Plan for today

- Scope
- Functions in C++
- Header files

Not So Fast!

- So far, we've learned data basic data manipulation and control structures

Not So Fast!

- So far, we've learned data basic data manipulation and control structures
- With this subset of the C++ language, we are able to write any program/algorithm

So are we done??

Key Concept: Abstraction

- For reasonable scalability, we have to manage complexity
- Use abstraction to build new tools to improve expressive power of the language

But first, a note on **scope**...

Scope in C++

- Scope defines a variable's lifespan

Scope in C++

- Scope defines a variable's lifespan
- A variable declared within a particular scope will die once you exit that scope

Example on Scope

- Will the following code compile?

```
if (a > 3) {
    int b = a + 1;
    a = 0;
}
cout << b << endl;
```

Example on Scope

- Will the following code compile?

```
if (a > 3) {
    int b = a + 1;
    a = 0;
}
cout << b << endl;
```

- No! b does not exist in this scope!!!

Easy rule to remember:
scope is defined by { }

The Big Picture

- Variables that don't live inside { } have **global scope**
- It means they won't go away until the program terminates...
- Useful for global constants, but in general, bad coding practice!!!

Global Scope in Action

```
int magic = 14;
int main( ) {
    cout << “The magic number is “ << magic << endl;
    return 0;
}
```

- No magic, just bad code!

Functions

Functions

- We have actually used functions already
- rand and srand

```
int rand(void);
void srand(unsigned int seed);
```

Calling Functions

- Parentheses after an identifier is interpreted as a function call
- Function arguments delimited by comma

```
srand(mySeed);
int myRandom = rand( );
```

Calling a Function **passes control** to that function

Calling a Function **passes control** to that function

Upon callee **return**, control is passed back to the caller

So What About Custom Functions?

- I can also define my own functions to call within my program

Declaration v.s. Definition

Informal Definition of a Declaration

- A Declaration specifies the function signature
  - Name of the function (symbol)
  - Return type
  - Parameter types

Informal Definition of a Declaration

- A Declaration specifies the function signature
  - Name of the function (symbol)
  - Return type (*not part of the signature)
  - Parameter types
- Function signature == function prototype (interchangeable)

Informal Definition of a Declaration

- A Declaration specifies the function signature
  - Name of the function (symbol)
  - Return type
  - Parameter types
- Function signature == function prototype (interchangeable)
- A note to the compiler that the function exists

Multiple Declaration

```
int foo(int a, int b);
int foo(int c, int d);
int foo(int e, int f);
int main( ) {
    …
    return 0;
}
```

- Will this compile?

Multiple Definition

```
int foo(int c, int d) { return c*d; }
int foo(int e, int f) { return e*f; }
int main( ) {
    ...
    return 0;
}
```

How about this?

Declaration vs Definition

- A declaration indicates to the compiler that a function exists
  - In general, applies to all “symbols
  - Can be replicated many times throughout the program
- A definition is the implementation of the function
  - Can only appear once

Some Examples

Write a function that implements **pow**

x^y = pow(x,y)

Function Example 1

```
double pow(double x, int y) {
    double result = 1;
    for (int i=0; i<y; ++i)
        result *= x;

    return result;
}
```

Write a function that determines whether an input string only contains alphabetic characters

```
bool alphaString(string input) {
    for (int i=0; i<input.length(); i++)
        if (!isAlpha(input[i]))
            return false;
    return true;
}
bool isAlpha(char c) {
    return (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z');
```

Functions Overloaded!

Q: What is the result of the following code

```
void foo(int a, int b) {
   cout << “two argument foo”;
}
void foo(int a) {
    cout << “one argument foo”;
}
int main( ) {
    foo(1);
}
```

Functions Overloaded!

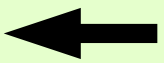
Q: What about this code?

```
void bar(int a, int b) {
    cout << “int arguments”;
}
void bar(double a, double b) {
   cout << “double trouble”;
}
int main( ) {
    foo(1,1);
    foo(2,3.0)
}
```

Functions Overloaded!

Q: What about this code?

```
void bar(int a, int b) {
    cout << “int arguments”;
}
void bar(double a, double b) {
   cout << “double trouble”;
}
int main( ) {
    foo(1,1);
    foo(2,3.0)
}
```



This is ambiguous to the compiler

Functions Overloaded!

Q: What about this code?

```
void foo(int a) {
    cout << “void foo”;
}
int foo(int a) {
   cout << “int foo”; return a;
}
int main( ) {
    int b = foo(1);
}
```

Functions Overloaded!

Q: What about this code?

```
void foo(int a) {
    cout << "void foo";
}
int foo(int a) {
   cout << "int foo"; return a;
}
int main( ) {
    int b = foo(1);
}
```

Also ambiguous

Return type is not part of the signature

Parameters

```
void bar(int foo) {
    foo *= foo;
    cout << “foo is “ << foo << endl;
}
int main( ) {
    int foo = 10;
    bar(foo);
    cout << “foo is “ << foo << endl;
}
```

Q: What is the output of this program

Parameters

```
void bar(int foo) {
    foo *= foo;
    cout << “foo is “ << foo << endl;
}
int main( ) {
    int foo = 10;
    bar(foo);
    cout << “foo is “ << foo << endl;
}
```

foo is 100
foo is 10

Parameters

```
void bar(int foo) {
    foo *= foo;
    cout << “foo is “ << foo << endl;
}
int main( ) {
    int foo = 10;
    bar(foo);
    cout << “foo is “ << foo << endl;
}
```

foo is 100
foo is 10

- **This is because foo is passed by value**

Passing By Value

- Recall scope... It applies to functions too!

Passing By Value

- Recall scope... It applies to functions too!
- A variable declared within a functions scope does not exist outside of the function
  - i.e. if a function declares local variables then they are not accessible by the outside world

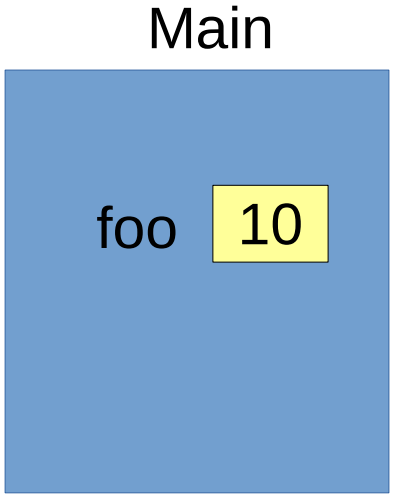
Passing By Value

- Recall scope... It applies to functions too!
- A variable declared within a functions scope does not exist outside of the function
  - i.e. if a function declares local variables then they are not accessible by the outside world
- Pass by value parameters are local to a function

Example Revisted

```
void bar(int foo) {
    foo *= foo;
    cout << “foo is “ << foo << endl;
}
```

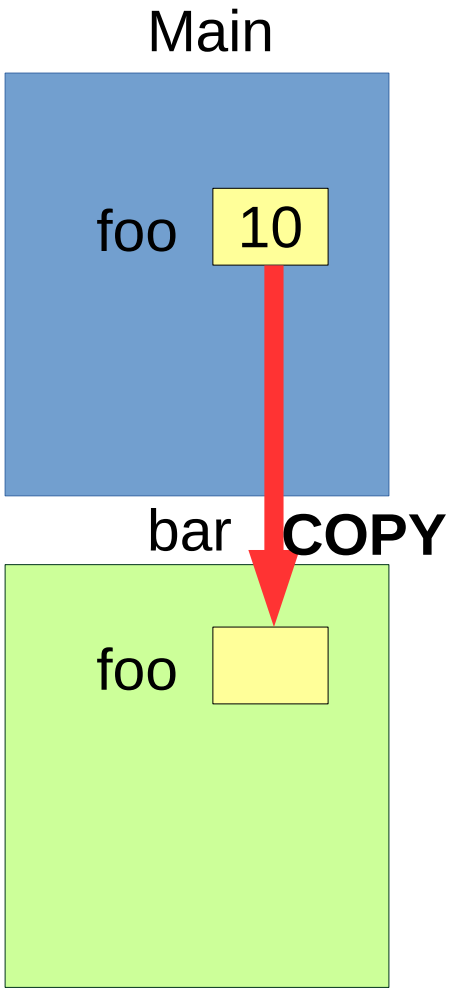
```
int main( ) {
    int foo = 10;
    bar(foo);
    cout << “foo is “ << foo << endl;
}
```



Example Revisted

```
void bar(int foo) {
    foo *= foo;
    cout << “foo is “ << foo << endl;
}
```

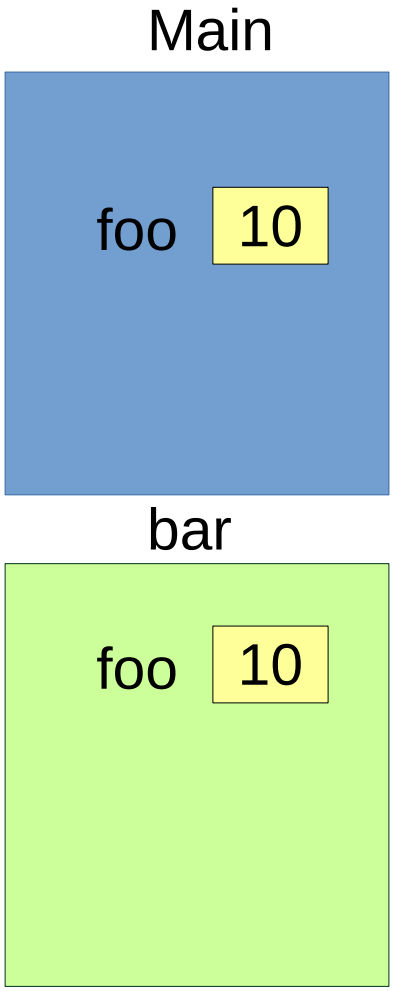
```
int main( ) {
    int foo = 10;
    bar(foo);
    cout << “foo is “ << foo << endl;
}
```



Example Revisted

```
void bar(int foo) {
    foo *= foo;
    cout << “foo is “ << foo << endl;
}
```

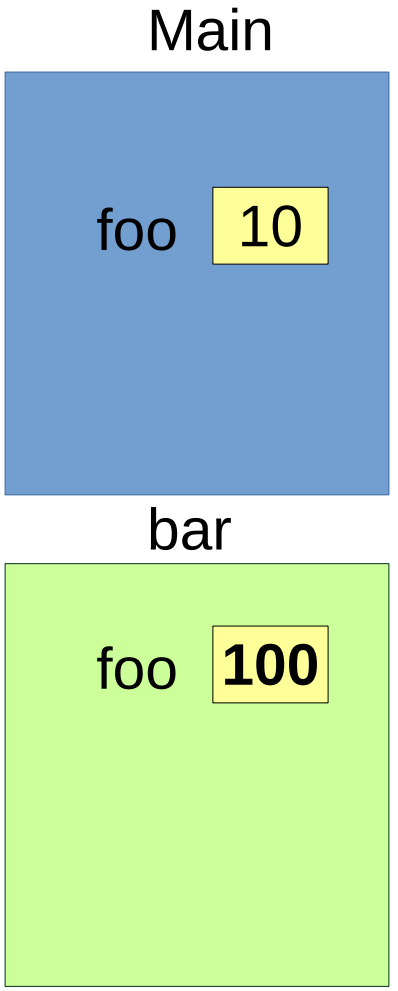
```
int main( ) {
    int foo = 10;
    bar(foo);
    cout << “foo is “ << foo << endl;
}
```



Example Revisted

```
void bar(int foo) {
    foo *= foo;
    cout << “foo is “ << foo << endl;
}
```

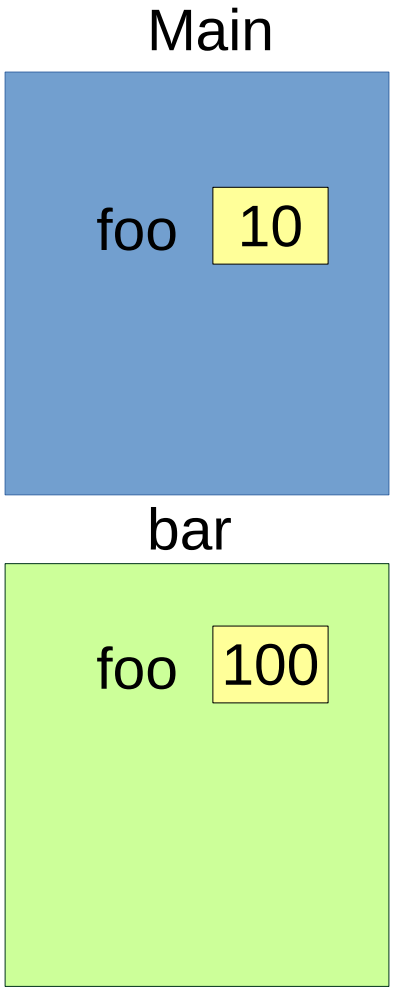
```
int main( ) {
    int foo = 10;
    bar(foo);
    cout << “foo is “ << foo << endl;
}
```



Example Revisted

```
void bar(int foo) {
    foo *= foo;
    cout << “foo is “ << foo << endl;
}
```

```
int main( ) {
    int foo = 10;
    bar(foo);
    cout << “foo is “ << foo << endl;
}
```



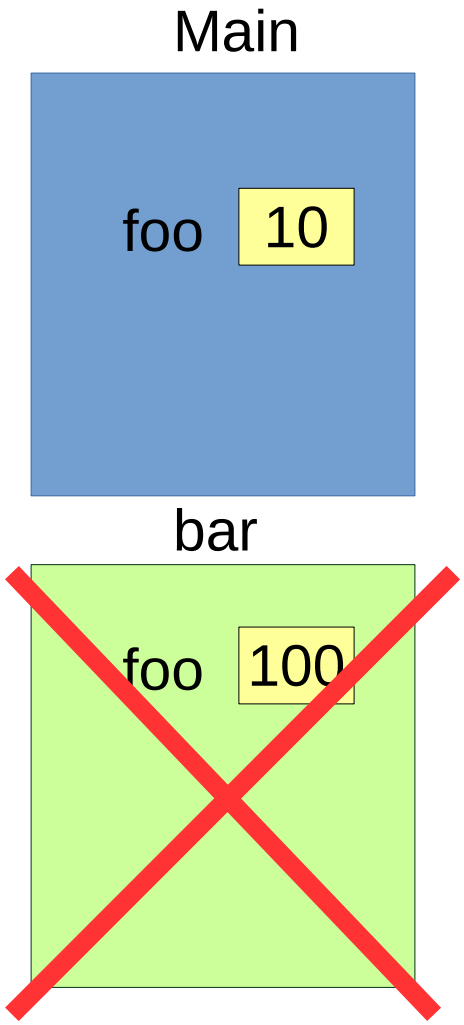
foo is 100

Example Revisted

```
void bar(int foo) {
    foo *= foo;
    cout << “foo is “ << foo << endl;
}
```

```
int main( ) {
    int foo = 10;
    bar(foo);
    cout << “foo is “ << foo << endl;
}
```

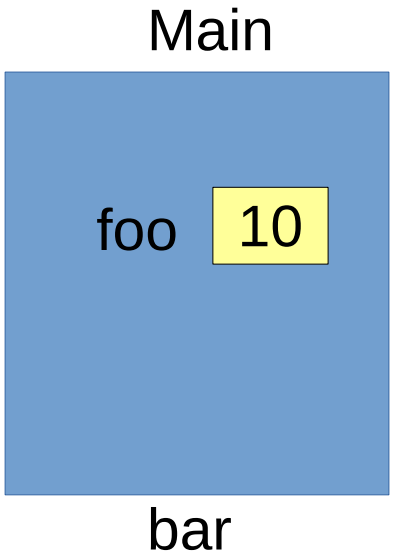
foo is 100



Example Revisted

```
void bar(int foo) {
    foo *= foo;
    cout << “foo is “ << foo << endl;
}
```

```
int main( ) {
    int foo = 10;
    bar(foo);
    cout << “foo is “ << foo << endl;
}
```



foo is 100
foo is 10

Pass By Value Summary

- Parameters passed to a function are just like variables declared in that functions local scope
- A variables in a particular scope are destroyed once you exit that scope
- So the only thing that survives after a function terminates is the return value

OK, then what is Pass By Reference?

Passing By Reference

- If we want more from the function than just the return value, use pass by reference

Passing By Reference

- If we want more from the function than just the return value, use pass by reference
- Don't copy the value of the argument when function is called

Passing By Reference

- If we want more from the function than just the return value, use pass by reference
- Don't copy the value of the argument when function is called
- Create a reference to the original object
  - Can edit this value by calling the functionS
  - Saves additional information from the result of the function call

Pass By Reference

- To indicate a parameter is pass by reference, write an '&' after indicating the type

```
bool eatCookies(int &numOfCookies);
```

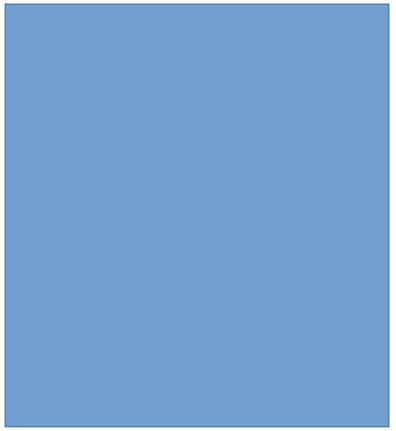
Example Revisited Again

```
void bar(int &foo) {
    foo *= foo;
    cout << "foo is " << foo << endl;
}
```

Note: Pass by Reference

```
int main( ) {
    int foo = 10;
    bar(foo);
    cout << "foo is " << foo << endl;
}
```

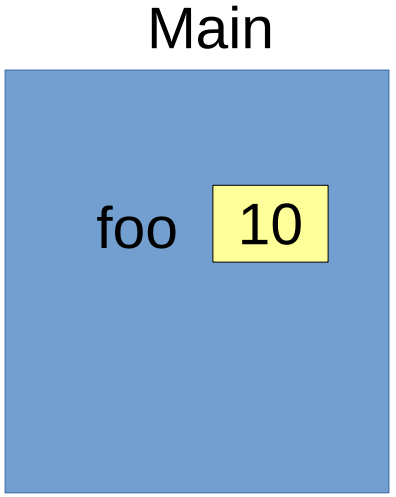
Main



Example Revisited Again

```
void bar(int &foo) {
    foo *= foo;
    cout << “foo is “ << foo << endl;
}
```

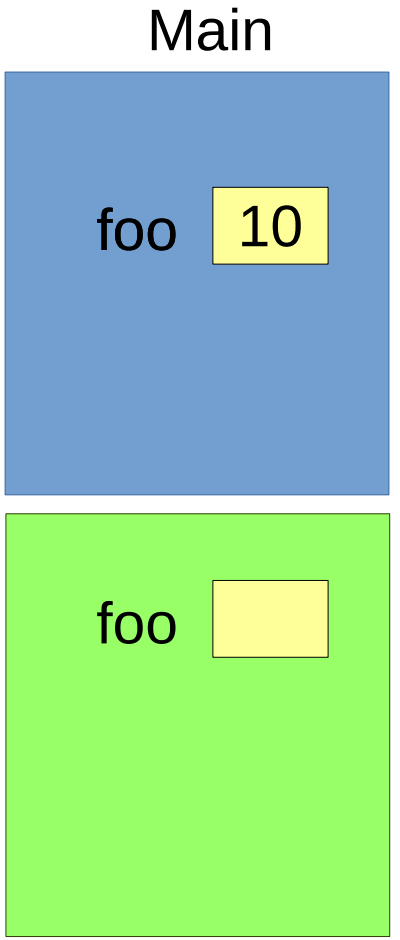
```
int main( ) {
    int foo = 10;
    bar(foo);
    cout << “foo is “ << foo << endl;
}
```



Example Revisited Again

```
void bar(int &foo) {
    foo *= foo;
    cout << “foo is “ << foo << endl;
}
```

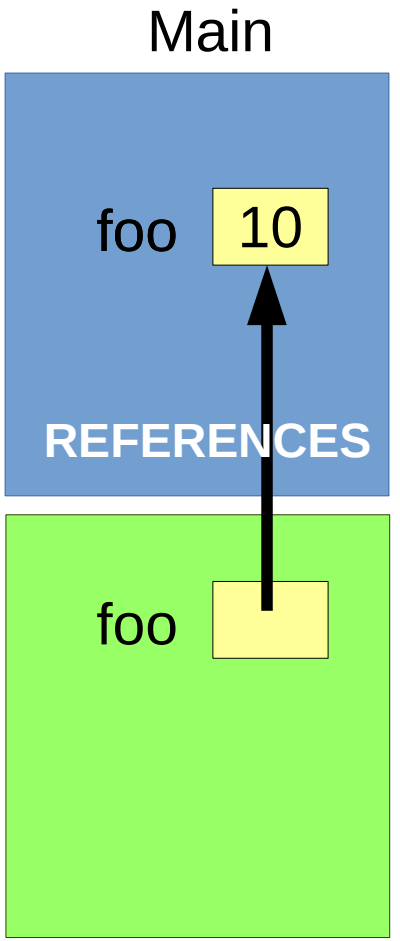
```
int main( ) {
    int foo = 10;
    bar(foo);
    cout << “foo is “ << foo << endl;
}
```



Example Revisited Again

```
void bar(int &foo) {
    foo *= foo;
    cout << “foo is “ << foo << endl;
}
```

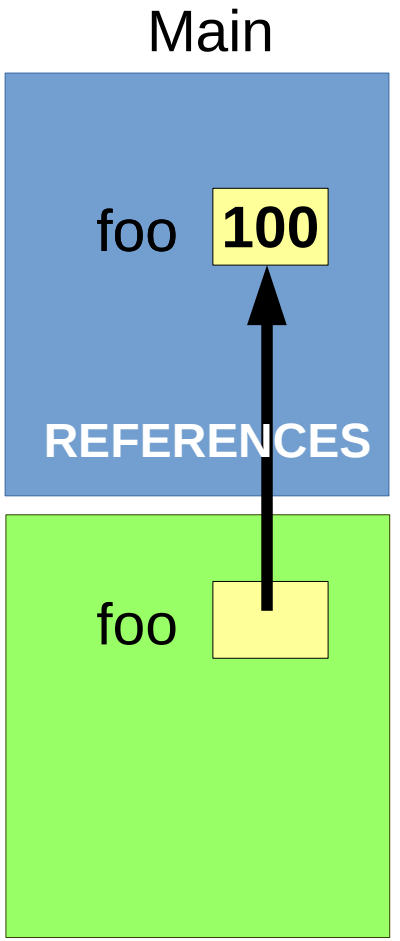
```
int main( ) {
    int foo = 10;
    bar(foo);
    cout << “foo is “ << foo << endl;
}
```



Example Revisited Again

```
void bar(int &foo) {
    foo *= foo;
    cout << “foo is “ << foo << endl;
}
```

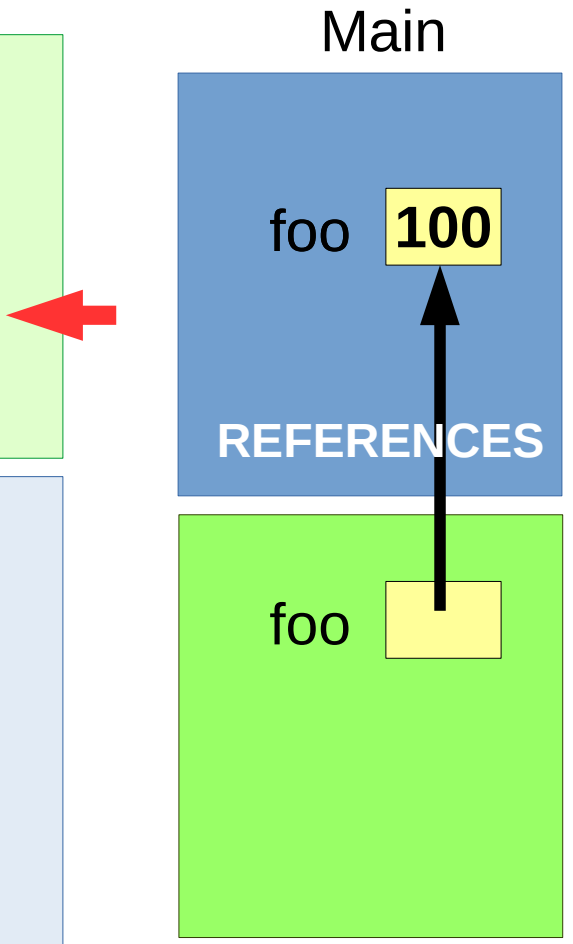
```
int main( ) {
    int foo = 10;
    bar(foo);
    cout << “foo is “ << foo << endl;
}
```



Example Revisited Again

```
void bar(int &foo) {
    foo *= foo;
    cout << “foo is “ << foo << endl;
}
```

```
int main( ) {
    int foo = 10;
    bar(foo);
    cout << “foo is “ << foo << endl;
}
```

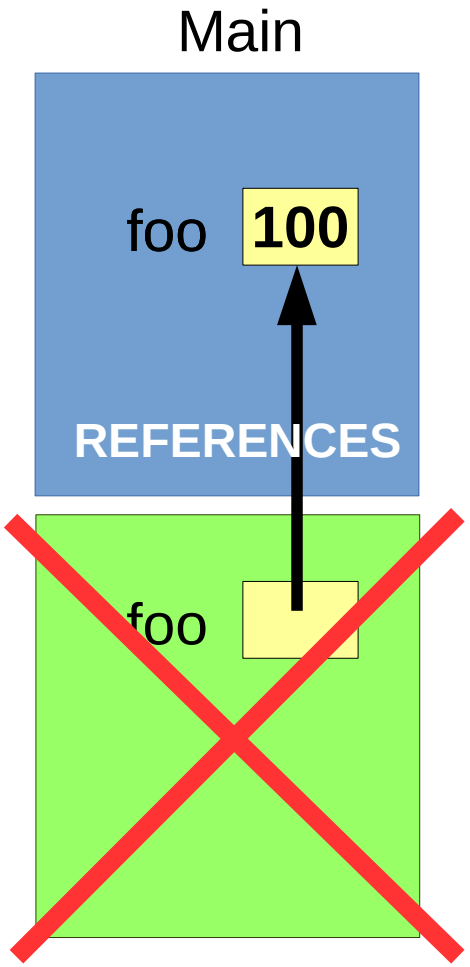


foo is 100

Example Revisited Again

```
void bar(int &foo) {
    foo *= foo;
    cout << "foo is " << foo << endl;
}
```

```
int main( ) {
    int foo = 10;
    bar(foo);
    cout << "foo is " << foo << endl;
}
```

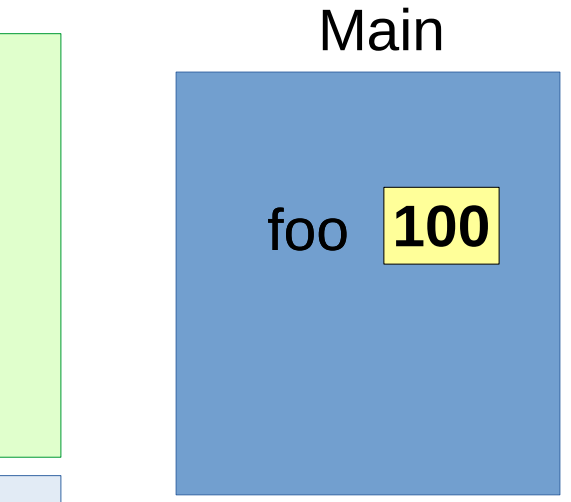


foo is 100

Example Revisited Again

```
void bar(int &foo) {
    foo *= foo;
    cout << “foo is “ << foo << endl;
}
```

```
int main( ) {
    int foo = 10;
    bar(foo);
    cout << “foo is “ << foo << endl;
}
```



foo is 100
foo is 100

Pass By Reference Summary

- A pass by reference parameter of a function is a reference (pointer) to the variable in the caller's scope
  - It's scope is not local to the function call
- Changes made within the function are seen after the function returns
- Pass by reference can be useful, but also dangerous

C++ Header Files

So we have seen things like this:

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

So we have seen things like this:

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

But what is it??

Including Header Files

- # indicates a preprocessor directive

Including Header Files

- # indicates a preprocessor directive
- Its not compilable C++ code (yet)

Including Header Files

- # indicates a preprocessor directive
- Its not compilable C++ code (yet)
- Preprocessor replaces all `#include` statements with the text of the header file

Including Header Files

- # indicates a preprocessor directive
- Its not compilable C++ code (yet)
- Preprocessor replaces all `#include` statements with the text of the header file
- Usually includes function declarations and global constant declarations

So Why Does This Work?

- We saw that multiple declarations of the same symbol is harmless
- Including header files tells compiler that the functions exist
  - They are defined elsewhere in libraries or other cpp files
- Compiler does not need to know what the function does
  - Just its signature

Quick Example

math.cpp

```
const double PI = 3.14;

double area(int radius){
        …
}

double
pow(double b, int p) {
        //Definitions here...
}

...
```

math.h

```
/* Declarations of math
 *  Functions  */

double area(int radius);

double
pow(double b, int p);
```

main.cpp

```
#include “math.h”

int main( ) {
        int x; double y;
        cout << pow(y,x);
        …
        return 0;
}
```

Quick Example

math.cpp

```
const double PI = 3.14;

double area(int radius){
        …
}

double
pow(double b, int p) {
        //Definitions here...
}

...
```

math.h

```
/* Declarations of math
 *  Functions  */

double area(int radius);

double
pow(double b, int p);
```

Contains declarations of library functions. May be included multiple times

main.cpp

```
#include “math.h”

int main( ) {
        int x; double y;
        cout << pow(y,x);
        …
        return 0;
}
```

Quick Example

math.cpp

```
const double PI = 3.14;

double area(int radius){
        …
}

double
pow(double b, int p) {
        //Definitions here...
}

...
```

Contains actual definitions of the functions.

math.h

```
/* Declarations of math
 *  Functions  */

double area(int radius);

double
pow(double b, int p);
```

main.cpp

```
#include “math.h”

int main( ) {
        int x; double y;
        cout << pow(y,x);
        …
        return 0;
}
```

Quick Example

math.cpp

```
const double PI = 3.14;

double area(int radius){
        …
}

double
pow(double b, int p) {
        //Definitions here...
}

...
```

math.h

```
/* Declarations of math
 *  Functions  */

double area(int radius);

double
pow(double b, int p);
```

main.cpp

```
#include “math.h”

int main( ) {
        int x; double y;
        cout << pow(y,x);
        …
        return 0;
}
```

All #include statements resolved in the preprocessing step

Quick Example

math.cpp

```
const double PI = 3.14;

double area(int radius){
        …
}

double
pow(double b, int p) {
        //Definitions here...
}

...
```

main.cpp

```
double area(int radius);

double
pow(double b, int p);

int main( ) {
        int x; double y;
        cout << pow(y,x);
        …
        return 0;
}
```

Now the compiler can generate object files for the 2 cpp's **independently**

Quick Example

math.cpp

```
const double PI = 3.14;

double area(int radius){
        …
}

double
pow(double b, int p) {
        //Definitions here...
}

...
```

main.cpp

```
double area(int radius);

double
pow(double b, int p);

int main( ) {
        int x; double y;
        cout << pow(y,x);
        …
        return 0;
}
```

Now the compiler can generate object files for the 2 cpp's independently

Resolving references to the math library functions is handled by the **linker**