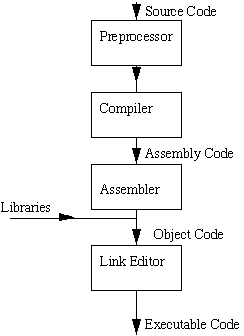
**Compilation Model**



The preprocessor is the first step of the compilation process. It prepares the source files for the compiler.

The preprocessor is responsible for . . .

* Removing all the comments from the source files.
* Executing the preprocessor directives (#define and #include).

The **gcc compiler** translate C code to assembler.

The **assembler** takes assembly code and transforms it into object code

If a source file references library functions or functions defined in other source files the link editor combines these functions (with main()) to create an executable file. External Variable references resolved here also.

**About GCC**

Gcc encapsulates all the different step of the compilation process.

* + Create main.i, the preprocessed version of main.c

gcc -E main.c

* + Create main.s, the assembler code of main.c

gcc -S main.c

* + Create main.o, the object code of main.c

gcc -c main.c

* + Create a.out, the compiled executable of main.c

gcc main.c

**Some options**

* **-o filename** : allows you to specify the name of the output executable (instead of a.out).
* **-v** : enable verbose mode (more output information).
* **-w** : suppresses warning messages (bad idea)
* **-W** : extra warning messages (good idea)
* -**Wall** : all warning messages (best idea)
* -**O1** : Optimize code for size and speed.
* **-O2** : Optimize even more
* **-g**: Invoke debugging option. This instructs the compiler to produce additional symbol table information that is used by a variety of debugging utilities.

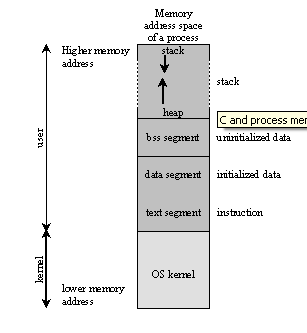
**Memory Segments in C**

The following Table summarizes the segments in the memory address space layout as illustrated in the previous Figure.

|  |  |
| --- | --- |
| **Segment** | **Description** |
| Code - text segment | Often referred to as the **text segment**, this is the area in which the executable or binary image instructions reside.  For example, Linux/Unix arranges things so that multiple running instances of the same program share their code if possible.  Only one copy of the instructions for the same program resides in memory at any time.  The portion of the executable file containing the text segment is the **text section**. |
| Initialized data – data segment | Statically allocated and global data that are **initialized with nonzero values** live in the **data segment**.  Each process running the same program has its own data segment.  The portion of the executable file containing the data segment is the **data section**. |
| Uninitialized data – bss segment | BSS stands for ‘Block Started by Symbol’.  Global and statically allocated data that **initialized to zero by default** are kept in what is called the BSS area of the process.  Each process running the same program has its own BSS area.  When running, the BSS, data are placed in the data segment.  In the executable file, they are stored in the **BSS section**.  For Linux/Unix the format of an executable, only variables that are initialized to a nonzero value occupy space in the executable’s disk file |
| Heap | The heap is where dynamic memory (obtained by malloc(), calloc(), realloc() and new – C++) comes from.  Everything on a heap is anonymous, thus you can only access parts of it through a pointer. As memory is allocated on the heap, the process’s address space grows.  Although it is possible to give memory back to the system and shrink a process’s address space, this is almost never done because it will be allocated to other process again.   Freed memory (free() and delete – C++) goes back to the heap, creating what is called holes.   It is typical for the heap to grow upward.  This means that successive items that are added to the heap are added at addresses that are numerically greater than previous items.  It is also typical for the heap to start immediately after the BSS area of the data segment.  The end of the heap is marked by a pointer known as the break. You cannot reference past the break. You can, however, move the break pointer (via brk() and sbrk() system calls) to a new position to increase the amount of heap memory available. |
| stack | The stack segment is where local (automatic) variables are allocated.  In C program, local variables are all variables declared inside the opening left curly brace of a **function's body** including the main() or other left curly brace that aren’t defined as static.  The data is **popped** up or **pushed** into the stack following the **Last In First Out** (LIFO) rule.  The stack holds local variables, temporary information, function parameters, return address and the like.  When a function is called, a **stack frame** (or a procedure activation record) is created and **PUSH**ed onto the top of the stack. This stack frame contains information such as the address from which the function was called and where to jump back to when the function is finished (return address), parameters, local variables, and any other information needed by the invoked function. The order of the information may vary by system and compiler.  When a function returns, the stack frame is **POP**ped from the stack.  Typically the stack grows downward, meaning that items deeper in the call chain are at numerically lower addresses and toward the heap. |

**The Process Memory Layout**

* A running program is called a **process** and when a program is run, its executable image is loaded into memory area that normally called a process address space in an organized manner.
* This is a physical memory space and do not confuse yourself with the virtual address space explained in [Module W](http://www.tenouk.com/ModuleW.html).
* Process address space is organized into three memory areas, called **segments**: the **text** segment, **stack** segment, and **data** segment (bss and data) and can be illustrated below.

****

* The text segment (also called a code segment) is where the compiled code of the program itself resides.

**Some terms**

|  |  |
| --- | --- |
| **Memory Allocation**  **Type** | **Description** |
| Static allocation | This allocation happens when you declare a **static** or **global** variable.  Each static or global variable defines one block of space, of a fixed size.  The space is allocated once, when your program is started, and is never freed.  In memory address space, for uninitialized variables are stored in bss segment while an initialized variables stored in data segment. |
| Automatic allocation(Dynamic) | This allocation happens when you declare an automatic variable, such as a function argument or a local variable. The space for an automatic variable is allocated when the compound statement containing the declaration is entered, and is freed when that compound statement is exited.  As discussed before this allocation done in the stack segment. |

* Dynamic allocation is not supported by C variables; there is no storage class called ‘dynamic’, and there can never be a C variable whose value is stored in dynamically allocated space. All must be done manually using related functions.
* The only way to refer to dynamically allocated space is **through a pointer**.  Because it is less convenient, and because the actual process of dynamic allocation requires more computation time, programmers generally use dynamic allocation only when neither static nor automatic allocation will serve.
* The dynamic allocation done by using functions and the memory used is heap area.

**Stack/Heap**

* The stack is where memory is allocated for automatic variables within functions.  A stack is a Last In First Out (LIFO) storage where new storage is allocated and de-allocated at only one end, called the top of the stack.  Every function call will create a stack (normally called stack frame) and when the function exit, the stack frame will be destroyed.
* When a program begins execution in the function main(), stack frame is created, space is allocated on the stack for all variables declared within main().
* Then, when main() calls a function, a(), new stack frame is created for the variables in a() at the top of the main() stack.  Any parameters passed by main() to a() are stored on this stack.
* If a() were to call any additional functions such as b() and c(), new stack frames would be allocated at the new top of the stack.  Notice that the order of the execution happened in the sequence.
* When c(), b() and a() return, storage for their local variables are de-allocated, the stack frames are destroyed and the top of the stack returns to the previous condition.  The order of the execution is in the reverse.
* As can be seen, the memory allocated in the stack area is used and reused during program execution.  It should be clear that memory allocated in this area will contain garbage values left over from previous usage.

**Function calling Convention**

For function call, compilers have some convention used for calling them.  A convention is a way of doing things that is standardized, but not a documented standard. for every function call there will be a creation of a stack frame. The order in which function arguments are pushed onto the stack. Whether the caller function or called function (callee) responsibility to remove the arguments from the stack at the end of the call that is the stack cleanup process.

**Dynamic Allocation – The Functions**

* The heap segment provides more stable storage of data for a program; memory allocated in the heap remains in existence for the duration of a program.
* Therefore, global variables (external storage class), and static variables are allocated on the heap.  The memory allocated in the heap area, if initialized to zero at program start, remains zero until the program makes use of it.  Thus, the heap area need not contain garbage.
* In ANSI C , there is a family of four functions which allow programs to dynamically allocate memory on the heap.
* In order to use these functions you have to include the **stdlib.h** header file in your program.

**Malloc(**)

void \* malloc (size\_t nbytes);

nbytes is the number of bytes that to be assigned to the pointer.  The function returns a pointer of type void\*.  When allocating memory, malloc() returns a pointer which is just a byte address.  Thus, it does not point to an object of a specific type.  A pointer type that does not point to a specific data type is said to point to void type, that is why we have to type cast the value to the type of the destination pointer, for example:

char \* test;

test = (char \*) malloc(10);

This assigns test a pointer to a usable block of 10 bytes.

**Calloc()**

void \* calloc (size\_t nelements, size\_t size);

calloc() is very similar to malloc() in its operation except its prototype have two parameters.  These two parameters are multiplied to obtain the total size of the memory block to be assigned. Usually the first parameter (nelements) is the number of elements and the second one (size) serves to specify the size of each element. For example, we could define test with calloc():

int \* test;

test = (int \*) calloc(5, sizeof(int));

Another difference between malloc() and calloc() is that calloc() initializes all its elements to 0.

**Realloc()**

void \* realloc (void \* pointer, size\_t elemsize);

It changes the size of a memory block already assigned to a pointer.  pointer parameter receives a pointer to the already assigned memory block or a null pointer (if fail), and size specifies the new size that the memory block shall have.  The function assigns size bytes of memory to the pointer.  The function may need to change the location of the memory block so that the new size can fit; in that case the present content of the block is copied to the new one.  The new pointer is returned by the function and if it has not been possible to assign the memory block with the new size it returns a null pointer.

**Free()**

void free (void \* pointer);

It releases a block of dynamic memory previously assigned using malloc(), calloc() or realloc().  This function must only be used to release memory assigned with functions malloc(), calloc() and realloc().

**NULL**

NULL is a defined constant used to express null pointers, that is, an unassigned pointer (pointing to the address 0) or a pointer that points to something but not useful.

In practice, one must always verify whether the pointer returned is NULL.  If malloc() is successful, objects in dynamically allocated memory can be accessed indirectly by dereferencing the pointer, appropriately cast to the type of required pointer. The size of the memory to be allocated must be specified, in bytes, as an argument to malloc().   Since the memory required for different objects is implementation dependent, the best way to specify the size is to use the sizeof operator.  Recall that the sizeof operator returns the size, in bytes, of the operand.

**Some memory Problems**

**Uninitialized memory**

In this example, p has been allocated 10 bytes. The 10 bytes might contain garbage data, as shown

|  |
| --- |
| char \*p = malloc ( 10 ); |

**Garbage data**  
http://www.ibm.com/developerworks/aix/library/au-toughgame/fig1.gif

If a code segment tries to access this p before a value has been assigned to it, it might get that garbage value and your program could behave mysteriously. p might have a value that your program never expected. A good practice is to always use memset along with malloc, or always use calloc.

|  |
| --- |
| char \*p = malloc (10);  memset(p,0,10); |

Now, even if the same code segment tries to access p before a value has been assigned to it and it has proper handling of the Null value (which ideally it should be), then it will behave properly.

**Memory Leak**

What happens if some memory is heap allocated, but never deallocated? A program which forgets to deallocate a block is said to have a "memory leak" which may or may not be a serious problem. The result will be that the heap gradually fill up as there continue to be allocation requests, but no deallocation requests to return blocks for re-use.

For a program which runs, computes something, and exits immediately, memory leaks are not usually a concern. Such a "one shot" program could omit all of its deallocation requests and still mostly work. Memory leaks are more of a problem for a program which runs for an indeterminate amount of time. In that case, the memory leaks can gradually fill the heap until allocation requests cannot be satisfied, and the program stops working or crashes. Many commercial programs have memory leaks, so that when run for long enough, or with large data-sets, they fill their heaps and crash. Often the error detection and avoidance code for the heap-full error condition is not well tested, precisely because the case is rarely encountered with short runs of the program — that's why filling the heap often results in a real crash instead of a polite error message. Most compilers have a "heap debugging" utility which adds debugging code to a program to track every allocation and deallocation. When an allocation has no matching deallocation, that's a leak, and the heap debugger can help you find them.

**Dangling Pointer/Memory leak**

**Dangling pointers** in computer programming are pointers that do not point to a valid object of the appropriate type. Dangling pointers arise when an object is deleted or deallocated, without modifying the value of the pointer, so that the pointer still points to the memory location of the deallocated memory.

**Memory leak**:A memory leak in computer science is a particular type of unintentional memory consumption by a

computer program where the program fails to release memory when no longer needed.

ex:

void f(void)

{

void\* s;

s = malloc(50); /\* get memory \*/

return;

}

//control comes out w/o freeing the memory....so s is a dangling pointer.

**Segmentation Fault**

An error in which a running program attempts to access memory not allocated to it and core dumps with a segmentation violation error. This is often caused by improper usage of pointers in the source code, dereferencing a null pointer, or (in C) inadvertently using a non-pointer variable as a pointer. The classic example is:

int i;

scanf ("%d", i); /\* should have used &i \*/

A segmentation fault occurs when a program attempts to access a [memory](http://en.wikipedia.org/wiki/Computer_memory) location that it is not allowed to access, or attempts to access a memory location in a way that is not allowed (for example, attempting to write to a [read-only](http://en.wikipedia.org/wiki/Read-only_memory) location, or to overwrite part of the [operating system](http://en.wikipedia.org/wiki/Operating_system)).

int main(void)

{

char \*s = "hello world";

\*s = 'H'; }

When the program containing this code is [compiled](http://en.wikipedia.org/wiki/Compiler), the [string](http://en.wikipedia.org/wiki/String_literal) "hello world" is placed in the section of the program [binary](http://en.wikipedia.org/wiki/Executable_file) marked as read-only; when loaded, the operating system places it with other strings and [constant](http://en.wikipedia.org/wiki/Constant_(programming)) data in a read-only segment of memory. When executed, a variable, *s*, is set to point to the string's location, and an attempt is made to write an *H* character through the variable into the memory, causing a segmentation fault. Compiling such a program with a compiler that does not check for the assignment of read-only locations at compile time, and running it on a Unix-like operating system produces the following [runtime error](http://en.wikipedia.org/wiki/Runtime_error):

$ gcc segfault.c -g -o segfault

$ ./segfault

Segmentation fault

**Stack Overflow**

In software, a **stack overflow** occurs when too much [memory](http://en.wikipedia.org/wiki/Computer_memory) is used on the [call stack](http://en.wikipedia.org/wiki/Call_stack). The call stack contains a limited amount of memory, often determined at the start of the program. The size of the call stack depends on many factors, including the programming language, machine architecture, multi-threading, and amount of available memory. When too much memory is used on the call stack the stack is said to overflow, typically resulting in a program crash. The most common cause of stack overflows is excessively deep or infinite recursion.

Main(){

Main();

}

The other major cause of a stack overflow results from an attempt to allocate more memory on the stack than will fit. This is usually the result of creating local array variables that are far too large. For this reason arrays larger than a few kilobytes should be allocated dynamically instead of as a local variable.

main() {

double x[1000000];

}

The declared array consumes 8 [megabytes](http://en.wikipedia.org/wiki/Megabytes) of data (assuming each double is 8 bytes), which is more memory than is available on the stack.

**Difference between NULL & null**

A NULL pointer is a pointer that's guarnteed to point to nothing. This may be 0 in a UNIX/Linux system or some other address in another system. Using the NULL macro to set/initialize your pointers will make your programs more portable among systems than using something like the 0.

#include <stdio.h>   
  
char \*c = 0; // initialize to NULL--not portable   
char \*p = NULL; // initialize to NULL as defined in stdio is portable

The NULL in C means zero or void.

**NULL Pointers**

If (pointer ==NULL)

NULL is defined to compare equal to a null pointer. It is implementation defined what the actual definition of NULL is, as long as it is a valid null pointer constant.

if (pointer == 0)

0 is another representation of the null pointer constant.

if (!pointer)

This if statement implicitly checks "is not 0", so we reverse that to mean "is 0".

**NULL characters**

'\0' is defined to be a null character - that is a character with all bits set to zero. This has nothing to do with pointers. Additionally, '\0' is (like all character literals) an integer constant, in this case with the value zero. So '\0' is completely equivalent to an unadorned 0 integer constant - the only difference is in the *intent* that it conveys to a human reader