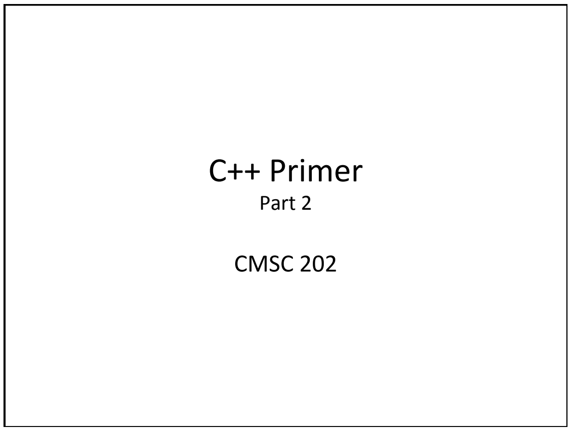
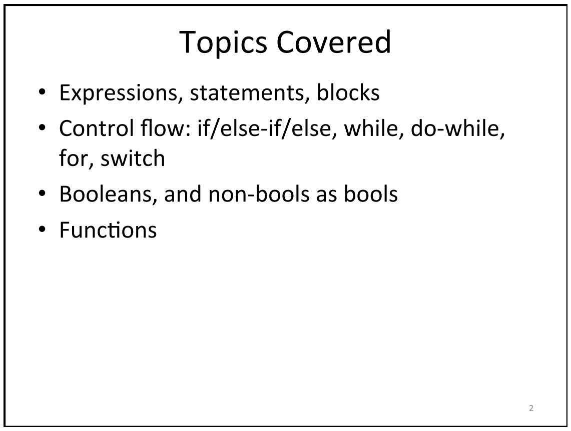


C++ Primer

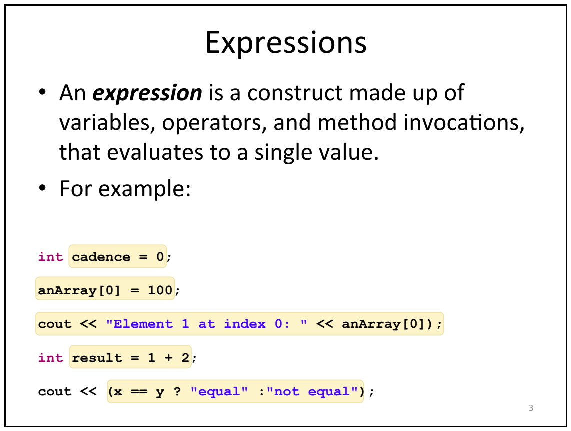
Part 2

CMSC 202



Topics Covered

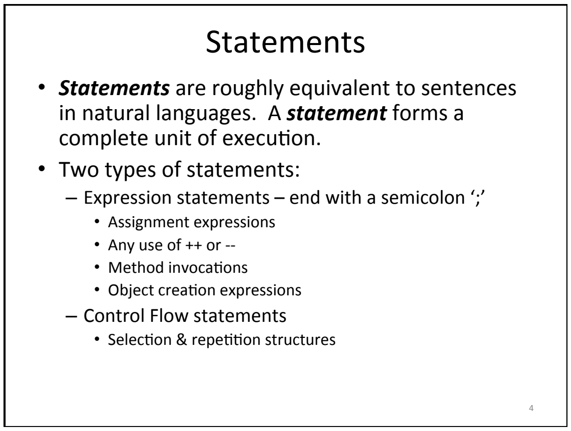
- Expressions, statements, blocks
- Control flow: if/else-if/else, while, do-while, for, switch
- Booleans, and non-bools as bools
- Functions



Expressions

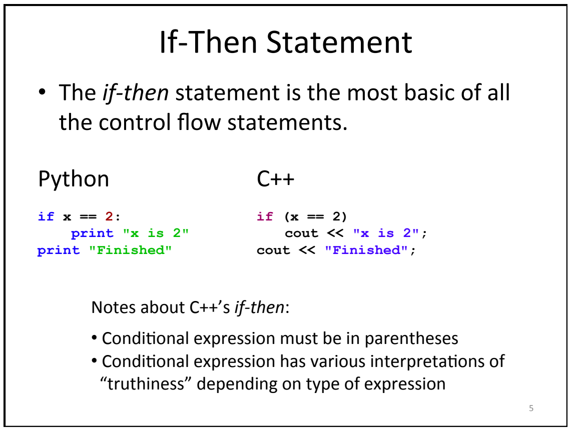
- An ***expression*** is a construct made up of variables, operators, and method invocations, that evaluates to a single value.
- For example:

```
int cadence = 0;
anArray[0] = 100;
cout << "Element 1 at index 0: " << anArray[0]);
int result = 1 + 2;
cout << (x == y ? "equal" :"not equal");
```



Statements

- ***Statements*** are roughly equivalent to sentences in natural languages. A ***statement*** forms a complete unit of execution.
- Two types of statements:
  - Expression statements – end with a semicolon ‘;’
    - Assignment expressions
    - Any use of ++ or --
    - Method invocations
    - Object creation expressions
  - Control Flow statements
    - Selection & repetition structures



If-Then Statement

- The *if-then* statement is the most basic of all the control flow statements.

Python

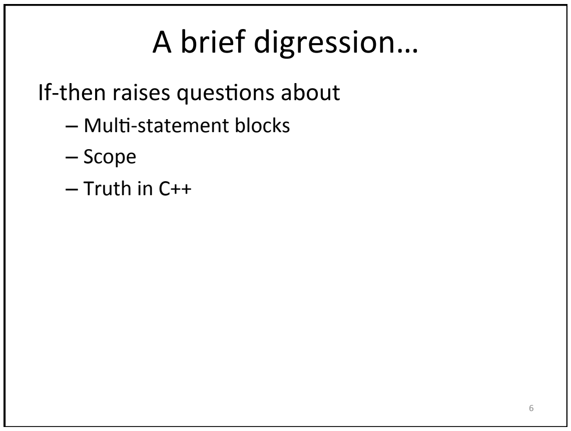
```
if x == 2:
    print "x is 2"
print "Finished"
```

C++

```
if (x == 2)
    cout << "x is 2";
cout << "Finished";
```

Notes about C++’s *if-then*:

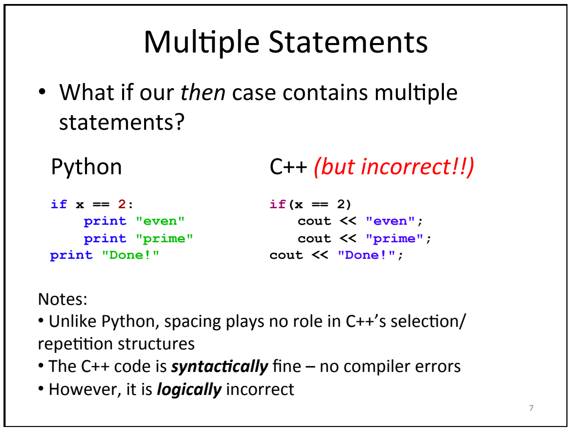
- Conditional expression must be in parentheses
- Conditional expression has various interpretations of “truthiness” depending on type of expression



A brief digression…

If-then raises questions about

- Multi-statement blocks
- Scope
- Truth in C++



Multiple Statements

- What if our *then* case contains multiple statements?

Python

```
if x == 2:
    print "even"
    print "prime"
print "Done!"
```

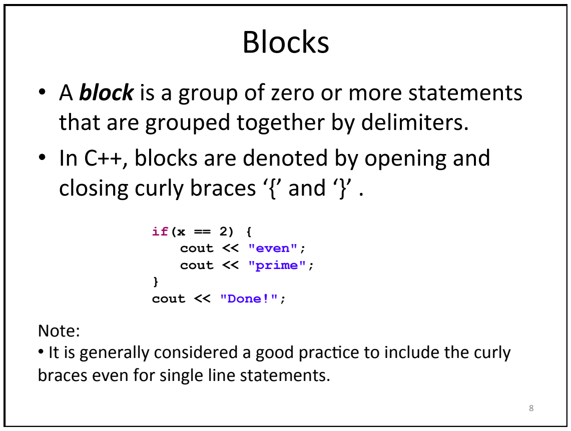
C++ *(but incorrect!!)*

```
if(x == 2)
    cout << "even";
    cout << "prime";
cout << "Done!";
```

Notes:

- Unlike Python, spacing plays no role in C++'s selection/ repetition structures
- The C++ code is ***syntactically*** fine – no compiler errors
- However, it is ***logically*** incorrect

7



Blocks

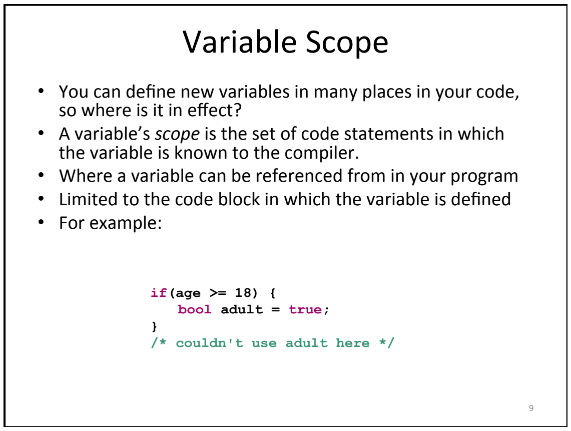
- A ***block*** is a group of zero or more statements that are grouped together by delimiters.
- In C++, blocks are denoted by opening and closing curly braces '{' and '}' .

```
if(x == 2) {
    cout << "even";
    cout << "prime";
}
cout << "Done!";
```

Note:

- It is generally considered a good practice to include the curly braces even for single line statements.

8

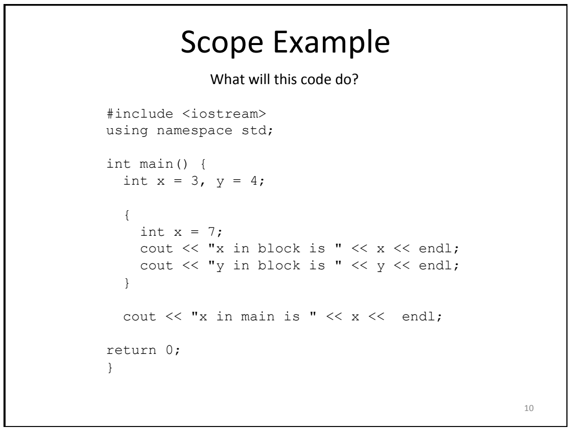


Variable Scope

- You can define new variables in many places in your code, so where is it in effect?
- A variable's *scope* is the set of code statements in which the variable is known to the compiler.
- Where a variable can be referenced from in your program
- Limited to the code block in which the variable is defined
- For example:

```
if(age >= 18) {
    bool adult = true;
}
/* couldn't use adult here */
```

9

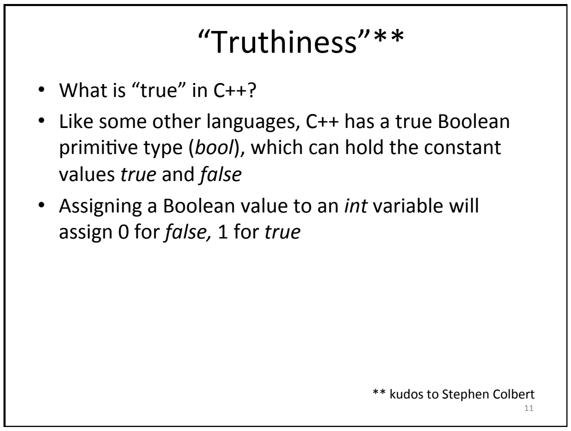


Scope Example

What will this code do?

```
#include <iostream>
using namespace std;

int main() {
  int x = 3, y = 4;

  {
    int x = 7;
    cout << "x in block is " << x << endl;
    cout << "y in block is " << y << endl;
  }

  cout << "x in main is " << x <<  endl;

return 0;
}
```

10

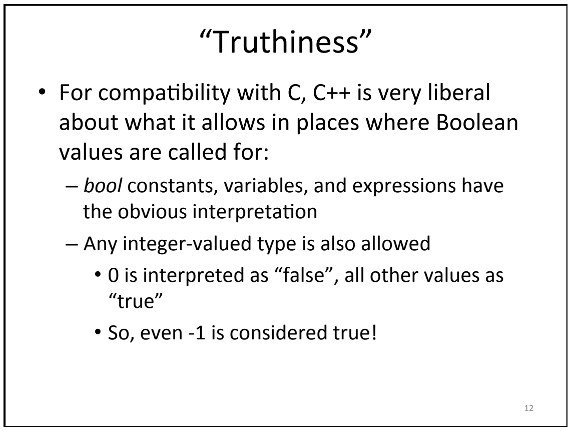


"Truthiness"**

- What is "true" in C++?
- Like some other languages, C++ has a true Boolean primitive type (*bool*), which can hold the constant values *true* and *false*
- Assigning a Boolean value to an *int* variable will assign 0 for *false,* 1 for *true*

** kudos to Stephen Colbert

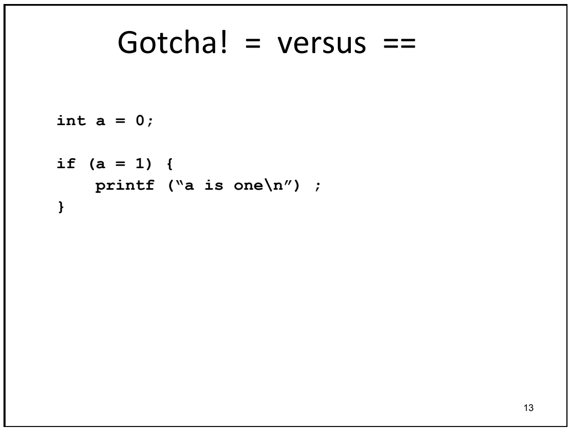
11



"Truthiness"

- For compatibility with C, C++ is very liberal about what it allows in places where Boolean values are called for:
  - *bool* constants, variables, and expressions have the obvious interpretation
  - Any integer-valued type is also allowed
    - 0 is interpreted as "false", all other values as "true"
    - So, even -1 is considered true!

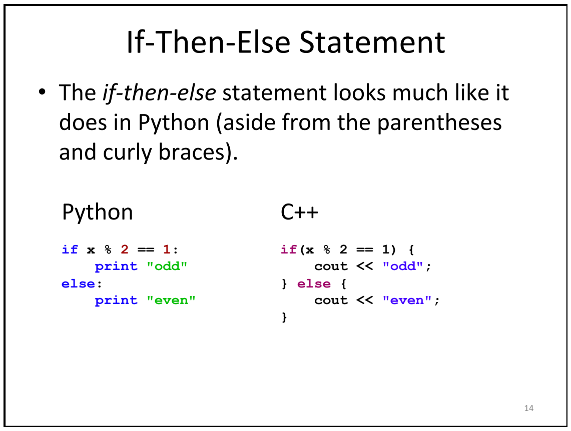
12



Gotcha! = versus ==

```
int a = 0;

if (a = 1) {
     printf (“a is one\n”) ;
}
```

13



If-Then-Else Statement

- The *if-then-else* statement looks much like it does in Python (aside from the parentheses and curly braces).

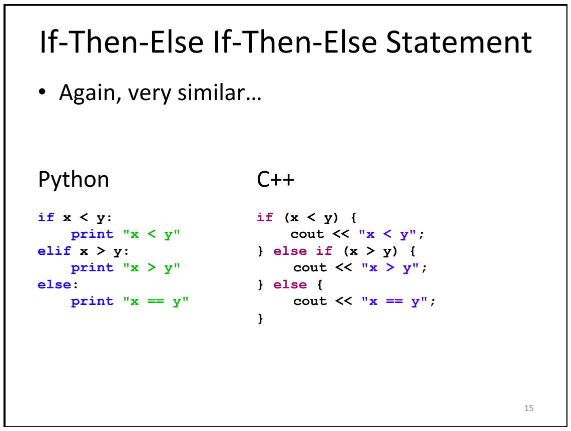
Python

```
if x % 2 == 1:
    print "odd"
else:
    print "even"
```

C++

```
if(x % 2 == 1) {
    cout << "odd";
} else {
    cout << "even";
}
```

14



If-Then-Else If-Then-Else Statement

- Again, very similar…

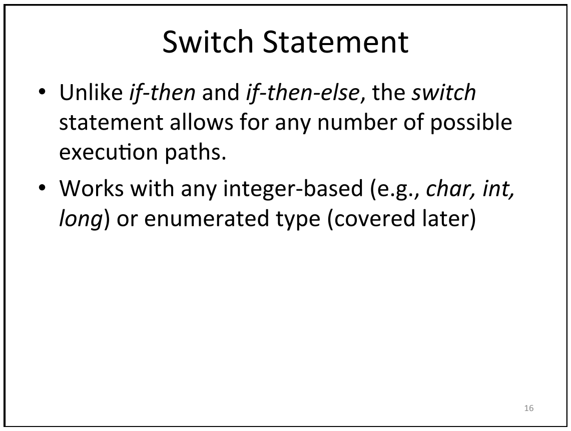
Python

```
if x < y:
    print "x < y"
elif x > y:
    print "x > y"
else:
    print "x == y"
```

C++

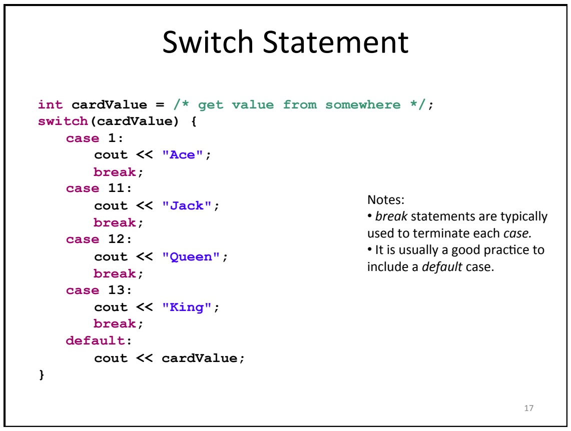
```
if (x < y) {
    cout << "x < y";
} else if (x > y) {
    cout << "x > y";
} else {
    cout << "x == y";
}
```

15



Switch Statement

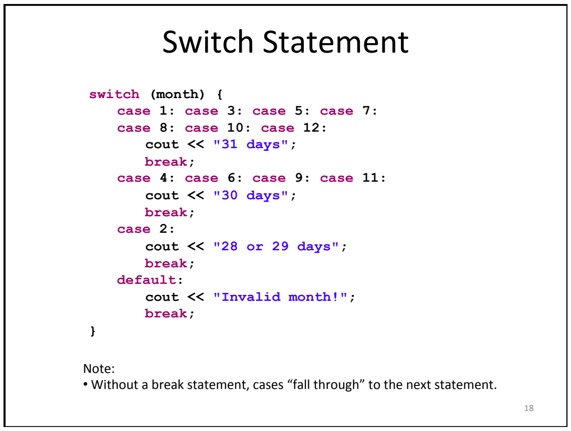
- Unlike *if-then* and *if-then-else*, the *switch* statement allows for any number of possible execution paths.
- Works with any integer-based (e.g., *char, int, long*) or enumerated type (covered later)



Switch Statement

```
int cardValue = /* get value from somewhere */;
switch(cardValue) {
    case 1:
        cout << "Ace";
        break;
    case 11:
        cout << "Jack";
        break;
    case 12:
        cout << "Queen";
        break;
    case 13:
        cout << "King";
        break;
    default:
        cout << cardValue;
}
```

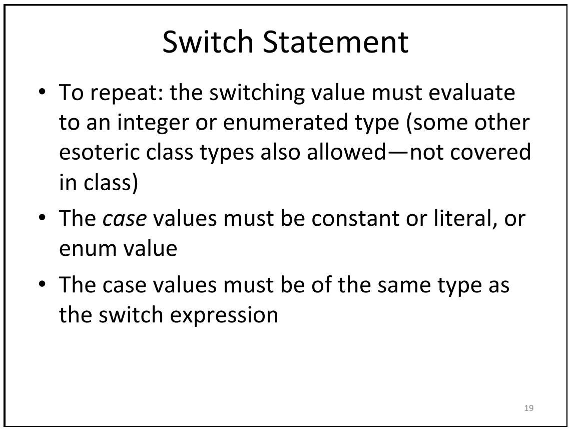
Notes:
- *break* statements are typically used to terminate each *case.*
- It is usually a good practice to include a *default* case.



Switch Statement

```
switch (month) {
    case 1: case 3: case 5: case 7:
    case 8: case 10: case 12:
        cout << "31 days";
        break;
    case 4: case 6: case 9: case 11:
        cout << "30 days";
        break;
    case 2:
        cout << "28 or 29 days";
        break;
    default:
        cout << "Invalid month!";
        break;
}
```

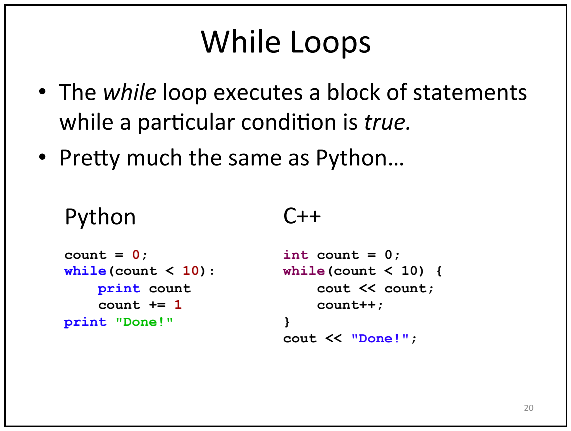
Note:
- Without a break statement, cases “fall through” to the next statement.



Switch Statement

- To repeat: the switching value must evaluate to an integer or enumerated type (some other esoteric class types also allowed—not covered in class)
- The *case* values must be constant or literal, or enum value
- The case values must be of the same type as the switch expression

19



While Loops

- The *while* loop executes a block of statements while a particular condition is *true.*
- Pretty much the same as Python…

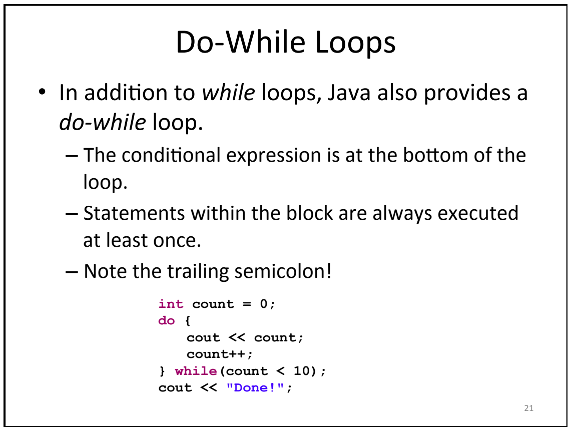
Python

```
count = 0;
while(count < 10):
    print count
    count += 1
print "Done!"
```

C++

```
int count = 0;
while(count < 10) {
    cout << count;
    count++;
}
cout << "Done!";
```

20

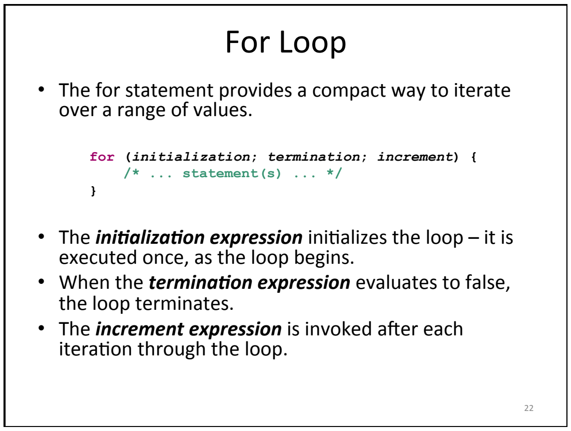


Do-While Loops

- In addition to *while* loops, Java also provides a *do-while* loop.
  - The conditional expression is at the bottom of the loop.
  - Statements within the block are always executed at least once.
  - Note the trailing semicolon!

```
int count = 0;
do {
    cout << count;
    count++;
} while(count < 10);
cout << "Done!";
```

21

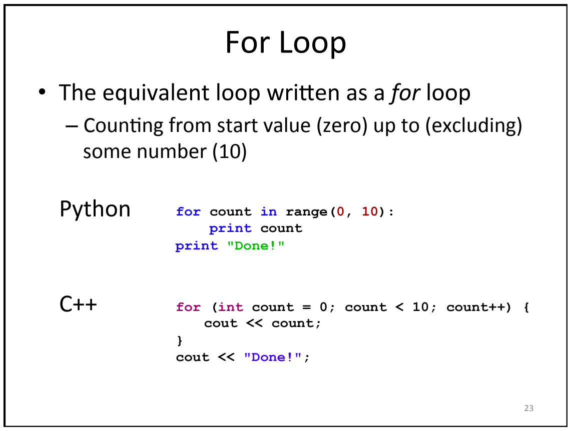


For Loop

- The for statement provides a compact way to iterate over a range of values.

```
for (initialization; termination; increment) {
    /* ... statement(s) ... */
}
```

- The ***initialization expression*** initializes the loop – it is executed once, as the loop begins.
- When the ***termination expression*** evaluates to false, the loop terminates.
- The ***increment expression*** is invoked after each iteration through the loop.



For Loop

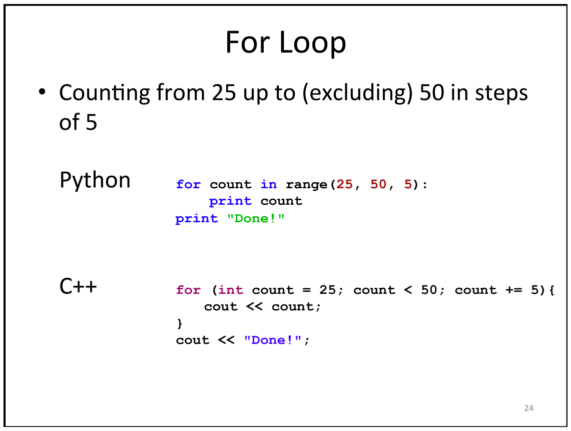
- The equivalent loop written as a *for* loop
  - Counting from start value (zero) up to (excluding) some number (10)

Python

```
for count in range(0, 10):
    print count
print "Done!"
```

C++

```
for (int count = 0; count < 10; count++) {
    cout << count;
}
cout << "Done!";
```



For Loop

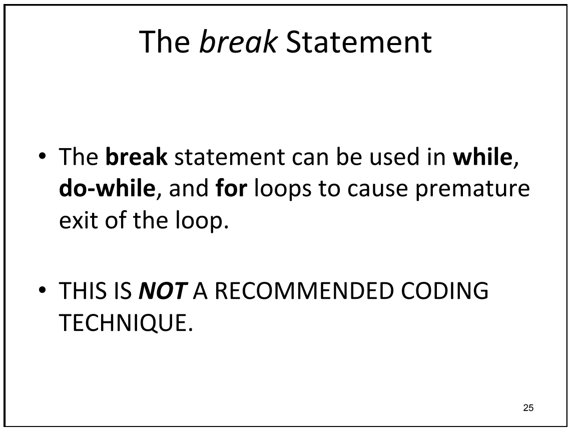
- Counting from 25 up to (excluding) 50 in steps of 5

Python

```
for count in range(25, 50, 5):
    print count
print "Done!"
```

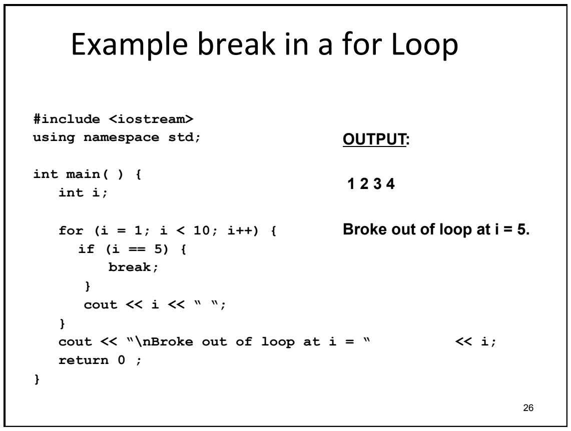
C++

```
for (int count = 25; count < 50; count += 5){
    cout << count;
}
cout << "Done!";
```



The *break* Statement

- The **break** statement can be used in **while**, **do-while**, and **for** loops to cause premature exit of the loop.
- THIS IS ***NOT*** A RECOMMENDED CODING TECHNIQUE.

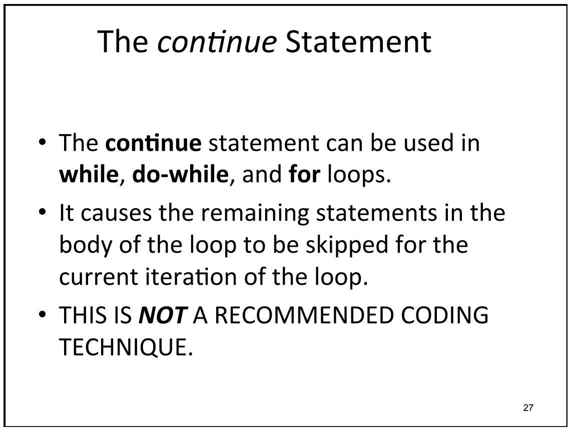


Example break in a for Loop

```
#include <iostream>
using namespace std;

int main( ) {
   int i;

   for (i = 1; i < 10; i++) {
     if (i == 5) {
        break;
     }
     cout << i << " ";
   }
   cout << "\nBroke out of loop at i = "          << i;
   return 0 ;
}
```

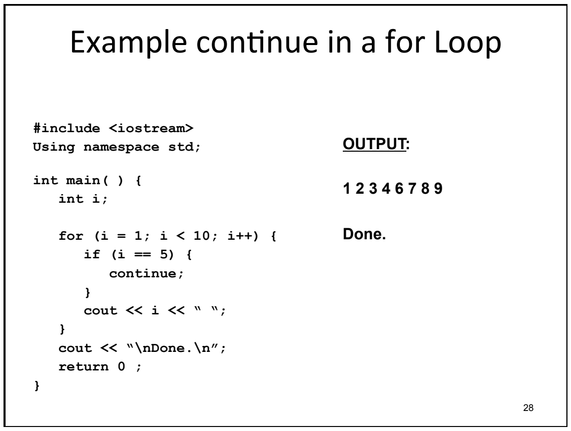
OUTPUT:

```
1 2 3 4

Broke out of loop at i = 5.
```



The *continue* Statement

- The **continue** statement can be used in **while**, **do-while**, and **for** loops.
- It causes the remaining statements in the body of the loop to be skipped for the current iteration of the loop.
- THIS IS ***NOT*** A RECOMMENDED CODING TECHNIQUE.



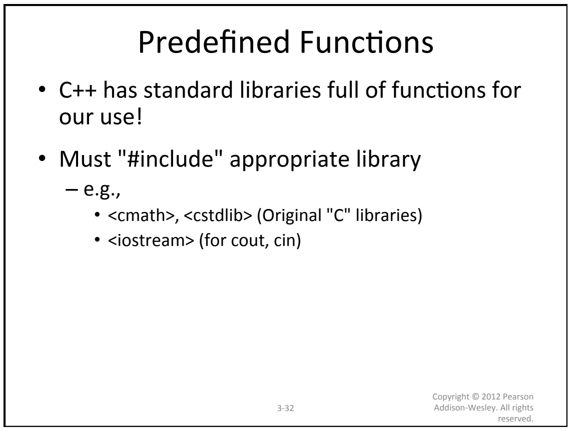
Example continue in a for Loop

```
#include <iostream>
Using namespace std;

int main( ) {
   int i;

   for (i = 1; i < 10; i++) {
      if (i == 5) {
         continue;
      }
      cout << i << “ “;
   }
   cout << “\nDone.\n”;
   return 0 ;
}
```

__OUTPUT__:

```
1 2 3 4 6 7 8 9

Done.
```

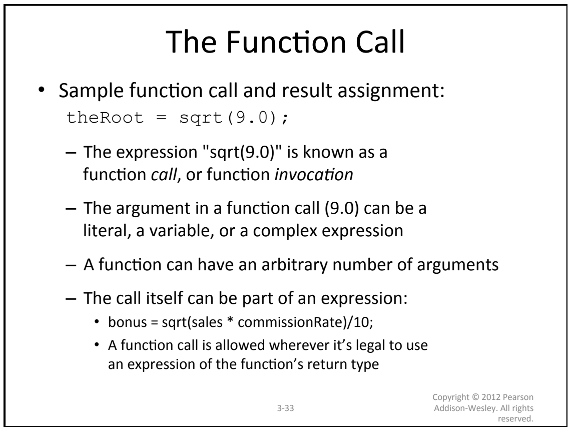
28



Predefined Functions

- C++ has standard libraries full of functions for our use!
- Must "#include" appropriate library
  – e.g.,
    - <cmath>, <cstdlib> (Original "C" libraries)
    - <iostream> (for cout, cin)

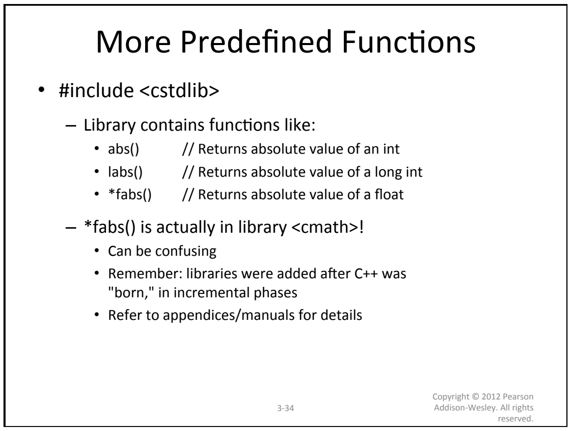
3-32



The Function Call

- Sample function call and result assignment:
  `theRoot = sqrt(9.0);`
  – The expression "sqrt(9.0)" is known as a function *call*, or function *invocation*
  – The argument in a function call (9.0) can be a literal, a variable, or a complex expression
  – A function can have an arbitrary number of arguments
  – The call itself can be part of an expression:
    - bonus = sqrt(sales * commissionRate)/10;
    - A function call is allowed wherever it’s legal to use an expression of the function’s return type

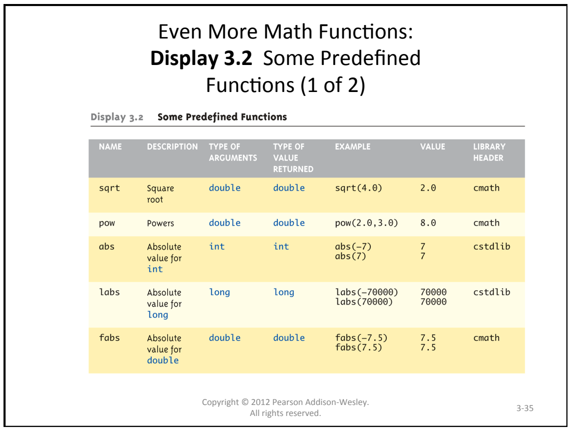
3-33



More Predefined Functions

- #include <cstdlib>
  - Library contains functions like:
    - abs() // Returns absolute value of an int
    - labs() // Returns absolute value of a long int
    - *fabs() // Returns absolute value of a float
  - *fabs() is actually in library <cmath>!
    - Can be confusing
    - Remember: libraries were added after C++ was "born," in incremental phases
    - Refer to appendices/manuals for details

3-34



Even More Math Functions: **Display 3.2** Some Predefined Functions (1 of 2)

Display 3.2 **Some Predefined Functions**

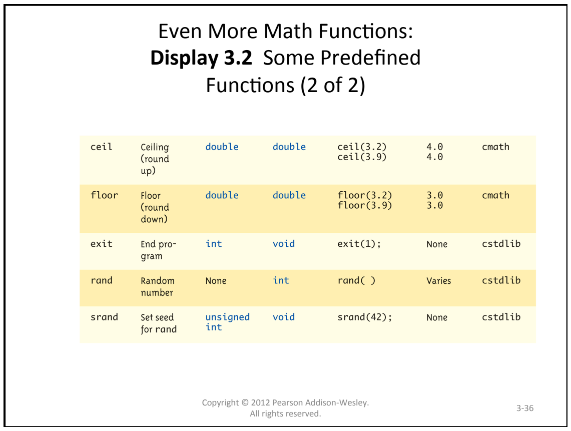
NAME	DESCRIPTION	TYPE OF ARGUMENTS	TYPE OF VALUE RETURNED	EXAMPLE	VALUE	LIBRARY HEADER
sqrt	Square root	double	double	sqrt(4.0)	2.0	cmath
pow	Powers	double	double	pow(2.0,3.0)	8.0	cmath

abs	Absolute value for int	int	int	abs(−7)
				abs(7)
					7
					7
						cstdlib

labs	Absolute value for long	long	long	labs(−70000)
				labs(70000)
					70000
					70000
						cstdlib

fabs	Absolute value for double	double	double	fabs(−7.5)
				fabs(7.5)
					7.5
					7.5
						cmath

3-35



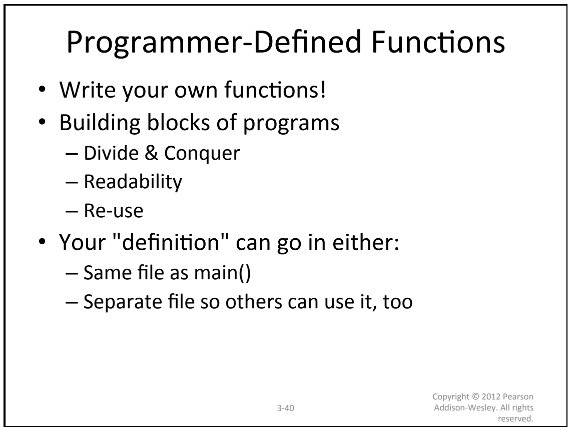
Even More Math Functions: **Display 3.2** Some Predefined Functions (2 of 2)

ceil	Ceiling (round up)	double	double	ceil(3.2)
				ceil(3.9)
					4.0
					4.0
						cmath

floor	Floor (round down)	double	double	floor(3.2)
				floor(3.9)
					3.0
					3.0
						cmath

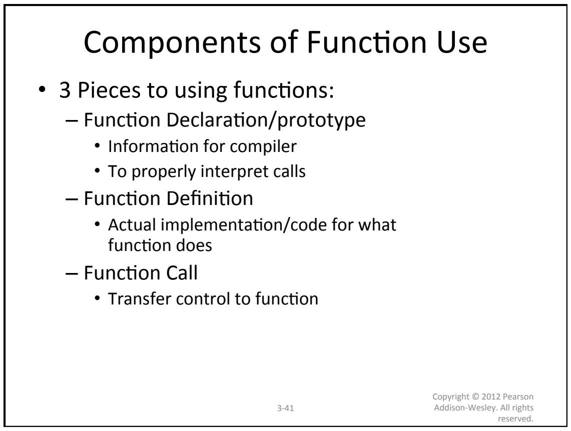
exit	End program	int	void	exit(1);	None	cstdlib
rand	Random number	None	int	rand( )	Varies	cstdlib
srand	Set seed for rand	unsigned int	void	srand(42);	None	cstdlib

3-36



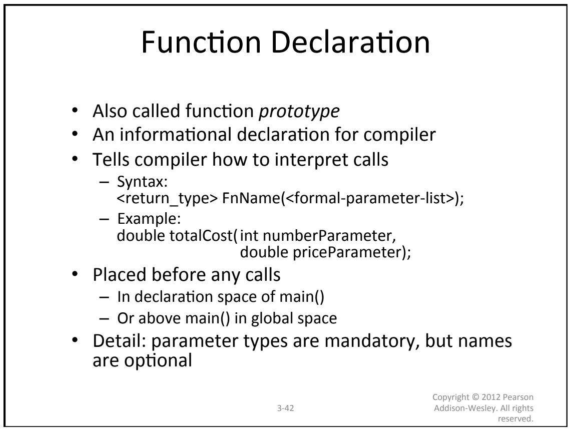
Programmer-Defined Functions

- Write your own functions!
- Building blocks of programs
  – Divide & Conquer
  – Readability
  – Re-use
- Your "definition" can go in either:
  – Same file as main()
  – Separate file so others can use it, too



Components of Function Use

- 3 Pieces to using functions:
  – Function Declaration/prototype
    - Information for compiler
    - To properly interpret calls
  – Function Definition
    - Actual implementation/code for what function does
  – Function Call
    - Transfer control to function



Function Declaration

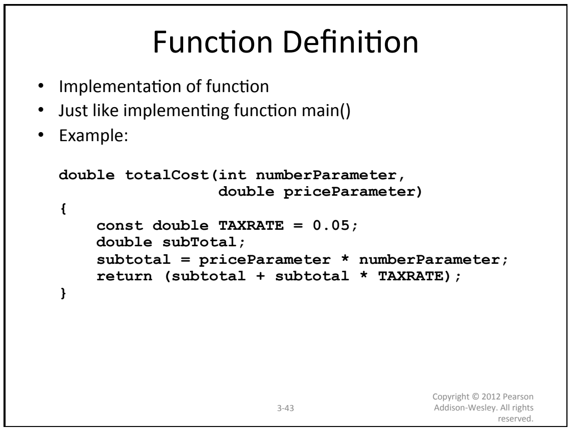
- Also called function *prototype*
- An informational declaration for compiler
- Tells compiler how to interpret calls
  – Syntax:

```
<return_type> FnName(<formal-parameter-list>);
```

  – Example:

```
double totalCost(int numberParameter,
                 double priceParameter);
```

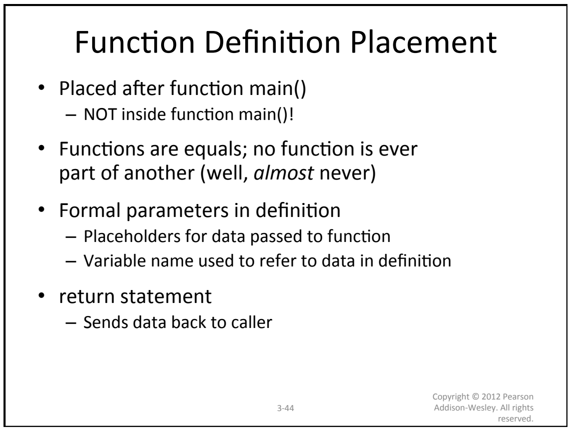
- Placed before any calls
  – In declaration space of main()
  – Or above main() in global space
- Detail: parameter types are mandatory, but names are optional



Function Definition

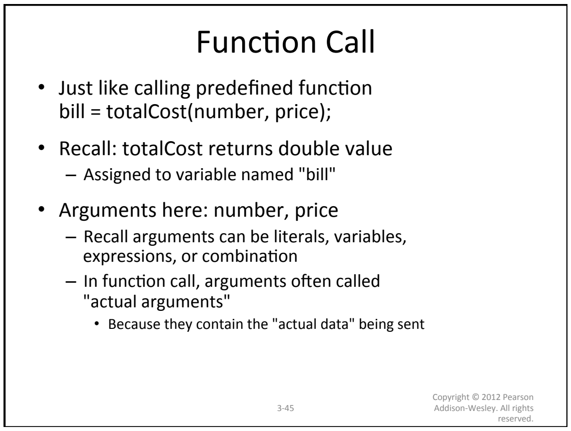
- Implementation of function
- Just like implementing function main()
- Example:

```
double totalCost(int numberParameter,
                 double priceParameter)
{
    const double TAXRATE = 0.05;
    double subTotal;
    subtotal = priceParameter * numberParameter;
    return (subtotal + subtotal * TAXRATE);
}
```



Function Definition Placement

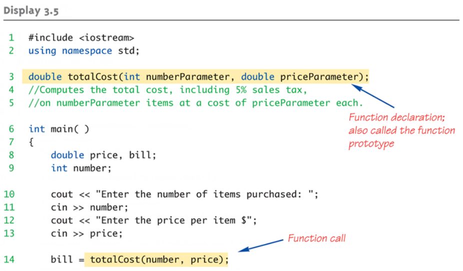
- Placed after function main()
  - NOT inside function main()!
- Functions are equals; no function is ever part of another (well, *almost* never)
- Formal parameters in definition
  - Placeholders for data passed to function
  - Variable name used to refer to data in definition
- return statement
  - Sends data back to caller



Function Call

- Just like calling predefined function
  bill = totalCost(number, price);
- Recall: totalCost returns double value
  - Assigned to variable named "bill"
- Arguments here: number, price
  - Recall arguments can be literals, variables, expressions, or combination
  - In function call, arguments often called "actual arguments"
    - Because they contain the "actual data" being sent

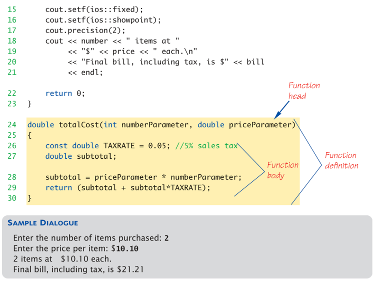
Function Example: **Display 3.5** A Function to Calculate Total Cost (1 of 2)



Display 3.5

```
#include <iostream>
using namespace std;

double totalCost(int numberParameter, double priceParameter);
//Computes the total cost, including 5% sales tax,
//on numberParameter items at a cost of priceParameter each.

int main( )
{
    double price, bill;
    int number;

    cout << "Enter the number of items purchased: ";
    cin >> number;
    cout << "Enter the price per item $";
    cin >> price;

    bill = totalCost(number, price);
```

Function Example: **Display 3.5** A Function to Calculate Total Cost (1 of 2)



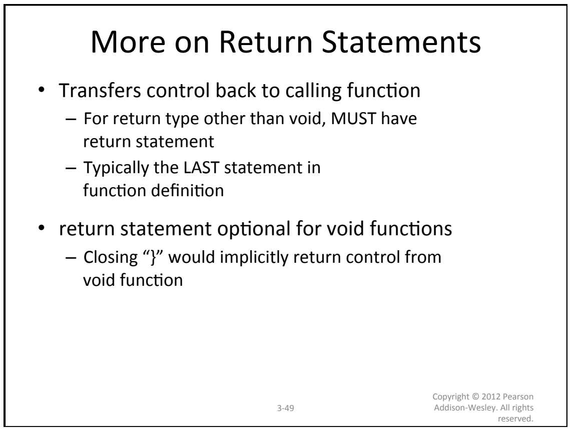
```
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << number << " items at "
         << "$" << price << " each.\n"
         << "Final bill, including tax, is $" << bill
         << endl;

    return 0;
}

double totalCost(int numberParameter, double priceParameter)
{
    const double TAXRATE = 0.05; //5% sales tax
    double subtotal;

    subtotal = priceParameter * numberParameter;
    return (subtotal + subtotal*TAXRATE);
}
```

Sample Dialogue

```
Enter the number of items purchased: 2
Enter the price per item: $10.10
2 items at   $10.10 each.
Final bill, including tax, is $21.21
```

Declaring Void Functions

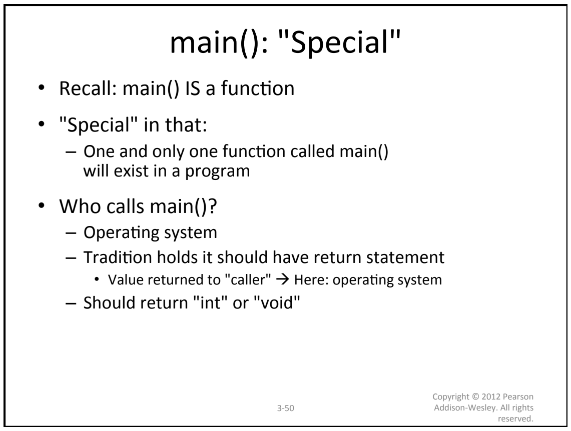
- “void” functions are called for side effects; they don’t return any usable value
- Declaration is similar to functions returning a value, but return type specified as "void"
- Example:
  - Function declaration/prototype:
    void showResults(double fDegrees,
                                double cDegrees);
    - Return-type is "void"
    - Nothing is returned



More on Return Statements

- Transfers control back to calling function
  - For return type other than void, MUST have return statement
  - Typically the LAST statement in function definition
- return statement optional for void functions
  - Closing "}" would implicitly return control from void function

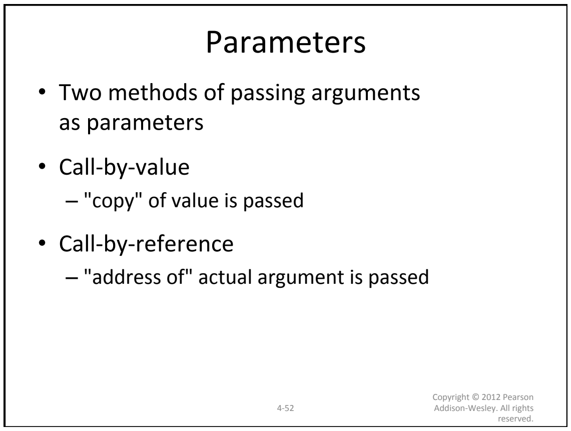
3-49



main(): "Special"

- Recall: main() IS a function
- "Special" in that:
  - One and only one function called main() will exist in a program
- Who calls main()?
  - Operating system
  - Tradition holds it should have return statement
    - Value returned to "caller" → Here: operating system
  - Should return "int" or "void"

3-50



Parameters

- Two methods of passing arguments as parameters
- Call-by-value
  - "copy" of value is passed
- Call-by-reference
  - "address of" actual argument is passed

4-52

Call-by-Value Parameters

- Copy of actual argument passed
- Considered "local variable" inside function
- If modified, only "local copy" changes
  – Function has no access to "actual argument" from caller
- This is the default method
  – Used in all examples thus far

Call-by-Value Example: **Display 4.1** Formal Parameter Used as a Local Variable (1 of 3)

Display 4.1 **Formal Parameter Used as a Local Variable**

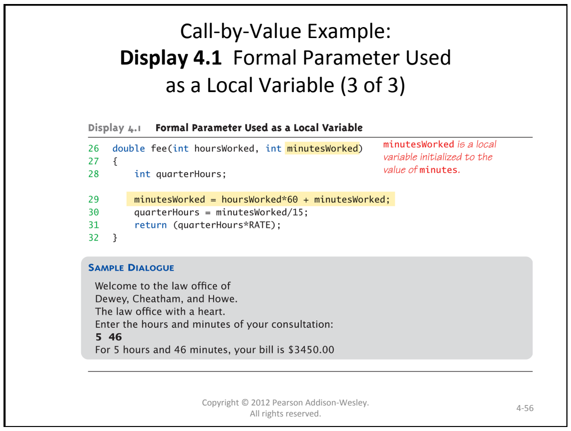
```
//Law office billing program.
#include <iostream>
using namespace std;

const double RATE = 150.00; //Dollars per quarter hour.

double fee(int hoursWorked, int minutesWorked);
//Returns the charges for hoursWorked hours and
//minutesWorked minutes of legal services.

int main( )
{
    int hours, minutes;
    double bill;
```

Call-by-Value Example: **Display 4.1** Formal Parameter Used as a Local Variable (2 of 3)

```
    cout << "Welcome to the law office of\n"
         << "Dewey, Cheatham, and Howe.\n"
         << "The law office with a heart.\n"
         << "Enter the hours and minutes"
         << " of your consultation:\n";
    cin >> hours >> minutes;

    bill = fee(hours, minutes);

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "For " << hours << " hours and " << minutes
         << " minutes, your bill is $" << bill << endl;

    return 0;
}
```

The value of `minutes` *is not changed by the call to* `fee`*.*

(continued)



Call-by-Value Example: **Display 4.1** Formal Parameter Used as a Local Variable (3 of 3)

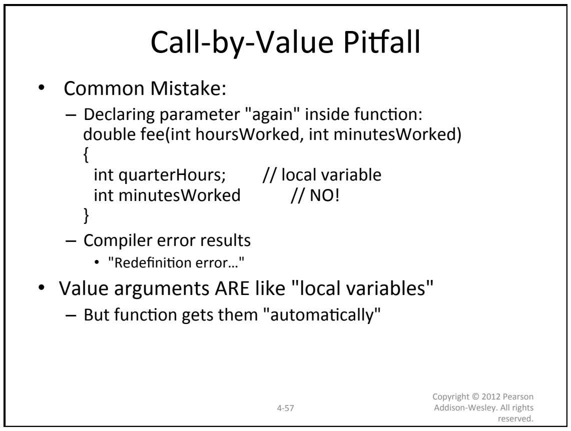
Display 4.1 **Formal Parameter Used as a Local Variable**

```
double fee(int hoursWorked, int minutesWorked)
{
    int quarterHours;

    minutesWorked = hoursWorked*60 + minutesWorked;
    quarterHours = minutesWorked/15;
    return (quarterHours*RATE);
}
```

minutesWorked is a local variable initialized to the value of minutes.

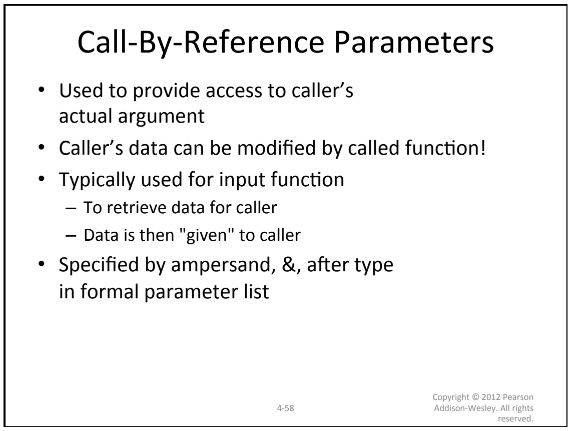
Sample Dialogue

```
Welcome to the law office of
Dewey, Cheatham, and Howe.
The law office with a heart.
Enter the hours and minutes of your consultation:
5 46
For 5 hours and 46 minutes, your bill is $3450.00
```



Call-by-Value Pitfall

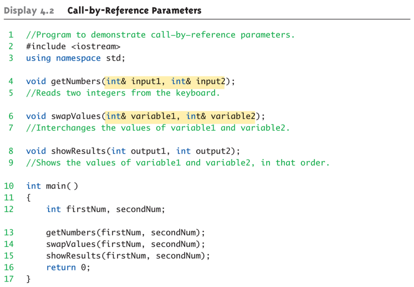
- Common Mistake:
  – Declaring parameter "again" inside function:
    double fee(int hoursWorked, int minutesWorked)
    {
      int quarterHours;   // local variable
      int minutesWorked   // NO!
    }
  – Compiler error results
    - "Redefinition error…"
- Value arguments ARE like "local variables"
  – But function gets them "automatically"



Call-By-Reference Parameters

- Used to provide access to caller's actual argument
- Caller's data can be modified by called function!
- Typically used for input function
  – To retrieve data for caller
  – Data is then "given" to caller
- Specified by ampersand, &, after type in formal parameter list

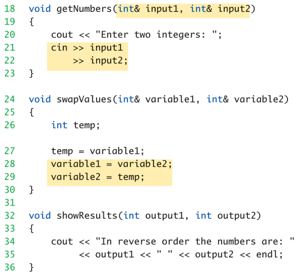
Call-By-Reference Example: **Display 4.1** Call-by-Reference Parameters (1 of 3)



Display 4.2 **Call-by-Reference Parameters**

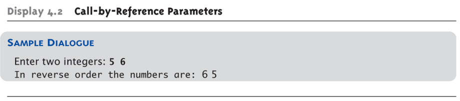
```
//Program to demonstrate call-by-reference parameters.
#include <iostream>
using namespace std;

void getNumbers(int& input1, int& input2);
//Reads two integers from the keyboard.

void swapValues(int& variable1, int& variable2);
//Interchanges the values of variable1 and variable2.

void showResults(int output1, int output2);
//Shows the values of variable1 and variable2, in that order.

int main( )
{
    int firstNum, secondNum;

    getNumbers(firstNum, secondNum);
    swapValues(firstNum, secondNum);
    showResults(firstNum, secondNum);
    return 0;
}
```

Call-By-Reference Example: **Display 4.1** Call-by-Reference Parameters (2 of 3)



```
void getNumbers(int& input1, int& input2)
{
    cout << "Enter two integers: ";
    cin >> input1
        >> input2;
}

void swapValues(int& variable1, int& variable2)
{
    int temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}

void showResults(int output1, int output2)
{
    cout << "In reverse order the numbers are: "
         << output1 << " " << output2 << endl;
}
```

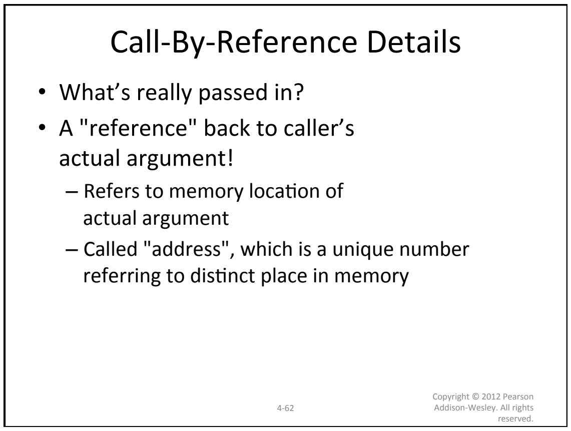
Call-By-Reference Example: **Display 4.1** Call-by-Reference Parameters (3 of 3)



Display 4.2 **Call-by-Reference Parameters**

SAMPLE DIALOGUE

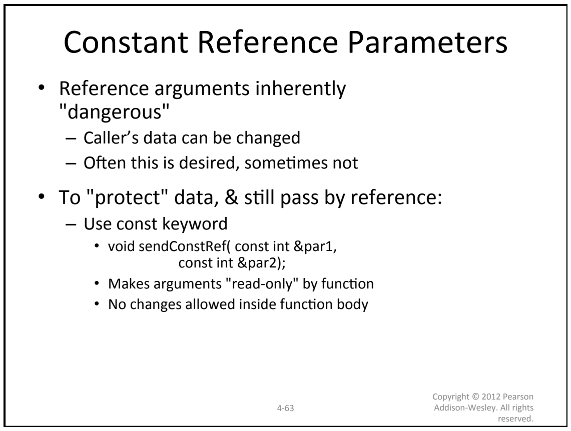
```
Enter two integers: 5 6
In reverse order the numbers are: 6 5
```



Call-By-Reference Details

- What's really passed in?
- A "reference" back to caller's actual argument!
  - Refers to memory location of actual argument
  - Called "address", which is a unique number referring to distinct place in memory

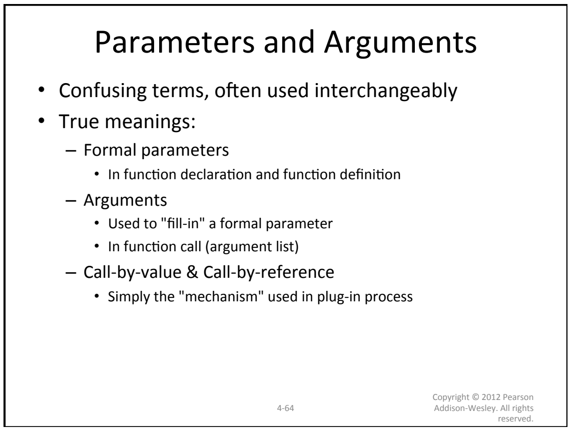
4-62



Constant Reference Parameters

- Reference arguments inherently "dangerous"
  - Caller's data can be changed
  - Often this is desired, sometimes not
- To "protect" data, & still pass by reference:
  - Use const keyword
    - void sendConstRef(const int &par1,
                   const int &par2);
    - Makes arguments "read-only" by function
    - No changes allowed inside function body

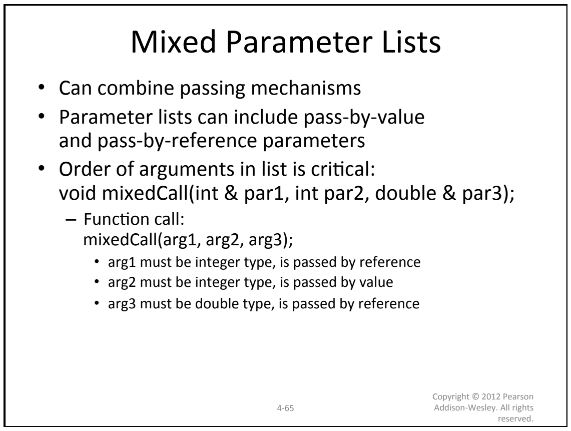
4-63



Parameters and Arguments

- Confusing terms, often used interchangeably
- True meanings:
  - Formal parameters
    - In function declaration and function definition
  - Arguments
    - Used to "fill-in" a formal parameter
    - In function call (argument list)
  - Call-by-value & Call-by-reference
    - Simply the "mechanism" used in plug-in process

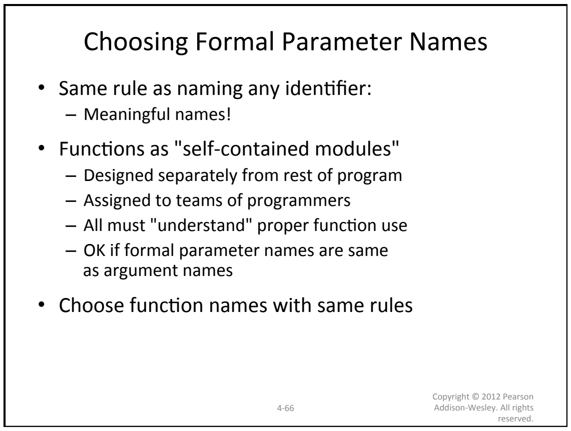
4-64



Mixed Parameter Lists

- Can combine passing mechanisms
- Parameter lists can include pass-by-value and pass-by-reference parameters
- Order of arguments in list is critical:
  void mixedCall(int & par1, int par2, double & par3);
  - Function call:
    mixedCall(arg1, arg2, arg3);
    - arg1 must be integer type, is passed by reference
    - arg2 must be integer type, is passed by value
    - arg3 must be double type, is passed by reference

4-65



Choosing Formal Parameter Names

- Same rule as naming any identifier:
  - Meaningful names!
- Functions as "self-contained modules"
  - Designed separately from rest of program
  - Assigned to teams of programmers
  - All must "understand" proper function use
  - OK if formal parameter names are same as argument names
- Choose function names with same rules

4-66