Lecture 2

Functions

Functions in C++

Let’s take a look at an example declaration:

```
long factorial(int n)
```

The declaration above has the following meaning:

- The return type is `long`. That means the function will return a long integer to the calling function.
- The name of the function is `factorial`. When we need to call this function we will use this name.
- The function takes one parameter which is an integer variable named `n`.

Functions in C++

How might our factorial function be implemented?

```
long factorial(int n)
{
  long result = 1, k = n;
  while(k > 1)
  {
    result *= k--;
  }
  return result;
}
```

- Note the use of the post-decrement operator.
  - Why do we use postfix here instead of prefix?
- Note the use of `return` to return the value to the caller.

Functions in C++

How might we call our function from a main() function?

```
#include <iostream>
long factorial(int);    // forward declaration
int main()
{
  int x;
  cout << “Please enter a number> “ << endl;
  cin >> x;
  cout << x << “! is “ << factorial(x) << endl;
}
```

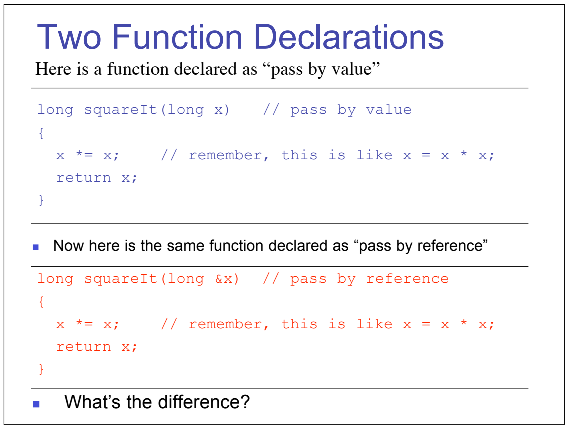
- Note forward declaration
  - Needed only if factorial() appears below main() in the file
  - Parameter names do not need to be specified but all types must!
- Function call--an expression which evaluates to its return value.
  - Could also be used in assignment

Demonstration #1

The factorial function

Argument Passing

- There are two ways to pass arguments to functions in C++:
  - Pass by VALUE
  - Pass by REFERENCE
- Pass by VALUE
  - The value of a variable is passed along to the function
  - If the function modifies that value, the modifications stay within the scope of that function.
- Pass by REFERENCE
  - A reference to the variable is passed along to the function
  - If the function modifies that value, the modifications appear also within the scope of the calling function.



Two Function Declarations

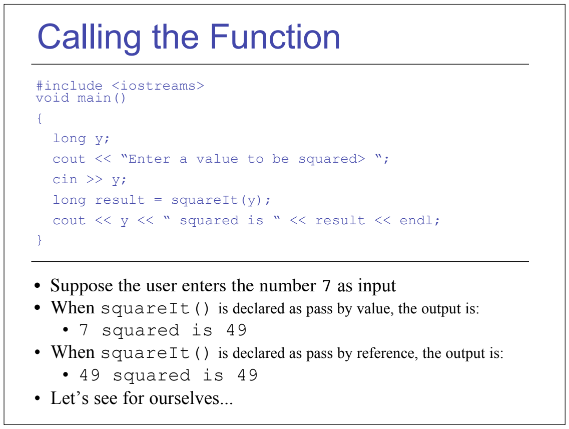
Here is a function declared as “pass by value”

```
long squareIt(long x)    // pass by value
{
   x *= x;     // remember, this is like x = x * x;
   return x;
}
```

- Now here is the same function declared as “pass by reference”

```
long squareIt(long &x)   // pass by reference
{
   x *= x;     // remember, this is like x = x * x;
   return x;
}
```

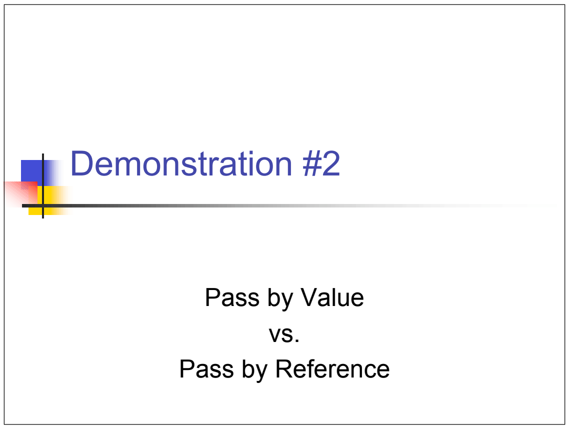
- What’s the difference?



Calling the Function

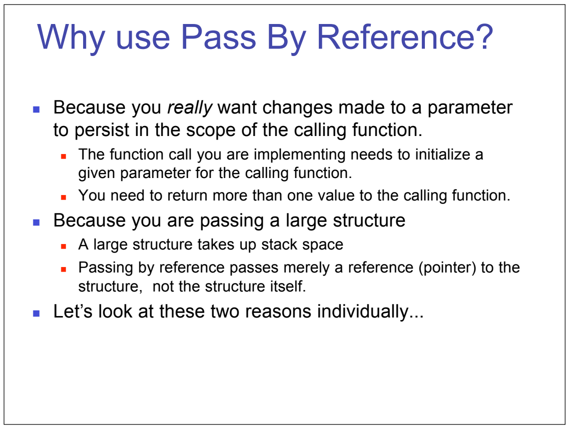
```
#include <iostreams>
void main()
{
  long y;
  cout << “Enter a value to be squared> “;
  cin >> y;
  long result = squareIt(y);
  cout << y << “ squared is “ << result << endl;
}
```

- Suppose the user enters the number 7 as input
- When `squareIt()` is declared as pass by value, the output is:
  - `7 squared is 49`
- When `squareIt()` is declared as pass by reference, the output is:
  - `49 squared is 49`
- Let’s see for ourselves...



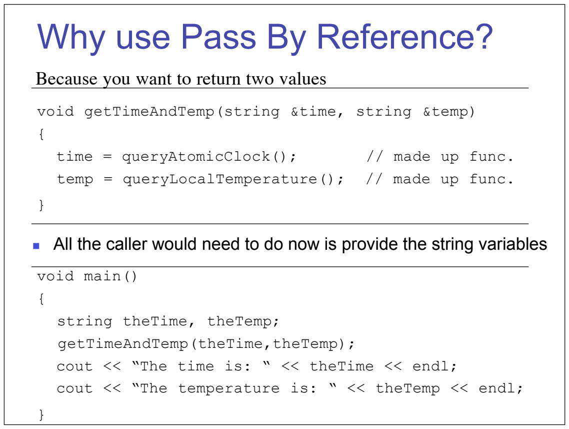
Demonstration #2

Pass by Value
vs.
Pass by Reference



Why use Pass By Reference?

- Because you *really* want changes made to a parameter to persist in the scope of the calling function.
  - The function call you are implementing needs to initialize a given parameter for the calling function.
  - You need to return more than one value to the calling function.
- Because you are passing a large structure
  - A large structure takes up stack space
  - Passing by reference passes merely a reference (pointer) to the structure, not the structure itself.
- Let’s look at these two reasons individually...



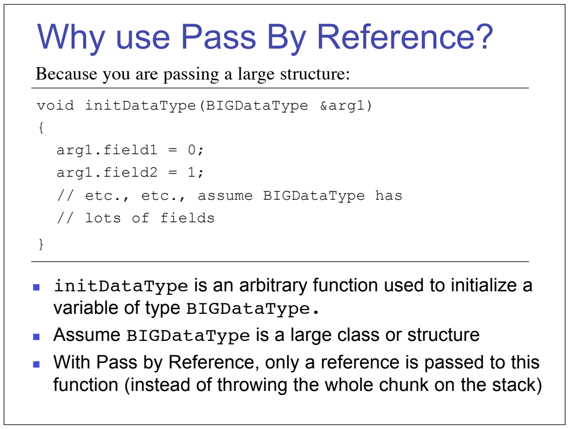
Why use Pass By Reference?

Because you want to return two values

```
void getTimeAndTemp(string &time, string &temp)
{
   time = queryAtomicClock();        // made up func.
   temp = queryLocalTemperature();   // made up func.
}
```

- All the caller would need to do now is provide the string variables

```
void main()
{
   string theTime, theTemp;
   getTimeAndTemp(theTime,theTemp);
   cout << “The time is: “ << theTime << endl;
   cout << “The temperature is: “ << theTemp << endl;
}
```

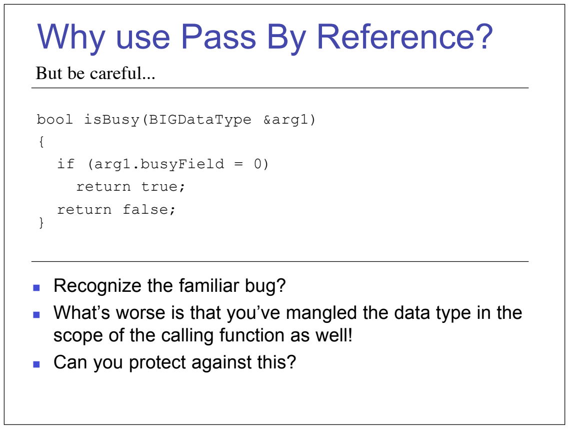


Why use Pass By Reference?

Because you are passing a large structure:

```
void initDataType(BIGDataType &arg1)
{
   arg1.field1 = 0;
   arg1.field2 = 1;
   // etc., etc., assume BIGDataType has
   // lots of fields
}
```

- `initDataType` is an arbitrary function used to initialize a variable of type `BIGDataType.`
- Assume `BIGDataType` is a large class or structure
- With Pass by Reference, only a reference is passed to this function (instead of throwing the whole chunk on the stack)

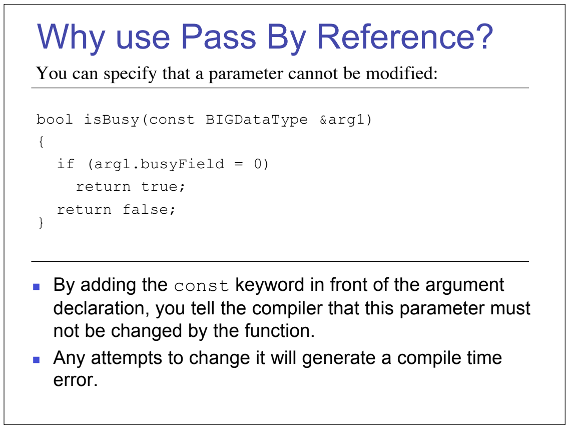


Why use Pass By Reference?

But be careful...

```
bool isBusy(BIGDataType &arg1)
{
  if (arg1.busyField = 0)
    return true;
  return false;
}
```

- Recognize the familiar bug?
- What's worse is that you've mangled the data type in the scope of the calling function as well!
- Can you protect against this?

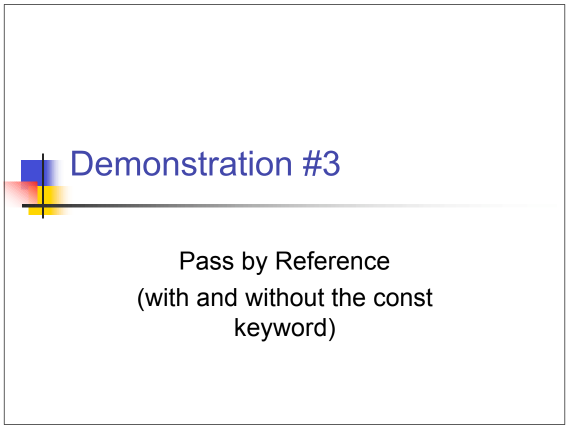


Why use Pass By Reference?

You can specify that a parameter cannot be modified:

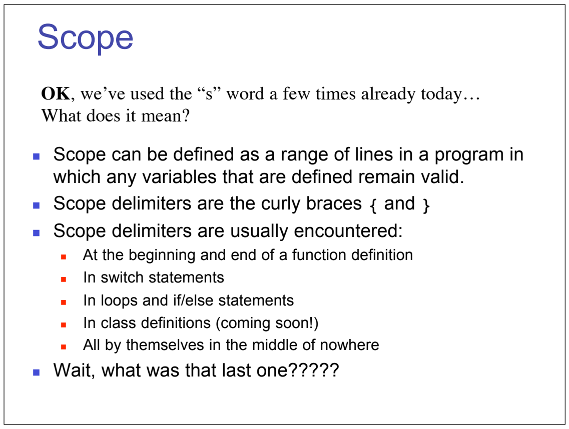
```
bool isBusy(const BIGDataType &arg1)
{
  if (arg1.busyField = 0)
    return true;
  return false;
}
```

- By adding the `const` keyword in front of the argument declaration, you tell the compiler that this parameter must not be changed by the function.
- Any attempts to change it will generate a compile time error.



Demonstration #3

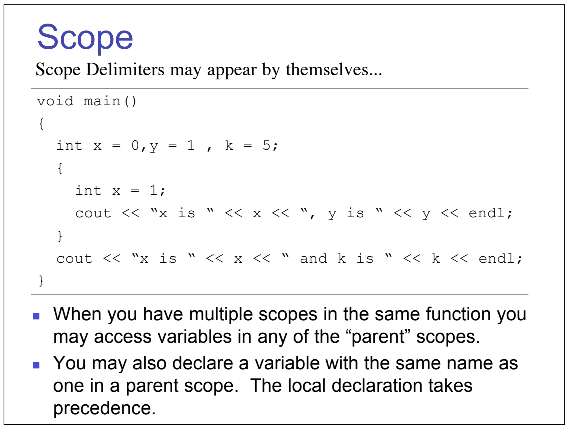
Pass by Reference
(with and without the const
keyword)



Scope

OK, we've used the "s" word a few times already today… What does it mean?

- Scope can be defined as a range of lines in a program in which any variables that are defined remain valid.
- Scope delimiters are the curly braces `{` and `}`
- Scope delimiters are usually encountered:
  - At the beginning and end of a function definition
  - In switch statements
  - In loops and if/else statements
  - In class definitions (coming soon!)
  - All by themselves in the middle of nowhere
- Wait, what was that last one?????

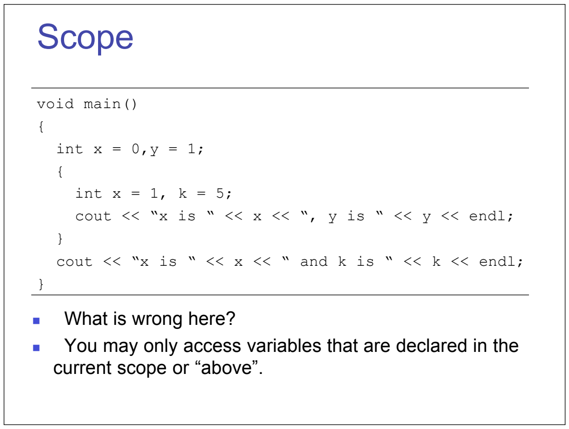


Scope

Scope Delimiters may appear by themselves...

```
void main()
{
  int x = 0,y = 1 , k = 5;
  {
    int x = 1;
    cout << “x is “ << x << “, y is “ << y << endl;
  }
  cout << “x is “ << x << “ and k is “ << k << endl;
}
```

- When you have multiple scopes in the same function you may access variables in any of the "parent" scopes.
- You may also declare a variable with the same name as one in a parent scope. The local declaration takes precedence.



Scope

```
void main()
{
  int x = 0,y = 1;
  {
    int x = 1, k = 5;
    cout << “x is “ << x << “, y is “ << y << endl;
  }
  cout << “x is “ << x << “ and k is “ << k << endl;
}
```

- What is wrong here?
- You may only access variables that are declared in the current scope or "above".

Scope

There is a global scope...

```
int globalX = 0;
int main()
{
  int x = 0,y = 1,k = 5;
  {
    int x = 1;
    cout << “x is ” << x << “, y is ” << y << endl;
    globalX = 10;
  }
  cout << “x is ” << x << “ and k is ” << k << endl;
  cout << “globalX is ” << globalX << endl;
}
```

- What happens here?

Demonstration #4

Miscellaneous Scope Issues

Function Declarations vs. Definitions

We’ve been somewhat lax about this...

```
long squareIt(long);        // Declaration (or prototype)
      .
      .
      .
long squareIt(long x)       // Definition
{
  return( x * x );
}
```

- Before a function may be called by any other function it must be either defined or declared.
- When a function is declared separately from its definition, this is called a forward declaration.
- Forward declarations need only to specify return type and parameter type. Parameter names are irrelevant.

Function Declarations and Header Files

- What happens when programs start getting really big?
  - We naturally want to separate all the functions we implement into logical groupings. These groupings are usually stored in their own files.
  - How, then, do we access a function from one file when we are working in another file?
- We move the function declarations into header files
- Then all we need to do is include the appropriate header file in whatever source file needs it.
- By convention, the function definitions go into a source file with a .cpp suffix, whereas function declarations go into a source file with a .h suffix.
- Consider the following example...

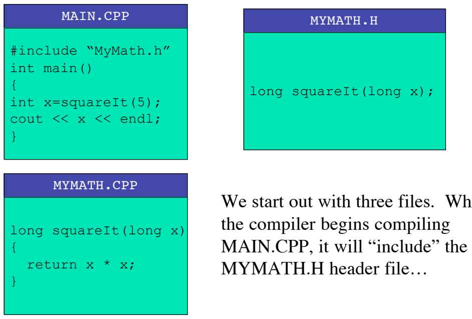
Function Declarations and Header Files

```
// mymath.h -- header file for math functions
long squareIt(long);
```

```
// mymath.cpp -- implementation of math functions
long squareIt(long x)
{
  return x * x;
}
```

```
// main.cpp
#include “mymath.h”
void main()
{
  cout >> “5 squared is ” >> squareIt(5) >> endl;
}
```

Function Declarations and Header Files



MAIN.CPP

```
#include “MyMath.h”
int main()
{
int x=squareIt(5);
cout << x << endl;
}
```

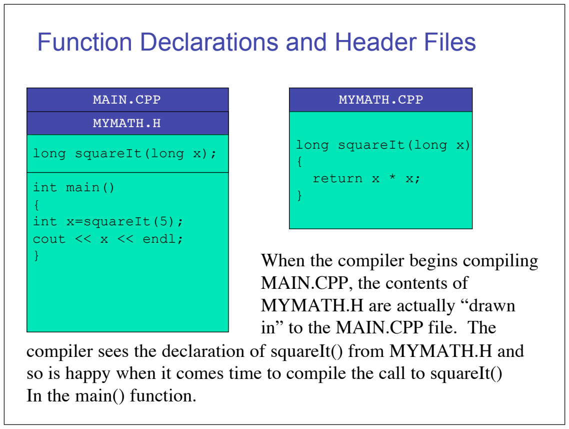
MYMATH.H

```
long squareIt(long x);
```

MYMATH.CPP

```
long squareIt(long x)
{
  return x * x;
}
```

We start out with three files. When the compiler begins compiling MAIN.CPP, it will “include” the MYMATH.H header file…



Function Declarations and Header Files

MAIN.CPP

MYMATH.H

```
long squareIt(long x);
```

```
int main()
{
int x=squareIt(5);
cout << x << endl;
}
```

MYMATH.CPP

```
long squareIt(long x)
{
  return x * x;
}
```

When the compiler begins compiling MAIN.CPP, the contents of MYMATH.H are actually “drawn in” to the MAIN.CPP file. The compiler sees the declaration of squareIt() from MYMATH.H and so is happy when it comes time to compile the call to squareIt() In the main() function.