UCLA

CS 31:
Introduction To Computer Science I

Howard A. Stahl

UCLA

Agenda

- Switch Statements
- For Loops
- Functions
- Parameter Passing Mechanisms
- Overloading
- Type-Casting

Selective Control Flow in C++

- Programs often choose between different instructions in a variety of situations
  - sometimes, code must be skipped because it does not apply in the current situation
  - other times, one of several code blocks must be chosen to be executed based on the current situtation

The `if-else if-else` Statement

- Multiple Action

```
if ( x < y )
{
  x++;
}
else if ( x > y )
{
  y++;
}
else {
  x++; y++;
}
```

The `if-else if-else` Statement

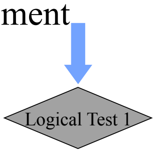
- Multiple Action

```
if ( x < y )
{
  x++;
}
else if ( x > y )
{
  y++;
}
else {
  x++; y++;
}
```

The `if-else if-else` Statement

- Multiple Action

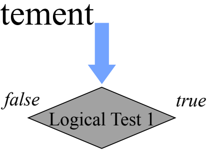
```
if ( x < y )
{
  x++;
}
else if ( x > y )
{
  y++;
}
else {
  x++; y++;
}
```



The `if-else if-else` Statement

- Multiple Action

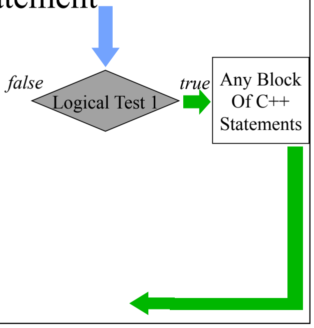
```
if ( x < y )
{
  x++;
}
else if ( x > y )
{
  y++;
}
else {
  x++; y++;
}
```



The `if-else if-else` Statement

- Multiple Action

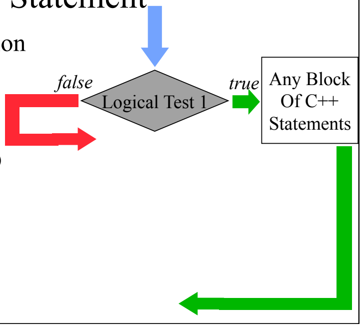
```
if ( x < y )
{
  x++;
}
else if ( x > y )
{
  y++;
}
else {
  x++; y++;
}
```

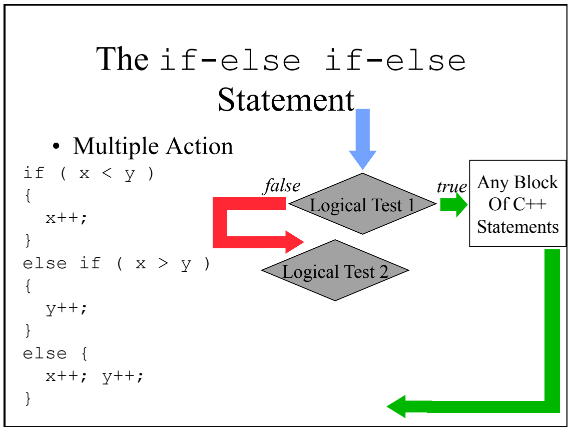


The `if-else if-else` Statement

- Multiple Action

```
if ( x < y )
{
  x++;
}
else if ( x > y )
{
  y++;
}
else {
  x++; y++;
}
```

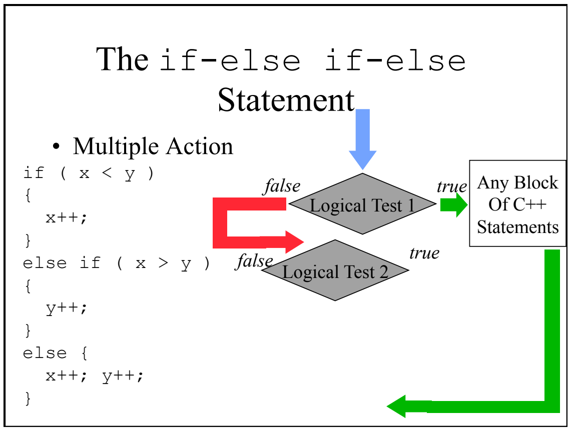




The `if-else if-else` Statement

- Multiple Action

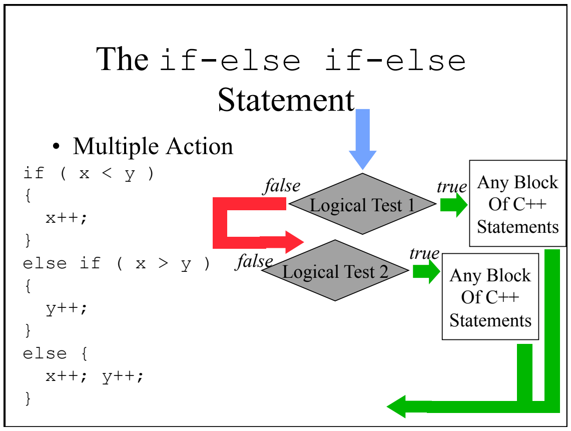
```
if ( x < y )
{
  x++;
}
else if ( x > y )
{
  y++;
}
else {
  x++; y++;
}
```



The `if-else if-else` Statement

- Multiple Action

```
if ( x < y )
{
  x++;
}
else if ( x > y )
{
  y++;
}
else {
  x++; y++;
}
```



The `if-else if-else` Statement

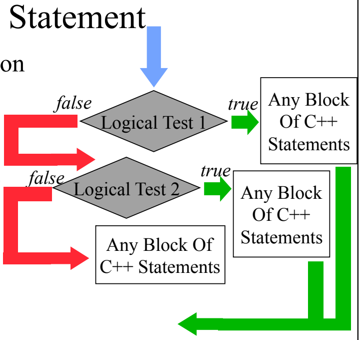
- Multiple Action

```
if ( x < y )
{
  x++;
}
else if ( x > y )
{
  y++;
}
else {
  x++; y++;
}
```

The `if-else if-else` Statement

- Multiple Action

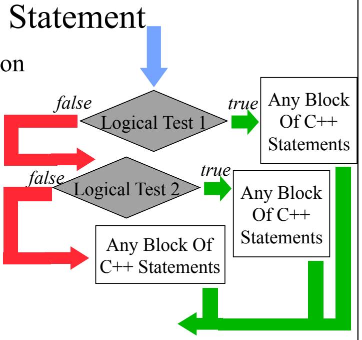
```
if ( x < y )
{
  x++;
}
else if ( x > y )
{
  y++;
}
else {
  x++; y++;
}
```


false
Logical Test 1
true
Any Block Of C++ Statements
false
Logical Test 2
true
Any Block Of C++ Statements
Any Block Of C++ Statements

The `if-else if-else` Statement

- Multiple Action

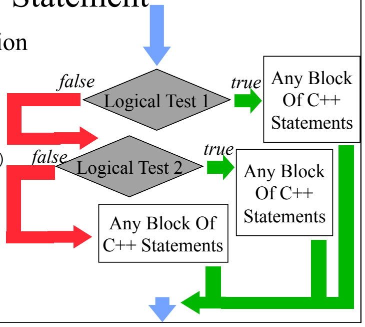
```
if ( x < y )
{
  x++;
}
else if ( x > y )
{
  y++;
}
else {
  x++; y++;
}
```


false
Logical Test 1
true
Any Block Of C++ Statements
false
Logical Test 2
true
Any Block Of C++ Statements
Any Block Of C++ Statements

The `if-else if-else` Statement

- Multiple Action

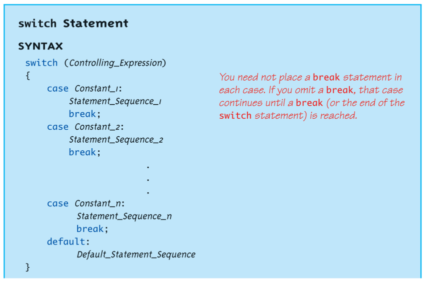
```
if ( x < y )
{
  x++;
}
else if ( x > y )
{
  y++;
}
else {
  x++; y++;
}
```


false
Logical Test 1
true
Any Block Of C++ Statements
false
Logical Test 2
true
Any Block Of C++ Statements
Any Block Of C++ Statements

The `if-else if-else` Statement

- Any Number Of `else-if` Alternatives Is Allowed
- The `else` Clause Is Completely Optional

switch Statement Syntax



switch Statement

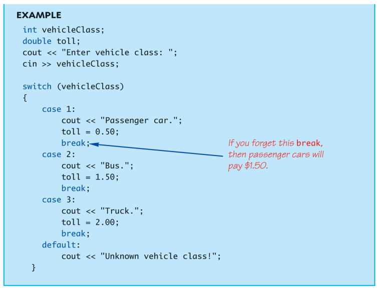
SYNTAX

```
switch (Controlling_Expression)
{
    case Constant_1:
        Statement_Sequence_1
        break;
    case Constant_2:
        Statement_Sequence_2
        break;
                        .
                        .
                        .
    case Constant_n:
          Statement_Sequence_n
          break;
    default:
          Default_Statement_Sequence
}
```

You need not place a **break** *statement in each case. If you omit a* **break**, *that case continues until a* **break** *(or the end of the* **switch** *statement) is reached.*

The controlling expression must be integral! This includes char.

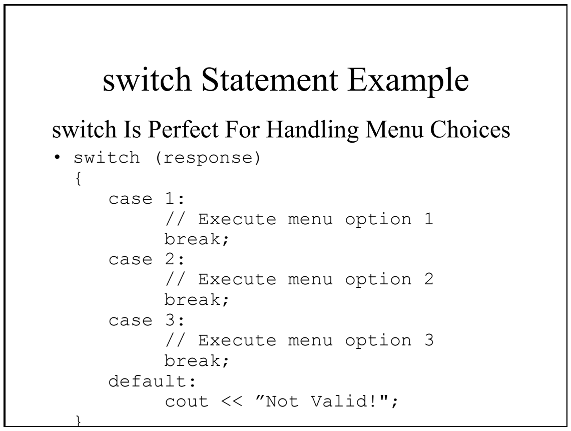
switch Statement Example



EXAMPLE

```
int vehicleClass;
double toll;
cout << "Enter vehicle class: ";
cin >> vehicleClass;

switch (vehicleClass)
{
    case 1:
        cout << "Passenger car.";
        toll = 0.50;
        break;
    case 2:
        cout << "Bus.";
        toll = 1.50;
        break;
    case 3:
        cout << "Truck.";
        toll = 2.00;
        break;
    default:
        cout << "Unknown vehicle class!";
 }
```

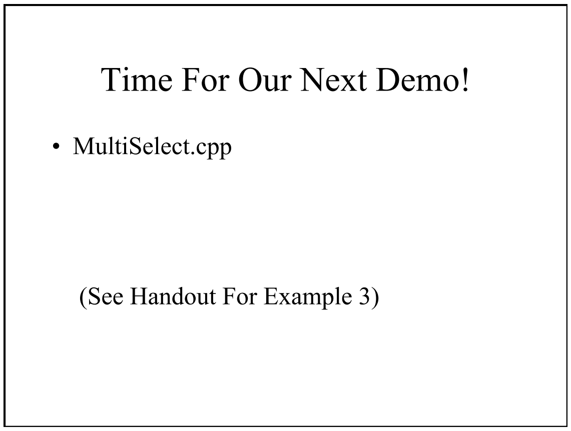
If you forget this **break**, *then passenger cars will pay $1.50.*



switch Statement Example

switch Is Perfect For Handling Menu Choices

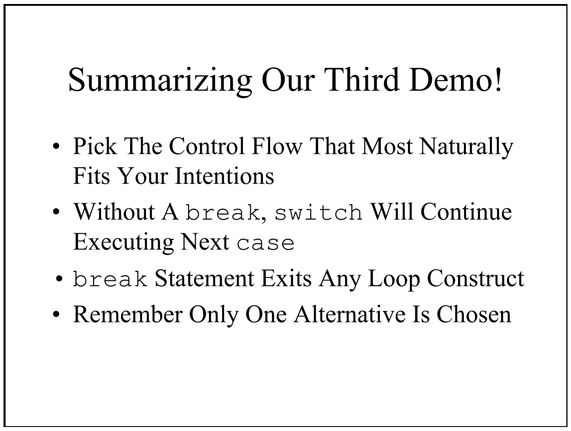
```
• switch (response)
  {
     case 1:
          // Execute menu option 1
          break;
     case 2:
          // Execute menu option 2
          break;
     case 3:
          // Execute menu option 3
          break;
     default:
          cout << ”Not Valid!";
  }
```



Time For Our Next Demo!

- MultiSelect.cpp

(See Handout For Example 3)



Summarizing Our Third Demo!

- Pick The Control Flow That Most Naturally Fits Your Intentions
- Without A `break`, `switch` Will Continue Executing Next `case`
- `break` Statement Exits Any Loop Construct
- Remember Only One Alternative Is Chosen

Repetitive Control Flow in C++

- Programs often must repeat different instructions in a variety of situations
  - sometimes, code must be repeated a determinate number of times
  - other times, code must be repeated an indeterminate number of times

The `while` Statement

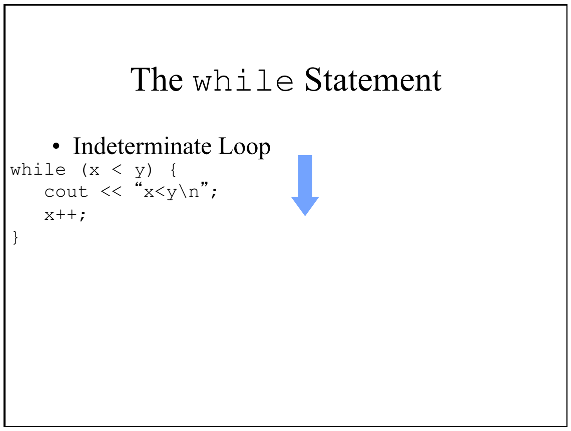
- Indeterminate Loop
  - Repeat While A Condition Is True

```
while ( logical-expression ) {
   ...block of statements...
}
```

The `while` Statement

- Indeterminate Loop

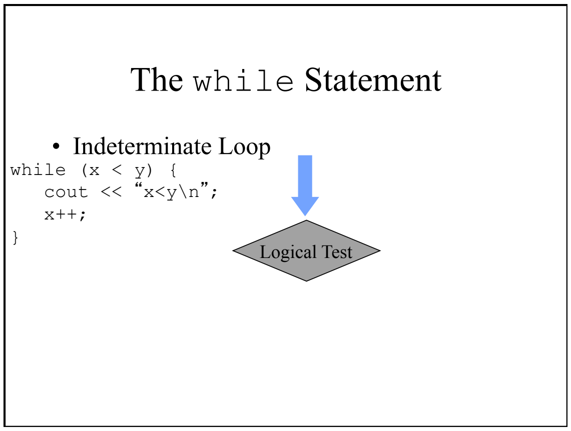
```
while (x < y) {
   cout << “x<y\n”;
   x++;
}
```



The `while` Statement

- Indeterminate Loop

```
while (x < y) {
   cout << “x<y\n”;
   x++;
}
```

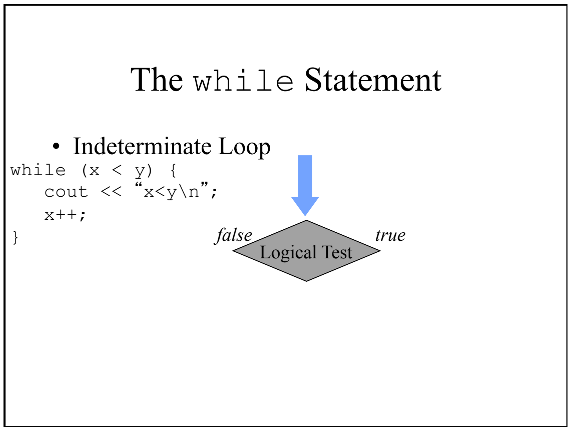


The `while` Statement

- Indeterminate Loop

```
while (x < y) {
   cout << “x<y\n”;
   x++;
}
```

Logical Test

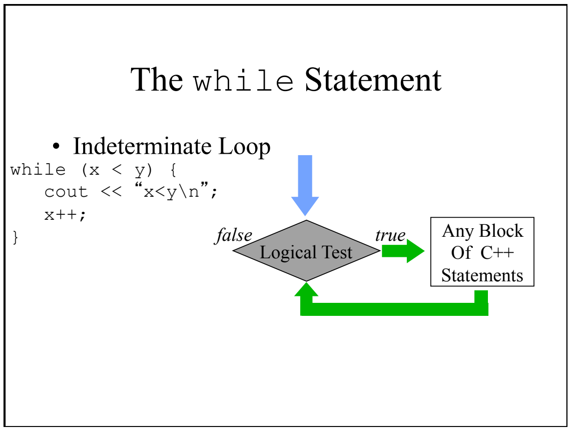


The `while` Statement

- Indeterminate Loop

```
while (x < y) {
   cout << “x<y\n”;
   x++;
}
```

false Logical Test *true*



The while Statement

- Indeterminate Loop

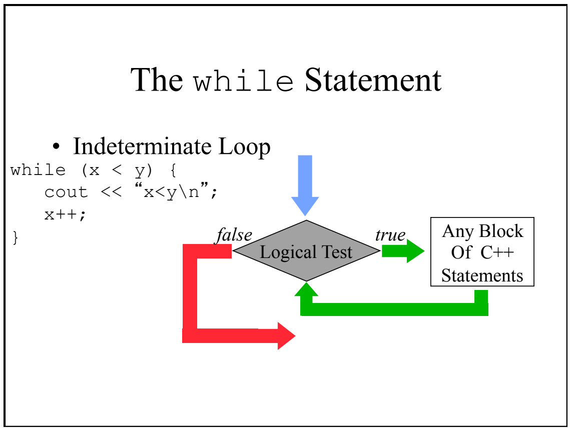
```
while (x < y) {
   cout << “x<y\n”;
   x++;
}
```

false

Logical Test

true

Any Block Of C++ Statements



The while Statement

- Indeterminate Loop

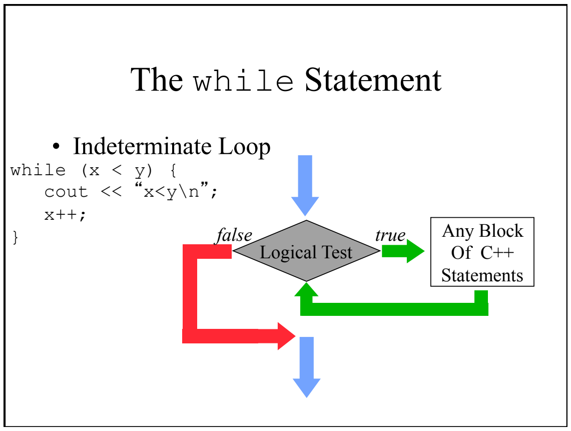
```
while (x < y) {
   cout << “x<y\n”;
   x++;
}
```

false

Logical Test

true

Any Block Of C++ Statements



The while Statement

- Indeterminate Loop

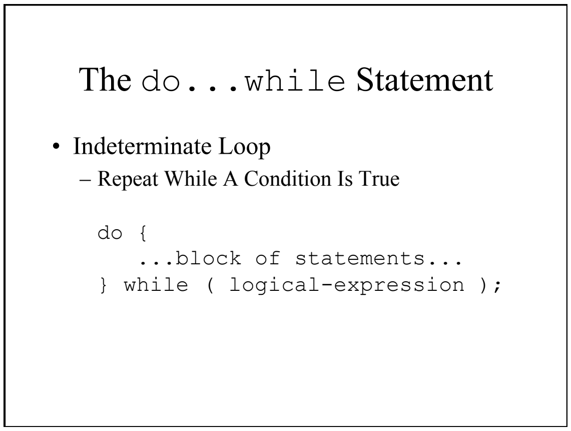
```
while (x < y) {
   cout << “x<y\n”;
   x++;
}
```

false

Logical Test

true

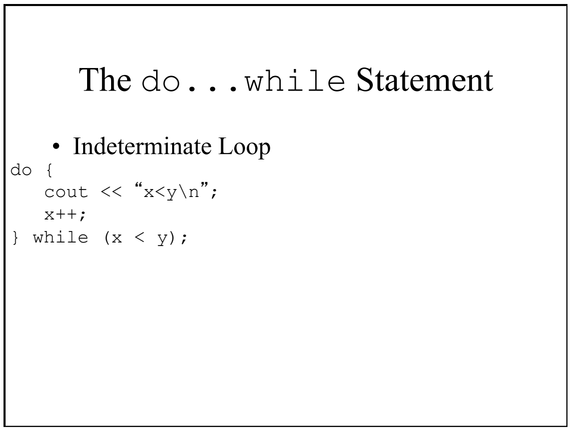
Any Block Of C++ Statements



The do...while Statement

- Indeterminate Loop
  - Repeat While A Condition Is True

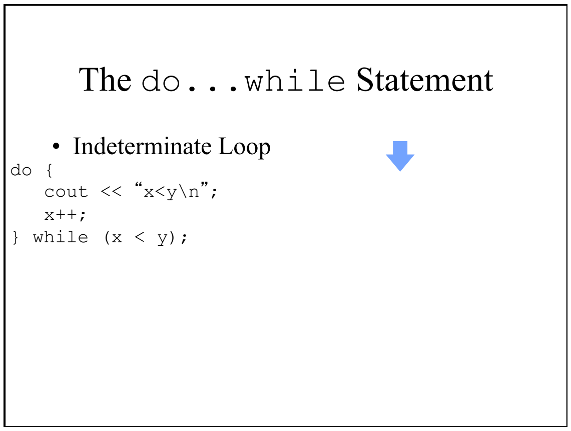
```
do {
   ...block of statements...
} while ( logical-expression );
```



The do...while Statement

- Indeterminate Loop

```
do {
   cout << “x<y\n”;
   x++;
} while (x < y);
```



The do...while Statement

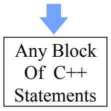
- Indeterminate Loop

```
do {
   cout << “x<y\n”;
   x++;
} while (x < y);
```

The do...while Statement

- Indeterminate Loop

```
do {
   cout << “x<y\n”;
   x++;
} while (x < y);
```

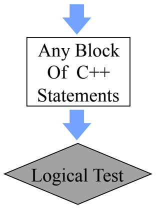


Any Block Of C++ Statements

The do...while Statement

- Indeterminate Loop

```
do {
   cout << “x<y\n”;
   x++;
} while (x < y);
```



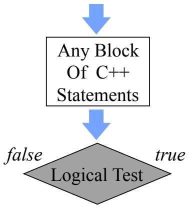
Any Block Of C++ Statements

Logical Test

The do...while Statement

- Indeterminate Loop

```
do {
   cout << “x<y\n”;
   x++;
} while (x < y);
```



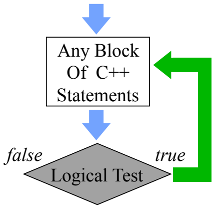
Any Block Of C++ Statements

false Logical Test *true*

The do...while Statement

- Indeterminate Loop

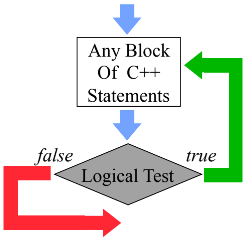
```
do {
   cout << “x<y\n”;
   x++;
} while (x < y);
```



The do...while Statement

- Indeterminate Loop

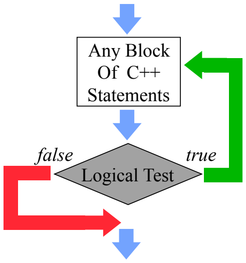
```
do {
   cout << “x<y\n”;
   x++;
} while (x < y);
```



The do...while Statement

- Indeterminate Loop

```
do {
   cout << “x<y\n”;
   x++;
} while (x < y);
```



Time For Our Next Demo!

- Loops.cpp

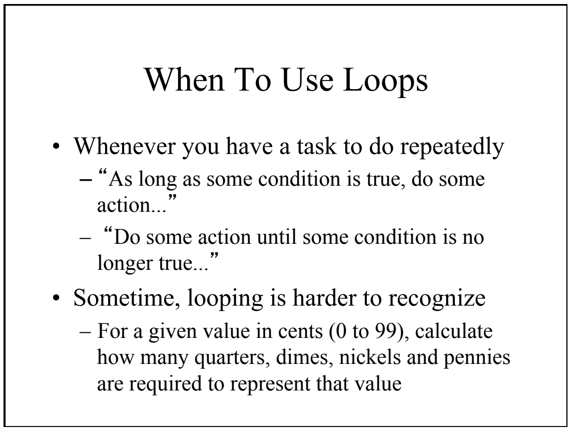
(See Handout For Example 4)

Summarizing Our Fourth Demo!

- Typically, one of the loop forms fits your problem better than the other
- However, any loop written in one form can be re-written in the other

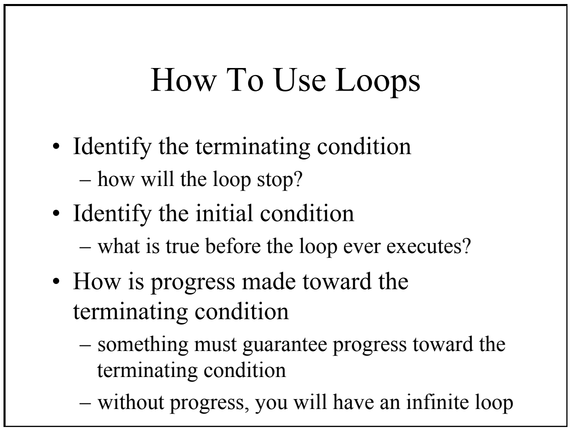
`while` versus `do...while`

- `while` loop may never execute
- `do...while` loop will always execute atleast once



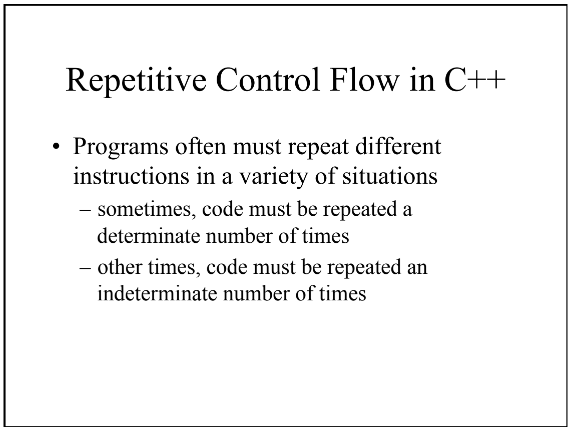
When To Use Loops

- Whenever you have a task to do repeatedly
  - "As long as some condition is true, do some action..."
  - "Do some action until some condition is no longer true..."
- Sometime, looping is harder to recognize
  - For a given value in cents (0 to 99), calculate how many quarters, dimes, nickels and pennies are required to represent that value



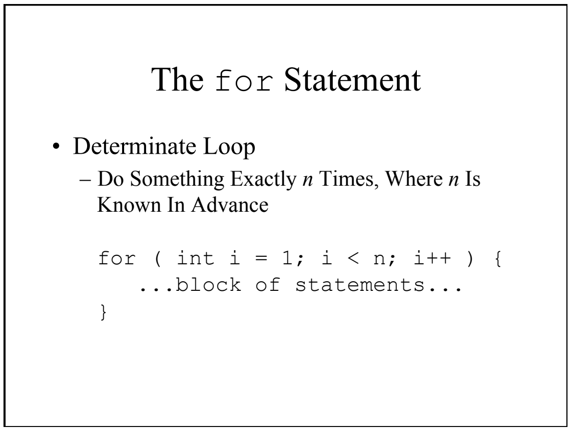
How To Use Loops

- Identify the terminating condition
  - how will the loop stop?
- Identify the initial condition
  - what is true before the loop ever executes?
- How is progress made toward the terminating condition
  - something must guarantee progress toward the terminating condition
  - without progress, you will have an infinite loop



Repetitive Control Flow in C++

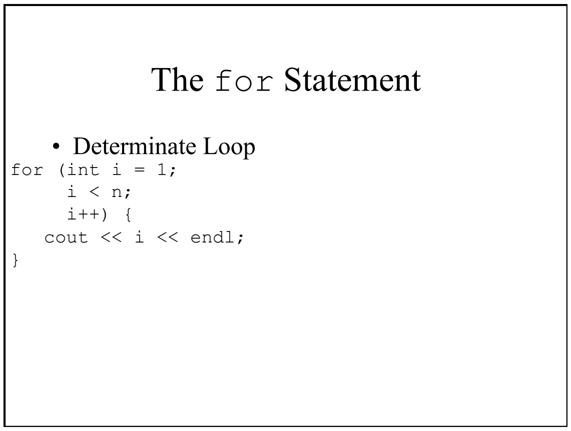
- Programs often must repeat different instructions in a variety of situations
  - sometimes, code must be repeated a determinate number of times
  - other times, code must be repeated an indeterminate number of times



The `for` Statement

- Determinate Loop
  - Do Something Exactly *n* Times, Where *n* Is Known In Advance

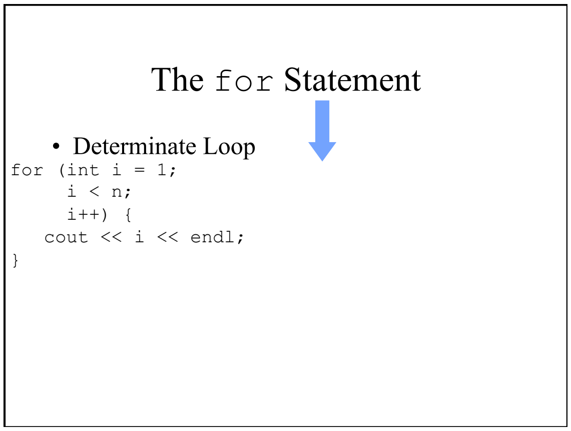
```
for ( int i = 1; i < n; i++ ) {
   ...block of statements...
}
```



The `for` Statement

- Determinate Loop

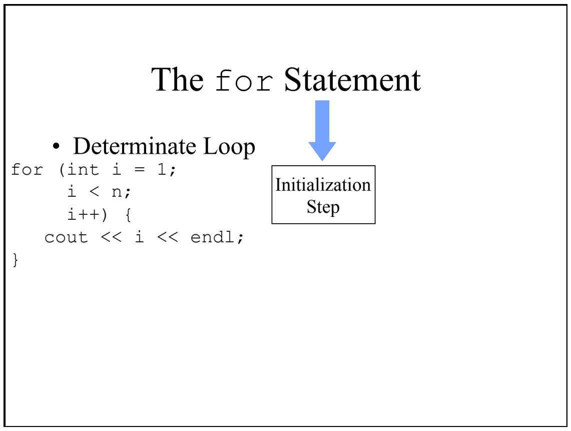
```
for (int i = 1;
     i < n;
     i++) {
   cout << i << endl;
}
```



The `for` Statement

- Determinate Loop

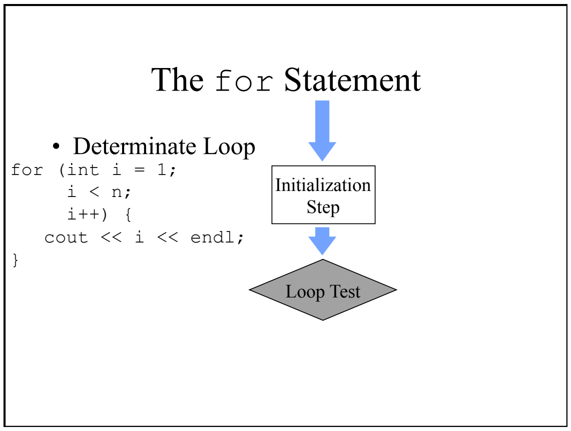
```
for (int i = 1;
     i < n;
     i++) {
   cout << i << endl;
}
```



The for Statement

- Determinate Loop

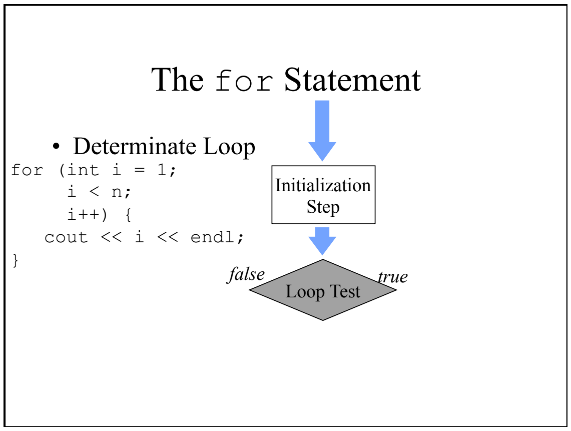
```
for (int i = 1;
     i < n;
     i++) {
   cout << i << endl;
}
```



The for Statement

- Determinate Loop

```
for (int i = 1;
     i < n;
     i++) {
   cout << i << endl;
}
```



The for Statement

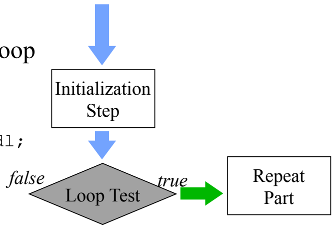
- Determinate Loop

```
for (int i = 1;
     i < n;
     i++) {
   cout << i << endl;
}
```

The `for` Statement

- Determinate Loop

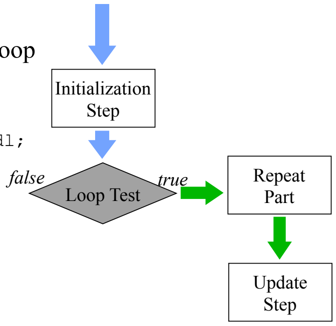
```
for (int i = 1;
      i < n;
      i++) {
   cout << i << endl;
}
```



The `for` Statement

- Determinate Loop

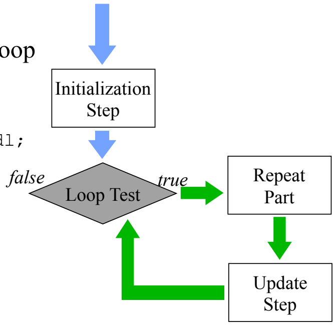
```
for (int i = 1;
      i < n;
      i++) {
   cout << i << endl;
}
```



The `for` Statement

- Determinate Loop

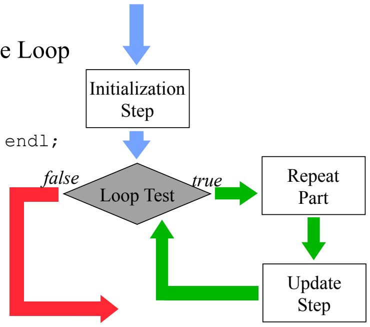
```
for (int i = 1;
      i < n;
      i++) {
   cout << i << endl;
}
```



The `for` Statement

- Determinate Loop

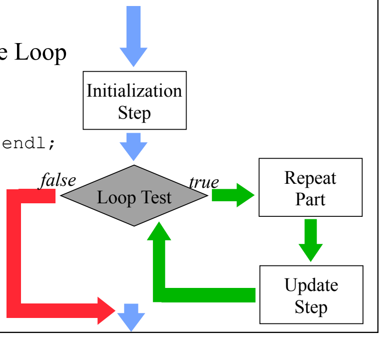
```
for (int i = 1;
     i < n;
     i++) {
   cout << i << endl;
}
```



The `for` Statement

- Determinate Loop

```
for (int i = 1;
     i < n;
     i++) {
   cout << i << endl;
}
```



Time For Our Next Demo!

- ForLoop.cpp

(See Handout For Example 5)

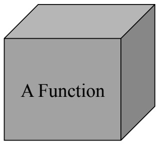
Summarizing Our Fifth Demo!

- Pick The Control Flow That Most Naturally Fits Your Intentions
- A `for` Loop May Never Execute At All

Functions Match The Real World

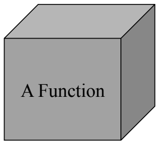
- Large Organizations Are Managed By Dividing Them Into Smaller Departments
- Humans Seem To Manage Complexity By This Process Of Subdivision
- Functions Match This Experience
  - Large Problems Get Broken Down Into Smaller SubPieces

Functions As “Black Boxes”



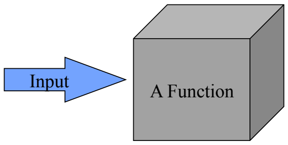
No One But The Function’s Author
Needs To Know What Goes On Inside

Functions As “Black Boxes”



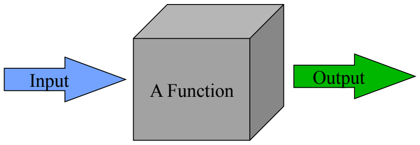
As Users, All We Know Is That The
Function Accepts Some Kind Of Input
And Generates Some Kind Of Output

Functions As “Black Boxes”



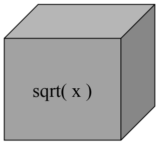
As Users, All We Know Is That The
Function Accepts Some Kind Of Input
And Generates Some Kind Of Output

Functions As “Black Boxes”

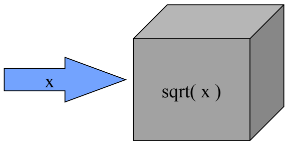


As Users, All We Know Is That The
Function Accepts Some Kind Of Input
And Generates Some Kind Of Output

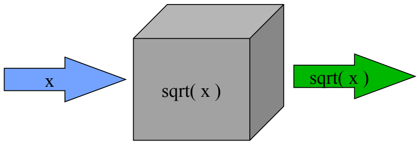
Functions As “Black Boxes”



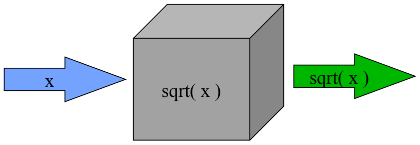
Functions As “Black Boxes”



Functions As “Black Boxes”



Functions As “Black Boxes”



As Users, We Know **What** It Does But Not **How** It Does It

“Information Hiding”

Functions

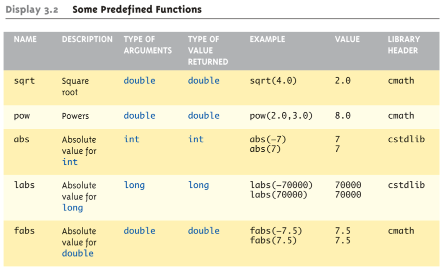
- A named subprogram that can take parameters and returns a result
  - `main( )` is a function that returns `int`
- Functions Are A Way To Reuse Code
- Functions Are An Important Part Of Programming
  - “divide and conquer” strategy

Predefined Functions

- C++ Libraries Offer Us Many Functions
  - `<cmath>` described in Appendix 4
  - `#include <cmath>` acquires all the declarations in this system library

Function	**Argument**	**Result**
`ceil(x)`	`double`	`double`
`fabs(x)`	`double`	`double`
`floor(x)`	`double`	`double`
`pow(x,y)`	`double`	`double`
`sqrt(x)`	`double`	`double`

Various Available Functions



Display 3.2 **Some Predefined Functions**

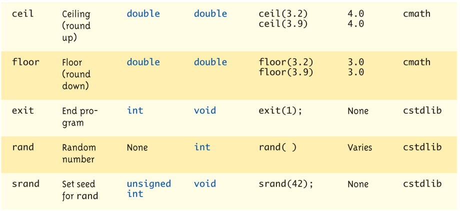
NAME	DESCRIPTION	TYPE OF ARGUMENTS	TYPE OF VALUE RETURNED	EXAMPLE	VALUE	LIBRARY HEADER
`sqrt`	Square root	`double`	`double`	`sqrt(4.0)`	`2.0`	`cmath`
`pow`	Powers	`double`	`double`	`pow(2.0,3.0)`	`8.0`	`cmath`

`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`

Various Available Functions



`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`

Syntax Of A Function Call

- The Call To A Function Call Is A Signature
- Syntax:
  `rv = funcname( `*`[arg-list]`*` );`
  where:
  - arg-list := `argument[, argument]*`
  - `rv` is the value returned by the function call

Time For Our First Demo!

- MathFuncs.cpp

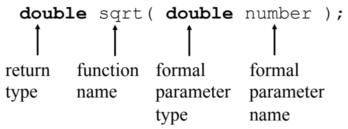
(See Handout For Example 1)

Summarizing Our First Demo!

- Functions Allow Chunks Of Code To Be Reused
- Generally, Functions Enhance Readability
- Parameters Are Passed By Position

Function Prototype

- A Function Prototype or Function Header Defines How A Function Is Called
  - tells everything you need to know to use it



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

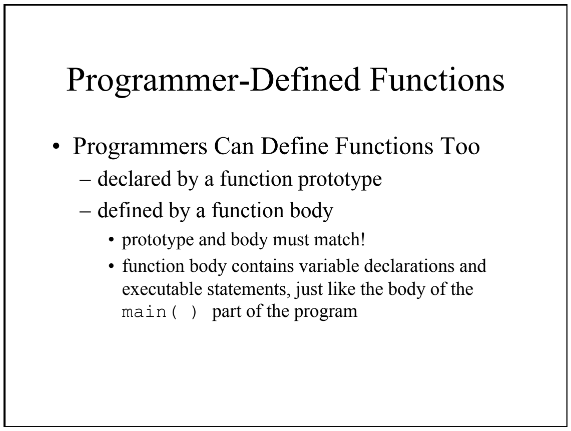
return type

function name

formal parameter type

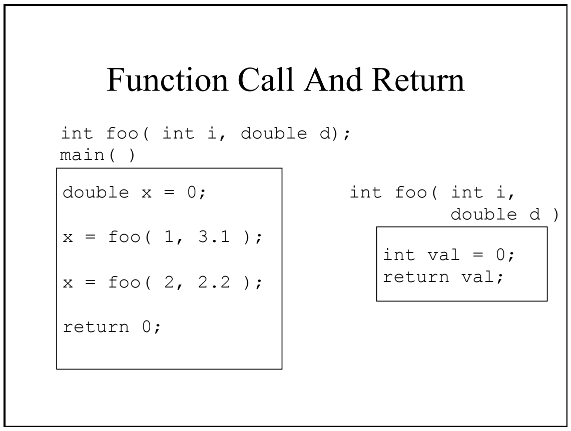
formal parameter name

  - formal parameter gets replaced by the actual parameter at run-time



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



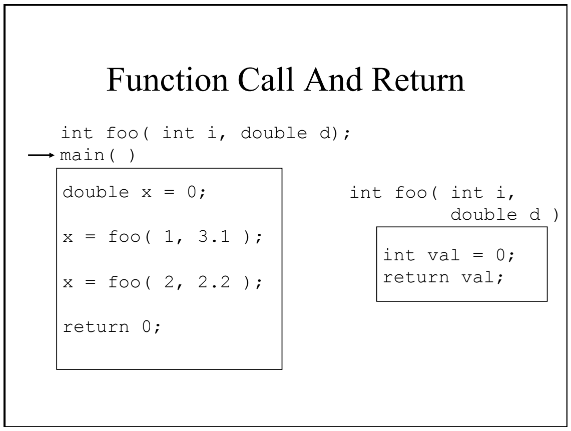
Function Call And Return

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

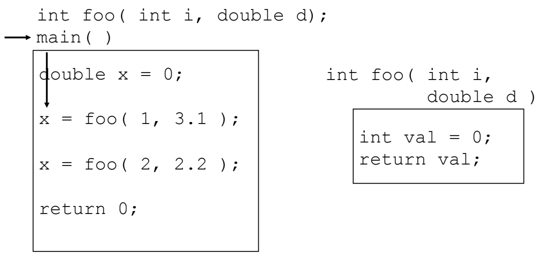
```
  int foo( int i, double d);
→ main( )
```

```
double x = 0;

x = foo( 1, 3.1 );

x = foo( 2, 2.2 );

return 0;
```

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

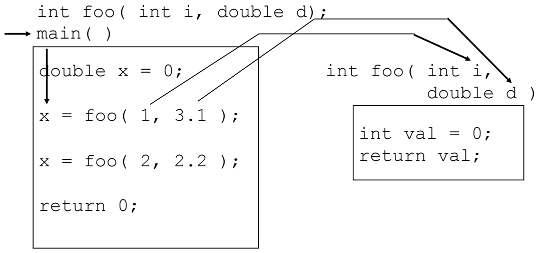
```
int val = 0;
return val;
```

Function Call And Return



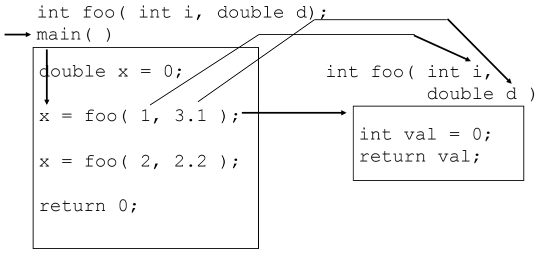
```
int foo( int i, double d);
main( )
double x = 0;                int foo( int i,
                                      double d )
x = foo( 1, 3.1 );
                             int val = 0;
x = foo( 2, 2.2 );           return val;

return 0;
```

Function Call And Return



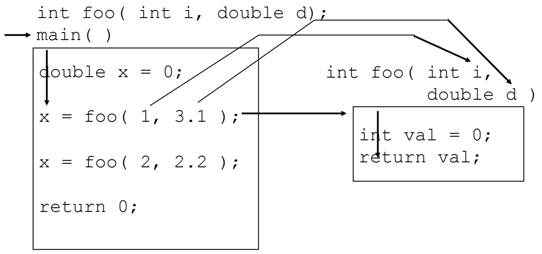
```
int foo( int i, double d);
main( )
double x = 0;                int foo( int i,
                                      double d )
x = foo( 1, 3.1 );
                             int val = 0;
x = foo( 2, 2.2 );           return val;

return 0;
```

Function Call And Return



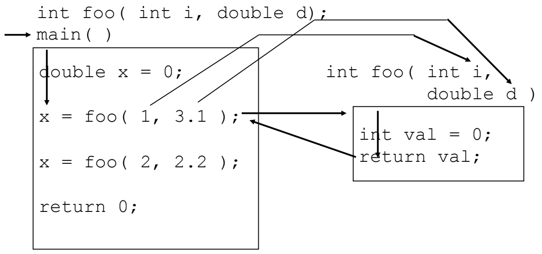
```
int foo( int i, double d);
main( )
double x = 0;                int foo( int i,
                                      double d )
x = foo( 1, 3.1 );
                             int val = 0;
x = foo( 2, 2.2 );           return val;

return 0;
```

Function Call And Return



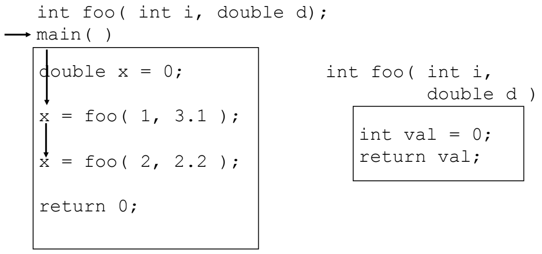
```
int foo( int i, double d);
main( )
double x = 0;
x = foo( 1, 3.1 );
x = foo( 2, 2.2 );
return 0;

int foo( int i,
         double d )
int val = 0;
return val;
```

Function Call And Return



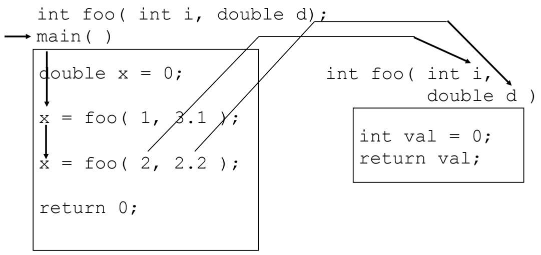
```
int foo( int i, double d);
main( )
double x = 0;
x = foo( 1, 3.1 );
x = foo( 2, 2.2 );
return 0;

int foo( int i,
         double d )
int val = 0;
return val;
```

Function Call And Return



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

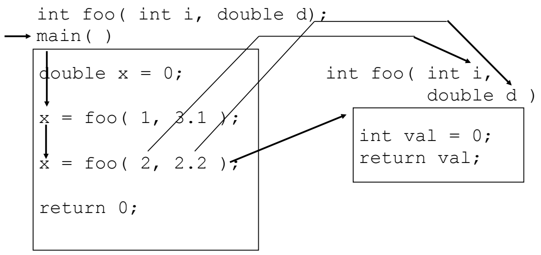
Function Call And Return



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

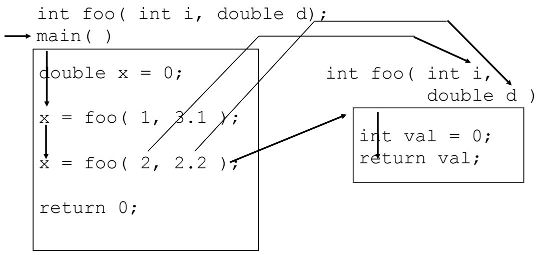
Function Call And Return



```
int foo( int i, double d);
main( )
double x = 0;

x = foo( 1, 3.1 );

x = foo( 2, 2.2 );

return 0;
```

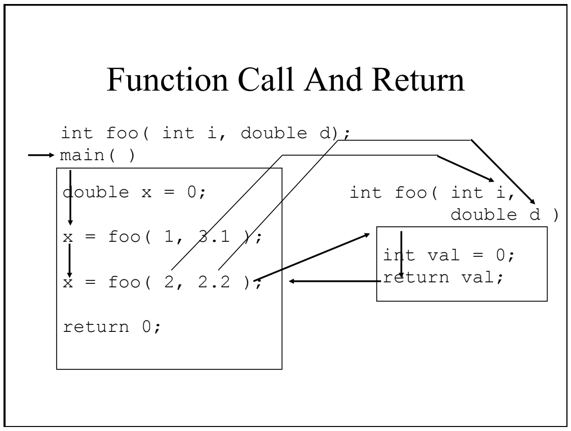
```
int foo( int i,
         double d )
int val = 0;
return val;
```

Function Call And Return



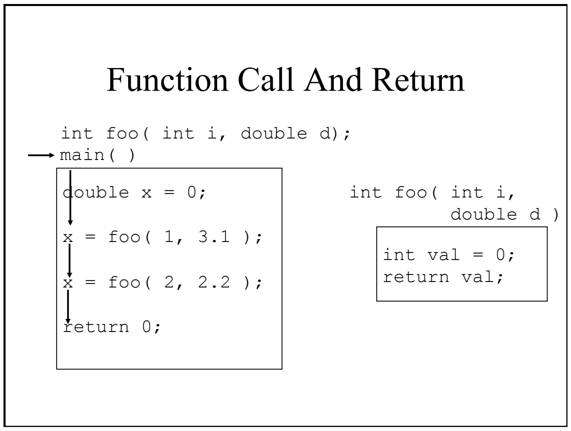
```
int foo( int i, double d);
main( )
double x = 0;

x = foo( 1, 3.1 );

x = foo( 2, 2.2 );

return 0;
```

```
int foo( int i,
         double d )
int val = 0;
return val;
```



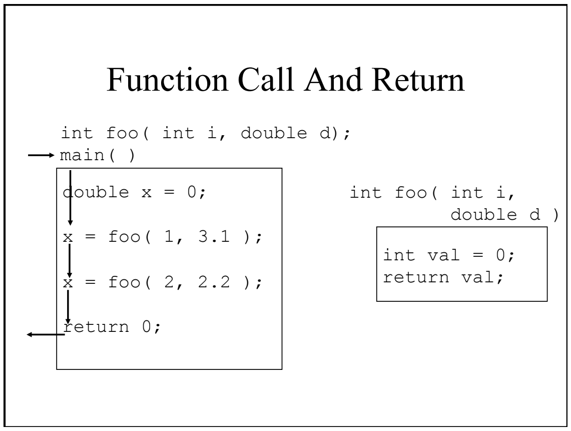
Function Call And Return

```
int foo( int i, double d);
main( )
double x = 0;             int foo( int i,
                                   double d )
x = foo( 1, 3.1 );
                           int val = 0;
x = foo( 2, 2.2 );         return val;

return 0;
```



Function Call And Return

```
int foo( int i, double d);
main( )
double x = 0;             int foo( int i,
                                   double d )
x = foo( 1, 3.1 );
                           int val = 0;
x = foo( 2, 2.2 );         return val;

return 0;
```



Function Call And Return

```
int foo( int i, double d);
main( )
double x = 0;             int foo( int i,
                                   double d )
x = foo( 1, 3.1 );
                           int val = 0;
x = foo( 2, 2.2 );         return val;

return 0;
```

Time For Our Next Demo!

- UserFuncs.cpp

(See Handout For Example 2)

Summarizing Our Second Demo!

- Functions Need To Be Documented!
- Formal Parameters Receive Copies Of The Runtime Function Parameters
- Return Values, Although Provided, May Be Ignored By The Caller
- Functions Are Defined Once But May Be Used Countless Times

Summarizing Functions

- Functions Are Like Small Programs
- Functions Use Formal Parameters For Input
- Functions Use `return` Statement To Communicate To The Caller
- Each Function Call Must Be Defined By A Function Prototype

Parameter Passing

- Pass-By-Value Scheme Is What We Have Seen So Far
  - Functions See A Copy Of The Value Passed, Not The Value Itself
  - *i*-th Formal Parameter Is A Local Variable Initialized To The *i*-th Actual Argument
- There Are Other Passing Schemes We' ll Mention Later

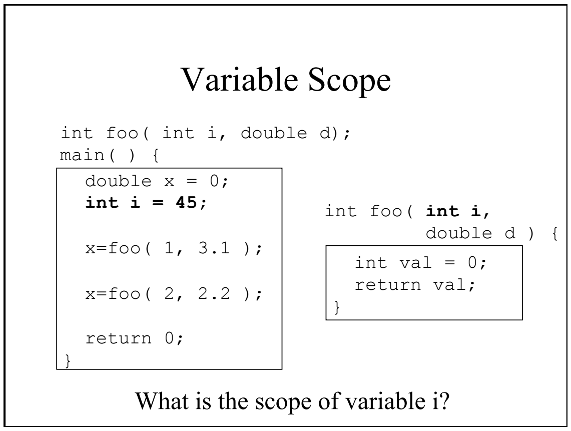
Variable Scope

- 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
  - generally, called functions have no access to variables available to the caller

Variable Scope

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

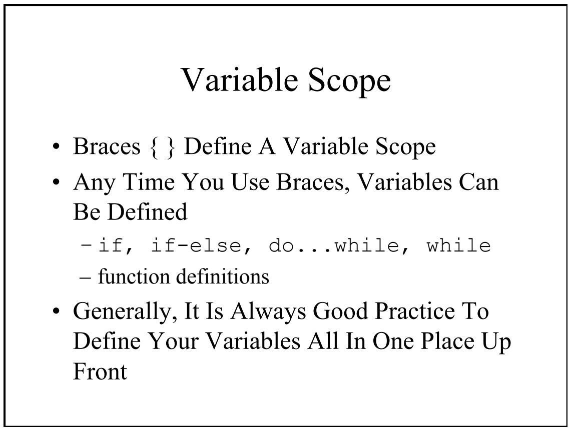


Variable Scope

```
int foo( int i, double d);
main( ) {
  double x = 0;
  int i = 45;

  x=foo( 1, 3.1 );

  x=foo( 2, 2.2 );

  return 0;
}
```

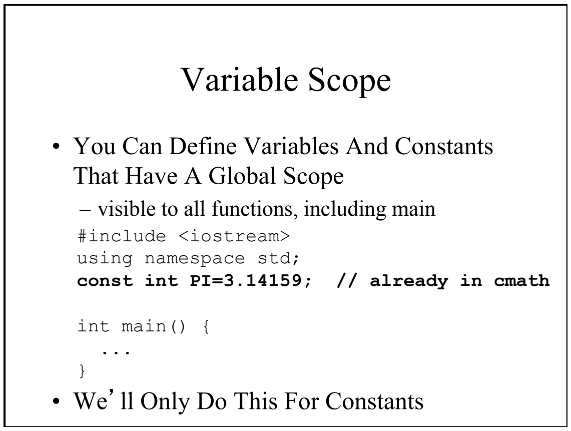
```
int foo( int i,
         double d ) {
  int val = 0;
  return val;
}
```

What is the scope of variable i?



Variable Scope

- Braces { } Define A Variable Scope
- Any Time You Use Braces, Variables Can Be Defined
  - `if, if-else, do...while, while`
  - function definitions
- Generally, It Is Always Good Practice To Define Your Variables All In One Place Up Front



Variable Scope

- You Can Define Variables And Constants That Have A Global Scope
  - visible to all functions, including main

```
#include <iostream>
using namespace std;
const int PI=3.14159;  // already in cmath

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

- We'll Only Do This For Constants

Time For Our Next Demo!

- Scope.cpp

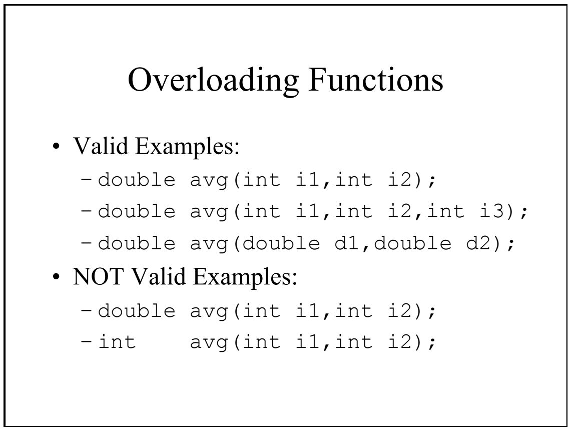
(See Handout For Example 3)

Summarizing Our Third Demo!

- Regardless How Formal Parameters Are Named, They Do Not Clash With Similarly Named Variables In The Caller
- Regardless How Local Variables Are Named, They Do Not Clash With Similarly Named Variables In The Caller

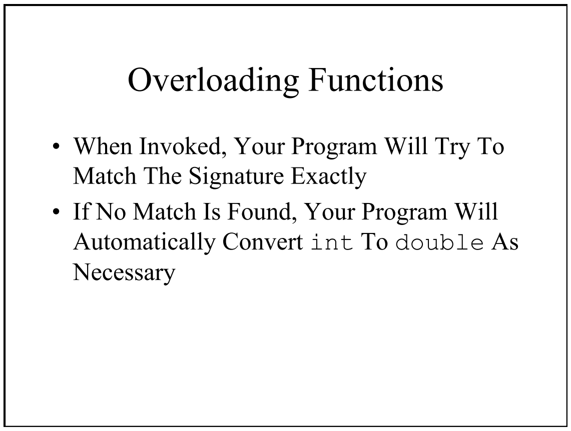
Overloading Functions

- In C++, Your Programs Can Have Two Or More Function Definitions For The Same Functions Name.
- These Functions Are Called “Overloaded”
- Each Definition Must Have A Prototype That Differs In The Number Of Parameters Or Their Types
  - value returned is not a valid difference



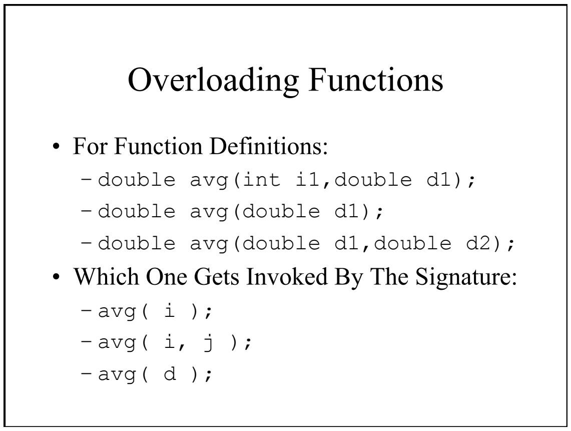
Overloading Functions

- Valid Examples:
  - `double avg(int i1,int i2);`
  - `double avg(int i1,int i2,int i3);`
  - `double avg(double d1,double d2);`
- NOT Valid Examples:
  - `double avg(int i1,int i2);`
  - `int    avg(int i1,int i2);`



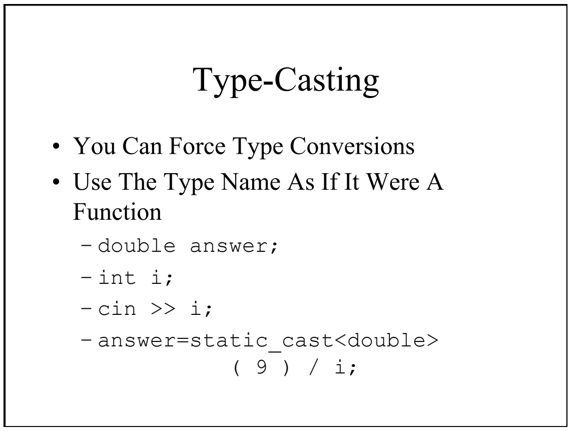
Overloading Functions

- When Invoked, Your Program Will Try To Match The Signature Exactly
- If No Match Is Found, Your Program Will Automatically Convert `int` To `double` As Necessary



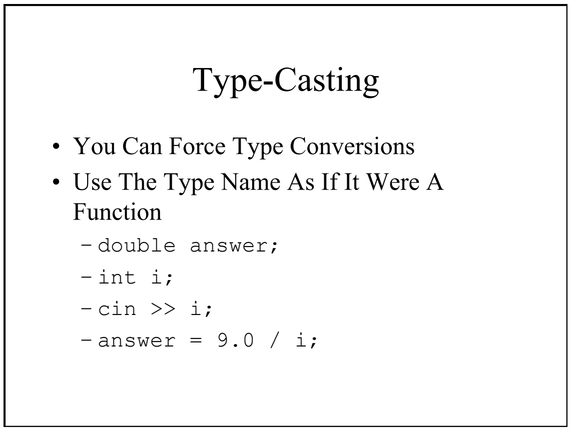
Overloading Functions

- For Function Definitions:
  - `double avg(int i1,double d1);`
  - `double avg(double d1);`
  - `double avg(double d1,double d2);`
- Which One Gets Invoked By The Signature:
  - `avg( i );`
  - `avg( i, j );`
  - `avg( d );`



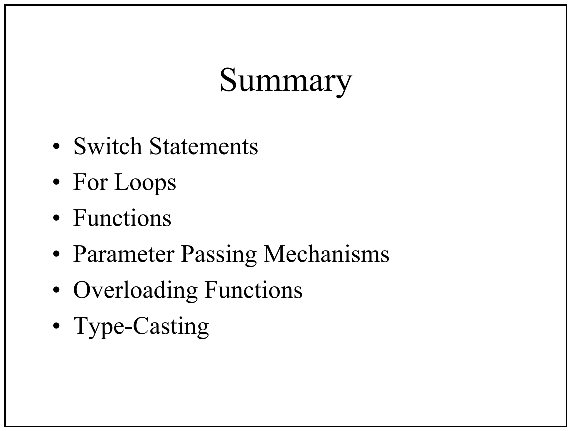
Type-Casting

- You Can Force Type Conversions
- Use The Type Name As If It Were A Function
  - `double answer;`
  - `int i;`
  - `cin >> i;`
  - `answer=static_cast<double>`
    `( 9 ) / i;`



Type-Casting

- You Can Force Type Conversions
- Use The Type Name As If It Were A Function
  - `double answer;`
  - `int i;`
  - `cin >> i;`
  - `answer = 9.0 / i;`



Summary

- Switch Statements
- For Loops
- Functions
- Parameter Passing Mechanisms
- Overloading Functions
- Type-Casting