

# C++ Functions

Variable Scope

Storage Class

Recursion

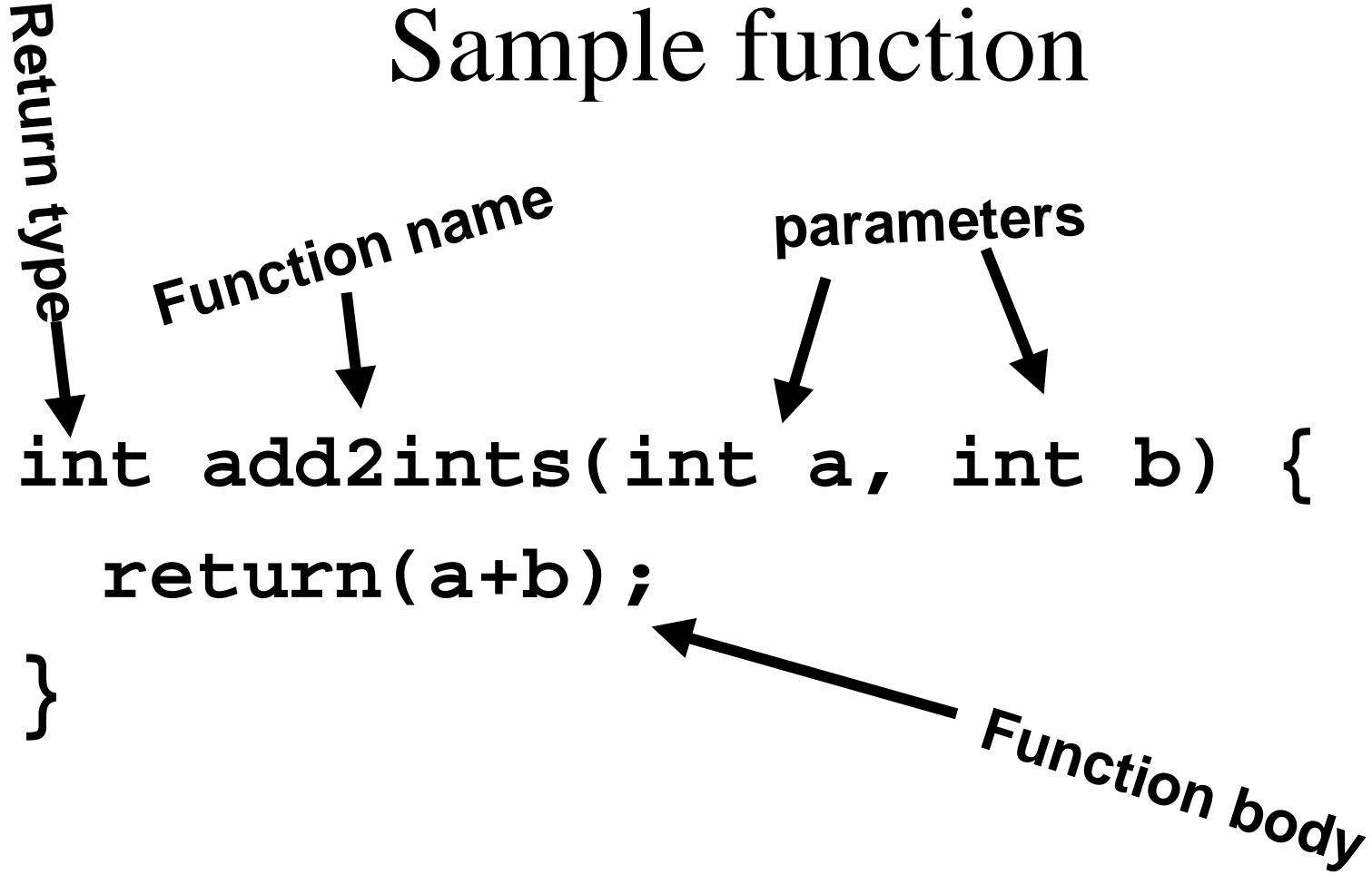
# C++ Functions

- In other languages called subroutines or procedures.
- C++ functions all have a *type*.
  - Sometimes we don't need to have a function return anything – in this case the function can have type **void**.

# C++ Functions (cont.)

- C++ functions have a list of *parameters*.
  - Parameters are the things we give the function to operate on.
    - Each parameter has a *type*.
  - There can be zero parameters.

# Sample function



# Using functions – Math Library functions

- C++ includes a library of Math functions you can use.
- You have to know how to *call* these functions before you can use them.
- You have to know what they return.
- You don't have to know how they work!

**double sqrt( double )**

- When *calling* **sqrt**, we have to give it a **double**.
- The **sqrt** function returns a **double**.
- We have to give it a **double**.

```
x = sqrt(y);  
x = sqrt(100);
```

**x = sqrt(y) ;**

- The stuff we give a function is called the argument(s). **Y** is the argument here.
- A C++ function can't change the value of an argument!
- If **y** was **100** before we call **sqrt**, it will always be **100** after we call **sqrt**.

# Table of square roots

```
int i;  
for (i=1;i<10;i++)  
    cout << sqrt(i) << "\n";
```

- But I thought we had to give `sqrt ( )` a double?
- C++ does automatic *type conversion* for you.

# Telling the compiler about `sqrt()`

- How does the compiler know about `sqrt` ?
- You have to tell it:

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

# Other Math Library Functions

**ceil**

**floor**

**cos**

**sin**

**tan**

**exp**

**log**

**log10**

**pow**

**fabs**

**fmod**

# Writing a function

- You have to decide on what the function will *look* like:
  - Return type
  - Name
  - Types of parameters (number of parameters)
- You have to write the body (the actual code).

# Function parameters

- The parameters are *local variables* inside the body of the function.
  - When the function is called they will have the values *passed in*.
  - The function gets *a copy* of the values passed in (we will later see how to pass a *reference* to a variable).

# Sample Function

```
int add2nums( int firstnum, int secondnum ) {  
    int sum;  
  
    sum = firstnum + secondnum;  
  
    // just to make a point  
    firstnum = 0;  
    secondnum = 0;  
  
    return(sum);  
}
```

# Testing add2nums

```
int main(void) {  
    int y,a,b;  
  
    cout << "Enter 2 numbers\n";  
    cin >> a >> b;  
  
    y = add2nums(a,b);  
  
    cout << "a is " << a << endl;  
    cout << "b is " << b << endl;  
    cout << "y is " << y << endl;  
    return(0);  
}
```

# What happens here?

```
int add2nums(int a, int b) {  
    a=a+b;  
    return(a);  
}
```

...

```
int a,b,y;
```

...

```
y = add2nums(a,b);
```

# Local variables

- Parameters and variables declared inside the definition of a function are *local*.
- They only exist inside the function body.
- Once the function returns, the variables no longer exist!
  - That's fine! We don't need them anymore!

# Block Variables

- You can also declare variables that exist only within the *body* of a compound statement (*a block*):

```
{  
  int foo;  
  ...  
  ...  
}
```

# Global variables

- You can declare variables outside of any function definition – these variables are *global variables*.
- Any function can access/change global variables.
- Example: flag that indicates whether debugging information should be printed.

# Scope

- The *scope* of a variable is the portion of a program where the variable has meaning (where it exists).
- A global variable has global (unlimited) scope.
- A local variable's scope is restricted to the function that declares the variable.
- A block variable's scope is restricted to the block in which the variable is declared.


# A note about Global vs. File scope

- A variable declared outside of a function is available everywhere, but only the functions that follow it in the file know about it.
- The book talks about *file scope*, I'm calling it *global scope*.

# Block Scope

```
int main(void) {  
    int y;  
  
    {  
        int a = y;  
        cout << a << endl;  
    }  
    cout << a << endl;  
}
```

**Error – a doesn't exist  
outside the block!**



# Nesting

- In C++:
  - There is no nesting of function definitions.
    - You don't need to know who calls a function to know the scope of its variables!
  - There is nesting of variable scope in blocks.

# Nested Blocks

```
void foo(void) {  
    .....  
    for (int j=0;j<10;j++) {  
        .....  
        int k = j*10;  
        cout << j << "," << k << endl;  
        {  
            .....  
            int m = j+k;  
            cout << m << "," << j << endl;  
            .....  
        }  
        .....  
    }  
    .....  
}
```

The diagram illustrates the scope of variables `j`, `k`, and `m` in the provided C++ code. Three vertical bars represent the lifetime of each variable:

- The bar for `j` spans the entire duration of the `foo` function.
- The bar for `k` spans the duration of the `for` loop.
- The bar for `m` spans only the innermost block.

# Storage Class

- Each variable has a *storage class*.
  - Determines the period during which the variable exists *in memory*.
  - Some variables are created only once (memory is set aside to hold the variable value)
    - Global variables are created only once.
  - Some variables are re-created many times
    - Local variables are re-created each time a function is called.

# Storage Classes

- **auto** – created each time the block in which they exist is *entered*.
- **register** – same as **auto**, but tells the compiler to make as fast as possible.
- **static** – created only once, even if it is a local variable.
- **extern** – global variable declared elsewhere.

# Specifying Storage Class

```
auto int j;
```

```
register int i_need_to_be_fast;
```

```
static char remember_me;
```

```
extern double a_global;
```

# Practical Use of Storage Class

- Local variables are **auto** by default.
- Global variables are **static** by default.
- Declaring a local variable as **static** means it will *remember* it's last value (it's not destroyed and recreated each time it's scope is entered).

# static example

```
int countcalls(void) {  
    static int count = 0;  
    count++;  
    return(count);  
}  
  
...  
cout << countcalls() << endl;  
cout << countcalls() << endl;  
cout << countcalls() << endl;
```

# The Scope of Functions

- In C++ we really talk about the scope of an identifier (name).
  - Could be a function or a variable (or a class).
- Function names have *file* scope
  - everything that follows a function definition in the same file can use the function.
- Sometimes this is not convenient
  - We want to call the function from the top of the file and define it at the bottom of the file.

# Function Prototypes

- A Function prototype can be used to *tell* the compiler what a function looks like
  - So that it can be called even though the compiler has not yet seen the function definition.
- A function prototype specifies the function name, return type and parameter types.

# Example prototypes

```
double sqrt( double );
```

```
int add2nums( int, int );
```

```
int counter(void);
```

# Using a prototype

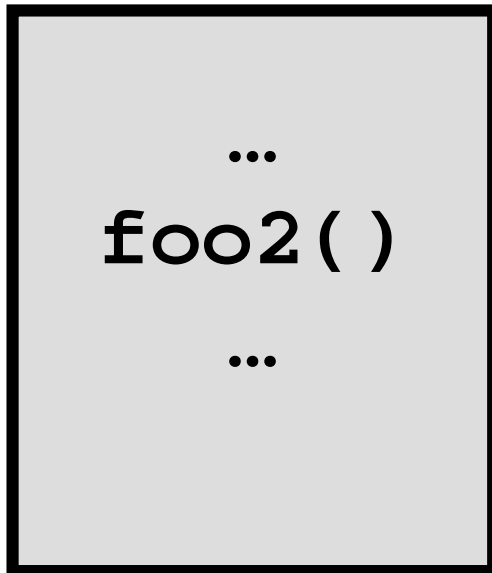
```
int counter(void);
```

```
int main(void) {  
    cout << counter() << endl;  
    cout << counter() << endl;  
    cout << counter() << endl;  
}
```

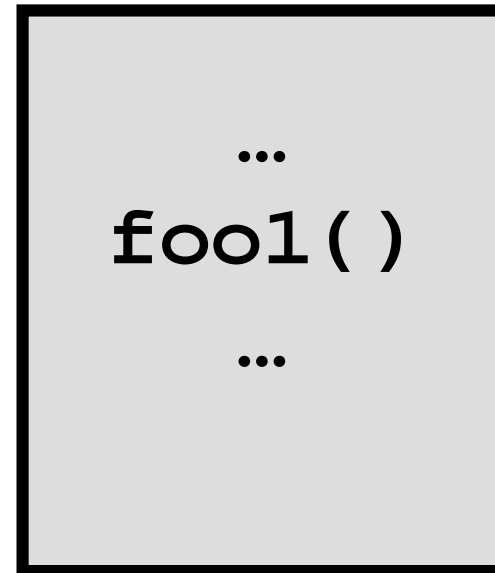
```
int counter(void) {  
    static int count = 0;  
    count++;  
    return(count);  
}
```

# Functions that call each other

**foo1**



**foo2**



# Dualing Functions

```
char *chicken( int generation ) {  
    if (generation == 0)  
        return("Chicken!");  
    else  
        return(egg(generation-1));  
}
```

```
char *egg( int generation ) {  
    if (generation == 0)  
        return("Egg!");  
    else  
        return(chicken(generation-1));  
}
```

# The rest of chicken vs. egg

```
char *egg( int );
char *chicken( int );

int main(void) {
    int startnum;

    cout << "Enter starting generation of
your chicken" << endl;
    cin >> startnum;
    cout << "Your chicken started as a " <<
        chicken(startnum) << endl;
    return(0);
}
```

# Recursion

- Functions can call themselves! This is called recursion.
- Recursion is very useful – it's often very simple to express a complicated computation recursively.

# The Recursive Chicken

(a new dance fad!)

```
char *chicken_or_egg( int gen ) {  
    if (gen == 0)  
        return( "Chicken!" );  
    else if (gen == 1)  
        return( "Egg!" );  
    else  
        return( chicken_or_egg( gen-1 ) );  
}
```

# A Better Example - Computing Factorials

```
int factorial( int x ) {  
    if (x == 1)  
        return(1);  
    else  
        return(x * factorial(x-1));  
}
```

# Designing Recursive Functions

- Define “Base Case”:
  - The situation in which the function does **not** call itself.
- Define “recursive step”:
  - Compute the return value the help of the function itself.

# Recursion Base Case

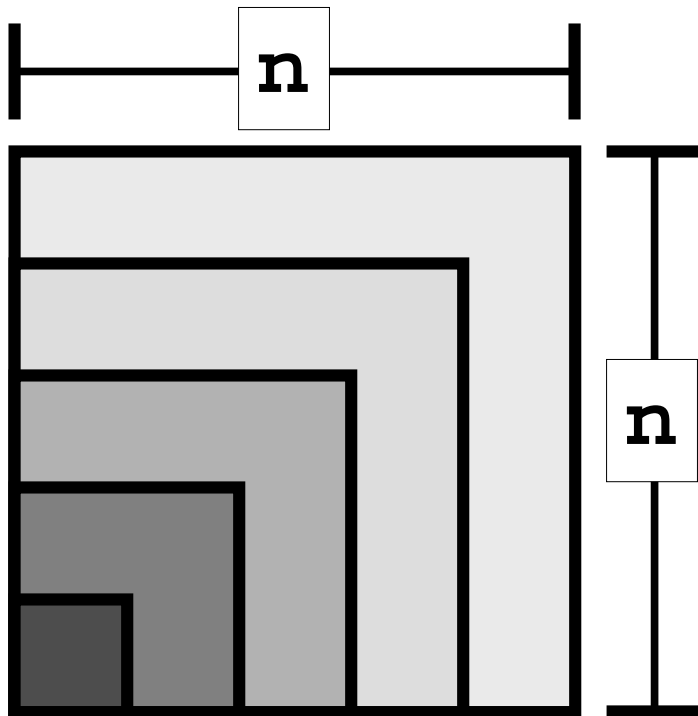
- The base case corresponds to a case in which you know the answer (the function returns the value immediately), or can easily compute the answer.
- If you don't have a base case you can't use recursion! (and you probably don't understand the problem).

# Recursive Step

- Use the recursive call to solve a **sub-problem**.
  - The parameters must be different (or the recursive call will get us no closer to the solution).
  - You generally need to do something besides just making the recursive call.

# Recursion is a favorite test topic

- Write a recursive C++ function that computes the area of an  **$n \times n$**  square.



**Base case:**

**$n=1$  area=1**

**Recursive Step:**

**$\text{area} = n + n - 1 + \text{area}(n - 1)$**

# Recursive area function

```
int area( int n) {  
    if (n == 1)  
        return(1);  
    else  
        return( n + n - 1 + area(n-1) );  
}
```

# Recursion Exercise

- Write a function that prints a triangle:

```
triangle(4);
```

```
  *
 * * *
* * * * *
* * * * * * *
```

```
triangle(5);
```

```
  *
   * * *
  * * * * *
 * * * * * *
* * * * * * * *
```

# Call-by-value vs. Call-by-reference

- So far we looked at functions that get a copy of what the *caller* passed in.
  - This is call-by-value, as the value is what gets passed in (the value of a variable).
- We can also define functions that are passed a *reference* to a variable.
  - This is call-by-reference, the function can change a callers variables directly.

# References

- A *reference* variable is an alternative name for a variable. A *shortcut*.
- A reference variable must be initialized to *reference* another variable.
- Once the reference is initialized you can treat it just like any other variable.

# Reference Variable Declarations

- To declare a reference variable you precede the variable name with a “&”:

```
int &foo;  
double &blah;  
char &c;
```

# Reference Variable Example

```
int count;  
int &blah = count;  
// blah is the same variable as count  
  
count = 1;  
cout << "blah is " << blah << endl;  
blah++;  
cout << "count is " << count << endl;
```


# Reference Parameters

- You can declare reference parameters:

```
void add10( int &x) {  
    x = x+10;  
}
```

...

```
add10(counter);
```

The parameter is a reference

# Useful Reference Example

```
void swap( int &x, int &y) {  
    int tmp;  
    tmp = x;  
    x = y;  
    y = tmp;  
}
```