

Introduction to Programming with C++

Student Lecture Workbook

Andi Toce



Last update: Monday 27th May, 2024

Contents

I	Part One - MAC 101	
1	Introduction	11
1.1	What is Computer Science?	11
1.2	What is Programming?	12
1.3	Programming Language Hierarchy	13
1.4	Getting Started with C++	14
1.5	Practice Questions	17
2	Numbers at Different Bases	21
2.1	Storing Information in the Computer	21
2.2	Converting Binary, Octal and Hexadecimal to Decimal	22
2.3	Converting Decimal Numbers to Other Bases	24
2.4	Conversion Between Binary, Octal and Hexadecimal	25
2.5	Addition and Subtraction of Numbers at Different Bases	26
2.6	Practice Questions	28
3	Data Types and Operators	31
3.1	C++ Syntax Tokens	31
3.2	Primitive Data Types	33
3.3	Arithmetic Operations	35
3.4	Practice Questions	38

4	Selection Statements	47
4.1	Relational and Comparison Operators	47
4.2	<i>if</i> and <i>if-else</i> Statements	47
4.3	The Nested <i>if-else</i> Statements	51
4.4	The <i>switch</i> Statement	52
4.5	Logical Operators	54
4.6	Practice Questions	57
5	Repetitions Statements. (Loops)	67
5.1	The <i>while</i> loop.	68
5.2	The <i>do - while</i> loop.	69
5.3	The <i>for</i> loop	70
5.4	The <i>break</i> and <i>continue</i> Statements	73
5.5	Practice Questions	74
6	Functions and Parameters	85
6.1	What are Functions?	85
6.2	Passing a Parameter by Value or by Reference	91
6.3	Default Function Arguments	92
6.4	Function Overloading	93
6.5	Practice Questions	94
7	Variable Scope	103
7.1	Local and Global Variables	103
7.2	Default and Static Variables	105
7.3	Pre-defined Functions	106
7.4	Practice Questions	108
8	Arrays	113
8.1	What are Arrays	113
8.2	Initializing Arrays	114
8.3	Array Size	117
8.4	Passing Arrays to Functions	118
8.5	Characters, Strings and Arrays of Strings	120
8.6	Multi-Dimensional Arrays	121
8.7	Practice Questions	123
9	Pointers	135
9.1	What are Pointers?	135
9.2	Arrays and Pointer Arithmetic	136
9.3	Practice Questions	139

10	Structures	143
10.1	What are Structures?	143
10.2	Structure Composition	145
10.3	Arrays of Structures	146
10.4	Passing Structures to Functions	147
10.5	Practice Questions	148
11	Introduction to Classes and Objects	151
11.1	Objects	151
11.2	Classes	152
11.3	The First Example	152
11.4	Constructors	154
11.5	Destructors	155
11.6	Separating Content in Different Files	156
11.7	Constant Objects	158
11.8	Member Initializers	159
11.9	Class Composition	160
11.10	Practice Questions	161
12	Strings	165
12.1	How C++ Stores Text?	165
12.2	C-style String Functions	167
12.3	The <string> Class	168
12.4	Practice Questions	173

II

Part Two - MAC 125

13	Classes, Pointers, and Arrays	177
13.1	Pointers Revisited	177
13.2	Pointers and Some Special Operators	178
13.3	Arrays and Pointers	179
13.4	The this Keyword	181
13.5	Operator Overloading	183
13.6	Practice Questions	186
14	Recursion	193
14.1	Recursive Functions	193
14.2	Practice Questions	201
15	Inheritance	207
15.1	Introduction	207

15.2	Constructors and Destructors in Inheritance	209
15.3	Inheriting Functionality and Method Overriding	211
15.4	Access Control in Inheritance	213
15.5	Multiple Inheritance	215
15.6	Multilevel Inheritance	217
15.7	Practice Questions	219
16	Polymorphism	225
16.1	Introduction	225
16.2	Polymorphism and Virtual Functions	226
16.3	Pure Virtual Functions and Abstract Classes	230
16.4	Practice Questions	233
17	Templates	239
17.1	Background	239
17.2	Function Templates	241
17.3	Template Specialization and Function Overloading	243
17.4	Class Templates	244
17.5	Practice Questions	245
18	Linked Data Structures	253
18.1	Linked Lists	253
18.2	Linked Lists using Classes	256
18.3	Adding and Removing Nodes	258
18.4	LinkedList class	261
18.5	Practice Questions	264
19	Exception Handling	269
19.1	Introduction	269
19.2	Deep Dive into Exception Handling	271
19.3	Exploring Different Types of Exceptions	273
19.4	Handling Multiple Exceptions	275
19.5	Throwing an Exception in a Function	276
19.6	One More Example	277
19.7	Practice Questions	279
20	Standard Template Library	281
20.1	Iterators	281
20.2	Vectors	282
20.3	Queue Iterator Example	283

21	File Processing	291
21.1	Stream Objects	291
22	Basic Sorting Algorithms	295
22.1	Sorting Algorithms	295
A	Answers to Exercises	299
A.1	Chapter 1	299
A.2	Chapter 3	300
A.3	Chapter 3	303
	Index	305



Part One - MAC 101

1	Introduction	11
2	Numbers at Different Bases	21
3	Data Types and Operators	31
4	Selection Statements	47
5	Repetitions Statements. (Loops)	67
6	Functions and Parameters	85
7	Variable Scope	103
8	Arrays	113
9	Pointers	135
10	Structures	143
11	Introduction to Classes and Objects .	151
12	Strings	165

1. Introduction

1.1 What is Computer Science?

Different sources give different definitions of Computer Science. Below are examples taken from two university websites.

George Washington University: <http://www.seas.gwu.edu/~simhaweb/misc/cscareers.html>

What exactly is Computer Science?

Computer Science is the science of using computers to solve problems. Mostly, this involves designing software (computer programs) and addressing fundamental scientific questions about the nature of computation but also involves many aspects of hardware and architecting the large computer systems that form the infrastructure of commercial and government enterprises. Computer scientists work in many different ways: pen-and-paper theoretical work on the foundations and fundamentals, programming work at the computer and collaborative teamwork in doing research and solving problems.

What Computer Science is not ...

Computer Science is not about using software, such as spreadsheets (like Excel), word processors (like Word) or image tools (like Photoshop). Many software packages are complicated to master (such as Photoshop or Excel) and it is true that many jobs depend on expertise in using such tools, but computer science is not about using the tools. It is not about expertise in computer games, it is not about writing content in websites, and it is not about not about assembling computers or knowing which computers are best buys. Edsger Dijkstra, a famous award-winning computer scientist once said, "Computer Science is no more about computers than Astronomy is about telescopes". Computer Science is about the principles behind building the above software packages, about the algorithms used in computer games, about the technology behind the Internet and about the architecture of computing devices.

Michigan Technological University: <http://www.cs.mtu.edu/~john/whatiscs.html>

Computer science is a discipline that spans theory and practice. It requires thinking both in abstract terms and in concrete terms. The practical side of computing can be seen everywhere. Nowadays, practically everyone is a computer user, and many people are even computer programmers. Getting computers to do what you want them to do requires intensive hands-on experience. But computer science can be seen on a higher level, as a science of problem solving. Computer scientists must be adept at modeling and analyzing problems. They must also be able to design solutions and verify that they are correct. Problem solving requires precision, creativity, and careful reasoning.

1.2 What is Programming?



What is programming?



What is a programming language?

1.3 Programming Language Hierarchy

Programming languages can be categorized into hierarchical levels based on their abstraction and proximity to machine code. Here is a description of the four commonly recognized levels of programming languages:

1. **Machine Language:** At the lowest level of programming, we find machine language. This level consists of binary code that directly corresponds to the machine instructions executed by a computer's hardware. It is specific to the processor architecture and is incredibly challenging for humans to read and write directly.
2. **Assembly Language:** Assembly language is the next level above machine language. It uses mnemonic codes and symbols to represent the machine instructions and memory addresses. Assembly language is specific to a particular processor architecture, making it more human-readable than machine language. However, it still requires a deep understanding of the underlying hardware.
3. **High-Level Languages:** High-level languages are designed to be more user-friendly and abstract away the complexities of the hardware. These languages provide a higher level of abstraction, allowing programmers to focus on problem-solving rather than low-level details. High-level languages, such as Python, Java, C++, and Ruby, use English-like syntax and offer a wide range of built-in functions and libraries to facilitate common programming tasks.
4. **Domain-Specific High-Level Languages:** Domain-Specific High-Level Languages (DSHLLs) represent the highest level of programming abstraction. They provide an even greater level of abstraction compared to general-purpose high-level languages. DSHLLs are specifically designed for particular application domains, such as data analysis, artificial intelligence, or web development. Examples of DSHLLs include MATLAB, R, SQL, and HTML.

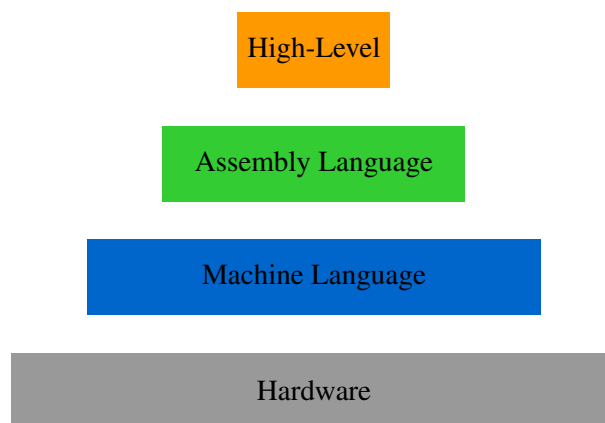


Figure 1.1: Pyramid of Programming Languages

1.4 Getting Started with C++

Why C++ ?

- C++ is a widely-used and highly versatile programming language.
- It has a rich set of libraries and frameworks, making it suitable for various domains such as game development, system programming, and scientific computing.
- C++ provides low-level control over hardware resources, allowing for efficient code.
- Many large-scale software projects, including operating systems and high-performance applications, are written in C++.
- Learning C++ enhances understanding of fundamental programming concepts like memory management, pointers, and object-oriented programming (OOP).
- C++ supports OOP principles, enabling modular and reusable code design.
- C++ code is portable, meaning it can be compiled and run on different platforms, making it suitable for cross-platform development.
- C++ is a good choice for developing performance-critical applications where speed and efficiency are crucial.

C++ IDEs (Integrated Development Environment):

- Eclipse C++
- CodeBlocks
- Microsoft Visual C++
- CLion
- xCode (Apple devices)

Online Compilers (This should only be temporary. You need to use an IDE):

- C++ Shell. (cpp.sh). Only C++.
- JDoodle. (www.jdoodle.com). Various languages.

Installing C++ in your computer:

1. Choose a C++ editor: While I recommend using Eclipse, you can choose any C++ editor that you feel comfortable with.
2. Install your chosen editor: For Eclipse, you can find installation guides on the internet. Search for "Install Eclipse for C++" on YouTube, and choose a video that suits the specifications of your computer. Follow the directions to complete the installation process. **BE PATIENT** and follow **ALL** directions carefully. Any small mistake can cause the IDE to fail.



Link (Cave of Programming) Intro discussion on pros and cons of C++, compilers and IDEs.

Running the first program:

- Open Eclipse (or other C++ editors like CodeBlocks, Microsoft Visual C++)
- Create a new C++ project
- Create a new “HelloWorld.cpp” file
- Compile and run the file

Code 1.1: First C++ program

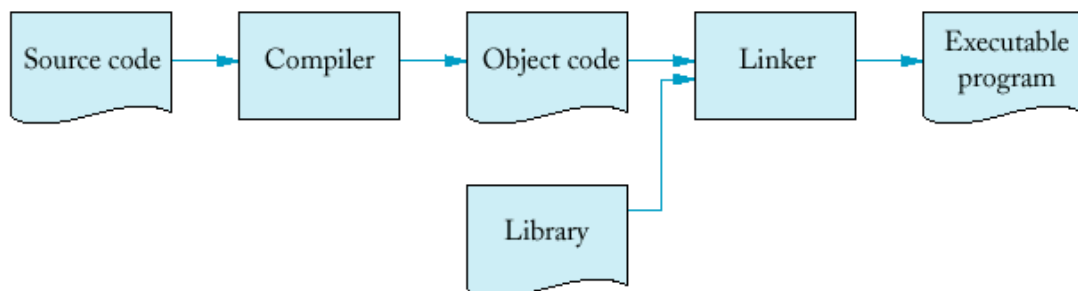
```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     cout << "Hello World";
6     return 0;
7 }
```



[Link \(Cave of Programming\) Hello World program.](#)



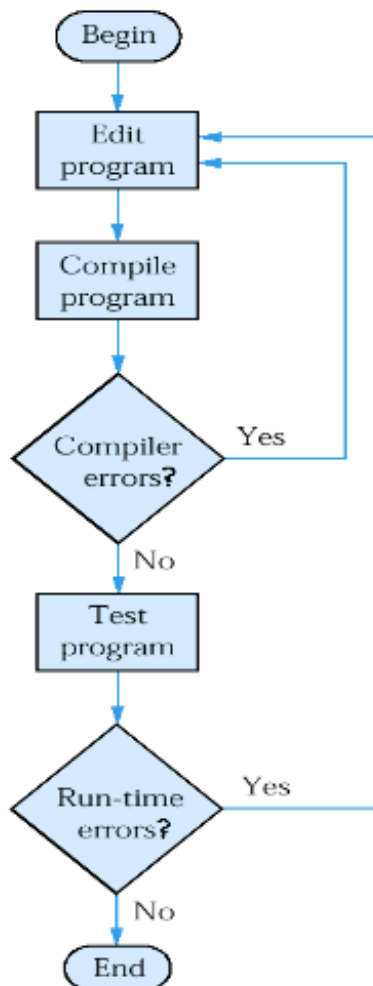
[Link \(Cave of Programming\) Outputting text.](#)

What happens when we build a C++ program?

Code 1.2: A simple input-output C++ program

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 int main () {
6     string mystr;
7     cout << "What's your name? ";
8     getline (cin, mystr);
9     cout << "Hello " << mystr << ".\n";
10    return 0;
11 }
```

Testing, revising and optimizing a program.



1.5 Practice Questions

1. What is the meaning of the following in the context of Computer Science?

(a) programming language:

(b) compiler:

(c) source code:

(d) algorithm:

(e) syntax and semantics:

(f) syntax error and logic error:

(g) debugging:

Write a program that prints two lines of text with an empty line in between as shown below.

2. **Desired output:**
- ```
This is the first line.

This is the second line.
```

Click [here](#) for answer. (Try to code it first yourself!)

Write a program that prints two lines of text with an empty line in between as shown below.

3. **Desired output:**
- ```
one  
two  
three
```

Click [here](#) for answer. (Try to code it first yourself!)

Write a program that produces the output shown below.

4. **Desired output:**
- ```
*
**


```

Click [here](#) for answer. (Try to code it first yourself!)

Write a program that produces the output shown below.

5. **Desired output:**
- ```
*  
***  
*****  
*****
```

Write a program that allows the user to input his/her age and prints the age on the screen.

6. **Sample output:**
- ```
Enter your age: 25
You are 25 years old
```

Click [here](#) for answer. (Try to code it first yourself!)

Write a C++ program that allows the user to input his/her name and prints a greeting on the screen.

7. **Sample output:**  
Enter your name: John  
Hello John! Have a wonderful day!

8. Write a C++ program that prints the following shape on the screen:

```
 /_/\
(o o)
> ^ <
/ ^ \
| / \ |
|/ \|
```

Make sure to use the appropriate 'cout' statements to generate the shape.



## 2. Numbers at Different Bases

### 2.1 Storing Information in the Computer

In computing, the smallest unit of information storage is the bit. A bit can take on only two values, 0 or 1, and all information in a computer is stored as a sequence of bits. To make it easier to talk about larger amounts of information, we use multiples of bits. The most common multiples are:

- Byte: a unit of 8 bits. It is the basic building block of most computer systems.
- Kilobyte (KB): a unit of a thousand bytes, or  $10^3$  bytes.
- Megabyte (MB): a unit of a million bytes, or  $10^6$  bytes.
- Gigabyte (GB): a unit of a billion bytes, or  $10^9$  bytes.
- Terabyte (TB): a unit of a trillion bytes, or  $10^{12}$  bytes.

Understanding bit multiples is important for understanding the capacity and performance of computer systems.

## 2.2 Converting Binary, Octal and Hexadecimal to Decimal

The base of a number system corresponds to the digits used by the system.

| Base | Digits Used                     | Name        | Example    | Decimal Equivalent |
|------|---------------------------------|-------------|------------|--------------------|
| 2    | 0,1                             | Binary      | $101_2$    | 5                  |
| 8    | 0,1,2,3,4,5,6,7                 | Octal       | $46_8$     | 38                 |
| 10   | 0,1,2,3,4,5,6,7,8,9             | Decimal     | $109_{10}$ | 109                |
| 16   | 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F | Hexadecimal | $2A3_{16}$ | 675                |



<— Watch this video to better understand why we use the binary system.



<— .... and another video.

In mathematics, we use different number bases or number systems to represent numbers. The most commonly used number system is the decimal or base-10 system, which uses ten digits (0 to 9) to represent numbers. However, there are other number systems as well, such as the binary (base-2), octal (base-8), and hexadecimal (base-16) systems. Understanding how different number bases work is important in computer science and engineering.

Consider the decimal number 2365 .

| Thousands | Hundreds | Tens   | Ones   |                   |
|-----------|----------|--------|--------|-------------------|
| 2         | 3        | 6      | 5      |                   |
| $10^3$    | $10^2$   | $10^1$ | $10^0$ |                   |
| 2000      | 300      | 60     | 5      | <b>Sum = 2365</b> |

Consider the binary number  $1001101_2$ . Find the equivalent decimal number.

|       |       |       |       |       |       |       |                 |
|-------|-------|-------|-------|-------|-------|-------|-----------------|
| 1     | 0     | 0     | 1     | 1     | 0     | 1     |                 |
| $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |                 |
| 64    | 0     | 0     | 8     | 4     | 0     | 1     | <b>Sum = 77</b> |

$$\begin{aligned}
 1001101_2 &= 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\
 &= 64 + 0 + 0 + 8 + 4 + 0 + 1 \\
 &= \boxed{77}_{10}
 \end{aligned}$$

Therefore, the equivalent decimal number of the binary number  $1001101_2$  is  $77_{10}$ .

Consider the octal number  $427_8$ . Find the equivalent decimal number.

|       |       |       |                  |
|-------|-------|-------|------------------|
| 4     | 2     | 7     |                  |
| $8^2$ | $8^1$ | $8^0$ |                  |
| 256   | 16    | 7     | <b>Sum = 279</b> |

$$\begin{aligned}427_8 &= 7 \times 8^0 + 2 \times 8^1 + 4 \times 8^2 \\&= 7 \times 1 + 2 \times 8 + 4 \times 64 \\&= 7 + 16 + 256 \\&= \boxed{279}_{10}\end{aligned}$$

Therefore, the equivalent decimal number of the octal number  $427_8$  is  $279_{10}$ .



Convert  $111001_2$  to a decimal number.



Convert  $3771_8$  to a decimal number.



Convert  $2B5_{16}$  to a decimal number.

### 2.3 Converting Decimal Numbers to Other Bases

To convert a decimal number to a different base, we use the method of repeated division by the base, while keeping track of the remainders. The remainders, read in reverse order, give us the corresponding digits in the new base. This process can be better understood through the following examples.

Example: convert  $43_{10}$  to binary.

| Divide by 2 | Quotient | Remainder |
|-------------|----------|-----------|
| $43 / 2$    | 21       | 1         |
| $21 / 2$    | 10       | 1         |
| $10 / 2$    | 5        | 0         |
| $5 / 2$     | 2        | 1         |
| $2 / 2$     | 1        | 0         |
| $1 / 2$     | 0        | 1         |

Take the binary digits in reverse order:  $43_{10} = 101011_2$



Convert  $94_{10}$  to binary.

Convert  $342_{10}$  to the octal equivalent.

| Divide by 8 | Quotient | Remainder |
|-------------|----------|-----------|
| $342/8$     | 42       | 6         |
| $42/8$      | 5        | 2         |
| $5/8$       | 0        | 5         |

$342_{10} = 526_8$



Convert  $189_{10}$  to octal.



Convert  $542_{10}$  to hexadecimal.



## 2.4 Conversion Between Binary, Octal and Hexadecimal

Converting between binary, octal, and hexadecimal is easier than converting to and from decimal. Can you tell why?

Example: Convert  $1011010_2$  to octal and hexadecimal.

| Octal |     |     |
|-------|-----|-----|
| 1     | 011 | 010 |
| 1     | 3   | 2   |

$$\begin{aligned}\text{Binary } 1011010_2 &= \text{Grouped } 001, 011, 010 \\ &= \text{Octal } 132_8\end{aligned}$$

Therefore, the binary number  $1011010_2$  is equivalent to the octal number  $132_8$ .

| Hexadecimal |      |
|-------------|------|
| 101         | 1010 |
| 5           | A    |

$$1011010_2 = 132_8 = 5A_{16}$$



Convert  $3617_8$  to binary and hexadecimal.



Convert  $1110001_2$  to octal and hexadecimal.

## 2.5 Addition and Subtraction of Numbers at Different Bases

In this section, we will explore the addition and subtraction of numbers at different bases, focusing on the techniques and strategies used to perform these operations efficiently and accurately.

### Addition

First, a closer look at the addition of decimal values:



$$3524_{10} + 692_{10}$$

Let's add the binary numbers 11011 and 11010:

$$\begin{array}{r} 1\ 1\ 0\ 1\ 1 \\ 1\ 1\ 0\ 1\ 0 \\ \hline 1\ 1\ 0\ 1\ 0\ 1 \end{array}$$



$$\text{Now try: } 1011101_2 + 10110_2$$



$$\text{Add } 1100110_2 \text{ and } 1111010_2$$



$$35A_{16} + E7_{16}$$



$$3527_8 + 726_8$$

**Subtraction**

First, a closer look at the subtraction of decimal values:



$$1932_{10} - 281_{10}$$



Now we subtract  $11001_2$  from  $1110011_2$



$$1101001_2 - 101110_2$$



$$35A_{16} - E7_{16}$$



$$3527_8 - 726_8$$

**2.6 Practice Questions**

1. Consider the binary number  $11011001_2$ . Find the equivalent decimal number.
  
2. Consider the octal number  $274_8$ . Find the equivalent decimal number.
  
3. Consider the hexadecimal number  $AC7_{16}$ . Find the equivalent decimal number.
  
4. Convert  $283_{10}$  to the octal equivalent.
  
5. Convert  $198_{10}$  to the corresponding hexadecimal number.
  
6. Convert  $10110011_2$  to octal and hexadecimal.
  
7. Convert  $4571_8$  to binary and hexadecimal.
  
8. Convert  $10011011_2$  to octal and hexadecimal.
  
9. Add  $11100101_2$  and  $111010110_2$
  
10. Subtract  $110100101_2 - 1010110_2$

11. Find  $35A_{16} + E7_{16}$

12. Find  $35A_{16} - E7_{16}$

13. Find  $3527_8 + 726_8$

14. Find  $3527_8 - 726_8$

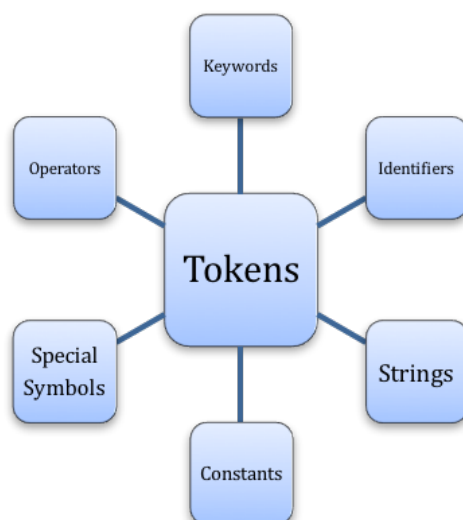


## 3. Data Types and Operators

### 3.1 C++ Syntax Tokens

In C++, tokens are the smallest individual units of a program that the compiler can understand. Each token has a specific meaning for the compiler, and C++ has a variety of different types of tokens, including keywords, identifiers, operators, literals, and punctuation.

Understanding the different types of C++ tokens is essential for writing correct and efficient code, as well as for debugging errors that may arise during the compilation process.



### Keywords in C++

Keywords are reserved words that have a specific meaning in a programming language, such as C++. They are used to define the syntax and structure of the language, and cannot be used for any other purpose. In C++, keywords are an essential component of the language, and it is important for developers to have a good understanding of their usage.

The following table lists some of the most commonly used keywords in C++:

Table 3.1: Commonly Used C++ Keywords

|          |          |           |          |          |
|----------|----------|-----------|----------|----------|
| auto     | bool     | break     | case     | catch    |
| char     | class    | const     | continue | default  |
| delete   | do       | double    | else     | enum     |
| explicit | extern   | false     | float    | for      |
| friend   | goto     | if        | inline   | int      |
| long     | mutable  | namespace | new      | nullptr  |
| operator | private  | protected | public   | register |
| return   | short    | signed    | sizeof   | static   |
| struct   | switch   | template  | this     | throw    |
| true     | try      | typedef   | typeid   | typename |
| union    | unsigned | using     | virtual  | void     |
| volatile | while    |           |          |          |

### Identifiers

Identifiers are typically used to name different components of a program such as variables and objects.

#### Rules for choosing identifiers:

- The first character of the name must be a letter or an underscore.
- You can only use letters, digits, or underscores in the rest of the name.
- You can't use spaces to separate words in a function name - instead, you can either use underscores to separate words or capitalize the first letter of words after the first word.
- You can't use C++ keywords as identifier names.



Which of the following is not a valid C++ identifier? Select all that apply.

- (a) firstInteger
- (b) first integer
- (c) first\_integer
- (d) 1st\_integer
- (e) double
- (f) 3ge4k2
- (g) \$dollar
- (h) My#one



## 3.2 Primitive Data Types

When you write a computer program or algorithm, you need to use and manipulate data. This data can take different forms, such as numbers or text, and needs to be stored in a specific location in the computer's memory. This is where variables come in - they provide a way to store and access information throughout the program.

To use variables effectively, you need to understand the different data types that can be stored. Here are some of the most commonly used data types, along with their descriptions:

| Type           | Keyword | Size             | Example             |
|----------------|---------|------------------|---------------------|
| Integer        | int     | at least 32 bits | int i=23;           |
| Floating point | float   | at least 32 bits | float f = 23.7;     |
|                | double  | at least 64 bits | double d = 243.2;   |
| Character      | char    | at least 8 bits  | char c = 'g';       |
| String         | string  | varies           | string s = "hello"; |

Code 3.1: Example of different data types and their maximum capacity.

```

1 #include <iostream>
2 using namespace std;
3
4 int main() {
5 cout << "Size of char : " << sizeof(char) << endl;
6 char c = 'g';
7 cout << "Example character type: " << c << endl;
8
9 cout << "Size of int : " << sizeof(int) << endl;
10 int i = 12;
11 cout << "Example integer type: " << i << endl;
12
13 cout << "Size of float : " << sizeof(float) << endl;
14 float f = 22.3;
15 cout << "Example floating point type: " << f << endl;
16
17 cout << "Size of double : " << sizeof(double) << endl;
18 double d = 125.345;
19 cout << "Example double precision type: " << d << endl;
20
21 return 0;
22 }
```



← Click here for an extended version of this program.

Code 3.2: An input-output program with integers.

```
1 #include <iostream>
2 using namespace std;
3
4 int main () {
5 int i;
6 cout << "Please enter an integer value: ";
7 cin >> i;
8 cout << "The value you entered is " << i;
9 return 0;
10 }
```



Enter different numeric and non-numeric values of different sizes as input. What do you notice?



What is the largest unsigned integer that can be represented with 32 bits?



What is the largest unsigned integer that can be represented with 64 bits?



<— Video Link (Cave of Programming) Variables.



<— Video Link (Cave of Programming) User Input.



<— Video Link (Cave of Programming) Integer Variables.



<— Video Link (Cave of Programming) Floating Point Variables.

### 3.3 Arithmetic Operations

Basic C++ arithmetic binary operators:

| Operation      | C++ Operator | Example     |
|----------------|--------------|-------------|
| Addition       | +            | H + 3       |
| Subtraction    | -            | A - B       |
| Multiplication | *            | 3 * D       |
| Division       | /            | 7 / 5       |
| Modulus        | %            | 10 % 3      |
| Parenthesis    | ( )          | (a + 2) - 5 |

Code 3.3: A program that adds two numbers:

```

1 #include <iostream>
2 using namespace std;
3
4 int main() {
5 int n1, n2, sum;
6
7 cout << "Enter first integer: ";
8 cin >> n1;
9
10 cout << "Enter second integer: ";
11 cin >> n2;
12
13 sum = n1 + n2;
14 cout << "Sum is " << sum << endl;
15
16 return 0;
17 }
```

(From book) Write a program that reads a four-digit integer, such as 1998, and then displays it, one digit per line. Your prompt should tell the user to enter a four-digit integer. You can then assume that the user follows directions. Hint: Use the division and remainder operators.



**Desired output:**

Enter a four-digit number: 1998

1  
9  
9  
8

Code 3.4: A program that finds the average of two numbers

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5
6 double d1;
7 cout << "Enter the first number: ";
8 cin >> d1;
9
10 double d2;
11 cout << "Enter the second number: ";
12 cin >> d2;
13
14 cout << "The average value is " << (d1+d2)/2;
15
16 return 0;
17 }
```

Code 3.5: A program using characters

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5
6 char myChar = 'A';
7 cout << "The initial character value is: " << myChar << endl;
8
9 myChar = myChar+1;
10 cout << "The character value is now: " << myChar;
11
12 return 0;
13 }
```

Basic C++ compound operators:

| Expresion             | Equivalent to           |
|-----------------------|-------------------------|
| <code>x+=5;</code>    | <code>x=x+5;</code>     |
| <code>x-=y;</code>    | <code>x=x-y;</code>     |
| <code>x*= y+2;</code> | <code>x=x*(y+2);</code> |
| <code>x/=2;</code>    | <code>x=x/2;</code>     |

Increment and decrement operators:

| Expresion         | Similar to          |
|-------------------|---------------------|
| <code>++x;</code> | <code>x=x+1;</code> |
| <code>x++</code>  | <code>x=x+1;</code> |
| <code>--x;</code> | <code>x=x-1;</code> |
| <code>x--</code>  | <code>x=x-1;</code> |

Here's an example program that demonstrates the use of prefix and suffix increment operators:

Code 3.6: Unary increment example

```

1 #include <iostream>
2 using namespace std;
3
4 int main() {
5 int x, y;
6
7 cout << "Unary Prefix Example: " << endl;
8 x = 5;
9 y = ++x;
10 cout << "x is : " << x << endl;
11 cout << "y is : " << y << endl << endl;
12
13 cout << "Unary Suffix Example: " << endl;
14 x = 5;
15 y = x++;
16 cout << "x is : " << x << endl;
17 cout << "y is : " << y << endl;
18
19 return 0;
20 }
```



← Get code and some explanations here.

### 3.4 Practice Questions

1. What is the meaning of the following in the context of Computer Science?

(a) variable:

---

---

(b) identifier:

---

---

(c) keyword:

---

---

(d) statement:

---

---

(e) declaring a variable:

---

---

(f) initializing a variable:

---

---

(g) assignment statement:

---

---

Click [here](#) for answer.

2. True-False questions – circle your choice.
- (a) All variables must be declared before being used. ( True - False )
  - (b) C++ considers the variable number and NUMBER to be identical. ( True - False )
  - (c) The arithmetic operators \*, /, +, and – have the same level of precedence. ( True - False )
  - (d) The statement `cout << "a = 5;"` ; is a typical assignment statement. ( True - False )
  - (e) The statements `n +=2;` and `n = n+2;` have the exact same outcome. ( True - False )
  - (f) The size of the int data type is guaranteed to be 4 bytes on all platforms. ( True - False )
  - (g) The double data type can hold larger values than the float data type. ( True - False )
  - (h) C++ has a built-in data type for representing Boolean values. (True - False)
  - (i) The "sizeof" operator in C++ returns the number of bytes occupied by a variable. ( True - False )

Click [here](#) for answer.

3. Assuming that x is equal to 6, which of the following statements will not result in y containing the value 7 after execution? Select ALL that apply.
- (a) `y = 7;`
  - (b) `y = ++x;`
  - (c) `y = x++;`
  - (d) `y = x + 1;`
  - (e) `y = x + 8 / 2;`
  - (f) `y = (x + 8) / 2;`
4. Which of the following is not a valid C++ identifier? Choose all that apply.
- (a) myNumber
  - (b) my number
  - (c) my\_number
  - (d) 1st\_number

- (e) float
- (f) w3e5r6t7
- (g) &number
- (h) my+number

5. A variable that can have values only in the range 0 to 65535 is a:

- (a) four – byte integer
- (b) four – byte unsigned integer
- (c) two – byte integer
- (d) two – byte unsigned integer

Click [here](#) for answer.

6. Given the algebraic expression  $2x^2 - 5x + 2$ , which of the following is (are) correct C++ statements for the equation. Circle all that apply.

- (a)  $y = 2 * x * x - 5 * x + 2$
- (b)  $y = 2 * x * x - 5 * (x + 2)$
- (c)  $y = (2 * x * x) - (5 * x) + 2$
- (d)  $y = 2 * x * (x - 5) * x + 2$

7. What is the largest **unsigned** integer that can be represented with 32 bits and 64 bits?

Click [here](#) for answer.

8. What is the largest **signed** integer that can be represented with 32 bits and 64 bits?

9. Which of the following is NOT a valid C++ data type? Circle your choice clearly.

- (a) int
- (b) char
- (c) double
- (d) number



10. Write four different C++ statements that each adds one to variable x. Embed the statements in a small c++ program for testing.

Click [here](#) for answer.

11. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5 int x = 13;
6 int y = 3;
7 cout<< "Summary of Results:" << endl;
8 cout<< x + y << endl;
9 cout<< x - y << endl;
10 cout<< x / y << endl;
11 cout<< x * y << endl;
12 cout<< x % y << endl;
13
14 return 0;
15 }
```

**Write exact output here:**

12. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 int main () {
5 int x = 7, y =10;
6 cout << x++ << " " << ++y << endl;
7 cout << x << " " << y << endl ;
8 cout <<x-- << " " << --y << endl ;
9 cout << x << " " << y << endl ;
10 cout << x / y << endl ;
11 cout << x % y << endl ;
12
13 return 0;
14 }
```

**Write exact output here:**

13. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5 cout << "Size of char : " << sizeof(char) << endl;
6 cout << "Size of int : " << sizeof(int) << endl;
7 cout << "Size of float : " << sizeof(float) << endl;
8 cout << "Size of double : " << sizeof(double) << endl;
9
10 return 0;
11 }
```

**Write exact output here:**

14. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6 int x, y;
7
8 cout << "Unary Prefix Example: " << endl;
9 x = 11;
10 y = --x;
11 cout << "The value of x is : " << x << endl;
12 cout << "The value of y is : " << y << endl << endl;
13
14 cout << "Unary Suffix Example: " << endl;
15 x = 11;
16 y = x--;
17 cout << "The value of x is : " << x << endl;
18 cout << "The value of y is : " << y << endl;
19
20 return 0;
21 }
```

**Write exact output here:**

15. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5
6 int x = 11;
7 double y = 6, z = 2;
8 y = x / z;
9 x = 3 * y;
10 cout << "y = " << y << endl;
11 cout << "x = " << x << endl;
12
13 return 0;
14 }
```

**Write exact output here:**

16. Write a program that allows the user to input a temperature value in Celsius and prints the corresponding temperature in Fahrenheit.

Use the following conversion formula: Fahrenheit = Celsius \* 1.8 + 32.

**Desired output:**

Enter the temperature in Celsius: **30**

The corresponding temperature in Fahrenheit is: 86

Click [here](#) for answer.

17. Write a program that allows the user to input a temperature value in Fahrenheit and prints the corresponding temperature in Celsius.

**Desired output:**

Enter the temperature in Fahrenheit: **86**

The corresponding temperature in Celsius is: 30

18. Write a program that allows the user to input two integers, one for the length of the rectangle and one for the width. The program then prints the value for the area of the rectangle.

**Desired output:**

Enter a number for the length: **5**  
Enter a number for the width: **4**  
The area of the rectangle is: 20

19. Write a program that allows the user to input a number for the side of a cube and prints the value for the volume of the cube.

**Desired output:**

Enter a number for the side of the cube: **2.0**  
The volume of the cube is: 8.0

20. Write a C++ program that allows the user to input an integer value  $n$  and prints the sum of the five consecutive integers starting with  $n$ . For example if the user enters 2 then the program prints the sum = 20 since  $2+3+4+5+6 = 20$ .

**Desired output:**

Enter the first integer value: **2**  
The sum is: 20

21. Write a C++ program that allows the user to input separately the number of years, months and days for a given time period and prints the total number of days for the same time period. For this exercise assume all months have 30 days.

**Desired output:**

Enter the number of years: **2**  
Enter the number of months: **5**  
Enter the number of days: **10**  
  
The time period you entered is equivalent to 880 days

22. Write a program that prompts the user to enter a five-digit integer and displays the sum of its digits. Your program should also verify that the input is a valid five-digit integer.

**Desired output:**

Enter a five-digit number: 12345  
The sum of the digits in 12345 is 15.

23. Write a C++ program that takes two integers as input and swaps their values. The program should print on the screen the values of each variable before and after swapping.

**Sample output:**

Enter two integers: 7 9  
Before swapping:  
x = 7  
y = 9  
After swapping:  
x = 9  
y = 7

24. What is the largest signed decimal integer that can be represented with 16 bits?
25. What is the largest unsigned decimal integer that can be represented with 16 bits?
26. What is the minimum number of bits needed to represent the decimal value 1000?

## 4. Selection Statements

In programming, we often encounter situations where we need to make decisions on whether to execute certain statements based on specific conditions. In C++, we can use selection statements to handle such scenarios.

### 4.1 Relational and Comparison Operators

Relational and comparison operators are used in C++ to compare two expressions and produce a result of either *true* or *false*. These operators are useful for making logical decisions in programs. Here are the commonly used relational and comparison operators:

| Operator | Comparison   | Operator | Comparison          |
|----------|--------------|----------|---------------------|
| ==       | Equal to     | >        | Greater than        |
| !=       | Not equal to | <=       | Less or equal to    |
| <        | Less than    | >=       | Greater or equal to |

### 4.2 *if* and *if-else* Statements

#### The *if* statement format

```
if (condition)
 statement: executed if condition is true;
```

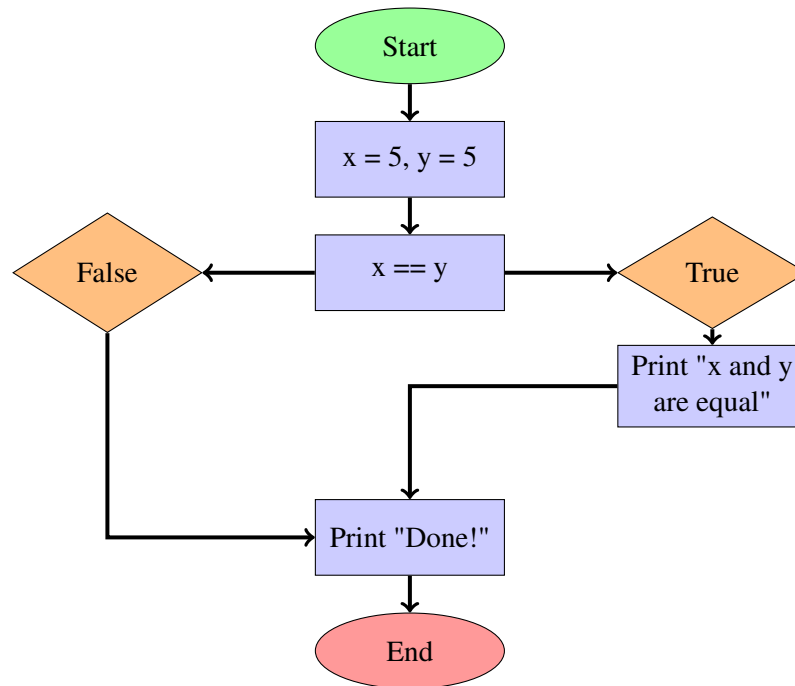


Figure 4.1: Flowchart of the program

Code 4.1: C++ if statement example

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5 int x = 5;
6 int y = 5;
7
8 if (x==y)
9 cout << "x and y are equal" << endl;
10
11 cout << "Done!";
12 return 0;
13 }
```



Change the value of  $x$  and run the program.



← Video Link (Cave of Programming) If Statements.



**The compound if statement format**

```
if (condition){
 statement1;
 statement2;
 statement3;
}
```

Code 4.2: C++ compound if statement example

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5 int x = 5;
6 int y = 5;
7
8 if (x == y){
9 cout << "x and y are equal \n";
10 cout << "x is " << x << " and y is " << y << endl;
11 }
12 cout << "Done!";
13 return 0;
14 }
```



Remove the curly brackets at the beginning and the end of the if statement and run the program.

Write a program that allows the user to input three integers (not necessarily distinct) and prints the largest of the three values.

**Desired output:**

Enter three integers: 5

2

1

The largest integer is: 5

**The compound if-else statement format**

```
if (condition)
 statement1: executed if condition is true;
else
 statement2: executed if condition is false;
```

Code 4.3: C++ if-else statement example

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5 int x = 5;
6 int y = 6;
7
8 if (x == y)
9 cout << "x and y are equal \n";
10 else
11 cout << "x and y are not equal \n";
12
13 cout << "Done!";
14 return 0;
15 }
```



← Video Link (Cave of Programming) If Else Statements.

Write a program that inputs an integer and prints if the number is even or odd.

**Desired output:**

Enter an integer: 7

The number you entered is odd.

### 4.3 The Nested *if-else* Statements

Code 4.4: C++ nested if-else statement example

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5 int score=0;
6 cout << "Enter Score: ";
7 cin >> score;
8
9 cout << "The Grade is: ";
10 if (score >= 90)
11 cout << "A";
12 else if (score >= 80)
13 cout << "B";
14 else if (score >= 70)
15 cout << "C";
16 else if (score >= 60)
17 cout << "D";
18 else
19 cout << "F";
20
21 return 0;
22 }
```



← Get code here.

Code 4.5: An alternative to if-else statement (using the ?: operator)

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5 int score=0; //score on exam
6 cout << "Enter Score: ";
7 cin >> score;
8
9 cout << (score >= 60 ? "Passed" : "Failed");
10 return 0;
11 }
```

## 4.4 The *switch* Statement

The switch statement is an alternative to if-else. It is commonly used if we have several condition cases. It is easier to write and read.

Example switch statement compared to the equivalent if-else statement.

Code 4.6: Switch statement example

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5 int day = 0;
6 cout << "Enter a number between 1 and 7: ";
7 cin >> day;
8
9 switch (day){
10 case 1 : cout << "Monday";
11 break;
12 case 2 : cout << "Tuesday";
13 break;
14 case 3 : cout << "Wednesday";
15 break;
16 case 4 : cout << "Thursday";
17 break;
18 case 5 : cout << "Friday";
19 break;
20 case 6 : cout << "Saturday";
21 break;
22 case 7 : cout << "Sunday";
23 break;
24 default : cout << "Not a valid entry";
25 break;
26 }
27 return 0;
28 }
```



← Get code here.



Remove the first 3 break statements and then enter 1 as an input. What do you notice?



Remove the default case. Does the program work? What can you conclude?



Write a program that does the same work as the switch example above but using a nested if statement instead.

Characters share some similar features with integers. Characters are saved in memory as numbers. We can use the same relational operators to compare characters. See example below.

Code 4.7: Example comparing characters

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5 char first = 'b';
6 char second = 'b';
7
8 if (first < second)
9 cout << "It looks like " << first
10 << " comes before " << second << endl;
11 else if (second < first)
12 cout << "It looks like " << second
13 << " comes before " << first << endl;
14 else
15 cout << first << " and " << second
16 << " are the same." << endl;
17
18 return 0;
19 }
```



Change the value of 'first' and 'second' and test the program.

## 4.5 Logical Operators

Logical operators in C++ are used to combine multiple conditions and perform logical operations. The three main logical operators are `&&` (logical AND), `||` (logical OR), and `!` (logical NOT).

The logical AND operator (`&&`) evaluates to *true* only if both of its operands are *true*. If any of the operands is *false*, the result is *false*. This operator is commonly used to check if multiple conditions are satisfied simultaneously.

The logical OR operator (`||`) evaluates to *true* if at least one of its operands is *true*. It returns *false* only if both operands are *false*. This operator is often used to check if at least one condition is true.

The logical NOT operator (`!`) reverses the logical state of its operand. If the operand is *true*, the result is *false*; if the operand is *false*, the result is *true*. It is typically used to negate a condition or the result of a logical expression.

These logical operators are essential for creating complex conditional statements and controlling the flow of execution in C++ programs. By combining conditions with logical operators, you can build powerful decision-making structures and control the behavior of your code based on various conditions and criteria.

| Operator | Logical Operation | Example                                |
|----------|-------------------|----------------------------------------|
| !        | NOT               | !(2==2) // evaluates to FALSE          |
| &&       | AND               | ( 2==2) && (2==3) //evaluates to FALSE |
|          | OR                | (2==2)    (2==3) //evaluates to TRUE   |

| A     | B     | A && B |
|-------|-------|--------|
| TRUE  | TRUE  | TRUE   |
| TRUE  | FALSE | FALSE  |
| FALSE | TRUE  | FALSE  |
| FALSE | FALSE | FALSE  |

| A     | B     | A    B |
|-------|-------|--------|
| TRUE  | TRUE  | TRUE   |
| TRUE  | FALSE | TRUE   |
| FALSE | TRUE  | TRUE   |
| FALSE | FALSE | FALSE  |

Code 4.8: || (OR) operator example.

```
1 #include <iostream>
2 using namespace std;
3
4 int main () {
5 char agree;
6 cout << "Would you like to meet? (y/n): ";
7 cin >> agree;
8
9 if (agree == 'y' || agree == 'Y')
10 cout << "Great !!!" << endl;
11 else if (agree == 'n' || agree == 'N')
12 cout << "Sorry to hear that !!!" << endl;
13 else
14 cout << "What did you say?" << endl;
15
16 return 0;
17 }
```

Code 4.9: &amp;&amp; (AND) operator example.

```
1 #include <iostream>
2 using namespace std;
3
4 int main () {
5 int number;
6 cout << "I only like numbers between 10 and 20. " << endl;
7 cout << "Enter an integer: ";
8 cin >> number;
9
10 if (number >= 10 && number <= 20)
11 cout << "You entered " << number << ". I am happy!";
12 else
13 cout << "I don't like the number you entered!";
14
15 return 0;
16 }
```



Determine whether the following expressions are true or false. Use a computer program to verify your answers.

- (a)  $(4 == 4) \ \&\& \ (4 \geq 3) =$
- (b)  $(4 == 4) \ || \ (10 < 5) =$
- (c)  $!(4 == 4) \ || \ (10 < 5) =$
- (d)  $!(4 == 4) \ || \ !(10 < 5) =$
- (e)  $(3 != 4) \ \&\& \ (10 < 15) \ || \ (4 == 5) =$
- (f)  $(3 != 4) \ \&\& \ (10 < 15) \ \&\& \ (4 == 5) =$

Here is a C++ program that can be used to verify the answer to each question.

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5 cout << boolalpha << ((4 == 4) && (4 >= 3)) << endl;
6 cout << boolalpha << ((4 == 4) || (10 < 5)) << endl;
7 cout << boolalpha << (!(4 == 4) || (10 < 5)) << endl;
8 cout << boolalpha << (!(4 == 4) || !(10 < 5)) << endl;
9 cout << boolalpha << ((3 != 4) && (10 < 15) || (4 == 5)) << endl;
10 cout << boolalpha << ((3 != 4) && (10 < 15) && (4 == 5)) << endl;
11
12 return 0;
13 }
```



## 4.6 Practice Questions

1. What is the meaning of the following in the context of Computer Science?

(a) optimizing a program / algorithm:

---

---

(b) C++ is case sensitive:

---

---

(c) unsigned and signed integer:

---

---

(d) A program is scalable:

---

---

2. True-False questions – Circle your choice:

- (a) (**True** - False) The AND (&&) operator stops after finding one condition to be false.
- (b) (True - **False**) The OR (||) operator stops after finding one condition to be false.
- (c) (True - **False**) The default portion is REQUIRED in a switch statement.
- (d) (**False** - True) A switch statement can use a string as the expression to be evaluated.
- (e) (**True** - False) The logical NOT operator (!) can only be used with boolean values.
- (f) (**True** - False) The switch statement can only be used with integer and character types.
- (g) (**True** - False) The AND (&&) operator has higher precedence than the OR (||) operator.
- (h) (**True** - False) The if statement is a selection statement.
- (i) (**True** - False) The if-else statement is a conditional statement.
- (j) (**True** - False) The switch statement is a selection statement.
- (k) (**True** - False) The break statement is used to exit a switch statement.
- (l) (**False** - True) The AND (&&) operator evaluates both conditions even if the first condition is false.
- (m) (**False** - True) The OR (||) operator evaluates both conditions even if the first condition is true.
- (n) (**True** - False) The ?: operator can be used in place of an if-else statement.
- (o) (**False** - True) The AND (&&) operator can be used with integer values.
- (p) (**False** - True) The OR (||) operator can be used with integer values.
- (q) (**True** - False) The if-else statement can be nested within another if-else statement.

3. Determine whether the following expressions are true or false. Assume  $x=3$  and  $y=5$ . Use a computer program to verify your answers.

- (a)  $(x \leq 5) \ \&\& \ (y == 5) =$
- (b)  $(x == y) \ || \ (y < 10) =$
- (c)  $!(x == 3) \ || \ (y != 5) =$
- (d)  $((x \% 3) == 1) \ || \ (y == 3) =$
- (e)  $(3 != 4) \ \&\& \ (x < 2) \ || \ (y == 5)$

4. What the following statements print? Assume  $i=1$ ,  $j=2$ ,  $k=3$  and  $m=2$ .

- (a) `cout << (j == 1) << endl;`
- (b) `cout << (j >= 1 && j < 4) << endl;`
- (c) `cout << (j >= i || k == m) << endl;`
- (d) `cout << (k + m < j || 3-j >= k) << endl;`
- (e) `cout << (!(j - m)) << endl;`

5. Which of the following is a selection statement? Clearly circle your answer.

- (a) `for`
- (b) `if...else`
- (c) `do...while`
- (d) `while`

6. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 int main ()
5 {
6 int number1 = 5;
7 int number2 = 12;
8 int number3 = 7;
9 int myChoice = 0;
10
11 myChoice = number1;
12
13 if (number2 < myChoice) myChoice = number2;
14 if (number3 < myChoice) myChoice = number3;
15
16 cout << "My Choice is: " << myChoice;
17 return 0;
18 }
```

7. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5 int year = 2000;
6
7 if ((year % 4 == 0 && year % 100 != 0) || (year % 400 == 0))
8 cout << year << " is a leap year";
9 else
10 cout << year << " is not a leap year";
11
12 return 0;
13 }
```

8. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5 int num1 = 5, num2 = -3;
6
7 if (num1 > 0 && num2 > 0)
8 cout << "The sum is " << num1 + num2;
9 else if (num1 <= 0 || num2 <= 0)
10 cout << "Invalid input";
11 else
12 cout << "The difference is " << num1 - num2;
13
14 return 0;
15 }
```

9. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 int main ()
5 {
6 int x = 5;
7 int y = 3;
8 int z = 7;
9
10 if(x<=y && y<= z)
11 cout << "My Choice is: " << y;
12 if(y<=x && x<=z)
13 cout << "My Choice is: " << x;
14 if(x<=z && z<=y)
15 cout << "My Choice is: " << z;
16
17 return 0;
18 }
```

10. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6 int x = 6;
7 int y = 6;
8
9 if (x >= y)
10 cout << "Hello " << endl;
11 else
12 cout << "Goodbye " << endl;
13 cout << "Done!";
14 return 0;
15 }
```

11. Convert the following switch-statement into the equivalent if-else-statement.

```
1 switch (x)
2 {
3 case 1 :
4 n=1;
5 break;
6 case 2 :
7 n=2;
8 break;
9 case 3 :
10 n=3;
11 break;
12 default :
13 n=10;
14 break;
15 }
```

12. Convert the following switch-statement into the equivalent if-else-statement.

```
1 #include <iostream>
2 using namespace std;
3
4 // Choose day of week program
5 int main()
6 {
7 int day = 0; // day of the week
8 cout << "Enter a number between 1 and 7 for the day of the week: ";
9 cin >> day;
10
11 if (day == 1) cout << "Monday";
12 else if (day == 2) cout << "Tuesday";
13 else if (day == 3) cout << "Wednesday";
14 else if (day == 4) cout << "Thursday";
15 else if (day == 5) cout << "Friday";
16 else if (day == 6) cout << "Saturday";
17 else if (day == 7) cout << "Sunday";
18 else cout << "Not a valid entry";
19
20 return 0;
21 }
```

13. Write a program that allows the user to input three integers (not necessarily distinct) and prints the largest of the three values.

**Desired output:**

Enter three integers: **2 5 1**  
The largest integer is: 5

14. Write a program that allows the user to input three integers (not necessarily distinct) and prints the median of the three values.

**Desired output:**

Enter three integers: **2 7 1**  
The median is: 2

15. Write a program that inputs an integer and prints if the number is even or odd.

**Desired output:**

Enter an integer: **7**  
The number you entered is odd.

16. Write a C++ program that allows the user to input two integers and prints the sum, difference, product, quotient, and remainder of the two values.

**Desired output:**

Enter two integers: **17 5**  
Sum: 22  
Difference: 12  
Product: 85  
Quotient: 3  
Remainder: 2

17. Write a C++ program that inputs a grade percentage and prints the corresponding letter grade according to the following scale: A for percentages 90-100, B for 80-89, C for 70-79, D for 60-69, and F for percentages below 60.

**Desired output:**

Enter grade percentage: **83**  
Letter grade: B

18. Write a C++ program that inputs a number and checks if it is a palindrome (i.e., the number reads the same forwards and backwards).

**Desired output:**

Enter a number: **12321**  
12321 is a palindrome.

19. Write a program that inputs a number between 1 and 12 inclusive and prints the corresponding month of the year. If a different value is entered the program prints: "Not a valid entry!". **Use the switch statement.** As an alternative you can also write a different version of the program using the if statement.

**Sample output:**

Enter a number between 1 and 12 for the month of the year: **2**  
The month you selected is: February



20. Write a C++ program that inputs three sides of a triangle and checks if it is a valid triangle. A valid triangle has the property that the sum of the lengths of any two sides is greater than the length of the remaining side.

**Desired output:**

Enter three sides of a triangle: **5 12 8**

It is a valid triangle.



## 5. Repetitions Statements. (Loops)

Loops are a vital part of programming languages, such as C++, as they enable the repetition of statements based on specific conditions. C++ offers several loop structures that provide different ways to control program flow. The main loop structures in C++ include:

C++ offers several loop structures, including the *while*, *do-while*, *for*, and *range-based for* loops, which provide various ways to control program flow and repeat statements based on specific conditions.

These loop structures in C++ offer flexibility and control to developers, allowing them to repeat actions efficiently and manipulate program execution based on conditions.

## 5.1 The while loop.

### The while statement format

```
while (condition)
 statement;
```

### The while statement format

```
while (condition){
 statement1;
 statement2;
 statement3;
}
```

Code 5.1: Program using a loop to print numbers from 1 to 10

```
1 #include <iostream>
2 using namespace std;
3
4 int main () {
5 int number = 1;
6
7 while (number <= 10){
8 cout << number << endl;
9 number++;
10 }
11 return 0;
12 }
```



Remove the statement number++; from the previous program and run it. What do you notice?

Write a program that allows the user to enter an integer value n and prints all the positive integers starting with n, in decreasing order.



#### Desired output:

Enter a positive integer: 7

The numbers in decreasing order are: 7 6 5 4 3 2 1

Write a program that allows the user to enter an integer value n and prints all the positive even integers smaller or equal to n, in decreasing order.



#### Desired output:

Enter a positive integer: 7

The even numbers in decreasing order are: 6 4 2

## 5.2 The *do - while* loop.

The *do...while* statement in C++ is similar to the *while* statement, with the key difference that the list of statements is executed before the condition is checked. This means that the *do...while* loop is guaranteed to be executed at least once, regardless of the condition.

### The *do...while* statement format

```
do {
 statement1;
 statement2;
 statement3;
} while(condition);
```

Code 5.2: Example *do...while* statement

```
1 #include <iostream>
2 using namespace std;
3
4 int main () {
5 int number = 1;
6
7 do {
8 cout << number << endl;
9 number++;
10 } while (number <= 10);
11 return 0;
12 }
```



Change the value of `number` to 15. What do you notice? Does the program print anything on the screen?

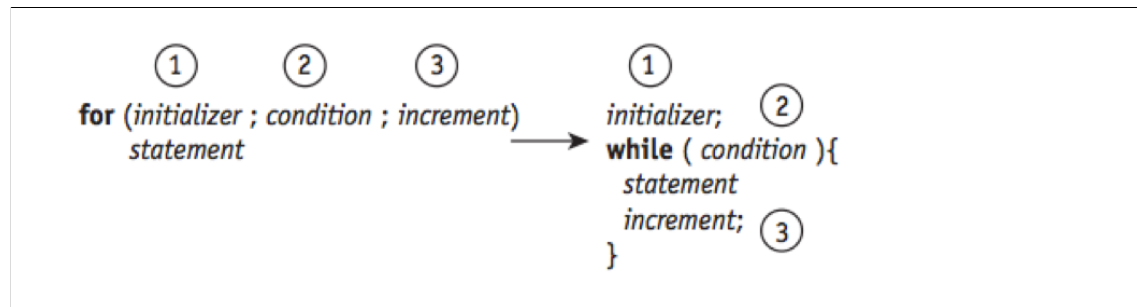
### 5.3 The *for* loop

The while loop can achieve everything that the for loop can accomplish. However, the for loop provides a more concise and widely used syntax.

#### The for statement format

```
for(initializer; condition; increment){
 statement1;
 statement2;
}
```

Comparison for and while loops:



Code 5.3: Printing 1 to 10 using the for loop

```
1 #include <iostream>
2 using namespace std;
3
4 int main () {
5 for(int number=1; number <=10; number++) {
6 cout << number << endl;
7 }
8 return 0;
9 }
```



Which one is your preferred loop statement? Why?



The next program calculates the factorial of a given number. What is the factorial of a number?

Code 5.4: Using a nested for loop to calculate the factorial of a number

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5 int factorial;
6 int n = 10;
7
8 cout << "N \t Factorial of N \n";
9
10 for (int i=1; i<=n; ++i) {
11 factorial=1;
12 for (int j=1; j<=i; ++j)
13 factorial*=j;
14 cout << i << '\t' << factorial << endl;
15 }
16 return 0;
17 }
```



Can we optimize the code in Factorial.cpp? Do we need a nested for loop?



<— Click here for an improved version of the factorial program with explanations. Try to answer the question above first before visiting this link.

Write a program that prints the shape below using only the following print statements:  
`cout << "*";` and `cout << endl;`

**NOTE: YOU MUST USE A NESTED LOOP FOR THIS QUESTION**



**Desired output:**

```
*
**


```

Write a program that prints the following diamond shape. You may use output statements that print a single asterisk (`cout << "*";`) a single blank (`cout << " ";`) or new line (`cout << endl;`). Maximize your use of repetition (with **nested for** structures) and minimize the number of output statements.



**Desired output:**

```
 *

 *
```



## 5.4 The *break* and *continue* Statements

The *break* statement when executed within any of the loop statements it causes the immediate exit from the loop statement.

Code 5.5: Example break statement.

```
1 #include <iostream>
2 using namespace std;
3
4 int main () {
5
6 for(int number=1; number <=10; number++) {
7 if(number == 5)
8 break;
9 cout << number << endl;
10 }
11 return 0;
12 }
```

The *continue* statement when used within any of the loop statements it causes the remaining statements in the body of the loop to be skipped. The loop then continues with the next iteration.

Code 5.6: Example continue statement.

```
1 #include <iostream>
2 using namespace std;
3
4 int main () {
5
6 for(int number=1; number <=10; number++) {
7 if(number == 5)
8 continue;
9 cout << number << endl;
10 }
11 return 0;
12 }
```

## 5.5 Practice Questions

1. What is the meaning of the following in the context of Computer Science?

(a) iteration:

---

---

(b) sentinel:

---

---

(c) initializer:

---

---

(d) Nested loop:

---

---

(e) increment / decrement:

---

---

2. True-False questions – circle your choice.

- (a) **(True/False)** In C++, a `while` loop executes the loop body at least once, regardless of the loop condition.
- (b) **(True/False)** A `do-while` loop in C++ checks the loop condition at the beginning of each iteration.
- (c) **(True/False)** The `break` statement can be used in a loop to immediately terminate the loop and exit.
- (d) **(True/False)** The `continue` statement in C++ skips the remaining statements in the current iteration and proceeds to the next iteration.
- (e) **(True/False)** C++ supports nested loops, where a loop can be placed inside another loop.
- (f) **(True/False)** The loop variable in a `for` loop in C++ can only be of type `int`.
- (g) **(True/False)** The `do-while` loop always executes the loop body at least once, regardless of the loop condition.
- (h) **(True/False)** The `while` loop can be used to create an infinite loop if the loop condition is always `true`.
- (i) **(True/False)** The `for` loop in C++ can have multiple loop variables.
- (j) **(True/False)** The `do-while` loop is particularly useful when you want to execute the loop body before checking the loop condition.
- (k) **(True/False)** In C++, the `while` loop and the `for` loop are interchangeable and can be used interchangeably in all scenarios.

3. Which of the following is a C++ repetition statement?

- (a) if
- (b) if...else
- (c) do...while
- (d) switch
- (e) repeat

4. What is wrong with the following while loop? Explain.

```
1 int x=5;
2 while (x <= 10)
3 x -=2;
```

5. The following code should output all even integers from 2 to 100 inclusive. Find and fix the errors.

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5
6 counter = 2;
7
8 do{
9 cout << counter << endl;
10 counter +=2;
11 } While (counter < 100);
12
13 return 0;
14 }
```

6. The following code should output all powers of 2 between 1 and 100 (as shown in the desired output column). Find and fix the errors.

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5 counter = 1;
6 do{
7 counter +=2;
8 cout << counter << endl;
9 } while (counter < 100);
10
11 return 0;
12 }
```

**Desired output:**

```
2
4
8
16
32
64
```

7. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 int main ()
5 {
6 for(int i=1;i<=2;i=i+1)
7 for (int j = 1; j <= 3; j = j + 1)
8 cout << i << " - " << j << endl;
9
10 return 0;
11 }
```

8. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5 int i = 1;
6 int j = 1;
7
8 while (i <= 3) {
9 cout << "Outer loop iteration: " << i << endl;
10
11 while (j <= 5) {
12 cout << "Inner loop iteration: " << j << endl;
13 j++;
14 }
15
16 j = 1;
17 i++;
18 }
19
20 return 0;
21 }
```

9. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5 int num1, num2, i = 1, gcd = 1;
6 cout << "Enter first number: ";
7 cin >> num1;
8 cout << "Enter second number: ";
9 cin >> num2;
10 while (i <= num1 && i <= num2) {
11 if (num1 % i == 0 && num2 % i == 0) {
12 gcd = i;
13 }
14 i++;
15 }
16 cout << "GCD of " << num1 << " and " << num2 << " is " << gcd << endl;
17 return 0;
18 }
```

10. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 int main ()
5 {
6 for(int number=1; number <10; number++) {
7 if(number % 3 != 0)
8 continue;
9 cout << number << endl;
10 }
11 return 0;
12 }
```

11. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 int main ()
5 {
6 int number = 1;
7 while (number < 10){
8 cout << number << endl;
9 number+=3;
10 }
11 return 0;
12 }
```

12. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5 int i = 0, j = 1, k = 1, sum=0;
6
7 while(i < 10){
8 cout << j << endl;
9 sum=j+k;
10 j=k;
11 k=sum;
12 i++;
13 }
14 }
```

13. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5 for (int i=1; i<=3; ++i){
6 for (int j=1; j<=3; ++j){
7 for (int k=1; k<=3; ++k)
8 cout << '*';
9 cout << endl;
10 }
11 cout << endl;
12 }
13 return 0;
14 }
```



14. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 int main ()
5 {
6 int number = 15;
7
8 do {
9 cout << number << endl;
10 number++;
11 } while (number <= 10);
12 return 0;
13 }
```

15. Make only one change to the program below so that the current output changes to the desired output.

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5 int y;
6 for (int i = 1; i <= 9; i++) {
7 y=0;
8 for (int j = 1; j <= i; j++) {
9 y += 1;
10 cout << i << " ";
11 }
12 cout << endl;
13 }
14 return 0;
15 }
```

**Current output:**

```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
6 6 6 6 6 6
7 7 7 7 7 7 7
8 8 8 8 8 8 8 8
9 9 9 9 9 9 9 9 9
```

**Desired output:**

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
1 2 3 4 5 6 7
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8 9
```

16. Write a C++ program that uses a nested loop to print the multiplication table for numbers from 1 to 10 as shown below.

**Desired output:**

|    |    |    |    |    |    |    |    |    |     |
|----|----|----|----|----|----|----|----|----|-----|
| 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10  |
| 2  | 4  | 6  | 8  | 10 | 12 | 14 | 16 | 18 | 20  |
| 3  | 6  | 9  | 12 | 15 | 18 | 21 | 24 | 27 | 30  |
| 4  | 8  | 12 | 16 | 20 | 24 | 28 | 32 | 36 | 40  |
| 5  | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 | 50  |
| 6  | 12 | 18 | 24 | 30 | 36 | 42 | 48 | 54 | 60  |
| 7  | 14 | 21 | 28 | 35 | 42 | 49 | 56 | 63 | 70  |
| 8  | 16 | 24 | 32 | 40 | 48 | 56 | 64 | 72 | 80  |
| 9  | 18 | 27 | 36 | 45 | 54 | 63 | 72 | 81 | 90  |
| 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |

17. Write a program that allows the user to enter an integer value n and prints all the positive integers starting with n, in decreasing order.

**Sample output:**

Enter a positive integer: **7**  
The numbers in decreasing order are: 7 6 5 4 3 2 1

18. Write a program that allows the user to enter an integer value n and prints all the positive even integers smaller or equal to n, in decreasing order.

**Sample output:**

Enter a positive integer: **7**  
The even numbers in decreasing order are: 6 4 2

19. Write a program that allows the user to enter an integer value n and prints all the positive odd integers smaller or equal to n, in decreasing order.

**Sample output:**

Enter a positive integer: **7**  
The odd numbers in decreasing order are: 7 5 3 1

20. Write a program that allows the user to enter a positive integer value  $n$  and prints all the integers from 1 to  $n$  and back down to 1.

**Sample output:**

```
Enter a positive integer: 5
The numbers are: 1 2 3 4 5 4 3 2 1
```

21. Write a program that has the output shown below and uses **only one for loop**. Hint: The numbers in the second column are the factorials of the numbers in the first column.

**Desired output:**

```
1 1
2 2
3 6
4 24
5 120
6 720
```

22. Write a C++ program that prompts the user to enter a positive integer and checks if it is a palindrome.

**Sample output:**

```
Enter a positive integer: 12321
12321 is a palindrome.
```

## 6. Functions and Parameters

### 6.1 What are Functions?

Functions are an essential part of any C++ program. They allow programmers to group together a set of statements that accomplish a specific task and make the program easier to read, maintain, and debug. C++ provides a rich set of predefined functions that can be used to perform a wide variety of tasks, and programmers can also create their own functions to perform specific tasks. Overall, functions are a crucial aspect of C++ programming and are used extensively in real-world applications to improve code organization, reusability, and performance.

#### Components of a function

```
return_type function_name (parameter_list) {
 statements

 return statement; (if applicable)
}
```

Code 6.1: A program that defines and tests a function that calculates the average of two numbers.

```
1 #include <iostream>
2 using namespace std;
3
4 // Function must be declared before being used.
5 double average(double x, double y);
6
7 int main() {
8
9 double a = 0.0;
10 double b = 0.0;
11
12 cout << "Enter first number: ";
13 cin >> a;
14 cout << "Enter second number: ";
15 cin >> b;
16
17 // Call the function average().
18 cout << "Average is: " << average(a, b) << endl;
19
20 return 0;
21 } // end main()
22
23
24 // Average-number function definition
25 double average(double x, double y) {
26
27 double avg = (x + y)/2;
28 return avg;
29
30 } //end average() function
```

Code 6.2: A program that defines and tests a factorial function.

```
1 #include <iostream>
2 using namespace std;
3
4 int factorial(int x);
5
6 int main() {
7
8 int number;
9
10 cout << "Enter an integer: ";
11 cin >> number;
12
13 cout << number << "! = " << factorial(number) << endl;
14
15 return 0;
16 } // end main()
17
18 int factorial(int x) {
19
20 int f=1;
21
22 for(int i=2; i<=x; i++){
23 f *= i;
24 }
25
26 return f;
27
28 } //end factorial()
```

Code 6.3: A program that defines and tests a function that prints the first n integers.

```
1 #include <iostream>
2 using namespace std;
3
4 void printAllIntegers(int x);
5
6 int main() {
7
8 int n;
9 cout << "Enter an integer: ";
10 cin >> n;
11
12 cout << "The first " << n << " positive integers are:" << endl;
13 printAllIntegers(n);
14
15 return 0;
16
17 } // end main()
18
19 void printAllIntegers(int x) {
20
21 for(int i=1; i<=x; i++){
22 cout << i << endl;
23 }
24
25 } // end printAllIntegers()
```

Write a C++ program that defines and tests a function **largest(...)** that takes as parameters any three integers and returns the largest of the three integers. Your output should have the same format and should work for any integers a user enters.

**Desired output:**

Enter three integers: 6 15 8

The largest integer is: 15



Code 6.4: A program that checks whether a number is prime.

```
1 #include <iostream>
2 #include <cmath>
3 using namespace std;
4
5 bool prime(int);
6
7 int main() {
8
9 int n;
10
11 while (true) {
12 cout << "Enter number (0 = exit): ";
13 cin >> n;
14 if (n == 0)
15 break;
16 if (prime(n))
17 cout << n << " is prime" << endl;
18 else
19 cout << n << " is not prime" << endl;
20 }
21 return 0;
22 } // end main
23
24 bool prime(int x) {
25 for (int i = 2; i < x; i++)
26 if (x % i == 0) return false;
27 return true;
28 } // end prime()
```



← Click here for a version of this program.



Optimize the prime-number function. Can we reduce the current number of operations?

Rewrite main so that it tests all the numbers from 2 to 10 (inclusive) and prints out the results, each on a separate line. (Hint: Use a for loop, with i running from 2 to 10.)

**Desired output:**

```
2 is prime
3 is prime
4 is not prime
5 is prime
6 is not prime
7 is prime
8 is not prime
9 is not prime
10 is not prime
```

Write a C++ program that defines and tests a function **power(base, exponent)** that takes two integers b and x and returns  $b^x$  by using only multiplication. Design your own function. Do not use the built in in function pow(). Your output should have the same format as shown below and should work for any integers.

**Desired output:**

```
Enter base: 2
Enter exponent: 3
2 to the power 3 is 8
```

## 6.2 Passing a Parameter by Value or by Reference

There are two ways to pass arguments to a function (by value or by reference). What we have done so far is to pass the value of the parameter. We illustrate both possibilities via examples.

Code 6.5: Passing by value example

```
1 #include <iostream>
2 using namespace std;
3
4 int addOne(int);
5
6 int main(){
7
8 int n = 3;
9
10 cout << "n before calling addOne(): " << n << endl;
11 cout << "Value returned by addOne(): " << addOne(n) << endl;
12 cout << "n after calling addOne() is: " << n << endl;
13 }
14
15 int addOne(int value){
16 value++;
17 return value;
18 }
```

Code 6.6: Passing by reference example

```
1 #include <iostream>
2 using namespace std;
3
4 int addOne(int &);
5
6 int main(){
7
8 int n = 3;
9
10 cout << "n before calling addOne(): " << n << endl;
11 cout << "Value returned by addOne(): " << addOne(n) << endl;
12 cout << "n after calling addOne() is: " << n << endl;
13 }
14
15 int addOne(int &value){
16 value++;
17 return value;
18 }
```

### 6.3 Default Function Arguments

It is possible to assign default values to each of the function arguments. This allows the caller to omit argument values. If the value is not specified by the caller, the default value will be used. See examples below.

Code 6.7: Default arguments example - Rectangle area

```

1 #include <iostream>
2 using namespace std;
3
4 int rArea(int length = 2, int width = 1);
5
6 int main() {
7
8 cout << "Default area : " << rArea() << endl;
9 cout << "length=10 and default width: " << rArea(10) << endl;
10 cout << "length=10 and width=5: " << rArea(10, 5) << endl;
11
12 return 0;
13 }
14
15 int rArea(int length, int width){
16
17 return length*width;
18 }
```



Can we assign a default value to only length or only width?

Code 6.8: Additional Example - Interest

```

1 #include <iostream>
2 using namespace std;
3
4 double interest(double p = 1000, double r = 0.05, double t = 1);
5
6 int main() {
7
8 cout << "interest(3000, 0.015, 5): "
9 << interest(3000, 0.015, 5) << endl;
10 cout << "interest(3000, 0.015): " << interest(3000, 0.015) << endl;
11 cout << "interest(3000): " << interest(3000) << endl;
12 cout << "interest(): " << interest() << endl;
13
14 return 0;
15 }
16
17 double interest(double p, double r, double t){
18
19 return p*r*t;
20 }
```

## 6.4 Function Overloading

It is possible to declare different functions using the same function identifier. These functions however have different signatures, typically different data types. The C++ compiler will select the version of the function that matches with the signature.

Code 6.9: Function overloading example

```

1 #include <iostream>
2 using namespace std;
3
4 int addNumbers(int , int);
5 double addNumbers(double , double);
6
7 int main () {
8 int xInt = 5;
9 int yInt = 7;
10 double xDouble = 3.3;
11 double yDouble = 2.4;
12
13 cout << "Calling the integer version." << endl;
14 cout << "Sum is: " << addNumbers(xInt , yInt) << endl << endl;
15 cout << "Calling the double version." << endl;
16 cout << "Sum is: " << addNumbers(xDouble , yDouble) << endl;
17
18 return 0;
19 }
20
21 int addNumbers(int first , int second) {
22 cout << "Inside the integer version." << endl;
23 return first + second;
24 }
25
26 double addNumbers(double first , double second) {
27 cout << "Inside the double version." << endl;
28 return first + second;
29 }

```



← Get here a similar version of the program.



Can we overload a function by changing the number of arguments?



Can we overload a function by only changing the return type?

## 6.5 Practice Questions

1. What is the meaning of the following in the context of Computer Science?

(a) function return type:

---

---

(b) pre-defined and user-defined functions:

---

---

(c) function arguments / parameters:

---

---

(d) function call:

---

---

(e) void return type:

---

---

(f) passing an argument by value or by reference:

---

---

(g) function overloading:

---

---

(h) default function values:

- 
- 
- (i) function header and function body:

- 
- 
- (j) function declaration / function prototype:

2. True-False questions – Circle your choice

- (a) (**True** - False) Function parameters in C++ can have default values, allowing the caller to omit them during function invocation.
- (b) (True - **False**) C++ functions can have multiple return values.
- (c) (**True** - False) A function in C++ can have the same name as a variable.
- (d) (**True** - False) C++ supports pass-by-value and pass-by-reference for function arguments.
- (e) (True - **False**) C++ functions can only return numeric data types.
- (f) (**True** - False) C++ functions must always have a return type specified.
- (g) (**True** - False) Function overloading in C++ allows multiple functions with the same name and different parameter list.
- (h) (**True** - False) C++ supports recursive functions, which are functions that call themselves.
- (i) (**True** - False) C++ allows functions to be defined with no parameters.
- (j) (**True** - False) C++ functions can have a void return type, indicating that they do not return a value.

3. If an argument to a function is declared as **const**, then ...

- (a) Function can modify the argument
- (b) Function can't modify the argument
- (c) const argument to a function is not possible
- (d) None of the above

4. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 int f(int b, int e);
5
6 int main() {
7 int n1 = 2, n2 = 6;
8 cout << n1 << "^" << n2 << " = " << f(n1, n2) << endl;
9 return 0;
10 } // end main
11
12 int f(int b, int e) {
13 int result=1;
14 for (int i = 1; i <= e; i++)
15 result *= b;
16 return result;
17 }
```



5. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 int rectangleArea(int length = 2, int width = 1);
5
6 int main(){
7 cout << "First Rectangle : " << rectangleArea() << endl;
8 cout << "Second Rectangle: " << rectangleArea(5) << endl;
9 cout << "Third Rectangle: " << rectangleArea(10, 2) << endl;
10 return 0;
11 }
12
13 int rectangleArea(int length , int width){
14 return length*width;
15 }
```

6. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 int myFunction(int);
5
6 int main(){
7 int number = 5;
8 cout << number << endl;
9 cout << myFunction(number) << endl;
10 cout << number << endl;
11 }
12
13 int myFunction(int value){
14 value*=5;
15 return value;
16 }
```

7. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 int myFunction(int &);
5
6 int main(){
7 int number = 5;
8 cout << number << endl;
9 cout << myFunction(number) << endl;
10 cout << number << endl;
11 }
12
13 int myFunction(int &value){
14 value*=5;
15 return value;
16 }
```

8. Write a program that defines and tests function **factorial()**. Use a **for loop** for this function. Sample output is given.

```
1 #include <iostream>
2 using namespace std;
3
4 int factorial(int x);
5
6 int main() {
7 int number;
8 cout << "Enter an integer for the desired factorial: ";
9 cin >> number;
10 cout << number << "! = " << factorial(number) << endl;
11 return 0;
12 }
13
14 int factorial(int x) {
15
16 }
```

**Sample output:**

Enter an integer for the desired factorial: 5  
5! = 120

9. Write a program that defines and tests a function that prints the first n positive integers followed by their sum. Use a **while loop** for this function. Sample output is given.

```
1 #include <iostream>
2 using namespace std;
3
4 void printAllIntegers(int x);
5
6 int main() {
7 int number;
8 cout << "Enter an Integer: ";
9 cin >> number;
10 cout << "The first " << number << "integers are listed below:" << endl;
11 printAllIntegers(number);
12 return 0;
13 } // end main() function
14
15 void printAllIntegers(int x) {
16
17
18
19
20
21
22
23 }
```

**Sample output:**

```
Enter an Integer: 6
The first 6 positive integers are listed below:
1
2
3
4
5
6
Sum = 21
```

10. Write a C++ program that defines and tests a function `largest(...)` that takes as parameters any three integers and returns the largest of the three integers.

**Sample output:**

Enter three integers: **6 15 8**  
The largest integer is: 15

11. Write a C++ program that defines and tests a function `absoluteValue(...)` that takes as a parameter a floating point number and returns its absolute value.

**Sample output:**

Enter a number: **-4.25**  
The absolute value is: 4.25

12. Write a C++ program that defines a function `reverseInt(int n)` that takes as an argument a positive integer and returns an integer with digits in reverse order. Print `n` before and after reversing the digits. Your output should have the same format and should work for any integer a user enters.

**Sample output:**

The value before reverse is: **425**  
The value after reverse is: 524

13. Write a C++ program that defines and tests a function `multiplyNumbers(m,n)` that takes two positive integers `m` and `n`, computes `m * n` by using only addition, and returns the product. Hint: Use iteration and notice for example that  $3*2=2+2+2$

**Sample output:**

Enter m: **2**  
Enter n: **3**  
**2\*3 = 6**

14. Write a C++ program that defines and tests a function **power(base, exponent)** that takes two integers  $b$  and  $x$  and returns  $b^x$  by using only multiplication. Design your own function. Do not use the built in function `pow()`.

**Sample output:**

```
Enter base: 2
Enter Exponent: 3
2 to the power 3 = 8
```

15. Write a boolean function **divisibleBySeven(...)** definition that takes as an input an integer and returns true if the integer is divisible by 7, false if not. Make sure you use the proper function return type and argument list. Just write the function definition. You do not need to write a complete C++ program.
16. Write a function **isDigit(...)** definition that takes one argument of type `char` and returns a `bool` value. The function returns true if the argument is a decimal digit; otherwise it returns false. Make sure you use the proper function return type and argument list. Just write the function definition. You do not need to write a complete C++ program.
17. Write a function **multiple(...)** definition that determines for a pair of integers whether the second integer is a multiple of the first. The function should take two integer arguments and return true if the second is a multiple of the first, false otherwise. . Make sure you use the proper function return type and argument list. Just write the function definition. You do not need to write a complete C++ program.



## 7. Variable Scope

### 7.1 Local and Global Variables

In C++, variable scope plays a crucial role in writing efficient and organized code. When it comes to variable scope, we encounter two primary types: local variables and global variables. A local variable is declared inside a function, making it accessible and usable only within that specific function. Conversely, a global variable is declared outside of any function, including the main function, making it accessible from any part of the program. Generally, it is considered good practice to use local variables unless there is a specific requirement for a global variable that needs to be shared across multiple functions.

Code 7.1: Global vs. local variables example

```
1 #include <iostream>
2 using namespace std;
3
4 int i = 1;
5
6 void firstFunction();
7 void secondFunction();
8 void thirdFunction();
9
10 int main() {
11 int i = 2;
12 cout << "Inside main(): i=" << i << endl;
13
14 firstFunction();
15 secondFunction();
16 thirdFunction();
17
18 return 0;
19 }
20
21 void firstFunction(){
22 int i = 3;
23 cout << "Inside firstFunction(): i=" << i << endl;
24 }
25
26 void secondFunction(){
27 cout << "Inside secondFunction(): i=" << i << endl;
28 }
29
30 void thirdFunction(){
31 int i = 4;
32 cout << "Inside thirdFunction(): i=" << ::i << endl;
33 }
```



← Get a version of the code here.



## 7.2 Default and Static Variables

In C++, the lifetime of a variable is determined by its storage category. Two commonly used storage categories are default (auto) and static. A default (auto) local variable is automatically allocated memory space when a function is executed and is released once the function returns, making the memory available for other processes. On the other hand, a static variable persists throughout the entire execution of the program, retaining its value and memory allocation.

Code 7.2: Example auto variable

```
1 #include <iostream>
2 using namespace std;
3
4 void test();
5
6 int main() {
7
8 for(int i=1; i<=3; i++)
9 test();
10
11 return 0;
12 } //end main()
13
14 void test(){
15
16 int number = 0;
17 cout << "number is: " << number << endl;
18 number++;
19
20 return;
21 } //end test()
```



Change the local variable 'number' to static and run the program. What do you observe?



Change the keyword int for variable 'number' to auto. What do you observe?

### 7.3 Pre-defined Functions

In C++, programmers have access to a multitude of pre-defined functions that can greatly enhance their programming capabilities. These functions are organized into collections known as libraries. By including the appropriate library, users gain access to these functions and can leverage their functionality within their programs. One such library is `<cmath>`, which provides a range of mathematical functions. Let's consider an example program that showcases some of the functions from this library.

Code 7.3: Using functions from the `cmath` library

```
1 #include <iostream>
2 #include <cmath>
3 #include <iomanip>
4 using namespace std;
5
6 int main() {
7 cout << fixed << setprecision(2);
8 cout << "sqrt(12.0) = " << sqrt(12.0) << endl;
9 cout << "exp(1.0) = " << setprecision(3) << exp(1.0) << endl;
10 cout << "log(8.32) = " << log(8.32) << endl;
11 cout << "log(exp(8.23)) = " << log(exp(8.23)) << endl;
12 cout << "pow(2,3) = " << pow(2,3) << endl;
13 cout << "cos(90) = " << setprecision(15) << cos(90) << endl;
14
15 return 0;
16 }
```



← Get code here.

Code 7.4: Random number generator

```
1 #include <iostream>
2 #include <cmath>
3 #include <cstdlib>
4 #include <ctime>
5 using namespace std;
6
7 int main() {
8 int n, r;
9 srand(time(NULL));
10 cout << "How many numbers do you want? ";
11 cin >> n;
12 for (int i = 1; i <= n; i++) {
13 r = rand() % 9 + 1;
14 cout << r << " ";
15 }
16 return 0;
17 }
```



What is the range of numbers generated by the program?



Modify the program to generate numbers in the range [5,10].



Modify the program to generate numbers in the range [-3,25].



Modify the program to allow the user to specify the range of values to be generated.

Modify the program to display a frequency distribution of the values and graph the results.



**Sample output:**

How many number do you want? 20

2 2 6 3 2 3 4 1 4 3 4 5 5 4 3 3 3 3 4 2

| Value | Freq | Graph |
|-------|------|-------|
| 1     | 1    | *     |
| 2     | 4    | ****  |
| 3     | 7    | ***** |
| 4     | 5    | ***** |
| 5     | 2    | **    |
| 6     | 1    | *     |

## 7.4 Practice Questions

1. What is the meaning of the following in the context of Computer Science?

(a) local and global variables:

---

---

(b) Stack:

---

---

(c) scope of a variable:

---

---

(d) static variable:

---

---

2. The statement `n = rand();` assigns to variable `n` a value between 0 and a very large positive integer. Change the statement to assign to `n` a value in the following ranges.

(a) `[1, 5]`

(b) `[5, 100]`

(c) `[-3, 18]`

## 3. True-False questions – Circle your choice

- (a) (**True** - False) In C++, local variables are only accessible within the block of code where they are defined.
- (b) (**True** - False) Global variables can be accessed by any part of the program.
- (c) (**True** - False) Local variables take precedence over global variables if they share the same name.
- (d) (**True** - False) Static local variables in C++ retain their value between function calls.
- (e) (**True** - False) The scope of a local variable is limited to the function in which it is defined.
- (f) (True - **False**) Global variables are always initialized with a default value by the compiler.
- (g) (**True** - False) Static variables are initialized only once during the entire program execution.
- (h) (**True** - False) Local variables can have the same name as a global variable without any conflict.
- (i) (**True** - False) Global variables can be modified by any function in the program.
- (j) (True - **False**) Static variables are accessed using the scope resolution operator (::).
- (k) (True - **False**) The scope of a global variable is limited to the function in which it is defined.
- (l) (True - **False**) Pre-defined functions in C++ are always accessible without including any specific library.
- (m) (True - **False**) Local variables are automatically initialized with a default value by the compiler.
- (n) (**True** - False) The <ctime> library in C++ provides functions for date and time manipulation.
- (o) (True - **False**) Local variables can be modified by any function in the program.

4. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 int myFunction();
5
6 int main() {
7 cout << myFunction() << endl;
8 cout << myFunction() << endl;
9 cout << myFunction() << endl;
10 return 0;
11 }
12
13 int myFunction() {
14 int number = 0;
15 return number++;
16 }
```

5. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 int myFunction();
5
6 int main() {
7 cout << myFunction() << endl;
8 cout << myFunction() << endl;
9 cout << myFunction() << endl;
10 return 0;
11 }
12
13 int myFunction() {
14 static int number = 0;
15 return number++;
16 }
```

6. Write the exact output for the following code.

```
1 #include <iostream>
2 using namespace std;
3
4 char myFunction();
5
6 int main() {
7 cout << myFunction() << endl;
8 cout << myFunction() << endl;
9 cout << myFunction() << endl;
10 return 0;
11 }
12
13 char myFunction(){
14 static char character = 'a';
15 return character++;
16 }
```

7. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 int i = 7;
5
6 void firstFunction();
7 void secondFunction();
8
9 int main() {
10 int i = 2;
11 cout << "Inside main(): i=" << i << endl;
12 firstFunction();
13 secondFunction();
14 return 0;
15 }
16
17 void firstFunction(){
18 int i = 5;
19 cout << "Inside firstFunction(): i=" << i << endl;
20 }
21
22 void secondFunction(){
23 cout << "Inside secondFunction(): i=" << i << endl;
24 }
```





## 8. Arrays

### 8.1 What are Arrays

An array is an ordered collection (potentially very large) of items. Arrays are indexed to allow easier access to its elements. Arrays are powerful structures that allow the user to store multiple data values of the same type without having to declare as many variables.

Assume we want to store 5 double data values for further processing (example find their average). One way is to declare 5 different double variables, one for each value. This is not an efficient way of storing data. Imagine we have to do this for a lot more data values. The following program uses an array to store the data values and calculate the average of all these values.

Code 8.1: Example using arrays

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5
6 double myArray[5];
7
8 cout << "Enter values: " << endl;
9
10 for (int i = 0; i < 5; i++)
11 cin >> myArray[i];
12
13 double sum=0;
14 for (int i = 0; i < 5; i++)
15 sum += myArray[i];
16
17 cout << "The Average is: " << sum / 5 << endl;
18 return 0;
19 }
```

## 8.2 Initializing Arrays

Arrays follow the same principles regarding initialization as variables. Global numerical variables (arrays) are all initialized to zero by default if the user does not specify any values. Local variables must be initialized by the user otherwise they will be possibly assigned random values.

Code 8.2: Initializing global and local arrays

```
1 #include <iostream>
2 using namespace std;
3
4 int globalArray[6];
5
6 int main() {
7
8 int localArray[8];
9
10 cout << "GlobalArray values before initializing." << endl;
11 for (int i=0; i<6; i++)
12 cout << globalArray[i] << " ";
13
14 cout << endl << endl;
15
16 cout << "LocalArray values before initializing." << endl;
17 for (int i=0; i<8; i++)
18 cout << localArray[i] << " ";
19
20 cout << endl << endl;
21
22 int initializedLocalArray[5] = {3,4,6,2,4};
23
24 cout << "InitializedLocalArray values." << endl;
25 for (int i=0; i<5; i++)
26 cout << initializedLocalArray[i] << " ";
27
28 return 0;
29 }
```



&lt;— Get code here.



Modify the following code to use an array instead of the individually declared variables. Does this approach improve code writing and code readability?



&lt;— Get code here.

Code 8.3: Random number simulator with statistics

```

1 #include <iostream>
2 #include <cstdlib>
3 #include <ctime>
4 using namespace std;
5
6 void keepRecord(int);
7 void displayStatistics();
8 int n1, n2, n3, n4, n5;
9
10 int main() {
11 int n, r;
12 srand(time(NULL));
13 cout << "How many numbers do you want: ";
14 cin >> n;
15
16 for (int i = 1; i <= n; i++) {
17 r = rand() % 5 + 1;
18 cout << r << " ";
19 keepRecord(r);
20 }
21 cout << endl << endl;
22 displayStatistics();
23 return 0;
24 }
25
26 void keepRecord(int value){
27 switch (value){
28 case 1 : n1++; break;
29 case 2 : n2++; break;
30 case 3 : n3++; break;
31 case 4 : n4++; break;
32 case 5 : n5++; break;
33 }
34 }
35
36 void displayStatistics(){
37 cout << "Value \t" << "Freq \t" << "Graph" << endl << endl;
38 cout << "1 \t" << n1 << "\t" ;
39 for(int i=0; i<n1;i++) cout << "*";
40 cout << endl;
41
42 cout << "2 \t" << n2 << "\t" ;
43 for(int i=0; i<n2;i++) cout << "*";
44 cout << endl;
45
46 cout << "3 \t" << n3 << "\t" ;
47 for(int i=0; i<n3;i++) cout << "*";
48 cout << endl;
49
50 cout << "4 \t" << n4 << "\t" ;
51 for(int i=0; i<n4;i++) cout << "*";
52 cout << endl;
53
54 cout << "5 \t" << n5 << "\t" ;
55 for(int i=0; i<n5;i++) cout << "*";
56 cout << endl;
57 }

```

### 8.3 Array Size

It is a good practice to declare a constant integer for the size of the array and use it during the program as needed.

Code 8.4: Using constant array size example

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5
6 const int myArraySize = 5;
7 int myArray[myArraySize] = {3,4,6,2,4};
8
9 cout << "Number of elements: " << myArraySize << endl;
10 cout << "List of elements: ";
11 for (int i=0; i<myArraySize; i++)
12 cout << myArray[i] << " ";
13
14 return 0;
15 }
```

Another way of getting the array size.

Code 8.5: Calculating number of array elements

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5
6 int myArray[5] = {3,4,6,2,4};
7
8 cout << "Array size in bytes: " << sizeof(myArray) << endl;
9 cout << "First element size: " << sizeof(myArray[0]) << endl;
10 int size = sizeof(myArray)/sizeof(myArray[0]);
11 cout << "The number of elements is: " << size << endl;
12 cout << "Elements in the arrays are: ";
13 for (int i=0; i<size; i++)
14 cout << myArray[i] << " ";
15
16 return 0;
17 }
```



← Get code here.

## 8.4 Passing Arrays to Functions

In C++ arrays are passed to functions by reference. Individual elements from the array are passed by value. See example below.

Code 8.6: Passing Arrays to Functions Example

```

1 #include <iostream>
2 using namespace std;
3
4 void modifyArray(int [], int);
5
6 int main() {
7
8 const int arraySize = 5;
9 int myArray[arraySize] = {3,4,6,2,8};
10
11 cout << "Elements in the original array are: " << endl;
12 for (int i=0; i<arraySize; i++)
13 cout << myArray[i] << " ";
14 cout << endl << endl;
15
16 modifyArray(myArray, arraySize);
17
18 cout << "Elements in the modified array are: " << endl;
19 for (int i=0; i<arraySize; i++)
20 cout << myArray[i] << " ";
21 cout << endl;
22
23 return 0;
24 } // end main
25
26 void modifyArray(int a[], int sizeOfArray){
27 for (int i=0; i<sizeOfArray; i++)
28 a[i]++;
29 } // end modifyArray()

```



← Get code here.



Add another value to the number set 3,4,6,2,8, in PassingArraysToFunctions.cpp, compile and run the program. Does the program run? What do you observe? What can you conclude?



Change the function signature to read void modifyArray(const int a[], int sizeOfArray). Compile and run the program. What do you notice?



Add a 'cout' statement that prints myArray[5]. Does it work? What do you think is happening?



Write a program that uses a function **largestValue(int[], int)** that takes as an argument an array of integers and the size of the array and returns the largest value in the array. Part of the code is given. Complete the remaining code (function definition) to produce the desired output.

**Desired output format:**

The largest value is: 25

Code 8.7: Class code activity

```
1 #include <iostream>
2 using namespace std;
3
4 int largestValue(int [], int);
5
6 int main() {
7
8 const int arraySize = 10;
9 int myArray[arraySize] = {3, 4, 6, 21, 8, 13, 25, 1, 7, 11};
10
11 cout << "The largest integer is: " ;
12 cout << largestValue(myArray, arraySize);
13
14 return 0;
15 } // end main
16
17 int largestValue(int a[], int sizeOfArray){
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33 } // end largestValue()
```

## 8.5 Characters, Strings and Arrays of Strings

Code 8.8: Example illustrating different char arrays.

```

1 #include <iostream>
2 using namespace std;
3
4 int main(){
5
6 char firstArray [8];
7
8 for(int i=0; i<8;i++)
9 firstArray [i]='d';
10
11 cout << "Print firstArray using for loop: " << endl;
12 for(int i=0; i<7;i++)
13 cout << firstArray [i];
14
15 firstArray [7]='\0';
16 cout << endl;
17
18 cout << "Print firstArray using identifier: " << endl;
19 cout << firstArray << endl;
20
21 char secondArray [] = { 'H', 'e', 'l', 'l', 'o', '\0' };
22 cout << "Print secondArray using for loop: " << endl;
23 for(int i=0; i<6;i++)
24 cout << secondArray [i];
25 cout << endl;
26
27 char thirdArray [] = "Test";
28 cout << "Print thirdArray using for loop: " << endl;
29 for(int i=0; i<4;i++)
30 cout << thirdArray [i];
31 cout << endl;
32
33 cout << "Changing one digit in third array: " << endl;
34 thirdArray [0]='N';
35 cout << "Print thirdArray using for loop: " << endl;
36 for(int i=0; i<4;i++)
37 cout << thirdArray [i];
38
39 return 0;
40 }

```



← Get code here.



Comment out the statement `firstArray[7]='\0';`. What changes do you see in the output?



Code 8.9: Array of Strings Example

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5
6 char *weekDays[7]= { "Monday", "Tuesday", "Wednesday",
7 "Thursday", "Friday", "Saturday", "Sunday" };
8
9 for(int i=0; i<7;i++)
10 cout << weekDays[i] << endl;
11
12 return 0;
13 }
```

## 8.6 Multi-Dimensional Arrays

C++ allows creation of arrays with 2 or more dimensions. Below is an example of declaring, initializing and printing a 2-dimensional array of numbers.

Code 8.10: Example 2D Array

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5 const int columns = 6;
6 const int rows = 4;
7 int my2DArray[rows][columns];
8
9 for (int i=0;i<rows;i++){
10 for(int j=0; j<columns;j++)
11 my2DArray[i][j] = i+j;
12 }
13
14 for (int i=0;i<rows;i++){
15 for(int j=0; j<columns;j++)
16 cout << my2DArray[i][j] << " ";
17 cout << endl;
18 }
19 return 0;
20 }
```

Code 8.11: Another 2D Array Example

```
1 #include <iostream>
2 using namespace std;
3
4 void printArray(const int [][][3]);
5 const int rows = 2;
6 const int columns = 3;
7
8 int main(){
9 int array1[rows][columns] = {{1,2,3},{4,5,6}};
10 int array2[rows][columns] = {1,2,3,4,5};
11 int array3[rows][columns] = {{1,2},{4}};
12
13 cout << "Values in array1 are:" << endl;
14 printArray(array1);
15
16 cout << "Values in array2 are:" << endl;
17 printArray(array2);
18
19 cout << "Values in array3 are:" << endl;
20 printArray(array3);
21
22 return 0;
23 }
24
25 void printArray(const int a[][columns]){
26 for (int i=0;i<rows;i++){
27 for(int j=0; j<columns;j++){
28 cout << a[i][j] << " ";
29 cout << endl;
30 }
31 }
```



← Get code here.

## 8.7 Practice Questions

1. What is the meaning of the following in the context of Computer Science?

(a) array index:

---

---

(b) array length / size:

---

---

2. True-False questions – Circle your choice

(a) (True - **False**) The statement `array[5][6]` reads the value of the array element in row 5 and column 6.

(b) (True - **False**) In C++ all arrays are passed as arguments to a function by value.

(c) (**True** - False) In C++ all arrays are passed as arguments to a function by reference.

(d) (**True** - False) In C++, arrays can only store elements of the same data type.

(e) (**True** - False) C++ arrays are always zero-indexed, meaning the first element is accessed using the index 0.

(f) (True - **False**) C++ allows arrays of variable size, where the size is determined at runtime.

(g) (**True** - False) The size of a C++ array can be determined using the `sizeof()` operator.

(h) (True - **False**) C++ arrays can have negative indices.

(i) (**True** - False) C++ arrays can be initialized with a list of values using braces `()` notation.

(j) (True - **False**) C++ arrays can have a size of zero.

(k) (**True** - False) C++ arrays can be passed as arguments to functions.

(l) (**True** - False) C++ arrays can be multidimensional, allowing for the storage of matrices and tables.

(m) (**True** - False) The size of a C++ array can be determined by dividing the total number

of bytes by the size of each element.

- (n) (**True - False**) C++ arrays can be directly compared using the greater than (>) or less than (<) operators.
- (o) (**True** - False) C++ arrays can have strings as elements.
- (p) (**True** - False) The subscript operator ([]) can be used to access elements of a C++ array.
- (q) (**True** - False) C++ arrays can have a maximum size limit imposed by the compiler or the system.

3. Fill in the blanks.

- (a) The number used to refer to a specific element in an array is called its \_\_\_\_\_.
- (b) A 5 x 3 array contains \_\_\_\_\_ rows, \_\_\_\_\_ columns and \_\_\_\_\_ elements.

4. Given the array declaration, `int a[20]`; The first element of this array is written as:

- (a) `a[1]`
- (b) `a[0]`
- (c) `a`
- (d) `a[19]`
- (e) `a[20]`

5. Fix the errors in the code to obtain the desired output.

```
1 #include <iostream>
2 using namespace std;
3
4 void modifyArray(int [], int);
5
6 int main() {
7
8 const int arraySize = 5;
9 int myArray[arraySize] = {3,4,6,2,8,9};
10
11 modifyArray(myArray, arraySize);
12
13 return 0;
14 } // end main
15
16 void modifyArray(const int a[]){
17 for (int i=0; i<sizeofArray; i++){
18 a[i]++;
19 cout << a[i] << endl;
20 }
21 }
```

**Desired output:**

```
4
5
7
3
9
10
```

6. Fix the errors in the following code.

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5
6 char weekDays[]= {"Monday", "Tuesday", "Wednesday",
7 "Thursday", "Friday", "Saturday", "Sunday"};
8
9 for(int i=0; i<7;i++){
10 cout << weekDays[i] << endl;
11 }
12 return 0;
13 }
```

7. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 void printArray(const int [][][3]);
5 const int rows = 2;
6 const int columns = 3;
7
8 int main(){
9 int array1[rows][columns] = {{1,2,3},{4,5,6}};
10 int array2[rows][columns] = {1,2,3,4,5};
11 int array3[rows][columns] = {{1,2},{4}};
12
13 cout << "Values in array1 are:" << endl;
14 printArray(array1);
15
16 cout << "Values in array2 are:" << endl;
17 printArray(array2);
18
19 cout << "Values in array3 are:" << endl;
20 printArray(array3);
21
22 return 0;
23 }
24
25 void printArray(const int a[][columns]){
26 for (int i=0;i<rows;i++){
27 for(int j=0; j<columns;j++){
28 cout << a[i][j] << " ";
29 cout << endl;
30 }
31 }
```

8. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5
6 int myArray[6] = {3,4,6,2,4,7};
7
8 cout << sizeof(myArray) << endl;
9 cout << sizeof(myArray[0]) << endl;
10
11 return 0;
12 }
```

9. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5 const int columns = 4;
6 const int rows = 3;
7 int my2DArray[rows][columns];
8
9 for (int i=0;i<rows;i++){
10 for(int j=0; j<columns;j++)
11 my2DArray[i][j] = 6-i+j;
12 }
13
14 for (int i=0;i<rows;i++){
15 for(int j=0; j<columns;j++)
16 cout << my2DArray[i][j] << " ";
17 cout << endl;
18 }
19 return 0;
20 }
```



10. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5
6 int my2DArray[3][3] = {{1,2,3},{4,5,6},{7,8,9}};
7
8 cout << my2DArray[1][0] << endl;
9 cout << my2DArray[2][1] << endl;
10 cout << my2DArray[1][2] << endl;
11
12 return 0;
13 }
```

11. Make only one change to the program below so that the current output changes to the desired output.

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5
6 char *weekDays[7]= {"Monday", "Tuesday",
7 "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"};
8
9 for(int i=0; i<7;i++){
10 cout << weekDays[i] << endl;
11 }
12
13 return 0;
14 }
```

**Current output:**

Monday  
Tuesday  
Wednesday  
Thursday  
Friday  
Saturday  
Sunday

**Desired output:**

M  
T  
W  
T  
F  
S  
S

12. Make only one change to the program below so that the current output changes to the desired output.

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5 char *weekDays[7]= {"Monday", "Tuesday",
6 "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"};
7
8 for(int i=0; i<7;i++){
9 cout << weekDays[0][i] << endl;
10 }
11
12 return 0;
13 }
```

**Current output:**

M  
o  
n  
d  
a  
y

**Desired output:**

Monday  
onday  
nday  
day  
ay  
y

13. Complete the program below so that it prints the content of My2Darray as shown in the output column. Use a nested for loop to print the values. Use the tab feature to print the values in distinct columns as shown.

```
1 ##include <iostream>
2 using namespace std;
3
4 int main(){
5 const int columns = 3;
6 const int rows = 4;
7 int my2DArray[rows][columns]= {1,2,3,4,5,6,7,8,9,10,11,12};
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24 return 0;
25 }
```

**Desired output:**

|    |    |    |
|----|----|----|
| 1  | 2  | 3  |
| 4  | 5  | 6  |
| 7  | 8  | 9  |
| 10 | 11 | 12 |

14. Assume we want to store 5 double data values for further processing (example find their average). Write a program that allows the user to input 5 numerical values and uses an array to store the data values and calculate the average of all these values.

**Desired output:**

Enter value:

5

6

8

2.3

3.2

The Average is: 4.9

15. Write a function definition `LinearSearch(...)` to perform a linear search of an array. The function should receive an integer array and the size of the array as arguments, and the key. If the search key is found, return the array index; otherwise, return -1. Make sure you use the proper function return type and argument list. Just write the function definition. You do not need to write a complete C++ program.
16. Write the function `void rotateRight (int n, char a[])` which moves each of the `n` characters one position to the right, but moves the last character to the first position. For example, if `a` contained `n = 4` characters, `ABCD`, then when the function is finished, it would contain `DABC`. Make sure you use the proper function return type and argument list. Just write the function definition. You do not need to write a complete C++ program.
17. Write a function definition `largestIndex(...)` that returns the index of the largest element of a global integer array `myArray` of length 100. Make sure you use the proper function return type and argument list. Just write the function definition. You do not need to write a complete C++ program.
18. Write a function `averageValue(...)` that returns the average of all the elements in a global array `myArray` of double values. (hint: the number of elements in array `a` must be passed to the function.) Make sure you use the proper function return type and argument list. Just write the function definition. You do not need to write a complete C++ program.



## 9. Pointers

### 9.1 What are Pointers?

Pointers are a powerful feature of C++. They allow the user to pass-by-reference which in turn allows manipulation of more complex, dynamic data structures. Pointers will be extensively used in later more advanced topics.

Code 9.1: Pointers first example

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5
6 int x = 5;
7 int *xPointer = &x;
8
9 cout << "Address of x: " << &x << endl;
10 cout << "Value of xPointer: " << xPointer << endl << endl;
11
12 cout << "Address of xPointer: " << &xPointer << endl << endl;
13
14 cout << "Value of x: " << x << endl;
15 cout << "Value of *xPointer: " << *xPointer << endl;
16
17 return 0;
18 }
```



← Get code here.

Code 9.2: Another example using pointers

```
1 #include <iostream>
2 using namespace std;
3
4 void doubleValue(int *p);
5
6 int main() {
7
8 int myInt = 5;
9 cout << "myInt before doubling: " << myInt << endl;
10 doubleValue(&myInt);
11 cout << "myInt after doubling: " << myInt << endl;
12
13 return 0;
14 } // end main()
15
16 void doubleValue(int *p) {
17 *p = *p * 2;
18 } //end doubleValue()
```

Write a C++ program that swaps the values of two integer variables a and b. Use a function `swap1(*p1, *p2)`. Print a and b before and after swapping. Your output should have the same format and should work for any integer a user enters. Use pointers.

**Desired output format:**

The value of a before swapping is: 4  
The value of b before swapping is: 7

The value of a after swapping is: 7  
The value of b after swapping is: 4

## 9.2 Arrays and Pointer Arithmetic

Arrays store values in consecutive memory locations. Pointer can be very useful when processing arrays. The example below illustrates some of the array properties and the use of pointers.



Code 9.3: Arrays and pointers example

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5
6 const int arraySize = 5;
7 int myArray[arraySize] = {2, 4, 7, 9, 1};
8 int *p = myArray;
9
10 cout << "Index \tValue \tAddress" << endl << endl;
11
12 for (int i=0; i< arraySize; i++){
13 cout << i << "\t";
14 cout << myArray[i] << "\t";
15 cout << &myArray[i] << "\t" << endl;
16 }
17 cout << endl;
18
19 cout << "p is pointing at address: " << p << endl;
20 cout << "The value of *p is: " << *p << endl << endl;
21
22 cout << "Moving p to the next memory address." << endl;
23 p = p+1;
24
25 cout << "p is pointing at address: " << p << endl;
26 cout << "The value of *p is: " << *p << endl;
27
28 return 0;
29 }
```



← Get code here.



Change myArray from an array of integers to a double array. Run the program, compare any two consecutive addresses. What do you notice?



Change line 8 from **int \*p = myArray;** to **int \*p = &myArray[0];**. Is there any difference in the output? Why?

Code 9.4: Using pointers to work with strings

```
1 #include <iostream>
2 using namespace std;
3
4
5 char *myStrings[4] = {"one", "two", "three", "four"};
6
7 int main() {
8
9 cout << "Printing the array content." << endl;
10 for (int i=0; i<4; i++)
11 cout << myStrings[i] << endl;
12 cout << endl;
13
14 cout << "Printing the address of each string." << endl;
15 for (int i=0; i<4; i++)
16 cout << &myStrings[i] << endl;
17 cout << endl;
18
19 cout << "Printing substrings of the third string." << endl;
20 for (int i=0; i<5; i++)
21 cout << &myStrings[2][i] << endl;
22 cout << endl;
23
24 cout << "Printing the value where pointers are." << endl;
25 for (int i=0; i<4; i++)
26 cout << *myStrings[i] << endl;
27 cout << endl;
28
29 return 0;
30 }
```



← Get code here.

### 9.3 Practice Questions

1. True-False questions – Circle your choice
  - (a) Pointers can only store the memory address of a variable. (**True/False**)
  - (b) It is not necessary to initialize a pointer variable before using it. (**True/False**)
  - (c) Pointers can be used to dynamically allocate memory at runtime. (**True/False**)
  - (d) The dereference operator (\*) is used to access the value stored in a pointer. (**True/False**)
  - (e) A null pointer is a pointer that does not point to any valid memory location. (**True/False**)
2. What is a pointer in C++?
  - (a) A variable that stores the address of another variable
  - (b) A variable that stores a constant value
  - (c) A variable that stores a string of characters
  - (d) A variable that stores the result of an arithmetic operation
3. How do you declare a pointer in C++?
  - (a) By using the 'var' keyword
  - (b) By using the 'ptr' keyword
  - (c) By using the 'point' keyword
  - (d) By using the '\*\*' symbol before the variable name
4. What does the '&' operator do when used with a variable?
  - (a) It returns the memory address of the variable
  - (b) It performs a bitwise AND operation
  - (c) It accesses the value pointed by a pointer
  - (d) It dereferences a pointer
5. How do you assign the address of a variable to a pointer?
  - (a) By using the '=' operator
  - (b) By using the '==' operator
  - (c) By using the '&' operator
  - (d) By using the '\*\*' operator
6. What does it mean to dereference a pointer?
  - (a) It assigns the value of a pointer to another pointer
  - (b) It accesses the address stored in a pointer
  - (c) It accesses the value stored at the address pointed by a pointer
  - (d) It performs a logical NOT operation on a pointer
7. What is the size of a pointer in C++?
  - (a) It depends on the size of the variable it points to
  - (b) 4 bytes on a 32-bit system and 8 bytes on a 64-bit system
  - (c) 2 bytes on all systems
  - (d) It is implementation-dependent

8. What is a null pointer?
- (a) A pointer that points to the address 0
  - (b) A pointer that is uninitialized
  - (c) A pointer that points to a variable with a value of zero
  - (d) A pointer that points to the end of an array
9. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5 int* ptr = nullptr;
6 cout << ptr << endl;
7 return 0;
8 }
```

10. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 void changeValue(int* ptr) {
5 *ptr = 20;
6 }
7
8 int main() {
9 int x = 10;
10 int* ptr = &x;
11 changeValue(ptr);
12 cout << x << endl;
13 return 0;
14 }
```

11. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5 int x = 10;
6 int* ptr = &x;
7 cout << *ptr << endl;
8 return 0;
9 }
```

12. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5 int arr[] = {1, 2, 3, 4, 5};
6 int* ptr = arr;
7 cout << *(ptr + 2) << endl;
8 return 0;
9 }
```

13. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5 int x = 5;
6 int* ptr = &x;
7 *ptr = 7;
8 cout << x << endl;
9 return 0;
10 }
```



## 10. Structures

### 10.1 What are Structures?

We have used variables that allow us to store information in the computer. Variable types such as `int`, `char`, etc. allow us to save information of a single type. Structures allow us to organize information as collections of values that belong to the same entity and are connected to one another in a meaningful way. Understanding structures is a helpful step to better understand classes and objects which we will discuss in the next chapter.

Code 10.1: Structure example

```
1 #include <iostream>
2 using namespace std;
3
4 struct Course {
5 int courseNumber;
6 char grade;
7 };
8
9 int main() {
10
11 Course myCourse;
12 myCourse.courseNumber = 101;
13 myCourse.grade = 'A';
14
15 cout << "I took MAC " << myCourse.courseNumber;
16 cout << " and I got a(n) " << myCourse.grade << endl;
17
18 return 0;
19 }
```



Add a new structure to the program above named Exam, which has two member variables, examNumber and grade. Assign values from main() to both variables, then print the results similar to what we did with the Course structure. Is it working? Is there a problem using the same name (grade) for both structures?

Code 10.2: Alternative way of initializing structures

```
1 #include <iostream>
2 using namespace std;
3
4 struct Course {
5 int courseNumber;
6 char grade;
7 };
8
9 int main(){
10
11 Course myCourse = {101, 'A'};
12
13 cout << "I took MAC " << myCourse.courseNumber;
14 cout << " and I got a(n) " << myCourse.grade << endl;
15
16 return 0;
17 }
```



Swap the values 101 and 'A'. Does it work? What do you notice?



## 10.2 Structure Composition

It is possible to have a structure be a member of another structure.

Code 10.3: Structure composition example

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 struct Course {
6 int courseNumber;
7 char grade;
8 };
9
10 struct Student {
11 string studentName;
12 Course studentCourse;
13 };
14
15 int main() {
16
17 Student st1;
18 st1.studentName = "Bob";
19 st1.studentCourse.courseNumber = 101;
20 st1.studentCourse.grade = 'A';
21
22 cout << st1.studentName;
23 cout << " took MAC " << st1.studentCourse.courseNumber;
24 cout << " and got a(n) " << st1.studentCourse.grade << endl;
25
26 return 0;
27 }
```



← Get code here.

### 10.3 Arrays of Structures

Multiple structures of the same type can be stored using an array.

Code 10.4: Example storing structures in an array

```
1 #include <iostream>
2 using namespace std;
3
4 struct Course {
5 int courseNumber;
6 char grade;
7 };
8
9 int main() {
10 const int numberOfCourses = 4;
11 Course courseList[numberOfCourses] = {
12 {101, 'A'},
13 {125, 'B'},
14 {190, 'C'},
15 {286, 'A'}
16 };
17
18 cout << "Course list:" << endl;
19 for(int i=0; i<numberOfCourses; i++){
20 cout << courseList[i].courseNumber;
21 cout << " " << courseList[i].grade << endl;
22 }
23
24 return 0;
25 }
```



← Get code here.



What are the similarities and differences between structures and arrays?

## 10.4 Passing Structures to Functions

Structures can also serve as arguments for function parameters, much like any primitive data types. By default structures are passed by value.

Code 10.5: Passing a structure to function example

```
1 #include <iostream>
2 using namespace std;
3
4 struct Course {
5 int courseNumber;
6 char grade;
7 };
8
9 void printCourse(Course);
10
11 int main() {
12 const int numberOfCourses = 4;
13 Course courseList[numberOfCourses] = {
14 {101, 'A'},
15 {125, 'B'},
16 {190, 'C'},
17 {286, 'A'}
18 };
19
20 cout << "Course list:" << endl;
21 for(int i=0; i<numberOfCourses; i++)
22 printCourse(courseList[i]);
23
24 return 0;
25 }
26
27 void printCourse(Course c){
28 cout << c.courseNumber;
29 cout << " " << c.grade << endl;
30 }
```



1. Change the previous program to pass each structure to the function by reference. The program should produce the same output.
2. Similarly change the program to give the function a pointer to each structure.

## 10.5 Practice Questions

1. True-False questions – Circle your choice
  - (a) (**True** - False) In C++, a structure is a user-defined data type that allows the storage of different types of data under a single name.
  - (b) (**True** - False) In C++, a structure can be passed as a parameter to a function by value or by reference.
  - (c) (**True** - False) C++ structures can be used to create objects similar to classes, but with some differences in terms of default access specifiers and member visibility.
  - (d) (**True** - False) C++ structures can be used to create collections of related data.
  - (e) (**True** - False) C++ structures can have member variables of different data types, including other structures.
  - (f) (**True** - False) C++ structures can be used to define data structures for efficient storage and retrieval of data.
  - (g) (**True** - False) C++ structures can have member functions that access and modify the values of member variables in other structures.
  - (h) (**True** - False) C++ structures can be used to organize related data and operations into a single unit.
2. What is the purpose of structures in C++? How do they differ from variables of primitive types?

3. Consider the following structure definition:

```
struct Person {
 string name;
 int age;
 char gender;
};
```

Write a program that creates an instance of the Person structure, assigns values to its members, and prints the details of the person.

4. Can a structure contain another structure as one of its members? Explain with an example.
5. What is the difference between passing a structure by value, by reference, and by pointer to a function? When would you choose one method over the others?

- 
6. Write a program that uses an array of structures to store information about books. Each structure should have members for the title, author, and year of publication. Prompt the user to enter details for multiple books and then display the information for each book.
  7. Explain the similarities and differences between structures and arrays in C++.
  8. How can you initialize a structure when declaring it? Provide an example.
  9. Create a program that declares a structure named `Rectangle` with members for length and width. Write a function that takes a `Rectangle` as a parameter and calculates its area. Test your program by creating multiple instances of `Rectangle` and calculating their areas.



## 11. Introduction to Classes and Objects

### 11.1 Objects

C++ is an object-oriented programming language that provides a powerful paradigm for developing software applications. With its support for OOP, C++ enables programmers to create custom data types that closely mirror real-life scenarios, making the code more intuitive, organized, and reusable. This approach to programming results in software that is easy to understand, maintain, and extend, and ultimately results in higher-quality applications.



What is an object?



What sets two objects apart?



Do different objects have anything in common?

## 11.2 Classes

Until now for the most part we have used primitive data types such as **int** or **char** to process information. Following the OOP principles, a programmer can specify his/her own type based on what objects he/she intends to work with.

To enable creation of such objects a programmer needs to identify common characteristics and behavior of these objects and create a **class** using this information.

A class is a **blueprint** (template) which is used when we create an object. A class describes collectively characteristics and behavior of the same **type** of objects.

We can create multiple objects from the same class. Each object is unique but they all share common features of all other objects derived from the same class.

An object created from a class is referred to as an **instance** of the class. The process of creating an object is called **instantiation**.

## 11.3 The First Example

Code 11.1: First example of a class

```
1 #include <iostream>
2 using namespace std;
3
4 class Rectangle{
5 public:
6 int width, length;
7 }; //end Rectangle class
8
9 int main() {
10 Rectangle r1;
11 r1.width=3;
12 r1.length=5;
13 cout << " r1 is a " << r1.width;
14 cout << "x"<<r1.length <<" rectangle."<< endl;
15
16 return 0;
17 } // end main()
```



Remove the public keyword and run the program. What do you notice?



What happens if we do not initialize class members?



Code 11.2: Adding class member functions

```
1 #include <iostream>
2 using namespace std;
3
4 class Rectangle{
5 public:
6 int width, length;
7 void printArea(){
8 cout << "Area = " << width*length << endl;
9 }
10 }; //end Rectangle class
11
12 int main() {
13 Rectangle r1;
14 r1.width=3;
15 r1.length=5;
16 r1.printArea();
17 return 0;
18 } // end main()
```

Code 11.3: Access specifiers, data hiding and encapsulation

```
1 #include <iostream>
2 using namespace std;
3
4 class Rectangle{
5 private:
6 int width, length;
7 public:
8 void set(int w, int l) { width = w; length = l;}
9 int getWidth() {return width;}
10 int getLength() {return length;}
11
12 void printArea(){
13 cout << "Area = " << width*length << endl;
14 }
15 }; //end Rectangle class
16
17 int main(){
18 Rectangle r1;
19 r1.set(4,7);
20 cout << "Dimensions: " << r1.getWidth();
21 cout << "x" << r1.getLength() << endl;
22 r1.printArea();
23 return 0;
24 }
```

## 11.4 Constructors

Code 11.4: Adding constructors

```
1 #include <iostream>
2 using namespace std;
3
4 class Rectangle {
5 private:
6 int width, length;
7 public:
8 Rectangle() {
9 width=1; length=2;
10 cout << "Hello world!!!" << endl;
11 }
12 void set(int w, int l) { width = w; length = l;}
13 int getWidth() {return width;}
14 int getLength() {return length;}
15
16 void printArea(){
17 cout << "Area = " << width*length << endl;
18 }
19 }; //end Rectangle class
20
21 int main() {
22 Rectangle r1;
23 r1.printArea();
24 return 0;
25 }
```



Add another constructor that takes two parameters and assigns those values to width and length.



What can we conclude regarding constructor overloading in C++?



Can we call the constructor explicitly, say: ***r1.Rectangle(3,5);***



Always include a default constructor.

## 11.5 Destructors

Code 11.5: Adding destructors

```
1 #include <iostream>
2 using namespace std;
3
4 class Rectangle {
5 private:
6 int width, length;
7 public:
8 Rectangle() {
9 width=1; length=2;
10 cout << "Hello world!!!" <<endl;
11 }
12 ~Rectangle(){
13 cout << "Goodbye everyone!!!" << endl;
14 }
15 void set(int w, int l) { width = w; length = l;}
16 int getWidth() {return width;}
17 int getLength() {return length;}
18
19 void printArea(){
20 cout << "Area = " << width*length << endl;
21 }
22 }; //end Rectangle class
23
24 int main() {
25 Rectangle r1;
26 r1.printArea();
27 return 0;
28 }
```

## 11.6 Separating Content in Different Files

It is common practice to separate the declaration and definition of each class into separate files. First comes the header file **ClassName.h** then the source / implementation file **ClassName.cpp**. Next, we create separate files to achieve the results from the previous example.

Code 11.6: Rectangle.h

```
1 #ifndef RECTANGLE_H_
2 #define RECTANGLE_H_
3
4 class Rectangle{
5 private:
6 int width, length;
7 protected:
8 public:
9 Rectangle();
10 Rectangle(int w, int l);
11 ~Rectangle();
12 void setWidth(int);
13 void setLength(int);
14 int getWidth();
15 int getLength();
16 void printArea();
17 };
18
19 #endif /* RECTANGLE_H_ */
```

Code 11.7: Rectangle.cpp

```
1 #include "Rectangle.h"
2 #include <iostream>
3 using namespace std;
4
5 Rectangle::Rectangle(){ width = 1; length = 2;}
6
7 Rectangle::Rectangle(int w, int l){ width = w; length = l;}
8
9 Rectangle::~Rectangle(){}
10
11 void Rectangle::setWidth(int w){ width = w;}
12
13 void Rectangle::setLength(int l){ length = l;}
14
15 int Rectangle::getWidth(){ return width;}
16
17 int Rectangle::getLength(){ return length;}
18
19 void Rectangle::printArea(){ cout << "Area = "<< width*length;}
```

Code 11.8: Main program

```
1 #include "Rectangle.h"
2 #include <iostream>
3 using namespace std;
4
5 int main() {
6 Rectangle r1;
7 r1.printArea();
8 return 0;
9 }
```



Write a class named **Person** that has data members **name** and **age**. Include a default and a parameterized constructor, a destructor and a public member function named **sayHello()** that that prints “Hello! I am “, followed by the name of the person on the screen.

## 11.7 Constant Objects

Similar to primitive data types we can set objects as constant to prevent any changes to the object during the execution of a program.

Code 11.9: Rectangle class example

```

1 #include <iostream>
2 using namespace std;
3 /***** Declaration Section *****/
4 class Rectangle{
5 private:
6 int width , length;
7 public:
8 Rectangle();
9 Rectangle(int w, int l);
10 ~Rectangle();
11 void setWidth(int);
12 void setLength(int);
13 int getWidth();
14 int getLength();
15 void printArea();
16 };
17 /***** Implementation Section *****/
18 Rectangle::Rectangle(){ width = 1; length = 2;}
19 Rectangle::Rectangle(int w, int l){ width = w; length = l;}
20 Rectangle::~~Rectangle(){}
21 void Rectangle::setWidth(int w){ width = w;}
22 void Rectangle::setLength(int l){ length = l;}
23 int Rectangle::getWidth(){ return width;}
24 int Rectangle::getLength(){ return length;}
25 void Rectangle::printArea(){ cout << "Area = " << width*length;}
26 /***** Main Section *****/
27 int main(){
28 Rectangle r1;
29 r1.printArea();
30 return 0;
31 }

```



← Get code here.



Consider the example above. Make the following changes and run the program.

1. Declare Rectangle r1 as constant. What do you observe?
2. Change the printArea() function to constant.
3. Change the Rectangle r1 back to non-constant. What can you conclude?

## 11.8 Member Initializers

C++ offers an alternative syntax for initializing member variable via the constructor. Below is an example using both alternatives. Consider the program below:

Code 11.10: Class example.

```

1 #include <iostream>
2 using namespace std;
3 /***** Declaration Section *****/
4 class Rectangle{
5 private:
6 int width, length;
7 public:
8 Rectangle();
9 Rectangle(int w, int l);
10 void printArea();
11 };
12 /***** Implementation Section *****/
13 Rectangle::Rectangle(){
14 width = 1, length = 2;
15 cout << "Hello there! My id is " << this << endl;
16 }
17 Rectangle::Rectangle(int w, int l){
18 width = w, length = l;
19 cout << "Hello there! My id is " << this << endl;
20 }
21 void Rectangle::printArea() {
22 cout << "Area of " << this << " = " << width*length << endl;
23 }
24 /***** Main Section *****/
25 int main(){
26 Rectangle r1, r2(5,7);
27 r1.printArea();
28 r2.printArea();
29 return 0;
30 }

```



← Get code here.



Change the implementation section using the code below:

Code 11.11: Using initializers code

```

1 /***** Implementation Section *****/
2 Rectangle::Rectangle() :width(1), length(2) {
3 cout << "Hello there! My id is " << this << endl;
4 }
5 Rectangle::Rectangle(int w, int l) :width(w), length(l) {
6 cout << "Hello there! My id is " << this << endl;
7 }
8 void Rectangle::printArea() {
9 cout << "Area of " << this << " = " << width*length << endl;
10 }

```

## 11.9 Class Composition

A class can have as a member an object from another class.

Code 11.12: Class composition example

```

1 #include <iostream>
2 #include <string>
3 using namespace std;
4 /***** Birthday Class *****/
5 class Birthday {
6 private:
7 int month, day, year;
8 public:
9 Birthday(int m, int d, int y)
10 : month(m), day(d), year(y) {}
11
12 void printDate(){
13 cout<<month <<"/" <<day <<"/" <<year <<endl;
14 }
15 };
16 /***** Person Class *****/
17 class Person {
18 private:
19 string name;
20 Birthday dateOfBirth;
21 public:
22 Person(string n, Birthday b)
23 : name(n), dateOfBirth(b){}
24
25 void printInfo(){
26 cout << name << " was born on: ";
27 dateOfBirth.printDate();
28 }
29 };
30 /***** Main *****/
31 int main() {
32 Birthday bDate(12,11,1882);
33 Person peopleObj("Fiorello H. LaGuardia", bDate);
34 peopleObj.printInfo();
35 }

```



← Get code here.



1. Create a Major class with two private attributes, major name, and number of credits and a public member function **displayMajorInfo()**.
2. Create a Student class with private attributes, student name, age major and a public member function **displayStudentInfo()**.



## 11.10 Practice Questions

### 1. True-False questions – Circle your choice

- (a) (**True** - False) In C++, a class is a user-defined data type that can have member functions and member variables.
- (b) (True - **False**) C++ classes can have static member variables that are shared among all instances of the class.
- (c) (**True** - False) C++ classes can be used to define objects, which are instances of the class.
- (d) (**True** - False) C++ classes can have member functions with default arguments.
- (e) (True - **False**) C++ classes cannot have member variables with different access specifiers, such as private, public, and protected.
- (f) (**True** - False) C++ classes can have constructors and destructors.
- (g) (True - **False**) C++ objects cannot have member functions.
- (h) (**True** - False) In C++, objects of the same class share the same member functions, but each object has its own copy of member variables.
- (i) (**True** - False) C++ classes can have constant member functions that cannot modify the member variables of the class.
- (j) (**True** - False) C++ classes can have multiple constructors with different parameter lists.
- (k) (**True** - False) C++ objects can be created without using the "new" keyword.
- (l) (**True** - False) C++ objects can be used to access member functions of a class.
- (m) (**True** - False) C++ classes can have member variables with different data types, including other classes.
- (n) (True - **False**) C++ classes can be passed as parameters to functions by value or by reference.
- (o) (**True** - False) C++ classes can have member variables with the same name as variables defined outside the class, without causing conflicts.

2. What is the purpose of classes and objects in C++? How do they differ from structures and variables of primitive types?

3. Consider the following class definition:

```
class Person {
public:
 string name;
 int age;
 char gender;
};
```

Write a program that creates an instance of the `Person` class, assigns values to its members, and prints the details of the person.

4. Can a class contain instances of another class as one of its members? Explain with an example.
5. Write a program that uses an array of objects to store information about books. Each object should belong to a class with members for the title, author, and year of publication. Prompt the user to enter details for multiple books and then display the information for each book.
6. Explain the similarities and differences between classes and structures in C++.
7. In what scenarios would you choose to use a class instead of a structure?
8. Create a class named `Circle` with a member variable for the radius and member functions to calculate its area and circumference. Test your class by creating multiple instances of `Circle` and calculating their areas and circumferences.

9. Write a class named **Product** with data members for **name**, **price**, and **quantity**, and public member functions *calculateTotalPrice()* and *displayInfo()* to calculate the total price of the product (price \* quantity) and display its information respectively. Include the following components:
- Appropriate access specifiers
  - A default constructor
  - A parameterized constructor
  - Separate declaration and implementation sections
  - Accessor and mutator functions for name, price, and quantity
  - Public functions *calculateTotalPrice()* and *displayInfo()*
  - Include a *main()* function, create a Product object using the default constructor, prompt the user to enter the product details, calculate its total price, and display the product's information
10. Develop a class called **Car** to represent information about a car. The class should have data members for **make** (string), **year** (int), **mileage** (float), and **isElectric** (bool). Include the following components:
- Appropriate access specifiers
  - A default constructor that initializes the make and year to empty string and 0 respectively, mileage to 0.0, and isElectric to false.
  - A parameterized constructor that allows you to set all the data members with provided values.
  - Separate declaration and implementation sections for organized code.
  - Accessor and mutator functions for make, year, mileage, and isElectric.
  - A public function *displayInfo()* that prints the car's make, year, mileage, and whether it is an electric car or not, on the screen.
  - Include a *main()* function where you create a **Car** object using the default constructor, prompt the user to enter the car details, update the electric status based on isElectric, and display the car's information.



## 12. Strings

### 12.1 How C++ Stores Text?

All information in the computer is stored using 0s and 1s. Therefore, all data is stored as a number. When it comes to text, each character is stored using the corresponding ASCII number. Each character gets one byte of memory space.



How many different characters can be represented with one byte of memory space?

Code 12.1: Example displaying the ASCII value of a char variable.

```
1 #include <iostream>
2 #include <bitset>
3 #include <iomanip>
4 using namespace std;
5
6 int main()
7 {
8 char c;
9
10 cout << "Enter any character: ";
11 cin >> c;
12 cout << "The character you entered is: " << c << endl;
13 cout << "Its ASCII code (decimal) is: " << (int)c << endl;
14 bitset<8> x(c);
15 cout << "Its ASCII code (binary) is: " << x << endl;
16 cout << "Its ASCII code (hexadecimal) is: ";
17 cout << setw(2) << setfill('0') << hex << (int)c << endl;
18
19 return 0;
20 }
```

Code 12.2: Example converting ascii value to char.

```
1 #include <iostream>
2 #include <bitset>
3 #include <iomanip>
4 using namespace std;
5
6 int main(){
7 int ascii;
8
9 cout << "Enter an integer between 0 and 255: ";
10 cin >> ascii;
11 cout << "The integer you entered is: " << ascii << endl;
12 cout << "The ASCII character is: " << (char) ascii << endl;
13
14 return 0;
15 }
```

Write a program that uses a for loop to print all the ASCII characters and their corresponding decimal values in tabular format. See the sample output for some of the characters below:

**Desired output format:**

```
...
...
35 #
36 $
37 %
...
...
```

## 12.2 C-style String Functions

C++ has a number of pre-defined functions inherited from C that help the programmer manipulate arrays of characters or strings. The example below illustrates some of these functions.

Code 12.3: Example using cstring functions.

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5 char s[80];
6 strcpy(s, "One");
7 strcat(s, "Two");
8 strcat(s, "Three ");
9 cout << "The string created is: " << s << endl;
10 cout << "The length of the string is: " << strlen(s);
11
12 return 0;
13 }
```



There is something missing in the above program which may prevent the program from running. Can you fix it?

Code 12.4: Reading string input example.

```
1 #include <iostream>
2 #include <cstring>
3 using namespace std;
4
5 int main() {
6 char s[80];
7 cout << "Enter string: ";
8 cin >> s;
9 cout << "The string entered is: " << s << endl;
10 cout << "The length of the string is: " << strlen(s);
11
12 return 0;
13 }
```



Enter a string consisting of more than one word. Does it work?



Substitute the line **cin >> s;** with the line **cin.getline(s,79);** Run the program and input the string “Hello World”. What changed?



Now input a long string of more than 80 characters. What do you notice?

### 12.3 The <string> Class

The <string> class makes it easier for programmers to manipulate strings. A string derived from this class is an object, but it has similarities to primitive data types such as int or char. We will discuss classes and objects in more detail in the upcoming chapters.



Code 12.5: Different ways to initialize a string object.

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 int main() {
6
7 string myString1;
8 string myString2 = "Hello MAC101 class!";
9 string myString3("Using Parenthesis");
10 string myString4(myString3);
11 string myString5(myString2, 6);
12 string myString6(myString2, 6, 6);
13
14 cout << "1: " << myString1 << endl;
15 cout << "2: " << myString2 << endl;
16 cout << "3: " << myString3 << endl;
17 cout << "4: " << myString4 << endl;
18 cout << "5: " << myString5 << endl;
19 cout << "6: " << myString6 << endl;
20
21 return 0;
22 }
```

Code 12.6: String input and concatenation example.

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 int main() {
6 string str, name, addr;
7
8 cout << "Enter name: ";
9 getline(cin, name);
10 cout << "Enter address: ";
11 getline(cin, addr);
12
13 str = "My name is " + name + ". " + "I live in " + addr + ".";
14 cout << str << endl;
15
16 return 0;
17 }
```

Code 12.7: Comparing strings example.

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 int main(){
6
7 string string1 = "one";
8 string string2("one");
9 string string3("five");
10 string string4("ten");
11
12 if(string1 == string2)
13 cout << "The strings are the same" << endl;
14 else
15 cout << "The strings are different" << endl;
16
17 if(!string1.compare(string2))
18 cout << "The strings are the same" << endl;
19 else
20 cout << "The strings are different" << endl;
21 cout << string1.compare(string2);
22
23 return 0;
24 }
```

Code 12.8: Working with strings additional example.

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 int main() {
6
7 string city = "New";
8 cout << "Initial string: " << city << endl;
9
10 city.append(" city!");
11 cout << "After appending: " << city << endl;
12
13 city[4]='C';
14 cout << "After char change: " << city << endl;
15
16 string s2 = "York ";
17 city.insert(4, s2);
18 cout << "After insert: " << city << endl;
19
20 city.erase(8, 5);
21 cout << "After erase: " << city << endl;
22
23 for (int i = 0; i < city.size(); i++){
24 for(int j=0;j<i;j++)
25 cout << " ";
26 cout << city[i] << endl;
27 }
28 return 0;
29 }
```

Write a C++ program that takes a string as an input and prints the string in reverse order. Use a loop for this program, do not use any built-in function to reverse the string.

**Desired output format:**

Enter a string: **Hello World**

The string you entered in reverse order is: dlroW olleH

Write a C++ program that takes as an input two strings from the user and determines whether the first string is a substring of the second. If yes, also print at what position the substring starts.

**Desired output format 1:**

Enter first string: Hello World

Enter second string: World

"World" is a substring of "Hello World".

It starts at position 6

**Desired output format 2:**

Enter first string: Hello World

Enter second string: apple

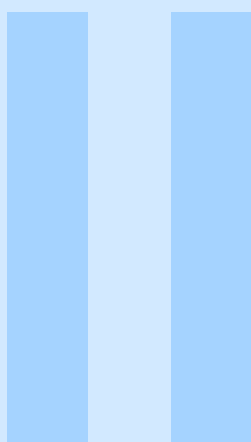
"apple" is NOT a substring of "Hello World".

## 12.4 Practice Questions

1. True-False questions – Circle your choice
  - (a) (**True** - False) In C++, a string is a built-in data type that can be used to store a sequence of characters.
  - (b) (True - **False**) C++ strings are mutable, meaning their contents can be changed after initialization.
  - (c) (**True** - False) C++ strings can only store ASCII characters and cannot handle Unicode characters.
  - (d) (**True** - False) In C++, a string literal is enclosed in double quotes ("").
  - (e) (True - **False**) C++ strings can be directly compared using the equality operator (==).
  - (f) (**True** - False) C++ strings can be concatenated using the "+" operator.
  - (g) (**True** - False) C++ strings can be treated as arrays of characters.
  - (h) (True - **False**) C++ strings can be passed as arguments to functions by value or by reference.
  - (i) (True - **False**) C++ strings can be directly compared using the greater than (>) or less than (<) operators.
  - (j) (**True** - False) C++ strings can have a fixed size, specified at the time of creation, that cannot be changed later.
  - (k) (**True** - False) C++ strings can be directly printed to the console using the cout object.
  - (l) (**True** - False) C++ strings can have a size of zero.
  - (m) (**True** - False) C++ strings can be directly compared using the "compare" method.
2. What is the difference between using the == operator and the compare() function to compare two strings?
3. What are some common C++ string member functions used for string manipulation?
4. Write a C++ program that takes a string as input and removes all the spaces from it, then prints the resulting string.
5. Write a C++ program that takes a string as input and counts the number of vowels (a, e, i, o,

u) in the given string.

6. Create a C++ program that takes a string as input and checks whether it is a palindrome (reads the same forwards and backwards) or not.
7. Create a C++ program that takes a string as input and counts the frequency of each character (case-insensitive) in the string, then prints the character and its corresponding count.



# Part Two - MAC 125

|    |                                     |     |
|----|-------------------------------------|-----|
| 13 | Classes, Pointers, and Arrays ..... | 177 |
| 14 | Recursion .....                     | 193 |
| 15 | Inheritance .....                   | 207 |
| 16 | Polymorphism .....                  | 225 |
| 17 | Templates .....                     | 239 |
| 18 | Linked Data Structures .....        | 253 |
| 19 | Exception Handling .....            | 269 |
| 20 | Standard Template Library .....     | 281 |





## 13. Classes, Pointers, and Arrays

### 13.1 Pointers Revisited

**IMPORTANT:** First please review the material from chapter 9 on page 135.

**Quick facts:**

- A pointer is a variable that holds the address of another variable.
- The operator & provides the address of a variable.
- We use \* when declaring a pointer. After declaration the operator \* is used to get the value from the variable the pointer is pointing to.

Code 13.1: Simple pointer example

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5
6 int a = 5;
7 int *aP = &a;
8 cout << "a = " << a << endl;
9 cout << "*aP = " << *aP << endl;
10
11 return 0;
12 }
```

## 13.2 Pointers and Some Special Operators

Code 13.2: Pointer example using some special operators

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5 // Declare and initialize pointers
6 int* one = new int; // Allocate memory for one integer
7 int* two = nullptr; // Initialize another pointer to nullptr
8
9 // Dereference and print the value of 'one' (uninitialized)
10 cout << "*one = " << *one << endl;
11
12 // Assign a value to the memory location pointed by 'one'
13 *one = 5;
14 cout << "*one = " << *one << endl;
15
16 *one = NULL;
17 cout << "*one = " << *one << endl;
18
19 (*one) += 1;
20 cout << "*one = " << *one << endl;
21
22 // Assign the address pointed by 'one' to 'two'
23 two = one;
24 cout << "*two = " << *two << endl;
25
26 delete one; // Deallocate memory
27 return 0;
28 }
```



← Get code here.

In modern C++, `nullptr` serves as the preferred method for representing a null pointer over the traditional `NULL` macro. While both `nullptr` and `NULL` are used to denote a pointer that does not point to any memory location, `nullptr` brings several advantages. Unlike `NULL`, which is typically defined as an integer zero, `nullptr` is a special keyword introduced in C++11 explicitly designed for null pointer representation. One key benefit of `nullptr` is its type safety. The use of `nullptr` enhances code readability and maintainability, aligning with modern C++ best practices. Therefore, in contemporary C++ development, it is recommended to adopt `nullptr` over `NULL` for clarity, type safety, and improved code quality.

### 13.3 Arrays and Pointers

In Chapter 9, we discussed examples of using pointers when working with arrays. Arrays in C++ are closely related to pointers. Here are some key concepts:

Code 13.3: Example of a dynamic array

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5 // Declaration and initialization of arrayOne
6 const int sizeOne = 5;
7 int arrayOne[sizeOne];
8
9 // Filling arrayOne with values
10 for (int i = 0; i < sizeOne; i++)
11 arrayOne[i] = i + 1;
12
13 // Printing the content of arrayOne
14 cout << "Content of arrayOne: ";
15 for (int i = 0; i < sizeOne; i++)
16 cout << arrayOne[i] << " ";
17 cout << endl;
18
19 // Declaration and initialization of arrayTwo dynamically
20 const int sizeTwo = 5;
21 int* arrayTwo = new int[sizeTwo];
22
23 // Filling arrayTwo with values
24 for (int i = 0; i < sizeTwo; i++)
25 arrayTwo[i] = i + 1;
26
27 // Printing the content of arrayTwo
28 cout << "Content of arrayTwo: ";
29 for (int i = 0; i < sizeTwo; i++)
30 cout << arrayTwo[i] << " ";
31 cout << endl;
32
33 // Freeing the dynamically allocated memory for arrayTwo
34 delete[] arrayTwo;
35
36 return 0;
37 }
```



← Get code here.

**Quick Facts:**

1. Array variables are essentially pointers. When we declare an array, we are also declaring a pointer to its first element.
2. When we pass an array to a function, we are actually passing a pointer to the first element of the array, not the entire array itself.



What are the differences between arrayOne and arrayTwo?



Can we remove the const keyword from sizeOne? Why?

### 13.4 The **this** Keyword

The special keyword **this** is used to allow an object to refer to itself, enabling access to its own address. This is particularly useful in member functions where there might be a need to disambiguate between member variables and function parameters.

Code 13.4: **this** pointer example

```

1 #include <iostream>
2 using namespace std;
3
4 /** ***** Class Declaration & Definition ***** */
5 class MyClass {
6 public:
7 // Constructor to initialize myInt
8 MyClass(int a) : myInt(a) {}
9
10 // Member function to print values
11 void printValues() {
12 // Printing myInt using different ways
13 cout << "myInt: " << myInt << endl;
14 // Direct access
15 cout << "this->myInt: " << this->myInt << endl;
16 // Using this pointer
17 cout << "(*this).myInt: " << (*this).myInt << endl;
18 }
19 private:
20 int myInt; // Member variable
21 };
22
23 /** ***** Main Section ***** */
24 int main() {
25 MyClass obj(100);
26 obj.printValues();
27 return 0;
28 }
```



← Get code here.



Explain the purpose of the **this** pointer in the context of the provided code example.



What is the significance of using the **this** pointer when accessing member variables in the `printValues` member function?

Code 13.5: Another example

```

1 #include <iostream>
2 using namespace std;
3
4 /***** Declaration Section *****/
5 class BankAccount{
6 private:
7 double balance;
8 public:
9 BankAccount();
10 BankAccount(double balance);
11 BankAccount& deposit(double money);
12 void withdraw(double money);
13 double getBalance();
14 };
15
16 /***** Implementation Section *****/
17 BankAccount::BankAccount(){ balance = 0.00;}
18 BankAccount::BankAccount(double money){ balance = money;}
19 BankAccount& BankAccount::deposit(double money){
20 balance += money;
21 return *this;
22 }
23 void BankAccount::withdraw(double money){ balance -= money;}
24 double BankAccount::getBalance(){ return balance;}
25
26 /***** Main Section *****/
27 int main(){
28 BankAccount myAccount;
29 cout << "Initial balance = " << myAccount.getBalance() << endl;
30
31 myAccount.deposit(50);
32 cout << "Balance 1 = " << myAccount.getBalance() << endl;
33
34 myAccount.deposit(30).deposit(50).deposit(20);
35 cout << "Balance 2 = " << myAccount.getBalance() << endl;
36
37 return 0;
38 }

```



← Get code here.

## 13.5 Operator Overloading

In addition to the existing functionality, you can further extend the versatility of operator overloading by defining new behaviors for operators to work with different types of operands. For instance, you can overload the + operator to work not only with two Rectangle objects but also with an integer, as demonstrated in the code above. This enables more intuitive and concise expressions in your code, enhancing readability and maintainability.

### Overloadable Operators:

- Arithmetic operators:
- Comparison operators:
- Logical operators:
- Bitwise operators:
- Increment and decrement operators:
- Assignment operator:
- Subscript and member access operators:

### Non-Overloadable Operators:

- Scope Resolution (::)
- Member Access (., .\* ->)
- Ternary Conditional (?:)
- Sizeof (sizeof)

Code 13.6: Operator overloading example

```

1 #include <iostream>
2 using namespace std;
3
4 /***** Declaration Section *****/
5 class Rectangle {
6 private:
7 int width, length;
8 public:
9 Rectangle();
10 Rectangle(int w, int l);
11 void printArea();
12 void displayInfo();
13 Rectangle operator+(Rectangle& r);
14 Rectangle operator+(int value); // Operator overload declaration
15 };
16 /***** Implementation Section *****/
17 Rectangle::Rectangle() { width = 1, length = 2; }
18 Rectangle::Rectangle(int w, int l) { width = w, length = l; }
19 void Rectangle::printArea() {
20 cout << "Area of " << this << " = " << width * length << endl;
21 }
22 void Rectangle::displayInfo() {
23 cout << "I am a " << width << "x" << length << " rectangle "
24 << "with id " << this << endl;
25 }
26 Rectangle Rectangle::operator+(Rectangle& r) {
27 Rectangle temp;
28 temp.width = this->width + r.width;
29 temp.length = this->length + r.length;
30 return temp;
31 }
32 /***** Main Section *****/
33 int main() {
34 Rectangle r1, r2(5, 7), r3;
35 r1.displayInfo();
36 r2.displayInfo();
37 r3.displayInfo();
38 r3 = r1 + r2;
39 r3.displayInfo();
40
41 r1.printArea();
42 r2.printArea();
43 r3.printArea();
44 return 0;
45 }

```



← Get code here.





Change the + operator overload to work with a rectangle object and an integer (e.g `r1+2`). The operation will add the integer to both the width and the length of the rectangle.



Overload the > operator for the Rectangle class in C++ to compare two Rectangle objects based on their areas.



Implement the unary ++ operator overload for the Rectangle class in C++ to increment both the width and length of a Rectangle object by one.



What are the benefits of operator overloading in C++?



Explain the difference between overloading a unary and a binary operator in C++.

### 13.6 Practice Questions

1. True / False Questions:

- (a) (**True - False**) Pointers and arrays are unrelated concepts in C++.
- (b) (**True - False**) In modern C++, it's recommended to use `NULL` instead of `nullptr` for representing null pointers.
- (c) (**True - False**) The `this` pointer in C++ is used to refer to the previous object of a class.
- (d) (**True - False**) Operator overloading in C++ enables the creation of new operators.
- (e) (**True - False**) Arrays are always passed by value to functions in C++.
- (f) (**True - False**) The `sizeof` operator can be overloaded in C++.
- (g) (**True - False**) Dynamic arrays in C++ are allocated using the `new` keyword.
- (h) (**True - False**) Overloaded operators in C++ must have the same number of operands as the original operator.
- (i) (**True - False**) In C++, the `=` operator cannot be overloaded.
- (j) (**True - False**) A pointer in C++ can only store the memory address of a variable of the same data type.
- (k) (**True - False**) The `delete[]` operator is used to deallocate memory for single objects allocated with `new`.
- (l) (**True - False**) In C++, the scope resolution operator (`::`) can be overloaded.
- (m) (**True - False**) In C++, the `new` operator returns a pointer to the allocated memory.
- (n) (**True - False**) The `->` operator in C++ is used to access members of a class or structure through a pointer.

2. Which of the following statements is true about pointers in C++?

- (a) Pointers cannot be used to access the members of a class.
- (b) Pointers cannot be dereferenced.
- (c) Pointers can be used to dynamically allocate memory.
- (d) Pointers can only store integer values.

3. What does the 'this' keyword refer to in C++?

- (a) It refers to the base class object.
- (b) It refers to the derived class object.
- (c) It refers to the current object instance.
- (d) It refers to the static member of a class.

- 
4. How are pointers and arrays related in C++?
    - (a) Arrays cannot be accessed using pointers.
    - (b) Pointers and arrays are completely unrelated concepts.
    - (c) Arrays can be accessed using pointers and vice versa.
    - (d) Pointers cannot be used to iterate through arrays.
  
  5. What does the arrow operator (->) do in C++?
    - (a) It performs subtraction.
    - (b) It accesses the member of an object pointed to by a pointer.
    - (c) It performs bitwise AND operation.
    - (d) It is used for logical negation.
  
  6. In C++, how is operator overloading achieved?
    - (a) By defining new operators.
    - (b) By overloading the operators with different parameters.
    - (c) By using predefined overloaded operators only.
    - (d) Operator overloading is not supported in C++.
  
  7. When does a pointer contain the address of a variable?
    - (a) Only when it is initialized.
    - (b) Only when it is declared.
    - (c) It always contains the address of a variable.
    - (d) It never contains the address of a variable.
  
  8. Which operator is used to access the value pointed to by a pointer?
    - (a) \*
    - (b) ->
    - (c) ::
    - (d) &

9. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5 // Dynamic array creation
6 int size = 5;
7 int *arr = new int[size];
8
9 // Filling the array
10 for (int i = 0; i < size; ++i) {
11 arr[i] = i + 1;
12 }
13
14 // Printing array elements using pointers
15 cout << "Array elements: ";
16 for (int i = 0; i < size; ++i) {
17 cout << *(arr + i) << " ";
18 }
19
20 // Freeing memory allocated for the dynamic array
21 delete [] arr;
22
23 return 0;
24 }
```

10. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5 int array[3] = {10, 20, 30};
6 int *ptr = array;
7
8 cout << "Value: " << *(ptr + 1) << endl;
9
10 return 0;
11 }
```

11. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 class MyNumber {
5 private:
6 int value;
7 public:
8 MyNumber(int val) : value(val) {}
9
10 MyNumber operator+(const MyNumber& other) {
11 return MyNumber(value + other.value);
12 }
13
14 void display() {
15 cout << "Value: " << value << endl;
16 }
17 };
18
19 int main() {
20 MyNumber num1(5);
21 MyNumber num2(10);
22
23 MyNumber result = num1 + num2;
24 result.display();
25
26 return 0;
27 }
```

12. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 class Complex {
5 private:
6 double real;
7 double imag;
8 public:
9 Complex(double r, double i) : real(r), imag(i) {}
10
11 Complex operator+(const Complex& other) {
12 return Complex(real + other.real, imag + other.imag);
13 }
14
15 void display() {
16 cout << "Real part: " << real << ", Imaginary part: " << imag << endl;
17 }
18 };
19
20 int main() {
21 Complex num1(3.5, 4.5);
22 Complex num2(1.5, 2.5);
23
24 Complex result = num1 + num2;
25 result.display();
26
27 return 0;
28 }
```

13. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 class Time {
5 private:
6 int hours;
7 int minutes;
8
9 public:
10 Time(int h = 0, int m = 0) : hours(h), minutes(m) {}
11
12 bool operator==(const Time& other) const {
13 return (hours == other.hours && minutes == other.minutes);
14 }
15 };
16
17 int main() {
18 Time t1(5, 30), t2(5, 30);
19 if (t1 == t2)
20 cout << "Times are equal." << endl;
21 else
22 cout << "Times are not equal." << endl;
23 return 0;
24 }
```

14. Write a C++ program that dynamically allocates memory for an array of integers of size 10. Initialize the array with consecutive even numbers starting from 2. Then, print the elements of the array.
15. Create a C++ program that dynamically allocates memory for an array of doubles of size 5. Prompt the user to input values for each element of the array. After filling the array, print the sum of all elements.
16. Write a C++ function named `reverseArray` that takes a pointer to an array of integers and the size of the array as arguments. The function should reverse the elements of the array in place. You do not need to write a complete program, just the function definition.
17. Design a C++ program that dynamically allocates memory for two arrays of integers, each of size 5. Fill the first array with user input and the second array with the squares of the elements of the first array. Finally, print both arrays.

18. Create a C++ program that dynamically allocates memory for an array of characters to store a string entered by the user. Then, write a function named `countVowels` that takes a pointer to this array and counts the number of vowels in the string. Print the count of vowels. You do not need to write a complete program, just the function definition.
19. Write a C++ program that defines a class `Complex` to represent complex numbers. Overload the addition operator `+` to add two complex numbers. Make sure to include the necessary member functions and data members in the class definition. You do not need to write a complete program, just the class definition with the overloaded operator.
20. Design a C++ program that defines a class `Date` to represent dates in a calendar. Overload the equality operator `==` to compare two dates for equality. Include necessary member functions and data members in the class definition. You do not need to write a complete program, just the class definition with the overloaded operator.



## 14. Recursion

### 14.1 Recursive Functions

Recursion in C++ is a programming technique where a function calls itself to solve a problem. In other words, a function is defined in terms of itself, which allows it to break down a complex problem into simpler subproblems.

A recursive function typically has two parts:

**Base Case:** A termination condition that stops the recursion and prevents the function from calling itself further. It is essential to have a base case in recursive functions; otherwise, the recursion will continue indefinitely.

**Recursive Case:** The part of the function where it calls itself with a modified version of the problem, moving closer to the base case. This recursive call helps in solving the original problem by solving smaller instances of the same problem.

#### Quick Facts:

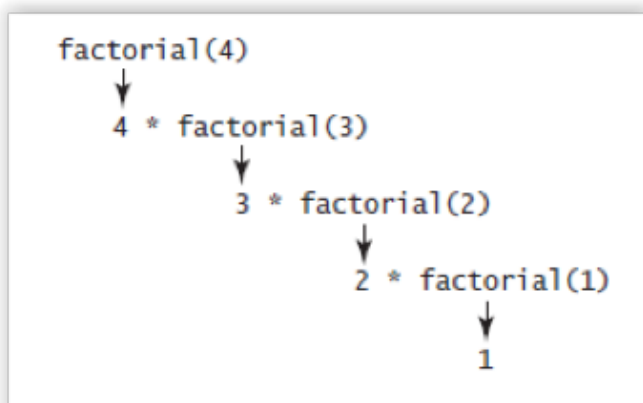
1. Recursion is a technique that solves a problem by first finding the solution for a smaller, simpler problem and use the information to build up to the final answer.
2. In C++, like many other languages, we implement a recursive solution by allowing a function to call itself.
3. Each recursive function has two components, the base case and the recursive step.
4. Both void functions and those returning a value can be recursive.

**Example:** Recall the program 5.4 that defines and tests a factorial function using a for loop. Here we change the factorial function to a recursive function.

Code 14.1: Using recursion to calculate the factorial of a number

```
1 #include <iostream>
2 using namespace std;
3
4 int factorial(int);
5
6 int main() {
7
8 int number=5;
9
10 cout << number << "! = " << factorial(number) << endl;
11
12 return 0;
13 } // end main()
14
15 int factorial(int x) {
16
17 if(x==0 || x==1)
18 return 1;
19 else
20 return x * factorial(x-1);
21
22 } //end factorial()
```

During recursion, a special area of memory called the stack is utilized to store the information as the multiple function calls are executed. Values that are stored in the stack follow the last in – first out (LIFO) order. The following picture and table illustrate the factorial recursive process.



| Step | $x$ | Recursive Call      | Result |
|------|-----|---------------------|--------|
| 1    | 5   | factorial(4)        |        |
| 2    | 4   | factorial(3)        |        |
| 3    | 3   | factorial(2)        |        |
| 4    | 2   | factorial(1)        |        |
| 5    | 1   | -                   | 1      |
| 4    | 2   | $2 \times 1 = 2$    | 2      |
| 3    | 3   | $3 \times 2 = 6$    | 6      |
| 2    | 4   | $4 \times 6 = 24$   | 24     |
| 1    | 5   | $5 \times 24 = 120$ | 120    |

Table 14.1: Steps of recursion to calculate the factorial of a number



Which of the two approaches (iterative or recursive) is better in this case?

Write a program that uses a function `sum(int)` that takes as an argument a positive integer `n` and returns the sum of the first `n` positive integers. Use a while loop. Then write a recursive function that performs the same task.



**Desired output:**

Enter a positive integer: 5

The sum of the first 5 positive integers is: 15



Try using `n=50,000` for each of the programs. What do you notice? Which version is a better choice for this example? Can we further optimize the code?

**Example: The Fibonacci series.**

What is the Fibonacci series?

Code 14.2: Fibonacci example using iteration

```
1 #include <iostream>
2 using namespace std;
3
4 int fibonacci(int x) {
5 int i = 0, j = 1, k = 1, sum=0;
6 while(i < x-1){
7 sum=j+k;
8 j=k;
9 k=sum;
10 i++;
11 }
12 return j;
13 } // end fibonacci()
14
15 int main() {
16 int number = 6; // <-- Enter term here
17 cout << "The " << number << "-th Fibonacci is: ";
18 cout << fibonacci(number) << endl;
19 return 0;
20 } // end main()
```

Code 14.3: Fibonacci example using recursion

```
1 #include <iostream>
2 using namespace std;
3
4 int fibonacci(int x) {
5 if (x==1 || x==2)
6 return 1;
7 else
8 return fibonacci(x-1) + fibonacci(x-2);
9 } // end fibonacci()
10
11 int main() {
12 int number = 6; // <-- Enter term here
13 cout << "The " << number << "-th Fibonacci is: ";
14 cout << fibonacci(number) << endl;
15 return 0;
16 } // end main()
```

Write a program that uses a function **power(base,exponent)** which when invoked returns  $base^{exponent}$ .

**Desired output:**

Enter base: 5

Enter exponent: 3

5 to the power 3 is: 125

**Example:** Prime factorization of a number.

Code 14.4: Prime factorization of a number

```
1 #include <iostream>
2 using namespace std;
3
4 void get_divisors(int n);
5
6 int main() {
7 int n = 0;
8 cout << "Enter an integer: ";
9 cin >> n;
10 get_divisors(n);
11 return 0;
12 } // end main
13
14 void get_divisors(int n) {
15 for (int i = 2; i < n; i++)
16 if (n % i == 0) {
17 cout << i << ", ";
18 get_divisors(n/i);
19 return;
20 }
21 cout << n;
22 } // end get_divisors()
```



← Get code here.

**Example:** Convert decimal to binary.

Code 14.5: Example void recursive function - Convert to Binary

```
1 #include <iostream>
2 using namespace std;
3
4 void toBinary(int);
5
6 int main() {
7 int number = 5;
8 cout << "Decimal: " << number << endl;
9 cout << "Binary: ";
10 toBinary(number);
11 return 0;
12 }
13
14 void toBinary(int n) {
15 if(n == 0)
16 return;
17
18 toBinary(n/2);
19 cout << n%2;
20 return;
21 }
```



Write an iterative version of the 'convert to binary' program.



Can we generalize this algorithm to convert to other bases?

**Example:** Reverse the digits of a number.

Code 14.6: Iterative program to reverse the digits of an integer

```
1 #include <iostream>
2 using namespace std;
3
4 void printReverse(int);
5
6 int main() {
7 int number = 12345;
8 cout << "Original: " << number << endl;
9 cout << "Reversed: ";
10 printReverse(number);
11 return 0;
12 }
13
14 void printReverse(int x) {
15 while(x!=0){
16 cout << x%10;
17 x=x/10;
18 }
19 }
```



Write a recursive version of the 'reverse digits' program.

**Example:** Determine if a string is a palindrome.

Code 14.7: Determine if string is a palindrome - recursive

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 bool isPalindrome (string);
6
7 int main(){
8 string mystr;
9 cout << "Enter a string: ";
10 getline (cin, mystr);
11 if (isPalindrome(mystr))
12 cout << mystr << " is a palindrome \n";
13 else
14 cout << mystr << " is not a palindrome \n";
15 }
16
17 bool isPalindrome(string str){
18
19 int length = str.length();
20
21 if(length <= 1)
22 return true;
23
24 if(str[0] == str[length-1])
25 return isPalindrome(str.substr(1, (length - 2)));
26
27 return false;
28 }
```



<— Get code here (recursive version).



<— Get code here (iterative version).



## 14.2 Practice Questions

1. What is the meaning of the following in the context of Computer Science?

(a) recursion base case:

---

---

(b) recursive step:

---

---

(c) iteration vs recursion:

---

---

2. True-False questions – Circle your choice

- (a) (**True** - **False**) Recursion is a programming technique in C++ where a function calls itself to solve a problem.
- (b) (**True** - **False**) A recursive function must have a base case, which is the condition that stops the recursion.
- (c) (True - **False**) Recursion is always more memory-efficient than iterative solutions for solving a problem.
- (d) (True - **False**) Recursion can lead to infinite loops if the base case is not properly defined or if there is no base case.
- (e) (**True** - **False**) In recursion, each recursive call creates a new copy of local variables and function arguments.
- (f) (**True** - **False**) In C++, recursion can be used to solve problems that can be broken down into smaller subproblems of the same type.
- (g) (True - **False**) Recursive functions in C++ cannot have more than one base case.
- (h) (**True** - **False**) In recursion, the computer uses a data structure called the call stack to keep track of the recursive calls.

3. The following program should print the sum of the first n positive integers. Find and fix the errors in the code.

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5 int number;
6 cout << "Enter a Positive Integer: ";
7 cin >> number;
8 cout << "The sum of the first " << number;
9 cout << " integers is: " << sum(number) << endl;
10 return 0;
11 } // end main() function
12
13 int sum(int x) {
14 if (x == 1)
15 return 1;
16 else
17 return x + sum(x);
18 }
```

**Sample output:**

Enter a Positive Integer: **5**  
The sum of the first 5 integers is: 15

4. Write the exact output.

```
1 #include <iostream>
2 using namespace std;
3
4 int f(int x);
5
6 int main() {
7 int number = 5;
8 cout << f(number);
9 return 0;
10 } // end main() function
11
12 int f(int x) {
13 if (x < 2)
14 return 1;
15 else
16 return f(x-1) * f(x-2) + 1;
17 }
```

5. The following program should print the prime factorization of a number entered by the user. Fix all the errors and make the necessary changes so that the code produces the desired output. Sample output is provided.

```
1 #include <iostream>
2 #include <cmath>
3 using namespace std;
4
5 void get_divisors(int n);
6
7 int main() {
8 int n = 0;
9 cout << "Enter an integer: ";
10 cin >> n;
11
12 cout << endl;
13 return 0;
14 }
15 void get_divisors(int n) {
16 int i;
17
18 for (i = 2; i <= sqrt_of_n; i++)
19 if (n % i == 5) {
20 cout << i << ", ";
21 get_divisors(n \ i);
22 return;
23 }
24 cout << n;
25 }
```

**Sample output:**

Enter an integer: **24**  
2,2,2,3

6. Write a program that uses a recursive function **power(base,exponent)** which when invoked returns  $base^{exponent}$ .

**Sample output:**

Enter the Base: **5**  
Enter the Exponent: **3**  
5 to the power 3 = 125

7. Write a program that uses a function **sum(int)** that takes as an argument a positive integer *n* and returns the sum of the first *n* positive integers. Use a while loop. Then write a recursive function that performs the same task. .

**Sample output:**

Enter a positive integer: **5**  
The sum of the first 5 positive integers is: 15

8. Write a program that uses a function **gcf(int,int)** that takes as arguments two positive integers and returns their GCF (Greatest Common Factor) sum of the first *n* positive integers. First use a loop. Then write a recursive function that performs the same task.

**Sample output:**

Enter first integer: **50**  
Enter second integer: **30**  
Their gcf is: 10



## 15. Inheritance

### 15.1 Introduction

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a class to inherit properties and behavior from another class. This mechanism promotes code reuse, enhances code organization, and facilitates the creation of more maintainable and extensible software systems.

In this chapter, we will explore the principles of inheritance in C++, covering its syntax, types, access control, constructor and destructor inheritance, virtual functions, and common best practices.

Code 15.1: Simple inheritance example

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 class Person {
6 public:
7 string name;
8 int age;
9 };
10
11 class Student: public Person {
12 public:
13 double gpa;
14 };
15
16 int main() {
17 // Creating an object of the Student class
18 Student st1;
19
20 // Accessing and assigning values to inherited members
21 st1.name = "Bob";
22 st1.age = 20;
23
24 // Accessing and assigning values to derived member
25 st1.gpa = 3.55;
26
27 // Displaying information about the student
28 cout << st1.name << " is a " << st1.age << " year old student";
29 cout << " with a GPA of " << st1.gpa << endl;
30
31 return 0;
32 }
```



Which class serves as the **base class** and which one serves as the **derived class** in this example?



Alternatively, the terms **parent class** and **child class** are also used to describe the relationship between classes. Which class is the **parent class** and which one is the **child class** in our example?



## 15.2 Constructors and Destructors in Inheritance

In C++, constructors and destructors play a crucial role in the context of inheritance. When a derived class is instantiated, the constructors of both the base and derived classes are called, followed by their respective destructors when the object goes out of scope. Let's explore how constructors and destructors work in our Person and Student example:

Code 15.2: Constructors and Destructors in Inheritance

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 class Person {
6 public:
7 string name;
8 int age;
9
10 // Constructor
11 Person(const string& n, int a) : name(n), age(a) {
12 cout << "Person constructor called" << endl;
13 }
14
15 // Destructor
16 ~Person() {
17 cout << "Person destructor called" << endl;
18 }
19 };
20
21 class Student : public Person {
22 public:
23 double gpa;
24
25 // Constructor
26 Student(const string& n, int a, double g) : Person(n, a), gpa(g) {
27 cout << "Student constructor called" << endl;
28 }
29
30 // Destructor
31 ~Student() {
32 cout << "Student destructor called" << endl;
33 }
34 };
35
36 int main() {
37 // Creating an object of the Student class
38 Student st1("Bob", 20, 3.55);
39
40 return 0;
41 }
```

In this code:

- We have defined constructors and destructors for both the `Person` and `Student` classes.
- The constructors initialize the members of their respective classes, including those inherited from the base class.
- The destructors clean up any resources allocated by the classes, ensuring proper cleanup when objects go out of scope.
- When an object of the `Student` class is created in the `main()` function, both the `Person` constructor and the `Student` constructor are called, followed by their respective destructors when the program exits.



In what order are the constructors and destructors called when an object of the `Student` class is created and destroyed?



What are the advantages of calling a constructor explicitly?



Is it best to keep attributes protected or private?

Write a C++ program that demonstrates the use of constructors and destructors in inheritance. Define a base class named `Vehicle` with a constructor that prints "Vehicle constructor called" and a destructor that prints "Vehicle destructor called". Then, define a derived class named `Car` that inherits from `Vehicle` and adds its own constructor and destructor, each printing "Car constructor called" and "Car destructor called" respectively. In the `main()` function, create an object of the `Car` class and observe the order in which the constructors and destructors are called.



**Expected Output:**

```
Vehicle constructor called
Car constructor called
Car destructor called
Vehicle destructor called
```

### 15.3 Inheriting Functionality and Method Overriding


Method inheritance and overriding facilitate code reuse and customization. Inheritance allows derived classes to inherit methods from their base classes. Method overriding enables derived classes to redefine the behavior of inherited methods, allowing them to provide specialized implementations while still adhering to the interface defined by the base class.

Code 15.3: Inheriting member functions

```

1 #include <iostream>
2 using namespace std;
3
4 class Person {
5 protected:
6 string name;
7 int age;
8 public:
9 Person(string n, int a): name(n), age(a)
10 { cout << "New person created." << endl; }
11 void sayHello() {
12 cout << "Hello I am " << name << ". ";
13 cout << "I am a person." << endl;
14 }
15 };
16
17 class Student: public Person {
18 private:
19 double gpa;
20 public:
21 Student(string n, int a, double g): Person(n,a), gpa(g)
22 { cout << "New student created." << endl; }
23 };
24
25 int main() {
26 Student st1("Bob", 20, 3.55);
27 st1.sayHello();
28 return 0;
29 }

```




```

1 void sayHello() {
2 cout << "Hello I am " << name << ". ";
3 cout << "I am a student." << endl;
4 }

```

Add the above function definition to the Student class definition and run the program. What change do you observe?



```

1 void sayHello(string mood) {
2 cout << "Hello I am " << name << ". ";
3 cout << "I am a " << mood << " student." << endl;
4 }

```

Change the function definition to the one above and run again the program. Any observations?

Another example of method overriding.

Code 15.4: Method overriding example

```

1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 class Animal {
6 public:
7 void makeSound() const {
8 cout << "Animal makes a generic sound" << endl;
9 }
10 };
11
12 class Dog : public Animal {
13 public:
14 void makeSound() const {
15 cout << "Dog barks: Woof! Woof!" << endl;
16 }
17 };
18
19 class Cat : public Animal {
20 public:
21 void makeSound() const {
22 cout << "Cat meows: Meow! Meow!" << endl;
23 }
24 };
25
26 int main() {
27 Animal animal;
28 Dog dog;
29 Cat cat;
30
31 animal.makeSound(); // Output: Animal makes a generic sound
32 dog.makeSound(); // Output: Dog barks: Woof! Woof!
33 cat.makeSound(); // Output: Cat meows: Meow! Meow!
34
35 return 0;
36 }

```

## 15.4 Access Control in Inheritance

In C++, access specifiers (`public`, `protected`, `private`) play a crucial role in determining the accessibility of base class members in derived classes. These access specifiers dictate how derived classes can access the members (attributes and methods) inherited from the base class.

### Public Inheritance Example:

Code 15.5: Public Inheritance

```
1 class Base {
2 public:
3 int publicMember;
4 protected:
5 int protectedMember;
6 private:
7 int privateMember;
8 };
9
10 class Derived : public Base {
11 // publicMember remains public
12 // protectedMember remains protected
13 // privateMember remains inaccessible
14 };
```

### Protected Inheritance Example:

Code 15.6: Protected Inheritance

```
1 class Base {
2 public:
3 int publicMember;
4 protected:
5 int protectedMember;
6 private:
7 int privateMember;
8 };
9
10 class Derived : protected Base {
11 // publicMember becomes protected
12 // protectedMember becomes protected
13 // privateMember remains inaccessible
14 };
```

**Private Inheritance Example:**

Code 15.7: Private Inheritance

```
1 class Base {
2 public:
3 int publicMember;
4 protected:
5 int protectedMember;
6 private:
7 int privateMember;
8 };
9
10 class Derived : private Base {
11 // publicMember becomes private
12 // protectedMember becomes private
13 // privateMember remains inaccessible
14 };
```

In summary, the choice of access specifier in inheritance determines the accessibility of base class members in derived classes, influencing the encapsulation and visibility of the inherited members.

## 15.5 Multiple Inheritance

Multiple inheritance is a feature in C++ that allows a child (derived) class to inherit from more than one parent (base) class. This enables the child class to inherit properties and behavior from multiple sources.

Code 15.8: Multiple inheritance example

```

1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 class Student {
6 public:
7 string name;
8 string major;
9 Student() { cout << "Student created." << endl; }
10 ~Student() { cout << "Student deleted." << endl; }
11 void sayHello() { cout << "Hello, I am a student." << endl; }
12 };
13
14 class Teacher {
15 public:
16 string name1;
17 string rank;
18 Teacher() { cout << "Teacher created." << endl; }
19 ~Teacher() { cout << "Teacher deleted." << endl; }
20 void sayHello() { cout << "Hello, I am a teacher." << endl; }
21 };
22
23 class GraduateInstructor : public Student, public Teacher {
24 public:
25 GraduateInstructor() { cout << "Instructor created." << endl; }
26 ~GraduateInstructor() { cout << "Instructor deleted." << endl; }
27 };
28
29 int main() {
30 GraduateInstructor gil;
31 gil.name = "Bob";
32 gil.rank = "Instructor";
33 cout << gil.name << " is an " << gil.rank << endl;
34 return 0;
35 }

```



← Get code here.



What are the benefits and risks of using multiple inheritance?

**The diamond problem example:**

Code 15.9: The diamond problem example

```
1 #include <iostream>
2 using namespace std;
3
4 class A {
5 public:
6 void foo() {
7 cout << "A's foo()" << endl;
8 }
9 };
10
11 class B : public A {
12 public:
13 void bar() {
14 cout << "B's bar()" << endl;
15 }
16 };
17
18 class C : public A {
19 public:
20 void baz() {
21 cout << "C's baz()" << endl;
22 }
23 };
24
25 class D : public B, public C {
26 public:
27 // Here, we override A's foo() method
28 void foo() {
29 cout << "D's foo()" << endl;
30 }
31 };
32
33 int main() {
34 D d;
35 d.foo(); // Which foo() method will be called?
36 return 0;
37 }
```



&lt;— Get code here.



## 15.6 Multilevel Inheritance

Multilevel inheritance involves creating a hierarchy of classes where a derived class inherits from another derived class, forming multiple levels of inheritance. This allows for the construction of complex class relationships and promotes code reusability and organization.

In the example that follows, we have a hierarchy of classes representing majors. The `Major` class serves as the base class, while `Engineering` and `CivilEngineering` are derived classes representing more specific majors.

The program demonstrates the object lifecycle by creating an object of the `CivilEngineering` class. As the object is created, constructors of all classes in the inheritance chain are called in the order of their inheritance. Similarly, when the object goes out of scope, destructors are called in the reverse order.

Code 15.10: Multilevel inheritance example

```
1 #include <iostream>
2 using namespace std;
3
4 // Base class representing a major
5 class Major {
6 public:
7 int id;
8 Major() {
9 cout << "New major." << endl;
10 }
11 ~Major() {
12 cout << "Major deleted." << endl;
13 }
14 };
15
16 // Derived class representing a specific major in engineering
17 class Engineering : public Major {
18 public:
19 Engineering() {
20 cout << "New engineering major." << endl;
21 }
22 ~Engineering() {
23 cout << "Engineering major deleted." << endl;
24 }
25 };
26
27 // Further derived class representing civil engineering major
28 class CivilEngineering : public Engineering {
29 public:
30 CivilEngineering() {
31 cout << "New civil eng. major." << endl;
32 }
33 ~CivilEngineering() {
34 cout << "Civil eng. deleted." << endl;
35 }
36 };
37
38 int main() {
39 // Creating an object of CivilEngineering class
40 CivilEngineering major1;
41 major1.id = 10; // Accessing inherited member from Major class
42 cout << "Major with id: " << major1.id << endl;
43 return 0;
44 }
```



← Get code here.

## 15.7 Practice Questions

### 1. True-False questions – Circle your choice

- (a) (**True - False**) In C++, inheritance allows a class to inherit properties and behavior from another class.
- (b) (**True - False**) Constructors and destructors are not inherited by derived classes in C++.
- (c) (**True - False**) Constructors of both the base and derived classes are called when an object of the derived class is created.
- (d) (**True - False**) It's not necessary to explicitly call a constructor when creating an object of a derived class in C++.
- (e) (**True - False**) In C++, it is best to keep attributes private rather than protected for better encapsulation.
- (f) (**True - False**) Public inheritance in C++ allows all members of the base class to retain their access levels in the derived class.
- (g) (**True - False**) In C++, protected inheritance makes all members of the base class private in the derived class.
- (h) (**True - False**) Multiple inheritance in C++ allows a class to inherit from more than two parent classes simultaneously.
- (i) (**True - False**) Method overriding in C++ allows derived classes to redefine the behavior of inherited methods.
- (j) (**True - False**) In C++, when a derived class overrides a method from the base class, the base class method is no longer accessible from the derived class.
- (k) (**True - False**) In multilevel inheritance, a derived class inherits from another derived class, forming multiple levels of inheritance.
- (l) (**True - False**) In multiple inheritance, ambiguity arises when a method is called that exists in more than one base class.
- (m) (**True - False**) Method overloading and method overriding are the same concepts in C++.

2. Which of the following statements about constructors and destructors in C++ inheritance is correct?
  - (a) Constructors are inherited by derived classes, but destructors are not.
  - (b) Destructors are inherited by derived classes, but constructors are not.
  - (c) Both constructors and destructors are inherited by derived classes.
  - (d) Neither constructors nor destructors are inherited by derived classes.
3. What happens when a derived class object is created in C++ regarding constructors and destructors?
  - (a) Only the derived class constructor is called.
  - (b) Only the base class constructor is called.
  - (c) Both the base class constructor and the derived class constructor are called.
  - (d) Only the derived class destructor is called.
4. In C++, how can a derived class constructor initialize the base class part of an object?
  - (a) By automatically inheriting the base class constructor.
  - (b) By explicitly calling the base class constructor in the derived class constructor's body.
  - (c) By using the base class constructor in the derived class destructor.
  - (d) By accessing the base class constructor directly from the derived class object.
5. What happens when a derived class overrides a method from the base class in C++?
  - (a) The base class method becomes inaccessible from the derived class.
  - (b) Both base class and derived class methods are accessible independently.
  - (c) The derived class method hides the base class method.
  - (d) The base class method is deleted.
6. What problem can occur with multiple inheritance in C++?
  - (a) Inability to inherit attributes.
  - (b) Difficulty in accessing base class methods.
  - (c) The diamond problem.
  - (d) Compiler errors.
7. What does protected inheritance in C++ mean?
  - (a) All members of the base class become private in the derived class.
  - (b) All members of the base class become protected in the derived class.
  - (c) All members of the base class become public in the derived class.
  - (d) The derived class inherits from multiple base classes.

8. Write the exact output for the code below:

```
1 #include <iostream>
2 using namespace std;
3
4 class Base {
5 public:
6 void display () {
7 cout << "Base class display function" << endl;
8 }
9 };
10
11 class Derived : public Base {
12 public:
13 void display () {
14 cout << "Derived class display function" << endl;
15 }
16 };
17
18 int main() {
19 Base obj1;
20 Derived obj2;
21
22 obj1.display ();
23 obj2.display ();
24
25 return 0;
26 }
```

9. Write the exact output for the code below:

```
1 #include <iostream>
2 using namespace std;
3
4 class Base {
5 public:
6 void display() {
7 cout << "Base class display function" << endl;
8 }
9 };
10
11 class Derived : public Base {
12 public:
13 void display() {
14 cout << "Derived class display function" << endl;
15 Base::display();
16 }
17 };
18
19 int main() {
20 Derived obj;
21 obj.display();
22
23 return 0;
24 }
```

10. Write a C++ program to demonstrate single inheritance. Define a base class `Shape` with a member function `display()` that prints "This is a shape." Then, derive a class `Rectangle` from `Shape` with a member function `draw()` that prints "Drawing a rectangle."
11. Create a C++ program to showcase hierarchical inheritance. Define a base class `Animal` with a member function `sound()` that prints "Animal sound." Then, derive two classes `Dog` and `Cat` from `Animal`, each with their own `sound()` function printing "Bark" and "Meow" respectively.
12. Implement multiple inheritance in C++ by defining three classes: `Person`, `Employee`, and `Student`. `Person` should have a member function `introduce()` that prints "I am a person." `Employee` should inherit from `Person` and have a member function `work()` that prints "I am an employee." Similarly, `Student` should inherit from `Person` and have a member function `study()` that prints "I am a student."
13. Write a C++ program to demonstrate the use of constructor inheritance. Define a base class `Parent` with a parameterized constructor that accepts an integer argument. Derive a class `Child` from `Parent` with a parameterized constructor that accepts two integer arguments and initializes the base class using these arguments.
14. Develop a C++ program to show how constructors and destructors work in inheritance. Define a base class `Vehicle` with a constructor that prints "Vehicle created" and a destructor that prints "Vehicle destroyed." Derive a class `Car` from `Vehicle` with its own constructor and destructor messages.
15. Write a C++ program to demonstrate the use of dynamic memory allocation in inheritance. Define a base class `Parent` with a dynamic array as a member variable. Derive a class `Child` from `Parent` and dynamically allocate memory for the array in `Child`. Test the program by dynamically allocating memory for `Child` objects and accessing the array.
16. Implement a C++ program to illustrate how access specifiers work in inheritance. Define a base class `Base` with private, protected, and public member variables and member functions. Derive a class `Derived` from `Base` and attempt to access each member variable and function from `Derived`. Explain the results.
17. Develop a C++ program to show how member function overriding works in inheritance. Define a base class `Shape` with a member function `draw()` that prints "Drawing a shape." Derive a class `Circle` from `Shape` and override `draw()` to print "Drawing a circle." Create objects of both classes and call their `draw()` functions to see the overridden behavior.





## 16. Polymorphism

### 16.1 Introduction

Polymorphism is an important mechanism in object oriented programming. The term itself means **many forms**. When two or more classes are related to each other via inheritance, we want to be able to determine which inherited member functions are executed during the life of a program. Polymorphism allows for this decision to be made during the execution of the program by adapting to the type of object is currently working with at the time the decision needs to be made. This is also called **late binding**. Virtual functions are used for this purpose. Before we look at the first example of virtual functions lets revisit functionality inheritance, function overloading and function redefinitions.

## 16.2 Polymorphism and Virtual Functions

Code 16.1: Overloading and redefining functions example

```

1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 class Person {
6 public:
7 Person() { cout << "New person created." << endl;}
8
9 void sayHello() const {
10 cout << "Hello , I am a person." << endl;
11 }
12 void sayHello(const string& name) const {
13 cout << "Hello , I am " << name << ". I am a person." << endl;
14 }
15 };
16
17 class Student : public Person {
18 public:
19 Student() { cout << "New student created." << endl;}
20
21 void sayHello() const {
22 cout << "Hello , I am a student." << endl;
23 }
24 void sayHello(const string& name) const {
25 cout << "Hello , I am " << name << ". I am a student." << endl;
26 }
27 };
28
29 int main() {
30 Person person;
31 Student student;
32 person.sayHello();
33 person.sayHello("Bob");
34 student.sayHello();
35 student.sayHello("John");
36 return 0;
37 }

```



← Get code here.

In the example above we have both function overloading and function redefining. The decision on which function will be called during execution is made at compilation time. This is also referred to as static binding. In the next example we will use virtual functions and discuss the difference between static and late binding.

Code 16.2: Example using virtual functions

```
1 #include <iostream>
2 using namespace std;
3
4 class Person {
5 public:
6 Person() { cout << "New person created." << endl;}
7
8 virtual void sayHello() const {
9 cout << "Hello, I am a person." << endl;
10 }
11
12 virtual void sayHello(const string& name) const {
13 cout << "Hello, I am " << name << ". I am a person." << endl;
14 }
15 };
16 class Student : public Person {
17 public:
18 Student() { cout << "New student created." << endl;}
19
20 void sayHello() const override {
21 cout << "Hello, I am a student." << endl;
22 }
23
24 void sayHello(const string& name) const override {
25 cout << "Hello, I am " << name << ". I am a student." << endl;
26 }
27 };
28 int main() {
29 Person* person = new Person();
30 Person* student = new Student();
31 person->sayHello();
32 person->sayHello("Bob");
33 student->sayHello();
34 student->sayHello("John");
35 delete person;
36 delete student;
37 return 0;
38 }
```

In this version of the code, I've added the virtual keyword to the sayHello() methods in the Person class. This means that when a Student object is being accessed through a pointer or reference to a Person object, it will call the Student class's version of the function, not the Person class's version. This is an example of dynamic binding in C++, which is achieved through the use of virtual functions. The override keyword in the Student class is used to indicate that the function is intended to override a virtual function in the base class. This can help catch errors if the function signature in the base class changes. The decision on which function will be called during execution is now made at runtime, not at compile time. This is also referred to as late binding.

Code 16.3: Another virtual function example

```

1 #include <iostream>
2 using namespace std;
3
4 class Shape {
5 public:
6 int height;
7 Shape() : height(1) {}
8 Shape(int h) : height(h) {}
9 virtual void drawShape() const {
10 cout << "Shape: height: " << height << endl;
11 }
12 };
13
14 class Square : public Shape {
15 public:
16 Square() : Shape(1) {}
17 Square(int h) : Shape(h) {}
18 void drawShape() const override {
19 cout << "Square: height: " << height << endl;
20 for(int i = 0; i < height; i++){
21 for(int j = 0; j < height * 2; j++){
22 cout << '*';
23 }
24 cout << endl << endl;
25 }
26 };
27
28
29 class Triangle : public Shape {
30 public:
31 Triangle() : Shape(1) {}
32 Triangle(int h) : Shape(h) {}
33 void drawShape() const override {
34 cout << "Triangle: height: " << height << endl;
35 for (int i = 1; i <= height; i++) {
36 for(int j = 1; j <= height - i; j++){
37 cout << ' ';
38 }
39 for (int k = 1; k <= 2 * i - 1; k++)
40 cout << '*';
41 cout << '\n';
42 }
43 cout << endl;
44 };
45
46 int main() {
47 Square mySquare(4);
48 Triangle myTriangle(5);
49 Shape myShape(6);
50 Shape* shapep1 = &mySquare;
51 Shape* shapep2 = &myTriangle;
52 Shape* shapep3 = &myShape;
53 shapep1->drawShape();
54 shapep2->drawShape();
55 shapep3->drawShape();
56 return 0;
57 }

```



← Get code here.

The previous C++ code demonstrates polymorphism in object-oriented programming using a Shape base class and Square and Triangle derived classes. Each class has a drawShape method, with the derived classes' methods overriding the base class's method. The program uses Shape pointers to call the appropriate drawShape method at runtime, showcasing dynamic binding.



The virtual keyword in C++ is used to allow a function in the base class to be overridden in a derived class. What happens when you remove it?



Use the dot (.) operator instead of pointers to call method drawShape(). In C++, the dot operator is used to access members of an object directly, while the arrow operator is used to access members of an object through a pointer. How does this change the behavior of the drawShape() method calls?



Add a function **newDraw()** to the program and change **main()** as shown below. What do you notice? After adding the newDraw() function and modifying main(), what is the output of the program? How does this demonstrate the usefulness of virtual functions?

Code 16.4: Try now additions/changes

```

1 void newDraw(Shape *s){
2 cout << "Hello I am the new function!!!" << endl;
3 s->drawShape ();
4 }
5
6 int main () {
7 Square mySquare (4);
8 Triangle myTriangle (5);
9 Shape myShape (6);
10 newDraw(&mySquare);
11 newDraw(&myTriangle);
12 newDraw(&myShape);
13 return 0;
14 }
```



In object-oriented programming, **redefining** and **overriding** are two ways that a derived class can provide its own implementation of a function from the base class. Can you explain the difference between these two concepts?

## 16.3 Pure Virtual Functions and Abstract Classes

### Pure Virtual Functions

A pure virtual function is a function declared in a base class that has no definition relative to the base. A class containing at least one pure virtual function is considered an abstract base class. Classes derived from the abstract base class must implement all pure virtual functions to be considered concrete, i.e., to be able to instantiate objects of that class.

In C++, a pure virtual function is declared as follows:

```
1 virtual void functionName() = 0;
```

Pure virtual functions are used when the base class has a function that will be implemented in all derived classes, but there is no meaningful definition of that function in the base class itself. It's a way of ensuring that each derived class implements this function.

### Abstract Classes

An abstract class in C++ is a class that has at least one pure virtual function. You cannot create objects of an abstract class type. However, you can use pointers and references to abstract class types.

Abstract classes act as expressions of general concepts from which more specific classes can be derived. You can't create an object of the abstract class type, but you can use pointers and references to abstract class types to achieve polymorphism and dynamic binding.

Here is an example of an abstract class with a pure virtual function, and a class that derives from it:

```
1 class AbstractClass {
2 public:
3 // Pure virtual function makes this class Abstract class
4 virtual void pureVirtualFunction() = 0;
5 };
6
7 class ConcreteClass : public AbstractClass {
8 public:
9 void pureVirtualFunction() override {
10 // implementation of pure virtual function
11 }
12 };
```

Code 16.5: Abstract class example

```
1 #include <iostream>
2 using namespace std;
3
4 class Animal {
5 public:
6 Animal(string n): name(n) {}
7 virtual ~Animal() {}
8 virtual void makeNoise() const = 0; // Added const here
9 protected:
10 string name;
11 };
12
13 class Dog: public Animal {
14 public:
15 Dog(string n): Animal(n) {}
16 void makeNoise() const override { // Added const here
17 cout << name << " says woof!" << endl;
18 }
19 };
20
21 class Cat: public Animal {
22 public:
23 Cat(string n): Animal(n) {}
24 void makeNoise() const override { // Added const here
25 cout << name << " says meow!" << endl;
26 }
27 };
28
29 int main() {
30 Dog myDog("Lassie");
31 myDog.makeNoise();
32 Cat myCat("Bella");
33 myCat.makeNoise();
34 return 0;
35 }
```



← Get code here.



Which of the three classes (Animal, Dog, Cat) in the code is abstract?



The **override** keyword in C++ is used to indicate that a virtual function is intended to override a function in the base class. Is it mandatory to use override when defining inherited virtual functions? What might be the benefits of using it?



Given that Animal is an abstract class, can we create an object of type Animal directly?



Remove the makeNoise() function definition from the Dog class. Does the program work?



In main() add an array of four animals, two dogs and two cats. Then use a for loop to call the makeNoise() function for each array element. After modifying main() as described, what is the output of the program? How does this demonstrate the concept of polymorphism?



## 16.4 Practice Questions

### 1. True-False questions – Circle your choice

- (a) (**True - False**) Polymorphism in C++ allows objects of different types to be treated as objects of a common type.
- (b) (**True - False**) In C++, function overloading is a type of static polymorphism.
- (c) (**True - False**) Function overriding in C++ is achieved by declaring a function in the base class as 'virtual' and then redefining it in the derived class.
- (d) (**True - False**) A class with at least one pure virtual function is called an abstract class.
- (e) (**True - False**) In C++, we can create an instance of an abstract class.
- (f) (**True - False**) The 'override' keyword is mandatory when overriding a function in C++.
- (g) (**True - False**) If a base class pointer points to a derived class object and a virtual function is invoked, the function from the base class will be executed.
- (h) (**True - False**) If a function is declared as 'virtual' in the base class, it is not necessary to use the 'virtual' keyword when overriding that function in the derived class.
- (i) (**True - False**) A derived class inherits all base class methods, but it cannot have its own unique methods.
- (j) (**True - False**) The 'virtual' keyword in C++ is used to allow a function in the base class to be redefined in a derived class.
- (k) (**True - False**) If a class contains at least one pure virtual function, then it cannot be instantiated.
- (l) (**True - False**) In C++, if a function is declared as 'virtual' in a base class, it automatically becomes 'virtual' in any class that directly or indirectly inherits from that base class.
- (m) (**True - False**) The 'override' keyword in C++ is used to indicate that a function is intended to override a function in the base class.
- (n) (**True - False**) A constructor in C++ can be virtual.
- (o) (**True - False**) A destructor in C++ can be virtual.
- (p) (**True - False**) If a base class destructor is not virtual, derived class destructors will not be called when an object is deleted through a base class pointer.
- (q) (**True - False**) Overloading a function changes the function's behavior during runtime.

2. What is the purpose of using virtual functions in C++?
  - (a) To restrict access to member functions.
  - (b) To enable function overloading.
  - (c) To allow for dynamic binding and polymorphism.
  - (d) To improve memory management.
3. Which of the following best describes polymorphism in C++?
  - (a) The ability to create multiple objects of the same class.
  - (b) The ability to store different types of objects in the same array.
  - (c) The ability of a function to behave differently based on the object it is called with.
  - (d) The ability to access private members of a class from outside the class.
4. What is the purpose of a pure virtual function in C++?
  - (a) To provide a default implementation for derived classes.
  - (b) To prevent derived classes from overriding the function.
  - (c) To allow the base class to be instantiated.
  - (d) To define an interface that derived classes must implement.
5. Which statement about abstract classes in C++ is true?
  - (a) Abstract classes cannot have any member functions.
  - (b) Abstract classes can be instantiated directly.
  - (c) Abstract classes must have at least one pure virtual function.
  - (d) Abstract classes cannot have derived classes.
6. Which type of binding occurs when a function call is resolved during program execution in C++?
  - (a) Early binding
  - (b) Static binding
  - (c) Compile-time binding
  - (d) Late binding
7. What is the purpose of using pointers or references to base class objects in C++?
  - (a) To improve memory allocation efficiency.
  - (b) To enable multiple inheritance.
  - (c) To achieve encapsulation.
  - (d) To achieve polymorphism and dynamic binding.

8. Write the exact output for the code below:

```
1 #include <iostream>
2 using namespace std;
3 class A {
4 public:
5 A() { cout << "Constructor A" << endl; }
6 ~A() { cout << "Destructor A" << endl; }
7 };
8
9 class B : public A {
10 public:
11 B() { cout << "Constructor B" << endl; }
12 ~B() { cout << "Destructor B" << endl; }
13 };
14
15 int main() {
16 B obj;
17 return 0;
18 }
```

9. Write the exact output for the code below:

```
1 #include <iostream>
2 using namespace std;
3 class Base {
4 public:
5 Base() { cout << "Base Constructor" << endl; }
6 virtual ~Base() { cout << "Base Destructor" << endl; }
7 };
8
9 class Derived : public Base {
10 public:
11 Derived() { cout << "Derived Constructor" << endl; }
12 ~Derived() { cout << "Derived Destructor" << endl; }
13 };
14
15 int main() {
16 Base* ptr = new Derived();
17 delete ptr;
18 return 0;
19 }
```

10. Write the exact output for the code below:

```
1 #include <iostream>
2 using namespace std;
3 class Animal {
4 public:
5 virtual void sound() const = 0;
6 };
7
8 class Dog : public Animal {
9 public:
10 void sound() const override {
11 cout << "Woof!" << endl;
12 }
13 };
14
15 int main() {
16 Dog dog;
17 dog.sound();
18 return 0;
19 }
```

11. Write the exact output for the code below:

```
1 #include <iostream>
2 using namespace std;
3 class Shape {
4 public:
5 virtual void draw() const {
6 cout << "Drawing Shape" << endl;
7 }
8 };
9
10 class Circle : public Shape {
11 public:
12 void draw() const override {
13 cout << "Drawing Circle" << endl;
14 }
15 };
16
17 class Square : public Shape {
18 public:
19 void draw() const override {
20 cout << "Drawing Square" << endl;
21 }
22 };
23
24 int main() {
25 Shape* arr[3];
26 arr[0] = new Shape();
27 arr[1] = new Circle();
28 arr[2] = new Square();
29
30 css
31 Copy code
32 for (int i = 0; i < 3; ++i) {
33 arr[i]->draw();
34 }
35
36 for (int i = 0; i < 3; ++i) {
37 delete arr[i];
38 }
39
40 return 0;
41 }
```

12. Implement polymorphism in C++ by creating an abstract base class `Shape` with a pure virtual function `area()`. Create two derived classes, `Circle` and `Rectangle`. `Circle` should override `area()` to calculate the area of a circle given a radius, and `Rectangle` should override `area()` to calculate the area of a rectangle given a length and width.
13. Create an abstract base class `Animal` with a pure virtual function `makeSound()`. Derive two classes from `Animal`: `Dog` and `Cat`. The `Dog` class should override `makeSound()` to print "Woof!" and the `Cat` class should override `makeSound()` to print "Meow!".
14. Define an abstract base class `Vehicle` with a pure virtual function `speed()`. Create two derived classes, `Car` and `Bike`. `Car` should have a data member for the number of doors and override `speed()` to return a speed specific to cars. `Bike` should have a data member for the type of bike and override `speed()` to return a speed specific to bikes.
15. Implement a simple banking system with a base class `Account` and two derived classes, `SavingsAccount` and `CheckingAccount`. The base class should have a virtual function `getAccountType()` and the derived classes should override this function to return "Savings" and "Checking" respectively.
16. Implement polymorphism by creating an abstract class `Employee` with a pure virtual function `calculateSalary()`. Create two derived classes, `FullTimeEmployee` and `PartTimeEmployee`. `FullTimeEmployee` should override `calculateSalary()` to calculate a fixed monthly salary, while `PartTimeEmployee` should override `calculateSalary()` to calculate an hourly wage based on hours worked.
17. Define a base class `Transport` with a virtual function `move()`. Create derived classes `Airplane` and `Train`. `Airplane` should override `move()` to print "Flying in the sky" and `Train` should override `move()` to print "Running on tracks".
18. Implement polymorphism with a base class `Game` having a pure virtual function `play()`. Create two derived classes, `Chess` and `Soccer`. `Chess` should override `play()` to print "Playing chess", while `Soccer` should override `play()` to print "Playing soccer".
19. Create a class hierarchy with a base class `Employee` and derived classes `Manager` and `Engineer`. The base class should have a virtual function `getRole()`. `Manager` should override `getRole()` to return "Manager", and `Engineer` should override `getRole()` to return "Engineer".
20. Define a base class `Person` with a virtual function `greet()`. Create derived classes `Teacher` and `Student`. `Teacher` should override `greet()` to say "Hello, class!" and `Student` should override `greet()` to say "Hi, everyone!".

## 17. Templates

### 17.1 Background

Before we introduce templates, we look back at a simple function which swaps the values of two variables.

Code 17.1: Function swapValues() example

```
1 #include <iostream>
2 using namespace std;
3
4 void swapValues(int& one, int& two) {
5 int temp = one;
6 one = two;
7 two = temp;
8 } // end swap
9
10 int main(){
11 int first=3, second=5;
12 cout << "Before swap." << endl;
13 cout << "first = " << first << endl;
14 cout << "second = " << second << endl << endl;
15
16 swapValues(first, second);
17
18 cout << "After swap." << endl;
19 cout << "first = " << first << endl;
20 cout << "second = " << second << endl << endl;
21
22 return 0;
23 } // end main
```



Change the **swapValues()** definition to work with two **double** values. How about two **char** values?



Add the following class definition to the program and make any necessary changes for the function to swap two Person objects.

Code 17.2: Class Person definition

```
1 class Person {
2 private:
3 string name;
4 public:
5 Person(string n):name(n) {}
6 string getName() {return name;}
7 };
```



What could we do if we want all of the above examples to work in the same program?



## 17.2 Function Templates

C++ templates allow you to write generic code that works with various data types. By using templates, you can define functions and classes that operate on different types without rewriting code. This chapter explores the basics of function templates, class templates, and advanced concepts like template specialization and constraints.

Code 17.3: First function template example

```
1 #include <iostream>
2 using namespace std;
3
4 template <typename T>
5 void swapValues(T& one, T& two) {
6 T temp = one;
7 one = two;
8 two = temp;
9 }
10
11 int main(){
12
13 int first=3, second=5;
14 cout << "Before swap." << endl;
15 cout << "first = " << first << endl;
16 cout << "second = " << second << endl << endl;
17
18 swapValues(first, second);
19
20 cout << "After swap." << endl;
21 cout << "first = " << first << endl;
22 cout << "second = " << second << endl << endl;
23
24 return 0;
25 } // end main
```

In the example above, we use the template keyword to define a template parameter ‘T’. This parameter acts as a placeholder for any data type. When the function is called with specific types (e.g., ‘int’, ‘double’, ‘char’, etc.), the compiler generates the correct version of the function.



Do we need to use T for the typename?



Change the keyword **typename** with the keyword **class** and run the program. Does it work?



Change **main()** to test other types of values and run the program. Do not change the definition of **swapValues()**. Does it work?



Can we separate the template function declaration from its definition?

Write a function template named **largest()**. The function takes three values of the same type as arguments and returns the largest of the three. The function should also work if the values are the same. Write a program to test the function for different values. A sample output is given below as a reference.



**Desired output format:**

Values are: 4 4 3

The largest value is: 4



Consider the **Rectangle** class definition below. Make all the necessary changes to make the **largest()** function template work with **Rectangle** types in a meaningful way.

Code 17.4: Rectangle class definition

```

1 /***** Declaration Section *****/
2 class Rectangle{
3 private:
4 int width , length ;
5 public:
6 Rectangle ();
7 Rectangle (int w, int l);
8 Rectangle operator+(Rectangle& r);
9 };
10 /***** Implementation Section *****/
11 Rectangle::Rectangle() { width = 1, length = 2; }
12 Rectangle::Rectangle(int w, int l) { width = w, length = l; }
13 Rectangle Rectangle::operator+(Rectangle& r) {
14 Rectangle temp;
15 temp.width = this->width + r.width;
16 temp.length = this->length + r.length;
17 return temp;
18 }

```

## 17.3 Template Specialization and Function Overloading

While template functions are designed to be generic, there are cases where you need specific behavior for certain data types. This is where template specialization comes in. Specialization allows you to define a specific implementation for a particular type.

Here's an example of a specialized 'swapValues()' function for strings:

Code 17.5: Specialization for string type

```
1 template <>
2 void swapValues<string>(string& one, string& two) {
3 // Specific behavior for strings
4 string temp = "Special swap: " + one;
5 one = "Special swap: " + two;
6 two = temp;
7 }
```

Function overloading with templates is another technique where you define multiple versions of a function, with different parameter lists. This can be useful to handle specific cases that require different behaviors.

Code 17.6: Another example:

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 // General template function
6 template <typename T>
7 T square(T x) { return x * x; }
8
9 // Overloaded template function
10 template <>
11 string square<string>(string x) {
12 return x + x; // "square" for strings could mean doubling the string
13 }
14
15 int main() {
16 int intValue = 4;
17 double doubleValue = 3.5;
18 string stringValue = "Hello";
19
20 // Using the general template function
21 cout << "Square of " << intValue << " is " << square(intValue) << endl;
22 cout << "Square of " << doubleValue << " is " << square(doubleValue) << endl;
23
24 // Using the overloaded template function for strings
25 cout << "Square of \" << stringValue << \" is \" << square(stringValue) << endl;
26 return 0;
27 }
```

## 17.4 Class Templates

Class templates are similar to function templates but allow you to create generic classes. This is useful for creating data structures and objects that work with various types. The syntax for defining class templates is like that for function templates.

Here's an example of a simple class template for a pair of values:

Code 17.7: Class template example

```

1 #include <iostream>
2 using namespace std;
3
4 template <typename T>
5 class Pair {
6 public:
7 Pair(T firstValue , T secondValue)
8 : first(firstValue), second(secondValue) {}
9 void printPair() const {
10 cout << "(" << first << ", " << second << ")";
11 }
12
13 private:
14 T first;
15 T second;
16 };
17
18 int main() {
19 Pair<int> intPair(3, 5);
20 Pair<char> charPair('a', 'b');
21
22 cout << "Integer pair: ";
23 intPair.printPair();
24
25 cout << "\nCharacter pair: ";
26 charPair.printPair();
27
28 return 0;
29 }

```



Give the definition for the default Pair constructor.



Define a non-member template function `T addValues(Pair<T> & myPair)` which returns the sum of the two values in the pair. Add any necessary components to the class `Pair`. Test the function from `main` for different value types.

## 17.5 Practice Questions

### 1. True-False questions – Circle your choice

- (a) (**True - False**) Function templates allow you to create generic functions that work with multiple data types.
- (b) (**True - False**) Class templates can have template specialization, allowing specific implementations for certain types.
- (c) (**True - False**) The keyword 'typename' in template declarations is equivalent to 'class'.
- (d) (**True - False**) You can overload template functions just like regular functions in C++.
- (e) (**True - False**) Class templates cannot have non-template member functions.
- (f) (**True - False**) Template specialization allows you to define a different implementation for a specific type, while keeping the general template intact.
- (g) (**True - False**) Templates cannot be used with standard C++ containers like 'std::vector'.
- (h) (**True - False**) When using class templates, you need to specify the type of the template when creating an object.
- (i) (**True - False**) Template function overloading can occur when different numbers of template parameters are used.

2. Multiple-choice questions – Choose the correct answer

- (a) What is the primary benefit of using templates in C++?
  - (i) Allows code reuse with different data types.
  - (ii) Increases compile-time safety.
  - (iii) Reduces code size.
  - (iv) All of the above.
- (b) What is required to use a class template in your C++ program?
  - (i) A specific type must be provided when creating an object.
  - (ii) The template must be defined in a separate header file.
  - (iii) The compiler must support C++17 or later.
  - (iv) A template specialization must be defined.
- (c) Which statement is true about template specialization?
  - (i) It allows specific behavior for certain types while retaining general template functionality.
  - (ii) It requires a separate definition file.
  - (iii) It is used to optimize code at runtime.
  - (iv) It prevents the use of other templates in the same file.
- (d) Which of the following operations can be performed using class templates?
  - (i) Creating generic data structures.
  - (ii) Defining functions for different data types.
  - (iii) Template specialization.
  - (iv) All of the above.
- (e) What does template function overloading allow you to do?
  - (i) Define multiple versions of a template function with different parameter lists.
  - (ii) Use templates with different data structures.
  - (iii) Overload regular functions with template functions.
  - (iv) Overload template functions with different numbers of template parameters.
- (f) When using templates, what is template instantiation?
  - (i) The process of generating code for specific types at compile-time.
  - (ii) The process of creating objects from a template.
  - (iii) The process of specializing a template for a specific type.
  - (iv) None of the above.
- (g) Which of the following is a common use case for class templates in C++?
  - (i) Creating container classes.
  - (ii) Defining mathematical operations.
  - (iii) Implementing custom algorithms.
  - (iv) All of the above.

3. Predict the output – Write the expected output for the following code snippets

(a)

```
1 #include <iostream>
2 using namespace std;
3
4 template <typename T>
5 void swapValues(T& a, T& b) {
6 T temp = a;
7 a = b;
8 b = temp;
9 }
10
11 int main() {
12 int x = 10, y = 20;
13 swapValues(x, y);
14 cout << "x = " << x << ", y = " << y << endl;
15
16 return 0;
17 }
```

(b)

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 template <typename T>
6 T square(T x) {
7 return x * x;
8 }
9
10 int main() {
11 cout << square(5) << endl;
12 cout << square(3.5) << endl;
13 cout << square(string("A")) << endl;
14
15 return 0;
16 }
```

(c)

```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 template <typename T>
6 void printVector(const vector<T>& v) {
7 for (const auto& elem : v) {
8 cout << elem << " ";
9 }
```

```
10 cout << endl;
11 }
12
13 int main() {
14 vector<int> vec = {1, 2, 3, 4};
15 printVector(vec);
16
17 return 0;
18 }
```

(d)

```
1 #include <iostream>
2 using namespace std;
3
4 template <typename T>
5 T add(T a, T b) {
6 return a + b;
7 }
8
9 int main() {
10 cout << add(3, 4) << endl;
11 cout << add(2.5, 3.7) << endl;
12
13 return 0;
14 }
```

(e)

```
1 #include <iostream>
2 using namespace std;
3
4 template <typename T>
5 void printValue(const T& value) {
6 cout << "Value: " << value << endl;
7 }
8
9 int main() {
10 printValue(42);
11 printValue("Hello");
12 printValue(3.14);
13
14 return 0;
15 }
```

(f)

```
1 #include <iostream>
2 using namespace std;
3
4 template <typename T>
5 T multiply(T a, T b) {
```



```
6 return a * b;
7 }
8
9 int main() {
10 cout << multiply(3, 4) << endl;
11 cout << multiply(1.5, 2.5) << endl;
12
13 return 0;
14 }
```

(g)

```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 template <typename T>
6 void reverseVector(vector<T>& v) {
7 std::reverse(v.begin(), v.end());
8 }
9
10 int main() {
11 vector<int> vec = {1, 2, 3, 4, 5};
12 reverseVector(vec);
13 for (const auto& elem : vec) {
14 cout << elem << " ";
15 }
16 cout << endl;
17
18 return 0;
19 }
```

(h)

```
1 #include <iostream>
2 using namespace std;
3
4 template <typename T>
5 void doubleValue(T& value) {
6 value *= 2;
7 }
8
9 int main() {
10 int x = 10;
11 doubleValue(x);
12 cout << "x = " << x << endl;
13
14 return 0;
15 }
```

(i)

```
1 #include <iostream>
2 using namespace std;
3
4 template <typename T>
5 T subtract(T a, T b) {
6 return a - b;
7 }
8
9 int main() {
10 cout << subtract(10, 5) << endl;
11 cout << subtract(7.5, 2.3) << endl;
12
13 return 0;
14 }
```

(j)

```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 template <typename T>
6 void clearVector(vector<T>& v) {
7 v.clear();
8 }
9
10 int main() {
11 vector<int> vec = {1, 2, 3, 4, 5};
12 clearVector(vec);
13 cout << "Vector size: " << vec.size() << endl;
14
15 return 0;
16 }
```

---

4. Write code – Write the following C++ programs

- (a) Write a C++ template function that returns the minimum of three values of any type. Test this function with integers, doubles, and characters to ensure it works correctly.
- (b) Define a class template 'Box' that can hold an object of any type. Add methods to get and set the value inside the 'Box'. Test this class with different data types.
- (c) Write a template function 'multiplyByTwo()' that multiplies a given value by two. Test this function with integers, doubles, and strings (by repeating the string twice).
- (d) Define a class template 'Point' that represents a point in 2D space with 'x' and 'y' coordinates. Add a method to calculate the distance to another 'Point'. Test this class with different types like 'int' and 'double'.
- (e) Write a template function 'findLargest()' that takes a container (e.g., 'std::vector') and returns the largest value in the container. Test this function with different container types and values.
- (f) Define a template function 'reverseArray()' that reverses an array of any type. Test this function with arrays of integers, doubles, and characters.
- (g) Write a template function 'addAll()' that takes a container (e.g., 'std::vector') and returns the sum of all elements in the container. Test this function with different data types, ensuring it works for numerical types.
- (h) Write a C++ template function that takes two generic values and swaps them. Test this function with integers, doubles, and characters to ensure it works as expected.



## 18. Linked Data Structures

### 18.1 Linked Lists

Linked Lists are a fundamental data structure in computer science, providing a flexible and dynamic way to store and manage collections of data. Unlike arrays, linked lists do not have a fixed size, allowing them to grow or shrink as needed. The elements in a linked list, known as nodes, are connected through pointers, which create a chain-like structure.

Before introducing a full example of a linked list, let's start with a basic program that demonstrates the use of structs and pointers to create and manipulate a single node.

Code 18.1: Starting example

```
1 #include <iostream>
2 using namespace std;
3
4 struct Node {
5 int value;
6 Node *next;
7 };
8
9 int main() {
10 Node *one = new Node;
11 one->next = nullptr; // Explicitly setting the next pointer to null
12 one->value = 1;
13 cout << one->value << endl;
14
15 delete one; // Always deallocate memory to avoid memory leaks
16 }
```

This simple example creates a single node with a value of '1' and a pointer to the next node, which is set to 'nullptr'. The use of 'nullptr' indicates that there is no subsequent node, effectively marking the end of the list.

Let's explore a couple of questions related to this example:



What is the default access mode for variables in a struct, and how does it differ from a class?



Why is it useful to set a pointer to 'nullptr' after initializing a node, and why is it important to deallocate memory?

Below is an example of a linked list in C++, demonstrating the creation of nodes, linking them together, traversing the list, and printing the values.

Code 18.2: Linked list first example

```
1 #include <iostream>
2 using namespace std;
3
4 // Linked list node
5 struct Node {
6 int value;
7 Node* next;
8
9 // Constructor with default values
10 Node(int v = 0, Node* n = nullptr) : value(v), next(n) {}
11 };
12
13 int main() {
14 // Creating linked list nodes
15 Node* one = new Node(1, nullptr);
16 Node* two = new Node(5, nullptr);
17 Node* three = new Node(6, nullptr);
18
19 // Linking the nodes
20 one->next = two; // Connecting first node to second
21 two->next = three; // Connecting second node to third
22
23 // Traversing the list and printing values
24 Node* current = one;
25 while (current != nullptr) {
26 cout << current->value << " ";
27 current = current->next;
28 }
29 cout << endl;
30
31 // Clean up memory
32 delete one;
33 delete two;
34 delete three;
35
36 return 0;
37 }
```



← Get code here.

## 18.2 Linked Lists using Classes

Next we extend our discussion to include classes, and add some more components to each node. Each Node will now be an object of a class.

Using classes for linked lists in C++ provides several benefits over structs, particularly in terms of encapsulation and method-based interactions. When linked lists are implemented using classes, you can control access to internal data, define clear methods for manipulating the list, and potentially include additional logic to maintain integrity.

A class-based implementation allows us to encapsulate the nodes' internals and introduce getter and setter methods to interact with them. This makes the code more maintainable and reduces the risk of accidental data modification or corruption.

Here's an example of a linked list implemented with classes. Each node is represented by a class with a value and a pointer to the next node. The code snippet demonstrates creating a list of three nodes, linking them, traversing the list, and then cleaning up memory at the end to avoid memory leaks.



Code 18.3: Linked list example using classes

```
1 #include <iostream>
2 using namespace std;
3
4 // Class representing a node in a linked list
5 class Node {
6 private:
7 int value;
8 Node* next;
9 public:
10 Node(int v = 0, Node* n = nullptr) : value(v), next(n) {}
11
12 int getValue() const { return value; }
13
14 Node* getNext() const { return next; }
15
16 void setValue(int v) { value = v; }
17
18 void setNext(Node* n) { next = n; }
19 };
20
21 int main() {
22 // Create a linked list with three nodes
23 Node* one = new Node(1);
24 Node* two = new Node(5);
25 Node* three = new Node(6);
26
27 // Link the nodes together
28 one->setNext(two);
29 two->setNext(three);
30
31 // Traverse the list and print the values
32 Node* current = one;
33 while (current != nullptr) {
34 cout << current->getValue() << " ";
35 current = current->getNext();
36 }
37 cout << endl;
38
39 delete one;
40 delete two;
41 delete three;
42
43 return 0;
44 }
```



← Get code here.

### 18.3 Adding and Removing Nodes

In this section, we show how to add nodes at the beginning of a linked list. The code snippet creates a linked list using classes, includes a method for inserting at the head, and provides a way to clean up memory.

Code 18.4: Adding Nodes at the Beginning of the List

```

1 #include <iostream>
2 using namespace std;
3
4 class Node {
5 private:
6 int value;
7 Node* next;
8 public:
9 Node(int v, Node* n = nullptr) : value(v), next(n) {}
10 int getValue() const { return value; }
11 Node* getNext() const { return next; }
12 void setValue(int v) { value = v; }
13 void setNext(Node* n) { next = n; }
14 };
15
16 void printList(Node* head) {
17 Node* current = head;
18 cout << "List content: ";
19 while (current != nullptr) {
20 cout << current->getValue() << " ";
21 current = current->getNext();
22 }
23 cout << endl;
24 }
25
26 void insertHead(Node*& head, int value) {
27 head = new Node(value, head);
28 }
29
30 int main() {
31 Node* head = nullptr; // Start with an empty list
32 printList(head);
33 insertHead(head, 6);
34 printList(head);
35 insertHead(head, 5);
36 printList(head);
37 insertHead(head, 1);
38 printList(head);
39
40 Node* current = head;
41 while (current != nullptr) {
42 Node* temp = current;
43 current = current->getNext();
44 delete temp;
45 }
46 return 0;
47 }

```





Add a function `insertAfter(Node* after, int v)` to the program above. This function should insert a new node with the given value `v` after the specified node `after`. Ensure the linked list is updated accordingly. Test the new function in `main()` by inserting nodes at various positions within the list.



Add a function `removeAfter(Node* after)` to the program above. This function should remove the node that comes immediately after the given node `after`. Ensure memory is properly deallocated to avoid leaks. Test this function in `main()` by removing specific nodes and then printing the updated list to verify the results.



Add a function `find(Node* head, int key)` to the program above. This function should search the linked list starting from the `head` for a node containing the given `key`. It should return `true` if a node with the `key` is found, and `false` otherwise. Test the function from `main()` by searching for existing and non-existing values in the list, and then print the results.

## 18.4 LinkedList class

In this section, we start by introducing a new LinkedList class and gradually build upon it by adding various functionalities. These include methods for inserting nodes at the head of the list, inserting nodes after a specific node, and removing nodes after a specific node. We also add a method to search for a node with a specific value in the list. By the end of this section, you will have a robust LinkedList class that can be used in a variety of applications.

Code 18.5: LinkedList class example

```
1 #include <iostream>
2 using namespace std;
3
4 class Node {
5 private:
6 int value;
7 Node* next;
8 public:
9 Node(int v, Node* n = nullptr) : value(v), next(n) {}
10 int getValue() const { return value; }
11 Node* getNext() const { return next; }
12 void setValue(int v) { value = v; }
13 void setNext(Node* n) { next = n; }
14 };
15
16 class LinkedList {
17 private:
18 Node* head;
19 public:
20 LinkedList() : head(nullptr) {}
21 ~LinkedList() {
22 Node* current = head;
23 while (current != nullptr) {
24 Node* temp = current;
25 current = current->getNext();
26 delete temp;
27 }
28 }
29 void insertHead(int value) {
30 head = new Node(value, head);
31 }
32 void insertAfter(Node* after, int value) {
33 if (after == nullptr) {
34 return;
35 }
36 after->setNext(new Node(value, after->getNext()));
37 }
38 void removeAfter(Node* after) {
39 if (after == nullptr || after->getNext() == nullptr) {
40 return;
```

```
41 }
42 Node* temp = after->getNext();
43 after->setNext(temp->getNext());
44 delete temp;
45 }
46 bool find(int key) {
47 Node* current = head;
48 while (current != nullptr) {
49 if (current->getValue() == key) {
50 return true;
51 }
52 current = current->getNext();
53 }
54 return false;
55 }
56 void printList() {
57 Node* current = head;
58 cout << "List content: ";
59 while (current != nullptr) {
60 cout << current->getValue() << " ";
61 current = current->getNext();
62 }
63 cout << endl;
64 }
65 };
66
67 int main() {
68 LinkedList list;
69 list.printList();
70 list.insertHead(6);
71 list.printList();
72 list.insertHead(5);
73 list.printList();
74 list.insertHead(1);
75 list.printList();
76 return 0;
77 }
```



<— Get code here.



Expand the 'main()' function in the program above to test all the methods in the LinkedList class. For each method, perform a series of tests that demonstrate its functionality and edge cases. After each operation, print the updated list to verify the results.



Add a function 'insertTail(int value)' to the LinkedList class in the program above. This function should insert a new node with the given value 'value' at the end of the list. Test this function in 'main()' by inserting nodes at the tail and then printing the updated list to verify the results.



Add a function 'size()' to the LinkedList class in the program above. This function should return the number of nodes in the list. Test this function in 'main()' by calling it after various operations and then printing the returned size to verify the results.



Add a function 'size()' to the LinkedList class in the program above. This function should return the number of nodes in the list. Test this function in 'main()' by calling it after various operations and then printing the returned size to verify the results.



Add a function 'isEmpty()' to the LinkedList class in the program above. This function should return 'true' if the list is empty and 'false' otherwise. Test this function in 'main()' by calling it on an empty list and a non-empty list, and then print the returned value to verify the results.

## 18.5 Practice Questions

1. True-False questions – Circle your choice

- (a) (**True - False**) A linked list has a fixed size that must be defined at compile-time.
- (b) (**True - False**) The node structure in a linked list typically contains a value and a pointer to the next node in the list.
- (c) (**True - False**) Inserting a new node at the beginning of a linked list changes the head pointer to the newly inserted node.
- (d) (**True - False**) When removing a node from a linked list, it is important to update the 'next' pointer of the previous node.
- (e) (**True - False**) A memory leak can occur if a node is removed from a linked list without deallocating its memory.
- (f) (**True - False**) In a singly linked list, each node has pointers to both the previous and the next node.
- (g) (**True - False**) Inserting a node at the end of a linked list always requires iterating through the entire list.
- (h) (**True - False**) A linked list's memory is automatically cleaned up when the program exits, so there's no need to manually delete nodes.



## 2. Multiple-choice questions – Choose the correct answer

- (a) Which of the following statements is true about linked lists?
  - (i) Linked lists can dynamically grow or shrink during program execution.
  - (ii) Linked lists are designed to maintain a fixed size.
  - (iii) Linked lists use indices like arrays for random access.
  - (iv) None of the above.
- (b) What is the primary advantage of linked lists over arrays?
  - (i) Linked lists offer better cache performance.
  - (ii) Linked lists allow fast random access.
  - (iii) Linked lists use less memory.
- (c) What happens when inserting a node at the beginning of a linked list?
  - (i) The head pointer remains the same.
  - (ii) The last node of the list becomes the new head.
  - (iii) The newly inserted node becomes the head of the list.
  - (iv) None of the above.
- (d) Which of the following is a potential risk when working with linked lists?
  - (i) Memory leaks if nodes are not properly deallocated.
  - (ii) Data loss if pointers are mismanaged.
  - (iii) Circular references causing infinite loops.
  - (iv) All of the above.
- (e) What is the role of the 'next' pointer in a linked list node?
  - (i) It points to the next node in the list.
  - (ii) It points to the previous node in the list.
  - (iii) It points to the head node.
  - (iv) It points to the tail node.

3. Predict the output – Write the expected output for the following code snippet

Code 18.6: Linked list output prediction

```
1 #include <iostream>
2 using namespace std;
3
4 class Node {
5 private:
6 int value;
7 Node* next;
8 public:
9 Node(int v, Node* n = nullptr) : value(v), next(n) {}
10
11 int getValue() const { return value; }
12 Node* getNext() const { return next; }
13
14 void setNext(Node* n) { next = n; }
15 };
16
17 void insertHead(Node*& head, int value) {
18 head = new Node(value, head);
19 }
20
21 void printList(Node* head) {
22 Node* current = head;
23 while (current != nullptr) {
24 cout << current->getValue() << " ";
25 current = current->getNext();
26 }
27 cout << endl;
28 }
29
30 int main() {
31 Node* head = nullptr;
32 insertHead(head, 1);
33 insertHead(head, 2);
34 insertHead(head, 3);
35 printList(head);
36 Node* middle = head->getNext();
37 insertAfter(middle, 4);
38 printList(head);
39 removeAfter(middle); // Remove the node after the second node
40 printList(head);
41
42 return 0;
43 }
```

- 
4. Write code – Write the following C++ programs
- (a) Write a C++ program that creates a linked list, then adds nodes at the beginning, middle, and end. Implement functions to insert, remove, and find nodes. Ensure proper memory management by deallocating the list at the end.



## 19. Exception Handling

### 19.1 Introduction

Exception handling in C++ provides a mechanism to react to exceptional circumstances (like runtime errors) in programs by transferring control to special functions called handlers. This chapter will cover the basics of exception handling in C++, along with some advanced topics.

Code 19.1: A simple example

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5 int friends = 0;
6 double money = 0.0;
7
8 cout << "How many friends do you have? ";
9 cin >> friends;
10 cout << "How much money do you have? ";
11 cin >> money;
12
13 double average = money/friends;
14 cout << "Each friend will get $" << average;
15 return 0;
16 }
```



Under which circumstances we will get an error and how we can deal with it?

Code 19.2: Dealing with error example

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5 int friends=0;
6 double money = 0.0, average = 0.0;
7
8 cout << "How many friends do you have? ";
9 cin >> friends;
10 cout << "How much money do you have? ";
11 cin >> money;
12
13 if(friends <= 0){
14 cout << "You need to find a friend."<< endl;
15 }
16 else {
17 average = money/friends;
18 cout << "Each friend will get $" << average;
19 }
20 return 0;
21 }
```

The potential error in the example above is easily predictable, so we can handle it in a simple way as we did. We will use the same scenario to illustrate the exception handling mechanism in C++. This use is not necessary in this example but is a good first example to explain how it works.

## 19.2 Deep Dive into Exception Handling

### Types of Exceptions

In C++, exceptions can be of any data type. This includes basic data types (like 'int' or 'double'), as well as any user-defined types. The C++ Standard Library provides a base class specifically designed to declare objects to deal with exceptions. In addition, programmers can define their own exception classes by inheriting and overriding exception class functionality.

### The try, catch, and throw Keywords

The 'try' keyword encloses the code that might throw an exception, the 'catch' keyword defines a block of code to be executed if an exception occurs in the 'try' block, and the 'throw' keyword is used to throw an exception. Here's an example:

Code 19.3: Using try, catch, and throw

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5 try {
6 throw "An error occurred!";
7 }
8 catch(const char* e) {
9 cout << "Exception Caught: " << e << endl;
10 }
11 return 0;
12 }
```

In this example, the string "An error occurred!" is thrown as an exception, which is caught by the 'catch' block and then printed to the console.

Next we use a modified version of our previous example to illustrate the use of the try-catch block.

Code 19.4: Using try and catch blocks

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5 int friends=0;
6 double money = 0.0, average = 0.0;
7
8 try {
9 cout << "How many friends do you have? ";
10 cin >> friends;
11 cout << "How much money do you have? ";
12 cin >> money;
13
14 if (friends <= 0)
15 throw money;
16
17 average = money/friends;
18 cout << "Each friend will get $" << average;
19 }
20
21 catch(double e){
22 cout << "$" << e;
23 cout << " will not make you happy if you have no friends.";
24 }
25 return 0;
26 }
```



## 19.3 Exploring Different Types of Exceptions

### Standard Exceptions

The C++ Standard Library provides several standard exceptions, defined in ‘<exception>’, that we can use in our programs. These are derived from the ‘std::exception’ class, and include ‘std::runtime\_error’, ‘std::logic\_error’, and many others.

Code 19.5: Using standard exceptions

```
1 #include <iostream>
2 #include <stdexcept> // Include for std::runtime_error
3 using namespace std;
4
5 int main() {
6 try {
7 throw runtime_error("A runtime error occurred!");
8 }
9 catch(runtime_error& e) {
10 cout << "Exception Caught: " << e.what() << endl;
11 }
12 return 0;
13 }
```

In this example, a ‘std::runtime\_error’ is thrown and caught. The ‘what()’ function returns a C-style character string describing the error.

### User-Defined Exceptions

In addition to the standard exceptions, C++ allows us to define our own exception classes. These can be useful for throwing and catching exceptions that are specific to our application. Here's an example:

Code 19.6: Using user-defined exceptions

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 // Define a new exception class
6 class MyException : public exception {
7 public:
8 MyException(const string& message) : msg(message) {}
9 const char* what() const noexcept override { return msg.c_str(); }
10 private:
11 string msg;
12 };
13
14 int main() {
15 try {
16 throw MyException("My exception occurred!");
17 }
18 catch (MyException& e) {
19 cout << "Exception Caught: " << e.what() << endl;
20 }
21 return 0;
22 }
```

In this example, we define a new exception class 'MyException' that inherits from 'std::exception'. We override the 'what()' function to return our own error message.

## 19.4 Handling Multiple Exceptions

In C++, a single ‘try’ block can have multiple ‘catch’ blocks to handle different types of exceptions. This allows us to write code that can handle various error conditions in a more specific manner.

Code 19.7: Handling multiple exceptions

```

1 #include <iostream>
2 using namespace std;
3
4 class NegativeNumber {
5 public:
6 NegativeNumber(const string& theMessage): message(theMessage) {}
7 string getMessage() const {return message;}
8 private:
9 string message;
10 };
11 class DivideByZero {};
12
13 int main() {
14 int pencils, erasers;
15 double ppe;
16 try{
17 cout << "How many pencils do you have?\n";
18 cin >> pencils;
19 if(pencils < 0)
20 throw NegativeNumber("Cannot have a negative number of pencils.");
21 cout << "How many erasers do you have?\n";
22 cin >> erasers;
23 if(erasers < 0)
24 throw NegativeNumber("Cannot have a negative number of erasers.");
25 if(erasers == 0)
26 throw DivideByZero();
27 ppe = static_cast<double>(pencils) / erasers;
28 cout << "Each eraser must last through " << ppe << " pencils.\n";
29 }
30 catch(const NegativeNumber& e){
31 cout << e.getMessage() << endl;
32 }
33 catch(const DivideByZero&){
34 cout << "Number of erasers cannot be zero.\n";
35 }
36 catch(...){
37 cout << "An unexpected error occurred." << endl;
38 }
39 cout << "End of program.\n";
40 return 0;
41 }

```



← Get code here.

## 19.5 Throwing an Exception in a Function

Code 19.8: From book. Throwing an exception in a function.

```
1 #include <iostream>
2 #include <cstdlib>
3 using namespace std;
4
5 class DivideByZero {};
6
7 double safeDivide(int top, int bottom) throw (DivideByZero);
8
9 int main() {
10 int numerator, denominator;
11 double quotient;
12
13 cout << "Enter numerator: \n";
14 cin >> numerator;
15 cout << "Enter denominator: \n";
16 cin >> denominator;
17
18 try {
19 quotient = safeDivide(numerator, denominator);
20 }
21
22 catch(DivideByZero){
23 cout << "Error. Division by Zero.\n";
24 exit(0);
25 }
26
27 cout << "Result = " << quotient << endl;
28
29 cout << "End of program.\n";
30 return 0;
31 }
32
33 double safeDivide(int top, int bottom) throw (DivideByZero){
34 if(bottom == 0)
35 throw DivideByZero();
36 return top / (double) bottom;
37 }
```



← Get code here.

## 19.6 One More Example

Code 19.9: Using the programmer defined exception class

```
1 #include <iostream>
2 #include <stdexcept>
3
4 using namespace std;
5
6 class DivideByZeroException : public runtime_error {
7 public:
8 DivideByZeroException() : runtime_error("Division by zero") {}
9 };
10
11 class DivideByZeroDemo {
12 private:
13 int numerator, denominator;
14 double quotient;
15
16 public:
17 void doIt();
18 void giveSecondChance();
19 };
20
21 void DivideByZeroDemo::doIt() {
22 try {
23 cout << "Enter numerator: ";
24 cin >> numerator;
25 cout << "Enter denominator: ";
26 cin >> denominator;
27
28 if (denominator == 0)
29 throw DivideByZeroException();
30
31 quotient = static_cast<double>(numerator) / denominator;
32 cout << numerator << "/" << denominator << " = " << quotient << endl;
33 }
34 catch (DivideByZeroException& e) {
35 cout << e.what() << endl;
36 giveSecondChance();
37 }
38 cout << "End of program." << endl;
39 }
40
41 void DivideByZeroDemo::giveSecondChance() {
42 cout << "Try again:" << endl;
43 cout << "Enter numerator: ";
44 cin >> numerator;
45 cout << "Enter denominator: " << endl;
46 cout << "Be sure the denominator is not zero." << endl;
```

```
47 cin >> denominator;
48
49 if (denominator == 0) {
50 cout << "I cannot do division by zero." << endl;
51 cout << "Since I cannot do what you want," << endl;
52 cout << "the program will now end." << endl;
53 exit(0);
54 }
55 quotient = static_cast<double>(numerator) / denominator;
56 cout << numerator << "/" << denominator << " = " << quotient << endl;
57 }
58
59 int main() {
60 DivideByZeroDemo oneTime;
61 oneTime.doIt();
62 return 0;
63 }
```



← Get code here.

## 19.7 Practice Questions

1. True-False questions – Circle your choice
  - (a) (**True - False**) Exception handling in C++ is achieved through the use of try, catch, and finally blocks.
  - (b) (**True - False**) If no catch block matches a thrown exception, the program terminates by calling 'std::terminate()'.
  - (c) (**True - False**) It is possible to throw an exception of any type in C++, including user-defined types.
  - (d) (**True - False**) The 'std::exception' class is the base class for all standard exceptions in the C++ Standard Library.
  - (e) (**True - False**) The 'catch(...)' block can catch any type of exception.
2. Multiple-choice questions – Choose the correct answer
  - (a) Which of the following statements is true about exception handling in C++?
    - (i) Exceptions are used for handling errors and other exceptional circumstances.
    - (ii) Exceptions must always be derived from 'std::exception'.
    - (iii) Exceptions are handled only by the operating system.
    - (iv) None of the above.
  - (b) What happens if an exception is thrown but not caught in a C++ program?
    - (i) The program continues execution from the point of the exception.
    - (ii) The program terminates by calling 'std::abort()'.
    - (iii) The program terminates by calling 'std::terminate()'.
    - (iv) The program enters an infinite loop.
  - (c) When should you use the 'throw' keyword in C++?
    - (i) To define a function that may throw exceptions.
    - (ii) To catch exceptions in a catch block.
    - (iii) To signal the occurrence of an error or exceptional condition.
    - (iv) To handle errors within a try block.

3. Predict the output – Write the expected output for the following code snippet

Code 19.10: Exception handling output prediction

```
1 #include <iostream>
2 #include <stdexcept>
3 using namespace std;
4
5 void mightGoWrong() {
6 bool error1 = true;
7 bool error2 = false;
8
9 if(error1) {
10 throw runtime_error("Error 1 occurred");
11 }
12
13 if(error2) {
14 throw runtime_error("Error 2 occurred");
15 }
16 }
17
18 void useMightGoWrong() {
19 mightGoWrong();
20 }
21
22 int main() {
23 try {
24 useMightGoWrong();
25 }
26 catch(const runtime_error& e) {
27 cout << "Caught runtime_error: " << e.what() << endl;
28 }
29 catch(...) {
30 cout << "Caught an unknown exception" << endl;
31 }
32 cout << "Program continues..." << endl;
33 return 0;
34 }
```

4. Write code – Write the following C++ programs

- (a) Write a C++ program that defines a custom exception class called 'MyException'. This class should inherit from 'std::exception' and override the 'what()' method to return a custom error message. Write a function that throws this exception under certain conditions and demonstrate how to catch it in a try-catch block.



## 20. Standard Template Library

The Standard Template Library is a collection of C++ template classes that allows us to create commonly used data structures such as lists, vectors, stacks, queues, etc. Since these classes (often referred to as container classes) are often used, they are part of the C++ standard and can be used with any of the C++ installations.

### 20.1 Iterators

A data structure is created to store and access information in an organized and efficient manner. One way of accessing and using this information is by using iterators. Iterators allow us to cycle through the data to obtain any necessary information stored in the data structure.

Previously we discussed Linked Lists. We used pointers to visit each node (or a subset of nodes) as needed for various operations on a Linked List. The pointer in this case would be one example of an iterator.

## 20.2 Vectors

Vectors are very similar to arrays. The main difference is the ability of vectors to change size during the execution of a program which is not true for arrays. Vectors are created by using a template from the STL. An iterator is available to allow access and manipulation of data in a given vector. The example below tests some of the basic functions of a vector data structure.

Code 20.1: From book. Vector example

```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 int main() {
6 vector<int> container;
7
8 for (int i = 1; i <= 4; i++)
9 container.push_back(i);
10
11 cout << "Here is what is in the container:\n";
12 vector<int>::iterator p;
13 for (p = container.begin(); p != container.end(); p++)
14 cout << *p << " ";
15 cout << endl;
16
17 cout << "Setting entries to 0:\n";
18
19 for (p = container.begin(); p != container.end(); p++)
20 *p = 0;
21
22 cout << "Container now contains:\n";
23 for (p = container.begin(); p != container.end(); p++)
24 cout << *p << " ";
25 cout << endl;
26
27 return 0;
28 }
```

## 20.3 Queue Iterator Example

The following example taken from your book shows an example of creating and using an iterator for a queue class. All separate content can be merged as is into one file or can be saved into separate files by adding the appropriate headers. The entire merged code can be obtained by following the link below:



← Get code here.

Code 20.2: From book. Node class declaration with inline definitions

```

1 template<class T>
2 class Node {
3 public:
4 Node(T theData , Node<T>* theLink): data(theData), link(theLink) {}
5 Node<T>* getLink() const {return link;}
6 const T& getData() const {return data;}
7 void setData (const T& theData) {data = theData;}
8 void setLink(Node<T>* pointer) {link = pointer;}
9 private:
10 T data;
11 Node<T> *link;
12
13 };

```

Code 20.3: From book. List Iterator class declaration with inline definitions

```

1 template<class T>
2 class ListIterator{
3 public:
4 ListIterator() : current(NULL){}
5 ListIterator(Node<T>* initial) : current(initial){}
6 const T& operator *() const {return current->getData();}
7
8 ListIterator& operator ++(){
9 current = current->getLink();
10 return *this;
11 }
12
13 ListIterator& operator ++(int){
14 ListIterator startVersion(current);
15 current = current->getLink();
16 return startVersion;
17 }
18
19 bool operator ==(const ListIterator rightSide) const {
20 return (current == rightSide.current);
21 }

```

```

22
23 bool operator !=(const ListIterator rightSide) const {
24 return (current != rightSide.current);
25 }
26
27 private:
28 Node<T> *current;
29 };

```

Code 20.4: From book. Class Queue declaration with some inline definitions

```

1 template<class T>
2 class Queue{
3 public:
4 typedef ListIterator<T> Iterator;
5 Queue();
6 Queue(const Queue<T>& aQueue);
7 Queue<T>& operator =(const Queue<T>& rightSide);
8 virtual ~Queue();
9 void add(T item);
10 T remove();
11 bool isEmpty() const;
12 Iterator begin() const { return Iterator(front); }
13 Iterator end() const { return Iterator(); }
14
15 private:
16 Node<T> *front;
17 Node<T> *back;
18 };

```

Code 20.5: From book. Main function

```

1 int main () {
2 char next, ans;
3 do{
4 Queue<char> q;
5 cout << "Enter a line of text:\n";
6 cin.get(next);
7 while(next != '\n'){
8 q.add(next);
9 cin.get(next);
10 }
11
12 cout << "You entered:\n";
13 Queue<char>::Iterator i;
14

```

```

15 for(i=q.begin(); i!=q.end(); i++)
16 cout << *i;
17 cout << endl;
18
19
20 cout << "Again?(y/n): ";
21 cin >> ans;
22 cin.ignore(10000, '\n');
23
24 } while(ans != 'n' && ans != 'N');
25
26 return 0;
27 }

```

Code 20.6: From book. Remaining function definitions from class Queue

```

1 template<class T>
2 Queue<T>::Queue(): front(NULL), back(NULL){ }
3
4 template<class T>
5 Queue<T>::Queue(const Queue<T>& aQueue){
6 if(aQueue.isEmpty()){
7 front = back = NULL;
8 }
9 else{
10 Node<T> *temp = aQueue.front;
11 back = new Node<T>(temp->getData(), NULL);
12 front = back;
13 temp = temp->getLink();
14 while(temp != NULL){
15 back->setLink(new Node<T>(temp->getData(), NULL));
16 back = back->getLink();
17 temp = temp->getLink();
18 }
19 }
20 }
21
22 template<class T>
23 Queue<T>& Queue<T>::operator = (const Queue<T>& rightSide){
24 if (front == rightSide.front)
25 return *this;
26 else {
27 T next;
28 while (! isEmpty())
29 next = remove();
30 }
31 if (rightSide.isEmpty()) {
32 front = back = nullptr;

```

```

33 return *this;
34 }
35 else {
36 Node<T> *temp = rightSide.front;
37 back = new Node<T>(temp->getData(), nullptr);
38 front = back;
39 temp = temp->getLink();
40 while (temp != nullptr)
41 {
42 back->setLink(
43 new Node<T>(temp->getData(), nullptr));
44 back = back->getLink();
45 temp = temp->getLink();
46 }
47 return *this;
48 }
49 }
50
51 template<class T>
52 Queue<T>::~~Queue(){
53 T next;
54 while (! isEmpty())
55 next = remove();
56 }
57
58 template<class T>
59 void Queue<T>::add(T item){
60 if (isEmpty())
61 front = back = new Node<T>(item, nullptr);
62 else
63 {
64 back->setLink(new Node<T>(item, nullptr));
65 back = back->getLink();
66 }
67 }
68
69 template<class T>
70 T Queue<T>::remove(){
71 if (isEmpty())
72 {
73 cout << "Error: Removing an item from an empty queue.\n";
74 exit(1);
75 }
76 T result = front->getData();
77 Node<T> *discard;
78 discard = front;
79 front = front->getLink();
80 if (front == nullptr)
81 back = nullptr;

```

```
82 delete discard;
83 return result;
84 }
85
86 template<class T>
87 bool Queue<T>::isEmpty() const{
88 return (back == NULL);
89 }
```







# Part Three - Other Topics

|    |                                |     |
|----|--------------------------------|-----|
| 21 | File Processing .....          | 291 |
| 22 | Basic Sorting Algorithms ..... | 295 |
| A  | Answers to Exercises .....     | 299 |
|    | Index .....                    | 305 |



## 21. File Processing

### 21.1 Stream Objects

The output generated by the C++ program will be lost unless it is stored somewhere in permanent memory. In this section we will discuss how to store and retrieve data to and from a file.

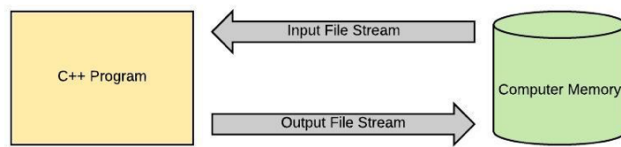
Code 21.1: Creating a new file in the current workspace.

```
1 #include <fstream>
2 using namespace std;
3
4 int main() {
5 ofstream myOutputFile("output.txt");
6
7 myOutputFile << "MAC 101" << endl;
8 myOutputFile << "Computer Science" << endl;
9 myOutputFile.close();
10
11 return 0;
12 }
```

**Note:** File stream objects support the same function calls as the cin/cout objects.



Change the output text and run the program again. What do you notice?



Code 21.2: Creating a new file in a new, specific location.

```

1 #include <fstream>
2 using namespace std;
3
4 int main() {
5 ofstream myOutputFile("c:\\TestFolder\\output2.txt");
6
7 myOutputFile << "MAC 101" << endl;
8 myOutputFile << "Computer Science" << endl;
9 myOutputFile.close();
10
11 return 0;
12 }

```



Why do we need to use double slash (\\) in the file path?

**Example:** We can use pre-defined functions to open a file, check the existence of a file and many other related operations.

Code 21.3: Open a file example.

```

1 include <fstream>
2 using namespace std;
3
4 int main() {
5 ifstream myOutputFile;
6 myOutputFile.open("output.txt");
7
8 if(myOutputFile.fail()){
9 cout << "Cannot open file !!!";
10 return -1;
11 }
12
13 cout << "The file is open.";
14 myOutputFile.close();
15 return 0;
16 }

```

**Example:** Read content from a file one page at a time and display on the console window.

Code 21.4: Read file example.

```
1 #include <iostream>
2 #include <fstream>
3 #include <string>
4 using namespace std;
5
6
7 int main() {
8 string fileName, lineOfText;
9
10 cout << "Enter a file name: ";
11 getline(cin, fileName);
12 ifstream myFile(fileName);
13
14 if (!myFile) {
15 cout << fileName << " could not be opened.";
16 return -1;
17 }
18
19 while (true) {
20 char c = 'a';
21 for (int i = 1; i <= 24 && ! myFile.eof(); i++) {
22 getline(myFile, lineOfText);
23 cout << lineOfText << endl;
24 }
25
26 if (myFile.eof())
27 break;
28 cout << "Enter 'Q' to quit.";
29 getline(cin, lineOfText);
30 c = lineOfText[0];
31 if (c == 'Q' || c == 'q')
32 break;
33 }
34 return 0;
35 }
```



- a) Create a folder PracticeFolder in the c: directory of your computer.
- b) Modify the C++ program from above to create a new file “numbers.txt”.
- c) Write in the file the first 100 positive integers, each value in a new line.
- d) Use the program to output the content of the “numbers.txt” file 10 numbers at a time.

**Example:** Append text at the end of a file.

Code 21.5: Append text example

```
1 #include <iostream>
2 #include <fstream>
3 #include <string>
4 using namespace std;
5
6 int main() {
7 string someText;
8 ofstream myFile("records.txt", ios::app);
9
10
11 if (!myFile) {
12 cerr << "File could not be opened." << endl;
13 exit(1);
14 }
15
16 cout << "Enter some text or type 'exit' to stop: " << endl;
17
18 while (getline(cin, someText) && someText != "exit")
19 myFile << someText << endl;
20
21 return 0;
22 }
```

**Note:** Besides `ios::app`, there several other available `fstream` modes.

Write a C++ program that allows the user to search a file “MyText.txt” located in the current workspace for any word (String). The program should print the result of the search. In the event the word is found the program should indicate the line where it was found in the text file.



**Desired output format sample 1:**

Enter the word you are looking for: [hello](#)  
The word hello was not found.

**Desired output format sample 2:**

Enter the word you are looking for: [hello](#)  
The word hello was found in line 3 of the text.

## 22. Basic Sorting Algorithms

### 22.1 Sorting Algorithms

Very often we need to arrange (sort) data for searching or other purposes. There is a large variety of sorting algorithms. We will only discuss here the simplest algorithms.

Code 22.1: Bubble Sort example.

```

1 #include <iostream>
2 #include <iomanip>
3 using namespace std;
4
5 int myArray[] = {7,2,4,2,9,5,6,1,5};
6 int arraySize = sizeof(myArray)/sizeof(myArray[0]);
7 void BubbleSort();
8 void swap(int one, int two);
9 void printMyArray(int, int);
10
11 int main(){
12 cout << "Array values before sorting: " << endl;
13 printMyArray(-1,-1);
14 BubbleSort();
15 cout << "\nThe array has been sorted.";
16 return 0;
17 } // end main
18
19 void BubbleSort() {
20 for(int i=0; i<arraySize; i++){
21 for(int j=0; j<arraySize-1-i; j++){
22 if(myArray[j] > myArray[j+1])
23 swap(j, j+1);
24 }
25 } // end BubbleSort
26
27 void swap(int one, int two) {
28 int temp = myArray[one];
29 myArray[one] = myArray[two];
30 myArray[two] = temp;
31
32 static int swapCount=1;
33 cout << "Swap " << setw(2) << swapCount++ << ": ";
34 printMyArray(one, two);
35 } // end swap
36
37 void printMyArray(int one, int two) {
38 for(int j=0; j<arraySize; j++){
39 if(j==one or j==two)
40 cout << "(" << myArray[j] << ")";
41 else
42 cout << " " << myArray[j] << " ";
43 }
44 cout << endl;
45 } //end printMyArray()

```



← Get code here.



Code 22.2: Selection sort example.

```

1 #include <iostream>
2 #include <iomanip>
3 using namespace std;
4
5 int myArray[] = {7,2,4,2,9,5,6,1,5};
6 int arraySize = sizeof(myArray)/sizeof(myArray[0]);
7 void SelectionSort();
8 void swap(int one, int two);
9 void printMyArray(int, int);
10
11 int main() {
12 cout << "Array values before sorting: " << endl;
13 printMyArray(-1,-1);
14 SelectionSort();
15 cout << "\nThe array has been sorted.";
16 return 0;
17 } // end main
18
19 void SelectionSort() {
20 int min;
21 for(int i=0; i<arraySize-1; i++){
22 min=i;
23 for(int j=i+1; j<arraySize; j++)
24 if(myArray[j] < myArray[min]) min=j;
25 swap(i, min);
26 }
27 } // end SelectionSort
28
29 void swap(int one, int two) {
30 int temp = myArray[one];
31 myArray[one] = myArray[two];
32 myArray[two] = temp;
33 static int swapCount=1;
34 cout << "Swap " << setw(2) << swapCount++ << ": ";
35 printMyArray(one, two);
36 } // end swap
37
38 void printMyArray(int one, int two) {
39 for(int j=0; j<arraySize; j++)
40 if(j==one or j==two)
41 cout << "(" << myArray[j] << ")";
42 else
43 cout << " " << myArray[j] << " ";
44 cout << endl;
45 } //end printMyArray()

```



← Get code here.

Code 22.3: Insertion sort example.

```

1 #include <iostream>
2 using namespace std;
3
4 int myArray[] = {7,2,4,2,9,5,6,1,5};
5 int arraySize = sizeof(myArray)/sizeof(myArray[0]);
6 void InsertionSort();
7 void printMyArray(int, int);
8
9 int main(){
10 cout << "Array values before sorting: " << endl;
11 printMyArray(-1, -1);
12
13 InsertionSort();
14
15 cout << "\nThe array has been sorted.";
16 return 0;
17 } // end main()
18
19 void InsertionSort() {
20 int here, next, temp;
21 printMyArray(-1, 1);
22 for(next=1; next<arraySize; next++){
23 temp=myArray[next];
24 here = next;
25 while (here>0 && myArray[here-1]>=temp){
26 myArray[here] = myArray[here-1];
27 here--;
28 }
29 myArray[here]=temp;
30 printMyArray(here, next+1);
31 } // end for
32 } // end InsertionSort()
33
34 void printMyArray(int inserted, int nextValue) {
35 for(int j=0; j<arraySize; j++){
36 if(j==nextValue)
37 cout << " <-- Sorted | Unsorted --> ";
38 if(j==inserted)
39 cout << "(" << myArray[j] << ")";
40 else
41 cout << " " << myArray[j] << " ";
42 }
43 cout << endl;
44 } //end printMyArray()

```



← Get code here.

## A. Answers to Exercises

### A.1 Chapter 1

→ Answer to question 2

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6 cout << "This is the first line." << endl << endl;
7 cout << "This is the second line." << endl;
8
9 return 0;
10 }
```

→ Answer to question 4

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5 cout << " *\n**\n***\n****" << endl;
6 return 0;
7 }
```

→ Answer to question 6

Write a C++ program that prompts the user to input their age, and then outputs a message stating their age. The program should use proper grammar and punctuation to form a complete sentence. For example, if the user enters "25", the program should output "You are 25 years old."

Your program should display a prompt asking the user to enter their age, and then read in the age from the user using the appropriate input method. Finally, the program should output the user's age in a message that includes the number of years and proper punctuation.

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5 int age;
6 cout << "Enter your age: ";
7 cin >> age;
8 cout << "You are " << age << " years old." << endl;
9 return 0;
10 }
```

→ Item 2

## A.2 Chapter 3

→ Answer to question 1

1. **variable:** A variable is a named storage location in computer memory that holds a value of a particular data type. It is a symbolic representation of a memory location that can hold different values at different times during the execution of a program.
2. **identifier:** An identifier is a name given to a variable, function, class, or any other user-defined item in a program. An identifier must follow certain naming rules and conventions and should be unique within its scope.
3. **keyword:** A keyword is a reserved word that has a special meaning in the programming language and cannot be used as an identifier. Keywords are used to define the syntax and structure of the language, and they have predefined functions that cannot be changed by the programmer.
4. **statement:** A statement is a unit of code that performs a specific action in a program. It can be a declaration, an assignment, a function call, a control statement, or any other operation that can be performed in the language.
5. **declaring a variable:** Declaring a variable is the process of defining a named storage location in memory with a specific data type. It involves specifying the variable name, the data type, and optionally an initial value.
6. **initializing a variable:** Initializing a variable is the process of assigning an initial value to a declared variable. It can be done at the time of declaration or later in the program.
7. **assignment statement:** An assignment statement is a statement that assigns a value to a variable. It involves using the assignment operator '=' to assign a value to the variable on the left-hand side of the operator.

→ Answer to question 2

1. All variables must be declared before being used. (True - False)  
**True.** This means that their type and name must be specified in a declaration statement before they can be used in any other statements.
2. C++ considers the variable number and NUMBER to be identical. (True - False)  
**False.** In C++, variable names are case sensitive, which means that "number" and "NUMBER" are two different identifiers and not identical.
3. The arithmetic operators \*, /, +, and – have the same level of precedence. (True - False)  
**False.** The arithmetic operators in C++ have different levels of precedence, which determine the order in which they are evaluated. The \* and / operators have a higher precedence than + and - operators.
4. The statement `cout << "a = 5;"` ; is a typical assignment statement. (True - False)  
**False.** The statement `cout << "a = 5;"` ; is an output statement that writes the value of the string "a = 5;" to the console. An assignment statement assigns a value to a variable, for example: `a = 5;`.
5. The statements `n += 2;` and `n = n + 2;` have the exact same outcome. (True - False)  
**True.** The statements `n += 2;` and `n = n + 2;` are equivalent and have the same outcome. The += operator is a compound assignment operator that adds the value on the right-hand side to the variable on the left-hand side and then assigns the result to the left-hand side variable.
6. The size of the int data type is guaranteed to be 4 bytes on all platforms. (True - False)  
**False.** The size of the int data type is not guaranteed to be 4 bytes on all platforms. The C++ standard only specifies that an int must be able to represent at least the range of values from -32767 to 32767, which requires 2 bytes.
7. The double data type can hold larger values than the float data type. (True - False)  
**True.** The double data type can hold larger values than the float data type because it uses 8 bytes to represent a floating-point number, while the float data type uses 4 bytes.
8. C++ has a built-in data type for representing Boolean values. (True - False)  
**True.** C++ has a built-in data type called bool that represents Boolean values, which can have the value true or false.
9. The "sizeof" operator in C++ returns the number of bytes occupied by a variable. (True - False)  
**True.** For example, the expression `sizeof(int)` returns the number of bytes occupied by an int variable on the current platform.

→ Answer to question 5

A variable that can have values only in the range 0 to 65535 is a two-byte unsigned integer.

An unsigned integer is a non-negative integer that can have values from 0 to a certain maximum value determined by the number of bytes used to store the value. A two-byte unsigned integer can store values in the range of 0 to  $2^{16} - 1$ , which is equivalent to 0 to 65535. Therefore, option (d) "two-byte unsigned integer" is the correct answer.

→ Answer to question 7

The largest unsigned integer that can be represented with 32 bits is  $2^{32} - 1$ , which is equal to 4,294,967,295.

The largest unsigned integer that can be represented with 64 bits is  $2^{64} - 1$ , which is equal to

18,446,744,073,709,551,615.

→ Answer to question 10

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5 int x = 0;
6
7 // Using the + operator and the = operator
8 x = x + 1;
9 cout << "x = " << x << endl;
10
11 // Using the += operator
12 x += 1;
13 cout << "x = " << x << endl;
14
15 // Using the postfix ++ operator
16 x++;
17 cout << "x = " << x << endl;
18
19 // Using the prefix ++ operator
20 ++x;
21 cout << "x = " << x << endl;
22
23 return 0;
24 }
```

→ Answer to question 16

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5 float celsius , fahrenheit;
6
7 cout << "Enter the temperature in Celsius: ";
8 cin >> celsius;
9
10 fahrenheit = celsius * 1.8 + 32;
11
12 cout << "The corresponding temperature in Fahrenheit is: ";
13 cout << fahrenheit << endl;
14
15 return 0;
16 }
```

→ Answer to question ??

→ Answer to question ??

### A.3 Chapter 3

→ Answer to question A.3

→ Item 2

→ Item 3





# Index

- if* and *if-else* Statements, 41
- Abstract Classes, 181
- Adding and Removing Nodes, 192
- Addition and Subtraction of Numbers at Different Bases, 20
- Arithmetic Operations, 29
- Array Size, 101
- Arrays and Pointer Arithmetic, 112
- Arrays and Pointers, 150
- Arrays of Structures, 118
- Background, 183
- Basic Exception Handling, 195
- C++ Loops. The *while* and *do...while* Statement., 55
- C++ Syntax Tokens, 25
- C-style String Functions, 127
- Characters, Strings and Arrays of Strings, 104
- Class Composition, 147
- Class Templates, 187
- Classes, 122
- Constant Objects, 145
- Constructors, 140
- Conversion Between Binary, Octal and Hexadecimal, 19
- Converting Binary, Octal and Hexadecimal to Decimal, 15
- Converting Decimal Numbers to Other Bases, 17
- Default and Static Variables, 90
- Default Function Arguments, 78
- Destructors, 142
- Function Overloading, 79
- Function Templates, 185
- Getting Started with C++, 11
- How C++ Stores Text?, 125
- Inheritance, 167
- Inheritance and Constructors, 170
- Inheriting Functionality, 172
- Initializing Arrays, 98
- Iterators, 201
- Linked Lists, 189
- Linked Lists using Classes, 191
- Local and Global Variables, 89
- Logical Operators, 46
- Member Initializers, 146
- More on Recursion, 159
- Multi-Dimensional Arrays, 105
- Multilevel Inheritance, 175
- Multiple Inheritance, 174
- Multiple Throws and Catches, 198
- Objects, 121
- Operator Overloading, 153

Passing a Parameter by Value or by Reference, 76

Passing Arrays to Functions, 102

Passing Structures to Functions, 119

Pointers and Some Special Operators, 150

Pointers Revisited, 149

Polymorphism and Virtual Functions, 177

Practice Questions 1, 13

Practice Questions 2, 22

Practice Questions 3, 32, 49

Practice Questions 5, 61

Practice Questions 6, 80

Practice Questions 7, 93, 162

Practice Questions 8, 106

Pre-defined Functions, 91

Primitive Data Types, 27

Queue Iterator Example, 203

Recursive Functions, 155

Relational and Comparison Operators, 41

Separating Content in Different Files, 143

Sorting Algorithms, 133

Storing Information in the Computer, 15

Stream Objects, 211

Structure Composition, 117

The `<string>` Class, 128

The *break* and *continue* Statements, 60

The *for* loop, 57

The *switch* Statement, 44

The First Example, 122

The Nested *if-else* Statements, 44

The **this** Keyword, 152

Throwing an Exception in a Function, 199

Vectors, 202

What are Arrays, 97

What are Functions?, 71

What are Pointers?, 111

What are Structures?, 115

What is Computer Science?, 9

What is Programming?, 10