CHAPTER 9

Templates and Exception Handling

In this chapter, we shall discuss two powerful features of C++ programming, namely templates and exception handling. The main advantage of templates is that it enables the programmer to develop generic code (classes or functions) and let the compiler generate data type specific code. Thus, the effort required on the programmer's side is reduced. Templates help in developing generic code. In fact, we have briefly discussed function templates in Chapter 3. In this chapter, we shall elaborate on both class and function templates. In the later half of the chapter, we shall elaborate on exception handling and another powerful feature of C++ that helps programmer decide the way exceptions or abnormal program behaviour to be handled.

9.1 Introduction

One requirement for function templates would be to develop a function called print array that should display the contents of either an integer or float or character array passed to it as an argument. In other words, the generic specification that displays the contents of the array, not relating to any specific integer or character or float array. This generic specification is referred to as function template. Based on the requirement or the specific data type with which a function template is called or invoked, the compiler on behalf of the programmer generates the data type specific code or the function definition, which handles the specific data type. Such compiler-generated versions of function definition are referred to as template functions. The reasoning behind calling them as template functions is that these are functions that are generated from templates (function templates to be precise).

On similar lines, generic class specifications or behaviours are referred to as class templates, and data type specific class specifications are referred to as template classes. One example for class template requirement would be to create a generic stack class and then instantiate or create specific integer, or float or character stack. To correlate the concept of templates with realistic example, function and class templates are similar to stencils from which differing shapes can be created or traced. Template functions and template classes are nothing but the separate tracings that are all of the same shape but can differ in their colour. Here, the similarity in shape of different coloured tracings is the genericness [template] and the various colours available are equivalent to the data type specific versions generate from the generic templates. In fact, templates promote the concept of software reusability to a great extent. We will first explore the concept of function templates and then proceed on to class templates.

9.2 Function Templates

We have already mentioned in Chapter 3 that function templates are similar to function overloading concept. Function overloading is normally the choice when similar but not equivalent operations need to be performed on different types of data. If the operations are identical or equivalent across different data types, the choice is function templates. Function overloading is the choice when the logic associated with different data types is slightly different, but definitely not identical or similar. Getting on with function templates, the programmer specifies a generic function behaviour or function template definition. Based on the arguments types (data type) passed to the function (at the invocation point). The compiler generates separate data type specific versions of the function.

To a certain extent this can be simulated using the defined macros in C, but macros suffer from the setback that there is no type checking enabled, whereas templates have in them type checking enabled. Function `template` definition is preceded by the keyword template followed by a list of formal parameters to the function template that shall be referred to within the function `template` definition. This list of formal parameter is of generic type and is specified within the angled brackets that succeed the `template` keyword. These formal parameters correlate with the generic data type, which would be predefined, or user-defined. The exact syntax is as follows:

```
Template <class T>
```

where T is some predefined or user-defined generic data type.

The formal type parameters of a template definition are used to specify the argument types to functions, functions return type and for declaring variables for use within the function definition. Here, in the function definition, we are referring to the generic behaviour. The function definition follows the function template header and the definition is quite similar to that of a normal function definition with the only difference that only generic data types and not specific data types are manipulated within the function template definition. Let us now develop a generic swap function `template` and then leave it to the compiler to generate data type specific code to swap integers or characters or floating point inputs, code for which is as shown in Figure 9.1.

The generic swap function `template` declares a single formal parameter `T` (`T` to be some valid identifier) for the type of data to be swapped by this function `template`. When the compiler detects an invocation of swap function, the type of data with which the swap function is invoked replaces the generic data type `T` throughout the function `template` definition of swap. This compiler substitute or generate code for a specific data type is the `template` function. For better understanding, the invocation of function swap with integers [template function for integer data type passed to a function `template swap(T)` generates the following integer data type specific version of the function `template swap`:

```
void swap (int & input1, int & input 2)
{
int temp;
temp = input1;
input1 = input2;
input2 = temp;
}
```

```
template <class T>
void swap (T & inp1, T & inp2)
{
T temp;
Temp = inp1;
inp1 = inp2;
inp2 = temp;
}
#include<iostream.h>
int main ( )
{
int num1 = 20, num2 =30;
float no1 = 5.5, no2 = 6.6;
char ip1 ='A', ip2 ='B';
cout << "Swapping Two Integers \n";
cout << "Before Swapping Values are \n";
cout << num1 <<"\t"<<num2<<"\n";
swap (num1,num2);
cout << "After Swapping Values are \n";
cout << num1 <<"\t"<<num2<<"\n";
cout << "Swapping Two Floating Point Numbers \n";
cout << "Before Swapping Values are \n";
cout << no1 <<"\t"<<no2<<"\n";
swap(no1,no2);
cout<< "After Swapping Values are \n";
cout<< no1 <<"\t"<<no2<<"\n";
cout<< "Swapping Two Characters \n";
cout<< "Before Swapping Values are \n";
cout<< ip1 <<"\t"<<ip2<<"\n";
swap(ip1,ip2);
cout<< "After Swapping Values are \n";
cout<< ip1 <<"\t"<<ip2<<"\n";
return 0;
}
```

Figure 9.1: C++ code to demonstrate function templates.

Similarly, for other invocations of swap with `float` and `char` data type the compiler generates the code for the respective template function from the generic function template definition by suitably replacing the generic data type `T` with the specific data type (`char` or `float`). This is left as an exercise for the reader. Every different formal type parameter of a function template definition should be specified or identified in the formal parameter list of the function template definition header. The name of a formal parameter type in the template header can be mentioned only once and cannot be duplicated. However, with template functions, these formal parameter types need or will not remain unique. The instantiation or invocation of function templates is illustrated in the definition of function `main()` where we have three calls to swap, first call accepting two integers, second call accepting two float point numbers, and the third call accepting two character data type inputs. As is shown the two entities [integer or `char` or `float`] are swapped.

The `call swap (&num1,&num2);` causes the compiler to replace `T` in the generic function `template swap (T &, T&);` with integer and generate the swap template function associated for integer arguments. Similarly, the calls to other data type specific inputs are han-

dled by the compiler. Thus, the above example saves the programmer effort in developing three separate overloaded functions with the following prototypes of `void swap`: `void swap (int &,int &); void swap(float &,float &);` `void swap (char &,char &);` all of which would have the same code and differ only in their data types. Templates though support the concept of software reusability and reduces the programmer's effort by generating code (data type specific), multiple copies of template functions and template classes are still instantiated in a program despite the fact that the function or class template is specified only once.

We have earlier said that function templates have the concept of function overloading supported after the template functions have been generated. Thus, there is a considerable amount of memory consumption involved with templates feature of C++. Note that function templates can also be overloaded to have different generic behaviours associated with the same function name. The way templates are handled by the C++ compiler is as follows. It has already been mentioned that when a function is invoked, the compiler performs a matching process to locate the function definition that matches in signature (function header) with the called functions signature.

This methodology is applicable with template based c++ programs as well. First, the compiler checks for such precise match. If this comparison fails, the compiler proceeds checking if a function `template` from which template functions can be generated possibly match the signature of the function call [in number of arguments and the individual data type]. If the compiler is able to locate such a template match, the compiler generates the appropriate `template` function and invokes it to handle the function call. In cases where the compiler cannot find a valid match or multiple matches are encountered a suitable compiler error is generated. Let us look at another example for function templates. A generic sort function to sort either an array of integers or characters or floating point numbers shall be developed in the following example, code for which is shown in Figure 9.2.

9.3 Class Templates

In this section, we will explore the class templates feature of C++. Stack is one important data structure that is so very useful in expression evaluation. A stack is a constrained version of a linked list, new nodes(data) can be added or removed or insertion / deletion is only at the top. This is also referred to as the LIFO (Last In, First Out) data structure. This differs from QUEUE (FIFO—First In, First Out) in that deletion from the front while insertion is at the back. Based on the realistic Q syntax of newly arriving people at the back or rear of the Q, while the server or the service person caters to or services the person at the front end [deletion]. One important application of stacks is function call mechanism.

For example, when a function call is made, the called function must know the return address [caller address] to resume program execution. Return addresses are normally pushed onto the stack. If a series of function calls occur, the successive return values are pushed onto the stack in LIFO order so that each function can return to its appropriate caller. Recursive function calls are supported by stacks similar to non-recursive function calls. Having elaborated on stacks and their applications, let us now get back to the development of a generic stack class template from which either stack of integers or floats or characters can be [template classes] created. We shall implement the stack data structure using arrays, code for which is as shown in Figure 9.3.

```
template < class T>
void bubblesort (T arr[], int size)
{
for (int pass =1,pass<size;pass++)
for (int count =0;count<size-1;count++)
if( arr[count]>arr[count+1] )
{
T temp;
temp = work[count];
work[count]=work[count+1];
work[count+1]= temp;
}
}
#include<iostream.h>
#define SIZE 5
int main ( )
{
int array1[SIZE] = {5,4,3,2,1 };
float array2[SIZE] = {5.5,4,4,3.3,2.2,1.1};
char array3[SIZE] = 'E','D','C','B','A';
cout << "Sorting in Progress \n";
cout << "Integer Array Being Sorted \n";
bubblesort(array1,SIZE);
cout << "The sorted Integer Array is \n";
for (int i=0;i<SIZE;i++)
cout << array1[i];
cout << "Floating Point Array Being Sorted \n";
bubblesort(array2,SIZE);
cout << "The sorted Floating Point Array is \n";
for (int i=0;i<SIZE;i++)
cout << array2[i];
cout << "Character Array Being Sorted \n";
bubblesort(array3,SIZE);
cout << "The sorted Character Array is \n";
for (int i=0;i<SIZE;i++)
cout << array3[i];
return 0;
}
```

Figure 9.2: Function `template` to sort an array of integers.

```
template <class T>
class Stack
{
public :
Stack (int =20);
~Stack ();
bool push (const T & );
```

Figure 9.3: Continued

```
bool pop ( T & );
private : int size;
int top;
T * stackpointer;
bool empty( ) const
{
return (top == -1);
}
bool isfull ( ) const
{
return (top == size -1);
}
};
template <class T>
Stack<T> :: Stack (int sz)
{
size = sz >0?sz:20;
top = -1;
stackpointer = new T [size];
}
template <class T>
Stack<T> :: ~Stack( )
{
delete [] stackpointer;
}
bool Stack<T> :: push (const T & pval)
{
if (!(isfull()))
{
stackpointer [++top] = pval;
return true;
}
return false;
}
bool Stack<T> :: pop (T & popval)
{
if (!(isempty()))
{
popval = stackpointer[top-];
return true;
}
return false;
}
#include <iostream.h>
int main ( )
{
float f = 2.2;
int val = 2;
Stack<float> sof (4);
while (sof.push(f))
```

Figure 9.3: Continued

```
{
cout << "pushed \t"<<f<<"\n";
f+ =1.5;
}
cout << "Sorry ! The Float Stack is Full \n";
cout << "Popping from the Float Stack \n";
while (sof.pop(f))
cout << f;
Stack<int> soi (4);
while (soi.push(val))
{
cout << "pushed \t"<<val<<"\n";
val =1.5;
}
cout << "Sorry ! The Int Stack is Full \n";
cout << "Popping from the Int Stack \n";
while (soi.pop(val))
cout <<val;
return 0;
}
```

Figure 9.3: C++ stack class template code.

9.4 Stack Class Template Code Interpretation

The stack class template contains at its data members size of the stack (size), a cumulative index (top) that refers to the current top index or element of stack. The `int * stackpointer` data member of the class refers or points the first element or the base address of the array or the first array element. The constructor for the class template stack (defaults the size of the stack to 20) validates the size value passed to it. If the size value passed is not >=0, then the size is set to the default value of 20.

At the point of construction since the stack would be empty [no elements], the top data member [stack pointer index] is set to -1 to indicate that the stack at the point of construction is empty. The constructor then allocates the space required for the respective stack type. The destructor of Stack class template deallocates the space allocated for the stack pointer in the constructor using the delete operator. The class template has two helper or utility functions, namely `isfull`, `isempty` to check for if the stack is currently full in which case more elements cannot be pushed or not. `Isempty` is used to test if the stack is empty or not. In case of empty stack, elements cannot be removed from the stack (popping). An empty stack condition occurs when the top index =-1 and full stack occurs when top index = size -1.

The class template stack should support two basic operations of push and pop. Push member function is used to [push or insert values onto the stack. The push member function accepts as argument the value to be pushed onto the stack. A value of generic type is pushed onto the stack after checking for the `isfull` condition. If the stack is not full (`!isfull()`), then the value `pval` is pushed onto the stack by the statement `stackpointer [++top] = pval`. This top index is preincremented since the value to be pushed should be stored in the next index of the array. At the end of a successful push operation the push member function returns a true result to indicate that the push operation was successful. Successive push

operations are performed only if an immediately preceding push operation was successful (for a new push to be allowed, the earlier push should be successful).

The pop function is used to delete or remove elements in the last in first our fashion. The pop member function accepts an argument of generic type T to store the popped value. Note that this argument cannot be made constant since for the popped value to be stored in T, one should have write or assignment operations over the argument. Both push and pop member functions accepts arguments by reference to avoid the duplicating effect in case of value pass.

The pop member function checks for the stack being empty condition. If the stack is not empty, we proceed with the pop operation by decrementing the top index [top –]. Note that this is a postdecrement operation, since we require first the element at the top of the stack to be popped and then proceed to the next element. Thus, here the element `stack[top]` is retrieved first and then later the top index is decremented so that after the topmost element has been popped, the new top element of the stack would be earlier top index -1. The retrieved value is stored in the `popval` argument. The driver routine creates stacks of the integer and floating point data types respectively and invokes the push and pop functions to push/retrieve elements from the stack.

9.5 Exceptions

The exhaustive features supported by any programming language and C++ in specific brings along the possibility of several types of errors and erroneous conditions that the program has to encounter and handle. The feature in C++ that supports this mechanism of handling errors and exceptions is what is referred to as Exception Handling. One possible and most often encountered exception in programming is the divide by zero exception (the code for which is shown in Figure 9.4), in which case most languages come out with a prompt of Invalid division operation and terminating further program execution. C++'s exception handling differs from other languages in the fact that the error handling code is clearly demarcated from the main processing code. But in languages such as error handling code often gets mixed up or interspersed with the main program code, contributing to poor maintenance and understanding of the code.

C++ exception handling allows programs to identify and handle errors rather letting them happen and result in serious consequences. It is generally the choice when the program can recover from the error situation and the recovery procedure that handles the error is referred to as the exception handler. It is primarily designed to handle synchronous errors such as divide by zero that occur as the program executes the concerned statement or instruction. Asynchronous errors such as those involving disk reads, mouse clicks, etc. are not handled by exception handlers. The exception handling feature of C++ is so designed such that in case of no exceptions in the program, then the overhead as a result of exception handling code is minimal to the extent possible.

9.6 The `try`, `throw` and `catch` Syntax

Exception handling in C++ incorporates three syntactic constructs or keywords, namely `try`, `throw` and `catch`. The `try` construct or block as it is often referred to encompasses the program that is susceptible to errors or exceptions. Once an exception is generated, it is

```
#include<iostream.h>
class Example
{
public:
Example()
{
message=new char[30];
strcpy(prompt, "Invalid Divide by 0 Attempted \n");
}
char * message()
{
return prompt;
}
private:
char*prompt;
};
int main ()
{
int no1,no2;
float ans;
try
{
if(no2==0)
throw Example();
ans=no1/no2;
}
catch(Example e)
{
cout<<"Exception has occurred \n"<<e.message();
}
cout<<"Program Execution Successful, result is \n"<<ans;
return 0;
}
```

Figure 9.4: C++ code to handle divide by zero exceptions.

thrown (sent out) for an appropriate handler or exception handler to process and handle the exception. The entity or block that handles the exception is referred to as the `catch` block. Ideally, a `try` block is followed by several `catch` blocks, so as to say that each handler takes of specific exception situations. In the event of an exception being generated, the appropriate `catch` handler is identified and code belonging to that `catch` block is executed.

Situations where an exception has been thrown and none of the existing handlers can handle it, function `terminate` is called, which in turn invokes the abort function. Cases where exceptions are not generated, the entire `catch` block(s) following the `try` block is ignored and program execution resumes from the point after the last `catch` block. Exceptions are thrown within the `try` block or from a function that either directly or indirectly contains a `try` block. The statement or point at which an exception is thrown is referred to as the `throw` point. Once an exception is thrown control cannot return back to the `thrown` point. Information about the exception such as type or nature can be passed on to the exception handlers or `catch` blocks. Yes, exception details are passed as arguments to the appropriate `catch` handlers, which based on a parameter match invoke the intended exception handling routine.

Let us now see an example to handle the divide by zero exception. The code is as shown in Figure 9.4 and the class Example that involves a division behaviour (function) is equipped to handle the divide by zero situation.

As can be seen in the above code, the `try` block throws an exception of the user-defined class type (Example), when the denominator is zero, which is caught by the appropriate `catch` handler (only one in this case which accepts as parameter an object of class Example). The program then executes the handler resulting in the display of the message Exception has occurred followed up by the specific prompt message that it was An attempt to divide by zero that occurred. In the case of no exceptions being generated, normal execution of the code follows, resulting in the display of the computed quotient value.

9.7 The `throw` Construct

The keyword `throw` in C++ denotes the throw point and signals the occurrence of the exception. Along with the `throw` keyword, one operand, which could be an object or some other type can be thrown. It is this object or data type that is communicated to the external handlers to appropriately get invoked and handle the exception in the intended manner. Once an exception is thrown, it is generally handled by the handler or the `catch` block that is closest to the `try` block in which the exception occurred (of course the thrown data type or exception type needs to match the `catch` parameter). All the exception handlers or `catch` blocks for a specific `try` block are specified immediately after the `try` block.

Once an exception is thrown, control from the `try` block is exited and shifted to the appropriate `catch` routine or handler that can handle the exception. Situations where the `throw` statement lies deep within a `try` block or nested in a function call, still the appropriate `catch` handler is invoked and activate, once an exception is generated. As an example, out of bounds array subscription could be one exception situation where the indexing statement checks for the bounds to fall within the limits or not. When this condition fails, an exception is thrown and the `catch` handler that handles it is executed.

9.8 The `catch` Construct

As stated above, exception handlers are generally part of the `catch` blocks. A `catch` block contains the keyword `catch` followed by an optional parameter list (similar to functions) that identifies the type of exception. When a thrown exception is caught or there is a parameter match in terms of the exception type, the code that forms the definition of the `catch` block is executed. This is referred to as an exception being caught and handled appropriately. The `catch` block can contain either named or unnamed parameters. In the case of named parameters, it can be referenced within the `catch` block, while in the other situation, only the type (exception type) would suffice to identify the exception and then have an appropriate handler executed.

The `catch` handler that matches a thrown exception type or catches an exception is the first one that lies after the currently active `try` block. A `catch` block such as `catch(...)` or ellipsis catches all types of exceptions or is a generic exception handler. Thus, such a `catch` block is best placed after all possible `catch` blocks. Situations where an ellipsis `catch` block

precedes all other possible handlers, would always have the generic handler executed, ignoring the other what could be case specific exception handlers.

`catch` blocks are searched in the order in which they are listed after the `try` block and the first
one that yields a match is executed. Once the appropriate `catch` block is executed, program execution switches to the first statement after the last `catch` block associated with the currently active `try` block. At any given point in time only one `catch` block gets executed, after which the other following `catch` block, if any are ignored. Cases where there is no match of the thrown exception with the handler types, function `terminate` which indirectly calls abort is called, resulting in program termination. This signifies the fact that the thrown exception cannot be handled by any of the user-defined handlers and hence control is transferred to the system-defined terminate procedure to stop further program execution. Since there can be the situation of several handlers that can handle a specific exception, the order of listing of the `catch` blocks is crucial to the entire process of exception handling.

Similar to the `catch` ellipsis syntax, a `catch` block that catches a base class type object would yield a match for the corresponding derived class object types as well. This is based on the argument that, derived class objects can always be treated as instances of the base class object and such a base class object `catch` block should generally constitute the trailing half of the exception handler codes. Alternatively, instead of having separate classes for different exception, a generic exception class with different private data members associating the different exception types could also be used. In fact, it is this approach that is adopted by the system-defined exception handler classes. There is no conversion (promotion or demotion) of the thrown data types in the process of exception matching and handling.

A `catch` block though similar in nature to a `switch` statement differs in the sense that every `catch` block has its own unique and distinct scope. However, with `switch` constructs, all the cases fall under the same scope of the outlying `switch`. Hence the need for `break` statement within switch-case constructs is alleviated with respect to exception handlers or `catch` blocks. Also, handlers cannot access variables or objects within the `try` block, because once an exception occurs control is exited the `try` for ever and transferred to a matching `catch` block. Throw point cannot be reached back from a `catch` block using a `return` statement.

Exception handlers differ in their approach of handling the exception. Some might decide to terminate program execution, while others might perform recovery operation and resume execution after the last handler. There can also be handlers which convert exceptions for other handlers to handle and process the exception. Ideally, the `catch` block is expected to perform resource clean-up (in the bare minimum). One such operation could be reclaiming memory spaces allocated using the new operator with the delete operation.

In certain cases, a handler or a `catch` block might also decide to rethrow an exception. That is when it is not able to process the exception, it might throw the same exception (rethrow) with a throw (just the keyword throw after an earlier throw operand) for other handlers to process the exception. In such cases, the `catch` blocks associated with the next following `try` block are the one that can possibly handle rethrown exception. Also, `catch` blocks themselves can throw new exceptions to be handled and processed by encompassing `try` associated `catch` handlers. Consider another example for situation handling where the user wishes to have an appropriate handler invoked in cases of failed memory allocation attempts.

One alternative is to check for non-null condition of the allocated block. The code shown in Figure 9.5 handles failure by the new operator to allocate the requisite space by invoking

the what function of the predefined exception class. Note the usage of the new header file to make use of the features supported by the `bad_alloc` class that is employed to generate the message related to failure of new to obtain the requisite blocks of memory. Alternatively, users can as well make use of the `set_new_handler` routine with argument as the customized handler function or routine that is to be called, instead of a case specific `try` and `catch` block syntax. C++ supports the `exception.h header file` with a what message interface support to identify the appropriate error message depending on the exception generated. exception is the base class with several possible derivations associated with errors to check for runtime error, overflow, underflow errors, etc. Interested readers must pursue the respective header file for further application and usage.

```
#include<iostream.h>
#include<new.h>
int main()
{
int * p [200];
try
{
for (int i=0;i<100;i++)
{
p[i]=new int[10000000];
cout<<"Allocated space for p["<<i<<"]";
}}catch (bad alloc exception)
{
cout<<"Exception Occurred \n";
cout<<exception.what();
}
return 0;
}
```

Figure 9.5: Exception handling in cases of new failure.

Review Questions

1. Discuss the need for templates-based programming.
2. Differentiate function from class templates.
3. Develop a function template MAX in C++ that computes the maximum of three integers or three floating points of three characters (based on ascii values).
4. Develop a function template SORT in C++ that sorts either an array of integers or characters or floating point numbers.
5. Develop a Matrix class template in C++ that supports floating or integer matrix arithmetic of addition and subtraction.
6. Develop a Linked List class template in C++ with support for operations insertion, deletion and search of an element. A linked list is basically a collection of elements with the first element pointing to the next and so on, The last element links to NULL. Note in a linked list elements need not be in contiguous memory locations as is the case with arrays.