

# Functions in C

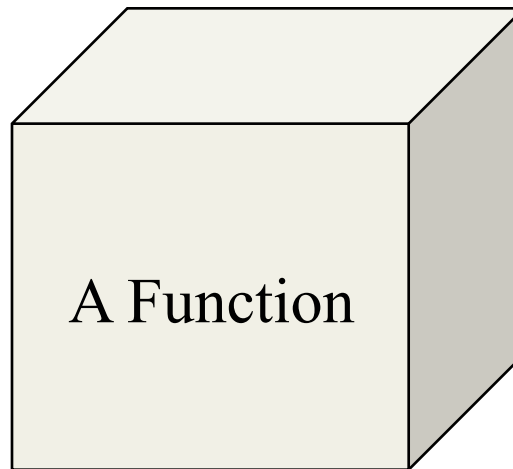
# Topics

- Functions
- By Value Parameter Passing
- Testing
- Scope

# Functions Match The Real World

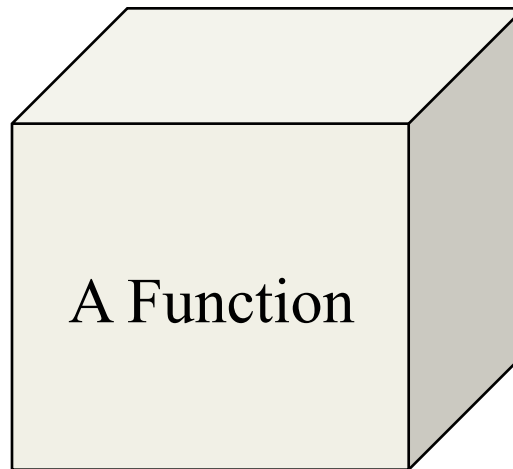
- Large organizations are managed by dividing them into smaller departments
- Humans seem to manage complexity by this process of subdivision
- Functions match this experience
  - Large problems get broken down into smaller sub pieces

# Functions As “Black Boxes”



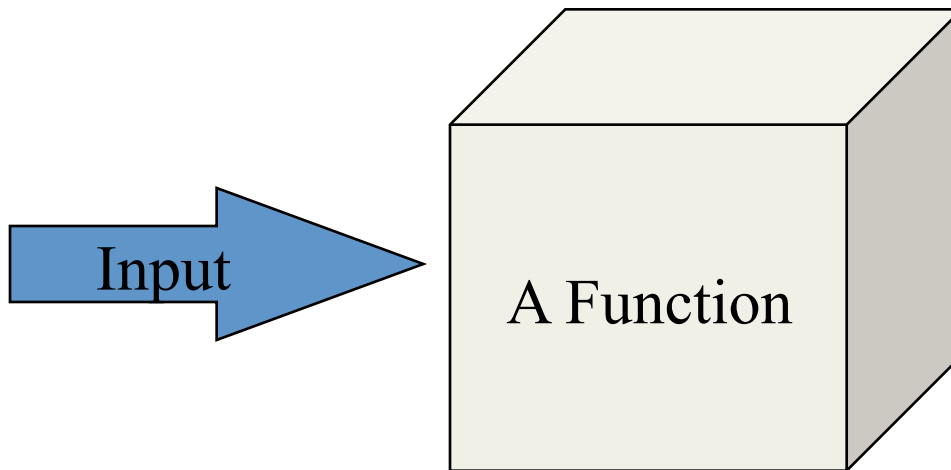
No one but the function's author  
needs to know what goes on inside

# Functions As “Black Boxes”



As users, all we know is that the Function accepts some kind of input and generates some kind of output

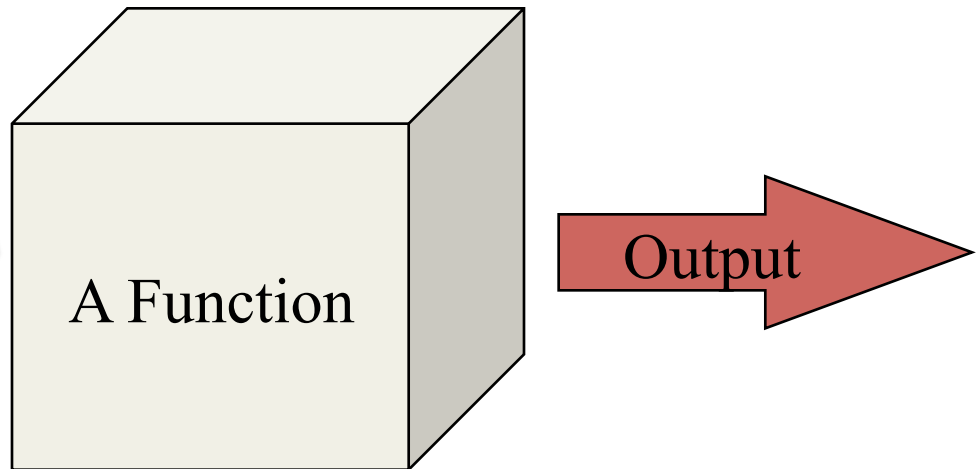
# Functions As “Black Boxes”



As users, all we know what input we should give to the function. For example, `fprint` we know if we pass it some variables, it “does” something with them

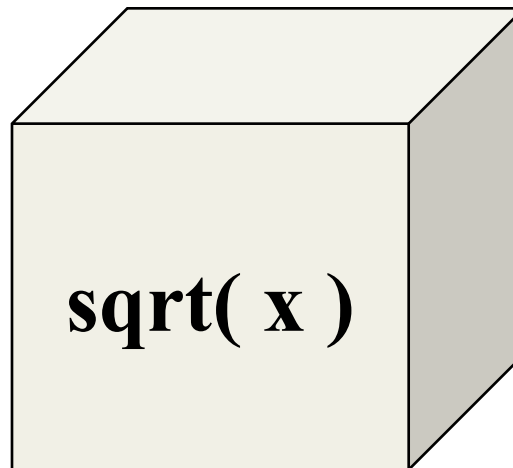
# Functions As “Black Boxes”

Kernighan and Ritchie:  
With properly designed functions, it is possible to ignore **HOW** the job is done; knowing **WHAT** is done is sufficient.



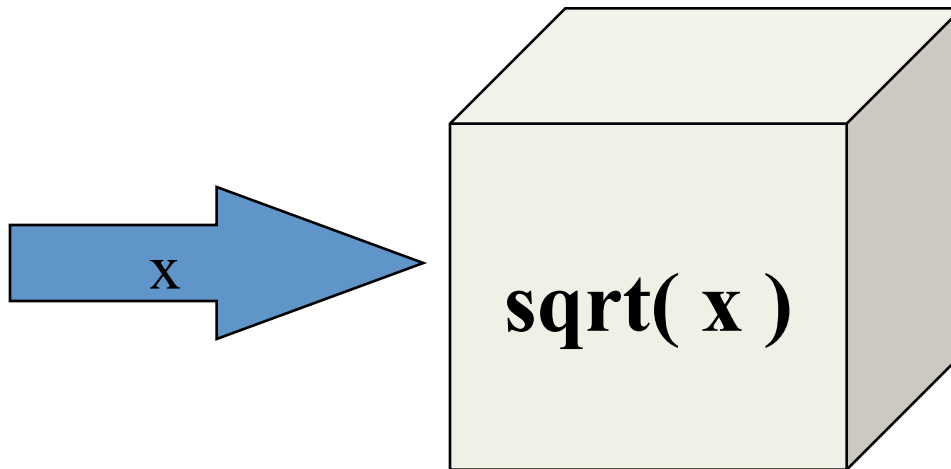
As users, all we care about is what the output is

# Functions As “Black Boxes”

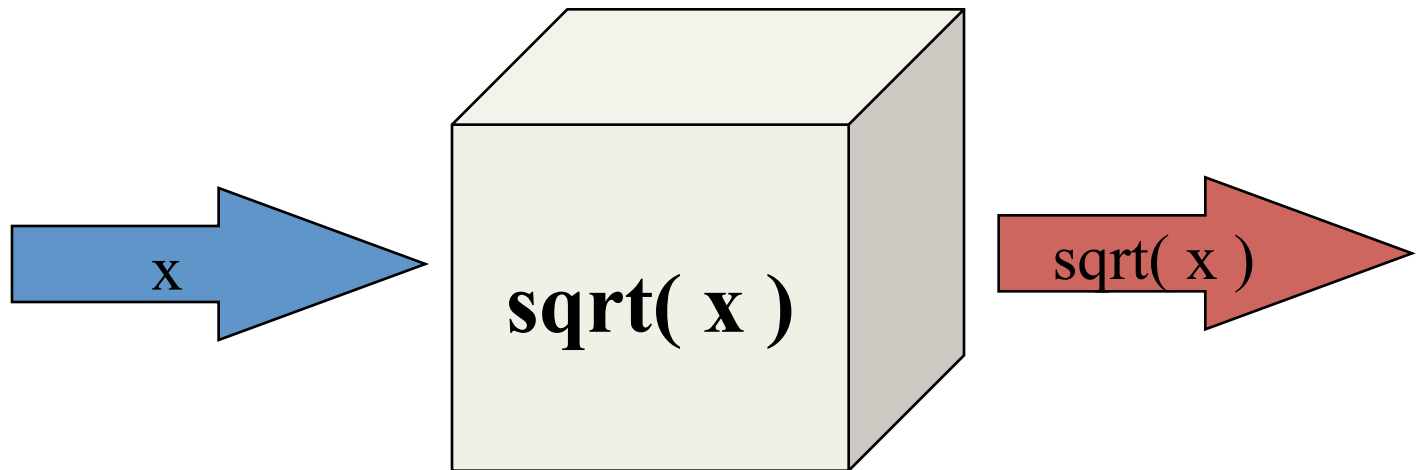




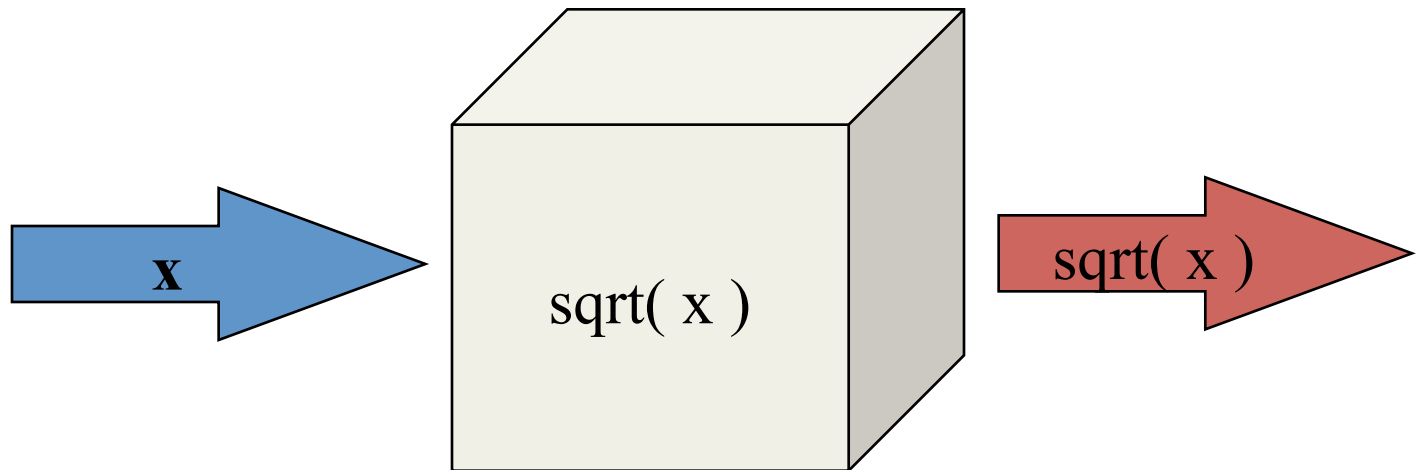
# Functions As “Black Boxes”



# Functions As “Black Boxes”



# Functions As “Black Boxes”



As users, we know **what** it  
Does but not **how** it does it

“Information hiding”

Important to also know, as a  
block box, the function does  
not use statements to get  
input from the end user at the  
keyboard

# Functions

- A named subprogram that can take parameters (input to function) and returns a result (output of a function)
  - `main ( )` is a function that returns `int`
- Functions are A way to reuse code
- Functions are an important part of programming
  - “divide and conquer”

# Predefined Functions

- C libraries offer many functions
  - `<Math>` **see book section 14** (no need to master all the listed functions)
  - `#Include <math>` acquires all the declarations in this system library

<u>Function</u>	<u>Argument</u>	<u>Result</u>
<code>ceil(x)</code>	double	double
<code>fabs(x)</code>	double	double
<code>floor(x)</code>	double	double
<code>pow(x, y)</code>	double	double
<code>sqrt(x)</code>	double	double

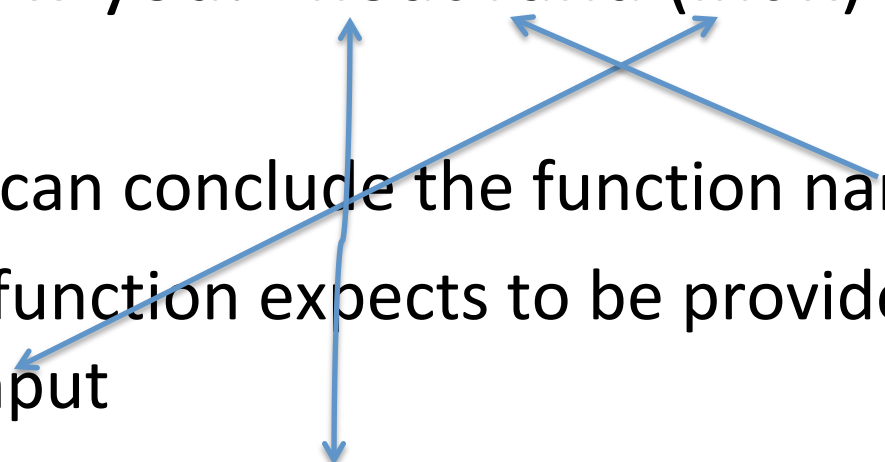
# Read Section 16

- The book does a good job on this topic.
- For now skip Variable Length Argument List.

# Syntax Of A Function Call

- To learn how to call a function you need to know its prototype (AKA signature).
- The prototype tells you the function **name**, how many **parameters** and their type, and what the **return** type is.
- Syntax:  
`rv = funcname ( [arg-list] );`  
where:
  - Arg-list := argument [, argument] \*
  - Rv is the value returned by the function call
  - This was a generic syntax so it sounds confusing; let's do an example.

# A Function Prototype

- If I show you: `float rand (int x)`
  - you can conclude the function name is rand
  - the function expects to be provided an int value as input
  - the function returns an float



# Function Signature

- A function signature or prototype or function header defines how a function is called
  - Tells everything you need to know to use it

**double** sqrt ( **double** number ) ;

↑            ↑            ↑            ↑

return    function    formal    formal  
type      name        parameter    parameter  
                         type            name

- Formal parameter gets replaced by the actual value at run-time

# Examples

- `double foo1(int x)` takes one int and returns a double
- `char foo2(int y)` takes one int and returns a char
- `float foo3(int x1, char x2)` takes one int as the first parameter and char as the second parameter and returns a float
- `void foo4(int y)` takes one int and does not return a value
- `double foo5(void)` takes no parameters and returns a double
- `void foo6(void)` takes no parameters and does not return a value
- Caution: Skipping the keyword `void` in C to mean that the function does not take parameters is ok by many C compilers. **However, some C compilers take `()` to mean: Any number of arguments (including none) of unknown types. Be as explicit as possible.**

# Inspecting a Function's Prototype

Let's take this example:

```
float foo3(int x1, char x2)
```

The above tells us that to call foo3 we must:

1. Pass foo3 an int value then add a comma then add a char value.
2. foo3 will return a float. If we want to “use” that value we must “save” it into a float declared var, or just show it in an output statement like printf using the proper type placeholder
3. We don't have to worry about the names x1 and x2

# Parameter or Argument

- Kernighan and Ritchie differentiate:
- The code that calls a function, uses arguments.
- The code that defines or prototypes the function, uses parameters.
- What? See examples next slide

# Parameter or Argument

- `float rand (int x);` // here x is a parameter
- `y=rand(z);` // here z is an argument
- You can use parameter to refer to everything, but you'll have to explain are we calling or are we in the function definition. This is needed once we talk about Passing By Reference.

# Calling...

- Based on: `int foo3(float x1, char x2)` //x1 and x2 are parameters
- Invalid or invalid calls :

```
int z;
```

```
z=foo3(4.0,qq); //valid if qq is a char var
```

```
z=foo3(4.5); //invalid: second param missing
```

```
foo3(0,'z'); // valid but return value is thrown out
```

```
foo3('w',3.5); //invalid, order of param matters
```

# Using C Functions math Demo!

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

```
#include <math.h>
```

```
int main() {
```

```
    const double PI = 3.14159; //good practice
```

```
    /*
```

```
        There are 3 steps involved when using a function.
```

```
        1. Declare your function.
```

```
            Officially, that is what the line:
```

```
                #include <math.h>
```

is for. #include performs "textual substitution". When the compiler chews up your program, it replaces that one line with the contents of that entire header file. In that header file is the declaration of `sin( )`, `cos( )`, `pow( )`, `sqrt( )` and many of other callable functions. You can see a lot of available functions you can call in the book.

The declaration needs to be found before the compiler finds a call to that function. Very often, programmers put these function declarations at the very top of the file to ensure they are found before any executable statements that could call that function.

```
        2. Call your function.
```

```
            Officially, I refer to the caller and the callee. These
```

```
are different parts of the code. The caller makes the
```

```
request. The callee responds to the request. I also like to call the caller "driver code" since it is often what gets the program moving along.
```

```
        3. Implement the body of your function
```

```
            You have to instruct the computer on what your function should
```

```
do when it is called. This is often the hardest part of all the three steps listed here.
```

In the case of these mathematical functions, #include <math.h> takes care of steps 1 and 3. So all we have to do it proceed with step 2. Eventually, we will get around to making and calling functions we built ourselves.

```
*/
```

```
/*
```

```
    Sin and Cosine functions take a parameter that is in radians.
```

```
    This web page shows the conversion formula between radians and degrees.
```

```
    http://math.rice.edu/~pcmi/sphere/drg\_txt.html
```

```
*/
```

```
printf( "Here are some math calculations:\n" );
```

```
printf( "The sine of 90 degrees = %f\n", sin( PI / 2.0 ) );
```

```
printf( "The sine of 180 degrees = %f\n", sin( PI ) );
```

```
printf( "The cosine of 90 degrees = %f\n", cos( PI / 2.0 ) );
```

```
printf( "The cosine of 180 degrees = %f\n", cos( PI ) );
```

```
return( 0 );
```

```
}
```

# Summarizing Our First Demo!

- Functions allow chunks of code to be reused
- Generally, functions enhance readability
- Arguments are passed by position – order matters



# Programmer-Defined Functions

- Programmers can define functions too
  - Declared by a function prototype
  - Defined by a function body
    - Prototype and body must match!
    - Function body contains variable declarations and executable statements, just like the body of the `main ( )` part of the program

# Function Call And Return

```
int foo( int i, double d); //prototype line  
int main( )
```

```
{ double x = 0;  
  
x = foo( 1, 3.1 );  
x = foo( 2, 2.2 );  
  
return 0;}
```

```
int foo( int i,  
         double d )
```

```
{ int val = 0;  
  return val;}
```

# Function Call And Return

start → `int foo( int i, double d );`  
`int main( )`

```
{double x = 0;  
  
x = foo( 1, 3.1 );  
  
x = foo( 2, 2.2 );  
  
return 0;}
```

```
int foo( int i,  
         double d )
```

```
{int val = 0;  
return val;}
```

# Function Call And Return

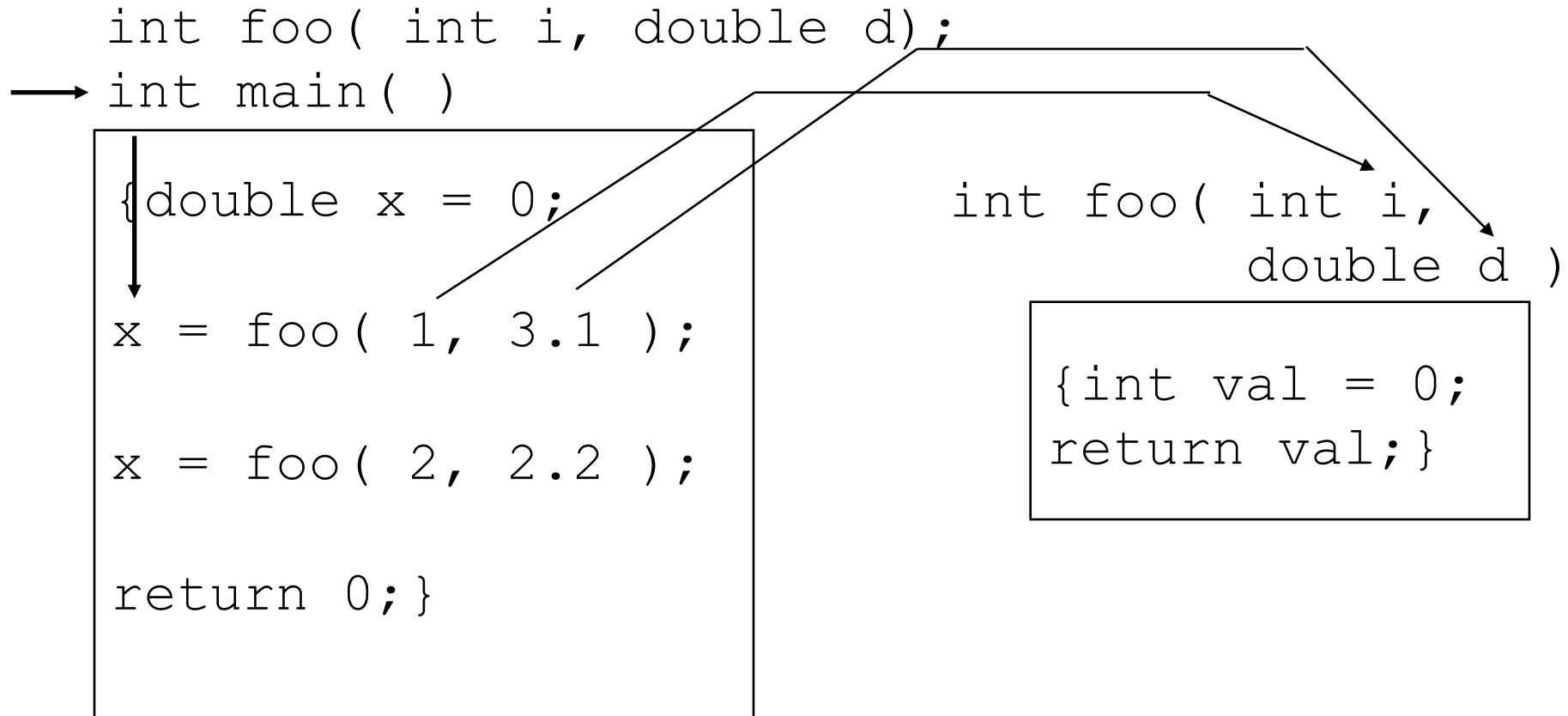
```
int foo( int i, double d);  
→ int main( ) //
```

```
{ double x = 0;  
x = foo( 1, 3.1 );  
x = foo( 2, 2.2 );  
return 0;}
```

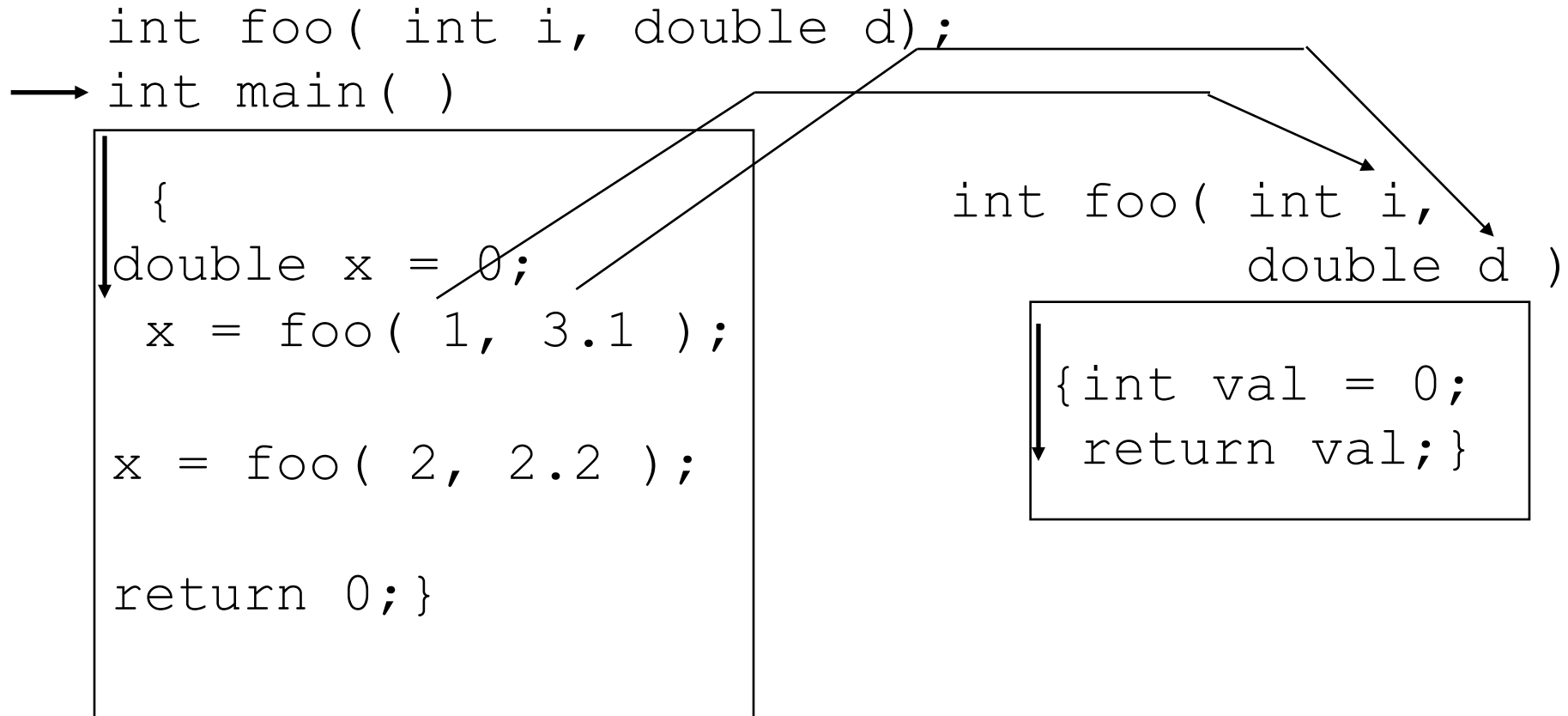
```
int foo( int i,  
         double d )
```

```
{int val = 0;  
return val;}
```

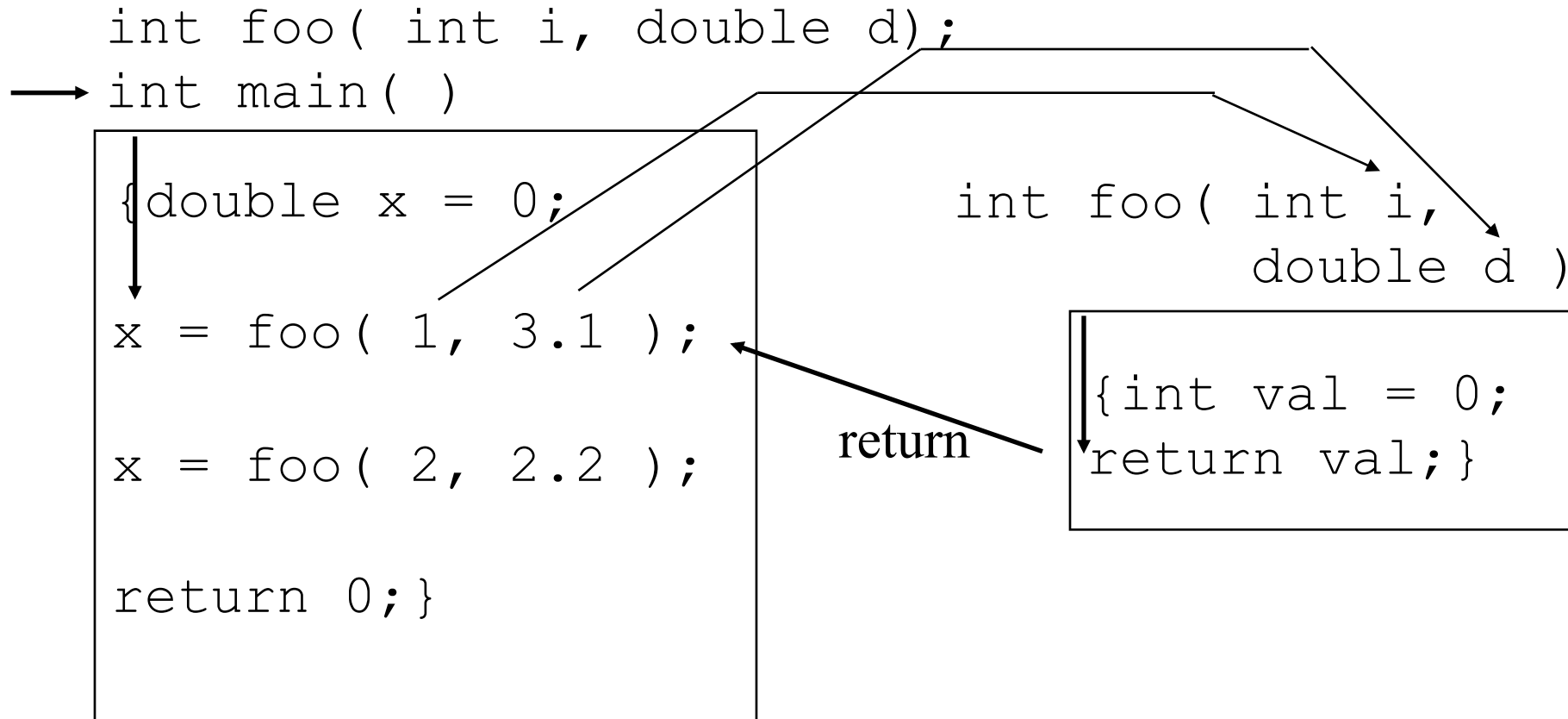
# Function Call And Return



# Function Call And Return



# Function Call And Return



# Function Call And Return

```
int foo( int i, double d);  
→ int main( )
```

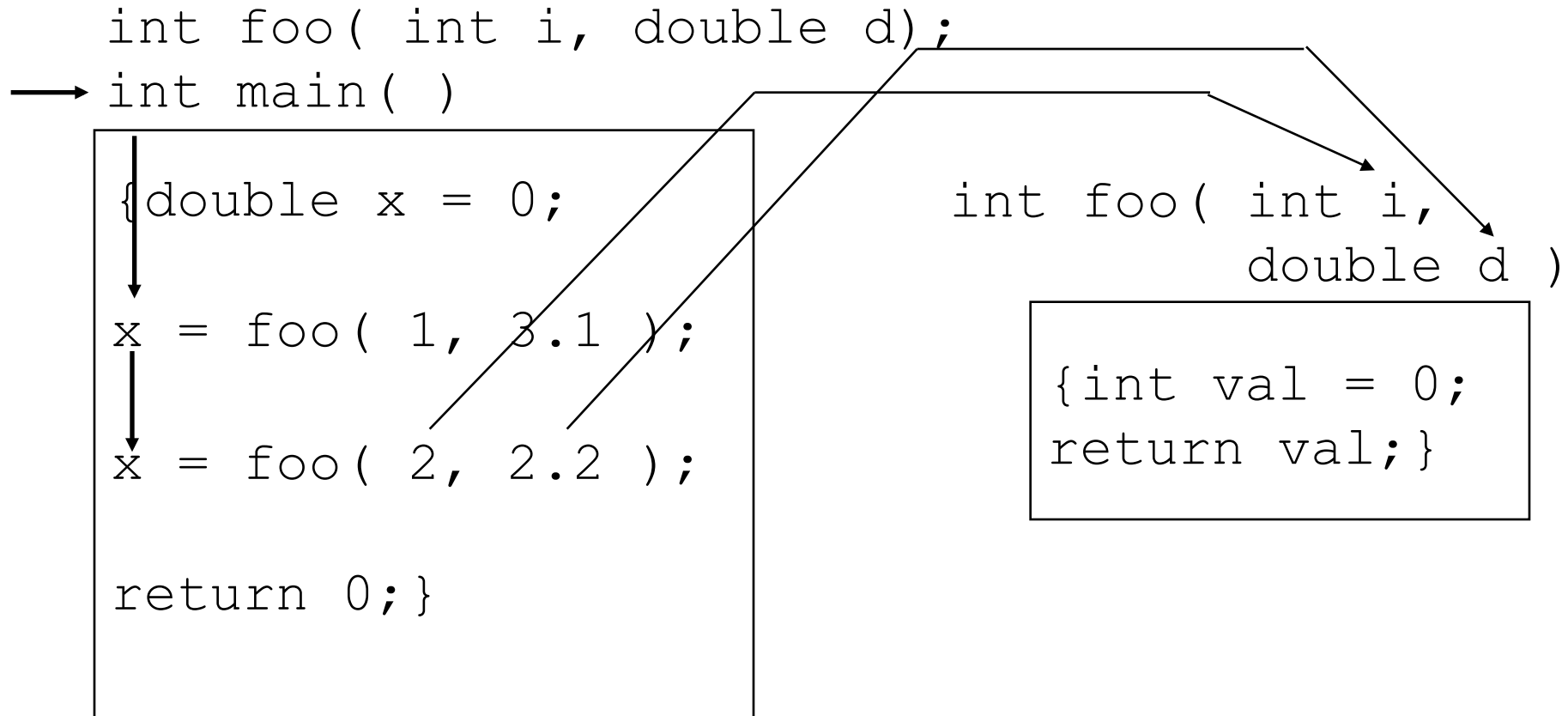
```
{double x = 0;  
↓  
x = foo( 1, 3.1 );  
next  
↓  
x = foo( 2, 2.2 );  
  
return 0;}
```

```
int foo( int i,  
        double d )
```

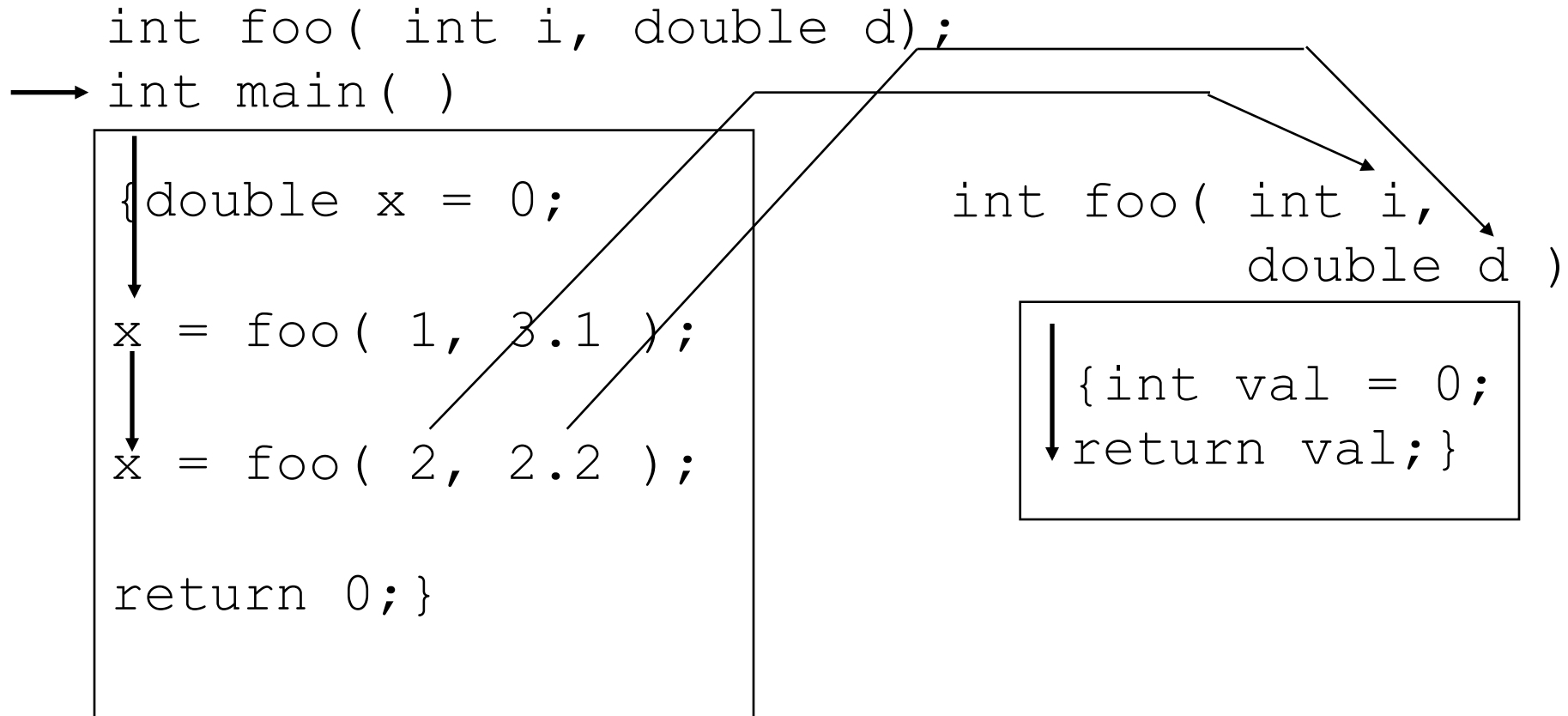
```
{int val = 0;  
return val;}
```



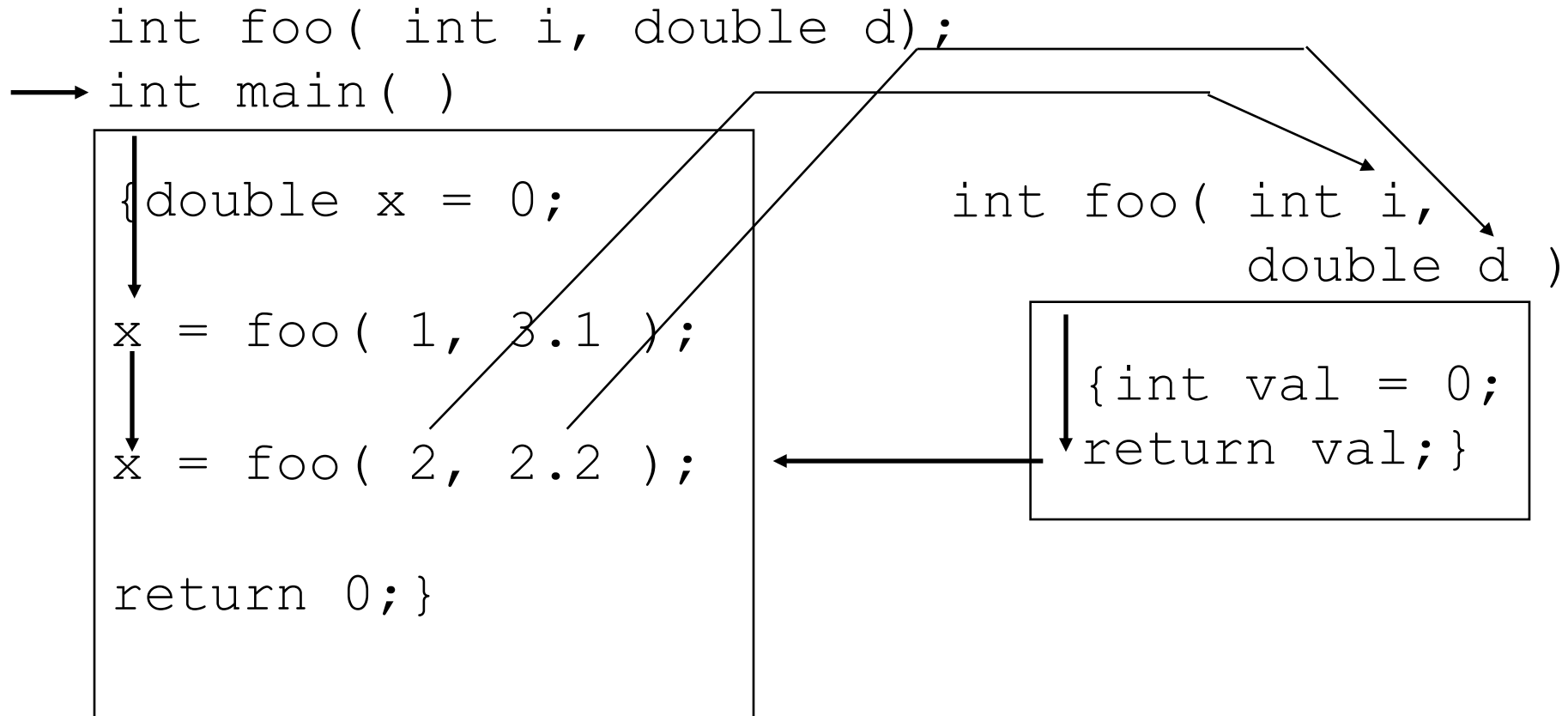
# Function Call And Return



# Function Call And Return



# Function Call And Return



# Function Call And Return

```
int foo( int i, double d);  
→ int main( )
```

```
{double x = 0;  
↓  
x = foo( 1, 3.1 );  
↓  
x = foo( 2, 2.2 );  
↓  
return 0;}
```

```
int foo( int i,  
        double d )
```

```
{int val = 0;  
return val;}
```

# Function Call And Return

```
int foo( int i, double d);  
→ int main( )
```

```
{double x = 0;  
↓  
x = foo( 1, 3.1 );  
↓  
x = foo( 2, 2.2 );  
↓  
return 0;}  
←
```

```
int foo( int i,  
         double d )
```

```
{int val = 0;  
return val;}
```

# In Our Next Demo

- Write a function that takes an integer, and returns the summation of each integer from 0 to the number, raised to the power 3. In mathematical terms:

Output Of Function =  $1^3 + 2^3 + 3^3 + \dots n^3$  where n is the input to the function

Before we do any typing, we must first think of the actual code that implements the above. Think, write on paper then proceed to the next slide.

# User\_defined Functions Demo!

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

```
/*
  There are 3 steps involved when using a function.
  1. Declare your function.
  */
int triplets( int trip); //note we used trip here but num there
//the parameter names here are not connected to the names there
//The order and type must match

int main() {
    int v,w;
    printf( "What value would you like to triplet:\n" );
    scanf_s("%d",&v);
    printf( "Next I'm calling the triplets function:\n" );
    /*
      2. Call your function
      Here is step 2
    */
    w =triplets(v); // v is an argument. Its content is dumped into
    return( 0 );
}
```

## 3. Implement the body of your function

User defined functions must be implemented with the exact definition listed at the top of the file. My advice is to copy and paste, so there is no doubt that these two match exactly .

```
*/
int triplets( int num)
{
    //do not use any print or scan statements in the function
    // remember: black box
    int counter;
    int total = 0;
    for (counter=0; counter<=num;counter++)
        total+=(counter*counter*counter);
    Return total
}
/*
  Is there a chance that v, which was used to pass the input be changed?
  NO! There is no connection between v and num. This is the
  way pass-by-value works...more on this later
*/
```

# Summarizing Our Second Demo!

- Functions need to be documented! Add comments as needed
- Parameters receive copies of the arguments
- Recall parameter is in the definition, while argument is in the calling code
- Return values, although provided, may be ignored by the caller
- Functions are defined once but may be used countless times



# Return...

- Functions use `return` statement to communicate to the caller
- It is possible to write a function that has no return statement. This is a bad practice. DON'T DO IT.
- When no return value is intended, use `return 0`
- It is possible to use `return` with no expression in which case control moves back to the calling code, then the next line
- The `return` statement can return a literal, a variable, or an expression
- It is my practice to enclose what is returned in parenthesis

# Really?

- A return statement, moves control back to the calling code
- Any code placed after a return statement is ignored and never executes

# Before we close on Functions

- If you wish to declare and define functions, the definition can be in any order
- A function such as:  
    `dummy() {}`  
does nothing, and its return type is assumed to be `int`.  
Why use it? As a placeholder while writing big programs
- If a return type of a function does not match the type of the expression returned, the expression is cast to match the defined return type when possible
- Please do not memorize this but try it out to learn it.

# Parameter Passing

- Arguments can be passed by Value or Reference.
- In the previous examples, we used Pass-by-value
  - Functions see A copy of the value passed, not the value or argument itself
- We will cover by-reference in a later unit

# Problem Solving Strategy

- One big problem is harder to solve than many smaller problems
- Understand the problem
  - what result is expected
  - what process can provide these results
  - what parameters are needed for these processes
  - write function descriptions in english telling what the function should do

# Problem Solving Strategy

- C Syntax Typically Obscures Understanding
  - write out your solution on paper FIRST
  - use flow charts or pseudocode
  - translate to C syntax on paper
  - try not to compose code at a terminal
- Great Answers Don't Come The First Time
  - iteratively refine and enhance partial solutions

# Variable Scope

- Variables declared in a function are only visible in that function
  - Referred to as a “local” variable
- More generally, every variable has a “scope” which defines its lifecycle

# Variable Scope

```
int foo( int i, double d);  
int main( ) {
```

```
    double x = 0;  
    int i = 45;  
  
    x=foo( 1, 3.1 );  
  
    x=foo( 2, 2.2 );  
  
    return 0;  
}
```

```
int foo( int i,  
        double d ) {
```

```
    int val = 1;  
    return val;  
}
```



# Variable Scope

```
int foo( int i, double d);  
int main( ) {
```

```
    double x = 0;  
    int i = 45;  
  
    x=foo( 1, 3.1 );  
  
    x=foo( 2, 2.2 );  
  
    return 0;  
}
```

```
int foo( int i,  
        double d ) {  
  
    int val = 1;  
    return val;  
}
```

What is the scope of variable i of main?  
The bolded square box above

# Variable Scope

```
int foo( int i, double d);  
int main( ) {
```

```
    double x = 0;  
    int i = 45;  
  
    x=foo( 1, 3.1 );  
  
    x=foo( 2, 2.2 );  
  
    return 0;  
}
```

```
int foo( int i,  
        double d ) {  
    int val = 1;  
    return val;  
}
```

What is the scope of variable i of foo?  
The bolded square box above

# Variable Scope

```
int foo( int i, double d);  
int main( ) {
```

```
    double x = 0;  
    int i = 45;  
  
    x=foo( 1, 3.1 );  
  
    x=foo( 2, 2.2 );  
  
    return 0;  
}
```

```
int foo( int i,  
        double d ) {
```

```
    int val = 1;  
    return val;  
}
```

First: *i* in main has NO connection to *i* in foo

# Variable Scope

**In the next few slides the arrow implies the flow of control, once we hit the runtime. The bolded box implies it has control over i**

```
int foo( int i, double d );  
int main( ) {
```

start ↓

```
    double x = 0;  
    int i = 45;  
  
    x=foo( 1, 3.1 );  
  
    x=foo( 2, 2.2 );  
  
    return 0;  
}
```

```
int foo( int i,  
        double d ) {
```

```
    int val = 1;  
    return val;  
}
```

# Variable Scope

```
int foo( int i, double d);  
int main( ) {
```

```
    double x = 0;  
    int i = 45;
```

Call foo

```
    x=foo( 1, 3.1 );
```

```
    x=foo( 2, 2.2 );
```

```
    return 0;
```

```
}
```

```
int foo( int i,  
        double d ) {
```

```
    int val = 1;  
    return val;
```

```
}
```

# Variable Scope

```
int foo( int i, double d);  
int main( ) {
```

```
    double x = 0;
```

```
    int i = 45;
```

```
    x=foo( 1, 3.1 );
```

```
    x=foo( 2, 2.2 );
```

```
    return 0;
```

```
}
```

**1 is dumped into i**

```
int foo( int i,  
        double d ) {
```

```
    int val = 1;
```

```
    return val;
```

```
}
```

# Variable Scope

```
int foo( int i, double d);  
int main( ) {
```

```
    double x = 0;
```

```
    int i = 45;
```

```
    x=foo( 1, 3.1 );
```

```
    x=foo( 2, 2.2 );
```

```
    return 0;
```

```
}
```

1 is dumped into i

**3.1 dumped into d**

```
int foo( int i,  
        double d ) {
```

```
    int val = 1;
```

```
    return val;
```

```
}
```

# Variable Scope

```
int foo( int i, double d);  
int main( ) {
```

```
    double x = 0;  
    int i = 45;
```

```
    x=foo( 1, 3.1 );
```

```
    x=foo( 2, 2.2 );
```

```
    return 0;
```

```
}
```

1 is dumped into i  
3.1 dumped into d

```
int foo( int i,  
        double d ) {
```

```
    int val = 1;  
    return val;
```

```
}
```



# Variable Scope

```
int foo( int i, double d);  
int main( ) {
```

```
    double x = 0;  
    int i = 45;
```

```
    x=foo( 1, 3.1 );
```

```
    x=foo( 2, 2.2 );
```

```
    return 0;
```

```
}
```

```
int foo( int i,  
        double d ) {
```

```
1  int val = 1;
```

```
    return val;
```

```
}
```

# Variable Scope

```
int foo( int i, double d );  
int main( ) {
```

```
    double x = 0;  
    int i = 45;
```

```
    x=foo( 1, 3.1 );
```

```
    x=foo( 2, 2.2 );
```

```
    return 0;
```

```
}
```

```
int foo( int i,  
         double d ) {
```

```
    int val = 1;  
    return val;  
}
```



**x is 1**

1

# Variable Scope

```
int foo( int i, double d);  
int main( ) {
```

```
    double x = 0;  
    int i = 45;
```

```
    x=foo( 1, 3.1 );
```

```
    x=foo( 2, 2.2 );
```

```
    return 0;
```

```
}
```

```
int foo( int i,  
        double d ) {
```

```
    int val = 0;  
    return val;
```

```
}
```

And the control continues as in the previous line, ...

# Variable Scope

```
int foo( int i, double d);  
int main( ) {
```

```
    double x = 0;  
    int i = 45;  
  
    x=foo( 1, 3.1 );  
  
    x=foo( 2, 2.2 );  
  
    return 0;  
}
```

```
int foo( int j,  
        double d ) {  
  
    int val = 1;  
    return val;  
}
```

**If we change i to j in the foo function, nothing will change.**

Is local scope good?

The BEST!!

# Variable Scope

- There may be cases when a bigger scope is needed.
- You can define variables and constants that have a global scope
  - Visible to all functions, including main

```
#include <stdio.h>
const int PI=3.14159;
```

```
int main() {
    ...
}
```

- We'll only do this for constants
- It's not recommended to use global variables

# From The Book:

- **Scope** describes the level at which a piece of data or a function is visible. There are two kinds of scope in C, local and **global**. When we speak of something being **global**, we speak of something that can be seen or **manipulated from anywhere in the program**. When we speak of something being local, we speak of something that can be seen or manipulated only within the block it was declared.
- Read the section on Scope from Variables but not yet. I'll remind you later

# Global vs Local: the battle continues...

```
int i = 5; /* i is a global variable that can be accessed anywhere */  
//that's how you make global scope  
// declare the variable outside and before main
```

*//next function ,main. all variables inside it are "local" to the function.*

```
int main(void)  
{  
int k = 6; /* 'k' is local to the braces that contain it */  
printf("%d\n %d\n", k,i); //prints a '6' to the screen and next line is 5  
return 0;  
}
```



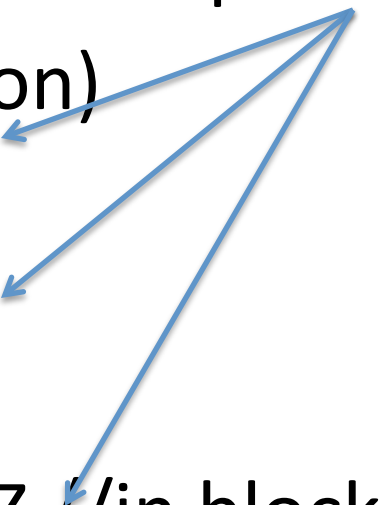
# First, What's a Code Block?

- Every opening and closing brace { } is a block.
- In C, you can have a block inside a block:

```
int main() { //start block 1
    while (someCondition)
        { // start block 2
            someStatements
        } //end block 2
    someStatementsXYZ //in block 1
}
```

# block 1 vars have scope in block 1

```
int main() { int a; //a's scope is  
    while (someCondition)  
    { // start block 2  
        someStatements  
    } //end block 2  
    someStatementsXYZ //in block 1  
} //end block 1
```



The diagram illustrates the scope of the variable `a`. Three blue arrows originate from the declaration `int a;` and point to its uses: the first arrow points to the opening curly brace of the `while` loop (the start of block 2), the second arrow points to the `someStatements` line inside the `while` loop, and the third arrow points to the `someStatementsXYZ` line outside the `while` loop (still within block 1). This demonstrates that `a` is in scope throughout block 1 and within block 2.

# Block 2 vars have scope inside block 2

```
int main() { int a; //a's scope is
    while (someCondition) {//start block 2
int b; //b's scope is online inside block 2
someStatements
    } //end block 2
    someStatementsXYZ //in block 1 – b is undeclared
} //end block 1
```

# It gets confusing when block 1 vars and block 2 vars have the same name

```
/* the main function */
int main(void)
{
/* this is the beginning of a 'block'*/
int i = 6; /* this is the first variable of this 'block', 'i' */
    { /* this is a new 'block', and because it's a different block, it has its own
      ((smaller))
        scope */
        /* this is also a variable called 'i', but in a different 'block', because it's in a
          different 'block' then the old 'i', it doesn't affect the old one! */
        int i = 5;
        printf("%d\n", i); /* prints a '5' onto the screen */
    }
    /* now we're back into the old block */
    printf("%d\n", i); /* prints a '6' onto the screen */
    return 0;
}
```

Inner block i scope is the bolded text, while outer I scope is over all main but loses its “visibility” while inside the inner block

```
/* the main function */
int main(void)
{
/* this is the beginning of a 'block'*/
int i = 6; /* this is the first variable of this 'block', 'i' */
{ /* this is a new 'block', and because it's a different block, it has its own  
((smaller)) scope */
/* this is also a variable called 'i', but in a different 'block', because it's in a  
different 'block' then the old 'i', it doesn't affect the old one! */
int i = 5;
printf("%d\n", i); /* prints a '5' onto the screen */
}
/* now we're back into the old block */
printf("%d\n", i); /* prints a '6' onto the screen */
return 0;
```

# Confusing?

- It is confusing to:
  - Use block variables, so DON'T
  - Use variables with the same name but in different scopes, so DON'T
- Programming is not about confusion, but about write CLEAR and CLEAN code
- Using Block variables is not clean and clear
- Know how it works but avoid it

# 3-Scopes Demo!

```
int i = 5; /* this is a 'global' variable, anywhere in the program can access it */  
/* this is a function, all variables inside of it are "local" to the function. */  
int main(void)  
{  
    /* this is the beginning of a 'block' */  
    int i = 6; /* this is the first variable of this 'block', 'i' */  
    {  
        /* this is a new 'block', and because it's a different block, it has its own scope */  
        /* this is also a variable called 'i', but in a different 'block', because it's in a  
different 'block' then the old 'i',  
        it doesn't affect the old one! */  
        int i = 5;  
        printf("%d\n", i); /* prints a '5' onto the screen */  
    }  
    /* now we're back into the old block */  
    printf("%d\n", i); /* prints a '6' onto the screen */  
    return 0;
```

# Time to practice Scope

- So far we covered two major topics:
  - Functions (really major)
  - Scope
- Next we cover Type Casting.
- Practice, and read the book sections Scope under Variables, and skim through Other Modifiers for the fun of it. We will not cover these scope types: static, extern(al), volatile, auto, and register.



# Summary

- Defining Functions
- Parameter Passing Mechanisms
- Scope