CS 31: Introduction To Computer Science I

Howard A. Stahl

Agenda

- Revisiting Output and Input
- Type Compatibility and Conversion
- Expressions and Precedence Rules
- Selective Control
- Repetition

Revisiting C++ Output

- cout is connected to the terminal screen
  - `cout << expr`$_1$ `<< ... << expr`$_n$ `;`
- `<<` referred to as the insertion operator
- expressions are normally variables or literals
- the identifier `endl` can be used to send a new-line-character and flush the buffer

Revisiting C++ Output

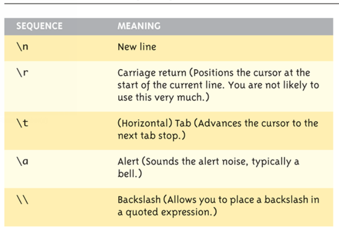
- Examples:
  - `cout << "Hello, World\n";`
  - `cout << "Hello" << "," << " " << "World" << endl;`
  - `cout << "1" << endl;`
  - `cout << 1 << endl;`
  - `cout << 5*3 << endl;`
  - `cout << 1 << 1 << endl;`

Escape Sequences

- Characters Following A Backslash Have A Different Meaning From The Character Themselves

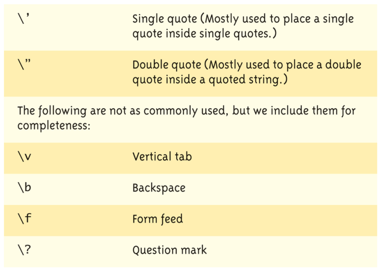
Escape Sequences

Some Escape Sequences



SEQUENCE	MEANING
\n	New line
\r	Carriage return (Positions the cursor at the start of the current line. You are not likely to use this very much.)
\t	(Horizontal) Tab (Advances the cursor to the next tab stop.)
\a	Alert (Sounds the alert noise, typically a bell.)
\\	Backslash (Allows you to place a backslash in a quoted expression.)

Escape Sequences



`\'`	Single quote (Mostly used to place a single quote inside single quotes.)
`\"`	Double quote (Mostly used to place a double quote inside a quoted string.)
The following are not as commonly used, but we include them for completeness:	
`\v`	Vertical tab
`\b`	Backspace
`\f`	Form feed
`\?`	Question mark

"Magic Formula"

- Use The Following To Output 2 Digits After The Decimal Point When Working With cout
  - `cout.setf( ios::fixed );`
  - `cout.setf( ios::showpoint );`
  - `cout.precision( 2 );`

Revisiting C++ Input

- cout is connected to the terminal screen
  - `cin >> var`$_1$ `>> ... >> var`$_n$ `;`
- `>>` referred to as the extraction operator
- variables are assigned values from standard input
- values read are separated by whitespace
  - spaces, tabs, CR
  - `cin` is greedy

Revisiting C++ Input

- Examples:
  - `cin >> fahrenheit;`
  - `cin >> x_coord >> y_coord;`
- Always better to issue prompt for input
  - `cout << “Enter temperature: “;`
  - `cin >> fahrenheit;`
  - `char symbol1, symbol2;`
  - `cout << "Enter your initials:";`
  - `cin >> symbol1 >> symbol2;`

Revisiting Character Data

- Character Literals use single quote
  - `‘A’      ‘5’      ‘?’      ‘\n’`
- Character variables can be assigned character literal values
  - `char first;`
  - `char last;`
  - `first = ‘P’;`
  - `last = ‘ ‘;`

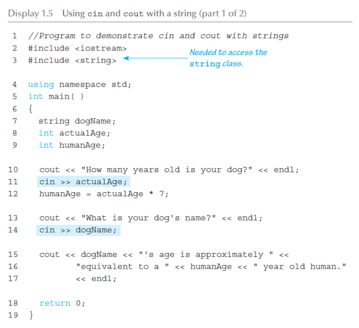
Revisiting String Data

- String Literals use double quotes
  - `“Hello World”`
  - `“Thank You, Maam”`

Revisiting String Data

- C++ has a data type of "string" to store sequences of characters
  - Not a primitive data type - A distinction that will become much more important later on…
  - Must say: **`#include <string>`**
  - Operator + when working on strings will concatenate two strings together
  - **`cin >> aString`** reads only up to the first whitespace character (tab, space, newline)

I/O Example

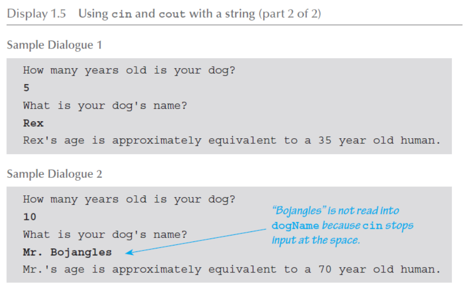


Display 1.5 Using `cin` and `cout` with a string (part 1 of 2)

```
//Program to demonstrate cin and cout with strings
#include <iostream>
#include <string>

using namespace std;
int main( )
{
  string dogName;
  int actualAge;
  int humanAge;

  cout << "How many years old is your dog?" << endl;
  cin >> actualAge;
  humanAge = actualAge * 7;

  cout << "What is your dog's name?" << endl;
  cin >> dogName;

  cout << dogName << "'s age is approximately " <<
          "equivalent to a " << humanAge << " year old human."
          << endl;

  return 0;
}
```

Needed to access the `string` class.

I/O Example



Display 1.5 Using `cin` and `cout` with a string (part 2 of 2)

Sample Dialogue 1

```
How many years old is your dog?
5
What is your dog's name?
Rex
Rex's age is approximately equivalent to a 35 year old human.
```

Sample Dialogue 2

```
How many years old is your dog?
10
What is your dog's name?
Mr. Bojangles
Mr.'s age is approximately equivalent to a 70 year old human.
```

"Bojangles" is not read into `dogName` because `cin` stops input at the space.

Important String Handling Detail

- **`cin >>`** eats leading whitespace but breaks on whitespace
- **`getline( cin , aString )`** reads a texual line ending with newline, consuming the newline character itself

Important String Handling Detail

- **`cin >>`** eats leading whitespace but breaks on whitespace
- **`getline( cin , aString )`** reads a texual line ending with newline, consuming the newline character itself
- **`cin.ignore( 1000, '\n')`** discards up to and including the next \n character or 1000 characters, which ever comes first

Important String Handling Detail

- **`cin >> actualAge;`**
- **`cin.ignore( INT_MAX, '\n' );`**
- **`getline( cin, dogName );`**
- **`getline( cin, dogBreed )`**

Constants

- It's a good idea to name values to prevent "magic" values showing up in your code
- Use `const` declaration to state that value cannot change after assignment
- Examples:

```
const double PI = 3.14159;
const int LIMIT = 15;
```

Type Compatibility

- Generally, you should not try storing values of one type in a variable of a different type
  - Type Mismatch Error
- Storing a `double` in `int` leads to truncation
- Storing an `int` in a `double` is OK
  - best to convert

Type Compatibility

- You can coerce types from one to another by saying:

```
double value( 12.510104 );
int i=static_cast<int>( value );
```

Type Compatibility

- Need to be careful if you mix types and values on assignment statements or arithmetic expressions
- When working with

```
        A operand B
```

  where operand may be +, -, *, /, or %
  - if `A` or `B` is `double`, the result will be `double`

Type Compatibility

- Examples:
  - 3 + 4.4 =
  - 2.2 * 3 =
  - 2.2 * 3.0 =
  - 2 * 3 =
  - 4.5 * 2 =
  - 9 * 2 =

Division

- When working with

```
        A / B
```

- If either operand is real, then the other will be converted to a real and the result will be real
- If both are `int`, then integer division occurs and the result will be an `int`
  - modulus operator `%` yeilds the remainder

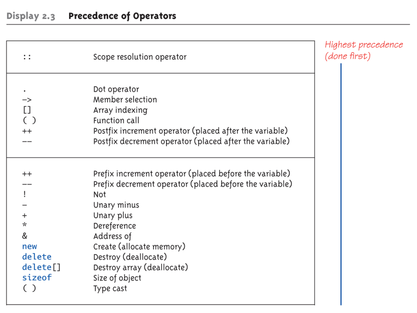
Division

- Examples:
  - 9 / 4 =
  - 9.0 / 4 =
  - 9 % 4 =
  - 11 / 4 =
  - 11 % 4 =
  - -11 % 4 =
  - TRICK QUESTION: `var_a * (1 / 4) =`

Precedence Rules

- Operators in an expression are evaluated according to precedence rules
  - ( )
  - *, /, %
  - +, -
  - =, +=, *=, /=, -=, %=
- Precedence described in Appendix 2, page 917

Precedence Rules


Display 2.3 **Precedence of Operators**

`::`	Scope resolution operator
`.`	Dot operator
`->`	Member selection
`[]`	Array indexing
`( )`	Function call
`++`	Postfix increment operator (placed after the variable)
`--`	Postfix decrement operator (placed after the variable)
`++`	Prefix increment operator (placed before the variable)
`--`	Prefix decrement operator (placed before the variable)
`!`	Not
`-`	Unary minus
`+`	Unary plus
`*`	Dereference
`&`	Address of
`new`	Create (allocate memory)
`delete`	Destroy (deallocate)
`delete[]`	Destroy array (deallocate)
`sizeof`	Size of object
`( )`	Type cast

Precedence Rules

`*`
`/`
`%`
	Multiply
	Divide
	Remainder (modulo)

`+` `-`	Addition Subtraction

`<<`
`>>`
	Insertion operator (console output)
	Extraction operator (console input)

Lower precedence (done later)

Precedence Rules

Display 2.3 Precedence of Operators

All operators in part 2 are of lower precedence than those in part 1.

`<`
`>`
`<=`
`>=`
	Less than
	Greater than
	Less than or equal to
	Greater than or equal to

`==`
`!=`
	Equal
	Not equal

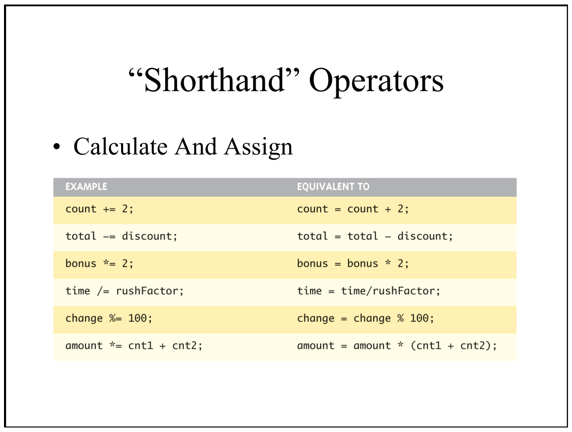
`&&`	And
`||`	Or

Precedence Rules

`=`
`+=`
`-=`
`*=`
`/=`
`%=`
	Assignment
	Add and assign
	Subtract and assign
	Multiply and assign
	Divide and assign
	Modulo and assign

`? :`	Conditional operator
`throw`	Throw an exception
`,`	Comma operator

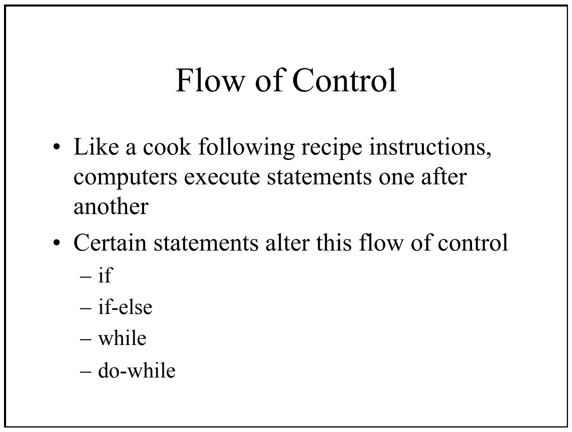
Lowest precedence (done last)



“Shorthand” Operators

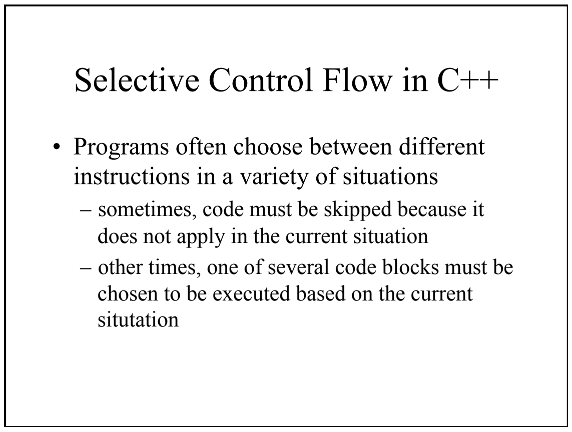
- Calculate And Assign

EXAMPLE	EQUIVALENT TO
`count += 2;`	`count = count + 2;`
`total -= discount;`	`total = total - discount;`
`bonus *= 2;`	`bonus = bonus * 2;`
`time /= rushFactor;`	`time = time/rushFactor;`
`change %= 100;`	`change = change % 100;`
`amount *= cnt1 + cnt2;`	`amount = amount * (cnt1 + cnt2);`



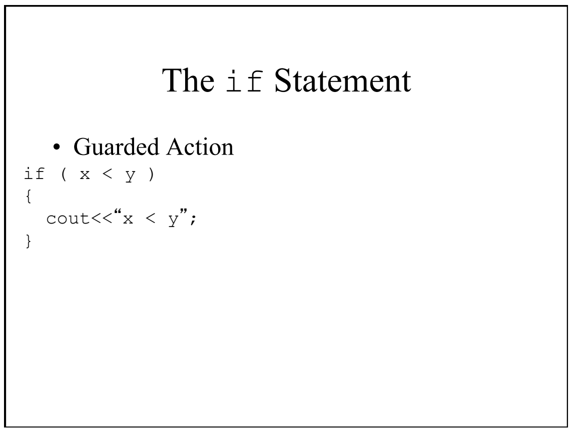
Flow of Control

- Like a cook following recipe instructions, computers execute statements one after another
- Certain statements alter this flow of control
  - if
  - if-else
  - while
  - do-while



Selective Control Flow in C++

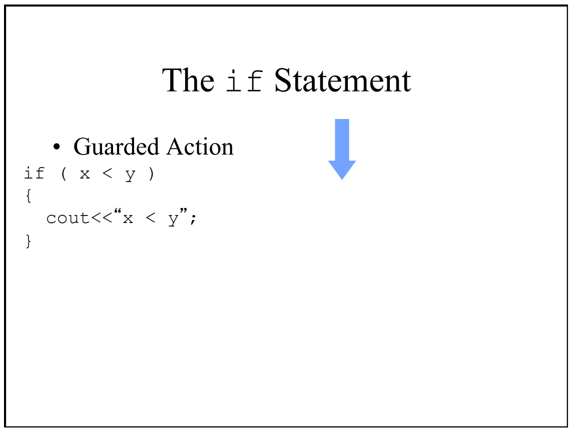
- Programs often choose between different instructions in a variety of situations
  - sometimes, code must be skipped because it does not apply in the current situation
  - other times, one of several code blocks must be chosen to be executed based on the current situtation



The `if` Statement

- Guarded Action

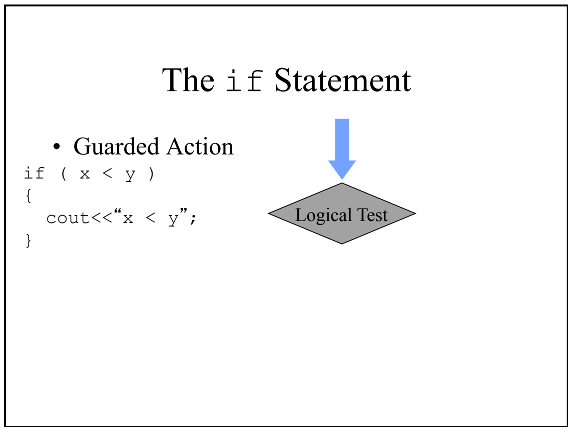
```
if ( x < y )
{
  cout<<"x < y";
}
```



The `if` Statement

- Guarded Action

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



The `if` Statement

- Guarded Action

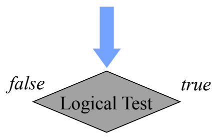
```
if ( x < y )
{
  cout<<"x < y";
}
```

Logical Test

The `if` Statement

- Guarded Action

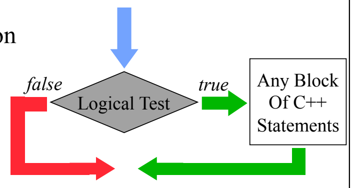
```
if ( x < y )
{
  cout<<“x < y”;
}
```



The `if` Statement

- Guarded Action

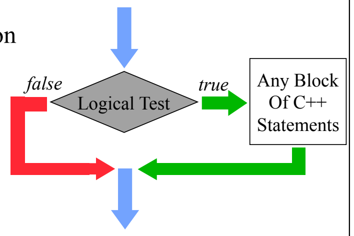
```
if ( x < y )
{
  cout<<“x < y”;
}
```



The `if` Statement

- Guarded Action

```
if ( x < y )
{
  cout<<“x < y”;
}
```

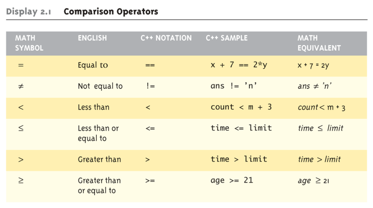


Comparison Operators

- Testing Ordering
  - <, <=, >, >=

Testing Equality

== , !=



Display 2.1 **Comparison Operators**

MATH SYMBOL	ENGLISH	C++ NOTATION	C++ SAMPLE	MATH EQUIVALENT
=	Equal to	==	`x + 7 == 2*y`	x + 7 = 2y
≠	Not equal to	!=	`ans != 'n'`	*ans* ≠ *'n'*
<	Less than	<	`count < m + 3`	*count* < m + 3
≤	Less than or equal to	<=	`time <= limit`	*time* ≤ *limit*
>	Greater than	>	`time > limit`	*time* > *limit*
≥	Greater than or equal to	>=	`age >= 21`	*age* ≥ 21

Common Mistake

- Assignment (=) is different from Equality (==)

```
if (salary = 100000)
{
    cout << "You’re fired!";
}
```

- Equality is always dangerous when working with real operands

More Complex Expressions

- Examples:

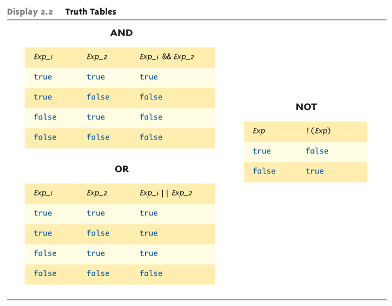
```
if (rate * balance > 1000)
if (a * b != c + d * e)
if (a / b > c)
```

- Never Hurts To Add Parenthesis To Make Your Intentions Clear
- Arithmetic Operators Have Higher Precendence Than Relational Operators
  - 24.00000001 != 24

Logical Operators

- `&&`means AND, `||`means OR, `!` means NOT
- Examples:
  - true and false =
  - false and true =
  - true or false =
  - false or true =
  - not true =
  - not false =

Logical Operators



Display 2.2 **Truth Tables**

AND

Exp_1	*Exp_2*	*Exp_1* && *Exp_2*
true	true	true
true	false	false
false	true	false
false	false	false

OR

Exp_1	*Exp_2*	*Exp_1* || *Exp_2*
true	true	true
true	false	true
false	true	true
false	false	false

NOT

Exp	!(*Exp*)
true	false
false	true

Logical Operators

- Logical Operators connect expressions
- Examples:
```
    if ((0 <= x) && (x > 3))
    if ((y != 1) && (x/y > 4))
```
- C++ uses short-circuit evaluation
  - The evaluation of condition stops because the condition turns false (in case of &&) or true (in case of ||)

Precedence Rules

- Parentheses
- Unary Operators: +, -, !
- Arithmetic Operators: *, / then +, -, then %
- Comparison Operators: <, <=, >, >=, ==, !=
  then && then ||
- See Appendix 2 for full set of rules

Time For Our Next Demo!

- Selection.cpp

Summarizing Our Second Demo!

- Proper Indentation Helps Express Your Intentions
  - But Remember, The Computer Cares Little For Whitespace....

The `if-else` Statement

- Alternative Action

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

The `if-else` Statement

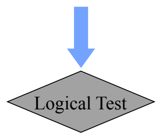
- Alternative Action

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

The `if-else` Statement

- Alternative Action

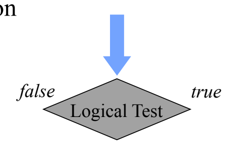
```
if ( x < y )
{
  x++;
}
else
{
  y++;
}
```



The `if-else` Statement

- Alternative Action

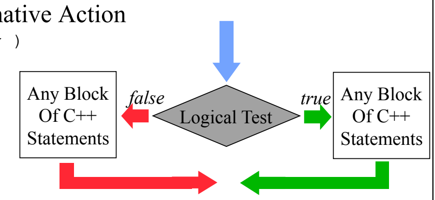
```
if ( x < y )
{
  x++;
}
else
{
  y++;
}
```



The `if-else` Statement

- Alternative Action

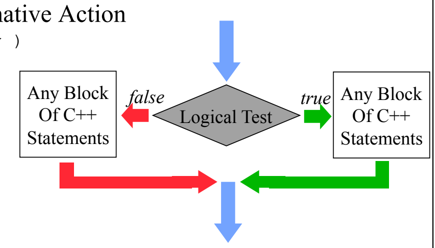
```
if ( x < y )
{
  x++;
}
else
{
  y++;
}
```



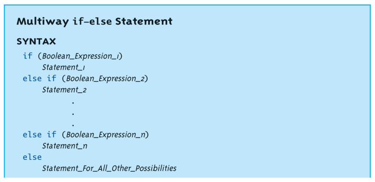
The `if-else` Statement

- Alternative Action

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



Multiway `if-else` Statement

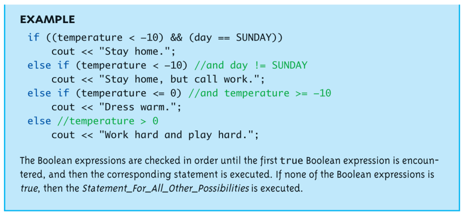


Multiway if-else Statement

SYNTAX

```
if (Boolean_Expression_1)
     Statement_1
else if (Boolean_Expression_2)
     Statement_2
            .
            .
            .
else if (Boolean_Expression_n)
     Statement_n
else
     Statement_For_All_Other_Possibilities
```

Multiway `if-else` Statement



EXAMPLE

```
if ((temperature < -10) && (day == SUNDAY))
    cout << "Stay home.";
else if (temperature < -10) //and day != SUNDAY
    cout << "Stay home, but call work.";
else if (temperature <= 0) //and temperature >= -10
    cout << "Dress warm.";
else //temperature > 0
    cout << "Work hard and play hard.";
```

The Boolean expressions are checked in order until the first `true` Boolean expression is encountered, and then the corresponding statement is executed. If none of the Boolean expressions is *true*, then the *Statement_For_All_Other_Possibilities* is executed.

Nested Conditional Statements

- Selection Statements can be used in combination
- Just be sure that the `else` clause is not dangling...

```
if (precipitating)
   if (temperature < 32)
        cout << "It's snowing";
else // HMMM...
   cout << "It's raining";
```

Time For Our Next Demo!

- Nesting.cpp

(See Handout For Example 3)

Summarizing Our Third Demo!

- Nested Conditionals Make For Complex Scenarios
- Use Parentheses To Prevent A Dangling `else`
- Remember Only One Guarded Action Or 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

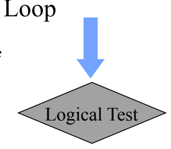
- Indeterminate Loop

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

The `while` Statement

- Indeterminate Loop

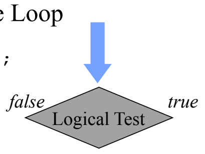
```
while (x < y) {
   cout << “x<y\n”;
   x++;
}
```



The `while` Statement

- Indeterminate Loop

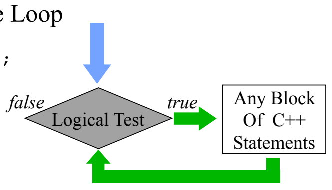
```
while (x < y) {
   cout << “x<y\n”;
   x++;
}
```



The `while` Statement

- Indeterminate Loop

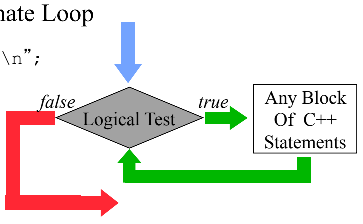
```
while (x < y) {
   cout << “x<y\n”;
   x++;
}
```



The `while` Statement

- Indeterminate Loop

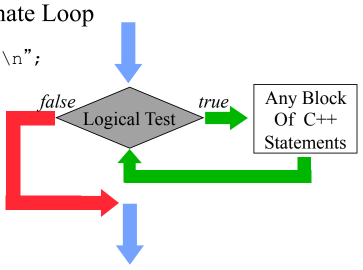
```
while (x < y) {
   cout << “x<y\n”;
   x++;
}
```



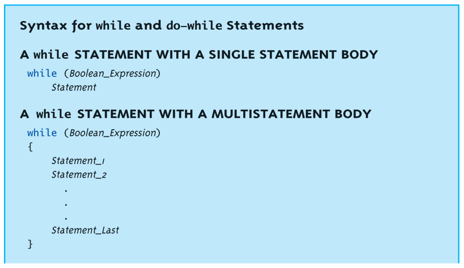
The `while` Statement

- Indeterminate Loop

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



The `while` Statement



Syntax for `while` and `do-while` Statements

A `while` STATEMENT WITH A SINGLE STATEMENT BODY

```
while (Boolean_Expression)
    Statement
```

A `while` STATEMENT WITH A MULTISTATEMENT BODY

```
while (Boolean_Expression)
{
    Statement_1
    Statement_2
        .
        .
        .
    Statement_Last
}
```

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

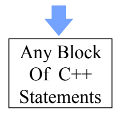
- Indeterminate Loop

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

The `do...while` Statement

- Indeterminate Loop

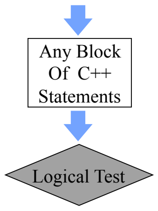
```
do {
   cout << "x<y\n";
   x++;
} while (x < y);
```



The `do...while` Statement

- Indeterminate Loop

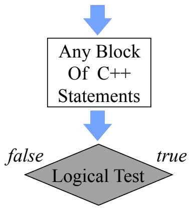
```
do {
   cout << "x<y\n";
   x++;
} while (x < y);
```



The `do...while` Statement

- Indeterminate Loop

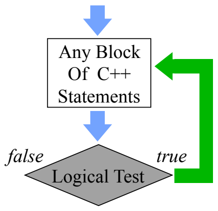
```
do {
   cout << "x<y\n";
   x++;
} while (x < y);
```



The `do...while` Statement

- Indeterminate Loop

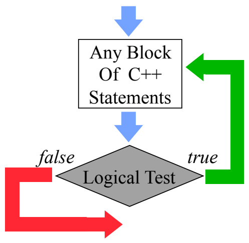
```
do {
   cout << “x<y\n”;
   x++;
} while (x < y);
```



The `do...while` Statement

- Indeterminate Loop

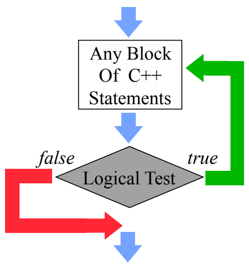
```
do {
   cout << “x<y\n”;
   x++;
} while (x < y);
```



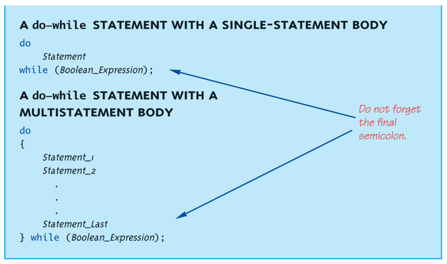
The `do...while` Statement

- Indeterminate Loop

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



The `do...while` Statement



A do-while STATEMENT WITH A SINGLE-STATEMENT BODY

```
do
    Statement
while (Boolean_Expression);
```

A do-while STATEMENT WITH A MULTISTATEMENT BODY

```
do
{
    Statement_1
    Statement_2
      .
      .
      .
    Statement_Last
} while (Boolean_Expression);
```

Do not forget the final semicolon.

Time For Our Next Demo!

- Loops.cpp

(See Handout For Example 3)

Summarizing Our Third 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

Summary

- Revisiting Output and Input
- Type Compatibility and Conversion
- Expressions and Precedence Rules
- Selective Control
- Repetition