CS 31:
Introduction To Computer Science I

Howard A. Stahl

Agenda

- C++ Review
  - Classes
  - Pointers
  - Inheritance
  - Virtual Functions

C++ Review

- A Class In C++ Consists Of Its Member Attributes And Functions
- Each Instance Of A Class Is An Object
- A Member Marked `public` May Be Access From Any Method Of Any Class
- A Member Marked `private` May Only Access From A Method Of Its Class

C++ Review

- A `friend` Of A Class May Access `private` Members Of That Class
  - one-way access
- Typically, Data Members Are Marked `private`
  - promotes information hiding

C++ Review

- A `constructor` Of A Class Describes How An Instance Of This Class May Be Formed
- A Member Function That Changes State Is Called A `mutator`
- A Member Function That Views But Does Not Change State Is Called An `accessor`

C++ Review

- C++ Implementation Techniques Promote The Separation Of Interface From Implementation
  - the interface answers WHAT a class does
  - the implementation answers HOW it does it

C++ Review

- Header Files Describe A Class' Interface
- Source Files Describe A Class' Implementation
- Drivers Code Manipulates Classes

C++ Review

- C++ Has All The Flow Of Control Statements That Exist In The C Language

C++ Review

- C++ Has All The Flow Of Control Statements That Exist In The C Language
  - selection
    - if-then, if-then-else if-else, switch
  - looping
    - for, 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

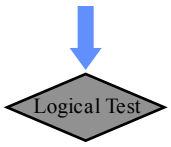
- Guarded Action

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

The `if` Statement

- Guarded Action

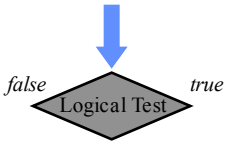
```
if ( x < y )
{
  cout<<"x < y";
}
```



The `if` Statement

- Guarded Action

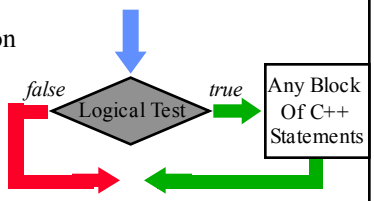
```
if ( x < y )
{
  cout<<"x < y";
}
```



The `if` Statement

- Guarded Action

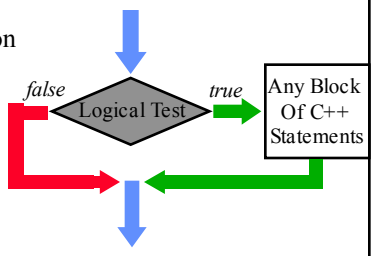
```
if ( x < y )
{
  cout<<"x < y";
}
```



The `if` Statement

- Guarded Action

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



Comparison Operators

- Operators Testing Ordering
  - <, <=, >, >=
- Operators Testing Equality
  - ==, !=

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

- 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 could not possibly be true (in case of &&) or false (in case of ||)

Precedence Rules

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

The `if-else` Statement

- Alternative Action

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

The `if-else` Statement

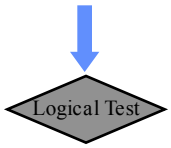
- Alternative Action

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

The `if-else` Statement

- Alternative Action

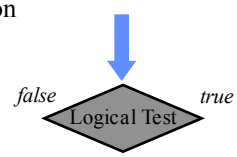
```
if ( x < y )
{
  x++;
}
else
{
  y++;
}
```



The `if-else` Statement

- Alternative Action

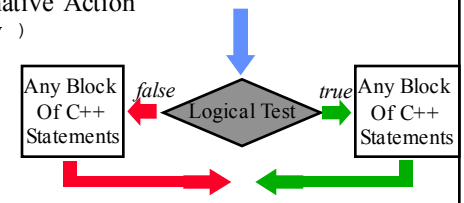
```
if ( x < y )
{
  x++;
}
else
{
  y++;
}
```



The `if-else` Statement

- Alternative Action

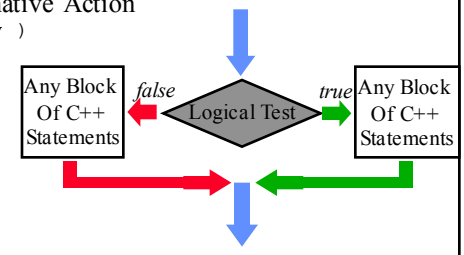
```
if ( x < y )
{
  x++;
}
else
{
  y++;
}
```



The `if-else` Statement

- Alternative Action

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



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";
```

C++ Review

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

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

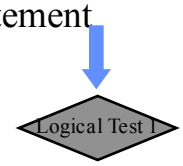
- Multiple Action

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

The `if-else if-else` Statement

- Multiple Action

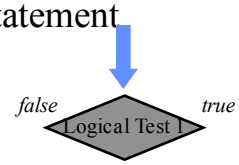
```
if ( x < y )
{
  x++;
}
else if ( x > y )
{
  y++;
}
else {
  x++; y++;
}
```



The `if-else if-else` Statement

- Multiple Action

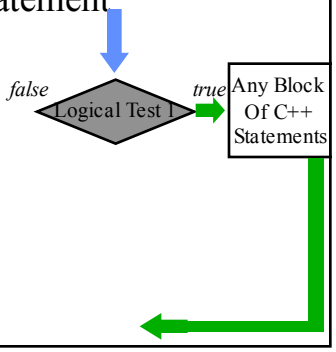
```
if ( x < y )
{
  x++;
}
else if ( x > y )
{
  y++;
}
else {
  x++; y++;
}
```



The `if-else if-else` Statement

- Multiple Action

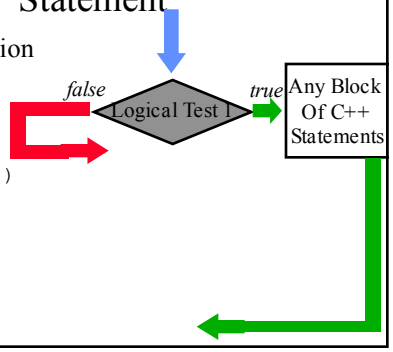
```
if ( x < y )
{
  x++;
}
else if ( x > y )
{
  y++;
}
else {
  x++; y++;
}
```



The `if-else if-else` Statement

- Multiple Action

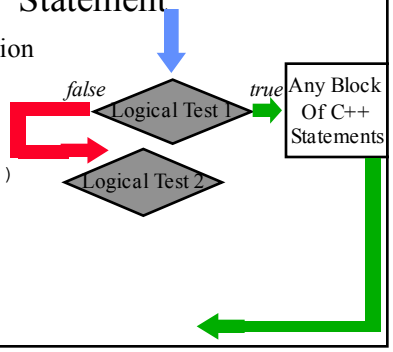
```
if ( x < y )
{
  x++;
}
else if ( x > y )
{
  y++;
}
else {
  x++; y++;
}
```



The `if-else if-else` Statement

- Multiple Action

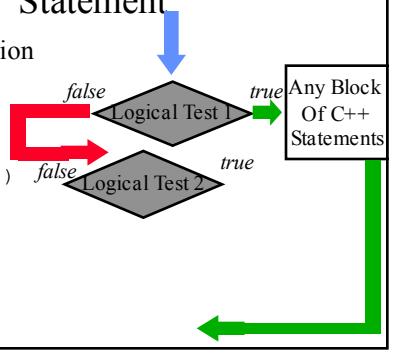
```
if ( x < y )
{
  x++;
}
else if ( x > y )
{
  y++;
}
else {
  x++; y++;
}
```



The `if-else if-else` Statement

- Multiple Action

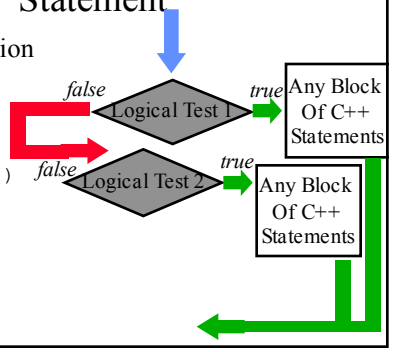
```
if ( x < y )
{
  x++;
}
else if ( x > y )
{
  y++;
}
else {
  x++; y++;
}
```



The `if-else if-else` Statement

- Multiple Action

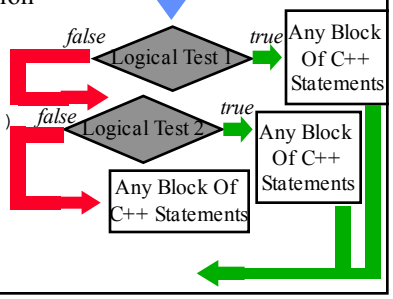
```
if ( x < y )
{
  x++;
}
else if ( x > y )
{
  y++;
}
else {
  x++; y++;
}
```



The `if-else if-else` Statement

- Multiple Action

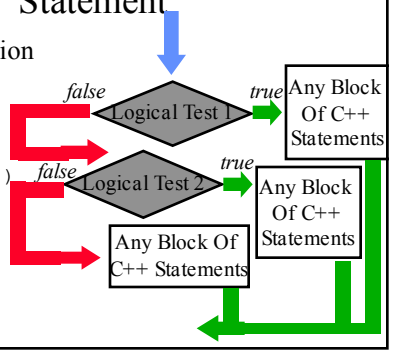
```
if ( x < y )
{
  x++;
}
else if ( x > y )
{
  y++;
}
else {
  x++; y++;
}
```



The `if-else if-else` Statement

- Multiple Action

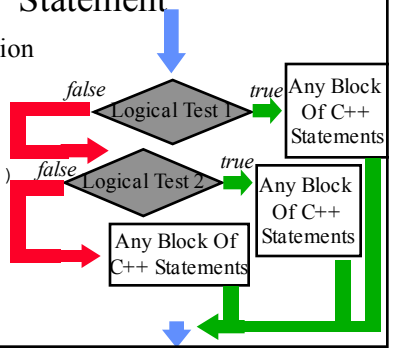
```
if ( x < y )
{
  x++;
}
else if ( x > y )
{
  y++;
}
else {
  x++; y++;
}
```



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

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

The `switch` Statement

- An Alternative To Lots Of `else-if` Choices

```
switch( option ) {
  case 1:
     cout << "1";
     break;
  case 2:
     cout << "2";
     break;
  default:
     cout << "other";
}
```

The `switch` Statement

- An Alternative To Lots Of `else-if` Choices

switch expression must evaluate to an integral value

```
switch( option ) {
  case 1:
     cout << "1";
     break;
  case 2:
     cout << "2";
     break;
  default:
     cout << "other";
}
```

The `switch` Statement

- An Alternative To Lots Of `else-if` Choices

switch expression must evaluate to an integral value

choice must be a constant value

```
switch( option ) {
  case 1:
     cout << "1";
     break;
  case 2:
     cout << "2";
     break;
  default:
     cout << "other";
}
```

The `switch` Statement

- An Alternative To Lots Of `else-if` Choices

switch expression must evaluate to an integral value

choice must be a constant value

break exits this control structure

```
switch( option ) {
  case 1:
     cout << "1";
     break;
  case 2:
     cout << "2";
     break;
  default:
     cout << "other";
}
```

The `switch` Statement

- An Alternative To Lots Of `else-if` Choices

```
switch( option ) {
  case 1:
     cout << "1";
     break;
  case 2:
     cout << "2";
     break;
  default:
     cout << "other";
}
```

switch expression must evaluate to an integral value

choice must be a constant value

break exits this control structure

default case for when no matches occur; completely optional

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

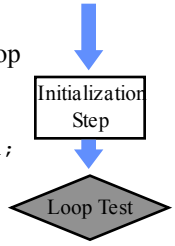
```
for (int i = 1;
     i < n;
     i++) {
   cout << i << endl;
}
```

Initialization Step

The `for` Statement

- Determinate Loop

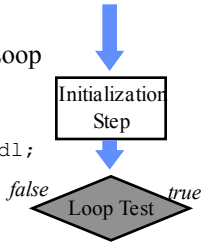
```
for (int i = 1;
     i < n;
     i++) {
   cout << i << endl;
}
```



The `for` Statement

- Determinate Loop

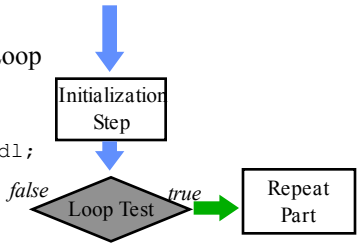
```
for (int i = 1;
     i < n;
     i++) {
   cout << i << endl;
}
```



The `for` Statement

- Determinate Loop

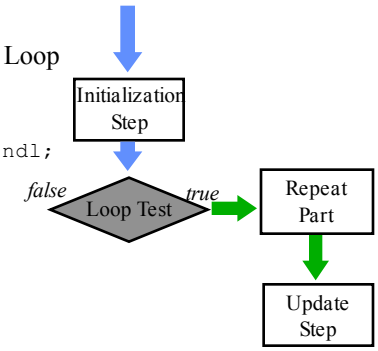
```
for (int i = 1;
     i < n;
     i++) {
   cout << i << endl;
}
```



The `for` Statement

- Determinate Loop

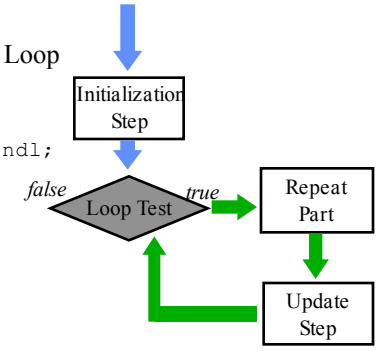
```
for (int i = 1;
     i < n;
     i++) {
   cout << i << endl;
}
```



The `for` Statement

- Determinate Loop

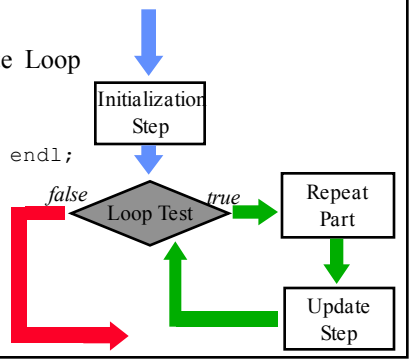
```
for (int i = 1;
     i < n;
     i++) {
   cout << i << endl;
}
```



The `for` Statement

- Determinate Loop

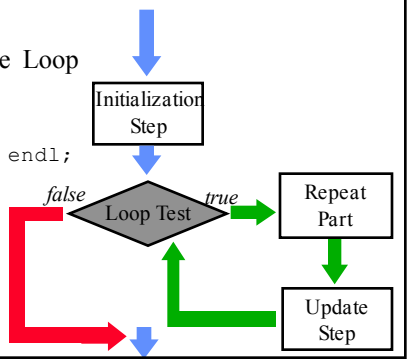
```
for (int i = 1;
     i < n;
     i++) {
   cout << i << endl;
}
```



The `for` Statement

- Determinate Loop

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



The `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

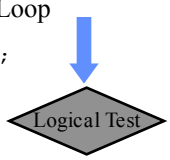
- Indeterminate Loop

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

The `while` Statement

- Indeterminate Loop

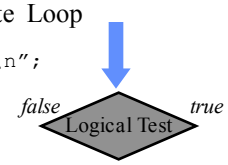
```
while (x < y) {
   cout << "x<y\n";
   x++;
}
```



The `while` Statement

- Indeterminate Loop

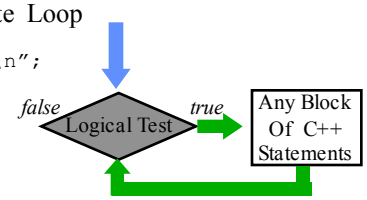
```
while (x < y) {
   cout << "x<y\n";
   x++;
}
```



The `while` Statement

- Indeterminate Loop

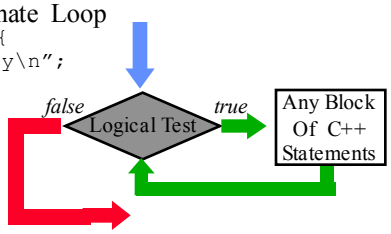
```
while (x < y) {
   cout << "x<y\n";
   x++;
}
```



The `while` Statement

- Indeterminate Loop

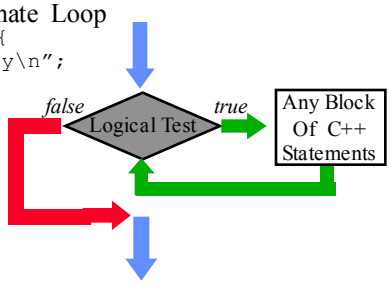
```
while (x < y) {
   cout << "x<y\n";
   x++;
}
```



The `while` Statement

- Indeterminate Loop

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



The `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

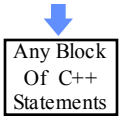
- Indeterminate Loop

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

The `do...while` Statement

- Indeterminate Loop

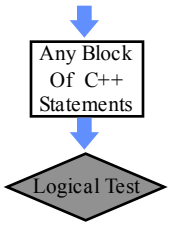
```
do {
   cout << "x<y\n";
   x++;
} while (x < y);
```



The `do...while` Statement

- Indeterminate Loop

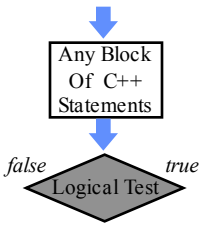
```
do {
   cout << "x<y\n";
   x++;
} while (x < y);
```



The `do...while` Statement

- Indeterminate Loop

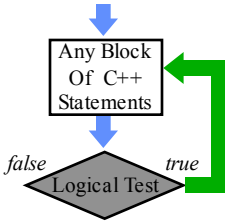
```
do {
   cout << "x<y\n";
   x++;
} while (x < y);
```



The `do...while` Statement

- Indeterminate Loop

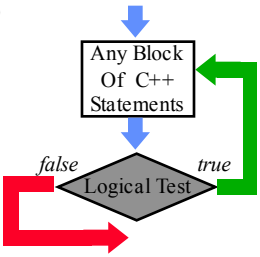
```
do {
   cout << "x<y\n";
   x++;
} while (x < y);
```



The `do...while` Statement

- Indeterminate Loop

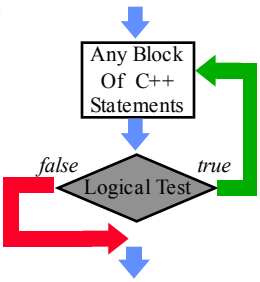
```
do {
   cout << "x<y\n";
   x++;
} while (x < y);
```



The `do...while` Statement

- Indeterminate Loop

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



C++ Review

- 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

C++ Review

- `cout` Is An Instance Of The Class `ostream`
- The Insertion Operator << Is Used To Write Data To `ostreams`
- `cin` Is An Instance Of The Class `istream`
- The Extraction Operator >> Is Used To Read Data From `istreams`
- `cin.getline( )` Reads One Line Of Input

C++ Review

- Use `setw(), setprecision(N)` And `width()` To Control Output Formatting
- The Classes `ifstream` And `ofstream` Support Stream Instances That Get Attached To Files
  - `.open(), .close(), .fail(), .eof(), <<, >>`

Parameter Passing

- C++ Passes All Parameters Using Call-By-Value
  - arguments are copied into the formal parameters
  - inefficient for large object graphs

Parameter Passing

- 3 Rules To Live By:
  - Call By Value May Be Used With Small Objects That Should Not Be Altered By A Function
  - Call By Constant Reference Should Be Used With Large Objects That Should Not Be Altered By A Function
  - Call By Reference Should Be Used For All Objects That May Be Altered By A Function

C++ Review

- I/O Is Pretty Easy
- Each Class Can Overload The << And >> Operators To Determine How Its State Should Be Persisted

Revisiting Pointers

- A Pointer Contains The Address Of A Variable

Revisiting Pointers

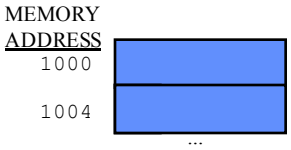
- A Pointer Contains The Address Of A Variable

```
int a = 12;
int* intPtr;
intPtr = &a;
*intPtr = 5;
```

Revisiting Pointers

- A Pointer Contains The Address Of A Variable

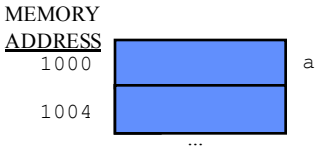
```
int a = 12;
int* intPtr;
intPtr = &a;
*intPtr = 5;
```



Revisiting Pointers

- A Pointer Contains The Address Of A Variable

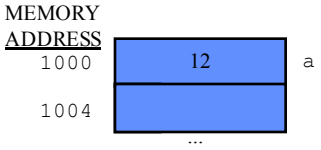
```
int a = 12;
int* intPtr;
intPtr = &a;
*intPtr = 5;
```



Revisiting Pointers

- A Pointer Contains The Address Of A Variable

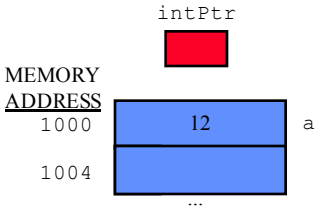
```
int a = 12;
int* intPtr;
intPtr = &a;
*intPtr = 5;
```



Revisiting Pointers

- A Pointer Contains The Address Of A Variable

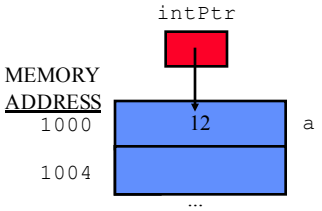
```
int a = 12;
int* intPtr;
intPtr = &a;
*intPtr = 5;
```



Revisiting Pointers

- A Pointer Contains The Address Of A Variable

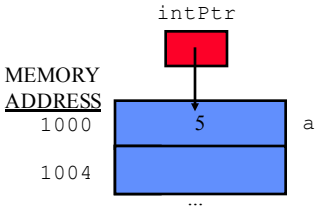
```
int a = 12;
int* intPtr;
intPtr = &a;
*intPtr = 5;
```



Revisiting Pointers

- A Pointer Contains The Address Of A Variable

```
int a = 12;
int* intPtr;
intPtr = &a;
*intPtr = 5;
```



Pointer Assignment

- = Operator Works With Pointers

Pointer Assignment

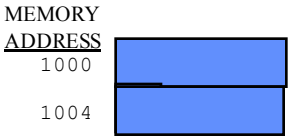
- = Operator Works With Pointers

```
int a = 12, b = 20;
int* p1, *p2;
p1 = &a;
p2 = &b;
p2 = p1;
p2 = 5;
```

Pointer Assignment

- = Operator Works With Pointers

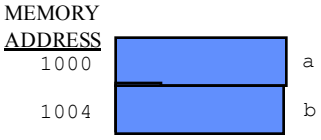
```
int a = 12, b = 20;
int* p1, *p2;
p1 = &a;
p2 = &b;
p2 = p1;
p2 = 5;
```



Pointer Assignment

- = Operator Works With Pointers

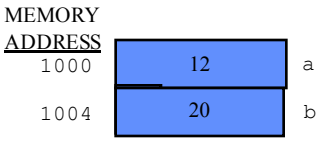
```
int a = 12, b = 20;
int* p1, *p2;
p1 = &a;
p2 = &b;
p2 = p1;
p2 = 5;
```



Pointer Assignment

- = Operator Works With Pointers

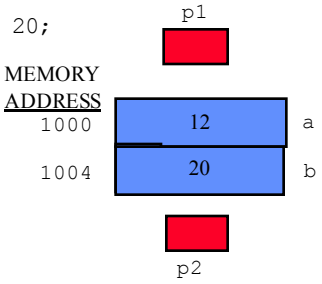
```
int a = 12, b = 20;
int* p1, *p2;
p1 = &a;
p2 = &b;
p2 = p1;
p2 = 5;
```



Pointer Assignment

- = Operator Works With Pointers

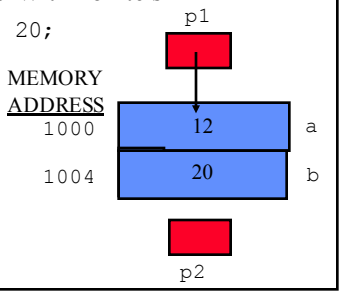
```
int a = 12, b = 20;
int* p1, *p2;
p1 = &a;
p2 = &b;
p2 = p1;
p2 = 5;
```



Pointer Assignment

- = Operator Works With Pointers

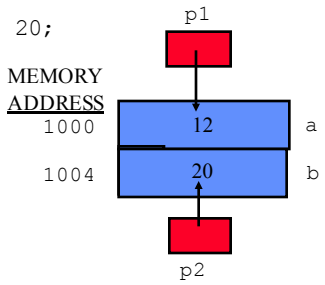
```
int a = 12, b = 20;
int* p1, *p2;
p1 = &a;
p2 = &b;
p2 = p1;
p2 = 5;
```



Pointer Assignment

- = Operator Works With Pointers

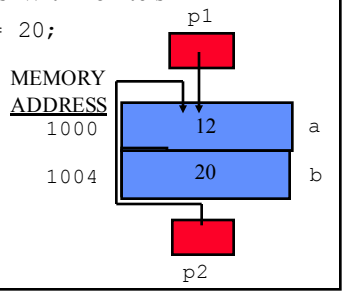
```
int a = 12, b = 20;
int* p1, *p2;
p1 = &a;
p2 = &b;
p2 = p1;
p2 = 5;
```



Pointer Assignment

- = Operator Works With Pointers

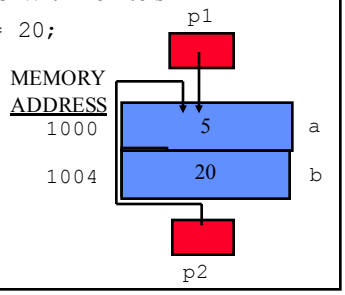
```
int a = 12, b = 20;
int* p1, *p2;
p1 = &a;
p2 = &b;
p2 = p1;
p2 = 5;
```



Pointer Assignment

- = Operator Works With Pointers

```
int a = 12, b = 20;
int* p1, *p2;
p1 = &a;
p2 = &b;
p2 = p1;
p2 = 5;
```



Working With Classes

- The Power Of C++ Comes From Classes
- Object-Orientation Offers Many Benefits

Revisiting Classes

Number
void setValue(int) int getValue( ) void printRomanNumeral( )
int value

```
class Number {
public:
  Number( );
  Number( int initValue )
  void setValue( int v );
  int getValue( );
  void printRomanNumeral(
private:
  int value;
};
```

Revisiting Classes

```
Number::Number() {
  value = 0;
}
Number::Number( int initValue ) {
  value = initValue;
}
int Number::getValue( ) {
  return( value );
}
void Number::setValue( int newValue ) {
  value = newValue;
}
```

Revisiting Classes

```
Number four = Number( 4 );
Number five = Number( 5 );
```

Revisiting Classes

```
Number four = Number( 4 );
Number five = Number( 5 );
```

Wouldn't be great to...

```
Number nine = add( four, five );
```

Revisiting Classes

```
Number four = Number( 4 );
Number five = Number( 5 );


Number nine = add( four, five );


Number add( Number left, Number right ) {
  Number temp=Number(left.value+right.value);
  return( temp );
}
```

Revisiting Classes

```
Number four = Number( 4 );
Number five = Number( 5 );


Number nine = add( four, five );


Number add( Number left, Number right ) {
  Number temp=Number(left.value+right.value);
  return( temp );
}
```

Trouble Is:
It's ILLEGAL!

`friend` Functions

- `friend` Function Of A Class Is **NOT** A Member Function But Has Access To Private Members Of That Class
  - `friend` functions must be named inside the class definition
- `friend` Functions Are Always `public`
  - regardless of where they are placed in the class definition

`friend` Functions

```
class Number {
public:
  digit( );
  digit( int initValue );
  void setValue( int v );
  int getValue( );
  void printRomanNumeral();
  friend Number add(Number left, Number right);
private:
  int value;
};
```

`friend` Functions

```
class Number {
public:
  ...
  friend Number add(Number left,
                    Number right);
  ...
}

Number add( Number left, Number right ) {
  Number t=Number( left.value + right.value );
  return( t );
}
```

`friend` Functions

```
class Number {
public:
  ...
  friend Number add(Number left,
                    Number right);
  ...
}

Number add( Number left, Number right ) {
  Number t=Number( left.value + right.value );
  return( t );
}
```

There is no :: operator

C++ Review

- Use `friend` Functions With Care
  - defeats encapsulation
- Use Member Functions When Working With Only One Object Instance
- Use `friend` Functions When Working With More Than One Object Instance

Revisiting `const` Modifier

- Named Constants Improve Readability

```
const int DAYS_IN_WEEK = 7;

for (int i = 0; i < DAYS_IN_WEEK; i++) {
    read_textbook_chapter();
    study();
}
```

`const` Modifier

- `const` Modifier Also Applies To Function Parameters
  - member functions or normal functions
- `const` Modifier Is Unnecessary With Call-By-Value Parameters
  - any changes made are never seen by the caller
- `const` Modifier Can Be Applied To Call-By-Reference Parameters

`const` Modifier

- Recall That Call-By-Value Results In Argument Copies
  - can be expensive when working with large object graphs
- Call-By-Reference Is Preferred When Passing Objects
- If You Know No Changes Are Made, Mark That Parameter With The `const` Modifier
  - compiler will complain if you alter its value

`const` Modifier

- `const` Modifier Can Also Apply To Member Functions
  - informs the compiler that a member function does not update the `this` pointer of the object being referenced

`const` Modifier

```
class Number {
public:
  digit( );
  digit( int initValue );
  void setValue( int v );
  int getValue( ) const;
  void printRomanNumeral() const;
  friend Number add(const Number& left,
                    const Number& right);
private:
  int value;
};
```

`const` Modifier

```
class Number {
public:
  ...
  friend Number add(const Number& left,
                    const Number& right);
  ...
}

Number add( const Number& left,
            const Number& right ) {
  Number t=Number( left.value + right.value );
  return( t );
}
```

`const` Modifier

```
class Number {
public:
  ...
  friend Number add(const Number& left,
                    const Number& right);
  ...
}

Number add( const Number& left,
            const Number& right ) {
  Number t=Number( left.value + right.value );
  return( t );
}
```

const Modifier
In Both Declaration
And Definition

C++ Review

- Using `const` Modifier Is An All-Or-Nothing Proposition
- Due To Function Calls Within Functions, The Compiler Will Cascade `const` Modifier Requirements

Revisiting Operators

```
Number four = Number( 4 );
Number five = Number( 5 );

Number nine = four + five;
```

Wouldn't be great to..

Revisiting Operators

```
Number four = Number( 4 );
Number five = Number( 5 );

Number nine = four + five;
```

Operator Overloading

- All The Operators You Have Learned About So Far Can Be Overloading By Class Definitions
  - +, -, ==, /, *, ++, --, +=, -=, *=, /=
  - CANNOT OVERLOAD ::, .
  - DON'T TRY =
- These Operators Are "Just" Functions That Use A Different Way Of Listing Their Arguments

Operator Overloading

- Operator Functions Are Typically Defined As `friend` Functions With `const` Parameter Arguments
  - be sure to use the `operator` keyword

```
friend Number operator +(const Number& left,
                         const Number& right);

friend bool operator ==(const Number& left,
                        const Number& right);
```

Operator Overloading

```
class Number {
public:
  ...
  friend Number operator +(const Number& left,
                           const Number& right)
  friend bool operator ==(const Number& left,
                          const Number& right);
  ...
}
```

Operator Overloading

```
Number operator +(const Number& left,
                  const Number& right) {
  Number t=Number( left.value + right.value );
  return( t );
}

bool operator ==(const Number& left,
                 const Number& right) {
  return( left.value == right.value );
}
```

Operator Overloading

```
Number operator +(const Number& left,
                  const Number& right) {
  Number t=Number( left.value + right.value );
  return( t );
}

bool operator ==(const Number& left,
                 const Number& right) {
  return( left.value == right.value );
}
```

operator Modifier
In Both Declaration
& Definition And No
:: Operator

C++ Review

- YUCK!!! Please Don't Use Visual Studio 6.0!
  - A Known Microsoft Bug (C2248) Requires Forward Declarations Of Operator Functions In Header And Implementation Files!
- Operator Overloading Is Cool!
  - binary or unary operators cannot be altered to accept different arguments, just overloaded
- Overloading Can Be Quite Confusing To Class Consumers

Overloading << and >>

- The Insertion And Extraction Operators Can Also Be Overloaded By A Class Definition
- These Operators Must Be Friends

```
friend ostream& operator <<(ostream& outs,
                           const Number& n);

friend istream& operator >>(istream& ins,
                           Number& n);
```

C++ Review

- Overloading << And >> Let A Class' Author Determine How A Class Should Be Dumped To And From A File Stream

Pointers Vs. References

- Pointers And Reference Variables Are Not The Same Thing
- Pointers Hold The Address Of A Variable Of A Specific Type And May Be Null
- References Always Point To An Object And May Never Be Null

`new` Operators

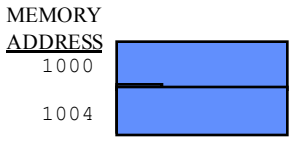
- Rather Assigning To Existing Variables, A Pointer Can Be Attached To Dynamic Variables Using The `new` Operator

```
int* p1;
p1 = new int;
*p1 = 10;
```

`new` Operators

- Rather Assigning To Existing Variables, A Pointer Can Be Attached To Dynamic Variables Using The `new` Operator

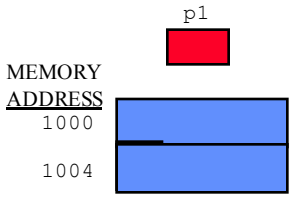
```
int* p1;
p1 = new int;
*p1 = 10;
```



new Operators

- Rather Assigning To Existing Variables, A Pointer Can Be Attached To Dynamic Variables Using The new Operator

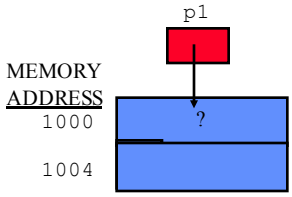
```
int* p1;
p1 = new int;
*p1 = 10;
```



new Operators

- Rather Assigning To Existing Variables, A Pointer Can Be Attached To Dynamic Variables Using The new Operator

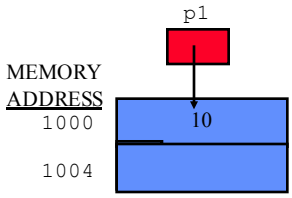
```
int* p1;
p1 = new int;
*p1 = 10;
```



new Operators

- Rather Assigning To Existing Variables, A Pointer Can Be Attached To Dynamic Variables Using The new Operator

```
int* p1;
p1 = new int;
*p1 = 10;
```



new Operators

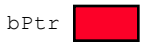
- Pointers Can Work With Any Class Type
- new Operator Makes A Constructor Call;

```
bankAccount*  bPtr;
bPtr = new bankAccount("howie",  10.0);
```

new Operators

- Pointers Can Work With Any Class Type
- new Operator Makes A Constructor Call;

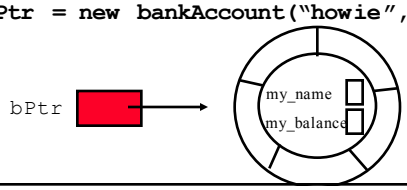
```
bankAccount*  bPtr;
bPtr = new bankAccount("howie",  10.0);
```



new Operators

- Pointers Can Work With Any Class Type
- new Operator Makes A Constructor Call;

```
bankAccount*  bPtr;
bPtr = new bankAccount("howie",  10.0);
```



`delete` Operators

- All Dynamic Variables Must Be delete'd To Recycle Memory Used

```
bankAccount*  bPtr;
bPtr = new bankAccount("howie",  10.0);
cout << (*bPtr).balance();
...
delete bPtr;
```

`delete` Operators

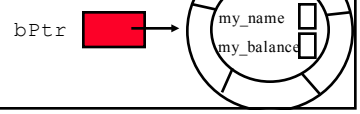
- All Dynamic Variables Must Be delete'd To Recycle Memory Used

```
bankAccount*  bPtr;
bPtr = new bankAccount("howie",  10.0);
cout << (*bPtr).balance();
...
delete bPtr;
```

`delete` Operators

- All Dynamic Variables Must Be delete'd To Recycle Memory Used

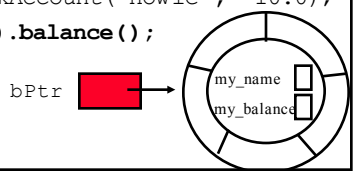
```
bankAccount*  bPtr;
bPtr = new bankAccount("howie",  10.0);
cout << (*bPtr).balance();
...
delete bPtr;
```



`delete` Operators

- All Dynamic Variables Must Be delete'd To Recycle Memory Used

```
bankAccount*  bPtr;
bPtr = new bankAccount("howie", 10.0);
cout << (*bPtr).balance();
...
delete bPtr;
```



`delete` Operators

- All Dynamic Variables Must Be delete'd To Recycle Memory Used

```
bankAccount*  bPtr;
bPtr = new bankAccount("howie", 10.0);
cout << (*bPtr).balance();
...
delete bPtr;
```

Pointer Basics

- A Pointer Must Point To Something Before You Dereference The Pointer
- Once Deleted, You Cannot Dereference The Pointer Anymore

C++ Review

- `typedef` Is A Convenient Aliasing Technique When Working With Pointers
- Pointers Must Point To Something Before They Are Dereferenced
- Once Deleted, Pointers Cannot Be Dereferenced
- The `->` Operator Is A Shorthand For `(*ptr_variable).member`

Constructors And Destructors

- Each Class Type Defines Hooks That Are Invoked When Variables Are Declared
  - overloaded constructors
- Each Class Type Can Also Define A Hook That Gets Invoked When Variables Are Deleted
  - default destructor

Constructors And Destructors

- A Destructor Allows A Class Type Containing Dynamic Variables To Delete Its Managed Memory
  - `~classname();` declared in class definition
  - a public member method
- Like Constructors, Destructors Are Not Programmer-Callable

C++ Review

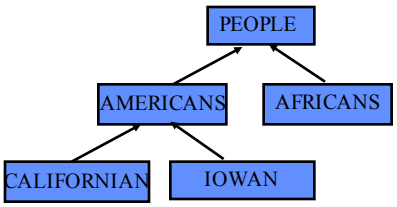
- Destructors Perform Cleanup On Variables That Fall Out Of Scope
  - typically, destructors release dynamic variables

Inheritance

- Often, Classes Are Made From Existing Classes
- Base Class
  - starting point for defining a set of classes
  - most general attributes and methods defined here
- Derived Class
  - extends the definition of a base class in some way

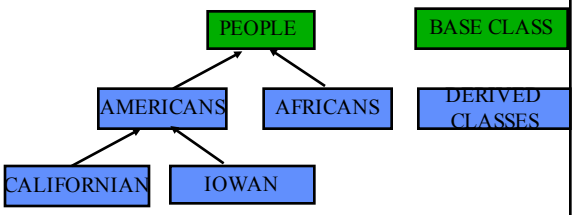
Hierarchy

- A Very Natural Notion



Hierarchy

- A Very Natural Notion



Doors Example

- Let's Design A Set Of Doors Classes For An Adventure Game
- What Are The Common Characteristics Of All Doors?

Door Object

- Knows:
  - Its Status (Open or Shut)
- Can Do:
  - Initialize Itself As Shut
  - Open Itself, If Possible
  - Close Itself
  - Tell Whether Or Not It Is Open

Class Door

- A Generic Base Class

```
class Door {
public:
  Door();
  bool isOpen() const;
  void open();
  void close();
protected:
  bool isShut;
}
```

`protected` Qualifier

- `protected` Is A Compromise Between Private And Public
  - `protected` Is Public To Base Classes
  - `protected` Is Public To Friends Of The Base Classes
  - `protected` Is Public To Derived Classes
  - `protected` Is Public To Friends Of Derived Classes
  - `protected` Is Private To Other Classes

C++ Review

- Inheritance Is An Important Part Of Good OO Design And Implementation

Derived Lockable Door From Door

- A Derived Class

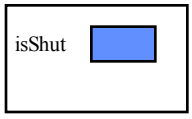
```
class LockableDoor : public Door {
public:
  LockableDoor();
  bool isLocked() const;
  void open();
  void lock();  void unlock();
protected:
  bool thelock;
}
```

Lockable Door Object

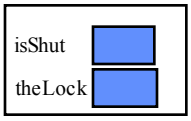
- Member Data
  - `isShut`
  - `theLock`
- Member Functions
  - `isLocked()`
  - `open()`
  - `lock()`
  - `unlock()`
  - `isOpen()`
  - `close()`

Comparing Door And LockableDoor

- Door



- LockableDoor



Inheritance Behavior

- By Default, All Member Functions And Member Attributes Are Inherited Down To Derived Classes
  - happens without mentioning these functions and attributes in the derived class definition
- Any Member Function Or Member Attribute Can Be Redefined In The Derived Class
  - hides access to the base class versions

Example Redefinition

- LockableDoor's open( ) Function

```
void LockableDoor::open( )
{
  if (!isLocked()) {
    Door::open();
  }
}
```

Using The Doors

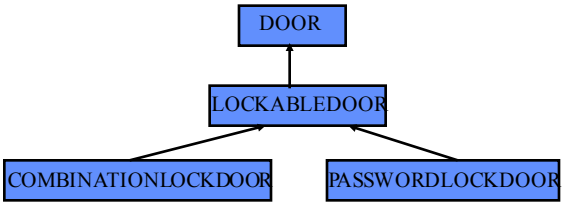
```
Door hallDoor;
LockableDoor frontDoor;

hallDoor.open();
frontDoor.lock();
frontDoor.open();
if (!frontDoor.isOpen())
   frontDoor.unlock();
```

C++ Review

- `protected` Members Are Accessible To Derived Classes
- Using The Scope Operator ::, You Can Specify Just Which Version Of A Function To Call

Derived Classes Can Be A Base Class



CombinationLockDoor

- Combinations Are Single Integers

```
class CombinationLockDoor : public
LockableDoor {
public:
  CombinationLockDoor( int combo = 0);
  void unlock( int combo );
protected:
  int thecombination;
}
```

CombinationLockDoor Object

- Member Data
  `isShut`
  `thelock`
  `thecombination`
- Member Functions
  `isLocked()`
  `open()`
  `lock()`
  `unlock( int )`
  `isOpen()`
  `close()`

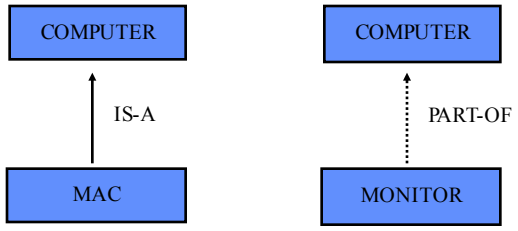
C++ Review

- `protected` Members Are Accessible To Derived Classes
- Redefined Members Hide Access To The Parent Class Versions

Relationships Between Object

- IS-A
  - one class "is a kind of" another class
  - base class is a general class
  - derived class is a specialization of the general concept
- PART-OF
  - one class "is a part of" another class
  - often used to represent compound objects

Relationships Between Objects



Relationships Between Objects

```
class Computer {
}

class Mac:public
          Computer {
}
```

IS-A

```
class Monitor {
}

class Computer {
private:
  Monitor theMonitor;
}
```

PART-OF

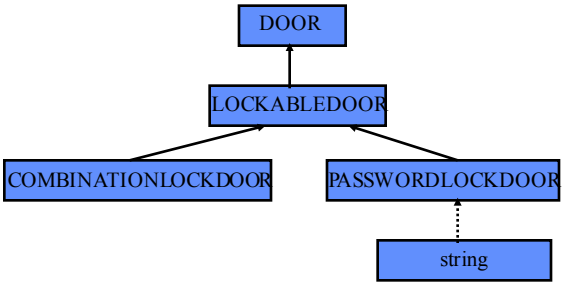
PasswordLockDoor

- Doors That Require A Password To Open Or Close
  - "open sesame" "sounds like fun"
- Let's Represent The Password As `string`
- The Password Is "Part-Of" A PasswordLockDoor

PasswordLockDoor Object

```
class PasswordLockDoor : public
LockableDoor {
public:
  PasswordLockDoor(const char c[]="");
  void unlock( const char c[]="");
protected:
  string thepassword;
}
```

Object Diagram For Doors



PasswordLockDoor Object

- Member Data
  - isShut
  - thelock
  - thepassword
- Member Functions
  - isLocked()
  - open()
  - lock()
  - unlock(char[])
  - isOpen()
  - close()

Pointers To Base Classes

- Pointers Can Be Made To Point To Derived Classes

```
typedef Door* DoorPtr;
DoorPtr p = new Door();
p->open();   // calls Door::open
...
p = new UnlockableDoor();
p->open();   // which open??
```

`virtual` Functions

- Late Binding Allows The Selection Of Which Implementation Of A Member Function To Execute To Be Determined At Run-Time
- C++ Performs Late Binding Via `virtual` Functions

`virtual` Functions

- A Generic Base Class

```
class Door {
public:
  Door();
  bool isOpen() const;
  virtual void open();
  void close();
protected:
  bool isShut;
}
```

C++ Review

- `virtual` Functions Allow For Late Binding To Runtime Objects To Determine Which Version Of A Function Actually Gets Called

Summary

- C++ Review
  - Classes
  - Pointers
  - Templates
  - Inheritance
  - Virtual Functions