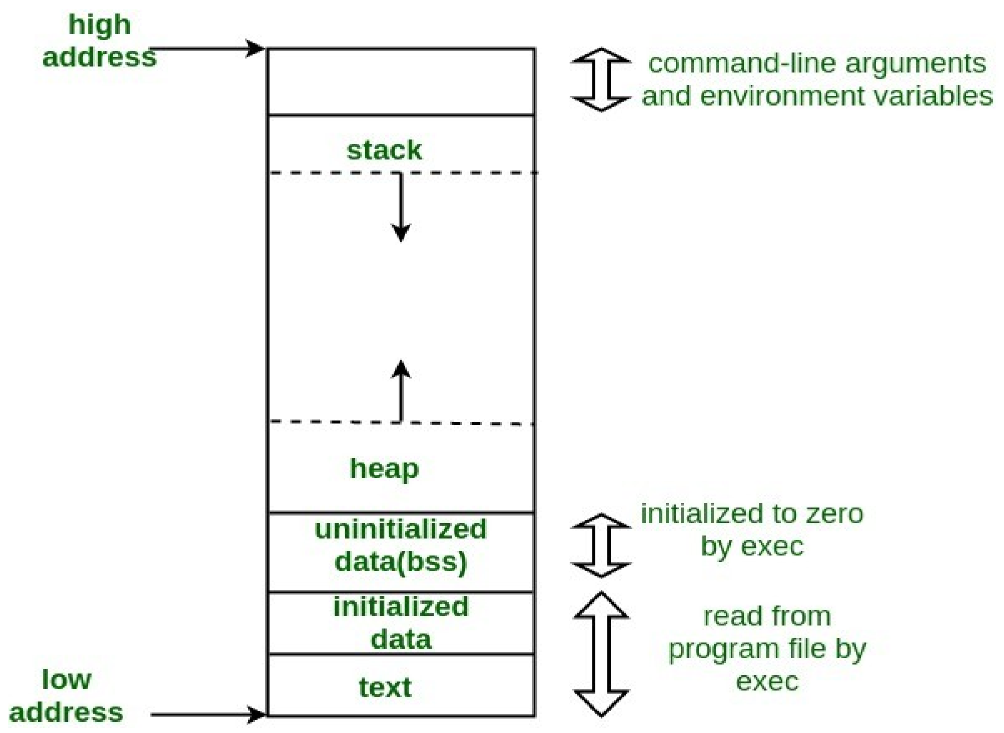
1. Memory Segments in C/C++


high address
command-line arguments and environment variables
stack
heap
uninitialized data(bss)
initialized to zero by exec
initialized data
read from program file by exec
text
low address

1. Text segment : Code segment or simply as text, is one of the sections of a program in an object file or in memory, which contains executable instructions. Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.

2. Initialized data segment (Data Segment): Global variables and static variables that are initialized by the programmer. This segment can be further classified into initialized read-only area and initialized read-write area.

Ex: **`static int i = 10`** will be stored in data segment and **`global int i = 10`** will also be stored in data segment

3. Uninitialized data segment (bss Segment) :Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing

For instance a variable declared **`static int i`**; would be contained in the BSS segment.
For instance a global variable declared **`int j`**; would be contained in the BSS segment.

bss => block started by symbol

4. Stack : The stack area contains the program stack, a LIFO structure, typically located in the higher parts of memory. A “stack pointer” register tracks the top of the stack; it is adjusted each time a value is “pushed” onto the stack. The set of values pushed for one function call is termed a “stack frame”; A stack frame consists at minimum of a return address.

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.

5. Heap : 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.

---

2. Compiling a C program : Behind the Scenes

```
$ vi filename.c
$ gcc –Wall filename.c –o filename
```

Option -Wall enables all compiler's warning, -o is used to specify output file name

```
$ ./filename
```

To execute program

Compiler converts a C program into an executable. There are four phases for a C program to become an executable:

```
$gcc –Wall –save-temps filename.c –o filename
```

We get the all intermediate files in the current directory along with the executable

"PCAL" => "ISOBinaryExe"

Pre-processing => Output : **filename.i**

Removal of Comments
Expansion of Macros
Expansion of the included files.
The preprocessed output is stored in the **filename.i**

Compilation => Output : **filename.s**

The next step is to compile filename.i and produce an; intermediate compiled output file filename.s . This file is in assembly level instructions.

Assembly => Output : **filename.o**

In this phase the filename.s is taken as input and turned into filename.o by assembler.
This file contain machine level instructions.
At this phase, only existing code is converted into machine language, the function calls like printf() are not resolved.

Linking => Output : **filename**

This is the final phase in which all the linking of function calls with their definitions are done. Linker knows where all these functions are implemented.
Linker does some extra work also, it adds some extra code to our program which is required when the program starts and ends.

For example, there is a code which is required for setting up the environment like passing command line arguments.

Linker task can be easily verified by using

```
$size filename.o
$size filename
```

Through these commands, we know that how output file increases from an object file to an executable file.

Note that GCC by default does dynamic linking, so printf() is dynamically linked in above program.

----

3. Namespace

Namespaces allow us to group named entities that otherwise would have global scope into narrower scopes, giving them namespace scope. This allows organizing the elements of programs into different logical scopes referred to by names.

Namespace is a feature added in C++ and not present in C.
A namespace is a declarative region that provides a scope to the identifiers (names of the types, function, variables etc) inside it.
Multiple namespace blocks with the same name are allowed. All declarations within those blocks are declared in the named scope.

In each scope, a name can only represent one entity. So, there cannot be two variables with the same name in the same scope. Using namespaces, we can create two variables or member functions having the same name.

A namespace definition begins with the keyword namespace followed by the namespace name as follows:

```
namespace namespace_name
{
   int x, y; // code declarations where
             // x and y are declared in
             // namespace_name's scope
}
```

Namespace declarations appear only at global scope.
Namespace declarations can be nested within another namespace.
Namespace declarations don’t have access specifiers. (Public or private)
No need to give semicolon after the closing brace of definition of namespace.
We can split the definition of namespace over several units.

```
// Creating namespaces
#include <iostream>
using namespace std;
namespace ns1
{
    int value()    { return 5; }
}
namespace ns2
{
    const double x = 100;
    double value() {  return 2*x; }
}

int main()
{
    // Access value function within ns1
    cout << ns1::value() << '\n';

    // Access value function within ns2
    cout << ns2::value() << '\n';

    // Access variable x directly
    cout << ns2::x << '\n';

    return 0;
}
```

----

4. What happen when we exceed valid range of built-in data types in C++?

Consider the below programs.
1) Program to show what happens when we cross range of ‘char’ :

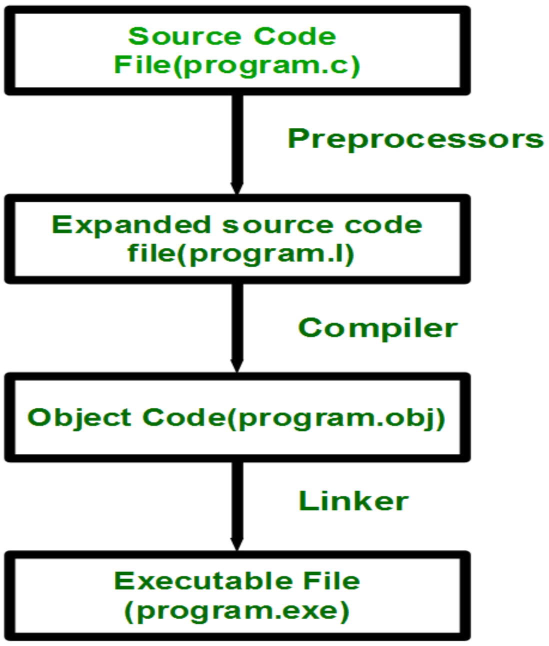
```
// C++ program to demonstrate
// the problem with 'char'
#include <iostream>
using namespace std;

int main()
{
    for (char a = 0; a <= 225; a++)
        cout << a;
    return 0;
}
```

Will this code print ‘a’ till it becomes 226? Well the answer is indefinite loop, because here ‘a’ is declared as a char and its valid range is -128 to +127. When ‘a’ become 128 through a++, the range is exceeded and as a result the first number from negative side of the range (i.e. -128) gets assigned to a. Hence the condition “a <= 225” is satisfied and control remains within the loop. The same example applies for signed short(-32767 to +32767) and unsigned short(0 to +65535)

* For bool any value >= 1 is true , and 0 is false

----

5. C/C++ Preprocessors


Source Code File(program.c)
Preprocessors
Expanded source code file(program.I)
Compiler
Object Code(program.obj)
Linker
Executable File (program.exe)

Preprocessors are programs that processes our source code before compilation.

Preprocessor programs provides preprocessors directives which tell the compiler to preprocess the source code before compiling. All of these preprocessor directive begins with a ‘#’ (hash) symbol. This (‘#’) symbol at the beginning of a statement in a C/C++ program indicates that it is a pre-processor directive. We can place these pre processor directives anywhere in our program. Examples of some preprocessor directives are: *#include* , *#define*, *#ifndef* etc.

There are 4 main types of preprocessor directives

Macros : Macros are piece of code in a program which is given some name. Whenever this name is encountered by the compiler the compiler replaces the name with the actual piece of code.

```
#include<iostream>
//macro definition
#define LIMIT 5
int main()
{
     for(int i=0; i < LIMIT; i++)
     {
          std::cout<<i<<"\n";
     }
     return 0;
}
```

File Inclusion :

Header File or Standard files => **`#include< file_name >`**

user defined files => **`#include"filename"`**

Conditional Compilation : Conditional Compilation directives are type of directives which helps to compile a specific portion of the program or to skip compilation of some specific part of the program based on some conditions. This can be done with the help of two preprocessing commands **ifdef** and **endif**.

```
ifdef macro_name
    statement1;
    statement2;
    statement3;
    .
    .
    .
    statementN;
endif
```

Other directives :

`#undef Directive` : The **`#undef`** directive is used to undefine an existing macro.

```
#undef LIMIT
```

Using this statement will undefine the existing macro LIMIT. After this statement every **#ifdef LIMIT** statement will evaluate to false.

`#pragma Directive` : This directive is a special purpose directive and is used to turn on or off some features. This type of directives are compiler-specific i.e., they vary from compiler to compiler

`#pragma startup` and **`#pragma exit`** : These directives helps us to specify the functions that are needed to run before program startup(before the control passes to main()) and just before program exit (just before the control returns from main()).

Note: Below program (#pragma command) will not work with GCC compilers.

```
#include<stdio.h>

void func1();
void func2();

#pragma startup func1
#pragma exit func2

void func1()
{
     printf("Inside func1()\n");
}
void func2()
{
     printf("Inside func2()\n");
}
int main()
{
     printf("Inside main()\n");
     return 0;
}
```

`Note :` `For gcc compiler below code can be used`

```
void __attribute__((constructor)) func1();
void __attribute__((destructor)) func2();
```

#pragma warn Directive: This directive is used to hide the warning message which are displayed during compilation.
We can hide the warnings as shown below:

- **#pragma warn -rvl**: This directive hides those warning which are raised when a function which is supposed to return a value does not returns a value.
- **#pragma warn -par**: This directive hides those warning which are raised when a function does not uses the parameters passed to it.
- **#pragma warn -rch**: This directive hides those warning which are raised when a code is unreachable. For example: any code written after the *return* statement in a function is unreachable.
