Lecture 04 - Storage classes (last updated: 2023-09-21)

In the last lecture, we saw how C variables occupy memory to hold their value. In this lecture, we will begin to learn about how memory is organized.

Storage classes

The term "storage class" refers to whether a variable has a transient or permanent lifetime. There are two kinds of storage classes in C:

- "Automatic" variables are transient; i.e., they are created and destroyed as the program enters and exits functions.
- "Static" variables are permanent; i.e., they are created when the program starts and has a value throughout the execution of the program.

Automatic variables

Automatic variables are also called "local" variables because they exist only inside the block where they are defined. This is called the "scope" of the variable. Here is an example showing how they work:

```
    void f(int p)             // p is a local variable whose scope is
    {                         // the body of the function f
        printf("%d\n", p);

        int x;  // defines a local variable x whose scope is the rest of the
        x = 0;                                                  // function

        {
            int x;  // defines another separate local variable x whose scope is
            x = 1;                                  // limited to the inner block
            printf("%d\n", x);  // prints 1
        }

        printf("%d\n", x);  // prints 0
    }
```

Automatic variables are also called stack variables because they are stored in the area of memory called stack. More on this later.

Static variables

There are three kinds of static variables:

(1) Global variables

Global variables are defined in the global scope, i.e., outside of any function:

```
    // in main.c for example:

    int x = 0;         // this is usually how a global variable is defined
```

```
extern int y = 0;  // extern keyword here is optional, and normally not used

int z;        // all static variables are initialized to 0 by default,
              // but it's better to initialize it explicitly as we did for x

int main() {  // code continues ...
```

Global variables defined in one file can then be referenced from another file:

```
// in foo.c for example:

extern int x;

int foo() { // code continues ...
```

Using the extern keyword without initialization makes the declaration a reference to a global variable defined elsewhere. When foo.c is compiled into foo.o, foo.o will contain an "undefined" reference to a symbol x, which will be resolved later when foo.o is linked with a object file that contains the definition of the symbol x, which is main.o.

(2) File-static variables

A variable can be defined in the global scope (just like global variables) but with the addition of the "static" keyword:

```
// in foo.c for example:

static int a = 0;
```

We call these variables file-static variables. (These variables are commonly -- and confusingly -- called just static variables; we will reserve "static variables" to refer to all three kinds of static variables.)

They behave the same way as global variables, except that their scope is limited to the file where it is defined. In other words, file-static variables cannot be referred to from other files using extern declarations.

(3) Block-static variables

A static variable's scope can be further confined into a local block. Consider the following program:

```
int f()
{
    static int i = 100;
    i++;
    return i;
}

int main()
{
    for (int i = 0; i < 5; i++) {
        printf("%d\n", f());
    }
}
```

The program produces the following output:

```
101
```

```
102
103
104
105
```

The variable i in the function f() looks like a local variable, but it doesn't act like one. Instead of getting initialized to 100 every time the function is called, it gets initialized to 100 once at the program start-up, and it keeps its value across multiple invocations of the function. In other words, the program acts exactly as if the following line had been pulled out of the function f() and placed in the global scope.

```
static int i = 100;
```

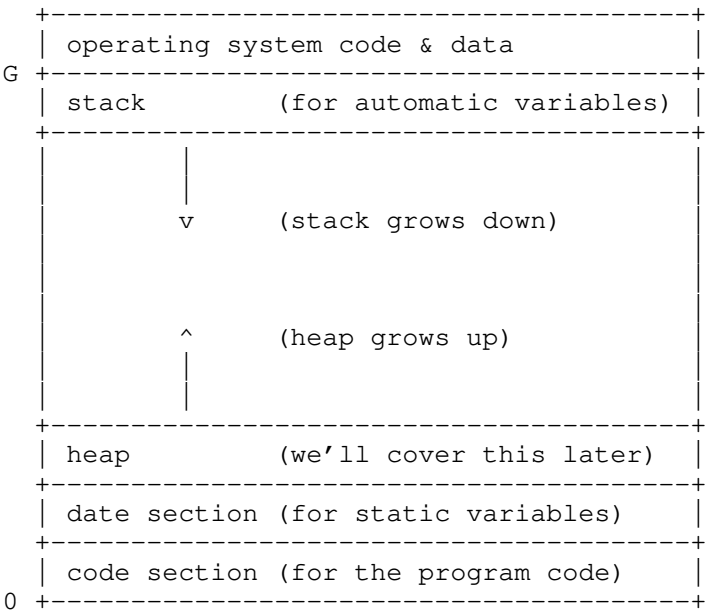
Indeed, block-static variables act just like global or file-static variables in that it gets created once at the program start-up and retains its value throughout the execution of the program. The only difference is that its scope is limited to the block where it's defined, and thus it is not available to code outside of the block.

In summary, all three kinds of static variables -- global, file-static, and block-static variables -- are created and initialized on program start-up and persists throughout the execution of the program. They differ only in the scope where they are visible to other parts of the program.

Process memory address space

Under 64-bit Linux operating system, every single process (i.e., a running instance of a program) gets a vast memory space. Each byte of that memory space is linearly addressed from 0 to a large number like 512G.

An over-simplified illustration of a process's memory space looks like this:

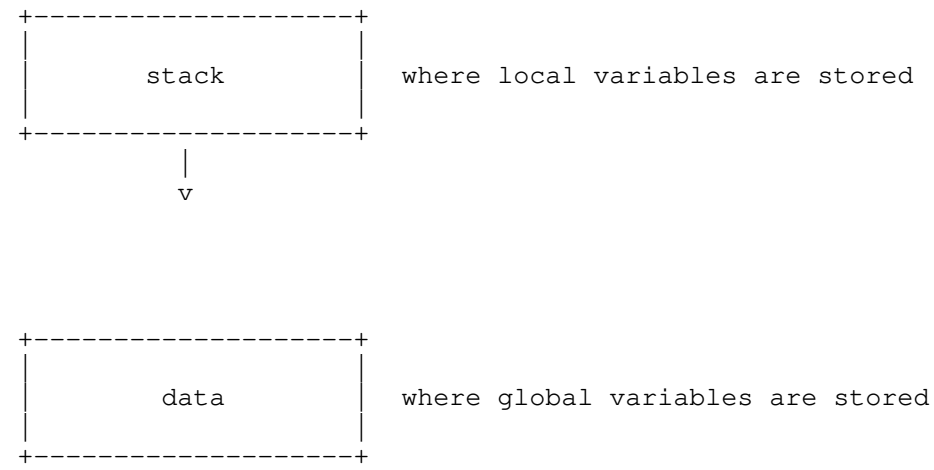


Most computers do not have 512G of physical memory (RAM), let alone enough memory to give 512G to each of the dozens or even hundreds of processes running at any given time. The operating system provides an illusion that each process

owns the entire memory space, and one that is much bigger than what's actually there. This is called virtual memory.

Stack and data section

For now, we will confine our discussion to the stack, which stores automatic variables, and the data section, which stores static variables.

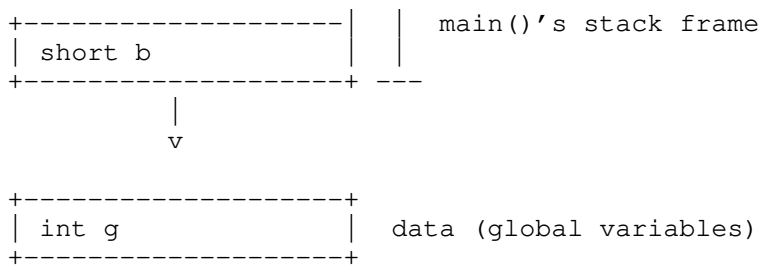


The "stack" stores local variables in each function, and grows downward. Because the number and sizes of global variables stay the same at runtime, the "data" section of memory is fixed at runtime.

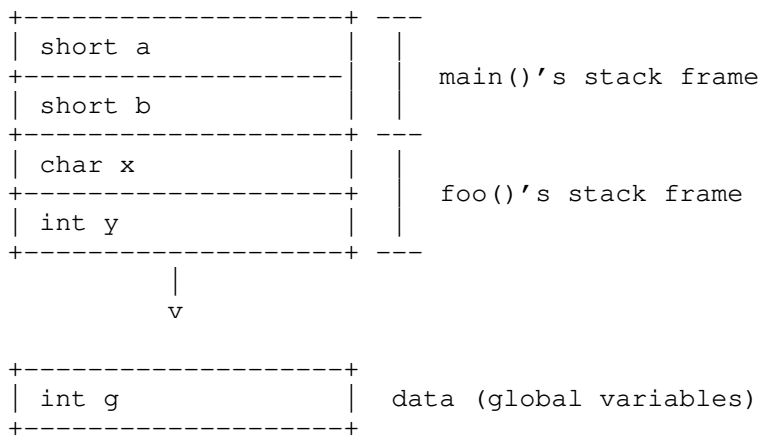
For example, consider the following program:

```
int g;

void foo(void) {
    char x;         // 1 byte
    int y;          // 4 bytes

    // DIAGRAM 2
}

int main(void) {
    short a;        // 2 bytes
    short b;        // 2 bytes

    // DIAGRAM 1

    foo();

    // DIAGRAM 3

    return 0;
}
```

When this program's execution reaches the comment "DIAGRAM 1", the memory layout might look something like this:



Once main() calls foo() and the program's execution reaches the comment "DIAGRAM 2", the memory layout might look something like this:



Then when foo() returns, execution will resume in the main() function at the comment "DIAGRAM 3", the memory layout will return to looking like what it did at the comment "DIAGRAM 1".

Some things to keep in mind about this rough illustration:

- The height of each variable's box is not drawn to scale, for brevity. e.g., the boxes for ints should be four times as tall as the box for char.
- The order of variables within a function's stack frame is not specified by the C standard; however, the order of stack frames is well-defined, and is determined by the order of function calls.
- In the stack, some small gaps between variables aren't shown.
- Even if a function does not declare any local variables, its stack frame will still occupy some memory, which isn't shown in this illustration.
- The distance between the stack and the data section is much, much larger than the illustration suggests.

Also note that if a function is called multiple times (e.g., if it is called recursively), it will have separate stack frames. Stack overflow occurs when there are so many nested function calls that memory can no longer accommodate the stack. For example, this function will cause a stack overflow:

```
int overflow(void) {
    return overflow();
}
```

The key takeaways here are:

- The stack grows downwards with each function call, and recedes upwards when a function returns.
- The data section is fixed.