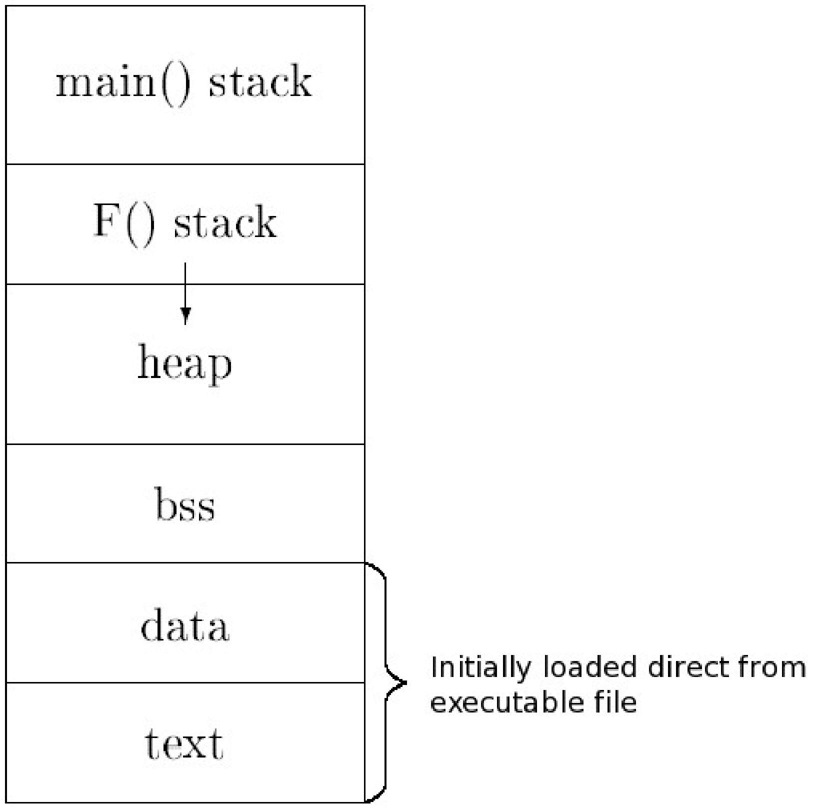
5.5. Variables in Memory

To understand *storage classes* and *scope* in detail, first we need to know more about how the compiler/computer stores variables in the computer’s memory. To accomplish this, we will first look at an overview of parts of the programs memory, then we will consider C’s storage classes and how the storage class affects a variable’s scope and placement in memory.

When a program is loaded into memory and started, a contiguous section of virtual memory is reserved for the program. On most systems, this memory section can be expanded when needed. We refer to this as virtual memory because in reality the memory may not be contiguous and in fact it may not all even be loaded into memory at the same time. It is the responsibility of the operating system’s paging (memory swapping) system to translate between virtual memory and real physical memory – well beyond the scope of this class. As far as our user level program knows, it has a contiguous block of memory available for its use.

When a program is started, the binary, executable file is first copied directly to the bottom portion of the memory block into two sections called the *text* and *data* sections of memory. Next, a fixed amount of memory (the amount determined by the compiler) is reserved and the memory values in this section are initialized to hold only zeros. This area of memory is called the *bss* section. Next the program skips to the top of our available memory block and starts executing the program. The program adds and removes memory as needed from the top of the memory block using a section of memory called the *stack*. As the program uses more memory and the stack grows, the address of the *top* of the stack decreases. So the top of the stack is actually has the lowest memory address in the stack section of memory. Whenever a function is called, new data is *pushed* or added to the stack. When the function returns, the stack data for the function is *popped* or removed from the stack. The area between the bss and stack sections of memory is referred to as the *heap*.



The contents of the sections of the memory can be described as follows:

`Text Section`

The text section of the program contains the executable instructions of the program. Thus, the program instruction counter is a pointer into the text section. Constants, such as the string constants of a printf() statement are also stored in the text section of memory.

`Data Section`

The data section of memory if for global and static data that is initialized when declared. Since, the initial value of initialized variables is known in advance, all global or static variables are saved together in the executable file and the data block is loaded into memory directly from the executable file just like the text section.

`Bss Section`
: The bss section, like the data section, is for storing global and static variables. The difference being that the bss section stores variables that were not initialized to a specific value when declared. Bss data is initialized to zero when the program starts.

`Stack Section`
: The stack stores various pointer values which are needed for the execution of the program and also is the default storage location for variables which are local to a function, i.e., declared within the body of the function. The stack data is created when program begins a function and is destroyed when that function exits. When the running function call another function, the new function's data is pushed in front of the original function's data.

`Registers`
: In addition to the computer's main memory, it is also possible to temporarily store a few variables directly in memory locations called registers which are a part of the computer's CPU.

5.5.1. Storage classes

Every variable declaration has 3 attributes.

1. type (int, float, double, char, ...), discussed in Topic 1.
2. placement of the declaration
   1. Global variables are declared outside of any functions and may, by default, be accessed by any function in the program.
   2. Variables declared inside a function (including *main()*), are accessible to only the function where they are defined.
3. storage class The following keywords may be used in a data declaration and have impact on where the variable is stored in the computer's memory and on the rules of scope for the variable.
   1. auto - the default.
   2. extern
   3. register
   4. static

5.5.1.1. auto

- This is the default for variables. The `auto` keyword is seldom used.
- Auto variables are placed on the stack. So once the function that uses the variable exits, the variable goes away and that memory location is available for other functions to use.
- Each function invocations sets up new automatic variables on the stack. The function is only aware of its own automatic variables, and can only access other function's automatic variables through the use of pointers.

5.5.1.2. extern

- When a variable is defined outside of any function (including main). It is a *global* variable and is placed in either the data or bss sections of memory.
- Any function in a the file where it is declared has access to a global variable.
- Using the keyword `extern` in another file gives access to the variable in the second file. If the `extern` declaration is inside a function, only that function has access to the global variable. If it is outside of any function, then all functions in the file have access to the variable.
- The `extern` statement does not declare a variable, but just references a global variable which was declared in another file.

```
/* File1.c: */
int Global;

extern void f( int ); /* functions as well as variables can be extern */
extern void g( int );
int main(void)
{
   ....
   f( i );
   g( j );
}
---------------------------------------
/* File2.c  */
extern int Global;  /* now f() and g() have access to Global */
void f( int );
void g( int );

void f( int i )
{
   ....
}

void g( int j )
{
   ....
}
```

5.5.1.3. register

- In addition to the computer's main memory, a limited amount of memory exists on the CPU itself. The computer can access register memory much faster than main memory.
- The keyword `register` is an *advisory request* to store a variable in one of the CPU's registers.
- *Advisory* means that the compiler decided if this request is honored. If you ask for a large number of variables to be stored in registers, then it is not possible for the compiler to honor your request.
- `register` should be used sparing and only for variables that are accessed many times in a tight loop. It is best to use `register` variables in small functions, so the variable is in use for a short amount of time.

5.5.1.4. static

The keyword `static` is used in two ways.

- A `static` variable declared inside a function is placed in the data or bss portion of memory and can retain it's value between calls to the same function. In this way it is similar to a global variable, except it is only used in one function. The restriction on the scope of static variables is a compiler implemented restriction; being that, an assembly language program could easily access any data stored in either the data or bss sections of memory.

- A `static` variable declared as a global variable is also called a *static external* variable. Here the keyword `static` produces the opposite results as the `extern` keyword. The variable is global to the file where it is declared, but may not be referenced in any other files. Thus the keyword `static` can produce a form of data hiding.

  Static variables provide a simple means to hide information in a C program. In other words, a variable may need to be persistent or global, but we may not want other programmers to know much about this variable - least they try to change it.

5.5.2. const qualifier

If the keyword `const` is used when declaring a variable, then we restrict the variable from being changed past its initial definition. `const` is not a storage class keyword because it does not affect the scope or memory location of the variable.

Note: Now complete Homework 7 - Using a Static Local Variable.