**Chapter 14: Templates and Exceptions**

Table of Contents

[Exceptions 2](#_Toc49596036)

[Why Do We Need Exceptions? 2](#_Toc49596037)

[Exception Syntax 3](#_Toc49596038)

[Multiple Exceptions 5](#_Toc49596039)

[Exceptions with Arguments 6](#_Toc49596040)

[The bad\_alloc Class 7](#_Toc49596041)

[Exception Notes 8](#_Toc49596042)

[Function Templates 10](#_Toc49596043)

[Syntax and Usage 11](#_Toc49596044)

[Inner Workings 12](#_Toc49596045)

[The Macro Alternative 13](#_Toc49596046)

[Class Templates 14](#_Toc49596047)

[Object Declarations 15](#_Toc49596048)

[External Member Functions 15](#_Toc49596049)

## Exceptions

Exceptions provide a systematic, object-oriented approach to handling errors generated by C++ classes. Exceptions are errors that occur at runtime. They are caused by a wide variety of problems, such as running out of memory, being unable to open a file, trying to initialize an object to an impossible value, or using an out-of-bounds index.

### Why Do We Need Exceptions?

C-language programs often signal an error by returning a particular value from the function in which it occurred. For example, disk-file functions often return NULL or 0 to indicate an error. We could check for these values when we call the functions using simple if-else statements, and design our program to behave accordingly by writing code to handle the errors (called error-handling routines).

One problem with this approach is that we would need to write error-handling routines every single time the function was called. This would make our code very disorganized. Another problem is when we use classes, which can cause exceptions without any explicit function calls, such as in the constructor function. The most serious problem we would face is when using class libraries, which are often designed by separate people. It is very difficult to communicate error values from a class member function to the program calling the function in this case.

Old-time C programming had another approach to catching errors, combining two functions, setjmp() and longjmp(). However, that method is inappropriate in an object-oriented environment, since it does not properly handle the destruction of objects.

### Exception Syntax

Consider an application that interacts with objects of a certain class. If the application makes a mistake, an error occurs in one of the member functions. The member function then informs the application about that error.

With exceptions, this scenario is called ‘throwing an exception’. In the application, we write a separate section of code, called the exception handler or the catch block, specifically to handle the error. When the member function throws the exception, the catch block catches it.

Any code that has a possibility of causes an exception is put inside a try block. Essentially, exceptions thrown due to the code in the try block are caught in the catch block.

Another thing to note is that we will have to create a new kind of class, called an exception class. An object of this class is what will be thrown.

class AClass  
{  
public:  
 class AnError {}; *// exception class; empty body for now* void func()  
 {  
 if (*/\* error condition \*/*) throw AnError();  
 *// throwing an exception class object (notice constructor)* }  
};  
  
int main()  
{  
 try *// try block* {  
 AClass obj1;  
 obj1.func();  
 }

catch (AClass::AnError) *// catch block* {  
 *// some code to deal with this situation* }  
}

C++

Due to the way exceptions work, we are able to have finer control over what happens when an error occurs. We can write code that simply notifies the user about the error and continues with program execution, or we can write code that takes care of any necessary tasks before terminating the program, such as deleting objects we created. When using exceptions in the former method, it is common to put the try and catch blocks inside a loop, so that the user is brought back to the top of the try block after the error.

Note that if there is an exception that we do not provide an exception handler for, the program is terminated abruptly by the system, as though we did not use any exceptions at all.

### Multiple Exceptions

We can make as many exception classes as we want inside a class. This allows us to throw a different class’s object depending on what error has occurred, which in turn allows us to write code specific to that error in the corresponding catch block.

class Stack  
{  
public:  
 class Full {};  
 class Empty {};  
 void someFunc()  
 {  
 if (*/\* condition to check that class is empty \*/*) throw Empty();  
 else (*/\* condition to check that class is full \*/*) throw Full();  
 }  
};  
  
int main()  
{  
 Stack s1;  
 try  
 {  
 s1.someFunc();  
 }  
 catch (Stack::Empty)  
 {  
 cout<<"Exception: Stack is empty."<<endl;  
 }  
 catch (Stack::Full)  
 {  
 cout<<"Exception: Stack is full."<<endl;  
 }  
}

C++

A group of catch blocks like this is called a catch ladder. Notice the similarities with a switch statement.

Note that all the catch blocks that are being used with a particular try block must immediately follow the try block. Only one of the catch blocks may be activated by a single exception.

### Exceptions with Arguments

It possible to pass extra information to the catch block when an exception occurs, such as what caused the exception or what function was being executed when it occurred. This is done using the exception object we throw.

If we add some attributes and a constructor to the exception class definition, and we initialize the corresponding object accordingly when we throw it, we can use that information in the catch block.

#include <iostream>  
using namespace std;  
class someClass  
{  
public:  
 class exceptionClass  
 {  
 public:  
 string functionName;  
 int someInteger  
 exceptionClass (string s, int i) : functionName(s), someInteger(i) {}; *// constructor* };  
 void someFunc()  
 {  
 throw exceptionClass ("someFunc", -1);  
 }  
};

int main()  
{  
 try  
 {  
 someClass c1;  
 c1.someFunc();  
 }  
 catch (someClass::exceptionClass c2)  
 {  
 cout<<"Exception in function: "<<c2.functionName<<endl;  
 cout<<"Error code: "<<c2.someInteger<<endl;  
 }  
}

C++

We can of course use this method to do whatever we want, but exception arguments usually carry information that helps diagnose errors.

### The bad\_alloc Class

C++ has many built-in exception classes. bad\_alloc is probably the most commonly used one. It throws an error when attempting to allocate memory with the new operator. This exception was called xalloc in older versions of C++.

int main()  
{  
 char\* ptr;  
 try  
 {  
 ptr = new char[1000];  
 }  
 catch (bad\_alloc)  
 { *// error code* }  
 delete[] ptr;  
}

C++

### Exception Notes

We have only covered the simplest and most common approaches to using exceptions. There is much more, but we will not be discussing it for now. Instead, we shall end this topic with a few notes.

It is not necessary that the statement that causes an exception be directly in the try block. The exception will still work if there is a function in the try block that contains a statement that causes the error. This sort of nesting can (theoretically) go on indefinitely. Thus, it is not necessary to burden all the functions with error-handling capabilities. However, it can sometimes be useful for a lower-level function to be able to add their own identifying data to an exception before throwing it to an upper-level function. Exceptions are transmitted up nested functions until a catch block is found.

We mentioned that exceptions are very useful with class libraries. Often, a library routine is written by people unrelated to the program using it. As such, they may not know what to do when a particular error occurs. Instead, they use exceptions to inform the main program about the error and pass along any relevant data.

Exceptions should not be used for every kind of error. They add some overhead in terms of program size and, if an error occurs, execution time. For example, user input errors are easily detectable by the program itself and should not be handled with exceptions.

When an exception is thrown, any objects that were created by the code up to that point in the try block automatically have their destructors called. This is necessary because the program cannot tell which particular statement inside the try block caused the error, and to recover from it, it will at least have to start over from the top of the try block. Thus, the exception mechanism guarantees that the code in the try block is ‘reset’, in this sense.

## Function Templates

Templates make it possible to use one function or class to handle many different data types. We shall be looking at function templates first.

Say we create a function that accepts an integer and returns its absolute value. If we want to find the absolute values for data of many different data types, we will need to write separate functions for each data type, making use of the concept of overloading.

This is perfectly valid in C++, but it is a waste of time and space. In C, we cannot even use the same function name, since the concept of overloading does not exist. Further, if we make a tiny mistake, we would need to go to every function we created to correct it, least we have inconsistencies in our program.

Function templates make tasks like this easier. We create one function that is capable of working with several different data types. Which ‘version’ of the function will be used depends entirely on what the data type of the argument we pass to it is.

### Syntax and Usage

When creating a template function, it is probably best to start with a normal function for a specific data type. Once we have that working, then we can switch it to a template function. This makes it easier to debug the function itself.

The final template function may look something like this:

template <class T> // function template  
T abs (T n)  
{  
 return (n < 0) ? -n : n;  
}

C++

This entire syntax, starting from the first line to the function definition, is called a function template. The one we have defined above will work with absolutely everything, including user-defined data types, provided that any operators used within the function can be used with those data types.

Notice the lack of a semicolon at the end of the first line. Also note that the name we have used, T, is nothing special. Any name is valid. T is called the template argument. The first line essentially declares the template argument. The class keyword is used, since classes can be used to define data types.

Throughout the template definition, wherever we would have used a normal data type, we use the template argument T instead.

Whenever a template function is invoked, all instances of the same template argument must be of the same type. This means that if we pass two arguments to a function using the same template argument as its parameters, both have to be the same data type for the template function to work. Of course, if we used two different template arguments for the two parameters of the function, then we can pass two arguments of different data types.

We can declare two separate template arguments that we want to use with a particular function like this:

template <class aType, class bType>  
bType someFunc(aType\* array, aType value, bType size);

C++

### Inner Workings

The function template itself does not cause the compiler to do anything at all. After all, the compiler does not know what data type it is. Code is not generated until the function is actually called.

At the function call, the compiler gets to know the data type of the argument passed to the function. It then generates a specific version of the function substituting that data type wherever it sees the template argument T. This is known as instantiating the function template, and each instantiated version of the function is called a template function (yes, confusing I know). Finally, the compiler generates a call to the newly instantiated function. The compiler will only generate one version of the function for each data type. Notice how the function template is like a blueprint for making functions and not a function in itself, similar to how a class is a blueprint for objects.

Note that the return type of the function is irrelevant to the generation of a new instance of the function. That depends solely on the arguments passed.

Also note that using templates does not cause any extra usage of system resources such as RAM. Generating a template function for a number of different data types will indeed cause a lot of memory usage, especially if there are multiple templates being used in one function, but this is no different than if we had simply written all the different versions of the function. This is because nothing is generated until a function call is actually made.

### The Macro Alternative

There is an alternative way to achieve the same thing we did with templates by using macros. This method was followed in old versions of C programs.

#define abs(n) ( (n < 0) ? (-n) : (n) )

C++

However, it is strongly discouraged to ever do this, since it has many small and big problems. For one thing, macros are hardly ever used in C++. For another, the macro will not provide any form of type checking. If we have multiple arguments that should be of the same data type, we could have used the same template argument in their definition to ensure that they are the same type. In macros, those arguments could be given different data types and we would not know until it is too late. Similarly, since a macro has no return type specified, the compiler will be unable to tell if the returned value is being assigned to an incompatible variable.

## Class Templates

Class templates are generally used for data storage classes, also known as container classes. Stacks and linked lists are examples of data-storage classes. These classes, as far as we have seen them, can only store data of a particular data type, such as an integer. Instead of writing a separate class for every type of data we want to use, we can use a class template.

template <class Type>  
class Stack  
{  
private:  
 Type st[100];  
 int top;  
public:  
 Stack()  
 { top = -1; }  
};

C++

The approach is similar to that used for function templates. The template keyword and the class definition indicate that the entire class is meant to be a template. The template argument Type should be used in every place where a data type would normally be used.

A template class can of course, also work with user-defined data types. However, we will need to ensure every function we use in the class can operate properly in this case. This may require that we overload certain operators for the user-defined data type. This limitation also applies to non-user-defined data types that cannot work with the required operators, such as character arrays (char\*) with the extraction (<<) operator.

### Object Declarations

The difference between a class template and a function template is in the way that they are instantiated. Function templates make use of the arguments provided to the function to determine what the data type of the template arguments should be. However, we have no way of doing this with a class, since we simply create objects of the class. Thus, we have to explicitly tell the compiler what data type to use with a particular class object.

Stack<float> s1;

C++

This creates an object that uses floats in place of the template arguments. Again, the compiler will generate a different class definition for each data type. Every new generation will cause a new set of member functions to be generated and every new object will cause some new memory space to be allocated to that object.

### External Member Functions

With classes, it is possible to define member functions externally, outside of the class definition. With template classes, this is still possible, albeit with a slightly different syntax.

template<class Type>  
Stack<Type>::Stack() // external constructor definition  
{ top = -1; }

C++

Notice how the line template<class Type> is written again. This line must precede every single external member function definition we write.

We must be careful to uses the correct syntax depending on what we are doing. Notice how in the original class definition, simply writing Stack was enough everywhere within the definition. However, we needed to use Stack<Type> both when defining a member function externally (and everywhere inside the function definition) and when declaring an object, where we gave Type a specific data type. Similarly, if we were to return a Type value, we would need to specify that yet again. However, if we pass an object to a function, we do not need to specify Type.

Template<class Type>  
Stack<Type> Stack<Type>::someFunc(Stack arg)  
{}

C++

This function may feel extremely confusing due to the different syntaxes we had to use. Essentially, we have some member function that takes an object of the Stack class as an argument. We do no need to care about the data type this object uses. The function also returns an object of the Stack class, but this time we do need to specify the data type the object being returned uses.