Chapter 1     The Preliminaries

This chapter briefly discusses the important C++ concepts that will be used throughout the book.

1.1 Introduction

Computer software has become a driving force in all kinds of activities. With each passing day more and more systems are controlled by software. The ever increasing demand has resulted in exponential increase in size and complexity of software systems, requiring better techniques and practices to develop software for these systems. Software engineering is the study and practice of these techniques and tools.

IEEE defines software engineering as:

1. The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

2. The study of approaches as in 1.

Broadly speaking, software goes through three phases in its lifetime. These are: (1) development, (2) operation and maintenance, and (3) retirement. Looking from this perspective, we can say that, software engineering mainly targets the activities corresponding to the first two phases of the software lifetime.

Let us discuss these activities briefly.

Development, as the name suggests, means the phase in the life of the software when it is being built. The development phase can be broken down into many different sub-activities. These include gathering and analyzing requirements (what is to be built), designing the system (how to build it), coding (actually building it), and then testing to ensure that the software achieves the desired objectives.
Operation and maintenance involve providing the infrastructure, training, support, and other resources required for the smooth functioning of the software. This seems just like operating any system or machine. But the similarity ends here as maintenance of software is a totally different ball game as compared to any other system. What is it and why is it different, let us try to understand.

As there are no physical parts involved, software systems do not go through the process of wear and tear which are the root causes for maintenance in other kinds of physical systems. In software, maintenance is required because long lasting software systems go through continuous change – for correcting faults, for adding new features and functionalities, for adapting to new operating environments, and for improving performance or other attributes. All these activities come under the umbrella of software maintenance.

Software maintenance is a very costly activity and it is estimated that for most large, long lifetime software systems, maintenance cost normally exceeds development cost by factors ranging from 2 to 3. For this reason, in most cases, maintainability is considered to be the most desirable feature of a software system. Therefore, many of the software engineering practices aim to bring the cost of maintenance down by improving the maintainability of the system.

Achieving a high degree of maintainability is not an easy task by any means. It requires a conscious effort during all the phases of software development cycle. Neglect or carelessness even in any one activity like requirement specification, design and analysis, coding, testing documentation etc, may ultimately lead to a maintenance nightmare. Therefore, it is of utmost importance that we first understand the factors that help in improving the maintainability of a system and then apply those principles in developing systems that are, relatively speaking, easier to maintain.

So, what is maintainable design and code how do we achieve that? This question is discussed in the next section.
Chapter 1 – The preliminaries

1.2 Characteristics of a maintainable design

A software design is said to be maintainable if the cost of system changes is minimal and it is flexible enough to allow modification to existing functionality and addition of new functionality. Achieving such a design is no easy task but it can be made less painful if the design and code are easy to understand and the changes are, to a large extent, local in effect. The principle of abstraction plays a major role in achieving these objectives. The following subsection discusses this principle briefly.

1.2.1 Abstraction and encapsulation

Abstraction is a technique in which we construct a model of an entity based upon its essential characteristics while ignoring the inessential details. The principle of abstraction thus helps in handling the inherent complexity of a system by allowing the user to work with its important external characteristic and hiding its inner complexity at the same time. Hiding the internal details is called encapsulation. Engineers of all fields, including computer science, employ abstraction for mastering complexity.

In order to understand the concept, let us consider the following example. Let us assume that a variable x occupies four bytes and is stored in memory starting from address 0012FF7C hex. Figure 1 shows the contents in binary of each of the four bytes constituting the variable x. Now the question is: what is x? That is, if you printed x, what would you get?

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0012FF7C</td>
<td>00000000</td>
</tr>
<tr>
<td>0012FF7D</td>
<td>01000011</td>
</tr>
<tr>
<td>0012FF7E</td>
<td>01000010</td>
</tr>
<tr>
<td>0012FF7F</td>
<td>01000001</td>
</tr>
</tbody>
</table>

Figure 1 – Variable x represented in binary
In order to answer this question, you have to know the data type of x as the data types provide the context for interpretation of raw binary data and for different data types the answer would be totally different. Figure -2 shows some possible values for the variable x for different data types.

It is important to note that the answer is dependent upon the underlying hardware, operating environment, and compiler and may change from one system to another.

Fortunately, a programmer has to work only with entities like number 4407873 and string “ABC” without bothering about how it is actually implemented and stored in binary format. That is, as a programmer one only needs to work with entities at a higher level of abstraction rather than worrying about the implementation details. From this example, it is clearly evident that it would have been extremely difficult to write any meaningful programs without the abstraction provided by the data types.

As another example, let us consider the case of a two-dimensional array. A two dimensional array, as conceived by a programmer is a two-dimensional matrix with certain number of rows and columns. An element in the matrix can be accessed by specifying its row and column number. Since the memory in a computer is a linear one-dimensional entity, two-dimensional matrix is usually mapped on to a single dimensional array by putting the rows (or columns) one after the other in a linear fashion. This arrangement is depicted in Figure -3. Once again, fortunately, in order to access an element, the programmer needs to work only with the two-dimensional abstraction. Offset calculation would

<table>
<thead>
<tr>
<th>Data type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>4407873</td>
</tr>
<tr>
<td>float</td>
<td>6.17675 × 10^{-39}</td>
</tr>
<tr>
<td>char*</td>
<td>“ABC”</td>
</tr>
</tbody>
</table>

Figure -2 – possible values of variable x
otherwise be quite difficult, especially in the case of multidimensional arrays.

The examples discussed above elaborate the kind of low-level support for abstraction built into the programming languages. But modern programming languages provide constructs that allow a programmer to apply the principle of abstraction at a much higher level, resulting in programs that are easier to understand and maintain. This support for abstraction is provided at two levels, code and data. These are discussed in the following subsections.

1.2.2 Code and Data Abstraction

As we have discussed earlier, in most cases, maintainability is the most desirable quality of a software artifact. Code is no exception. Good software ought to have code that is easy to maintain. That is, it is not sufficient to write code that works, it is important to write code that works and is easy to understand so that it can be maintained. The three basic principles that guide maintainability are: simplicity, clarity, and generality (or flexibility). The software will be easier to maintain if it is easier to understand and easier to enhance. Simplicity and clarity help in making the code easier to understand while flexibility facilitates easier enhancement of the software.

From a maintenance perspective, what we need is what is called self-documenting code. A self-documenting code explains itself without the
need of comments and extraneous documentation. That is, the meaning of the code should be evident just by reading the code without having to refer to information present outside this code. A number of attributes contribute towards improving the maintainability of a program and producing self-documented code. The size of a programming entity, such as a function, plays a significant role in this respect. In general, as a piece of software becomes larger in size, it becomes more difficult to understand and hence more difficult to maintain. Principles of abstraction and encapsulation are two important tools that can help in managing and mastering the complexity of a system, including reduction in the size of individual programming components.

Software basically has two components – data and code to manipulate the data. From a programming perspective, principle of abstraction can be applied to code as well as to data. Code abstraction is related to algorithmic steps and actions whereas the concept of data abstraction helps us in better organizing and manipulating data.

1.2.2.1 Code abstraction

Code abstraction is mainly achieved with the help of subprograms. It helps us in hiding the micro-level implementation details of an algorithm and lets us focus on the macro-level algorithmic steps required to solve a particular problem. Code abstraction has been the corner stone of programming from the very beginning. Initially, reusability was considered to be the prime reason for making subprograms. However, research in software engineering showed that dividing a program into smaller subprograms for managing the complexity is of equal concern (if not more) and a program should be broken into smaller pieces to have better code-level abstraction, even if those pieces are not reused.

As an example consider the Selection Sort algorithm given in Figure -4. With the help of this example we demonstrate how code abstraction helps in generating code which is easier to understand and hence is more maintainable.
As mentioned in the code, each iteration of the selection sort algorithm has two basic algorithmic steps:

a) find the index \( i \) of the minimum value in the array
b) exchange the minimum value found in step (a) with the value at index \( i \)

We can abstract out these two steps by converting them into subprograms – minimum and swap – and replacing these steps by corresponding function calls. This resultant version of the selection sort is shown in Figure 5. It is easy to see that the new `selectionSort` function is much more readable and does not need any comments as both the statements inside the loop are self explanatory. Hence it is more self-documenting than the first version. It may also be noted that this code for selection sort is not only shorter but also, as a by-product, we now have two functions (minimum and swap) that can be reused. In general, code abstraction yields smaller components that are more generic in nature and hence are more reusable.
From a maintenance viewpoint, abstraction helps us in achieving another very important objective – the ability to replace equal with equals. As code abstraction hides the details of how a subprogram was implemented, therefore a particular subprogram could be replaced by another that has the same overall behavior. For example, a sort function (e.g. selection sort) could be replaced by another sort function (e.g. bubble sort) without any change in the behavior of the program that uses the sort function. This is an extremely powerful concept and can be quite handy in issues relating to performance and portability.

**1.2.2.2 Data Abstraction and Abstract Data Types (ADT)**

As mentioned earlier, code abstraction has been the corner stone of programming from the very beginning and was fundamental to the concept of structured programming. However, it has been shown that data abstraction plays an even bigger and more important role in managing the inherent complexity in computer programs.
Data abstraction separates the implantation of a data type from its abstract properties and allows the user to work with the abstract properties of the data type by making them visible in the form of an interface and hiding the implementation details. As the user program is dependent only upon the interface provided by the data type, any change in the implementation that does not require a change in the interface remains transparent to the user programs. This allows changing the implementation of a data type without disturbing the user programs.

Data abstraction leads to the concept of abstract data types.

A data type is a template for objects and a set of operations that define the behavior of the objects (or instances) of that type. A data type is called an Abstract Data Type (ADT) when it is defined in terms of operations on it and its implementation is hidden. That is, an ADT is a data type in which the implementation details are hidden and the user is only concerned with the properties (or behavior) of that type. It is called abstract because, from a user’s perspective, it is independent of any concrete implementation.

A very obvious example of an abstract data types is the data type int. For a user, int is an abstract representation of integers. The programmer is only interested in its abstract properties - the range of numbers supported by int on a particular computer and the permissible operations on it. How it is actually represented inside the machine and how the supported operations are implemented are details that are hidden and programmer need not worry about these.

From a pure theoretical point of view, an array is also an abstract data type. The basic properties of an array are:

1. we can create an array of a specific size with a specific data type,
2. we can store a value in the array by providing an index of the desired location, and

---

1 Arrays in C++ are not really first class ADT’s as implementation details are not truly hidden. Rather, in many situations, knowledge of implementation details is essential for proper use of arrays.
3. we can retrieve data stored at a specific index in the array by providing the index.

While using an array, the programmer only needs to deal with these properties and is not required to know how an array is actually represented in the system.

In addition to such built-in abstract data types, C++ also provides support for user defined abstract data types and that is elaborated in the next section.

1.3 ADTs and C++ Classes

In C++, classes are used to develop user defined Abstract Data Types. A class consists of data and functions that operate on that data. A class in C++ has two parts – public and private (let’s ignore the protected members for now). The data and functions defined in the class are called the members of the class. Data encapsulation is achieved by declaring all data members of a class to be private. Private members can only be used by other members of the class (let’s also ignore the friends for now). The interface to the class is provided through the use of public member functions. Users of the class can only access and manipulate the class state through the public members of the class.
As an example consider the specification in C++ of an ADT for a Point in a 2-dimensional Cartesian system (Figure 6).

As mentioned earlier, the public part specifies the interface or behavior and the implementation is encapsulated in the private part. For example, in this case the xCoord and yCoord are declared as private. The user of the Point ADT is only concerned with the public interface and cannot directly access these private members that are used in the implementation of the ADT. Thus, any change in the implementation would not affect the user program as long as the interface stays the same. For example, the designer of the point ADT could decide to change the representation of point from Cartesian to Polar without requiring any change in the user program. Figure 7 shows the implementation of the Point ADT, which is usually kept separate from the specification².

² It is good programming practice to keep the specification of a class in .h file and the implementation in the .cpp file.
1.4 Classes with dynamically allocated memory

In general, the basic characteristics of the problem at hand are known. However, at the time of writing the code, the exact size and shape of the particular instance of the problem that is being solved may be unknown. This requires a mechanism through which the structure that is used to organize data can grow dynamically according to the need of the problem. In C++, pointers provide the facility to dynamically allocate memory on-demand and then manipulate this memory area. They give programmers a direct access to the memory and hence provide perhaps the most powerful

```cpp
Point::Point(float x, float y) {
    xCoord = x;
    yCoord = y;
}
float Point::getX() { return xCoord; }
float Point::getY() { return yCoord; }
float Point::distance(Point other) {
    float x = xCoord - other.xCoord;
    float y = yCoord - other.yCoord;
    return sqrt(x*x + y*y);
}
void Point::moveRelative(float x, float y) {
    xCoord = xCoord + x;
    yCoord = yCoord + y;
}
void Point::moveAbsolute(float x, float y) {
    xCoord = x;
    yCoord = y;
}
void Point::print(ostream &str) {
    str << "(" << xCoord << "," << yCoord << ")";
}
```

Figure 7 - Implementation of Point ADT
programming tool. This most powerful tool is also the most dangerous one – pointers, if not handled with utmost care, introduce the nastiest of errors that are very hard to debug and fix. The three most important concerns while dealing with pointers are: (a) the issue of shallow and deep copy, (b) garbage creation or memory leaks, and (c) dangling references. Following is a brief discussion of these topics.

1.4.1 Deep versus shallow copy

Many a times it is required that a copy of the dynamically allocated memory area is made. As shown in Figure 8, simply copying the pointer into another pointer variable does not do the job as it only copies the address of the memory area and both the pointers point to the same physical memory location. This is called shallow copy because the copy of the address rather than the contents of the memory is made.

When the copy is made by first allocating a separate memory area and then copying the contents, it is called deep copy. This is depicted in Figure 9.

```
TypeA *p, *q;
p = new TypeA;
*p = something;
q = p;
```

Figure 8 - Shallow copy
1.4.2 Dangling references and garbage

Space may be acquired at run-time on request from free store\(^3\) using \textit{new} and the address of the memory area so acquired can be stored in a pointer type variable for future references. At a point in time when a memory area pointed to by a pointer is no longer required, it can be returned back to free store using \textit{delete}. We should always return storage after we no longer need it. Once an area is freed, it is improper and quite dangerous to use it.

A memory area that is no longer required if not returned back to the free store, may eventually become inaccessible. Such memory area is called garbage or memory leak. On the other hand, a pointer that has an address of a memory area which is not under its control (it may have been returned back to the free store or it is not initialized and has junk value in it), then it is called a dangling reference. In other words, a dangling reference is an access path that continues to exist after the lifetime of the associated data object. A very common programming error that creates dangling references is outlined below. In order to understand the concept, let us refer back to Figure 8.

\begin{verbatim}
TypeA *p, *q;
p = new TypeA;
*p = something;
q = new TypeA;
*q = *p;
\end{verbatim}

Figure 9 - Deep copy

---

\(^3\) The terms \textit{heap} and \textit{free-store} are used interchangeably when referring to dynamically allocated objects. However, in the context of C++, the two are similar but they are not the same and using “free store” is preferable as it is more conformant with the C++ standard.
At the moment, both p and q point to the same memory area and they can be used quite legitimately to manipulate data stored in the memory area pointed to by these pointers. But situation changes if something like the following is done.

\[
\text{delete } q; \\
q = \text{NULL;}
\]

The programmer seems to have done the proper thing by setting the value of q to NULL after the memory has been returned to the free store. This very innocent looking piece of code, which seems to be correct at a first glance, has however created a dangling reference! In order to understand the problem, we have to remember that p was also pointing to the same memory area. When the memory was deleted, what happened to p? As depicted in Figure 10, p now points to a memory area which has been returned back to the free store and it does not anymore have a legitimate right to use it. It’s a dangling reference! Now an attempt to access the memory area whose address is present in p may result in an undesirable situation, which, at times, can be quite nasty.

Let us now look at the problem of memory leak.

When all access paths to a data object are destroyed but the data object continues to exist, the data object is said to be garbage. Let us consider the code and corresponding diagram of Figure 11.
Let us now execute the following statement:

\[ q = p; \]

The new situation is shown in Figure 12. What happens to the space that was pointed to by \( q \)?

![Figure 12 - Garbage or memory leak](image)

This memory area has not been returned back to the free store but, at the same time, it is no longer accessible. As far as the run-time environment is concerned, this memory area is still in use and hence cannot be assigned against another request for memory allocation. It has virtually “leaked” from the system and is called garbage. Now, for all practical purposes, the system’s memory has been reduced by that much.

In C++, garbage is only reclaimed by the operating system after the termination of the program.

Garbage slowly eats up the memory and perpetual accumulation of garbage slows down the system to an extent that no effective work can be done and the corresponding program needs to be terminated.

### 1.4.3 Classes and dynamic memory

If not created by the programmer, C++ compiler automatically creates four member functions for a class. These are: (1) the default constructor, (2) the destructor, (3) the copy constructor, and (4) the operator=.

When dynamically allocated memory is used inside classes, the last three (the destructor, the copy constructor, and assignment operator) become extremely important. In that case the programmer is required to explicitly write these three methods or face the pointer related problems mentioned
earlier. Let us try to understand these issues with the help of a simple example of a class given in Figure 13.

As can be easily seen, in this example, the name of the student is stored in a dynamically allocated memory area pointed to be the studentName. Let us now see what happens when the destructor, copy constructor, and operator= are not implemented by the programmer.

1.4.3.1 The Destructor

A destructor is a member function with the same name as its class prefixed by a ~ (tilde). The destructor of a class is called when an object of that type is destroyed. There are two basic scenarios: (1) when an object goes out of scope, or (2) when an object pointed to by a pointer is destroyed explicitly by using the delete operator.

For most cases, the default destructor is enough. However, when the object has some dynamically allocated memory, the default destructor is
not sufficient. It does not return the dynamically allocated area back to the free store. As a result, as depicted in Figure 14, garbage is created in the process. Therefore, when dynamically allocated memory is used inside a class, the programmer must write a destructor or garbage would be accumulated. Writing a destructor in this case is easy – all you have to do is explicitly delete the dynamically allocated memory as shown below:

```cpp
Student::~Student()
{
    delete [] studentName;
}
```

### 1.4.3.2 The Copy Constructor

The copy constructor is a special constructor that is used to make a copy of the object. It is called implicitly to make a hidden temporary copy of the object in three situations listed below:

a) It is called when an object is initialized by another object at the time of declaration. For example, the copy constructor will be implicitly called in the statement shown below:

```cpp
TypeA   a = b;
```
In this case a temporary copy of $b$ will be made to initialize $a$ and that copy will be destroyed at the completion of the job.

It may be noted that the copy constructor will not be called in the following scenario:

```cpp
TypeA   a;
a = b;
```

In this case the assignment operator (operator=) will be used. Assignment operator is similar to the copy constructor but there are certain differences that we will discuss later.

b) The second place where the copy constructor is implicitly called is when an object is returned by value from a function. As an example, consider the following code:

```cpp
TypeA foo(){
    TypeA x;
    // something is done here
    return x;
}
```

Just before returning from the function, a copy of $x$ will be made. That copy will be used to copy the data in the target. Once again this implicitly created temporary object (in this case copy of $x$) will be destroyed when the job is complete and the function terminates.

c) The third place for the use of copy constructor is when an object is passed by value as a parameter to a function.

As mentioned above, the life time of the implicitly created temporary object by the copy constructor is limited to the life time of the corresponding operation. At the completion of the required job, this object is destroyed. This creates problems when dynamic memory is involved.
The default copy constructor generated by the compiler simply copies the class data members. That is, if a pointer was involved, it would make a shallow copy whereas, in most cases, a deep copy would be required. This scenario is shown in Figure 15.

But that is not all – the destructor, if implemented, plays a nasty role here and joins hands with the default copy constructor in creating dangling references.

When the destructor destroys the implicitly created copy, it also disposes-off the dynamically allocated memory area. But if a shallow copy of the object was made, destroying the copy would destroy the original because the same memory area was shared by both. As shown in Figure 16, this creates a dangling reference.

Therefore, in order to avoid such situations it is required to make a deep-copy of the object. Hence, the programmer must write a copy constructor when dynamically allocated memory is involved. Once again, writing the copy constructor is easy – just make sure to allocate memory explicitly and then copy the data. The copy constructor for the Student is shown below:
Student::Student(const Student& other)
{
    makeCopy(other.studentName, other.creditsEarned);
}

The only other point to note here is that the reference of the object whose copy is to be made is passed as parameter in the copy constructor. It is important to do that otherwise if the object is passed by value, its copy will be made which will call the copy constructor again and hence will result in an infinite recursive call of the copy constructor.

1.4.3.3 The assignment operator (operator=)

The assignment operator comes into action when value of one object is assigned to another object. In this case, just like the copy constructor, the default assignment operator simply makes a shallow copy. Therefore, once again the programmer must write an overloaded assignment operator when dynamically allocated memory is involved. Overloading the assignment operator is not much different from the copy constructor and is written below:

```cpp
const Student& Student::operator=(const Student& rhs)
{
    if (this != &rhs) {
        delete[] this->studentName;
        makeCopy(rhs.studentName, rhs.creditsEarned);
    }
    return *this; // return self-reference so cascaded assignment works
}
```

The important points to note are:

(a) The memory already allocated to the object on the left hand side of the assignment operator must be deleted to avoid memory leaks.

(b) Before deleting memory it is important to check for self assignment (like a = a;). This is done by comparing the this pointer with the address of the object on the right hand side. Not doing so will create dangling references when an object is assigned to itself.
(c) Reference to self is returned back to allow cascading (e.g. \( a = b = c; \)) of assignments.

Figure 17 shows the scenario when all three methods mentioned above are properly written.

### 1.5 Error Handling

Except for very simple programs, it is common for a program to fail because of some rare condition occurring at run-time. This could be the result of a logical programming error, bad input data, unavailability of a resource, or some kind of a hardware failure. Simply terminating the program in such cases may result in the loss of some critical data or even a catastrophic disaster. Therefore, a programmer is required to design
programs that can either recover from such error and then continue after taking the corrective measure or at least shut down gracefully to minimize the amount of damage. Proper handling of run-time errors is thus considered a key distinction between high quality professionally written programs and a poor amateurishly written code. To handle run-time errors two basic techniques are used. These are error code testing and exception handling. A brief discussion of these follows.

### 1.5.1.1 Error code testing

Error code testing is a very common error handling technique. In this approach, functions are written to return values to indicate success or failure. Values that indicate failure are called error codes. The programmer can then test for those return values and decide what to do next. This concept is elaborated with the help of the following example. In this example, the program would go into an infinite loop if, at any stage, the data entered by the user was non-numeric.

```cpp
int i = 0;
while (i != -1)
{
    cout << "Enter a number (-1 to exit): ";
    cin >> i;
    if(i != -1) {
        cout << "The number is: " << i << endl;
    }
}
```

If the user enters something like “abc”, the input stream will go into an error state and will not work properly until it is cleaned-up and flushed. In order to avoid such problems, the user should check for the validity of the input by testing the input stream for failure and then take the corrective measure as shown next:
int i = 0;
while (i != -1)
{
    cout << "Enter a number (-1 to exit): ";
    cin >> i;
    if (cin.fail()) {
        cout << "Not a valid number!" << endl;
        cin.clear();
        cin.ignore(1);
    } else if (i != -1) {
        cout << "The number is: " << i << endl;
    }
}

If used properly, error code testing is a useful and effective mechanism to
detect and recover from errors. However, its effectiveness is easily
compromised if a programmer fails to check the error code. In many such
cases, ignoring an error code results in nasty problems that appear to have
occurred at a place which is far removed from the actual problem area and
are thus extremely difficult to debug and fix. For example, failing to check
for NULL value as a result of using the malloc function may consequently
lead to an inappropriate use of the pointer, hence resulting in program
break and loss of data. In addition, if we use error code testing to handle
errors, recovery from deeply nested function calls may also be a non-
trivial exercise.

1.5.1.2 Exception handling

Modern programming languages offer an alternative approach, called
exception handling, to handle run-time errors. In most situations, this
provides a much simpler and cleaner strategy for error handling as
compared to error code testing. In addition, exceptions are much harder
to be ignored by the programmers and hence help in producing code
which is more robust and is easier to debug.

Exception handling mechanism is briefly discussed next as a detailed
discussion on the topic is beyond the scope of this book.
Exception handling framework in C++ is based upon three activities: (a) indicating an exception, (b) monitoring for exception condition, and (c) handling the exception. The language provides the support for these activities through throw, try, and catch statements respectively. The throw statement is used to indicate the exceptional condition that needs to be taken care of. To catch an exception, code must be written inside a try block and the code to handle any exceptions occurring in the try block is written in a corresponding catch block. To handle different types of errors arising from a single try block, there may be more than one corresponding catch blocks. All catch blocks related to a try block are written one by one immediately after the try block. In order to understand, let us consider the code segment of Figure 18.
In this example, the main program calls the function exceptionExample within its try block. exceptionExample calls mightThrow which might throw an exception if cin goes into an error state. If the exception is raised, the control goes back to a catch block which is equipped to handle that particular type of error. So, if INPUT_ERROR is thrown by mightThrow, then the control returns to exceptionExample. As exceptionExample does not have a catch block to handle INPUT_ERROR, the control goes back to the main program without printing the value of x. Associated with the try block, there are two catch blocks in the main function. The first one can handle exceptions of type MyExceptionType and the second one can catch any exception. Assuming that INPUT_ERROR is of type MyExceptionType, it will be handled by the first catch block and the necessary corrective measure can be taken at that point.

In addition, exception has another advantage over error codes. As a constructor cannot return a value, hence if it fails error codes cannot be used. Therefore throwing an exception is the only real option to report an error from a constructor.

For these reasons, rather than using error codes, we have taken the exception handling approach. Therefore, throughout this book, error conditions will be indicated by throwing an exception and it will be up to the user programs to handle these exceptions and use an appropriate strategy to recover from these error conditions.

Exception handling is an involved topic and, as stated earlier, a detailed discussion on this topic is beyond the scope of this book. An interested reader is therefore advised to look at some of the reference material mentioned at the end of the chapter.

1.6 Generic Programming

Let us revisit our Selection Sort function once again. It sorts an array of integers in ascending order. If we wanted to sort an array of floating point numbers, we would have to rewrite the function. The only difference in the two functions would be the data types of some variables and
parameters. It may be noted that in the case of second version of the selection sort algorithm, all three functions would need to be rewritten with data types being the only changes required.

As an illustration, the two swap functions are given in Figure 19.

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

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

*Figure 19 – Duplicated code because of change in data type*

It may be noted that function overloading allows us to use the same name for the function and hence that does not need to be modified.

It is obviously inconvenient to write the same code again simply because there was a change in the data type (though it is mostly copy-paste). From a maintenance point of view, it is a nightmare – every time there is some change required in the code for whatever reason, the same change has to be applied to all the different versions of the same code. It is not very unusual that some version of the code is missed in the process, hence making it inconsistent and faulty.

From this perspective, it would have been quite useful if, just like function parameters, data types could also be parameterized. Generic programming is a mechanism to do exactly that.

In C++, support of generic programming is provided through *templates*. In this case, the generic entity is declared as *template* and it takes special parameters through which types can be passed as arguments. These parameters are used inside the entity as if they were any other regular type. C++ supports two kinds of templates, function and class. Both of these kinds of templates are briefly discussed in the following subsections.
1.6.1 Function templates

A function template defines a function in which types of function parameters and return type are parameterized. Thus, with the help of templates, the swap function can be written generically as shown in Figure 20.

In this example we have created a template function with T as its template parameter. It is a placeholder for whatever type is specified when swap is called.

To use swap with a concrete type we can either specify the type explicitly, or let the compiler figure it out. So, assuming that x and y are two integers, we can either write

\[
\text{swap} \ <\text{int}\> (x, y); \quad // \text{type explicitly specified}
\]

or

\[
\text{swap}(x, y); \quad // \text{type to be determined by the compiler}
\]

When the compiler encounters a call to a function template, it uses the template to automatically (and transparently) generate a function replacing each occurrence of the dummy type parameters by the type passed as the actual template parameter (int in this case) and then calls it.

1.6.2 Class templates

Like function, many classes are also intrinsically type independent. In fact, container classes used for implementing different data structures that we will study later are independent of the type of the data that is stored in these entities. A classical example of such an entity is an array. In C++, class templates facilitate the development of such classes.
Following is an example of a generic container, SmartArray, which is essentially a one-dimensional array with support for bound checking in it. Since an array can store elements of any type, SmartArray needs to be made generic. Therefore, we will create a class template for SmartArray and then we can use it instead of a single dimensional array.

### 1.6.2.1 SmartArray Specification

Specification of the SmartArray is given in Figure 21. It shows that it is a class template where the data type to be stored in the array is parameterized. In the private part of the specification we have three data members. mArray is a pointer to the memory area used for storing the data in the SmartArray. mSize maintains the size of the array and will be used in checking the array bounds. The role of mRefCount will be discussed later. In addition, the operator= is made private to disallow assignment and checkBounds is a private method that will be used internally by other member functions of the class. Except for the differences enumerated below, a SmartArray will behave exactly like an array. The differences are:

```cpp
template <class T>
class SmartArray
{
public:
    SmartArray(int n); // constructor
    SmartArray(const SmartArray &other); // copy constructor
    ~SmartArray(); // destructor
    T& operator[] (int index); // overloaded subscript operators
    const T& operator[] (int index) const;

private:
    int mSize; // size of the array
    T *mArray; // array to be created dynamically
    int *mRefCount; // to be used in the copy
                     // constructor and destructor
    SmartArray& operator =(const SmartArray& other); // assignment operator is declared
    void checkBounds(int index);
};
```

Figure 21 – Class template Specification
a) It has bound checking in it.
b) Syntax for declaring an entity of SmartArray type is different.
c) An expression (to be evaluated at run-time) can be used in its declaration.

The implementation of the SmartArray class template requires some special attention.

In order to make it behave like a normal array, we need to take the following measures:

a) To disallow a declaration of the form
   
   ```
   SmartArray<int> x;
   ```
   
   the default constructor is not written.

b) As mentioned above, in order to disallow assignment of one SmartArray to another, the assignment operator is declared as private. This is sufficient to achieve the desired objective and no implementation of the `operator=` is required. It is important to note that if the assignment operator is not overloaded, default assignment operator will be used, which will yield undesirable results in this case.

c) The effect of passing a SmartArray to a function should be similar to a normal array. That is, it should behave as if it was passed by reference so that any changes made to any of its elements inside the function should be reflected in the original SmartArray that was passed as the argument to the function.

1.6.2.2 SmartArray implementation – A copy constructor that makes a shallow copy

Usually, the copy constructor is written to facilitate pass by value. However, as mentioned above, in our case we want it to support pass by reference. Therefore we need to write it differently.

In order to achieve this effect, a shallow copy of the object is made. The resultant copy constructor is shown next:
On a cursory glance it appears that the default copy constructor would have done exactly the same therefore there really was no need to write our own copy constructor. However, there is some complication in this case that makes it necessary to write a copy constructor to make a shallow copy of the data. In order to understand the difficulty, the role of the destructor needs to be revisited. Implementation of the constructor and destructor and are discussed next.

1.6.2.3 SmartArray implementation – The destructor and the constructor

You may recall that the destructor is called when the object of a class is destroyed for any reason. As mentioned earlier, the temporary copy made at the time when an object is passed by value is destroyed when the function terminates. As we want to achieve the effect of pass by reference, the copy and the original object would share the same data. Therefore, when the function returns, invocation of the destructor will destroy the copy and hence the original object will also be destroyed. This will create a dangling reference.

In order to avoid that situation, we have used a reference count. The reference count, maintained in the dynamic variable pointed to by the pointer mRefCount, is initialized to 1 when an object of type SmartArray is declared. It is shared by all the copies of this object (a shallow copy of mRefCount is made) and is incremented by one when a copy of the object is made. When the destructor is called, it is decremented by one and the array is deleted only when it is zero. That is, the array containing the data is deleted when no other copy of the object exists. The resultant constructor and destructor are shown next:
template<class T>
SmartArray<T>::SmartArray(int n) 
{ 
   if (n < 1) throw ILLEGAL_ARRAY_SIZE;
   else {
      mSize = n;
      mArray = new T[mSize];
      if (mArray == NULL) throw OUT_OF_MEMORY;
   }
   mRefCount = new int;
   if (mRefCount == NULL) throw OUT_OF_MEMORY;
   *mRefCount = 1;
}

template<class T>
SmartArray<T>::~SmartArray() 
{ (*mRefCount)--; 
   if (*((mRefCount) == 0))
      delete [] mArray;
}

1.6.2.4 SmartArray implementation – Overloading the subscript operator

Finally, in order to support the same syntax in SmartArray to access the elements directly just like regular arrays, we need to overload the subscript operator (operator[]). The subscript operator, [], must be a member function.

In order to allow the use an array that has been passed to a function as a constant in an expression, we need to overload it twice – one for providing the read-only access and the other for read-write access so that it can also be used on the left hand side of the assignment. The former is implemented by returning a constant reference of the element at the specified index and the latter by returning the non-constant reference of the element at the given index. Except for the difference in the return type, these two versions of the overloaded subscript operator would look exactly the same. From the context, the compiler will automatically
decide which particular function needs to be called and it is totally transparent to the user. These two overloaded functions are given below:

```cpp
template<class T>
const T& SmartArray<T>::operator[] (int index) const
{
    checkBounds(index);
    return mArray[index];
}

template<class T>
T& SmartArray<T>::operator[] (int index) {
    checkBounds(index);
    return mArray[index];
}
```

The bound checking function is trivial and is shown below:

```cpp
template<class T>
void SmartArray<T>::checkBounds (int index) {
    if (index < 0 || index >= mSize)
        throw OUT_OF_BOUNDS;
}
```

1.7 Summary

In this chapter we discussed the importance of maintenance in the software development lifecycle and then looked at different tools and techniques available in C++ for developing data structures with high degree of maintainability. The main points discussed in the chapter are enumerated below:

1. After being put into use, long-lasting software goes through continuous change. Maintenance cost normally exceeds development cost by factors ranging from 2 to 3. Thus reduction in the cost associated with maintenance could significantly reduce the overall cost associated with the software system. Maintainability of the
system can improve if the design and code is understandable and the changes are, to a large extent, local in effect. The three basic principles that guide maintainability are: simplicity, clarity, and generality.

2. The principle of abstraction plays a major role in achieving these objectives. In C++, support for abstraction is provided at the code as well as at the data level. Code abstraction is mainly achieved by breaking a larger function into smaller subprograms. In C++, classes are used to achieve data abstraction through user defined Abstract Data Types (ADT’s).

3. When working with dynamically allocated memory, one has to be careful about memory leaks and dangling pointers. When a class uses dynamic memory, the programmer must write the destructor, copy constructor, and operator=.

4. Error handling can be achieved by using error code checking or exception handling. Error codes can be ignored by the programmers whereas exceptions must be caught and processed. Exception handling thus provides a more robust error handling mechanism.

5. Generic programming is very powerful feature of the C++ programming language that allows the development of data type independent algorithms and data structures through the use of templates. This rids the programmer to write the same (similar) code again and again for different data types. Generic algorithms are achieved through function templates whereas data structures use class templates.
1.8 Exercises

1) Execute the following program and explain the output. You may need to include appropriate header files for the required libraries.

```c
int main( )
{
    int   i, *pi;
    float f, *pf;
    i = 1024;
    pi = &i;
    pf = (float *) pi;
    f = *pf;
    cout << i << " " << f << "\n";
    f = i;
    cout << i << " " << f << "\n";
    return 0;
}
```

2) Write a function in C++ to swap two entities of arbitrary type without using templates.

3) Write function template for binary search.

4) Convert the selection sort given in Figure 5 into a template based version.

5) Write a template based function to insert data into a sorted list.

6) Use the function developed in Q 4 to write a template based function for insertion sort.

7) Develop a class in C++ for complex numbers.

8) Develop a class in C++ to store a set of points in an array that is created dynamically in a constructor.

9) Develop a generic class for two-dimensional array with bound checking.

10) The C++ standard library includes a smart pointer called auto_ptr. These types of pointers are said to be safe from the usual pointer related issues. Study auto_ptr and compare it with C++ pointers.

11) Rewrite SmartArray (section 1.6.2) using auto_ptr.
12) Identify and correct errors in the following class

```cpp
class X {
private:
    const int size;
    int *storage;
    int count;
public:
    X (int _size) : size(_size) {
        if (size < 1) throw ILLEGAL_SIZE;
        else {
            storage = new int[size];
            if (storage == NULL)
                throw OUT_OF_MEMORY;
            count = 0;
        }
    }
    void insert(int data) {
        if (count >= size) throw OUT_OF_SPACE;
        storage[count] = data;
        count++;
    }
    void print() {
        for (int i = 0; i < count; i++)
            cout << storage[i] << " ";
        cout << endl;
    }
};
```