EEE110 - Computer Programming
Week 12: Inheritance, Polymorphism, Virtual Functions, Exceptions, Templates, and the STL

ADANA ALPARSLAN TÜRKEŞ
SCIENCE AND TECHNOLOGY UNIVERSITY

Dr Kasım Zor

Department of Electrical and Electronic Engineering

Spring 2020

Chapter 15:

Inheritance, Polymorphism, and Virtual Functions

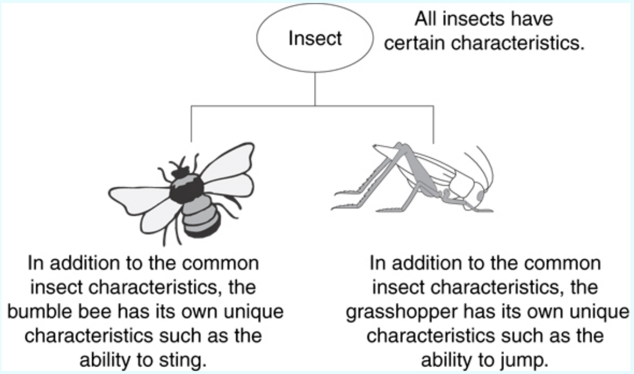
15.1

- What Is Inheritance?

What Is Inheritance?

- Provides a way to create a new class from an existing class
- The new class is a specialized version of the existing class

Example: Insects



The "is a" Relationship

- Inheritance establishes an "is a" relationship between classes.
  - A poodle is a dog
  - A car is a vehicle
  - A flower is a plant
  - A football player is an athlete

Inheritance – Terminology and Notation

- <u>Base</u> class (or parent) – inherited from
- <u>Derived</u> class (or child) – inherits from the base class
- Notation:

```
class Student              // base class
{
     . . .
};
class UnderGrad : public student
{                                    // derived class
     . . .
};
```


Back to the ‘is a’ Relationship

- An object of a derived class 'is a(n)' object of the base class
- Example:
  - an `UnderGrad` is a `Student`
  - a `Mammal` is an `Animal`
- A derived object has all of the characteristics of the base class

What Does a Child Have?

An object of the derived class has:
- all members defined in child class
- all members declared in parent class

An object of the derived class can use:
- all `public` members defined in child class
- all `public` members defined in parent class

15.2

- Protected Members and Class Access

Protected Members and Class Access

- `protected` member access specification: like `private`, but accessible by objects of derived class
- Class access specification: determines how `private`, `protected`, and `public` members of base class are inherited by the derived class

Class Access Specifiers

1) `public` – object of derived class can be treated as object of base class (not vice-versa)
2) `protected` – more restrictive than `public`, but allows derived classes to know details of parents
3) `private` – prevents objects of derived class from being treated as objects of base class.

Inheritance vs. Access

Base class members		How inherited base class members appear in derived class

`private: x`
`protected: y`
`public: z`
	`private`
	base class
		`x` is inaccessible
		`private: y`
		`private: z`

`private: x`
`protected: y`
`public: z`
	`protected`
	base class
		`x` is inaccessible
		`protected: y`
		`protected: z`

`private: x`
`protected: y`
`public: z`
	`public`
	base class
		`x` is inaccessible
		`protected: y`
		`public: z`

More Inheritance vs. Access

class Grade

```
private members:
   char letter;
   float score;
   void calcGrade();
public members:
   void setScore(float);
   float getScore();
   char getLetter();
```

class Test : public Grade

```
private members:
   int numQuestions;
   float pointsEach;
   int numMissed;
public members:
   Test(int, int);
```

When `Test` class inherits from `Grade` class using `public` class access, it looks like this:

```
private members:
   int numQuestions:
   float pointsEach;
   int numMissed;
public members:
   Test(int, int);
   void setScore(float);
   float getScore();
   float getLetter();
```


More Inheritance vs. Access (2)

class Grade

```
private members:
   char letter;
   float score;
   void calcGrade();
public members:
   void setScore(float);
   float getScore();
   char getLetter();
```

class Test : protected Grade

```
private members:
   int numQuestions;
   float pointsEach;
   int numMissed;
public members:
   Test(int, int);
```

When `Test` class inherits from `Grade` class using `protected` class access, it looks like this:

```
private members:
   int numQuestions:
   float pointsEach;
   int numMissed;
public members:
   Test(int, int);
protected members:
   void setScore(float);
   float getScore();
   float getLetter();
```


More Inheritance vs. Access (3)

class Grade

```
private members:
   char letter;
   float score;
   void calcGrade();
public members:
   void setScore(float);
   float getScore();
   char getLetter();
```

class Test : private Grade

```
private members:
   int numQuestions;
   float pointsEach;
   int numMissed;
public members:
   Test(int, int);
```

When `Test` class inherits from `Grade` class using `private` class access, it looks like this:

```
private members:
   int numQuestions:
   float pointsEach;
   int numMissed;
   void setScore(float);
   float getScore();
   float getLetter();
public members:
   Test(int, int);
```

15.3

- Constructors and Destructors in Base and Derived Classes

Constructors and Destructors in Base and Derived Classes

- Derived classes can have their own constructors and destructors
- When an object of a derived class is created, the base class's constructor is executed first, followed by the derived class's constructor
- When an object of a derived class is destroyed, its destructor is called first, then that of the base class

Constructors and Destructors in Base and Derived Classes

Program 15-4

```
// This program demonstrates the order in which base and
// derived class constructors and destructors are called.
#include <iostream>
using namespace std;

//*********************************
// BaseClass declaration          *
//*********************************

```


Program 15-4 *(continued)*

```
class BaseClass
{
public:
   BaseClass()  // Constructor
      { cout << "This is the BaseClass constructor.\n"; }

   ~BaseClass() // Destructor
      { cout << "This is the BaseClass destructor.\n"; }
};

//*********************************
// DerivedClass declaration       *
//*********************************

class DerivedClass : public BaseClass
{
public:
   DerivedClass()  // Constructor
      { cout << "This is the DerivedClass constructor.\n"; }

   ~DerivedClass()  // Destructor
      { cout << "This is the DerivedClass destructor.\n"; }
};

```

Program 5-14 (Continued)

```
//*********************************
// main function                  *
//*********************************

int main()
{
   cout << "We will now define a DerivedClass object.\n";

   DerivedClass object;

   cout << "The program is now going to end.\n";
   return 0;
}
```

Program Output

```
We will now define a DerivedClass object.
This is the BaseClass constructor.
This is the DerivedClass constructor.
The program is now going to end.
This is the DerivedClass destructor.
This is the BaseClass destructor.
```

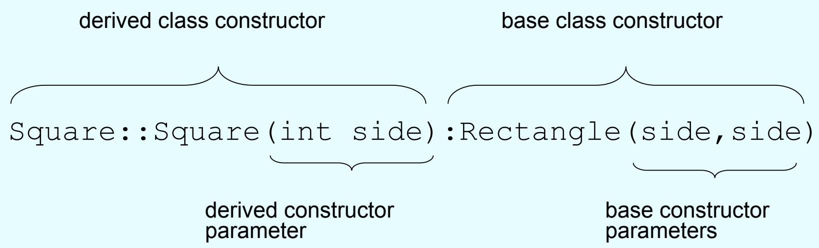

Passing Arguments to Base Class Constructor

- Allows selection between multiple base class constructors
- Specify arguments to base constructor on derived constructor heading:

```
Square::Square(int side) :
      Rectangle(side, side)
```

- Can also be done with inline constructors
- Must be done if base class has no default constructor

Passing Arguments to Base Class Constructor



15.4

- Redefining Base Class Functions

Redefining Base Class Functions

- Redefining function: function in a derived class that has the *same name and parameter list* as a function in the base class
- Typically used to replace a function in base class with different actions in derived class

Redefining Base Class Functions

- Not the same as overloading – with overloading, parameter lists must be different
- Objects of base class use base class version of function; objects of derived class use derived class version of function

Base Class

```
class GradedActivity
{
protected:
   char letter;            // To hold the letter grade
   double score;           // To hold the numeric score
   void determineGrade();  // Determines the letter grade
public:
   // Default constructor
   GradedActivity()
      { letter = ' '; score = 0.0; }

   // Mutator function
   void setScore(double s)
      { score = s;
        determineGrade();}

   // Accessor functions
   double getScore() const
      { return score; }

   char getLetterGrade() const
      { return letter; }
};
```

Note `setScore` function

Derived Class

```
#ifndef CURVEDACTIVITY_H
#define CURVEDACTIVITY_H
#include "GradedActivity.h"

class CurvedActivity : public GradedActivity
{
protected:
   double rawScore;     // Unadjusted score
   double percentage;   // Curve percentage
public:
   // Default constructor
   CurvedActivity() : GradedActivity()
      { rawScore = 0.0; percentage = 0.0; }

   // Mutator functions
   void setScore(double s)
      { rawScore = s;
        GradedActivity::setScore(rawScore * percentage); }

   void setPercentage(double c)
      { percentage = c; }

   // Accessor functions
   double getPercentage() const
      { return percentage; }

   double getRawScore() const
      { return rawScore; }
};
#endif
```

Redefined `setScore` function

From Program 15-7

```
   // Define a CurvedActivity object.
   CurvedActivity exam;

   // Get the unadjusted score.
   cout << "Enter the student's raw numeric score: ";
   cin >> numericScore;

   // Get the curve percentage.
   cout << "Enter the curve percentage for this student: ";
   cin >> percentage;

   // Send the values to the exam object.
   exam.setPercentage(percentage);
   exam.setScore(numericScore);

   // Display the grade data.
   cout << fixed << setprecision(2);
   cout << "The raw score is "
        << exam.getRawScore() << endl;
   cout << "The curved score is "
        << exam.getScore() << endl;
   cout << "The curved grade is "
        << exam.getLetterGrade() << endl;
```

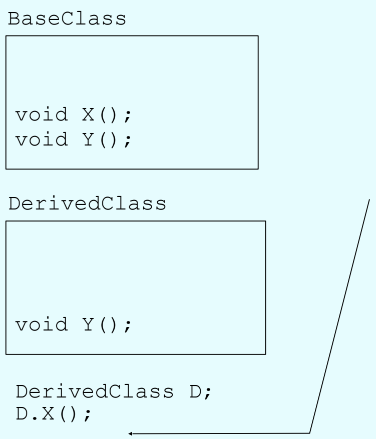
Program Output with Example Input Shown in Bold

```
Enter the student's raw numeric score: 87 [Enter]
Enter the curve percentage for this student: 1.06 [Enter]
The raw score is 87.00
The curved score is 92.22
The curved grade is A
```


Problem with Redefining

- Consider this situation:
  - Class `BaseClass` defines functions `x()` and `y()`. `x()` calls `y()`.
  - Class `DerivedClass` inherits from `BaseClass` and redefines function `y()`.
  - An object `D` of class `DerivedClass` is created and function `x()` is called.
  - When `x()` is called, which `y()` is used, the one defined in `BaseClass` or the the redefined one in `DerivedClass`?

Problem with Redefining



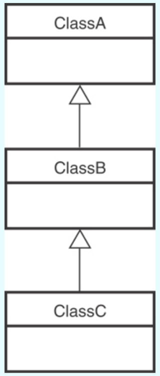
Object `D` invokes function `X()` In `BaseClass`. Function `X()` invokes function `Y()` in `BaseClass`, not function `Y()` in `DerivedClass`, because function calls are bound at compile time. This is <u>static binding.</u>

15.5

- Class Hierarchies

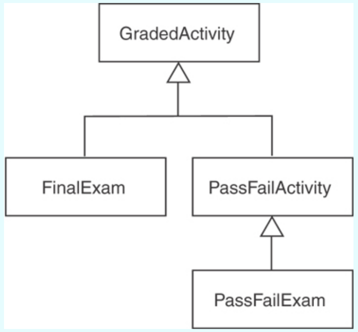
Class Hierarchies

- A base class can be derived from another base class.



Class Hierarchies

- Consider the GradedActivity, FinalExam, PassFailActivity, PassFailExam hierarchy in Chapter 15.



15.6

- Polymorphism and Virtual Member Functions

Polymorphism and Virtual Member Functions

- Virtual member function: function in base class that expects to be redefined in derived class
- Function defined with key word `virtual`:
  `virtual void Y() {...}`
- Supports dynamic binding: functions bound at run time to function that they call
- Without virtual member functions, C++ uses static (compile time) binding

Consider this function (from Program 15-9)

```
void displayGrade(const GradedActivity &activity)
{
   cout << setprecision(1) << fixed;
   cout << "The activity's numeric score is "
        << activity.getScore() << endl;
   cout << "The activity's letter grade is "
        << activity.getLetterGrade() << endl;
}
```

Because the parameter in the `displayGrade` function is a GradedActivity reference variable, it can reference any object that is derived from GradedActivity. That means we can pass a GradedActivity object, a FinalExam object, a PassFailExam object, or any other object that is derived from GradedActivity.

A problem occurs in Program 15-10 however...

Program 15-10

```
#include <iostream>
#include <iomanip>
#include "PassFailActivity.h"
using namespace std;

// Function prototype
void displayGrade(const GradedActivity &);

int main()
{
   // Create a PassFailActivity object. Minimum passing
   // score is 70.
   PassFailActivity test(70);

   // Set the score to 72.
   test.setScore(72);

   // Display the object's grade data. The letter grade
   // should be 'P'. What will be displayed?
   displayGrade(test);
   return 0;
}
```



```

//***************************************************************
// The displayGrade function displays a GradedActivity object's *
// numeric score and letter grade.                              *
//***************************************************************

void displayGrade(const GradedActivity &activity)
{
   cout << setprecision(1) << fixed;
   cout << "The activity's numeric score is "
        << activity.getScore() << endl;
   cout << "The activity's letter grade is "
        << activity.getLetterGrade() << endl;
}
```

Program Output

```
The activity's numeric score is 72.0
The activity's letter grade is C
```

As you can see from the example output, the `getLetterGrade` member function returned 'C' instead of 'P'. This is because the GradedActivity class's `getLetterGrade` function was executed instead of the PassFailActivity class's version of the function.

Static Binding

- Program 15-10 displays 'C' instead of 'P' because the call to the `getLetterGrade` function is statically bound (at compile time) with the GradedActivity class's version of the function.
- We can remedy this by making the function *virtual*.

Virtual Functions

- A virtual function is dynamically bound to calls at runtime.
- At runtime, C++ determines the type of object making the call, and binds the function to the appropriate version of the function.

Virtual Functions

- To make a function virtual, place the virtual key word before the return type in the base class's declaration:

```
virtual char getLetterGrade() const;
```

- The compiler will not bind the function to calls. Instead, the program will bind them at runtime.

Updated Version of GradedActivity

```
class GradedActivity
{
protected:
   double score;   // To hold the numeric score
public:
   // Default constructor
   GradedActivity()
      { score = 0.0; }

   // Constructor
   GradedActivity(double s)
      { score = s; }

   // Mutator function
   void setScore(double s)
      { score = s; }

   // Accessor functions
   double getScore() const
      { return score; }

   virtual char getLetterGrade() const;
};
```

The function is now virtual.

The function also becomes virtual in all derived classes automatically!

If we recompile our program with the updated versions of the classes, we will get the right output, shown here: (See Program 15-11 in the book.)

Program Output

```
The activity's numeric score is 72.0
The activity's letter grade is P
```

This type of behavior is known as polymorphism. The term *polymorphism* means the ability to take many forms.

Program 15-12 demonstrates polymorphism by passing objects of the GradedActivity and PassFailExam classes to the displayGrade function.

Program 15-12

```
#include <iostream>
#include <iomanip>
#include "PassFailExam.h"
using namespace std;

// Function prototype
void displayGrade(const GradedActivity &);

int main()
{
   // Create a GradedActivity object. The score is 88.
   GradedActivity test1(88.0);

   // Create a PassFailExam object. There are 100 questions,
   // the student missed 25 of them, and the minimum passing
   // score is 70.
   PassFailExam test2(100, 25, 70.0);

   // Display the grade data for both objects.
   cout << "Test 1:\n";
   displayGrade(test1);     // GradedActivity object
   cout << "\nTest 2:\n";
```



```
   displayGrade(test2);     // PassFailExam object
   return 0;
}

//****************************************************************
// The displayGrade function displays a GradedActivity object's *
// numeric score and letter grade.                              *
//****************************************************************

void displayGrade(const GradedActivity &activity)
{
   cout << setprecision(1) << fixed;
   cout << "The activity's numeric score is "
        << activity.getScore() << endl;
   cout << "The activity's letter grade is "
        << activity.getLetterGrade() << endl;
}
```

Program Output

```
Test 1:
The activity's numeric score is 88.0
The activity's letter grade is B

Test 2:
The activity's numeric score is 75.0
The activity's letter grade is P
```


Polymorphism Requires References or Pointers

- Polymorphic behavior is only possible when an object is referenced by a reference variable or a pointer, as demonstrated in the `displayGrade` function.

Base Class Pointers

- Can define a pointer to a *base* class object
- Can assign it the address of a *derived* class object

```
GradedActivity *exam = new PassFailExam(100, 25, 70.0);
```

```
cout << exam->getScore() << endl;
cout << exam->getLetterGrade() << endl;
```

Base Class Pointers

- Base class pointers and references only know about members of the base class
  - So, you can't use a base class pointer to call a derived class function
- Redefined functions in *derived* class will be ignored unless *base* class declares the function `virtual`

Redefining vs. Overriding

- In C++, redefined functions are statically bound and overridden functions are dynamically bound.
- So, a virtual function is overridden, and a non-virtual function is redefined.

Virtual Destructors

- It's a good idea to make destructors virtual if the class could ever become a base class.
- Otherwise, the compiler will perform static binding on the destructor if the class ever is derived from.
- See Program 15-14 for an example

15.7

- Abstract Base Classes and Pure Virtual Functions

Abstract Base Classes and Pure Virtual Functions

- Pure virtual function: a virtual member function that must be overridden in a derived class that has objects
- Abstract base class contains at least one pure virtual function:

```
virtual void Y() = 0;
```

- The `= 0` indicates a pure virtual function
- Must have no function definition in the base class

Abstract Base Classes and Pure Virtual Functions

- Abstract base class: class that can have no objects. Serves as a basis for derived classes that may/will have objects
- A class becomes an abstract base class when one or more of its member functions is a pure virtual function

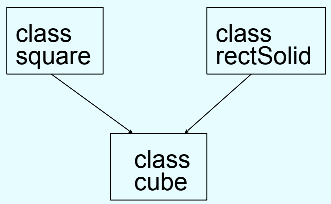
15.8

- Multiple Inheritance

Multiple Inheritance

- A derived class can have more than one base class
- Each base class can have its own access specification in derived class's definition:

```
class cube : public square,
             public rectSolid;
```



Multiple Inheritance

- Arguments can be passed to both base classes' constructors:

```
 cube::cube(int side) :
square(side),
             rectSolid(side, side,
side);
```

- Base class constructors are called in order given in class declaration, not in order used in class constructor

Multiple Inheritance

- Problem: what if base classes have member variables/functions with the same name?
- Solutions:
  - Derived class redefines the multiply-defined function
  - Derived class invokes member function in a particular base class using scope resolution operator `::`
- Compiler errors occur if derived class uses base class function without one of these solutions

Chapter 16:

Exceptions, Templates, and the Standard Template Library (STL)

6.1

- Exceptions

Exceptions

- Indicate that something unexpected has occurred or been detected
- Allow program to deal with the problem in a controlled manner
- Can be as simple or complex as program design requires

Exceptions - Terminology

- Exception: object or value that signals an error
- Throw an exception: send a signal that an error has occurred
- Catch/Handle an exception: process the exception; interpret the signal

Exceptions – Key Words

- `throw` – followed by an argument, is used to throw an exception
- `try` – followed by a block `{ }`, is used to invoke code that throws an exception
- `catch` – followed by a block `{ }`, is used to detect and process exceptions thrown in preceding `try` block. Takes a parameter that matches the type thrown.

Exceptions – Flow of Control

1) A function that throws an exception is called from within a try block
2) If the function throws an exception, the function terminates and the try block is immediately exited. A catch block to process the exception is searched for in the source code immediately following the try block.
3) If a catch block is found that matches the exception thrown, it is executed. If no catch block that matches the exception is found, the program terminates.

Exceptions – Example (1)

```
// function that throws an exception
int totalDays(int days, int weeks)
{
   if ((days < 0) || (days > 7))
     throw "invalid number of days";
// the argument to throw is the
// character string
  else
     return (7 * weeks + days);
}
```


Exceptions – Example (2)

```
try // block that calls function
{
    totDays = totalDays(days, weeks);
   cout << "Total days: " << days;
}
catch (char *msg) // interpret
           //  exception
{
   cout << "Error: " << msg;
}
```


Exceptions – What Happens

1) `try` block is entered. `totalDays` function is called
2) If 1st parameter is between 0 and 7, total number of days is returned and `catch` block is skipped over (no exception thrown)
3) If exception is thrown, function and `try` block are exited, `catch` blocks are scanned for 1st one that matches the data type of the thrown exception. `catch` block executes

From Program 16-1

```
int main()
{
   int num1, num2;  // To hold two numbers
   double quotient; // To hold the quotient of the numbers

   // Get two numbers.
   cout << "Enter two numbers: ";
   cin >> num1 >> num2;

   // Divide num1 by num2 and catch any
   // potential exceptions.
   try
   {
      quotient = divide(num1, num2);
      cout << "The quotient is " << quotient << endl;
   }
   catch (char *exceptionString)
   {
      cout << exceptionString;
   }

   cout << "End of the program.\n";
   return 0;
}
```

From Program 16-1

```
//*********************************************
// The divide function divides numerator by  *
// denominator. If denominator is zero, the  *
// function throws an exception.             *
//*********************************************

double divide(int numerator, int denominator)
{
   if (denominator == 0)
      throw "ERROR: Cannot divide by zero.\n";

   return static_cast<double>(numerator) / denominator;
}
```

Program Output with Example Input Shown in Bold

```
Enter two numbers: 12 2 [Enter]
The quotient is 6
End of the program.
```

Program Output with Example Input Shown in Bold

```
Enter two numbers: 12 0 [Enter]
ERROR: Cannot divide by zero.
End of the program.
```


What Happens in theTry/Catch Construct

```
try
{
   quotient = divide(num1, num2);
   cout << "The quotient is " << quotient << endl;
}
catch (char *exceptionString)
{
   cout << exceptionString;
}

cout << "End of the program.\n";
return 0;
```

If this statement throws an exception... (→ `quotient = divide(num1, num2);`)

... then this statement is skipped. (→ `cout << "The quotient is " << quotient << endl;`)

If the exception is a string, the program jumps to this catch clause. (→ `catch (char *exceptionString)`)

After the catch block is finished, the program resumes here. (→ `cout << "End of the program.\n";`)

What if no exception is thrown?

```
try
{
     quotient = divide(num1, num2);
     cout << "The quotient is " << quotient << endl;
}
catch (char *exceptionString)
{
   cout << exceptionString;
}

cout << "End of the program.\n";
return 0;
```

If no exception is thrown in the try block, the program jumps to the statement that immediately follows the try/catch construct.

Exceptions - Notes

- Predefined functions such as `new` may throw exceptions
- The value that is thrown does not need to be used in `catch` block.
  - in this case, no name is needed in catch parameter definition
  - `catch` block parameter definition *does* need the type of exception being caught

Exception Not Caught?

- An exception will not be caught if
  - it is thrown from outside of a `try` block
  - there is no `catch` block that matches the data type of the thrown exception
- If an exception is not caught, the program will terminate

Exceptions and Objects

- An <u>exception class</u> can be defined in a class and thrown as an exception by a member function
- An exception class may have:
  - no members: used only to signal an error
  - members: pass error data to `catch` block
- A class can have more than one exception class

Contents of `Rectangle.h` (Version 1)

```
// Specification file for the Rectangle class
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle
{
   private:
      double width;     // The rectangle's width
      double length;    // The rectangle's length
   public:
      // Exception class
      class NegativeSize
         { };            // Empty class declaration

      // Default constructor
      Rectangle()
         { width = 0.0; length = 0.0; }

      // Mutator functions, defined in Rectangle.cpp
      void setWidth(double);
      void setLength(double);

```


Contents of Rectangle.h (Version1) (Continued)

```
      // Accessor functions
      double getWidth() const
         { return width; }

      double getLength() const
         { return length; }

      double getArea() const
         { return width * length; }
};
#endif
```

Contents of `Rectangle.cpp` (Version 1)

```
// Implementation file for the Rectangle class.
#include "Rectangle.h"

//***********************************************************
// setWidth sets the value of the member variable width.    *
//***********************************************************

void Rectangle::setWidth(double w)
{
   if (w >= 0)
      width = w;
   else
      throw NegativeSize();
}

//***********************************************************
// setLength sets the value of the member variable length.  *
//***********************************************************

void Rectangle::setLength(double len)
{
   if (len >= 0)
      length = len;
   else
      throw NegativeSize();
}
```


Program 16-2

```
// This program demonstrates Rectangle class exceptions.
#include <iostream>
#include "Rectangle.h"
using namespace std;

int main()
{
   int width;
   int length;

   // Create a Rectangle object.
   Rectangle myRectangle;

```


Program 16-2 *(continued)*

```
   // Get the width and length.
   cout << "Enter the rectangle's width: ";
   cin >> width;
   cout << "Enter the rectangle's length: ";
   cin >> length;

   // Store these values in the Rectangle object.
   try
   {
      myRectangle.setWidth(width);
      myRectangle.setLength(length);
      cout << "The area of the rectangle is "
           << myRectangle.getArea() << endl;
   }
   catch (Rectangle::NegativeSize)
   {
      cout << "Error: A negative value was entered.\n";
   }
   cout << "End of the program.\n";

   return 0;
}
```


Program 16-2 (Continued)

Program Output with Example Input Shown in Bold

```
Enter the rectangle's width: 10 [Enter]
Enter the rectangle's length: 20 [Enter]
The area of the rectangle is 200
End of the program.
```

Program Output with Example Input Shown in Bold

```
Enter the rectangle's width: 5 [Enter]
Enter the rectangle's length: -5 [Enter]
Error: A negative value was entered.
End of the program.
```

What Happens After `catch` Block?

- Once an exception is thrown, the program cannot return to throw point. The function executing `throw` terminates (does not return), other calling functions in `try` block terminate, resulting in unwinding the stack
- If objects were created in the `try` block and an exception is thrown, they are destroyed.

Nested `try` Blocks

- `try/catch` blocks can occur within an enclosing `try` block
- Exceptions caught at an inner level can be passed up to a `catch` block at an outer level:

```
catch ( )
{
  . . .
           throw;  // pass exception up
}              // to next level
```


16.2

- Function Templates

Function Templates

- Function template: a pattern for a function that can work with many data types
- When written, parameters are left for the data types
- When called, compiler generates code for specific data types in function call

Function Template Example

```
template <class T>
T times10(T num)
{
        return 10 * num;
}
```

template prefix

generic data type

type parameter

What gets generated when `times10` is called with an `int`:	What gets generated when `times10` is called with a `double`:
`int times10(int num) { return 10 * num; }`	`double times10(double num) { return 10 * num; }`

Function Template Example

```
template <class T>
T times10(T num)
{
        return 10 * num;
}
```

- Call a template function in the usual manner:

```
int ival = 3;
double dval = 2.55;
cout << times10(ival); // displays 30
cout << times10(dval); // displays 25.5
```


Function Template Notes

- Can define a template to use multiple data types:

```
template<class T1, class T2>
```

- Example:

```
template<class T1, class T2>      // T1 and T2 will be
double mpg(T1 miles, T2 gallons) // replaced in the
{                                 // called function
    return miles / gallons         // with the data
}                                 // types of the
                                  // arguments
```


Function Template Notes

- Function templates can be overloaded Each template must have a unique parameter list

```
template <class T>
T sumAll(T num) ...
template <class T1, class T2>
T1 sumall(T1 num1, T2 num2) ...
```

Function Template Notes

- All data types specified in template prefix must be used in template definition
- Function calls must pass parameters for all data types specified in the template prefix
- Like regular functions, function templates must be defined before being called

Function Template Notes

- A function template is a pattern
- No actual code is generated until the function named in the template is called
- A function template uses no memory
- When passing a class object to a function template, ensure that all operators in the template are defined or overloaded in the class definition

16.3

- Where to Start When Defining Templates

Where to Start When Defining Templates

- Templates are often appropriate for multiple functions that perform the same task with different parameter data types
- Develop function using usual data types first, then convert to a template:
  - add template prefix
  - convert data type names in the function to a type parameter (*i.e.*, a T type) in the template

16.4

- Class Templates

Class Templates

- Classes can also be represented by templates. When a class object is created, type information is supplied to define the type of data members of the class.
- Unlike functions, classes are instantiated by supplying the type name (`int`, `double`, `string`, etc.) at object definition

Class Template Example

```
template <class T>
class grade
{
    private:
            T score;
        public:
            grade(T);
            void setGrade(T);
            T getGrade()
};
```


Class Template Example

- Pass type information to class template when defining objects:

```
grade<int> testList[20];
grade<double> quizList[20];
```

- Use as ordinary objects once defined

Class Templates and Inheritance

- Class templates can inherit from other class templates:

```
template <class T>
class Rectangle
    { ... };
template <class T>
class Square : public Rectangle<T>
    { ... };
```

- Must use type parameter `T` everywhere base class name is used in derived class

16.5

- Introduction to the Standard Template Library

Introduction to the Standard Template Library

- Standard Template Library (STL): a library containing templates for frequently used data structures and algorithms
- Not supported by many older compilers

Standard Template Library

- Two important types of data structures in the STL:
  - containers: classes that stores data and imposes some organization on it
  - iterators: like pointers; mechanisms for accessing elements in a container

Containers

- Two types of container classes in STL:
  - sequence containers: organize and access data sequentially, as in an array. These include `vector`, `deque`, and `list`
  - associative containers: use keys to allow data elements to be quickly accessed. These include `set`, `multiset`, `map`, and `multimap`

Iterators

- Generalization of pointers, used to access information in containers
- Four types:
  - forward (uses `++`)
  - bidirectional (uses `++` and `--`)
  - random-access
  - input (can be used with `cin` and `istream` objects)
  - output (can be used with `cout` and `ostream` objects)

Algorithms

- STL contains algorithms implemented as function templates to perform operations on containers.
- Requires `algorithm` header file
- `algorithm` includes

`binary_search`	`count`
`for_each`	`find`
`find_if`	`max_element`
`min_element`	`random_shuffle`
`sort`	and others