CSE 220: Systems Programming
Functions and Automatic Variables

Ethan Blanton

Department of Computer Science and Engineering
University at Buffalo
Stack Manipulations

Function calls and **automatic variables** use the stack.

**Automatic variables are variables declared inside a function.**

Usage of the stack is platform-specific, and part of the application binary interface (ABI).

As previously discussed, stack memory is **implicitly allocated**.
The Stack — Again

A reminder: the stack grows downward in memory.

The base of the stack is at some high address.

The top of the stack is at a lower address.

Stack operations:

- **push**: Move the top of the stack down to make room for an item, then place it at the top.
- **pop**: Remove an item from the stack by moving the top up above it.

The stack is last in, first out (LIFO):
Only the most recently pushed item(s) can be popped!
Stack Operations

(An empty stack; each row is 32 bits.)
Stack Operations

```
push int i = 42;
```
Stack Operations

push double d = 2.0;
(Remember padding!)
Stack Operations

push struct { int x = 3; int y = 5; } pos;
Stack Operations

Stack items are typically referenced with respect to its top. E.g., $d$ is at $top + 8$.
Stack Operations

Low Addresses

pop 20 bytes to remove pos and d

Note that the unused data remains present on the stack.
Variable Declarations

A variable declaration does two things:

- Asks the compiler to reserve space on the stack for data
- Names the location of that data

```c
int array[32];
```

“Make space for 32 integers and call that space array.”
Variable Declarations

A variable declaration does two things:
- Asks the compiler to reserve space on the stack for data
- Names the location of that data

```
int array[32];
```

“Make space for 32 integers and call that space array.”
Variable Declarations

A variable declaration does two things:

- **Asks the compiler to reserve space on the stack for data**
- **Names the location of that data**

```c
int array[32];
```

“Make space for 32 integers and call that space array.”
Variable Declarations

A variable declaration does two things:
- Asks the compiler to reserve space on the stack for data
- Names the location of that data

```c
int array[32];
```

“Make space for 32 integers and call that space array.”

Every non-static local variable is an automatic variable.
Automatic Variable Lifetime

Automatic variables are:

- Guaranteed to be *allocated* before they *are first referenced*
- Guaranteed to be *valid* until their *enclosing block is done*

In many cases they are created when the *function is entered*.

Placing automatic variables on the stack *allows this*. 
Automatic Variable Placement

Automatic variables may be allocated anywhere.

The programmer cannot predict their order or location.

They may only be in registers!

Their structure will be preserved.
Automatic Variable Placement

Automatic variables may be allocated anywhere.

The programmer cannot predict their order or location.

They may only be in registers!

Their structure will be preserved.

```
int i;
struct {
    int x; int y;
} pos;
```
Function Call Nesting

Note that:
- Function calls form a **tree** over the life of a program
- Function calls form a **stack** at any point in time

This is because:
- A function may call *many functions* consecutively
- A function can call *only one function* at a time

These properties directly affect the program stack.
Function Calls

At its simplest, a function call consists of:

- A jump to a new program location
- Execution of the function code
- A jump back to the calling location

However, many function calls are more complicated. They may:

- Allocate automatic variables
- Call other functions
- Temporarily save registers
- …

In these cases, functions require a stack frame.
Stack Frames

A stack frame\(^1\) holds information for a single function invocation.

While the details vary by platform, it will include:

- Saved processor registers
- Local variables for the current function
- Arguments for any called function
- The return location for any called function

We will discuss all of these except saved processor registers. (Maybe we’ll get to those later.)

\(^1\)You will sometimes see this called an activation record.
Local Variables

We have previously discussed automatic variables.

Often, all local variables for a function are allocated together.

When the function is entered, it will immediately move the top of the stack to make room for its local storage.

This portion of the stack frame is then of fixed size.

Its size is often not saved, but recorded in the program instructions by the compiler.

The location of individual variables are likewise recorded.
Function Arguments

The platform ABI will determine how arguments are passed. Normally, it is a combination of registers and stack space.

On x86-64 Linux, the first six 64 bit values are passed in registers. Any additional arguments are pushed onto the stack.

Therefore, many functions have no arguments on the stack.
Function Arguments Layout

If function arguments are pushed onto the stack, they are normally pushed in reverse order.

That is, the first function argument is closest to the top.

Among other reasons, this allows for a variable number of arguments.

Consider printf: it takes 1 or more arguments. The first format argument tells it how many.
The Program Counter

The other major item that must be tracked for the function call stack is the program counter.

The program counter is the address of the machine instruction the processor is currently executing.

For a function call:

- the current program counter is pushed before jumping to the called function
- the called function pops the program counter in order to return

On some architectures there is a dedicated instruction for this.
A Stack Frame

(Exactly which elements are part of which frame is negotiable.)
A Call Stack

```c
void foo() {
    int i = 3;
    bar(i);
    /* ... */
}

void bar(int i) {
    int j = 2;
    i = 5 + j;
}
```
A Call Stack

```c
void foo() {
    int i = 3;
    bar(i);
    /* ... */
}

void bar(int i) {
    int j = 2;
    i = 5 + j;
}
```

Stack

top →
calling pc

call foo()
A Call Stack

```c
void foo() {
    int i = 3;
    bar(i);
    /* ... */
}

void bar(int i) {
    int j = 2;
    i = 5 + j;
}
```

Stack

calling pc

Reserve space for `foo()`’s locals
A Call Stack

```c
void foo() {
    int i = 3;
    bar(i);
    /* ... */
}

void bar(int i) {
    int j = 2;
    i = 5 + j;
}
```

Execute `foo()`
A Call Stack

```c
void foo() {
    int i = 3;
    bar(i);
    /* ... */
}

void bar(int i) {
    int j = 2;
    i = 5 + j;
}
```

Execute `foo();`
prepare to call `bar()`
A Call Stack

```c
void foo() {
    int i = 3;

    bar(i);
    /* ... */
}

void bar(int i) {
    int j = 2;

    i = 5 + j;
}
```

Push PC; call bar()
A Call Stack

```c
void foo() {
    int i = 3;
    bar(i);
    /* ... */
}

void bar(int i) {
    int j = 2;
    i = 5 + j;
}
```

Stack:
- **calling pc**: foo
- **foo(pc)**: 3
- **bar(i)**: 3
- **foo() pc**: foo

Reserve space for `bar()`’s locals
A Call Stack

```c
void foo() {
    int i = 3;
    bar(i);
    /* ... */
}

void bar(int i) {
    int j = 2;
    i = 5 + j;
}
```

Stack

Execute `bar()`
A Call Stack

```c
void foo() {
    int i = 3;
    bar(i);
    /* ... */
}

void bar(int i) {
    int j = 2;
    i = 5 + j;
}
```

Stack diagram:
- `foo: i = 3`
- `bar: i`
- `bar: j`
- `top: j = 2`
- `foo() pc: i = 5 + j`

Execute `bar()`
A Call Stack

```c
void foo() {
    int i = 3;
    bar(i);
    /* ... */
}

void bar(int i) {
    int j = 2;
    i = 5 + j;
}
```

Return from bar();
Pop bar()’s stack frame;
Execute foo()
Summary

- Automatic variables are allocated on the stack.
- Stack frames track function calls.
- The stack grows downward.
- Items removed from the stack are not cleared.
- Stack-allocated arguments are why C is call-by-value.
Next Time …

- The Heap
References

Required Readings


License

Copyright 2018, 2019 Ethan Blanton, All Rights Reserved.

Reproduction of this material without written consent of the author is prohibited.

To retrieve a copy of this material, or related materials, see https://www.cse.buffalo.edu/~eblanton/.