./

**Learning Report –**

**Intermediate C++**



Submitted by: Nilanjana Das

PS No. 99003748

Version Number:

Team Members :

Team No:

Module: Model Based System Engineering

**Document History**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Ver. Rel. No.** | **Release Date** | **Prepared. By** | **Reviewed By** | **Approved By** | **Remarks/Revision Details** |
|  | 18-3-2021 | Nilanjana Das |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

# 

Contents

[1. Compilation stages 4](#_Toc69077347)

[1.1. **Pre-processing:** 4](#_Toc69077348)

[1.2. **Complier**: 4](#_Toc69077349)

[1.3. **Assembler**: 4](#_Toc69077350)

[1.4. **Linking:** 4](#_Toc69077351)

[2. Memory layout of a C++ program 5](#_Toc69077352)

[2.1. **Text or Code Segment:** 5](#_Toc69077353)

[2.2. **Initialized Data Segments:** 5](#_Toc69077354)

[2.3. **Uninitialized Data Segments (bss):** 6](#_Toc69077355)

[2.4. **Stack Segment:** 6](#_Toc69077356)

[2.5. **Heap Segment:** 6](#_Toc69077357)

[2.6. **Unmapped or Reserved Segment:** 6](#_Toc69077358)

[3. Operator overloading 7](#_Toc69077359)

[4. Function Overloading in C++ 8](#_Toc69077360)

[5. Templates 9](#_Toc69077361)

[6. Friend Function in C++: 10](#_Toc69077362)

[ACTIVITY 1: Program on Use of template to perform calculator operations, complex function, operation on fractions 11](#_Toc69077363)

[Activity 2: SOLOLEARN ON C++ 11](#_Toc69077364)

# Compilation stages

## **Pre-processing:**

The source code is the code which is written in a text editor and the source code file is given an extension ".c". This source code is first passed to the preprocessor, and then the preprocessor expands this code. After expanding the code, the expanded code is passed to the compiler.

## **Complier**:

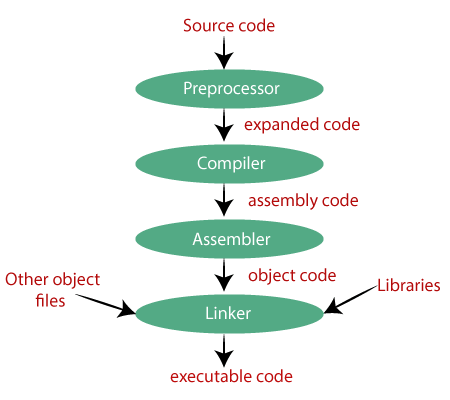
The code which is expanded by the preprocessor is passed to the compiler. The compiler converts this code into assembly code. Or we can say that the C compiler converts the pre-processed code into assembly code.

## **Assembler**:

The assembly code is converted into object code by using an assembler. The name of the object file generated by the assembler is the same as the source file. The extension of the object file in DOS is '.obj,' and in UNIX, the extension is 'o'. If the name of the source file is 'hello.c', then the name of the object file would be 'hello.obj'.

* 1. **Linking:**

Mainly, all the programs written in C use library functions. These library functions are pre-compiled, and the object code of these library files is stored with '.lib' (or ‘. a') extension. The main working of the linker is to combine the object code of library files with the object code of our program. Sometimes the situation arises when our program refers to the functions defined in other files; then linker plays a very important role in this. It links the object code of these files to our program. Therefore, we conclude that the job of the linker is to link the object code of our program with the object code of the library files and other files. The output of the linker is the executable file. The name of the executable file is the same as the source file but differs only in their extensions. In DOS, the extension of the executable file is '.exe', and in UNIX, the executable file can be named as 'a.out'. For example, if we are using printf () function in a program, then the linker adds its associated code in an output file.



**Figure 1: Stages of compilation**

# Memory layout of a C++ program

This memory layout is organized in following fashion:

## **Text or Code Segment:**

Code segment, also known as text segment contains machine code of the compiled program. The text segment of an executable object file is often read-only segment that prevents a program from being accidentally modified. It will be .bin or .exe or. hex etc.

## **Initialized Data Segments:**

Data segment stores program data. This data could be in form of initialized or uninitialized variables, and it could be local or global. Data segment is further divided into four sub-data segments (initialized data segment, uninitialized or. bss data segment, stack, and heap) to store variables depending upon if they are local or global, and initialized or uninitialized.

## **Uninitialized Data Segments (bss):**

Uninitialized data segment is also called BSS segment. BSS stands for ‘Block Started by Symbol’ named after an ancient assembler operator. Uninitialized data segment contains all global and static variables that are initialized to zero or do not have explicit initialization in source code.

## **Stack Segment:**

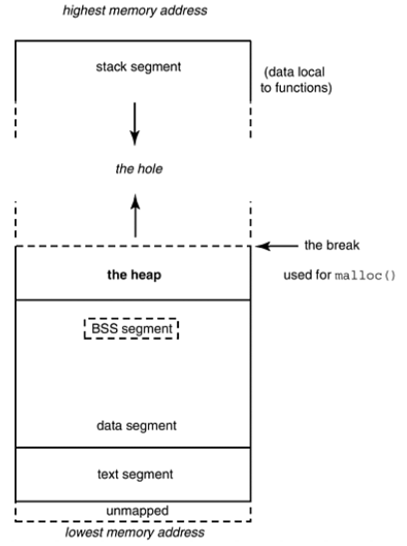
Stack, where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller’s environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables. This is how recursive functions in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn’t interfere with the variables from another instance of the function. So Stack frame contain some data like return address, arguments passed to it, local variables, and any other information needed by the invoked function.

## **Heap Segment:**

Heap is the segment where dynamic memory allocation usually takes place. The heap area begins at the end of the BSS segment and grows to larger addresses from there. The Heap area is managed by malloc, realloc, and free, which may use the brk and sbrk system calls to adjust its size (note that the use of brk/sbrk and a single “heap area” is not required to fulfill the contract of malloc/realloc/free; they may also be implemented using mmap to reserve potentially non-contiguous regions of virtual memory into the process’ virtual address space). The Heap area is shared by all shared libraries and dynamically loaded modules in a process.

## **Unmapped or Reserved Segment:**

Unmapped or reserved segment contain command line arguments and other program related data like lower address-higher address of executable image, etc.



**Figure 2: memory map of a C++ program**

# Operator overloading

In C++, we can make operators to work for user defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. For example, we can overload an operator ‘+’ in a class like String so that we can concatenate two strings by just using +. Other example classes where arithmetic operators may be overloaded are Complex Number, Fractional Number, Big Integer, etc.

# Function Overloading in C++

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

#include <iostream>

using namespace std;

class printData {

public:

void print(int i) {

cout << "Printing int: " << i << endl;

}

void print(double f) {

cout << "Printing float: " << f << endl;

}

void print(char\* c) {

cout << "Printing character: " << c << endl;

}

};

int main(void) {

printData pd;

// Call print to print integer

pd.print(5);

// Call print to print float

pd.print(500.263);

// Call print to print character

pd.print("Hello C++");

return 0;

}

# Templates

Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.

A template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.

There is a single definition of each container, such as **vector**, but we can define many different kinds of vectors for example, **vector <int>** or **vector <string>**.

**Syntax:**

template <class type> ret-type func-name (parameter list) {

// body of function

}

**Example:**

#include <iostream>

#include <string>

using namespace std;

template <typename T>

inline T const& Max (T const& a, T const& b) {

return a < b ? b:a;

}

int main () {

int i = 39;

int j = 20;

cout << "Max(i, j): " << Max(i, j) << endl;

double f1 = 13.5;

double f2 = 20.7;

cout << "Max(f1, f2): " << Max(f1, f2) << endl;

string s1 = "Hello";

string s2 = "World";

cout << "Max(s1, s2): " << Max(s1, s2) << endl;

return 0;

}

# Friend Function in C++:

A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.

**Synatax:**

friend class ClassTwo;

Sample program:

#include <iostream>

using namespace std;

class Box {

double width;

public:

friend void printWidth( Box box );

void setWidth( double wid );

};

// Member function definition

void Box::setWidth( double wid ) {

width = wid;

}

// Note: printWidth() is not a member function of any class.

void printWidth( Box box ) {

/\* Because printWidth() is a friend of Box, it can

directly access any member of this class \*/

cout << "Width of box : " << box.width <<endl;

}

// Main function for the program

int main() {

Box box;

// set box width without member function

box.setWidth(10.0);

// Use friend function to print the wdith.

printWidth( box );

return 0;

}

# ACTIVITY 1: Program on Use of template to perform calculator operations, complex function, operation on fractions

# Activity 2: SOLOLEARN ON C++



**Figure 3: Sololearn C++ certificate**