C Review

Functions

K&R C (old)	ANSI C
Declaration	Prototype

K&R C (old)

Declaration

```
int fcomp();
```

Definition

```
int fcomp(a,b)
    int a;
    int b;
{
   …
}
```

ANSI C

Prototype

```
int fcomp(int a, int b);
            or
int fcomp(int, int);
```

← preferable

The use of a function requires a prototype. For functions like printf(), the function prototypes are contained in stdio.h or related headers.

Definition

```
int fcomp(int a, int b){
…
}
```

Not so much an issue now, but problems can occur if one mixes K&R C with ANSI C in function declarations and definitions in separate files. This is because under K&R C, type promotion can occur because the compiler makes everything a standard size, int, double, pointer (and thus types might have changed from what you expected)

C Review

```
#include <stdio.h>

#define  abs(t)       (((t)>=0)?(t):-(t))
#define  square(t)   ((t)*(t))

double sqr(double);

void main(void) {

    int i;
    double c;

    for( i=1; i<11; i++){
        c = sqr(i);
        printf(" \t  %d  \t  %f  \t  %f  \n", i, c, square(c));
    }

}
```

Notice parenthesis around the variable t and body of macro

No semicolon

Square Root Function Prototype

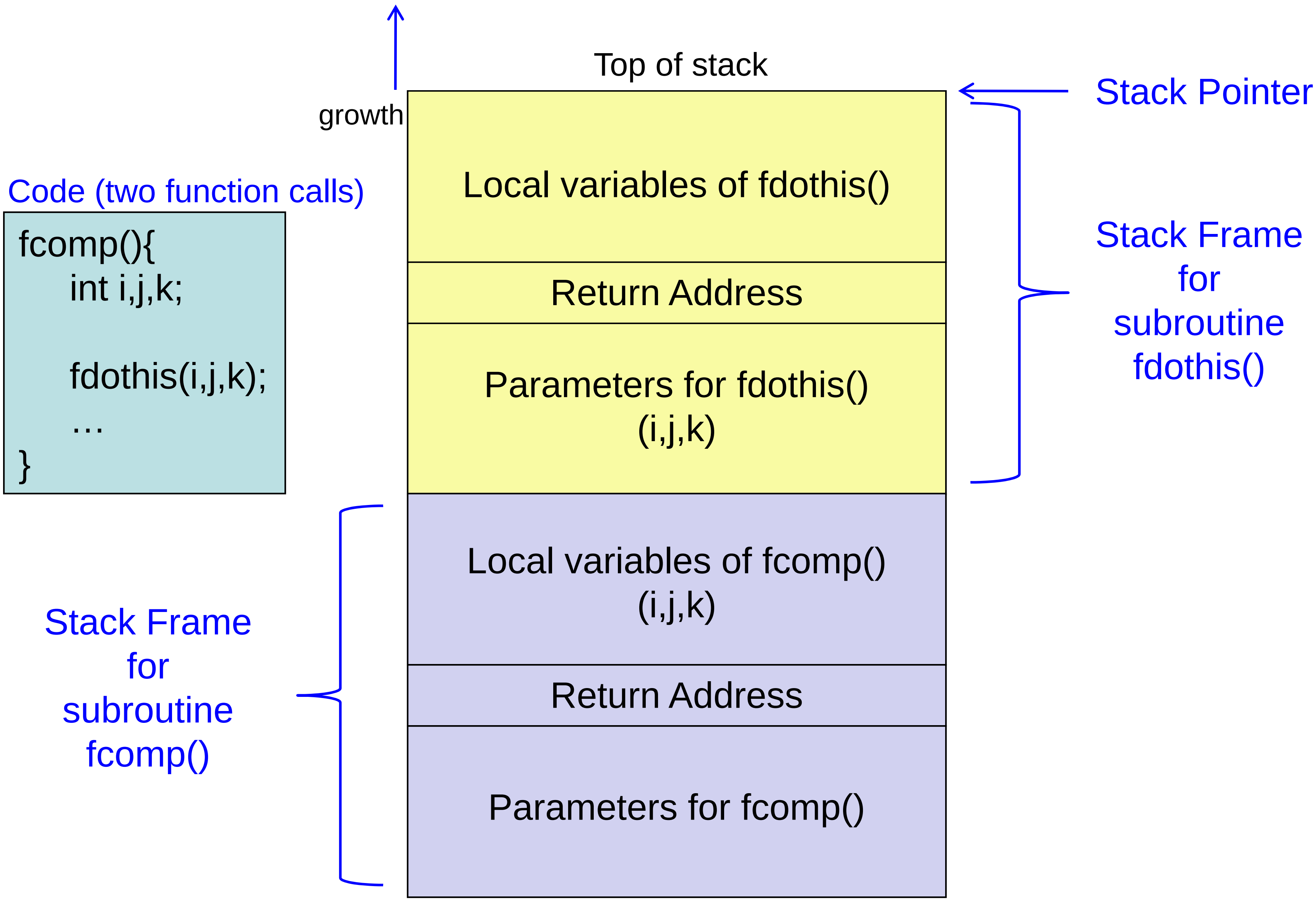
```
double sqr( double x ){
    double x1=1;
    double x2=1;
    double c;
    do{
        x1=x2;
        x2=(x1 + x/x1)/2;
        c = x1-x2;
    }while( abs(c) >= 0.00000001);
    return x2;
}
```

Why not:
while(abs(x1-x2) >=…) ?

Because the macro would cause the argument x1-x2 to be evaluated three times. (speed issue)

C Review

The function call stack



C Review

C requires a function to have an argument list
even if there are no arguments

If f is a function;

```
f();
```

is a statement that calls the function, but

```
f;
```

does nothing at all.

More precisely, it evaluates the address of the function
but does not call it.

C Review

Recursion

C permits a function to call itself

Example: factorial

n! = n * (n-1) * (n-2) * … * 2 * 1;

n! = n * (n-1)!

```
long fact( int n ) {
    if (n == 0){
        return 1;
    }else{
        return n*fact(n-1);
    }
}
```

Hardware/Software Tradeoff

Recursion is not used much in embedded systems because of limited stack space. Recursion can cause stack overflow problems.

C Review

Variable number of arguments

Suppose you wanted to create a function that could take an arbitrary number of parameters.

```
double ave(int N, double a1, double a2,  …. double aN)
```

```
double ave(int N, …) // This is how your function would be written in C
{
…
}
```

You will need to use the following include file and functions

```
# include <stdarg.h>  // necessary header file

void va_start(va_list  ap,  parmN);  // parmN is the last fixed parameter

type va_arg(va_list  ap,  type);  // returns the next type value

void va_end(va_list  ap);  // called when all the arguments have been processed.
```

C Review

Variable argument example

```
#include <stdarg.h>

double average(int count, ...) {
    va_list ap;
    int j;
    double tot = 0;

    va_start(ap, count); //Requires the last fixed parameter (to get the address of it)

    for(j=0; j<count; j++) {
        tot+=va_arg(ap, double);  //Requires the type for type casting.
     }                            // Increments ap to the next argument.
    va_end(ap);

    return tot/count;
}
```

C Review

Pointers

A pointer to a variable is the ***address*** of a variable

```
int *px;
int x;

px = &x;  // the ampersand & notifies the compiler to use the
          // address of x rather than the value of x.

x  =  *px; // * is a unary operator that applies to a pointer and
           // directs the compiler to use the integer pointed to by
           // the pointer px.

           // * is referred to as the dereference operator

           // Remember: if px is a pointer to an int, *px is the int.
```

C Review

Pointers

A pointer to a variable is the ***address*** of a variable

```
int *px;    // Note:  this provides memory space for the pointer to the
            // type int, but does not provide any memory for the
            // int itself.
int x;      // allocates memory for the int itself

px = &x;    // the ampersand & notifies the compiler to use the
            // address of x rather than the value of x.

x  =  *px;  // * is a unary operator that applies to a pointer and
            // directs the compiler to use the integer pointed to by
            // the pointer px.

            // * is referred to as the dereference operator

            // Remember: if px is a pointer to an int, *px is an int.
```

C Review

Pointers

What does n = ?

```
int m,n;
int *pm;

m   = 10;
pm = &m;
n    = *pm;
```

n = 10

C Review

Unary pointer operators (*) have higher precedence than arithmetic operators;

What do the following statements do?

```
int *pi;

*pi = *pi + 10;
```

// The integer pointed to by pi will increase by 10.

```
y = *pi + 1;
```

// y will be replaced by one more than the integer
// pointed to by the pointer pi

```
*pi += 1;
```

// the integer pointed to by pi will increase by 1

C Review

Unary pointer operators (*,++) have the same precedence,
but are associated right to left.

What do the following statements do?

```
int *pi;

++*pi;
```

// The integer pointed to by pi is increased by 1

```
*pi++;
```

// The pointer pi is incremented *after* the
// dereference operator is applied.

```
(*pi)++;
```

// The integer pointed to by pi is post-incremented.

```
*++pi;
```

// The pointer is pre-incremented before the
// dereference.

C Review

Pointers and Arrays

```
int *pa;
int a[20];

pa = a;
pa = &a[0];       // The pointer points to the beginning address;

pa++              // The pointer increments to the next location in the array.
pa = pa + 1;      // If pa points to an integer, p++ points to the next integer
pa += 1;          // If pa points to a long,     p++ points to the next long
                  // If pa points to a double,   p++ points to the next double

                  // C automatically takes care of knowing the number of bytes
                  // that must be added to a pointer to point to the next element
                  // in an array.

pa = &a[0];       // *pa      = first element
                  // *(pa+1) = second element
                  // *(pa+2) = third element, etc.
```

C Review

Pointer Operations

There is a set of arithmetic operations that can be applied to pointers. They have to be of like types and pointing within the same array.

- Pointers can be compared.
- Pointers can be subtracted. (The result will be the number of elements between the two pointers, not the difference in values of the pointers.)
- Pointers can be incremented or decremented. (The result will be a pointer that points to the next (previous) element.)
- Pointers can be assigned. (It is a good practice to initially assign pointers to NULL before use. Why?)

C Review

Arrays are NOT Pointers

File 1:

```
int  mango[100];   // defines mango as an array of ints.
```

File 2:

```
extern int *mango;  // declares mango as a pointer to int.
.
.
.
// some code in file 2 that references mango[i]
```

Why will this crash your program?

Because it is as bad as confusing integer and floats

File 1: int x;

File 2: extern float x; // nobody expects this to work!

C Review

Some C terminology
(to help explain why the previous example is a problem)

Objects (variables) in C must have exactly one ***definition***, but may have multiple ***external declarations***.

Definition: Occurs in only one place // - Specifies the type of an object
// - **Reserves storage for it**
// - It is used to create new objects
// example: int my_array[100];

Declaration: Can occur multiple times // - describes the type of object
// - it is used to refer to objects defined
// elsewhere (in another file)
// example: extern int my_array[];

A declaration is like a customs declaration, it is not the thing itself, but a description of it.
A definition is a special kind of declaration, it fixes the storage for an object, i.e. it creates the defined object.

C Review

How Arrays and Pointers are Accessed.

- We are going to show the difference between a reference using an array and a reference using a pointer.
- We must distinguish between *address y* and the *contents of address y*.
- This is a subtle point because in most programming languages we use the same symbol to represent both, and the compiler figures out which is meant from the context.

Take the simple assignment:

```
x = y;
```

- The symbol x, in this context, means the *address* that x represents.
- This is termed an “L-value” (for “left-hand-side” or “Locator” value)
- An L-value is known at compile-time.
- An L-value says where to store the result.

- The symbol y, in this context, means the *contents of the address* that y represents.
- This is termed an “R-value” (for “right-hand-side”).
- An R-value is not known until run-time. “The value of y”, means the R-value unless otherwise stated.

A key point is that the address of each symbol is known at compile-time.

C Review

A Subscripted Array Reference

```
char a[9] = “abcdefgh”;
char c;

…
c=a[i];
…
```

Suppose at ***compile-time***, the compiler has given ***a*** the address 9980

a	b	c	d	e	f	g	h			x	
9980	+1	+2	+3	+4	+5	+6	+7	…		+i	

Run-time step #1 : get value i, multiply by 1 (byte), and add it to 9980

Run-time step #2 : get the contents from address (9980 + i*1)

```
extern char a[ ];      // Both are valid declarations, the compiler doesn't
extern char a[100];    // need to know how long a is, it merely generates
                       // address offsets from the start.
```

What does a[3] = ? a[3] = ‘d’

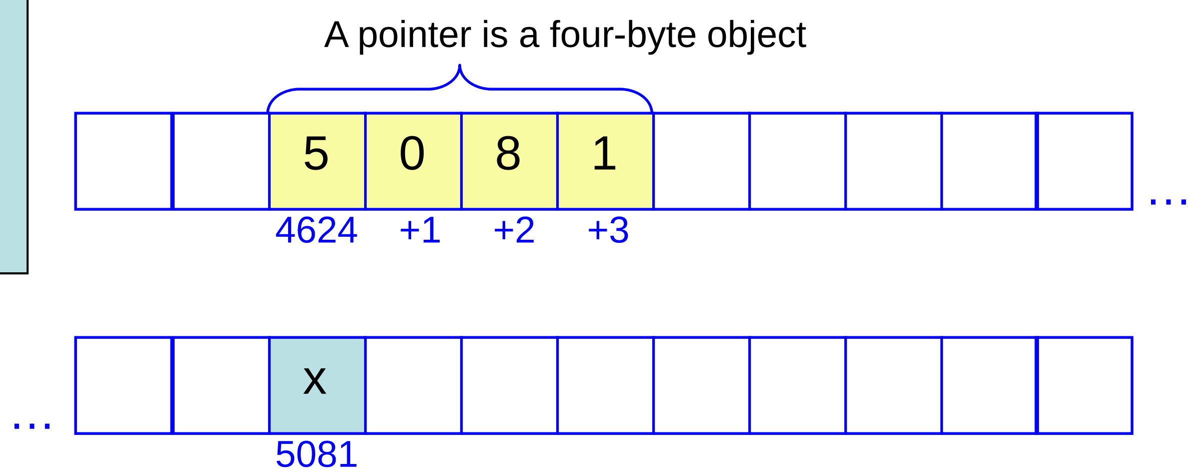
C Review

A Pointer Reference

```
char *p;
char c;

…
c=*p;
…
```

Suppose at ***compile-time***, the compiler has given ***p*** the address 4624



Run-time step #1 : get the contents from address 4624, say 5081
Run-time step #2 : get the contents from address 5081

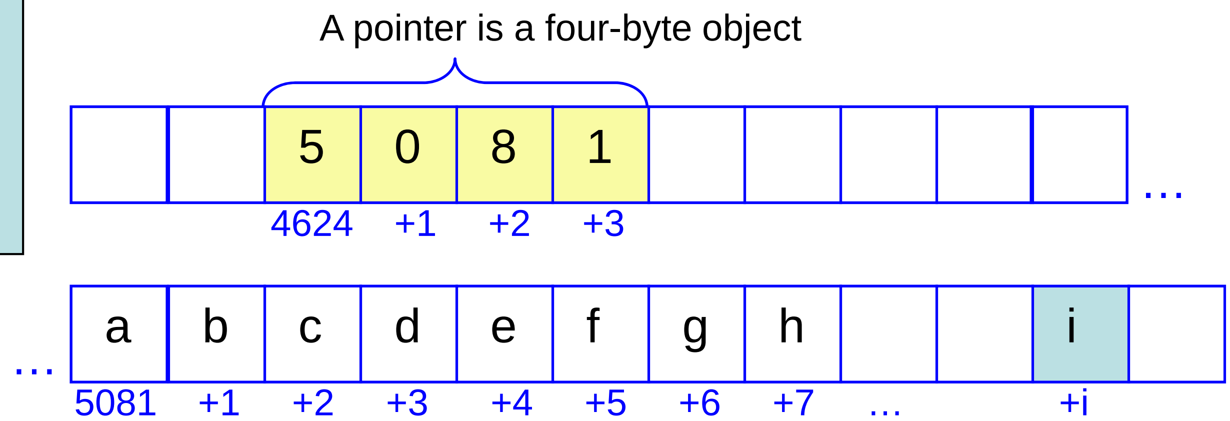
C Review

A Subscripted Pointer Reference

```
char *p = “abcdefgh”;
char c;

…
c=p[i];
…
```

Suppose at *compile-time*, the compiler has given *p* the address 4624



Run-time step #1 : get the contents from address 4624, say 5081
Run-time step #2 : get the value i, multiply by 1 (byte), and add it to 5081.
Run-time step #3 : get the contents from address (5081 + i*1)

What does p[3] = ?

p[3] = ‘d’
Note: a[3] and p[3] both give ‘d’, but by ***very different*** look-ups.

C Review

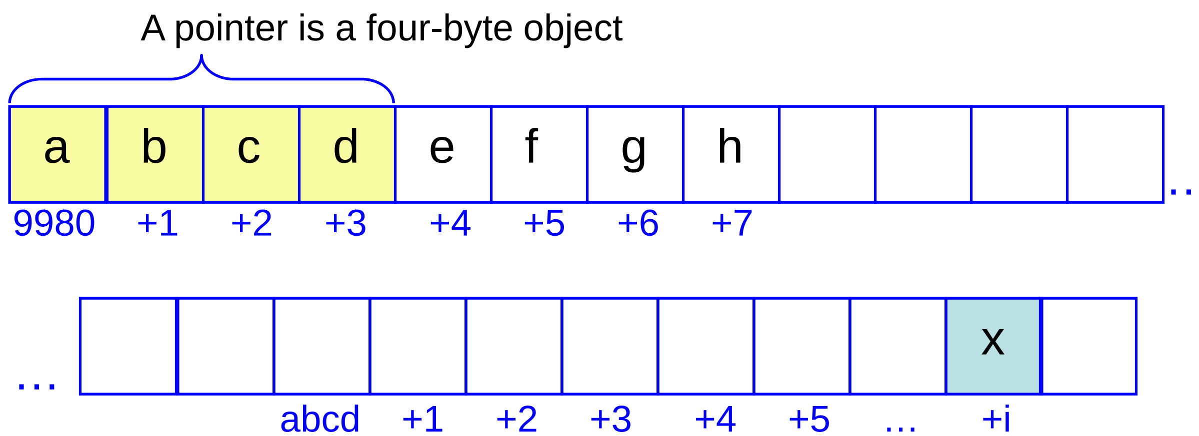
The Error that will occur if you *Define* as an Array and *Declare* as a Pointer

```
File 1:
char a[9] = "abcdefgh";
```

```
File 2:
extern char *a;
char c;

…
c=a[i];
…
```

Suppose at *compile-time*, the compiler has given *a* the address 9980



Run-time step #1 : get the contents from address 9980, which is abcd
Run-time step #2 : get the value i, multiply by 1 (byte), and add it to abcd.
Run-time step #3 : get the contents from address (abcd + i*1)

but this is wrong, since the ascii characters are giving us a bogus address. This will start corrupting memory in random places, which will cause mysterious errors later on. Hopefully, you will get lucky and get a core dump.

Solution:
Change the declaration so that it matches the definition.

C Review

Other differences between Arrays and Pointers

Pointer	Array
Holds the address of data	Holds data
Data is accessed indirectly, so you first retrieve the contents of the pointer, load that as an address (call it "L"), then retrieve the contents at "L" If the pointer has a subscript [i] you retrieve the contents of the location "i" *units* (data type) past "L".	Data is accessed directly, so for a[i] you simply retrieve the contents of the location i units past a.
Commonly used for dynamic data structures.	Commonly used for holding a fixed number of elements of the same type of data.
Commonly used with malloc() free()	Implicitly allocated and deallocated.
Typically points to anonymous data.	It is a named variable in its own right.

C Review

Function Arguments

Suppose we have the following code:

```
int x,y;

x = 4;
y = 5;
.
.
.
swap(x,y)
…
```

```
void swap(int x, int y){
    int temp;

    temp = x;
    x = y;
    y = temp;
}
```

What are the values of x and y after the swap(x,y) function is called?

x=4
y=5

They are the same. Why?

C Review

Function Arguments

fcomp()

```
int x,y;

x = 4;
y = 5;
.
.
.
swap(x,y)
…
```

```
void swap(int x, int y){
     int temp;

     temp = x;
     x = y;
     y = temp;
}
```

Top of stack
(or bottom if growing down)

Stack Pointer

Local variables of fcomp()
x = 4;
y = 5;

Return Address
Parameters for fcomp()

Location of
Variables x,y

C Review

Function Arguments

fcomp()

```
int x,y;

x = 4;
y = 5;
.
.
.
swap(x,y)
…
```

Location of program counter

```
void swap(int x, int y){
    int temp;

    temp = x;
    x = y;
    y = temp;
}
```

Top of stack
(actually bottom if growing down)

Stack Pointer

Local variables of fcomp()
x = 4;
y = 5;

Return Address
Parameters for fcomp()

C Review

Function Arguments

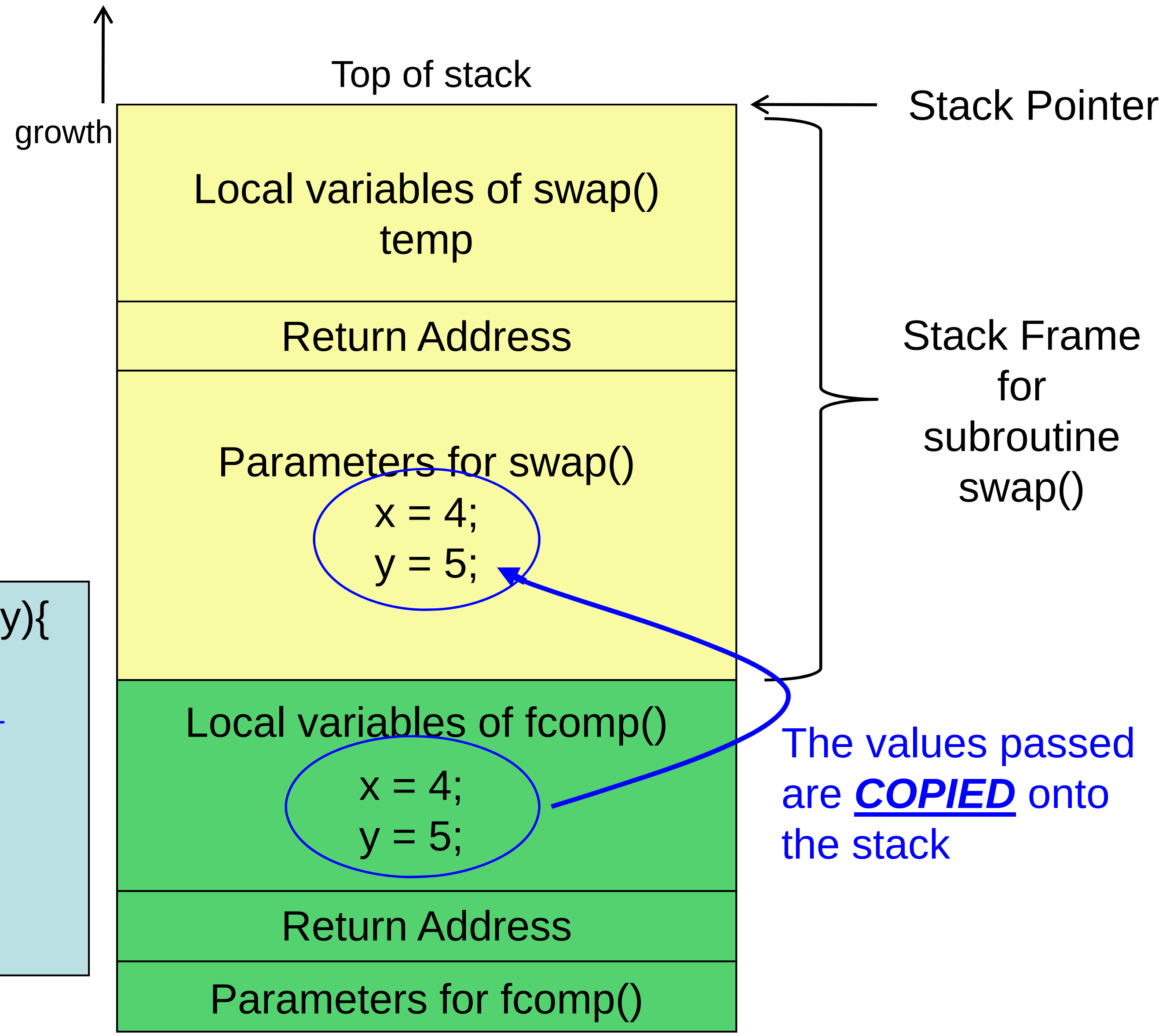
fcomp()

```
int x,y;

x = 4;
y = 5;
.
.
.
swap(x,y)
…
```

```
void swap(int x, int y){
    int temp;

    temp = x;
    x = y;
    y = temp;
}
```



C Review

Function Arguments

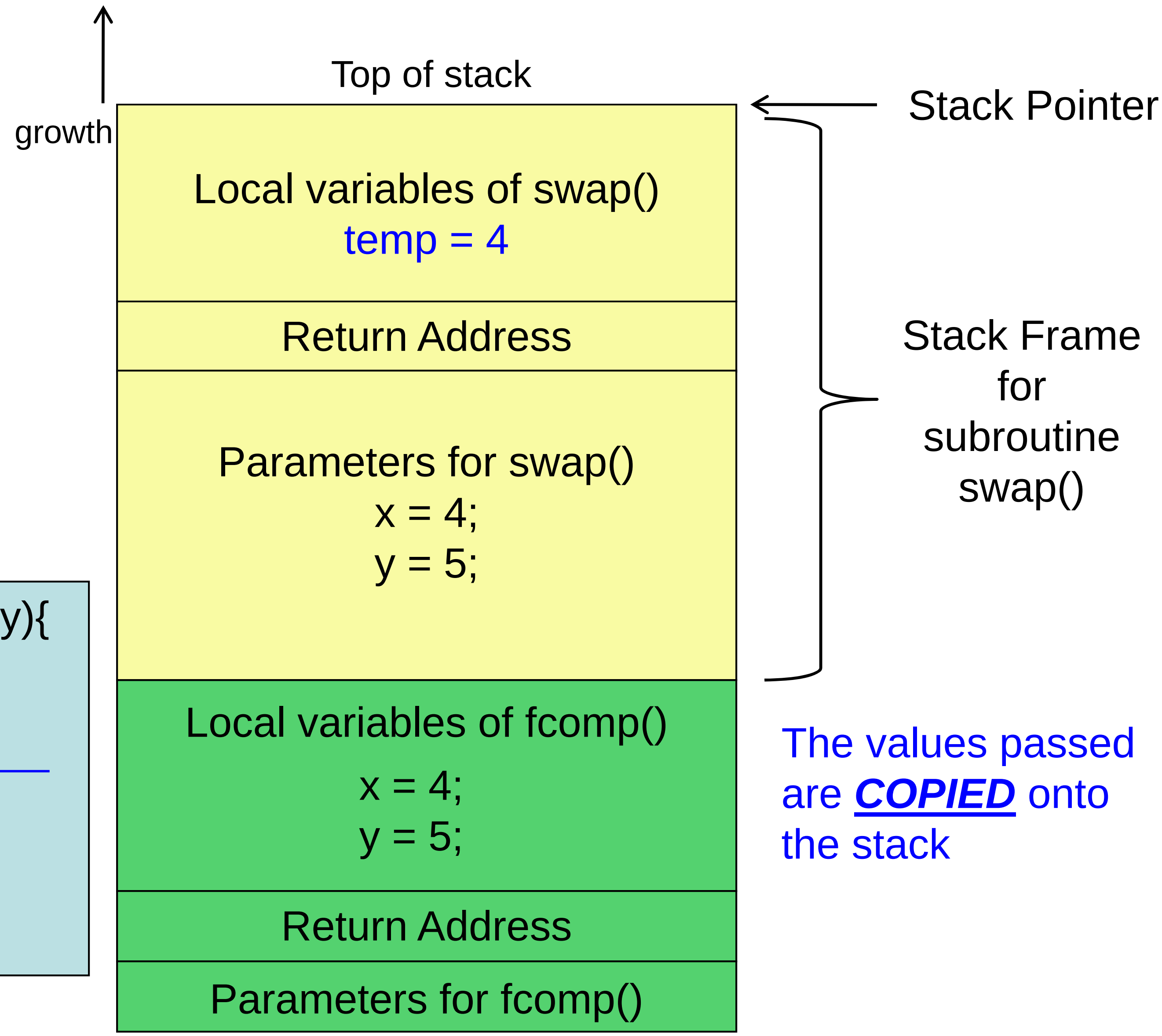
fcomp()

```
int x,y;

x = 4;
y = 5;
.
.
.
swap(x,y)
…
```

```
void swap(int x, int y){
    int temp;

    temp = x;
    x = y;
    y = temp;
}
```



C Review

Function Arguments

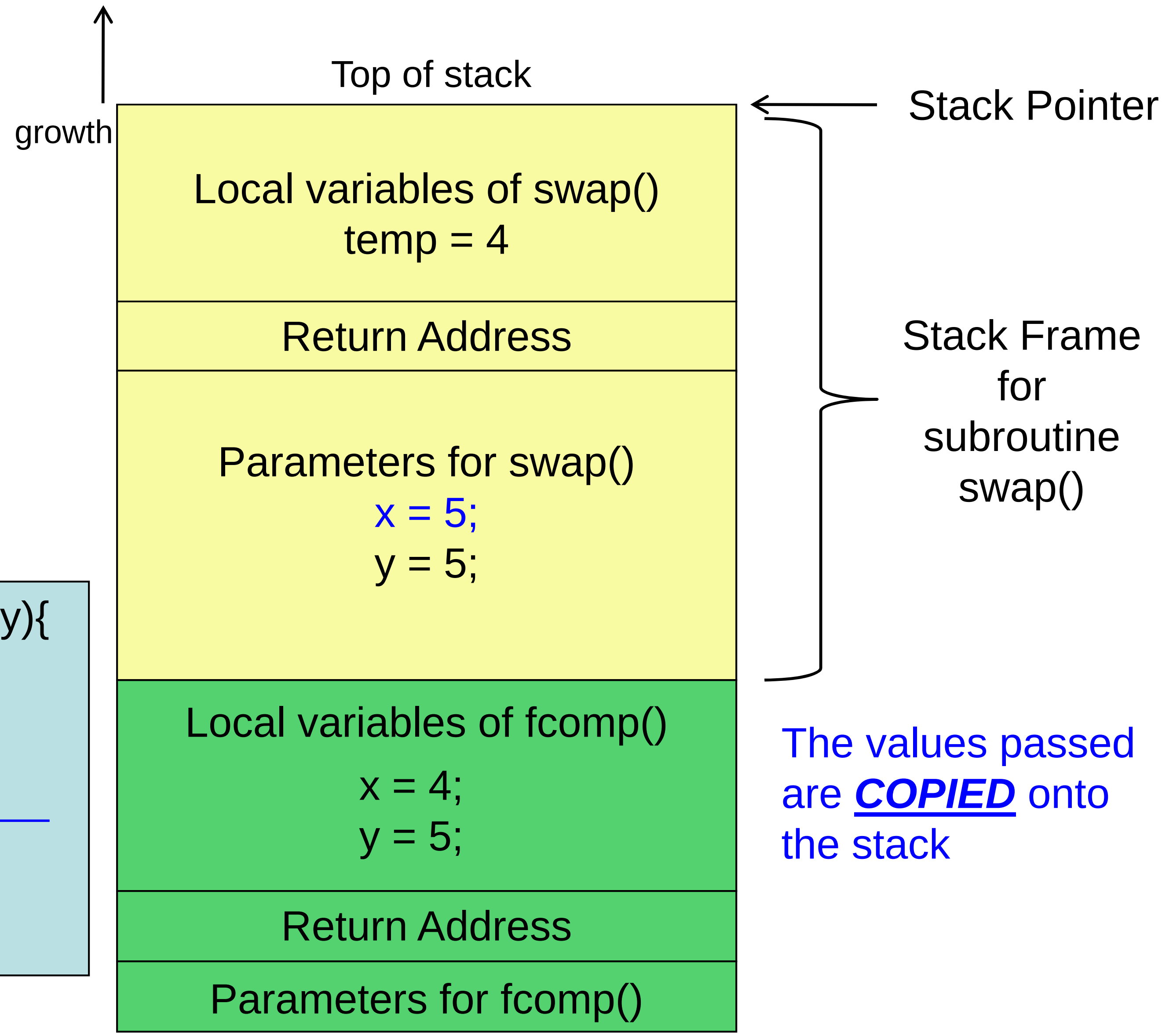
fcomp()

```
int x,y;

x = 4;
y = 5;
.
.
.
swap(x,y)
…
```

```
void swap(int x, int y){
    int temp;

    temp = x;
    x = y;
    y = temp;
}
```



The values passed are ***COPIED*** onto the stack

C Review

Function Arguments

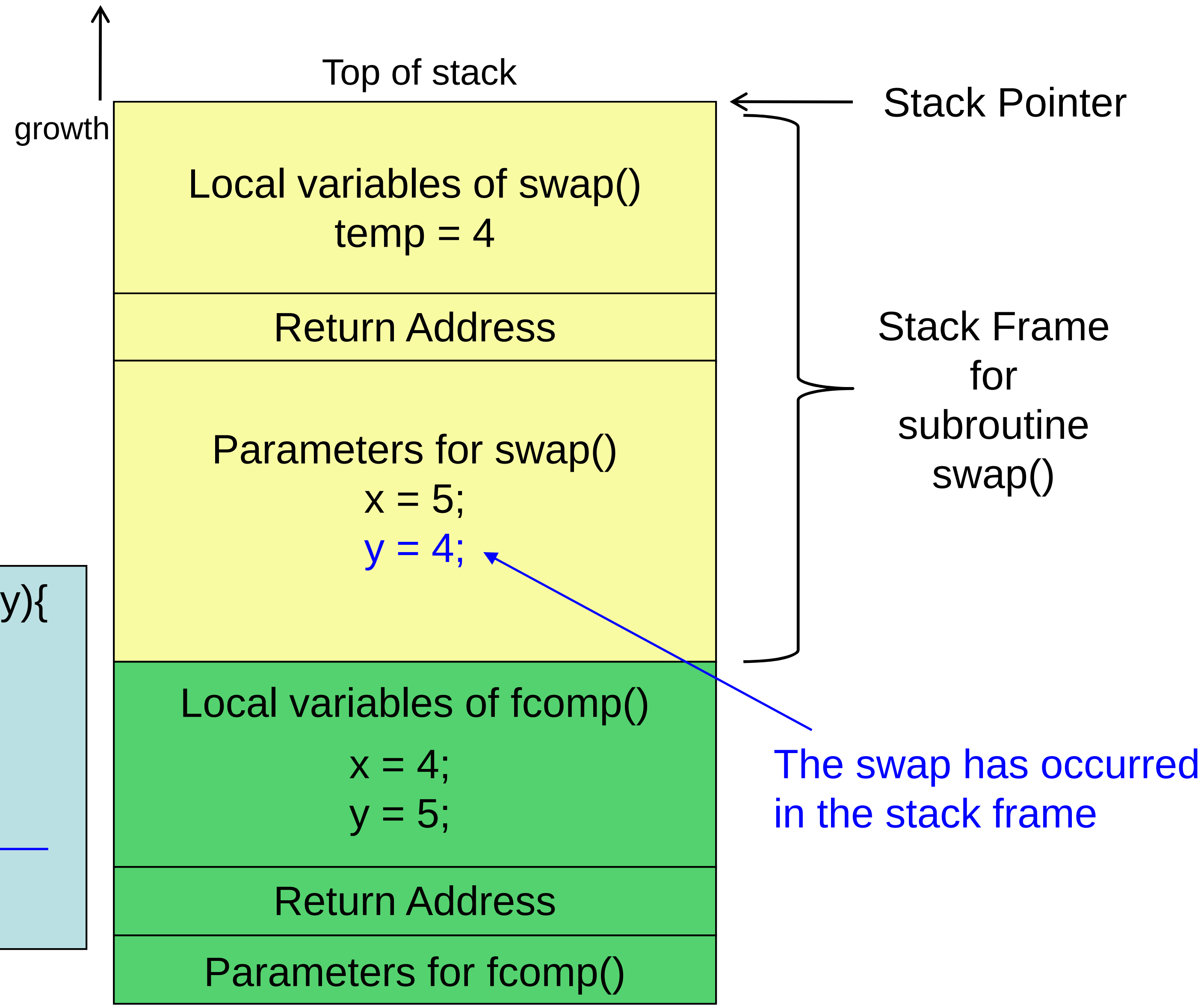
fcomp()

```
int x,y;

x = 4;
y = 5;
.
.
.
swap(x,y)
…
```

```
void swap(int x, int y){
    int temp;

    temp = x;
    x = y;
    y = temp;
}
```



C Review

Function Arguments

fcomp()

```
int x,y;

x = 4;
y = 5;
.
.
.
swap(x,y)
…
```

Values passed to functions as arguments are *__copies__* of the real values, i.e. they are ***__COPIED__*** onto the stack.

Data values passed to a function are pushed on the stack prior to the function call. If a function modifies any of the arguments, this modification only happens on the stack and is lost when the function returns.

```
void swap(int x, int y){
    int temp;

    temp = x;
    x = y;
    y = temp;
}
```

??

The swap occurred in the stack frame, which is now gone!

Top of stack

Local variables of fcomp()
x = 4;
y = 5;

Return Address
Parameters for fcomp()

We wanted these values swapped!?

C Review

Pointers and Function Arguments

The correct solution.

```
int x,y;

x = 4;
y = 5;
.
.
.
swap(&x,&y)
…
```

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

    temp = *px;
    *px = *py;
    *py = temp;
}
```

C Review

Pointers and Function Arguments

fcomp()

```
int x,y;

x = 4;
y = 5;
.
.
.
swap(&x,&y)
…
```

Location of program counter

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

     temp = *px;
     *px = *py;
     *py = temp;
}
```

Top of stack
(actually bottom if growing down)

Stack Pointer

Local variables of fcomp()
x = 4;
y = 5;

Return Address
Parameters for fcomp()

C Review

Pointers and Function Arguments

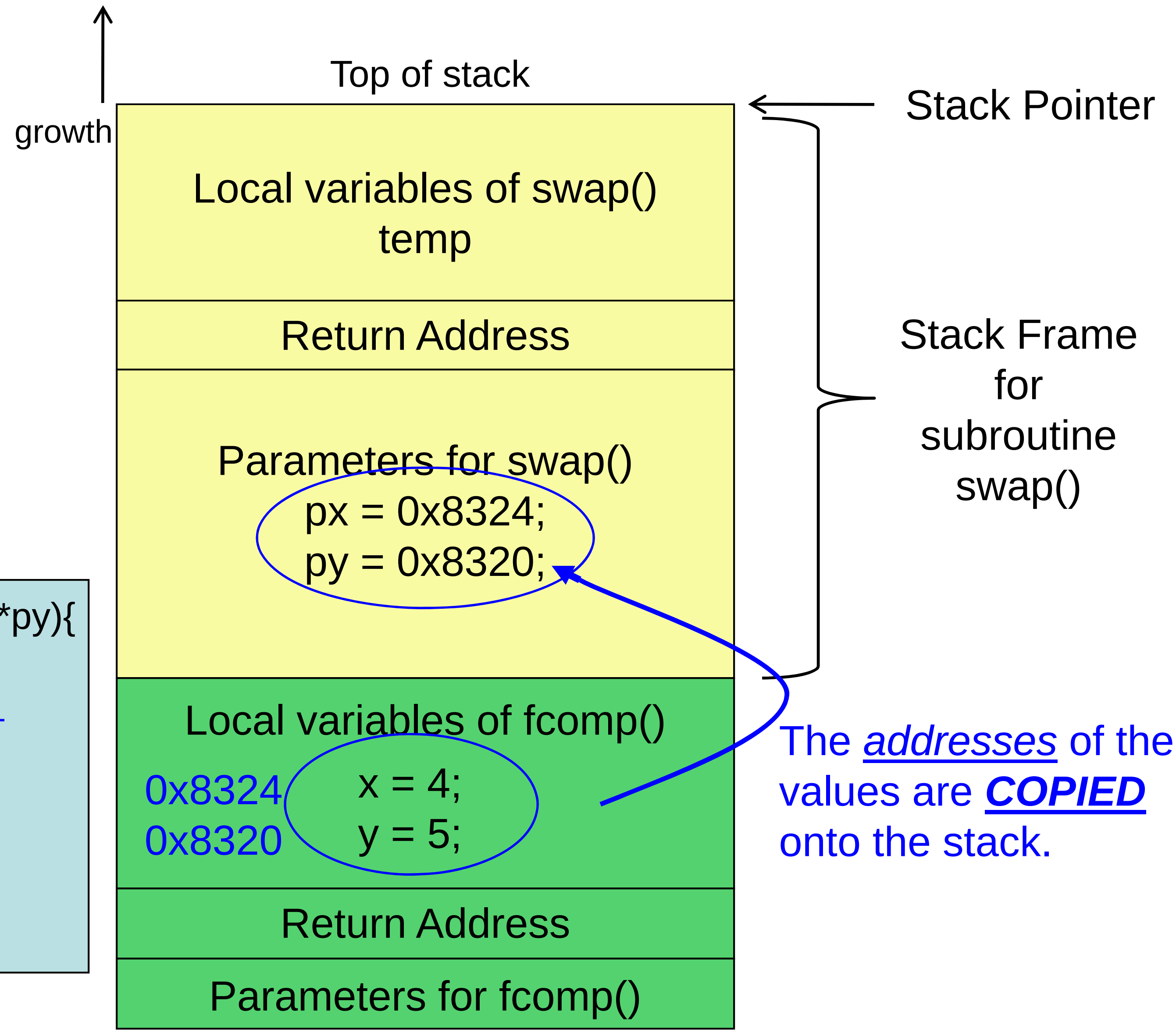
fcomp()

```
int x,y;

x = 4;
y = 5;
.
.
.
swap(&x,&y)
…
```

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

    temp = *px;
    *px = *py;
    *py = temp;
}
```



C Review

Pointers and Function Arguments

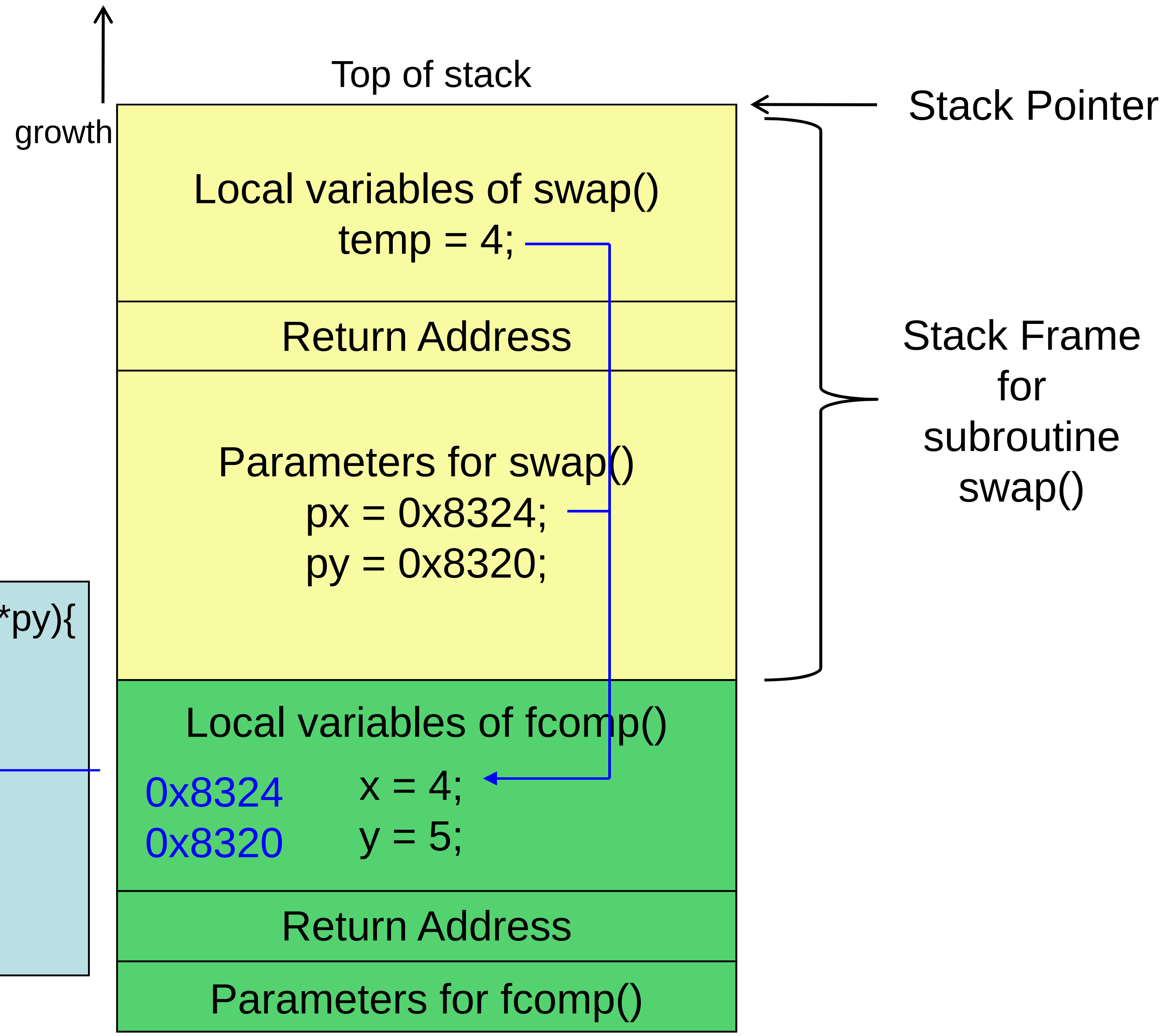
fcomp()

```
int x,y;

x = 4;
y = 5;
.
.
.
swap(&x,&y)
…
```

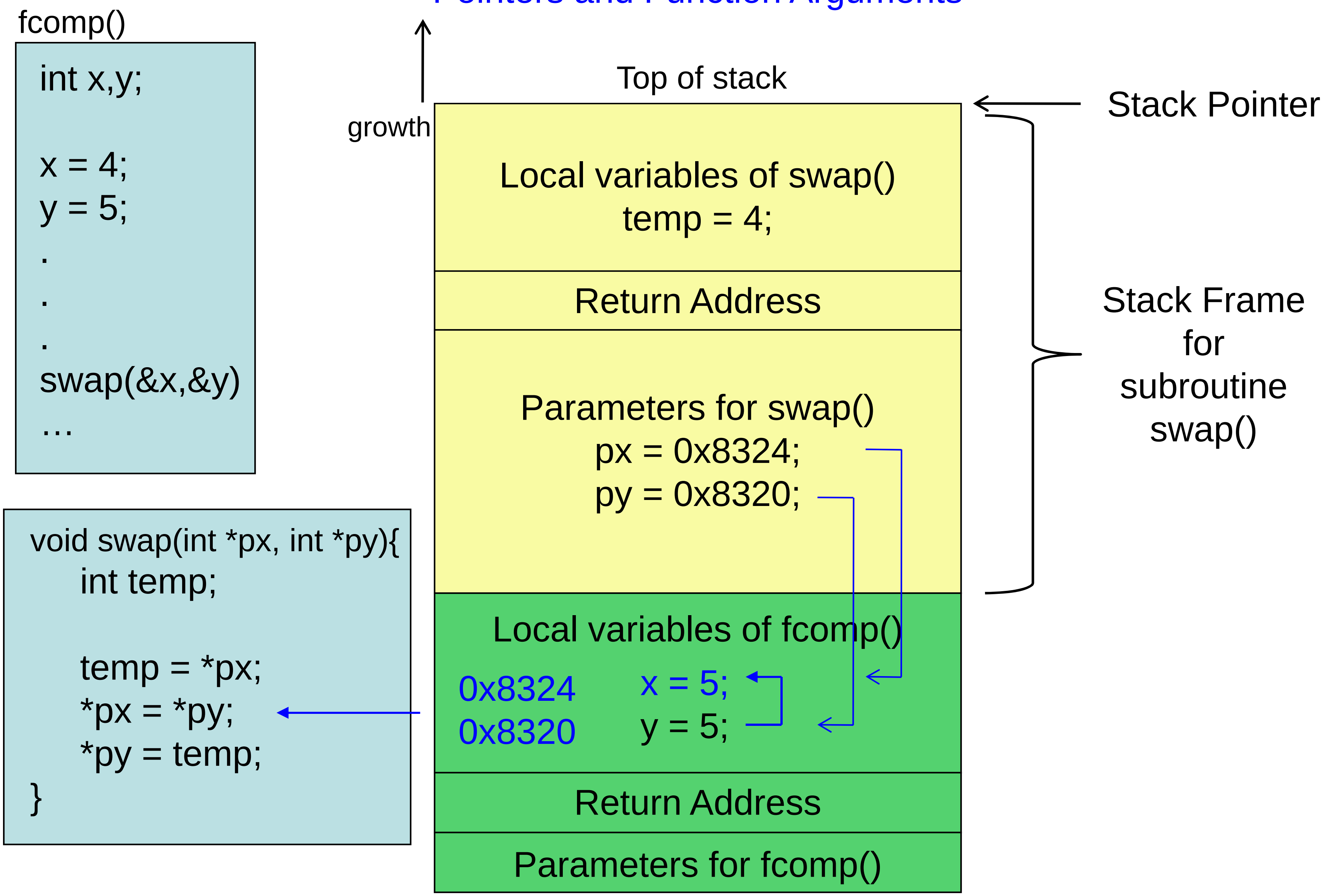
```
void swap(int *px, int *py){
    int temp;

    temp = *px;
    *px = *py;
    *py = temp;
}
```



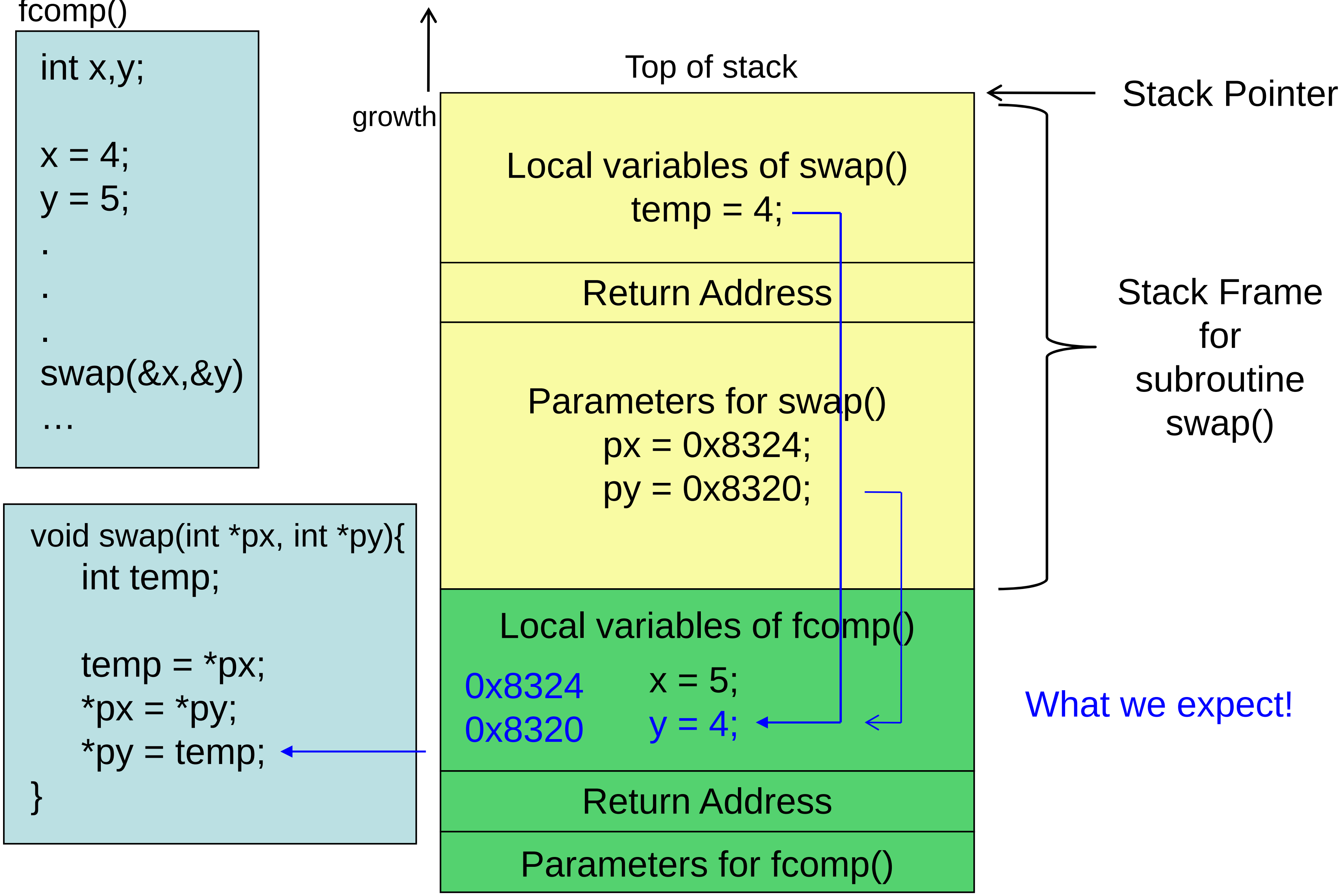
C Review

Pointers and Function Arguments



C Review

Pointers and Function Arguments



C Review

Pointers and Function Arguments

Suppose you want a ***function*** to create an array
and also return the size of the array.

Remember
For a function to modify ANY variable,
you copy the variables ADDRESS.

But here we want to modify a ***pointer*** in a function.

For a function to modify a pointer, you copy the ***pointer's address***
Thus you need a pointer pointer in a function

C Review

Pointer Pointers and Function Arguments

Example:
Have a function create an array and return the size.

```
#include <stdio.h>
#include <stdlib.h>

main(){
    float *duck;
    int i,N;
    void create_array(float **, int *);  // function prototype

    create_array(&duck,&N); // pass pointer address

    for(i=0; i<N; i++){
        printf("%d %f\n",i,duck[i]);
    }
    free(duck); // free allocated memory
}
```

```
void create_array(float **duck, int *N){
    int i;
                    // *duck is a pointer so it gets the
    *N = 5;         // address returned by calloc()
    *duck = (float *)calloc(5,sizeof(float);
    for(i=0; i<*N, i++){
        (*duck)[i] = (float)i*5;
    }               // The values are assigned to
}                   // **duck or (*duck)[]
```

C Review

Accessing a memory mapped register

Suppose you have a memory mapped register in memory located at 0x1000. How do you access it?

Type cast to the appropriate bit size

```
#define REGISTER1  (*(char *) 0x1000) // 8-bit register

REGISTER1 = 0xff;  // writing to the register

y = REGISTER1;   // reading from the register
```

C Review

Setting up an Interrupt Service Routine (ISR)

```
#define vector(isr,address) (* (void **)(address) = (isr))

vector(timer,0xfff0);   // places the address of the function timer() at
                        //  the location 0xfff0
```