IS 0020
Program Design and Software Tools

Polymorphism, Template, Preprocessor

Lecture 6

June 28, 2004

Introduction

- Polymorphism
  - "Program in the general"
  - Derived-class object can be treated as base-class object
    - "is-a" relationship
    - Base class is not a derived class object
  - Virtual functions and dynamic binding
  - Makes programs extensible
    - New classes added easily, can still be processed
- Examples
  - Use abstract base class **`Shape`**
    - Defines common interface (functionality)
    - **`Point`**, **`Circle`** and **`Cylinder`** inherit from **`Shape`**

Invoking Base-Class Functions from Derived-Class Objects

- Pointers to base/derived objects
  - Base pointer aimed at derived object
    - "is a" relationship
      - **`Circle`** "is a" **`Point`**
    - Will invoke base class functions
  - Can cast base-object's address to derived-class pointer
    - Called down-casting
    - Allows derived-class functionality
- Key point
  - Base-pointer can aim at derived-object - but can only call base-class functions
    - Data type of pointer/reference determines functions it can call

Virtual Functions

- **`virtual`** functions
  - Object (not pointer) determines function called
- Why useful?
  - Suppose **`Circle`**, **`Triangle`**, **`Rectangle`** derived from **`Shape`**
    - Each has own **`draw`** function
  - To draw any shape
    - Have base class **`Shape`** pointer, call **`draw`**
    - Program determines proper **`draw`** function at run time (dynamically)
    - Treat all shapes generically

Virtual Functions

- Declare **`draw`** as **`virtual`** in base class
  - Override **`draw`** in each derived class
    - Like redefining, but new function must have same signature
  - If function declared **`virtual`**, can only be overridden
    - **`virtual void draw() const;`**
    - Once declared **`virtual`**, **`virtual`** in all derived classes
      - Good practice to explicitly declare **`virtual`**
- Dynamic binding
  - Choose proper function to call at run time
  - Only occurs off pointer handles
    - If function called from object, uses that object's definition

Virtual Functions

- Polymorphism
  - Same message, “print”, given to many objects
    - All through a base pointer
  - Message takes on “many forms”
- Summary
  - Base-pointer to base-object, derived-pointer to derived
    - Straightforward
  - Base-pointer to derived object
    - Can only call base-class functions
  - Derived-pointer to base-object
    - Compiler error
    - Allowed if explicit cast made

Polymorphism Examples

- Suppose designing video game
  - Base class **`SpaceObject`**
    - Derived **`Martian`**, **`SpaceShip`**, **`LaserBeam`**
    - Base function **`draw`**
  - To refresh screen
    - Screen manager has **`vector`** of base-class pointers to objects
    - Send **`draw`** message to each object
    - Same message has "many forms" of results
  - Easy to add class **`Mercurian`**
    - Inherits from **`SpaceObject`**
    - Provides own definition for **`draw`**
  - Screen manager does not need to change code
    - Calls **`draw`** regardless of object's type
    - **`Mercurian`** objects "plug right in"

Type Fields and switch Structures

- One way to determine object's class
  - Give base class an attribute
    - **`shapeType`** in class **`Shape`**
  - Use **`switch`** to call proper **`print`** function
- Many problems
  - May forget to test for case in **`switch`**
  - If add/remove a class, must update **`switch`** structures
    - Time consuming and error prone
- Better to use polymorphism
  - Less branching logic, simpler programs, less debugging

Abstract Classes

- Abstract classes
  - Sole purpose: to be a base class (called abstract base classes)
  - Incomplete
    - Derived classes fill in "missing pieces"
  - Cannot make objects from abstract class
    - However, can have pointers and references
- Concrete classes
  - Can instantiate objects
  - Implement all functions they define
  - Provide specifics

Abstract Classes

- Abstract classes not required, but helpful
- To make a class abstract
  - Need one or more "pure" virtual functions
    - Declare function with initializer of 0

```
virtual void draw() const = 0;
```

  - Regular virtual functions
    - Have implementations, overriding is optional
  - Pure virtual functions
    - No implementation, must be overridden
  - Abstract classes can have data and concrete functions
    - Required to have one or more pure virtual functions

Case Study: Inheriting Interface and Implementation

- Make abstract base class **`Shape`**
  - Pure virtual functions (must be implemented)
    - **`getName`**, **`print`**
    - Default implementation does not make sense
  - Virtual functions (may be redefined)
    - **`getArea`**, **`getVolume`**
      - Initially return **`0.0`**
    - If not redefined, uses base class definition
  - Derive classes **`Point`**, **`Circle`**, **`Cylinder`**

Case Study: Inheriting Interface and Implementation

	getArea	getVolume	getName	print
Shape	0.0	0.0	= 0	= 0
Point	0.0	0.0	"Point"	[x,y]
Circle	πr^2	0.0	"Circle"	center=[x,y]; radius=r

Cylinder	$2\pi r^2 + 2\pi rh$	$\pi r^2 h$	"Cylinder"	center=[x,y];
				radius=r;
				height=h

Outline

shape.h (1 of 1)

```
// Fig. 10.12: shape.h
// Shape abstract-base-class definition.
#ifndef SHAPE_H
#define SHAPE_H

#include <string>  // C++ standard string class

using std::string;

class Shape {

public:

   // virtual function that returns shape area
   virtual double getArea() const;

   // virtual function that returns shape volume
   virtual double getVolume() const;

   // pure virtual functions; overridden in derived classes
   virtual string getName() const = 0; // return shape name
   virtual void print() const = 0;     // output shape

}; // end class Shape

#endif
```

Virtual and pure virtual functions.

Outline

shape.cpp (1 of 1)

```
// Fig. 10.13: shape.cpp
// Shape class member-function definitions.
#include <iostream>

using std::cout;

#include "shape.h"  // Shape class definition

// return area of shape; 0.0 by default
double getArea() const
{
   return 0.0;

}  // end function getArea

// return volume of shape; 0.0 by default
double getVolume() const
{
   return 0.0;

}  // end function getVolume
```

Polymorphism, Virtual Functions and Dynamic Binding “Under the Hood”

- Polymorphism has overhead
  - Not used in STL (Standard Template Library) to optimize performance
- **`virtual`** function table (vtable)
  - Every class with a **`virtual`** function has a vtable
  - For every **`virtual`** function, vtable has pointer to the proper function
  - If derived class has same function as base class
    - Function pointer aims at base-class function

Virtual Destructors

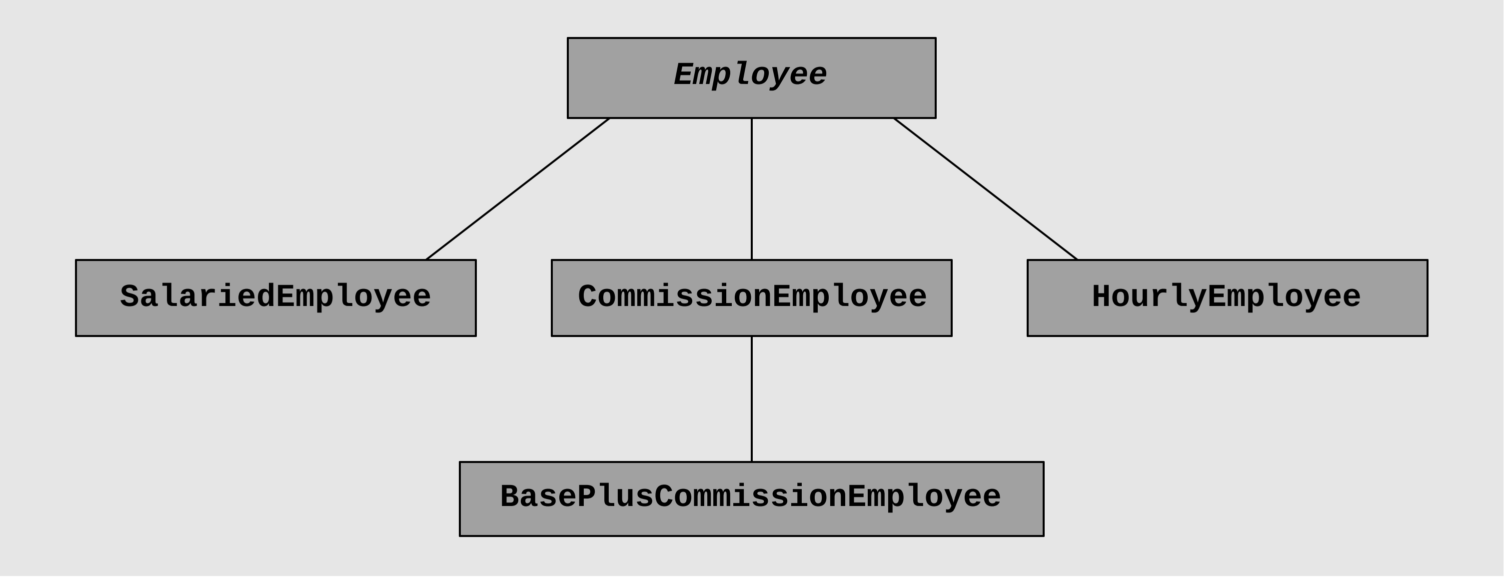
- Base class pointer to derived object
  - If destroyed using **`delete`**, behavior unspecified
- Simple fix
  - Declare base-class destructor virtual
    - Makes derived-class destructors virtual
  - Now, when **`delete`** used appropriate destructor called
- When derived-class object destroyed
  - Derived-class destructor executes first
  - Base-class destructor executes afterwards
- Constructors cannot be virtual

Case Study: Payroll System Using Polymorphism

- Create a payroll program
  - Use virtual functions and polymorphism
- Problem statement
  - 4 types of employees, paid weekly
    - Salaried (fixed salary, no matter the hours)
    - Hourly (overtime [>40 hours] pays time and a half)
    - Commission (paid percentage of sales)
    - Base-plus-commission (base salary + percentage of sales)
      - Boss wants to raise pay by 10%

Payroll System Using Polymorphism

- Base class **`Employee`**
  - Pure virtual function **`earnings`** (returns pay)
    - Pure virtual because need to know employee type
    - Cannot calculate for generic employee
  - Other classes derive from **`Employee`**



Dynamic Cast

- Downcasting
  - **dynamic_cast** operator
    - Determine object's type at runtime
    - Returns 0 if not of proper type (cannot be cast)

```
NewClass *ptr = dynamic_cast < NewClass *> objectPtr;
```

- Keyword **typeid**
  - Header **<typeinfo>**
  - Usage: **typeid(object)**
    - Returns **type_info** object
    - Has information about type of operand, including name
    - **typeid(object).name()**

IS 0020
Program Design and Software Tools

Templates

Introduction

- Overloaded functions
  - Similar operations but Different types of data
- Function templates
  - Specify entire range of related (overloaded) functions
  - Function-template specializations
  - Identical operations
    - Different types of data
  - Single function template
    - Compiler generates separate object-code functions
  - Unlike Macros they allow Type checking
- Class templates
  - Specify entire range of related classes
    - Class-template specializations

Function Templates

- Function-template definitions
  - Keyword **`template`**
  - List formal type parameters in angle brackets (**`<`** and **`>`**)
    - Each parameter preceded by keyword **`class`** or **`typename`**
      - **`class`** and **`typename`** interchangeable

```
template< class T >
template< typename ElementType >
template< class BorderType, class FillType >
```

    - Specify types of
      - Arguments to function
      - Return type of function
      - Variables within function

fig11_01.cpp
(1 of 2)

```
// Fig. 11.1: fig11_01.cpp
// Using template functions.
#include <iostream>

using std::cout;
using std::endl;

// function template printArray defi
template< class T >
void printArray( const T *array, const int count )
{
   for ( int i = 0; i < co
      cout << array[ i ] <

   cout << endl;

} // end function print

int main()
{
   const int aCount = 5;
   const int bCount = 7;
   const int cCount = 6;

```

Function template definition; declare single formal type parameter **T**.

T is type parameter; use any valid identifier.

If **T** is user-defined type, stream-insertion operator must be overloaded for class **T**.

fig11_01.cpp
(2 of 2)

```
   int a[ aCount ] = { 1, 2, 3, 4, 5 };
   double b[ bCount ] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
   char c[ cCount ] = "HELLO";  // 6th position for null

   cout << "Array a contains:" << endl;

   // call integer function-template specialization
   printArray( a, aCount );

   cout << "Array b contains

   // call double funct
   printArray( b, bCoun

   cout << "Array c con

   // call character fu
   printArray( c, cCoun

   return 0;

} // end main
```

Crea... pecialization for printing array...

```
void ... , const int count )
{
   f ... + )

   c
} // end function printArray
```

Compiler infers **T** is **double**; instantiates function-template specialization where **T** is **double**.

Compiler infers **T** is **char**; instantiates function-template specialization where **T** is **char**.

Overloading Function Templates

- Related function-template specializations
  - Same name
    - Compiler uses overloading resolution
- Function template overloading
  - Other function templates with same name
    - Different parameters
  - Non-template functions with same name
    - Different function arguments
  - Compiler performs matching process
    - Tries to find precise match of function name and argument types
    - If fails, function template
      - Generate function-template specialization with precise match

Class Templates

- Stack
  - LIFO (last-in-first-out) structure
- Class templates
  - Generic programming
  - Describe notion of stack generically
    - Instantiate type-specific version
  - Parameterized types
    - Require one or more type parameters
      - Customize “generic class” template to form class-template specialization

```
// Fig. 11.2: tstack1.h
// Stack class template.
#ifndef TSTACK1_H
#define TSTACK1_H

template< class T >
class Stack {

public:
   Stack( int = 10 );  // default constructor (stack size 10)

   // destructor
   ~Stack()
   {
      delete [] stackPtr;

   } // end ~Stack destructor

   bool push( const T& );  // push an element onto the stack
   bool pop( T& );         // pop an element off the stack

```

Specify class-template definition; type parameter **T** indicates type of **Stack** class to be created.

Function parameters of type **T**.

Outline

tstack1.h (2 of 4)

```
   // determine whether Stack is empty
   bool isEmpty() const
   {
      return top == -1;

   } // end function isEmpty

   // determine whether Stack is full
   bool isFull() const
   {
      return top == size - 1;

   } // end function isFull

private:
   int size;     // # of elements in the stack
   int top;      // location of the top element
   T *stackPtr;  // pointer to the stack

}; // end class Stack

```

Array of elements of type **T**.

```
// constructor
template< class T >
Stack< T >::Stack( int s )
{
   size = s > 0 ? s : 10;
   top = -1;  // Stack initially empt
   stackPtr = new T[ size ]; // alloc

} // end Stack constructor

// push element onto stack;
// if successful, return true; ot
template< class T >
bool Stack< T >::push( const T &p
{
   if ( !isFull() ) {
      stackPtr[ ++top ] = pushValue;  // place item on Stack
      return true;  // push successful

   } // end if

   return false;  // push unsuccessful

} // end function push

```

Constructor creates array of type **`T`**. For example, compiler generates

`stackPtr = new T[ size ];`

for class-template specialization **`Stack< double >`**.

Use binary scope resolution operator (**`::`**) with class-template name (**`Stack< T >`**) to tie definition to class template's scope.

`T >`

```
// pop element off stack;
// if successful, return true; otherwise, return false
template< class T >
bool Stack< T >::pop( T &popValue )
{
   if ( !isEmpty() ) {
      popValue = stackPtr[ top-- ];  // r
      return true;  // pop successful

   } // end if

   return false;  // pop unsuccessful

} // end function pop

#endif
```

Member function preceded with header

Use binary scope resolution operator (`::`) with class-template name (`Stack< T >`) to tie definition to class template's scope.

```cpp
// Fig. 11.3: fig11_03.cpp
// Stack-class-template test program.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

#include "tstack1.h"  // Stack class template definition

int main()
{
   Stack< double > doubleStack( 5 );
   double doubleValue = 1.1;

   cout << "Pushing elements onto doubleSt

   while ( doubleStack.push( doubleValue ) ) {
      cout << doubleValue << ' ';
      doubleValue += 1.1;

   } // end while

   cout << "\nStack is full. Cannot push " << doubleValue
        << "\n\nPopping elements from doubleStack\n";
```

Link to class template definition.

Instantiate object of class **`Stack< double >`**.

Invoke function **`push`** of class-template specialization **`Stack< double >`**.

```
   while ( doubleStack.pop( doubleValue ) )
      cout << doubleValue << ' ';

   cout << "\nStack is empty. Cannot pop\n

   Stack< int > intStack;
   int intValue = 1;
   cout << "\nPushing elements onto intStack\n";

   while ( intStack.push( intValue ) ) {
      cout << intValue << ' ';
      ++intValue;

   } // end while

   cout << "\nStack is full. Cannot push " << intValue
        << "\n\nPopping elements from intStack\n";

   while ( intStack.pop( intValue ) )
      cout << intValue << ' ';

   cout << "\nStack is empty. Cannot pop\n";

   return 0;
```

Invoke function **pop** of class-template specialization **Stack< double >**.

Note similarity of code for **Stack< int >** to code for **Stack< double >**.

```

} // end main
```

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

fig11_03.cpp
(3 of 3)

fig11_03.cpp
output (1 of 1)

Outline

fig11_04.cpp
(1 of 2)

```
// Fig. 11.4: fig11_04.cpp
// Stack class template test program. Function main uses a
// function template to manipulate objects of type Stack< T >.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

#include "tstack1.h"  // Stack class template definition

// function template to manipulate Stack< T >
template< class T >
void testStack(
   Stack< T > &theStack,   // reference to Stack< T >
   T value,                // initial value to push
   T increment,            // increment for subsequent values
   const char *stackName ) // name of the Stack < T > object
{
   cout << "\nPushing elements onto " << stackName << '\n';

   while ( theStack.push( value ) ) {
      cout << value << ' ';
      value += increment;

   } // end while
```

Function template to manipulate **`Stack< T >`** eliminates similar code from previous file for **`Stack< double >`** and **`Stack< int >`**.

Outline

fig11_04.cpp
(2 of 2)

```

   cout << "\nStack is full. Cannot push " << value
        << "\n\nPopping elements from " << stackName << '\n';

   while ( theStack.pop( value ) )
      cout << value << ' ';

   cout << "\nStack is empty. Cannot pop\n";

} // end function testStack

int main()
{
   Stack< double > doubleStack( 5 );
   Stack< int > intStack;

   testStack( doubleStack, 1.1, 1.1, "doubleStack" );
   testStack( intStack, 1, 1, "intStack" );

   return 0;

} // end main
```

fig11_04.cpp
output (1 of 1)

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

Note output identical to that of **fig11_03.cpp**.

Class Templates and Nontype Parameters

- Class templates
  - Nontype parameters
    - Default arguments
    - Treated as **`consts`**

```
template< class T, int elements >
Stack< double, 100 > mostRecentSalesFigures;
```

      - Declares object of type **`Stack< double, 100>`**
  - Type parameter
    - Default type example: **`template< class T = string >`**
- Overriding class templates
  - Class for specific type
    - Does not match common class template
  - Example:

```
template<>
Class Array< Martian > {
      // body of class definition
};
```

Templates and Inheritance

- Several ways of relating templates and inheritance
  - Class template derived from class-template specialization
  - Class template derived from non-template class
  - Class-template specialization derived from class-template specialization
  - Non-template class derived from class-template specialization
- Friendships between class template and
  - Global function
  - Member function of another class
  - Entire class

Templates and Friends

- **friend** functions
  - Inside definition of **template< class T > class X**
    - **friend void f1();**
      - **f1() friend** of all class-template specializations
    - **friend void f2(X< T > &);**
      - **f2(X< float > &) friend** of **X< float >** only,
        f2(X< double > &) friend of **X< double >** only,
        f2(X< int > &) friend of **X< int >** only,
        …
    - **friend void A::f4();**
      - Member function **f4** of class **A friend** of all class-template specializations

Templates and Friends

- **friend** functions
  - Inside definition of **template< class T > class X**
    - **friend void C< T >::f5(X< T > &);**
      - Member function **C<float>::f5(X< float> &)** **friend** of **class X<float>** only
- **friend** classes
  - Inside definition of **template< class T > class X**
    - **friend class Y;**
      - Every member function of **Y** friend of every class-template specialization
    - **friend class Z<T>;**
      - **class Z<float>** **friend** of class-template specialization **X<float>,** etc.

Templates and static Members

- Non-template class
  - **`static`** data members shared between all objects
- Class-template specialization
  - Each has own copy of **`static`** data members
  - **`static`** variables initialized at file scope
  - Each has own copy of **`static`** member functions

Introduction

- Preprocessing
  - Occurs before program compiled
    - Inclusion of external files
    - Definition of symbolic constants
    - Macros
    - Conditional compilation
    - Conditional execution
  - All directives begin with `#`
    - Can only have whitespace before directives
  - Directives not C++ statements
    - Do not end with `;`

The #include Preprocessor Directive

- **`#include`** directive
  - Puts copy of file in place of directive
  - Two forms
    - **`#include <filename>`**
      - For standard library header files
      - Searches pre-designated directories
    - **`#include "filename"`**
      - Searches in current directory
      - Normally used for programmer-defined files
- Usage
  - Loading header files
    - **`#include <iostream>`**
  - Programs with multiple source files
  - Header file
    - Has common declarations and definitions
    - Classes, structures, enumerations, function prototypes
    - Extract commonality of multiple program files

The #define Preprocessor Directive: Symbolic Constants

- **`#define`**
  - Symbolic constants
    - Constants represented as symbols
    - When program compiled, all occurrences replaced
  - Format
    - **`#define`** ***`identifier replacement-text`***
    - **`#define PI 3.14159`**
  - Everything to right of identifier replaces text
    - **`#define PI=3.14159`**
    - Replaces **`PI`** with **`"=3.14159"`**
    - Probably an error
  - Cannot redefine symbolic constants
- Advantage: Takes no memory
- Disadvantages
  - Name not be seen by debugger (only replacement text)
  - Do not have specific data type
- **`const`** variables preferred

The #define Preprocessor Directive: Macros

- Macro
  - Operation specified in **`#define`**
  - Intended for legacy C programs
  - Macro without arguments
    - Treated like a symbolic constant
  - Macro with arguments
    - Arguments substituted for replacement text
    - Macro expanded
  - Performs a text substitution
    - No data type checking

The #define Preprocessor Directive: Macros

- Example

```
#define CIRCLE_AREA( x ) ( PI * ( x ) * ( x ) )
area = CIRCLE_AREA( 4 );
```

becomes

```
area = ( 3.14159 * ( 4 ) * ( 4 ) );
```

- Use parentheses
  - Without them,

```
#define CIRCLE_AREA( x )  PI * x * x
area = CIRCLE_AREA( c + 2 );
```

becomes

```
area = 3.14159 * c + 2 * c + 2;
```

which evaluates incorrectly

The #define Preprocessor Directive: Macros

- Multiple arguments

```
#define RECTANGLE_AREA( x, y )  ( ( x ) * ( y ) )
rectArea = RECTANGLE_AREA( a + 4, b + 7 );
```

becomes

```
rectArea = ( ( a + 4 ) * ( b + 7 ) );
```

- **#undef**
  - Undefines symbolic constant or macro
  - Can later be redefined

Conditional Compilation

- Control preprocessor directives and compilation
  - Cannot evaluate cast expressions, **`sizeof`**, enumeration constants
- Structure similar to **`if`**

```
#if !defined( NULL )
   #define NULL 0
#endif
```

  - Determines if symbolic constant **`NULL`** defined
  - If **`NULL`** defined,
    - **`defined( NULL )`** evaluates to **`1`**
    - **`#define`** statement skipped
  - Otherwise
    - **`#define`** statement used
  - Every **`#if`** ends with **`#endif`**

Conditional Compilation

- Can use else
  - **`#else`**
  - **`#elif`** is "else if"
- Abbreviations
  - **`#ifdef`** short for
    - **`#if defined(name)`**
  - **`#ifndef`** short for
    - **`#if !defined(name)`**
- "Comment out" code
  - Cannot use **`/* ... */`** with C-style comments
    - Cannot nest **`/* */`**
  - Instead, use

```
#if 0
  code commented out
#endif
```

  - To enable code, change **`0`** to **`1`**

Conditional Compilation

- Debugging

```
#define DEBUG 1
#ifdef DEBUG
  cerr << "Variable x = " << x << endl;
#endif
```

  - Defining **DEBUG** enables code
  - After code corrected
    - Remove **#define** statement
    - Debugging statements are now ignored

The #error and #pragma Preprocessor Directives

- **`#error` *`tokens`***
  - Prints implementation-dependent message
  - Tokens are groups of characters separated by spaces
    - **`#error 1 - Out of range error`** has 6 tokens
  - Compilation may stop (depends on compiler)
- **`#pragma` *`tokens`***
  - Actions depend on compiler
  - May use compiler-specific options
  - Unrecognized **`#pragma`**s are ignored

The # and ## Operators

- **#** operator
  - Replacement text token converted to string with quotes

```
#define HELLO( x ) cout << "Hello, " #x << endl;
```

  - **HELLO(JOHN)** becomes
    - **cout << "Hello, " "John" << endl;**
    - Same as **cout << "Hello, John" << endl;**
- **##** operator
  - Concatenates two tokens

```
#define TOKENCONCAT( x, y )  x ## y
```

  - **TOKENCONCAT(O, K)** becomes
    - **OK**

Line Numbers

- **`#line`**
  - Renumbers subsequent code lines, starting with integer
    - **`#line 100`**
  - File name can be included
  - **`#line 100 "file1.cpp"`**
    - Next source code line is numbered **`100`**
    - For error purposes, file name is **`"file1.cpp"`**
    - Can make syntax errors more meaningful
    - Line numbers do not appear in source file

Predefined Symbolic Constants

- Five predefined symbolic constants
  - Cannot be used in **#define** or **#undef**

Symbolic constant	Description
__LINE__	The line number of the current source code line (an integer constant).
__FILE__	The presumed name of the source file (a string).
__DATE__	The date the source file is compiled (a string of the form **"Mmm dd yyyy"** such as **"Jan 19 2001"**).
__TIME__	The time the source file is compiled (a string literal of the form **"hh:mm:ss"**).

Assertions

- **`assert`** is a macro
  - Header **`<cassert>`**
  - Tests value of an expression
    - If **`0`** (**`false`**) prints error message, calls **`abort`**
      - Terminates program, prints line number and file
      - Good for checking for illegal values
    - If **`1`** (**`true`**), program continues as normal
  - **`assert( x <= 10 );`**
- To remove **`assert`** statements
  - No need to delete them manually
  - **`#define NDEBUG`**
    - All subsequent **`assert`** statements ignored