Random Number Generation

C++

It is often useful to generate random numbers to produce simulations or games (or homework problems :) One way to generate these numbers in C++ is to use the function rand(). Rand is defined as:

```
#include <cstdlib>
int rand();
```

The rand function takes no arguments and returns an integer that is a pseudo-random number between 0 and RAND_MAX. On transformer, RAND_MAX is 2147483647. What is a pseudo-random number? It is a number that is not truly random, but appears random. That is, every number between 0 and RAND_MAX has an equal chance (or probability) of being chosen each time rand() is called. (In reality, this is not the case, but it is close).

For example, the following program might print out:

```
cout <<rand() << endl;
cout <<rand() << endl;
cout <<rand() << endl;
```

```
1804289383
846930886
1681692777
```

Here we "randomly" were given numbers between 0 and RAND_MAX. What if we only wanted numbers between 1 and 10? We will need to scale the results. To do this we can use the modulus (%) operator.

Any integer modulus 10 will produce a number from 0-9. In other words, if we divide a number by 10, the remainder has to be from 0 to 9. It's impossible to divide a number by 10 and end up with a remainder bigger than or equal to ten. In our case:

```
1804289383% 10  = 3

846930886% 10    = 6

1681692777% 10  = 7
```

Consequently, we can take rand() % 10 to give us numbers from 0-9. If we want numbers from 1-10 we can now just scale up by adding one. The final result is:

```
cout  << (rand() % 10) + 1 << endl;
```

If you run this program many times, you'll quickly notice that you end up with the same sequence of random numbers each time. This is because these numbers are not truly random, but pseudorandom.

Repeat calls to rand merely return numbers from some sequence of numbers that appears to be random. Each time we call rand, we get the next number in the sequence.

If we want to get a different sequence of numbers for each execution, we need to go through a process of *randomizing*. Randomizing is "seeding" the random number sequence, so we start in a different place. The function that does this is srand() which takes an integer as the seed:

```
        void srand(int seed);
```

It is important to only invoke the srand call ONCE at the beginning of the program. There is no need for repeat calls to seed the random number generator (in fact, it will make your number less evenly distributed).

A commonly used technique is to seed the random number generator using the clock. The time() function will return the computer's time. On transformer, this is expressed in terms of the number of seconds that have elapsed since Jan 1, 1970 (the Epoch). The function time(NULL) will return the number of seconds elapsed in computer time.

```
        #include <ctime>
        srand(time(NULL));
        cout << (rand() %10) + 1;
```

This produces different values each time the program is run.

Call By Value – C++ and Java

Both Java and C++ invoke functions using a mechanism called "Call By Value". If we pass a variable to a function then the function gets the value contained in the variable. However, any changes that are made to the variable in the function are not reflected back in the calling program. These parameters are considered local variables. To illustrate, consider the following:

```
void  ChangeValues(int x);                    // Prototype
int main()
{
        int x=5;
        ChangeValues(x);
        cout << "Back in main: " << x << endl;
        return 0;
}

void ChangeValues(int x)
{
        cout << "In change values: " <<  x << endl;
        x = 10;
        cout << "In change values: " <<  x << endl;
        return;
}
```

When this program is run, we get:

```
        In change values: 5
        In change values: 10
        Back in main: 5
```

Note that inside the function ChangeValues, we initially get x=5, or the value that was assigned in main to x. Then x is changed to 10, and the new change is reflected in the print statement. However, when we return back to the main function, x is unchanged and is back to 5.

The parameter x inside ChangeValues is referred to as a local variable, because changes to it are only enforced within the scope of the function. We'll have more to say about scoping later. Note that we can still have functions that use pass by value, but return a value:

```
int  ChangeValues(int x);                       // Prototype

int main()
{
        int x=5, y=0;
        y=ChangeValues(x);
        cout << “Back in main x= “ << x << “ y=” << y << endl;
        return 0;
}

int ChangeValues(int x)
{
        int y;
        y = 20;
        x = 10;
        return 10;
}
```

This will output:

```
        Back in main, x=5 y=10
```

Or we returned a value of 10 which is reflected in main. The variable 'y' defined in ChangeValues is also a local variable, and only exists within the scope of the ChangeValues function. Back in main, we are referencing main's variable y, not the y defined in ChangeValues.

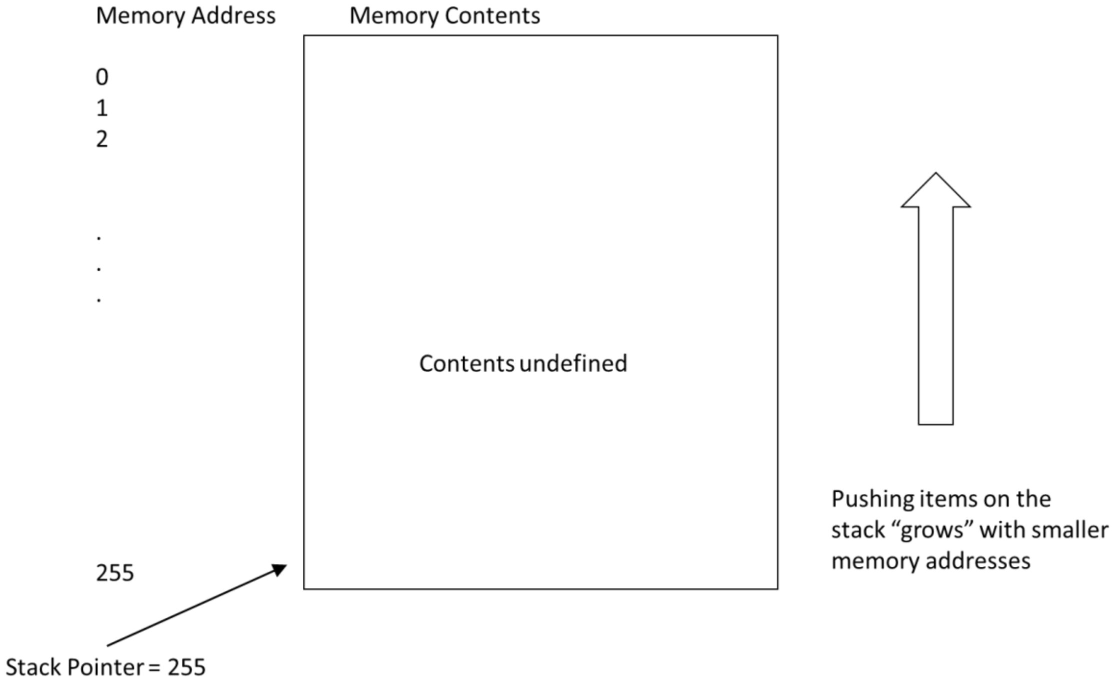
Underneath the Hood – The Stack and Parameter Passing

What is happening when we call functions and pass parameters back and forth? It is very useful to know how this is done architecturally in the computer. Local variables and parameters are passed using the stack in the computer.

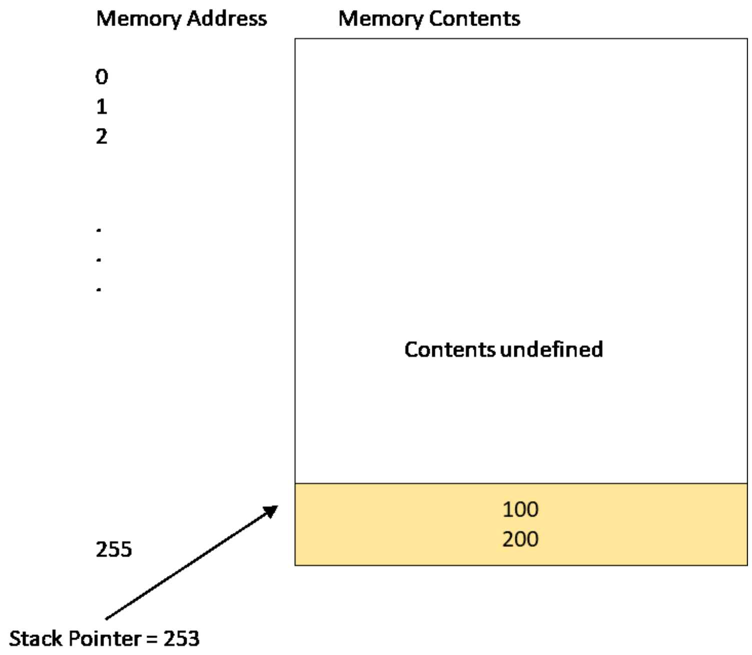
The Stack

You can visualize the stack like a stack of trays in a cafeteria. A person grabs a tray from the top of the stack. Clean trays are also added to the top of the stack. In general operation, you never touch anything that is not on top of the stack. This has the property of LIFO – Last In, First Out. The last thing you put into the stack is the first thing that will be taken out. In contrast, the first thing put into the stack is the last thing that is taken out.

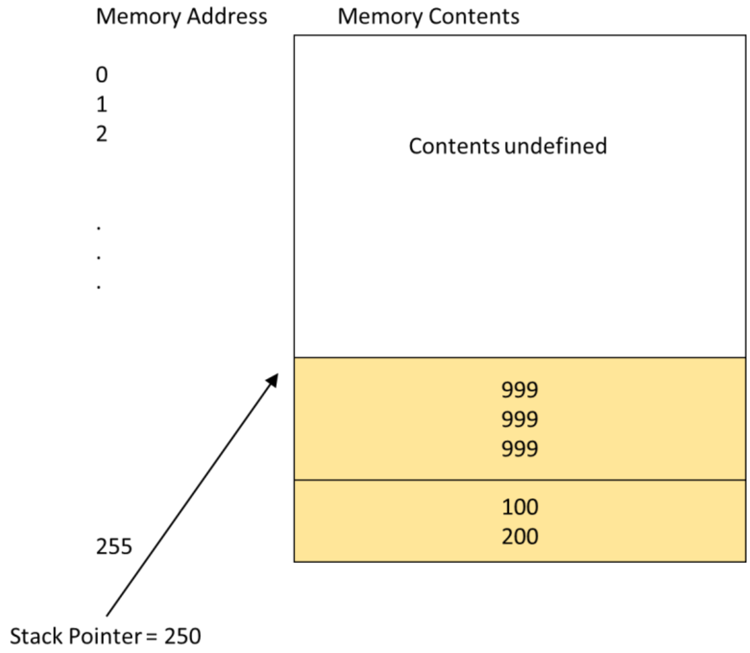
The computer implements the stack using a chunk of memory, with a special register named the **stack pointer** that remembers the place in memory that contains the top of the stack. Let's say our stack pointer is currently pointing to memory address 255. We push **stack frames** onto the stack, where each stack frame contains data of variable size that we would like to store. Here is an initial picture:



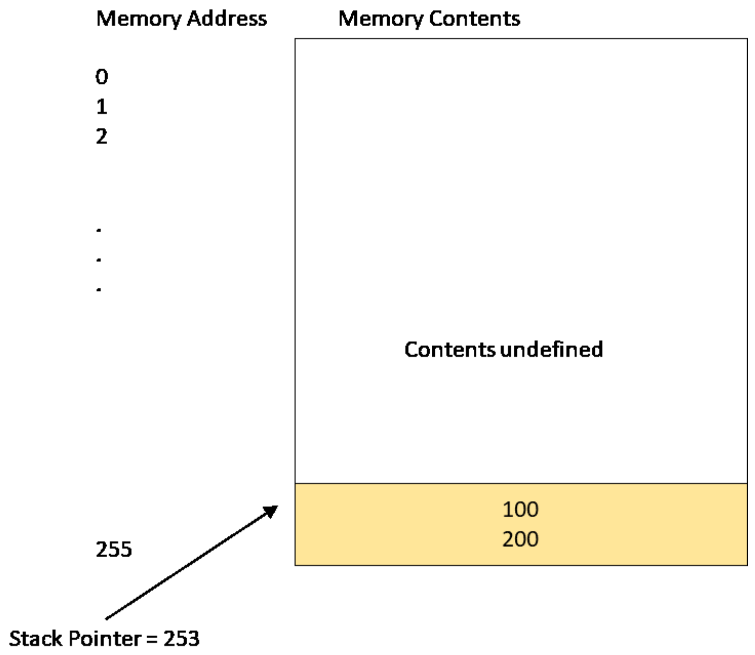
Let's say that we push a stack frame containing the numbers "100" and "200" that occupies two memory locations. The memory picture now looks like this:



If we push another stack frame with the numbers 999, 999, and 999 that occupies three memory locations then now the memory picture looks like this:



If we pop the top stack frame then we are back to this:



Note that the computer must know where the bounds of the stack are, or we might overrun into memory that is not allocated for the stack.

What is the stack used for? When we call a function, it is used to store the contents of variables and to remember what the next instruction is that we should execute when the function returns. This is needed in case there is an arbitrary number of nested functions (or recursion!)

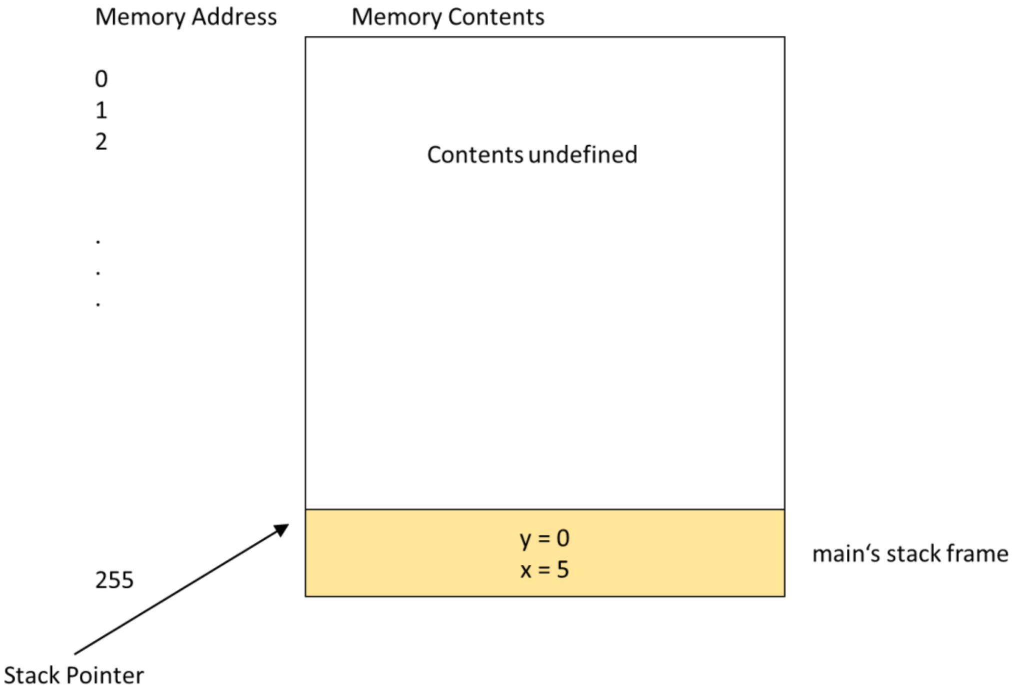
Let's see how our stack might look upon this simple ChangeValues sample program. I have added line numbers to indicate lines of code, which would map to memory addresses in the computer.

```
1     int main()
2     {
3           int x=5, y=0;
4           y=ChangeValues(x);
5           cout << "Back in main x= " << x << " y=" << y << endl;
6           return 0;
7     }

8     int ChangeValues(int x)
9     {
10          int y;
11          y = 20;
12          x = 10;
13          return 10;
14    }
```

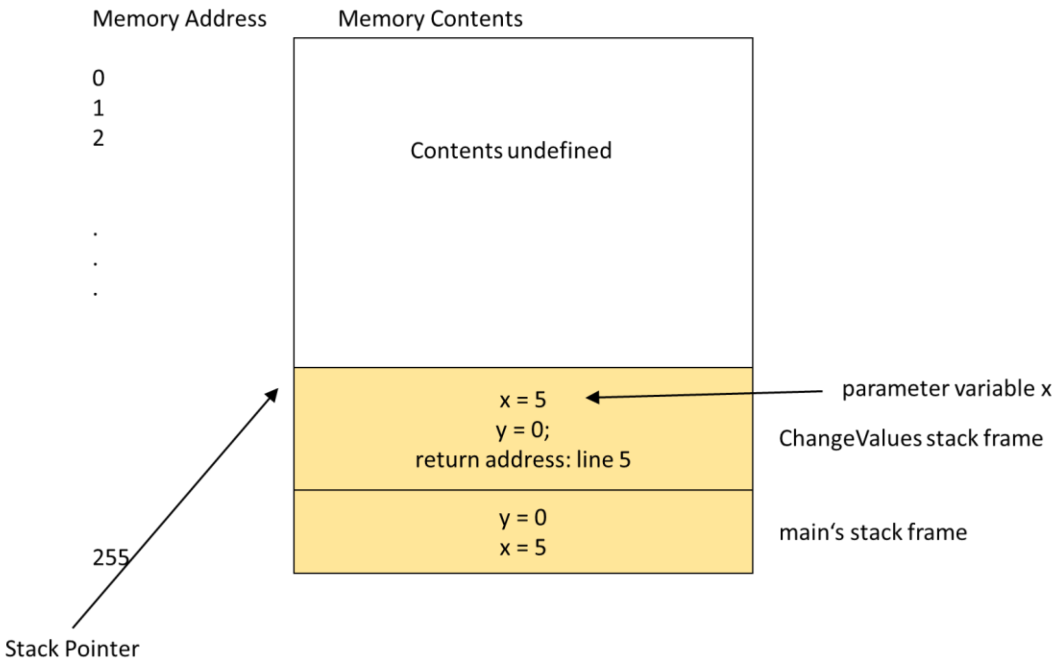
Variables are actually defined on the stack, so when main executes, it is going to declare variables x and y in terms of a stack frame. These are called **automatic** or **auto** variables.

Upon starting main:



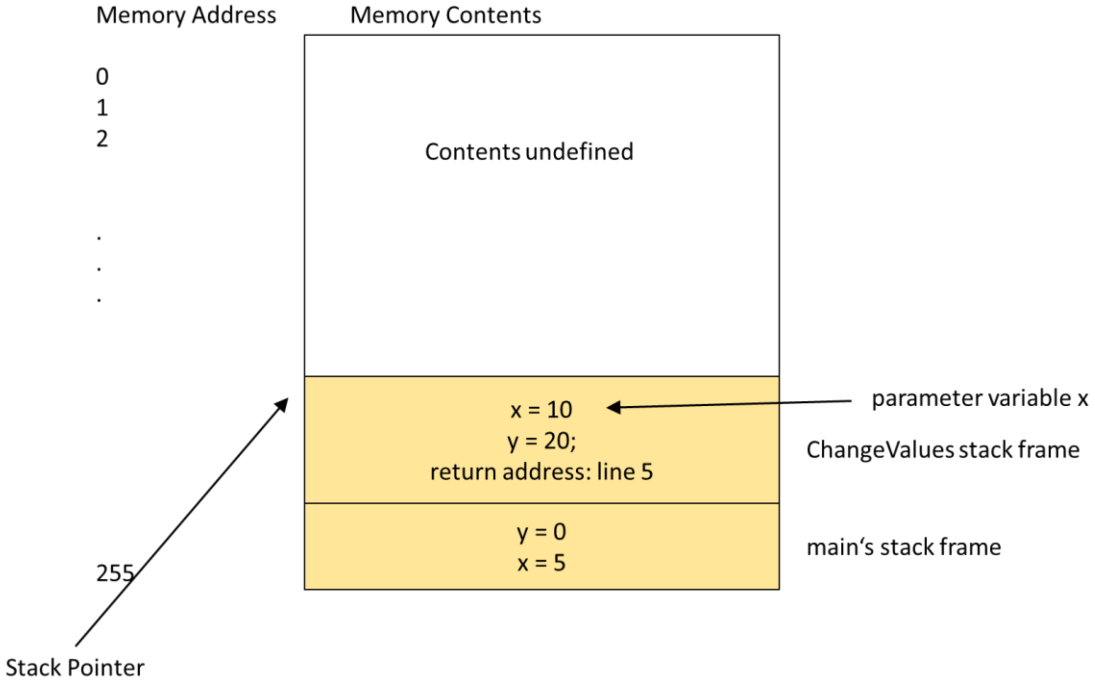
When main is executing, the knows which memory addresses are used to store variables x and y.

What happens when we make the function call to ChangeValues() ? We make a new stack frame for ChangeValues that contains data for the parameters and local variables. We also need to remember the place in memory we should resume execution after the function returns. This is called the return address, which in this example maps back to line 5 in the code (actually line 4 since we need to do the assignment to y). This stack frame is pushed onto the stack and we decrement the stack pointer to hold the new stack frame.

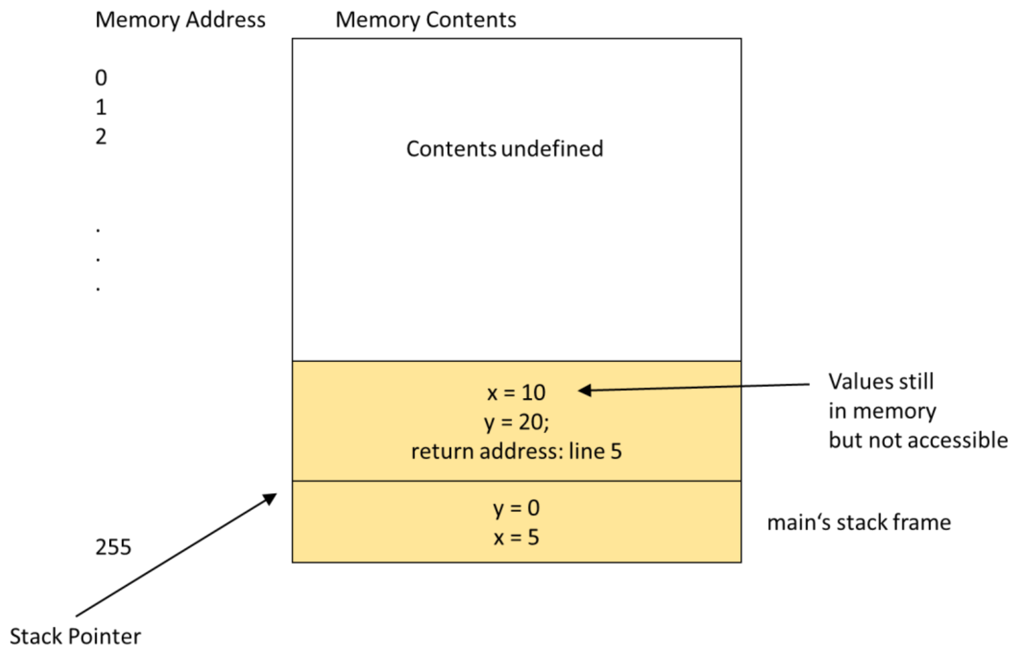


There is other bookkeeping data to keep track of on the stack, but we will skip those details for this class. You will cover it in more detail in computer organization & architecture.

Now, when ChangeValues changes the variables x and y they are changed only in its stack frame. The values in main are unchanged even though they have the same names in the source code.



When the return function is executed, we pop off the stack frame and resume execution at the return address. This has the effect of throwing away the local variables. We are back to the original state of the stack before making the function call:



Note that the old values are still sitting on the stack, but the next time a new stack frame is pushed onto the stack we may overwrite the old values.

The process of using the stack varies from machine to machine and compiler to compiler, but the basic process is the same. Note that each time we call a function, we'll have to allocate some memory off the stack – this will have implications later when we do recursion and make many function calls. (In class example – can you figure out what the stack looks like for a recursive function?) If we make too many calls, we will exhaust the amount of available memory on the stack.

C++ Call By Reference

C++ has a feature called call by reference that doesn't exist in Java. There is also a way to define function parameters so that the contents of variables change between the function and the caller of the function. This is called call by reference. As an example, consider the following changes to the previous program:

```
void  ChangeValues(int &y);                    // Prototype

int main()
{
        int x=5;
        cout << "Before call, x= " << x << endl;
        ChangeValues(x);
        cout << "Back in main x= " << x << endl;
        return 0;
}

void ChangeValues(int &y)
{
        cout << "In ChangeValues, Y=" << y << endl;
        y = 20;
        cout << "In ChangeValues, Y=" << y << endl;
        return;
}
```

This will output:

```
        Before call, x=5
        In ChangeValues, Y= 5
        In ChangeValues, Y=20
        Back in main, x=20
```

By adding a & in the parameter definition for ChangeValues, we now have the property that changes made inside the local function are reflected back in the caller's code. Note that the variable names do not need to be the same; the local variable Y is "bound" to the variable X in main, and changes to Y make changes to X.

Also note that without checking the function prototypes or the function definitions, it is not possible to tell from the calls alone whether a function can modify its arguments. We'll discuss later a way to explicitly use pointers that make it clear when a call is made by value or if a call is made by reference.

How does call by reference work? Briefly, instead of copying the value of X into the stack, instead a pointer containing the memory address of variable X is put on the stack. When accessing this location the computer ends up making changes to the one and only copy of the data, which happens to be referred to by variable X.

In Java it is possible to get a similar effect by passing an object to a function/method.

Scoping

C++ and Java use a type of scoping called **static scope**. It is called static because the scope of identifiers is determined at compile time. In contrast, some languages (like Lisp) use dynamic scope, where the scope is determined at runtime.

There are four categories of scope for an identifier in C++: class scope, local scope, namespace scope, and global scope. Class scope will be discussed a bit later. Local scope is the scope of an identifier defined within a block and extends from the point where the identifier is declared to the end of the block. Global scope is the scope of an identifier declared outside all functions and extends to the end of the file containing the program.

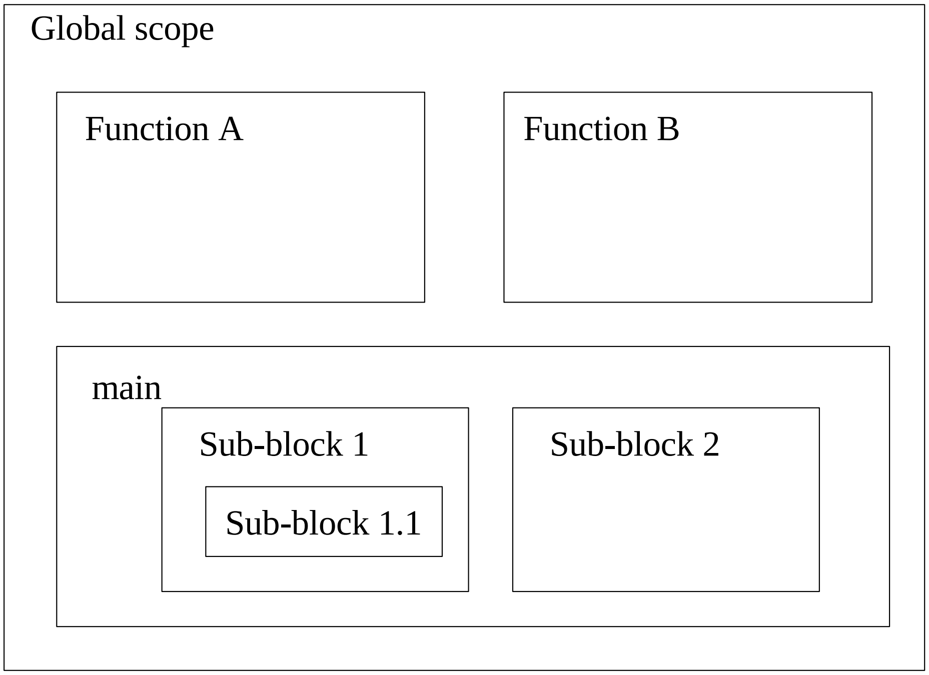
Function names in C++ have global scope. Once a function name has been declared, any subsequent function can call it. In C++ you cannot nest a function definition within another function. When a function defines a local identifier with the same name as a global identifier, the local identifier takes precedence. That is, the local identifier hides the existence of the global identifier. For obvious reasons this principle is called *name precedence* or *name hiding*.

The rules of C++ that govern who knows what, where, and when are called *scope rules*.

- A function name has global scope.
- The scope of a global identifier extends from its declaration to the end of the file in which it is defined.
- The scope of a parameter is the same as the scope of a local variable declared in the outermost block of the function body.
- The scope of a local identifier includes all statements following the declaration of the identifier to the end of the block in which it is declared and includes any nested blocks unless a local identifier of the same name is declared in a nested block.
- The scope of an identifier begins with its most recent declaration.

The last three rules mean that if a local identifier in a block is the same as a global or nonlocal identifier to the block, the local identifier blocks access to the other identifier. That is, the block's reference is to its own local identifier.

The figure below depicts the scope rules for global and local blocks:



Each block has access to identifiers defined
in their own block and to outer blocks

In the figure above, a variable defined in global scope (outside all functions) can be accessed from any function or sub-block, providing there is not a variable of the same name within that block. However, the process doesn’t work the other way. A variable defined in Function A can’t be directly accessed from Function B, for example. If we wanted to access a variable in Function A from Function B, we would have to access it indirectly through the function call. Similarly, we can’t access variables defined in sub-blocks from outside the sub-block. For example, code in Sub-block 1 can’t directly access variables defined in sub-block 1.1.