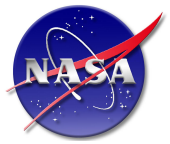
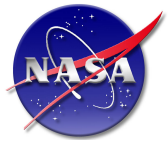


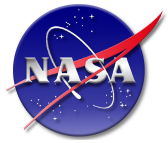
The Basics of C/C++

Program Memory and Variables

John Penn (L-3Com/ER7)



Variables and Vocabulary



Attributes of Variables

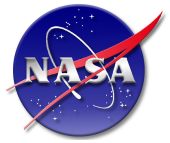
A **variable** is a portion of memory used by a program to store data values.

Attributes of a variable are:

Reference – provides access to the variable. A reference can be :
a. name or
b. address.

Data Type – a description of how data is represented within the variable. For example: int, double[10], MyClass

Lifetime – the period during which it is valid to access the variable.



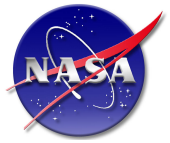
Variable Reference Examples

```
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.



Scope

Variables have **scope**.

Scope – where in a program a variable can be "seen".

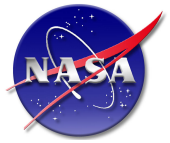
```
double circle_area(double radius) {
    double area;
    area = 3.14159265 * radius *radius;
    return area;
}
```

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.

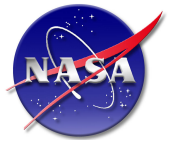
A variable is **out of scope** when your program is at a point where it can't see the variable.



Association of Scope and Lifetime

A variable's **lifetime** and its **scope** are strongly associated. Scope implies life and life (usually*) implies scope.

* static variables go in and out of scope during their lifetimes.

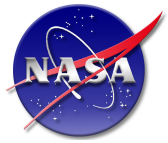


Declarations and Definitions

A **declaration** proclaims the existance of a variable or function.

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

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



Declarations and Definitions

Examples of declarations that are also definitions:

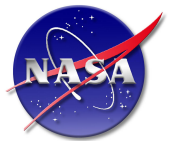
```
double acceleration;
```

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

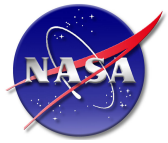
Examples of declarations that are not definitions:

```
extern double distance;
```

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



How Programs Work



What happens when you run a program:

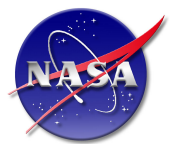
The shell (csh, bash, etc.) invokes the **program-loader** to:

1. Allocate memory for the program.

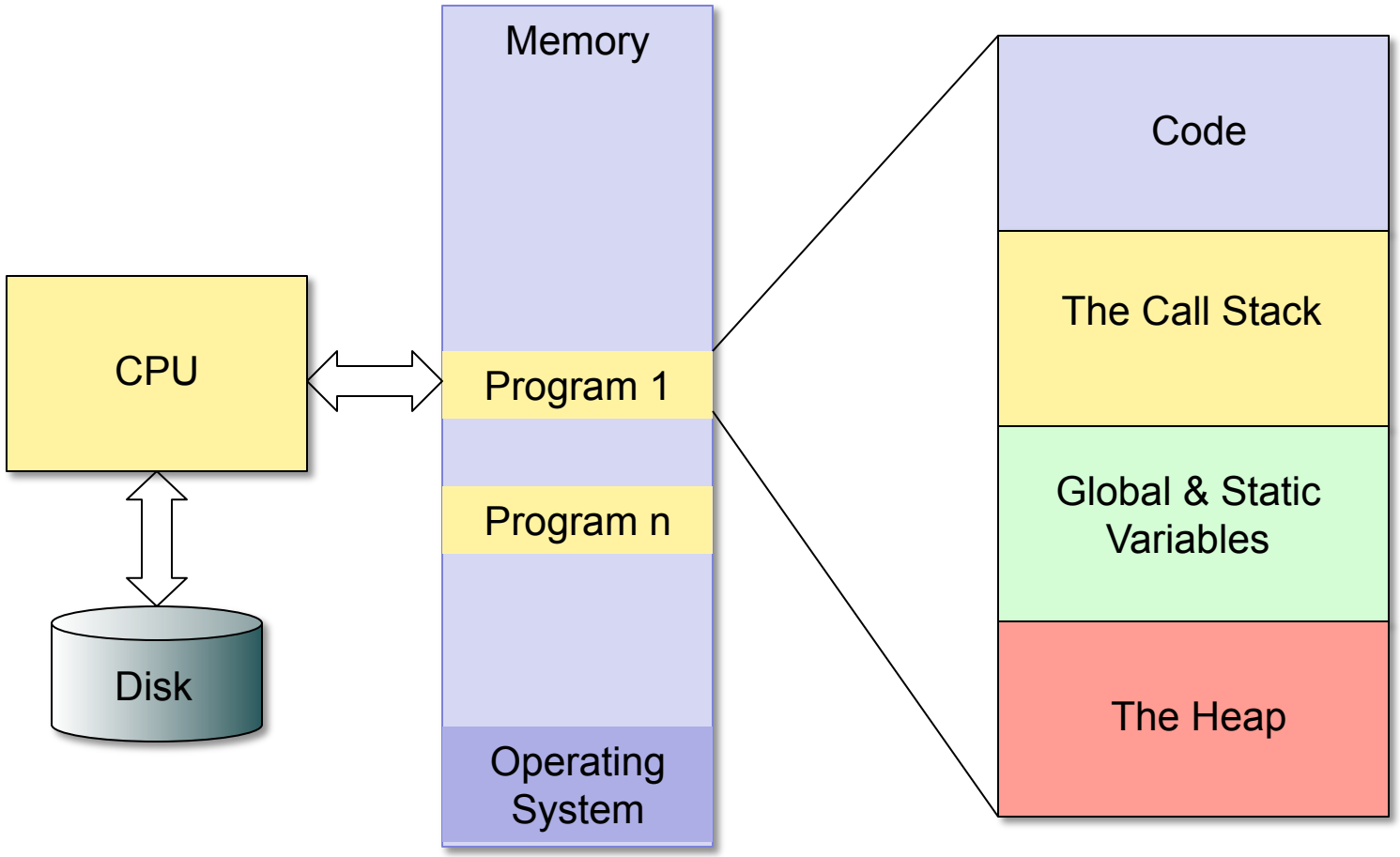
2. Copy the contents of the program file into the memory.

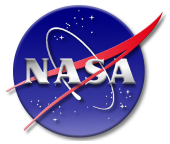
3. Setup the program so it's ready to run.

The operating system's process scheduler then calls the program's **main()** function.

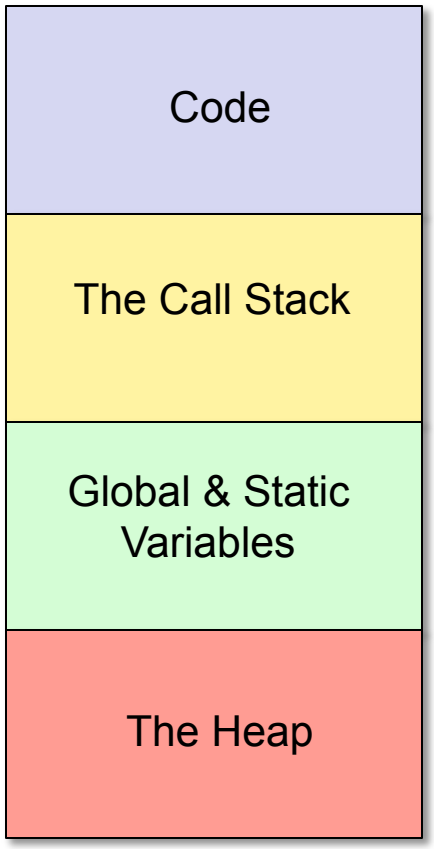


A Program in Memory





Program Memory

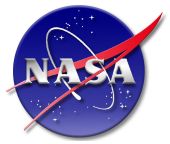


This is set at load-time. It cannot be altered at runtime.

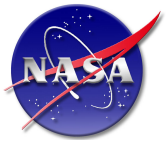
Used at run-time for function parameters and automatic local variables.

Variables in this memory are live all during runtime.

Memory is allocated from here when you call malloc() or new().

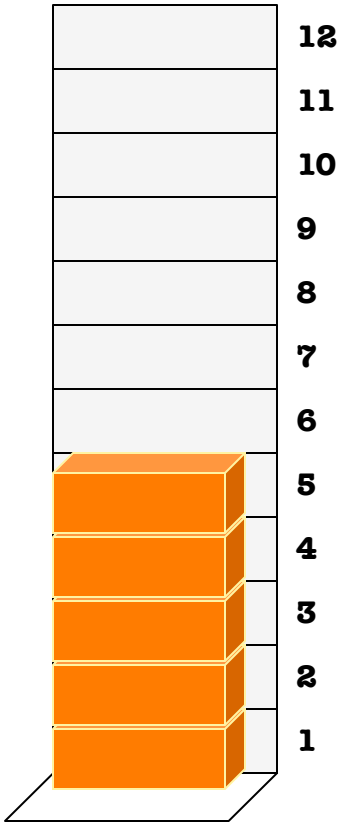


The Call Stack

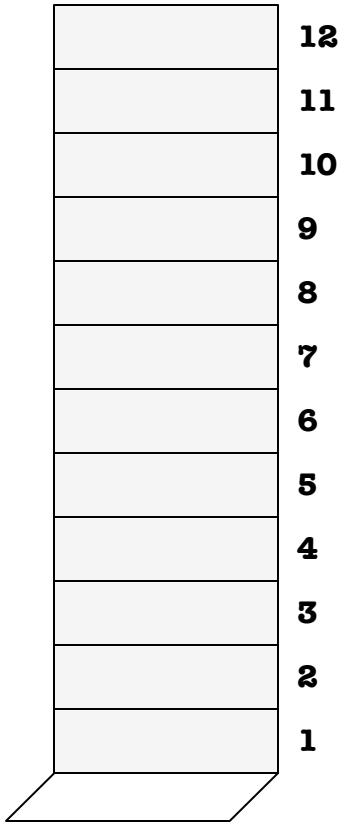


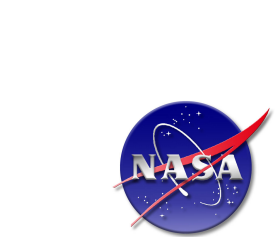
A Stack

Stack of five elements



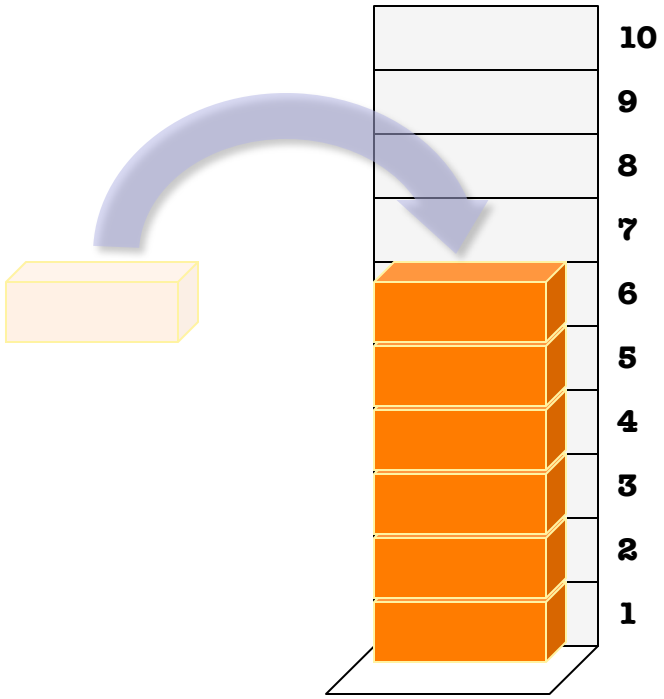
Empty Stack



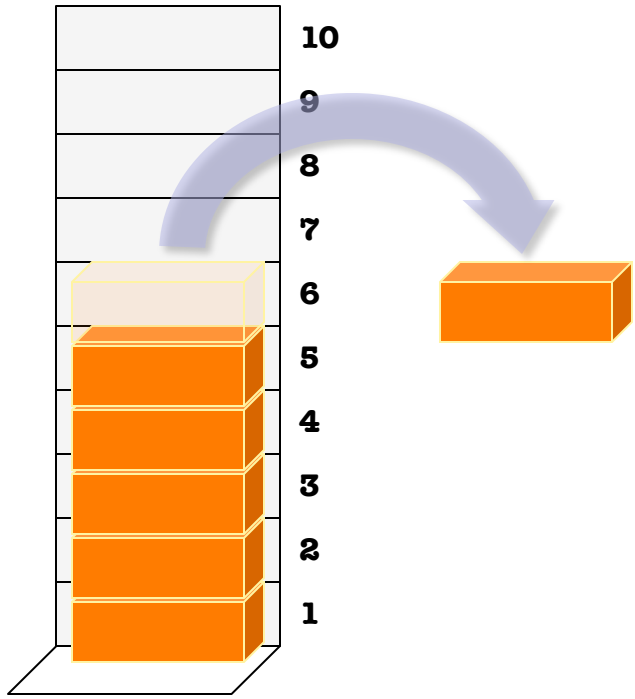


A Stack

Items are “pushed” onto the stack

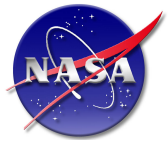


Items are “popped” off of the stack



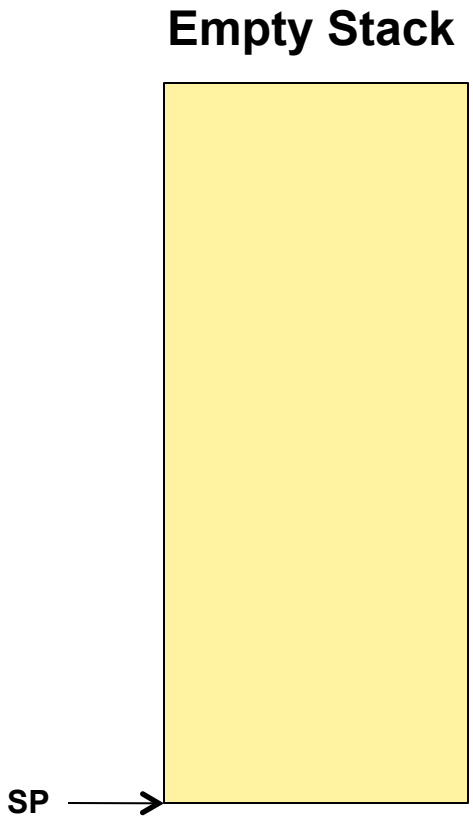
Sometimes rather than “push” or “pop”, people just say “allocate” or “deallocate” because its more general.

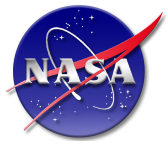
Stacks are always “Last In, First Out”.



The Call Stack

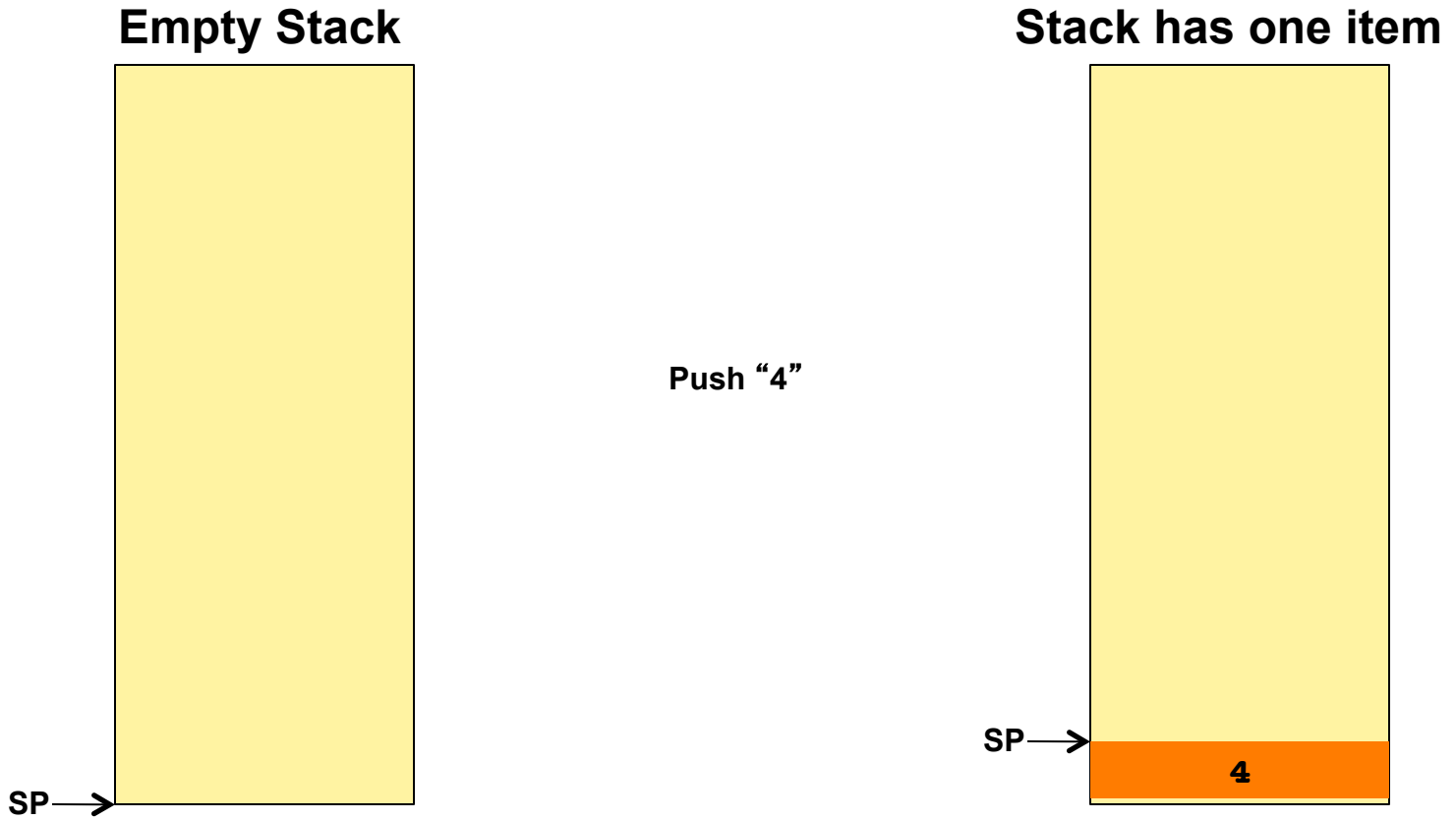
In a program, a Call Stack is represented by a chunk of memory and a **Stack Pointer** (SP). SP points to the "top" of the stack.

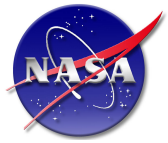




The Call Stack

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.





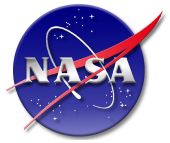
How Function Calls Work

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

A function call :

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.



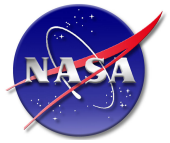
What Functions Do When They Are Called

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

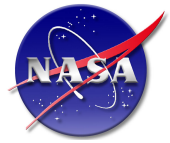
Suppose we are calling the following function named foo as shown.

Function-call

a = foo(3,4);

Called-function

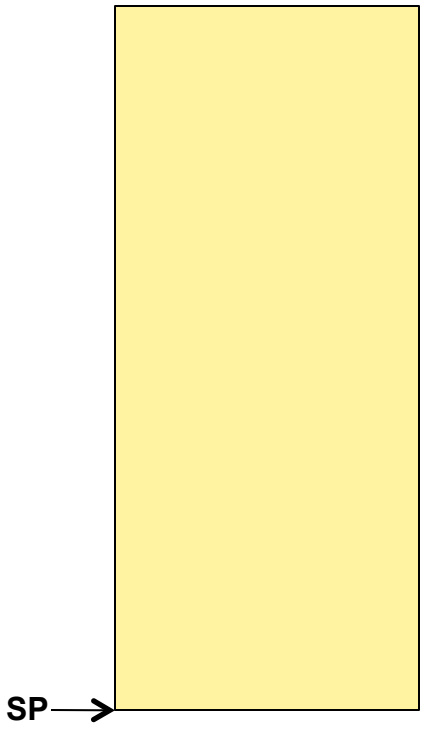
```
int foo(int x, int y) {
    int z;
    z = x + y;
    return z;
}
```



Function Call Example

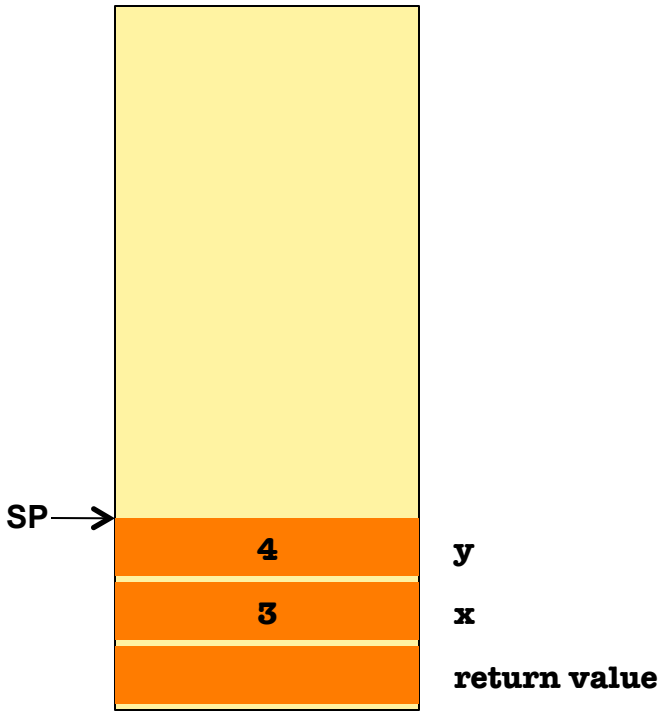
First, our function call will push its parameters onto the stack. It will also push a space for the return value.

Before the call



a = foo(3,4);

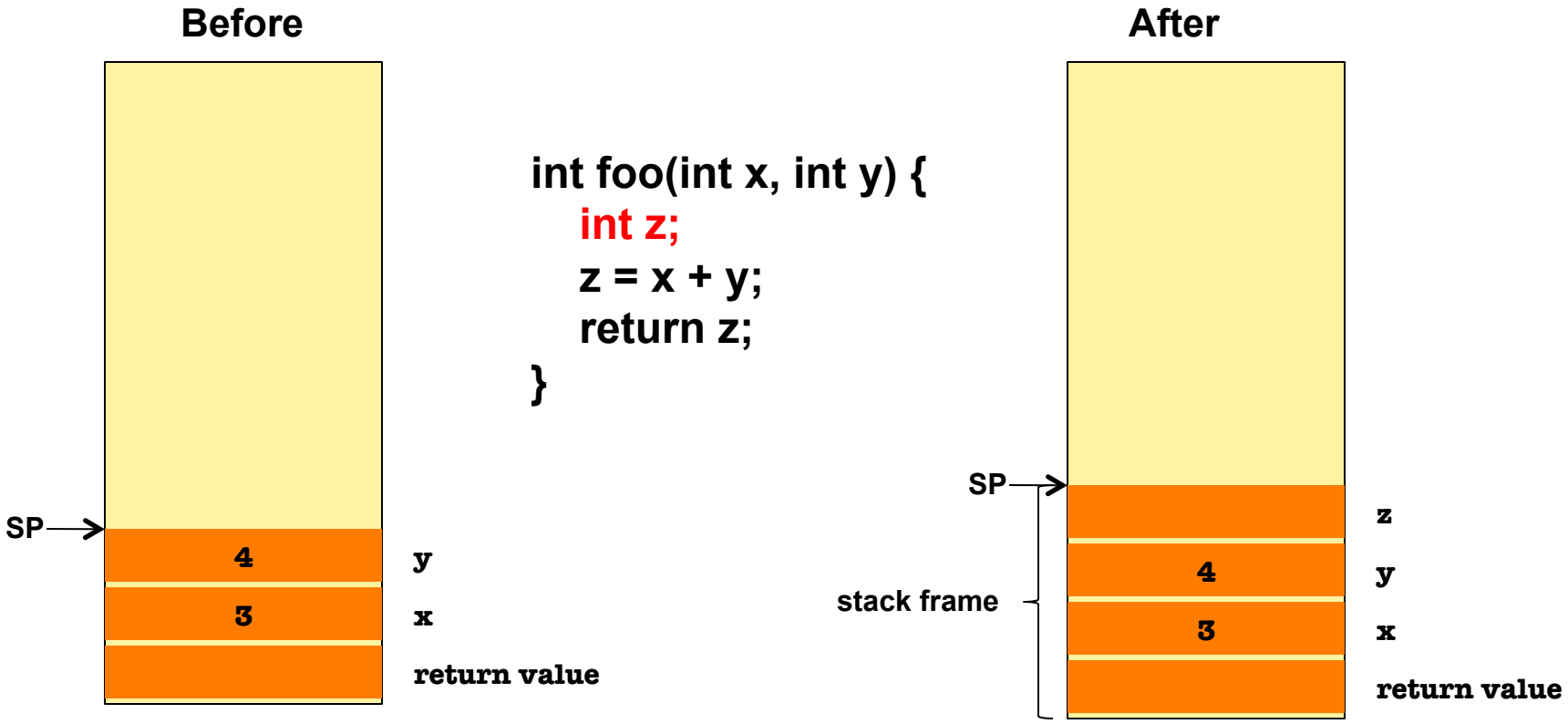
After pushing parameters



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

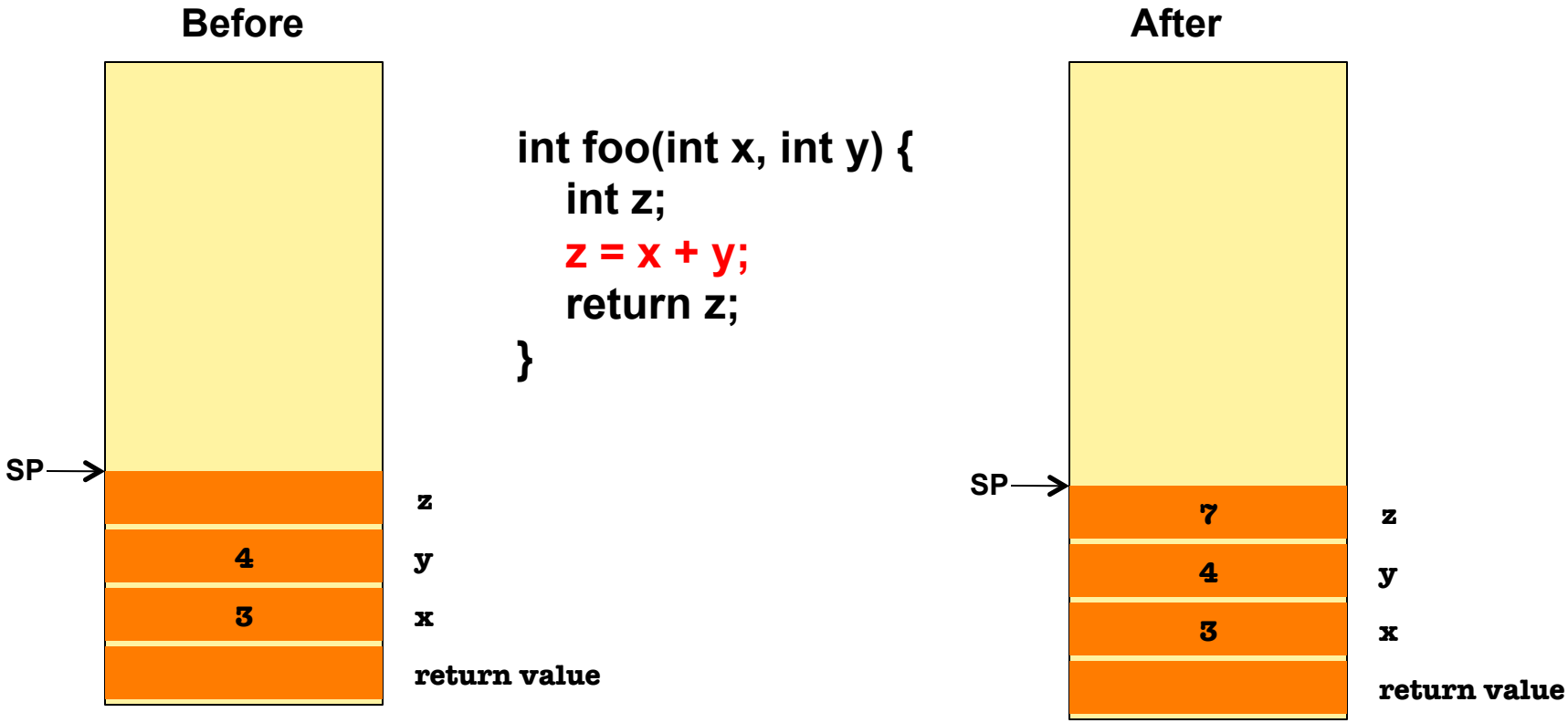
Function Call Example

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**.



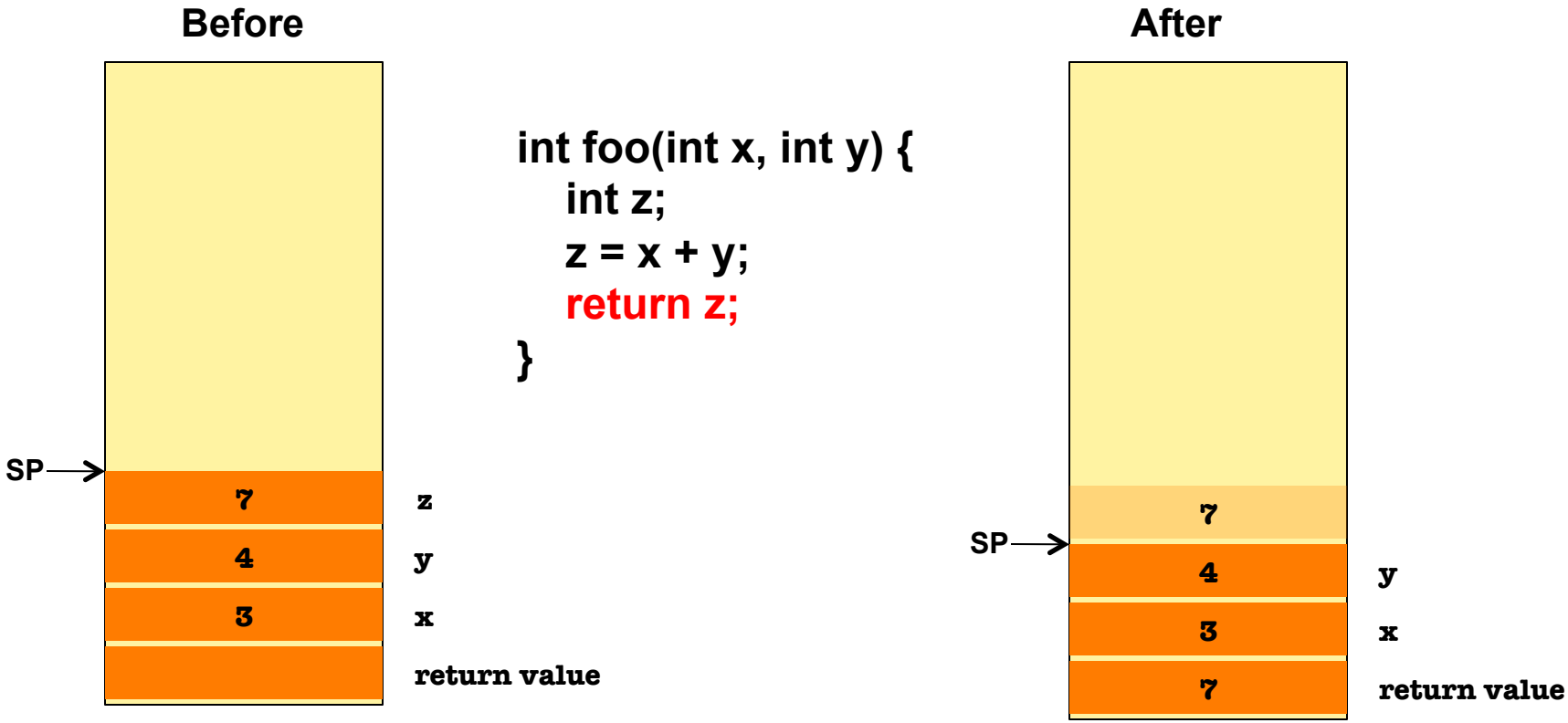
Function Call Example

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



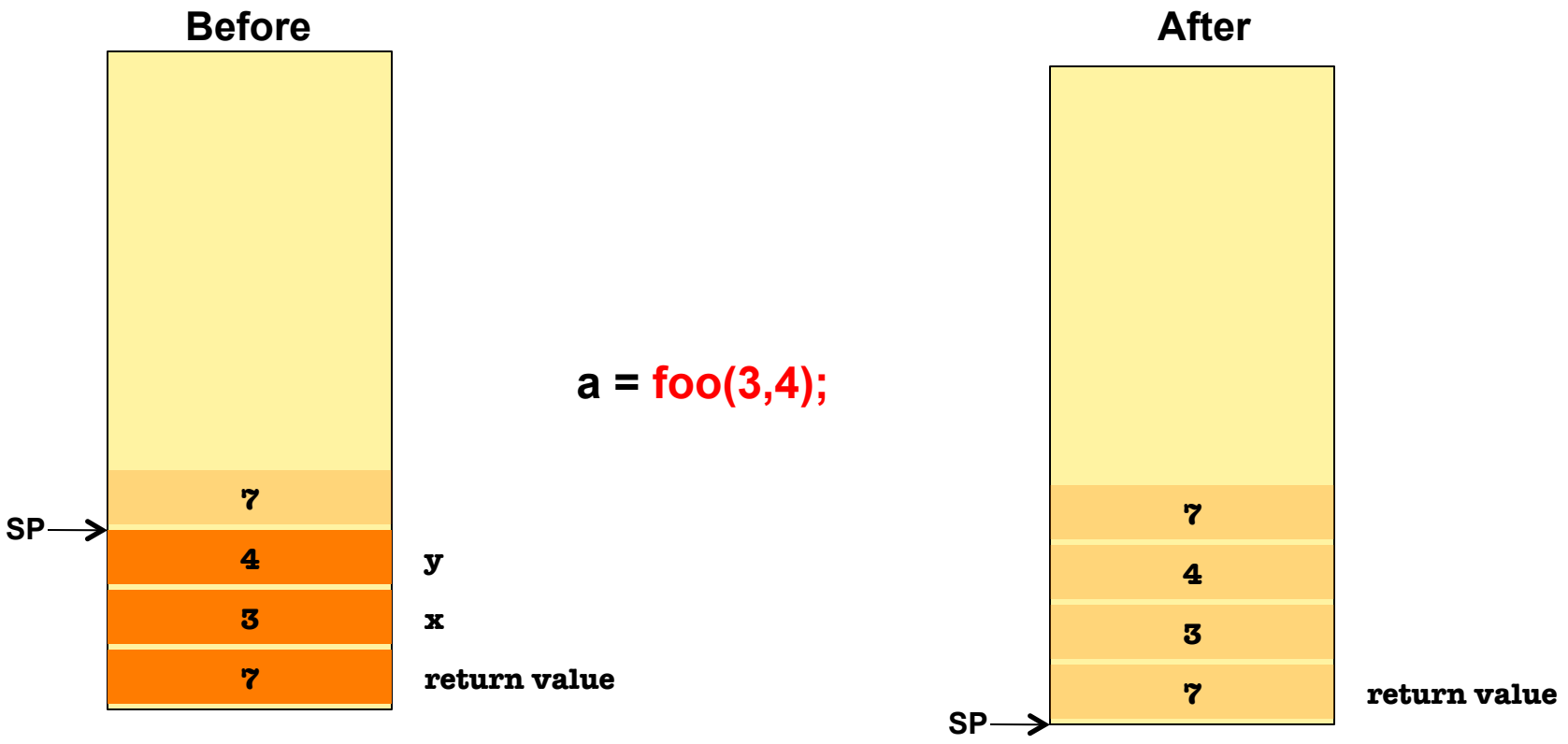
Function Call Example

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.

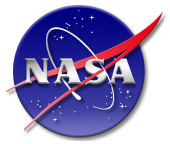


Function Call Example

Finally, the function call pops the parameters back off the stack.

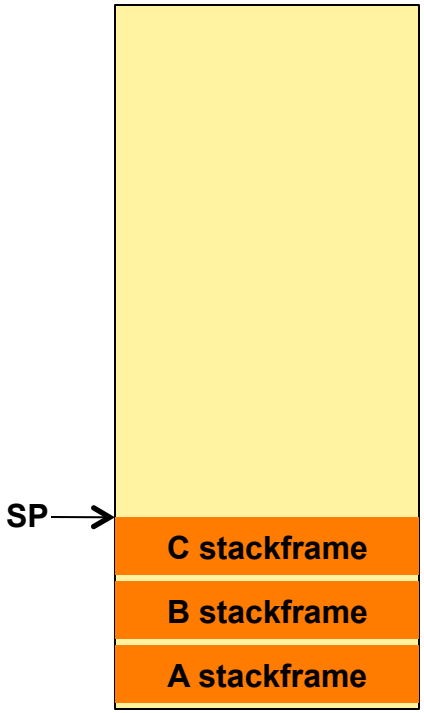


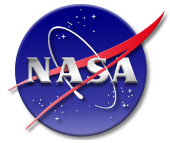
The last value popped is the return parameter.



Multiple Stack Frames

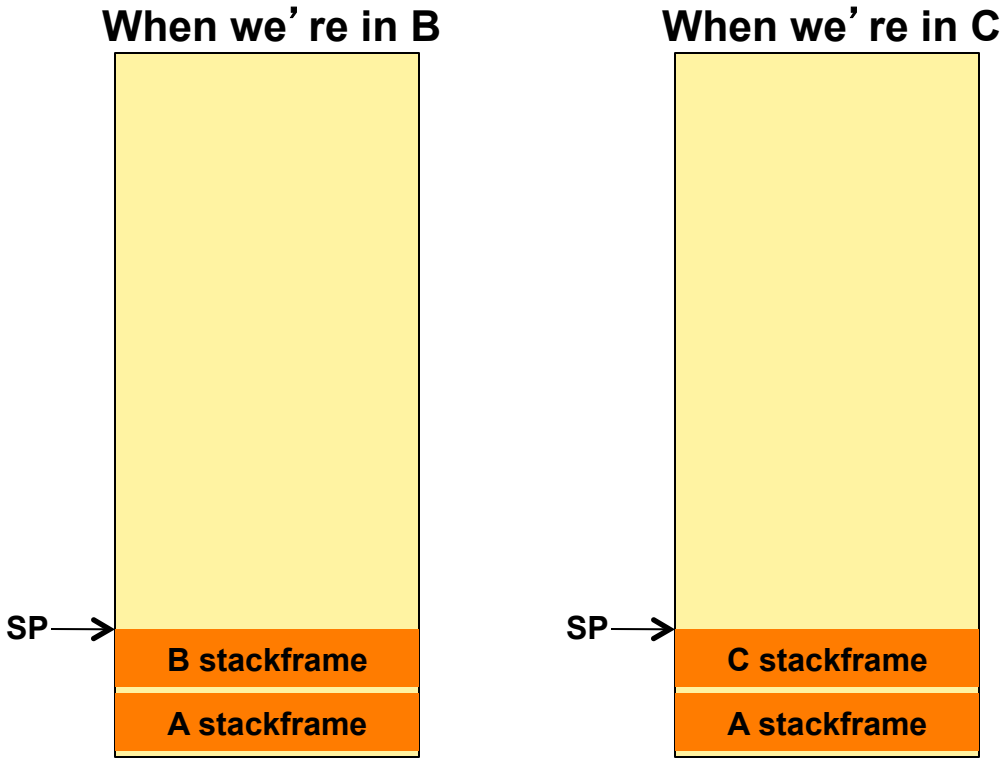
Suppose function A calls function B, and function B calls C. When our program is running code in function C, the stack will look like:



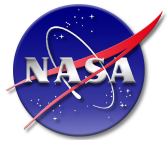


Multiple Stack Frames

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

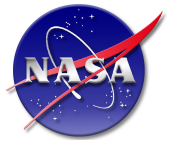


Notice that C is using the **same** part of the stack that B **was** using.



Things to Remember About the Stack

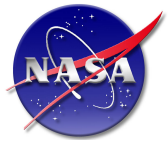
- A function's local variables are created on the **Stack** when they are defined.
- A function's local variables are removed from the **Stack** when it exits.
- If your program has entered a function, but not yet exited that function, then it has a stack frame on the call stack.
- It's OK to pass a pointer to a local variable to a function that you are calling but, **NEVER** 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.
- A program's stack is pretty big, but it's not limitless. It is possible to overrun the stack, especially in recursive functions that don't achieve their termination case.



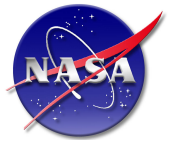
What's wrong with this?

```
#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?



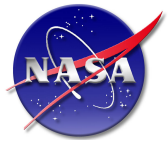
The Heap



The Heap

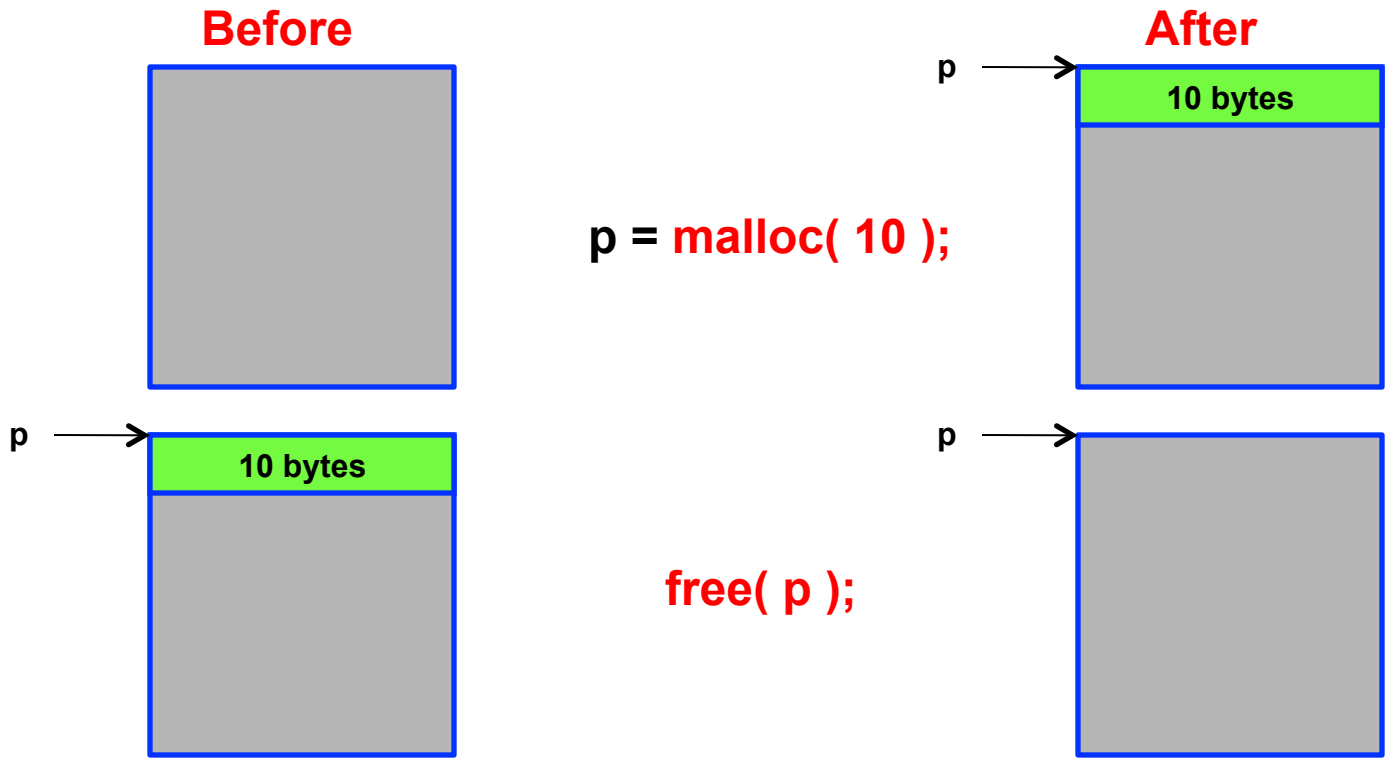
The **heap** is the chunk of memory from which **`malloc`** and **`new`** dynamically allocate smaller chunks of memory (variables).

	Allocation	Deallocation
C/C++	**malloc**	**free**
C++ Only	**new**	**delete**

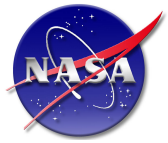


The Heap

malloc and **new** each allocate memory from the heap, and return a pointer to that allocation. Variables allocated from the heap are called **dynamic variables**.



Note that p's value (even though it hasn't changed) is invalid after **free** is called. So, don't attempt to use it or you will likely get a segmentation violation.



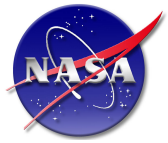
A pointer to a dynamic variable is invalid after **free** or **delete** is called.

A pointer variable containing an invalid reference is a **dangling** pointer.

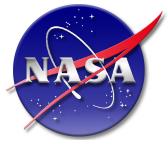
A dynamic variable without scope (we've somehow lost our reference to it, oops!) is called a **memory leak**.

Because the heap is limited in size, you might consider checking that the pointer returned by **malloc** is not NULL. **new** throws an exception.

It's generally a bad idea to intermix the usage of **malloc** and **free** with **new** and **delete**.



Static & Global Variables



Static & Global Variables

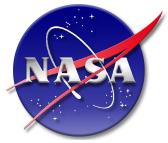
Lifetime of **static** and **global** variables extends throughout the entire the lifetime of the program.

Global variables are in scope within the **translation unit** in which they are defined. They can be brought into the scope of other translation units using the **extern** keyword.

Translation unit – a source code file plus any files that are #include'd.

The scope of a static global variable is only the translation unit in which it's declared.

The scope of a static local variable is only the **function** in which it's declared.



The End