C/C++ Lab Sessions - COL 106

Last updated: 20 July 2014

This module is targeted to kick-start students in programming C/C++ by introducing them to the basic syntax and useful/essential features of C/C++. This module is by no means complete reference of C/C++ as it is nearly impossible to squeeze a book's worth of material into 2-3 lab session modules.

In imperative programming languages like C, a program is made of a set of functions which are invoked by a main function in some order to perform a task. C++ provides a number of features that spruce up the C language mainly in the object- oriented programming aspects and data type safety (which lack in C language).

Basic structure Let us explore the basic structure of a program in C++.

1. The program starts with what is called preprocessor directive, `#include`.
2. The main program starts with keyword `main()`, each program must have a `main()` function.
3. All the coding is included in the body of the `main()`, within the curly braces { }.

A typical C++ program has the following structure:

Listing 1: Simplest C++ program

```
1 #include <iostream>
2 using namespace std;
3 // main() is where program execution begins.
4 int main( )
5 {
6 cout<<"This is my first C++ program.";
7 // return to operating system, no error
8 return 0;
9 }
```

- The C++ language defines several headers, which contain information that is either necessary or useful to your program. For this program, the header <iostream> is needed.
- The line `using namespace std;` tells the compiler to use the `std` namespace.
- The next line `// main() is where program execution begins.` is a single-line comment available in C++. Single-line comments begin with // and stop at the end of the line.
- The line `int main()` is the main function where program execution begins.
- The next line `cout << "This is my first C++ program.";` causes the message "This is my first C++ program" to be displayed on the screen.
- The next line `return 0;` terminates main()function and causes it to return the value 0 to the calling process.

The `main()` function Method `main` is like entry point for a C++ program. According to the C/C++ standard, only two definitions of `main()` functions are portable:

```
 int main()
{
return 0;
}
```

and

```
int main (int argc, char* argv[])
{
return 0;
}
```

where:

- argc: Is an argument count that is the number of command line arguments the program was invoked with.
- argv[ ]: Is an argument vector that is a pointer to an array of character strings that contain the arguments, one string.

Note that some compilers might generate a warning or error message regarding this thing and some compiler may need to explicitly put the void if there is no return or empty return statement as shown below:

```
 void main(){
 ..
 }
```

Having covered the most essential aspect of a `main()` in C++, now let us look at the process of compiling and executing a C/C++ program.

File naming convention for C/C++ programs

- Any C (C++) program must be saved in a file with extension ".c" (".cpp").
- File names should begin with lower case letters.
- File names should limited up to 30 characters.
- File names and directory paths should not contain blanks, parentheses "(", or crazy characters.
- Smallest non-empty C/C++ file, which is compilable, must have at least the `main()`.

Compiling a C/C++ program The base command for the Gnu C compiler is "gcc". The base command for the Gnu C++ compiler is "g++". To compile a C++ program which is saved in a file, say "first.cpp", execute the following command on shell/command line.

```
 g++ first.cpp
```

This command will create an executable program called "a.out" (the default executable target name when one is not specified).
You can also specify an output filename of your choice.

```
 g++ -o first first.cpp
```

Executing a C/C++ program Once you get an executable file after compiling a C/C++ program, you execute the program with the following command.

```
 ./first
```

where "first" is the name of the output filename you have specified (default, a.out).

Using library functions/helper functions defined elsewhere In **C/C++**, helper functions or library functions (like `printf`, `scanf`, `cout`, textttcin etc.) are used by including appropriate header file with **# include** syntax.

Now we are ready to write, compile and execute our first C++ program which prints a string on command line/ console.

Listing 2: Printing "hello world" on screen

```
1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5     cout << "Hello World!";
6     return 0;
7 }
```

Compile and execute the program of listing 2.

Loops and Condition constructs C++ has three constructs for implementing loops,

- `do{Statements} while(boolean condition);`,
- `while(boolean condition){Statements}`, and
- `for(initialization; terminating condition; post-operation){Statements}`.

Listing 3 shows a program to print all numbers from 10 to 100 using all three looping constructs.

Listing 3: Different looping constructs in C++

```
1  #include<iostream>
2  using namespace std;
3  int main()
4     {
5       int i=10;
6       cout<<"Using do-while";
7       do{
8           cout<<i;
9           i++;
10        }while(i<100);
11
12
13      for(i=10; i< 100; i++){
14          cout<<i;
15        }
16
17      i=10;
18      while(i<100){
19          cout<<i;
20        }
21
22
23
24    }
```

For selection (if then else), C++ provides two constructs.

- `if(condition) {Statements} else {Statements}`
- Switch-case. `break` is used to break the flow of program once a case is satisfied.

An example program to check if a given number is even or odd is implemented using both conditional constructs in listing 4.

Listing 4: Different conditional constructs in C++

```
1 #include<iostream>
```

```
2 using namespace std;
3  int main()
4     {
5       int inp;
6       cin>>inp;
7
8       cout<<"Using if-else construct";
9       if(inp \% 2 == 0)
10            cout<<inp << " is even";
11      else
12           cout<<inp << " is odd";
13
14      cout<<"Using switch case construct";
15      switch(inp \% 2){
16        case 0:
17                cout<<inp << " is even";
18                break;
19        case 1:
20                cout<<inp << " is odd";
21                break;
22      }
23    }
```

Compile and execute the program of listing 4. After that remove the `break` of line 21 and again execute the program. What is the difference?

Write a C++ program to print all prime numbers between 10 and 1000 (including 10 and 1000 both).

Pointers Every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator which denotes an address in memory. Consider the following which will print the address of the variables defined:

Listing 5: Printing addresses of variables

```
1 #include <iostream>
2 using namespace std;
3 int main ()
4 {
5    int  var1;
6    char var2[10];
7
8    cout << "Address of var1 variable: ";
9    cout << &var1 << endl;
10
11   cout << "Address of var2 variable: ";
12   cout << &var2 << endl;
13
14   return 0;
15 }
```

A pointer is a variable whose value is the address of another variable. Like any variable or constant, you must declare a pointer before you can work with it. The general form of a pointer variable declaration is:

```
type *var-name;
```

Here, type is the pointer's base type; it must be a valid C++ type and var-name is the name of the pointer variable. The asterisk you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration:

```
int    *ip;    // pointer to an integer
double *dp;    // pointer to a double
float  *fp;    // pointer to a float
char   *ch     // pointer to character
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

There are few important operations, which we will do with the pointers very frequently. (a) we define a pointer variables (b) assign the address of a variable to a pointer and (c) finally access the value at the address available in the pointer variable. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand. Following example makes use of these operations:

Listing 6: Pointer operations

```
#include <iostream>
using namespace std;
int main ()
{
   int  var = 20;   // actual variable declaration.
   int  *ip;        // pointer variable

   ip = &var;       // store address of var in pointer variable

   cout << "Value of var variable: ";
   cout << var << endl;

   // print the address stored in ip pointer variable
   cout << "Address stored in ip variable: ";
   cout << ip << endl;

   // access the value at the address available in pointer
   cout << "Value of *ip variable: ";
   cout << *ip << endl;

   return 0;
}
```

Compile the code of listing 6 and check the output.

There are four arithmetic operators that can be used on pointers: ++, –, +, -. To understand pointer arithmetic, let us consider that ptr is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmatic operation on the pointer:

```
ptr++;
```

The ptr will point to the location 1004 because each time ptr is incremented, it will point to the next integer. This operation will move the pointer to next memory location without impacting actual value at the memory location. If ptr points to a character whose address is 1000, then above operation will point to the location 1001 because next character will be available at 1001.

Write a C++ program having the various pointer operations.

Pointers can also be compared.

Write a C++ program showing pointer comparisons.

Functions The general form of a C++ function definition is as follows:

```
return_type function_name( parameter list )
{
   body of the function
}
```

A C++ function definition consists of a function header and a function body. Here are all the parts of a function:

- Return Type: A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void.

- Function Name: This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- Parameters: A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- Function Body: The function body contains a collection of statements that define what the function does.

Following is the source code for a function called max(). This function takes two parameters num1 and num2 and returns the maximum between the two:

Listing 7: function returning the max between two numbers

```
1
2 int max(int num1, int num2)
3 {
4    // local variable declaration
5    int result;
6
7    if (num1 > num2)
8       result = num1;
9    else
10      result = num2;
11
12   return result;
13 }
```

Function Declarations: A function declaration tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately. A function declaration has the following parts:

```
return_type function_name( parameter list );
```

For the above defined function max(), following is the function declaration:

```
int max(int num1, int num2);
```

Parameter names are not important in function declaration, only their type is required, so following is also valid declaration:

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

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function. When a program calls a function, program control is transferred to the called function. A called function performs defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program. To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value. For example:

Listing 8: Calling a function

```
1 #include <iostream>
2  using namespace std;
3 // function declaration
4 int max(int num1, int num2);
5
6 int main ()
7 {
8    // local variable declaration:
```

```
    int a = 100;
    int b = 200;
    int ret;

    // calling a function to get max value.
    ret = max(a, b);

    cout << "Max value is : " << ret << endl;

    return 0;
}

// function returning the max between two numbers
int max(int num1, int num2)
{
    // local variable declaration
    int result;

    if (num1 > num2)
       result = num1;
    else
       result = num2;

    return result;
}
```

Compile and execute the program of listing 8.

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, you can use any of the following ways to pass the arguments to the function:

Call by value The call by value method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

By default, C++ uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function. Consider the function swap() definition as follows.

Listing 9: Function definition to swap the values

```

void swap(int x, int y)
{
    int temp;

    temp = x; /* save the value of x */
    x = y;    /* put y into x */
    y = temp; /* put x into y */

    return;
}
```

Now, let us call the function swap() by passing actual values as in the following example:

Listing 10: Call by value example

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

// function declaration
void swap(int x, int y);

int main ()
{
```

```
 9    // local variable declaration:
10    int a = 100;
11    int b = 200;
12
13    cout << "Before swap, value of a :" << a << endl;
14    cout << "Before swap, value of b :" << b << endl;
15
16    // calling a function to swap the values.
17    swap(a, b);
18
19    cout << "After swap, value of a :" << a << endl;
20    cout << "After swap, value of b :" << b << endl;
21
22    return 0;
23 }
```

When the above code is put together in a file, compiled and executed, it produces the following result:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200
```

which shows that there is no change in the values though they had been changed inside the function.

Call by pointer The call by pointer method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

To pass the value by pointer, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function swap(), which exchanges the values of the two integer variables pointed to by its arguments.

Listing 11: Function definition to swap the values

```
1 void swap(int *x, int *y)
2 {
3    int temp;
4    temp = *x; /* save the value at address x */
5    *x = *y; /* put y into x */
6    *y = temp; /* put x into y */
7
8    return;
9 }
```

Let us call the function swap() by passing values by pointer as in the following example:

Listing 12: Program to show pass by pointer method

```
1 #include <iostream>
2 using namespace std;
3
4 // function declaration
5 void swap(int *x, int *y);
6
7 int main ()
8 {
9    // local variable declaration:
10    int a = 100;
11    int b = 200;
12
13    cout << "Before swap, value of a :" << a << endl;
14    cout << "Before swap, value of b :" << b << endl;
15
16    /* calling a function to swap the values.
17     * &a indicates pointer to a ie. address of variable a and
18     * &b indicates pointer to b ie. address of variable b.
```

```
19     */
20    swap(&a, &b);
21
22    cout << "After swap, value of a :" << a << endl;
23    cout << "After swap, value of b :" << b << endl;
24
25    return 0;
26 }
```

When the above code is put together in a file, compiled and executed, it produces the following result:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100
```

Call by reference The call by reference method of passing arguments to a function copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

To pass the value by reference, argument reference is passed to the functions just like any other value. So accordingly you need to declare the function parameters as reference types as in the following function swap(), which exchanges the values of the two integer variables pointed to by its arguments.

Listing 13: Function definition to swap the values

```
1 void swap(int &x, int &y)
2 {
3    int temp;
4    temp = x; /* save the value at address x */
5    x = y;    /* put y into x */
6    y = temp; /* put x into y */
7
8    return;
9 }
```

Let us call the function swap() by passing values by reference as in the following example:

Listing 14: Pass by reference example

```
1 #include <iostream>
2 using namespace std;
3
4 // function declaration
5 void swap(int &x, int &y);
6
7 int main ()
8 {
9    // local variable declaration:
10    int a = 100;
11    int b = 200;
12
13    cout << "Before swap, value of a :" << a << endl;
14    cout << "Before swap, value of b :" << b << endl;
15
16    /* calling a function to swap the values using variable reference.*/
17    swap(a, b);
18
19    cout << "After swap, value of a :" << a << endl;
20    cout << "After swap, value of b :" << b << endl;
21
22    return 0;
23 }
```

When the above code is put together in a file, compiled and executed, it produces the following result:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100
```

Default parameters When you define a function, you can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function. This is done by using the assignment operator and assigning values for the arguments in the function definition. If a value for that parameter is not passed when the function is called, the default given value is used, but if a value is specified, this default value is ignored and the passed value is used instead.

```
int sum(int a, int b=20)
```

Here, if the value of `b` is not passed, it will take a default value of 20.

Write a C++ program where functions have default parameters.

Data Types There are two data types in C/C++:

- Primitive: C++ offer the programmer a rich assortment of built-in as well as user defined data types, some of which you might have already encountered by now. Following table lists down seven basic C++ data types:

Type	**Keyword**
Boolean	bool
Character	char
Integer	int
Floating point	float
Double floating point	double
Valueless	void
Wide character	wchar_t

  Several of the basic types can be modified using one or more of these type modifiers:

  - signed
  - unsigned
  - short
  - long

- User Defined/Derived: C++ offers the following derived data types:

  - `typedef`:
    You can create a new name for an existing type using typedef. Following is the simple syntax to define a new type using `typedef`:

    ```
    typedef type newname;
    ```

    For example, the following tells the compiler that feet is another name for int:

    ```
    typedef int feet;
    ```

    Now, the following declaration is perfectly legal and creates an integer variable called distance:

    ```
    feet distance;
    ```

  - enumerate:
    An enumerated type declares an optional type name and a set of zero or more identifiers that can be used as values of the type. Each enumerator is a constant whose type is the enumeration. To create an enumeration requires the use of the keyword `enum`. The general form of an enumeration type is:

```
enum enum-name { list of names } var-list;
```

Here, the enum-name is the enumeration's type name. The list of names is comma separated. For example, the following code defines an enumeration of colors called colors and the variable c of type color. Finally, c is assigned the value "blue".

```
enum color { red, green, blue } c;
c = blue;
```

By default, the value of the first name is 0, the second name has the value 1, the third has the value 2, and so on. But you can give a name a specific value by adding an initializer. For example, in the following enumeration, green will have the value 5.

```
enum color { red, green=5, blue };
```

Here, blue will have a value of 6 because each name will be one greater than the one that precedes it.

– Arrays:
C++ provides a data structure, the array, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.
Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index. All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

Declaring Arrays: To declare an array in C++, you have to specify the type of the elements and the number of elements required by an array as follows:

```
type arrayName [ arraySize ];
```

This is called a single-dimension array. The arraySize must be an integer constant greater than zero and type can be any valid C++ data type. For example, to declare a 10-element array called balance of type double, use this statement:

```
double balance[10];
```

Initializing Arrays: You can initialize C++ array elements either one by one or using a single statement as follows:

```
double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

The number of values between braces { } can not be larger than the number of elements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array:

```
balance[4]=50.0;
```

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:

```
double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

You will create exactly the same array as you did in the previous example.

Write a C++ program to declare and initialize a 10 size array of integers. Hint: Use loops.

Write a C++ program to declare and initialize a 100 size array of integers. Print all its value.

– Structure:
C/C++ arrays allow you to define variables that combine several data items of the same kind but structure is another user defined data type which allows you to combine data items of different kinds. Structures are used to represent a record, suppose you want to keep track of your books in a library. You might want to track the following attributes about each book:

```
Title
Author
Subject
Book ID
```

Defining a Structure: To define a structure, you must use the `struct` statement. The `struct` statement defines a new data type, with more than one member, for your program. The format of the struct statement is this:

```
struct [structure tag]
{
   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

The structure tag is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure:

```
struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
}book;
```

Accessing Structure Members: To access any member of a structure, we use the member access operator (.). The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use struct keyword to define variables of structure type. Following is the example to explain usage of structure:

Listing 15: Accessing structure members

```
1 #include <iostream>
2 #include <cstring>
3  using namespace std;
4 struct Books
5 {
6    char  title[50];
7    char  author[50];
8    char  subject[100];
9    int   book_id;
10 };
11
12 int main( )
13 {
14    struct Books Book1;        // Declare Book1 of type Book
15    struct Books Book2;        // Declare Book2 of type Book
16
17    // book 1 specification
18    strcpy( Book1.title, "Learn C++ Programming");
19    strcpy( Book1.author, "Chand Miyan");
20    strcpy( Book1.subject, "C++ Programming");
21    Book1.book_id = 6495407;
```

```
22
23    // book 2 specification
24    strcpy( Book2.title, "Telecom Billing");
25    strcpy( Book2.author, "Yakit Singha");
26    strcpy( Book2.subject, "Telecom");
27    Book2.book_id = 6495700;
28
29    // Print Book1 info
30    cout << "Book 1 title : " << Book1.title <<endl;
31    cout << "Book 1 author : " << Book1.author <<endl;
32    cout << "Book 1 subject : " << Book1.subject <<endl;
33    cout << "Book 1 id : " << Book1.book_id <<endl;
34
35    // Print Book2 info
36    cout << "Book 2 title : " << Book2.title <<endl;
37    cout << "Book 2 author : " << Book2.author <<endl;
38    cout << "Book 2 subject : " << Book2.subject <<endl;
39    cout << "Book 2 id : " << Book2.book_id <<endl;
40
41    return 0;
42 }
```

Compile the program in listing 15 and check its output.

Structures as Function Arguments: You can pass a structure as a function argument in very similar way as you pass any other variable or pointer. You would access structure variables in the similar way as you have accessed in the above example:

Listing 16: Structures as function arguments

```
1 #include <iostream>
2 #include <cstring>
3  using namespace std;
4 void printBook( struct Books book );
5
6 struct Books
7 {
8    char  title[50];
9    char  author[50];
10   char  subject[100];
11   int   book_id;
12 };
13
14 int main( )
15 {
16   struct Books Book1;        // Declare Book1 of type Book
17   struct Books Book2;        // Declare Book2 of type Book
18
19   // book 1 specification
20   strcpy( Book1.title, "Learn C++ Programming");
21   strcpy( Book1.author, "Chand Miyan");
22   strcpy( Book1.subject, "C++ Programming");
23   Book1.book_id = 6495407;
24
25   // book 2 specification
26   strcpy( Book2.title, "Telecom Billing");
27   strcpy( Book2.author, "Yakit Singha");
28   strcpy( Book2.subject, "Telecom");
29   Book2.book_id = 6495700;
30
31   // Print Book1 info
32   printBook( Book1 );
33
34   // Print Book2 info
35   printBook( Book2 );
36
37   return 0;
38 }
39 void printBook( struct Books book )
```

```
40 {
41    cout << "Book title : " << book.title <<endl;
42    cout << "Book author : " << book.author <<endl;
43    cout << "Book subject : " << book.subject <<endl;
44    cout << "Book id : " << book.book_id <<endl;
45 }
```

Compile the program in listing 15 and check its output.

Structures can also be used as function arguments. We can also declare pointers to structures. The individual elements are then accessed using $\longrightarrow$ instead of the '.' operator. You can avoid writing `struct <struct-name>` each time by using typedef.

Write a C++ program using structure as a function argument.

Write a C++ program using pointers to structure Books defined in listing 16.

Write a C++ program using typedef to define structure.

Now, let us learn about classes in C++.

1. **Classes and objects**. A class is similar to a C *structure*, except that the definition of the data structure, *and* all of the functions that operate on the data structure are grouped together in one place. An *object* is an instance of a class (an instance of the data structure); objects share the same functions with other objects of the same class, but each object (each instance) has its own copy of the data structure. A class thus defines two aspects of the objects: the *data* they contain, and the *behavior* they have.

2. **Member functions**. These are functions which are considered part of the object and are declared in the class definition. They are often referred to as *methods* of the class. In addition to member functions, a class's behavior is also defined by:

   (a) What to do when you create a new object (the **constructor** for that object) – in other words, initialize the object's data.

   (b) What to do when you delete an object (the **destructor** for that object).

3. **Private vs. public members**. A public member of a class is one that can be read or written by anybody, in the case of a data member, or called by anybody, in the case of a member function. A private member can only be read, written, or called by a member function of that class.

Classes are used for two main reasons: (1) it makes it much easier to organize your programs if you can group together data with the functions that manipulate that data, and (2) the use of private members makes it possible to do *information hiding*, so that you can be more confident about the way information flows in your programs.

Classes C++ classes are similar to C structures in many ways. In fact, a C++ struct is really a class that has only public data members. A class in C++ is written as below.

Listing 17: An empty C++ class.

```
1 class intro
2 {
3
4 }
```

In the following explanation of how classes work, we will use a stack class as an example.

1. **Member functions.** Here is a (partial) example of a class with a member function and some data members:

Listing 18: Example of a class with member functions and data members.

```
1 class Stack {
2   public:
3     void Push(int value); // Push an integer, checking for overflow.
4     int top;           // Index of the top of the stack.
5     int stack[10];     // The elements of the stack.
6 };
7
8 void
9 Stack::Push(int value) {
10    ASSERT(top < 10);           // stack should never overflow
11    stack[top++] = value;
12 }
```

This class has two data members, `top` and `stack`, and one member function, `Push`. The notation *class*::*function* denotes the *function* member of the class *class*. (In the style we use, most function names are capitalized.) The function is defined beneath it. You can also define it inside the class definition.

Note that we use a call to `ASSERT` to check that the stack hasn't overflowed; ASSERT drops into the debugger if the condition is false. It is an extremely good idea for you to use ASSERT statements liberally throughout your code to document assumptions made by your implementation. Better to catch errors automatically via ASSERTs than to let them go by and have your program overwrite random locations.

Inside a member function, one may refer to the members of the class by their names alone. In other words, the class definition creates a scope that includes the member (function and data) definitions.

Note that if you are inside a member function, you can get a pointer to the object you were called on by using the variable `this`. If you want to call another member function on the same object, you do not need to use the `this` pointer, however. Let's extend the Stack example to illustrate this by adding a `Full()` function.

```
class Stack {
  public:
    void Push(int value); // Push an integer, checking for overflow.
    bool Full();       // Returns TRUE if the stack is full, FALSE otherwise.
    int top;          // Index of the lowest unused position.
    int stack[10];    // A pointer to an array that holds the contents.
};
```

```
bool
Stack::Full() {
    return (top == 10);
}
```

Now we can rewrite `Push` this way:

```
void
Stack::Push(int value) {
    ASSERT(!Full());
    stack[top++] = value;
}
```

We could have also written the ASSERT:

```
    ASSERT(!(this->Full());
```

but in a member function, the `this->` is implicit.

The purpose of member functions is to encapsulate the functionality of a type of object along with the data that the object contains. A member function does not take up space in an object of the class.

2. **Private members.** One can declare some members of a class to be *private*, which are hidden to all but the member functions of that class, and some to be *public*, which are visible and accessible to everybody. Both data and function members can be either public or private.

   In our stack example, note that once we have the `Full()` function, we really don't need to look at the `top` or `stack` members outside of the class – in fact, we'd rather that users of the Stack abstraction *not* know about its internal implementation, in case we change it. Thus we can rewrite the class as follows:

```
class Stack {
  public:
    void Push(int value); // Push an integer, checking for overflow.
    bool Full();       // Returns TRUE if the stack is full, FALSE otherwise.
  private:
    int top;           // Index of the top of the stack.
    int stack[10];     // The elements of the stack.
};
```

   Before, given a pointer to a `Stack` object, say `s`, any part of the program could access `s->top`, in potentially bad ways. Now, since the `top` member is private, only a member function, such as `Full()`, can access it. If any other part of the program attempts to use `s->top` the compiler will report an error.

   You can have alternating `public:` and `private:` sections in a class. Before you specify either of these, class members are private, thus the above example could have been written:

```
class Stack {
    int top;           // Index of the top of the stack.
    int stack[10];     // The elements of the stack.
  public:
    void Push(int value); // Push an integer, checking for overflow.
    bool Full();       // Returns TRUE if the stack is full, FALSE otherwise.
};
```

   Which form you prefer is a matter of style, but it's usually best to be explicit, so that it is obvious what is intended.

   What is not a matter of style: **all data members of a class should be private.** All operations on data should be via that class' member functions. Keeping data private adds to the modularity of the system, since you can redefine how the data members are stored without changing how you access them.

3. **Constructors and the operator new.** In C, in order to create a new object of type `Stack`, one might write:

```
struct Stack *s = (struct Stack *) malloc(sizeof (struct Stack));
InitStack(s, 17);
```

The `InitStack()` function might take the second argument as the size of the stack to create, and use `malloc()` again to get an array of 17 integers.

The way this is done in C++ is as follows:

```
Stack *s = new Stack(17);
```

The `new` function takes the place of `malloc()`. To specify how the object should be initialized, one declares a *constructor* function as a member of the class, with the name of the function being the same as the class name:

Listing 19: Constructor Example

```
1 class Stack {
2   public:
3     Stack(int sz);     // Constructor:  initialize variables, allocate space.
4     void Push(int value); // Push an integer, checking for overflow.
5     bool Full();        // Returns TRUE if the stack is full, FALSE otherwise.
6   private:
7     int size;          // The maximum capacity of the stack.
8     int top;           // Index of the lowest unused position.
9     int* stack;        // A pointer to an array that holds the contents.
10 };
11
12 Stack::Stack(int sz) {
13     size = sz;
14     top = 0;
15     stack = new int[size];   // Let's get an array of integers.
16 }
```

There are a few things going on here, so we will describe them one at a time.

The `new` operator automatically creates (i.e. allocates) the object and then calls the constructor function for the new object. This same sequence happens even if, for instance, you declare an object as an automatic variable inside a function or block – the compiler allocates space for the object on the stack, and calls the constructor function on it.

In this example, we create two stacks of different sizes, one by declaring it as an automatic variable, and one by using `new`.

```
void
test() {
    Stack s1(17);
    Stack* s2 = new Stack(23);
}
```

Note there are two ways of providing arguments to constructors: with `new`, you put the argument list after the class name, and with automatic or global variables, you put them after the variable name.

It is crucial that you **always** define a constructor for every class you define, and that the constructor initialize **every** data member of the class. If you don't define your own constructor, the compiler will automatically define one for you, and believe me, it won't do what you want ("the unhelpful compiler"). The data members will be initialized to random, unrepeatable values, and while your program may work anyway, it might not the next time you recompile (or vice versa!).

As with normal C variables, variables declared inside a function are deallocated automatically when the function returns; for example, the `s1` object is deallocated when `test` returns. Data allocated

with `new` (such as `s2`) is stored on the heap, however, and remains after the function returns; heap data must be explicitly disposed of using `delete`, described below.

The `new` operator can also be used to allocate arrays, illustrated above in allocating an array of `ints`, of dimension `size`:

```
stack = new int[size];
```

Note that you can use `new` and `delete` (described below) with built-in types like `int` and `char` as well as with class objects like `Stack`.

4. **Destructors and the operator delete.** Just as `new` is the replacement for `malloc()`, the replacement for `free()` is `delete`. To get rid of the `Stack` object we allocated above with `new`, one can do:

```
delete s2;
```

This will deallocate the object, but first it will call the *destructor* for the `Stack` class, if there is one. This destructor is a member function of `Stack` called `~Stack()`:

Listing 20: Destructor Example

```
1 class Stack {
2   public:
3     Stack(int sz);     // Constructor:  initialize variables, allocate space.
4     ~Stack();          // Destructor:   deallocate space allocated above.
5     void Push(int value); // Push an integer, checking for overflow.
6     bool Full();       // Returns TRUE if the stack is full, FALSE otherwise.
7   private:
8     int size;          // The maximum capacity of the stack.
9     int top;           // Index of the lowest unused position.
10    int* stack;        // A pointer to an array that holds the contents.
11 };
12
13 Stack::~Stack() {
14    delete [] stack;  // delete an array of integers
15 }
```

The destructor has the job of deallocating the data the constructor allocated. Many classes won't need destructors, and some will use them to close files and otherwise clean up after themselves.

The destructor for an object is called when the object is deallocated. If the object was created with `new`, then you must call `delete` on the object, or else the object will continue to occupy space until the program is over – this is called "a memory leak." Memory leaks are bad things – although virtual memory is supposed to be unlimited, you can in fact run out of it – and so you should be careful to **always** delete what you allocate. Of course, it is even worse to call `delete` too early – `delete` calls the destructor and puts the space back on the heap for later re-use. If you are still using the object, you will get random and non-repeatable results that will be very difficult to debug. It is good to always explicitly allocate and deallocate objects with `new` and `delete`, to make it clear when the constructor and destructor is being called. For example, if an object contains another object as a member variable, we use `new` to explicitly allocated and initialize the member variable, instead of implicitly allocating it as part of the containing object. C++ has strange, non-intuitive rules for the order in which the constructors and destructors are called when you implicitly allocate and deallocate objects. In practice, although simpler, explicit allocation is slightly slower and it makes it more likely that you will forget to deallocate an object (a bad thing!), and so some would disagree with this approach.

When you deallocate an array, you have to tell the compiler that you are deallocating an array, as opposed to a single element in the array. Hence to delete the array of integers in `Stack::~Stack`:

```
delete [] stack;
```

We can modify the program of listing 17 as following

- Create a public function in `class intro` which prints *Hello world* on screen.
- Create an object of this class in function `main` and invoke/call the function defined above on this object.

Along the same line,

Complete the program of listing 21 (in place of $\cdots$) such that it prints "Hello world" on console.

Listing 21: Modified hello world program

```
1  #include<iostream>
2  using namespace std;
3  int main(){
4  ...
5  }
6
7  public class intro
8  {
9   public void printinfo()
10  {
11       cout<<"Hello world";
12  }
13 }
```

Reading from console To read from command line we use the function `cin` which is defined in the header file iostream. Following program takes one string (name) and an integer (age) from the command line and store it in two fields.

Listing 22: Reading from command line

```
1 #include<iostream>
2 using namespace std;
3 int main()
4     {
5       intro obj = new intro();
6
7       cout<<"Enter your name";
8       cin>>bj.name;
9       cout<<"Enter your age ";
10      cin>>obj.age;
11      cout<<"Name: " << obj.name;
12      cout<<"Age: " << obj.age;
13    }
14 public class intro
15 {
16  public String name;
17  public int age;
18 }
```

Compile the code of listing 22 and check the output.

Write a C++ program which takes two integer numbers as input and returns the minimum of these two.

File handling A file must be opened before you can read from it or write to it. Either the ofstream or fstream object may be used to open a file for writing and ifstream object is used to open a file for reading purpose only. When a C++ program terminates it automatically closes flushes all the streams, release all the allocated memory and close all the opened files. But it is always a good practice that a programmer should close all the opened files before program termination.

Listing 23: File I/O

```
#include <fstream>
#include <iostream>
 using namespace std;
int main ()
{

   char data[100];

   // open a file in write mode.
   ofstream outfile;
   outfile.open("afile.dat");

   cout << "Writing to the file" << endl;
   cout << "Enter your name: ";
   cin.getline(data, 100);

   // write inputted data into the file.
   outfile << data << endl;

   cout << "Enter your age: ";
   cin >> data;
   cin.ignore();

   // again write inputted data into the file.
   outfile << data << endl;

   // close the opened file.
   outfile.close();

   // open a file in read mode.
   ifstream infile;
   infile.open("afile.dat");

   cout << "Reading from the file" << endl;
   infile >> data;

   // write the data at the screen.
   cout << data << endl;

   // again read the data from the file and display it.
   infile >> data;
   cout << data << endl;

   // close the opened file.
   infile.close();

   return 0;
}
```

- While doing C++ programming, you write information to a file from your program using the stream insertion operator (<<) just as you use that operator to output information to the screen. The only difference is that you use an ofstream or fstream object instead of the cout object.
- You read information from a file into your program using the stream extraction operator (>>) just as you use that operator to input information from the keyboard. The only difference is that you use an ifstream or fstream object instead of the cin object.

Compile and execute the program given in listing 23.

You can also seek for a defined location within a file. Following example illustrates it.

Listing 24: Seeking within a file

```
#include<iostream>
#include<fstream>

using namespace std;

```

```
6 int main()
7 {
8         fstream f;
9
10        //open file "abc.txt" for reading and writing
11        f.open("abc.txt", ios::in | ios::out | ios::trunc);
12 // ios::trunc is to be used if the file does not already exist
13 // if file exists, its contents will be deleted
14
15        if(f.is_open()) //if file open succeeded
16        {
17                // write to file
18                f << "File Handling Program" << endl;
19                f << "Hello World" << endl;
20
21                // move read pointer to start of file
22                f.seekg(0, ios::beg);
23
24                // read entire file line-by-line
25                string line;
26                cout<<"File Contents are : "<< endl;
27                while( getline(f, line) )
28                {
29                        cout << line << endl;
30                }
31
32                f.close();
33                f.open("abc.txt", ios::in | ios::out);
34 // file has to be re-opened again as reading the file till
35 // the end causes the fstream object 'f' to be invalidated
36
37                //move write pointer to position 28
38                cout << endl << "before seek : ";
39                cout << "write pointer is at : " << f.tellp() << endl;
40                f.seekp(28, ios::beg);
41                cout << "after seek : ";
42                cout << "write pointer is at : " << f.tellp() << endl;
43                cout << endl;
44
45                //overwrite contents
46                f << "My World";
47
48                f.seekg(0, ios::beg);
49
50                cout<<"File Contents are : "<< endl;
51                while( getline(f, line) )
52                {
53                        cout << line << endl;
54                }
55
56                f.close();
57        }
58        else
59        {
60                perror("file open error");
61        }
62
63        return 0;
64 }
```

Compile and execute the program given in listing 24.

Please refer `http://www.cplusplus.com/doc/tutorial/files/` for a more thorough treatment on file handling.

Qualifiers

- **const** : you can use *const* to declare that a variable is never modified. This is apt for constants like the maximum size of structures. To define constants to be used in multiple source files, it is not

prudent to place definitions in all source files. Instead, place the constant definition in a header file, and include this file in each of the source files.

```
const int MaxHashTableSize = 8;
```

- **volatile** : The volatile type qualifier declares an item whose value can legitimately be changed by something beyond the control of the program in which it appears, such as a concurrently executing thread. Using the volatile qualifier is a popular way to direct the compiler to not optimize code that works with the particular variable.

```
volatile int shared_key = 5;
```

- **extern** : This qualifier is used to indicate that the variable is declared in another file, and used in this one. To declare a variable to be used in multiple files, this comes in handy.

```
//in file 1.cpp
int shared_variable;
//in files 2.cpp, 3.cpp, 4.cpp
extern int shared_variable;
```

- **static** : The static qualifier is used to indicate that the qualified entity has a lifetime equal to the lifetime of the program. The following examples will explain this further.

Listing 25: `static` qualifier example

```
1 void func1()
2 {
3     int counter = 0;
4     cout << counter++;
5 }
6
7 void func2()
8 {
9     static int counter = 0;
10    cout << counter++;
11 }
12
13 void main()
14 {
15    cout << "\\nfunc1\\n";
16    for(int i = 0; i < 5; i++)
17        func1();
18
19    cout << "\\nfunc2\\n";
20    for(int i = 0; i < 5; i++)
21        func2();
22 }
```

Listing 26: Another `static` qualifier example

```
1 class test {
2     static int i; //declaration
3 }
4
5 int test::i = 1;  //definition -- only allowed once,
6                   //available for the rest of the execution
7
8 void main()
9 {
10    test obj1, obj2;
11    cout << "obj1.i = " << obj1.i << endl;
12    cout << "obj2.i = " << obj2.i << endl;
13 }
14
15 //output
16 obj1.i = 1
17 obj2.i = 1
```

Compile and execute the program given in listings 25 and 26.

Inheritance Inheritance captures the idea that certain classes of objects are related to each other in useful ways. For example, lists and sorted lists have quite similar behavior – they both allow the user to insert, delete, and find elements that are on the list. There are two benefits to using inheritance:

1. **Shared Behavior** : You can write generic code that doesn't care exactly which kind of object it is manipulating. For example, inheritance is widely used in windowing systems. Everything on the screen (windows, scroll bars, titles, icons) is its own object, but they all share a set of member functions in common, such as a routine `Repaint` to redraw the object onto the screen. This way, the code to repaint the entire screen can simply call the `Repaint` function on every object on the screen. The code that calls `Repaint` doesn't need to know which kinds of objects are on the screen, as long as each implements `Repaint`.

2. **Shared Implementation** : You can share pieces of an implementation between two objects. For example, if you were to implement both lists and sorted lists in C, you'd probably find yourself repeating code in both places – in fact, you might be really tempted to only implement sorted lists, so that you only had to debug one version. Inheritance provides a way to re-use code between nearly similar classes. For example, given an implementation of a list class, in C++ you can implement sorted lists by replacing the insert member function – the other functions, delete, isFull, print, all remain the same.

Shared Behavior

A Stack can be implemented with an array or a linked list. Any code using a Stack shouldn't care which implementation is being used.

To allow the two implementations to coexist, we first define an *abstract* Stack, containing just the public member functions, but no data.

Listing 27: Shared behaviour example

```
1 class Stack {
2   public:
3     Stack();
4     virtual ~Stack();           // deallocate the stack
5     virtual void Push(int value) = 0;
6                                 // Push an integer, checking for overflow.
7     virtual bool Full() = 0;    // Is the stack is full?
8 };
9
10 // For g++, need these even though no data to initialize.
11 Stack::Stack {}
12 Stack::~Stack() {}
```

The `Stack` definition is called a *base class* or sometimes a *superclass*. We can then define two different *derived classes*, sometimes called *subclasses* which inherit behavior from the base class. (Of course, inheritance is recursive – a derived class can in turn be a base class for yet another derived class, and so on.) Note that I have prepended the functions in the base class is prepended with the keyword `virtual`, to signify that they *CAN* be redefined by each of the two derived classes. The virtual functions are initialized to zero, to tell the compiler that those functions *MUST* be defined by the derived classes. Such virtual functions are commonly called "pure virtual functions".

Here's how we could declare the array-based and list-based implementations of `Stack`. The syntax `: public Stack` signifies that both `ArrayStack` and `ListStack` are kinds of `Stacks`, and share the same behavior as the base class.

Listing 28: Inheritance example

```
1 class ArrayStack : public Stack {  // the same as in Section 2
2   public:
3     ArrayStack(int sz);   // Constructor:  initialize variables, allocate space.
4     ~ArrayStack();        // Destructor:   deallocate space allocated above.
```

```
5     void Push(int value); // Push an integer, checking for overflow.
6     bool Full();      // Returns TRUE if the stack is full, FALSE otherwise.
7   private:
8     int size;         // The maximum capacity of the stack.
9     int top;          // Index of the lowest unused position.
10    int *stack;       // A pointer to an array that holds the contents.
11 };
12
13 class ListStack : public Stack {
14   public:
15    ListStack();
16    ~ListStack();
17    void Push(int value);
18    bool Full();
19  private:
20    List *list;          // list of items pushed on the stack
21 };
22
23 //function definitions not shown for the sake of brevity
```

The neat concept here is that we can assign pointers to instances of `ListStack` or `ArrayStack` to a variable of type `Stack`, and then use them as if they were of the base type.

Listing 29: Usage of Derived classes

```
1     Stack *s1 = new ListStack;
2     Stack *s2 = new ArrayStack(17);
3
4     if (!stack->Full())
5         s1->Push(5);
6     if (!s2->Full())
7         s2->Push(6);
8
9     delete s1;
10    delete s2;
```

The compiler automatically invokes `ListStack` operations for `s1`, and `ArrayStack` operations for `s2`; this is done by creating a procedure table for each object, where derived objects override the default entries in the table defined by the base class. For the code above, it invokes the operations `Full`, `Push`, and `delete` by indirection through the procedure table, so that the code doesn't need to know which kind of object it is.

In this example, since an instance of the abstract class `Stack` is never created, there is no need to *implement* its functions. This might seem a bit strange, but remember that the derived classes are the various implementations of Stack, and Stack serves only to reflect the shared behavior between the different implementations.

Also note that the destructor for `Stack` is a virtual function but the constructor is not. Clearly, when creating an object, we have to know which kind of object it is, whether `ArrayStack` or `ListStack`. The compiler makes sure that no one creates an instance of the abstract `Stack` by mistake – you cannot instantiate any class whose virtual functions are not completely defined (in other words, pure virtual functions).

But when deallocating an object, I may no longer know its exact type. In the above code, we want to call the destructor for the derived object, even though the code only knows that an object of class `Stack` is being deleted. If the destructor were not virtual, then the compiler would invoke `Stack`'s destructor, which is not at all what we want. This is an easy mistake to make – if you don't define a destructor for the abstract class, the compiler will define one for you implicitly (and by the way, it won't be virtual, since you have a *really* unhelpful compiler). The result for the above code would be a memory leak, and who knows how you would figure that out!

Shared Implementation

What about sharing code, the other reason for inheritance? In C++, it is possible to use member functions of a base class in its derived class.

Suppose that we wanted to add a new member function, `NumberPushed()`, to both implementations of `Stack`. One alternative is to have a counter in both the `ArrayStack` class and the `ListStack` class,

and update this as and when elements are pushed or popped. The other alternative (and the better one) is to place the counter, and update functions, and the `NumberPushed()` function in the base class. Objects of any derived class use these definitons when invoked.

Listing 30: Modified Stack class

```
1 class Stack {
2   public:
3     virtual ~Stack();           // deallocate data
4     virtual void Push(int value); // Push an integer, checking for overflow.
5     virtual bool Full() = 0;    // return TRUE if full
6     int NumPushed();            // how many are currently on the stack?
7   protected:
8     Stack();                    // initialize data
9   private:
10    int numPushed;
11 };
12
13 Stack::Stack() {
14    numPushed = 0;
15 }
16
17 void Stack::Push(int value) {
18    numPushed++;
19 }
20
21 int Stack::NumPushed() {
22    return numPushed;
23 }
```

We can then modify both `ArrayStack` and `ListStack` to make use the new behavior of `Stack`. Listing one of them :

Listing 31: Overloaded functions example

```
1 class ArrayStack : public Stack {
2   public:
3     ArrayStack(int sz);
4     ~ArrayStack();
5     void Push(int value);
6     bool Full();
7   private:
8     int size;         // The maximum capacity of the stack.
9     int *stack;       // A pointer to an array that holds the contents.
10 };
11
12 ArrayStack::ArrayStack(int sz) : Stack() {
13    size = sz;
14    stack = new int[size];   // Let's get an array of integers.
15 }
16
17 void
18 ArrayStack::Push(int value) {
19    ASSERT(!Full());
20    stack[NumPushed()] = value;
21    Stack::Push();           // invoke base class to increment numPushed
22 }
```

There are a few things to note:

1. The constructor for `ArrayStack` needs to invoke the constructor for `Stack`, in order to initialize `numPushed`. It does that by adding `: Stack()` to the first line in the constructor:

   ```
   ArrayStack::ArrayStack(int sz) : Stack()
   ```

2. A new keyword, `protected`, has been introduced in the new definition of `Stack`. For a base class, `protected` signifies that those member data and functions are accessible to classes derived (recursively) from this class, but inaccessible to other classes. In other words, protected data is

`public` to derived classes, and `private` to everyone else. For example, we need `Stack`'s constructor to be callable by `ArrayStack` and `ListStack`, but we don't want anyone else to create instances of `Stack`. Hence, we make `Stack`'s constructor a protected function.

Note however that `Stack`'s data member is `private`, not `protected`. Although there is some debate on this point, as a rule of thumb you should never allow one class to see directly access the data in another, even among classes related by inheritance. Otherwise, if you ever change the implementation of the base class, you will have to examine and change all the implementations of the derived classes, violating modularity.

3. The interface for a derived class automatically includes all functions defined for its base class, without having to explicitly list them in the derived class. Although we didn't define `NumPushed()` in `ArrayStack`, we can still call it for those objects:

```
ArrayStack *s = new ArrayStack(17);
ASSERT(s->NumPushed() == 0); // should be initialized to 0
```

4. Conversely, even though we have defined a routine `Stack::Push()`, because it is declared as `virtual`, if we invoke `Push()` on an `ArrayStack` object, we will get `ArrayStack`'s version of `Push`:

```
Stack *s = new ArrayStack(17);
if (!s->Full()) // ArrayStack::Full
    s->Push(5); // ArrayStack::Push
```

5. `Stack::NumPushed()` is not `virtual`. That means that it cannot be re-defined by `Stack`'s derived classes.

6. Member functions in a derived class can explicitly invoke public or protected functions in the base class, by the full name of the function, `Base::Function()`, as in:

```
void ArrayStack::Push(int value)
{
    ...
    Stack::Push();      // invoke base class to increment numPushed
}
```

Of course, if we just called `Push()` here (without prepending `Stack::`, the compiler would think we were referring to `ArrayStack`'s `Push()`, and so that would recurse, which is not exactly what we had in mind here.

Templates Templates are another useful concept in C++. With templates, you can parameterize a class definition with a *type*, to allow you to write generic type-independent code. For example, our `Stack` implementation above only worked for pushing and popping *integers*; what if we wanted a stack of characters, or floats, or pointers, or some arbitrary data structure?

In C++, this is pretty easy to do using templates:

Listing 32: Templates' application example

```
template <class T>
class Stack {
  public:
    Stack(int sz);     // Constructor:  initialize variables, allocate space.
    ~Stack();          // Destructor:   deallocate space allocated above.
    void Push(T value); // Push an integer, checking for overflow.
    bool Full();       // Returns TRUE if the stack is full, FALSE otherwise.
  private:
    int size;          // The maximum capacity of the stack.
    int top;           // Index of the lowest unused position.
    T *stack;        // A pointer to an array that holds the contents.
};
```

To define a template, we prepend the keyword `template` to the class definition, and we put the parameterized type for the template in angle brackets. If we need to parameterize the implementation with two or more types, it works just like an argument list: `template <class T, class S>`. We can use the type parameters elsewhere in the definition, just like they were normal types.

When we provide the implementation for each of the member functions in the class, we also have to declare them as templates, and again, once we do that, we can use the type parameters just like normal types:

Listing 33: Template version of Stack::Stack

```
template <class T>
Stack<T>::Stack(int sz) {
    size = sz;
    top = 0;
    stack = new T[size];   // Let's get an array of type T
}

     // template version of Stack::Push
template <class T>
void
Stack<T>::Push(T value) {
    ASSERT(!Full());
    stack[top++] = value;
}
```

Creating an object of a template class is similar to creating a normal object:

Listing 34: Creating objects of template classes

```
void
test() {
    Stack<int> s1(17);
    Stack<char> *s2 = new Stack<char>(23);

    s1.Push(5);
    s2->Push('z');
    delete s2;
}
```

Everything operates as if we defined two classes, one called `Stack<int>` – a stack of integers, and one called `Stack<char>` – a stack of characters. `s1` behaves just like an instance of the first; `s2` behaves just like an instance of the second. In fact, that is exactly how templates are typically implemented – you get a complete *copy* of the code for the template for each different instantiated type. In the above example, we'd get one copy of the code for `ints` and one copy for `chars`.

Container Classes in the C++ Standard Library Optimized implementations of various container classes are available in the C++ Standard Library. A rich, clean API to query and update the different structures are provided. A brief overview will be given here.

- **Array** : Represents a clean interface to an array data structure.
- **Forward List** : Represents a Singly-Linked List
- **List** : Represents a Doubly-Linked List
- **Vector** : It is similar to an array in that it refers to a linear collection of data, located in contiguous memory locations. The additional feature that a vector offers is the provision to "grow" or "shrink" the structure dynamically. Its capacity, or the maximum number of elements it can hold, can be dynamically changed.
- **Stack** : Represents a LIFO (Last-In-First-Out) stack. The underlying container (a stack can be based on an array, list, etc.) can be specified.
- **Queue** : Represents a FIFO (First-In-First-Out) queue. The underlying container (a queue can be based on an array, list, etc.) can be specified.

- **Priority Queue** : It is a data structure that provides efficient querying of the smallest (or largest, or any other pre-defined metric) element in the collection.
- **Set** : Ordered collection of elements. Searching in a set is efficient.
- **Map** : Represents a hash table – a collection of <key>:<value> pairs. Elements ordered by key.

Debugging This section will explain how to use the popular "Gnu Debugger" (GDB).

- Compile the program to include debugging symbols. This is done as :

  ```
  g++ -ggdb hello.cpp
  ```

- Load the program :

  ```
  gdb a.out
  ```

- Run the program :

  ```
  run [arguments]
  ```

  If your program takes any command line arguments, specify them in the run command.

- If the program crashes, or is suspended at a breakpoint (see Breakpoints ahead), `gdb` tells us at which line of the program the control currently is.
  - To know the current value of some variable at this point, use print.

    ```
    print counter
    // OR
    p counter
    ```

    This displays the value of `counter` at this point.
  - A low-level alternative to print is the examine command which displays the value at a memory address.

    ```
    examine 0x12341234
    // OR
    x 0x12341234
    ```

    This displays the value of the word at $0x12341234$. The number of bytes to consider and the format of printing can be configured.
  - To know the stack trace of functions that were called to reach this point, use backtrace.

    ```
    backtrace
    // OR
    bt
    ```

- Breakpoints : A breakpoint is an instruction to the debugger to suspend execution at a particular line number, or at a particular function. These can be registered at any point during the debugging.

  ```
  break MyClass::MyFunc
  ```

  This will cause the execution to suspend when the function `MyFunc` of class `MyClass` is called.

- When the execution is suspended, we have three options :
  - Continue normally until the next breakpoint, crash or the end of the program.

    ```
    continue
    // OR
    c
    ```
  - Execute a single instruction and suspend again.

```
step
// OR
s
```

– Execute the current function and suspend again.

```
next
// OR
n
```

- To exit gdb,

```
quit
```

For a more detailed tutorial on GDB, refer `http://www.delorie.com/gnu/docs/gdb/gdb.html#SEC_Top`.

Common errors that cause a program to crash include accessing an array out of its bounds

```
int a[5];
a[6] = 10;
```

and dereferencing a pointer before setting it.

```
int a;
int * p;
a = *p;
```

This precisely completes the broad overview/tutorial of C++ features which you might require during your assignments. In case of any difficulty in C++, internet (googling) is your best friend.

References

- "A Quick Introduction to C++", homes.cs.washington.edu/~tom/c++example/
- "Introduction to C and C++", http://www.tenouk.com/Module1.html
- "Debugging Under Unix: gdb Tutorial", http://www.cs.cmu.edu/~gilpin/tutorial/
- "Debugging with GDB", http://www.delorie.com/gnu/docs/gdb/gdb.html#SEC_Top
- "Tutorials Point", http://www.tutorialspoint.com/cplusplus/
- "MSDN Library", http://msdn.microsoft.com/en-us/library/dtefa218.aspx
- http://www.cplusplus.com
- http://www.studytonight.com/cpp/