MECH215
Lecture Notes 4

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 pseudo-random sequence of numbers can be generated every time a program is run, without asking the user for a seed value:

```
// Author: Ted Obuchowicz and John Peach
// timeSeed.cpp
// 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
    }

    return 0;
}
```

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

```
1599787664
297162175
1458544492
128465630
1079387178
1915432695
```

```
1751250703
209895636
1022942420
698728349
```

Second Run

```
1212611733
818690775
105705768
826058055
624637668
289764914
438681437
1659074143
904665038
63106795
```

User-Defined Functions

In addition to library functions C++ allows the programmer to define their own functions. A function 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], ...]);

Note: [ ] is a standard notation to mean that something is optional.

where return_type can be any of the following:

1. A primitive data type (i.e. char, short, int, long, float, double, double long)
2. Defined data type (i.e. enum, struct)
3. A class (we will learn about this type later on)
4. A reference variable (i.e., int& or a reference to any of the above types, not only an int)
5. A pointer variable (i.e. double * or a pointer to any of the above type.) We will learn about pointers later
6. A void (signifies the absence of a return type)

Neither an array (we will learn about arrays 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.

The argument names are optional but it is a good idea to include them so that your code is self-documenting. For example:

return_type function_name(type, type, ...);

It is also possible to leave out the formal parameter specification all together. In this case, the compiler does not check the parameters but it can cause problems when linking the application (putting the pieces of a large program to act as a single program). For example we could have the following:

return_type function_name();

There is no good reason to do this for the type of programming that you are likely to do.

If there are no formal parameters in a function, then you declare this with the void data type as the formal parameter

return_type function_name(void);

Example 1:
A function which returns the maximum of the values passed as parameters.

```
// Author: Ted Obuchowicz and John Peach
// maxInt.cpp
// example program illustrating use of a user-defined function
// maxInt(int, int) which returns the max of two numbers

#include <iostream>
#include <string>

using namespace std;

// return the largest of two numbers
int maxInt(int num1, int num2) {
    if (num1 >= num2){
        return num1;
    } else {
        return num2;
    }
}

int main() {

    int num1, num2;     // User entered numbers

    //Get values from the user
    cout << "Please enter two integers: ";

    // Loop until the EOF signal
    while (cin >> num1 >> num2){
        cout << "The maximum is " << maxInt(num1,num2) << endl;
        cout << "Please enter two integers";
    }

    return 0;
}
```

The program output is:

```
Please enter two integers 2 56
The maximum is 56
```

Note that in the above example, we gave the complete definition of the function called maxInt in the program source code. The definition must be before the first time the function is invoked. In C++, the definition of the function also serves as its prototype. However, this is considered poor programming style. A function prototype should always be declared in the header file (.h file) and included in the .cpp file.

Example 2:

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

We can write the following and save it in a file called maxIntWithHeader.cpp

```
// Author: Ted Obuchowicz and John Peach
// maxIntWithHeader.cpp
// example program illustrating use of a user-defined function
// maxInt(int, int) which returns the max of two numbers

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

using namespace std;

int main() {

    int num1, num2;     // User entered numbers

    //Get values from the user
    cout << "Please enter two integers: ";

    // Loop until the EOF signal
    while (cin >> num1 >> num2){
        cout << "The maximum is " << maxInt(num1,num2) << endl;
        cout << "Please enter two integers";
    }

    return 0;
}
```

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

```
#include "maxInt.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 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 maxInt(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. However, it is a good style include the variable names.

We now have write the C++ code which defines our function maxInt. This will be saved in a third file called maxIntDefinition.cpp:

```
// return the largest of two numbers
int maxInt(int num1, int num2) {
    if (num1 >= num2){
        return num1;
    } else {
        return num2;
    }
}
```

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

```
#> g++ -o maxIntWithHeader maxIntWithHeader.cpp maxIntDefinition.cpp
#> ./maxIntWithHeader
Please enter two integers: 4 5
The maximum is 5
```

Call-by-Value versus Call-by-Reference

All of the function examples we have seen thus far 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.

Each time 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 only exist during a 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.

```
// Author: Ted Obuchowicz and John Peach
// passByValue.cpp
// example program illustrating
// call-by-value parameter 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) {
    cout << "The memory address of 'a' inside inc is: " << &a << endl;
    cout << "Value passed to inc is: " << a << endl;
    // increment the value of the passed parameter by 1
    cout << "inc changed the value to: " << ++a << endl;
}

int main() {

    int a = 5;
    cout << "The memory address of 'a' inside main is: " << &a << endl;
    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 << endl;

    return 0;
}
```

The program output may surprise some of you:

```
The memory address of 'a' inside main is: 0x7fff3f7b0dac
The value of 'a' before we call the function is: 5
The memory address of 'a' inside inc is: 0x7fff3f7b0d7c
Value passed to inc is: 5
inc changed the value to: 6
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?

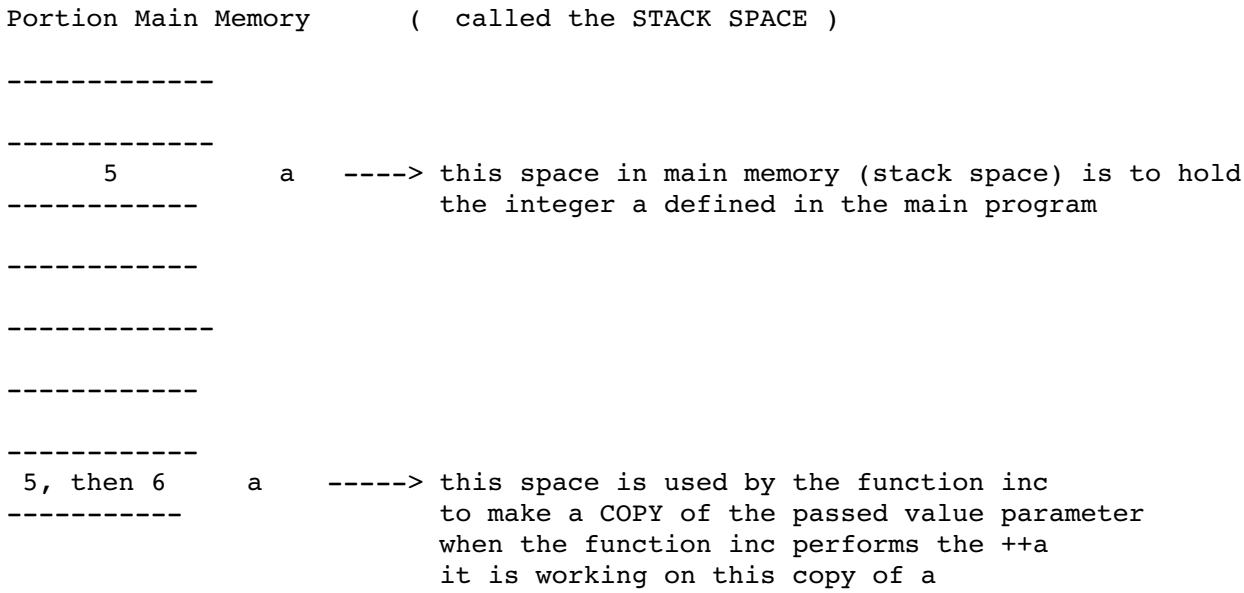
The program will exhibit the same behaviour even if we change the name of the formal parameter to something other than 'a'. For example, we can call it keith instead:

```
// Author: Ted Obuchowicz and John Peach
// passByValueKeith.cpp
// example program illustrating
// call-by-value parameter 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) {
    cout << "The memory address of 'keith' inside inc is: " << &keith << endl;
    cout << "Value passed to inc is: " << keith << endl;
    // increment the value of the passed parameter by 1
    cout << "inc changed the value to: " << ++keith << endl;
}

int main() {

    int a = 5;
    cout << "The memory address of 'a' inside main is: " << &a << endl;
    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 << endl;

    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 declare (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:



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++. 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 and John Peach
// passByReference.cpp
// example program illustrating
// call-by-value parameter 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) {
    cout << "The memory address of 'a' inside inc is: " << &a << endl;
    cout << "Value passed to inc is: " << a << endl;
    // increment the value of the passed parameter by 1
    cout << "inc changed the value to: " << ++a << endl;
}

int main() {

    int a = 5;
    cout << "The memory address of 'a' inside main is: " << &a << endl;
    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 << endl;

    return 0;
}
```

The program now produces the 'expected' output of:

```
The memory address of 'a' inside main is: 0x7fffc8d0bf2c
The value of 'a' before we call the function is: 5
The memory address of 'a' inside inc is: 0x7fffc8d0bf2c
Value passed to inc is: 5
inc changed the value to: 6
Hello, I'm back from the function, this is the value of 'a': 6
```

Amazing what a single & makes in a program.

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 and John Peach
// swapPassByValue.cpp
// 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
```

```
    cout << "In swap (before the switch)\tnum1:\t" << num1 << "\tnum2:\t"
         << num2 << endl;
    temp = num1;
    num1 = num2;
    num2 = temp;
    cout << "In swap (after the switch)\tnum1:\t" << num1 << "\tnum2:\t"
         << num2 << endl;
}

int main() {

    int x = 10;     // first value that we want to swap
    int y = 20;     // first value that we want to swap

    cout << "In main: (before call to swap)\tx:\t" << x << "\ty:\t" << y << endl;

    // call the swap function and pass the VALUES of x and y to it
    swap(x,y);

    cout << "In main: (after call to swap)\tx:\t" << x << "\ty:\t" << y << endl;

    return 0;
}
```

Here is the program output:

```
In main: (before call to swap)     x:   10   y:   20
In swap (before the switch)   num1:     10   num2:     20
In swap (after the switch)    num1:     20   num2:     10
In main: (after call to swap)      x:   10   y:   20
```

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 is in the main memory at the instantaneous swap is called with parameters x and y has executed the line temp = num1. temp is a LOCAL variable, it is only accessible within function swap, no other function 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 and John Peach
// swapPassByReference.cpp
// 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
    cout << "In swap (before the switch)\tnum1:\t" << num1 << "\tnum2:\t"
         << num2 << endl;
```

```
    temp = num1;
    num1 = num2;
    num2 = temp;
    cout << "In swap (after the switch)\tnum1:\t" << num1 << "\tnum2:\t"
         << num2 << endl;
}

int main() {

    int x = 10;     // first value that we want to swap
    int y = 20;     // first value that we want to swap

    cout << "In main: (before call to swap)\tx:\t" << x << "\ty:\t" << y << endl;

    // call the swap function and pass the VALUES of x and y to it
    swap(x,y);

    cout << "In main: (after call to swap)\tx:\t" << x << "\ty:\t" << y << endl;

    return 0;
}
```

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

```
In main: (before call to swap)     x:   10   y:   20
In swap (before the switch)   num1:     10   num2:     20
In swap (after the switch)    num1:     20   num2:     10
In main: (after call to swap)      x:   20   y:   10
```

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 and John Peach
// countMeUpNotStatic.cpp
// 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 countMeUp(void) {
    int counter = 0;    // The num of times the function is called
    counter++;          // Increment the counter
    return counter;     // Return the counter
```

```
}

int main() {

    for(int i = 0 ; i < 10; i++) {
        cout << countMeUp() << endl;
    }

    return 0;
}
```

When we run the program, we will see 10 lines of:

```
1
1
1
```
and so on...

Every time function countMeUp() 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 countMeUp(). 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 and John Peach
// countMeUpStatic.cpp
// example program illustrating lifetime of static 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 countMeUp(void) {
```

```
    static int counter = 0; // 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
                            // initialized once and its value will be
                            // remembered across function invocations

    counter++;              // Increment the counter
    return counter;         // Return the counter
}

int main() {

    for(int i = 0 ; i < 10; i++) {
        cout << countMeUp() << endl;
    }

    return 0;
}
```

The program output is:

```
1
2
3
4
5
6
7
8
9
10
```

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 and John Peach
// countMeUpGlobal.cpp
// example program illustrating lifetime of 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 increment a global variable.

int countMeUp(void) {
    counter++;              // Increment the counter
    return counter;         // Return the counter
}

int main() {

    for(int i = 0 ; i < 10; i++) {
        cout << countMeUp() << endl;
    }

    return 0;
}
```

The program out is the 10 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 side-effects. Side-effects 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 side-effects, a program contains a countMeUp and and countMeDown function:

```
// Author: Ted Obuchowicz and John Peach
// countMeUpGlobalSideEffect.cpp
// example program illustrating the use of a GLOBAL
// variable with side effects

#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 increment a global variable.

void countMeUp(void) {
    counter++;              // Increment the counter
    cout << "Value of counter is: " << counter << endl;
}

void countMeDown(void) {
    counter--;              // Increment the counter
    cout << "Value of counter is: " << counter << endl;
}

int main() {

    for(int i = 0 ; i < 10; i++) {
        countMeUp();
        countMeDown();
    }

    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
```

The extern Keyword

Suppose we had written our above program in 5 separate files; main program (countMeExtern.cpp); two for the function countMeUp (countMeUp.h and countMeUp.cpp); and two for the function countMeDown (countMeDown.h and countMeDown.cpp):

File 1: The main program:

```
// Author: Ted Obuchowicz and John Peach
// countMeExten.cpp
// example program illustrating exten

#include <iostream>
#include <string>
#include "countMeUp.h"
#include "countMeDown.h"
using namespace std;

// define a GLOBAL integer variable
// which is seen and can be accessed by all functions

int counter = 0;

int main() {

    for(int i = 0 ; i < 10; i++) {
        countMeUp();
        countMeDown();
    }

    return 0;
}
```

File 2: header for countMeUp

```
// countMeUp.h
void countMeUp(void);
```

File 3: definition for countMeUp

```
// countMeUp.cpp
#include <iostream>
using namespace std;

void countMeUp(void) {
```

```
    extern int counter; // the extern tells the compiler that
                        // this variable was declared
                        // "somewhere else"

    cout << "Value of counter is: " << ++counter << endl;
}
```

File 4: header for countMeDown

```
// countMeDown.h
void countMeDown(void);
```

File 5: definition of countMeDown

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

void countMeDown(void) {

    extern int counter; // the extern tells the compiler that
                        // this variable was declared
                        // "somewhere else"

    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 do not complain about it"

To compile this program we have to list all the .cpp files on the command line but not the .h files.

```
#> g++ -o countMeExten countMeExten.cpp countMeUp.cpp countMeDown.cpp
```

NOTE: IT WOULD BE AN ERROR TO DO SOMETHING LIKE:

```
void countMeDown(void) {
    extern int counter = 1; // THIS IS INVALID
    cout << "Value of counter is: " << --counter << endl;
}
```

Scope and Storage Classes

The following program does not compile:

```
// Author: Ted Obuchowicz and John Peach
// outOfScope.cpp
// example program illustrating use of illegal access to
// an out-of-scope variable

#include <iostream>
#include <string>

using namespace std;

void outOfScope(int dummy1, int dummy2) {

    // Report where we are and the parameters
    cout << "Hello from function outOfScope" << endl;
    cout << dummy1 << "\t" << dummy2 << endl;

    // Change the value of the parameters and report
    dummy1 = 10;
    dummy2 = 20;
    cout << dummy1 << "\t" << dummy2 << endl;

    // Create a new var and report its value
    int dummy3 = 500;
    cout << dummy3 << endl;
}

int main() {

    int a = 1;          // dummy var
    int b = 2;          // dummy var

    // Call outOfScope to create dummy1, dummy2 and dummy3
    outOfScope(a,b);
    cout << dummy1 << "\t" << dummy2 << endl;
    cout << dummy3 << endl;

    return 0;
}
```

These are the errors reported by the g++ compiler:

```
outOfScope.cpp: In function ‘int main()’:
outOfScope.cpp:34:13: error: ‘dummy1’ was not declared in this scope
outOfScope.cpp:34:31: error: ‘dummy2’ was not declared in this scope
outOfScope.cpp:35:13: error: ‘dummy3’ was not declared in this scope
```

The compiler is complaining about lines 34 and 35 in the program which correspond to the following lines of code in function main():

```
    cout << dummy1 << "\t" << dummy2 << endl;   // Line 34
    cout << dummy3 << endl;                     // Line 35
```

The compiler is saying that the identifiers dummy1, dummy2 and dummy3 are undeclared. These attempts to access these variables are illegal since dummy1, dummy2, and dummy3 are said to be out of scope for function main(). dummy1, dummy2, and dummy3 are LOCAL to function outOfScope(). This means that they are only ACCESSIBLE inside the function outOfScope(). 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 variable can only be used in the block in which it is declared 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 and John Peach
// scopeBlock.cpp
// example program illustrating scope of variables

#include <iostream>
#include <string>

using namespace std;
```

```
int main() {

    int dummy = 5;
    cout << "The value of dummy in main is:  " << dummy
         << ", dummy's address is " << &dummy << endl;

    // define a block with another different dummy integer in it
    {
        int dummy = 28;
        cout << "The value of dummy in the inner block is: " << dummy
             << ", dummy's address is " << &dummy << endl;
    }

    return 0;
}
```

The program output is:

```
The value of dummy in main is:  5, dummy's address is 0x7fff9cc711d8
The value of dummy in the inner block is: 28, dummy's address is 0x7fff9cc711dc
```

You can see how two different main memory locations are used to store the two dummy.

As another example, here is a program which contains a block which contains a nested block:

```
// Author: Ted Obuchowicz and John Peach
// scopeNestedBlock.cpp
// example program illustrating use of scope and nested blocks

#include <iostream>
#include <string>

using namespace std;

int main() {

    int i = 0;
    cout << " i = " << i << "\taddress 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 << "\taddress of i = " << &i << endl;
        {
            // define a new block nested inside
            int j = 4;
            cout << " j = " << j << "\taddress of j = "  << &j << endl;
            // in this nested block, the i with value 6 is still visible
            cout <<  " i = " << i << "\taddress 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 << "\taddress of i = " << &i << endl;

    return 0;
}
```

Here is the output:

```
 i = 0    address of i = 0x7fff02318474
 i = 6    address of i = 0x7fff02318478
 j = 4    address of j = 0x7fff0231847c
 i = 6    address of i = 0x7fff02318478
 i = 0    address of i = 0x7fff02318474
```

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 0x7fff02318474, then there is the i defined in the block which has value 6 and is stored in location 0x7fff02318478. Note how this a second i is also visible in the nested block.

Global Scope

Declarations not in any statement block are said to be of GLOBAL SCOPE. Objects of global scope are visible in all scopes and therefore by all functions.

Here is a simple example:

```
// Author: Ted Obuchowicz and John Peach
// global.cpp
// example program illustrating use of a global scope object

#include <iostream>
#include <string>

using namespace std;

int visibleEverywhere = 55;

void f1(void) {
    cout << "value of visibleEverywhere = " << visibleEverywhere
         << "\taddress = " << &visibleEverywhere << endl;
}

void f2(void) {
    cout << "value of visibleEverywhere = " << visibleEverywhere
```

```
         << "\taddress = " << &visibleEverywhere << endl;
}

int main() {

    cout << "value of visibleEverywhere = " << visibleEverywhere
         << "\taddress = " << &visibleEverywhere << endl;
    f1();
    f2();

    return 0;
}
```

The output is:

```
value of visibleEverywhere = 55    address = 0x601058
value of visibleEverywhere = 55    address = 0x601058
value of visibleEverywhere = 55    address = 0x601058
```

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 visibleEverywhere with value of 4567:

```
// Author: Ted Obuchowicz and John Peach
// globalLocal.cpp
// example program illustrating use of a global scope object

#include <iostream>
#include <string>

using namespace std;

int visibleEverywhere = 55;

void f1(void) {
    int visibleEverywhere = 4567;
    cout << "value of visibleEverywhere = " << visibleEverywhere
         << "\taddress = " << &visibleEverywhere << endl;
}

void f2(void) {
    cout << "value of visibleEverywhere = " << visibleEverywhere
         << "\taddress = " << &visibleEverywhere << endl;
}
```

```
int main() {

    cout << "value of visibleEverywhere = " << visibleEverywhere
         << "\taddress = " << &visibleEverywhere << endl;
    f1();
    f2();

    return 0;
}
```

The program output is now:

```
value of visibleEverywhere = 55        address = 0x601058
value of visibleEverywhere = 4567      address = 0x7fff5a50a9cc
value of visibleEverywhere = 55        address = 0x601058
```

Note how there are now two different variables called visibleEverywhere 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 visibleEverywhere 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 visibleEverywhere variable seen in function f1() :

```
// Author: Ted Obuchowicz and John Peach
// globalScopeResolution.cpp
// example program illustrating use of a global scope object

#include <iostream>
#include <string>

using namespace std;

int visibleEverywhere = 55;

void f1(void) {
    int visibleEverywhere = 4567;
    cout << "value of visibleEverywhere = " << visibleEverywhere
         << "\taddress = " << &visibleEverywhere << endl;

    // Use scope resolution to access the global var
    cout << "value of visibleEverywhere = " << ::visibleEverywhere
         << "\taddress = " << &::visibleEverywhere << endl;
}

void f2(void) {
    cout << "value of visibleEverywhere = " << visibleEverywhere
         << "\taddress = " << &visibleEverywhere << endl;

    // note: f2 can also refer to the global visibleEverywhere using the
```

```
    // scope resolution but there is on need as there is no local/global
    // collision so ::visibleEverywhere is the same as visibleEverywhere

}

int main() {

    cout << "value of visibleEverywhere = " << visibleEverywhere
         << "\taddress = " << &visibleEverywhere << endl;
    f1();
    f2();

    return 0;
}
```

This is the output:

```
value of visibleEverywhere = 55    address = 0x601058
value of visibleEverywhere = 4567  address = 0x7fff549a405c
value of visibleEverywhere = 55    address = 0x601058
value of visibleEverywhere = 55    address = 0x601058
```

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 later in the 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 'persistence' 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 program or when the program leaves the defining block. Function parameters are considered to be automatic variables. Automatic variables are stored in a special 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 someFunction() {
    auto int dummy;
    // 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 location. 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 i = 0 ; i < 9 ; i++) {
    // do something inside the loop
}
```

Sometimes the compiler cannot honour 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, i, 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 program duration. The compiler sets aside a fixed size of main memory (distinct from the stack area) to use for a program's static variables. If a static variable is not explicitly given an initial value by the programmer, the compiler gives it an initial value of 0. We have already seen the use of a static variable in our example program which used a function which declared a static variable 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 fewer or greater number of arguments, the compiler will report an error as in the following (incorrect) program:

```
// Author: Ted Obuchowicz and John Peach
// missingParameter.cpp
// example program illustrating use of incorrect function call

#include <iostream>
#include <string>

using namespace std;

int doSomething(int x, int y, int z) {
    return ( x - y - z);
```

```
}

int main() {

    int a, b, c;    // dummy vars
    a = b = c = 5;
    cout << doSomething(a,b) << endl; // compile time error
    return 0;
}
```

The compiler output is:

```
missingParameter.cpp: In function ‘int main()’:
missingParameter.cpp:19:28: error: too few arguments to function ‘int
doSomething(int, int, int)’
missingParameter.cpp:11:5: note: declared here
```

line 11 is the opening function definition for doSomething. Line 19 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 default values.

Here is the same program, where we specify a default value for the last parameter:

```
// Author: Ted Obuchowicz and John Peach
// missingParameter.cpp
// example program illustrating use of incorrect function call

#include <iostream>
#include <string>

using namespace std;


int doSomething(int x, int y, int z = 0) {
    // if a third parameter is not given, z will be set to 0
    // if a third parameter is given, z will be set to that value
    return ( x - y - z);
}

int main() {

    int a, b, c;    // dummy vars
    a = b = c = 5;
    cout << doSomething(a,b) << endl; // compiles and prints 0
    return 0;
}
```

It is important to remember that when defining a function with default parameters, the default parameters must 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 and John Peach
// overrideDefault.cpp
// 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
values

    return 0;

}
```

The output is:

```
 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. The actual error message will vary from compiler to compiler but it will be something like:

```
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:

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 & f2 assume defaults

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 definitions must vary in their parameter list. The compiler then determines which definition of the overloaded function is to be used by examining the argument list and finding a definition 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++.

```
// Author: Ted Obuchowicz and John Peach
// swapNoOverload.cpp
// example program illustrating swapping

#include <iostream>
#include <string>

using namespace std;

struct someStruct {
    int first;
    float second;
};

void swapInt(int& a, int& b) {
    int temp;
    temp = b;
    b = a;
    a = temp;
}

void swapSomeStruct( someStruct& a, someStruct& b) {
    someStruct temp;            // temp var

    // Copy b into temp
    temp.first  = b.first;
    temp.second = b. second;

    // Copy a into b
    b.first  = a.first;
    b.second = a.second;

    // Copy temp into a
    a.first = temp.first;
    a.second = temp.second;
}

int main() {

    int dummy1 = 10;
    int dummy2 = 20;
    someStruct struct1, struct2;

    // Report the values, swap and report again
    cout << "dummy1 = " << dummy1 << "\tdummy2 = " << dummy2 << endl;
    swapInt(dummy1, dummy2);
    cout << "After the swap" << endl;
    cout << "dummy1 = " << dummy1 << "\tdummy2 = " << dummy2 << endl;

    // initialize the values in the struct
    struct1.first =  5;
    struct1.second = 5.5;

    struct2.first =  6;
    struct2.second = 6.6;
```

```
    // Report the struct values
    cout << "struct1: first: " << struct1.first << "\t" << struct1.second << endl;
    cout << "struct2: first: " << struct2.first << "\t" << struct2.second << endl;

    // swap the structs
    swapSomeStruct(struct1, struct2);

    // Report the struct values after the swap
    cout << "After the swap" << endl;
    cout << "struct1: first: " << struct1.first << "\t" << struct1.second << endl;
    cout << "struct2: first: " << struct2.first << "\t" << struct2.second << endl;

    return 0;
}
```

The output of the program is:

```
dummy1 = 10    dummy2 = 20
After the swap
dummy1 = 20    dummy2 = 10
struct1: first: 5   5.5
struct2: first: 6   6.6
After the swap
struct1: first: 6   6.6
struct2: first: 5   5.5
```

If we overload the function swap with two different definitions (one for swapping two integers, the other for swapping two someStructs), then we obtain the following:

```
// Author: Ted Obuchowicz and John Peach
// swapOverload.cpp
// example program illustrating overloading

#include <iostream>
#include <string>

using namespace std;

struct someStruct {
    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 'someStruct' arguments instead
```

```
// of integer arguments

void swap( someStruct& a, someStruct& b) {
    someStruct temp;            // temp var

    // Copy b into temp
    temp.first  = b.first;
    temp.second = b. second;

    // Copy a into b
    b.first  = a.first;
    b.second = a.second;

    // Copy temp into a
    a.first = temp.first;
    a.second = temp.second;
}

int main() {

    int dummy1 = 10;
    int dummy2 = 20;
    someStruct struct1, struct2;

    // Report the values, swap and report again
    cout << "dummy1 = " << dummy1 << "\tdummy2 = " << dummy2 << endl;
    swap(dummy1, dummy2);
    cout << "After the swap" << endl;
    cout << "dummy1 = " << dummy1 << "\tdummy2 = " << dummy2 << endl;

    // initialize the values in the struct
    struct1.first =  5;
    struct1.second = 5.5;

    struct2.first =  6;
    struct2.second = 6.6;

    // Report the struct values
    cout << "struct1: first: " << struct1.first << "\t" << struct1.second << endl;
    cout << "struct2: first: " << struct2.first << "\t" << struct2.second << endl;

    // swap the structs
    swap(struct1, struct2); // compiler will choose the definition of swap
                            // which has the two someStruct as arguments


    // Report the struct values after the swap
    cout << "After the swap" << endl;
    cout << "struct1: first: " << struct1.first << "\t" << struct1.second << endl;
    cout << "struct2: first: " << struct2.first << "\t" << struct2.second << endl;

    return 0;
}
```

The program output is:

```
dummy1 = 10    dummy2 = 20
After the swap
dummy1 = 20    dummy2 = 10
struct1: first: 5   5.5
struct2: first: 6   6.6
After the swap
struct1: first: 6   6.6
struct2: first: 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
// ambiguates.cpp
#include <iostream>
using namespace std;

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.

```
ambiguates.cpp: In function ‘float foo(int)’:
ambiguates.cpp:11:17: error: new declaration ‘float foo(int)’
ambiguates.cpp:6:5: error: ambiguates old declaration ‘int foo(int)’
```

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 definition of a function to use as follows:

"When a function f is called, the compiler must figure out which of the functions 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 arguments of all functions called f. The idea is to invoke the function that is the BEST match on the arguments and give a compile-time error if no function is a best match. ... A series of criteria are 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 overloaded functions, the compiler examines the list of actual arguments passed to the overloaded function and then chooses the function whose formal argument list BEST matches the passed argument list. Sometimes the compiler is not able to resolve the function name overloading and will report an error:

```
// Author: Ted Obuchowicz and John Peach
// swapWhichVersion.cpp
// example program illustrating overloading

#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;
    cout "int swap" << endl;
}

// 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;
    cout "double swap" << endl;
}

int main() {

    int intDummy1 = 10;
    int intDummy2 = 20;

    double dbDummy1 = 100.0;
    double dbDummy2 = 200.0;

    // Report the values for int, swap and report again
    cout "int version" << endl;
    cout << "intDummy1 = " << intDummy1 << "\intDummy2 = " << intDummy2 << endl;
    swap(intDummy1, intDummy2); // compiler will choose the swap with two integers
    cout << "intDummy1 = " << intDummy1 << "\intDummy2 = " << intDummy2 << endl;

    // Report the values for double, swap and report again
    cout "double version" << endl;
    cout << "dbDummy1 = " << dbDummy1 << "\dbDummy2 = " << dbDummy2 << endl;
    swap(dbDummy1, dbDummy2); // compiler will choose the swap with two double
    cout << "dbDummy1 = " << dbDummy1 << "\dbDummy2 = " << dbDummy2 << endl;

    // Report the values, swap and report again
    cout "Which version?" << endl;
    cout << "intDummy1 = " << intDummy1 << "\dbDummy1 = " << dbDummy1 << endl;
    swap(intDummy1, dbDummy1); // compiler is UNABLE to convert the data types and
                               // resolve which overloaded definition to use
    cout << "intDummy1 = " << intDummy1 << "\dbDummy1 = " << dbDummy1 << endl;

    return 0;
}
```

The compiler reports the following errors:

```
swapWhichVersion.cpp: In function ‘int main()’:
swapWhichVersion.cpp:51:29: error: no matching function for call to ‘swap(int&,
double&)’
swapWhichVersion.cpp:51:29: note: candidates are:
swapWhichVersion.cpp:12:6: note: void swap(int&, int&)
swapWhichVersion.cpp:12:6: note:   no known conversion for argument 2 from
‘double’ to ‘int&’
swapWhichVersion.cpp:23:6: note: void swap(double&, double&)
swapWhichVersion.cpp:23:6: note:   no known conversion for argument 1 from ‘int’
to ‘double&’
/usr/include/c++/4.6/bits/basic_string.h:2658:5: note: template<class _CharT,
class _Traits, class _Alloc> void std::swap(std::basic_string<_CharT, _Traits,
_Alloc>&, std::basic_string<_CharT, _Traits, _Alloc>&)
/usr/include/c++/4.6/bits/move.h:136:5: note: template<class _Tp, long unsigned
int _Nm> void std::swap(_Tp (&)[_Nm], _Tp (&)[_Nm])
/usr/include/c++/4.6/bits/move.h:122:5: note: template<class _Tp> void
std::swap(_Tp&, _Tp&)
```

The compiler is referring to the swap(intDummy1, dbDummy1). It is unable to find a best match since there are two possible candidates. The compiler could choose to convert the double variable called dbDummy1 to an reference integer and invoke the integer version of swap, or alternatively the compiler could choose to promote the integer intDummy to its reference double version and invoke the swap which uses two reference doubles as arguments. It will try to widen the data to convert and will also narrow the data but there are data types that cannot be converted and in that case it will generate an error.

Function overload resolution is a complicated and tricky matter, 14 pages are allocated to the subject in "The Annotated C++ Reference Manual" but it is very logical.