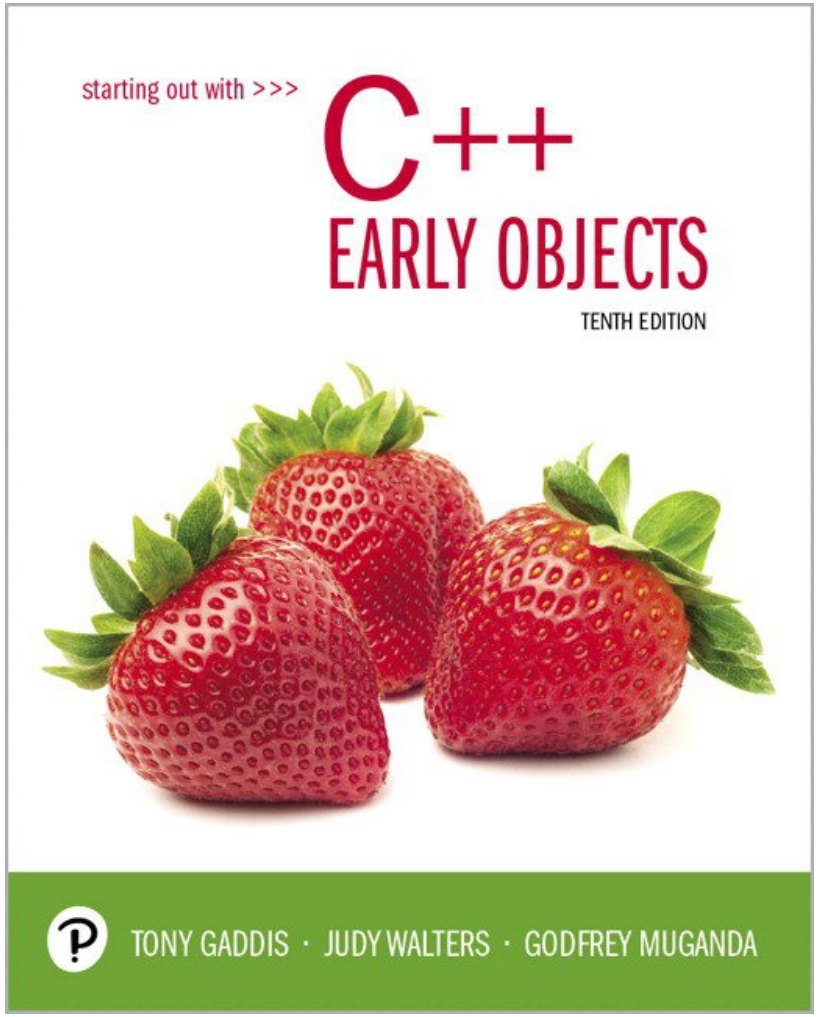
Starting Out with C++ Early Objects

Tenth Edition



Chapter 16

Exceptions and Templates

Topics

16.1 Exceptions

16.2 Function Templates

16.3 Class Templates

16.4 Class Templates and Inheritance

16.1 Exceptions

- An exception is a value or an object that indicates that an error has occurred
- When an exception occurs, the program must either terminate or jump to special code for handling the exception.
- The special code for handling the exception is called an exception handler

Exceptions – Key Words

- **`throw`** – followed by an argument, is used to signal an exception
- **`try`** – followed by a block **`{ }`**, is used to invoke code that may throw an exception
- **`catch`** – followed by a block **`{ }`**, is used to process exceptions thrown in a preceding **`try`** block. It takes a parameter that matches the type of exception thrown

Throwing an Exception

- Code that detects the exception must pass information to the exception handler. This is done using a **`throw`** statement:

```
throw string("Emergency!");
throw 12;
```

- In C++, information thrown by the **`throw`** statement may be a value of any type

Catching an Exception 1 of 2

- Block of code that handles the exception is said to catch the exception and is called an exception handler
- An exception handler is written to catch exceptions of a given type: For example, the code

```
catch(string str)
{
   cout << str;
}
```

can only catch exceptions that are string objects

Catching an Exception 2 of 2

Another example of a handler:

```
catch(int x)
{
  cerr << "Error: " << x;
}
```

This can catch exceptions of type **`int`**

Connecting to the Handler

Every catch block is attached to a try block of code and is responsible for handling exceptions thrown from that block

```
try
{
   // code that may throw an exception
   // goes here
}
catch(char e1)
{
   // This code handles exceptions
   // of type char that are thrown
   // in the try block
}
```

Execution of Catch Blocks

- The catch block syntax is similar to a that of a function
- A catch block has a formal parameter that is initialized to the value of the thrown exception before the block is executed

Exception Example

- An example of exception handling is code that computes the square root of a number.
- It throws an exception in the form of a string object if the user enters a negative number

Example

```
int main( )
{
  try
  {
    double x;
    cout << "Enter a number: ";
    cin >> x;
    if (x < 0) throw string("Bad argument!");
    cout << "Square root of " << x << " is " << sqrt(x);
  }
  catch(string str)
  {
       cout << str;
  }
  return 0;
}
```

Flow of Control

1. The computer encounters a **`throw`** statement in a **`try`** block
2. The computer evaluates the **`throw`** expression, and immediately exits the **`try`** block
3. The computer selects an attached **`catch`** block that matches the type of the thrown value, places the thrown value in the catch block's formal parameter, and executes the catch block

Uncaught Exception

- An exception may be uncaught if
  - there is no **`catch`** block with a data type that matches the exception that was thrown, or
  - it was not thrown from within a **`try`** block
- The program will terminate in either case

Throwing an Exception Class

- An exception class can be defined and thrown
- A catch block must be designed to catch an object of the exception class
- The exception class object can pass data to the exception handler via data members

Handling Multiple Exceptions

Multiple catch blocks can be attached to the same try block of code. The catch blocks should handle exceptions of different types

```
try{...}
catch(int iEx){ }
catch(string strEx){ }
catch(double dEx){ }
```

Exception When Calling `new`

- If **`new`** cannot allocate memory, it throws an exception of type **`bad_alloc`**
- Must **`#include <new>`** to use **`bad_alloc`**
- You can invoke **`new`** from within a **`try`** block, then use a **`catch`** block to detect that memory was not allocated.

Where to Find an Exception Handler?

- The compiler looks for a suitable handler attached to an enclosing `try` block in the same function
- If there is no matching handler in the function, it terminates execution of the function, and continues the search for a handler starting at the point of the call in the calling function.

Unwinding the Stack

- An unhandled exception propagates backwards into the calling function and appears to be thrown at the point of the call
- The computer will keep terminating function calls and tracing backwards along the call chain until it finds an enclosing **`try`** block with a matching handler, or until the exception propagates out of **`main`** (terminating the program).
- This process is called unwinding the call stack

Nested Exception Handling

`try` blocks can be nested in other **`try`** blocks

```
try
  {
     processData();  // function
     // containing a try block
  }
 catch(string s)
 { }
```

Rethrowing an Exception

- Sometimes an exception handler may need to do some tasks, then pass the exception to a handler in the calling environment.
- The statement

```
throw;
```

with no parameters can be used within a **`catch`** block to pass the exception to a handler in the outer block

16.2 Function Templates

- Function template: A pattern for creating definitions of functions that differ only in the type of data they manipulate. It is a generic function

- They can be better than overloaded functions, since the code defining the algorithm of the function is only written once

Function Template Example

Two functions that differ only in the type of the data they manipulate

```
void swap(int &x, int &y)
{ int temp = x; x = y;
  y = temp;
}

void swap(char &x, char &y)
{ char temp = x; x = y;
  y = temp;
}
```

A swap Template

The logic of both functions can be captured with one template function definition

```
template<class T>
void swap(T &x, T &y)
{ T temp = x; x = y;
  y = temp;
}
```

Using a Template Function

- When a function defined by a template is called, the compiler creates the actual definition from the template by inferring the type of the type parameters from the arguments in the call:

```
int i = 1, j = 2;
swap(i,j);
```

- This code makes the compiler instantiate the template with type `int` in place of the type parameter `T`

Function Template Notes 1 of 2

- 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 referred to in the template are defined or overloaded in the class definition

Function Template Notes 2 of 2

- 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
- Function templates can be overloaded – need different parameter lists
- Like regular functions, function templates must be defined before being called

Where to Start When Defining Templates

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

Function templates and C++ 11

- As of C++ 11, the key word **`typename`** may be used instead of **`class`** in the template prefix.
- Thus,

```
template<class T>
```

May be written as

```
template<typename T>
```

16.3 Class Templates

- It is possible to define templates for classes. Such classes define abstract data types
- Unlike functions, a class template is instantiated by supplying the type name (**`int`**, **`float`**, **`string`**, etc.) at object definition

Class Template

Consider the following classes

1. Class used to join two integers by adding them:

```
class Joiner
{ public:
     int combine(int x, int y)
     {return x + y;}
};
```

2. Class used to join two strings by concatenating them:

```
class Joiner
{ public:
     string combine(string x, string y)
     {return x + y;}
};
```

Example class Template

A single class template can capture the logic of both classes: it is written with a template prefix that specifies the data type parameters:

```
template <class T>
class Joiner
{
public:
    T combine(T x, T y)
        {return x + y;}
};
```

Using Class Templates

To create an object of a class defined by a template, specify the actual parameters for the formal data types

```
Joiner<double> jd;
Joiner<string> sd;
cout << jd.combine(3.0, 5.0);
cout << sd.combine("Hi ", "Ho");
```

Prints `8.0` and `Hi Ho`

Member Function Definitions Outside of a Template Class

- If a member function is defined outside of the class, then the definition requires the template header to be prefixed to it, and the template name and type parameter list to be used to refer to the name of the class:

```
template<class T>
 T Joiner<T>::combine(T x, T y)
       {return x + y;}
```

16.4 Class Templates and Inheritance

- Templates can be combined with inheritance
- You can derive a template class from a template class
- Other combinations are possible:
  - Derive a template from an ordinary class
  - Derive an ordinary class from a template

Copyright