34 3.33 4.44
5.55 6.66 7.77 8.88
9.99 10.1 11.11 12.12
13.13 14.14 15.15 16.16



STRINGS AND CHARACTER ARRAYS
---------------------------

There is a subtle difference between the two following declarations:

char * s = "ABC";

and

char s1[4] = {'A', 'B', 'C', '\0' );

The first creates a pointer to a character which is assigned the address of a
CHARACTER STRING LITERAL CONSTANT stored in a portion of memory which may only
be read from (the CHARACTER STRING LITERAL CONSTANT is terminated by the '\0'
character).

The second declares and initializes a 4 element of array of characters. The last
element in the array is thr null character '\0'.  The elements of the array
s1 may both be read from and written to at will.

Here is a program which explains the subtleties of the char * s = "ABC";

// Author: Ted Obuchowicz
// March 19, 2002
// example program illustrating use of



#include <iostream>
#include <string>

using namespace std;



int main()
{

char * s = "ABC";  // the compiler treats the string "ABC\0"
                   // as constant since string literals are
                   // treated as constants
                   // the pointer s points to the first character
                   // in the string... this means that strings
                   // declared in this manner may be READ from
                   // but not written to

cout << "s is " << s  << endl; // this is OK, read from the string
```

```
// try to modify s

*s = 'Z'; //  this will compile fine, but cause a SEGMENTATION FAULT
          //  during run-time

cout << "modified s is " << s  << endl;
cout << "modified s is " << s  << endl;


return 0;
}
```

As noted in the program comments, a SEGMENTATION FAULT is generated at
run-time by the above program. Here is a different version of the program
which uses the array style method of working with character strings:

```
// Author: Ted Obuchowicz
// March 19, 2002
// example program illustrating use of



#include <iostream>
#include <string>

using namespace std;



int main()
{

char  s[4] = {'A', 'B', 'C', '\0' };


cout << "s is " << s  << endl; // this is OK, it will simply start printing
                               // characters starting at address s until a '\0'
                               // is encountered

// try to modify s

*s = 'Z'; //   this is ok as the array is both read and writeable


cout << "modified s is " << s  << endl;


return 0;
}
```

This program runs fine:

```
ted@flash Programs 10:07pm >array_of_characters
s is ABC
modified s is ZBC
```

```
ted@flash Programs 10:08pm >


A string reverse function
------------------------

Here is a nice function which joins two character arrays but the second
array is copied BACKWARDS. It makes use of pointers.

// Author: Ted Obuchowicz
// March. 13, 2002
// example program illustrating use of



#include <iostream>
#include <string>

using namespace std;

void reverse_string(char s1[], char s2[])
{
 // find the end of the first string designated by the '/0' character

int string1_end = 0;
for( ; s1[string1_end] != '\0' ; string1_end++);

// the above for loop is a sneaky way of writing
// for( ; s1[string1_end] != '\0' ; string1_end++)
// {
//
// }
// note that the ; in the original for loop terminates the
// statement consisting of the for loop with no body...


int string2_end = 0;
for( ; s2[string2_end] != '\0' ; string2_end++);

// another sneaky for loop with an empty {  }

string2_end--;


int i;
for( i = string2_end ; i >= 0; i--)
 s1[ string1_end++] = s2[i];

// now add the '\0' character to s1


string1_end++;
s1[string1_end] = '\0';
}



int main()
{
```

```
char string1[50] =  "Keith";
char string2[50] = "Richards";

cout << string1 << endl;
cout << string2 << endl;

reverse_string(string1, string2);

cout << string1 << endl;

return 0;
}



ARRAY OF POINTERS
----------------

C++ lets a programmer declare and use an array of pointers:

/ Author: Ted Obuchowicz
// March 20, 2002
// example program illustrating use of
// an array of pointers


#include <iostream>
#include <string>

using namespace std;



int main()
{

char* rolling_stones[4] ;  // declare an array of 4 character pointers


// initialize the 4 pointers to chars to some string literal constants
// note, we may only READ these string constants and cannot write to
// them..

rolling_stones[0] = "Keith";
rolling_stones[1] = "Mick";
rolling_stones[2] = "Charlie";
rolling_stones[3] = "Ron";


// this will set the values of the 4 pointers to point to the starting
// address of the 4 strings

//   rolling_stones[0] -------->  _________________________
                                |'K'|'e'|'i'|'t'|'h'|'\0'|
                                --------------------------

//   rolling_stones[1] -------> ____________________
                               |'M'|'i'|'c'|'k'|'\0'|
                               ----------------------
                               _________________________________
```

```
//   rolling_stones[2] -------> |'C'|'h'|'a'|'r'|'l'|'i'|'e'|'\0'|
                                 ---------------------------------
                                 __________________
//   rolling_stones[3] -------> |'R'|'o'|'n'|'\0'|
                                 ------------------


// print out the strings

for(int i = 0; i < 4 ; i++)
  cout << rolling_stones[i] << endl;

return 0;
}


The line:

char* rolling_stones[4] ;

declares the variable named rolling_stones to be a 4 element array of
pointers to characters (i.e. each element of the array is a pointer to
a character)

the lines:

rolling_stones[0] = "Keith";
rolling_stones[1] = "Mick";
rolling_stones[2] = "Charlie";
rolling_stones[3] = "Ron";


initializes the 4 pointers to characters stored in the elements of the
array rolling_stones to 4 different string literal constants
(which may only be READ FROM ).

The 4 strings may be printed out by using the loop:

for(int i = 0; i < 4 ; i++)
  cout << rolling_stones[i] << endl;



POINTERS TO FUNCTIONS
--------------------

C++ allows one to declare POINTERS TO FUNCTIONS.

Consider the following two function prototypes:

void*   f1 (void);
void (*f2) (void);


The first declares a function which is called f1 , this function receives
no arguments and returns a void pointer.

The second declares f2 to be a POINTER TO A FUNCTION which receives no arguments
and returns nothing.
```

Here is a program which makes us of a pointer to a function and invokes the function through its pointer:

```
// Author: Ted Obuchowicz
// March 20, 2002
// example program illustrating use of
// pointers to functions


#include <iostream>
#include <string>

using namespace std;



void print_message(void)
{
 cout << "Hello World" << endl;
}


int main()
{

// declare f as a pointer to a function which returns a void and takes no argume
nt

void (*f) (void);


// now assign f the address of the print_message function

f = &print_message;

// can also do f = print_message

// run the function by invoking it indirectly through the pointer f

(*f)();


return 0;
}
```

The program output is:

```
ted@flash Programs 10:40pm > pointers_to_functions1
Hello World
ted@flash Programs 10:40pm >
```

ARRAYS OF POINTERS TO FUNCTIONS:

Consider the following declaration:

```
void (*  array[4]) (void)
```

This declares the variable called array to be an array of 4 elements. Each element of the array is a POINTER TO A FUNCTION which takes no arguments and returns nothing.

We can use this array be assigning its 4 elements the 4 addresses of some functions, then invoke the functions indirectly by dereferencing the 4 pointers stored in the array. The following program does this:

```
// Author: Ted Obuchowicz
// March 20, 2002
// example program illustrating use of
// pointers to functions


#include <iostream>
#include <string>

using namespace std;

int global_array[5] ;  // an integer array of 5 uninitialized elements

void clear(void)
{
 for(int i = 0; i < 5 ; i++)
    global_array[i] = 0;  // set all the 5 elements to 0
}

void add_one(void)
{
 for(int i = 0; i < 5 ; i++)
    global_array[i] = global_array[i] + 1;
}

void print_out(void)
{
 for(int i = 0; i < 5 ; i++)
   cout <<   global_array[i] << endl;
}



int main()
{

void (*  array[4]) (void) ; // declare an array of 4 pointers to FUNCTIONS which
 do not return a value

array[0] = &clear;
array[1] = &print_out;
array[2] = &add_one;
array[3] = &print_out;

// now call each function one-by-one

for(int i = 0; i < 4 ; i++)
   (*array[i])() ;
```

```
return 0;
}

The program output is:

ted@flash Programs 10:45pm >pointers_to_functions
0
0
0
0
0
1
1
1
1
1

POINTERS AND STRUCTS:
--------------------

Consider a struct declared as:

struct a_struct
{
int i;
char t;
};


We can declare a  variable to be of this type with the line:

a_struct struct1;

Next, we can declare a pointer to the struct with:

a_struct* ptr_to_struct;

Now, we can give the address of the variable struct1 to the pointer:

ptr_to_struct = &struct1;

We can access the individual fiels of the struct indirectly by pointer
dereferencing:

(*ptr_to_struct).i = 5;  // derefence the pointer and access the integer field
(*ptr_to_struct).t = 'x' ;  // dereference the pointer and access the char field


This method is a bit awkward and clumsy. C++ allows for a 'shorthand notation'
when working with pointers to structs:

ptr_to_struct -> i = 10;
ptr_to_struct -> t = 'z';


These ideas are summarized in the following program:

// Author: Ted Obuchowicz
//March 19 , 2002
```

```
// example program illustrating use of



#include <iostream>
#include <string>

using namespace std;



int main()
{

struct a_struct
{
int i;
char t;
};

a_struct struct1;

a_struct* ptr_to_struct;

ptr_to_struct = &struct1;  // assign the address of the variable struct1 to
                           // the pointer

(*ptr_to_struct).i = 5;  // derefence the pointer and access the integer field
(*ptr_to_struct).t = 'x' ;  // dereference the pointer and access the char field

cout << "i is " << struct1.i << endl;
cout << "t is " << struct1.t << endl;

// alternate method of accessing the elements of the struct using the
// pointer to the struct

ptr_to_struct -> i = 10;
ptr_to_struct -> t = 'z';

cout << "i is " << ptr_to_struct -> i << endl;
cout << "t is " << ptr_to_struct -> t << endl;

return 0;
}

The program output is:

ted@flash Programs 10:50pm > struct_pointers
i is 5
t is x
i is 10
t is z
ted@flash Programs 10:51pm >



THE LIBRARY <string>
---------------------
```

There is a built-in libary called <string> which has some useful
string manipulating functions. Some of these functions are:

```
char *strcpy(char *, const char *);
char *strcat(char *, const char *);
int strcmp(const char *, const char *);
```

The Deitel and Deitel book gives numerous examples of the use of
these functions. The reader is advised to refer to these examples.

DYNAMIC MEMORY ALLOCATION WITH NEW AND DELETE

C++ has another method of assigning a value to a pointer variable.
This method makes use of the new operator to DYNAMICALLY ALLOCATE
MEMORY.

It is often convenient in certain programming situations to be
able to request new memory space during program run time. C++
allows this with the new operator. The new operator is used to
request a certain amount of memory at program run time, if the call
to new is successful , new returns the starting address of this dynamically
allocate memory. If the call to new is not successful (if there is no
more memory left for example) then new returns the value 0.

Here is a simple program which dynamically allocates enough room for an
integer:

```
/ Author: Ted Obuchowicz
// Mar. 20, 2002
// example program illustrating use of
// new and delete operators


#include <iostream>
#include <string>

using namespace std;



int main()
{

int* integer_pointer ;

integer_pointer = new int;  // request enough memory from the heap
                            // to hold an integer, if the request
                            // is successful, new returns the address,
                            // if there is no more memory, new returns
                            // the value 0
                            // the bits in the dynamically allocated
                            // memory are random 1's and 0s..
```

```
// the following will print out some garbage number
// since we did not initialize the allocated memory with any
// value

cout << *integer_pointer << endl;
// before assigning a value to this dynamically and newly allocated
// memory space, we should check that the call to new was successful

if ( integer_pointer != 0 )
    *integer_pointer = 10 ; // assign the value 10 to this memory location

cout << "Address of newly allocated memory = " << integer_pointer << endl;
cout << "value stored at this address = " << *integer_pointer << endl;


// once a program no longer needs the memory which it has asked for at run
// time, it should return it to back to the heap. This is performed with the
// delete operator

delete integer_pointer ;  // this returns the 4 bytes of memory which integer_po
inter
                          // pointed to and was used to store the value of 10 in
                          // note that the pointer itself still exists and can b
e
                          // assigned another address (though a second call to n
ew for
                          // example .




return 0;
}
```

The program output is:

```
261944
Address of newly allocated memory = 0x3ff30
value stored at this address = 10
```

There is a variation of the new operator which initializes the memory allocated from the heap with some user-specified value:

```
// Author: Ted Obuchowicz
// Mar. 20, 2002
// example program illustrating use of
// new and delete operators


#include <iostream>
#include <string>

using namespace std;
```

```
int main()
{

int* integer_pointer ;

integer_pointer = new int(55);  // request enough memory from the heap
                                // to hold an integer, and give this
                                // location an initial value of 55


cout << *integer_pointer << endl;

// before assigning a value to this dynamically and newly allocated
// memory space, we should check that the call to new was successful

if ( integer_pointer != 0 )
    *integer_pointer = 10 ; // assign the value 10 to this memory location

cout << "Address of newly allocated memory = " << integer_pointer << endl;
cout << "value stored at this address = " << *integer_pointer << endl;


// once a program no longer needs the memory which it has asked for at run
// time, it should return it to back to the heap. This is performed with the
// delete operator

delete integer_pointer ;  // this returns the 4 bytes of memory which integer_po
inter
                          // pointed to and was used to store the value of 10 in
                          // note that the pointer itself still exists and can b
e
                          // assigned another address (though a second call to n
ew for
                          // example .




return 0;
}
```

The program output is:

```
55
Address of newly allocated memory = 0x3fff0
value stored at this address = 10
```

Rather than space to hold a single variable, new may be asked to dynamically allocate a contiguous sequence of memory locations (i.e. an array) this is done with

```
new <data_type> [size]
```

Here is a program which dynamically allocates an array of integers. The size of the array is specified by the user as input at run-time:

```
// Author: Ted Obuchowicz
// March 20, 2002
// example program illustrating use of



#include <iostream>
#include <string>

using namespace std;

// to circumvent this limitation of arrays


int main()
{

int size;

int* pointer ; // this pointer will be set to the starting address
               // of an array of integers. This array will be dynamically
               // allocated at run time through a call to the new operator
               // we will pass the requested size as to new

cout << "enter the size of the array you wish to create" << endl;
cin >> size;


pointer = new int [size] ;  // request space to hold size integers and assign
                            // starting address of this space to pointer

for(int i = 0 ; i < size ; i++)
{
  *(pointer + i)  = i;   // assign some values to these integers
// same as doing    pointer[i] = i;
// note: can only do this for 1 d arrays.. will not work for two dynamic arrays
}


// now print out the array

for(int i = 0 ; i < size ; i++)
{
   cout << *pointer << " " << pointer <<  endl;
   pointer++;
}


cout << pointer << endl;

pointer = pointer -size ;  // set pointer back to start of array

cout << pointer << endl;

// once we're done with the memory, give  it back

delete [] pointer;
```

```
return 0;
}

The program output is:

ted@flash Programs 10:24pm > dynamic_size_array_new
enter the size of the array you wish to create
4
0 0x40018
1 0x4001c
2 0x40020
3 0x40024
0x40028
0x40018



A variation of the above program is to use new to dynamically allocate
a two-dimensional array. The program uses the address translation mechanism
discussed earlier:

// Author: Ted Obuchowicz
// March 20, 2002
// example program illustrating use of



#include <iostream>
#include <string>

using namespace std;

// to circumvent this limitation of arrays


int main()
{

int rows;
int cols;

int* pointer ; // this pointer will be set to the starting address
               // of an array of integers. This array will be dynamically
               // allocated at run time through a call to the new operator
               // we will pass the requested size as to new

int* pointer_backup ;

cout << "enter the number of rows of the array you wish to create" << endl;
cin >> rows;

cout << "enter the number of  cols of the array you wish to create" << endl;
cin >> cols;


pointer = new int [rows*cols] ;  // request space to hold a 2-d array rows x col
s
```

```
pointer_backup = pointer;

for(int i = 0 ; i < rows ; i++)
 for(int j = 0; j < cols; j++)
{
  *(pointer + (i*cols + j))  = i+j;   // assign some values to these integers
}


// now print out the array


for(int i = 0 ; i < rows ; i++)
 for(int j = 0; j < cols; j++)
{
  cout <<   *(pointer + (i*cols + j))  << " " ;
  if (j == cols - 1 )
    cout << endl;
}



// once we're done with the memory, give  it back

delete [] pointer_backup;



return 0;
}



Note that in the above example, we had no choice but to use the
2-address translation formula together with the pointer deference
to pretend that the dynamically allocated 1-d array consisting of
enough contiguous bytes from the heap to store row*col number of integers
is organized as a 2d array of 'rows' rows and 'cols' columns:

*(pointer + (i*cols + j))

if we want to treat the dynamically allocated array as 'cols' number
of rows by 'rows' number of columns we would have used the formula:

*(pointer + (i*rows + j))


unlike the 1-d dynamic array, we do NOT have the option of using good old
array square bracket notation such as:

pointer[i][j] = some_value ; // will result in compile time error

since the compiler does not know what value to "put in the box" when
it converts the pointer[i][j] notation into an address :


                  ---
   pointer + ( i* |? |  +  j)
                  ---
```

```
Should the poor confused compiler put the number 'rows' in the box?
Or should it put the number 'cols' in the box ??? . It can't determine what
to put in the box , so it gives a compile time error..  (maybe it should
put Jack in the Box... ha ha  or maybe Jumping Jack Box)



Here is another example of the use of the new operator together
with pointers. It is a C++ function which receives a char* and will return anoth
er
char*.  The function will dynamically allocate sufficient memory to
hold the contents string pointed to by the passed parameter with ALL
VOWELS (a,e,i,o,u) removed.  The function will return a pointer to the
start of the dynamically allocated memory.  We will make use of the
strlen function found in the library <string> within your function.

#include <iostream>
#include <string>

using namespace std;

char* remove_vowels(char * str)
{
 int length = strlen(str);
 length++ ; // add one more to account for end-of-string character
 char * start = new char  [length] ; // dynamically allocate sufficient memory
 char * tmp = start; // make a copy
 while( * str != '\0')
 {
   if ((*str != 'a') && (*str != 'e') && (*str != 'i') && (*str != 'o') &&  (*st
r != 'u') )

    {
      *tmp = *str ; // copy the non-vowel characters
       tmp++;
    }

   str++; // advance the two pointers
 }
  *tmp =  '\0' ;  // add the end of string character to the end of the new strin
g
  return start;
}


int main()
{

char s1[] = "xcvbnmhjk";
char s2[] = "";
char s3[] = "aeiou";
char s4[] = "hello";
char s5[] = "good";


cout << s1 << "    " << remove_vowels(s1) << endl;
cout << s2 << "    " << remove_vowels(s2) << endl;
```

```
cout << s3 << "   " << remove_vowels(s3) << endl;
cout << s4 << "   " << remove_vowels(s4) << endl;
cout << s5 << "   " << remove_vowels(s5) << endl;

return 0;
}
```

The program output is:

```
ted@flash Programs 10:54pm >remove_vowels
xcvbnmhjk   xcvbnmhjk

aeiou
hello   hll
good   gd
```

CLASSES

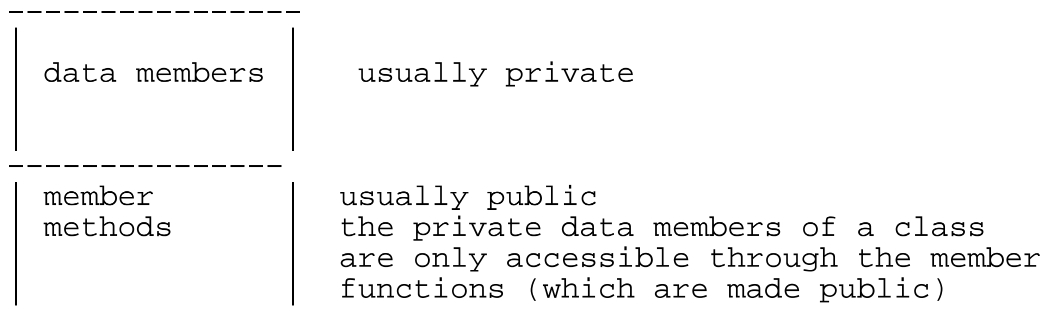
Classes are the foundation of object-oriented programming in the C++ language.

A C++ class is a collection of DATA MEMBERS (which may be of different types such as int, char, float, int*, etc.) and MEMBER FUNCTIONS (which are given the special name of METHODS).

The class data type allows for the concept of ENCAPSULATION - a grouping of data members and the functions which operate on this data into a unified "object".

The class construct allows for information hiding - a class can restrict access to either its data members or its methods. Information within a class can be hidden by declaring the information to be 'private'. Conversely, i nformation may be visible to users of the class by declaring it to be 'public'.

Here is a conceptual view of a class:



```
Circle Class Example
---------------------

Here is a very simple example of a circle class in C++.
A circle has associated with it a radius (stored as a float).
We can define two methods called Area and Circumference which will
return the area of a circle.  We will also define a method called
SetRadius(float r) which will set the radius to a certain value which
is passed as a parameter.


// Author: Ted Obuchowicz
// April 23, 2002
// example program illustrating use of
// a simple Circle Class

const float pi = 3.1415926;


#include <iostream>
#include <string>

using namespace std;




// define a class called Circle, class is a C++ keyword

class Circle
{
float radius;    // everything in a class is private by default

public:          // we want the following methods to be public, so we
                 // make them public by using the public keyword followed
                 // by a colon :

float Area();
float Circumference();
void SetRadius(float r);

};


// we now define the methods of the class, we use the scope resolution
// operator to inform the compiler that the method belongs to the class
// called Circle

float Circle::Area()
{
 return ( pi * radius * radius) ;
}

float Circle::Circumference()
{
 return ( 2 * pi * radius) ;
}

 void Circle::SetRadius(float r)
```

```
{
radius = r;
}




// let's use the class definition and the definitions of the three methods
// in a main program

int main()
{

// create two Circle "objects"

Circle mick, keith;

mick.SetRadius(2.00) ; // set the Radius of circle mick to 2.00
cout << "Circle mick has area " << mick.Area() << endl;
cout << "Circle mick has circumference " << mick.Circumference() << endl;

keith.SetRadius(5.00);
cout << "Circle keith has area " <<  keith.Area() << endl;
cout << "Circle keith has circumference " << keith.Circumference() << endl;

return 0;
}
```

The program output is:

```
Circle mick has area 12.5664
Circle mick has circumference 12.5664
Circle keith has area 78.5398
Circle keith has circumference 31.4159
```

Some comments on the program:

1) Note how we give the full name of the methods when we define
   them using the form :

   return_type   class_name::method_name

   (note there is no space between the class_name::method_name)

   This is because there may be similaryly named methods belonging to
   different classes.

2) We could have also defined the methods within the class declaration.
   This style of defining a method is known as INLINE DEFINITION.
   Here is the same program using the inline definition form for the
   method definitions:

```
// Author: Ted Obuchowicz
// April 23, 2002
// example program illustrating use of
// a simple Circle Class using inline definition for the methods
```

```
const float pi = 3.1415926;


#include <iostream>
#include <string>

using namespace std;




// define a class called Circle, class is a C++ keyword

class Circle
{
float radius;   // everything in a class is private by default

public:         // we want the following methods to be public, so we
                // make them public by using the public keyword followed
                // by a colon :

// use the inline form to define the three class methods

float Area() { return ( pi * radius * radius) ; } ;
float Circumference() { return ( 2 * pi * radius) ;} ;
void SetRadius(float r) { radius = r;};

};






// let's use the class definition and the definitions of the three methods
// in a main program

int main()
{

// create two Circle "objects"

Circle mick, keith;

mick.SetRadius(2.00) ; // set the Radius of circle mick to 2.00
cout << "Circle mick has area " << mick.Area() << endl;
cout << "Circle mick has circumference " << mick.Circumference() << endl;

keith.SetRadius(5.00);
cout << "Circle keith has area " <<  keith.Area() << endl;
cout << "Circle keith has circumference " << keith.Circumference() << endl;

return 0;
}


Of course, the output produced by this version is exactly the same as the
output produced by the earlier version:
```

```
ted@flash Programs 7:35pm >class_circle_inline
Circle mick has area 12.5664
Circle mick has circumference 12.5664
Circle keith has area 78.5398
Circle keith has circumference 31.4159


3) IT IS AN ERROR TO ATTEMPT TO DIRECTLY ACCESS THE PRIVATE
   DATA OF A  CLASS.  Suppose in a main program we try to
   read the value of mick's radius:

For example, with the same class definition as above (either the
inline one or the first one), if we write a main program such as:

// let's use the class definition and the definitions of the three methods
// in a main program
// and we try to read mick's radius directly..

int main()
{

// create two Circle "objects"

Circle mick, keith;

mick.SetRadius(2.00) ; // set the Radius of circle mick to 2.00
cout << "Circle mick has radius " << mick.radius<< endl;
return 0;

}


When we compile this program , the following errors are reported:

ted@flash Programs 7:38pm >g++ -o circle_class_read_private circle_class_read_pr
ivate.C
circle_class_read_private.C: In function `int main()':
circle_class_read_private.C:21: `float Circle::radius' is private
circle_class_read_private.C:68: within this context




USING HEADER FILES FOR CLASS DEFINITIONS
----------------------------------------

Just as we used header files to include function prototypes, we may
make use of header files to include class definitions. The following
example is our class Circle example redone using header files.  We
also make use of the #ifndef compiler directive to prevent multiple
redefinitions of identifiers in cases where a header file is included
more than one time:


We can put the definition of the class Circle in a file called
circle.h (any file name may be used... it is good to follow
the class_name.h  convention):
```

```
all this goes into the file called circle.h

-------------------------------------------------------------

#ifndef circle_h
#define circle_h

const float pi = 3.1415926;

// define a class called Circle, class is a C++ keyword

class Circle
{
float radius;   // everything in a class is private by default

public:         // we want the following methods to be public, so we
                // make them public by using the public keyword followed
                // by a colon :
Circle();   // default constructor
Circle(float r); // overloaded constructor
float Area();
float Circumference();
float GetRadius();
};  // NOTE: THIS LAST SEMICOLON IS REQUIRED !!!!

#endif

-------------------------------------------------------------------



Next, we can have another file containing the implementation of the methods
found in the class . This file can be called circle_methods.C :


file: circle_methods.C
------------------------------------------------------------------

#include "circle.h"   // this will read the file circle.h which contains the cla
ss
                      // declaration
#include <iostream>

using namespace std;






float Circle::Area()
{
 return ( pi * radius * radius) ;
}

float Circle::Circumference()
{
 return ( 2 * pi * radius) ;
```

```
}

float  Circle::GetRadius()
{
return radius;
}

// define the default constructor

Circle::Circle()
{
  cout << "Default constructor called... " << endl;
  cout << "Setting circle's radius to 1.00 " << endl;
  radius = 1.00 ;
}

// define the overloaded constructor

Circle::Circle(float r)
{
 cout << "Overloaded constructor invoked ... " << endl;
 radius = r;
}



--------------------------------------------------------------------------
```

Next, the main program can be written in a third file. Let us
call it circle_main.C:

```
file: circle_main.C
----------------------------------------------------------------

#include "circle.h"
#include <iostream>

using namespace std;




int main()
{

// create two Circle "objects"

Circle mick, keith(456.89);

cout << "Circle mick has radius " << mick.GetRadius() << endl;
cout << "Circle keith has radius " <<  keith.GetRadius() << endl;

return 0;
}

------------------------------------------------------------------------
```

We can compile the above files using the command:

g++ -o circle_main circle_methods.C circle_main.C

LINKING YOUR PROGRAM WITH A PRE-COMPILED CLASS IMPLEMENTATION

The above example showed how we separated the class DEFINITION
from its IMPLEMENTATION into two separate files. It is even
possible (and desirable in some cases) to further isolate the user
of the class from its implementation by providing only the precompiled
version of the class implementation (that is providing only the .o file
(object file)). All the user of a class needs to know is the interface
of the class: the class name, the name of its methods, etc. All this
information is provided in the class definition. The actual details of
how the methods operate are left to the implementor to decide.

We can precompile our class_methods.C program into a binary object
file called class_methods.o using the following option to
the g++ compiler:

```
-c   Compile or assemble the source files, but do not  link.
          The  compiler output is an object file corresponding to
          each source file.
```

(this was generated from the man page for g++ with the command man g++)

```
ted@townshend Programs 9:01pm >g++ -c circle_methods.C
```

This will create a binary object file called circle_methods.o
Binary files are not human readable, try doing a more
on circle_methods.o :

```
ted@townshend Programs 9:02pm >more circle_methods.o
EL4(
.shstrtab.text.rodata.eh_frame.symtab.strtab.rela.text.rela.eh_frame.commentã¿Å
Ç@ #Ç # "Çà
èã¿Å Ç@ # "Çèã¿Å "Çèã¿ ‘ @       @‘ @    @‘ÒÒ$ºÇèã¿ ò’ H‘ @      @Ð HÐ$ºÇè@IÚ@ÉÚ
Default constructor called... Setting circle’s radius to 1.00 ?Overloaded constr
u
-tor inveÿñ!t$480K\aXlh «½Ç8Õcircle_methods.Cgcc2_c
mpiled.pi__FRAME_BEGIN__Circumference__6Circleendl__FR7ostreamcout__6Circlef__ls
__7ostreamPFR7ostream_
R7ostream_Q_qtodGetRadius__6Circle__ls__7ostreamPCc__6CircleArea__6Circle__throw
^L       ^LD     H^L

^L      ^ º               ´        ^L¸^LÀ
        Ä
^LÈ      (Ì^L(Ðà         ä       ^Lè^Lð  Lô^LL
        $
```

```
^L(     P,^LP0@         D       ^LH^L88Thpas: Sun WorkShop 6 99/08/18
GCC: (GNU) 2.95.3 20010315 (release)4V
                                     løp #+@Ý3 h^L><^LMÄC
```

This is what you will see... lots of garbage.

Next, we can link our circle_methods.o with our main program by doing:

```
ted@townshend Programs 9:04pm >g++ -o circle_main circle_methods.o circle_main.C
```

This command instructs the g++ compiler to compile the source code in file
circle_main.C and LINK it together with the precompiled binary file circle_metho
ds.o
and create a single executable file called circle_main.

When you compile and link a program which makes use of library routines such
as cin , cout, etc. the linker uses precompiled object files for these routines.
Some of these routines are found in the directory /usr/lib. For example, there i
s
a file called /usr/lib/libCrun.so.1 which contains various C runtime library ut
ilities.
The .so file name extension is the so called "shared object" library extension n
ame.
There is a very nice explanation of libraries and shared libraries in the textbo
ok
"Deep C Secrets: Expert C Programming". This is a wonderfully written book; I
read it over my summer vacation at the cottage during the rainy evenings...

CONSTRUCTORS

The above circle class contains a method called SetRadius which must
be manually invoked by the user to set the radius of a certain circle
object to a specific value. This is burdensome and error-prone, a
user of the class may forget to do so.

C++ has a way of automatically initializing the data members of a class.
The method makes use of CONSTRUCTORS . Constructors are special purpose
methods (which have the same name as the class), constrcutors are
automatically called by the compiler when an object of the class is defined.

Constructors differ from ordinary functions and methods in that NO RETURN
TYPE (NOT EVEN A RETURN TYPE OF VOID) is specified in the definition of
a constructor.

DEFAULT CONSTRUCTORS

If a constructor takes no arguments, it is known as a default constructor.

OVERLOADED CONSTRUCTORS

A class defintion may contain more than one definition of a constructor,

we may define different versions of the constructor function by
OVERLOADING the constructor (that is multiple defintions which vary in their
argument list).

Circle class with a default and overloaded constructor:

We will modify our circle class example to include a default constructor
(which will set the radius of a circle object to 1.00) and an overloaded
constructor (which will set the radius to some user specified value). We
will also include a new method called GetRadius, this method will return
the radius of the circle object:

```
// Author: Ted Obuchowicz
// April 23, 2002
// example program illustrating use of
// a simple Circle Class
// using constructors

const float pi = 3.1415926;


#include <iostream>
#include <string>

using namespace std;




// define a class called Circle, class is a C++ keyword

class Circle
{
float radius;    // everything in a class is private by default

public:          // we want the following methods to be public, so we
                 // make them public by using the public keyword followed
                 // by a colon :
Circle();    // default constructor
Circle(float r); // overloaded constructor
float Area();
float Circumference();
float GetRadius();
};


// we now define the methods of the class, we use the scope resolution
// operator to inform the compiler that the method belongs to the class
// called Circle

float Circle::Area()
{
 return ( pi * radius * radius) ;
}
```

```
float Circle::Circumference()
{
 return ( 2 * pi * radius) ;
}

float  Circle::GetRadius()
{
return radius;
}

// define the default constructor

Circle::Circle()
{
  cout << "Default constructor called... " << endl;
  cout << "Setting circle's radius to 1.00 " << endl;
  radius = 1.00 ;
}

// define the overloaded constructor

Circle::Circle(float r)
{
 cout << "Overloaded constructor invoked ... " << endl;
 radius = r;
}



// let's use the class definition and the definitions of the three methods
// in a main program

int main()
{

// create two Circle "objects"

Circle mick, keith(456.89);

cout << "Circle mick has radius " << mick.GetRadius() << endl;
cout << "Circle keith has radius " <<  keith.GetRadius() << endl;

return 0;
}



The output is:

ted@flash Programs 8:23pm >circle_class_with_constructors
Default constructor called...
Setting circle's radius to 1.00
Overloaded constructor invoked ...
Circle mick has radius 1
Circle keith has radius 456.89



Here's another example of providing a value of an argument to
```

an overloaded constructor, instead of passing a literal value
as in the earlier example, we can use a variable which has been
assigned a value (either through an cin input statement or
an assignment statement). This was actually investigated by a student
who passed it along to me:

```
// Thomas Savage
// April 2016


#include <iostream>
using namespace std;

const float pi=3.14159;

class circle
{
float rad;
public:
float area()
        {
        float temp;
        return temp=rad*rad*pi;
        };
circle(float x)
        {
        rad=x;
        };
};


int main()
{
float size;
cout<<"Enter initial value of circle v1: ";
cin>>size;

circle v1(size);

cout<<"The area of circle v1 is: "<<v1.area()<<endl;

return 0;
}
```

Operator Overloading

C++ allows to overload built-in operators such as +, - , *, etc.
Let us overload the + operator such that it will "add" two circle
objects together. For the purposes of our example, we will define
the addition of two circles to give a third circle whose radius
is the sum of the two circles being added together.

```
// Author: Ted Obuchowicz
// April 23, 2002
// example program illustrating use of
```

```
// a simple Circle Class
// using constructors

const float pi = 3.1415926;


#include <iostream>
#include <string>

using namespace std;




// define a class called Circle, class is a C++ keyword

class Circle
{
float radius;   // everything in a class is private by default

public:         // we want the following methods to be public, so we
                // make them public by using the public keyword followed
                // by a colon :
Circle();   // default constructor
Circle(float r); // overloaded constructor
float Area();
float Circumference();
float GetRadius();
Circle operator+(Circle);
};


// we now define the methods of the class, we use the scope resolution
// operator to inform the compiler that the method belongs to the class
// called Circle

float Circle::Area()
{
 return ( pi * radius * radius) ;
}

float Circle::Circumference()
{
 return ( 2 * pi * radius) ;
}

float  Circle::GetRadius()
{
return radius;
}

// define the default constructor

Circle::Circle()
{
  cout << "Default constructor called... " << endl;
  cout << "Setting circle's radius to 1.00 " << endl;
  radius = 1.00 ;
}
```

```
// define the overloaded constructor

Circle::Circle(float r)
{
 cout << "Overloaded constructor invoked ... " << endl;
 radius = r;
}


Circle Circle::operator+(Circle a_circle)
{
 Circle temp ; // note default constructor will be called on temp
              // so the class better contain a definition of it

 temp.radius = radius + a_circle.radius;

 return temp;
}

// let’s use the class definition and the definitions of the three methods
// in a main program

int main()
{

// create two Circle "objects"

Circle mick, keith(456.89);

// create a third circle

Circle ron;


cout << "Circle mick has radius " << mick.GetRadius() << endl;
cout << "Circle keith has radius " <<  keith.GetRadius() << endl;


// add mick and keith to get ron

ron = mick + keith ; // ron’s radius should be 1.00 + 456.89

cout << "Circle ron has radius " << ron.GetRadius() << endl;

return 0;
}


The program output is:

ted@flash Programs DING! >circle_class_overload
Default constructor called...
Setting circle’s radius to 1.00
Overloaded constructor invoked ...
Default constructor called...
Setting circle’s radius to 1.00
Circle mick has radius 1
Circle keith has radius 456.89
```

```
Default constructor called...
Setting circle's radius to 1.00
Circle ron has radius 457.89


NOTE:

The line

ron = mick + keith ;

can also be written as:

ron = mick.operator+(keith);


Written in this manner, it is clearer that we are actually invoking the
method called operator+ belonging to the circle mick and passing to this
method the circle keith as a parameter. The return value of the operator+
method is then assigned to the circle ron.

If we write it as ron = mick + keith, it is more apparent that we are
"adding" two circles together and assigning the result of the "addition"
to another circle.




DIFFERENCES BETWEEN A C++ STRUCT AND A C++ CLASS
------------------------------------------------

A C++ struct  is very similar to a C++ class.  In fact , the only
difference between C++ structs and classes is that in A STRUCT
EVERYTHING IS CONSIDERED PUBLIC (unless explicitly denoted as private)
BY DEFAULT AND IN A CLASS EVERYTHING IS CONSIDERED PRIVATE (unless
explicitly denoted as public by use of the public keyword).  Traditionally,
C++ programmers use the class construct when we want to encapsulate
the data members togethers with the methods which operate on these members.
A struct is traditionally used only when there are data members (all public)
and no methods.  As always, one can abandon tradition and be a rebel. So,
let us be a bit rebellious and use a struct which has some methods in addition
to data members:

// Author: Ted Obuchowicz
//  Oct. 29, 2002
// file: fraction_struct_class.C



#include <iostream>
#include <string>

using namespace std;


struct fraction
{
 private: // in a struct everything is public unless explicitly
          // made private
```

```
int numerator, denominator;

 public:

fraction() {  numerator = 1; denominator = 1; } ; // inline definition of defaul
t
                                                   // constructor

fraction(int top, int bottom ) { numerator = top; denominator = bottom; };

fraction  add(fraction x)
{
 fraction temp;
 temp.numerator = numerator * x.denominator + x.numerator * denominator;
 temp.denominator = denominator * x.denominator;
 return temp;
};

void print_fraction()
{
 cout << numerator << " / " << denominator << endl;
}

};




int main()
{

fraction f1(2,3) ;
fraction f2(3,4) ;
fraction sum;
f1.print_fraction();
f2.print_fraction();
sum =  f1.add(f2);
sum.print_fraction();






return 0;
}
```

This example illustrates the INLINE method of defining the methods of the struct (or a class as well) . Note if we use the inline methos there is no need to use the scope resolution operator together with the class name to give the "full name" as was done in the previous circle class examples. Note that we use a method called add (instead of overloading the + operator which we could have done if we so desired... this is left as an exercise to the interested reader) which performs the addition of two fraction objects. N

o reduction
of the fraction to its "lowest common denominator" is performed. Again, the inte
rested
reader is encouraged to do this as an exercise.

A COMPLEX NUMBER CLASS

Here is a more representative example of a C++ class which
provides for addign two complex numbers together (by overloading
the + operator). Recall that a complex number consists of a
real part and an imaginary part:

complex_number = real_part + i (imaginary_part)

where i is the square root of -1.

Complex numbers are used frequently in electrical engineering,
most notably in the study of electromagnetic theory. Complex
numbers are less frequently used in digital design ... but I digress..

Some comments on the program follow:

```
#include <iostream.h>

class Complex
{
 double real;
 double imag;

public:


Complex() {real = 0.0; imag = 0.0;}
Complex(double,double);
void write()
{
 cout << "Real = " << real << endl;
 cout << "Imaginary = " << imag << endl;
}

Complex operator+(const  Complex) const  ;

};

Complex::Complex(double r, double i)
{
 real = r;
 imag = i;
}

Complex  Complex::operator+(const  Complex c) const
{
 Complex temp; // note default constructor will be called on object temp
               // hence one must be provided or else
 temp.real = real + c.real;
```

```
 temp.imag = imag + c.imag;
 return temp;

}

void main(void)
{
 Complex c1;
 Complex c2(7.7,4.4);
 Complex c3(1.1,2.2);
 c1 = c2 + c3;
 c1.write();
}


The output is:

ted@flash Programs 9:11pm >complex
Real = 8.8
Imaginary = 6.6


Comments:
--------

The const at the end of the declaration of the operator+

Complex  Complex::operator+(const  Complex c) const
                                              ^^^^^


                                            This const over here
simply guarantees that the object which is calling
(ie. object c2 in our sample program)  the method
operator+ will NOT be changed by the method.

The const in the :

Complex  Complex::operator+(const  Complex c) const
                             ^^^^
                          this const guarantees that the
operator+ method will not change the object which is passed to it
as a paramter (ie. the object c3 in the sample program).

If the method operator+ attempted to do something stupid like change
the values of either the calling object or the object received
as a parameter, it would be flagged as an error by the compiler.



DESTRUCTORS
-----------

Constructors are used to perform INITIALIZATION of data members.
This initialization may  involve some DYNAMIC MEMORY allocation
(for example assign some pointer an address returned by the new
operator).  Recall that when an object which dynamically allocates
some memory goes out of scope, the memory used for that object's data
members are returned to the operating system (and given back to the
heap space0, HOWEVER ANY DYNAMICALLY ALLOCATED MEMORY IS NOT RETURNED.
```

Thus, a memory leak is said to occur, and we have the possibility
of exhausting the heap space causing the program to crash.

This is illustrated in the program below:

```
#include <iostream.h>
#include <stdlib.h>

const long OneMillion = 1000000;

class Big
{
 char* string ; // a data member which is a pointer to a character..i.e a string

public:

 Big(); // default constructor which will do some DYNAMIC memory allocation

};

Big::Big()
{
 string = new char [OneMillion];  // get 1 000 000 char cells
 if (!string)
   {
    cout << "OUT OF MEMORY!!!!" << endl;
    exit(1);
   }
}


void garbage(void)
{
 Big oops;

 // do some local processing of object oops

}

void main(void)
{

int i;

for(i = 0; i < 10000; i++)
 {
   cout << "I = " << i << endl;  // eventually we will get a segmentation fault
   garbage();
 }

}
```

We have a class which has a single character pointer has a data member.
There is also a default constructor defined which assigns the pointer
the starting address of a dynamically allocated array of 1 000 000 bytes.

There is a function called garbage which simply creates an object
of class Big (recall that the default constructor will be called

```
whenever an object of class Big is declared).

In the main program, we have a for loop which calls the function garbage
10000 times.  Each time through the loop, the default constructor
will be called (allocating 1 000 000 bytes from the heap), eventually
we will exhaust all of the available memory and the program will crash.
On my workstation , this occured during loop iteration 229:

My output is:

I = 223
I = 224
I = 225
I = 226
I = 227
I = 228
I = 229
Abort


Solutions:
----------

There are two solutions. One is to have the user of the class,
explicitly give back any dynamically allocate memory with the delete
operator when that memory is no longer needed:

#include <iostream.h>
#include <stdlib.h>

const long OneMillion = 1000000;

class Big
{
 char* string ; // a data member which is a pointer to a character..i.e a string

public:

 Big(); // default constructor which will do some DYNAMIC memory allocation
void  clear(); // method  which will deallocate dynamic memory

};

Big::Big()
{
 string = new char [OneMillion];  // get 1 000 000 char cells
 if (!string)
    {
     cout << "OUT OF MEMORY!!!!" << endl;
     exit(1);
    }
}

void  Big::clear()
{
 delete [] string;
}

void garbage(void)
```

```
{
 Big oops;

 // do some local processing of object oops

 // using this cumbersome method we must remember to invoke the method
 // clear before returning from from the function garbage

 oops.clear();

 // this is not elegant as it relies upon the person implementing function garba
ge
 // to remember to delete any memory which was dynamically allocated by the
 // constructor function Big()

 // woudn't it be simpler if there were a way to have this done automatically so
mehow..?

}

void main(void)
{

int i;

for(i = 0; i < 10000; i++)
 {
   cout << "I = " << i << endl;
   garbage();
 }

}
```

Now, there is no problem. However, this method is cumbersome. It
relies upon the user to call the method clear() explicitly.

There is a better way which uses DESTRUCTORS.

DESTRUCTOR functions are special functions inside of class objects,
they NEITHER TAKE ANY ARGUMENTS, NOR DO THEY RETURN AVALUE; DESTRUCTORS
ARE HIGHLY SPECIALIZED FUNCTIONSWHICH EXIST ONLY TO RELEASE ANY MEMORY
WHICH WAS DYNAMICALLY ALLOCATED BY A CONSTRUCTOR!!!!!!

A DESTRUCTOR's name is similar to the constructor in the respect that
each is the same as the class name. However, a destructor name is
preceded with the tilde (~) symbol.

Here is a program which makes use of the destructor ~Big()

```
#include <iostream.h>
#include <stdlib.h>

const long OneMillion = 1000000;

class Big
```

```
{
 char* string ; // a data member which is a pointer to a character..i.e a string

public:

 Big(); // default constructor which will do some DYNAMIC memory allocation
 ~Big(); // destructor which will deallocate dynamic memory

};

Big::Big()
{
 string = new char [OneMillion];  // get 1 000 000 char cells
 if (!string)
    {
     cout << "OUT OF MEMORY!!!!" << endl;
     exit(1);
    }
}

Big::~Big()
{
 delete [] string;
}

void garbage(void)
{
 Big oops;

 // do some local processing of object oops

}

void main(void)
{

int i;

for(i = 0; i < 10000; i++)
 {
   cout << "I = " << i << endl;
   garbage();
 }

}



This program complets its for loop with no memory problems:

I = 9988
I = 9989
I = 9990
I = 9991
I = 9992
I = 9993
I = 9994
I = 9995
I = 9996
```

```
I = 9997
I = 9998
I = 9999
ted@flash Programs 9:43pm >
```

MORE ON CONSTRUCTORS

Constructors are automatically invoked by the compiler when a class object is declared. As such, they cannot be invoked in the "regular" manner as a method function is invoked. The following program will generate a compile-time error:

```
#include <iostream>
#include <string>

using namespace std;

class C
{
 int x ;
public:
 C();
 void print(void);
} ;

// define the default constructor

C :: C()
{
 cout << "default constructor called" << endl;
 x = 1;
};

 void C :: print(void)
{
 cout << x << endl;
}




int main()
{

C x;
x.print();

// see if we can invoke a constructor explicitly...

x.C();
```

```
// nope it gives a compile time error
//constructor.C: In function ‘int main()’:
//constructor.C:44: error: calling type ‘C’ like a method



return 0;
}


In this program, we attempt to invoke explicitly the constructor C() using
the "object_name.method_name() style.  This results in a compile time error.

There is a way to explicitly invoke the constructor function, but is has to be
called at the time of the object’s declaration as in:

#include <iostream>
#include <string>

using namespace std;

class C
{
 int x ;
public:
 C();
 C(int num);
 void print(void);
} ;

// define the default constructor

C :: C()
{
 cout << "default constructor called" << endl;
 x = 1;
};


C :: C(int num)
{
 cout << "overloaded constructor called" << endl;
 x = num;
};

 void C :: print(void)
{
 cout << x << endl;
};




int main()
{

C x = C(55);   // make a call to the overloaded constructor at time of object de
claration
```

```
                // to perform initialization
x.print();


C y = C();  // make a call to the default constructor at time of object declarat
ion
            // to perform initialization of data member

y.print();


return 0;
}
```

The program output is:

```
overloaded constructor called
55
default constructor called
1
```

Member Initialization in Constructors:

C++ allows an alternate form for a constructor to perform member initialization by a mechanism known a "base/member initialization". The following program illustrates its use:

```
#include <iostream>
#include <string>

using namespace std;


class X
{
private:

 int i;
 float f;
 char c;

public:

X(int first=1, float second=2.0, char third='a') : i(first) , f(second) , c(thir
d) { }
void print() { cout << i << " " << f << " " << c << endl;}

};



int main()
{

X var1;
```

```
var1.print();

return 0;
}


We may override the default arguments provided by the definition of the construc
tor,
by providing them when we declare an object:

int main()
{

X var1, var2(6,2.34,'z');
var1.print();
var2.print();

return 0;
}

will produce the output:

1 2 a
6 2.34 z


What if we had a third object declared as:

X var3(7) ;

the value of 7 would be used to initialize the data member i, and
the other two date members would use the provided default values in the
constructor ( 2.0 and 'a' )




COPY CONSTRUCTORS
=================

There is another type of constructor known as a "copy" constructor
which is provided by the compiler when a class does not contain
a user-provided version.  It is invoked whenever an object is
declared and soem existing object is used to make a "copy" of the
existing object into the newly declared object.  The following program
shows how the compiler copy constructor is used:


// Author: Ted Obuchowicz
// file: copy_constructor.C



#include <iostream>
#include <string>

using namespace std;
```

```
class X
{
private:

 int i;
 float f;
 char c;

public:

X() { i = 1 ; f = 2.0 ; c = 'a' ; cout << "default called" << endl;}
X(int first, float second, char third) { i = first; f = second; c = third;
                                          cout << "overloaded called" << endl;
                                        }
void print() { cout << i << " " << f << " " << c << endl;}

};



int main()
{

X var1, var2(10,100.0,'z');
X var3(var1) ; // the compiler provided version of a "copy" constructor will be
used
               // to copy the data values of var1 into var3
var1.print();
var2.print();
var3.print(); // will be the same as var1




return 0;

}



The output is :

default called
overloaded called
1 2 a
10 100 z
1 2 a


In the line:

X var3(var1) ;

the previously declared object var1 is used as an argument to the
compiler provided version of a a copy construcor which simply copies
the values of the data members of object var1 into those of object
var3. In other words, the compiler provided version copy constructor
performs:
```

```
var3.i = var1.i
var3.f = var1.f
var3.c = var1.c


The programmer can provide their own version of the so-called
copy cosnturctor as in the following example:


// Author: Ted Obuchowicz
// file: copy_constructor2.C



#include <iostream>
#include <string>

using namespace std;


class X
{
private:

 int i;
 float f;
 char c;

public:

X() { i = 1 ; f = 2.0 ; c = 'a' ; cout << "default called" << endl;}
X(int first, float second, char third) { i = first; f = second; c = third;
                                          cout << "overloaded called" << endl;
                                        }
X(const X& arg) { cout << "programmer provided copy constructor invoked " << end
l;
           i = arg.i ;
           f = arg.f;
           c = arg.c;
         }

void print() { cout << i << " " << f << " " << c << endl;}

};



int main()
{

X var1, var2(10,100.0,'z');
X var3(var1) ; // the programmer  provided version of a "copy" constructor will
be used
               // to copy the data values of var1 into var3
var1.print();
var2.print();
var3.print(); // will be the same as var1
```

```
return 0;

}


The program output is:


default called
overloaded called
programmer provided copy constructor invoked
1 2 a
10 100 z
1 2 a


NOTE: The argument to the copy constructor MUST BE A
CONSTANT REFERENCE ARGUMENT or else a compile time
error would result as in:

copy_constructor2.C:26:8: error: invalid constructor; you probably meant âM-^@
M-^XX (const X&)âM-^@M-^Y
 X(X arg) { cout << "programmer provided copy constructor invoked " << endl;


The reason for requiring that the argument be a constant reference is
in the interest of efficiency, otherwise if pass by value were
the parameter passing mechanism for the copy constructor, a copy
of the argument would be created on the stack resulting in run-time
inefficiencies.  The const simply guarantees that the passed
asrgument CANNOT be modified by the copy constructor. Attempting to
do so would again result in a compile-time error :


X(const X& arg) { cout << "programmer provided copy constructor invoked " << end
l;
           i = arg.i ;
           f = arg.f;
           c = arg.c;
           // this will cause a compile time error
          arg.i = 55 ;
         }



The compiler generates the following error:


copy_constructor3.C: In copy constructor âM-^@M-^XX::X(const X&)âM-^@M-^Y:
copy_constructor3.C:31:17: error: assignment of member âM-^@M-^XX::iâM-^@M-^Y in
 read-only object
           arg.i = 55 ;




WHEN IS A COPY CONSTRUCTOR INVOKED ?
```

In addition to explicity invoking a copy cosntructor as in the
previous example, whenever a class contains a method which
receives by value an argument of the class type, the copy
constructor is invoked (either the compiler provided version,
or if provided the programmers version. We extend our example
to include a method is_equal(X arg) :

```
bool X::is_equal(X arg ) // note the copy constructor will be called whenever me
thod
                         // is equal is invoked
{
  return ( (i == arg.i ) && ( f == arg.f ) && ( c == arg.c ) )  ;
}
```

consider the line in main():

```
if ( var3.is_equal(var1) )
 cout << "Equal" << endl;
else
 cout << "Not equal" << endl;
```

The programmer provided copy constructor will be invoked since
the values of the passed argument var1 must be copied into the
argument arg in method is_equal. This is indicated by the program output:

```
default called
overloaded called
programmer provided copy constructor invoked  --> from the line: X var3(var1) ;
1 2 a
10 100 z
1 2 a
programmer provided copy constructor invoked  --> from the line: if ( var3.is_eq
ual(var1)

Equal
```

If we do not want to copy constructor to be invoked when a method
whichr receives an argument of a class, we can simply define the
method such it RECEIVES a REFERENCE ARGUMENT RATHER THAN A VALUE
ARGUMENT:

```
bool X::is_equal(X& arg ) // note the copy constructor will NOT BE INVOKED
                          // since the method receives a reference to an argumen
t

{
  return ( (i == arg.i ) && ( f == arg.f ) && ( c == arg.c ) )  ;
}
```

This time the output is:

```
ted@deadflowers Programs >copy_constructor5
default called
overloaded called
programmer provided copy constructor invoked  --> produced by X var3(var1)
1 2 a
10 100 z
1 2 a
Equal
```

If we extend this simple example to contain a destructor,
it should be noted that if pass-by-value is used as the
parameter passing mechansim for a class method, then the
destructor would be invoked upon return from the method (since
the argument goes out of scope upon the return from the method).
If pass-by-reference is used for the argument, then neither the
copy constructor nor a descturctor would be invoked . This is
illustrated in the following examples:

```
// Author: Ted Obuchowicz
// file: copy_constructor6.C



#include <iostream>
#include <string>

using namespace std;


class X
{
private:

 int i;
 float f;
 char c;

public:

X() { i = 1 ; f = 2.0 ; c = 'a' ; cout << "default called" << endl;}
X(int first, float second, char third) { i = first; f = second; c = third;
                                         cout << "overloaded called" << endl;
                                       }
X(const X& arg) { cout << "programmer provided copy constructor invoked " << end
l;
           i = arg.i ;
           f = arg.f;
           c = arg.c;
         }

~X() { cout << "Destrutor called " << endl; }

void print() { cout << i << " " << f << " " << c << endl;}
bool is_equal(X arg ) ;

};
```

```
bool X::is_equal(X arg ) // note the copy constructor will BE INVOKED

{
  return ( (i == arg.i ) && ( f == arg.f ) && ( c == arg.c ) )  ;
 // also the destructor will be invoked
}

int main()
{

X var1, var2(10,100.0,'z');
X var3(var1) ; // the programmer  provided version of a "copy" constructor will
be used
               // to copy the data values of var1 into var3
var1.print();
var2.print();
var3.print(); // will be the same as var1


if ( var3.is_equal(var1) )
 cout << "Equal" << endl;
else
 cout << "Not equal" << endl;



return 0;

}
```

The output is:

```
default called
overloaded called
programmer provided copy constructor invoked
1 2 a
10 100 z
1 2 a
programmer provided copy constructor invoked  --> from line if ( var3.is_equal(v
ar1) )
Destrutor called  --> produced when program returns from method is_equal
Equal
Destrutor called  --> when v3 goes out of scope
Destrutor called  --> when v2 goes out of scope
Destrutor called  --> when v1 goes out of scope.. note the exact order is not kn
own
```

If is_equal is changed to pass-by-reference, neither the copy constructor nor destructor are invoked upon entry and return from the function.

```
bool X::is_equal(X& arg ) // note the copy constructor will NOT BE INVOKED

{
  return ( (i == arg.i ) && ( f == arg.f ) && ( c == arg.c ) )  ;
 // also the destructor will NOT  be invoked
}


ted@deadflowers Programs  >copy_constructor7
default called
overloaded called
programmer provided copy constructor invoked
1 2 a
10 100 z
1 2 a
Equal
Destrutor called
Destrutor called
Destrutor called
```

ASSIGNMENT of one class object to another of the same type

An object of a class may be assigned to another object of the same class.

```
#include <iostream>
#include <string>
using namespace std;


class X
{
private:

 int i;
 float f;
 char c;

public:

X() { i = 1 ; f = 2.0 ; c = 'a' ; cout << "default called" << endl;}
X(int first, float second, char third) { i = first; f = second; c = third;
                                         cout << "overloaded called" << endl;
                                       }
void print() { cout << i << " " << f << " " << c << endl;}

};



int main()
{
```

```
X var1, var2(10,100.0,'z');
var1.print();
var2.print();

var2 = var1 ;  // the compiler provides a default assignment operator which simp
ly
               // performs a member by member assignment

var2.print();


return 0;

}


The output is:

default called
overloaded called
1 2 a
10 100 z
1 2 a


We can always provide our own version of the overloaded assignment (=) operator
instead of relying upon the one which is provided default by the compiler:



#include <iostream>
#include <string>
using namespace std;

class X
{
private:

 int i;
 float f;
 char c;

public:

X() { i = 1 ; f = 2.0 ; c = 'a' ; cout << "default called" << endl;}
X(int first, float second, char third) { i = first; f = second; c = third;
                                         cout << "overloaded called" << endl;
                                       }
void print() { cout << i << " " << f << " " << c << endl;}
void operator=(X some_object); // copy const. will be invoked
};


void X::operator=(X some_object)
{
 cout << "using the user-defined assignment operator instead of compiler provide
d"
      << endl;
```

```
// now simply copy the values of the data members of some_object

i = some_object.i;
f = some_object.f;
c = some_object.c;

};


int main()
{

X var1, var2(10,100.0,'z');
var1.print();
var2.print();

var2 = var1 ;  // use the programmer supplied version of the assignment operator
               // this is the same as var2.operator=(v1);



var2.print(); // print out the values of the data members after the assignment

return 0;

}
```

The output is:

```
default called
overloaded called
1 2 a
10 100 z
using the user-defined assignment operator instead of compiler provided
1 2 a
```

Why would we want to provide our own version of the assignment operator
when the compiler always provides its own default one??? There are certain
cases in which a class contains a pointer variable in which the compiler provide
d
assignment operator is "not good enough". In order to understand why we have to
learn a little about garbage and dangling pointers.....

Garbage and Dangling Pointers

The following program produces what is known as a "dangling pointer":

```
#include <iostream>
#include <string>

using namespace std;



int main()
{

int* ptr ; // declare a pointer to an int

ptr = new int (5); // request space from the heap , initialize this space to 5
                  // and assign address of allocated space to ptr


cout << "The value of ptr is " << ptr << " and the data stored in this location
is "
     << *ptr << endl;


delete ptr ; // give back the allocated space  to the heap


// ptr is now called a dangling pointer.. it does not point to any
// valid memory location... it should not be dereferenced until it
// has been assigned some valid address... note the value of
// ptr is not affected by the delete operator... it still
// holds the same address... except that this address is "no longer
// in existence"

cout << "The value of ptr after the delete is " << ptr << endl;



return 0;
}
```

Garbage is the result of when we dynamically allocate memory from the heap, assign it’s address to some pointer variable, and then assign some other address to the pointer variable BEFORE we have deleted the allocated memory (returned it back to the heap). The allocated memory can now NEVER be returned back to the heap (at least until the program completes execution) since we now longer have its address available. Such memory is referred to as GARBAGE. The following program illustrates how a program may generate garbage.

```
#include <iostream>
#include <string>

using namespace std;



int main()
{
```

```
int* a = new int(10);

cout << "an integer with value 10  has been allocated from the heap at address "
 << a << endl;


a = new int(2) ;    // the memory which was allocated to hold the integer
                    // with initial value is lost and cannot be reclaimed
                    // since we no longer have its address ... It is
                    // said to be GARBAGE.


cout << " the original pointer holding the integer value with 10 is now assigned
 a new address of" << a << endl;

delete a  ; // give back the second integer to the heap


return 0;
}
```

The output is:

```
an integer with value 10  has been allocated from the heap at address 0x21200
 the original pointer holding the integer value with 10 is now assigned a new ad
dress of0x21210
```

If a class contains data members which have been made to point to dynamically
allocated memory , and a default assignment operator is used to assign one objec
t
to another, garbage can result as in the following example:

```
#include <iostream>
#include <string>
using namespace std;


class X
{
private:

int* ptr;

public:

X() { ptr = new int ;}
void show_address() { cout << "Value of ptr is: " << ptr << endl;}
};



int main()
{
```

```
X var1, var2;
var1.show_address();
var2.show_address();

var2 = var1 ;  // the compiler provides a default assignment operator which simp
ly
               // performs a member by member assignment


var1.show_address();
var2.show_address();


return 0;

}


The output is:

Value of ptr is: 0x211e8
Value of ptr is: 0x211f8
Value of ptr is: 0x211e8
Value of ptr is: 0x211e8
```

After the statement var2 = var1 ; we no longer have a pointer to the
memory which was allocated from 0x211f8 because var2.ptr has been overwritten
with value of var1.ptr (which points to some other place in the heap). We have
created some garbage at address 0x211f8 and this garbage cannot be de-allocated
with the delete operator since we no longer have its address.

Even if the class contains a destructor function, the use of the compiler
provided assignment operator will result in a dangling pointer and garbage.
The details of the "picture" of main memory in the case of a user-provided
destructor functions which simply performs ~X() { delete ptr } is left to
the interested reader... I highly encourage you to go through this constructive
exercise.

We can provide our own version of the assignment operator which circumvents
this problematic behaviour:

```
// Author: Ted Obuchowicz
//  nov. 14, 2008




// Author: Ted Obuchowicz
//  nov. 14, 2008
```

```
#include <iostream>
#include <string>

using namespace std;


class X
{
private:

int* ptr;

public:

X() { ptr = new int(5) ;} // allocated memory from the heap and intialize it to
5
void show_address() { cout << "Value of ptr is: " << ptr <<
                      " Value stored in this location is " << *ptr << endl;}
void operator=(X arg);
};


void X::operator=(X arg)
{
 if ( ptr != 0)
 {
   cout << "About to delete the space at address " << ptr << endl;
   delete ptr;
 };
 ptr = new int; // allocate space from the heap
 *ptr = *(arg.ptr); // fill this newly allocated memory with the same value as t
he argument
};


int main()
{

X var1, var2;
var1.show_address();
var2.show_address();


var2 = var1 ; // no more garbage

var1.show_address();
var2.show_address();


return 0;

}


//output:


//Value of ptr is: 0x213e8 Value stored in this location is 5
```

```
//Value of ptr is: 0x213f8 Value stored in this location is 5
//About to delete the space at address 0x213f8
//Value of ptr is: 0x213e8 Value stored in this location is 5
//Value of ptr is: 0x213f8 Value stored in this location is 5




When a class contains a constructor which dynamically allocates memory,
and a destructor which returns any allocate dmemory back to the heap,
one then needs to be concerned with the subtleties involved in copy
constructors, overloading of the = operator, and pass-by-reference vs
pass-by value.  The following example shows the subtle errors which
ma arise from incorrect programming.



#include <iostream>
using namespace std;

class Vector
{
 int * elements;  // each vector will be consist of 3 int values
                  // pointed to by elements which is initialized
                  // by the constructor

public:  // some methods

Vector() ; // default constructor will simply initialize all elems to 0
Vector(int,int,int); // overloaded constructor with passed values
~Vector(); // destructor

// some overloaded operators

Vector operator+(Vector& v1); // will return as a Vector the element by element
addition of two vectors
void show(void); // prints out the address of where the array is stored at and t
he values of the array contents
};

Vector::Vector()
{
 cout << "Default Constructor called!!" << endl;
 elements  = new int[3]; // allocate room for 3 integers
 for(int i = 0; i < 3; i++)
    elements[i] = 0;     // and initialize values to zero
                         // same as *(elements + i ) = 0
  cout << "elements found at address " << elements << endl;
}

Vector::Vector(int i1, int i2, int i3)
{
 cout << "Constructor Called !! " << endl;
 elements  = new int[3]; // allocate room for 3 integers
 elements[0] = i1; // same as *(elements + 0 ) = i1
 elements[1] = i2; // same as *(elements + 1 ) = i2
 elements[2] = i3;
}
```

```
Vector::~Vector()
{
 cout << "Destructor called!!" << endl;
 delete [] elements;
 cout << "Returned to the heap the array at address " << elements << endl;
}



Vector Vector::operator+(Vector& v1) // what would happen if we passed by value
instead of
                                     // reference ?????
                                     // answer  .. default copy const would be i
nvoked and
                                     // destructor would be invoked when we retu
rn from operator+
                                     // and the program crashes since destructor
 deletes the "copy" array
{
 cout << "Inside method operator+ " << endl;
 Vector temp; // constructor will be called for temp
 temp.elements[0] = elements[0] + v1.elements[0];
 temp.elements[1] = elements[1] + v1.elements[1];
 temp.elements[2] = elements[2] + v1.elements[2];
 cout << "about to return temp from methos operator+ " << endl;
 return temp; // note the destructor will be called for temp when it goes out of
 scope
}



void Vector::show(void)
{
 cout << "Array stored at address " << elements << endl;
 for(int i = 0; i < 3; i++)
   cout << elements[i] << "   " ;
 cout << endl << endl;

}

int  main(void)
{

cout << "Main beginning execution" << endl;

Vector v1, v2(3,3,3), v3(3,3,3) ;
cout << "V1 = " <<  endl;
v1.show();
cout << "V2 = " <<  endl;
v2.show();
cout << "V3 = " <<  endl;
v3.show();

v1 = v2 + v3 ; // same as v1 = v2.operator+(v3);
cout << "This is result of vector addition " <<
endl;
cout << "V1 = " <<  endl;
v1.show();
```

```
cout << "Main ending execution...." << endl;

}



The program output is:

ted@deadflowers Programs 4:26pm >vector_with_subtle_error_stripped
Main beginning execution
Default Constructor called!!
elements found at address 0xc12010
Constructor Called !!
Constructor Called !!
V1 =
Array stored at address 0xc12010
0   0   0

V2 =
Array stored at address 0xc12030
3   3   3

V3 =
Array stored at address 0xc12050
3   3   3

Inside method operator+
Default Constructor called!!   ---> constructor called for temp
elements found at address 0xc12070  ---> address of temp
about to return temp from methos operator+
Destructor called!!    ---> called for temp upon return from operator+
Returned to the heap the array at address 0xc12070
This is result of vector addition
V1 =
Array stored at address 0xc12070
0   0   6     ---> WRONG VALUES SHOULD BE 6 6 6

Main ending execution....
Destructor called!!
Returned to the heap the array at address 0xc12050   --> address of  v3
Destructor called!!
Returned to the heap the array at address 0xc12030   ---> address of v2
Destructor called!!
Returned to the heap the array at address 0xc12070   ---> address of new v1 (not
 same as 0xc12010)
                                                     ---> garbage created at 0xc
12010



The line: v1 = v2 + v3 , which is the same as
v1 = v2.operator+(v3) causes the return values of operator+ of
v2 (i.e. the fileds of temp) to be copied into the fileds of
v1 since we the compiler provided version of the = operator
will be used.  This will cause GARBAGE at original address
of v1 ( = 0xc12010) since this value is overwritten with
the value of 0xc12070) and the new value of v1.elements
( which is now 0xc12070 ) is in reality a DANGLING POINTER.
```

DRAW THE PICTURE OF MEMORY TO VERIFY THIS FOR YOURSELF.

As a first attempt to overload the = operator to solve this
problem, let's first consider a more simple version of this
program which does not use the operator+, buit simply
defines an overloaded version of the = operator.

```
#include <iostream>
using namespace std;

class Vector
{
 int * elements;  // each vector will be consist of 3 int values
                  // pointed to by elements which is initialized
                  // by the constructor

public:  // some methods

Vector() ; // default constructor will simply initialize all elems to 0
Vector(int,int,int); // overloaded constructor with passed values
~Vector(); // destructor

// some overloaded operators

Vector operator+(Vector& v1); // will return as a Vector the element by element
addition of two vectors
void operator=(Vector arg);
void show(void); // prints out the address of where the array is stored at and t
he values of the array contents
};

Vector::Vector()
{
 cout << "Default Constructor called!!" << endl;
 elements  = new int[3]; // allocate room for 3 integers
 for(int i = 0; i < 3; i++)
    elements[i] = 0;     // and initialize values to zero
                         // same as *(elements + i ) = 0
 cout << " constructor found elements at " << elements << endl;
}

Vector::Vector(int i1, int i2, int i3)
{
 cout << "Constructor Called !! " << endl;
 elements  = new int[3]; // allocate room for 3 integers
 elements[0] = i1; // same as *(elements + 0 ) = i1
 elements[1] = i2; // same as *(elements + 1 ) = i2
 elements[2] = i3;
}

Vector::~Vector()
{
 cout << "Destructor called!!" << endl;
 delete [] elements;
 cout << "Returned to the heap the array at address " << elements << endl;
```

```
}



Vector Vector::operator+(Vector& v1) // what would happen if we passed by value
instead of
                                     // reference ?????
                                     // answer  .. default copy  constructor  wo
uld be invoked and
                                     // destructor would be invoked when we retu
rn from operator+
{
 cout << "Inside method operator+ " << endl;
 Vector temp; // constructor will be called for temp
 temp.elements[0] = elements[0] + v1.elements[0];
 temp.elements[1] = elements[1] + v1.elements[1];
 temp.elements[2] = elements[2] + v1.elements[2];
 cout << "about to return temp from methos operator+ " << endl;
 return temp; // note the destructor will be called for temp when it goes out of
 scope
}


void Vector::operator=(Vector arg) // copy constructor invoked
{
  cout << "Using programmer provided definition of = operator " << endl;
  for ( int i = 0 ; i < 3 ; i++)
  {
    elements[i] = arg.elements[i] ;
  }

// destructor for arg will be invoked
}


void Vector::show(void)
{
 cout << "Array stored at address " << elements << endl;
 for(int i = 0; i < 3; i++)
   cout << elements[i] << "   " ;
 cout << endl << endl;

}

int  main(void)
{

cout << "Main beginning execution" << endl;

Vector v1, v2(3,3,3), v3(3,3,3) ;
cout << "V1 = " <<  endl;
v1.show();
cout << "V2 = " <<  endl;
v2.show();
cout << "V3 = " <<  endl;
v3.show();

v1 = v2 ;
cout << "This is result of vector addition " <<
```

```
endl;
cout << "V1 = " <<  endl;
v1.show();
cout << "V2 = " <<  endl;
v2.show();
cout << "V3 = " <<  endl;
v3.show();

cout << "Main ending execution...." << endl;

}
```

This program is contains a subtle error as indicated by the
incorrect output:

```
Main beginning execution
Default Constructor called!!
 constructor found elements at 0x1ecc010
Constructor Called !!
Constructor Called !!
V1 =
Array stored at address 0x1ecc010
0   0   0

V2 =
Array stored at address 0x1ecc030
3   3   3

V3 =
Array stored at address 0x1ecc050
3   3   3

Using programmer provided definition of = operator
Destructor called!! --> destructor invoked for argument arg (which is a copy of
v2)
Returned to the heap the array at address 0x1ecc030
This is result of vector addition
V1 =
Array stored at address 0x1ecc010
3   3   3

V2 =
Array stored at address 0x1ecc030
0   0   3

V3 =
Array stored at address 0x1ecc050
3   3   3

Main ending execution....
Destructor called!!
Returned to the heap the array at address 0x1ecc050
Destructor called!!
Returned to the heap the array at address 0x1ecc030
Destructor called!!
Returned to the heap the array at address 0x1ecc010
```

since the operator=(Vector arg) will invoke the copy constructor
when we call it in main with the line v1 = v2 in main(), the
fields of arg are the same as those of v2 . Futhermore, upon return
from method operator=(), the destructor for argument arg is invoked
and the memory at address of v2.elements will be deleted. Our version
of the = operator copies the array elements pointed to by v2.elements
correctly into the v1.elements (without changing the original value of
v1.elements), but when we subsequently try to access the array elements of
v2 with the v2.show(), we see wrong value of:

0   0   3

since v2.elements has become a DANGLING pointer due to the desctructor
beign invoked on the passed argument to operator=()...

THE SOLUTION TO THIS IS TO USE PASS BY REFERENCE for the
definition of operator=(). Recall, no copy constructor
will be invoked and no destrucor will be invoked:

file: vector_with_subtle_error_stripped_with_overloaded_assignment_operator_pass_by_ref.C

The changes to the operator=() are simply:

```
void Vector::operator=(Vector& arg)
{
  cout << "Using programmer provided definition of = operator " << endl;
  for ( int i = 0 ; i < 3 ; i++)
  {
    elements[i] = arg.elements[i] ;
  }
}
```

The correct output is:

```
Main beginning execution
Default Constructor called!!
 constructor found elements at 0x1489010
Constructor Called !!
Constructor Called !!
V1 =
Array stored at address 0x1489010
0   0   0

V2 =
Array stored at address 0x1489030
3   3   3

V3 =
Array stored at address 0x1489050
3   3   3
```

```
Using programmer provided definition of = operator
This is result of vector addition
V1 =
Array stored at address 0x1489010
3   3   3

V2 =
Array stored at address 0x1489030
3   3   3

V3 =
Array stored at address 0x1489050
3   3   3

Main ending execution....
Destructor called!!
Returned to the heap the array at address 0x1489050
Destructor called!!
Returned to the heap the array at address 0x1489030
Destructor called!!
Returned to the heap the array at address 0x1489010



Now, we can correct the original program by using operator+() together
with the correct version of operator=() which uses pass by reference:

file: vector_with_subtle_error_stripped_solved_with_overloaded_assignment_operat
or_pass_by_ref.C


#include <iostream>
using namespace std;

class Vector
{
 int * elements;  // each vector will be consist of 3 int values
                  // pointed to by elements which is initialized
                  // by the constructor

public:  // some methods

Vector() ; // default constructor will simply initialize all elems to 0
Vector(int,int,int); // overloaded constructor with passed values
~Vector(); // destructor

// some overloaded operators

Vector operator+(Vector& v1); // will return as a Vector the element by element
addition of two vectors
void operator=(const Vector& arg);
void show(void); // prints out the address of where the array is stored at and t
he values of the array contents
};

Vector::Vector()
{
 cout << "Default Constructor called!!" << endl;
 elements  = new int[3]; // allocate room for 3 integers
```

```
 for(int i = 0; i < 3; i++)
    elements[i] = 0;      // and initialize values to zero
                          // same as *(elements + i ) = 0
 cout << " constructor found elements at " << elements << endl;
}

Vector::Vector(int i1, int i2, int i3)
{
 cout << "Constructor Called !! " << endl;
 elements  = new int[3]; // allocate room for 3 integers
 elements[0] = i1; // same as *(elements + 0 ) = i1
 elements[1] = i2; // same as *(elements + 1 ) = i2
 elements[2] = i3;
}

Vector::~Vector()
{
 cout << "Destructor called!!" << endl;
 delete [] elements;
 cout << "Returned to the heap the array at address " << elements << endl;
}



Vector Vector::operator+(Vector& v1) // what would happen if we passed by value
instead of
                                     // reference ?????
                                     // answer  .. default copy  constructor  wo
uld be invoked and
                                     // destructor would be invoked when we retu
rn from operator+
{
 cout << "Inside method operator+ " << endl;
 Vector temp; // constructor will be called for temp
 temp.elements[0] = elements[0] + v1.elements[0];
 temp.elements[1] = elements[1] + v1.elements[1];
 temp.elements[2] = elements[2] + v1.elements[2];
 cout << "about to return temp from methos operator+ " << endl;
 return temp; // note the destructor will be called for temp when it goes out of
 scope
}


void Vector::operator=(const Vector& arg)
{
  cout << "Using programmer provided definition of = operator " << endl;
  for ( int i = 0 ; i < 3 ; i++)
  {
    elements[i] = arg.elements[i] ;
  }
}


void Vector::show(void)
{
 cout << "Array stored at address " << elements << endl;
 for(int i = 0; i < 3; i++)
   cout << elements[i] << "   " ;
 cout << endl << endl;
```

```
}

int  main(void)
{

cout << "Main beginning execution" << endl;

Vector v1, v2(3,3,3), v3(3,3,3) ;
cout << "V1 = " <<  endl;
v1.show();
cout << "V2 = " <<  endl;
v2.show();
cout << "V3 = " <<  endl;
v3.show();

v1 = v2 + v3  ;
cout << "This is result of vector addition " <<
endl;
cout << "V1 = " <<  endl;
v1.show();
cout << "V2 = " <<  endl;
v2.show();
cout << "V3 = " <<  endl;
v3.show();

cout << "Main ending execution...." << endl;

}


The correct output is now:

Main beginning execution
Default Constructor called!!
 constructor found elements at 0x14e7010
Constructor Called !!
Constructor Called !!
V1 =
Array stored at address 0x14e7010
0   0   0

V2 =
Array stored at address 0x14e7030
3   3   3

V3 =
Array stored at address 0x14e7050
3   3   3

Inside method operator+
Default Constructor called!!
 constructor found elements at 0x14e7070
about to return temp from methos operator+
Using programmer provided definition of = operator
Destructor called!!
Returned to the heap the array at address 0x14e7070
This is result of vector addition
V1 =
```

```
Array stored at address 0x14e7010
6   6   6

V2 =
Array stored at address 0x14e7030
3   3   3

V3 =
Array stored at address 0x14e7050
3   3   3

Main ending execution....
Destructor called!!
Returned to the heap the array at address 0x14e7050
Destructor called!!
Returned to the heap the array at address 0x14e7030
Destructor called!!
Returned to the heap the array at address 0x14e7010
```

Friend Functions

The private data members belonging to a class are only accessible to the
member functions of the class and special functions which are defined to
be friend functions.

Defining a function to be a friend function to a class allows it to access
the private date members of the class.

A friend function can either be a method belonging to some other class, or
it may be a top-level function not belonging to any particular class.

The code below declares two classes : friendlyclass and a_friend. Class
friendlyclass defines the method a_friend::f() to be its friend. Class
friendly_class also declares a top-level function top_func() to be a friend
as well. This allows both the method a_friend::f() and the function top()
direct access to all the private date members belonging to class
friendly_class.

```
// Author: Ted Obuchowicz
//  April 16, 2008
// file: friend.C



#include <iostream>
#include <string>

using namespace std;

class friendlyclass ; // a forward declaration telling the compiler that
                       // friendly_class is the name of a class

class a_friend
{
 int member;
```

```
 public:
 void  f(friendlyclass) ; // method f will receive an object of type
                          // friendlyclass as an argument
};




class friendlyclass
{
 int my_private;
 public :
 friendlyclass(); // constructor
 void  my_method(); // friendlyclass's method
 friend  void  top_func(friendlyclass);
 friend  void a_friend::f(friendlyclass); // note the use of the scope
                                          // resolution operator
};


void a_friend::f(friendlyclass some_friend )
{
 cout << "The value of the private data as seen by the friend is " <<
          some_friend.my_private << endl;
};


void friendlyclass:: my_method()
{
 cout << "The value of the private data as seen by the my_method is " << my_priv
ate << endl;
};


friendlyclass::friendlyclass() // constructor function
{
 my_private = 5;
};


void top_func(friendlyclass some_friend)
{
 cout << " The value of the private data as seen by the top function is " <<
 some_friend.my_private << endl;
};


int main()
{


friendlyclass keith;
a_friend mick;

mick.f(keith); // invoke method f() of object mick which is a friend function
               // of object keith
top_func(keith);
keith.my_method();
```

```
return 0;
}
```

This program contains what is known as a FORWARD DECLARATION of a class. The lin
e

class friendlyclass ;

tells the compiler that the string of characters friendlyclass is the name of a
class which will be declared at a later point in the source code. This is neces
sary
since the class a_friend contains a method f which receives an argument of type
friendlyclass and the definition of the class called friendlyclass appears after
the definition of the class a_friend. If the forward definition did not exist,
a compile time error would oocur at the line:

```
#include <iostream>
#include <string>

using namespace std;

// class friendlyclass ; // a forward declaration telling the compiler that
                       // friendly_class is the name of a class

class a_friend
{
 int member;
 public:
 void  f(friendlyclass) ;  // compiler would give an error here if the
                           // forward definition of class friendlyclass
                           // did not exist as the compiler would have no
                           // idea what the string of characters friendlyclass
                           // represents.. by putting in the forward declaration
                           // we are in effect telling the compiler that
                           // friendlyclass is the name of a class
};
```

If we want to declare all the methods of a class to be friend functions to
some other class, we can list all of them as friends as in the following example
:

```
// Author: Ted Obuchowicz
//  April 16, 2008
// file: friend2.C



#include <iostream>
#include <string>
```

```
using namespace std;

class friendlyclass ; // a forward declaration telling the compiler that
                       // friendly_class is the name of a class

class a_friend
{
 int member;
 public:
 void  f(friendlyclass) ; // will be declared as a friend function to class
                          // friendlyclass
 void  g(friendlyclass); // is another  friend function to class friendlyclass
};




class friendlyclass
{
 int my_private;
 public :
 friendlyclass(); // constructor
 friendlyclass(int); // overloaded constructor
 void  my_method(); // friendlyclass’s method
 friend  void  top_func(friendlyclass);
 friend  void a_friend::f(friendlyclass); // note the use of the scope resolutio
n
                                          //operator
 friend  void a_friend::g(friendlyclass);
};


// in the definition of class friendlyclass we list all the method of class
a_friend to be friend functions to class friendlyclass (along with top_func() )

void a_friend::f(friendlyclass some_friend )
{
 cout << "The value of the private data as seen by the friend is " <<
          some_friend.my_private << endl;
};

void a_friend::g(friendlyclass some_friend)
{
  cout << "Hello from g method   the value of the private data is"  <<
           some_friend.my_private << endl;
}




void friendlyclass:: my_method()
{
 cout << "The value of the private data as seen by the my_method is " << my_priv
ate << endl;
};


friendlyclass::friendlyclass()
{
```

```
 my_private = 5;
};

friendlyclass::friendlyclass(int x)
{
 my_private = x;
};


void top_func(friendlyclass some_friend)
{
 cout << " The value of the private data as seen by the top function is " <<
 some_friend.my_private << endl;
};


int main()
{


friendlyclass keith, ron(25);
a_friend mick;

mick.f(keith); // invoke method f() of object keith which is a friend function o
f object mick
mick.g(ron);
top_func(keith);
top_func(ron);
keith.my_method();
ron.my_method();

return 0;
}


The output produced is:

ted@brownsugar Programs 11:49am >friend2
The value of the private data as seen by the friend is 5
Hello from g method   the value of the private data is25
 The value of the private data as seen by the top function is 5
 The value of the private data as seen by the top function is 25
The value of the private data as seen by the my_method is 5
The value of the private data as seen by the my_method is 25



Note that in this program, mick is an object of type a_friend, hence mick's meth
ods
( f and g ) may access the private data of objects of type friendlyclass. The ma
in
program declares two such objects : keith and ron.  Hence, we can invoke methods
f and g of object mick and pass to these methods either objects keith or ron
as illustrated in the program.




Instead of explicitly listing all the methods of class a_friend as friend functi
ons
in class friendlyclass, we can define the ENTIRE class a_friend as a FRIEND CLAS
```

```
S
to class friendlyclass. This makes all the methods declared in class a_friend to
be friend functions to class friendlyclass.  This method is shown in the
following example:



// Author: Ted Obuchowicz
//  April 16, 2008
// file: friend3.C




#include <iostream>
#include <string>

using namespace std;

class friendlyclass ; // a forward declaration telling the compiler that
                       // friendly_class is the name of a class

class a_friend
{
 int member;
 public:
 void  f(friendlyclass) ; // will be declared as a friend function to class frie
ndlyclass
 void  g(friendlyclass); // is another  friend function to class friendlyclass
};





class friendlyclass
{
 int my_private;
 public :
 friendlyclass(); // constructor
 friendlyclass(int); // overloaded constructor
 void  my_method(); // friendlyclass's method
 friend  void  top_func(friendlyclass);
//  friend  void a_friend::f(friendlyclass); // note the use of the scope resolu
tion operator
//  friend  void a_friend::g(friendlyclass);
    friend  class a_friend ;                // note use of "class a_friend"

//  instead of explicitly listing all the methods of class a_friend which we wan
t to
//  declare as friends to class friendlyclass, we can simply declare the ENTIRE
class
//  a_friend to be a friend to friendlyclass, this makes ALL the methods declare
d in class
//  a_friend to be friends to class friendlyclass.
//  a_friend to be friends to class friendlyclass.
//  note that the keyword "class" must precede the name of the class (a_friend)

};


void a_friend::f(friendlyclass some_friend )
```

```
{
 cout << "The value of the private data as seen by the friend is " <<
          some_friend.my_private << endl;
};

void a_friend::g(friendlyclass some_friend)
{
  cout << "Hello from g method   the value of the private data is"  <<
           some_friend.my_private << endl;
}




void friendlyclass:: my_method()
{
 cout << "The value of the private data as seen by the my_method is " << my_priv
ate << endl;
};


friendlyclass::friendlyclass()
{
 my_private = 5;
};

friendlyclass::friendlyclass(int x)
{
 my_private = x;
};


void top_func(friendlyclass some_friend)
{
 cout << " The value of the private data as seen by the top function is " <<
 some_friend.my_private << endl;
};


int main()
{


friendlyclass keith, ron(25);
a_friend mick;

mick.f(keith); // invoke method f() of object keith which is a friend function o
f object mick
mick.g(ron);
top_func(keith);
top_func(ron);
keith.my_method();
ron.my_method();

return 0;
}


The program output is identical to the precvious example where we explicitly
```

listed all the functions of a_friend to be friends of friendlyclass:

```
ted@brownsugar Programs 12:12pm >friend3
The value of the private data as seen by the friend is 5
Hello from g method   the value of the private data is25
 The value of the private data as seen by the top function is 5
 The value of the private data as seen by the top function is 25
The value of the private data as seen by the my_method is 5
The value of the private data as seen by the my_method is 25
```

STATIC DATA MEMBERS

A static data member is shared all the objects of a particular class, unlike
a non-static data member which belongs to a particular object of the class.

Here is a simple example which makes use of a static data member:

```
// Author: Ted Obuchowicz
// April 16, 2008
// file:  static_member.C



#include <iostream>
#include <string>

using namespace std;

class stones
{
 int x ;
 static int static_data_member; //  declare (but not define ) shared static data
                                // member
public:
 stones() ; // default constructor
 void inc_data();
 void show_static();
};

int stones::static_data_member = -1965;  // define the value of the static membe
r
                                         // OUTSIDE OF ALL BLOCKS , even outside
 of
                                         // main()


void stones::inc_data()
{
 static_data_member++;
};

void stones::show_static()
{
```

```
 cout << "Value of static data member is " << static_data_member << endl;
}

stones::stones()
{
 x =1 ;
};


int main()
{

stones mick, keith;

mick.inc_data();
keith.inc_data();
mick.show_static();
keith.show_static();

return 0;
}
```

The program output is:

```
ted@brownsugar Programs 1:52pm >static_member
Value of static data member is -1963
Value of static data member is -1963
```

There is only copy of the integer static_data_member and this is shared
by the two objects of class stones. Each object of class stones (mick and
keith) have their own copy of their private data member called x.

Static data members are useful for keeping track of the number of objects
of a particular class which exist during the course of a program's execution
(i.e. during run-time). We can have the class constructor function increment
the value of a static data member and have the class desctructor function
decrment the value. A class method can be used to display the value of the
static data member which represents the number of objects of the particular
class. The following program illustrates the method:

```
// Author: Ted Obuchowicz
// April 16, 2008
// file:  static_member2.C



#include <iostream>
#include <string>

using namespace std;




class stones
{
```

```
 int x ;
 static int count; //  declare (but not define ) shared static data member
public:
 stones() ; // default constructor
~stones() ; // destructor
 void show_static();
};

int stones::count = 0;  // define the value of the static member
                        // OUTSIDE OF ALL BLOCKS , even outside of
                       // main()



void stones::show_static()
{
 cout << " At this point in time, we have  " << count  <<
         " number of objects of the type class stones " << endl;
}

stones::stones() // constructor
{
 x =1 ;   //initialize the value of the private and non-static data member
 count++; // and update the static data member which is used to keep track of th
e number
          // of class objects in existence
};


stones::~stones() // destructor
{
 count--;

// destructor will be invoked everytime the object goes out of scope
}





int main()
{

stones mick, keith;

mick.show_static(); // will show 2
keith.show_static(); // will still show 2

stones ron ;
mick.show_static() ; // will now display 3

// define a block
{

stones charlie;
keith.show_static() ; // will show 4

} ; // at this point, object charlie has gone out of scope.. poor charlie..
```

```
ron.show_static(); // back to 3 objects of class stones now
return 0;
}
```

The program output is as expected:

```
ted@brownsugar Programs 2:12pm >static_member2
 At this point in time, we have  2 number of objects of the type class stones
 At this point in time, we have  2 number of objects of the type class stones
 At this point in time, we have  3 number of objects of the type class stones
 At this point in time, we have  4 number of objects of the type class stones
 At this point in time, we have  3 number of objects of the type class stones
```

This example illustrated another use for a class descructor function
, other than the use we see previosly whihc was used to delete any
constrcutor dynamically allocated memory .

Consider the following program:

```
#include <iostream>
#include <string>
#include <math.h>
using namespace std;


class One
{
 protected:
 float a;
 public:
  One()  ;      // a  constructor
  void f1(void);   // a method
  void f2(void);   // another method ... f2() will call f1() as part of its exec
ution
};

One::One()
{
 a =  1.0;
}

void One::f1(void)
{
 cout << "Hello there... I am function f1() found inside class One " << endl <<
endl << endl;
}

void One::f2(void)
```

```
{
 cout << "Hi ! i'm function f2() found inside class One " << endl;
 cout << "I'm about to call f1() but which f1() will I call ???" << endl << endl
;;
 // now invoke function f1();
 f1();
}

// derived class

class Two : public One
{
 public:
 void f1(void) ; // this overrides class One's f1()
};

void Two::f1()
{
 cout << "This is f1() found inside class Two " << endl << endl << endl;
}




int main()
{


One object1;
Two object2;

object1.f2() ;

object2.f2() ;

object2.f1()  ;



return 0;
}


The output is:

Hi ! i'm function f2() found inside class One
I'm about to call f1() but which f1() will I call ???

Hello there... I am function f1() found inside class One


Hi ! i'm function f2() found inside class One
I'm about to call f1() but which f1() will I call ???

Hello there... I am function f1() found inside class One


This is f1() found inside class Two
```

As before, the first two lines of output are caused by the line:

```
object1.f2() ;
```

The next two lines are caused by:

```
object2.f2() ;
```

Even though there is a definition of f1() inside of class Two which overrides
the definition found in class One, the one found in class One will be the versi
on
which is actually called at program run-time. This is known as STATIC BINDING a
nd
is performed by the compiler at compile time. In other words, since in the defi
nition
of f2() inside of class One, the compiler encountered f1(), this will cause any
invocations of f2() to always use the definition of f1() as defined in the base
class,
irrespective of whether it was an object of the derived class Two which actually
called
it's inherited f2(). Of course, we can always make a call to any "overridden" d
efinitions
appearing in class Two directly as in the line:

```
object2.f1()  ;  // this will use "overridden" definition found in Class Two
```

In order to allow the overridden defintion of f1() to be used IF f2() is invoked
by
an object of class Two, we MUST DEFINE f1() as VIRTUAL functions. This is done b
y
simply adding the keyword virtual in front of the defintion of the function with
in
each class as in:

```
#include <iostream>
#include <string>
#include <math.h>
using namespace std;


class One
{
 protected:
 float a;
 public:
  One()  ;      // a  constructor
  virtual  void f1(void);   // a method declared to be virtual
  void f2(void);   // another method ... f2() will call f1() as part of its exec
ution
};

One::One()
```

```
{
 a =  1.0;
}

void One::f1(void)
{
 cout << "Hello there... I am function f1() found inside class One " << endl <<
endl
<< en
dl;
}

void One::f2(void)
{
 cout << "Hi ! i'm function f2() found inside class One " << endl;
 cout << "I'm about to call f1() but which f1() will I call ???" << endl << endl
;;
 // now invoke function f1();
 f1();
}

// derived class

class Two : public One
{
 public:
 virtual void f1(void) ; // this overrides class One's f1()
};

void Two::f1()
{
 cout << "This is f1() found inside class Two " << endl << endl << endl;
}




int main()
{


One object1;
Two object2;

object1.f2() ;

object2.f2() ;

object2.f1()  ;



return 0;
}


The output will be;

Hi ! i'm function f2() found inside class One
```

```
I'm about to call f1() but which f1() will I call ???

Hello there... I am function f1() found inside class One


Hi ! i'm function f2() found inside class One
I'm about to call f1() but which f1() will I call ???

This is f1() found inside class Two


This is f1() found inside class Two



The first two lines of output are produced by

object1.f2() ;


object1 is of type class One and f2 in class One eventually calls f1. since
there is a f1() found in class One, this definition of f1 is used.



The next two lines of output are produced by:

object2.f2() ;


object2 is of type class Two, it inherits f2() from class One, BUT since
f1() is decalred as a virtual function, at run-time, we are able to
determine that since there is a definition of f1() in class two which
OVERRIDES the definition found in class One, this invocation of f2() will
call the overridden definition of f1() since the OBJECT WHICH IS CALLING F2()
is of class Two.


The next two lines are produced by object2 invoking the "overridden" definition
of f1() ( it will se the definition of f1 as found in class Two):

object2.f1()  ;




Here is another example, except this time we have three nested functions.
This is the non-virtual version:


#include <iostream>
#include <string>
#include <math.h>
using namespace std;


class One
{
 protected:
```

```
 float a;
 public:
  One()  ;      // a  constructor
  void f1(void);   // a method
  void f2(void);   // another method ... f2() will call f1() as part of its exec
ution
  void f3(void);   // another method ... f3() will call f2()   ( which will call
 f1() )
};

One::One()
{
 a =  1.0;
}

void One::f1(void)
{
 cout << "Hello there... I am function f1() found inside class One " << endl <<
endl << endl;
}

void One::f2(void)
{
 cout << "Hi ! i'm function f2() found inside class One " << endl;
 cout << "I'm about to call f1() but which f1() will I call ???" << endl << endl
;;
 // now invoke function f1();
 f1();
}


void One::f3(void)
{
 cout << "Hi ! i'm function f3() found inside class One " << endl;
  cout << "I'm about to call f2() but which f2() will I call ???" << endl << end
l;;
 f2();
}

// derived class

class Two : public One
{
 public:
 void f1(void) ; // this overrides class One's f1()
 void f2(void) ; // this overrides class Twos' f2()
};

void Two::f1()
{
 cout << "This is f1() found inside class Two " << endl << endl << endl;
}


void Two::f2()
{
 cout << "This is f2() found inside class Two " << endl << endl << endl;
}
```

```
int main()
{


One object1;
Two object2;

object1.f3() ;

object2.f3() ;

object2.f1()  ;
object2.f2();




return 0;
}



The output is:

Hi ! i'm function f3() found inside class One
I'm about to call f2() but which f2() will I call ???

Hi ! i'm function f2() found inside class One
I'm about to call f1() but which f1() will I call ???

Hello there... I am function f1() found inside class One


Hi ! i'm function f3() found inside class One
I'm about to call f2() but which f2() will I call ???

Hi ! i'm function f2() found inside class One
I'm about to call f1() but which f1() will I call ???

Hello there... I am function f1() found inside class One


This is f1() found inside class Two


This is f2() found inside class Two



Now, we make f2() and f1() to be virtual functions:


#include <iostream>
#include <string>
#include <math.h>
```

```
using namespace std;


class One
{
 protected:
 float a;
 public:
  One()  ;      // a  constructor
  virtual void f1(void);   // a method
  virtual void f2(void);   // another method ... f2() will call f1() as part of
its execution
  void f3(void);   // another method ... f3() will call f2()   ( which will call
 f1() )
};

One::One()
{
 a =  1.0;
}

void One::f1(void)
{
 cout << "Hello there... I am function f1() found inside class One " << endl <<
endl << endl;
}

void One::f2(void)
{
 cout << "Hi ! i'm function f2() found inside class One " << endl;
 cout << "I'm about to call f1() but which f1() will I call ???" << endl << endl
;;
 // now invoke function f1();
 f1();
}


void One::f3(void)
{
 cout << "Hi ! i'm function f3() found inside class One " << endl;
  cout << "I'm about to call f2() but which f2() will I call ???" << endl << end
l;;
 f2();
}

// derived class

class Two : public One
{
 public:
 virtual void f1(void) ; // this overrides class One's f1()
 virtual void f2(void) ; // this overrides class Twos' f2()
};

void Two::f1()
{
 cout << "This is f1() found inside class Two " << endl << endl << endl;
}
```

```
void Two::f2()
{
 cout << "This is f2() found inside class Two " << endl << endl << endl;
}




int main()
{


One object1;
Two object2;

object1.f3() ;

object2.f3() ;

object2.f1()  ;
object2.f2();





return 0;
}



Can you guess what the output will be this time?

Hi ! i'm function f3() found inside class One
I'm about to call f2() but which f2() will I call ???

Hi ! i'm function f2() found inside class One
I'm about to call f1() but which f1() will I call ???

Hello there... I am function f1() found inside class One


Hi ! i'm function f3() found inside class One
I'm about to call f2() but which f2() will I call ???

This is f2() found inside class Two


This is f1() found inside class Two


This is f2() found inside class Two
```

```
Stepwise Refinement
-------------------

Stepwise refinement is a method of developing an algorithm starting
with a very high-level description of the approach, and then refining
the description into a sequence of lower-level steps.  Once we have described
the the solution to the problem as a sequence of very low-level steps, it is a
very simple matter to translate the steps in the syntax of a C++ program (or any
other language ).

Several examples will be used to illustrate the method of using stepwise refinem
ent.

Example: Perfect Numbers

An integer is called PERFECT if it is equal to the sum of its divisors. The firs
t
two perfect numbers are 6 ( 6 = 1 + 2 + 3) and 28 ( 28 = 1 + 2 + 4 + 7 + 14 ).
Stepwise refinement will be used to develop a program which prints the first 100
perfect numbers.

Print the first 100 perfect numbers    ------->  ( we will use the -----> symbol
                                                   as the refinement operator)


First Refinement
-----------------

count = 0  ; // use a counter to keep track of how many perfect numbers
             // have been found

number = 1 ;  // start the search from 1

do
{
  check to see if the number is perfect;

  if (number is perfect)
     print the number;
     increment the value of count;

  number = number + 1 ; // repeat the above with the next number in the sequence
 1,2,3,...

} while (count < 100);




Second Refinement
-----------------
```

```
We will break down the step:

check to see if the number is perfect;

into a sequence of more primitive operations:

check to see if the number is perfect -------->

add up all the divisors of the number;
number will be perfect if sum of its divisors == number;



Third Refinement:
----------------

add up all the divisors of the number   -------->


sum_of_divisors = 0 ;
for( int j = 1 ; j < number ; j++ )   // loop thru all the possible numbers
   if ( number % j == 0 )   // check to see that the remainder is 0
      sum_of_divisors = sum_of_divisors + j;  // it remainder is 0 then j is a d
ivisor
                                              // of number so add it to the sum
so far


number will be perfect if sum of its divisors == number   ------->


if (number == sum_of_divisors )
{
   print out the number;
   count = count + 1;
}




Now, we can put all the refinements together into a complete C++ program
We must make sure to reassign the value of sum_of_divisors back to zero,
every time we finished processing a number in our do-while loop.

The complete program is:

// Author: Ted Obuchowicz
// Oct. 10, 2002
// file : perfect2.C



#include <iostream>

using namespace std;
```

```
int main()
{

unsigned long int sum_of_divisors = 0;
unsigned long int number = 1;
int count = 0;

do
{
   sum_of_divisors = 0;
   for( unsigned long int j = 1 ; j < number; j++)
      if ( number % j == 0)
          sum_of_divisors = sum_of_divisors + j;


   if (sum_of_divisors == number)
   {
     cout << number << " is a perfect number " << endl;
     count = count + 1;
   }

   number = number + 1;

} while ( count < 100 );


return 0;
}
```

It is interesting to note thar running this program on a Sun Blade 1000
workstation eqipped with a 900 MHZ UltraSparc III 64-bit processor for
over three hours only prodcues the first four perfect numbers:

```
ted@townshend Programs 2:43pm >perfect2
6 is a perfect number
28 is a perfect number
496 is a perfect number
8128 is a perfect number
```

Also, the unsigned long int data type was used just in case an int was too small
.

Example: The 3k + 1 sequence

The not-so-famous "3k + 1" sequence is defined as follows:

Begin with an integer k and repeat the following steps:

```
    if (k is an even number)
         k = k / 2  ;   // integer division wanted !
    else
```

```
         k = 3*k + 1;


The lenght of a sequence is defined as the number of terms it contains.
The sequence terminates when a term becomes equal to 1.

Here are some typical sequences for several starting numbers:

k  = 3:

3 10 5 16 8 4 2 1   --> this sequence has length 8

k = 4:

4 2 1   ---> this sequence has length 3

k = 5:

5 16 8 4 2 1

k = 7:

7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

k = 9:

9 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1


First refinement:
----------------

Print smallest value of SEED for which lenth >= 100.



Second Refinement:
------------------


Start with SEED = 1 and increase it until a solution is found:


SEED = 1 ;

do
{
  SEED = SEED + 1;
  compute the length of the resulting sequence for this SEED;

} while ( lenght < 100 );

print out the SEED


Third Refinement:
----------------

We will break down the step:
```

```
compute the length of the resulting sequence for this SEED

as follows:

length = 1;
term = SEED;

while (the sequence is not terminated)
  {
    compute the next term;
     length = length + 1;
  }



We can refine the :

the sequence is not terminated   ---->


(term != 1 )

We can refine the :

compute the next term ----->


if (term is even)
   term = term /2;
else
   term = 3 * term + 1;



We can refine the

(term is even)   ----> ( term % 2 == 0)


Final Refinement:
----------------

Putting all the refinements together yields:

SEED = 1 ;

do
{
  SEED = SEED + 1;
  length = 1;
term = SEED;

while (term != 1)
  {
   if (term % 2 == 0)
     term = term /2;   // make sure this is performed as integer division
   else
    term = 3 * term + 1;
```

```
   length = length + 1;
  }


} while ( lenght < 100 );

print out the SEED



Once we have our algorithm fully refined and described it is a simple
task to translate it into a legal C++ program:


// Author: Ted Obuchowicz
// Oct. 10, 2002
// file:  3k+1.C



#include <iostream>

using namespace std;



int main()
{

unsigned long int seed = 1;
unsigned long int term;
unsigned long int length;

do
{
 seed = seed + 1;
 term = seed;
 length = 1;

 cout << "Testing seed value = " << seed << endl;

 while ( term != 1)
 {
    if (term % 2 == 0 )   // term is even
       term = term / 2 ;   // perform in integer division
    else
       term = (3*term + 1);

    cout << " "  << term ;
    length = length + 1;
 };

 } while (length < 100);

cout << seed << endl;
```

```
return 0;
}




Example:  Generating a bar graph giving the frequency distribution of random
          numbers between 0 and 6 .


First refinement:
----------------

 Generate a bar graph giving the frequency  distribution of
 a sample of random numbers all between 0 and 6.

For example,  the program should generate:

0. *************************************
1. **********************************
2. ***********************************
3. ****************************************
4. **************************************
5. ***************************************
6. **********************************


As we generate a larger number of random numbers, the "height" of
each bar will become more and more equal to each other. The above
distribution corresponds to a total of 2000 random numbers being
generated. If we generate 10 times this amount, the bar chart
will appear as:


0. ***************************************
1. ***************************************
2. ****************************************
3. ***************************************
4. **************************************
5. ***************************************
6. **************************************








Second refinement :
-------------------



for (int i = 0 ; i < 2000 ; i++)    // this controls how many random numbers
{                                    // we will compute
    generate a random number between 0 and 6;
    keep track of how many times each number occurred;
};

Print the bar chart using 40 *s for the number which occurred the MOST
```

```
number of times, and all the other numbers will have their number of *s
scaled accordingly ;





Third refinement:
-----------------

we will refine the "generate a random number between 0 and 6"
by making use of the built-in rand() function:

random_value = rand() % 7 ;

The logic behind this is that the  possible remainders when we
divide any integer by 7 will all be between 0 and 6:

     3            2
  ----         ----
7 | 23       7 | 15
    21           14
   ---         ----
     2            1



the "keep track how many times each number occurred" can be easily refined
by introducing 6 integer counter variables  one for each of the
possible values of the random values being generated, initialized to 0. The
appropriate counter is incremented each time that the generated random value
corresponds to the value that the counter is keeping track of. This can be done
using either if statements or a switch statement...


initialize 6 counters to 0; // NOTE: this is to be done before the for loop


The above can be refined to actual C++ code as:

int zero_count, one_count, two_count, three_count, four_count,five_count, six_co
unt;
zero_count = one_count = two_count = three_count = four_count = five_count = six
_count = 0;


Now  we refine "keep track of how many times each number was generated
by using a switch statement:


// keep track of number of 0s, 1s, 2s, 3s, ...
   switch (random_value)
   {
    case 0 : zero_count++;
             break;
    case 1 : one_count++;
             break;
    case 2 : two_count++;
             break;
```

```
    case 3 : three_count++;
             break;
    case 4:  four_count++;
             break;
    case 5:  five_count++;
             break;
    case 6:  six_count++;
             break;
    default: cout << "Something wrong happened as we should never get to this ca
se" << endl;
   }  // switch





Now, we refine the:

"Print the bar chart using 40 *s for the number which occurred the MOST
number of times, and all the other numbers will have their number of *s
scaled accordingly ; "



 find the largest value of all the 6 counters ;

 compute a "scale"  factor  which will be used to
  control for loops which print out the appropriate number
  of *s for every value of zero_count, one_count, etc.



Refine the "find the largest value of all the 6 counters " :


largest = zero_count;

if (one_count > largest)
  largest = one_count;

if (two_count > largest)
  largest = two_count;

if (three_count > largest)
  largest = three_count;

if (four_count > largest)
  largest = four_count;

if (five_count > largest)
  largest = five_count;

if (six_count > largest)
  largest = six_count;


cout << "largest = " << largest << endl; // simply as a sanity check to verify
```

```
Let’s explain the scale factor:

suppose the counts are as  follows ( with a total of 20 numbers being
generated):


0: count 3
1: count 5
2: count 4
3: count 1
4: count 2
5: count 4
6: count 2

We can readily see that the value 1 was occured the greatest number of times.
The value of largest will be equal to 5 in this example.

So, the bar chart for value 1 should have 40 *s in it.

Introduce scale_factor as:


scale_factor =  ( ( value_of_counter ) * 40  ) / largest


If we substitute the values of 5 for value_of_counter and 5 for largest into
the scale_factor expression we obtain:

scale_factor = ( 5 * 40 ) / 5
             =   200 / 5
             = 40 ;

next, we use a for loop to print out up to "scale_factor" *’s:

for(int i = 0 ; i <  scale_factor ; i ++)
  cout << "*" ;
cout << endl;



We can use a for loop for every value of counter computing a new scale_factor
for every counter value.


Every value will have it’s bar chart ’scaled’ appropriately. For example, the
value 3 was generated a total of 1 times:

scale_factor = ( 1 * 40 ) / 5
             = 40 / 5
             = 8

so it’s chart will have 8 *s printed  corresponding to 1/5 the number of stars
printed for value 1.


Putting all the final refinements into a complete C++ program gives the final
version:
```

```
// Author: Ted Obuchowicz
// Jan. 29, 2002
// example program illustrating use of
// random seed function


#include <iostream>
#include <string>
#include <stdlib.h>

using namespace std;



int main()
{
int how_many_numbers_you_want;

int seed_value;
int random_value;
int largest;
int zero_count, one_count, two_count, three_count, four_count,five_count, six_co
unt;

// initialize all the counters to 0;

zero_count = one_count = two_count = three_count = four_count = five_count = six
_count = 0;


cout << "How many numbers do you want to be generated? " ;
cin >> how_many_numbers_you_want;

cout << "Please enter a seed value " ;
cin >> seed_value;
srand(seed_value);
for(int i = 0 ; i <= how_many_numbers_you_want ; i++)
 {
   random_value = rand() % 7;
   cout << random_value << endl;  // print out 20 random numbers between 0 and 6
   // keep track of number of 0s, 1s, 2s, 3s, ...
   switch (random_value)
   {
    case 0 : zero_count++;
             break;
    case 1 : one_count++;
             break;
    case 2 : two_count++;
             break;
    case 3 : three_count++;
             break;
    case 4:  four_count++;
             break;
    case 5:  five_count++;
             break;
    case 6:  six_count++;
             break;
```

```
    default: cout << "Something wrong happened as we should never get to this ca
se" << endl;
   }  // switch
  } // for


cout << "0 count " << zero_count << endl;
cout << "1 count " << one_count << endl;
cout << "2 count " << two_count << endl;
cout << "3  count " << three_count << endl;
cout << "4 count " << four_count << endl;
cout << "5  count " << five_count << endl;
cout << "6  count " << six_count << endl;

// next we draw our 'sideways' bar chart

// first determine which counter is the largest... we will print 40 * for this
// counter and all the others will be scaled accordingly

largest = zero_count;

if (one_count > largest)
  largest = one_count;

if (two_count > largest)
  largest = two_count;

if (three_count > largest)
  largest = three_count;

if (four_count > largest)
  largest = four_count;

if (five_count > largest)
  largest = five_count;

if (six_count > largest)
  largest = six_count;


cout << "largest = " << largest << endl;

// print out the bar chart for the zero_count

int scale_factor = (zero_count * 40) / largest;

cout << "0. " ;
for(int i = 0 ; i <  scale_factor ; i ++)
  cout << "*" ;
cout << endl;

 scale_factor = (one_count * 40) / largest;

cout << "1. " ;
for(int i = 0 ; i <  scale_factor ; i ++)
  cout << "*" ;
cout << endl;

 scale_factor = (two_count * 40) / largest;
```

```
cout << "2. " ;
for(int i = 0 ; i <  scale_factor ; i ++)
  cout << "*" ;
cout << endl;

 scale_factor = (three_count * 40) / largest;

cout << "3. " ;
for(int i = 0 ; i <  scale_factor ; i ++)
  cout << "*" ;
cout << endl;

 scale_factor = (four_count * 40) / largest;

cout << "4. " ;
for(int i = 0 ; i <  scale_factor ; i ++)
  cout << "*" ;
cout << endl;

 scale_factor = (five_count * 40) / largest;

cout << "5. " ;
for(int i = 0 ; i <  scale_factor ; i ++)
  cout << "*" ;
cout << endl;

 scale_factor = (six_count * 40) / largest;

cout << "6. " ;
for(int i = 0 ; i <  scale_factor ; i ++)
  cout << "*" ;
cout << endl;


return 0;
}
```

```
Pretty Pascal's Triangle:
-------------------------



I remember seeing a question about generating Pascal's Triangle in a programming
 book way book in
either CEGEP or first year Con. U so I sat down and finally figured it out :

ted@deadflowers Programs 7:38pm >pretty_triangle3
How big do you want the pretty triangle to be? 20
                    *
                   ***
                  *****
                 *******
                *********
```

```
               ***********
              *************
             ***************
            *****************
           *******************
          *********************
         ***********************
        *************************
       ***************************
      *****************************
     *******************************
    *********************************
   ***********************************
  *************************************
 ***************************************


ted@deadflowers Programs 8:08pm >pretty_triangle3
How big do you want the pretty triangle to be? 10
          *
         ***
        *****
       *******
      *********
     ***********
    *************
   ***************
  *****************
 *******************
ted@deadflowers Programs 8:09pm >pretty_triangle3
How big do you want the pretty triangle to be? 2
  *
 ***

The solution uses simple nested for loops, an outer loop to control how many row
s the triangle will have.
Using the method of stepwise refinement we have:


First refinement:
----------------



Draw a pretty triangle like the one above. The number of rows is a user
supplied value.


Second Refinement:
----------------

If we look carefully at the figure and give little thought we realize two things
about this pretty triangle:

1) the first row has "size" number of leading blanks, where size is the user sup
plied value

2) the number of *s in each row gows by 2 each row. The first row has 1 * in con
tained in it.
```

```
If we think a little more about 1) we find out :

the number of leading blanks is decreased by one per each row
or in other words the # of leading blanks in row i = 30 - i ===> this is the CRU
X of the problem.

Third refinement:
-------------------

get the "size" of the triangle from the user.

for (all the rows in the triangle)
{
   print out a row of the triangle;
}



Fourth Refinement:
---------------

int size;
cin >> size;
int stars = 1 ;  // controls how many stars are printed out in eaxch row and we
                 // start off with 1 star in the first row

for(int i = 0 , i <  size ; i++ )  first row is i=0 , next row is i=1, etc.
{

  figure out how many leading blanks there are in this row ;

  print out a row containing  the proper number of "blanks" and  "star" number o
f *s;

  stars = star + 2 ; // the next row will have two more stars in it

}


Fifth Refinement:
----------------

int size;
cin >> size;
int stars = 1 ;  // controls how many stars are printed out in eaxch row and we
                 // start off with 1 star in the first row

int blanks;

for(int i = 0 , i <  size ; i++ )  first row is i=0 , next row is i=1, etc.
{
  blanks = size - i ;

// use a for loop to print out the required number of leading blanks in row i
// BEFORE we print out any *s on the same row

  for (int j = 0 ; j < blanks ; j++ )
  {
```

```
    cout << " " ;  // note we do not print out an endl since we want the *s to b
e printed on the same
                  // line as the blanks
  }

// NOW print out the required number of stars in each row
// the first * is printed AFTER the last blank on the given row
// using a for loop

  for (int k = 0 ; k < stars ; k++ )
  {
    cout << "*" ;
  }

  cout << endl ;  //  now we print out a new line so that the next row of the
                 //   triangle begins on a new line.. one line down from the
                 //   previous


 stars = stars + 2 ;  // the number of stars in each row "grows" by two

}
```

Hello,

I have been informed bu our AITS staff that the Arduino
development environment has been installed on our CentOS7 Linux
systems in the Hall building labs... I tested in in H913 and it
works fine.. it should be available from any Liux system in the
Hall labs.

A few points:

0. The Arduino program can be invoked by simply typing:

```
  arduino  &
```

from the Linux prompt. The main Arduino GUI widow will appear
after a few second.

Note: The full path is /encs/bin/arduino

1.Tools -> Select board as Arduino Pro Mini (it should work with Nano
as well, but I haven’t tested it.

2. for the serial port: Tools -> Port should alreadyh by selected
as /dev/ttyUSB0

a sa sanity check, after you plugged in the USB cable to the Nano board,
froim the Linux prompt issue the command:

```
ted@panther ~ 5:32pm >ls -al /dev/ttyUSB0
crw-rw-rw- 1 root dialout 188, 0 Sep 22 17:22 /dev/ttyUSB0
ted@panther ~ 5:32pm >
```

it is important that the file permssions be:

crw-rw-rw-

Thius should be the default case as oiur systme admins have set this up.

3. Here is the equivalent "Hello World" progrma for ARduino:

ted@panther hello_world 5:27pm >more hello_world.ino

```
void setup() {
  // put your setup code here, to run once:

Serial.begin(9600);
Serial.println("Hello World from Arduino" );
}
```

4. Open up a terminal window by selecting the "Serial Monitor"
icon on the top right hand portion of the Arduino GUI.
A Linux xterm will appear with the titlebar /dev/ttyUSB0.
This window will contain the output produced by the Arduino program
(called a 'sketch' in the Arduino vernacular).

5. Type the Hello World sketch inside the Arduino window which contains the
initial (empty) program. and select the arrow icon (=>).
This will save, compile, download and run the program.

ACtually, not certian if it saves.. so do File -> save as specify a filename

"SEIHCEM ROF ONIUDRA"

ZCIWOHCUBO DET : YB

(Hint: hold this page in front of a mirror,
or Google : mirror code)

What is an Arduino board?
=========================

The Arduino board is an example of a microcontroller board - literally a
"small" controller. It consists of a small microcontroller chip
(the Atmega 328). A microcontroller integrated circuit chip is slightly
different from a microprocessor chip in that a microcontroller contains

```
the processor, and memory and additional hardware ALL IN ONE tiny integrated
circuit chip.  A microprocessor contains on the central processing unit
and requires additional support chips ( memory chips , chips containing
hardware to support input and output, etc ).

Microcontrollers are found almost everywhere in modern electronic devices -
microwave ovens, refrigerators, electronic toys (think of toy drones ),
and automobiles. The typical modern car contains about 20 different
microcontrollers controlling the electronic fuel injection, climate
control, antilock braking, cruise control , etc.   Microcontroller
(together with their more robust versions called "programmable logic
controllers) are used in factory automation turning valves on and off
to control some sort of industrial manufactuing process to a high degree
of precision and automation.

Microcontrollers have limited computing "muscle" - they typically work
on data which is byte size (8 bits) at a time.  Modern microprocessors
found in desktop personal computers work with data in chunks of 64 bits
at a time.  Due to this so called "small word size", microcontrollers
are slower than microprocessors. A small analogy explains why:
Consider a make believe microcontroller called the "KEITHIAC".  Suppose
the electronic hardware inside KEITHIAC can only perform addition
with 8bit wide binary data.  This data is found inside the memory, and has
to be FETCHED into the addition hardware inside of KEITHIAC one byte
at a time. To add an integer data type consisting of two numbers of 4
bytes, KEITHIAC would have to perform:


      get the two bytes of the numbers from memory.
      add them together and store the result somewhere
      and remember if there was a carry generated from the addition

      get the next two bytes from memory and
      add them together with the previous carry,
      saving the resulting answer and resulting carry

      get the next two bytes , etc..

      get the last pair of bytes


pictorially, this can be represented by:

                 carry2     carry1    carry0

   number1     = byte3      byte2     byte1     byte0
 + number2     = byte3      byte2     byte1     byte0
  --------       ------------------------------------
  answer        ans_byte3 ans_byt2  ans_byte1  ans_byte0



The benefit of being small (in terms of word size) is that
microcontrollers are INEXPENSIVE.

The Arduino board consists of the ATmega microcontroller together
with some extra hardware which allows it to be programmed with the
ATmega machine code via a host PC's USB (universal serial bus) port ,
hardware to provide a +5 V power supply (ie.e a voltage regulator),
```

```
some indicator LEDs, all mounted on a pretty blue printed circuit
board with pins proviidng access to the "ports" of the microcontroller-
more on the subject of ports later.

The Arduino Development environment  allows a programmer to edit
Arduino source code(loosely based on C/C++ together with many "Arduino
libraries contianing many useful "Arduino" functions) in a text editor window
, compile the source code
into the ATmega 328 machine code and send the resulting machine code
via the USB cable into the microcontrollers on chip  FLASH memory (memory
whose contents is retained even when power is removed - very similar to
the type of memory found in the ubiquitous USB "memory sticks".


Memory
=====

The ATmega 328 microcontroller contains the following memory
(all within the integrated circuit chip)


(Jumping Jack) FLASH memory :

  32 Kilobytes = 32767 bytes of FLASH memory used to store
 the program machine code.  Note: some of this
 memory is used to store the Arduino "bootloader"
 a small program which is automatically run whenver the
 board is powered up which allows the board to be programmed
without any extra supporting hardware.



FLASH is NON-VOLATILE, meaning the contents of memory is NOT lost
when the Arduino board is powered off.

Incidentally, when you compile your Arduino program from within the
Arduino IDE, the program size (in terms of number of byte of machine
code) is displayed in the bottom portion of the window:

"Sketch uses 4,872 bytes (15%) of program storage space. Maximum is 30,720
 bytes."


SRAM
====

SRAM is Static Random Access Memory - a type of memory that is volatile,
its contents are destroyed when power is removed.  SRAM is used to
store a programs variables.  The Arduino IDE reports the total amout of
SRAM used by a sketch's ( Arduino programs are commonly referred to as
'sketches' ) variables. For example:

Global variables use 188 bytes (9%) of dynamic memory, leaving 1,860 bytes for l
ocal variables. Maximum is 2,048 bytes.

EEPROM
======

EEROM is a non-volatile type of memory (electrically eraseable programmable
```

read only memory) which can be used to store a program's read-only variables
(variables whose values once initialized do not change). This is useful to have
in case a program exhausts its SRAM memory.... most Arduino sketches are small
programs and do not need to use the EEPROM memory.

The Arduino "Hello World" program
==================================

The following is an example of the Hello World program for the
Arduino microcontroller. It simply displays the message :

"Hello World from Arduino on deadflowers"

in a special window (opened up from the Arduino GUI before the sketch
is downloaded to the Arduiono board) called the "Serial Monitor" .
The Serial Monitor window is used to display any output produced
by the Arduinio sketch and it can also be used to provide input from
the keyboard to the program running on the Arduino board (through the
serial USB cable connecting the Arduino board and the host PC).

```
// Ted Obuchowicz
// file: hello_world.ino
// Arduino sketches are saved with the filename extension of .ino
// derived from the last three letters in the word "Arduino"
// I suppose it gives it a cute sounding name, like "bambino", "neutrino",
// "keithino", "tedino" ...

void setup()
{

// put your setup code here, to run once:

Serial.begin(9600); // specify the baud rate as 9600 bits per second
                    // and setup the serial communications between the USB port
                    // of the host PC and ant Arduino board
Serial.println("Hello World from Arduino on deadflowers" );
}

void loop()
{

// put your main code here, to run repeatedly:

}
```

The sketch consists of two functions: setup() and loop().
The setup() function is run once at the the beginning of program
execution. Any code which is meant to be performed one time at the
start of a sketch should be placed within the setup() function.
This particular version of the setup() function simply establishes
serial communication (via the USB port) between a host PC and the
Arduino board. This is done with the :

```
Serial.begin(9600);
```

statement.

the 9600 refers to the baud rate (or bits per second) used during the
serial communication between the host PC and the Arduino board .
There is nothing "special" about the value 9600. If one were to consult
the Arduino reference at https://www.arduino.cc/en/Serial/Begin ,
one would find the complete description of Serial.begin() :

"Sets the data rate in bits per second (baud) for serial data transmission.
For communicating with the computer, use one of these rates: 300, 600, 1200,
2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, or 115200."

The function prototype for Serial.begin() is:

```
void begin(long speed)
```

in other words, function begin (from the Serial) Arduino library
receives an argument of type long which specifies the bit rate and returns
nothing.

NOTE: there is a second version of begin() which recieves an additional
argument which is used to confugre to so called "data bits, stop bits,
parity" but we won't concern ourselves too much with this as the default values
are good enough.

The higher the baud rate, the faster the serial communcations over the USB
cable will be. This means that Serial.println() statements will take
LESS time to complete.

HISTORICAL NOTE: Early computers used bit rates as slow as 300 bits per
second, approxilately 0.02 second to send one ASCII character, so to send
the string "KEITH RICHARDS" on a 300 baud modem would require 0.37 seconds..
My first modem which my brother bought for me back in 1987 was a 2400 baud
modem... The slow bit rates of early computers was a primary reason why the
developers of the UNIX operating systems (Linux is a descendent of UNIX) adopted
"short-hand" versions of command names instead of their completely spelled out
English versions such as :

```
'list'   ( which became simply 'ls' for directory listing)
'copy'   ( 'cp' )
 'cd'    (instead of changedir)
```

The next line:

```
Serial.println("Hello World from Arduino on deadflowers" );
```

is the Arduino equivalent of :

```
cout << "Hello World from Arduino on deadflowers" ) ;
```

to perform output to the Serial Monitor.

The loop() function, as its name implies is an infinite loop
and usually contains code which is meant to be performed
repeatedly over and over. More examples on how to use the
loop() function will be shown later.

As mentioned before, the Arduino programming language is loosely
based on the C and C++ programming languages, with many different
"ARduino" library functions provided within the programming environment.

As an aide in understanding the above two functions (and how setup()
is called once at the beginning, and the loop() function is run repeatedly
over and over, the following C++ style fo code is functionally equivalent
to the two Arduino functions:

```
void setup()
{
  // put your setup code here, to run once:

Serial.begin(9600); // specify the baud rate as 9600 bits per second
                    // and setup the serial communications between the USB port
                    // of the host PC and ant Arduino board
Serial.println("Hello World from Arduino on deadflowers" );
}



int main()
{
   setup() ; // make a call to the setup() function at the start of the program

   while(true)
   {
     loop() ;  // call the loop() function over and over forever ...
   }

   return 0;
}
```

For the people who wish to explore Arduino programming in greater detail,
the Arduino IDE supports the programming of the ATMEGA microcontroller
using real bona fide C language .. but this is rather advanced stuff so
we won’t go in any further details.

Hello world - version 2:

Here is an alternate version of the "Hello world" sketch:

```
// T. Obuchowicz
// file: hello_world_in_loop.ino

void setup()
{
  // put your setup code here, to run once:

Serial.begin(9600); // specify the baud rate as 9600 bits per second
                    // and setup the serial communications between the USB port
                    // of the host PC and ant Arduino board
```

```
}

void loop()
{
  // put your main code here, to run repeatedly:

 Serial.println("Hello World from Arduino on deadflowers" );

}
```

Note that in this version, we only setup the serial communications
from the setup() function and in the loop() function , we forever
display the message:

"Hello World from Arduino on deadflowers"

For those who are wondering what 'deadflowers' refers to, it is the
name I selected for my Linux desktop and it the title of a Rolling Stones
song (from the album "Sticky Fingers" 1971, side 2 song 4, 4 min. 03s)

Hello world - version 3 (also known as The Blinking LED "
==

Microcontrollers such as the Arduino board are meant to interact with the
physical world by controlling devices such as light emittind diodes, sensors,
motors and actuators. The first two "Hello world" programs did not interact
with any devices. In the microcontroller world, a "true" Hello world program
is one which interacts with the devices it is meant to control. The following
"Blinking LED" is such an example. Before showing the code , the concept of
input and output ports must be explained.

A microcontroller interacts with devices through input/output PORTS. A port
is literally a "doorway" in and out from the microcontroller to the outside worl
d.
Devices such a LEDs can be connectec to ports and made to turn on and off under
the
control of the microcontroller.

A port can either be setup to act as an input port or an output port.
An input port READS information which is present on the physical pin
to which to port is connected to, and an output port can be WRITTEN
with data.

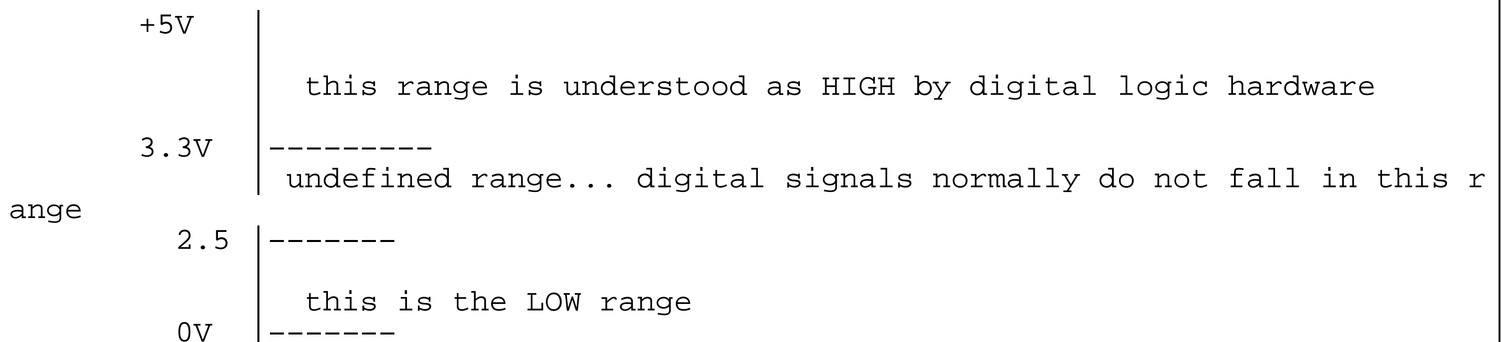
The Arduino port contains 14 DIGITAL ports, each port being 1 bit wide.
These are ports D0-D13. Of these, D0 and D1 are used for serial transmission,
so thety are not normally used as general purpose ports.

NOTE: when referring to ports in an Arduino sketch, we use the port
number such as 3 for DIO port 3. Note that the port number is usually distict
from the physical pin number of the board contianing the microcontroller.

DIGITAL PORTS

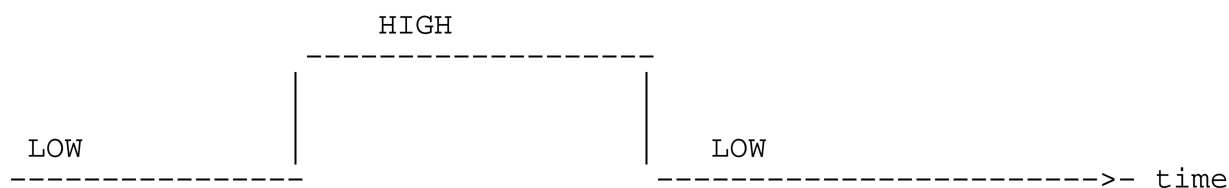
A digital signal is one in which there are only two distinct values
such as 0 or 1 (physically this means either 0 volts DC or +5 V DC, or in Ardu
ino talk LOW
or HIGH).

In reality, a RANGE of voltages represents a logic 1 (HIGH) and an
RANGE of voltages represents a LOW . These ranges are separated to
allow for NOISE IMMUNITY. Here is a picture of what we mean
by digital logic levels (i.e. Arduino HIGH and LOW) and
voltage:



Any DC voltage between 3.3 V and +5 V will be understood by the Arduino controll
er
as logic 1 (HIGH) and any voltage from 0 to 2.5 V will be intepretated as LOW.
Voltages in the so called undefined range are indeterminate, under normal operat
ion
a digital signal is NEVER in the undefined range...

The following is a timing diagram which shows a digital signal versus
time:



A digital port configured as an output port with the value of HIGH sent to the
port can be thought of as a low resistance

path to the +5 Volts of the power supply. If we write the value of HIGH to digital
output port, +5 V will appear on the physical pin - we can use this voltage to make a
light turn on.

Conversely, a digital port setup as output and we send to the port the digital value
LOW can be considered a wire to the ground of the power supply.

If we connect some type of circuit (i.e. a LED - a solid state lightbulb)
between the port pin and either ground or +5 V, the LED can be either turned on
or off under microcontroller control.

Pictorially:

```
                                                 wire      wire  --------------
 pin as output with value HIGH ====\-------|\|------|   resistor  | ---|
                               ====/       |/|      ---------------    |

                                +5V        LED                          |
                                                                      -----
                                                                       ---
                                                                        -

                                                                      ground
                                                                       0 V
```

In the somewhat crude ASCII text pseudo schematic diagram above, the LED will be
illuminated
since a LED lights up whenver +5V is applied to the the wire called the ANODE (shown as a
triangle symbol in the schematic) and ground applied to the other wire (called the
CATHODE).
If we write LOW to the port, the LED will be off. The rectangle labelled resistor is
there to limit the current through the LED to prevent it from burning out due
to excessive current (typical value of a current limiting resistor is 330 ohms).
This is known as an "active-high" configuration.

We can also make a LED turn on, if we wire it "backwards" and make the port output a LOW:

```
                                            wire      wire   --------------
port pin as output with value LOW ====>---------|/|------|   resistor  | ----- to
  +5V
                             0V(gnd)            |\|          --------------


                                                LED
```

In the so called "active-LOW" configuration, the LED is on when the port is outp
utting
LOW (0 volts or ground) and will be turned OFF when the port is outputting HIG
H
since a LED is only lit when a VOLTAGE DIFFERENCE appears across its two wires.

This is enough background on how digital output ports and LEDS work for this cou
rse,
if you want to learn more take MECH 368 Electronics for Mechanical Engineers.

Here is the blinking LED code:

```
// Ted Obuchowicz
// file blinking_LED.ino
// SEpt. 27, 2016

const int ledPin =  13; // pin number of LED on the Arduino board...
                        // declare it in the so called global
                       // space so that it is "visible' in the
                       // functions setup() and loop()

void setup()
{
// put your setup code here, to run once:

Serial.begin(9600);
Serial.println("The LED connected to DIO pin 3 will blink with aperiod of 0.5 se
c" );

// set the digital pin as output:
pinMode(ledPin, OUTPUT);

}

void loop()
{

// put your main code here, to run repeatedly:

digitalWrite(ledPin, HIGH) ; // turn the LED ON
delay(500); // and dealy for 500 milliseconds = 0.5 s
digitalWrite(ledPin, LOW) ; // turn the LED OFF
delay(500);  // wait for 0.5 seconds  and go back and loop all over

}
```

The code begins with a call to setup() which establishes serial communication
and configures digital pin 13 as OUTPUT using the Arduino function:

```
void pinMode(int pin_number, value)
```

which receives an integer argument corresponding to the port number we wish
to configure and a value of either HIGH or LOW - these are constants defined wit

```
hin
the Arduino environment of 1 and 0 respectively.

The loop() function is where it all happens:

we write HIGH to the port with the digitalWrite(ledPin, HIGH) function.
we call a function to delay 500 milliseconds
we write LOW to the port with the digitalWrite(ledPin, LOW) function
we dealy again for 500 ms.
go back and repeat this loop forever.


The Arduino board already contains a LED and resitor connected (in the active HI
GH
configuration)  wired up to digital port 13.  Compile and download this code
to the Arduino board and the LED will blink with a period of 0.5 seconds. Experi
ment
with difference delay values.

The prototype of the Arduino functions are:


void delay(int number_of_milliseconds_to_delay)

This function simply kills time for the specified integer number of milliseconds
:

delay(1) ; // will kill time for 1 ms
delay(25) ; // delay for 23 ms

void digitalWrite(int pinnumber , value)  ; // value can be HIGH or LOW




DIGITAL input PORTS:
===================

If a digital port is configured for input, it can read a digital value (with +5
V
being read as 1 (or HIGH using Arduino language) or 0 ( Arduino LOW). A digital
port
configured as input represents a very high resitance to circuit ground. This mea
ns
that very little current will be drawn away from the circuit which is connected
to it.  This is important as if the port did not present this so called "high im
pedance"
it would conduct some current ( we use the term "sink" some current) awat from t
he circuit
and this may cause the voltage at the port pin to drop from +5V to some lower vo
ltage
which may be misinterpreted as a logic 0 (LOW) value.    Enough talk of high imp
edance
input ports, lets read a digital signal through an input port:



// Ted Obuchowicz
// file digital_INPUT.ino
```

```
// Sept. 28, 2016

const int DIO_3 =  3  ;  // digital io pin #3 (physical pin 6 on the nano)
                        // declare it in the so called global
                       // space so that it is "visible' in the
                       // functions setup() and loop()


int input_value ;  // a variable to hold the return value of digitalRead

void setup()
{
// put your setup code here, to run once:

Serial.begin(9600);
Serial.println("This program repeatedly reads the value on D3" );

// set the digital pin as output:
pinMode(DIO_3, INPUT );

}

void loop()
{

// put your main code here, to run repeatedly:

input_value = digitalRead(DIO_3) ; // read the value of the DIO3 pin
Serial.println(input_value); // print it out as either 0 (for a LOW )
                             // 1 (for a HIGH = +5 volts

}
```

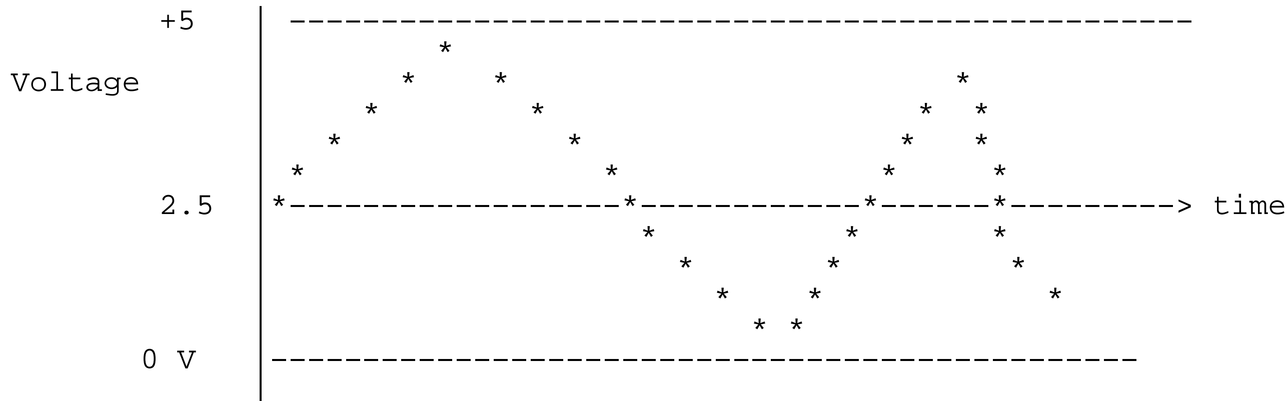
The setup functions intials serial communication and configure digital port
D3 as INPUT. In the loop() function, we READ the value on this input port
with the digitalRead(int pin_number) function and its return value is assigned
to the integer variable input_value which is then printed to the Serial Monitor.
The loop repeats forever. If we attach a wire to +5V of the Arduino board to di
gital pin 3, we will
read a HIGH value on the input port (which will be displayed in the Serial Monit
or
window as integer value 1 . If we connect pin 3 to GROUND, we will read a LOW
(displayed
as intger value 0).

ARDUINO EXPERIMENT FOR THE CURIOUS: what value is read if no wire is connected
to the port? This is termed a "floating "
input.

ANALOG SIGNALS
==============

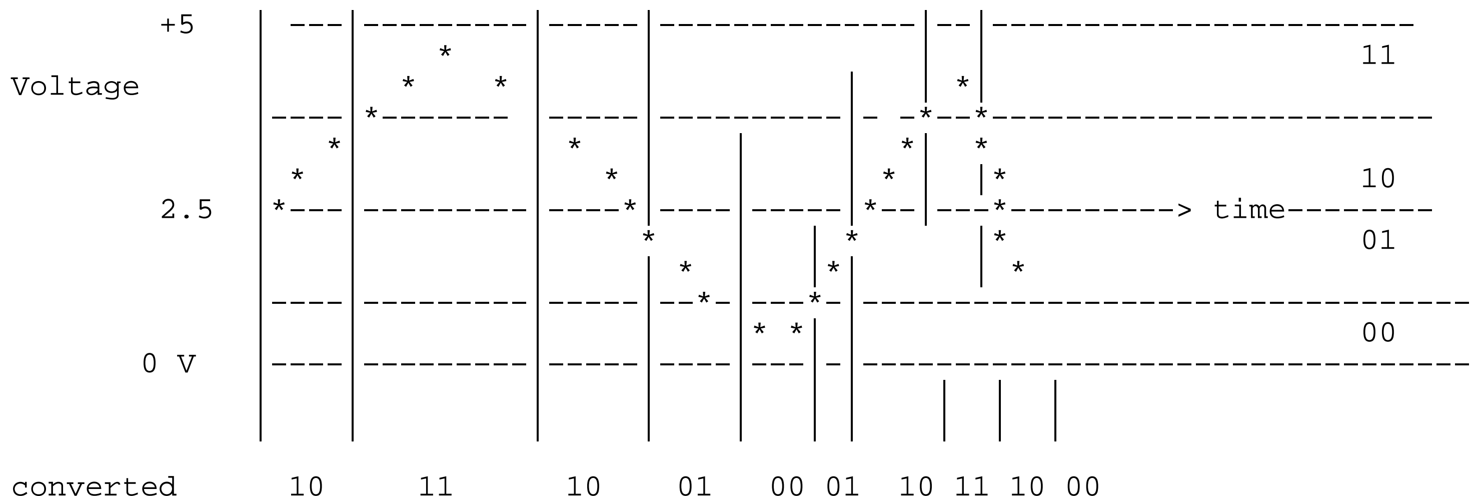
The real world is an analog one. Many electronic devices produce output
voltages which are CONTINUOUS over soem range of values such as 0V and +5 V.
Examples are photoresisors, electric guitar pickups, a function generating produ

ce
a sine wave output, etc. Here is a crude illustration of an analog signal:



Connect the * with a pencil and it will be a true continuos analog signal..
NOTE: A pencil is something people used to write with before YouTube came along
and ruined everything with videos....

Computers are digital, thet work with only 0s and 1s (HIGHs and LOWs). We need to
convert an analog signal into digital values (ie. numbers). This process is known'
as analog to digital conversion (ADC) and the ARduino contains 6 of these ADCs
connected to the its 8 ANALOG INPUTS (lablled A0, A1, A2, A3,A4, A5, A6, A7).
Suppose we use only 1 bit to perform the conversion. If we very quickly take a
sampel of the input signal and compatre it to the midway voltage of 2.5V, and if the
measrued voltage is LESS than 2.5, we represent it with the digital number 0, if the
signal is greater than or equal to 2.5, we convert it as a 1. Now suppose, we have
two bits to store the converted value (as a two bit binary number of 00, 01, 10, and 11).
Now we have 4 distinct voltage levels to compare to 0 , (1/4)* 5V , (1/2) * 5V,
(3/4)*5 and 5



```
value
as a 2bit
binary number


with more bits, we "chop" up the continous voltage into dicrete chunks, the chun
ks
are smaller if we have more bits:

3 bits : 8 chunks from 000, 001, 010, 011, 100, 101, 110, 111
4 bits : 16 chunks from 0000, 0001, etc,.
10 bits : 1024 chunks from 0 to 1023 (expressed in decimal, believe me you dont
want
           to read 1024 10bit wide binary unsigned numbers as in:


             0000000000
             0000000001  = ( 1)
             0000000010  = ( 2)
             0000000011  = ( 3)
             0000000100  = ( 4)

              etc.

             1111111110 = 1022
             1111111111 = 1023



The analog to digital converters on the Arduino are 10 bit converters. They conv
ert
an analog voltage into a 10 bit binary number giving a range of numbers
from 0 to 1023. The RESOLUTION is 5 V / 1023 = 0.004887 V = 4.8 millivolts.
This means that two voltages separated by less than 4.8 mV will convert to the s
ame
10 bit number.

Another aspect to consider when converting analog inputs to digital numbers
is the SPEED of conversion. Obviously, to accurately convert a signal, the conve
rsion
hardware has to be fast (faster than the signal is changinf) and we have to take
 a sufficient
number of sample to accurately convert the signal. This is heavy duty theory kno
wn as the
"sampling theorem" and is covered in great details in signal processing courses.
 The Arduiono
ADCs take approx. 100 microseconds to convert and are fast enough for most real
world "slow"
signals such as audio signals, etc.


Analog input:
============

This example uses the :


int analogRead(int pin_number)
// input argument is an int specifying which Arduino analog input to read from
```

```
// returns an integer value between 0 and 1023 , with 0 meaning 0V and 1023
// meaning full +5V.



function.



Here is the code which makes use of the light sensor (photoresistor)
module to provide a changing voltage between 0 and +5V depending on the
amount of light shining on it ( Rolling Stones reference - "Shine a Light"
from the album Exile on Main Street, 1972 )


// Ted Obuchowicz
// file: analog_READ.ino
// date: Sep 30 2016

int v ;// an integer to hold the converted analog voltage signal
       // between 0V and +5V (the output of the photoresitor )
       // connected to analog input pin 3 (physical pin 23 )

void setup() {
  // put your setup code here, to run once:
Serial.begin(9600);
Serial.println("World") ;

// note : analog input pins do not have to be setup, they
// are already preconfigured as "inputs" in hardware
// no no special software configuration is required for
// the analog input pins..
}

void loop() {
  // put your main code here, to run repeatedly:

v=analogRead(3); //Connect some analog source of the board to the A3 port
delay(1) ; // reading an analog input takes about 100 microseconds
           // so delay 1 ms before taking the next reading
           // can actually delay less but this is just an example and
           // I know my analog input signal is changing slowly..
Serial.println(v);  // print out the converted value as a number
                    // between 0 and 1023 , 0 = 0V and 1023 representing +5V


}


// for the light sensor module
// G = ground
// V = +5 V
// S = analog signal output to be connected to any analog input
//     pin of the Arduino.. i.e. A3
//  note: the slide slwitch on the sensro should be in the A (Analog)
//  position so that it ouitputs an analog voltage between 0 and +5 V
//  depending on the amoutn of light shining on the photoresistor.
```

The code is quite self explanatory:

we setup dor serial communicatiosn since we will be using Serial.println()
to perform output and print a little message (as a sanity check to make sure we
got into the setup() function) , then enter the loop() function where we:

* read the analog pin 3 using the Arduino function analogRead(3)
  and assign the return value to integer value v (v for "voltage" , a good name
  for a variable.. not as good as "keith" though...)

* delay for a whopping 1 ms (which is way overkill since conversion
* takes approx 100 ms...) but 1 ms is the shortest delay we can do
* using the delay function..

* print out the converted value

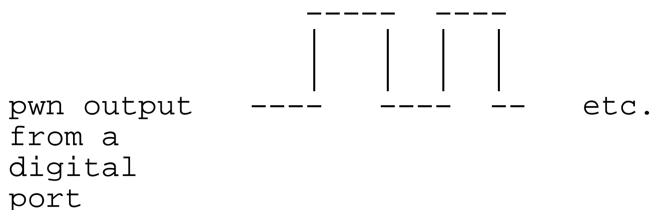
* go back to the top of loop() and again read the
  (new) voltage which is present on the pin.

You don’t even need to wire up the
light sensor to use this code. Simply use a wire to connect analog pin
3 to +5V and this will be read as 1023, connect the port to ground and the
number printed will be 0.

ANALOG OUTPUT
=============

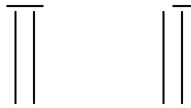
The Arduino microcontroller contains 6 so called "pulse-width-modulators"
which are hardware devices which can rapidly turn on and off a digital
port creating a "pseudo-analog" output whcih when connected to a device
such as a LED or motor will either dim or brighten the LED or speed up or
slow down the speed at which the motor is rotating,

The idea behind PWM is shown in the figure below:



the fraction of time which the output signal is HIGH as compared to LOW is
known as the "duty cycle". In the crude ASCII text drawing the duty cycle is
about 75%.

Here is a 10% duty cycle PWM output:



```
------ ------------  etc.

90% of a single period, the signal is LOW, and for 10% it is HIGH.

Chaging the duty cycle changes the AVERAGE voltage presented to whatever
device is connected to the pin and the device will behave accordingly
(speed up or slow down in the case of a DC motor, dim or brighten in the
case of a LED )


Here is the code for analog output. It makes use
of the function :

void analogWrite(int pin_number byte duty_cycle)

which takes two input arguments: an integer
represensting which pin to use and a byte
( an integer number between 0 and 255 which represents
the duty cycle with 0 = full OFF or 0% duty cycle and
255 = FULL ON = 100% duty cycle. )

NOTE: In the Arduino version of C language, there is
a byte data type which is used to represent unsigned
integers between 0 and 255.



// Ted Obuchowicz
// file: analog_WRITE.ino


int pwm_output = 3 ; // digital pin 3 used for analog writing
                     // of pulse width modulated signal (physical pin 6)

void setup()
{
  // put your setup code here, to run once:
Serial.begin(9600) ;  // to setup serial comminucations via USB port


// Note; no special pin configuration needed for analog output
// since the special analogWrite() function handles all the low
// level details..

}

void loop()
{
  // put your main code here, to run repeatedly:

// a little for loop which changes the duty cycle,
// in increments of 31

for(byte duty_cycle  = 0 ; duty_cycle <= 255 ; duty_cycle = duty_cycle + 31 )
```

```
{
  Serial.print("Duty cycle = " );
  Serial.println(duty_cycle);
  analogWrite(pwm_output,  duty_cycle) ;
  delay(2000)  ; // light the LED for 2 seconds incrementing over
                // 8  duty cycles from full OFF to full ON

}

}

// NOTE: analogWrite(int pin byte duty_cycle)
// only DIO pins 3 ,5,6,9,10,11 of the Arduino board can be specified as the pin
 number
//  the duty cycle is a byte in the range from 0 to 255
```

As noted in the last comment, only a select few of the 14 DIO pins
can be used as analog output.

PROVIDING KEYBOARD INPUT TO AN ARDUINO SKETCH WHILE
IT IS RUNNING ON THE BOARD

A natural question to ask yourselves is (at least I asked myself
this question) is:

"How can I provide some keyboard input to a sketch while it it
 running?"

Arduino sketches are meant to control things such as LEDs and motors,
etc. as such they don’t very often read keyboard input very often.
There are occasions when it is necessary and the Arduino environment
provides a mechanism to do so as long as the board is connected
to a host PC with the USB cable. The library "Serial" which we’ve
used for things such as Serial.begin(9600) and Serial.println()
has some functions used for keyboard input (Serial.read()).

Here is a sample sketch which reads a single ASCII character from
the Serial Monitor window and dispalys it on the Serial monitor
window:

```
// T. Obuchowicz
// file: serial_READ.ino

char incomingChar = 0;   // for incoming serial data
char letter = ’T’ ;      // I added this for testing purposes.. not really neede
d

void setup()
{
        Serial.begin(9600);     // opens serial port, sets data rate to 9600 bps
        Serial.println(" Position the cursor in the top part of the ");
        Serial.println(" Serial Monitor window, type any key and press the Send
```

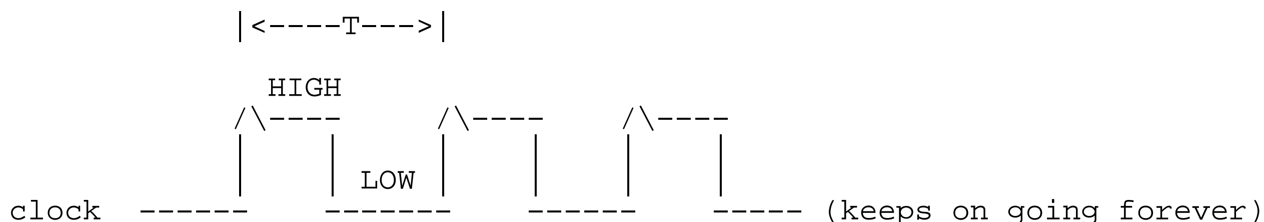
```
" ) ;
        Serial.println("button the send the ASCII code for the character to the
Arduino board");
        Serial.println(letter);


// ACHTUNG BABY ( A U2 musical reference for a change )
// READ THE ABOVE COMMENTS AS THEY EXPLAIN WHERE YOU TYPE IN THE CHARACTERS
// FROM ... TOOK ME A WHILE TO FIGURE IT OUT AS THE ARDUINO GUI IS NOT
// THAT INTUITIVE...


}

void loop()
{


        if (Serial.available() > 0)  // check if there is ASCII chars in the
                                     // buffer, everytime you send a char
                                     // by pressing SEND, it is sent to a
                                     // small areas in the Arduino's memory
                                    //   known as a buffer , a counter is
                                    // also incremented by oen with each
                                    // char which is sent
                                    // the function Serial.available() reads thi
s
                                    // counter value and if greater than 0 it me
ans
                                    // that there are still unread chars in the
buffer.
        {
                // read the incoming byte:
                incomingChar = Serial.read();  // read a char from the buffer
                                               // and assign it to the variable
                                               // we use the Serial.read(void) f
unction
                                               // to do this.. note: every read
                                               // decrements that special counte
r by 1

                // say what you got:
                Serial.print("I received: ");
                Serial.print(incomingChar, DEC); // print the ASCII value as a d
ecimal number
                Serial.print( "    " ) ;
                Serial.print(incomingChar); // will print it out as an ASCII cha
racter


        }
}




Read the comments in the loop() function as they explain what the
Serial.available() and Serial.read() functions do.
```

```
You may be wondering : "Great, but how do I read an integer or a float??"

Here’s the answer: Arduino has special functions to read an integer sent from th
e
keyboard to the Serial Monitor to the microcontroller and another function to
read a float .

Here’s the code, by now you should understand what its doing
by READING it.. better yet, manually type it in with a
text editor and run it... yes I said TYPE it in, not
simply "cut and paste". The mere act of physically
typing it in gives the brain time to absorb and understand
each line of code... if you do resort to using some
mouse magic to cut and paste.. at least take the time
to read over each line of code...



// T. Obuchowicz
// file: read_INTEGER.ino

int number ;  // for incoming integer serial data
float a_float ;

void setup()
{
        Serial.begin(9600);     // opens serial port, sets data rate to 9600 bps
        Serial.println(" Position the cursor in the top part of the ");
        Serial.println(" Serial Monitor window, enter an integer  and a float  a
nd press the Send " ) ;
        Serial.println("button the send the data   to the Arduino board");
}

void loop()
{


        if (Serial.available() > 0)
        {
                // read the incoming byte:
                number = Serial.parseInt();
                a_float = Serial.parseFloat();
                // say what you got:
                Serial.print("I received: ");
                Serial.print(number);
                Serial.print(" ") ;
                Serial.println(a_float);


        }
}


NOTE:  You would type in number in the Serial Monitor input portion in the
following manner

56       -5.678
34  123.56
```

parseInt() parses the input buffer and assemble an integer number from whatever
literal are present until it reaches a char which is not a valid literal for
an integer, it then figures out what integer number these diverse ASCII chars re
presents.
parsefloatt() then it skips over any blanks until it reaches a char which is a l
egal literal for a
float and keeps reading these literal until it reaches a char which is not a leg
al literal
for a float and it then it converts all these ASCII chars such as -5.678 into t
he IEEE 754
represenstation of a flaot with value of -5.678..... this is why we learned the
different
types of literals way back in the first or second class....

Keeping track of time with the Arduino
(or "Ticking away the moments that make up a dull day" - A Pink Floyd reference)
==

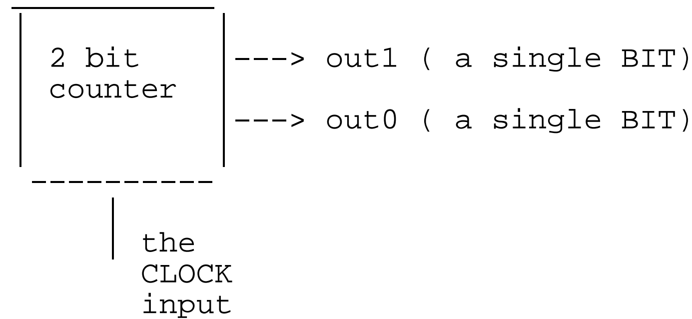
The microcontroller on the Arduino board contains several so-called HARDWARE
TIMER/COUNTERS used are used to keep track of how much time has elapsed since th
e
microcontroller was turned on (or reset). All processors use a special digital
input
called the CLOCK input - for the ATmega 328 microcontroller on the Arduino board
the FREQUENCY of this clock is 16 megahertz (Mhz). A timing diagram of a
clock signal looks like:



The time between successive RISING clock edges represented by the up
arrow :

is referred to as the CLOCK PERIOD , T = 1 / (clock frequency).

Certain digital logic hardware respond to rising clock edges. A COUNTER
is such a piece of hardware. A counter is a digital logic circuit which
has a CLOCK input and special extra hardware called FLIP-FLOPS (a flip-flop
you can think of a type of memory which can hold a BIT). A 2bit counter contains
two flip-flops and can store a 2bit binary number. The value of the 2 bit binar
y
number is INCREMENTED with every occurrance of a rising clock edge.




```
This 2bit counter would count in the following manner :

out1 out0
========

0     0             0     1       1   0     1    1   then it wraps around back t
o

0     0             0     1       1   0     1    1    and wraps around back to 0
0


The more bits in the counter, the larger number the counter can count up to
before it wraps around back to all 0s.  Since the clock period is constant,
by keeping track of the value of the counter, we know how many clock periods
have elapsed and it is straightforward to convert from clock periods to unit
of time such as milliseconds or microseconds.

The Arduino programming environment provides two special functions which use the
hardware counters :

unsigned long millis(void) ; // returns as an unsigned long the elapsed time
                             // in MILLISECONDS

unsigned long micros(void) ; // returns as an unsigned long the elapsed time
                             // in MICROSECONDS
                             // 1 MICROSECOND == 1000 MILLISECONDS


Here is an Arduino sketch which uses these timing related function to
measure the execution time of several for loops:



// T. Obuchowicz
// file: timer_millis.ino

unsigned long start_time ;
unsigned long end_time ;




void setup() {
  // put your setup code here, to run once:
```

```
Serial.begin(9600);  // estalish serial port for communications

unsigned long diff ;


//Serial.println("Hello");

start_time =  micros() ;  // start timing
                         // to determine
                         // time in ms needed
                         // to perform loop

for(int i = 0 ; i < 100 ; i++ )
{
  Serial.println(i) ;
}
end_time =  micros() ;

diff = end_time - start_time ;
Serial.print("The loop of 100 iterations took " );
Serial.println(diff);


start_time = micros() ;
for(int i = 0 ; i < 200; i++ )
{
  Serial.println(i) ;
}
end_time = micros() ;

diff = end_time - start_time ;
Serial.print("The loop of 200 iterations took " );
Serial.println(diff);

start_time = micros() ;
for(int i = 0 ; i < 400; i++ )
{
  Serial.println(i) ;
}
end_time = micros() ;
diff = end_time - start_time ;
Serial.print("The loop of 400 iterations took " );
Serial.println(diff);

Serial.println("Goodbye" ) ;




}

void loop() {
  // put your main code here, to run repeatedly:
  // nothing to do here , it's all done in the setup()
  // function for this example..
}
```

```
The output is:



0
1
2
3
4
5
...
The loop of 100 iterations took 338080

0
1
2
3
4
...
197
198
199
The loop of 200 iterations took 925604


0
1
2
3
...
397
398
399
The loop of 400 iterations took 1965604

Goodbye
```

We can note that the time to execute the loop is approximately LINEAR with respect to the # of loop iterations.

Experiment with the above code to see the effect of using the millis() function instead of micros(). You should also keep in mind that both the millis() and micros() function will WRAP AROUND back to 0 after a certain amount of time. Brain Evans, the author of the book entitled "Beginning Arduino Programming" states on pages 98 and 99:

"because this function (millis) returns a value in an unsinged long int, it will overlfow in about 50 days"

"micros() will overflow, or reset back to 0, every 70 minutes or so"

He goes on to state:

"It only has a resolution of 4 microseconds, meaning that every value returned by the micros() will be a multiple of 4."

```
Let’s do the math to understand why these timing functions wrap around.
An unsigned long in the Arduiono is 4 bytes (32 bits) with a range of:


0 to 4 294 967 295

Every 1 ms, millis() will increment by 1.  It will take 4 294 967 296 * 0.001s
= 4 294 967.296 seconds to wrap around.  Omitting the fractional part:

# days to wrap around = 4294967 seconds / ( 86400 sec /day) = 49,71 days
(close enough to 50 days).

You do the math and verify the value of 70 minutes for the time for
micros() to wrap around back to 0. (My math resulted in 71.56 minutes..)


Arduino Basic Data Types
=========================


The preceding examples have introduced some of the Arduino data
types, some of which are the same as the C++ data types we have learned,
and some are new.  Let’s tabulate the Arduino built-in data types:

NAME                          size        range
-----------------------------------------------
boolean                     1 bit            true or false ( in C++ we call it b
ool)

byte                        8 bits            0 to 255  (unsigned only)

char                        8 bits            ASCII character set
                                              (and can also be used for numbers
                                               from -128 to + 127)

int                         2 bytes            -32768 to +32767  (same as C++ sh
ort )

long                        4 bytes             -2 147 483 648 to +2 147 483 647

float                       4 byte               IEEE754 32 bit float range
                                                  we discussed in class


NOTE: Arduino does not support the double or long double floating point data
types.

NOTE: The modifier unsigned can be applied to int , long to extend the
      range if only positive numbers will be used as a value for a variable
      of this data type.






One more example:
===============
```

```
This example uses the ultrasound sensor to measure distance.
Read the data sheet for the sensor before reading this example.


// Ted Obuchowicz
// file: ultasound_Sensor_without_pulseIN.ino
// Oct. 3, 2016

int trig = 9  ;  //use DIO9 as an output pin to send the TRIG
                 // signal to the ultrasonic sensor

int echo = 8 ; // use DIO8 as an input pin to reasd the ECHO
              // signal sent by the sensor

unsigned long  duration ; // used to hold the duration in microseconds
unsigned long start_time, end_time ;
float distance ; // the distance the sensor is from the reflecting object

void setup()
{

  Serial.begin(9600);
  pinMode(trig, OUTPUT);
  pinMode(echo, INPUT);
  digitalWrite(trig, LOW); // make sure TRIG signal is intially LOW
  delay(2) ;   // give enough time for TRIG to be LOW
  Serial.println("Ready to measure the distance using the ultrasonic sensor ");


}


void loop()
{
 digitalWrite(trig, HIGH); // send out the HIGH to tell the sensor to start
 delayMicroseconds(11) ; // data sheets specifies min trig widt of 10 micro
                         // so give 1 more 1micro just to make sure

 digitalWrite(trig, LOW);  // bring TRIG signal low and wait for the
                           // sensor to send back the ECHO signal and
                           // this version does not use pulseIn
                          //  we simply loop as long as echo is LOW
                          // once it goes HIGH we start timing and
                          // wait again as long as it is still HIGH
                          // and when it goes LOW we stop timing..
                          // no need for fancy pants pulseIn function
                          // gives nice examples of what is know as
                          // "busy waiting loops"

//         HIGH
//        ------
//        |    |
//        |    |  LOW
//   -----     ---------  TRIG signal 10 microseconds min duration pulse
//        |<-->|          output on DIO 8
//         10 microseconds

//
```

```
//   after we output the above pulse, we WAIT until TRIG goes HIGH
//   as per the pretty timing diagram below:

//
//                           HIGH
//                        ---------------------
//                       |                     |
//                       |                     |  LOW
//    ----- -----------                         -------------  ECHO signal
//                       |  <- duration  --->|
//
//  the width of the ECHO duration is proportional to the
//  distance , the further away the sensor is , the longer
//   the duration
//   can use the formula in the lab handout to convert from pulse
//   duration to distance in cm
//   or basically the formula is: distance = speed of sound * duration
//    the speed of sound is 343,4 m/s or 0,0343 cm/microsecond (at a warm
//    20 deg. C ..
//   distance = 0,0343 * duration
//   we divide for 2 because the duration is the time spent from ultrasonic sign
al
//   to go from the sensor, bounce off the sound reflecting object and travel
//   back to the sensor.. in other words the duration times the 2-way trip,
//   we want the 1 way trip, so we divide by 2.
//   distance = 0.0343 * duration/2 = (0.0343/2) * duration
//   distance = ( 1/58.31) * duration
//   distance (to nearest whole cm) is approx = duration/58


while(digitalRead(echo) != HIGH)
{
   // busy waiting while ECHO is still low..
  // wait until sensor responds with ECHO = HIGH}
}

// start timing the duration of the ECHO pulse
start_time = micros() ;


while(digitalRead(echo) != LOW)
{
  // busy waiting while the sensor ECHO is still HIGH
  // Pink Floyd lyric : "Mother , did it need to be so HIGH" );
  // wait for the sensor to lower the ECHO signal back to 0
}

// now get the current time and the difference between the
// two times is the duration ..

end_time = micros();


duration = end_time - start_time;

Serial.print("Duration = ");
Serial.println(duration);

// now use the provided formula to convert from duration in
```

```
//microseconds top distance in cm

distance = duration/58;

Serial.print("Distance = ");
Serial.println(distance);


delay(500) ; // wait 0.5 seconds before sendign out a new
             // TRIF signal to commence a new reading.. this
             // gives the person enough time to move the
             // sensor to a new postion for testing purposes...



}










Contolling a DC servo motor with the  <Servo.h> library:
==========================================================




// controls a single servo motor attached to a PWM analog output
// DIO pins 3 ,5,6,9,10,11 are the pins with PWM output capability
// connect red wire of motor to +5 V of ARduino board, black wire to GND and
// yellow wire to the DIO pin used for PWM output (DIO 9 in this example )
// use a separate power supply if connecting two or more motors as the
// arduino cannot provide sufficient current for more than one motor...


#include <Servo.h> // for the Servo library

Servo mymotor; // declare a variable of type class "Servo"

void setup() {
  // put your setup code here, to run once:
Serial.begin(9600) ;  // to setup serial comminucations via USB port
Serial.println("The servo will be moved to the midpoint position");

mymotor.attach(9) ; // attach the servo called 'myservo' to dio pin 9
                    // if using more than one servo motor, then "attach"
                    // each motor to a different DIO pin .

mymotor.write(90) ; // write(int angle_in_degrees )
                    // postion the motor at 90 degrees intially at startup

}

void loop()
{
  // put your main code here, to run repeatedly:
```

```
int pos ; // angle  in degrees

// move motor forwards

for(pos = 0 ; pos <= 180 ; pos = pos + 5 )
{
   mymotor.write(pos); // NOTE: do not control a servo motor
                       // directly using analogWrite() (i.e. a PWM
                       // signal ) since the motor may be damaged
                       // one should use the Servo library function
                       // write to control a servo since this function
                       // generates the required pulse durations the
                       // servo requires.
   delay(500) ; // arbitrarily delay 0.5 seconds before moving the servo
                // the the next angle position.
}


// move motor backwards
for(pos = 180 ; pos >= 0 ; pos = pos -5 )
{
 mymotor.write(pos);
 delay(500) ;
}


}
```