The Basics of C/C++ Variables & Program Memory

John Penn (CACI/ER7)

Variables

Definition: Variable

A *variable* is a portion of memory used by a program to store data values.

Attributes of a variable are:

- Reference
- Data Type
- Scope
- Lifetime

Definition: Variable-Reference

A ***reference*** provides access to the variable. A reference can be :

- name or
- address

Definition: Variable-Data-type

A data-type is a description of how data is represented within the variable.

Definition: Variable-Scope

The *scope* of a variable is where in a program a variable can be "seen".

Variable-Scope

Usually the scope of a variable is described as *local*, or *global*.

Definition: Variable-Lifetime

The *lifetime* of a variable is the period during which it is valid to access the variable.

Variable-Lifetime

The *lifetime* of a variable is described as *static*, *dynamic*, or *automatic*.

More Details

Variable-Reference

```
double velocity[10];
double *p = &velocity;
```

- The name *velocity* is a reference to a variable whose type is double[10].

- *p* is a reference to a variable whose type is double*, (pointer to double).

- The value of p is also a reference. It is a reference to the same variable (of type double[10]) to which *velocity* refers.

Variable-Reference

```
double thrust;
double& force = thrust;
```

1. Allocates storage for a variable, of type double, with a reference name, *thrust*.
2. Creates a reference name, *force*, to a double, that refers to the same variable as *thrust*. Note that we are <u>not</u> creating a new variable here. The variable already exists. We are just creating another way to refer to that variable.

Variable-Datatype

Primitive Types

- int
- char
- bool
- float
- double
- void

etc …

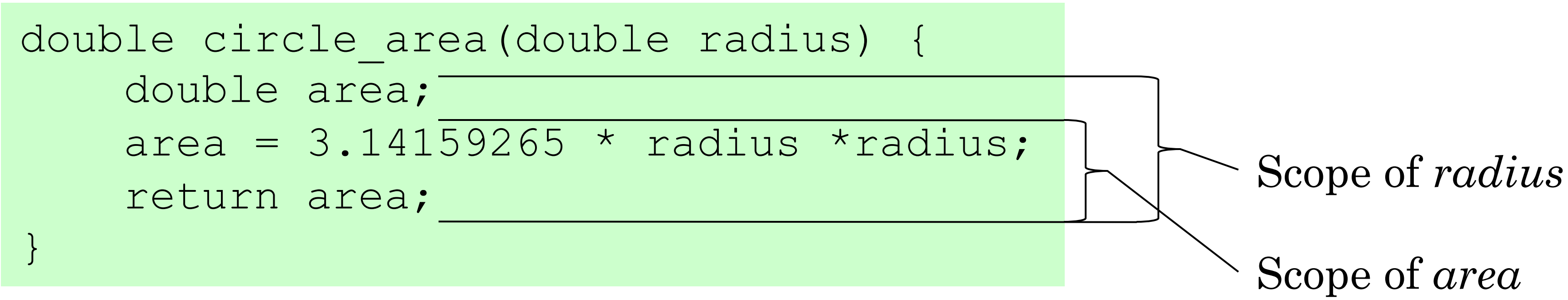
Derived Types

- array [ ]
- function ( )
- pointer *
- reference &

User-defined Types

- class
- struct
- union
- enum
- typedef

Variable-Scope



```
double circle_area(double radius) {
    double area;
    area = 3.14159265 * radius *radius;
    return area;
}
```

Scope of *radius*

Scope of *area*

A variable is said to be *“in scope”* when your program is at a point where it can see the variable.

A variable is *“out of scope”* when your program is at a point where it can’ t see the variable.

Variable Declarations and Definitions

Declarations and Definitions

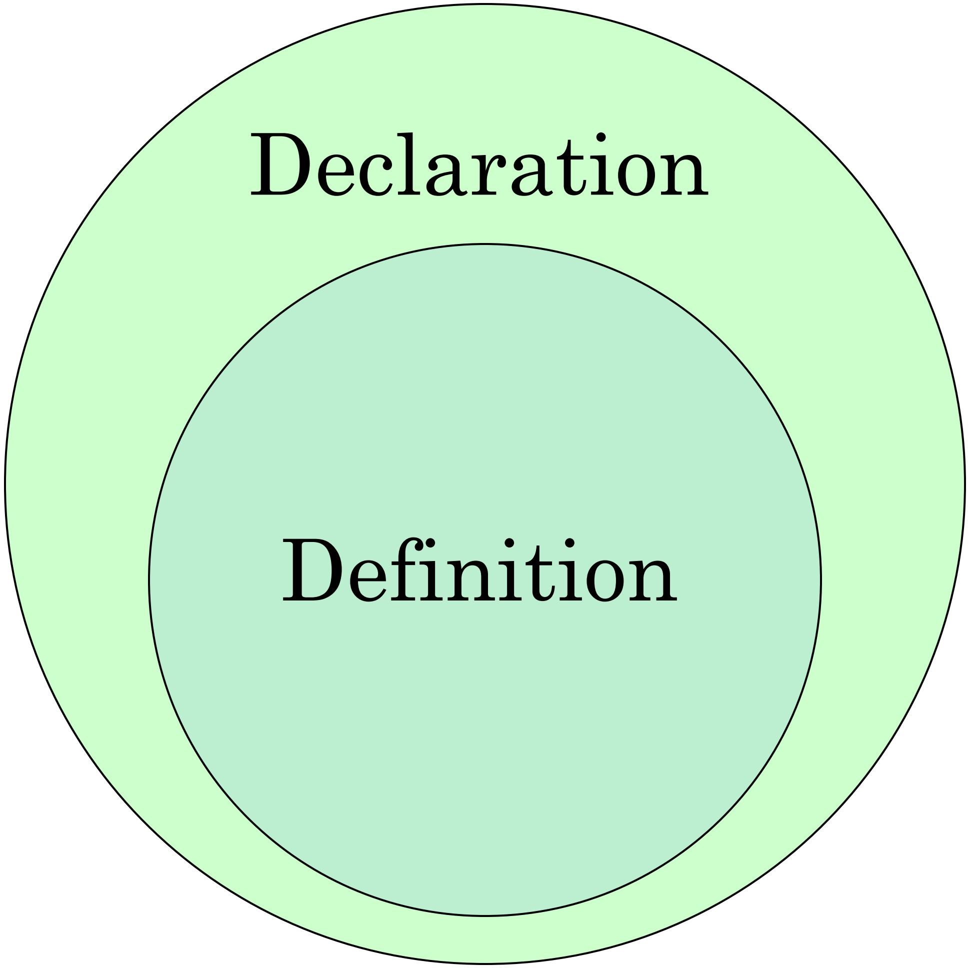
A variable-*declaration* simply proclaims the existence of a variable or a function.

Declarations and Definitions

A variable-*definition* is a *declaration* that also creates storage for a variable or function.

Declarations and Definitions

Declarations are usually definitions, but they don’t have to be.



Declarations and Definitions

Examples of declarations that are also definitions:

```
double acceleration;
```

```
double length(double x, double y){
    return sqrt(x*x + y*y);
}
```

Examples of declarations that are not definitions:

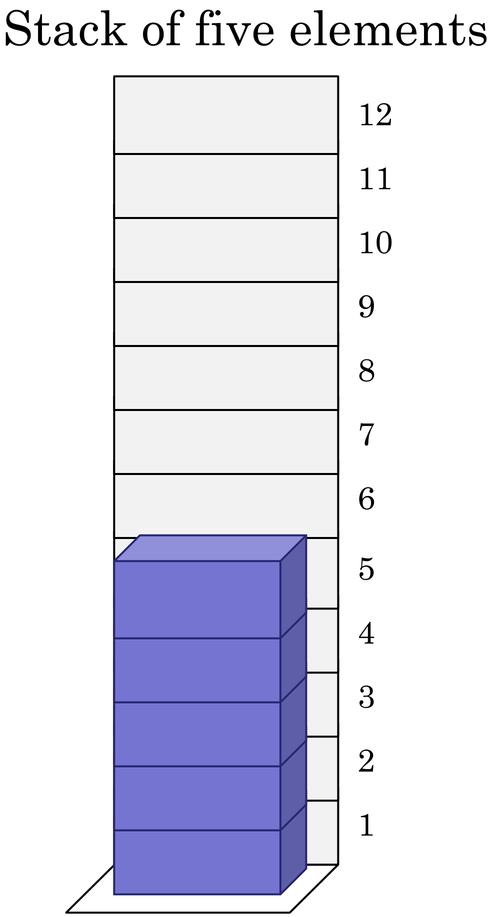
```
extern double distance;
```

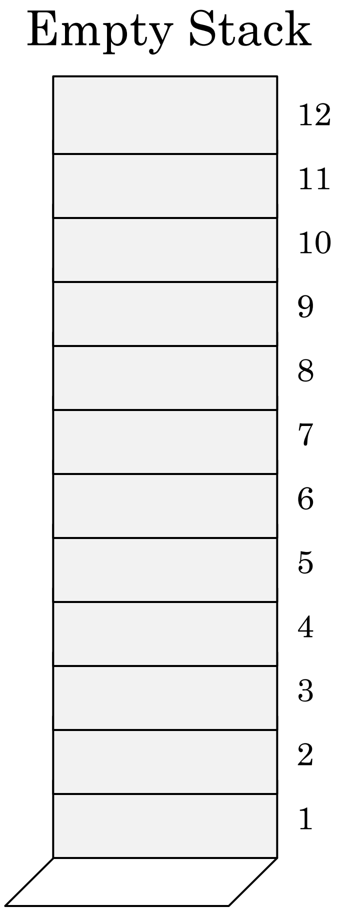
```
void length(double x, double y);
```

Automatic Storage

(The Stack)

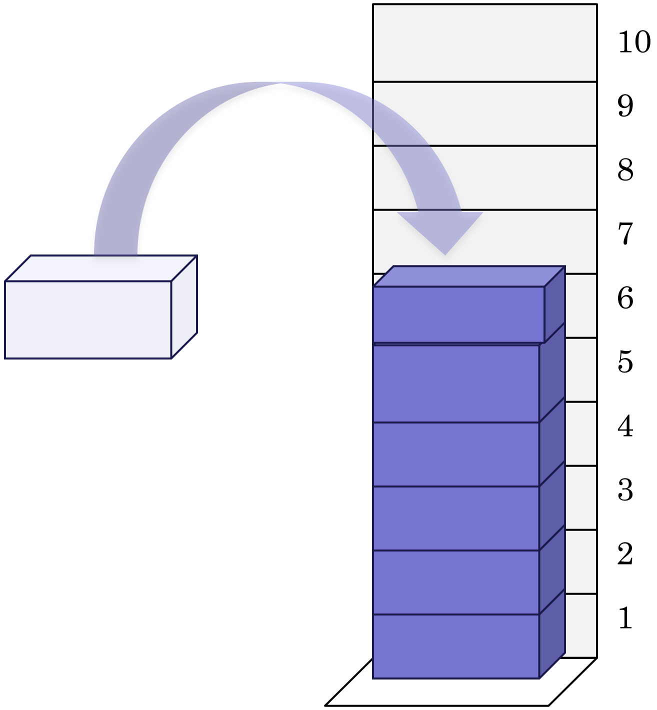
A Stack


Stack of five elements
12
11
10
9
8
7
6
5
4
3
2
1

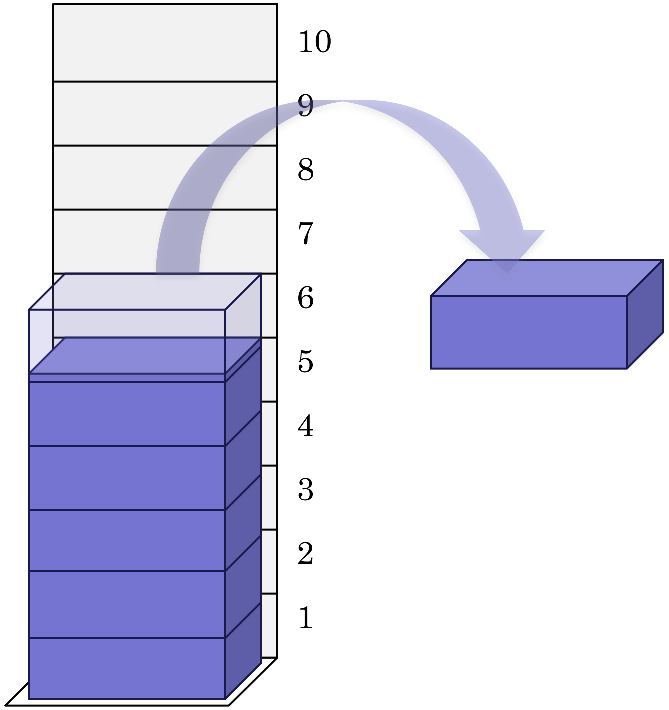

Empty Stack
12
11
10
9
8
7
6
5
4
3
2
1

A Stack

Items are “**pushed**” onto the stack



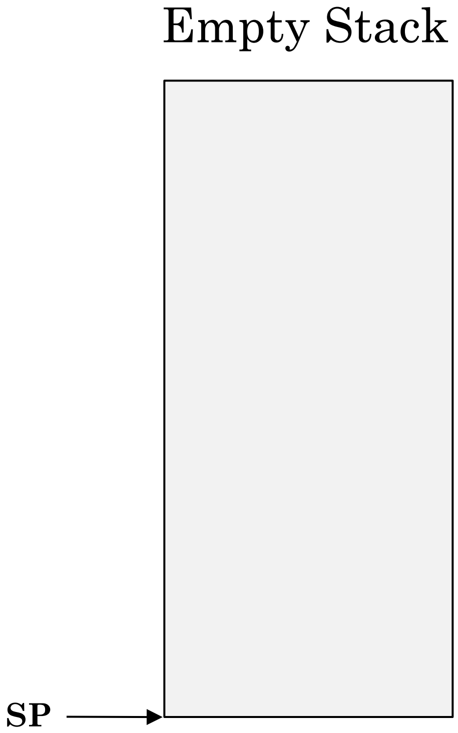
Items are “**popped**” off of the stack



- Sometimes rather than “push” or “pop”, people just say “allocate” or “deallocate” because its more general.
- Stacks are always LIFO (Last In, First Out).

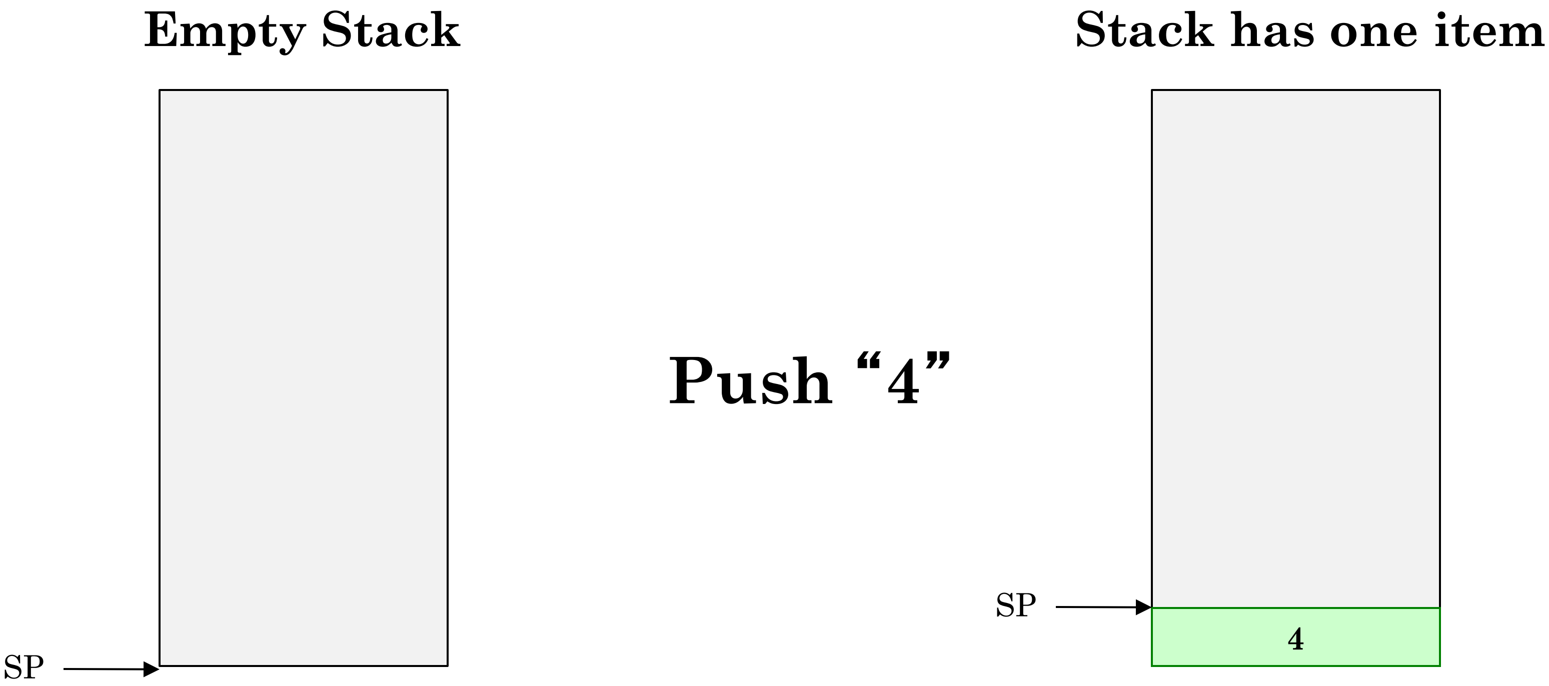
The Stack

In a program, the stack is represented by a chunk of memory and a *stack pointer* (SP). SP points to the "top" of the stack.



The Stack

To push a value onto the stack, a program writes the value to the top of the stack and updates SP to again point to the top of the stack.



How Function Calls Work

How Function Calls Work

Function calls use the stack to communicate with the function that they are calling.

A function call performs the following steps:

1. Pushes its parameters onto the stack.
2. Jumps to the function that it's calling.
3. Pops its parameters back off of the stack.

After a function call, the stack pointer will **always** be **exactly** where it was before the call.

What Functions Do When They are Called

Called-functions find their parameters on the stack.

A called-function :

1. Allocates it's local variables on the stack. Variables created in this way are called *automatic variables*.
2. Executes its code body.
3. Pops its local variables back off the stack.
4. Jumps back (returns) to the caller.

When we return from a function, the stack pointer will always be **exactly** where it was when we entered the function.

Function Call Example

Suppose we are calling the following function named *foo* as shown.

Function-call

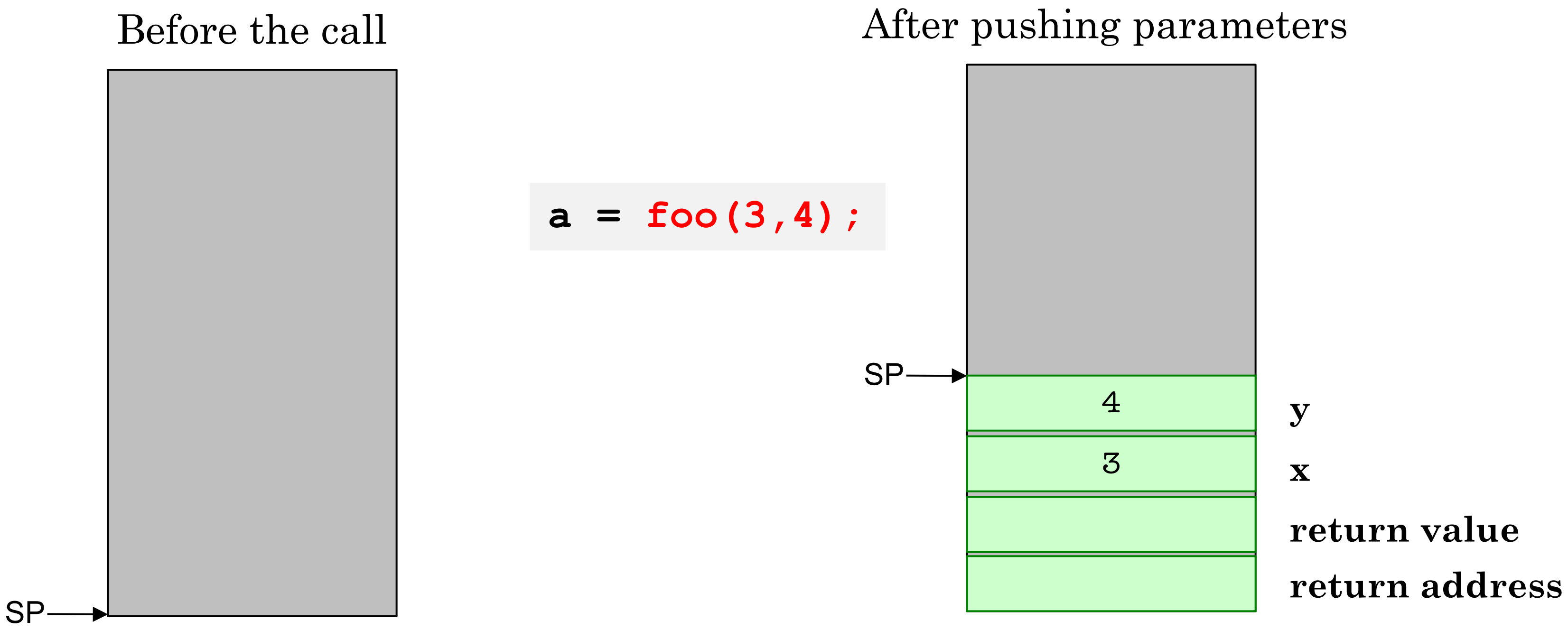
```
a = foo(3,4);
```

Called-function

```
int foo(int x, int y) {
    int z;
    z = x + y;
    return z;
}
```

Function Call Example

First, our function call will push its parameters onto the stack. It will also push a space for the return value, and return address.



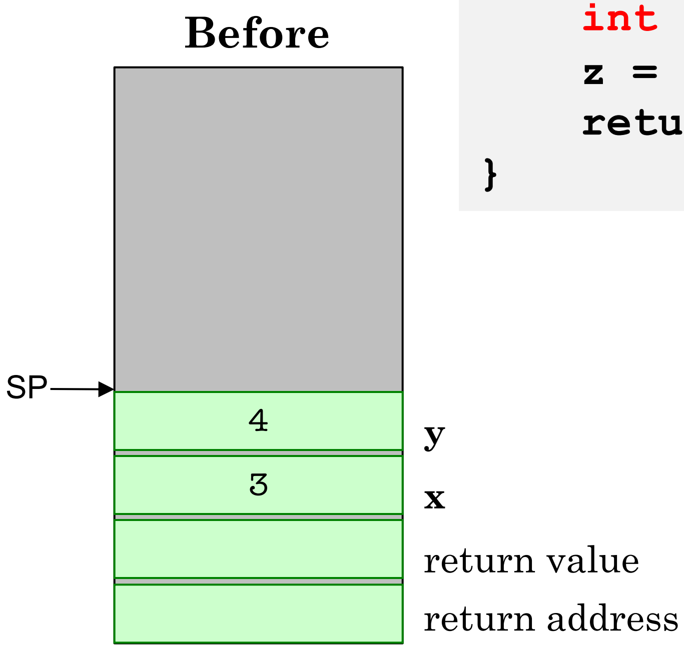
```
a = foo(3,4);
```

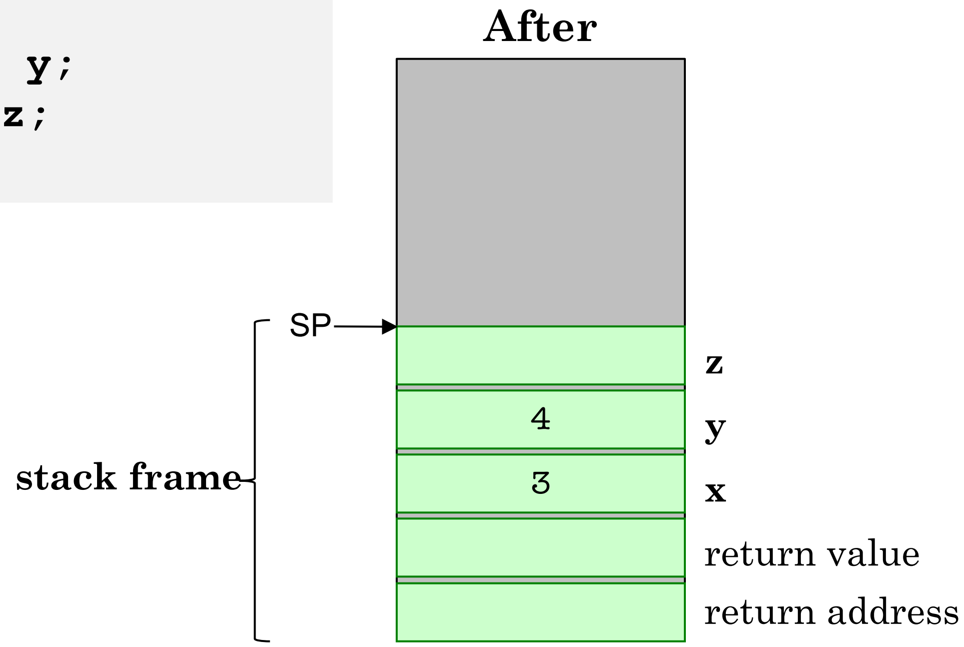
Second, the function call jumps to the function it is calling.

Function Call Example

The function first allocates its local variables on the stack. Here, space for the local variable z is allocated. At this point the function has everything that it needs on the stack. This is called the function's *stack frame*.

```
int foo(int x, int y) {
    int z;
    z = x + y;
    return z;
}
```

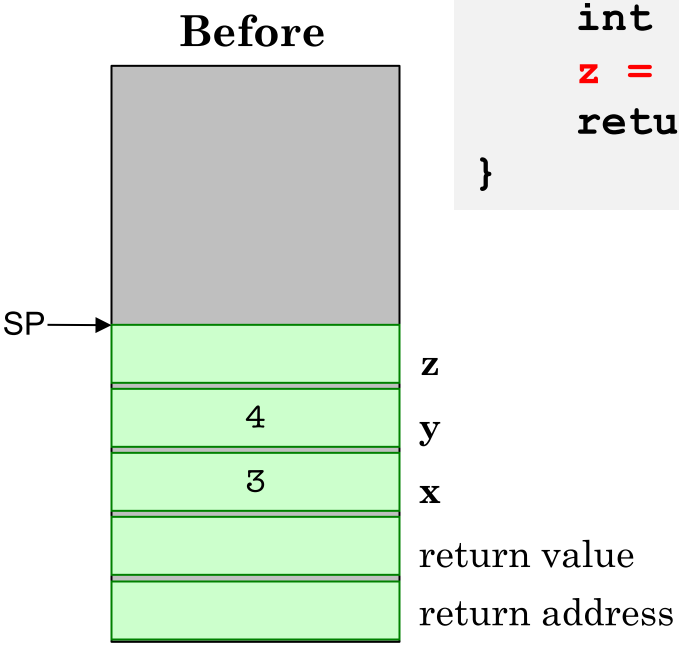


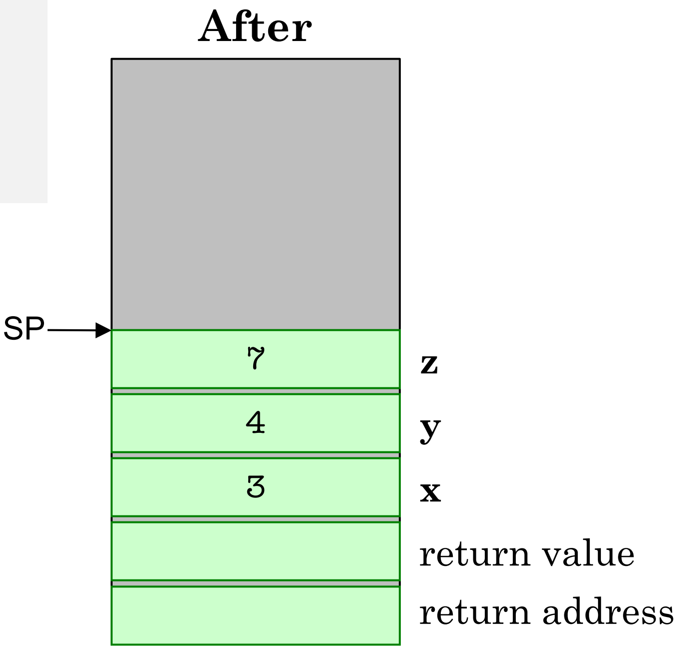


Function Call Example

The function then executes its statements. In this case x is added to y and the result is placed in z.

```
int foo(int x, int y) {
    int z;
    z = x + y;
    return z;
}
```

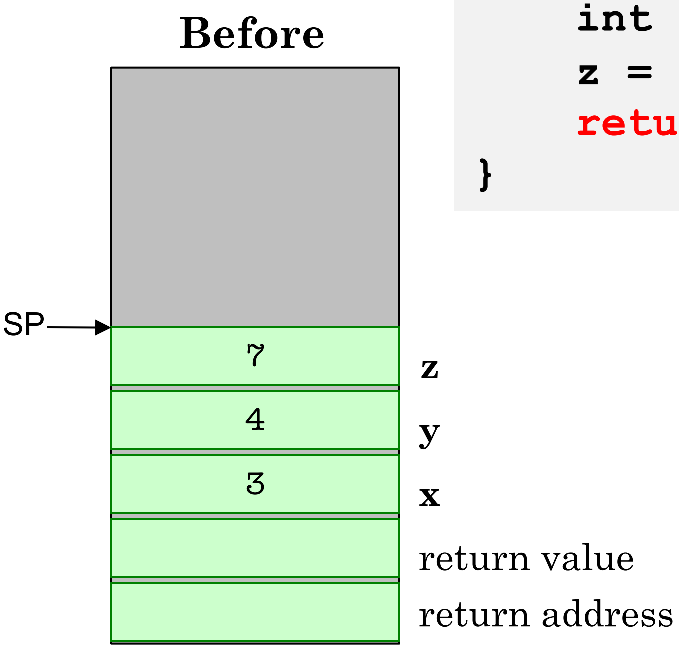


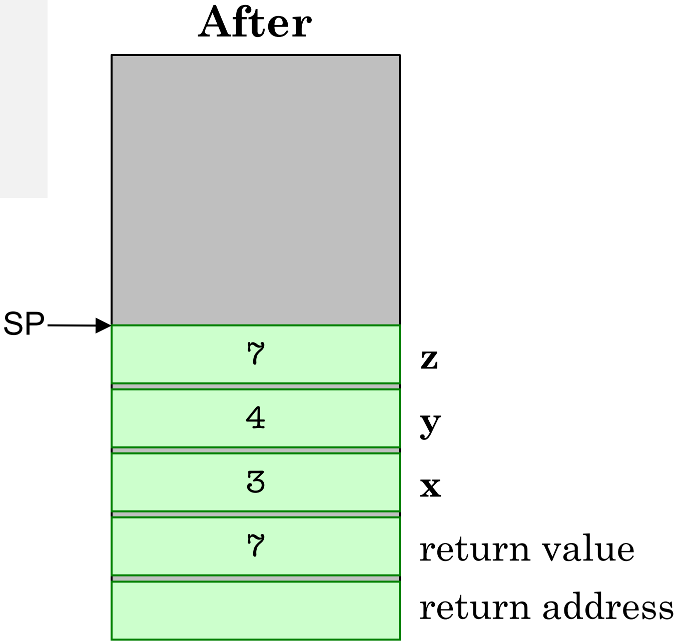


Function Call Example

Finally, the function executes its return statement by copying the value of z into the return value location and executing a JUMP back to the function call. **Remember, the function call is not done yet.**

```
int foo(int x, int y) {
    int z;
    z = x + y;
    return z;
}
```

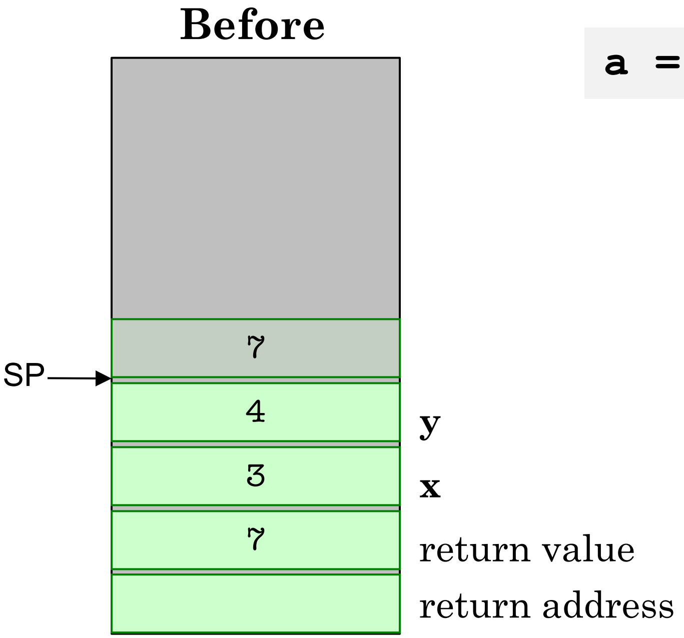


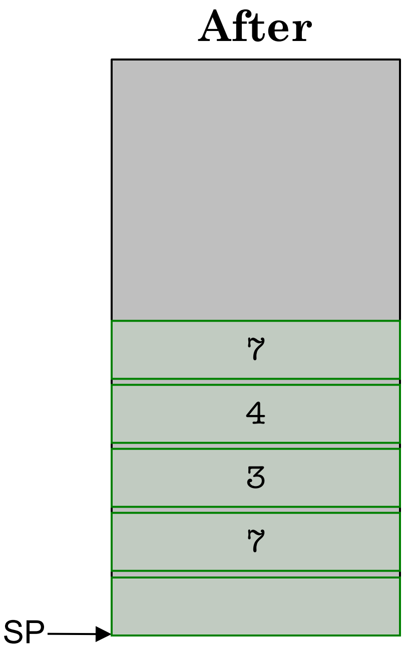


Function Call Example

Finally, the function call moves the stack pointer to where it was before the call.

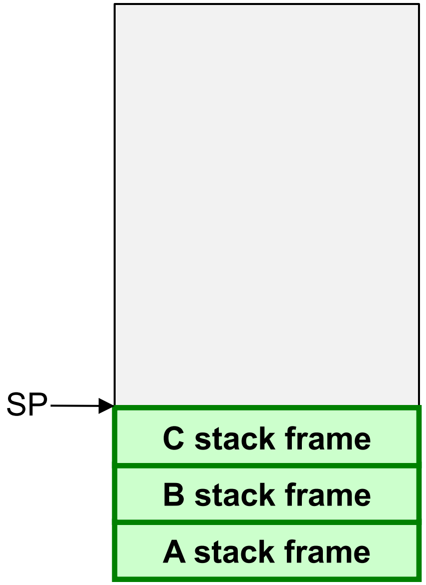
```
a = foo(3,4);
```





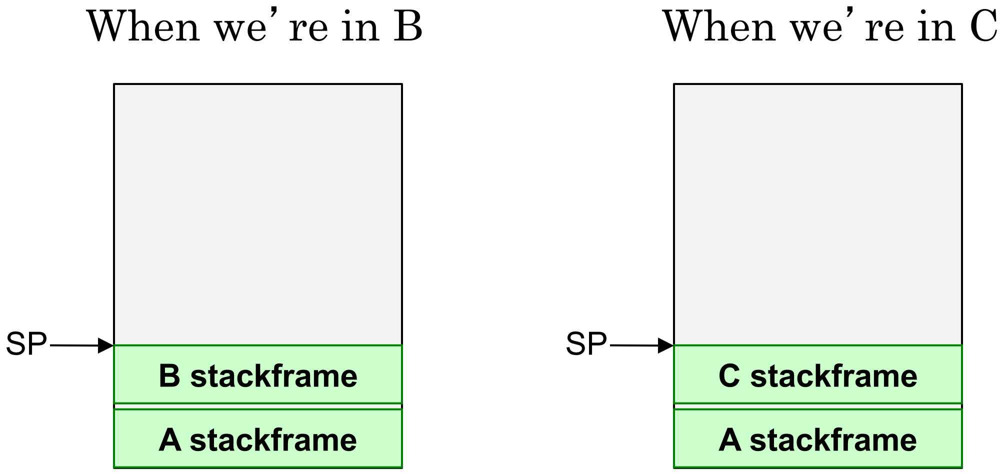
Multiple Stack Frames

Suppose function A calls function B, and function B calls C. When our program is running code in function C, the stack will look like:



Multiple Stack Frames

Suppose function A calls B, and then A calls C.



Notice that C is using the **same** part of the stack that B **was** using.

Things to Realize About the Stack

- A function's local, automatic variables are created on the stack when they are defined, thus beginning their lifetimes.

- A function's local, automatic variables are removed from the stack when it exits, thus ending their lifetimes.

Things to Realize About the Stack

- If your program has entered a function, but not yet exited that function, then it has a stack frame on the call stack.

- During code execution, the call stack represents the hierarchy of function calls currently in effect.

Things to Realize About the Stack

Don’t return a pointer to a local variable back to the function’s caller. Remember that once the function returns, the life of the local variable is over. It should not be treated as if it were still in scope.

It’ s OK to pass a pointer to a local automatic variable to a function that you are calling but.

What's wrong with this?

```
#include <stdio.h>

char* make_line(int n) {
    char temp[40];
    sprintf(temp,"%d bottles of beer on the wall",n);
    return temp;
}

int main(int argc, char* argv[]) {
    char *line;
    line   = make_line(100);
    printf("%s\n", line);
}
```

How can we fix it?

Dynamic Storage

(The Heap)

The Heap

- The *heap* is the region of memory from which `malloc` and `new` dynamically allocate smaller chunks of memory.

- Variables allocated from the heap are called *dynamic variables*.

The Heap

- Calling `malloc` or `new` allocates a portion of memory, and returns a pointer to it.
  - The pointer provides scope to the allocation.

- Calling `free` or `delete` on the pointer deallocates the previously allocated memory, and invalidate the pointer.
  - A pointer variable containing an invalid reference is called a *dangling* pointer.

Things to Realize About the Heap

- A dynamic variable without scope (we've somehow lost our reference to it, oops!) is called a *memory leak*.
- It's a bad idea to intermix the usage of `malloc` and `free` with `new` and `delete`.
- Using an invalid pointer is … a bad thing.

Static Storage

Static Variables

The lifetime of a **static** variable extends throughout the entire the lifetime of the program.

Static Variables

A static global variable is in scope within the translation unit in which it is defined. A translation unit is a source code file plus any header files that are included. It can be brought into the scope of other translation unit using the **`extern`** keyword.

A static local variable is in scope within the function in which it's declared. It, and its value persists between calls to the function.

Code Storage

(The *code* segment, also known as the *text* segment)

Code Storage

The code segment contains your compiled code, and global constant objects. It’s read-only. That’s it!

Terms

- **variable (reference, data-type, scope, lifetime)**
- **reference (name, address)**
- **data-type (int, float, double, … , user-defined types, …**
- **scope (local, global)**
- **lifetime (static, dynamic, automatic)**
- **declaration**
- **definition**
- **the stack (automatic variables)**
- **stack pointer**
- **stack frame**
- **automatic variable**
- **dynamic variable**
- **static variable**
- **local variable**
- **global variable**
- **the heap (dynamic variables)**
- **dangling pointer**
- **memory leak**

The End