C++: A Quick Introduction

Supplemental Material for Data Structures and Algorithms for Engineers at Carnegie Mellon University Africa

Compiled by: George Okeyo

Semester: Spring 2025

Table of Contents

1 INTRODUCTION TO C++

1.1 Learning Outcomes

By the end of this session, students should be able to:

a) *Explain the basic concepts of programming*
b) *Describe the features and uses of the C++ programming language*
c) *Set up the development environment for C++ programming*
d) *Write and run a simple C++ program*

1.2 Outline

- Overview of programming concepts
  - What is programming?
  - Why do we need programming?
  - Introduction to algorithms and problem-solving
- Introduction to C++ language
  - History and significance of C++
  - Features and applications of C++
- Setting up the development environment
  - Installing a C++ compiler in Windows/Linux/both (e.g., GCC, Visual Studio)
  - Configuring the IDE (Integrated Development Environment)
- Writing and running the first C++ program
  - Syntax and structure of a C++ program
  - Printing "Hello, World!" on the console"

1.3 C++: Brief History

✓ Designed by Bjarne Stroustrup at Bell Labs in 1979, initially as a successor to C.
✓ All C programs are also C++ programs – therefore, you can think of the C++ language as being as a superset of the C language.
✓ However, C++ is an object-oriented language, unlike C, with support for ***encapsulation***, ***abstraction***, ***inheritance***, and ***polymorphism***.
✓ Through use of classes, objects, member functions, and member data, C++ lets the programmer create programs that *model the data* and the *operations on the data*----a key plank of object-oriented programming.
✓ Most C++ compilers also compile C code. Therefore, you may learn C concepts as part of C++.
  - C is a procedural language.
  - C++ is both a procedural and object-oriented language.

1.4 Why C++

✓ Can be used for both high-level (no need to know the details of underlying hardware platform) and low-level (requires knowledge of underlying hardware platform e.g. to write device drivers) programming.
✓ Ability to write complex applications without compromising performance or resource management.
✓ Language of choice for AI & ML, device drivers, web services, compilers & interpreters for other languages, operating systems, and databases.

1.5 C++ standards

- ✓ Aimed at achieving interoperability and portability.
- ✓ First standard in 1998 by ISO, i.e., ISO/IEC 14882:1998.
- ✓ ISO/IEC 14882:2020, popularly called C++20, is the current ratified version.
- ✓ Not all compilers may support all features of current standards.
- ✓ Conforming to the ISO C++ Standard ensures that the code you develop can be compiled and executed on multiple operating systems.
- ✓ It also ensures code is cross-platform and portable.

1.6 Edit, Compile, Link C/C++ program

- ✓ Type the source code into a (plain) text *editor* (a code editor e.g. Notepad++, Visual Studio Code or Integrated Development Environment (IDE)) (.cpp).
- ✓ Compile the source code into object code(.o or .obj), machine language version of the source code, using a C++ *compiler*.
- ✓ Link the object codes with a *linker* to create the executable code (.exe in Windows)
- ✓ Compiler:
    - g++ for Linux, clang++ for macOS, Microsoft Visual C++ (MSVC) for Windows.
    - Compiles source code to byte code, one source file at a time.
- ✓ Linker: Resolves dependencies between object files.
- ✓ Building: when compiling and linking succeeds an executable is created that can be distributed and executed.
- ✓ Online C++ compiler: a good place to start.(e.g. https://www.programiz.com/cpp-programming/online-compiler/,)
- ✓ Debugging: analyse errors in code and fix them. Development environments provide tools for the task.

1.7 IDEs

- ✓ Combines text editing, compiling, linking, and debugging tools in one package.
- ✓ Popular IDEs include Eclipse & Code::Blocks for Linux, MS Visual Studio for Windows, and Xcode for macOS.
- ✓ Ensure you set up a suitable C++ compiler.
- ✓ Here is a set of instructions for setting up a compiler:
- ✓ Here is an example of how to set up an environment. Follow the instructions on the provided link below to install a C++ compiler and vscode to run C++ programs.
    - Windows: https://code.visualstudio.com/docs/cpp/config-mingw
    - Linux: https://code.visualstudio.com/docs/cpp/config-linux

```
Listing 1.1
#include<iostream>
int main()
{
        std::cout<<"Hello world!"<<std::endl;
        return 0;
}
```

- ✓ Compile and link in Linux: `g++ -o hello hello.cpp`

- ✓ Run in Linux: `./hello`

1.8 Errors

- ✓ *Syntax errors*: occur due to violating syntax rules e.g. leaving out a semi-colon.
- ✓ *Semantic errors*: relate to a program that runs but gives the wrong results.
- ✓ *Runtime errors*: arise as the program is executing causing abnormal program execution or termination.

1.9 Structure of a C++ Program

- ✓ When following object-orientation, a C++ program consists of classes. Each class defines member variables and member functions.
- ✓ However, a procedural C++ program will only contain functions.
- ✓ In general, a C++ program consists of preprocessor directives and the main body of the program.

1.9.1 Preprocessor directives

- ✓ In general, the program begins with preprocessor directives, that start with #.
- ✓ Preprocessor: A special utility that runs just before compilation to perform textual substitution.
- ✓ Preprocessor directive: A statement that invokes the preprocessor to run.
- ✓ #include directive: Takes the format `#include<filename>` or `#include”filepath/filename.h”`
  - o Allows the preprocessor to replace the line with the contents of filename.
  - o `#include<filename>`: filename should be included from standard include directory. For instance #include<iostream> includes the standard header file that supports input/output (IO) stream operations such us the use of `std::cout`, `std::endl`, `std::cin`, `std::cerr`, etc.
  - o `#include”filepath/filename.h”` : filename should be retrieved from `filepath`, including current directory. Generally used for headers created by the programmer – like your own files.

1.9.2 Body of program

- ✓ The main body contains the main function as well as other functions, classes, and other program elements.
- ✓ `main()` function: The `main` function is where execution begins and terminates.
  - o The header takes the form int main(). Other options include `int main(int argc, char* argv[])`, `int main(int argc, char** argv)`, `void main()`, `void main(int argc, char* argv[])`,and `void main(int argc, char** argv)`.
  - o The body of main is enclosed by `{}`. Where the header takes the form `int main(…)`, the body includes the statement return `EXIT_STATUS`;
  - o The exit_status is defined in stdlib.h with values `EXIT_SUCCESS` (or zero (0)) or `EXIT_FAILURE` (a non-zero int).
  - o The variant `int main(…)` is preferred because it allows the program to return an exit status to the caller – typically the operating system. It is also standards compliant.
  - o The variants `int main(int argc, char* argv[])` and `int main(int argc, char** argv)` allow the program to receive commandline inputs when executed.

- In the program above, the line `std::cout<<"Hello world!"<<std::endl;` displays the string `Hello World` to the screen.
- `"Hello World"` is a string literal.
- `cout`: console out. Output stream(typically screen).
- `cin`: console in. Input stream (typically keyboard).
  - Therefore, `std::cin>>num;` allows reading from the keyboard into the variable named `num`.
- `<<`: insertion operator --- inserts output to the output stream.
- `>>`: extraction operator ---extracts input from the input stream to a named variable.
- std: standard namespace.
- `endl`: end of line marker. Therefore `std::cout<<std::endl` inserts the end of line market to the standard output stream.
- `::` - scope resolution operator. Therefore, `std::cout` indicates `cout` that belongs to the scope of `std` (the `std` namespace).

```
Listing 1.2
#include<iostream>
int main()
{
        int age; //declares an int variable
        std::cout<<"Enter your age:";
        std::cin>>age; //reads a value
        std::cout<<"Your age is : "<<age<<std::endl;
        return 0;
}
```

1.9.3 Comments

- ✓ Comments provide internal documentation – they are ignored by the compiler.
- ✓ Internal documentation makes code easy to read, understand, and maintain.
- ✓ Single line comments begin with `//`
- ✓ Multiline comments are enclosed between `/* and */`.

1.9.4 Namespaces

- ✓ A namespace gives a name to sections of code, thereby resolving name conflicts whenever certain identifiers appear in multiple locations.
- ✓ Using a namespace, you can specify that the specific identifier (e.g. `cout`) that you plan to use belongs to a certain namespace (e.g. `std`) using the form `std::cout` (where `::` is the scope resolution operator).
- ✓ Std namespace gives access to functions, streams, and utilities that are part of the ISO C++ standards.
- ✓ You can avoid prefixing the `std::` by using the `using` directive as shown in code listing 1.3.

```
Listing 1.3
#include<iostream>
using namespace std; //tells the compiler what namespace to search in
                      //includes the entire namespace
int main()
{
        int age; //declares an int variable
        cout<<"Enter your age:";
        cin>>age; //reads a value
        cout<<"Your age is : "<<age<<endl;
        return 0;
}
```

- ✓ Other options include using the `using` directive with specific names that you plan to use as shown in Listing 1.4.

1.10 Summary

✓

```
Listing 1.4
#include<iostream>
//includes only certain names from the namespace
using std::cout;
using std::cin;
using std::endl;

int main()
{
        int age; //declares an int variable
        cout<<"Enter your age:";
        cin>>age; //reads a value
        cout<<"Your age is : "<<age<<endl;
        return 0;
}
```

2 C++ BUILDING BLOCKS

2.1 Learning Outcomes

By the end of this session, students should be able to:

a) *Define variables and explain their purpose in programming*
b) *Declare and initialize variables of various data types in C++*
c) *Perform basic arithmetic operations using numeric data types*
d) *Use cin and cout to take user input and display output in C++*

2.2 Outline

- Understanding variables and their types in C++
  - What is a variable?
  - Naming conventions and rules for variables
- Declaring and initializing variables
  - Syntax for variable declaration
  - Assigning values to variables during declaration
- Numeric data types and operators
  - Integers: int, short, long
  - Floating-point numbers: float, double
  - Arithmetic operators: +, -, *, /, %
- Input and output operations using cin and cout
  - Reading user input using cin
  - Formatting output with cout

2.3 Variables, Constants, and Data Types

2.3.1 Variables and Constants

✓ A variable refers to a named memory location for storing values.
  - The values can change.

✓ A constant is a named memory location that stores a value that does not change during program execution.

✓ Variables and constants make it possible to address memory locations.

✓ Constants in C++ can be:
  - Literal constants
  - Constants declared with const keyword
  - Enumerated constants using enum keyword
  - etc

2.3.1.1 Rules for Naming Variables (or other identifiers)

✓ Should not be keywords.

✓ Should contain letters, digits, and the underscore. No special characters are allowed.

✓ Should begin with a letter or the underscore.

✓ Identifiers are case sensitive; therefore sum and Sum are different.

2.3.1.2 *Declaring and Initializing Variables*

- ✓ Declaring a variable reserves a memory address and provides a name for accessing that memory address.
- ✓ A variable declaration specifies the data type of the value to be stored in the memory address, the name of the variable, and (optionally) an initial value for the variable.
- ✓ The syntax is:

```
<data-type> <variable-name>; //OR

<data-type> <variable-name>=<initial-value>;
```

- ✓ Where:
    - o `<data-type>`: a built in or user-defined data type. Specifies the nature of the value that can be stored in the variable.
    - o `<variable-name>`: a valid identifier for referencing the memory address.
    - o `<initial-value>`: a valid value of `<data-type>`
- ✓ Several variables can be declared(and initialized) using the same variable declaration statement.
- ✓ Examples include:

```
int x;

float a,b,c=10.4;

char letter;

bool flag =false;

long long num=131313131313;
```

- ✓ See Table 1 for list of fundamental data types.
- ✓ Scope: the scope of a variable refers to the region of code within which the variable is meaningful.
    - o A variable's scope is defined by the enclosing {}.
    - o As a result, there are global variables (scope: the entire program), and local variables(scope: the enclosing {}).

2.3.2 Fundamental Data Types

- ✓ C++ provides multiple data types including char, bool, and numerous integer and floating-point data types.

Table 1: Fundamental Data Types, Values, and Memory Needs

Type	**Values**	**Memory Size (in bytes)**[1]	**Comment**	**Usage**
bool	true or false	1		bool flag=false;
char	256 characters	1		char letter='B';
unsigned short int	0 to 65,535	2	Whole number	
short int	-32,768 to +32,767	2	Negative and positive whole numbers	
unsigned long int	0 to 4,294,967,295	4	Positive whole numbers	
long int	-2,147,483,648 to +2,147,483,647	4	Negative and positive whole numbers	
unsigned log long	0 to 18,446,744,073,709,551,615	8	Positive whole numbers	
long long	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807	8	Negative and positive whole numbers	

[1] The size depends on the platform---compiler, hardware architecture, and operating system!

int(16 bit)	-32,768 to +32,767	2	Negative and positive whole numbers	
int(32 bit)	-2,147,483,648 to +2,147,483,647	4	Negative and positive whole numbers	
unsigned int (16 bit)	0 to 65,535	2	Positive whole numbers	
unsigned int (32 bit)	0 to 4,294,967,295	4	Positive whole numbers	
float	1.2e-38 to 3.4e38	4	Single precision floating point numbers	float value=1.35;
double	2.2e-308 to 1.8e308	8	Double precision floating point numbers	double pi=22/7.0;
long double		16	Supported differently on multiple platforms.	

- ✓ Use the sizeof() operator to determine the size(hence, memory size) of a variable.
  - o sizeof() returns the size in bytes.

```
cout<<"size of int is "<<sizeof(int)<<" bytes"<<endl;
```

- ✓ You can allow type inference with the auto keyword. In that case, the compiler infers the type from the initial value.

```
auto flag=true;
```

  - o The auto keyword allows initialization similar to Python, whereby the type is inferred from the value.
- ✓ Use typedef to substitute variable type, e.g.:

```
typedef unsigned short int POSITIVE_INT; //positive short int

POSITIVE_SINT x=10;
```

2.3.3 Declaring constants

- ✓ Literal constants are used to initialize constants or variables or in output statements. Literal constants belong to any of the fundamental types or strings.
- ✓ For const variables, the syntax is:

```
const <data-type> <constant-name>=literal-value; e.g:

const float PI=3.14;
```

- ✓ Enums are used when a variable should accept values from only a set of values.
  - o Enums restrict the possible values.
  - o It constants enumerators: a set of constants.
- ✓ For enums the syntax is:

```
enum AfricanRegions{

        WestAfrica,

        EastAfrica,

        NorthAfrica,

        SouthernAfrica,

        CentralAfrica

};
```

- ✓ An enum can be used as a user-defined type, e.g.:

```
AfricanRegions west=WestAfrica;
```

- ✓ It can also be used to retrieve each of the enumerators.

```
cout<<"East Africa:"<<EastAfrica<<endl;
```

- ✓ You may also define constants using #define directive, but this should not be used as it is deprecated.

```
#define pi 3.14
```

2.4 Arithmetic Operators and Expressions

- ✓ Operator: Tools provided by the C++ language to work with data to transform, process or make decisions with it.
- ✓ Examples include:
    - The assignment operator =
    - Arithmetic operators +(add), -(subtract), *(multiply), / (divide), and %(modulo divide).
- ✓ Expressions are formed using operators and operands.
- ✓ See Listing 2.1 for an example using the assignment operator and the arithmetic operators.

```
Listing 2.1
#include<iostream>
using std::cout;
using std::endl;
int main()
{

    int a=11,b=3;
    int quotient, sum, product, remainder,difference;
    sum=a+b; //operands: a, b, sum; operators: +, =
    difference=a-b;
    quotient=a/b;
    product=a*b;
    remainder=11%3;
    cout<<"The two numbers are: "<<a<<", "<<b<<endl;
    cout<<"\tsum:"<<sum<<endl;
    cout<<"\tdifference:"<<difference<<endl;
    cout<<"\tquotient:"<<quotient<<endl;
    cout<<"\tproduct:"<<product<<endl;
    cout<<"\tremainder:"<<remainder<<endl;
    return 0;
}
```

2.5 Increment, Decrement, and Operator Assignment operators

- ✓ Table 2 provides a summary of additional operators that can be used for arithmetic or to increment or decrement values.

Table 2: Increment, Decrement, and Operator Assignment Operators

Operator	Symbol	Usage Example	Comment

Unary [2]increment	++	x++;
		++x;
			Equivalent to x=x+1;

[2] *Unary* operator acts on a single operand. *Binary* operator acts on two operands. *Ternary* operator acts on three operands.

Unary decrement	--	x--; --x;	Equivalent to x=x-1;
Operator assignment operators	+=	x+=y;	Equivalent to x=x+y;
	-=	x-=y;	Equivalent to x=x-y;
	*=	x*=y;	Equivalent to x=x*y;
	/=	x/=y;	Equivalent to x=x/y;
	%=	x%=y;	Equivalent to x=x%y;

2.6 Standard Input and Output

- ✓ Input from the keyboard and output to the screen.
- ✓ Standard input: uses cin stream and the extraction operator (>>).
- ✓ Standard output: uses cout stream and the insertion operator (<<).
- ✓ Read user input:

```
Format: cin>>target_variable;
```

- ✓ Display output:

```
Format: cout<<output_value;
```

  - o Where output_value can be a literal, variable, or an expression.
- ✓ See listing 2-2 for an example.

Listing 2-2
```
#include<iostream>
using std::cout;
using std::cin;
using std::endl;
int main()
{
    int a,b;
    int quotient, sum, product, remainder,difference;
    cout<<"Enter two numbers."<<endl;
    cout<<"First number:";
    cin>>a;
    cout<<"Second number:";
    cin>>b;
    sum=a+b;
    difference=a-b;
    quotient=a/b;
    product=a*b;
    remainder=11%3;
    cout<<"The two numbers are: "<<a<<", "<<b<<endl;
    cout<<"\tsum:"<<sum<<endl;
    cout<<"\tdifference:"<<difference<<endl;
    cout<<"\tquotient:"<<quotient<<endl;
    cout<<"\tproduct:"<<product<<endl;
    cout<<"\tremainder:"<<remainder<<endl;
    return 0;
}
```

2.7 Summary

3 CONTROL FLOW AND DECISION MAKING

3.1 Learning Outcomes

By the end of this session, students should be able to:

a) *Use conditional statements to make decisions in a program*
b) *Combine conditions using logical operators*
c) *Implement switch statements for multiple branching options*
d) *Utilize loops to perform repetitive tasks*

3.2 Outline

- Conditional statements: if, else if, and else
  - Syntax and usage of conditional statements
  - Logical conditions and comparison operators (<, >, ==, !=, etc.)
- Logical operators: &&, ||, !
  - Combining conditions with logical operators
  - Short-circuit evaluation
- Switch statements
  - Syntax and structure of switch statements
  - Multiple branching options based on different cases
- Loops: for, while, and do-while
  - Syntax and usage of different loop structures
  - Controlling loop execution with loop control statements (break, continue)

3.3 Program Flow

✓ Using conditional statements, a programmer can vary program execution based on inputs.

3.4 Relational and Logical Operators

✓ Programmers form conditional statements using conditional expressions.
✓ Conditional expressions evaluate Boolean values (true or false).
✓ Simple conditional expressions can be created using relational (or comparison operators).
✓ Complex conditional expressions can be formed from simple conditional expressions by using logical operators.

3.4.1 Relational operators

✓ They make it possible to check for equality or inequality.

```
Format: operand-1 operator operand-2
```

✓ The operator can be `==` (equal to), `!=`(not equal to), `>` (greater than), `<`(less than), `>=`(greater than or equal to), or `<=`(less than or equal to).
✓ See table 1 for each relational operator.

3.4.2 Logical operators

✓ Logical AND (&&): takes two operands and evaluate to true of both operands are true.
✓ Logical NOT (!): takes one operand and simply reverses the supplied Boolean value.
✓ Logical OR (||): takes two operands and returns if at least one operand is true.

- ✓ Logical XOR (exclusive OR)(^): takes two operands and returns true when one operand (but not both) is true.

Table 3: Usage of Relational and Logical Operators

Operator	**Usage**	**Explanation**
Relational operators		
==	operand1 ==operand2	`true` if operand1 and operand2 have the `same` value; `false` otherwise.
!=	operand1 !=operand2	`true` if operand1 and operand2 `don't have the same` value; `false` otherwise.
>	operand1 > operand2	`true` if operand1 is `greater than` operand2; `false` otherwise.
<	operand1 <operand2	`true` if operand1 is `less than` operand2; `false` otherwise.
>=	operand1 >=operand2	`true` if operand1 is `greater than or equal` to operand2; `false` otherwise.
<=	operand1 <=operand2	`true` if operand1 is `less than or equal to` operand2; `false` otherwise.
Logical operators (complex conditional statements)		
&&	operand1 && operand2	`true` if both operand1 and operand2 are `true`; `false` otherwise.
||	operand1 ||operand2	`true` if either operand1 or operand2 is `true`; `false` otherwise.
!	!operand	`true` if operand is `false`; `false` otherwise.
^	Operand1 ^ operand2	`true` if only one operand is `true`; `false` otherwise.

- ✓ See Listing 3-1 for some relational operators.

```
Listing 3-1
/* Logical and relational operators
Displays 1 for (true) and 0 for (false).
*/
#include<iostream>
using std::cout;
using std::cin;
using std::endl;
int main()
{
    int a=10,b=5;
    cout<<a<<" is equal to "<<b<<": "<<(a==b)<<endl;
    cout<<a<<" is not equal to "<<b<<": "<<(a!=b)<<endl;
    cout<<a<<" is greater than "<<b<<": "<<(a>b)<<endl;
    cout<<a<<" is less than "<<b<<": "<<(a<b)<<endl;
    cout<<a<<" is greater than or equal to "<<b<<": "<<(a>=b)<<endl;
    cout<<a<<" is less than or equal to "<<b<<": "<<(a<=b)<<endl;

    cout<<"(a>b)&&(a!=b): "<<((a>b)&&(a!=b))<<endl;
    cout<<"(a>b)||(a!=b): "<<((a>b)||(a!=b))<<endl;
    cout<<"!(a>b): "<<(!(a>b))<<endl;

    cout<<"(a>b)^(a!=b): "<<((a>b)^(a!=b))<<endl; //both LHS  and RHS true
    cout<<"(a>b)^(a==b): "<<((a>b)^(a==b))<<endl;//LHS true and RHS false
    cout<<"(a==b)^(a>b): "<<((a==b)^(a>b))<<endl;//LHS false and RHS true
    cout<<"(a<b)^(a==b): "<<((a<b)^(a==b))<<endl;//both LHS and RHS false
    return 0;
}
```

3.5 Branching/selection control flow

3.5.1 if, if…else, and else

- ✓ The syntax is:

```
if(conditional expression)
       statement(s)//execute if true
else //optional---not a must to have for 1-way selection
       statement(s) //execute if false
```

- ✓ If statement(s) is made up of several statements, the statements must be enclosed within `{ }`.
- ✓ See Listing 3-2 for an example.

```
Listing 3-2
/*Purpose: Read two numbers. Also lets the user choose an action- addition or multiplication. Display result.
Inputs: two numbers: a &b; users choice (a number based on a menu)
Output: the result based on users choice of action.
Logic: if…else if
*/
#include<iostream>
using std::cout;
using std::cin;
using std::endl;
int main()
{
    float a,b;
    cout<<"Enter two numbers."<<endl;
    cout<<"First number:";
    cin>>a;
    cout<<"Second number:";
    cin>>b;
    int choice;
    cout<<"Choose action.\n-----1-Perform Addition.\n-----2-Perform Multiplication.\nEnter choice:(1 or 2):";
    cin>>choice;
    cout<<"The two numbers are: "<<a<<", "<<b<<endl;
    int result;
    if(choice==1){
                result=a+b;
                cout<<"\tSum:"<<result<<endl;
    }
    else if(choice==2){
                result=a*b;
                cout<<"\tProduct:"<<result<<endl;
    }
    return 0;
```

- ✓ Use the if…else if……else statement for multiway selection, with more two or more conditions to test.
- ✓ The syntax is as shown below:

```
if(conditional expression #1)
       statement(s)
else if(conditional expression #2)
       statement(s)
       …
else if(conditional expression #N)
       statement(s)
else
       statement(s) //execute if every condition is false
```

- ✓ See Listing 3-3 for an example.

```
Listing 3-3
/*Purpose: Read an average mark and output the degree classification
(A 1-First class, a 2.1-Second Upper, and 2.2-Second Lower, and a 3-Pass).
Inputs: average mark)
Criteria: 1: average>=75; 2.1:average>=65; 2.2: average>=55; 3: average>=50
Output: degree classification.
Assumption: The mark entered is between 50 and 100  inclusive.
*/
#include<iostream>
using std::cout;
using std::cin;
using std::endl;
int main()
{
    float avg_mark;
    cout<<"Enter the average mark. Range(50-100):";
    cin>>avg_mark;
    if(avg_mark>=75 && avg_mark<=100){
            cout<<"1: First Class"<<endl;
    }
    else if(avg_mark>=65){
          cout<<"2.1: Second Upper"<<endl;
    }
    else if(avg_mark>=55){
          cout<<"2.2: Second Lower"<<endl;
    }
    else if(avg_mark>=50){
          cout<<"3: Pass"<<endl;
    }else{
         cout<<"Mark out of range! Valid range is 50 -100"<<endl;
    }

    return 0;
```

3.5.2 switch statement

- ✓ Lets you compare an expression (for equality) against some constant values.
- ✓ Uses the keywords switch, case, break, and default.
- ✓ The syntax is:

```
switch(expression)
{
        case constant#1:
                statement(s);
        break;
        case constant#2:
                statement(s);
        break;
        …
        case constant#n:
                statement(s);
        break;
        default://optional
                default-statement(s); //if no other branch is executed.
        break;
}
```

- ✓ See Listing 3-4 for an example.

```
Listing 3-4
/*Purpose: Read two numbers. Also lets the user choose an action- addition or multiplication. Display result.
Inputs: two numbers: a &b; users choice (a number based on a menu)
Output: the result based on users choice of action.
Logic: uses switch-case
*/
#include<iostream>
using std::cout;
using std::cin;
using std::endl;
int main()
{
    float a,b;
    cout<<"Enter two numbers."<<endl;
    cout<<"First number:";
    cin>>a;
    cout<<"Second number:";
    cin>>b;
    int choice;
    cout<<"Choose action.\n-----1-Perform Addition.\n-----2-Perform Multiplication.\nEnter choice:(1 or 2):";
    cin>>choice;
    cout<<"The two numbers are: "<<a<<", "<<b<<endl;
    int result;
    switch(choice)
    {
        case 1:
                result=a+b;
                cout<<"\tSum:"<<result<<endl;
        break;
        case 2:
                result=a*b;
                cout<<"\tProduct:"<<result<<endl;
        break;
        default:
            cout<<"Wrong choice!"<<endl;
        break;
    }
    return 0;
}
```

3.6 Looping/Repetition/Iteration control flow

- ✓ Makes it possible to execute a set of statements repeatedly based on a condition.
- ✓ Use iterative statements `while`, `do…while` and `for`.

3.1.1. while loop

- ✓ Executes a set of statements as long as the condition remains true.
- ✓ Evaluates the condition before the loop body; hence, the loop body may not execute at all.
- ✓ Requires a loop control variable (LCV) that is modified each time the loop is executed.
- ✓ The LCV should be used in formulating the condition – modification of the LCV should lead to the condition becoming false; otherwise, the loop will execute without stopping!
- ✓ Therefore, there must be an expression or statement that modifies the LCV.
- ✓ Syntax:

```
while(condition){
        statement(s)
}
```

- ✓ See Listing 3.5 for an example:

```
/*Listing 3-5: While loop to compute the sum and average of numbers between 1 and 10*/
#include<iostream>
using std::cout;
using std::endl;
int main()
{
   int counter=1,numValues=0;
   int sum=0;
   float avg;
   while(counter<=10)
   {
       sum=sum+counter;
       counter=counter+1;
       numValues++;
   }
   avg=(float)sum/numValues;
   cout<<"Sum:"<<sum<<", Average:"<<avg<<endl;
  return 0;
}
```

- ✓ The following while loop executes without stopping.

```
While(true){
        statement(s);
}
```

3.1.2. do…while loop

- ✓ Evaluates the condition after the first execution of the loop body; hence, the loop body will execute at least once.
- ✓ Just like the while loop, the LCV should be modified to prevent the loop from executing endlessly.
- ✓ Syntax:

```
do{
        statement(s);
} while(condition);
```

- ✓ See Listing 3.6 for an example:

```
/*Listing 3-6: Do--while loop to compute the sum and average of numbers between 1 and 10*/
#include<iostream>
using std::cout;
using std::endl;
int main()
{
   int counter=1,numValues=0;
   int sum=0;
   float avg;
   do
   {
       sum=sum+counter;
       counter=counter+1;
       numValues++;
   }while(counter<=10);
   avg=(float)sum/numValues;
   cout<<"Sum:"<<sum<<", Average:"<<avg<<endl;
  return 0;
}
```

- ✓ The following do…while loop executes without stopping.

```
do{
        statement(s);
} while(true);
```

- ✓

3.1.3. for loop

- ✓ Syntax:

```
for(initializer; condition; updater){
        statement(s);
}
```

- ✓ The initializer is executed once to initialize the LCV.
- ✓ The condition is evaluated each time before the loop body is executed.
- ✓ The updater expression updates the LCV after the loop body is executed.
- ✓ Just like the while loop, the for loop may not execute at all.
- ✓ See Listing 3-7 for an example:

```
/*Listing 3-7: for loop to compute the sum and average of numbers between 1 and 10*/
#include<iostream>
using std::cout;
using std::endl;
int main()
{
   int numValues=0;
   int sum=0;
   float avg;
   for(int counter=1;counter<=10; counter=counter+1)
   {
       sum=sum+counter;
       numValues++;
   }
   avg=(float)sum/numValues;
   cout<<"Sum:"<<sum<<", Average:"<<avg<<endl;
  return 0;
}
```

- ✓ The following for loop executes without stopping.

```
for(;;){
        statement(s);
}
```

- ✓ Other variants of the for loop may exclude the initializer or updater or both.
- ✓ You may also initialize and update multiple variables in the initializer and updater, e.g.:

```
for(int x=10,y=10;x<=15&&y>=5;x=x+1,y=y-1){
        statement(s);
}
```

- ✓ The above uses two loop control variables.
- ✓ `x=x+1` retrieves the current value in x, adds 1 to it and stores the new value in x, thereby incrementing and updating x.
- ✓ `y=y-1` retrieves the current value in y, subtracts 1 from it and stores the new value in y, thereby decrementing and updating y.

```
/*Listing 3-8:Range-based for loop
*/
#include<iostream>
using std::cout;
using std::endl;
int main()
{
   int nums[]={2,4,6,8,10};
   for(int num:nums)
   {
       cout<<num<<endl;
   }
   cout<<endl;
   //another way--uses auto to enable inference of type
   for(auto num:nums)
   {
       cout<<num<<endl;
   }
    return 0;
}
```

3.1.4. Range-based for loop

- ✓ Syntax:

```
for(data-type variable-name: sequence){
        statement(s);
}
```

- ✓ sequence: can be array type or std::string type.
- ✓ See Listing 3-8 for an example.

3.1.5. continue and break statements

- ✓ continue and break can be used to modify the program behaviour within the loop
- ✓ `continue`: causes the next iteration of the loop to be considered. Other code after the continue are ignored.
- ✓ `break`: causes the loop to be exited immediately and the statement immediately after the loop to be considered.
    - o The break statement can be used to terminate infinite loops like those shown in 3.1.1 to 3.1.3.
    - o E.g.

```
for(;;){
        statement(s);
        if(condition)
                break;//exit loop if condition is true.
}
```

3.1.6. Nested Loops

- ✓ A nested loop is one in which one loop is enclosed inside another loop.

Syntax:

```
outer-loop{
        inner-loop{
        }
}
```

3.7 Summary

4 ARRAYS AND STRINGS

4.1 Learning Outcomes

By the end of this session, students should be able to:

a) *Declare and initialize arrays in C++*
b) *Access and manipulate elements in arrays*
c) *Explain the concept of multi-dimensional arrays*
d) *Perform basic operations on strings in C++*

4.2 Outline

- Declaring and initializing arrays
  - Syntax for array declaration
  - Assigning values to array elements
- Accessing array elements
  - Indexing and addressing individual elements in an array
  - Looping through array elements using index variables
- Multi-dimensional arrays
  - Creating and accessing elements in 2D arrays
  - Working with rows and columns in a matrix
- Manipulating strings in C++
  - Declaring and initializing strings
  - String concatenation and comparison
  - String length and accessing individual characters

4.3 Overview

✓ Array: a collection of elements of the same type stored in contiguous memory.
✓ Each array member (also called element) is accessed using an index – the index shows the position of the element in the array.
✓ The elements are store in a sequential and orderly fashion.
✓ The index starts from zeros up to arraysize-1.

4.4 One-dimensional arrays

4.4.1 Declaring and Initializing arrays

✓ *Static arrays:* the length of the array is fixed by the programmer at compile time.
✓ *Dynamic arrays*: the array length is decided at runtime.
✓ The syntax to declare a static arrays is:

```
<data-type> arrayName[ARRAYSIZE]={optional list of elements}; //eg
const int ARRAYSIZE=5;
int nums[ARRAYSIZE];//no initializer
int nums[ARRAYSIZE]={}; //all elements initialized to 0
int nums[ARRAYSIZE]={1,3}; //first two elements initialized. The remaining
                        // elements initialized to 0
int nums[ARRAYSIZE]={1,3,5,7,9};
```

✓ It is also possible to give an initializer without specifying the size of the array. In that case, the size will be determined based on the number of elements in the initializer. For example `int nums[]={1,3,5,7,9};`

- ✓ The size is inferred as 5.
    - Using `sizeof(nums)/sizeof(int)`, you can determine the number of elements.

4.4.2 Accessing array data

- ✓ Data are accessed from an array using the array index – a value from 0 to length-1.
- ✓ The syntax is arrayName[index]
- ✓ The compiler used the memory address of the first element to determine the relative positions of the other elements.
- ✓ Accessing an array element beyond the bounds of the array leads to ***undefined behavior***!
- ✓ It is convenient to access array elements using a counting loop e.g., a for loop.
- ✓ See Listing 4-2 for an example.

```
/*Listing 4-2
Purpose: Program initializes an array and displays each value.
Logic: Uses for loop to step through the array.*/
#include<iostream>
using std::cout;
using std::endl;

int main()
{
    const int ARRAYSIZE=5;
    int nums[ARRAYSIZE]={1,3,5,7,9};
    for (unsigned int index=0;index<ARRAYSIZE;index++)
        cout<<nums[index]<<" ";
    cout<<endl;
    return 0;
}
```

4.5 Multidimensional arrays

- ✓ Represent arrays with more than one dimension.
- ✓ The simplest multidimensional array is the two-dimensional array, which can be conceptualized as a table with rows and columns.

4.5.1 Declare a multidimensional array

- ✓ You can specify the number of elements for each dimension.
- ✓ For a two-dimensional array, the syntax is:

```
<data-type> arrayName[DIMENSION1-SIZE][DIMENSION2-SIZE];//e.g.,
float table[3][2]; //declares a table of 3 rows, 2 columns
```

- ✓ `DIMENSION1-SIZE` and `DIMENSION2-SIZE` are integer values (preferably constants) representing the size of each dimension.
- ✓ Just like a one-dimensional array, it can also be initialized.
- ✓ The first dimension must be specified but the second can be inferred from initial values.

4.5.2 Accessing elements of a multidimensional array

- ✓ You must use an index for each dimension. The indexing is similar to one-dimensional arrays, with the first position in each dimension being index 0.
- ✓ It is common to use nested counting loops to step through the elements of a multidimensional array.
- ✓ See Listing 4-3 for an example.

```
/*Listing 4-3
Purpose: Program creates a 9x9 matrix to contain the multiplication table of 1 to 9.
The program displays the table.
Logic: Uses nested for loops.*/
#include<iostream>
using std::cout;
using std::endl;

int main()
{
    const int ROWS=9, COLS=9;
    int table[ROWS][COLS]={}; //initializes everything to zero
    cout<<"*";
    for(unsigned int col=0;col<COLS;col++)
        cout<<"\t"<<(col+1);
        cout<<"\n--------------------------------------------------------------------------";

    for (unsigned int row=0;row<ROWS;row++)
        {
        cout<<"\n"<<(row+1)<<"|";
        for(unsigned int col=0;col<COLS;col++)
        {
            table[row][col]=(row+1)*(col+1);//why?
            cout<<"\t"<<table[row][col];
        }
    }
    cout<<endl;
    return 0;
}
```

4.1. Strings

4.1.1. C-style character strings

- ✓ Character arrays where the last element is represented bt '\0' to represent the string terminator.
- ✓ Therefore the string CMUA will be represented by the list {'C','M','U','A','\0'}.
    - o This implies that you need five memory locations – 4 for the string, and 1 for the string terminator.
    - o The last character precedes the string terminator '\0'.
- ✓ Programming with C-style strings can be dangerous if the string terminating character '\0' is not provided for.
- ✓ Listing 4-4 is an example of unsafe use of C-style strings,

```
/*Listing 4-4
Purpose: Read a string into a C-style string.
Display the string.*/
#include<iostream>
using std::cout;
using std::cin;
using std::endl;
int main()
{
    const int WORDSIZE=8;
   char word[WORDSIZE];
   cout<<"Enter a word. The word should be 8 characters of less....";
   cin>>word; //whats the danger?
   cout<<"You entered: "<<word<<endl;
    return 0;
}
```

- ✓ Therefore, it is encouraged to use the C++ standard library class std::string.

- ✓ C functions like `strcat()` (to concatenate two strings), `strcpy()` (to copy one string into another) and `strlen()` (to determine the length of a string) and other similar functions operate on C-style strings.
- ✓ Such C functions look for the null terminator (‘\0’) to determine the end of the string – if the null character is missing, the functions can exceed the bounds of the character array.
- ✓ ***Important Note***: When using C-style strings, use secure coding practices to avoid buffer overflows.
- ✓ See Listing 4-6 for safe use of C-style strings.
    - o The program in Listing 4-6 may run without showing any problems.
    - o Try to give it an input that is larger than 6 characters and observe the output. What did you notice?

4.1.2. C++ Strings: Using std::string

- ✓ Suitable for string input and string manipulation operations.
- ✓ Avoid challenges associated with handling C-style strings.
- ✓ `std::string` strings are not static in size ad can expand to accommodate longer strings.
- ✓ See Listing 4-5 for an example:

```
/*Listing 4-5
Purpose: Read a string into a C++ strings.
Perform some manipulation and display results.*/
#include<iostream>
#include<string> //for C++ strings
using namespace std; //for cout,endl,cin, and string
int main()
{
  string firstName, secondName, fullName;
   cout<<"Enter a first name:";
   getline(cin,firstName);
   cout<<"Enter a second name:";
   getline(cin,secondName);
   //concatenate
   fullName=firstName+" "+secondName;

   cout<<"Full Name: "<<fullName<<endl;
   cout<<"length(Full Name): "<<fullName.length()<<endl;
    return 0;
}
```

4.2. Summary

- ✓ Be careful to avoid buffer overflows – accessing addresses beyond the bounds of the array.
- ✓ It is recommended to use C++ strings instead of C-style strings in order to avoid ***buffer overflow*** issues associated with C-style strings.

```
/*Listing 4-6: Demonstrate unsafe use of C-style strings. */
#include<iostream>
#include<cstring> //for C-style string functions
 using std::cout;
 using std::cin;
 using std::endl;

int main(int argc, char * argv[])
{
    char text[6];
         cout<<"Enter text of 5 characters:";
         cin>>text; //unsafe- does not check bounds of the text array –
                  //the cin may lead to overflow if the user enters more than 5 characters.
                  cout<<"Text: "<<text<<endl;
         char targetText[6];
    strcpy(targetText, text); //unsafe--strcpy will cause overflow if txt has more than 6 characters.
    cout<<" Target Text: "<<targetText<<endl;
         return 0;
}
```

5 FUNCTIONS AND SUB-PROGRAMS

5.1 Learning Outcomes

By the end of this session, students should be able to:

a) *Define and call functions in C++*
b) *Explain the concept of function parameters and return types*
c) *Implement function overloading to create multiple versions of the same function*
d) *Use recursion to solve problems*

5.2 Outline

- Defining and calling functions
    - Syntax and structure of a function
    - Function definition and declaration/prototypes
    - Function calls and return statements
- Function return types
    - Returning values from functions
    - Void functions
- Function parameters and arguments
    - Passing values to functions
    - Default parameters
    - Function overloading and function templates
- Recursion
    - Recursive functions and their structure
    - Base case and recursive calls
    - Recursive algorithms (e.g., factorial, Fibonacci sequence)

5.3 Overview

- ✓ A large and complex program solution cannot be handled only using the main function.
- ✓ Such a program can be broken down into sub-programs.
- ✓ Each sub-program should be defined to handle a single well-defined task.
- ✓ Such a sub-program can be defined using a *function*.
- ✓ Program execution will occur by invoking the program's functions.
- ✓ *Definition*: A function is a subprogram that (optionally) takes parameters and (optionally) returns a value when invoked to perform a task.

5.4 Defining and calling functions

- ✓ You create a function in two steps:
- ✓ Step 1: Declare the function by creating a function prototype.
    - A function prototype specifies the function return type, function name, and parameter list.
    - A function prototype must appear before the function can be called.
    - The syntax is:

```
<return-type> functionName(type-name1,type-name2,…, type-nameN);
```

    - Where:

        - `<return-type>`: specifies the data type (e.g. int, float, etc) of the value the function should return. Use void for functions that don't return a value. A function has only one return type.
        - `functionName`: a valid identifier for identifying the function.
        - `type-name1…type-nameN`: the parameter list. It specifies a comma-separated list of the data types of the inputs the function should receive. The type-name may also be followed by the `parameterName`, but this is optional. If the function does not receive inputs, the parameter list will be empty.
    - Examples:
        - `int area(int,int);` //receives two integer values and returns int
        - `int area(int length, int width);`
        - `void print(float);`
        - `void print();`//does not return a value; receives no inputs.
- ✓ Step 2: Define the function by providing the function definition.
    - A function definition specifies the implementation of the function, i.e., the logic the function uses to perform its task.
    - It takes the form:

```
<return-type> functionName(type-name1 var1, type-name2 var2,…,typenameN
varN)
{
        Statement(s)
}
```

    - Where:
        - `<return-type>` , `functionName`, and `type-name1…type-nameN` are as per function declaration.
        - `var1 …varN` are names of the parameters the function should receive as input.
        - The `statement(s)` refers to one or more statements the function uses to perform its tasks. For value returning functions (i.e., where the return type is not `void`), there must be at least one `return statement`.
        - The `return statement` takes the form:

```
return expression; where expression must have the same type as
<return-type>.
```

5.5 Function parameters, arguments, and calls

- ✓ A function is invoked to execute using a function call.
- ✓ When a function prototype declares parameters, the function call sends arguments.
    - An argument refers to a value the function expects based on its parameters.
- ✓ A function call expression takes the form:
    - functionName(arguments)
    - e.g.:
        - area(10,3)
        - area(length, width),etc.
- ✓ For value returning functions, the function call expression can be assigned to a variable, be inserted into an output stream, or be passed as an argument to another function.

- ✓ Consider Listing 5-1 for an example that shows function declaration, function definition, and function calls.

```
/*Listing 5-1: Function declaration, function definition, function calls
Purpose: Read the length and width of a rectangle and print out the area and perimeter.
Input: length, width.
Output: area, perimeter
Logic: uses functions main(), area(), and perimeter().
*/
#include<iostream>
using namespace std;

/*
Declare functions
*/
float area(float, float);
float perimeter(float,float);
int main()
{
    int length, width;
    cout<<"Enter length and width of the rectangle.\n";
    cout<<"Length:";
    cin>>length;
    cout<<"Width:";
    cin>>width;
    /*Compute and display the area and perimeter. Use function calls.*/
    cout<<"\nRectangle: length="<<length<<",width="<<width<<endl;
    cout<<"Area:"<<area(length,width)<<endl;
    cout<<"Periumeter:"<<perimeter(length,width)<<endl;
    return 0;
}
/*Define area function*/
float area(float length, float width)
{
    return length*width;
}
/*Define perimeter function*/
float perimeter(float length, float width)
{
    return 2*(length+width);
}
```

- ✓ Listing 5-1 defines three functions – area(), perimeter(), and main().
- ✓ You may also define a function to take default arguments.

5.6 Argument passing: by value, by reference

- ✓ When an argument is passed by value, all the changes made by the called function will be discarded as soon as the called function returns.
- ✓ When an argument is passed by reference, all the changes made by the called function to the corresponding parameter will be reflected in the arguments when the called function returns.
- ✓ To support argument passing by reference, the parameter types must be followed with the &.
- ✓ The use of & means that you are passing the actual memory address of the corresponding argument. This is why any changes on the parameter reflect in the argument since the argument's address is available to the called function.
- ✓ Given that a function returns only one value, argument passing by reference can be used to allow a function to modify multiple values or variables.

- ✓ See Listing 5-2 for an example. Run and examine the output.

```
/*Listing 5-2: Call-by-value vs. Call-by-reference
Purpose: Show call-by-value and call-by-reference.
Logic: uses functions main(), doesNotModify(), and modifies().
*/
#include<iostream>
using namespace std;
/*Declare functions*/
void doesNotModify(int);
void modifies(int &);
int main()
{
    int num;
    cout<<"Enter a number:";
    cin>>num;
    //modify the number by multiplying by 3 and printing the result.
    doesNotModify(num);
    cout<<"doesNotModify():"<<num<<" multiplied by 3 = "<<num<<endl;
   modifies(num);
    cout<<"modifies():"<<num<<" multiplied by 3 = "<<num<<endl;
    return 0;
}
/*Define functions.*/
void doesNotModify(int num)
{
    num=num*3;

}
void modifies(int & num)
{
    num=num*3;
}
```

5.7 Argument passing: pass array values

- ✓ An array, just like any other variable can be passed as a parameter to a function.
- ✓ For a function to receive an array as input, it needs an array reference parameter and another parameter representing the size/length of the array.
    - o The length ensures that you cannot process the array beyond its bounds.
- ✓ See Listing 5-3 for an example.

5.8 Recursion

- ✓ In recursion, a function calls itself.
- ✓ A function that calls itself must define both the base and recursive cases. The base case allows the function execution to stop.
- ✓ See Listing 5-5 for an example with the factorial function.

5.9 Overloaded functions

- ✓ Function overloading refers to a situation where the same function name is used but with different parameter lists.
- ✓ The function to call will be determined based on the argument list.
- ✓ See Listing 5-5 for an example.

```
/*Listing 5-3: Array passing
Purpose: Show how to pass an array of numbers to two function.
One function computes and returns the sum.
The second function displays the values in the array.
Logic: uses functions main(), print(), and sum().
Use counting for loop.
*/
#include<iostream>
using namespace std;
/*Declare functions*/
void print(float [], int);
float sum(float [],int);
int main()
{
    const int MAX=7;
    float nums[MAX]={4.5,2.5, 3.1, 6.1, 2.7, 8.0};
    //pass the array
    print(nums,MAX);
    cout<<"\nSum:"<<sum(nums,MAX)<<endl;
    return 0;
}
/*Define functions.*/
void print(float nums[], int length)
{
    for(int index=0; index<length;index++)
        cout<<nums[index]<<endl;
    cout<<endl;
}
float sum(float nums[], int length)
{
    float sum=0.0;
    for(int index=0; index<length;index++)
    {
        sum+=nums[index];
    }
    return sum;
}
```

```
/*Listing 5-4: Recursion
Purpose: Input a number and compute and display the factorial of the number.
Logic: uses functions main() and factorial(). Recursive definition: n!=n*(n-1)!
*/
#include<iostream>
using namespace std;
/*Declare functions*/
int factorial(int n);

int main()
{
    int n;
    cout<<"Enter a number between 0 and 20:";
    cin>>n;
    cout<<"\n"<<n<<"! ="<<factorial(n)<<endl;
    return 0;
}
/*Define functions.*/
int factorial(int n)
{
    if(n==0||n==1)
        return 1;//execute for base case
    else
            return n*factorial(n-1); //recursive case
}
```

```
/*Listing 5-5: Function overloading
Purpose: Compute and print the average of a set of numbers. Use two or three numbers.
Numbers are both int and double
Logic: uses functions main() and average().
*/
#include<iostream>
using namespace std;
/*Declare overloaded average functions*/
int average(int, int);
int average(int, int,int);
double average(double, double);
double average(double, double,double);
int main()
{
    cout<<"Average(10,6):"<<average(10,6)<<endl;
    cout<<"Average(10.1,6.1):"<<average(10.1,6.1)<<endl;
    cout<<"Average(10,6,3):"<<average(10,6,3)<<endl;
    cout<<"Average(10.1,6.1,3.1):"<<average(10.1,6.1,3.1)<<endl;
    return 0;
}
/*Define overloaded average functions.*/
int average(int x, int y){
    return (x+y)/2;
}
int average(int x, int y,int z){
    return (x+y+z)/3;
}
double average(double x, double y){
    return (x+y)/2;
}
double average(double x, double y,double z){
     return (x+y+z)/3;
}
```

5.10 Summary

- ✓ Functions help simplify large and complex programs.
- ✓ Each function should perform a single well-defined task.
- ✓ A function only returns one value. However, if you want it to modify several values, you can pass arguments by reference.
- ✓ Recursion allows you to define a function that calls itself. However, you must remember to define both the base case and the recursive case.
- ✓ Function overloading allows the programmer to use the same function name as long as the parameter lists are different. The compiler figures out the version of the overloaded function to call based on the arguments list and the types of arguments.

6 POINTERS, REFERENCES, AND MEMORY MANAGEMENT

6.1 Learning Outcomes

By the end of this session, students should be able to:

a) Explain the concept of pointers and memory addresses in C++.
b) Declare and initialize pointers.
c) Use pointer arithmetic and explain its applications.
d) Explain the concept of passing arguments by reference and its advantages.
e) Write programs that pass arguments by reference.
f) Apply dynamic memory allocation using the 'new' and 'delete' operators when writing programs.

6.2 Outline

- Introduction to Pointers:
  - Memory addresses and pointers
  - Declaring and initializing pointers
  - Accessing variables through pointers
- Pointer Arithmetic:
  - Incrementing and decrementing pointers
  - Pointer arithmetic operations
  - Pointer comparisons
- Passing Arguments by Reference:
  - Reference variables in C++
  - Pass-by-reference vs. pass-by-value
  - Modifying variables using references
- Dynamic Memory Allocation with new and delete:
  - The 'new' and 'delete' operators
  - Allocating and releasing memory dynamically
  - Memory leaks and deallocation best practices

6.3 Overview

- ✓ C++ lets you write high-level programs that are abstracted from the hardware as well as those that directly consider the hardware.
- ✓ Pointer and references make it possible to manipulate memory.

6.4 Pointers

- ✓ Pointer: a variable that stores a memory address instead of an ordinary value.
  - It is said a pointer points to a memory location. This can be visualized in Fig 1.
- ✓ Like every other variable, a pointer occupies a memory space.
- ✓ Fig. 1 shows a pointer `ptr` that occupies the memory address `200`. The pointer `ptr` points to the memory address `600`. Memory address `600` holds the value of the variable named `a`. Memory `600` holds the value `15`. The arrow establishes the pointing relationship between the memory locations for `ptr` and `a`.
- ✓ Note: 200 and 600 are hypothetical memory location that can be replaced by suitable hexadecimal memory addresses.

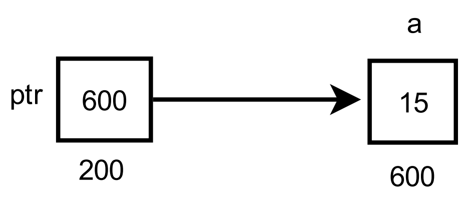


Figure 1: Visualization of a Pointer

6.4.1 Declare a pointer

✓ The syntax to declare a pointer is:

```
<pointer-type> * pointerVariable;
```

- Where:
  - `<pointer-type>` indicates the type of value that can be stored in the memory address that `pointerVariable` points to.
  - `<pointer-type>` can be set to void as well. When the type is void, `pointerVariable` points to a block of memory of unspecified type.

✓ Once declared, the pointer needs to be initialized. If not initialized, accessing the pointer will cause access to a random memory location. This is very *unsafe*!

- It is advisable that when declaring a pointer, it should be initialized to NULL.

```
<pointer-type> * pointerVariable=NULL;//e.g.
int * ptrInt=NULL;
float * ptrFloat=NULL;
double * ptrDouble=NULL;
char * ptrChar=NULL; //etc
```

6.4.2 Set a pointer to a variable

✓ Once a pointer is declared, you can set it to point to the memory address of another variable using.

✓ You obtain the memory address of a variable using the address-of (&) operator.

✓ The following example show how the pointer (**ptr**) in Fig. 1 can be associated with the memory of variable **a**.

```
int a=15;
int * ptr=NULL;
ptr=&a;
```

✓ You may access the data in the memory pointed to by a pointer using the dereference operator (also called indirection operator) (*).

- The * (dereference operator) can be placed before the pointer name in an expression, output statement, etc.
- It should only be used with a valid pointer – otherwise the result is undefined.

✓ Look at Listing 6-1 for pointer declaration, use of the &(address-of) operator and the * (dereference operator).

- Using Listing 6-1, there are two ways of accessing **15**, the value in **a** - using a or the pointer to **a**.
- *Line 8* shows address-of; Line 12 shows the use of dereference operator.

```
/*Listing 6-1: Declare a pointer, initialize it, then display the addresses and
values.*/
#include<iostream>
using std::cout;
using std::endl;
int main()
{
int a=15;
int * ptr=NULL;
ptr=&a;
cout<<"Address of a:"<<&a<<endl; //address of a
cout<<"Address of pointer a:"<<&ptr<<endl; //address of the pointer
cout<<"a:"<<a<<endl; //displays the value in a
cout<<"Another way of accessing a:"<<*ptr<<endl;//also displays the value
in a
return 0;
}
```

6.5 Dynamic Memory Management

- ✓ The previous example uses static memory allocation when allocating memory to the pointer variable.
- ✓ However, you can allocate and deallocate memory to a pointer using the **new** and **delete** operators.
- ✓ The memory is allocated and de-allocated from the **free store** (also called the **heap memory**).

6.5.1 new operator

- ✓ When using **new**, you need to specify the data type for the types of values to be stored in the memory that you are allocating.
- ✓ The syntax is:

```
<data-type> * pointerVariable=new <data-type>; /reserves memory for one
element of <data-type>.
e.g.: int *ptrInt=new int;
```

- ✓ If you need to store a collection of elements (like in an array), you can specify so using the syntax:

```
o  <data-type> * pointerVariable=new <data-type>[NumberOfElements];
o  e.g.:
       ▪  int *ptrVar=new int[10]; reserves memory for 10 integers.
       ▪  char * name=new char[10];//reserves memory for 10 characters, a C-
          style string.
```

- ✓ **Note**:
  1. There is no guarantee that new will return memory – when the heap memory is exhausted, i.e., when the memory resources are depleted, new will return the *null pointer*.
  2. Therefore, it is important to check what is returned by new before trying to manipulate the memory.

6.5.2 delete operator

- ✓ Any memory that is allocated using **new** should be freed(de-allocated) using **delete**.
- ✓ For a single element pointer, the syntax is:

```
delete pointerVariable;
e.g.: int *ptrInt=new int;
      delete ptrInt;
```

- ✓ For a multiple element pointer, the syntax is:

```
delete [] pointerVariable; //releases a block of memory
e.g.: int *ptrInt=new int[10];
      delete [] ptrInt;
```

- ✓ Failure to de-allocate memory means that the memory will be unavailable for applications to use – a scenario referred to as a *memory leak*.
- ✓ Look at Listing 6-2 for dynamic memory allocation and de-allocation.

```
/*Listing 6-2: Demonstrates dynamic memory allocation and de-allocations*/
#include<iostream>
using std::cout;
using std::endl;
int main()
{
   int *ptrInt=new int; //for single int value
   *ptrInt=33;
    cout<<"Value in memory pointed to by ptrInt:"<<*ptrInt<<endl;
    delete ptrInt;
    char * name=new char[15]; //for multiple characters
    name="Bellingham"; //remember unsafe C-style pointer use.
    cout<<"name:"<<name<<endl;
    delete name;
    return 0;
}
```

6.5.3 Pointer arithmetic: ++ and -- operators on pointers

- ✓ Given that a pointer contains a memory address, applying decrement(++) or increment (--) causes the pointer to access the new or previous value in the memory block.
    - o The magnitude of the movement is determined by the size (in bytes) of the pointer type (e.g. *pointer to char (char*)* will make a 1-byte movement; *pointer to 4-byte int(int*)* will make a 4-byte movement).
- ✓ Look at Listing 6-3, which reads a set of integer values into memory block and processes the values by incrementing from the first memory address until all blocks are accessed.

6.5.4 Passing Pointers to functions

- ✓ Pointers can be passed to functions, giving the called function access to the original memory addresses.
- ✓ The syntax requires that the relevant formal parameters be of pointer type. The caller function will pass the address of the arguments.
- ✓ Look at Listing 6-4 and compare that with Listing 5-2.

```
/*Listing 6-3: Allocates a block of memory and steps through the block.
Reads a set of numbers and prints out the numbers, the sum, and average*/
#include<iostream>
using std::cout;
using std::cin;
using std::endl;
int main()
{
   cout<<"How many numbers do we expect to process?";
   int numberOfValues;
   cin>>numberOfValues;
    if(numberOfValues>=1)
    {
          int *nums=new int[numberOfValues]; //this is similar to an array of values
          //now read a rest of numbers and compute and print the sum and average
          for(int index=0;index<numberOfValues; index++)
          {
              cout<<"Enter number#:"<<(index+1)<<":";
              cin>>*(nums+index); //nums gives first address; Adding 1 to it makes an int
                                  //size move in the memory block.
          }
          int sum=0;
          float avg;
          for(int index=0;index<numberOfValues;index++)
          {
              sum=sum+*(nums+index);//*(nums+index) retrieves each value in the memory block
          }
          avg=(float)sum/numberOfValues;

          //printout the values, sum, and average.
          cout<<endl<<endl<<"Values: ";
           for(int index=0;index<numberOfValues;index++)
          {
              cout<<*(nums+index)<<"\t";
          }
          cout<<"\nSum:"<<sum<<endl;
          cout<<"Average:"<<avg<<endl;

          //deallocate memory
          delete []  nums;
    }
    else{
        cout<<"You entered "<<numberOfValues<<". An invalid value!!"<<endl; //
    }

   return 0;
}
```

6.6 Problems with pointers

- ✓ *Memory leaks*: when memory is unavailable for use by applications because it was not deallocated.
- ✓ *Dangling pointers*: when pointer is not initialized or when you try to use it after it has been deallocated.

6.7 References

- ✓ *Reference*: an *alias* for a variable - it allows you to use the reference instead of the variable.
- ✓ A reference is declared using the reference operator (**&**).
- ✓ Once a reference is declared, it should be initialized to a variable.
- ✓ Therefore, a reference gives you direct access to the memory address of a variable.
    - o The syntax to declare a reference is:

```
<data-type> & referenceVar=variableName;
```

    - o The datatype of the reference must be the same as that of `variableName`.

- ✓ Look at Listing 6.5 for use of a reference variable.
- ✓ We have also seen the use of pass-by-reference when defining functions.

```
/*Listing 6-4
Purpose: Show call-by-pointer.
Logic: uses functions main() and modifies().
*/
#include<iostream>
using namespace std;
/*Declare functions*/
void modifies(int *);
int main()
{
    int num;
    cout<<"Enter a number:";
    cin>>num;
    //modify the number by multiplying by 3 and printing the result.
    cout<<"modifies():"<<num;
   modifies(&num);
    cout<<" multiplied by 3 = "<<num<<endl;
    return 0;
}
/*Define functions.*/
void modifies(int * num)
{
    *num=*num*3;
}
```

6.8 Summary

- ✓ Pointers make it possible for C++ programs to manage memory resources.
- ✓ You can access data from a memory block using pointer arithmetic.

```
/*Listing 6-5: Using references*/
#include<iostream>
using std::cout;
using std::cin;
using std::endl;
int main()
{
   float a=10.3;
   float & refVar=a; //refVar is an alias for a.
   cout<<"a "<<a<<endl;
   cout<<"another way to access a "<<refVar<<endl<<endl;
   refVar+=3.0;
    cout<<"after change: a "<<a<<endl;
   cout<<"after change: another way to access a "<<refVar<<endl<<endl;
   return 0;
}
```

7 OBJECT-ORIENTED PROGRAMMING (OOP) BASICS

7.1 Learning Outcomes:

By the end of this session, students should be able to:

- Describe the fundamental principles and concepts of object-oriented programming.
- Explain, define and use classes and objects in C++.
- Explore the concept of access specifiers and their role in encapsulation and use them to achieve encapsulation in code.
- Explain and use member functions and data members within a class.
- Explain and use the concept of encapsulation and information hiding in writing programs.
- Explain the purpose and usage of constructors and destructors and their importance in object initialization and cleanup.

7.2 Outline

- Overview of OOP Concepts:
  - Encapsulation, inheritance, and polymorphism
  - Benefits of OOP in software development
- Classes and Objects:
  - Defining classes and objects
  - Data members and member functions
  - Accessing class members
- Access Specifiers:
  - Public, private, and protected access specifiers
  - Encapsulation and information hiding
- Member Functions and Data Members:
  - Member function declaration and definition
  - Accessing member functions and data members
  - Static member functions and data members
- Encapsulation and Information Hiding:
  - Encapsulation and its benefits
  - Access specifiers and encapsulation
  - Getters and setters for data member access
- Constructors and Destructors:
  - Default constructors and parameterized constructors
  - Constructor overloading
  - Destructors and object cleanup

7.3 Overview

✓ So far, we have used (mostly) the procedural aspects of C++.
✓ This lesson considers object-oriented C++.

7.4 Overview of OOP Concepts

- Encapsulation, inheritance, and polymorphism
- Benefits of OOP in software development

- ✓ C++ supports both procedural and object-oriented programming.
- ✓ OOP makes it possible to model real-world objects in computer programs.
- ✓ Object-oriented languages use *abstraction, encapsulation, inheritance, and polymorphism* to support object-oriented programming.
- ✓ An object-oriented program consists of a collection of objects. Program execution involves message passing between objects.
- ✓ To model a real-world object, we use the class construct to define the object's properties.
- ✓ Once a class is defined, several objects can be created from it.
- ✓ *Definitions*:
  - o Object: An object is an instance of a class.
  - o Class: A blueprint for a collection of objects.
- ✓ There are two types of properties for each class:
  - o The attributes (the data) – these are defined by data members (or member variables). They define the object. Also called the state of the object.
  - o The activities an object can perform (functions). They define the behavior of the object. They are defined by member functions or methods.

7.5 Classes and Objects

- o Defining classes and objects
- o Data members and member functions
- o Accessing class members

- ✓ A class is declared using the class keyword using the following syntax:

```
class ClassName
{
       //class specification—member variables and member functions
};
```

- o e.g.:

```
class A{
       //member variable
       int x;
       //member function
       void print(){
              cout<<x<<endl;
              }
};
```

- ✓ Once a class is declared, you can use it like any other datatype to declare variables. Such variables are called objects.
  - o For instance:

```
A a1,a2; //declares two objects of the class A.
```

- ✓ You can access the members of each object using the dot (.) operator
  - o For instance:

```
a1.print(); //calls the print function of the object stored in a1.
```

- ✓ When the dot operator is used with member functions, e.g., `a1.print()`, the process is called *message passing* and the expression is called *message*.
- ✓ The class can also be used as a pointer type as shown below:
  - o `A * b=new A;//`
  - o Once a pointer to an object type is declared, you use the pointer operator (`->`) to access the members, e.g.:

            ▪ `b->print();`

- ✓ Encapsulation refers to the process of bundling into an object all the variables and the functions that operate on the variables such that only the member functions can access and manipulate the variables. Encapsulation is achieved through the *class construct* and *access specifiers*.

7.6 Access Specifiers

  - o Public, private, and protected access specifiers
  - o Encapsulation and information hiding

- ✓ Class members can be made public, protected, or private using access specifiers.
  - o *public members*: Can be accessed anywhere using an object of the class.
  - o *private members*: Can only be accessed within the class or within friend classes.
  - o *protected members*: Can be access within the class as well as within classes that inherit from the current class.
- ✓ In general, member functions are public because they provide a *public interface*. On the other hand, member variables are private to support *information hiding*.
- ✓ Look at Listing 7-1 for a badly designed Circle class.
  - o All members are public, making them accessible anywhere.

```
/*Listing 7-1
Purpose: A badly designed class. No information hiding.
Data members can be modified anyhow.Logic: .
*/
#include<iostream>
using namespace std;

class Circle{
    public:
    float radius;
    float const PI=3.14;
    float area(){return PI*radius*radius;}
    float circumference(){return 2*PI*radius;}
};

int main()
{
    Circle c;
    c.radius=2.3;//set radius
    cout<<" Radius:  "<<c.radius<<endl;
    cout<<" Area:  "<<c.area()<<endl;
    cout<<" Circumference:  "<<c.circumference()<<endl;

    cout<<endl;
    c.radius=1; //modify radius
    cout<<" Radius:  "<<c.radius<<endl;
    cout<<" Area:  "<<c.area()<<endl;
    cout<<" Circumference:  "<<c.circumference()<<endl;
    return 0;
}
```

- ✓ Look at Listing 7-2 for a better designed Circle class.
  - o All data members are public while member functions are private.
  - o Two member functions to get and set the data member radius have been provided to enhance encapsulation.

```
/*Listing 7-2
Purpose: A better designed class. Achieves information hiding.
Better encapsulated class.
Data members can only be accessed through methods.
Two methods are provided:
    a) setRadius() to modify radius- a getter member function.
    b) getRadius() to retrieve and return radius- -a setter member function
*/
#include<iostream>
using namespace std;

class Circle{
    private:
        float radius;
    public:
        float const PI=3.14;
        float getRadius(){return radius;}
        void setRadius(float r){ radius=r;}
        float area(){return PI*radius*radius;}
        float circumference(){return 2*PI*radius;}
};

int main()
{
    Circle c;
    c.radius=2.3; //set radius--this won't work---comment it out for the code to work.
    c.setRadius(2.3);
    cout<<" Radius:  "<<c.getRadius()<<endl;
    cout<<" Area:  "<<c.area()<<endl;
    cout<<" Circumference:  "<<c.circumference()<<endl;
    cout<<endl;
    c.setRadius(1.5);
    cout<<" Radius:  "<<c.getRadius()<<endl;
    cout<<" Area:  "<<c.area()<<endl;
    cout<<" Circumference:  "<<c.circumference()<<endl;
    return 0;
}
```

7.7 Constructors and Destructors

- Default constructors and parameterized constructors
- Constructor overloading
- Destructors and object cleanup

7.7.1 Constructor

- ✓ A special member function that is invoked when a class is instantiated. The constructor can be used to initialize the data members of the newly instantiated object. The constructor has the same name as the name of the class.
- ✓ Like any other function, a constructor can be *overloaded*.
- ✓ Two types of constructors can be defined:
  - a default constructor and
  - a parametrized constructor.

7.7.1.1 Default constructor

- ✓ A default constructor takes no parameters and takes the form:

```
ClassName(){//notice there are no parameters
        //body of constructor
}
```

- e.g.:

```
Circle()
{
        radius=0.0;
}
```

✓ The default constructor is called automatically.

7.7.1.2 *Parametrized constructor*

✓ A parametrized constructor takes inputs (parameters).
✓ The syntax of a parametrized constructor definition is given below:

```
//notice there are no parameters
ClassName(datatype var1, datatype var2,…datatype var3){
        //body of constructor
}
```

○ e.g.:

```
Circle(float r)
{
        radius=r;
}
```

✓ Look at Listing 7-3 that shows the use of default and parametrized constructors.

```
/*Listing 7-3
Purpose: Class showing good encapsulation and overloaded constructors.
*/
#include<iostream>
using namespace std;

class Circle{
    private:
        float radius;
    public:
        float const PI=3.14;
        Circle(){radius=0.0;}
        Circle(float r){radius=r;}
        float getRadius(){return radius;}
        void setRadius(float r){ radius=r;}
        float area(){return PI*radius*radius;}
        float circumference(){return 2*PI*radius;}
};

int main()
{
    Circle c;//calls the default constructor
    cout<<" Radius:  "<<c.getRadius()<<endl;
    cout<<" Area:  "<<c.area()<<endl;
    cout<<" Circumference:  "<<c.circumference()<<endl;

    c.setRadius(2.3);//modify radius
    cout<<endl;
    cout<<" Radius:  "<<c.getRadius()<<endl;
    cout<<" Area:  "<<c.area()<<endl;
    cout<<" Circumference:  "<<c.circumference()<<endl;
    cout<<endl;

    Circle c1(1.5); //calls the parametrized constructor
    cout<<" Radius:  "<<c1.getRadius()<<endl;
    cout<<" Area:  "<<c1.area()<<endl;
    cout<<" Circumference:  "<<c1.circumference()<<endl;
    return 0;
}
```

✓ Note: You can exclude the default constructor from a class definition to ensure that only parametrized constructors are used.
✓ Look at Listing 7-4 that shows a class with only the parametrized constructors.

```
/*Listing 7-4
Purpose: Class showing good encapsulation, excludes the default constructor
and provides only a parametrized constructor.
The parametrized constructor forces the client code -main in this case- to
provide the radius when instantiating a Circle class.
*/
#include<iostream>
using namespace std;

class Circle{
    private:
        float radius;
    public:
        float const PI=3.14;
        Circle(float r){radius=r;}
        float getRadius(){return radius;}
        void setRadius(float r){ radius=r;}
        float area(){return PI*radius*radius;}
        float circumference(){return 2*PI*radius;}
};

int main()
{
   /*
   //Uncomment the following four lines of code and attempt to run the program.
   What did you notice?
    Circle c;//calls the default constructor---this will
    cout<<" Radius:  "<<c.getRadius()<<endl;
    cout<<" Area:  "<<c.area()<<endl;
    cout<<" Circumference:  "<<c.circumference()<<endl;
*/

    Circle c1(1.5); //calls the parametrized constructor
    cout<<" Radius:  "<<c1.getRadius()<<endl;
    cout<<" Area:  "<<c1.area()<<endl;
    cout<<" Circumference:  "<<c1.circumference()<<endl;
    return 0;
}
```

7.7.2 Destructors and object cleanup

- ✓ Destructor: A special function that is automatically invoked when an object goes out of scope.
- ✓ It performs object clean-up.
    - Therefore, it can be used to free up any resources that are currently being used by the object.
    - Use the destructor to reset variables or to release dynamically allocated memory.
- ✓ The syntax of a destructor is:

```
~ClassName(){//notice the tilde ~
        //body of destructor
}
```

- ✓ Unlike a constructor, a destructor cannot be overloaded.
- ✓ Look at Listing 7-5 that demonstrates how to use destructors to handle a class that has dynamically allocated pointers.

Note:

1. The constructor and the destructor are the only methods that have no return type (not even void).
    - However, despite the constructor not having a return type, it returns an object of its class type.

2. Constructors, the destructor, and operator overloading are used to implement utility classes in C++ such as std::string, container classes such as std::vector available in the Standard Templates Library (STL).

```
/*Listing 7-5
Purpose: Demonstrating the use of a destructor.
*/
#include<iostream>
using namespace std;

class NumberApp{
    private:
        int *nums;
        unsigned int length;
    public:
        NumberApp(unsigned int len){
                length=len;
                nums=new int[len];//attempt to allocate memory
            ;}
            ~NumberApp(){
                cout<<"Destructor called. releasing memory!"<<endl;
                delete [] nums;
            }
            //other member functions
};

int main()
{
   unsigned int noOfValues;
   cout<<"Enter the number of integers to store:";
   cin>>noOfValues;
   NumberApp numApp(noOfValues);
    return 0;
}
```

7.8 Scope resolution operator ::

- ✓ So far, all our member functions have been defined inside the class declaration. This method works well for simple methods like the ones we have used so far. However, the code can become unreadable for many large and complex member functions.
- ✓ Another way to define member functions is to *include only the function prototypes in the class declaration* and then *define the member functions outside the class*.
- ✓ To do this, we can use the scope resolution operator `::`. For instance, `Circle::getRadius` is read as `getRadius` that belongs to the `Circle` class.
- ✓ Look at Listing 7.6 for an example where all member functions are defined outside the class declaration. Compare the codes in Listing 7-6 and Listing 7-3.

7.9 The this pointer

- ✓ `this` keyword gives access to an object of the current object. It gives accesses to the address of the current object. Its scope is the class.
- ✓ When a member function invokes another member function, the compiler sends `this` pointer as an implicit parameter in the function call.
- ✓ `this` can also be used in setters if the parameter name is the same as the member variable name.
    - o Using `this`, it is not necessary to use different names for parameters and member variables as we have been doing in Listing 7-2 to 7-6. Therefore, using this will help avoid name collisions.
- ✓ Look at Listing 7-7 for an illustration. Examine the definition of `setRadius`. Compare it with the code in Listing 7-2.

```
/*Listing 7-6
Purpose: Class showing good encapsulation and overloaded constructors.
The member functions are defined outside the class declaration.
*/
#include<iostream>
using namespace std;

class Circle{
    private:
        float radius;
    public:
        float const PI=3.14;
        //provide function prototypes
        Circle();
        Circle(float r);
        float getRadius();
        void setRadius(float r);
        float area();
        float circumference();
};
//provide member function definitions
Circle::Circle(){//default constructor
    radius=0.0;
}
Circle::Circle(float r){//parametrized constructor
    radius=r;
}
float  Circle::getRadius(){//getter function
    return radius;
}
void  Circle::setRadius(float r){ //setter function
    radius=r;
}
float  Circle::area(){
    return PI*radius*radius;
}
float  Circle::circumference(){
    return 2*PI*radius;
}
//define main function
int main()
{
    Circle c;//calls the default constructor
    cout<<" Radius:  "<<c.getRadius()<<endl;
    cout<<" Area:  "<<c.area()<<endl;
    cout<<" Circumference:  "<<c.circumference()<<endl;

    c.setRadius(2.3);//modify radius
    cout<<endl;
    cout<<" Radius:  "<<c.getRadius()<<endl;
    cout<<" Area:  "<<c.area()<<endl;
    cout<<" Circumference:  "<<c.circumference()<<endl;
    cout<<endl;

    Circle c1(1.5); //calls the parametrized constructor
    cout<<" Radius:  "<<c1.getRadius()<<endl;
    cout<<" Area:  "<<c1.area()<<endl;
    cout<<" Circumference:  "<<c1.circumference()<<endl;
    return 0;
}
```

7.10 Static member functions and data members

- ✓ Static members are associated with all objects, not specific objects.
    - o Therefore, a static data member will have the same value shared by all objects. For instance, a constant that is defined within a class can be made a static member.

    - Similarly, a static member function is called using the name of the class.
- ✓ A static member has class scope.
- ✓ A class's private or protected static members can be accessed through the class's member functions or friend classes and functions.
- ✓ A public static member can be accessed by merely prefixing the name with the name of the class and the scope resolution operator.
- ✓ For instance, if pi is a public static member of Circle, we can access it using Circle::pi.
- ✓ It is advisable to provide static member functions to access other static members.
- ✓ Static data members and static member functions can be used when no object of the class has been instantiated.
- ✓ Look at Listing 7-8 for an example with static members. The example helps count the number of Circle objects that have been instantiated.

7.11 struct vs class

- ✓ The `struct` keyword is used to create structs in C++. It is derived from C language.
- ✓ However, the C++ compiler treats a struct the same way it treats a class in C++. However, the following is an important exception:
    - If access specifiers are not given for `struct` type, the members default to `public`. The default is `private` for `class` type.
- ✓ Understanding how to use structs (an aggregate type to create record types) is left as an exercise (further reading) for you.

```
/*Listing 7-7
Purpose: A better designed class. But not very good because there are no constructors.
Achieves information hiding.
Data members can only be accessed through methods. Uses this keyword.
Two methods are provided:
    a) setRadius() to modify radius;
    b) getRadius() to retrieve and return radius
*/
#include<iostream>
using namespace std;

class Circle{
    private:
        float radius;
    public:
        float const PI=3.14;
        float getRadius(){return radius;}
        void setRadius(float radius){
         this->radius=radius; /*See the use of "this" on the left hand-side.
            "this->radius": "radius" that belongs to "this object".
            This distinguishes it from the right hand-side "radius"---
            which refers to the value being received as input.*/

         }
        float area(){return PI*radius*radius;}
        float circumference(){return 2*PI*radius;}
};

int main()
{
    Circle c;
    //c.radius=2.3; //set radius--this won't work---comment it out for the code to work.
    c.setRadius(2.3);
    cout<<" Radius:  "<<c.getRadius()<<endl;
    cout<<" Area:  "<<c.area()<<endl;
    cout<<" Circumference:  "<<c.circumference()<<endl;
    cout<<endl;
    c.setRadius(1.5);
    cout<<" Radius:  "<<c.getRadius()<<endl;
    cout<<" Area:  "<<c.area()<<endl;
    cout<<" Circumference:  "<<c.circumference()<<endl;
    return 0;
}
```

7.12 Summary

- ✓ Object oriented programming supports encapsulation and information hiding.
- ✓ Encapsulation is achieved when all the data and the methods (member functions) that operate on the data *are bundled into an object* such that any operations on the data only occur through the methods.
- ✓ Information hiding is achieved by *use of access specifiers*.
  - o By default, data members should be private while member functions should be public. However, some member functions can also be private.
  - o The protected access specifier can also be used to indicate members that are accessible in classes that are derived from the current class through inheritance (See Lesson 8).
- ✓ With good encapsulation, it is advisable to provide *setter methods to modify member variables* and *getter methods to retrieve values from member variables*.
- ✓ When defining classes, provide constructors to make it possible to properly initialize the data members.
  - o Constructors take the same name as the class and have no return type.
  - o Constructors can be overloaded just like any other function.
- ✓ Provide a destructor to be used to deallocate resources when an object is destroyed.
- ✓ Member functions can be defined outside the class declaration to improve readability. We use the scope resolution operator (`::`) to achieve this as seen in Listing 7-6.
- ✓ Objects communicate with each other through message passing.
- ✓ Key object-oriented concepts:
  - o *Abstraction*: a method for simplifying the solution to a problem by hiding the implementation details. This is the key concept used when defining classes.
  - o *Encapsulation*: the process of bundling into an object all the data and methods (member functions) that operate on the data.
  - o *Inheritance*: the process of creating a new class by acquiring properties from an existing class. The existing class is called the base class while the new class is the derived class. The new class may modify some of the properties of the existing class.
  - o *Polymorphism*: the ability of an entity to exist in many forms. In programming, this occurs when an object responds differently to the same *message*.
- ✓ Other object-oriented concepts such as *inheritance* and *polymorphism* will be considered in Lesson 8.

```
/*Listing 7-8
Purpose: Demonstrating the use of static members. */
#include<iostream>
using namespace std;

class Circle{
    private:
        float radius;
        static unsigned int count;//number of circles---static data member.
    public:

        float const PI=3.14;
        //provide function prototypes
        Circle();
        Circle(float );
        ~Circle();
        float getRadius();
        void setRadius(float );
        float area();
        float circumference();
        static unsigned int getCount();//returns the count of objects --static member function.

};
//provide member function definitions
Circle::Circle(){//default constructor
    radius=0.0;
    count++;//increment count of Circles.
}
Circle::Circle(float radius){//parametrized constructor
    this->radius=radius;
    count++; //increment count of Circles.
}
//Destructor should reduce the count of Circle objects
Circle::~Circle()
{
    cout<<"~Circle() destructor called for Circle of radius:"<<radius<<endl;
    count--; //decrement static count of Circles.
}
float  Circle::getRadius(){//getter function
    return radius;
}
void  Circle::setRadius(float radius){ //setter function
     this->radius=radius;
}
float  Circle::area(){
    return PI*radius*radius;
}
float  Circle::circumference(){
    return 2*PI*radius;
}
//returns number of Circle objects
 unsigned int Circle::getCount()
{
    return count;
}
//Initializes the static data member at the global namespace scope
unsigned int Circle::count{0};
//define main function
int main()
{
    cout<<"Number of Circle objects before first instantiation:"<<Circle::getCount()<<endl;
    Circle c;//calls the default constructor
    cout<<" Radius:  "<<c.getRadius()<<endl;
    cout<<" Area:  "<<c.area()<<endl;
    cout<<" Circumference:  "<<c.circumference()<<endl;
    cout<<"Number of Circle objects after first instantiation:"<<Circle::getCount()<<endl;

    c.setRadius(2.3);//modify radius
    cout<<endl;
    cout<<" Radius:  "<<c.getRadius()<<endl;
    cout<<" Area:  "<<c.area()<<endl;
    cout<<" Circumference:  "<<c.circumference()<<endl;
    cout<<endl;

    Circle c1(1.5); //calls the parametrized constructor
    cout<<" Radius:  "<<c1.getRadius()<<endl;
    cout<<" Area:  "<<c1.area()<<endl;
    cout<<" Circumference:  "<<c1.circumference()<<endl;
    cout<<"Number of Circle objects after second instantiation:"<<Circle::getCount()<<endl;
    return 0;
}
```

8 CLASSES AND OBJECTS

8.1 Learning Outcomes:

By the end of this session, students should be able to:

- Gain a deeper understanding of classes and objects in C++.
- Write C++ programs using inheritance and derived classes.
- Explain and apply polymorphism and function overriding in writing programs.

8.2 Outline

- Inheritance and Derived Classes:
    - Introduction to inheritance and its types
    - Creating derived classes from base classes
    - Inheriting member functions and data members
- Polymorphism and Function Overriding:
    - Polymorphism and its importance in OOP
    - Virtual functions and function overriding
    - Dynamic binding and run-time polymorphism

8.3 Inheritance and Derived Classes:

8.3.1 Introduction to inheritance and its types

✓ *Inheritance*: You "*create a new class that absorbs an existing class's capabilities and customizes or enhances them*"----C++ How to Program by Deitel & Deitel.

✓ Inheritance makes it possible to reused tested and high-quality code.

✓ The idea is to specify the new class in terms of the existing class using an "is-a" relationship.

- The new class inherits members from an existing class.
- Give three classes BaseClass, Derived Class1, and DerivedClass2, Fig. 1 shows a Unified Modeling Language (UML[3]) class diagram that shows the "is-a" relationship.

[3] The UML 2 class diagram - IBM Developer

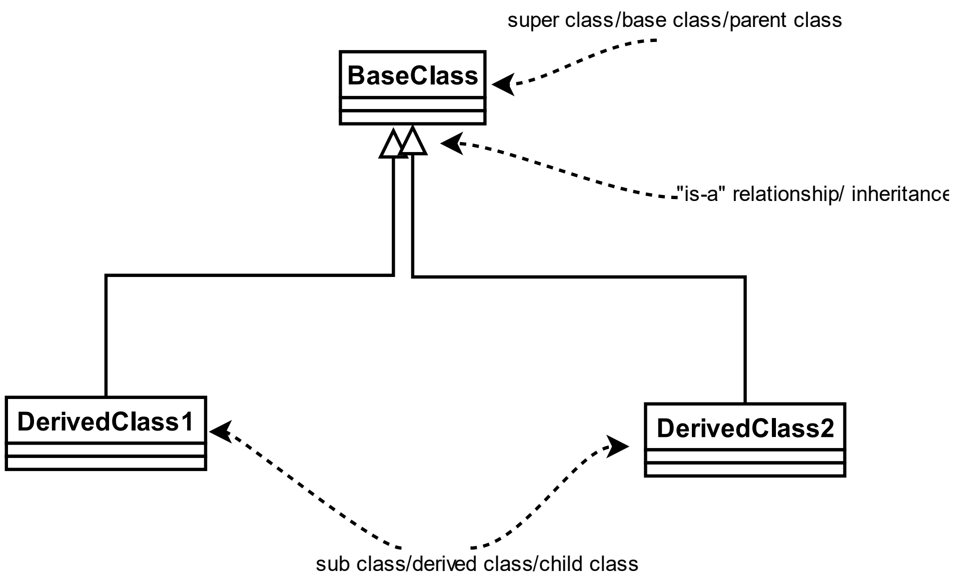

super class/base class/parent class
BaseClass
"is-a" relationship/ inheritance
DerivedClass1
DerivedClass2
sub class/derived class/child class

Figure 2: UML Diagram showing inheritance

- ✓ The class from which you inherit is called a base class (also superclass or parent class).
- ✓ The class created through inheritance is called the derived class (also subclass or child class).
- ✓ The derived class acquires all the properties of the base class but may provide additional properties.
    - o Therefore, it is considered a *specialization* of the base class.
    - o The base class is considered more *generalized*.
- ✓ Fig 1. Can also be considered a class hierarchy since it shows the inheritance relationships among classes.
- ✓ Figures 2 and 3 show two class hierarchies.
    - o Fig 2 shows the hierarchy involving Employee, Faculty, Staff, and StudentWorker.
    - o Fig 3 shows the hierarchy involving Shape, Circle, Rectangle, Triangle, Cylinder, Sphere, and Cube.
- ✓ C++ supports <u>*multiple inheritance*</u>: this enables a class to be created by *inheriting* properties from two or more classes at the same time.

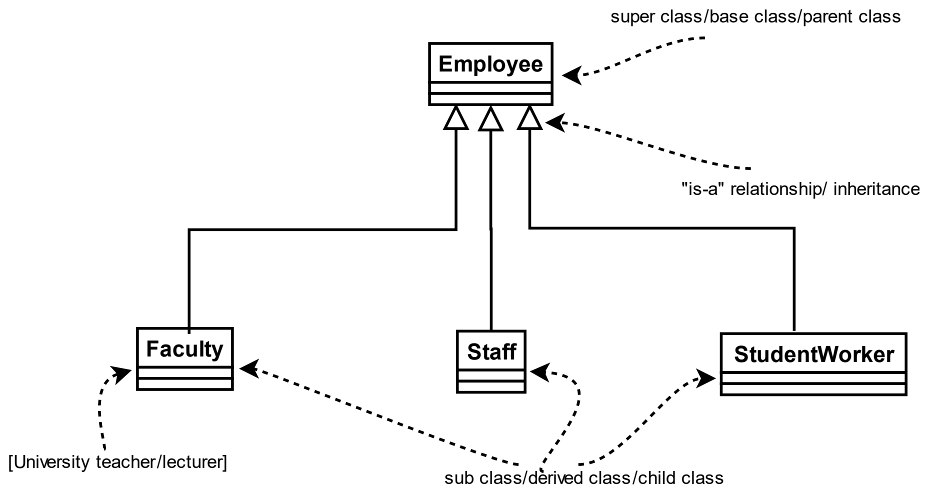
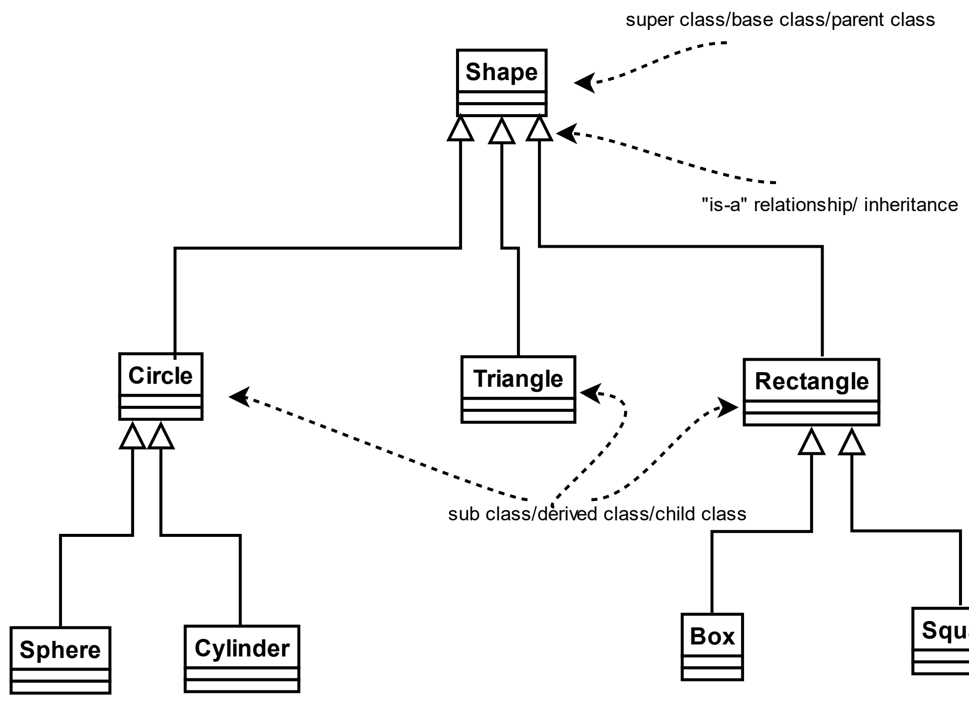

super class/base class/parent class
Employee
"is-a" relationship/ inheritance
Faculty
Staff
StudentWorker
[University teacher/lecturer]
sub class/derived class/child class

Figure 3: Employee class hierarchy



super class/base class/parent class
Shape
"is-a" relationship/ inheritance
Circle
Triangle
Rectangle
sub class/derived class/child class
Sphere
Cylinder
Box
Square

Figure 4: Shape class hierarchy with multi-level inheritance

- ✓
- ✓ C++ offers three types on inheritance:
    - Public inheritance: derived class is a base class relationship.
    - Protected or private inheritance: derived class has a base class relationship.
- ✓ Private inheritance: All public members of the base class are private and inaccessible to anyone with instance of the derived class.
    - Derived class has access to all public and protected members of the base class.
    - Classes outside the inheritance hierarchy with an instance of the derived class cannot access public members of the base class.
- ✓ Protected inheritance: All public members of the base class are private and inaccessible to anyone with instance of the derived class.
    - Similar to private inheritance except that new classes derived from the derived class can access public and protected members of the base class.

8.3.2 Creating derived classes from base classes

- ✓ The syntax for inheritance is:

```
class BaseClassName
{
        //base class specification
};
class DerivedClassName:<access-specifier> BaseClassName
{
        //derived class specification
};
```

- Where:
    - `<access-specifier>` can be public, private, and protected as appropriate.
- ✓ All protected members of the parent class can be modified my members of the child class.

✓ Figure 4 provides a more elaborate UML class diagram. All data members have been marked as protected.

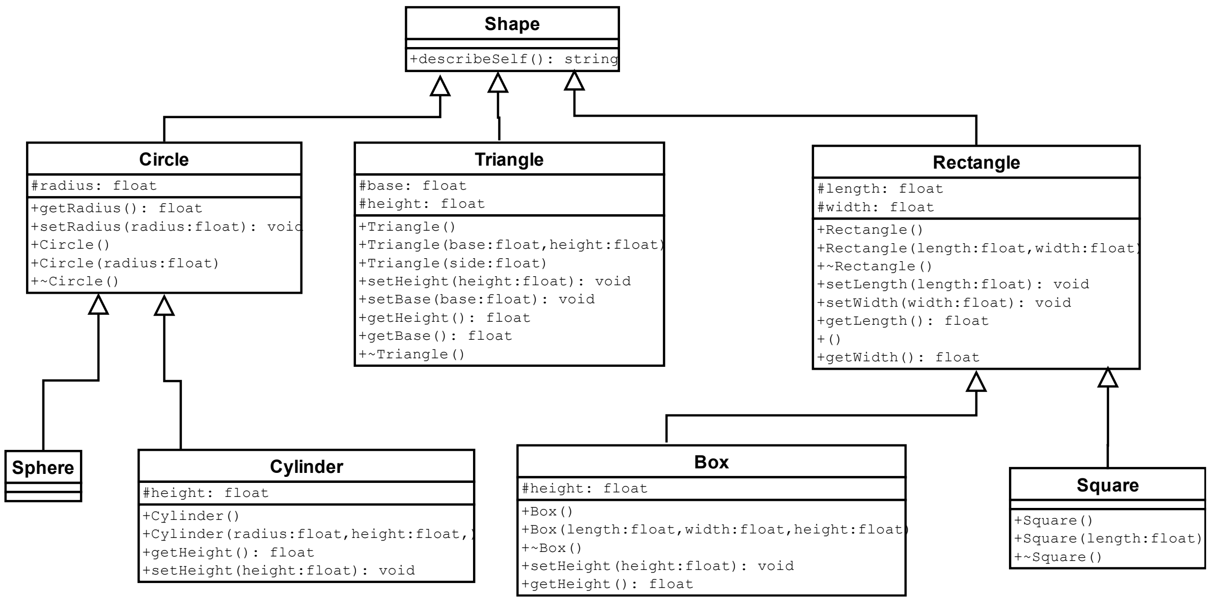


Figure 5: Shape class hierarchy with details about a subset of members

8.3.3 Base Class Initialization

✓ When initializing a derived class, its base class part also needs to be initialized.
✓ This requires that you pass parameters to the base class constructor. Recall that constructors are used to initialize objects.
✓ You can pass parameters to a base class constructor using an initialization list.
✓ The initialization list takes the form `BaseClassName(param1,param2,…,para,N)`.

8.3.4 Function overriding

✓ You can change the definition of a base class member function through function overriding.
✓ The function will have the same return type, name, and parameter list in both the base and derived classes.

8.3.5 Invoking an overridden method

✓ You can use the scope resolution operator with the name of the base class to invoke an overridden base class function.
✓ This method allows you to call overridden member functions within the derived class functions or main.
✓ Listing 8-1 shows inheritance but no base class initialization.
✓

```cpp
/*Listing 8-1
Purpose: Provides a partial implementation of the Shape class hierarchy.
It shows public inheritance; protected and public members
and function overriding.
*/
#include<iostream>
#include<string>
using namespace std;

class Shape{
public:
    string describeSelf(){
            return "I am a shape.";
    }
};
class Circle: public Shape{
    protected:
        float radius;
    public:
        float const PI=3.14;
        Circle();
        Circle(float);
        ~Circle();
        float getRadius();
        void setRadius(float);
        float area();
        float circumference();
        string  describeSelf();
};
//provide member function definitions
Circle::Circle(){//default constructor
    radius=0.0;
}
Circle::Circle(float radius){//parametrized constructor
    this->radius=radius;
}
//Destructor should reduce the count of Circle objects
Circle::~Circle()
{
    cout<<"~Circle() destructor called for Circle of radius:"<<radius<<endl;
}
float  Circle::getRadius(){//getter function
    return radius;
}
void  Circle::setRadius(float radius){ //setter function
     this->radius=radius;
}
float  Circle::area(){
    return PI*radius*radius;
}
float  Circle::circumference(){
    return 2*PI*radius;
}
string  Circle::describeSelf(){//f]unction overriding
    string desc="\n-----------Circle-----------";
    desc+="\n\tRadius:"+to_string(radius); //using string concatenation
                                          //std::to_string converts a number to a string.
                                          // to_string()—used in C++11 and above only
    desc+="\n\tArea:"+to_string(area());
    desc+="\n\tCircumference:"+to_string(circumference())+"\n";
    return desc;
}

int main()
{
    Circle c;
    c.setRadius(2.3);
    cout<<c.describeSelf()<<endl;
    cout<<endl;
    Circle c2(3);
    cout<<c2.describeSelf()<<endl;
    return 0;
}
```

8.4 Polymorphism and Function Overriding:

8.4.1 Polymorphism and its importance in OOP

8.4.2 Virtual functions and function overriding

8.4.3 Dynamic binding and run-time polymorphism

8.5 Summary

9 FILE HANDLING

9.1 Learning Outcomes

By the end of this session, students should be able to:

- Describe the concept of file handling in C++.
- Learn how to open, read, and write files using file streams.
- Gain familiarity with input/output stream manipulations for file handling.
- Learn how to handle errors and exceptions during file operations.

9.2 Outline

- Working with Files in C++:
  - File streams: ifstream and ofstream
  - Opening and closing files
  - File modes and flags
- Reading and Writing Files:
  - Reading data from files
  - Writing data to files
  - File position pointers
- Handling Input/Output Streams:
  - Manipulating input/output streams
  - Formatting data in file operations
  - File stream error handling
- Error Handling with Exceptions:
  - Introduction to exceptions
  - Throwing and catching exceptions
  - Handling file-related exceptions

9.3 Working with Files in C++

9.4 Reading and Writing Files

9.5 Handling Input/Output Streams

9.6 Error Handling with Exceptions

9.7 Summary

REFERENCE RESOURCES/BOOKS

- Sams Teach Yourself C++ in One Hour a Day, 9th Edition. Sams, 2022. Print. Link: https://cmu.primo.exlibrisgroup.com/permalink/01CMU_INST/6lpsnm/alma991019904891304436
- Deitel, Paul J., Harvey M. Deitel, and Paul J. Deitel. C++ How to Program: Introducing the New C++ 14 Standard. Tenth edition. Hoboken, New Jersey: Pearson Education, 2017. Print. Link: https://cmu.primo.exlibrisgroup.com/permalink/01CMU_INST/6lpsnm/alma991019578087204436