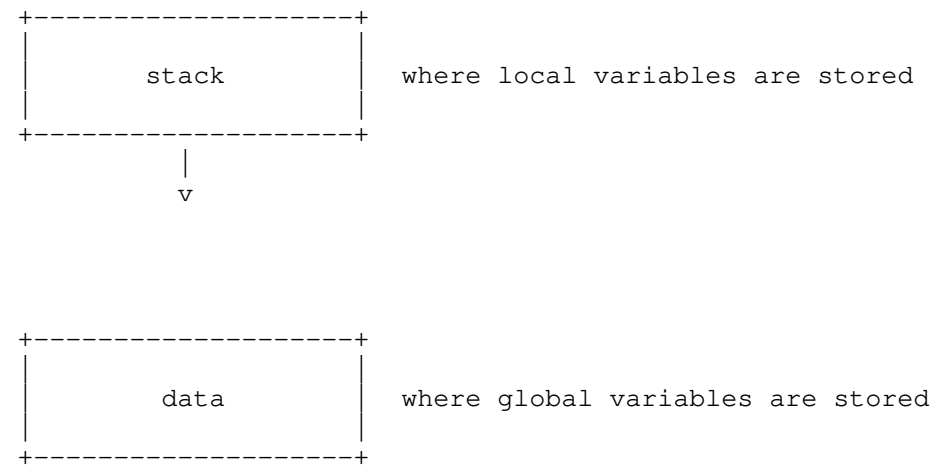
Lecture 05 - Pointers and addresses (last updated: 11/03/22)

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.

Linear memory and the stack

Memory is an array of bytes; a memory address is an index of this array.

An over-simplified illustration of memory looks like this (where larger memory addresses appear higher up in the diagram):

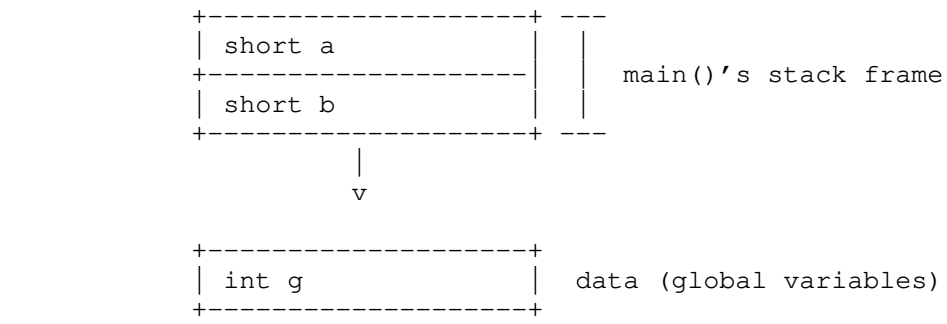


The "stack" stores local variables in each function, and grows downward. Because the number and sizes of global variables stay the same at runtime, the "data" section of memory is fixed at runtime.

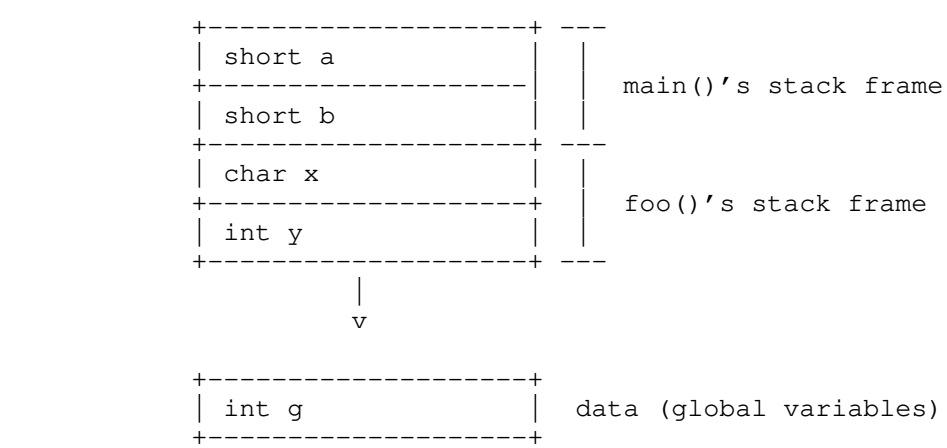
For example, consider the following program:

```
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.

Pointers and addresses

Locations in memory are indexed by memory addresses. We call values that hold memory addresses "pointers," because they "point" to a location in memory. The type of a pointer that points to some type T is written "T *". For example:

```
int *p;             // p is a pointer to an integer
```

We can assign p the location of integers in memory. For example:

```
int x = 1;

p = &x;             // p now holds the address of x
```

We can obtain the address of x (i.e., a pointer to x) using the prefix "address-of" operator, &. You may hear this colloquially referred to as the "reference" operator, since the pointer "refers" to x. However, we will insist on calling it the "address-of" operator because the term refers to something else in C++.

You can read from the address held by a pointer using the "dereference" operator, *:

```
int y = *p;         // read 1 from x (pointed to by p) and assign it to y
```

We can also write to the address held by a pointer by dereferencing it on the left-hand side of an assignment:

```
*p = 0;             // x is now 0
```

We can of course reassign p to point at something else:

```
p = &y;             // p now points to y

*p = 2;             // y is now 2; x is still 0

++*p;               // y is now 3

(*p)++;             // y is now 4
```

Note the necessary parentheses in (*p)++, which means "dereference p, and increment what points to); *p++ is grouped as *(p++), which means "increment p, but dereference what p currently points to."

Note that you can't mix pointers to different types without casting:

```
int i = 1234;
double d = 3.14;

int *pi = &i;
double *pd = &d;
```

```
    // pi = pd;         // compiler error

    pi = (int *) pd;    // compiles, but you better know what you're doing
```

Casting can be very unsafe, because you are telling the compiler that you want to reinterpret the bits of one type as another. At this point, if you tried to print the value of *pi, you would find a very surprising value.

There is an exception to C's pointer type rules: void pointers (i.e., void *). Void pointers can point to any type, but do not tell us anything about what they point to (i.e., the size of the data they point to). We can freely convert to and from void pointers without casting:

```
    void *pv = pi;      // no cast necessary; no compiler error
```

However, C does not allow us to dereference void pointers, because it does not know how many bytes we mean to read or write:

```
    // i = *pv;         // compiler error: can't dereference a void pointer
```

Simulating pass-by-reference with pointers

C is "pass-by-value" language, meaning the values of function parameters are copied from the caller to the callee. For example, consider this broken implementation of the swap() function (i.e., the callee):

```
    void bad_swap(int x, int y) {
        int temp = x;
        x = y;
        y = temp;
    }
```

Here is a snippet of code that calls our broken bad_swap():

```
    int x = 1, y = 2;
    bad_swap(x, y);     // Didn't work: x is still 1, y is still 2
```

bad_swap() fails to swap the values of x and y in the caller's context because bad_swap() only swaps its local variables.

In order to make swap() work, we need "pass-by-reference" semantics, which we can fake in C using a level of indirection, i.e., pointers:

```
    void swap(int *px, int *py) {
        int temp = *px;
        *px = *py;
        *py = temp;
    }
```

Now we can call swap() with pointers to x and y:

```
    int x = 1, y = 2;
    swap(&x, &y);       // Now x is 2 and y is 1
```

Note that we are still passing arguments to swap() by value, except this time the values are pointers.