MECH215
Lecture Notes 0

Getting Started With C++

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

```
// Author: Ted Obuchowicz & John Peach
// hello.cpp
// hello world program

#include <iostream>
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:

```
#>g++ -o hello hello.cpp
```

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

```
#>./hello
Hello World
```

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 & John Peach
// Jan. 7, 2001
// hello world program
```

The next two lines:

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

are known as COMPILER DIRECTIVES. They are instructions telling to compiler where 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 function 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 terminates. Historically, a return value of 0 has been used to indicate success. Any other value is indicates an error condition that is specific to that program.

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 & John Peach
// print_message.cpp
// Example program illustrating use of function calls and not using
// the “using namespace std” directive.

#include <iostream>
#include <string>

void print_message(std::string 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

  std::cout << s << std::endl;

}

int main() {

  print_message("Hello MECH 215");
  print_message("Goodbye MECH 215");
  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()` function 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 delimiters are comments */

/*  ******************************************** */
/*  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 upper 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 .

Do not 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

Literals 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 & John Peach
// literals.cpp
// 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:

```
#>./literals
24 24 24  24
```

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 specifying literal integer constants by appending L (or l), and U (or u) respectively to the number:

5U, 5u
5L, 5l,
5Ul, 5uL

Floating-point Literals

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

Character Literals

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 & John Peach
// bell.cpp
// 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 Literals

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' denotes the end of the c-string

When printed the above string would print:
```
a	b
```

(an a followed by a number of blank spaces as specified by the horizontal tab setting of your terminal, a b, followed 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 built in integer 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

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 integer 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 number.

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 position designates a positive number, a 1 represents a negative number. You do not 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 positive 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 & John Peach
// sizeof.cpp
// 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>
using namespace std;

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, 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:

```
#>./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 *
```

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

$0.\text{mantissa} \times 2^{\text{exponent}}$

The value of the sign bit determines whether the number is positive or negative.
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, programmers had to use to PREPROCESSOR DIRECTIVE #define to create a symbolic named constant:

```
// Author: Ted Obuchowicz & John Peach
// define.cpp
// Example program illustrating use of C-style #define methods 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:

```
#>./define
area of circle with radius 2 is 12.5664
```

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 & John Peach
// const.cpp
// example program illustrating use of the 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 methods 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/Pointers

C++ allows for the declaration of reference variables. Essentially, reference variables allow a variable to have more than one name associated with it. They are often called pointers as they contain the memory address that points to the value.

```
// Author: Ted Obuchowicz & John Peach
// pointer.cpp
// Example program illustrating use of reference variables

#include <iostream>
#include <string>

using namespace std;

int main() {

    int value ;                   // some value
    int & pointer1Value = value; // Create a pointer to value
    int & pointer2Value = value; // Create a second pointer to value
    value = 5;                    // initialize value

    cout << value << " " << pointer1Value << " "
         << pointer2Value << endl;

    pointer2Value = 7;

    cout << value << " "  << pointer1Value << " "
         << pointer2Value << endl;

    return 0;
}
```

The program output is:

```
#>./reference
5 5 5
7 7 7
```

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:

value
pointer1Value
pointer2Value

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 initialize a reference to some variable
```

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  MONDAY       = 1;
const int  TUESDAY      = 2;
const int  WEDNESDAY    = 3;
const int  THURSDAY     = 4;
const int  FRIDAY       = 5;
const int  SATURDAY     = 6;
const int  SUNDAY       = 7;
```

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

```
int day;
day = MONDAY;
if ( day == SATURDAY) {
   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 & John Peach
// enum.cpp
// Example program illustrating use of enumerated type declaration

#include <iostream>
#include <string>

using namespace std;

int main() {

    enum Days {MONDAY=1, TUESDAY, WEDNESDAY, THURSDAY,
               FRIDAY, SATURDAY, SUNDAY};

    Days today ;

    today = SATURDAY;
    if (today == SATURDAY) {
       cout << "Go windsurfing" << endl;
    }

    return 0;
}
```

The program output (as expected) is:

```
#>./enum
Go windsurfing
```

Had we not done `{MONDAY=1, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY}` but simply written `{MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY}` then the symbolic constant MONDAY would be associated with the value 0, TUESDAY 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 & John Peach
// struct.cpp
// example program illustrating use of
// nested structs

#include <iostream>
#include <string>

using namespace std;

int main() {

    enum Month  {JANUARY = 1, FEBUARY, MARCH, APRIL, MAY, JUNE, JULY,
                 AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER };
    enum DayOfTheWeek {SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
                       FRIDAY,  SATURDAY};

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

    struct date {
        int             year;
        Month           month;
        int             dayOfTheMonth;
        DayOfTheWeek    dayOfTheWeek;
        time            currentTime;
    };

    date today;
```

```
    today.year                  = 2012;
    today.month                 = JANUARY;
    today.dayOfTheMonth         = 16;
    today.dayOfTheWeek          = WEDNESDAY;
    today.currentTime.hour      = 7;
    today.currentTime.minute    = 12;
    today.currentTime.second    = 35;
    today.currentTime.isPM      = true;


    cout << "today's date is " << today.dayOfTheWeek << "-"
         << today.month << "-" << today.dayOfTheMonth << "-"
         << today.year << endl;
    cout << "time now is " << today.currentTime.hour << ":"
         << today.currentTime.minute << ":"
         << today.currentTime.second ;

    if (today.currentTime.isPM ) {
      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 & John Peach
// music.cpp
// example program illustrating use of structs

#include <iostream>

using namespace std;

int main() {

    enum guitarPlayers {KEITH_RICHARDS, DAVE_GILMOUR, ERIC_CLAPTON, MARK_KNOPFLER,
                        CHET_ATKINS, PETE_TOWNSHEND, JIMMY_PAGE, RON_WOOD,
                        ANGUS_YOUNG, MALCOLM_YOUNG };
    enum bassPlayers   {BILL_WYMAN, JOHN_ENTWISTLE, ROGER_WATERS, ROGER_GLOVER,
                        JOHN_PAUL_JONES };
    enum vocalists     {MICK_JAGGER, ROGER_DALTREY, MADONNA, ROBER_PLANT,
                        SYD_BARRETT };
    enum drummers      {KEITH_MOON, NICK_MASON, CHARLIE_WATTS, JOHN_BONHAM};

    typedef struct band {
        guitarPlayers      leadGuitar;
        guitarPlayers      rhythmGuitar;
        bassPlayers        bassist;
        vocalists          singer;
        drummers           percussionist;
```

```
    bool               isGood;
    bool               isDrummerDead;
    unsigned long int  annualIncome;
} BAND;

// Bands
BAND rollingStones, pinkFloyd, ledZeppelin, theWho;

// Store the information about the Rolling Stones
rollingStones.rhythmGuitar          = KEITH_RICHARDS;
rollingStones.leadGuitar            = RON_WOOD;
rollingStones.bassist               = BILL_WYMAN;
rollingStones.singer                = MICK_JAGGER;
rollingStones.percussionist         = CHARLIE_WATTS;
rollingStones.isGood                = true;
rollingStones.isDrummerDead         = false;
rollingStones.annualIncome          = 20123456;

// Store the information about Pink Floyd
pinkFloyd.rhythmGuitar              = DAVE_GILMOUR;
pinkFloyd.leadGuitar                = DAVE_GILMOUR;
pinkFloyd.bassist                   = ROGER_WATERS;
pinkFloyd.singer                    = SYD_BARRETT;
pinkFloyd.percussionist             = NICK_MASON;
pinkFloyd.isGood                    = true;
pinkFloyd.isDrummerDead             = false;
pinkFloyd.annualIncome              = 50000000;

// Store the information about Led Zeppelin
ledZeppelin.rhythmGuitar            = JIMMY_PAGE;
ledZeppelin.leadGuitar              = JIMMY_PAGE;
ledZeppelin.bassist                 = JOHN_PAUL_JONES;
ledZeppelin.singer                  = ROBER_PLANT;
ledZeppelin.percussionist           = JOHN_BONHAM;
ledZeppelin.isGood                  = true;
ledZeppelin.isDrummerDead           = true;
ledZeppelin.annualIncome            = 30000000;

// Store the information about The Who
theWho.rhythmGuitar                 = PETE_TOWNSHEND;
theWho.leadGuitar                   = PETE_TOWNSHEND;
theWho.bassist                      = JOHN_ENTWISTLE;
theWho.singer                       = ROGER_DALTREY;
theWho.percussionist                = KEITH_MOON;
theWho.isGood                       = true;
theWho.isDrummerDead                = true;
theWho.annualIncome                 = 25000000;

// Output some of the information so that we can see how it is stored
cout << rollingStones.rhythmGuitar << endl;
cout << rollingStones.leadGuitar << endl;
cout << rollingStones.bassist << endl;
cout << rollingStones.singer << endl;
cout << rollingStones.percussionist << endl;
cout << rollingStones.isGood << endl;
cout << rollingStones.isDrummerDead << endl;
```

```
    cout << rollingStones.annualIncome << endl;

    return 0;
}
```

Note that the program output is:

```
0
7
0
0
2
1
0
20123456
```