ADVANCED GUIDE TO

LEARN C++

PROGRAMMING EFFECTIVELY

BENJAMIN SMITH

=main;char
(char*)O?*(char*
ass Hell
world"
r_eac
c;});o
hello
[]=",!(
O==
ar*)O
tic std::s
string
am& operator <<

C++

Advanced Guide to Learn C++ Programming Effectively

Table of Contents

Introduction

I want to thank you for choosing this book, *'C++ - Advanced Guide to Learn C++ Programming Effectively,'* and I hope you find the book informative.

If you have read the previous book, you have gathered a basic idea of some concepts in C++ and how you can use loops and conditional statements to address different problems. This, however, does not mean you have mastered the art of programming in C++. You need to have more information to help you write robust programs and applications. This book will shed some light on some advanced topics in C++, which will enhance your understanding of C++.

The book will shed some light on the references and pointers in C++ and their importance. It also provides information on data structures and how you can use them in C++. Since object-oriented programming (OOP) is an important concept in most programming languages, this book sheds some light on what it is and the various concepts in OOP.

In this book, you will learn more about how you can optimize the performance of your code. When you write any code, you need to test it to determine if it runs correctly. You need to find the errors in your code and find a way to overcome those errors. So, what are you waiting for? Grab a copy of this book now and get started. By the end of the book, you will learn how to write code and improve it, so there are no errors and issues when you compile the code.

Chapter One: Using Pointers in C++

Pointers make it easier to perform specific types of tasks in C++. They are easy to use, and it is best to use them to perform activities or tasks, such as dynamic memory allocation. We have looked at the basics of memory allocation in the previous book. This chapter will shed some light on how best you can use pointers in C++.

Every variable you enter into a program or code will be stored in a memory location. Each location has its own address, and these addresses can be accessed in the code using the '&' operator. This operator denotes that section in the memory where the variable is stored. Let us look at the following example to see how you can print the location or every variable defined in the code.

```
#include <iostream>

using namespace std;

int main () {

   int  var1;

   char var2[10];

   cout << "Address of var1 variable: ";

   cout << &var1 << endl;

   cout << "Address of var2 variable: ";

   cout << &var2 << endl;

   return 0;

}
```

When you compile the code written above, you obtain the following output:

```
Address of var1 variable: 0xbfebd5c0

Address of var2 variable: 0xbfebd5b6
```

The terms 0xbfebd5c0 and 0xbfebd5b6 are the locations in the memory where these variables are stored.

Introduction to Pointers

Before we look at how you can use pointers, let us first understand what a pointer is. Pointers are variables that take the address of a different variable in the code. The syntax of a pointer is as follows:

```
type *var-name;
```

The keyword type in the above syntax is the data or base type of the pointer. Make sure the type is a valid data type in C++. The value var-name is the pointer's name. You need to use the asterisk in the syntax when you define the pointer. C++ throws an error if you forget to use it. The following are some methods to define pointers.

//The following statements are used to define or declare integer, double, float, and character pointers.

```
int    *ip;

double *dp;

float  *fp;

char   *ch;
```

Pointers will only take hexadecimal values since they only take the values of the variables you point them to. You can define a pointer as an integer, double, character, string, etc., but it only represents an address in the memory. The only difference is that when you assign a data type to a pointer when you define it, you indicate to the compiler that you are pointing to a variable with the same data type.

How to Use Pointers in C++

You can perform different operations in C++ using pointers:

1. Defining a pointer variable
2. Assigning the pointer with a variable whose address it stores
3. Accessing the value present in the memory location stored in the pointer

You can perform these operations using the operator '*' which indicates to the compiler that it needs to return the value of the variable stored at the memory location or address indicated by the pointer. The following example uses these operations:

```
#include <iostream>


using namespace std;


int main () {

   int  var = 20;   // actual variable declaration.

  int  *ip;        // pointer variable


   ip = &var;       // store address of var in pointer variable


   cout << "Value of var variable: ";
```

```
    cout << var << endl;

    // print the address stored in ip pointer variable
    cout << "Address stored in ip variable: ";
    cout << ip << endl;

    // access the value at the address available in pointer
    cout << "Value of *ip variable: ";
    cout << *ip << endl;

    return 0;
}
```

When you run the code and compile it, you obtain the following output:

```
Value of var variable: 20
Address stored in ip variable: 0xbfc601ac
Value of *ip variable: 20
```

Types of Pointers

It is easy to understand how you can use pointers in C++. Having said that, if you make mistakes when you use them in your code, you will receive multiple errors. The following are some concepts to bear in mind when it comes to pointers:

S. No.	**Concept**	**Description**
1	Null Pointers	You can use null pointers in C++. This pointer is a constant variable that has a value of zero defined in numerous libraries used in C++.
2	Pointer Arithmetic	You can use the following operators on pointers: 1. ++ 2. + 3. – 4. - -
3	Pointer vs. arrays	There is a very close relationship between arrays and pointers.
4	Arrays of pointers	If you do not want to introduce numerous variables for pointers, you can create an array to store the same data type

		pointers.
5	Pointer to pointer	C++ allows you to use one pointer to indicate to another pointer.
6	Passing a pointer as an argument in a function	You can pass pointers as arguments in functions using either a reference or address. These allow the compiler to pass the pointer as the argument in the function.
7	Returning pointers from functions	You can use a function to indicate a local variable to store the value of the pointer. You can use: 1. A static variable 2. A local variable 3. Dynamically allocated memory

Chapter Two: References in C++

Unlike pointers, references are used as aliases in C++. a reference is used to refer to a variable present in the existing code. When you initialize a reference and assign it to a variable in the code, you can use the variable itself or the reference variable to call the value stored in the variable if you need to use it in a different function.

Difference Between References and Pointers

People often confuse themselves when it comes to references and pointers. There are three differences between the two:

1. As mentioned in the previous chapter, you can have a null pointer, but you cannot have a null reference in your code. Make sure the reference is always tagged to a variable or function which has a return value.
2. When you initialize and assign a reference to a specific object, you cannot change its value to another object in the code at any point. You can use pointers to look at different objects at varied points in the code.
3. Every reference needs to be initialized before it is tagged to any variable. Unlike pointers, you cannot initialize a reference in any line of the code.

How to Create References

From the first book, you know that every variable has a name. Let us assume that this name is the label attached to the location of the variable's value in the memory. When you tag a reference to the variable, it becomes the second label attached to the location. Therefore, you can refer to the value in the memory location using either the reference or the original variable name. Let us consider the following example:

```
// Initialize a variable 'i' and assign it a value

int i = 4;

float j = 2.8;

// Declare a reference variable in your code for the above variable

int &r1 = i;

float &r2 = j;
```

The ampersand (&) in the above line is your reference. Read the above two lines of code as follows:

1. The integer reference, r1, has been initialized and tagged to the variable i
2. The integer reference, r2, has been initialized and tagged to the variable j

In the following example, we look at how you can use references on variables with the data types double and int.

```
#include <iostream>
```

```
using namespace std;

int main () {

   // The following statements are used to declare the simple variables in the code

   int    i;

   double d;

    // The following statements are used to declare and assign the reference variables to the simple variables

   int&    r = i;

   double& s = d;

    i = 5;

   cout << "Value of i : " << i << endl;

   cout << "Value of i reference : " << r  << endl;

    d = 11.7;

   cout << "Value of d : " << d << endl;

   cout << "Value of d reference : " << s  << endl ;

    return 0;

}
```

When you compile the above code, you will obtain the following output:

```
Value of i : 5

Value of i reference : 5

Value of d : 11.7

Value of d reference : 11.7
```

Coders often use references as function return values or argument lists. The following are two points to bear in mind when you write code in C++:

S. No.	**Concept**	**Description**
1	Using references as function parameters	You can pass references as parameters in functions. It is safer to use them as parameters instead of using simple variables
2	Using references as function values	References can be used like other parameters or data types as return values

Chapter Three: Introduction to Data Structures in C++

C++ allows you to use different variables and structures, such as arrays and lists. We have looked at these in brief in the first book. This chapter introduces the different ways you can use these data structures to perform different activities in C++. You can use arrays to define different variables or combine different elements across the program or code into one variable, as long as they fall into the same category. A structure, however, allows you to combine different variables and data types. You can use a structure to define or represent records. Let us assume you want to track the books on your bookshelf. You can use a structure to track various attributes of every book on your shelf, such as:

1. Book ID
2. Book title
3. Genre
4. Author

The Struct Statement

You need to use the struct statement to define a structure in your code. This statement allows you to develop or define a new data type for your code. You can also define the number of elements or members in the code. The syntax of this statement is as follows:

```
struct [structure tag] {

   member definition;

   member definition;

   ...

   member definition;

} [one or more structure variables];
```

It is not mandatory to use the structure tag when you use the statement. When you define a member in the structure, you can use the variable definition method we discussed in the previous book. For instance, you can use the method int i to define an integer variable. The section before the semicolon in the struct syntax is also optional, but this is where you define the structure variables you want to use. Continuing with the example above, let us look at how you can define a book structure.

```
struct Books {
   int book_id;
   char book_title[50];
   char genre[50];
   char author[100];
} book;
```

How to Access Members

Once you define the structure, you can access it using a full stop, which is also called the member access operator. This operator is used as a period or break between the structure member and the variable name. Make sure to enter the variable name you want to access. You can define the variable of the entire structure using the struct keyword. Let us look at an example of how you can use structures:

```
#include <iostream>
#include <cstring>
using namespace std;
struct Books {
   int book_id;
   char book_title[50];
   char genre[50];
   char author[100];
};
```

```
int main() {
    struct Books Book1; // This is where you declare the variable Book1 in the Book structure
   struct Books Book2; // This is where you declare the variable Book2 in the Book structure
// Let us now look at how you can specify the details of the first variable
    Book1.book_id = 120000;
    strcpy( Book1.book_title, "Harry Potter and the Philosopher’s Stone");
    strcpy( Book1.genre, "Fiction");
    strcpy( Book1.author, "JK Rowling");
// Let us now look at how you can specify the details of the second variable
    Book2.book_id = 130000;
    strcpy( Book2.book_title, "Harry Potter and the Chamber of Secrets");
    strcpy( Book2.genre, "Fiction");
    strcpy( Book2.author, "JK Rowling");
// The next statements are to print the details of the first and second variables in the structure
    cout << "Book 1 id: " << Book1.book_id <<endl;
    cout << "Book 1 title: " << Book1.book_title <<endl;
    cout << "Book 1 genre: " << Book1.genre <<endl;
    cout << "Book 1 author: " << Book1.author <<endl;
    cout << "Book 2 id: " << Book2.book_id <<endl;
    cout << "Book 2 title: " << Book2.book_title <<endl;
    cout << "Book 2 genre: " << Book2.genre <<endl ;
    cout << "Book 2 author: " << Book2.author <<endl;

    return 0;

}
```

The code above will give you the following output:

```
Book 1 id: 120000

Book 1 title: Harry Potter and the Philosopher's Stone

Book 1 genre: Fiction

Book 1 author: JK Rowling

Book 2 id: 130000

Book 2 title: Harry Potter and the Chamber of Secrets

Book 2 genre: Fiction

Book 2 author: JK Rowling
```

Using Structures as Arguments

You can use structures as arguments in a function similar to how you pass a pointer or variable as part of the function. You need to access the variables in the structure in the same way as we did in the example above.

```
#include <iostream>

#include <cstring>

using namespace std;

void printBook( struct Books book );

struct Books {

    int book_id;

    char book_title[50];

    char genre[50];
    char author[100];
};
int main() {
   struct Books Book1; // This is where you declare the variable Book1 in the Book structure
    struct Books Book2; // This is where you declare the variable Book2 in the Book structure
// Let us now look at how you can specify the details of the first variable
    Book1.book_id = 120000;
    strcpy( Book1.book_title, "Harry Potter and the Philosopher’s Stone");
    strcpy( Book1.genre, "Fiction");
    strcpy( Book1.author, "JK Rowling");
// Let us now look at how you can specify the details of the second variable
    Book2.book_id = 130000;
    strcpy( Book2.book_title, "Harry Potter and the Chamber of Secrets");
    strcpy( Book2.genre, "Fiction");
    strcpy( Book2.author, "JK Rowling");
// The next statements are to print the details of the first and second variables in the structure
    printBook( Book1 );
    printBook( Book2 );
    return 0;
}
void printBook(struct Books book ) {
    cout << "Book id: " << book.book_id <<endl;

    cout << "Book title: " << book.book_title <<endl;

    cout << "Book genre: " << book.genre <<endl;

    cout << "Book author: " << book.author<<endl;

}
```

When you compile the code written above, you receive the following output:

```
Book 1 id: 120000

Book 1 title: Harry Potter and the Philosopher’s Stone

Book 1 genre: Fiction

Book 1 author: JK Rowling

Book 2 id: 130000

Book 2 title: Harry Potter and the Chamber of Secrets

Book 2 genre: Fiction

Book 2 author: JK Rowling
```

Using Pointers

You can also refer to structures using pointers, and you can use a pointer similar to how you would define a pointer for regular variables.

```
struct Books *struct_pointer;
```

When you use the above statement, you can use the pointer variable defined to store the address of the variables in the structure.

```
struct_pointer = &Book1;
```

You can also use a pointer to access one or members of the structure. To do this, you need to use the -> operator:

```
struct_pointer->title;
```

Let us rewrite the example above to indicate a member or the entire structure using a pointer.

```
#include <iostream>

#include <cstring>

 using namespace std;

void printBook( struct Books *book );

struct Books {

   int book_id;

   char book_title[50];

   char genre[50];

   char author[100];

};

int main() {

  struct Books Book1; // This is where you declare the variable Book1 in the Book structure

   struct Books Book2; // This is where you declare the variable Book2 in the Book structure

// Let us now look at how you can specify the details of the first variable

   Book1.book_id = 120000;

   strcpy( Book1.book_title, "Harry Potter and the Philosopher’s Stone");

   strcpy( Book1.genre, "Fiction");

   strcpy( Book1.author, "JK Rowling");

// Let us now look at how you can specify the details of the second variable

   Book2.book_id = 130000 ;
```

```
    strcpy( Book2.book_title, "Harry Potter and the Chamber of Secrets");
    strcpy( Book2.genre, "Fiction");
    strcpy( Book2.author, "JK Rowling");
```

// The next statements are to print the details of the first and second variables in the structure

```
    printBook( Book1 );
    printBook( Book2 );
    return 0;
}
// We will now use a function to accept a structure pointer as its parameter.
void printBook( struct Books *book ) {
    cout << "Book id: " << book->book_id <<endl;
    cout << "Book title: " << book->book_title <<endl;
    cout << "Book genre: " << book->genre<<endl;
    cout << "Book author: " << book->author <<endl;
}
```

When you write the above code, you obtain the following output:

```
Book id: 120000
Book title: Harry Potter and the Philosopher’s Stone
Book genre: Fiction
Book author: JK Rowling
Book id: 130000
Book title: Harry Potter and the Chamber of Secrets
Book genre: Fiction
```

Book author: JK Rowling

Typedef Keyword

If the above methods are a little tricky for you, you can use an alias type to define a structure. For instance,

```
typedef struct {

    int book_id;

    char book_title[50];

    char genre[50];

    char author[100];

} Books;
```

This is an easier syntax to use since you can directly define all the variables in the structure without using the keyword ‘struct.’

```
Books Book1, Book2;
```

You do not have to use a typedef key only to define a structure. It can also be used to define regular variables.

```
typedef long int *pint32;

pint32 x, y, z;
```

The type long ints point to the variables x, y and z.

Chapter Four: Introduction to Object-Oriented Programming in C++

The objective of the development of C++ was to add the concept of object-oriented programming to the C programming language. The classes used in C++ programming are the variables or structures used in object-oriented programming. These classes are often termed as user-defined data types. These concepts were discussed in brief in the previous chapter.

Classes are used to define the form and type of object used in the code. This data type uses both data methods and representation to manipulate the data present in the memory into one package. The functions and data within these classes are termed as class members.

Definition of Classes

Since classes are user-defined data types, you can define how the data type should be structured. When you do this, you do not define the data to be stored in the class, but you define the name of the class. You will also define the objects used in the class and the operations you can perform on the class's objects.

Use the keyword **class** when you define the class in your code. This keyword is followed by the class name and the body of the class. You enclose these data in curly braces. You can end a class declaration with a semicolon or list of data types, declarations, and functions. The following example defines a class named triangle.

```
class Triangle {

   public:

      double length;   // This variable denotes the length of the triangle

      double height;   // This variable denotes the height of the triangle

     double breadth;  // This variable denotes the breadth of the triangle

};
```

When you use the keyword ‘public,’ it denotes that different functions in the program can access the class's attributes or members. All you need to do is call the members accurately when you want to use the values in the function. If you do not want the values of the members in the class to change, you should use the keyword ‘private.’ We will discuss this in detail in this chapter.

Defining Class Objects

You can use a class to provide the detail of how an object should be defined in the program. Every object in the class is defined in the same way you define simple variables in the code. The following statements are examples of how to declare objects in a class. We are going to declare two objects for the class Triangle.

//The following lines of code are used to declare the objects Triangle1 and Triangle2 in the class Triangle

```
Triangle Triangle1;

Triangle Triangle2;
```

Now, each of these objects will have the same members (length, breadth, and height), which we declared while defining the class Triangle.

How to Access the Class Members

If the members in the class are public members, you can access them anywhere in the code using the access operator (.).

```
#include <iostream>

using namespace std;

class Triangle {

    public:

        double length;   // This variable denotes the length of the triangle

        double height;   // This variable denotes the height of the triangle

        double breadth;  // This variable denotes the breadth of the triangle

};

int main() {

//The following lines of code are used to declare the objects Triangle1 and Triangle2 in the class Triangle

Triangle Triangle1;

Triangle Triangle2;

//The objective of the code is to calculate the area of the triangle. We will now initialize a variable ‘area’ with the data type double and assign it the value 0.0

double area = 0.0

    // We will now specify the parameter values for each of the class members

    Triangle1.height = 5.0;

    Triangle1.length = 6.0;

    Triangle1.breadth = 7.0;

    Triangle2.height = 10.0;

   Triangle2.length = 12.0;

    Triangle2.breadth = 13.0;

        // We now define the function to us to calculate the area of the triangles.
    volume = Triangle1.height * Triangle1.length * Triangle1.breadth;
    cout << "Area of the first triangle: " << volume <<endl;
    volume = Triangle2.height * Triangle2.length * Triangle2.breadth;
    cout << "Area of the second triangle: " << volume <<endl;
    return 0;
}
```

When you execute the above code, you receive the following output:

```
Area of the first triangle: 105
Area of the second triangle: 780
```

Note that you cannot access protected and private class members using the access operator (.). We will discuss how you can access protected and private class members in the code.

Classes and Objects

You now have a brief idea of what classes and objects in C++ are and how you can access the class members and objects. We will look at other aspects of object-oriented programming in further detail later in the book. Before we move onto the next chapter, let us look at some points you need to keep in mind when you work on object-oriented programming.

S. No.	Concept	Description
1	Class members and functions	You can define member functions in classes similar to the way you define member data types or variables.
2	Class access modifiers	The keywords public, private, and protected are termed as class access modifiers. If you have not defined the access modifier for the class members, the compiler takes the value as ‘private.’
3	Constructors and destructors	Class constructors are special functions in C++, and these can only be used within classes, especially when you create a new class object. A destructor is another function created when you delete an object from the class.
4	Copy constructor	This function creates another object in the class by initializing and declaring the object using another object in the same class.
5	Friend functions	If defined as such in the code, Friend functions can access protected and private members present in the class.
6	Inline functions	An inline function is one that instructs the compiler to expand the entire code in the class using the details of the function without using a call to access the function.

7	This pointer	Objects in classes are assigned pointers, and every object has only one pointer assigned to it. This pointer only points to the memory location of the object.
8	Pointer to classes	Any pointer in a class works the same as a pointer to a structure does. It is important to note that a class is only a structure with different members, objects, and functions.
9	Static class members	Both function members and data members defined in a class can always be static class members.

Chapter Five: Differences Between Classes and Structures

We have looked at data structures and classes in detail in the last two chapters. Let us now understand the difference between a data structure and classes.

Structures and classes have similar characteristics, but there are some important differences to bear in mind. One of the most important differences is one surrounding security. Data structures are not secure, and you cannot hide any variables or members in the structure from the user. This means you cannot use data abstraction to hide any implementation details of the data, variables, and members in the structure. On the other hand, a class is secure since you can use specific keywords, such as protected and private, to hide the implementation details of the members. The following are some ways to understand this difference: The data and function members in a class are created as private members by default. Any variable or data in a structure is the default. Consider the following examples. The first program gives you a compilation error while the second one compiles accurately.

Program 1

```
// Program 1

#include <stdio.h>

  class Test {

    int x; // x is private

};

int main()

{

  Test t;

  t.x = 20; // compiler error because x is private

  getchar() ;
  return 0;
}
```

Program 2

```
// Program 2
#include <stdio.h>
struct Test {
    int x; // x is public
};
int main()
{
  Test t;
  t.x = 20; // works fine because x is public
  getchar();
  return 0;
}
```

When you choose to derive structures from another structure or class, the base structure's access specifier or class is public. When you derive a class from a structure or class, the default access modifier is specified as private by the compiler.

Program 3

```
// Program 3
#include <stdio.h>
class Base {
public:
    int x;
};

class Derived : Base { }; // is equivalent to class Derived : private Base {}
int main()
{
  Derived d;
  d.x = 20; // compiler error because inheritance is private
  getchar();
  return 0;
}
```

Program 4

```
// Program 4
#include <stdio.h>
  class Base {
public:
    int x;
};
  struct Derived : Base { }; // is equivalent to struct Derived : public Base
{}
  int main()
{
  Derived d;
  d.x = 20; // works fine because inheritance is public
  getchar();
  return 0;
}
```

Chapter Six: Encapsulation in C++

Every C++ program has two elements:

- **Code or program statements** : This part of the program has many statements or actions that the compiler should perform. It holds different statements, such as methods, functions, calls to functions, etc.
- **Program data** : This section of the program is the information relevant to the program. This information in the program gets affected by different program functions and methods.

Encapsulation is another method of object-oriented programming that binds various functions and data together. These functions manipulate the different variables and data stored in the program. Encapsulation also ensures that the data is safe from external factors and interference. This concept is directly related to the concept of data hiding. Data encapsulation is the concept of combining the data and functions using the data.

C++ allows you to encapsulate the data and hide it from external factors by creating user-defined data types, known as classes. We have looked at the different access modifiers in the previous chapters. As mentioned earlier, every item in a class is labeled private by default. For instance,

```
class Box {
   public:
      double getVolume(void) {
         return length * breadth * height;
      }
   private:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box
};
```

In the above code, the variables breadth, length, and height are termed as private variables. It also means these variables can only be accessed by members in the same class. They cannot be accessed by any other function or section of the program. This is one of the easiest ways to encapsulate data. If you want to make any section of the code public, or accessible to various sections of the code, you need to declare that these variables are public. Any variable you declare after this keyword is accessible to every section of your code. If you allow one class to access the data and functions in another class, you reduce the encapsulation in the code. The objective is to ensure that the elements in the class are private.

Example

When you write code using both public and private data members and functions, you are using both data abstractions and encapsulation. Let us look at the following example:

```
#include <iostream>

using namespace std;

class Adder {

   public:

      // constructor

      Adder(int i = 0) {

         total = i;

      }

      // interface to outside world

      void addNum(int number) {

         total += number;

      }
```

```
        // interface to outside worl d
        int getTotal() {
            return total;
        };
    private:
        // hidden data from outside world
        int total;
};
int main() {
    Adder a;
    a.addNum(10);
    a.addNum(20);
    a.addNum(30);
    cout << "Total " << a.getTotal() <<endl;
    return 0;
}
```

When you run the above code, you receive the following output:

```
Total 60
```

Chapter Seven: Understanding Inheritance

Inheritance is an important concept in object-oriented programming. Using this characteristic, you can define a new class in terms of an existing class in the code. This makes it easy for you to create, maintain, and update any application. Inheritance also allows you to reuse code and its functionality, thereby reducing implementation time when you develop new applications. When you create a class, you do not have to create or write a new class with new function members and data members. As a programmer, you can choose to let a new class inherit members present in an existing class. This class, known as the base class, is used by the derived class.

The objective of inheritance is to create a relationship between the base and derived classes. Consider the following statements:

1. Mammals are animals.
2. Dogs are mammals.

What do you infer from these statements? Dogs are animals. You can develop such relationships in different parts of the code.

Introduction to Base and Derived Classes

You can derive classes from one or more classes in the existing code. This means a class can inherit data, variables, class members, and function members from numerous base classes. You need to use a class derivation list to specify the child or derived class's base classes. The class derivation list has the following syntax:

```
class derived-class: access-specifier base-class
```

In the above syntax, the access specifier is the access specifier modifiers we discussed in the previous chapter – private, public, or protected. The base class is the name of an existing class in the code. If you do not use an access-specifier, then the compiler chooses the private access modifier as the default.

Consider the following example where we are using a Shape class to derive the class Rectangle.

```
#include <iostream>

using namespace std;

// Base class

class Shape {

   public:

      void setWidth(int w) {

         width = w;

     }
```

```
        void setHeight(int h) {
            height = h;
        }
    protected:
        int width;
        int height;
};
// Derived class
class Rectangle: public Shape {
    public:
        int getArea() {
            return (width * height);
        }
};
int main(void) {
    Rectangle Rect;
    Rect.setWidth(5);
    Rect.setHeight(7) ;
    // Print the area of the object.
    cout << "Total area: " << Rect.getArea() << endl;
    return 0;
}
```

When you run the above code, you obtain the following output:

```
Total area: 35
```

Inheritance and Access

When you use a base class to create a new class, the derived class can access every public and protected member in the base class. If you do not want a derived class to access a specific member in the base class, you should declare those variables should be declared as private members. The following table lists the different forms of access modifiers used in base classes and who can access those members.

Access Modifier	**Public**	**Protected**	**Private**

Same class	Yes	Yes	Yes
Derived class	Yes	Yes	No
Outside classes	Yes	No	No

From above, you can see that a derived class can inherit the different members in the base class as long as they are defined as public, except for the following:

1. Friend functions present in the base class
2. Overloaded operators present in the base classes
3. Destructors, constructors, and copy constructors defined in the base class

Inheritance Types

The access modifiers public, private, and protected can be used to determine what members or characteristics the derived class can derive from the base class. Most programmers do not use the access modifiers protected and private when they create derived classes. They prefer to use the public access modifier since this makes it easier for the derived class to obtain the base class's members and characteristics. You need to apply the following rules when you use inheritance in your code:

Public Inheritance

If you use a public base class to create a derived class, the members in the base class become the members of the derived class. These members will still be public members of the derived class. The derived class also obtains the protected members in the base class, and they remain protected members even in the derived class. If the base class has any private members, the derived class cannot access those members. Having said that, you can use calls to functions to access the private members through protected and public members in the base class.

Protected Inheritance

When you create a derived class from a protected base class, the protected and public members in the base class are accessible to the derived class. They become the protected members of the derived class.

Private Inheritance

When you derive members from a private base class, every member of the base class becomes a private member in the derived class.

Multiple Inheritance

A class in your code can inherit members from one or more classes in the code. You can use the following syntax for the same:

class derived-class: access baseA, access baseB...

The word access in the above syntax is the access modifier (private, public, or protected), and you need to use this keyword against any base class you create in the code. You can separate the base classes in the code using a comma. Look at the following example to understand the same:

```
#include <iostream>

using namespace std;

// Base class Shape

class Shape {

   public:

      void setWidth(int w) {

         width = w;

      }

      void setHeight(int h) {

         height = h;

      }

   protected:

      int width;

      int height;

};

// Base class PaintCost

class PaintCost {

   public:

     int getCost(int area) {

         return area * 70;

      }

};

// Derived class

class Rectangle: public Shape, public PaintCost {

   public:

      int getArea() {

         return (width * height);

      }
};
int main(void) {
   Rectangle Rect;
   int area;
   Rect.setWidth(5);
   Rect.setHeight(7);
   area = Rect.getArea();
   // Print the area of the object.
   cout << "Total area: " << Rect.getArea() << endl;
   // Print the total cost of painting
   cout << "Total paint cost: $" << Rect.getCost(area) << endl;
  return 0;
}
```

When you run the above code, you receive the following output:

```
Total area: 35
Total paint cost: $2450
```

Chapter Eight: Overloading in C++

C++ also allows you to use function overloading and operator overloading in the code. You can specify more than one operator in the scope of the same function or give different definitions to a function name. When you call an overloaded operator or function in the code, you are giving the compiler the liberty to choose the function name definition or operator in the current section of the code. The compiler does this based on the parameters used in the function and its definition. This process where the compiler chooses the appropriate overloaded operator or function is called overload resolution.

Introduction to Function Overloading

You can always define a function name in different ways in the same code or scope. If you want to use function overloading in your code, you need to define the function differently in the code. You can do this by using different parameters or arguments in the function and types. It is important to note that you cannot use overload function declarations only by adding a different return type. The following is an example where we are using the print() function to look at different data types in the code:

```
#include <iostream>

using namespace std;

class printData {

   public:

      void print(int i) {

        cout << "Printing int: " << i << endl;

      }

      void print(double  f) {

        cout << "Printing float: " << f << endl;

      }

      void print(char* c) {

        cout << "Printing character: " << c << endl;

      }

};

int main(void) {

   printData pd;

   // Call print to print integer

   pd.print(5);

   // Call print to print float
```

```
    pd.print(500.263);
    // Call print to print character
    pd.print("Hello C++");
    return 0;
}
```

When you compile the above code, you receive the following output:

```
Printing int: 5
Printing float: 500.263
Printing character: Hello C++
```

Introduction to Operator Overloading

C++ allows you to overload or redefine most operators built-in in the code. Therefore, as a programmer, you can use different operators even if you have user-defined types. An overloaded operator is a type of function with a special name – the keyword ‘operator.’ You need to define the operator's symbol after the keyword to define the overloaded operator. When you define an overloaded operator, you also need to define the parameter list and return type. Consider the following example:

```
Triangle operator+(const Triangle&);
```

In the above example, we are declaring the addition operator you can use to add two triangle objects. It then returns the output for the Triangle object. You can define an overloaded operator as an ordinary non-member function. Alternatively, you can define the operator using a class member function. In case you want to define the above function using a non-member function in the class, you need to pass two arguments in the following manner:

```
Triangle operator+(const Triangle&, const Triangle&);
```

The following is an example of how you can use operator overloading using member functions. We pass an object as an argument in a function, and the function accesses the properties of the argument using the object. The object will then call the operator, and this object can be accessed using the operator.

```
#include <iostream>
using namespace std;
class Triangle {
    public:
        double getArea(void) {
            return 0.5*length * breadth * height;
        }
        void setLength( double len ) {
            length = len;
        }
        void setBreadth( double bre ) {
            breadth = bre;
        }
        void setHeight( double hei ) {
            height = hei;
        }
        // In this section, we are looking at the overload addition operator to add two triangle objects.
        Triangle operator+(const Triangle& b) {
            Triangle;
            triangle.length = this->length + t.length;
            triangle.breadth = this->breadth + t.breadth;
            triangle.height = this->height + t.height;
            return triangle;
        }
            private:
        double length;      // Length of a triangle
        double breadth;     // Breadth of a triangle
        double height;      // Height of a triangle
};
// Main function for the program
int main() {
//The next three statements are used to declare three triangle objects
    Triangle Triangle1;
    Triangle Triangle2;
    Triangle Triangle3;
    double area = 0.0;     // We are declaring the variable area which will be used to store the area of
the three objects.
     // We will now specify the values for the variables defined for each of the objects
```

```
   Triangle1.setLength(6.0);
   Triangle1.setBreadth(7.0);
   Triangle1.setHeight(5.0);

   Triangle2.setLength(12.0);
   Triangle2.setBreadth(13.0);
   Triangle2.setHeight(10.0);
    // Let us now calculate the area of the two objects
  area = Triangle1.getArea();
   cout << "Area of the first triangle: " << area <<endl;
   area = Triangle2.getArea();
   cout << "Area of the second triangle: " << area <<endl;
   // We will now add the area of the first two triangles and store it in a third triangle
   Triangle3 = Triangle1 + Triangle2;
   // The area of the third triangle is calculated as follows
   Area = Triangle3.getArea();
   cout << "Area of the third triangle: " << area <<endl;
   return 0;
}
```

When you run the above code, you will receive the following output:

```
Area of the first triangle: 105
Area of the second triangle: 780
Area of the third triangle: 2700
```

Non-Overloadable or Overloadable Operators

The following are some operators that you can use for operator overloading.

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()

->	->*	new	new []	delete	delete []

The following are some operators you cannot overload in your code:

::	.*	.	?:

Example of Operator Overloading

The following are some examples of operator overloading to help you understand the concept of operator overloading:

Unary Operator Overloading

A unary operator, as the name suggests, can only operate on one operand in the code. The following are some example of unary operators:

- The decrement and increment operators, -- and ++ respectively
- The not (!) logical operator
- The minus (-) unary operator

You can use these unary operators to work on specific objects. The operator always appears on the left of the object, as in ++obj, --obj, !obj, and -obj. You can also use the operators on the right side of the object if needed. The following is an example where we overload a minus operator.

```
#include <iostream>

using namespace std;

class Distance {

   private:

      int feet;           // 0 to infinite

      int inches;          // 0 to 12

   public:

      // required constructors

      Distance() {

         feet = 0;

         inches = 0;

     }

      Distance(int f, int i) {

        feet = f;

         inches = i;

      }

      // method to display distance
      void displayDistance() {
         cout << "F: " << feet << " I:" << inches <<endl;
      }
      // overloaded minus (-) operator
     Distance operator- () {
         feet = -feet;
         inches = -inches;
         return Distance(feet, inches) ;
      }
};
int main() {
   Distance D1(11, 10), D2(-5, 11);
   -D1;                  // apply negation
   D1.displayDistance();   // display D1
   -D2;                  // apply negation
   D2.displayDistance();   // display D2
   return 0;
}
```

When you run the above code, you will obtain the following output:

```
F: -11 I:-10
F: 5 I:-11
```

You can use the above example if you want to overload any of the unary operators mentioned above.

Binary Operator Overloading

A binary operator takes two variables or arguments. The following are some examples of binary operators:

- Addition (+)
- Subtraction (-)
- Division (/)

The following is an example of how you can use the different operators mentioned above.

```
#include <iostream>
using namespace std;
```

```
class Triangle {
//The following are the data members or variables of the class triangle
   double length;
   double breadth;
   double height;
      public:
    double getArea(void) {
      return 0.5 * length * breadth * height;
   }
      void setLength( double len ) {
      length = len;
   }
    void setBreadth( double bre ) {
      breadth = bre;
   }
    void setHeight( double hei ) {
      height = hei;
   }
    // We are now going to use the addition operator to perform an overload on the existing values
in the classes
   Triangle operator+(const Triangle& t) {
      Triangle triangle;
      triangle.length = this->length + t.length;
      triangle.breadth = this->breadth + t.breadth;
      triangle.height = this->height + t.height;
      return triangle ;
   }
};
// Main function for the program
int main() {
```

```
//Declare the three objects Triangle1, Triangle2 and Triangle3
   Triangle Triangle1;
   Triangle Triangle2;
   Triangle Triangle3;
//Declaring the variable area to store the area of the triangle
  double area = 0.0;
    // We will now specify the 1 specification
   Triangle1.setLength(6.0);
   Triangle1.setBreadth(7.0);
   Triangle1.setHeight(5.0);
    Triangle2.setLength(12.0);
   Triangle2.setBreadth(13.0);
   Triangle2.setHeight(10.0);
    area = Triangle1.getVolume();
   cout << "Area of the first triangle: " << area <<endl;
    area = Triangle2.getVolume();
   cout << "Area of the second triangle: " << area <<endl;
    // We will now calculate the area of the third triangle using the binary operator
   Triangle3 = Triangle1 + Triangle2;
    area = Triangle3.getArea() ;
   cout << "Area of the third triangle: " << area <<endl;
    return 0;
}
```

When you run the above code, you obtain the following output:

```
Area of the first triangle: 105
Area of the second triangle: 780
Area of the third triangle: 2700
```

Relational Operator Overloading

C++ supports different relational operators, such as:

- Greater than (>)
- Less than (<)

- Greater than or equal to (>=)
- Less than or equal to (<=)
- Equal to (==)

You can overload the above operators in C++ and use them to compare the values of one object in a class to another. The following example shows you how you can use the less-than operator and overload it. You can similarly overload the other relational operators:

```
#include <iostream>

using namespace std;

class Distance {

   private:

      int feet;            // 0 to infinite

      int inches;          // 0 to 12

   public:

      // required constructors

      Distance() {

         feet = 0;

         inches = 0;

      }

      Distance(int f, int i) {

         feet = f;

         inches = i;

      }

      // method to display distance

      void displayDistance() {

         cout << "F: " << feet << " I:" << inches <<endl;

      }

      // overloaded minus (-) operator

      Distance operator- () {

         feet = -feet;

         inches = -inches;

         return Distance(feet, inches);

      }
      // overloaded < operator
      bool operator <(const Distance& d) {
         if(feet < d.feet) {
            return true;
         }
         if(feet == d.feet && inches < d.inches) {
            return true;
         }
         return false;
      }
};
int main() {
   Distance D1(11, 10), D2(5, 11);
    if( D1 < D2 ) {
      cout << "D1 is less than D2 " << endl;
   } else {
      cout << "D2 is less than D1 " << endl;
   }
     return 0;
}
```

When you run the above code, you receive the following output:

```
D2 is less than D1
```

Input and Output Operator Overloading

You can use the stream extraction operator (>>) and stream insertion operator (<<) to use built-in data types as inputs and outputs. C++ allows you to overload these operators and perform different input and output operations on the different classes and user-defined objects. That said, you need to convert the operator overloading function into a friend function for the class since you call it without having to create the object. The following is an example of how you can use the insertion and extraction operator and overload it.

```
#include <iostream>
using namespace std;
class Distance {
   private :
      int feet;          // 0 to infinite
      int inches;        // 0 to 12
  public:
      // required constructors
      Distance() {
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i) {
         feet = f;
         inches = i;
      }
      friend ostream &operator<<( ostream &output, const Distance &D ) {
         output << "F : " << D.feet << " I : " << D.inches;
         return output;
      }
      friend istream &operator>>( istream  &input, Distance &D ) {
         input >> D.feet >> D.inches;
         return input;
      }
};
int main() {
   Distance D1(11, 10), D2(5, 11), D3;
   cout << "Enter the value of object : " << endl ;
   cin >> D3;
   cout << "First Distance : " << D1 << endl;
   cout << "Second Distance :" << D2 << endl;
   cout << "Third Distance :" << D3 << endl;
   return 0;
```

```
}
```

When you compile and execute the above code, you obtain the following result:

```
$./a.out
Enter the value of object :
70
10
First Distance : F : 11 I : 10
Second Distance :F : 5 I : 11
Third Distance :F : 70 I : 10
```

Assignment Operator Overloading

C++ allows you to overload the assignment operator in the same way you overload other operators in C++. You can use the overloaded object to create an object in the same way you use a copy constructor for the same. The following is an example of how you can overload an assignment operator in C++.

```
#include <iostream>
using namespace std;
class Distance {
   private:
      int feet;             // 0 to infinite
      int inches;           // 0 to 1 2
   public:
      // required constructors
      Distance() {
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i) {
         feet = f;
         inches = i;
      }
      void operator = (const Distance &D ) {
            feet = D.feet;
            inches = D.inches;
        }
        // method to display distance
        void displayDistance() {
            cout << "F: " << feet <<  " I:" <<  inches << endl;
        }
};
int main() {
    Distance D1(11, 10), D2(5, 11);
    cout << "First Distance : ";
    D1.displayDistance();
    cout << "Second Distance :";
    D2.displayDistance() ;
    // use assignment operator
    D1 = D2;
    cout << "First Distance :";
    D1.displayDistance();
    return 0;
}
```

On running the above code, you obtain the following output:

```
First Distance : F: 11 I:10
Second Distance :F: 5 I:11
First Distance :F: 5 I:11
```

Function Call Operator Overloading

You can overload the function call operator on any object in the class. When you overload the function operator, you create a new way to call the function and create a new operator function using which you can pass numerous arbitrary parameters. The following is an example of how you can overload the operator:

```
#include <iostream>
using namespace std;
```

```
class Distance {
    private:
        int feet;             // 0 to infinite
        int inches;           // 0 to 12
    public:
        // required constructors
        Distance() {
            feet = 0 ;
            inches = 0;
        }
        Distance(int f, int i) {
            feet = f;
            inches = i;
        }
        // overload function call
        Distance operator()(int a, int b, int c) {
            Distance D;
           // just put random calculation
            D.feet = a + c + 10;
            D.inches = b + c + 100 ;
            return D;
        }
        // method to display distance
        void displayDistance() {
            cout << "F: " << feet << " I:" << inches << endl;
        }
};
int main() {
    Distance D1(11, 10), D2;
    cout << "First Distance : ";
```

```
   D1.displayDistance();
   D2 = D1(10, 10, 10); // invoke operator()
   cout << "Second Distance :" ;
   D2.displayDistance();
   return 0;
}
```

When you run the above code, you will obtain the following result:

```
First Distance : F: 11 I:10
Second Distance :F: 30 I:120
```

Subscript Operator Overloading

A subscript operator is used to access different elements in an array. You can overload this operator to enhance or improve the functionality of arrays in C++. The following is an example of how you can overload the operator:

```
#include <iostream>
using namespace std;
const int SIZE = 10;
class safearay {
   private:
    int arr[SIZE];
   public:
     safearay() {
         register int i;
         for(i = 0; i < SIZE; i++) {
           arr[i] = i;
         }
      }
      int &operator[](int i) {
         if( i > SIZE ) {
               cout << "Index out of bounds" <<endl;
               // return first element.
               return arr[0];
```

```
            }
             return arr[i];
         }
};
int main() {
    safearay A;
    cout << "Value of A[2] : " << A[2] <<endl;
    cout << "Value of A[5] : " << A[5]<<endl;
    cout << "Value of A[12] : " << A[12]<<endl;
    return 0;
}
```

On running the above code, the output received is:

```
Value of A[2] : 2
Value of A[5] : 5
Index out of bounds
Value of A[12] : 0
```

Chapter Nine: Polymorphism in C++

Polymorphism is a concept in C++ which allows classes to have multiple forms. This only occurs when you have numerous classes in the code related through inheritance. Polymorphism in C++ means that the object determines the type of function to be executed by the compiler. When you call a function within a class, the compiler will look at the object type being used as an argument or parameter in the function before it invokes the function relevant to the object. In the example, we will look at how we can derive a base class based on two other classes.

```
#include <iostream>

using namespace std;

class Shape {

    protected:

        int width, height;

    public:

        Shape( int a = 0, int b = 0){

            width = a;

            height = b;

        }

        int area() {

            cout << "Parent class area :" <<endl;

            return 0;

        }

};

class Rectangle: public Shape {

    public :
        Rectangle( int a = 0, int b = 0):Shape(a, b) { }
        int area () {
            cout << "Rectangle class area :" <<endl;
            return (width * height);
        }
};
class Triangle: public Shape {
    public:
        Triangle( int a = 0, int b = 0):Shape(a, b) { }
        int area () {
            cout << "Triangle class area :" <<endl;
            return (width * height / 2);
        }
};
// This is the main function of the program
int main() {
    Shape *shape;
    Rectangle rec(10,7);
    Triangle  tri(10,5);
   // The variable shape is used to store the values for the rectangle
    shape = &rec;
    // This statement is used to call the function to calculate the area of the
rectangle
    shape->area();
   // The variable shape is used to store the values for the triangl e
    shape = &tri;
    // This statement is used to call the function to calculate the area of the triangle
    shape->area();
       return 0;
}
```

When the above code is run, the following is the output you will receive on your screen:

```
Parent class area :
Parent class area :
```

This is not the output we want. If you are unable to identify the reason for this incorrect output, let me tell you what change needs to be made to the code. Look at how the function area() is being called. We have defined this function in the base class, and the compiler uses this version. This method of linking functions is termed as static linkage or resolution. The call of this function should be fixed before you execute the program. This process is termed early binding. The compiler will set this function when it debugs the program. Let us now make a slight change to the above program. We will declare the area function within the Shape class itself. We will also use the virtual keyword. The updated code is as follows:

```
class Shape {
    protected:
       int width, height;
        public:
        Shape( int a = 0, int b = 0) {
            width = a;
            height = b ;
        }
        virtual int area() {
            cout << "Parent class area :" <<endl;
            return 0;
        }
};
```

When you make this modification to the code, you will receive the following output:

```
Rectangle class area
Triangle class area
```

The compiler will now look at the pointer and the contents of the variable it is pointing to. It will no longer look at the data type. The compiler calls the area function since the rec and tri class objects are not stored in shape but stored in *shape. Every child class in the above code has its own implementation of the area function. This is how programmers use the polymorphism concept in C++. The above code has different classes, and each class has the same function. The functions also take the same parameters, but the implementation of the function is different in each case.

Understanding Virtual Functions

When you define a function using the virtual keyword, it becomes a virtual function. C++ allows you to define a virtual function in the base class and a different version of the virtual function in the derived class. This only signals to the compiler that you do not want any link to exist between the functions. The only thing you need to be aware of is the position of the function you want to call in the code. To do this, you need to ensure the function selected by the compiler is based on the object you want to use it on. This type of connection or link is termed as late binding or dynamic linkage.

Pure Virtual Functions

You can include virtual functions in the base class and use that in a derived class. It is important to note that these functions can be redefined in the derived classes, so the functions suit the objects present in the derived class. There is, however, no meaningful definition you can give the function in the class. The following example shows you how you can change a virtual function in the base class.

```
class Shape {
    protected:
        int width, height;
    public:
        Shape(int a = 0, int b = 0) {
            width = a;
            height = b;
        }
        // pure virtual function
        virtual int area() = 0;
};
```

We see that the virtual function area() has been assigned the value zero in the above code. This indicates to the compiler that the function does not have a body. This is an example of a pure virtual function.

Chapter Ten: Abstraction in C++

Data abstraction is the process of hiding details or information in the code from other functions or classes in the code. The objective of data abstraction is to only present the details relevant to the other classes without sharing the actual details present in the class. Abstraction is a design and programming technique that relies only on two aspects – interfaces and implementation.

Consider the following example. When you use a television, you can switch it on and off, switch between channels, add speakers and other external devices, such as DVDs and VCRs or even increase or decrease the volume. You can do this using a remote, but you do not know what happens inside the device for you to be able to do this. You do not know how the signals pass through the cables or air or how the television interprets those signals before it displays them on the screen. Therefore, we can definitively state that a television will separate the interface's internal functionalities or implementation. You can use a remote to play with the external interface without learning anything about the internal components.

C++ gives every class you define some level of abstraction. When you define a class, you can determine the different class methods that the other classes in the code can use. This allows those classes to manipulate the data members and function members in the class without knowing how the base class works. For instance, you can use the sort function anywhere in your code without knowing what the function does or the algorithm it uses. Bear in mind that the functionality and algorithm used in the sort function will vary from one release to the next. If the interface being used still stays the same, any call made to the function will work in the code.

You can define abstract data types (ADTs) in the classes defined in the code. You can also use the cout object from the ostream class to move data into standard output files.

```
#include <iostream>

using namespace std;

int main() {
    cout << "Hello C++" <<endl;

    return 0;

}
```

From the above code, it is clear you do not have to worry about how the function cout works. You know it is used to display the output on the screen. It is only important for you to know the public interface used since the implementation of the cout function or keyword can change.

Benefits

There are two advantages of using abstraction in your code:

- The internal members of the class are always protected from any errors which users are bound to make. This prevents the corruption of objects in the classes.
- The implementation of classes and functions can evolve or change over time in response to the needs or requirements that constantly change. The developers may also make fixes or implementation changes in case of any bug report.

When you define data members or function members in the class, especially in private sections in the class, you can make any changes to the data as the author of the class. If the implementation changes, you only need to examine the code written in the class to see how the implementation affects the classes. If the data used in the classes is public, then a function related to the data or function members in the class may break in case of an implementation change.

How to Enforce Abstraction

In C++, you can use different access labels to define the interfaces in the class where you want to protect the data. Any class you define can contain zero, one, or more labels.

- When you define a class member using the public label, you allow that member to be accessible to any function in the program. Public members determine how the members in the class are viewed.

- If you define members using the private or protected labels, these values will not be accessible to any function or class in the code. The implementation of the members will always remain hidden.

C++ does not restrict the number of times you use an access label in your code. Every access label used in the code specifies the level of access every member in the firm has. The specific access level will always remain in effect until the compiler comes across another access level in the code.

Example

When you write a code with private or public class members, you are using the concept of data abstraction.

```
#include <iostream>

using namespace std;

class Adder {

   public:

      // constructor

      Adder(int i = 0) {

         total = i;

      }

      // interface to outside world

      void addNum(int number) {

         total += number;

      }

      // interface to outside world

      int getTotal() {

         return total;

      } ;
```

```
    private:
        // hidden data from outside world
        int total;
};
int main() {
    Adder a;
    a.addNum(10);
    a.addNum(20);
    a.addNum(30);
    cout << "Total " << a.getTotal() <<endl;
    return 0;
}
```

When you run the above code, you will obtain the following output:

```
Total 60
```

In the above code, we add two numbers and return the sum of those numbers. We have two public members in the code:

1. addNum
2. getTotal

The compiler uses these members as the interface to the outside world. A user only needs to know how to work with the class.

The private member in the code is total, and this is a value the user does not know about. The class, however, needs this variable for it to function.

Why Use Abstraction?

Through abstraction, you can separate the code into two parts – implementation and interface. When you design any code or program component, make sure the interface and implementation are independent. This is the only way you can ensure that any change made to the implementation would not affect the interface. This helps you control the functioning of any program to ensure there is no impact on the component and application because changes are made to the implementation.

Chapter Eleven: Abstract Classes or Interfaces

You may need to describe or use classes in your programs or code without committing or linking the class to a specific type of implementation. You can do this using an interface. You can implement interfaces in C++ using a concept termed as an abstract class. Do not confuse an abstract class with the concept of data abstraction. The latter is a concept used in object-oriented programming wherein the code's implementation details are kept away or apart from the interface and data used in the code.

You can make any class abstract by declaring a pure virtual function. We discussed this in an earlier chapter. a pure virtual function is one where the value of the function is equated to zero. For example:

```
class Box {
   public:
      // pure virtual function
      virtual double getVolume() = 0;
   private:
     double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box
};
```

Most programmers use an abstract class as a base class using which they create derived classes. An abstract class is one type of class using which you cannot create, declare, assign, or initialize an object. These classes only serve the purpose of an interface. If you try to create an object and instantiate it in an abstract class, it only leads to compilation errors. Therefore, if you want to instantiate a subclass or object in an abstract class, you need to implement numerous virtual functions. Virtual functions can support any interface created or declared within an abstract class. If you do not override a pure virtual function in any derived class and declare or instantiate an object in the class, it will lead to an error. These mistakes are very small but hard to detect in the code. You can also create a concrete class to instantiate and declare objects.

Example

The following is an example of how you can use abstract classes to implement functions. In the example below, we will look at how you can implement the function getArea().

```
#include <iostream>

using namespace std;

// Base class

class Shape {

   public:

      // pure virtual function providing interface framework.

      virtual int getArea() = 0;

      void setWidth(int w) {

         width = w;

      }

      void setHeight(int h) {

         height = h;

      }

   protected:

      int width;

      int height;

};

// Derived classes
```

```
class Rectangle: public Shape {
    public:
        int getArea() {
            return (width * height);
        }
};
class Triangle: public Shape {
    public:
        int getArea() {
            return (width * height)/2;
        }
};
int main(void) {
    Rectangle Rect;
    Triangle  Tri;
    Rect.setWidth(5);
    Rect.setHeight(7);
    // Print the area of the object.
    cout << "Total Rectangle area: " << Rect.getArea() << endl;
    Tri.setWidth(5);
    Tri.setHeight(7);
   // Print the area of the object.
    cout << "Total Triangle area: " << Tri.getArea() << endl;
```

```
  return 0;

}
```

When you run the above code, you receive the following output:

```
Total Rectangle area: 35

Total Triangle area: 17
```

In the above example, we see how you can use an abstract class to define an interface using the function getArea(). We also saw how you could implement this function in two other classes in the code. Each of these classes, however, uses a different algorithm to calculate the area of the shape.

When you develop an application using object-orientation, you may need to use an abstract class to provide a standard and common interface. This interface will be appropriate for any external class, application, or function you may want to use. Through inheritance, the derived classes obtain the necessary methods and data from the abstract base class. The functions, classified as public in the abstract class, should be pure virtual functions. These pure functions can only be used in the derived classes, which correspond to the specific functions and methods used in the application.

This makes it easier for you, as a programmer, to add new objects and applications to the existing code even after you have defined the system.

Chapter Twelve: Constructors in C++

As mentioned earlier, a constructor is a function created in a class when you create an object. This function is used to initialize the object in a class. A constructor is termed as a member function in every class. Before we look at the details of what constructors are, let us understand the difference between constructors and member functions. The following are some differences between constructors and member functions in a class:

- A constructor, unlike a member function, will have the same name as the class
- There is no return type for a constructor
- When you create an object in a class, the compiler automatically creates a constructor. You need to create a member function separately if you want to perform any operations.
- You need not define any constructor in the code since the compiler automatically generates one with no body or parameters

Let us understand what a constructor is better using an example. Let us assume you went to the store to purchase a pen. Do you consider all the options available when you choose to buy a pen? The first thing you do is think about the store you want to go to. Once you reach the store, you ask the shopkeeper to give you a pen. When you ask for just a pen, it indicates you have not thought about the brand you want to use or which color you prefer. The shopkeeper will hand you a pen or give you a pen that people have bought frequently. This is exactly what a default constructor in your class is.

The other option you have is to go to the store and let the shopkeeper know you want a blue color pen sold by ABC brand. When you mention this to him, he will hand the exact product to you. The shopkeeper knows what you want because you gave him the parameters. This is an example of a parameterized constructor.

The last option is to take a pen you have at home and show the shopkeeper a physical copy of the pen and ask for the same thing. The shopkeeper will give you exactly that pen. This new pen is a copy of the pen you own. This is how a copy constructor works.

Constructor Types

The following are the types of constructors we discussed previously.

Default Constructors

A default constructor is one that does not use any parameters or arguments. There is also no function or operation defined within this constructor.

Consider the following example:

```
// This program is an example of a default constructor

#include <iostream>

using namespace std;

class construct

{

public:

    int a, b;

    // Default Constructor

    construct()

    {

       a = 10;

        b = 20;

    }

};

int main()

{

    // Default constructor called automatically

    // when the object is create d
    construct c;

    cout << "a: " << c.a << endl

         << "b: " << c.b;

    return 1;

}
```

On compiling the code, you will receive the following output:

```
a : 10

b : 20
```

It is important to note that the compiler always defines a constructor in the code when you create an object in the class.

Constructors with Parameters

Constructors are like functions in the sense that you can pass arguments or parameters. These parameters or arguments allow you to declare and initialize an object in the class when you create it. If you want to create a constructor that accepts parameters and arguments, you need to add them to the constructor, similar to how you would add them to functions. When you define the body of the constructor, you should use the parameters or arguments to initialize or instantiate the objects in the class.

Consider the following example:

```
// This example is used to create a constructor with parameters and arguments

#include <iostream>

using namespace std;

class Point

{

private:
    int x, y;
public:
    // Parameterized Constructor
    Point(int x1, int y1)
    {
        x = x1;
        y = y1;
    }
     int getX()
    {
        return x;
    }
    int getY()
    {
        return y;
  }
};
int main()
{
    // Constructor called
    Point p1(10, 15);
     // Access values assigned by constructor
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
    return 0;
}
```

When you run the above code, you will obtain the following output:

```
p1.x = 10, p1.y = 15
```

When you use a parameterized constructor to create or declare an object, you need to pass the values you want to assign the object as an argument or parameter in the function. The compiler will not allow you to declare and assign a value to an object normally. Therefore, you need to call a constructor, and you can either do this implicitly or explicitly.

Consider the following example:

```
Example e = Example(0, 50); // Explicit call
Example e(0, 50);           // Implicit call
```

Uses of Parameterized Constructors

Parameterized constructors can be used for the following:

1. You can use this constructor to initialize different data elements in the code with different objects. Each of these objects can be assigned different values depending on when they are created.
2. You can use this function for constructor overloading. This method is similar to the process of overloading discussed above.

Copy Constructors

Copy constructors are member functions using which you can initialize an object in the classes. You can do this by using other objects in the same class. We will look at these in further detail later in the book.

When you define more than one constructor in the class with parameters, you also need to declare a default constructor without parameters. This should be declared explicitly in the class. The compiler will not create a default constructor if you have defined a constructor in the code. It is, however, not required for you to always declare a default constructor. However, this is the best practice.

```
// Example to create a copy constructor
#include "iostream"
using namespace std;
class point
{
private:
  double x, y;
public:
  // Non-default Constructor &
  // default Constructor
  point (double px, double py)
  {
    x = px, y = py;
  }
};
int main(void)
{
  // Define an array of size
  // 10 & of type point
  // This line will cause error
  point a[10];
  // Remove above line and program
  // will compile without error
```

```
   point b = point(5, 6);
}
```

The following is the output when the code above is compiled:

```
Error: point (double px, double py): expects 2 arguments, 0 provided
```

Chapter Thirteen: Copy Constructors in C++

We looked at the different types of constructors in the previous chapter. In this chapter, we will look at a copy constructor in further detail.

Definition

A copy constructor, like other constructors, is a member function in the class. This constructor is a copy of an existing member function or constructor in the class. You can use this function to initialize an object in the same class. Every copy constructor has the general syntax:

```
    ClassName (const ClassName &old_obj);
```

The following is an example of how you can create or use copy constructors.

```
#include<iostream>

using namespace std;

class Point

{

private:

    int x, y;

public:

    Point(int x1, int y1) { x = x1; y = y1; }

    // Copy constructor

    Point(const Point &p2) {x = p2.x; y = p2.y; }

    int getX()          {  return x; }

    int getY()          {  return y; }

};

int main()
```

```
{
    Point p1(10, 15); // Normal constructor is called here
    Point p2 = p1; // Copy constructor is called here
    // Let us access values assigned by constructors
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
    cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();
    return 0;
}
```

When you run the above code, you receive the following output:

```
p1.x = 10, p1.y = 15
p2.x = 10, p2.y = 15
```

When Do You Call a Copy Constructor?

The following are the instances when you can call a copy constructor.

1. When the member functions in the class return the object as a value
2. When you pass the object as a parameter or argument in a function instead of declaring it in the class
3. When you construct an object using a different object present in the same class
4. When the compiler uses the constructors and member functions to create temporary objects

However, you do not have to create a copy constructor or use it in any of these situations since C++ lets the compiler take the decision to save the memory. The compiler can choose to update and monitor the way a copy constructor is called, and objects and functions are copied.

When Should You Define a Copy Constructor?

If you do not define the copy constructor, the compiler will create a default constructor for every class. This does not necessarily have to be a copy made as per the member functions and data. The compiler then creates a copy constructor, which will work fine in general. When a compiler creates a copy constructor, it works like every other copy constructor. If you want to define a copy constructor for a certain function, make sure the object has a pointer attached to it.

If you let the compiler create a copy constructor, it will only be a shallow copy of the constructor. You can only ensure that the object or class is fully copied by creating a user-defined copy constructor. In these constructors, you can define the references and pointers of copied objects to every location.

Assignment Operators Versus Copy Constructors

From the example below, which statement do you think will call the assignment operator and which one will call the copy constructor?

```
MyClass t1, t2;

MyClass t3 = t1;  // ----> (1)

t2 = t1;          // -----> (2)
```

The compiler will call the copy constructor when you create a new object in the code from an existing object. It only calls the constructor since you create a copy of the existing copy. It calls the assignment operator when you assign a value to an existing object in the code. In the example above, the first statement will call the copy constructor, and the second constructor will call upon the assignment operator.

Example Where You Use Copy Constructors

The example below shows how you can use copy constructors in C++. We will define a copy constructor in the String class.

```
#include<iostream>

#include<cstring>

using namespace std;
```

```
  class String
{
private:
    char *s;
    int size;
public:
    String(const char *str = NULL); // constructor
    ~String() { delete [] s;  }// destructor
    String(const String&); // copy constructor
    void print() { cout << s << endl; } // Function to print string
    void change(const char *);  // Function to change
};
  String::String(const char *str)
{
    size = strlen(str);
    s = new char[size+1];
  strcpy(s, str);
}
  void String::change(const char *str)
{
    delete [] s;
    size = strlen(str);
    s = new char[size+1];
    strcpy(s, str);
}
  String::String(const String& old_str)
{
    size = old_str.size;
    s = new char[size+1];
    strcpy(s, old_str.s);
}
  int main()
{
    String str1("GeeksQuiz");
    String str2 = str1;
      str1.print(); // what is printed ?
    str2.print();
      str2.change("GeeksforGeeks");
      str1.print(); // what is printed now ?
    str2.print();
    return 0;
}
```

The output of this code is:

```
GeeksQuiz
GeeksQuiz
GeeksQuiz
```

GeeksforGeeks

What Happens When You Remove a Copy Constructor From the Code?

When you remove copy constructors from the code above, you will not receive the output. The changes you make to variable str2 will reflect the value in str1, which is not the expected outcome.

```
#include<iostream>

#include<cstring>

using namespace std;

class String

{

private:

    char *s;

    int size;

public:

    String(const char *str = NULL); // constructor

    ~String() { delete [] s;  }// destructor

    void print() { cout << s << endl; }

    void change(const char *);  // Function to change

};

String::String(const char *str)

{

    size = strlen(str);

    s = new char[size+1];
    strcpy(s, str);
}
void String::change(const char *str)
{
    delete [] s;
   size = strlen(str) ;
    s = new char[size+1];
    strcpy(s, str);
}
int main()
{
    String str1("GeeksQuiz");
    String str2 = str1;
    str1.print(); // what is printed ?
    str2.print();
    str2.change("GeeksforGeeks");
    str1.print(); // what is printed now ?
    str2.print();
    return 0;
}
```

The output of the above code is:

```
GeeksQuiz
GeeksQuiz
```

GeeksforGeeks

GeeksforGeeks

Can Constructors be Made Private?

You can make copy constructors private variables in the code. It is important to bear in mind that every object or member of the class can no longer be copied. It is useful to do this only when the class you define the constructor in has pointers, or the resources are allocated memory space dynamically. In these situations, you can write a copy constructor like the above example. You can also make the constructor private so that a user will receive a compiler error instead of runtime errors.

Passing Arguments in Copy Constructors

You can only pass arguments in a copy constructor in the form of a reference. When you pass an argument using a value in the copy constructor, a call to the constructor will lead to an error since the compiler only looks at the constructor and not the values of the arguments passed as parameters.

Chapter Fourteen: Destructors in C++

Now that we have looked at what a constructor is let us learn more about destructors. A destructor is another member function that is created by the compiler when you delete any object in the existing class. The syntax used for a destructor is as follows:

~constructor-name();

Properties

The following are properties of destructors:

- A destructor function is created by the compiler when you delete an object in a class
- You cannot declare a constant or static member function
- C++ does not allow you to enter any arguments
- The destructor does not have any return type. You cannot enter void as a return type
- If you have an object in a destructor class, it does not become a member of the class
- Destructors can only be declared in the public section of any class
- You do not have access to the address of the destructor function

When Do You Call a Destructor?

The compiler calls a destructor function when the object is no longer out of scope.

1. The program ends
2. The function ends
3. An object is deleted using the delete operator
4. A block containing the local variable will end

Difference Between Destructors and Member Functions

A destructor will have the same name as the class where an object has been deleted. The only way to differentiate between a destructor and class is the presence of a tilde (~). a destructor does not take arguments or parameters and cannot return any value.

Consider the following example:

```
class String {
private:
    char* s;
    int size;
public:
    String(char*); // constructor
    ~String(); // destructor
};
String::String(char* c)
{
    size = strlen(c);
    s = new char[size + 1];
    strcpy(s, c);
}
String::~String() { delete[] s; }
```

Can You Have More Than One Destructor?

Most people wonder if you can have multiple destructors in a class. If you delete more than one object, the compiler is bound to create more than one destructor, right? This, however, is not a correct assumption. There can only be one destructor in a class since the function takes the name of the class. There will be no parameters or functions within the destructor.

When Should You Define a Destructor?

The compiler creates a destructor automatically in the code if you do not declare one. It is okay to use the default destructor if none of the objects are allocated memory space dynamically. A default destructor does not work the way it should if you have a pointer. When classes contain pointers, the memory allocated to the class object will first need to be removed before you delete the instance. This is the only way to prevent a memory leak.

Can You Define Virtual Destructors?

Experts recommend that you describe a virtual destructor. We will look at virtual destructors in detail in the following chapter.

Chapter Fifteen: Virtual Destructors in C++

You can create a derived class where the objects use pointers to obtain the information of relevant objects from the base class. If you delete such objects using a non-virtual destructor, it will lead to runtime errors. To improve or rectify this situation, you need to define a virtual destructor in the base class. If you were to compile the code in the example below, it would result in an error in the code.

```
// This program is an example of how a class without a virtual destructor leads to runtime errors in the code

#include<iostream>

  using namespace std;

  class base {

  public:

    base()

    { cout<<"Constructing base \n"; }

    ~base()

    { cout<<"Destructing base \n"; }

};

  class derived: public base {

  public:

    derived()

    { cout<<"Constructing derived \n"; }

    ~derived()

    { cout<<"Destructing derived \n"; }

};
```

```
  int main(void)
{
  derived *d = new derived();
  base *b = d;
  delete b;
  getchar();
  return 0;
}
```

The above code's output may be slightly different between different C++ instances depending on the compiler used. If you use the Dev-CPP compiler, you will obtain the following output:

```
Constructing base
Constructing derived
Destructing base
```

When you create a virtual destructor in the base class in the code, it indicates to the compiler that the object in the derived class, when deleted, will be removed correctly from the code. These lines of code ensure that both the derived and base class destructors are called when you run the code. For instance,

```
// This program is an example of how you can use a virtual destructor to avoid a runtime error
#include<iostream>
using namespace std;
class base {
  public:
    base()
    { cout<<"Constructing base \n"; }
    virtual ~base( )
    { cout<<"Destructing base \n"; }
};
class derived: public base {
  public:
    derived()
    { cout<<"Constructing derived \n"; }
    ~derived()
    { cout<<"Destructing derived \n"; }
};
int main(void)
{
  derived *d = new derived();
  base *b = d;
  delete b;
  getchar();
  return 0;
}
```

You will receive the following output when you run the above code:

```
Constructing base
Constructing derived
Destructing derived
Destructing base
```

If you use a virtual function in your class, you need to add a virtual destructor to the code immediately, regardless of whether or not you use it. This is the only way you can prevent any runtime errors.

Pure Virtual Destructors

You can create a pure virtual destructor in C++ if you need to. When you define these functions in a class, you need to add a body to the destructor. This contradicts the definition of a virtual function, doesn’t it? Why does a virtual function need a body? We need to do this since the compiler does not override the destructor. The compiler calls a pure virtual destructor in the reverse order when a class is derived. This indicates that a destructor in a class is invoked first by the compiler. It is only after this that the compiler calls the destructor in the class.

If you do not define a pure virtual destructor in the code, what function or statements will the compiler call upon when it needs to destroy or delete an object? It is only when you define the right objects that the linker and compiler in the code enforce the presence of the function body. Consider the example below:

```
#include <iostream>

class Base

{

public:

    virtual ~Base()=0; // Pure virtual destructor

};

class Derived : public Base

{

public:

    ~Derived()
    {
        std::cout << "~Derived() is executed";
    }
} ;
  int main()
{
    Base *b=new Derived();
    delete b;
    return 0;
}
```

The linker in the code will give you the following error:

```
test.cpp:(.text$_ZN7DerivedD1Ev[__ZN7DerivedD1Ev]+0x4c):
undefined reference to `Base::~Base()'
```

If you define a pure virtual destructor in the code, the program will compile the code without any errors.

```
#include <iostream>
class Base
{
public:
    virtual ~Base()=0; // Pure virtual destructor
};
Base::~Base()
{
    std::cout << "Pure virtual destructor is called";
}
class Derived : public Base
{
public:
    ~Derived( )
    {
        std::cout << "~Derived() is executed\n";
    }
};
int main()
{
   Base *b = new Derived();
    delete b;
    return 0;
}
```

The output of the above code is:

```
~Derived() is executed
Pure virtual destructor is called
```

Bear in mind that a class in your code becomes an abstract class if you have pure virtual destructors. Consider the program below. Write it in your C++ window and see how it runs.

```
#include <iostream>
class Test
{
```

```
public:
    virtual ~Test()=0; // Test now becomes abstract class
};
Test::~Test() { }
int main()
{
    Test p;
    Test* t1 = new Test;
    return 0;
}
```

When you run the above code in the compiler, you receive the following messages:

```
[Error] cannot declare variable ‘p’ to be of abstract type ‘Test’
[Note] because the following virtual functions are pure within ‘Test’:
[Note] virtual Test::~Test()
[Error] cannot allocate an object of abstract type ‘Test’
[Note] since type ‘Test’ has pure virtual functions
```

If you make the changes indicated in the error to the code, the program compiles without any errors.

Chapter Sixteen: Introduction to Private Destructors

```
// This program is an example of a private destructor

#include <iostream>

using namespace std;

  class Test {

private:

    ~Test() {}

};

int main()

{

}
```

If you run the above code, you will see it compile with no errors. Therefore, you can say that there is no compiler error in the code, which indicates that the compiler does not throw an error when it comes across a private destructor. Consider the program below:

```
// This program is used to explain how a private destructor functions

#include <iostream>

using namespace std;

class Test {

private:

    ~Test() {}

};

int main()
```

```
{
    Test t;
}
```

When you run the above code, you will receive a compile error. The compiler notes that you have declared a variable ‘t’ and it cannot delete it from the class or code since the destructor you have defined is private. What do you think happens in the following code?

```
// Code to understand private destructors
#include <iostream>
using namespace std;
  class Test {
private:
    ~Test() {}
};
int main()
{
    Test* t;
}
```

When you run the above code, you do not receive any error. There is no object constructed as part of the code, and the compiler uses the pointer. Therefore, there is no use of a destructor. Now, what about the following program?

```
// Example of a private destructor
  #include <iostream>
using namespace std;
  class Test {
```

```
private:

    ~Test() {}

};

int main()

{

    Test* t = new Test;

}
```

When you run the above code, you will not receive any error in the code. Why do you think this is the case? The above code uses dynamic memory allocation to store the variables. Therefore, it is your duty to delete the object stored in the dynamic memory. It is for this reason why the compiler does not care.

In case you create a destructor and label it as a private member function, you can also create another instance in the class using the malloc() function (memory allocation). Consider the following example:

```
// Example of a private destructor

  #include <bits/stdc++.h>

using namespace std;

  class Test {

public:

    Test() // Constructor

    {

        cout << "Constructor called\n";

    }

  private:

    ~Test() // Private Destructor
    {
        cout << "Destructor called\n" ;
    }
};
  int main()
{
    Test* t = (Test*)malloc(sizeof(Test));
    return 0;
}
```

You do not receive any output when you run the code. The following code fails to compile accurately.

```
// Example of a private destructor
#include <iostream>
using namespace std;
class Test {
private:
  ~Test() {}
};
int main()
{
    Test* t = new Test;
    delete t;
}
```

When you create a class with a private destructor, a dynamic object is created for those classes. The following example is one where you can create a class using a private destructor. You can also create a friend function in the class. This function is only used to delete objects.

```
// Example of a private destructor

#include <iostream>

  // a class with private destructor

class Test {

private:

    ~Test() {}

    friend void destructTest(Test*);

};

  // Only this function can destruct objects of Test

void destructTest(Test* ptr)

{

    delete ptr;

}

  int main()

{

    // create an object

    Test* ptr = new Test;

      // destruct the object

    destructTest(ptr);

     return 0;
}
```

When you want to control or monitor the destruction or deletion of objects in a class, you need to make the destructor private. If you use dynamically created objects in your classes, you can delete the object by passing pointers in the function as arguments or parameters. These functions delete the objects in the code. It is important to note that a reference to an object after the function is called will lead to a dangler.

Chapter Seventeen: Exception Handling in C++

C++ is very different from C in terms of exception handling. An exception is any abnormal condition or a runtime error in the code. These are anomalies the compiler encounters when it executes the lines of code written. The following are the types of exceptions you may encounter when the compiler runs the program:

1. Synchronous
2. Asynchronous

An asynchronous error is beyond the compiler's control and the code written in the application. If you want to prevent the occurrence of an asynchronous error, you can use the following keywords:

1. *Try* : If you add this keyword at the beginning of a block of code, you can indicate to the compiler that this block of code may throw an error or exception
2. *Catch* : This keyword depicts that block of code that should be executed when the compiler throws a specific error
3. *Throw* : This keyword throws an exception or lists the different exceptions the block of code may throw. The compiler can ignore these errors and avoid handling them if it chooses to.

Importance of Exception Handling

The following are the reasons why you need to use or include error handling code or keywords in your program:

Separation of Normal Code From Error Handling Code

If you use traditional methods to write error handling codes, you can include if-else functions and other conditional statements to handle any errors. This only leads to confusion since the error handling code gets mixed with the normal flow, which makes it difficult for you to read the code. It also becomes difficult to maintain the code. If you use try-catch blocks of code in your program, you can separate the normal code from the error handling code.

Methods and Functions Can Choose How to Handle Exceptions

Functions and methods have different operations within the body, which may throw some exceptions, but these functions and methods can choose to handle these exceptions differently. If the function is called elsewhere in the program, the compiler can choose how to handle the other exceptions in the code. If the function caller does not care about the exceptions or misses them, then the caller of the caller will need to handle the error in the code.

When you write code in C++, you can write the throw keyword against the statements where you think there will be an error. The caller of the function with this keyword will need to identify a way to handle the exceptions. It can either catch the error or specify it again to the compiler.

Grouping Errors

Exceptions in C++ can either be objects or basic types of member functions and variables. You can write a few lines of code by creating a hierarchy of certain exception objects. You can group these objects based on their classes and namespaces and then categorize them based on their type.

Exception Handling Examples

The following is an example of how you can use exception handling in C++. This program's output will explain how the try-catch blocks of code are used by the compiler.

```
#include <iostream>

using namespace std;

int main()

{

    int x = -1;

    // Some code

    cout << "Before try \n" ;

    try {

        cout << "Inside try \n";
        if (x < 0)
        {
            throw x;
            cout << "After throw (Never executed) \n";
        }
    }
    catch (int x ) {
        cout << "Exception Caught \n";
    }
    cout << "After catch (Will be executed) \n";
    return 0;
}
```

The following is the output of the code:

```
Before try
Inside try
Exception Caught
After catch (Will be executed)
```

C++ also allows you to use a specific type of exception handler called the ‘catch-all’ block. You can use this function to catch any exceptions in the code. For instance, the following program throws an exception as an integer value. There is, however, no way to catch the error. In this case, the catch(…) block of code will be executed by the compiler.

```
#include <iostream>
using namespace std;
int main()
```

```
{
    try  {
        throw 10;
    }
    catch (char *excp)  {
         cout << "Caught " << excp;
    }
    catch (...)  {
         cout << "Default Exception\n";
    }
    return 0;
}
```

The output of the following code is: Default Exception.

As mentioned in the first book, you cannot convert primitive data types in the code. This means you cannot use implicit type conversions in the code. Consider the following example where we are trying to convert a character into an integer.

```
#include <iostream>
using namespace std;
int main()
{
    try  {
        throw 'a';
    }
    catch (int x) {
       cout << "Caught " << x;
    }
    catch (...)  {
        cout << "Default Exception\n";
    }
    return 0;
}
```

You receive the following output when you run the code: Default Exception.

If the compiler throws an exception that is not caught anywhere in the code, the program will end. For instance, the following program throws a char exception since there is no catch block or exception handling code to catch this error.

```
#include <iostream>
using namespace std;
int main()
{
    try  {
       throw 'a';
   }
    catch (int x)  {
        cout << "Caught ";
    }
    return 0;
}
```

The output of the above code is:

```
terminate called after throwing an instance of 'char'


This application has requested the Runtime to terminate it in an

unusual way. Please contact the application's support team for

more information.
```

You can change the way the code behaves by writing blocks of code to indicate unexpected functions.

There are instances when you may receive an error in the derived class block of code. It is important to ensure the compiler can catch the error in this code before it looks at the base class exception. Therefore, you should first write exception handling code in the derived class. C++, like Java, is built with a library of exceptions that caters to all forms of base or standard exceptions. Any standard exception in your code can be caught by the compiler using this type.

It is unfortunate that the exceptions in C++ are left unchecked by the compiler. The compiler does not care if the exception in the code is identified or not. C++ does not require the compiler to specify the exceptions caught in the code or program. It is recommended that the exceptions are identified. Consider the following example. You receive an output with no exceptions, but the function fun() should have listed a set of unchecked exceptions.

```
#include <iostream>

using namespace std;

// This function signature is fine by the compiler, but not recommended.

// Ideally, the function should specify all uncaught exceptions and function

// signature should be "void fun(int *ptr, int x) throw (int *, int)"
```

```
void fun(int *ptr, int x)
{
    if (ptr == NULL)
        throw ptr;
    if (x == 0)
        throw x;
    /* Some functionality */
}
int main()
{
    try {
       fun(NULL, 0);
    }
    catch(...) {
        cout << "Caught exception from fun()";
    }
    return 0;
}
```

The output: Caught exception from fun().

It is better to write the above code in the following manner:

```
#include <iostream>
using namespace std;
// Here we specify the exceptions that this function
// throws.
void fun(int *ptr, int x) throw (int *, int)
{
    if (ptr == NULL)
        throw ptr;
    if (x == 0)
        throw x;
    /* Some functionality */
}
int main()
{
    try {
       fun(NULL, 0);
    }
    catch(...) {
        cout << "Caught exception from fun()";
    }
    return 0;
}
```

The output of the above code is: Caught exception from fun().

You can also use try-catch blocks and nest them in the code if you want to re-throw exceptions in the output:

```
#include <iostream>
using namespace std;
```

```
int main()
{
    try {
        try {
            throw 20;
        }
        catch (int n) {
            cout << "Handle Partially ";
           throw; // Re-throwing an exception
        }
    }
    catch (int n) {
        cout << "Handle remaining ";
    }
    return 0;
}
```

The output is: Handle Partially Handle remaining

You can also use the ‘throw’ keyword if you want to re-throw an exception. The function can then handle a part of the exception and let the caller handle the other part. When the code throws an exception, every object created in the class or try block will be removed from the code before the compiler moves back to the code's catch block.

```
#include <iostream>
using namespace std;
class Test {
```

```
public:
    Test() { cout << "Constructor of Test " << endl; }
    ~Test() { cout << "Destructor of Test " << endl; }
};
int main()
{
    try {
        Test t1;
        throw 10;
    }
    catch (int i) {
        cout << "Caught " << i << endl;
    }
}
```

The following is the output of the code:

```
Constructor of Test
Destructor of Test
Caught 10
```

Chapter Eighteen: Stack Unwinding

You can remove functions in the code using an option known as stack unwinding. Using stack unwinding, you can remove function arguments, parameters, operators, and objects from the function's call stack. Stack handling is closely related to exception handling. When the compiler identifies an error in the code, it will look at the entire call stack to identify the error. The compiler looks for the exception handler in the code and the different entries associated with the handler. It will then remove the entire function with the exception handler code block from the function stack. This indicates that stack unwinding is a part of the exception handling process, especially since it relates to a function that is not handled in the same function.

Consider the following program:

```
#include <iostream>


using namespace std;


// a sample function f1() that throws an int exception

void f1() throw (int) {

  cout<<"\n f1() Start ";

  throw 100;

  cout<<"\n f1() End ";

}


// Another sample function f2() that calls f1()

void f2() throw (int) {

  cout<<"\n f2() Start ";
  f1() ;
  cout<<"\n f2() End ";
}

// Another sample function f3() that calls f2() and handles exception thrown
by f1()
void f3() {
  cout<<"\n f3() Start ";
  try {
    f2();
  }
  catch(int i) {
   cout<<"\n Caught Exception: "<<i;
  }
  cout<<"\n f3() End";
}

// a driver function to demonstrate Stack Unwinding  process
int main() {
  f3();


  getchar();
  return 0;
}
```

The following is the output of the code:

```
f3() Start

f2() Start

f1() Start

Caught Exception: 100

f3() End
```

Let us look at the different functions in the code. When the first function [f1()] throws an exception in your output, the entry is removed from the call stack. The compiler does this since the function with the error does not contain an exception handler code. The second function [f2()] does not have an exception handler code in it. Therefore, the compiler will remove the code from the stack. The last function in the code is [f3()]. The code in this function contains an exception handler block of code. When the compiler identifies the error, it will execute the catch block in the code. The compiler will run the code after the exception handler code. It is important to note that the compiler does not run the lines of code found in the first two functions.

```
//inside f1()

  cout<<"\n f1() End ";

//inside f2()

  cout<<"\n f2() End ";
```

If you have some local objects created in the first two functions, the destructors used in the base functions will be called to remove any lines or blocks of code during the process of stack unwinding.

Chapter Nineteen: Identifying Exceptions in Base and Derived Classes

If the compiler identifies exceptions in both the base and derived classes in the code, then it will not run the code or give you an output. When you write the code, you need to write an exception handler code in both the base and derived classes, and the exception handler block should be present before the block for the base class.

You may wonder why you need to add the exception handler block of code for the derived class before the base class. If you write the code for the base class before the derived class, the compiler will not reach the exception handler block of code in the derived class. For instance, the program below will give you the output: “Caught Base Exception.”

```
#include<iostream>

using namespace std;

  class Base {};

class Derived: public Base {};

int main()

{

   Derived d;

   // some other stuff

   try {

       // Some monitored code

       throw d;

   }

   catch(Base b) {

       cout<<"Caught Base Exception";
```

```
    }
    catch(Derived d) {  //This catch block is NEVER executed
          cout<<"Caught Derived Exception";
    }
    getchar();
    return 0;
}
```

If you change the order of the exception handler statements in the code above, the compiler can access both the statements easily. The following is a modified program of the above code, but it prints the following output: “Caught Derived Exception.”

```
#include<iostream>
using namespace std;
  class Base {};
class Derived: public Base {};
int main()
{
    Derived d;
    // some other stuff
    try {
         // Some monitored code
         throw d;
    }
    catch(Derived d) {
        cout<<"Caught Derived Exception";
    }
    catch(Base b) {
          cout<<"Caught Base Exception";
    }
    getchar();
    return 0;
}
```

In C++, the compiler may throw an error if it catches or goes through the exception handler code in the base class before it looks at the derived class. It will, however, continue to compile the code. Java, another object-oriented programming language, does not throw any exceptions or errors if this happens. Consider the example below. This code returns the following output: “exception Derived has already been caught.”

```
//filename Main.java
class Base extends Exception {}
class Derived extends Base  {}
public class Main {
  public static void main(String args[]) {
    try {
        throw new Derived();
    }
    catch(Base b) {}
    catch(Derived d) {}
  }
}
```

Differentiating Between Block and Type Conversions

Consider the following code:

```
#include <iostream>
using namespace std;

int main()
{
    try
    {
        throw 'x';
    }
    catch(int x)
    {
        cout << " Caught int " << x;
    }
    catch(...)
    {
        cout << "Default catch block";
    }
}
```

The output of this block of code is: “Default catch block.”

In the program above, an exception is thrown by the compiler in the form of a character. There is an exception handler block of code, but this is only to catch an int error in the code. You may think that the compiler will match the character's ASCII code and use the int exception handler to take care of the error, but this is not what happens in C++. Consider the following example where the exception handler code is not called for the object thrown as an error.

```
#include <iostream>

using namespace std;


class MyExcept1 {};

class MyExcept2

{

public:

     // Conversion constructor

    MyExcept2 (const MyExcept1 &e )

    {

        cout << "Conversion constructor called";

    }

};

int main()

{

    try

    {

        MyExcept1 myexp1;

        throw myexp1;

    }
    catch(MyExcept2 e2)
    {
        cout << "Caught MyExcept2 " << endl;
    }
    catch(... )
    {
        cout << " Default catch block " << endl;
    }
    return 0;
}
```

Chapter Twenty: Object Destruction and Error Handling

Before you run the code in C++, try to predict the output of the following code:

```
#include <iostream>

using namespace std;

  class Test {

public:

  Test() { cout << "Constructing an object of Test " << endl; }

  ~Test() { cout << "Destructing an object of Test "  << endl; }

};

  int main() {

  try {

    Test t1;

    throw 10;

  } catch(int i) {

    cout << "Caught " << i << endl;

  }

}
```

The output of this code is:

```
  Constructing an object of Test

  Destructing an object of Test

  Caught 10
```

When the compiler throws an exception, the code's destructor functions will be called to remove the objects whose scope ends with the entire block. This destructor is called before the compiler executes the exception handler code. For this reason, the code above gives you the output “Caught 10” after “Destructing an object of Test.” How do you think the compiler will act when it identifies an exception in the constructor? Consider the following example:

```
#include <iostream>

using namespace std;

  class Test1 {

public:

  Test1() { cout << "Constructing an Object of Test1" << endl; }

  ~Test1() { cout << "Destructing an Object of Test1" << endl; }

};

  class Test2 {

public:

  // Following constructor throws an integer exception

  Test2() { cout << "Constructing an Object of Test2" << endl;

             throw 20; }

  ~Test2() { cout << "Destructing an Object of Test2" << endl; }

};

  int main() {

  try {

    Test1 t1;  // Constructed and destructed

    Test2 t2;  // Partially constructed

   Test1 t3;  // t3 is not constructed as this statement never gets executed
  } catch(int i) {
    cout << "Caught " << i << endl ;
  }
}
```

The output of the above program is:

```
  Constructing an Object of Test1
  Constructing an Object of Test2
  Destructing an Object of Test1
  Caught 20
```

The compiler calls the destructor only when it uses completely constructed objects. If the constructor of the object leaves an exception, the compiler does not call for the destructor. Before you execute the following program, try to predict its outcome.

```
#include <iostream>
using namespace std;

class Test {
  static int count;
  int id;
public:
  Test() {
    count++;
    id = count;
    cout << "Constructing object number " << id << endl;
    if(id == 4)
```

```
            throw 4;
    }
    ~Test() { cout << "Destructing object number " << id << endl; }
};

int Test::count = 0;

int main() {
    try {
        Test array[5];
    } catch(int i) {
        cout << "Caught " << i << endl;
    }
}
```

Chapter Twenty-One: Searching Algorithms

A searching algorithm is designed to look for an element or print the same element from the program's data structures or variables. There are numerous algorithms you can use to search for elements in the structures. The algorithms are classified into the following categories:

- *Interval search* : An interval search is one where the algorithm will look for the element in a sorted structure. These algorithms are better to use when compared to the next category since the structure is broken down and divided into parts before the element is identified in the structure—for example, binary search.
- *Sequential search* : In these types of algorithms, the compiler moves from one element to the next to look for the element in the data structure. An example of this algorithm is linear search.

Let us look at how these search algorithms work in C++.

Linear Search

Let us understand how the search algorithm works in C++ using an example. Consider a problem where you have an array ‘arr[]’ with n elements; how would you look for the value ‘x’ in the arr[]?

```
Input : arr[] = {10, 20, 80, 30, 60, 50,

                    110, 100, 130, 170}

          x = 110;

Output : 6

Element x is present at index 6

Input : arr[] = {10, 20, 80, 30, 60, 50,

                    110, 100, 130, 170}

           x = 175;

Output : -1
```

Element x is not present in arr[].

The simplest way to perform a linear search is as follows:

1. Begin at the end of the array and compare the element you are looking for against each array element.
2. If the element matches one of the elements in your array, return the index
3. If the element does not match, move to the next element
4. If the element is not present in the array, return -1.

```
// C++ code to linearly search x in arr[]. If x

// is present then return its location, otherwise

// return -1


#include <iostream>

using namespace std;


int search(int arr[], int n, int x)

{

    int i;

    for (i = 0; i < n; i++)

        if (arr[i] == x)

            return i;

    return -1;

}


// Driver code

int main(void)
```

```
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int x = 10;
    int n = sizeof(arr) / sizeof(arr[0]);

    // Function call
    int result = search(arr, n, x);
    (result == -1)
        ? cout << "Element is not present in array"
        : cout << "Element is present at index " << result;
   return 0;
}
```

Binary Search

As mentioned earlier, a binary search is based on the interval search algorithm, where you look for an element in a sorted array. When compared to a linear search algorithm, a binary search algorithm has a higher time complexity.

A binary search uses the whole array as the interval when the search starts. It then breaks the interval into parts to look for the search element. It divides the array into half and looks for the element in the array's lower and upper sections. Depending on where the element lies, the algorithm will break the interval into a smaller section to look for it. It continues to do this until it finds the element.

A binary search aims to use the existing information in the array after it sorts the elements. This reduces the time complexity of the algorithm to O (log n). In a binary search, half the elements are not considered after making one comparison.

1. Sort the array.

2. Compare the search element x with the element in the middle of the array.
3. If x is less than the middle element, ignore the right section of the array since x can only lie in the left section of the array.
4. We then perform the same functions with the left section of the array.
5. If x is greater than the middle element in Step 3, we consider the array's right section.

We will now look at two ways to implement the binary search algorithm: recursive and iterative.

Before that, let us understand the time complexity of the binary search algorithm. You can calculate the time complexity of an algorithm using the following formula: T(n) = T(n/2) + c

You can remove the recurrence by using a master or recurrence tree method.

Recursive Implementation

```
// C++ program to implement recursive Binary Search

#include <bits/stdc++.h>

using namespace std;


// a recursive binary search function. It returns

// location of x in given array arr[l..r] is present,

// otherwise -1

int binarySearch(int arr[], int l, int r, int x)

{

    if (r >= l) {

        int mid = l + (r - l) / 2;
        // If the element is present at the middl e
        // itself
        if (arr[mid] == x)
            return mid;

        // If element is smaller than mid, then
        // it can only be present in left subarray
        if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);

        // Else the element can only be present
        // in right subarray
        return binarySearch(arr, mid + 1, r, x);
    }

    // We reach here when element is not
    // present in array
    return -1;
}

int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int x = 10;
    int n = sizeof(arr) / sizeof(arr[0]);
    int result = binarySearch(arr, 0, n - 1, x) ;
    (result == -1) ? cout << "Element is not present in array"
                   : cout << "Element is present at index " << result;
    return 0;
}
```

The output of the code is:

```
Element is present at index 3
```

Iterative Implementation

```
// C++ program to implement recursive Binary Search
#include <bits/stdc++.h>
using namespace std;

// a iterative binary search function. It returns
// location of x in given array arr[l..r] if present,
// otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    while (l <= r) {
        int m = l + (r - l) / 2;

        // Check if x is present at mid
        if (arr[m] == x)
            return m;
        // If x greater, ignore left half
        if (arr[m] < x )
            l = m + 1;

       // If x is smaller, ignore right half
        else
            r = m - 1;
    }

    // if we reach here, then element was
    // not present
    return -1;
}

int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
   int x = 10;
    int n = sizeof(arr) / sizeof(arr[0]);
    int result = binarySearch(arr, 0, n - 1, x);
    (result == -1) ? cout << "Element is not present in array"
                   : cout << "Element is present at index " << result;
    return 0;
}
```

The output of the code is: Element is present at index 3

Jump Search

The jump search algorithm is similar to the binary search algorithm in the sense that it looks for the search element in a sorted array. This algorithm's objective is to search for the search element from fewer elements in the array. The compiler can jump ahead by skipping a few elements or jumping ahead by a few steps.

Let us understand this better using an example. Suppose you have an array with n elements in it, and you need to jump between the elements by m fixed steps. If you want to look for the search element in the array, you begin to look at the following indices a[0], a[m], a[2m], ….. a[km]. When you find the interval where the element may be, the linear search algorithm kicks in.

Consider the following array: (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610). The length of this array is 16. Let us now look for element 55 in the array. The block size is 4, which means the compiler will jump four elements each time.

Step 1: The compiler moves from index 0 to 3.

Step 2: The compiler moves from 4 to 8.

Step 3: The compiler jumps from 9 to 12.

Step 4: The element in position 12 is larger than 55, so we go back to the previous step.

Step 5: The linear search algorithm kicks in and looks for the index of the element.

Optimal Block Size

When you use the jump search algorithm, you need to choose the right block size, so there are no issues in the algorithm. In the worst-case scenario, you need to perform n/m jumps. If the element the compiler last checked was greater than the search element, you need to perform m-1 comparisons when the linear search algorithm kicks in. Therefore, in the worst-case scenario, the number of jumps should be ((n/m) + m-1). This function's value will be minimum if the value of the element ‘m’ is square root n. The step size, therefore, should be m = √n.

```
// C++ program to implement Jump Search


#include <bits/stdc++.h>

using namespace std;


int jumpSearch(int arr[], int x, int n)

{

    // Finding block size to be jumped

    int step = sqrt(n);


    // Finding the block where element is

    // present (if it is present)

    int prev = 0;

    while (arr[min(step, n)-1] < x)

    {

        prev = step;

        step += sqrt(n);

        if (prev >= n)

            return -1;

    }


    // Doing a linear search for x in block
    // beginning with prev.
    while (arr[prev] < x)
    {
        prev++ ;

        // If we reached next block or end of
        // array, element is not present.
       if (prev == min(step, n))
            return -1;
    }
    // If element is found
    if (arr[prev] == x)
        return prev;

    return -1;
}

// Driver program to test function
int main()
{
    int arr[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21,
                34, 55, 89, 144, 233, 377, 610 };
    int x = 55;
   int n = sizeof(arr) / sizeof(arr[0]);
```

```
    // Find the index of 'x' using Jump Search
    int index = jumpSearch(arr, x, n);

    // Print the index where 'x' is located
    cout << "\nNumber " << x << " is at index " << index ;
    return 0;
}
```

The output of this code: Number 55 is at index 10

Some important points to note about this algorithm are:

- This algorithm only works when an array is sorted
- Since the optimal length the compiler should jump is √ n, the time complexity of this algorithm is O (√ n). This means the time complexity of this algorithm is between the binary search and linear search algorithms
- The jump search algorithm is not as good as the binary search algorithm, but it is better than the binary search algorithm since the compiler only needs to move once back through the array. If the binary search algorithm is too expensive, you should use the jump search algorithm.

Chapter Twenty-Two: Sorting Algorithms

You use sorting algorithms when you want to arrange a list or array of elements based on the comparison operator you want to include. This comparison operator will be used to decide how the elements will be sorted in the data structure where you want to arrange the elements. Let us look at some common sorting algorithms.

Bubble Sort

The bubble sort algorithm is one of the simplest algorithms used in C++, and it works by swapping the elements adjacent to each other in the array in case they are not in the right order.

Consider the following example:

First Pass

(5 1 4 2 8) –> (1 5 4 2 8): In this step, the algorithm will compare the elements in the array and swap the numbers 1 and 5.

(1 5 4 2 8) –> (1 4 5 2 8): In this step, the numbers 4 and 5 are swapped since the number 5 is greater than 4.

(1 4 5 2 8) –> (1 4 2 5 8): In this step, the numbers 5 and 2 are swapped.

(1 4 2 5 8) –> (1 4 2 5 8): In the last step, the elements are ordered, so there is no more swapping necessary.

Second Pas s

(1 4 2 5 8) –> (1 4 2 5 8)

(1 4 2 5 8) –> (1 2 4 5 8): In this step, the numbers 4 and 2 are swapped since the number 4 is greater than 2.

(1 2 4 5 8) –> (1 2 4 5 8)

(1 2 4 5 8) –> (1 2 4 5 8)

The array is already sorted, but since the compiler is not aware that the algorithm is complete, it will complete another round on the array and check if any elements need to be swapped.

Third Pass

(1 2 4 5 8) –> (1 2 4 5 8)

(1 2 4 5 8) –> (1 2 4 5 8)

(1 2 4 5 8) –> (1 2 4 5 8)

(1 2 4 5 8) –> (1 2 4 5 8)

Consider the following implementations of the bubble sort algorithm:

```
// C++ program for implementation of Bubble sort
#include <bits/stdc++.h>
using namespace std;

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// a function to implement bubble sort
void bubbleSort(int arr[], int n)
{
    int i, j;
    for (i = 0; i < n-1; i++)

    // Last i elements are already in place
    for (j = 0; j < n-i-1; j++)
        if (arr[j] > arr[j+1])
            swap(&arr[j], &arr[j+1]);
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver code
int main()
{
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);
    bubbleSort(arr, n);
    cout<<"Sorted array: \n";
    printArray(arr, n);
    return 0;
}
```

The output of this code is:

```
Sorted array:

11 12 22 25 34 64 90
```

We can optimize the implementation of this sorting algorithm. The above code runs more number of times than necessary, although the array is sorted. You cannot stop the optimization since the inner loop does not perform any swaps.

```
// Optimized implementation of Bubble sort

#include <stdio.h>


void swap(int *xp, int *yp)

{

    int temp = *xp;

    *xp = *yp;

    *yp = temp;

}


// An optimized version of Bubble Sort

void bubbleSort(int arr[], int n)

{

   int i, j;

   bool swapped;

   for (i = 0; i < n-1; i++)

  {

     swapped = false;
```

```
        for (j = 0; j < n-i-1; j++)
        {
            if (arr[j] > arr[j+1])
            {
                swap(&arr[j], &arr[j+1]);
                swapped = true;
            }
        }

        // IF no two elements were swapped by inner loop, then break
        if (swapped == false)
            break;
    }
}

/* Function to print an array */
void printArray(int arr[], int size)
{
     int i;
     for (i=0; i < size; i++)
          printf("%d ", arr[i]);
    printf("n");
}
```

```
// Driver program to test above functions
int main()
{
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);
    bubbleSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

The output of the above code is:

```
Sorted array:
11 12 22 25 34 64 90
```

Selection Sort

Using the selection sort algorithm, you can sort the elements in the array by looking at the smallest element in the array in ascending order. The compiler will only perform this sorting algorithm in the part of the array with unsorted elements. The algorithm divides the array into two arrays:

- The section of the array with sorted elements
- The section of the array which does not have any sorted elements

When the selection sort algorithm iterates, the element which is the smallest in the array from the unsorted section of the array. This element is then moved to the sorted section of the array. Consider the following section:

```
arr[] = 64 25 12 22 11


// Find the minimum element in arr[0...4]
// and place it at the beginning
11 25 12 22 64

// Find the minimum element in arr[1...4]
// and place it at the beginning of arr[1...4]
11 12 25 22 64

// Find the minimum element in arr[2...4]
// and place it at the beginning of arr[2...4]
11 12 22 25 64

// Find the minimum element in arr[3...4]
// and place it at beginning of arr[3...4]
11 12 22 25 64
```

Let us write down the above example in the form of a program:

```
// C++ program for implementation of selection sort
#include <bits/stdc++.h>
using namespace std;

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
```

```
}

void selectionSort(int arr[], int n)
{
    int i, j, min_idx;

    // One by one move boundary of unsorted subarray
    for (i = 0; i < n-1; i++)
    {
        // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i+1; j < n; j++)
        if (arr[j] < arr[min_idx])
            min_idx = j;

        // Swap the found minimum element with the first element
       swap(&arr[min_idx], &arr[i]);
    }
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver program to test above functions
int main()
{
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);
    selectionSort(arr, n);
    cout << "Sorted array: \n";
    printArray(arr, n);
    return 0;
}
```

The output of the program is:

```
Sorted array:
11 12 22 25 64
```

Insertion Sort

The insertion sort algorithm is a simple and straightforward algorithm that works in the same way you would sort or arrange playing cards. In this algorithm, the array is split into two parts – sorted and unsorted arrays. The values in the unsorted array will then be sorted and moved into the right positions.

Let us understand this algorithm better by considering how it works when the compiler tries to sort the elements in the ascending order:

1. Create an array with n elements.
2. Iterate from the first element in the array to the nth element in the array
3. Compare every element in the array with its predecessor
4. If the element is smaller than the predecessor, you should compare it to the other elements before the predecessor. Move the elements around to ensure there is enough space for the element which is swapped

Consider the following example: The array is 12, 11, 13, 5, 6

12, 11, 13, 5, 6

We will add a loop where the iterative element ‘i’ is assigned the value 1. Since there are only 5 elements in the array, the value of ‘i' can be incremented until 4.

i = 1. Since element 11 is smaller than 12, the number 12 is moved after element 11.

11, 12, 13, 5, 6

i = 2. The number 13 will stay in its position since the first three elements are sorted.

11, 12, 13, 5, 6

i = 3. The number 5 will move to the start of the array since it is the smallest number when compared to the other elements in the array. The other elements will be moved ahead.

5, 11, 12, 13, 6

i = 4. The number 6 will move next to 5 since it is smaller than all the other elements in the array.

5, 6, 11, 12, 13

Let us write the above example in C++.

```
// C++ program for insertion sort
#include <bits/stdc++.h>
using namespace std;

/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;

       /* Move elements of arr[0..i-1], that are
        greater than key, to one position ahead
       of their current position */
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

```
// a utility function to print an array of size n
void printArray(int arr[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

/* Driver code */
int main()
{
    int arr[] = { 12, 11, 13, 5, 6 };
    int n = sizeof(arr) / sizeof(arr[0]);

    insertionSort(arr, n);
    printArray(arr, n);

    return 0;
}
```

Quicksort

The quicksort algorithm is a divide and conquer algorithm. In this algorithm, the compiler picks an element in the array as the pivot and divides the elements in the array based on that pivot. There are different ways to implement the quick sort algorithm in C++.

1. The algorithm can choose the first element in the array as the pivot
2. The algorithm can choose the last element as the pivot (we will look at this in detail in the section below)
3. Choose any element in the array as the pivot
4. Choose the median of the elements in the array and pick that as the pivot

One of the most important processes in a quick sort algorithm is the partition() function. This function's target is to take an element from the array as the pivot and put it in the right position. The elements in the array which are smaller than the pivot will be moved to one side of the array while the others will move to the other section of the pivot. Consider the following pseudo-code for this algorithm:

```
/* low  --> Starting index,  high  --> Ending index */

quickSort(arr[], low, high)

{

    if (low < high)

    {

        /* pi is partitioning index, arr[pi] is now

           at right place */

        pi = partition(arr, low, high);


        quickSort(arr, low, pi - 1);  // Before pi

        quickSort(arr, pi + 1, high); // After pi

    }

}
```

Understanding the Partition Algorithm

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

The pseudo code for the partition algorithm is:

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
    array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
  of pivot */
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1)  // Index of smaller element

    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

Let us look at the illustration of this function:

```
arr[] = {10, 80, 30, 90, 40, 50, 70}
Indexes:  0   1   2   3   4   5   6

low = 0, high =  6, pivot = arr[h] = 70
Initialize index of smaller element, i = -1

Traverse elements from j = low to high-1
j = 0 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 0
arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j
                                     // are same

j = 1 : Since arr[j] > pivot, do nothing
// No change in i and arr[]

j = 2 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 1
arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30

j = 3 : Since arr[j] > pivot, do nothing
```

// No change in i and arr[]

j = 4 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 2

arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped

j = 5 : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]

i = 3

arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped

We come out of loop because j is now equal to high-1.

Finally we place pivot at correct position by swapping

arr[i+1] and arr[high] (or pivot)

arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped

Now 70 is at its correct place. All elements smaller than

70 are before it and all elements greater than 70 are after

it.

Let us look at how to implement this algorithm in C++:

```
/* C++ implementation of QuickSort */
#include <bits/stdc++.h>
using namespace std;

// a utility function to swap two elements
void swap(int* a, int* b)
```

```
{
    int t = *a;
    *a = *b;
    *b = t;
}

/* This function takes last element as pivot, places
the pivot element at its correct position in sorted
array, and places all smaller (smaller than pivot)
to left of pivot and all greater elements to right
of pivot */
int partition (int arr[], int low, int high)
{
    int pivot = arr[high]; // pivot
    int i = (low - 1); // Index of smaller element

    for (int j = low; j <= high - 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++; // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

/* The main function that implements QuickSort
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
        at right place */
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

```
/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver Code
int main()
{
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    quickSort(arr, 0, n - 1);
    cout << "Sorted array: \n";
    printArray(arr, n);
    return 0;
}
```

Chapter Twenty-Three: Tips to Optimize Code in C++

When you write code in C++ or any other programming language, your main objective should be to write code that works correctly. Once you accomplish this, you need to change the code to improve the following:

1. The security of the code
2. The quantity of memory used while running the code
3. Performance of the code

This chapter gives you a brief idea of the areas to consider if you want to improve the performance of your code. Some points to keep in mind are:

- You can use numerous techniques to improve the performance of your code. This method, however, can lead to the creation of a larger file.
- If you choose to optimize multiple areas in your code at the same time, it may lead to some conflict between the areas of your code. For instance, you may not be able to optimize both the performance of the code and memory use. You need to strike a balance between the two.
- You may always need to optimize your code, and this process is never-ending. The code you write is never fully optimized. There is always room to improve some parts of your code if you want the code to run better.
- You can use different tricks to improve the performance of the code. While you do this, you should ensure that you do not forget about some coding standards. Therefore, do not use cheap tricks to make the code work better.

Using the Appropriate Algorithm to Optimize Code

Before you write any code, you need to sit down and understand the task. You then need to develop the right algorithm to use to optimize the code. We are going to understand how the algorithm affects your code using a simple example. In the program, we are going to use a two-dimensional segment to identify the maximum value and, for this, we will take two whole numbers. In the first code, we will not look at the program's performance. We will then look at a few methods to use to improve the performance of the code.

Consider the following parameters used in the code: both numbers should lie between the interval [-100, 100]. The maximum value is calculated using the function: (x * x + y * y) / (y * y + b).

There are two variables used in this function – x and y. We are also using a constant ‘b’ which is a user-defined value. The value of this constant should always be greater than zero but less than 1000. In the example below, we do not use the pow() function from the math.h library.

```
#include <iostream>

#define LEFT_MARGINE_FOR_X -100.0

#define RIGHT_MARGINE_FOR_X 100.0

#define LEFT_MARGINE_FOR_Y -100.0

#define RIGHT_MARGINE_FOR_Y 100.0

using namespace std;

int

main(void)

{

//Get the constant value

cout<<"Enter the constant value b>0"<<endl;

cout<<"b->"; double dB; cin>>dB;

if(dB<=0)   return EXIT_FAILURE;

if(dB>1000) return EXIT_FAILURE;

//This is the potential maximum value of the function

//and all other values could be bigger or smaller
double dMaximumValue = (LEFT_MARGINE_FOR_X*LEFT_MARGINE_FOR_X+LEFT_MARGINE_FOR_Y*LEFT_MARGINE_FOR_Y)/(LEFT_MARGINE_FOR_Y*LEFT_MARGINE_FOR_Y+dB);

double dMaximumX = LEFT_MARGINE_FOR_X;

double dMaximumY = LEFT_MARGINE_FOR_Y;

for(double dX=LEFT_MARGINE_FOR_X; dX<=RIGHT_MARGINE_FOR_X; dX+=1.0)

  for(double dY=LEFT_MARGINE_FOR_Y; dY<=RIGHT_MARGINE_FOR_Y; dY+=1.0)

    if( dMaximumValue<((dX*dX+dY*dY)/(dY*dY+dB)))

    {

      dMaximumValue=((dX*dX+dY*dY)/(dY*dY+dB));

      dMaximumX=dX;

      dMaximumY=dY;

    }

cout<<"Maximum value of the function is="<< dMaximumValue<<endl;

cout<<endl<<endl;

cout<<"Value for x="<<dMaximumX<<endl

    <<"Value for y="<<dMaximumY<<endl;

            return EXIT_SUCCESS;

}
```

Look at the code carefully. You notice that the function and value dX * dX is run by the process too many times, and the value is stored multiple times in the memory. This is a waste of CPU time and memory. What do you think we could do to improve the speed of the code? An alternative to writing the operation multiple times in the code is to declare a variable and assign this function to it. Let us define a variable d, which stores the value of the function dX * dX. You can use the variable ‘d’ everywhere in the code where you need to use the calculation. You can optimize other sections of the above code as well. Try to spot those areas.

The next area we need to look at is how general the lines of code are. You need to see whether the program runs as fast as you want it to. If you want to increase the speed of the algorithm, you need to tweak some functions based on the size of your input. What does this mean?

You can improve the speed of the code you have written using multiple algorithms instead of only one algorithm. When you use two algorithms, you can instruct the compiler to switch between the algorithms based on a condition.

Optimizing Code

When you write code, every element in your code uses some space in the memory. It is important to understand how each word in your code uses memory to reduce consumption or usage. Let us consider a simple example where we try to swap the values in two variables in the memory. You can do this using numerous sorting algorithms. To understand this better, let us take a real-world example – you have two people sitting in two different chairs. You introduce a third or temporary chair to hold one of the individuals when they want to swap chairs.

Consider the following code:

```
int nFirstOne =1, nSecondOne=2;

int nTemp = nFirstOne;

nFirstOne = nSecondOne;

nSecondOne = nTemp;
```

This code is easy to use, but when you create a temporary variable in your code, the compiler will assign some space in the memory for this object. You can avoid wasting memory space by avoiding the usage of a temporary variable in the code.

```
int nFirstOne = 3, nSecondOne = 7;

nFirstOne += nSecondOne;

nSecondOne = nFirstOne ? nSecondOne;

nFirstOne -= nSecondOne;
```

You may need to swap large values in the memory to a different section. How would you do this? The easiest way to do this is to use pointers. Instead of copying the same value across the memory, use a pointer to obtain the address of the value in the memory. You can then change their address instead of moving the value from one location to the next in the memory.

You may wonder how you can determine if your code is faster or how you can calculate this. When you finish writing your code, the system will translate it into a language it understands using the assembler. It will then translate this into machine code, which it quickly interprets. Every operation you write in the code takes place in the processor. It may also take place in the graphic card or mathematical coprocessor.

One operation can take place in one clock cycle, or it may take a few. For this reason, it is easier for the computer to multiply numbers compared to division. This could be the case because of the optimization the computer performs. You can also leave the task of optimization to the compiler in some cases.

If you want to learn more about how fast your code is, you should know the architecture of the computer you are using. The code can be faster because of one of the following reasons:

1. The program runs in the cache memory
2. The mathematical coprocessor processes sections of the code
3. A branch predictor was used correctly by the compiler

Let us now consider the following numbers: O(n), O(log(n) *n), n*n, n!. When you use this type of code, the program's speed depends on the number you key into the system. Let us assume you enter n = 10. The program may take ‘t’ amount of time to run and compile. What do you think will happen when you enter n = 100? The program may take 10 times longer to run. It is important to understand the limits a small number can have on your algorithm.

Some people also take time to see how fast the code runs. This is not the right thing to do since not every program or algorithm you key in is completed first by the processor. Since an algorithm does not run in the computer’s kernel mode, the processor can get another task to perform. This means the algorithm is put on hold. Therefore, the time you write down is not an accurate representation of how fast the code can run. If you have more than one processor in the system, it is harder to identify which processor is running the algorithm. It is tricky to calculate the speed at which the processor completes running your code.

If you want to optimize or improve the speed at which the program runs, you need to prevent the processor from shifting the code to a different core during the run. You also need to find a way to prevent the counter from switching between tasks since that only increases the time the processor takes to run the code. You may also notice some differences in your code since the computer does not transfer all optimizations into machine code.

Using Input and Output Operators

When you write code, it is best to identify the functions you can use which do not occupy too much space in the memory. Most times, you can improve the speed of the program by using a different function to perform the same task. Printf and scanf are two functions used often in C programming, but you can use the same keywords in C++ if you can manipulate some files. This increases the speed of the program and can save you a lot of time and memory.

Let us understand this better through an example. You have two numbers in a file and need to read those numbers. It is best to use the keywords cin and cout on files in terms of security since you have instructions passed to the compiler from the header library in C++. If you use printf or scanf, you may need to use other functions of keywords to increase the speed of the program. If you want to print strings, you can use the keyword put or use an equivalent from file operations.

Optimizing the Use of Operators

You need to use operators to perform certain functions in C++. Basic operators, such as +=, *= or -= use a lot of space in the memory. This is especially true when it comes to basic data types. Experts recommend you use a postfix decrement or increment along with the prefix operator if you want to improve the functioning of the code. You may also need to use the << or >> operators instead of division and multiplication, but you need to be careful when you use those operators. This may lead to a huge mistake in the code. It takes some time to identify these mistakes and, to overcome the mistake, you need to add more lines of code. This is only going to reduce the speed and performance of the program.

It is best to use bit operators in your code since these increase the speed of the program. If you are not careful about how you use these operators, you may end up with machine-dependent code, and this is something you need to avoid.

C++ is a hybrid language and allows you to use an assembler’s support to improve the functioning of your program. It also allows you to develop solutions to problems using object-oriented programming. If you are adept at coding, you can develop libraries to improve your code's functioning.

Optimization of Conditional Statements

You may need to include numerous conditional statements in your code, depending on the type of code you are writing. Most people choose to use the ‘if’ conditional statement, but it is advised that you do not do this. It is best to use the switch statement. When you use the former conditional statement, the compiler needs to test every element in the code, and this creates numerous temporary variables to store the code. This reduces the performance of the code.

It is important to note that the ‘if’ conditional statement has many optimizations built into the statement itself. If you only have a few conditions to test, and if these are connected to the or operator, you can use the ‘if’ conditional statement to calculate the value. Let us look at this using an example. We have two conditions, and each of these uses the and operator. If you have two variables and want to test if both values are equal to a certain number, you use the and operator. If the compiler notes that one value does not meet the condition, it returns false and does not look at the second value.

When you use conditional statements in your code, it is best to identify the statements which often occur before the other conditional statements. This is the best way to determine if an expression is true or false. If you have too many conditions, you need to sort them and split them into a nested conditional statement. There may be a possibility that the compiler does not look at every branch in the nest you have created. Some lines of code may be useless to the compiler, but they simply occupy memory.

You may also come across instances where you have long expressions with numerous conditions. Most programmers choose to use functions in this instance, but what they forget is that functions take up a lot of memory. They create calls and stacks in memory. It is best to use a macro to prevent the usage of memory. This increases the speed of the program. It is important to remember that negation is also an operation you can use in your code.

Dealing with Functions

If you are not careful when you use functions, you may end up with bad code. Consider the following example. If you have code written in the same format as the statements below, it will lead to a bad code:

```
for(int i=1; i<=10; ++i)
    DoSomething(i);
```

Why do you think this is the case? When you write some code similar to the above, you need to call the function a few times. It is important to remember that the calls the compiler makes to the functions in the code use a lot of memory. If you want to improve the performance of the code, you can write the statement in the following format:

```
DoSomething(n);
```

The next thing you need to learn more about is inline functions. The compiler will use an inline function similar to a macro if it is small. This is one way to improve the performance of your code. You can also increase the reusability of the code in this manner. When you pass large objects from one function to another, it is best to use references or pointers. It is better to use a reference since this allows you to write code that is easy to read. Having said that, if you are worried about changing the value of the actual variable being passed to the function, you should avoid using references. If you use a constant object, you should use the keyword const since it will save you some time.

It is important to note that the arguments and parameters passed in the function will change depending on the situation. When you create a temporary object for a function, it will only reduce the speed of the program. We have looked at how you can avoid using or creating temporary variables in the code.

Some programmers use recursive functions depending on the situation. Recursive functions can slow the code down. So you should avoid the use of recursive functions if you can since these reduce the performance of your code.

Optimizing Loops

Let us assume you have a set of numbers, and you are to check if the value is greater than 5 or less than 0. When you write the code, you need to choose the second option. It is easier for the compiler to check if a value is greater than zero than to check which number is greater than 10. In simple words, the statement written below makes the program slower when compared to the second statement in this section.

```
for( i =0; i<10; i++)
```

As mentioned, it is best to use this loop instead of the above. If you are not well-versed with C++ programming, this line of code may be difficult for you to read.

```
for(i=10; i--; )
```

Similarly, if you find yourself in a situation where you need to pick from <=n or !=0, you should choose the second option since that is faster. For instance, if you want to calculate a factorial, do not try to use a loop since you can use a linear function. If you ever find yourself in a situation where you need to choose between a few loops or one loop with different tasks, you should choose the second option. This method may help you develop a better performing program.

Optimizing Data Structures

Do you think a data structure affects the performance of your code? It is not easy to answer this question. Since data structures are used everywhere in your code, the answer is difficult to formulate and vague. Let us look at the following example to understand this better. If you are tasked with creating permutations (using the pattern below), you may choose to use a linked list or array.

1, 2, 3, 4,

2, 3, 4, 1,

3, 4, 1, 2,

4, 1, 2, 3,

If you use an array, you can copy the first element in the array and move every other element in the array towards that element. You then need to move the first element in the array to the end of the list. To do this, you need to use multiple operations, and your program will be very slow. If you leave the data in a list, you can develop a program, which will improve the performance of the code. You can also store the data in the form of a tree. This data structure allows you to develop a faster program.

Bear in mind that the type of data structure you use affects the performance of your program. You can solve any problem you have in the code without using arrays or any other data structure.

Sequential or Binary Search?

When you look for a specific object or variable in the code, which method should you opt for – binary or sequential search?

No matter what you do in your code, you always look for some value in a data structure. You may need to look for data in tables, lists, etc. There are two ways to do this:

1. The first method is simple. You create an array and assign some values to the array. If you want to look for a specific value in the array, you need to start looking at the start of the array until you find the value in the array. If you do not find the value at the start of the array, the compiler moves to the end of the array. This reduces the speed at which the program is compiled.
2. In the second strategy, you need to sort the array before you search for an element in the array. If you do not sort the array before you look for the element, you cannot obtain the results on time. When the array is sorted, the compiler will break it into two parts from the middle. It will then look for the value in either part of the array depending on the values in the sections. When you identify the part where the element may be, you need to divide it through the middle again. You continue to do this until you find the value you are looking for. If you do not, then you know the array does not have the value.

What is the difference between these strategies? When you sort the elements in the array, you may lose some time. Having said that, if you give the compiler time to do this, you will benefit faster from the search. When it comes to choosing between a sequential and binary search, you need to understand the problem before you implement the method you want to use.

Optimizing the Use of Arrays

We looked at arrays in the previous book, and this is one of the basic data structures used in C++. An array contains a list of objects of a similar data type. Every object in an array holds a separate location in the memory.

If you want to learn more about optimizing the work or use of an array, you need to understand the structure of this structure. What does this mean? An array is similar to a pointer, and it points to the elements in the array. You can access array methods using arithmetic pointers or any other type of pointer if needed. Consider the following example:

```
for(int i=0; i<n; i++) nArray[i]=nSomeValue;
```

The code below is better than the statement above. Why do you think this is the case?

```
for(int* ptrInt = nArray; ptrInt< nArray+n; ptrInt++) *ptrInt=nSomeValue;
```

The second line of code is better than the first line of code since the operations rely only on pointers. In the example above, we are using pointers to access the values stored in the integer data type. The pointer takes the address of the variable in the memory. In the case of the example, the pointer points to the variable nArray. When we add the increment operator to the variable, the pointer will move from the first element in the array to the next until it reaches the end of the array. If you use the double data type, the compiler will know how far it should move the address.

It is difficult to interpret and read the code using this method, but this is the only way to increase the speed of the program. In simple words, if you do not use a good algorithm, you can increase compiling speed by writing code using the right syntax.

Consider the following example: You have a matrix with the required elements. A matrix is a type of array, and it will be stored in your memory based on the rows. So, how do you think you should access the elements in the array? You should access every element in the matrix row by row. It does not make sense for you to use any other method because you reduce the speed of the program.

It is best to avoid initializing large sections of the memory for only one element. If you know the size of the element, make sure to stick to that size. Do not allot more memory space. You can use the function memset or other commands to allot some space in the memory to the variables used in the code.

Let us assume you want to create an array of characters or strings. Instead of defining the variables or assigning the array to specific variables, it is best to use pointers. You can assign each element in the array to a string, but this would only reduce the speed of the program. The compiler will run the code faster, even if the file is big. If you use the new keyword to create or declare an array in the code, your program will not do well since it will use a lot of memory the minute you try to run the code. It is for this reason you should use vectors. These objects add some space to your memory, allowing the program to do well.

If you want to move large volumes of data from one section in the memory to another, it is best to use an array of pointers. When you do this, you do not change the original values of the data but only replace the addresses of the objects stored in the memory.

Chapter Twenty-Four: Debugging and Testing

Before we look at the different aspects of debugging and testing, let us understand what these terms mean.

Definition

Testing

Testing is the process of identifying the behavior of the code and identifying the correct behavior. You can test the code at different stages of developing the code, including:

1. Module development
2. Requirements analysis
3. Interface design
4. Algorithm development
5. Implementation
6. Integration

In this chapter, we will look at what implementation testing is and how it helps to improve the functioning of the code. It is important to note that implementation testing does not mean you only test the code when you execute it. You can also perform this testing to check the correctness of the code used. Some programmers also use peer review to help them improve their code.

Debugging

It is important to note that debugging is an activity every coder must perform to ensure the code runs correctly. During the debugging process, you can correct any lines in the code which do not run correctly. Implementation testing is very different from debugging since the latter is used to locate any errors in the code, while the former is to test whether the code gives you the correct output. The testing strategies you implement during debugging and testing are based on this difference.

Conditions for Debugging

It is important to avoid spending too much time when you debug code. You, as a programmer, should be prepared. You need to put in a lot of effort to debug the code. Follow the steps given below to prepare yourself for the arduous task:

Understand the Algorithm and Design

It becomes very difficult to debug the code written if you do not understand the algorithm and design. You cannot test the module if you do not understand the design since you have no idea what the objective of the module is. If you do not understand the algorithm, you cannot locate any error in the code when you test it. Another reason why it is important to understand the algorithm is the test cases. If you do not know how the algorithm functions, you cannot develop effective test cases, and this is true when you use data structures in your code.

Check the Correctness of the Code

There are numerous methods used to check if the code is correct and runs smoothly.

Proof of Correctness

One of the easiest ways to check for any errors in the code is to examine every algorithm used in the code using some methods. For instance, if you know the invariants, preconditions, postconditions, and terminating conditions in a loop statement, then you can perform some easy checks in the code. Here are some questions you can ask to determine the correctness of the code:

1. If the compiler enters the loop, does this mean the invariant used is true based on the precondition?
2. If the compiler moves through the loop, does it indicate that the loop is closer to termination?
3. If the loop is nearing the end, does it mean the compiler will move towards the postcondition?

Some of these checks may not indicate the errors in the code, but you will understand the algorithm better.

Code Tracing

It can be easy to detect some errors in the code by tracing how the modules or functions execute, especially when calls are made to the function or module in different parts of the program. Experts suggest that you, as the code writer, should trace the working of the modules and functions along with someone else. If you want this process to be effective, you should trace the modules and functions by assuming that other code work functions and procedures work accurately. You may need to deal with levels or layers of abstraction in the procedure and function. Tracing does not catch all errors, but it will improve your understanding of the algorithm used.

Peer Reviews

As the name suggests, peer review is asking a peer to examine and check your code for any errors. If you want the review to be effective, you must ensure the peer has the required information and knowledge to check the code. You may also need to give the peer the code in advance so they know what to read or expect.

As the code writer, you should meet with the reviewer and explain how the algorithms in the code work. If the reviewer disagrees or does not understand some parts of the implementation, you need to discuss it with him until you both reach an agreement. The reviewer’s objective should be to detect the errors in the code. You can then correct the errors identified.

You can identify or discover errors in your code when you review it. Having said that, it is useful if you have someone from the outside looking at the code and identifying some blind spots in the code. Peer reviews, like code tracing, may take some time. Make sure to restrict the reviews only to those sections in the code where you know there can be some problems.

Anticipate Errors

It is unfortunate that people make errors when they write code, and some arguments may not be called accurately by the functions and modules. We also make mistakes when it comes to tracing the code, and peer reviews may not catch all the errors in the code. Therefore, you need to be prepared for the errors your code may run into using the requirements mentioned below.

Debugging Requirements

You need to have two capabilities in your code when you try to debug your program. The first thing to do is call the functions and services used in the module, while the second thing to do is obtain the information about the results of those calls. You also need to learn more about the internal state and data available in the function or module.

How to Drive the Module

When you try to debug any module in your code, you should ensure there is a method available to call the services or functions used in the module. This can be done using the following methods:

Hardwired Drivers

Hardware drivers are sections in the main program which contain a sequence of calls to various functions in the program. You can modify the sequence of the calls by rewriting the code in the driver. If you want to test the modules which vary because of different variables, it is best to use a hardwired driver. These drivers cannot work with numerous cases, which is a shortcoming.

Command Onterpreters

When the compiler runs the code in the program, it uses a command interpreter to test the code by running the input statements and interpret the commands which execute the calls to functions and modules. You can design these interpreters so you can enter the command either from a file or interactively. It is best to use an interactive command interpretation, especially in the first stages of debugging, but it is best to use a batch mode in later stages. A disadvantage of using a command interpreter is that it is slightly complicated to write. You may also spend too much time debugging the code you have written for the interpreter. This disadvantage is overcome since most of the code can be used to test other sections of the module or function.

Learn More About the Module

When it comes to debugging, you need to learn more about the program's various modules and functions. It makes no sense to control the functions and their sequence if you do not have the information about the effect of those functions on your code's variables and data structures. If the statements provide an output, then you have enough information available. Having said that, most functions and statements in the code will change some aspects of the module. This would mean you need to learn more about the modules for debugging.

Module State

Most data structures and modules in C++ allow you to insert and delete data used in the module. These statements do not generate the necessary output because these statements do not return the information from the arguments and parameters entered in the functions. Therefore, if you want to debug or test any code in the module, include the right statements in the code to display the workings or changes made in the module. Most programmers add some procedures to display the various contents in the modules. The compiler uses the procedures while testing the code, but it removes them when the testing is complete. It is important to have the compiler show the internal structure of the modules during debugging.

Module Errors

If you develop modules with complex statements, it is hard to determine where the error has occurred. It is possible the compiler may call upon the wrong private subroutine. To avoid such errors, you need to write practical code.

Execution

To locate where the errors in the module are, it is important to know which subroutine or program the compiler used when the error occurred. If you want to know when the compiler is in the execution state, add print statements to the code to indicate when the compiler enters and exits some sections of the code.

Debugging Principles

Report Conditions

Most programmers spend a lot of time identifying the errors in the code. It is important to detect these errors early so that you identify the cause easily.

If the compiler detects the error early on, it is important to find the cause. If the compiler detects the errors in the module early, you can identify the cause easily. If you detect the errors in the code only when the symptoms of the errors appear, it may be hard to identify the code.

Improve the Ease of Interpretation and Useful Information

It is important to maximize the information you obtain on executing debugging code. It is also important to learn to interpret the meaning of the information. As a programmer, you need to interpret the data and detect the error location in the code. You cannot rely on the error or debugging codes you write since each module relies on the entire structure. Therefore, it is important to display the entire structure in an easily understandable form.

Avoid Distracting and Meaningless Information

If you have too much information in the code, it can be a little overwhelming. If you have very little information, the error testing process is rendered useless. Let us assume you have a printout of all the times the compiler has entered and exited the code. When you look at this sheet, you cannot identify where the first error was identified. You should only use module execution reports only if you know where the error has occurred. As a rule, you should prefer code that tells you where the problem is and not that it is not in the code you are looking at.

Do Not Use Single-Use Testing Code

Most programmers make the mistake of using one single test code to check the error in the entire code. This code is extremely complex to write and understand. How would you feel if you were testing the debugging code, but there is an error in the debugging code itself? This is a waste of time. It is only practical to write complex test scripts if you know you can use different parts of the code in other debugging methods.

Functionalities to Use

Every programming language has some built-in functionalities you can use to debug the code.

Assertion Statements

Some C++ procedures have assertion procedures in the code that only take one parameter, often a Boolean statement. When you add a call to an assertion procedure in your code, the compiler executes the Boolean expression first. If the compiler executes this statement and the result is true, nothing additional is done but, when you receive a false, the compiler will end the execution and throw an error message. You can use this procedure to detect and report the error condition.

Tracebacks

Most compilers come with generic debugger codes that allow you to perform traceback operations. The compiler uses these, especially when there are runtime errors in your code. a traceback method of debugging gives you the list of active subroutines in the code. Tracebacks also give you the line numbers where you have active subroutines. If the error code identifies the sections where you have runtime errors, the traceback gives you the line numbers where the error has occurred. It is, however, up to you to identify which line in the subprogram caused the error.

Debugger Keywords

Most compilers and computers have different debugging programs or sections in the code. For instance, you can use the debugging keywords dbx and sdb in a Unix operating system. a debugging program gives you a way out – there is no need to develop a block of code for the sole purpose of debugging.

A debugging program runs through every line of code and identifies the errors in each line of the code. When the compiler executes a line with an error or break within it, the debugging code will create a break in the process, which allows the user to examine or modify the program data.

You can also use a debugging program or keyword to perform traceback operations in case you encounter any run-time errors. It is difficult to learn how a debugging code or keyword works. If this is the only tool you are using for debugging, then you may not save a lot of time. For instance, if you have a good debugger but a terrible test driver, the results of your debugging will not lead to the right results.

Techniques for Debugging

Incremental Testing

It is important to break the code into multiple subroutines if you are designing a complex module or function. It is important to note that the subroutines in the code should not be longer than 10 statements. If a module is designed this way, it is best to use incremental testing to work on your code.

When you perform incremental testing, you can classify subroutines as different levels. For instance, if you have a subroutine at a lower level, it means lower level subroutines do not call higher subroutines. Let us assume you have two subroutines – a and B. If a calls B, then B is a lower subroutine.

This form of testing aims to look at each subroutine as an individual block of code and test it. Begin with the lower subroutines. To do this, develop a test script that calls a lower subroutine directly. Otherwise, you need to include numerous test cases in your code, and each of these sections should include lower level subroutines.

To develop these test cases, you need to have a good understanding of the algorithms used in the module. The objective of this form of testing is to identify the errors in different sections of the code. This makes the process effective to debug the code and identify different sections in the code with errors.

Through incremental testing, you learn to work only with one error in the code. Most debugging and testing techniques look at multiple errors at once.

Sanity Checks

You can write a low-level code within a data structure under the assumption that the compiler can implement the code with the help of the high-level code. Most programmers write low-level code under the assumption that certain parameters or variables should not be null. The condition or assumption may be justified by the statements written in the method or program, but it is best to include a test script or block to see if the statements in the algorithm are implemented correctly. This is a sanity check, and the advantage of including these blocks of code is that the compiler can easily detect errors in the code.

Using Boolean Constants to Turn Off Debugging

If you want to include debugging code in your functions and methods, it is best to enclose those statements within a conditional statement. You can control the functioning of the conditional statement using a Boolean constant. This process allows you to turn the debugging code off easily, but the compiler can access it later if it needs to. It is best to use a different constant at every stage of the testing so that the compiler does not consider useless information.

Error Variables to Control the Program's Behavior

When you add debugging a print statement to your code, there is a possibility that the program may contain too much information which the compiler cannot use. The trouble with including such print statements is that the compiler will execute the statement regardless of whether there is an error or not. If the print statement is executed multiple times by the compiler, it indicates that it does not identify the error.

If the print statements display too much information of an existing data structure in the code, it indicates that the issue is magnified. If you do include a sanity check in your code to detect any errors, you should include a Boolean variable in the same module. Initialize the Boolean variable to false, which indicates there is no error in the code. You can create most data structures during the data initialization period, and you can initialize the error variable at the same time. Doing this will ensure that the compiler sets the error to true but does not exit the sanity check block of code. You can then enclose the debug code in a conditional statement, so the information in the block of code is printed only when the compiler identifies an error. You can do this using the next method.

Traceback Methods

If you want to use traceback to identify the error in your code, it is best to use Boolean sets to identify the error. It is easier to do this by performing a sanity check. Experts recommend that you add some error code in the functions or methods. This error code needs to be controlled by the variable causing the error. It is best to run the code and perform a sanity check before you add any error code to the program. You should add the debug code to control the errors in the program when the method or function calls on the sanity check.

How to Correct the Errors In Your Code

You need to keep the following objective in mind when you test code and identify any errors in your code – identify the cause and fix it. Do not worry about the symptoms.

Let us assume you have run the code, and you identify an issue with the segment. When you check the code carefully, you notice that you were passing a null pointer into a function or module. You, unfortunately, have not entered a statement to check if the pointer carries a null value. Does this mean you should check every method or function in the code for a null pointer? To do this, you may need to use an if statement and enclose the entire method or function in that if statement. You cannot determine this without understanding the design of the program and the algorithm you have used. The algorithm may have been implemented correctly in your code, and you can determine if the pointer is null using the algorithm. If this happens in your code, it means the if statement does not identify or solve the issue's cause.

When you add the if statement to the code, you only cover the indicators; you may hide the issue in one section of the code, but the issue will be elsewhere. It is important to note that you should include new code to your algorithm only if you are certain the statements should be added to the algorithm. Although the algorithm does not require it, if you still want to add a check, ensure it returns error codes and conditions. In simple words, you need to perform a sanity check.

Conclusion

If you have the basics of C++ down pat and want to learn more about programming in C++, you have come to the right place. This book has all the information you need about the language and how you can use object-oriented programming in C++. You will gather information about different concepts in object-oriented programming, such as abstraction, encapsulation, etc.

The book also introduces the concepts of searching and sorting algorithms and gives you some examples of how you can implement these in C++. Since it is important to optimize the code, the book also leaves you with some tips to help you do the same. The book also provides some information on how you can test the code you have written and debug the errors. There are numerous programs and examples given in this book to help you understand how to implement various concepts of C++. Best of luck to you. I hope you enjoy your journey.

R eferences

10 Tips for C and C++ Performance Improvement Code Optimization. (n.d.). www.thegeekstuff.com website: https://www.thegeekstuff.com/2015/01/c-cpp-code-optimization/

C++ Tutorial - Tutorialspoint. (2019). Tutorialspoint.com website: https://www.tutorialspoint.com/cplusplus/index.htm

C++ Programming Language - GeeksforGeeks. (2012). GeeksforGeeks website: https://www.geeksforgeeks.org/c-plus-plus/

Debugging and Testing. (2020). Umn.edu website: https://www.d.umn.edu/~gshute/softeng/testing.html