Modern C++

An effective short way

By

Mustapha Ossama Abdelhalim

2024

Contents

[Chapter 1 Starter and Installation 7](#_Toc173506055)

[1.1 For windows 7](#_Toc173506056)

[1.2 For Linux 7](#_Toc173506057)

[Chapter 2 Basics 9](#_Toc173506058)

[Introduction 10](#_Toc173506059)

[1.1. Hello World 11](#_Toc173506060)

[Chapter 1 Variables and data types 12](#_Toc173506061)

[2.1 Primitive datatypes 13](#_Toc173506062)

[2.2 Derived datatypes 17](#_Toc173506063)

[2.2.1 Arrays 17](#_Toc173506064)

[2.2.2 Functions 19](#_Toc173506065)

[2.3 User-defined datatypes 20](#_Toc173506066)

[2.3.1 Structs 20](#_Toc173506067)

[2.3.2 Enum 28](#_Toc173506068)

[2.3.3 Union 30](#_Toc173506069)

[2.4 Operators and Expressions 33](#_Toc173506070)

[2.4.1 Arithmetic operators: +, -, \*, /, % 33](#_Toc173506071)

[2.4.2 Relational operators: ==, !=, >, <, >= 34](#_Toc173506072)

[2.4.3 Logical operators: &&, ||, ! 34](#_Toc173506073)

[2.4.4 Bitwise operators: &, |, ^, ~, <<, >> 35](#_Toc173506074)

[2.4.5 Assignment operators: =, +=, -=, \*=, /=, %=, &=, |=, ^=, <<=, >>= 35](#_Toc173506075)

[2.4.6 unary operators (Increment and decrement): ++, -- 38](#_Toc173506076)

[2.4.7 ternary operator: ?: 38](#_Toc173506077)

[2.5 Control Structures 39](#_Toc173506078)

[2.5.1 Conditional statements: if, if-else, switch-cases 39](#_Toc173506079)

[2.5.2 Loops 43](#_Toc173506080)

[2.5.3 Jump statements: break, continue, goto, return 47](#_Toc173506081)

[2.6 Final project 49](#_Toc173506082)

[Chapter 3 Pointers and Memory management 51](#_Toc173506083)

[3.1 Introduction to Pointers 52](#_Toc173506084)

[3.1.1 Pointer definition 54](#_Toc173506085)

[3.1.2 Operations on Pointers 56](#_Toc173506086)

[3.2 Dynamic Memory allocation 58](#_Toc173506087)

[3.2.1 new and delete operators 58](#_Toc173506088)

[3.2.2 Allocating memory for single variables and arrays 58](#_Toc173506089)

[3.2.3 Linked List 59](#_Toc173506090)

[3.2.4 Memory Leaks 65](#_Toc173506091)

[3.3 Smart pointers 67](#_Toc173506092)

[3.3.1 Unique pointer 67](#_Toc173506093)

[1.1.1. Shared pointer 69](#_Toc173506094)

[1.1.1. Weak pointer 70](#_Toc173506095)

[Chapter 4 Functions 72](#_Toc173506096)

[4.1 Function Declaration and Definition 74](#_Toc173506097)

[4.1.1 Function declaration 74](#_Toc173506098)

[4.1.2 Function definition 74](#_Toc173506099)

[4.1.3 Default arguments 77](#_Toc173506100)

[4.2 Overloading and Inline Functions 78](#_Toc173506101)

[4.2.1 Inline functions 78](#_Toc173506102)

[4.2.2 Overloading 80](#_Toc173506103)

[4.3 Recursive Functions 82](#_Toc173506104)

[4.4 Pass by value, reference and pointer 84](#_Toc173506105)

[4.4.1 Pass by value 84](#_Toc173506106)

[4.4.2 Pass by reference 85](#_Toc173506107)

[1.1.1. Pass by pointer 86](#_Toc173506108)

[4.5 Final Project 89](#_Toc173506109)

[Chapter 5 Preprocessor Directives 94](#_Toc173506110)

[5.1 Macros 94](#_Toc173506111)

[5.1.1 #define and #undef 94](#_Toc173506112)

[5.1.2 Function-like macro 96](#_Toc173506113)

[5.2 Conditional Compilation 99](#_Toc173506114)

[5.3 File guards 101](#_Toc173506115)

[Chapter 6 Compilation Process 105](#_Toc173506116)

[6.1 Compilation process 106](#_Toc173506117)

[6.1.1 Preprocessor directive 108](#_Toc173506118)

[6.1.2 Compiler 109](#_Toc173506119)

[6.1.3 Assembler 111](#_Toc173506120)

[6.1.4 Linker 112](#_Toc173506121)

[6.2 Compile multiple files 113](#_Toc173506122)

[6.2.1 Convert each file into object files then link 113](#_Toc173506123)

[6.2.2 Convert all .cpp files into one executable file 113](#_Toc173506124)

[Chapter 7 Object oriented programming OOP 116](#_Toc173506125)

[7.1 Classes and Objects 118](#_Toc173506126)

[7.1.1 Class definition and declaration 121](#_Toc173506127)

[7.1.2 Access specifiers: public, private, protected 121](#_Toc173506128)

[7.1.3 Member variables and member functions 122](#_Toc173506129)

[7.1.4 Object instantiation 122](#_Toc173506130)

[7.2 Constructors and Destructors 123](#_Toc173506131)

[7.2.1 Default constructor 123](#_Toc173506132)

[7.2.2 Parameterized constructor 124](#_Toc173506133)

[7.2.3 Copy constructor 125](#_Toc173506134)

[7.2.4 Destructor 127](#_Toc173506135)

[7.3 Inheritance 128](#_Toc173506136)

[7.3.1 Base and derived classes 128](#_Toc173506137)

[7.3.2 Types of inheritance 130](#_Toc173506138)

[7.3.3 Constructor and destructor calls in inheritance 131](#_Toc173506139)

[1.1. Encapsulation 132](#_Toc173506140)

[7.3.4 Data hiding 133](#_Toc173506141)

[7.3.5 Setter and Getter (Accessor and mutator functions) 133](#_Toc173506142)

[7.4 Polymorphism 136](#_Toc173506143)

[7.4.1 Compile-time polymorphism: function overloading, operator overloading 137](#_Toc173506144)

[7.4.2 Runtime polymorphism: virtual functions, pure virtual functions, abstract classes 139](#_Toc173506145)

[7.5 Abstraction 140](#_Toc173506146)

[7.5.1 Abstract classes and interfaces 140](#_Toc173506147)

[7.5.2 Virtual function and pure virtual function 141](#_Toc173506148)

[7.6 Static Members 142](#_Toc173506149)

[7.6.1 Class-level data and behavior 142](#_Toc173506150)

[7.6.2 Static member variables and functions 142](#_Toc173506151)

[7.7 Multiple Inheritance 145](#_Toc173506152)

[7.7.1 Diamond problem and virtual inheritance 146](#_Toc173506153)

[7.8 Operator overloading 148](#_Toc173506154)

[7.8.1 Overloading unary operators 149](#_Toc173506155)

[7.8.2 Overloading binary operators 150](#_Toc173506156)

[7.8.3 Insertion and extraction overloading 151](#_Toc173506157)

[7.9 Rules 152](#_Toc173506158)

[7.9.1 Rule of Three 152](#_Toc173506159)

[7.10 Final Project 153](#_Toc173506160)

[Chapter 8 Templates 154](#_Toc173506161)

[8.1 Function templates 155](#_Toc173506162)

[8.1.1 Template syntax and usage 155](#_Toc173506163)

[8.1.2 Specialization of function templates 157](#_Toc173506164)

[8.2 Class Templates 158](#_Toc173506165)

[8.2.1 Template classes 158](#_Toc173506166)

[8.2.2 Specialization of class templates 160](#_Toc173506167)

[Chapter 9 Standard Template Library (STL) 163](#_Toc173506168)

[Chapter 10 Exception Handling 164](#_Toc173506169)

[Chapter 11 File I/O 166](#_Toc173506170)

[Chapter 12 Multithreading and Concurrency 167](#_Toc173506171)

[Chapter 13 GDB Debugger 168](#_Toc173506172)

[Chapter 14 Others 169](#_Toc173506173)

[Chapter 15 Modern C++ 170](#_Toc173506174)

# Starter and Installation

Modern C++ starts with C11, this book introduce C11 and later on, the moving to C17 section

## For windows

1. Go to [winlibs.com](https://winlibs.com/)
2. Determine which list you will choose from UCRT runtime if you are using windows 10 or 11, or choose MSVCRT runtime if you are using older versions of windows.
3. If you will use the gcc for application that runs only on windows choose MCF threads, if you are using application that runs on windows and later maybe used on Linux distribution; choose POSIX threads

I will choose Win64 in UCRT runtime in POSIX thread section as I have windows 10 x64 and have 7zip installed see Figure 1 gcc releases

Figure 1 gcc releases

See this video for more details [LINK](https://www.youtube.com/watch?v=COZw6XetvR0)

## For Linux

Gcc is installed by default in ubuntu distribution

A screenshot of a computer

Description automatically generatedAfter downloading and extracting, move the mingw to c directory and get the bin path in environment variable and make sure to delete the old gcc form environment variables if exists. See Figure 2 adding bin folder path to environment variables

Figure 2 adding bin folder path to environment variables

A screen shot of a computer error

Description automatically generatedType in cmd gcc –version and you should see that gcc installed see Figure 3 verifying gcc installation

Figure 3 verifying gcc installation

# Basics

In this chapter, the Basics of C++ will be introduced as a refresher, the following topics will be introduced:

* **First program**
  + Compilation Hello World
* **Variables and Data Types**
  + Primitive types: int, char, float, double, bool
  + Derived types: arrays, pointers, references
  + User-defined types: structs, enums, classes
* **Operators and Expressions**
  + Arithmetic operators: +, -, \*, /, %
  + Relational operators: ==, !=, >, <, >=, <=
  + Logical operators: &&, ||, !
  + Bitwise operators: &, |, ^, ~, <<, >>
  + Assignment operators: =, +=, -=, \*=, /=, %=, &=, |=, ^=, <<=, >>=
  + Increment and decrement operators: ++, --
  + Conditional operator: ?:
* **Control Structures**
  + Conditional statements: if, if-else, nested if, switch-case
  + Looping statements: for, while, do-while
  + Jump statements: break, continue, goto, return

## Introduction

A **programming language** is set of instruction to perform a task, that’s it

In this book we will use notepad++ (even the simple preinstalled notepad will work fine) and compile our program in command prompt CMD, also its completely fine to use any integrated development environment (IDE), but make sure that you are using C11 gcc version.

C++ language has two types of files headers files(.h files) and source files (.cpp files), to compile the program and make it executable for windows (aka converted to .exe files to run on windows). you will use the following command in cmd

g++ -std=c++11 name.cpp -o name.exe

let’s break it down

* **g++** is the gcc command to perform compilation
* **-std-c++11** is flag to specify the version of c11
* **name.cpp** is our source file
* **-o** is the flag for output the .exe file
* **name.exe** is the name of output

## Hello World

1. Lets compile our first program !

#include<iostream>

int main**(){**

std**::**cout**<<**"Hello World"**;**

**return** 0**;**

**}**

* #include<iostream>

is library that permit us to output data and take input from user

* int main(){

return 0;}

Is the entry point for our program, all programs and applications should have that function (later functions will be expressed)

* std::cout<<”hello world”;

is the command to output hello world on the screen

1. make a file named Hello.cpp for example
2. type the code above
3. open cmd in the same directory as the file Hello.cpp
4. type: g++ -std=c++11 Hello.cpp -o Hello.exe
5. to run the program type: Hello.exe

the output should be as follows in Figure 4 first programA screenshot of a computer

Description automatically generated

Figure 4 first program

## Variables and data types

C++ has types to declare each variable, each variable should have a keyword to define if it integer (like 10, 99, and120) or decimal aka float like (10.2, 0.2, and 22.8) or character (like ‘a’ , ‘b’ and ‘c’) , this declaration specify:

* + How the variable stored in memory and takes how much of program memory
  + How operation change that variable

The types in C++ is as follows in Figure 5 Types in C++ :

Figure 5 Types in C++

## Primitive datatypes

Primary (primitive) data types are compiler dependent that means that the data types could be stored in different sizes for different compilers, in gcc compiler:

Type the following to examine the sizes of different datatypes, for example int (integer saved in 4 bytes in gcc).

#include<iostream>

**using** **namespace** std**;**

int main**(){**

cout**<<**"char has: "**<<sizeof(**char**)<<**endl**;**

cout**<<**"wide char has: "**<<sizeof(**wchar\_t**)<<**endl**;**

cout**<<**"int has: "**<<sizeof(**int**)<<**endl**;**

cout**<<**"float has: "**<<sizeof(**float**)<<**endl**;**

cout**<<**"double has: "**<<sizeof(**double**)<<**endl**;**

cout**<<**"long has: "**<<sizeof(**long**)<<**endl**;**

cout**<<**"long double has: "**<<sizeof(**long double**)<<**endl**;**

**return 0;**

**}**

A computer screen with white text

Description automatically generatedThe output should be in gcc compiler (maybe different for other compilers) see Figure 6:

Figure 6 datatypes sizes in gcc compiler

WHY we use different types of primitive (primary) variables?

To answer this question lets examine the following table

|  |  |  |  |
| --- | --- | --- | --- |
|  | details | Memory allocation (in GCC) | Syntax |
| Char | Store characters (‘a’,’b’,etc ) and integers from -128 to 127 | 1 | char x = ‘a’; |
| wchar\_t | Store much more characters than char | 2 | wchar\_t x = L’あ’ |
| Int | Store integer numbers till 2^31 positive integers and 2^31 negative integers | 4 | int x = 15; |
| float | Store decimal numbers | 4 | float x = 15.12; |

Also you have some modifiers like long/short and signed and unsigned

* Short: shorten integer to be usually stored in 2 bytes instead of 4 bytes which means that the value of short int will from 2^15 positives and 2^15 negatives not 2^31 positive integers and 2^31 negative integers.
* Long: will long the integers to be usually 12 bytes instead of 4 bytes which enlarge the range of that variable
* unsigned: signed (char or int or even short int) will store all bytes in positive for example, unsigned char has range of 0-255 while signed char (or char) has -128 to 127 (2^7 positives and 2^7 negatives)

back to our question, why we have different primitive data types?  
simply if I have variable that store integer variable of human age, I want only a variable that store positive integers of range 0 yrs old -150 yrs old, so char will be chosen or even short int (aka short) no need to take 4 bytes of integer as no human ever lived 2billion years !! so it waste of memory to choose int.

remember ! char variable store integers like 15 and characters like ‘a’ not only characters

what happen if:

1. what happen if: signed short int (aka short) which have range of -32768 to 32767, store number like 32770?

ans: the variable will overflow (aka return to zero and start to count gain the reminder) which mean that 32770is higher than the capability of unsigned short (32767) by 3 so the value will be 3 like in Figure 7 Variables overflow, note: same thing to unsigned short variable the start 0 and max is 65635 so if the number exceeds; it will start counting the reminder from 0.

Remember: when you exceed the variable range; overflow will happen



Figure 7 Variables overflow

1. what happen if: storing float number like 15.02 in integer variable like

int x = 15.02 ?

Ans: the float point (.02) will be truncated i.e. s is 15 only

SO always remember which primitive data types to choose !!;

Exercises on primitive (primary) Data types:

**Exercises** : introduction

Write C++ code to introduce someone, the introduction must include:

* + Name (string): like “Ahmed” , to declare string datatype called string like:

string name;

cin>>name;

* + Age (unsigned short) like 28
  + Salary (unsigned short) like 15000
  + GPA (float) like 3.5
  + NOTE: the data should be as input from user: to get input from user use cin>>var;

Answer:

#include <iostream>

**using** **namespace** std**;**

int main**()** **{**

string name**;**

unsigned short age**,**salary**;**

float gpa**;**

cout**<<**"enter your name"**<<**endl**;**

cin**>>**name**;**

cout**<<**"enter your age and salary "**<<**endl**;**

cin**>>**age**>>**salary**;**

cout**<<**"enter your gpa"**<<**endl**;**

cin**>>**gpa**;**

cout**<<**"Introduction\nMy name is:"**<<**name**<<**endl**;**

cout**<<**"I am "**<<**age**<<**"years old "**<<**"my salary is: "**<<**salary**<<**endl**;**

cout**<<**"my GPA is: "**<<**gpa**;**

**return** 0**;**

**}**

NOTE: \n between “ “ is as same as endl after cout which means start from new line (i.e start printing at the beginning of the new following line)

NOTE: using namesapace std; is used to write cout and cin without typing std::cout and std::cin

**Exercise** : bankClient

Write C++ program to show:

* Client name: string
* ID: int
* Deposit money: float

Answer in the GitHub repository: [LINK](https://github.com/MuOssama/MasteringCPP/tree/main/Basics/VariablesAndDatatypes)

All the previous was all about primitive datatypes, but how about derived and user defined datatypes? Recall Figure 5 Types in C++

**derived** datatypes are datatypes made from primitive

* Arrays
* Functions
* Pointers

**User defined** datatypes are datatypes that user build

* Struct
* Enum
* Union
* Class

Lets take them one by one:

## Derived datatypes

### Arrays

are list of some variables but must be same data type variable Like int list[3] clientAges; which means that we collect clientAges in one list instead of doing this: int client1Age; int client2Age; int client3Age;

So, to make the life easier we collect similar datatypes in one place called array

* + **Declaration**: datatype nameOfArray[number of item];

For example: int salaries[5];

* + **Accessing each element**: salaries[i] (i must be number from 0 to 4 as salaries have 5 items

The previous array called C-Array, C++ has much powerful arrays, these arrays have built-in method like size() and other to shorten your code

* + **Declaration**: array<datatype, itemNumbers> name;

For example: array<int, 5> salaries;

NOTE: don’t forget to include array (i.e #include <array>)

* + **Accessing each element**: salaries[i] (i must be number from 0 to 4 as salaries have 5 items

**Exercise** : arrays

Write C++ array of 5 integer contains some user salaries, don’t use c arrays, use C++ std array

#include<iostream>

#include<array>

**using** **namespace** std**;**

int main**(){**

array**<**int**,** 5**>**salaries**;**

//filling the array

**for(**int i**=**0**;**i**<**salaries**.**size**();**i**++){**

cout**<<**"enter the "**<<**i**<<**" element:"**;**

cin**>>**salaries**[**i**];**

cout**<<**"\n"**;**

**}**

//printing the array

**for(**int i**=**0**;**i**<**salaries**.**size**();**i**++){**

cout**<<**"the element "**<<**i**<<**" is: "**<<**salaries**[**i**]<<**"\n"**;**

**}**

**}**

### Functions

Imagine you want to intoduce 10 peaple (like in **Exercises 1**: introduction) the program was about 10 line for one person, do write same code for the 10 person (100 lines !!) OR you can write the code for general person once in a place called function and whenever you want to use that function, call that general function and specify your details

void introduction**(**string name**,** short age**,** short salary**,** float gpa **){**

cout**<<**"enter your name"**<<**endl**;**

cout**<<**"enter your age and salary "**<<**endl**;**

cout**<<**"enter your gpa"**<<**endl**;**

cout**<<**"Introduction\nMy name is:"**<<**name**<<**endl**;**

cout**<<**"I am "**<<**age**<<**"years old "**<<**"my salary is: "**<<**salary**<<**endl**;**

cout**<<**"my GPA is: "**<<**gpa**;**

**}**

You build the general function, you can now call it as many times as you want !!

introduction**(**“Ahmed”**,**26**,**15000**,**3.6**);**

introduction**(**Gamal**,**30**,**2500**,**3.8**);**

introduction**(**Mahmoud**,**22**,**1200**,**3.2**);**

we will know more about functions and pointers later.

## User-defined datatypes

### Structs

Struct is used when you want to declare and object that have many attributes (i.e. variable) but different data types, e.g you want to describe a student how have name (String), id (int), gpa (float), struct came to hold these attributes (variables) in one place called struct

**Example:** studentStruct

In this example, struct is made for student who have 3 atributes for example name (String), id (int), gpa (float).

//declaration

struct student**{**

string name**;**

int id**;**

float gpa**;**

**};**

int main**(){**

//create instance of a struct

student Ahmed**={**"Ahmed"**,**202410**,**3.45**};**

/\*Accessing

Accessing is done by dot operator .

\*/

cout**<<**"Name:"**<<**Ahmed**.**name**<<**" ID:"**<<**Ahmed**.**id**<<**" GPA:"**<<**Ahmed**.**gpa**<<**endl**;**

//Assigning an instance of struct

Ahmed**.**gpa **=** 3.58**;**

cout**<<**"Name:"**<<**Ahmed**.**name**<<**" ID:"**<<**Ahmed**.**id**<<**" GPA:"**<<**Ahmed**.**gpa**;**

**}**

NOTE: you can use comment to improve code readability:

* + - One line comment: using // comment
    - Multiline comment: using /\* comment \*/

1. **Declaration of struct**

struct name{

variable1;

variable2;

.

.

};

1. **Creating instance**
   * 1st way: after the deceleration

//declaration

struct student**{**

string name**;**

int id**;**

float gpa**;**

**};**

* + 2nd way: by using.. struct\_type struct\_name;

student Ahmed**={**"Ahmed"**,**202410**,**3.45**};**

NOTE: struct objects (instances) could be initialized of left to be assigned later

student Ahmed**;**

NOTE: in C++ you don’t have to use struct keyword in contrast in C

In C:

struct student Ahmed**={**"Ahmed"**,**202410**,**3.45**};**

in C++ struct is not necessary :

student Ahmed**={**"Ahmed"**,**202410**,**3.45**};**

1. **Accessing and Assigning**

Accessing done by dot operator

e.g cout**<<**"Name:"**<<**Ahmed**.**name**<<**" ID:"**<<**Ahmed**.**id**<<**" GPA:"**<<**Ahmed**.**gpa**<<**endl**;**

Assigning:

Ahmed**.**name=”Ahmed”;

**Exercise 3:** employee

Write a struct that refer to an employee that have name , salary, working hours

The answer in basics folder in the repository, see Figure 8 Exercise 3

A black background with white text

Description automatically generated

Figure 8 Exercise 3

1. **Methods**

Unlike C, in C++ we have methods in struct, methods are function inside structs or classes, Lets see how methods work

**Example:** structMethod

write employee struct that has name, salary, working hours, that get user data and print this data and apply bonus, so we must have 3 method(functions), see the output in Figure 9 Example

#include<iostream>

**using** **namespace** std**;**

struct employee**{**

string Name**;**

int salary**;**

short workingHrs**;**

//Method to enter employ data

void setData**(){**

cout**<<**"enter Name, Salary, Working Hrs respctivily:\n"**;**

//entering the employee data from user

cin**>>**Name**>>**salary**>>**workingHrs**;**

//printing the employee data

**}**

//Method to print employee data

void print**(){**

cout**<<**"employee: "**<<**Name**<<**" salary: "**<<**salary**<<**" working hours: "**<<**workingHrs**<<**endl**;**

**}**

//Method to apply bonus

char applyBonus**(**int bonus**){**

salary **=** salary **+** bonus**;**

**return** 's'**;**

**}**

**};**

int main**(){**

//create object of struct employee

employee emp1**;**

emp1**.**setData**();**

emp1**.**applyBonus**(**500**);**

emp1**.**print**();**

**}**

A black screen with white text

Description automatically generated

Figure 9 Example

1. **Constructors**

Constructor is type of method that is called by default when an instance is made, the purpose of a constructor is to initialize the object, setting up initial values for its members and performing any setup required.

**Example:** structConstructor

#include <iostream>

**using** **namespace** std**;**

struct Person **{**

string name**;**

int age**;**

// Constructor

Person**(**string n**,** int a**)** **:** name**(**n**),** age**(**a**)** **{**

cout **<<** "Constructor called for " **<<** name **<<** endl**;**

**}**

// Member function to display person details

void display**()** const **{**

cout **<<** "Name: " **<<** name **<<** ", Age: " **<<** age **<<** endl**;**

**}**

**};**

int main**()** **{**

// Creating an object of the Person struct

Person person1**(**"John Doe"**,** 30**);**

// Displaying the details of person1

person1**.**display**();**

**return** 0**;**

**}**

1. Inheritance

Inheritance used to create a child class of parent class or struct , e.g. if we created a class for employee that has name and age and member function named (method) role that is either writing() or reviewing() , we could create child of struct that inherit name and age but in writers employee child struct, writing() method will be created and in reviewer child struct, reviewing() method will be created.

**Example:** inheritance

#include <iostream>

#include <string>

// Base struct

struct Employee **{**

std**::**string name**;**

int age**;**

// Constructor for Employee

Employee**(**const std**::**string**&** n**,** int a**)** **:** name**(**n**),** age**(**a**)** **{}**

**};**

// Derived struct for Writer

struct Writer **:** public Employee **{**

// Constructor for Writer

Writer**(**const std**::**string**&** name**,** int age**)** **:** Employee**(**name**,** age**)** **{}**

// Specific method for Writer

void writing**()** const **{**

std**::**cout **<<** name **<<** " is writing a document." **<<** std**::**endl**;**

**}**

**};**

// Derived struct for Reviewer

struct Reviewer **:** public Employee **{**

// Constructor for Reviewer

Reviewer**(**const std**::**string**&** name**,** int age**)** **:** Employee**(**name**,** age**)** **{}**

// Specific method for Reviewer

void reviewing**()** const **{**

std**::**cout **<<** name **<<** " is reviewing a document." **<<** std**::**endl**;**

**}**

**};**

int main**()** **{**

// Create instances of Writer and Reviewer

Writer writer**(**"Alice"**,** 30**);**

Reviewer reviewer**(**"Bob"**,** 45**);**

// Use specific methods

writer**.**writing**();** // Output: Alice is writing a document.

reviewer**.**reviewing**();**// Output: Bob is reviewing a document.

**return** 0**;**

**}**

1. **Access Modifiers : Public, Private, Protected**

In the previous example, we could access display() method and any attribute (e.g name, age) anywhere, there are 3 places could a method or attribute called:

1. In the struct or class itself such enterData() call of age attribute check in the following example

struct Person **{**

string name**;**

int age**;**

// Member function to enter member data

void enterData**()** const **{**

cin **>>** name **>>** age**;**

**if(age<0)** cout **<<** “invalid age\n”;

**}**

// Member function to display person details

void display**()** const **{**

enterData();

cout **<<** "Name: " **<<** name **<<** ", Age: " **<<** age **<<** endl**;**

**}**

**};**

***All access modifiers are accessible within a class or struct***

1. In function like main() function after creating an instance of class of struct like person1.name = “void”, and person1.display();the following example:

int main**()** **{**

// Creating an object of the Person struct

Person person1**(**"John Doe"**,** 30**);**

// Displaying the details of person1

person1**.**name = “void”;

person1**.**display**();**

**return** 0**;**

**}**

***If age and name are private or protected, they wont be called outside the class or struct***

1. Last call or access of attributes and method (member function) is used in inheritance like public in line 12 the inheritance example:
2. // Base struct
3. struct Employee **{**
4. std**::**string name**;**
5. int age**;**
6. // Constructor for Employee
7. Employee**(**const std**::**string**&** n**,** int a**)** **:** name**(**n**),** age**(**a**)** **{}**
8. **};**
9. // Derived struct for Writer
10. struct Writer **:** public Employee **{**
11. // Constructor for Writer
12. Writer**(**const std**::**string**&** name**,** int age**)** **:** Employee**(**name**,** age**)** **{}**
13. // Specific method for Writer
14. void writing**()** const **{**
15. std**::**cout **<<** name **<<** " is writing a document." **<<** std**::**endl**;**
16. **}**

**};**

A screenshot of a computer

Description automatically generatedNote: the line struct Writer **:** public Employee is public inheritance see Figure 10 public, protected, private inheritance, members are attributes and methods

Figure 10 public, protected, private inheritance

The following table in Figure 11 Access Modifiers introduce how access modifiers work

Figure 11 Access Modifiers

For now we introduced only structs in user-defined data types, also we have union and enums

### Enum

Enum is abbreviation of enumeration, which used to give some related integers names as humans don’t remember and work with number well, e.g. if a worker get 500$ on Sunday and 600$ on Monday and 700$ on Tuesday ….. . an enum could hold these number and when we want give the worker 500$ on Monday, we could use Monday instead of using 500 number

**Example:** enum

Write C++ enum that define workday wage for a worker,

Sunday = 500, Monday = 600, Tuesday = 700, Wednesday = 800,

Thursday = 900, Friday = 1000, Saturday = 1100

#include<iostream>

**using** **namespace** std**;**

enum days**{**

Sunday **=** 500**,**

Monday **=** 600**,**

Tuesday **=** 700**,**

Wednesday **=** 800**,**

Thursday **=** 900**,**

Friday **=** 1000**,**

Saturday **=** 1100

**};**

int main**(){**

days workDay**;**

cout**<<**"Worker earned: "**<<**Sunday**<<**"$ wage"**<<**endl**;**

cout**<<**"Worker earned: "**<<**Monday**<<**"$ wage"**<<**endl**;**

cout**<<**"Worker earned: "**<<**Tuesday**<<**"$ wage"**<<**endl**;**

cout**<<**"Worker earned: "**<<**Wednesday**<<**"$ wage"**<<**endl**;**

cout**<<**"Worker earned: "**<<**Thursday**<<**"$ wage"**<<**endl**;**

cout**<<**"Worker earned: "**<<**Friday**<<**"$ wage"**<<**endl**;**

cout**<<**"Worker earned: "**<<**Saturday**<<**"$ wage"**<<**endl**;**

**}**

### Union

Union is user-defined data type that all attributes of that union share the same memory see Figure 12 Union vs struct, if I changed n in union; m will be changed too



Figure 12 Union vs struct

**Example:** union

Write C++ union that holds char x=1 and short y=65535 , show the size of the that union and change value of x to 2 and print y and values

#include<iostream>

**using** **namespace** std**;**

union storage**{**

unsigned char x**;**

unsigned short y**;**

**};**

int main**(){**

storage var**;**

var**.**x **=** 1**;**

var**.**y **=** 65535**;**

cout**<<**"size of var is: "**<<sizeof(**var**)<<**endl**;**

cout**<<**"x y resp: "**<<(**unsigned short**)**var**.**x**<<**" "**<<**var**.**y**<<**endl**;**

var**.**x **=** 2**;**

cout**<<**"x y resp: "**<<(**unsigned short**)**var**.**x**<<**" "**<<**var**.**y**<<**endl**;**

**}**

You can see the output in Figure 13 union example, x is unsigned char that holds 1 byte, while y is unsigned short that holds 2 bytes, the first byte is shared by x and y

Like in Figure 14 union example explanation

A black screen with white text

Description automatically generated

Figure 13 union example

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
|  |  |  |  |  |  |  |  | X = 255 | | | | | | | |
| Y = 65535 | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  | when x changed to 2, y is affected as they have 1 byte shared | | | | | | | | |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2nd byte | | | | | | | | 1st byte | | | | | | | |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
|  |  |  |  |  |  |  |  | X = 2 | | | | | | | |
| Y = 65282 | | | | | | | | | | | | | | | |

Figure 14 union example explanation

**Bitfield**

Bitfield is used in struct and union to specify bit values, e.g. if we have an 8bit register that we want to change every bit, we could do that.

**Example:** bitfield

Write a bitfield to mimic an 8bit register by union

#include<stdio.h>

**using** **namespace** std**;**

union Reg**{**

struct**{**

unsigned char B0**:**1**;**

unsigned char B1**:**1**;**

unsigned char B2**:**1**;**

unsigned char B3**:**1**;**

unsigned char B4**:**1**;**

unsigned char B5**:**1**;**

unsigned char B6**:**1**;**

unsigned char B7**:**1**;**

**}**Bits**;**

unsigned char byte**;**

**};**

int main**(){**

Reg DDRA**;**

DDRA**.**Bits**.**B0**=**1**;**

DDRA**.**Bits**.**B1**=**1**;**

DDRA**.**Bits**.**B2**=**1**;**

DDRA**.**Bits**.**B3**=**0**;**

DDRA**.**Bits**.**B4**=**0**;**

DDRA**.**Bits**.**B5**=**0**;**

DDRA**.**Bits**.**B6**=**0**;**

DDRA**.**Bits**.**B7**=**0**;**

printf**(**"%d"**,**DDRA**.**byte**);**

**}**

NOTE: in this example, printf must be used instead of cout, so we have to include stdio.h library

## Operators and Expressions

* Arithmetic operators: +, -, \*, /, %
* Relational operators: ==, !=, >, <, >=, <=
* Logical operators: &&, ||, !
* Bitwise operators: &, |, ^, ~, <<, >>
* Assignment operators: =, +=, -=, \*=, /=, %=, &=, |=, ^=, <<=, >>=
* unary operators (Increment and decrement): ++, --
* ternary operator: ?:

***#Let var1 = 4 and var2 = 3***

### Arithmetic operators: +, -, \*, /, %

Addition (+) e.g. var1 + var 2 = 4 + 3 =7

Subtraction (-) e.g. var1 - var 2 = 4-3=1

Multiplication (\*) var1 \* var 2 = 4\*3=12

Division (/) e.g. var1 / var 2 =4/3 = 1

Modulo or reminder (%) e.g. var1 % var 2 4%3 = 1

see Figure 15 Division and modulo

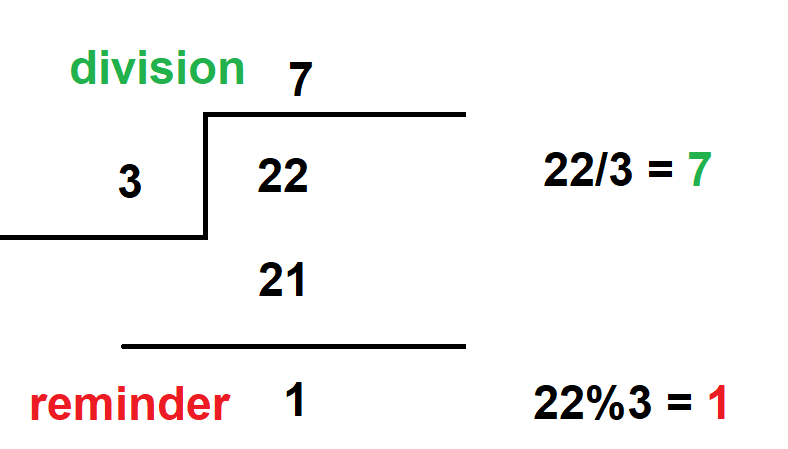


Figure 15 Division and modulo

### Relational operators: ==, !=, >, <, >=

These operators used to determine relational between variables i.e. make comparisons as follows:

Is var1 equal var2 : var1 == var2 (return false as 4 not equal 3)

Is var1 not equal var2 : var1 != var2 (return true as 4 not equal 3)

Is var1 bigger than var2 : var1 > var2 (return true as 4 bigger than 3)

Is var1 less than var2 : var1 < var2 (return false as 4 bigger than 3)

Is var1 bigger than or equal var2 : var1 >= var2 (return true as 4 bigger than 3)

### Logical operators: &&, ||, !

* && means **AND**
* || means **OR**
* !means **NOT**

**Example:** logicalOp

Whatif we want to combine 2 conditions?

The var1 is bigger than var2 **AND** var1 is odd:

The var1 is bigger than var2 **OR** var1 is odd:

The var1 is bigger than var2 **AND** var1 is not odd:

#include<iostream>

**using** **namespace** std**;**

int main**(){**

int var1 **=** 5**,** var2 **=** 6**;**

cout**<<**"The var1 is bigger than var2 AND var1 is odd: "**<<((**var2**>**var1**)&&(**var1**%**2 **==** 0**))<<**endl**;**

cout**<<**"The var1 is bigger than var2 OR var1 is odd: "**<<((**var2**>**var1**)||(**var1**%**2 **==** 0**))<<**endl**;**

cout**<<**"The var1 is bigger than var2 AND var1 is not odd: "**<<((**var2**>**var1**)** **&&** **(**var1**%**2 **!=** 0**))<<**endl**;**

**return** 0**;**

**}**

NOTE: false means 0 and true is anything except 0, the previous code should outputs: 0 1 1 (i.e. false true true)

### Bitwise operators: &, |, ^, ~, <<, >>

bitwise operators, used to change variable in bit level, if you have a 1-byte unsigned char of example, you can do operation on all these 8-bits freely like the following in the Figure 16 Bitwise operations :

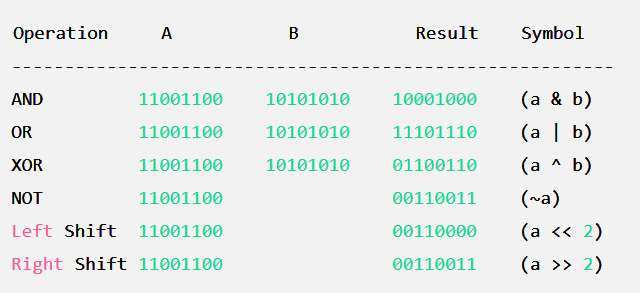


Figure 16 Bitwise operations

### Assignment operators: =, +=, -=, \*=, /=, %=, &=, |=, ^=, <<=, >>=

These operators used to assign variables, e.g.

var1 = 2 (set var1 to 2)

var1+=2 (means var1 =var1+2 which increment var1 by 2)

var1&=1 (means var1 = var1 & 1)

The following figure contains table of what are set, clr, tog, get bit

| **Operation** | **Original** | **Index** | **Result** | **Explanation** |
| --- | --- | --- | --- | --- |
| Set Bit | 11001100 | 2 | 11001100 | Bit at index 2 is already 1, no change. |
| Set Bit | 11001100 | 1 | 11001110 | Bit at index 1 is set to 1. |
| Clear Bit | 11001100 | 2 | 11001000 | Bit at index 2 is cleared to 0. |
| Clear Bit | 11001100 | 3 | 11000100 | Bit at index 3 is already 0, no change. |
| Toggle Bit | 11001100 | 2 | 11001000 | Bit at index 2 is toggled to 0. |
| Toggle Bit | 11001100 | 3 | 11000100 | Bit at index 3 is toggled to 0. |
| Get Bit (original) | 11001100 | 2 | 1 | Bit at index 2 is 1. |
| Get Bit (original) | 11001100 | 1 | 0 | Bit at index 1 is 0. |

Set bit: byte |= (1<<index) (Oring)

Clear bit: byte &=~ (1<<index) (Anding the complement )

Toggle bit: byte ^= (1<<index) (Xoring)

Get Bit: (byte>>index) & 1

**Example:** bitManipulation

In this example we want to make a struct named bit math that has 1 variable and 4 methods setBit(var, bit) clrBit(var, bit) togBit(var, bit) and getBit(var, bit)

NOTE: recall bitfield example and add the method mentioned: setBit() clrBit() ) togBit(var, bit) and getBit(var, bit)

#include<stdio.h>

**using** **namespace** std**;**

struct Register**{**

union Reg**{**

struct**{**

unsigned char B0**:**1**;**

unsigned char B1**:**1**;**

unsigned char B2**:**1**;**

unsigned char B3**:**1**;**

unsigned char B4**:**1**;**

unsigned char B5**:**1**;**

unsigned char B6**:**1**;**

unsigned char B7**:**1**;**

**}**Bits**;**

unsigned char byte**;**

**}**reg**;**

void setBit**(**int index**){**

reg**.**byte **|=** **(**1**<<**index**);**

**}**

void clrBit**(**int index**){**

reg**.**byte **&=~** **(**1**<<**index**);**

**}**

void togBit**(**int index**){**

reg**.**byte **^=** **(**1**<<**index**);**

**}**

int getBit**(**int index**){**

**return** **(**reg**.**byte**>>**index**)** **&** 1**;**

**}**

**};**

int main**(){**

Register DDRA**;**

DDRA**.**reg**.**byte **=** 0**;**

DDRA**.**setBit**(**0**);** //setting bit number 0

DDRA**.**setBit**(**1**);** //setting bit number 0

printf**(**"bit number 0 is %d\n"**,**DDRA**.**getBit**(**0**));**

printf**(**"bit number 1 is %d\n"**,**DDRA**.**getBit**(**1**));**

printf**(**"bit number 2 is %d\n"**,**DDRA**.**getBit**(**2**));**

**}**

NOTE: this is pretty hard solution, but fell easier way.

### unary operators (Increment and decrement): ++, --

increment and decrement is used on one operand (unary)

* post decrement/ increment

var1 = 5; cout<< var1++; //outputs 6

* pre decrement/increment

var1 = 5; cout<< ++var1; //outputs 5 but var1 after cout becomes 6

### ternary operator: ?:

Ternary operator is type of conditionals in C++

Syntax: (condition) what to do if true: what to do if false

e.g.

var1=5;

(var1%2==0) cout<<”even” :cout<<”odd”; // the output is “even”

## Control Structures

* Conditional statements: if, if-else, nested if, switch-case
* Looping statements: for, for range, while, do-while
* Jump statements: break, continue, goto, return

### Conditional statements: if, if-else, switch-cases

Program is set of instructions to perform a task, some instructions require certain conditions to be performed, e.g. if(day == Friday) give all workers weekend wage weekend Wage()

There are 2 types of conditionals: if, else if, else AND switch

**Switch case:**

Switch is used to check a variable

Syntax:

switch(variable){

case 1: instructions; break;

case 2: instructions; break;

case 3: instructions; break;

.

.

default: instructions;break

}

NOTE: case 1: means if variable == 1

case ‘a’: means if variable == ‘a’

NOTE: default is used when variable has value not include in cases

NOTE: don’t forgot to put break after any condition

NOTE: don’t make a variable case e.g. case var

**if, else if, else case:**

if else used when you want to check for conditions

Syntax:

if(condition){ instructions}

else if(condition){ instructions}

else { instructions}

NOTE: else if is not consider unless if conditional is not fulfilled

NOTE: else is not consider unless if conditional and else if conditionals are not fulfilled

NOTE: don’t make instructions between if- else if – else

e.g. if(x==5){ cout<<”5”’}

cin>>x; //wrong !!

else {cout<<” x is not 5;}

**Example**: switch

Write C++ code to determine whether the letter is vowel or not by using switch case, Vowels are: a, e, i, o, u . Consonants are the rest of the letters .

#include<iostream>

**using** **namespace** std**;**

int main**(){**

char x **=** ' '**;**

cout**<<**"enter a letter: "**;**

cin**>>**x**;**

**switch(**x**){**

**case** 'a'**:** cout**<<**"\n the letter "**<<**x**<<**" is Vowel\n"**;break;**

**case** 'e'**:** cout**<<**"\n the letter "**<<**x**<<**" is Vowel\n"**;break;**

**case** 'i'**:** cout**<<**"\n the letter "**<<**x**<<**" is Vowel\n"**;break;**

**case** 'u'**:** cout**<<**"\n the letter "**<<**x**<<**" is Vowel\n"**;break;**

**case** 'o'**:** cout**<<**"\n the letter "**<<**x**<<**" is Vowel\n"**;break;**

**default:** cout**<<**"\n the letter "**<<**x**<<**" is Consonant\n"**;** //break at last condition doesn’t matter

**}**

**}**

**Example**: ifElse

In switch example, if 5 is entered, the output is: the letter 5 is Consonants, as its in default case, but 5 is not letter, complete the previous code to check first if the input is letter

Hint: isalpha() use this to determine if the input is letter or not

#include<iostream>

**using** **namespace** std**;**

int main**(){**

char x **=** ' '**;**

cout**<<**"enter a letter: "**;**

cin**>>**x**;**

**if(**isalpha**(**x**)){**

**switch(**x**){**

**case** 'a'**:** cout**<<**"\n the letter "**<<**x**<<**" is Vowel\n"**;break;**

**case** 'e'**:** cout**<<**"\n the letter "**<<**x**<<**" is Vowel\n"**;break;**

**case** 'i'**:** cout**<<**"\n the letter "**<<**x**<<**" is Vowel\n"**;break;**

**case** 'u'**:** cout**<<**"\n the letter "**<<**x**<<**" is Vowel\n"**;break;**

**case** 'o'**:** cout**<<**"\n the letter "**<<**x**<<**" is Vowel\n"**;break;**

**default:** cout**<<**"\n the letter "**<<**x**<<**" is Consonant\n"**;** //break at last condition doesn’t matter

**}**

**}**

**else{**

cout**<<**'\n'**<<**x**<<**" is not letter"**<<**endl**;**

**}**

**}**

### Loops

What if we need to execute certain code many times? e.g. printing “hello” 100 time or until user enters quit

* We could type cout<<”hello” 100 times
* OR we could use loops

Loop is C++ are:

* for(start; end ;update){instructions}
* for (range){instructions}
* while(condition){instructions}
* do{instructions} while(condition)

**for loop**

**syntax:** for(start; end ;update){instructions}

e.g.

for(int itr=0;itr<10;itr++){

cout<<”hello ”<<itr<<” times<<endl;

}

Used when number of iterations is known, the previous example demonstrates printing hello itr times when such that itr starts with 0 and ends when itr = 9 (itr<10), and the update is how does the variable itr changers, in this case the update is itr is increased by 1 (i.e. itr++ means itr=itr+1 )

**Example**: forLoop

Write C++ code to print even numbers from 10 to 20

#include <iostream>

**using** **namespace** std**;**

int main**(){**

**for(**int k**=**10**;** k**<=**20**;** k**=**k**+**2 **){**

cout**<<**"the number "**<<**k**<<**" is even\n"**;**

**}**

**return** 0**;**

**}**

**for range loop**

**syntax:** for(datatype item: list){instructions}

this is used to get the item of list (array or vector) without subscript Operator (i.e []), like python for loop

**Example**: forRangeLoop

print array of vowels without using subscriptor operator

#include <iostream>

#include<array>

**using** **namespace** std**;**

int main**(){**

array**<**char**,**5**>** vowels **=** **{**'a'**,**'e'**,**'u'**,**'i'**,**'o'**};**

//remember array<,> differs from c arrays (vowels[])

**for(**char x**:** vowels**)**

**{**cout**<<**x**<<**" is vowel"**<<**endl**;}**

**}**

**While loop**

**syntax:** while(condition){instructions}

while loop is used when number of iteration is unknown but the condition is clear

**Example**: whileLoop

Write C++ code to calculate the sum of user single integer input, e.g. if user entered 1251 the sum is 1+2+5+1 which is 9

NOTE: the algorithm is take the reminder and divide the number by 10

#include <iostream>

**using** **namespace** std**;**

int main**(){**

int x **=**0**,**sum**=**0**,** cont**=**0**;**

cout**<<**"enter a number "**;**

cin**>>**x**;**

cont **=** x**;**

**while(**x**/**10 **>** 0**){**

sum **+=** x**%**10**;**

x**=** x**/**10**;**

**}**

sum **+=** x**;** //adding the most left number

cout**<<**"the sum of "**<<**cont**<<**" is: "**<<** sum**;**

**}**

**Do While loop**

**syntax:** do{instructions} while(condition);

same as while loop but the instructions are done first, then check on condition, remember the whileLoop (previous example), we had to write the following line

sum **+=** x**;** //adding the most left number

as the condition is reaching before getting the most left number

I.e.

Sum=0 and x=123

Reminder and divide first time Sum=3 and x=12 (**(**x**/**10 **>** 0**)** check is valid)

Reminder and divide second time Sum=6 and x=1 (**(**x**/**10 **>** 0**)** check isn’t valid)

As 1/10 not bigger that 0 so number 1 (most left number of 123) is not added,

**Example**: doWhileLoop

Rewrite the whileLoop by by dowhile loop instead of while loop

#include <iostream>

**using** **namespace** std**;**

int main**(){**

int x **=**0**,**sum**=**0**,** cont**=**0**;**

cout**<<**"enter a number "**;**

cin**>>**x**;**

cont **=** x**;**

**do{**

sum **+=** x**%**10**;**

x**=** x**/**10**;**

**}**

**while(**x **>** 0**);**

cout**<<**"the sum of "**<<**cont**<<**" is: "**<<** sum**;**

**}**

### Jump statements: break, continue, goto, return

**break** is used in loops to get out of the loop

**continue** is used to skip an iteration in the loop

**goto** is used to jump to any line in the code

**return** is used in functions to get out the function

**Example**: breakContinue

In this example, break and continue are used to illustrate the difference, two for loops will be written, break will be used in one loop and continue in the other when the itr is equal 6, the break gets out when itr gets 6 but the continue, skips the 6 and continue the loop, see Figure 17 break and continue

#include <iostream>

**using** **namespace** std**;**

int main**(){**

**for(**int itr**=**0**;** itr**<=**10**;** itr**++){**

**if(**itr**==**6**)** {**continue;}**

**else{**cout**<<**"continue loop: "**<<**itr**<<**endl**;}**

**}**

**for(**int itr**=**0**;** itr**<=**10**;** itr**++){**

**if(**itr**==**6**)**{**break;}**

**else{**cout**<<**"break loop: "**<<**itr**<<**endl**;}**

**}**

**}**

A screen shot of a computer

Description automatically generated

Figure 17 break and continue

**Example**: goto

Print even number from 30 to 40 without using loops and use only one cout

#include <iostream>

**using** **namespace** std**;**

int main**(){**

int itr **=** 30**;**

a**:**

**if(**itr**%**2 **==**0**){**

cout**<<**"the number "**<<**itr**<<**" is even\n"**;**

**}**

itr**++;**

**if(**itr**<=**40**)**

**goto** a**;**

**}**

## Final project

Project Requirements: Sign-Up Application

We are developing a user registration application to store user names and ages.

We will use a struct to represent each user, stored in an array (limited to 100 records).

Users can add records and retrieve them by ID.

Steps to Complete:

1-Include necessary headers.

2-Define a struct for user records (Person), and declare an array of this type (Person records[100]).

3-Implement functions:

A-void AddRecord(const std::string& name, int age): Adds a new record.

B-FetchRecord(int id): Retrieves a record by ID.

C-Quit().

4-In main(), use a loop to present options (Add Record, Fetch Record, Quit).

Handle user input using a switch statement:

Case 1: Prompt for name and age, then call AddRecord().

Case 2: Prompt for ID, then call FetchRecord() and display the result.

Case 3: Exit the loop.

You can fine code in the github repository [LINK](https://github.com/MuOssama/MasteringCPP/blob/main/Basics/FinalProject/finalProject.cpp)

An example: see Figure 18 Final project snapshoot

A screenshot of a computer program

Description automatically generated

Figure 18 Final project snapshoot

# Pointers and Memory management

One of strengths of C++ is access hardware directly specially memory, pointers is derived data type that can modify a variable (e.g. single variable or list (array or vector or even structs), you can for sure modify a variable like what we did in last chapters, but in functions it isn’t applicable (next chapter).

Pointers carry the memory address of variable (e.g. single variable or list (array or vector or even structs), so that you can modify this variable, pointer like a key for accessing a flat , to enter the flat you must have a key, the pointer carry the address of the variable in memory so it can gets in and change the variable

Another benefit from using pointers is to use it to allocate a place in memory for a variable (e.g. single variable or list (array or vector)), and the pointer is the key to access the element(s) of that variable

So in this chapter and the following one, the main two benefits of pointers will be introduced which are pointer to allocate variables and pointer in functions (call by reference) (in the next chapter)

* **Introduction to Pointers**
  + Pointer declaration, initialization, and dereferencing
  + Operations on Pointers
  + References and reference variables
* **Dynamic Memory Allocation**
  + new and delete operators
  + Allocating memory for single variables and arrays
  + Linked List
  + Memory leaks and how to avoid them
* **Smart Pointers**
  + std::unique\_ptr
  + std::shared\_ptr
  + std::weak\_ptr

## Introduction to Pointers

A diagram of a memory

Description automatically generatedMemory store variables in memory in bytes, its compiler dependent but in our gcc compiler char has 1 byte, short has 2 bytes, int and floats have 4 bytes, double has 8 bytes, Figure 19 how memory store variables, x is store in one byte in memory address 2000 and y also but stored in 2001 memory address, and short store 2 bytes which are 2002 and 2003 as short requires 2 bytes

#include<iostream>

**using** **namespace** std**;**

int main**(){**

char x**;**

char y**;**

short z**;**

**}**

Figure 19 how memory store variables

Pointers must be as same type of what it points to (so it will be same size also) !!

NOTE: in C++ pointers forces not to change a const variable, but you can change the const variable in C

**Example**: ptrVar

Write C++ code to pointer to char and other to int, print the values and addresses of variables and size of the two pointers to these variables

#include<iostream>

**using** **namespace** std**;**

int main**(){**

char x **=** 'a'**;**

short y **=** 15**;**

char**\*** ptr\_x **=** **&**x**;**

short**\*** ptr\_y **=** **&**y**;**

//derefrence operator \* to access or modifiy variables

cout**<<**"x has: "**<<\***ptr\_x**<<**" while y has: "**<<\***ptr\_y**<<**endl**;**

cout**<<**"x is stored in : "**<<(**void**\*)**ptr\_x**<<**" whil

e y is stored in: "**<<**ptr\_y**<<**" Addresses"**<<**endl**;**

**\***ptr\_x **=** 'b'**;**

**\***ptr\_y **=** 66**;;**

cout**<<**"After Modifiynig\nx has: "**<<\***ptr\_x**<<**" while y has: "**<<\***ptr\_y**<<**endl**;**

cout**<<**"x is stored in : "**<<(**void**\*)**ptr\_x**<<**" while y is stored in: "**<<**ptr\_y**<<**" Addresses"**<<**endl**;**

**}**

### Pointer definition

A table with a number and a memory

Description automatically generated with medium confidencethe array is simply a block of memory that holds some consecutive data of same type, when an array is created, \*e.g. array ***named x***), x is a pointer to first element of the array (x is the memory address of first element), so to access any element you have to use x pointer to access any element by x[i] or \*(x+i), remember that x pointer to first address (carry address of 1st element) let the address of first element is 2000 so x is 2000 and to access the 2nd element you have to use \*(x+1) which is \*(2001) and 3rd element is \*(x+2) which is \*(2002) so the general to access any element use \*(x+i) which is also x[i] see Figure 20 how memory store arrays

Figure 20 how memory store arrays

**Example**: ptrArray

Write array with 5 elements and get them from user and print them, don’t use subscriptor operator (i.e. arr[]) use \*(ptr+i) which means \*(pointer\_to\_)

#include<iostream>

#include<array>

**using** **namespace** std**;**

int main**(){**

array**<**int**,**5**>** x**;**

int**\*** ptr\_x **=** x**.**data**();** //x.data() is used to get the pointer of array x

//filling array

**for(**int i**=**0**;**i**<**x**.**size**();**i**++){**

int x**=**0**;**

cout**<<**"\nenter element: "**<<**i**<<**" "**;**

cin**>>\*(**ptr\_x **+** i**);** //\*(ptr\_x + i) == ptr[i]

**}**

//printing array

**for(**int i**=**0**;**i**<**x**.**size**();**i**++){**

cout**<<**"\nelement "**<<**i**<<**" is"**<<\*(**ptr\_x **+** i**);** //\*(ptr\_x + i) == ptr[i]

**}**

**}**

If you used C array (i.e. char x[5]) instead of stl array (array<char,5>x;)

#include<iostream>

#include<array>

**using** **namespace** std**;**

int main**(){**

int x**[**5**];**

int**\*** ptr\_x **=** x**;** //&x is not used as x is address itself

//filling array

**for(**int i**=**0**;**i**<**5**;**i**++){**

int x**=**0**;**

cout**<<**"\nenter element: "**<<**i**<<**" "**;**

cin**>>**x**;**

**\*(**ptr\_x **+** i**)** **=** x**;** //\*(ptr\_x + i) == ptr[i]

**}**

//printing array

**for(**int i**=**0**;**i**<**5**;**i**++){**

cout**<<**"\nelement "**<<**i**<<**" is "**<<(\*(**ptr\_x **+** i**));** //\*(ptr\_x + i) == ptr[i]

**}**

**}**

### Operations on Pointers

Arithmetic operators

+ and – but not /, \*, %

You can add and subtract values from pointers like what we did in array \*(arr+i) pointer arr (first element of the array) could be added or subtracted to iterate over an array

**Question**: what happens when pointer to array of 3 int is incremented ?? see Figure 21 pointer increment (remember : int has 4 bytes in gcc compiler

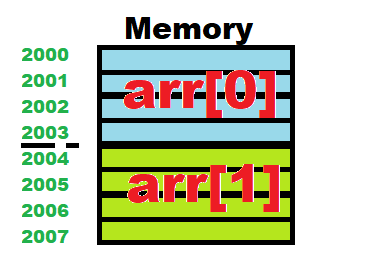


Figure 21 pointer increment

The pointer incremented by four as size of int is 4 so every increment is 4 memory addresses, if short is used; the increment will be 2-byte step as short in gcc has 2 bytes see Figure 22 pointer increment step

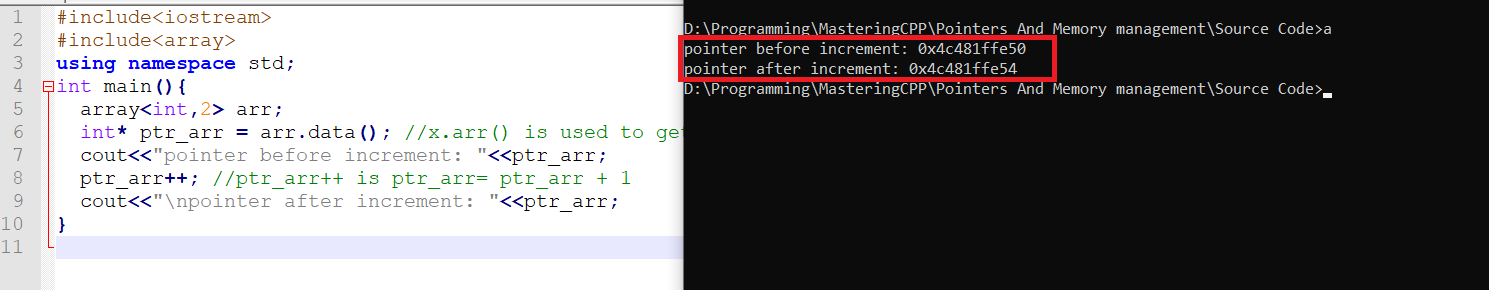


Figure 22 pointer increment step

**Arrow operator**

When you use dot operator (.) and dereference operator (\*) like in pointer to struct you could easily use arrow operator (->) for simplicity

e.g. \*(ptr\_to\_struct).member is equivalent to ptr\_to\_struct->member

don’t forgot: for simplicity also we use arr[i] instead of \*(arr+i)

NOTE:

* when we use pointer to int, the datatype of the pointer should be int
* when we use pointer to char, the datatype of the pointer should be char
* SO, when we use pointer to struct, the datatype of the pointer should be as same as the struct

**Example**: ptrStruct

Use arrow operator to modify age of student struct that have name and age

#include<iostream>

**using** **namespace** std**;**

struct Student**{**

int age**;**

string name**;**

//constructor

Student**(**int a**,** string n**):** age**(**a**),** name**(**n**){}**

**};**

int main**(){**

Student Ahmed**(**25**,**"Ahmed"**);**

//assume age is 26 so we have to modifiy

//we could modifi directly by Ahmed.age = 26

//but in function, it is not applicable

Student**\*** ptr\_struct **=** **&**Ahmed**;**

ptr\_struct**->**age **=** 26**;** //same as \*(ptr\_struct).age = 26

cout**<<**"The age of "**<<**ptr\_struct**->**name**<<**" and have "**<<**ptr\_struct**->**age**<<**" yrs old"**;**

**}**

## Dynamic Memory allocation

Pointers are used to allocate memory for array or single variable, the memory allocation is like in C but mostly we use new and delete instead off malloc and calloc and realloc and free

### new and delete operators

new is used to allocate memory for single variable and array or even structs

### Allocating memory for single variables and arrays

Allocation:

* datatype\* ptr = new datatype //for single variable
* datatype\* ptr = new datatype[num] //for array

e.g. int\* ptr = **new** int;

int\* ptr = **new** int[];

deallocation (deletion):

* delete ptr //for single variable
* delete[num] ptr//for array

e.g. delete ptr;

delete[] ptr;

**Example**: arrayAlloc

Write C++ to allocate 5-element (integers) array using new (don’t use malloc() )

#include<iostream>

**using** **namespace** std**;**

int main**(){**

int size **=** 5**;**

int**\*** ptr\_arr **=** **new** int**[**size**];**

//filling array

**for(**int i**=**0**;**i**<**size**;**i**++){**

cout**<<**"\nenter element :"**<<**i**<<**" "**;**

cin**>>\*(**ptr\_arr**+**i**);**

**}**

//printing array

**for(**int i**=**0**;**i**<**size**;**i**++){**

cout**<<**"\nenter element "**<<**i**<<**" is "**<<** **\*(**ptr\_arr**+**i**);**

**}**

**}**

### Linked List

Array and vector allocate element consecutively in memory, but what happens if we want to allocate 100 element (char) in array and we have these 100 bytes in memory to store these 100 char elements, but we don’t have these free 100 bytes consecutively?? Linked list came to help, in linked list you can store elements of array in different locations in memory ( not consecutively primarily)

In linked list, make a struct to store data and address (pointer) of the next node, linked list primarily is struct carry node data and pointer carry address of the next node see Figure 23 Linked List basics

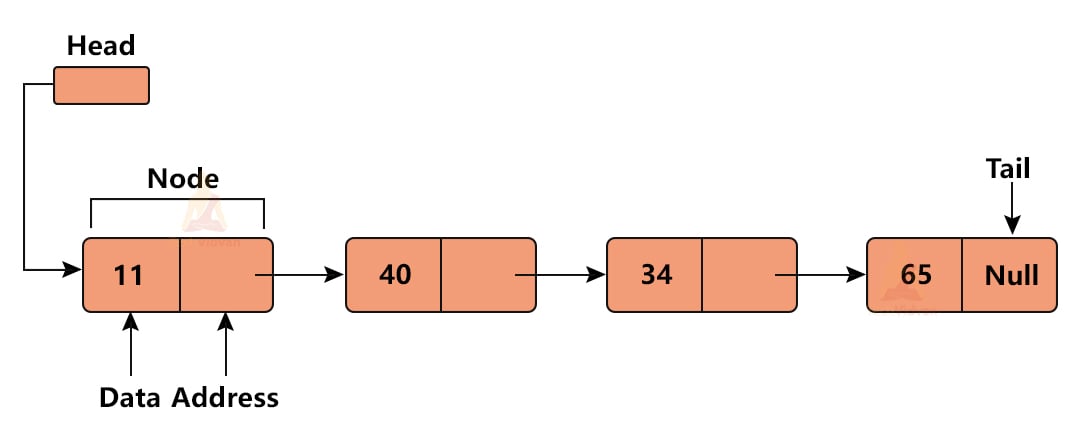


Figure 23 Linked List basics

A qr code on a white background

Description automatically generatedA qr code on a white background

Description automatically generatedBefore getting into linked lists, I am not big fan to illustrate such thing on books and leave graphical illustrations, so open the either YouTube links that illustrate the basics if you are totally newbie to linked list and C++ , the two links one of them in [Arabic](https://www.youtube.com/watch?v=t7WnaypBoNw) and the other in [English](https://www.youtube.com/watch?v=N6dOwBde7-M).

Figure 24 English video

Figure 25 Arabic video

The other thing is to recall pointer to struct:

When you use dot operator (.) and dereference operator (\*) like in pointer to struct you could easily use arrow operator (->) for simplicity

e.g. \*(ptr\_to\_struct).member is equivalent to ptr\_to\_struct->member

don’t forgot: for simplicity also we use arr[i] instead of \*(arr+i)

NOTE:

* when we use pointer to int, the datatype of the pointer should be int
* when we use pointer to char, the datatype of the pointer should be char
* SO, when we use pointer to struct, the datatype of the pointer should be as same as the struct

So after we make the pointer to list, we could access what is in the struct by either ways ptr\_struct->member **or** \*( ptr\_struct).member

**Steps to create linked list**

* Create struct carry data and pointer

struct Node**{**

int data**;**

Node**\*** next**;**

**};**

* Create the head, pointer of that struct that carry is ready for storing the 1st node address

//create the head

Node**\*** head **=** **new** Node**;**

* **To create 1st node** 
  1. Create instance of that struct

//create 1st node

Node**\*** newNode **=** **new** Node**;**

2- Put the data you want and make the next points to NULL

cout**<<**"enter the data of the first node "**;**

cin**>>**newNode**->**data**;**

newNode**->**next **=** **NULL;**

3- The address (ptr\_struct) is stored in head (link the new node to head)

//link the new node to the head

head**->**next **=** newNode**;**

* To create further node
  1. Create instance of that struct

//create 2nd node

Node**\*** newNode2 **=** **new** Node**;**

2- Put the data you want and make the next points to NULL

cin**>>**newNode2**->**data**;**

newNode2**->**next **=** **NULL;**

3- The address (ptr\_struct) is stored in the last node that has null\_ptr (link the new node to the last node

**(**head**->**next**)->**next **=** newNode2**;**

* To delete node

Let the node to be deleted is nth node

* 1. Iterate to n-1th node

//1- iterate to previous node of 2nd node which is 1st node

itr **=** head**;**

**for(**int i**=**0**;**i**<**1**;**i**++){**

itr**=**itr**->**next**;** //we 1st node

**}**

* 1. In n-1th address make the address is n+1th address instead of nth address

//2- (unlink 2nd node) replace 1st node address by the 3rd node instead of 2nd node

Node**\*** temp **=** itr**->**next**;** //save 2nd node before unlinking

itr**->**next **=** **(**itr**->**next**)->**next**;** //pointer to pointer (address of 3rd node)

* 1. Till here, the node is not deleted but unlinked from the list, so we have to delete the node by **delete**

//3-delete the 2nd node

**delete** temp**;**

* Print the list

//printing the linked list

Node**\*** itr **=** head**->**next**;**

**while(**itr **!=** **NULL){**

cout**<<**itr**->**data**<<**endl**;**

itr**=**itr**->**next**;**

**}**

Example: linkedList

Here is to sum up:

#include<iostream>

**using** **namespace** std**;**

struct Node**{**

int data**;**

Node**\*** next**;**

**};**

int main**(){**

//create the head

Node**\*** head **=** **new** Node**;**

//create 1st node

Node**\*** newNode **=** **new** Node**;**

cout**<<**"enter the data of the first node "**;**

cin**>>**newNode**->**data**;**

newNode**->**next **=** **NULL;**

//link the new node to the head

head**->**next **=** newNode**;**

//create 2nd node

Node**\*** newNode2 **=** **new** Node**;**

cout**<<**"enter the data of the second node "**;**

cin**>>**newNode2**->**data**;**

newNode2**->**next **=** **NULL;**

**(**head**->**next**)->**next **=** newNode2**;** //Link the node

//create 3nd node

Node**\*** newNode3 **=** **new** Node**;**

cout**<<**"enter the data of the third node "**;**

cin**>>**newNode3**->**data**;**

newNode3**->**next **=** **NULL;**

**((**head**->**next**)->**next**)->**next **=** newNode3**;** //Link the node

//printing the linked list

Node**\*** itr **=** head**->**next**;**

**while(**itr **!=** **NULL){**

cout**<<**itr**->**data**<<**endl**;**

itr**=**itr**->**next**;**

**}**

//deletion of 2nd node

//1- iterate to previous node of 2nd node which is 1st node

itr **=** head**;**

**for(**int i**=**0**;**i**<**1**;**i**++){**

itr**=**itr**->**next**;** //we 1st node

**}**

//2- (unlink 2nd node) replace 1st node address by the 3rd node instead of 2nd node

Node**\*** temp **=** itr**->**next**;** //save 2nd node before unlinking

itr**->**next **=** **(**itr**->**next**)->**next**;** //pointer to pointer (address of 3rd node)

//3-delete the 2nd node

**delete** temp**;**

//printing the linked list again after deletion

cout**<<**"after deletion of 2nd node"**<<**endl**;**

itr **=** head**->**next**;**

**while(**itr **!=** **NULL){**

cout**<<**itr**->**data**<<**endl**;**

itr**=**itr**->**next**;**

**}**

**}**

### Memory Leaks

Memory leaks happens when manual memory allocation is performed badly such that many allocation with no deletion

**Example**: memLeaks

Write C++ code that allocates array of 100 integers memory and the pointer to that array assign it to nullptr or NULL, the whole code in while(true)

#include<iostream>

**using** **namespace** std**;**

int main**(){**

**while(true){**

int**\*** ptr **=** **new** int**[**100**];**

ptr **=** **nullptr;** //null pointer

**}**

**}**

The previous example is like buying flats and throw away the key of each flat, the issue of throwing the key is you cannot get into the flat and the worse is takes portion of your wealth with no benefits, so the pointer (key) when forgotten (ptr **=** **nullptr )** the array or variable allocated remains in memory and never deleted, this causing software aging which means the program crashes and even worse the whole system. See Figure 26 Software crash (termination) due to bad memory allocation

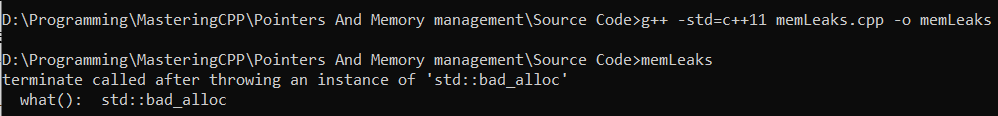


Figure 26 Software crash (termination) due to bad memory allocation

Remember, one of strengths of C++ is controlling hardware (e.g. memory) freely which makes C++ fast, but if this strength used badly, it becomes weakness

NOTE: C/C++ use manual memory management while programming languages like python use garbage collection which is automatic memory management, its little bit slower bit lesser risks

**Pointer types:**

* Wild pointer

when using pointers before assigning value to that pointer

#include<iostream>

**using** **namespace** std**;**

int main**(){**

int**\*** ptr**;**

cout**<<\***ptr**;**

**}**

**NOTE: ptr points to nothing !!**

Defending against wild pointers

Always initialize pointer to nullptr (null pointer) like:

int**\*** ptr = **nullptr;**

* Dandling pointer

When using a pointer that is deallocated

#include<iostream>

**using** **namespace** std**;**

int main**(){**

int**\*** ptr **=** **new** int**[**3**];**

ptr**[**0**]** **=** 5**;**

ptr**[**1**]** **=** 55**;**

ptr**[**2**]** **=** 555**;**

**delete** ptr**;**

cout**<<**ptr**[**1**];** //printing deallocated pointer

**}**

Using the ptr pointer after deletion

REMEMBER:

* Void pointer e.g. (void \*) is generic pointer that could be casted (converted) to point to any data
* nullptr is null pointer that points to nothing
* nullptr is preferred over NULL

## Smart pointers

When memory leaks happens (allocate memory and not deleting them), memory is consumed with no usage as variables are still in heap memory and are not deleted, some developers don’t deallocate memory well, so smart pointers came to help. Smart pointers are pointers that deallocate memory by itself (i.e. there is no need to call delete)

Smart pointers deallocate memory by keep tracking of usage of that pointer, if the pointer is not used, the deallocation done automatically.

Types of smart pointers:

* Unique pointer
* Shared pointer
* Week pointer

### Unique pointer

This type of smart pointer keeps track of user usage of itself, if user don’t use this unique pointer, it deallocates the memory allocated. See Figure 27 unique pointer

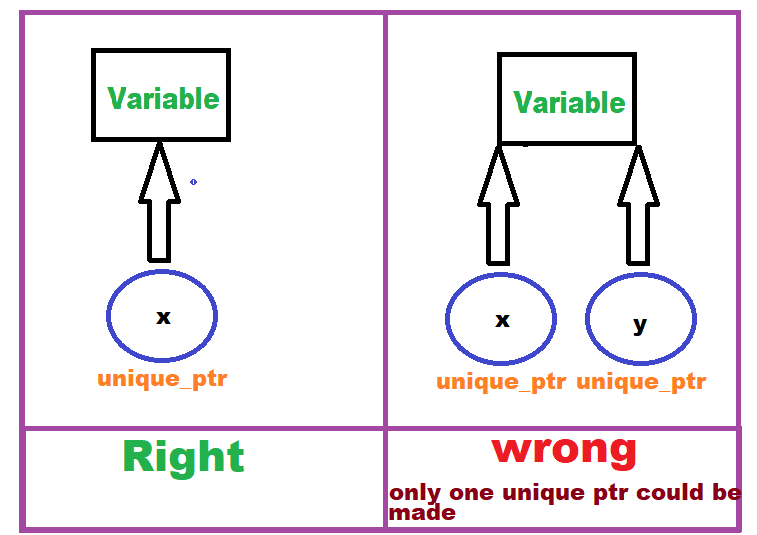


Figure 27 unique pointer

Syntax:

For allocating single variable via unique pointers

unique\_ptr<datatype> name(new datatype)

or

unique\_ptr<datatype> name = make\_unique<datatype>

For allocating array via unique pointers

unique\_ptr <datatype[]> name(new datatype[])

**Example:** allocate memory for single and array of int via unique\_ptr

#include<iostream>

#include<memory>

**using** **namespace** std**;**

int main**(){**

unique\_ptr**<**int**>** ptr1**(new** int**(**15**));**

//make unique ptr to var init with 15

//or use unique\_ptr<int> ptr1 = make\_unique<int>();

unique\_ptr**<**int**[]>** ptrArr**(new** int**[**10**]);**

//allocate array via unique pointer

//filling the array

**for(**int i**=**0**;**i**<**10**;**i**++){**

cout**<<**"enter element "**<<**i**<<**" "**<<**endl**;**

cin**>>\*(**ptrArr**.**get**()+**i**);** //or ptrArr[i]

//note to get the address of unique pointer, use .get() method

**}**

//printing the single value

cout**<<**"unique single var is :"**<<\***ptr1**<<**endl**;**

//printing the array

**for(**int i**=**0**;**i**<**10**;**i**++){**

cout**<<**"unique array var is, element number : "**<<**i**<<**" "**<<**ptrArr**[**i**]<<**endl**;**

**}**

**}**

NOTE: to get the address of unique pointer, use .get() method

### Shared pointer

this type of smart pointer is totally like unique pointer but could have many shortcut pointers so that deallocation is done if and only if user stopped using all the shortcuts and the main pointers itself. See Figure 28 shared pointer

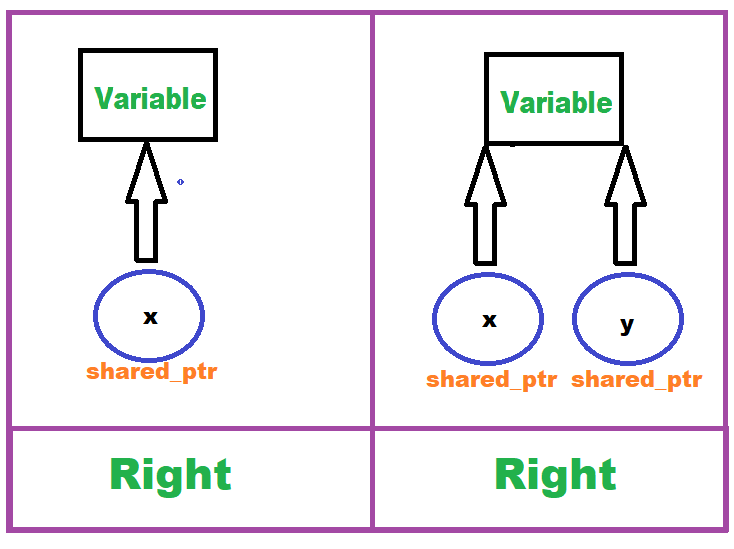


Figure 28 shared pointer

**Example:** allocate memory for single via 2 shared\_ptr and use .use\_count() method to show how many shared\_ptr are used, should be 2 as we created 2 shared pointers.

#include<iostream>

#include<memory>

**using** **namespace** std**;**

int main**(){**

shared\_ptr**<**int**>** ptr1**(new** int**(**15**));** //make unique ptr to var init with 15

//or use unique\_ptr<int> ptr1 = make\_shared<int>();

shared\_ptr**<**int**>** ptr2 **=** ptr1**;**

cout**<<**"first shared ptr to single var is :"**<<\***ptr1**<<**endl**;**

cout**<<**"second shared ptr to single var is :"**<<\***ptr1**<<**endl**;**

cout**<<**"there are "**<<**ptr1**.**use\_count**()<<**" shared pointer"**;**

**}**

### Weak pointer

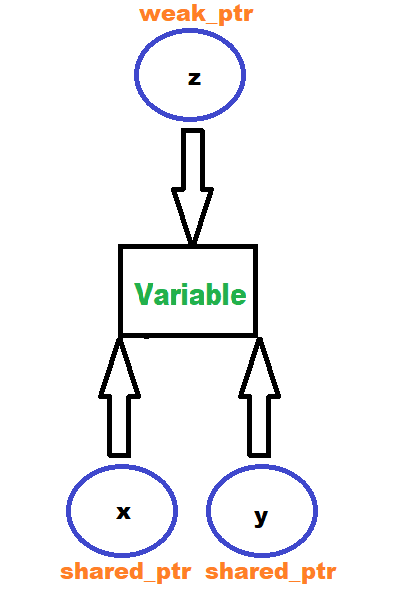
This type of smart pointer is like shared but the deallocation is done by only stopping usage the main pointers (shared pointers i.e. the main and its shortcuts). See Figure 29 Weak pointers, NOTE: you cannot dereference with weak pointers, you should create a var of shared\_ptr type and use .lock() i.e. var = wearPointer.lock()

Figure 29 Weak pointers

**Example:** allocate memory for single via 2 shared\_ptr and and 1 weak pointer then use .use\_count() method to show how many pointers are used (active), should be 2 as we created 2 shared pointers as weak pointer doesn’t count

#include<iostream>

#include<memory>

**using** **namespace** std**;**

int main**(){**

shared\_ptr**<**int**>** ptr1**(new** int**(**15**));** //make unique ptr to var init with 15

//or use unique\_ptr<int> ptr1 = make\_shared<int>();

shared\_ptr**<**int**>** ptr2 **=** ptr1**;**

weak\_ptr**<**int**>** ptr3 **=** ptr1**;** //weak pointer is created

//you cannot derefrence with weak ptr directly

//so do this:

//shared\_ptr<int> tempPtr3 = ptr3.lock();

cout**<<**"first shared ptr to single var is :"**<<\***ptr1**<<**endl**;**

cout**<<**"second shared ptr to single var is :"**<<\***ptr2**<<**endl**;**

//cout<<"third ptr (weak) to single var is :"<<\*tempPtr3<<endl;

cout**<<**"there are "**<<**ptr1**.**use\_count**()<<**" pointer"**; //the answer is 2**

**}**

NOTE: if we uncomment //shared\_ptr<int> tempPtr3 = ptr3.lock();

The answer will be 3 despite the fact of weak pointers don’t count as tempPtr3 is shared\_ptr type !!

# Functions

Function is set of instructions used to decrease program size, whenever a function called, it performs the instructions it has.

* **Function Declaration and Definition**
  + Syntax: return\_type function\_name(parameters)
  + Function prototypes
* **Parameter Passing**
  + Pass-by-value
  + Pass-by-reference
  + Pass-by-pointer
  + Default arguments
* **Overloading and Inline Functions**
  + Function overloading
  + Inline functions
* **Recursive Functions**
  + Base case and recursive case
  + Examples: factorial, Fibonacci sequence

e.g. you have 4 users to print their names, ID, age which is 12 lines of code (i.e. 3 lines to print for 4 users)

#include<iostream>

**using** **namespace** std**;**

int main**(){**

//print first user

cout**<<**"hello "**<<**"Ahmed"**<<**endl**;**

cout**<<**"you've "**<<**20**<<**" yrs old"**<<**endl**;**

cout**<<**"your ID is "**<<**202014**<<**endl**;**

//print second user

cout**<<**"hello "**<<**"Gamal"**<<**endl**;**

cout**<<**"you've "**<<**25**<<**" yrs old"**<<**endl**;**

cout**<<**"your ID is "**<<**202015**<<**endl**;**

//print third user

cout**<<**"hello "**<<**"Sameh"**<<**endl**;**

cout**<<**"you've "**<<**18**<<**" yrs old"**<<**endl**;**

cout**<<**"your ID is "**<<**202016**<<**endl**;**

//print fourth user

cout**<<**"hello "**<<**"Emad"**<<**endl**;**

cout**<<**"you've "**<<**23**<<**" yrs old"**<<**endl**;**

cout**<<**"your ID is "**<<**202017**<<**endl**;**

**}**

Its headache to do this for only 4 users, what if 100 users !!

Function could save the instructions and call it whenever you want with only one line !

#include<iostream>

**using** **namespace** std**;**

void printUser**(**string name**,** int age**,** short ID**);** //function Declaration

int main**(){**

printUser**(**"Ahmed"**,** 20**,** 202014**);**

printUser**(**"Gamal"**,** 25**,** 202015**);**

printUser**(**"Sameh"**,** 18**,** 202016**);**

printUser**(**"Emad"**,** 23**,** 202017**);**

**}**

void printUser**(**string name**,** int age**,** short ID**){**//function Definition

cout**<<**"hello "**<<**name**<<**endl**;**

cout**<<**"you've "**<<**age**<<**" yrs old"**<<**endl**;**

cout**<<**"your ID is "**<<**ID**<<**endl**;**

**}**

SEE the code decreased a lot

## Function Declaration and Definition

### Function declaration

Is telling the compiler what your return type and input parameters and name of your function

i.e.

**return\_datatype** name(datatype param1, datatype param2, datatype param3 ,.);

e.g.

int add (int x, int y);

NOTE: **return\_datatype** means the output of your function will be in what type, void mean the function don’t return anything, int means it return in

The add function takes 2 input integers and return sum as output which is integer too

If the function output (sum):

float the declaration will be **float** add (int x, int y);

its critical to determine the output (return) data types

### Function definition

Is telling the compiler what instructions the function does.

So to sum up, the declaration is:

This is user **declaration** of function takes 3 parameters and return nothing

void printUser**(**string name**,** int age**,** short ID**);** //function Declaration

and the **definition** is:

void printUser**(**string name**,** int age**,** short ID**){**//function Definition

cout**<<**"hello "**<<**name**<<**endl**;**

cout**<<**"you've "**<<**age**<<**" yrs old"**<<**endl**;**

cout**<<**"your ID is "**<<**ID**<<**endl**;**

**}**

**Example**: addFunc

Write function to add 2 integers and return their sum as long int

**Return datatype**: long int

**Input parameters**: int x and int y

**Function name**: add

#include<iostream>

**using** **namespace** std**;**

long int add**(**int x**,** int y**);** //function Declaration

int main**(){**

int a**,**b**;**

long int sum **=** 0**;**

cout**<<**"enter the two addition operands :"**;**

cin**>>**a**>>**b**;**

sum **=** add**(**a**,**b**);**

cout**<<**"\n the sum is "**<<**sum**;**

**}**

long int add**(**int x**,** int y**){** //function Definition

**return** x **+** y**;**

**}**

**Example**: structFunc

write function to set a struct, the struct student which have name and id, the function return a struct after asking the user to enter name and id of the student

**Return datatype**: struct Student

**Input parameters**: nothing *//as the user will enter them in the function*

**Function name**: fillStruct

#include<iostream>

**using** **namespace** std**;**

struct Student**{**

string name**;**

int id**;**

**};**

Student fillStruct**();** //function Declaration

int main**(){**

Student Ahmed **=** fillStruct**();**

cout**<<**"Student: "**<<**Ahmed**.**name**<<**" has ID of "**<<**Ahmed**.**id**;**

**}**

Student fillStruct**(){** //function Definition

Student student**;**

string name**;**

int id**;**

cout**<<**"enter your name: "**;**

cin**>>**student**.**name**;**

cout**<<**"enter your ID: "**;**

cin**>>**student**.**id**;**

**return** student**;**

**}**

NOTE: the return datatype of the fillStruct() was Student datatype !

### Default arguments

Default parameters are used to set default values t function inputs, e.g. if function has input called age, if the user didn’t enter his age, the programmer may set 18 by default to handle the state when user didn’t enter his age ;

**Example**: areaFunc

Write function to calculate square and circle area, the function will have 2 parameters inputs int length to specify Radius or side, the parameter to determine length is for radius or side and determine the law of area is string shape which could be “circle” or “square” and by default it will be circle.

**Return datatype**: float

**Input parameters**: int length, string shape

**Function name**: calc\_area

#include<iostream>

**using** **namespace** std**;**

float calc\_area**(**int length**,** string shape **=** "circle"**);**

int main**(){**

double len**,** area**=**0**;**

string choice**;**

cout**<<**"do you want to calculate area of circle or square?" **;**

cin**>>**choice**;**

cout**<<**"\nplease enter the length (side or radius) :"**;**

cin**>>**len**;**

area **=** calc\_area**(**len**,** choice**);**

**}**

float calc\_area**(**int length**,** string shape**){**

**if(**shape **==** "circle"**){**

double pi **=** 3.14159265359**;**

cout**<<**"\nthe area of circle is: "**<<**pi**\***length**\***length**;**

**return** pi**\***length**;**

**}**

**else** **if(**shape **==** "square"**){**

cout**<<**"\nthe area of square is: "**<<**length**\***length**;**

**return** length**\***length**;**

**}**

**else{**

cout**<<**"\nshape must be 'circle' or 'square'"**<<**endl**;**

**return** 0**;**

**}}**

NOTE: All default parameters should be at last (most right) in declaration

float calc\_area**(**int length**,** string shape **=** "circle"**);**

NOT float calc\_area**(**string shape **=** "circle”, int length**); //error**

## Overloading and Inline Functions

### Inline functions

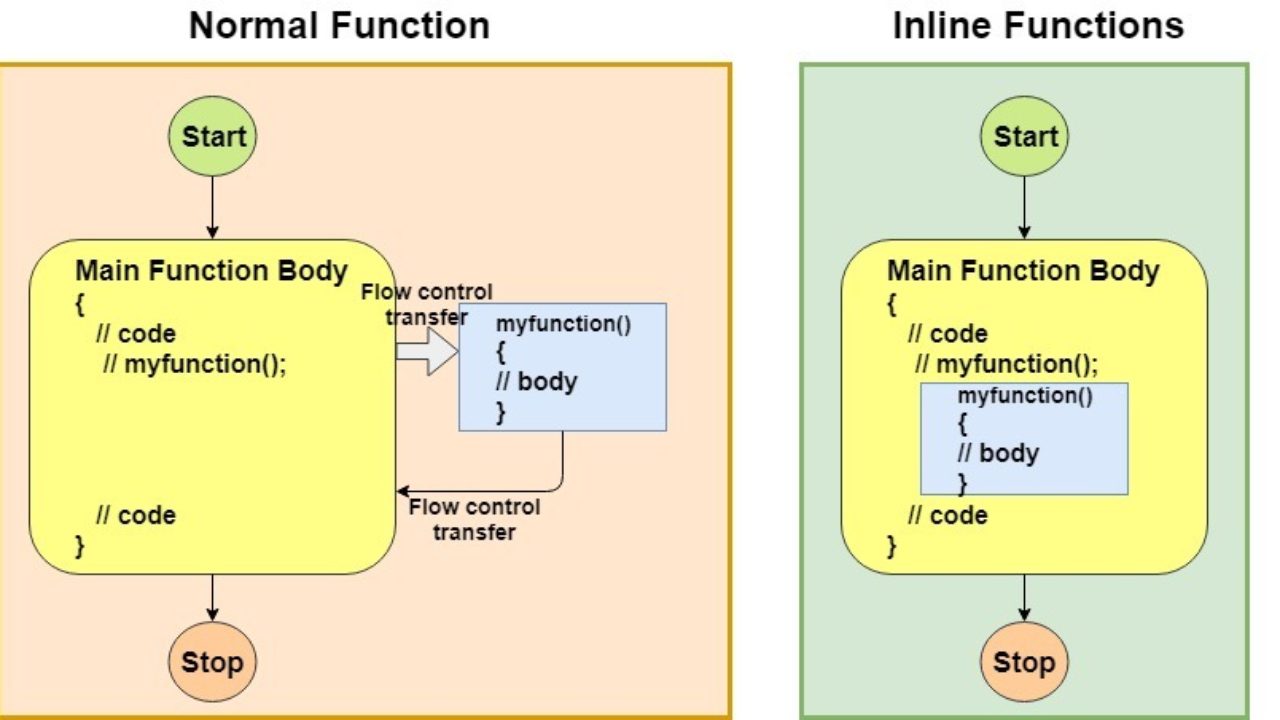
Inline functions are optimization technique done by compiler to replace the call function by function code (i.e. instructions), this makes the code faster as it decrease function call overhead (e.g. passing parameters and return the output) see Figure 31 inline function vs normal function

Figure 30 inline function vs normal function

Figure 31 inline function vs normal function

NOTE: inline function done for small functions

NOTE: compiler determine whether let inline function act as inline or normal fuction unless you pass \_\_attribute\_\_((always\_inline)) before inline

**Example**: inlineFunc

Make inline function to add 2 floats and return a float

**Return datatype**: float

**Input parameters**: float x , float y

**Function name**: add

#include<iostream>

**using** **namespace** std**;**

inline float add**(**float x**,** float y**);**

int main**(){**

float product**;**

float x**,**y**;**

cout**<<**"enter the 2 operands :"**;**

cin**>>**x**>>**y**;**

product **=** add**(**x**,**y**);**

cout**<<**"the addition is: "**<<**product**;**

**}**

inline float add**(**float x**,** float y**){**

**return** x **+** y**;**

**}**

NOTE: the compiler may decline the inline based on optimization level so add \_\_attribute\_\_((always\_inline)) before inline

#include<iostream>

**using** **namespace** std**;**

\_\_attribute\_\_**((**always\_inline**))**inline float add**(**float x**,** float y**);**

int main**(){**

float product**;**

float x**,**y**;**

cout**<<**"enter the 2 operands :"**;**

cin**>>**x**>>**y**;**

product **=** add**(**x**,**y**);**

cout**<<**"the addition is: "**<<**product**;**

**}**

\_\_attribute\_\_**((**always\_inline**))**inline float add**(**float x**,** float y**)**

**{**

**return** x **+** y**;**

**}**

### Overloading

Overloading is done when 2 or more function having same name but different parameter list (i.e. different parameter: datatypes, order, or number )

Recall **Example** addFunc

#include<iostream>

**using** **namespace** std**;**

long int add**(**int x**,** int y**);** //function Declaration

int main**(){**

float a**,**b**;**

long int sum **=** 0**;**

cout**<<**"enter the two addition operands :"**;**

cin**>>**a**>>**b**;**

sum **=** add**(**a**,**b**);**

cout**<<**"\n the sum is "**<<**sum**;**

**}**

long int add**(**int x**,** int y**){** //function Definition

**return** x **+** y**;**

**}**

What happens if we entered a and b 5.2 and 1.3 (i.e. entering floats) but the function takes integer as in declaration ?! truncation will be happen as the compile waits for integer and user provide float, the compiler may truncate the decimal point and treat 5.2 as 5 and 1.3 as 1 so output is 6 !! see Figure 32 truncation input parameters

A screen shot of a computer

Description automatically generated

Figure 32 truncation input parameters

how to SOLVE?? Overloading came to help

**Example**: overloading

Use overloading to make add() in addFunc example handles both integers and floats

#include<iostream>

**using** **namespace** std**;**

double add**(**int x**,** int y**);** //function Declaration

double add**(**float x**,** float y**);** //function Declaration

int main**(){**

float a**,**b**;**

double sum **=** 0**;**

cout**<<**"enter the two addition operands :"**;**

cin**>>**a**>>**b**;**

sum **=** add**(**a**,**b**);**

cout**<<**"\n the sum is "**<<**sum**;**

**}**

double add**(**int x**,** int y**){** //function Definition

**return** x **+** y**;**

**}**

double add**(**float x**,** float y**){** //function Declaration

**return** x **+** y**;**

**}**

See! The function behave based upon the input, the function call function with float inputs when the input parameters are floats and call int function when inputs are int, overloading solved the problem

A computer screen with white text

Description automatically generated

Figure 33 output of overloading example

**VI NOTE: templates solve the problem better, later we will discuss it**

## Recursive Functions

Recursive functions are functions that call itself, this must contain what in the Figure 34 recursion composition



Figure 34 recursion composition

NOTE: The recursion must contains base case to stop the recursion

**Example**: factRecursion

Write recursive function to calculate factorial of a number

#include<iostream>

**using** **namespace** std**;**

int fact**(**int num**);**

int main**(){**

int n**;**

cin**>>**n**;**

cout**<<**"the answer is: "**<<**fact**(**n**);**

**}**

int fact**(**int num**){**

//base case

**if(**num**==**1**)**

**return** 1**;**

**else{**

**return** num**\***fact**(**num**-**1**);**

**}**

**}**

**Example**: powerRecursion

Write recursive function to calculate power of a base and power expression

#include<iostream>

**using** **namespace** std**;**

int pwr**(**int base**,** int pow\_num**);**

int main**(){**

int base**,** power**;**

cout**<<**"enter base and power resp. "**;**

cin**>>**base**>>**power**;**

cout**<<**"the answer is: "**<<**pwr**(**base**,**power**);**

**}**

int pwr**(**int base**,** int pow\_num**){**

//base case

**if(**pow\_num**==**0**)**

**return** 1**;**

**else{**

**return** base**\***pwr**(**base**,**pow\_num**-**1**);**

**}**

**}**

## Pass by value, reference and pointer

In functions, the input parameters could be passed by value or by reference or by pointer, let’s see the differences.

### Pass by value

Till now, all parameters passed in this book is passed by value which is a copy of the variable not the variable itself, so you can’t modify the variable by passing by value, pass by value is useful when a variable is wanted but not to modify it,

**Example:** passByVal

Write function to add 2 integers and save the result in another value

**Return datatype**: int

**Input parameters**: int a , int b, int sum

**Function name**: add

#include<iostream>

**using** **namespace** std**;**

int add**(**int a**,** int b**,** int sum**){**

sum **=** a**+**b**;**

**return** sum**;**

**}**

int main**(){**

int x**=**5 **,**y**=**6**,** product**=**0**;**

//pass by value add(x,y, product);

cout**<<**"the addition is :" **<<**add**(**x**,**y**,** product**)<<**" the product var is "**<<**product**;**

**}**

See the output in Figure 35 pass by value output the product variable has not changes despite the fact we passed it and modify it, because we did not pass the product variable itself we passed an image (copy)

A black background with white text

Description automatically generated

Figure 35 pass by value output

### Pass by reference

If we want to modify the variable itself, we could do it by passing by reference, to do so, just add reference operator & before parameter name in function declaration **e.g.** int add(int a , int b, int &sum); and pass it by add(x, y, product) the product passed by reference and could be modified but x and y passed by value and could not be modified.

**Example**: passByRef

Modify example passByVal to make the product modified correctly.

#include<iostream>

**using** **namespace** std**;**

int add**(**int a**,** int b**,** int **&**sum**){**

sum **=** a**+**b**;**

**return** sum**;**

**}**

int main**(){**

int x**=**5 **,**y**=**6**,** product**=**0**;**

//pass by reference

cout**<<**"the addition is :" **<<**add**(**x**,**y**,** product**)<<**" the product var is "**<<**product**;**

**}**

NOTE: the only change was the reference operator & before sum in function declaration

NOTE: this function is declared and defined at one block not separated, you can separate the declaration and definition or combine them (same meaning), but some times we declare function in file and the definition in other file (will be discussed later in compilation process chapter)

### Pass by pointer

passing by pointer is as same as pass by reference but in pass by reference we pass the variable itself but in pass by pointer we pass the address of that variable that also could modified after dereferencing it (i.e. using \*)

the key feature of pass by value is that we could move the variable to different memory location as we pass the address, but remember, passing by pointer is ticky sometimes as pointers could be NULL (i.e. pointing to noting) this fault is programmer’s fault when he forgot to assign the pointer to point to the variable before calling the function

**Example**: passByPointer

Write function to perform swap of two variable without using the third variable temp like in Figure 36 swap by using third variable temp instead use the method in Figure 37 swap without using third variable temp

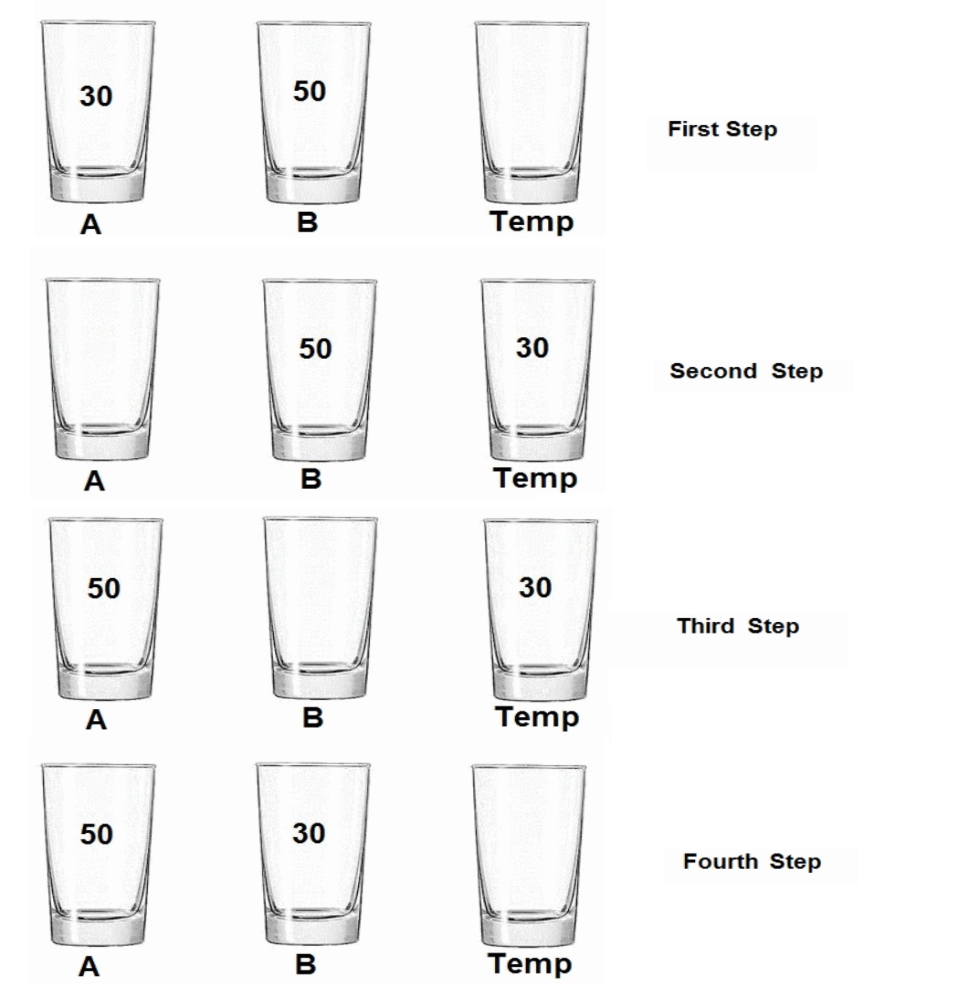


Figure 36 swap by using third variable temp

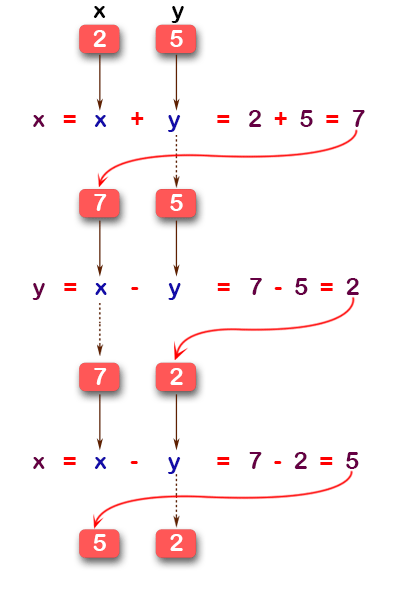


Figure 37 swap without using third variable temp

**Return datatype**: void //as we don’t output we only want to swap

**Input parameters**: int a, int b

**Function name**: swapping

#include<iostream>

**using** **namespace** std**;**

void swapping**(**int **\***a**,** int **\***b**){**

**\***a **=** **\***a **+** **\***b**;**

**\***b **=** **\***a **-** **\***b**;**

**\***a **=** **\***a **-** **\***b**;**

**}**

int main**(){**

int x**=**5 **,**y**=**6**;**

//pass by pointer

cout**<<**"Before swapping:\nx is:" **<<**x**<<**"\ny is:"**<<**y**<<**endl**;**

//swap

swapping**(&**x**,&**y**);**

/\*dont forget in pass by pointer

to add in reference operator in call\*/

cout**<<**"After swapping:\nx is:" **<<**x**<<**"\ny is:"**<<**y**<<**endl**;**

**}**

NOTE: don’t forget in pass by pointer to add in reference operator in call

swapping**(&**x**,&**y**);**

See Figure 38 pass by pointer example output

A black screen with white text

Description automatically generated

Figure 38 pass by pointer example output

## Final Project

Write Linked List program as functions

The program has the following functions:

* Append element at last
* Insert element
* Delete element
* Print element

**Code**: linkedList

#include <iostream>

**using** **namespace** std**;**

//creating linked list structure

struct Node**{**

int DATA**;**

Node **\***NEXT**;**

**};**

//function declarations

void Append**(**Node**\*** **&**head**,**int data**);**

void Insert**(**Node**\*** **&**head**,**int data**,** int index**);**

void Delete**(**Node**\*** **&**head**,** int index**);**

void Print**(**Node**\*** head**);**

int main**()** **{**

//creating head

Node**\*** head**=nullptr;**

Append**(**head**,** 15**);**

Append**(**head**,** 42**);**

Append**(**head**,** 70**);**

Insert**(**head**,** 61**,** 2**);**

Print**(**head**);**

Delete**(**head**,** 2**);**

Print**(**head**);**

**return** 0**;**

**}**

//function Definitions

void Append**(**Node**\*** **&**head**,**int data**){**

cout**<<**"Appending Node "**<<**data**<<**" \n"**;**

//create the node

Node**\*** newNode **=** **new** Node**;**

newNode**->**DATA **=** data**;**

newNode**->**NEXT **=** **nullptr;**

//detect if there no node (only head)

**if(**head **==** **nullptr){**

head **=** newNode**;**

**}**

**else{**

//get the last node

Node**\*** temp **=** head**;**

**while(**temp**->**NEXT **!=** **nullptr){**

temp **=** temp**->**NEXT**;**

**}**

temp**->**NEXT **=** newNode**;**

**}**

**}**

void Insert**(**Node**\*** **&**head**,**int data**,** int index**){**

cout**<<**"Inserting Node "**<<** data**<<**" at "**<<**index**<<**" \n"**;**

//move to node before index node

Node**\*** temp **=** head**;**

**for(**int i**=**0**;**i**<**index**-**1**;**i**++){**

temp **=** temp**->**NEXT**;**

**}**

//crete new node

Node**\*** newNode **=** **new** Node**;**

newNode**->**DATA **=** data**;**

newNode**->**NEXT **=** temp**->**NEXT**;** //make the new node to point to node index+1

temp**->**NEXT **=** newNode **;**//name the node at index-1 points to new node

**}**

void Delete**(**Node**\*** **&**head**,** int index**){**

cout**<<**"Deleting Node at: "**<<**index**<<**endl**;**

//move to node before index node

Node**\*** temp **=** head**;**

**for(**int i**=**0**;**i**<**index**-**1**;**i**++){**

temp **=** temp**->**NEXT**;**

**}**

//unlinking:

//put the address saved in node(index-1) to be address of node(index+1)

Node**\*** toDelete **=** temp**->**NEXT**;**

temp**->**NEXT **=** **(**temp**->**NEXT**)->**NEXT**;**

//to delete the node ar index

**delete** toDelete**;**

**}**

void Print**(**Node**\*** head**){**

cout**<<**"Printing Nodes..\n"**;**

**while(**head **!=** **nullptr){**

cout**<<**"item: "**<<**head**->**DATA**<<**endl**;**

head **=** head**->** NEXT**;**

**}**

**}**

**NOTES**:

In decelerations:

//function declarations

void Append**(**Node**\*** **&**head**,**int data**);**

void Insert**(**Node**\*** **&**head**,**int data**,** int index**);**

void Delete**(**Node**\*** **&**head**,** int index**);**

void Print**(**Node**\*** head**);**

Node\* &head: means pass by reference (for modifying the linked list) and the data type is pointer (Node\*)

SO: Node\* is the data type

&head is passing by reference to be able to modify (add and delete nodes)

Entry point function main()

int main**()** **{**

// Creating head node

Node**\*** head **=** **nullptr;**

// Append nodes to the list

Append**(**head**,** 15**);**

Append**(**head**,** 42**);**

Append**(**head**,** 70**);**

// Insert a node at position 2

Insert**(**head**,** 61**,** 2**);**

// Print the list

Print**(**head**);**

// Delete the node at position 2

Delete**(**head**,** 2**);**

// Print the list again

Print**(**head**);**

**return** 0**;**

**}**

NOTE: the head node init with nullptr so next have nothing

In Append() function:

//function Definitions

void Append**(**Node**\*** **&**head**,**int data**){**

cout**<<**"Appending Node "**<<**data**<<**" \n"**;**

//create the node

Node**\*** newNode **=** **new** Node**;**

newNode**->**DATA **=** data**;**

newNode**->**NEXT **=** **nullptr;**

//detect if there no node (only head)

**if(**head **==** **nullptr){**

head **=** newNode**;**

**}**

**else{**

//get the last node

Node**\*** temp **=** head**;**

**while(**temp**->**NEXT **!=** **nullptr){**

temp **=** temp**->**NEXT**;**

**}**

temp**->**NEXT **=** newNode**;**

**}**

**}**

NOTE: head = newNode;

NOT head->next = newNode;

As head is the pointer to access the first node not a node itself

A qr code on a white background

Description automatically generatedGet the code from the repository [LINK](https://github.com/MuOssama/MasteringCPP/blob/main/Functions/Final%20Project/linkedList.cpp) or scan Qr code in Figure 39 linked List Qr code

Figure 39 linked List Qr code

# Preprocessor Directives

Preprocessor directive are keywords followed by hashtag #. Each keyword has functionality.

* **Macros**
  + #define, #undef
  + Function-like Macro
* **Conditional Compilation**
  + #ifdef, #ifndef, #if, #else, #elif, #endif
  + Conditional compilation directives
* **File Guards**

## Macros

Macros is text replacement, that replace the name by it value e.g. #define num 5, whenever num is typed, it will be replaced by 5 while #undef delete the effect of #define (undo the #define)

### #define and #undef

**Example**: define

Write code to print age and name without declaring any variables

#include<iostream>

#define name "Ahmed"

#define age 19

int main**(){**

std**::**cout**<<**"My name is "**<<**name**<<**" and I've "**<<**age**<<**"yrs old"**;**

**return** 0**;**

**}**

NOTE: preprocessor directives aren’t compiled its only replacement, think about it like after you wrote the code:

std**::**cout**<<**"My name is "**<<**name**<<**" and I've "**<<**age**<<**"yrs old"**;**

and then name and age you came and replace them by their values

std**::**cout**<<**"My name is "**<<**”Ahmed”**<<**" and I've "**<<**19**<<**"yrs old"**;**

**then the compiler change C++ code to binary**

* **The first issue** of not being compiled is that if you are not careful enough, some unintended behavior may occur and you will never knew like:

#include<iostream>

#define square(x) x\*x

int main**(){**

std**::**cout**<<**square**(**1**+**2**);**

//the answers is 5 as it replaces square(1+2)

//with x\*x (but x = 1+2)

//so after replacement

// 1+2\*1+2 which is 5

//but the intended value is 3\*3 which is 9

**return** 0**;**

**}**

the answers is 5 as it replaces square(1+2) with x\*x (but x = 1+2)

so after replacement 1+2\*1+2 which is 5

but the intended value is 3\*3 which is 9

Blind text replacement !!

* **The second issue** is that there is no type check

if you define a macro that performs an operation on a variable, the preprocessor won't check if the variable type is appropriate for the operation.

### Function-like macro

Object-like macro Is like what we did #define name “Ahmed”, function-like macro is a pseudo function e.g. #define square(x) x\*x, i.e.

multi-line marco is by its name function macro but with multiple lines, to do so, put \ at end of each line except the last macro line

**Example**: macroFunction

Write function-like macro with multi-lines macro, that print data of student who has name, age and id.

#include<iostream>

**using** **namespace** std**;**

#define print(name, age, id) cout<<"my name is "<< name<<endl; \

cout<<"I've "<< age<<" yrs old "<<endl; \

cout<<"my ID is "<< id;

int main**(){**

print**(**"Ahmed"**,** 19**,** 202014**)**

**return** 0**;**

**}**

NOTE: the semicolon ; is not written in print**(**"Ahmed"**,** 19**,** 202014**)**

As the last macro line have semicolon cout<<"my ID is "<< id;

You could put semicolon in print call but remove last line semicolon like:

#include<iostream>

**using** **namespace** std**;**

#define print(name, age, id) cout<<"my name is "<< name<<endl; \

cout<<"I've "<< age<<" yrs old "<<endl; \

cout<<"my ID is "<< id

int main**(){**

print**(**"Ahmed"**,** 19**,** 202014**);**

**return** 0**;**

**}**

Some developers what to write real multi line function like macro like simple macro, as

print(name, age, id) cout<<"my name is "<< name<<endl; \

cout<<"I've "<< age<<" yrs old "<<endl; \

cout<<"my ID is "<< id;

lacks the {} to hold function body (in this case the 3 cout lines).

the modification:

print(name, age, id) {cout<<"my name is "<< name<<endl; \

cout<<"I've "<< age<<" yrs old "<<endl; \

cout<<"my ID is "<< id;}

**Example**: multiLineMacro

#include<iostream>

**using** **namespace** std**;**

#define valid true

#define print(name, age, id) {cout<<"my name is "<< name<<endl; \

cout<<"I've "<< age<<" yrs old "<<endl; \

cout<<"my ID is "<< id;\

}

int main**(){**

**if(**valid**)**

print**(**"Ahmed"**,** 19**,** 202014**);**

**else**

cout**<<**"not valid"**;**

**return** 0**;**

**}**

***This will throw and error as expansion will be:***

int main**(){**

**if(**valid**)**

{cout<<"my name is "<< “Ahmed”<<endl; \

cout<<"I've "<< 19<<" yrs old "<<endl; \

cout<<"my ID is "<< 202014;\

};

**else**

cout**<<**"not valid"**;**

**return** 0**;**

**}**

So in this case you are forced to remove semicolon from the call

print**(**"Ahmed"**,** 19**,** 202014**)**

**but it not what C++ developer used to do, so another approach is do-while multi lines**

remember: do while statement is as follows:

do{statements}

while(condition);

so after while a semicolon should be add, we could use this fact to put semicolon at calls

#include<iostream>

**using** **namespace** std**;**

#define valid true

#define print(name, age, id) do{cout<<"my name is "<< name<<endl; \

cout<<"I've "<< age<<" yrs old "<<endl; \

cout<<"my ID is "<< id;\

}while(0)

int main**(){**

**if(**valid**)**

print**(**"Ahmed"**,** 19**,** 202014**);**

**else**

cout**<<**"not valid"**;**

**return** 0**;**

**}**

The expansion will be:

int main**(){**

**if(**valid**)**

do{cout<<"my name is "<< name<<endl; \

cout<<"I've "<< age<<" yrs old "<<endl; \

cout<<"my ID is "<< id;\

}while(0);

**else**

cout**<<**"not valid"**;**

**return** 0**;**

**}**

Works well !!

## Conditional Compilation

#if, #else, #elif, #endif

These macros used to do conditionals (i.e if and elif and else) before runtime, as preprocessor directives are text replacement and is done in compilation not run time, this will speed the code as many code lines are determined before the run time

NOTE: you must add #endif at end of contidions

**Example**: conditionalCompilation

Write C++ code that determines the area of shape , don’t use if and else if and else, instead use #if #elif #else

#include<iostream>

**using** **namespace** std**;**

// you have to options:

// 0 for circle

// 1 for square

#define circle 0

#define square 1

#define shape square

int main**(){**

#if shape == circle

float pi **=** 3.14**,**raduis**=**0**;**

cout**<<**"enter the radius "**;**

cin**>>**raduis**;**

cout**<<**"\nArea of cricle is "**<<**pi**\***raduis**\***raduis**;**

#elif shape == square

float len**=**0**;**

cout**<<**"enter the length "**;**

cin**>>**len**;**

cout**<<**"\nArea of square is "**<<**len**\***len**;**

#endif

**return** 0**;**

**}**

NOTE: compilation conditionals don’t perform string comparison like:

#include<iostream>

**using** **namespace** std**;**

// you have to options: circle square

#define shape "circle"

int main**(){**

#if shape == "circle"

float pi **=** 3.14**,**r**=**0**;**

cout**<<**"enter the radius "**;**

cin**>>**raduis**;**

cout**<<**"\nArea of "**<<**shape**<<**" is "**<<**pi**\***r**\***r**;**

#elif shape == "square"

float len**=**0**;**

cout**<<**"enter the length "**;**

cin**>>**len**;**

cout**<<**"\nArea of "**<<**shape**<<**" is "**<<**len**\***len**;**

#endif

**return** 0**;**

**}**

***This is an error !***

#if shape == "circle"

#elif shape == "square"

Are wrong !

## File guards

In C/C++ .cpp files are source files that contains code implementation while another type of file has suffix of .h files which are header file that contains variable and functions declarations, the header file is included in the .cpp file by #include”filename.h”, remember #include<> is used for bultin function like iostream and stdio.h while user header files are included by “ ” not <> see

**Example**: headers

Write application that has .cpp file contains the implementation of printing student data (name, age, id) and 2 headers, first header contains the student struct and other contains the student data to be filled in the struct

* The first file headers.cpp is the source file

#include <iostream>

#include "headers1.h"

#include "headers2.h"

**using** **namespace** std**;**

void fillStruct**(**Student **&**stud**){**

stud**.**name **=** studentName**;**

stud**.**age **=** studentAge**;**

stud**.**id **=** studentId**;**

**}**

int main**(){**

Student Ahmed**;**

fillStruct**(**Ahmed**);**

cout**<<**"Student "**<<**Ahmed**.**name**<<**" has "**<<**Ahmed**.**age**<<**" yrs old.\nID: "**<<**Ahmed**.**id**;**

**}**

* The first header headers1.h

#include<string>

struct Student**{**

std::string name**;**

int age**;**

int id**;**

**};**

void fillStruct**(**Student **&**stud**);**

* The second header headers2.h

#define studentName "Ahmed"

#define studentAge 15

#define studentId 202015

The source file headers.cpp will replace each file by its contents, like this:

* Headers.cpp before replacing headers1.h and headers2.h

#include <iostream>

#include "headers1.h"

#include "headers2.h"

**using** **namespace** std**;**

void fillStruct**(**Student **&**stud**){**

stud**.**name **=** studentName**;**

stud**.**age **=** studentAge**;**

stud**.**id **=** studentId**;**

**}**

int main**(){**

Student Ahmed**;**

fillStruct**(**Ahmed**);**

cout**<<**"Student "**<<**Ahmed**.**name**<<**" has "**<<**Ahmed**.**age**<<**" yrs old.\nID: "**<<**Ahmed**.**id**;**

**}**

* Headers.cpp After replacing headers1.h and headers2.h

#include <iostream>

#include<string>

struct Student**{**

std::string name**;**

int age**;**

int id**;**

**};**

void fillStruct**(**Student **&**stud**);**

#define studentName "Ahmed"

#define studentAge 15

#define studentId 202015

void fillStruct**(**Student **&**stud**){**

stud**.**name **=** studentName**;**

stud**.**age **=** studentAge**;**

stud**.**id **=** studentId**;**

**}**

int main**(){**

Student Ahmed**;**

fillStruct**(**Ahmed**);**

cout**<<**"Student "**<<**Ahmed**.**name**<<**" has "**<<**Ahmed**.**age**<<**" yrs old.\nID: "**<<**Ahmed**.**id**;**

**}**

See the output Figure 40 headers example output

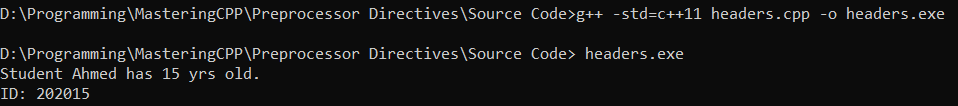


Figure 40 headers example output

But there is one problem, what if we include a header.h multiple times?? the .cpp files include the same header

The compiler will throw redefinition error see Figure 41 redefinition error

A black screen with white text

Description automatically generated

Figure 41 redefinition error

Because struct is defined twice so we could use file guard to protect against redefinition error, the file guard method is done by typing:

at start the header file:

#ifndef anyHeaderName

#define anyHeaderName

At and the header file:

#endif

So Coding the headers example again:

* The headers.cpp remain as its
* The headers1.h will be:

#ifndef HEADERS1

#define HEADERS1

#include<string>

struct Student**{**

std**::**string name**;**

int age**;**

int id**;**

**};**

void fillStruct**(**Student **&**stud**);**

#endif

* The headers2.h will be:

#ifndef HEADERS2

#define HEADERS2

#define studentName "Ahmed"

#define studentAge 15

#define studentId 202015

#endif

Even in headers.cpp if one header .h included multiple times, no redefinition error

# Compilation Process

Programing languages have ways to make sure that code is valid and convert the programming lines of code into format that computer machines understand, the two ways are compilation and interpretation

* **Compilation** is changing C/C++ files (.c files ot .cpp files) into executable format that computers or microcontrollers understand (e.g. .exe for computers run windows or .hex for microcontroller), after conversion into binaries (i.e. .exe or .hex) the target machine can run the code
* **Interpretation** converting lines of code like python lines into something targes understand on runtime, the interpreter interprets line by line on runtime
* Preprocessing
* Compilation
* Assembly
* Linking
* Compiling many source files

C and C++ are compiled languages, the compilation process is as follows in Figure 42 Compilation process,

* the source code is the .cpp file that contains C++ code, it then gets in the process of removing preprocessor directives (e.g. #define #if etc.)
* then the compiler converts the output into assembly code.
* then the assembler converts the assembly code into machine code.
* typically, there will be many source files each will be converted into machine code (object code), the linker will combine all object files into only one executable .

## Compilation processCompiling a C Program: Behind the Scenes - GeeksforGeeks

Figure 42 Compilation process

In the following, examples we will demonstrate the process:

**Example**: main

Write C++ code that have preprocessor directives and print name of user named Ahmed

#include<iostream>

#define username "Ahmed"

int main**(){**

std**::**cout**<<**"username is "**<<**username**;**

**return** 0**;**

**}**

Here is the source code main.cpp

### Preprocessor directive

**Example**: preprocessor

Convert main.cpp into preprocessor directive free using command

g++ -std=c++11 -E main.cpp -o preprocessor.i

The output which is preprocessor directives free in this small example is 30088 lines, and what last few lines will be:

# 4 "main.cpp"

int main(){

std::cout<<"My name is "<<"Ahmed"<<" and I've "<<19<<"yrs old";

return 0;

}

Lets explain:

#include<iostream>

#define username "Ahmed"

int main**(){**

std**::**cout**<<**"username is "**<<**username**;**

**return** 0**;**

**}**

The source file, have preprocessor directive #include, so this line will be replaced by built in file of iostream which is around 30080 lines

Then the next line there is another preprocessor directive #define so every line has username keword, will be replacement with the value of username which is “Ahemd”, thas why the line:

std**::**cout**<<**"username is "**<<**username**;**

changed into:

std::cout<<"My name is "<<"Ahmed"<<" and I've "<<19<<"yrs old";

in preprocceor deirecives free output after command: g++ -std=c++11 -E main.cpp -o preprocessor.i

### Compiler

The compiler will receive the preprocessor directive free output (preprocessor.i)a nd convert it into assembly code named assembly.S

**Example**: compiler

Convert preprocessor.i into assembly code using command

g++ -std=c++11 -S preprocessor.i -o assembly.s

.file "main.cpp"

.text

.section .rdata,"dr"

\_ZStL19piecewise\_construct**:**

.space 1

.LC0**:**

.ascii "My name is \0"

.LC1**:**

.ascii "Ahmed\0"

.LC2**:**

.ascii " and I've \0"

.LC3**:**

.ascii "yrs old\0"

.text

.globl main

.def main; .scl 2; .type 32; .endef

.seh\_proc main

main**:**

.LFB1808**:**

pushq %rbp

.seh\_pushreg %rbp

movq %rsp, %rbp

.seh\_setframe %rbp, 0

subq **$**32, %rsp

.seh\_stackalloc 32

.seh\_endprologue

call \_\_main

leaq .LC0**(**%rip), %rax

movq %rax, %rdx

movq .refptr.\_ZSt4cout**(**%rip), %rax

movq %rax, %rcx

call \_ZStlsISt11char\_traitsIcEERSt13basic\_ostreamIcT\_ES5\_PKc

movq %rax, %rcx

leaq .LC1**(**%rip), %rax

movq %rax, %rdx

call \_ZStlsISt11char\_traitsIcEERSt13basic\_ostreamIcT\_ES5\_PKc

movq %rax, %rcx

leaq .LC2**(**%rip), %rax

movq %rax, %rdx

call \_ZStlsISt11char\_traitsIcEERSt13basic\_ostreamIcT\_ES5\_PKc

movl $19, %edx

movq %rax, %rcx

call \_ZNSolsEi

movq %rax, %rcx

leaq .LC3(%rip), %rax

movq %rax, %rdx

call \_ZStlsISt11char\_traitsIcEERSt13basic\_ostreamIcT\_ES5\_PKc

movl $0, %eax

addq $32, %rsp

popq %rbp

ret

.seh\_endproc

.def \_\_main; .scl 2; .type 32; .endef

.ident "GCC: (MinGW-W64 x86\_64-ucrt-posix-seh, built by Brecht Sanders, r1) 14.1.0"

.def \_ZStlsISt11char\_traitsIcEERSt13basic\_ostreamIcT\_ES5\_PKc; .scl 2; .type 32; .endef

.def \_ZNSolsEi; .scl 2; .type 32; .endef

.section .rdata$.refptr.\_ZSt4cout, "dr"

.globl .refptr.\_ZSt4cout

.linkonce discard

.refptr.\_ZSt4cout:

.quad \_ZSt4cout

the output will be this, the mov and add and call are the assembly code related to my machine that output this which is intel x86 architecture.

NOTE: the compilation oupout (assembly code) will be different with every target, if you run this on ARM or MIPS for example, the output will be different as the instruction set architecture is different from target to another

NOTE: instruction set architecture(ISA) is the interface between hardware (the real register of the target) and the software that user writes, so the commands like in intel x86 mov add call will be changed into machine code and the ISA inside the processor will interpret this software command and perform it by using the hardware like the registers inside the target.

### Assembler

The Assembler will receive the assembly code (assembly.s)a and convert it into object file (hex decimal code)

**Example**: objectFile

Convert main.cpp into assembly code using command

g++ -std=c++11 -S assembly.s -o object.o

to open the output object file, you require a hex viewer, you can download free hex viewer neo from this [LINK](https://hhdsoftware.com/free-hex-editor)

In the following: Figure 43 Object file

A screenshot of a computer

Description automatically generated

Figure 43 Object file

The output is the hex decimal (bineries that the target (intel x86 computer) understands)

### Linker

The linker links all the object files into one executable, this is used by this command: g++ -std=c++11 object.o -o main.exe

In this book till now we used to use one command that do all what we have done in this chapter which is

g++ -std=c++11 main.cpp -o main.exe

see the summary in Figure 44 Compiling multiple source files

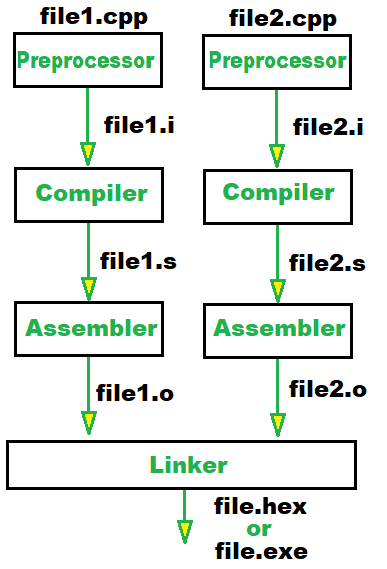


Figure 44 Compiling multiple source files

## Compile multiple files

All Examples in this book so far have only one .cpp file, what if we have multiple cpp files like in Figure 44 Compiling multiple source files? We can convert each .cpp file into object file then all object files into one executable file OR convert all .cpp files into one executable file by only one command

### Convert each file into object files then link

Compile each source file:

g**++** **-**c file1**.**cpp **-**o file1**.**o

g**++** **-**c file2**.**cpp **-**o file2**.**o

g**++** **-**c file3**.**cpp **-**o file3**.**o

Link all source files:

g**++** file1**.**o file2**.**o file3**.**o **-**o myExecutable

### Convert all .cpp files into one executable file

This is the easy way, compile all source files into one executable

g**++** file1**.**cpp file2**.**cpp file3**.**cpp **-**o myExecutable

NOTE: don’t put any .h files, put only source files (i.e. .cpp files)

**Example**: MultpileCompilations

Write main.cpp files to setUser() and printUser, these functions is declared one in file1.h and the other in file2.h while file1.cpp and file2.cpp have the implementation of these function, the User is a struct that has name and id

NOTE: use file guards !!

file1.h:

#ifndef FILE1

#define FILE1

#include "file2.h"

void printUser**(**USER **&**user**);**

#endif

file1.cpp:

#include"file1.h"

void printUser**(**USER **&**user**){**

std**::**cout**<<**"Username "**<<**user**.**name**<<**" ID "**<<**user**.**id**<<**std**::**endl**;**

**}**

file2.h:

#ifndef FILE2

#define FILE2

#include<iostream>

struct USER**{**

std**::**string name**;**

int id**;**

**};**

void setUser**(**USER **&**user**);**

#endif

file2.cpp:

#include"file2.h"

void setUser**(**USER **&**user**){**

std**::**cout**<<**"enter username and id: "**;**

std**::**cin**>>**user**.**name**>>**user**.**id**;**

**}**

main.cpp:

#include"file2.h"

#include"file1.h"

int main**(){**

USER Ahmed**;**

setUser**(**Ahmed**);**

printUser**(**Ahmed**);**

**}**

**Compilation**:

g++ -std=c++11 main.cpp file1.cpp file2.cpp -o main.exe

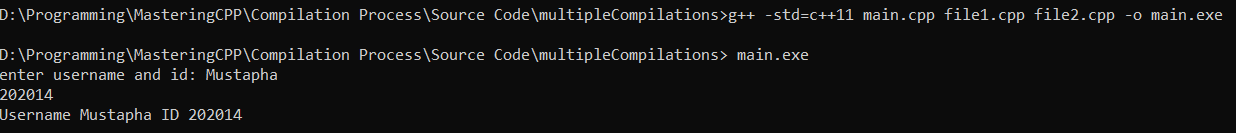
See the output in Figure 45 multiple file compilation and output:

Figure 45 multiple file compilation and output

# Object oriented programming OOP

In this chapter, object-oriented programming will be introduced, a tic tac toe game will be the final project, after starting from creating empty class then stacking and implementing OOP concepts concept by concept using animal class examples.

PLEASE: read the whole chapter one time fast, before starting studying thoroughly

NOTE: OOP has main four items to know despite the programming language you use

* **Inheritance** (make child class from parent class)
* **Encapsulation** (hide data for safety)
* **Polymorphism** (make multiple (poly) use for one function depending which child class use this function)
* **Abstraction** (abstract user from studying the code and let him build on your code easily)

Topics:

* **Classes and Objects**
  + Class definition and declaration
  + Access specifiers: public, private, protected
  + Member variables and member functions
  + Object instantiation
* **Constructors and Destructors**
  + Default constructor
  + Parameterized constructor
  + Copy constructor
  + Destructor
* **Inheritance**
  + Base and derived classes
  + Types of inheritance: single, multiple, multilevel, hierarchical, hybrid
  + Constructor and destructor calls in inheritance
* **Polymorphism**
  + Compile-time polymorphism: function overloading, operator overloading
  + Runtime polymorphism: virtual functions, pure virtual functions, abstract classes
* **Encapsulation**
  + Data hiding
  + Setter and Getter (Accessor and mutator functions)
* **Abstraction**
  + Abstract classes and interfaces
  + Virtual function and pure virtual function
* **Operator Overloading**
  + Overloading unary and binary operators
  + Overloading operators using member and friend functions
* **Static Members**
  + Static member variables and functions
  + Class-level data and behavior
* **Multiple Inheritance**
  + Diamond problem and virtual inheritance
* **Rules**
  + Rule of Three

## Classes and Objects

Classes are as same as structs, the only difference between them are the access modifiers defaults , when you make a class, its members are private by default and in structs are public by defaults. Like:

**Example**: structVsClass1

#include<iostream>

**using** **namespace** std**;**

struct StudentStruct**{**

int gpa**;**

**};**

class StudentClass**{**

int gpa**;**

**};**

int main**(){**

StudentClass Mohamed**;**

StudentStruct Ahmed**;**

//Ahmed is struct instance so gpa is public

//(could be accesed and modified)

Ahmed**.**gpa **=** 3.45**;**

//Mohamed is class instance so gpa is private

//couldn't be accesed and modified)

// Mohamed.gpa = 3.58; //error

**return** 0**;**

**}**

Also in inheritance the default child modifier is private for classes and public for structs

**Example**: structVsClass2

#include<iostream>

**using** **namespace** std**;**

struct AnimalStruct**{**

//parent struct

int age**;**

**};**

struct TigerStruct**:** AnimalStruct**{**

//child struct

int speed**;**

**};**

class AnimalClass**{**

//parent class

int age**;**

**};**

class TigerClass**:** AnimalClass**{**

//child struct

int speed**;**

**};**

int main**(){**

TigerClass tiger**;**

TigerStruct tigress**;**

//tigress inherits age as public

tigress**.**age **=** 15**;**

//tiger error its private inheritance

//so age becomes private for the child

tiger**.**age **=** 15**;**

**return** 0**;**

**}**

In TigerStruct child the default is public inheritance i.e.

struct TigerStruct**: public** AnimalStruct**{**

//child struct

int speed**;**

**};**

While in TigerClass child the default is private inheritance i.e.

struct TigerStruct**: private** AnimalStruct**{**

//child struct

int speed**;**

**};**

Anyway, if you specify the access modifiers like:

struct TigerStruct**: public** AnimalStruct**{**

//child struct

int speed**;**

**};**

struct TigerStruct**: protected** AnimalStruct**{**

//child struct

int speed**;**

**};**

struct TigerStruct**: private** AnimalStruct**{**

//child struct

int speed**;**

**};**

A screenshot of a computer

Description automatically generatedRecall Access modifier from Basics in Figure 46 Access modifiers in inheritance

Figure 46 Access modifiers in inheritance

It seems there is no big difference but mostly all OOP topics either in book or courses, will use classes so we will stick to convention !

### Class definition and declaration

As we introduced, the classes like struct in everything like definition and declaration but different in default access modifiers, just replace struct keyword by class keyword

**Example**: animalClass

Write class for animal that has sound and have name and age and length

#include<iostream>

**using** **namespace** std**;**

class Animal**{**

public**:**

string name**;**

int age**;**

float length**;**

//function member

void sound**(**string sound**){**

cout**<<**name**<<**"have "**<<**sound**<<**" sound\n"**;**

**}**

**};**

int main**(){**

**return** 0**;**

**}**

### Access specifiers: public, private, protected

As we mentioned earlier access modifiers determine how to access class/struct members (i.e. function (method) and attributes)

**public**: is to access or modify members anywhere

**private**: is accessed inside the class/struct only, even child class could modify or access these members

**protected:** like private but also child class could access or modify these members

|  |  |  |  |
| --- | --- | --- | --- |
|  | public members | private members | protected members |
| Accessed inside class ? | YES | YES | YES |
| Accessed inside child classes ? | YES | NO | YES |

### Member variables and member functions

Members are variables (attributes) or functions (method), objects in real life have attributes (e.g. name, length, age etc..) and have something done by them (e.g. playing, eating, make sounds etc..) or something happen to them (being eaten, dies, grow etc…) so in object oriented programming we have:

* **Variables** : to mimic object attributes
* **Functions** (methods): to mimic object function or what object can do or how the object is changes.

in animal example, name, age, length are class variable (class attributes) while sound(string sound) is class function (class method)

### Object instantiation

To use the objects, we made we have to instantiate (i.e. make instance), like we have mentioned, classes are like structs, so same instantiation but replace struct by class keyword

**Example**: animalInstantiation

Instantiate tiger as instance or animal class that we made previously

#include<iostream>

**using** **namespace** std**;**

class Animal**{**

public**:**

string name**;**

int age**;**

float length**;**

//function member

void sound**(**string sound**){**

cout**<<**name**<<**" have "**<<**sound**<<**" sound\n"**;**

**}**

**};**

int main**(){**

Animal tiger**;**

tiger**.**name **=** "tiger"**;**

tiger**.**sound**(**"roar"**);**

**return** 0**;**

**}**

The output: tiger have roar sound

## Constructors and Destructors

Constructors are function that got called when an instance is made while a destructor are called when objects go out of scope or when cleaned (i.e. deleted)

### Default constructor

Constructor that have no parameters at all

**Example**: defaultConstructor

Make a default constructior to set name of the animal class we made

#include<iostream>

**using** **namespace** std**;**

class Animal**{**

public**:**

string name**;**

int age**;**

float length**;**

//default constructor

Animal**(){**

cout**<<**"constructor called, enter animal name: \n"**;**

cin**>>**name**;**

**}**

//function member

void sound**(**string sound**){**

cout**<<**name**<<**" have "**<<**sound**<<**" sound\n"**;**

**}**

**};**

int main**(){**

Animal tiger**;**

tiger**.**sound**(**"roar"**);**

**return** 0**;**

**}**

The output should be as : Figure 47 default constructor example output

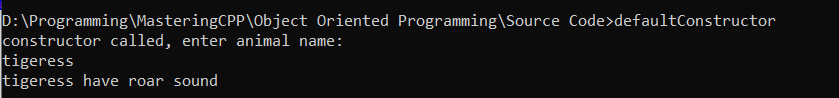


Figure 47 default constructor example output

### Parameterized constructor

Parameterized constructor is like default constructor but have parameters

**Example**: parameterizedConstructor

Modify *defalutConstructor* example to make it more easier by using Parameterized constructor

#include<iostream>

**using** **namespace** std**;**

class Animal**{**

public**:**

string name**;**

int age**;**

float length**;**

//Parameterized constructor

Animal**(**string s**):** name**(**s**){**

**}**

//function member

void sound**(**string sound**){**

cout**<<**name**<<**" have "**<<**sound**<<**" sound\n"**;**

**}**

**};**

int main**(){**

Animal tiger**(**"tigress"**);**

tiger**.**sound**(**"roar"**);**

**return** 0**;**

**}**

output: tigress have roar sound

NOTE: you can initialize all attributes not only name like:

//Parameterized constructor

Animal**(**string s, int a, float l**):** name**(**s)**,** age**(**a), length**(**l)**{**

**}**

### Copy constructor

Copy constructor is made when an instance is set by another existing instance, this is a parameterized constructor, and the input type must be as same as its class type, copy constructors are made to prevent compiler to make the default copy, as the default copy is shallow copy.

Shallow is copying the old object data to the new made instance, the problem comes when the class have dynamically allocated data which in this case, one of the class attributes is a pointer to point to this allocated data, when shallow copy happens, the pointer of new object will be the old one so the old and new object have the same pointer !! which means no new object hasn’t allocated memory for itself see Figure 48 shallow vs deep copy

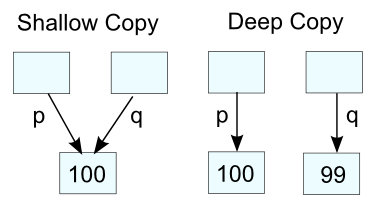


Figure 48 shallow vs deep copy

**Example**: copyConstructor

Make a copy constrctor to set tiger to be as same as tigress instance.

#include<iostream>

**using** **namespace** std**;**

class Animal**{**

public**:**

string name**;**

int age**;**

float length**;**

//Parameterized constructor

Animal**(**string s**):** name**(**s**){**

**}**

//copy constructor

Animal**(**const Animal**&** oldObject**){**

name **=** oldObject**.**name**;**

**}**

//function member

void sound**(**string sound**){**

cout**<<**name**<<**" have "**<<**sound**<<**" sound\n"**;**

**}**

**};**

int main**(){**

Animal tiger**(**"tigress"**);**

Animal newtiger **=** tiger**;**

newtiger**.**sound**(**"roar"**);**

**return** 0**;**

**}**

The output should be as same as last example

NOTE: don’t forget to in constructor to use const to prevent the old object to be modified unintentionally and take the old object by reference.

i.e. Animal**(**const Animal**&** oldObject**){}**

### Destructor

Destructor are called when the class gets out of the scope or deleted

**Example**: destructor

Make a destructor for animal class

#include<iostream>

**using** **namespace** std**;**

class Animal**{**

public**:**

string name**;**

int age**;**

float length**;**

//Parameterized constructor

Animal**(**string s**):** name**(**s**){**

**}**

//copy constructor

Animal**(**const Animal**&** oldObject**){**

name **=** oldObject**.**name**;**

**}**

//function member

void sound**(**string sound**){**

cout**<<**name**<<**" have "**<<**sound**<<**" sound\n"**;**

**}**

//destructor

**~**Animal**(){**

cout**<<**"destructor is called for instance: "**<<**name**<<**endl**;**

**}**

**};**

int main**(){**

Animal tiger**(**"tigress"**);**

tiger**.**sound**(**"roar"**);**

**return** 0**;**

**}**

Output:

*tigress have roar sound*

*destructor is called for instance: tigress*

## Inheritance

Inheritance comes when a child’s class (i.e. derived class) takes attributes and methods of parent class

### Base and derived classes

Base class is the parent class that a child class (derived class) will take its attributes and methods.

**Example**: inheritance

Drive tiger and lion child classes from animal class we made previously

#include<iostream>

**using** **namespace** std**;**

class Animal**{**

public**:**

string name**;**

int age**;**

float length**;**

//default constructor

Animal**(){**

cout**<<**"enter animal name "**;**

cin**>>**name**;**

**}**

//copy constructor

Animal**(**const Animal**&** oldObject**){**

name **=** oldObject**.**name**;**

**}**

//function member

void sound**(**string sound**){**

cout**<<**name**<<**" have "**<<**sound**<<**" sound\n"**;**

**}**

//destructor

**~**Animal**(){**

cout**<<**"destructor is called for instance: "**<<**name**<<**endl**;**

**}**

**};**

class Tiger**:** Animal**{**

**};**

class Lion**:** public Animal**{**

**};**

int main**(){**

Tiger teg**;**

Lion leo**;**

leo**.**sound**(**"growls"**);**

teg.sound("roar");

**return** 0**;}**

see the output in A screen shot of a computer code

Description automatically generatedFigure 49 default private inheritance

A screen shot of a computer code

Description automatically generatedFigure 49 default private inheritance

The problem is that teg (i.e. Tiger instance) is inherited private (by default), so public members are changes to private, so they are only accessed in Tiger class, so to remove the error we either comment the private member .sound() in main function teg.sound("roar");

Or to make the Tiger class inherit from Animal class as public like Lion class

class Tiger**:** public Animal**{};**

NOTE: Always remember, the class inheritance is private by default while the struct is public by default

### Types of inheritance

**Single inheritance**: when a child class inherit from one base (parent) class

**Multiple inheritance**: when a one child class inherit from many base (parent) class

**Hierarchical inheritance**: when many child classes inherit from one base (parent) class

**Multi-level inheritance**: : when a grandchild class inherit child class which inherited from one base (parent) class

**Hybrid inheritance**: : when multiple types are involved

See Figure 50 inheritance types

A diagram of a structure

Description automatically generated

Figure 50 inheritance types

### Constructor and destructor calls in inheritance

If the parent do something when initialized in constructor (e.g. print his class name which is Animal in our examples), when the child (e.g. Tiger) is initialized, sometimes we want to specialized the constructor for that child, so we have to override the parent class constructor.

**Example**: childConstructor

Override parent Animal class constructor in child class Tiger

#include<iostream>

**using** **namespace** std**;**

class Animal**{**

public**:**

string name**;**

int age**;**

float length**;**

//default constructor

Animal**(**string s**,** int a**,** float l**):**name**(**s**),**age**(**a**),**length**(**l**){**

cout**<<**"enter animal name "**;**

cin**>>**name**;**

**}**

//copy constructor

Animal**(**const Animal**&** oldObject**){**

name **=** oldObject**.**name**;**

**}**

//function member

void sound**(**string sound**){**

cout**<<**name**<<**" have "**<<**sound**<<**" sound\n"**;**

**}**

//destructor

**~**Animal**(){**

cout**<<**"destructor is called for instance: "**<<**name**<<**endl**;**

**}**

**};**

class Tiger**:** public Animal**{**

public**:**

Tiger**(**string s**,** int a**,** float l**):**Animal**(**s**,**a**,**l**){**

cout**<<**"Tiger "**<<**s**<<**" came\n"**;**

**}**

**};**

int main**(){**

Tiger teg**(**"teg"**,**2**,**2.5**);**

teg**.**sound**(**"roar"**);**

**return** 0**;}**

see the example output in Figure 51 inheritance example

Figure 51 inheritance example

## Encapsulation

Encapsulation is to hide the data from developers to prevent fatal mistakes, accessing and modifying members of class directly without any validation may ruin the code.

### Data hiding

make members private or protected and to access or modify these members use getters and setters.

e.g.

class Animal**{**

private**:**

string name**;**

int age**;**

float length**;**

…

…

};

### Setter and Getter (Accessor and mutator functions)

getters to get the member data and setter to set member data.

e.g.

class Animal**{**

private**:**

**…**

**…**

public**:**

**…**

**…**

//getters

string getName**(){**

**return** name**;**

**}**

int getAge**(){**

**return** age**;**

**}**

float getLength**(){**

**return** length**;**

**}**

//setter

void setName**(**string s**){**

name **=** s**;**

**}**

void setAge**(**int a**){**

age **=** a**;**

**}**

void setLength**(**float l**){**

length **=** l**;**

**}**

**…**

**…**

**};**

**Example**: encapsulation

Encapsulate the animal class and print Tiger data

#include<iostream>

**using** **namespace** std**;**

class Animal**{**

private**:**

string name**;**

int age**;**

float length**;**

public**:**

//constructor

Animal**(**string s**,** int a**,** float l**):**name**(**s**),**age**(**a**),**length**(**l**){**

**}**

//copy constructor

Animal**(**const Animal**&** oldObject**){**

name **=** oldObject**.**name**;**

**}**

//function member

void sound**(**string sound**){**

cout**<<**name**<<**" have "**<<**sound**<<**" sound\n"**;**

**}**

//getters

string getName**(){**

**return** name**;**

**}**

int getAge**(){**

**return** age**;**

**}**

float getLength**(){**

**return** length**;**

**}**

//setter

void setName**(**string s**){**

name **=** s**;**

**}**

void setAge**(**int a**){**

**if(**a**>**0**)**

age **=** a**;**

**}**

void setLength**(**float l**){**

**if(**l**>**0**)**

length **=** l**;**

**}**

//destructor

**~**Animal**(){**

cout**<<**"destructor is called for instance: "**<<**name**<<**endl**;}**

**};**

class Tiger**:** public Animal**{**

public**:**

Tiger**(**string s**,** int a**,** float l**):**Animal**(**s**,**a**,**l**){**

**}**

**};**

int main**(){**

Tiger teg**(**" "**,**0**,**0.0**);**//init empty

teg**.**setName**(**"teg"**);**

teg**.**setAge**(**2**);**

teg**.**setLength**(**1.5**);**

cout**<<**"Tiger '"**<<**teg**.**getName**()<<**"' has "**<<**teg**.**getAge**()<<**" yr(s) and "**<<**teg**.**getLength**()<<**" in length\n"**;**

**return** 0**;**

**}**

See the output in A black background with white text

Description automatically generatedFigure 52 encapsulation example:

A black background with white text

Description automatically generatedFigure 52 encapsulation example

## Polymorphism

What is child class what to override a member method? e.g. the method .sound() in parent class Animal is general , what if we want to build specific sound for each inherited child classes (i.e. lion child class growls , tiger child class, and so on), we can for each inherited child class override the base (parent) class.

NOTE: overriding is not overloading

Remember overload functions with different signatures (different return type, input parameters (number, type, order) but similar function names) within the same scope or class

While overriding is like overloading but in a different scope i.e. child class

See Figure 53 Overloading vs Overriding

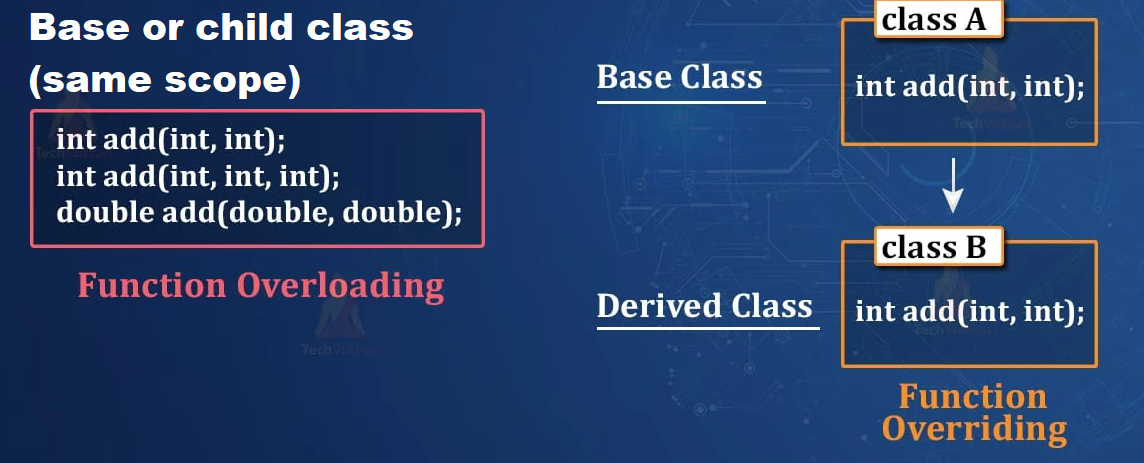


Figure 53 Overloading vs Overriding

NOTE: return type alone is insufficient for function to be overloading

### Compile-time polymorphism: function overloading, operator overloading

In base or derived class you can overload a method e.g.

#include <iostream>

**using** **namespace** std**;**

class Printer **{**

public**:**

// Overloaded functions

void print**(**int i**)** **{**

cout **<<** "Printing integer: " **<<** i **<<** endl**;**

**}**

void print**(**double d**)** **{**

cout **<<** "Printing double: " **<<** d **<<** endl**;**

**}**

void print**(**string s**)** **{**

cout **<<** "Printing string: " **<<** s **<<** endl**;**

**}**

**};**

int main**()** **{**

Printer p**;**

p**.**print**(**5**);** // Calls print(int)

p**.**print**(**3.14**);** // Calls print(double)

p**.**print**(**"Hello"**);** // Calls print(string)

**return** 0**;**

**}**

### Runtime polymorphism: virtual functions, pure virtual functions, abstract classes

You can override (change) function implementation of parent class to use it in child class (make the function member appropriate for child class) by using virtural keyword

#include <iostream>

**using** **namespace** std**;**

class Base **{**

public**:**

// Virtual function for overriding

virtual void show**()** **{**

cout **<<** "Base class show function" **<<** endl**;**

**}**

**};**

class Derived **:** public Base **{**

public**:**

// Overriding the base class function

void show**()** override **{**

cout **<<** "Derived class show function" **<<** endl**;**

**}**

**};**

int main**()** **{**

Base baseObj**;** // Create an object of Base

Derived derivedObj**;** // Create an object of Derived

// Call show() function for Base object

baseObj**.**show**();** // Outputs: Base class show function

// Call show() function for Derived object

derivedObj**.**show**();** // Outputs: Derived class show function

**return** 0**;**

**}**

## Abstraction

### Abstract classes and interfaces

Imagine a personal computer PC company that produces lots of PCs products like desktop PC and laptop,

the production of laptop in a section and production of desktop PC on other section, each section completed its section and pass it for branding section, the branding section doesn’t need to know how the laptop battery charging or how the desktop PC boots up,

the branding section only want to know the desktop PC and laptop both boots up, charging and so on, so we have to provide some sort of abstraction to make the life easier for branding section

in this case we could make like in polymorphism but not to provide any implementation for each member function and rely on implementation of function in the derived (child) classes

this is done by:

class Base **{**

public**:**

// Virtual function for overriding

virtual void show**()** **= 0;**

**};**

Instead of what we used in polymorphism:

class Base **{**

public**:**

// Virtual function for overriding

virtual void show**()** **{**

cout **<<** "Base class show function" **<<** endl**;**

**}**

**};**

### Virtual function and pure virtual function

this is pure virtual function, which used in abstraction

class Base **{**

public**:**

// Virtual function for overriding

virtual void show**()** **= 0;**

**};**

While this is virtual function which used in polymorphism

class Base **{**

public**:**

// Virtual function for overriding

virtual void show**()** **{**

cout **<<** "Base class show function" **<<** endl**;**

**}**

**};**

## Static Members

### Class-level data and behavior

Class level data is when a variable that all instances of classes share, e.g. if you want to keep track of number of instances made, so in class constructor, you can make static variable to increment every time an instance is made.

### Static member variables and functions

Imagine in Animal class we want to keep track of number of animals in zoo, we could make a static counter so that every time and animal come to the zoo, the constructor increment that counter.

**Example**: staticvariable

Count number of animals in the zoo

#include<iostream>

**using** **namespace** std**;**

class Animal**{**

private**:**

string name**;**

int age**;**

float length**;**

public**:**

static int animalCount**;**

//constructor

Animal**(**string s**,** int a**,** float l**):**name**(**s**),**age**(**a**),**length**(**l**){**

animalCount**++;**

**}**

…

…

//getters

…

…

//setter

…

…

//destructor

**~**Animal**(){**

cout**<<**"destructor is called for instance: "**<<**name**<<**endl**;**

**}**

**};**

// static member variables need to be declared inside

// the class definition and then defined outside of it

int Animal**::**animalCount **=** 0**;**

int main**(){**

Animal Tiger**(**"teg"**,** 2**,** 2.5**);**

Animal Lion**(**"leo"**,**1**,**2**);**

Animal Bird**(**"fly"**,** 1**,** 0.2**);**

cout**<<**"The zoo has "**<<**Animal**::**animalCount**<<**" animals\n"**;**

**return** 0**;}**

The output should be: *The zoo has 3 animals*

*destructor is called for instance: fly*

*destructor is called for instance: leo*

*destructor is called for instance: teg*

static methods are class level that also don’t need objects to be called, it could call without instantiating any object as its class level not instance(object) level

**Example**: staticMethod

Make a static method getAnimalCount() to show the animals count even if no object is made!   
#include<iostream>

**using** **namespace** std**;**

class Animal**{**

private**:**

string name**;**

int age**;**

float length**;**

public**:**

//static memver indicating for animals count

static int animalCount**;**

// Static member function to get animal count

static int getAnimalCount**()** **{**

cout**<<**"The zoo has "**<<**animalCount**<<**" animals\n"**;**

**return** animalCount**;**

**}**

//constructor

Animal**(**string s**,** int a**,** float l**):**name**(**s**),**age**(**a**),**length**(**l**){**

cout**<<**"Animal: "**<<**s**<<**" come\n"**;**

animalCount**++;**

**}**

…

//getters

…

//setter

…

//destructor

**~**Animal**(){**

cout**<<**"destructor is called for instance: "**<<**name**<<**endl**;**

**}**

**};**

// static member variables need to be declared inside

// the class definition and then defined outside of it

int Animal**::**animalCount **=** 0**;**

int main**(){**

//calling static method getAnimalCount() without any object made !

Animal**::**getAnimalCount**();**

Animal Tiger**(**"teg"**,** 2**,** 2.5**);**

Animal**::**getAnimalCount**();**

Animal Lion**(**"leo"**,**1**,**2**);**

Animal**::**getAnimalCount**();**

Animal Bird**(**"fly"**,** 1**,** 0.2**);**

Animal**::**getAnimalCount**();**

**return** 0**;**

**}**

See the output in Figure 54 Static method

A screen shot of a computer

Description automatically generated

Figure 54 Static method

## Multiple Inheritance

We knew previously that there are types of inheritance, a grandchild class that inherits from 2 parent class (multiple inheritance) then these 2 parent classes inherit from a parent class like Figure 55 hybrid inheritance case

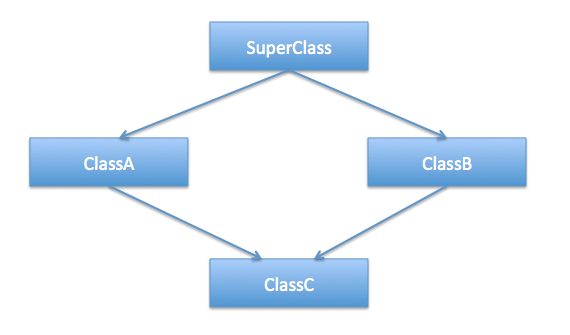


Figure 55 hybrid inheritance case

### Diamond problem and virtual inheritance

Imagine if the class named SuperClass has int x, while ClassA has int y while ClassB has int z

The ClassA will have x and y attributes

The ClassB will have x and z

So when ClassC inherits from ClassA and ClassB

The ClassC will have y and z attributes and also inherits the same x attribute twice

So when you try to access ClassC.x will throw an error because compiler doesn’t know which x attribute you want to access as ClassC have 2 of attribute x

#include <iostream>

**using** **namespace** std**;**

class SuperClass **{**

public**:**

int x**;**

**};**

class ClassA **:** public SuperClass **{**

public**:**

int y**;**

**};**

class ClassB **:** public SuperClass **{**

public**:**

int z**;**

**};**

class ClassC **:** public ClassA**,** public ClassB **{**

// ClassC has both y and z, but x is ambiguous

**};**

int main**()** **{**

ClassC obj**;**

obj**.**x **=** 15**;** //error: x is ambiguous!

**return** 0**;**

**}**

**Example**: diamondProblem

Solve the problem of diamond problem by inheriting ClassA and ClassB from SuberClass in virtual inheritance mode.

#include <iostream>

**using** **namespace** std**;**

class SuperClass **{**

public**:**

int x**;**

**};**

class ClassA **:**virtual public SuperClass **{**

public**:**

int y**;**

**};**

class ClassB **:** virtual public SuperClass **{**

public**:**

int z**;**

**};**

class ClassC **:** public ClassA**,** public ClassB **{**

// ClassC has both y and z, but x is ambiguous

**};**

int main**()** **{**

ClassC obj**;**

obj**.**x **=** 15**;** //error: x is ambiguous!

**return** 0**;**

**}**

## Operator overloading

Compiler know how to process operators well, e.g. it knows what how to add integer, floats, even strings, when it comes to user-defined data types, it becomes pretty hard , the complier don’t know your intention if an instance of class (e.g. Person) is added together, the compiler don’t know what do you mean of Person1+Person2, so you must tell the compiler what does this addition mean (e.g. the addition for the 2 person class might be adding their salaries).

Firstly, lets know what unary and binary operator are (see Figure 56 unary and binary operators (the photo from (geek for geek)), unary operators have one operand while binary operators have 2 operands .

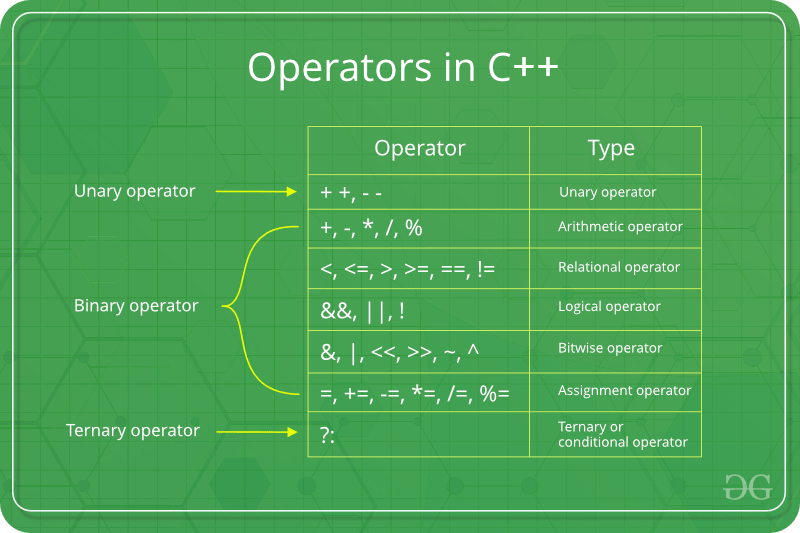


Figure 56 unary and binary operators (the photo from (geek for geek)

### Overloading unary operators

**Example**: unaryOverloading

#include<iostream>

**using** **namespace** std**;**

class Animal**{**

private**:**

string name**;**

int age**;**

float length**;**

public**:**

//constructor

Animal**(**string s**,** int a**,** float l**):**name**(**s**),**age**(**a**),**length**(**l**){**

**}**

//copy constructor

Animal**(**const Animal**&** oldObject**){**

name **=** oldObject**.**name**;**

**}**

//function member

void sound**(**string sound**){**

cout**<<**name**<<**" have "**<<**sound**<<**" sound\n"**;**

**}**

//getters

…

//setter

…

//destructor

**~**Animal**(){**

cout**<<**"destructor is called for instance: "**<<**name**<<**endl**;**

**}**

friend Animal**&** **operator++(**Animal**&** obj**);**

**};**

Animal**&** **operator++(**Animal**&** obj**){**

obj**.**age**++;**

**return** obj**;**

**}**

int main**(){**

Animal Tiger**(**"Teg"**,**2**,**1.5**);**

cout**<<**"Tiger age before increament "**<<**Tiger**.**getAge**()<<**endl**;**

**++**Tiger**;**

cout**<<**"Tiger age after increament "**<<**Tiger**.**getAge**()<<**endl**;**

**return** 0**;**

**}**

### Overloading binary operators

**Example**: binaryOverloading

Make a binary operator overloading + to add 2 Animal length of class we made, to make sure that a two animal could be lifted on a car that has a length of 4 meter

#include<iostream>

**using** **namespace** std**;**

#define cageLength 4

class Animal**{**

private**:**

string name**;**

int age**;**

float length**;**

public**:**

//constructors

…

//function member

void sound**(**string sound**){**

cout**<<**name**<<**" have "**<<**sound**<<**" sound\n"**;**

**}**

//getters

…

//setter

…

//destructor

…

friend float **operator+(**Animal**&** obj1**,** Animal**&** obj2**);**

**};**

float **operator+(**Animal**&** obj1**,** Animal**&** obj2**){**

**return** obj1**.**length **+** obj2**.**length**;**

**}**

int main**(){**

Animal Tiger**(**"Teg"**,**2**,**1.5**);**

Animal Liger**(**"Leg"**,**2**,**2**);**

**if(**Tiger**+**Liger**<**cageLength**){**

cout**<<**"they could fit in the cage\n"**;**

**}**

**else{**

cout**<<**"they couldn't fit in the cage\n"**;**

**}**

**return** 0**;**

**}**

REMEMBER: two classes by default couldn’t have + operator by default, that’s why we make + operator overloading, also you can overload any operator

### Insertion and extraction overloading

if we tried to cout the instance of a class, an error would pop up as insertion and extraction (i.e << in cout OR >> in cin) don’t have a definition for printing a user defined instance, so to print an instance, we have to define what will happen when we use cout<<instanceClass for example

**Example**: insertionOverload

Print the Animal instance name and age and length using cout<<Tiger;

#include<iostream>

**using** **namespace** std**;**

#define cageLength 4

class Animal**{**

private**:**

string name**;**

int age**;**

float length**;**

public**:**

//constructors

…

//getters

…

//destructor

…

friend ostream**&** **operator<<(**ostream**&** out**,** Animal**&** obj**);**

**};**

ostream**&** **operator<<(**ostream**&** out**,** Animal**&** obj**){**

out**<<**"Animal name: "**<<**obj**.**name**<<**" has "**<<**obj**.**age**;**

out**<<**" yrs and "**<<**obj**.**length**<<**" in length\n"**;**

**return** out**;**

**}**

int main**(){**

Animal Tiger**(**"Teg"**,**2**,**1.5**);**

**return** 0**;**

**}**

NOTE: cout**<<**Tiger**;** has no meaning by default so we overload it to define what happens when we use cout<<Tiger or any instance

## Rules

### Rule of Three

The "Rule of Three" in C++ is a guideline that helps manage resources properly and avoid common pitfalls related to resource management, especially when dealing with classes that allocate dynamic memory or other resources.

When a class is built, make three things:

* Copy Constructor (discussed in constructor section)
  + To prevent shallow copy by using deep copy
  + Used to create a new object as a copy of an existing object.
* Destructor (discussed in destructor section)
  + Used to release resources that were acquired by the object.
* Copy Assignment operator
  + Used to copy the contents of one object to an already-existing object

## Final Project

Tic-tac-toe game in console will be made with OOP flavored, the code may have:

**Inheritance**

* **Base Class**: Player is the base class.
* **Derived Classes**: HumanPlayer and ComputerPlayer are derived from Player

**Abstraction**

* **Play()**: virtual play()= 0;

**Encapsulation**

* **Maybe not necessary as no data to hide so no need for getters and setters**

**Polymorphism**

* **Add polymorphism is possible**

**MODEL answer fount at** [LINK](https://github.com/MuOssama/MasteringCPP/tree/main/Object%20Oriented%20Programming/Final%20Project) **or Scan QR code**

A qr code on a white background

Description automatically generated

# Templates

In C++, class members and function inputs and return must have **determined** datatypes as we introduced throughout this book, but what to do if we want to make these class members and function inputs and return varies with user use? In function we could overload like in function chapter, section overloading we used to make this overloading example

RECALL **Example**: overloading

Use overloading to make add() in addFunc example handles both integers and floats

#include<iostream>

**using** **namespace** std**;**

double add**(**int x**,** int y**);** //function Declaration

double add**(**float x**,** float y**);** //function Declaration

int main**(){**

float a**,**b**;**

double sum **=** 0**;**

cout**<<**"enter the two addition operands :"**;**

cin**>>**a**>>**b**;**

sum **=** add**(**a**,**b**);**

cout**<<**"\n the sum is "**<<**sum**;**

**}**

double add**(**int x**,** int y**){** //function Definition

**return** x **+** y**;**

**}**

double add**(**float x**,** float y**){** //function Declaration

**return** x **+** y**;**

**}**

This earlier example permit us to use add() function with different datatypes (floats and int), we can do more easier way using templates, not only for function inputs and return, but also for class members !!

## Function templates

like we said, the templates permit us not be constrained with the determined datatypes like the following:

//the return datatype will be treated as int

//the inputs datatype will be treated as floats

int add **(**float x**,** float y**);**

//the return datatype will be treated as int

//the inputs datatype will be treated as int

int mult **(**int x**,** int y**);**

could we make these functions make the datatypes of inputs and return datatypes variable???

Like:

variableDatatype mult **(**variableDatatype x**,** variableDatatype y**);**

variableDatatype add **(**variableDatatype x**,** variableDatatype y**);**

Of course we could without overloading !!

### Template syntax and usage

Declaration

template <typename name>

OR, template<class name>

e.g.

template **<**typename T1**,** typename T2**,** typename T3**>**

T1 mult**(**T2 x**,** T3 y**)** **{**

**return** x**\***y**;**

**}**

This permit the developer to use this function with different inputs and return datatypes

**Example**: funcTemplates

We previously overloaded add function to use different datatypes input, make this with templates, let the inputs are same in datatypes (make one template for input not 2), use the add function to return float with int inputs, and another use with float return and double inputs.

#include<stdio.h>

**using** **namespace** std**;**

template**<**typename R**,** typename T**>**

R add**(**T x**,** T y**){**

**return** x**+**y**;**

**}**

int main**(){**

float result **=** 0**;**

int in1**=**5**,**in2**=**2**;**

//we must determine the return as float

//as compiler cannot deduce by itself

result **=** add**<**float**>(**in1**,**in2**);**

printf**(**"%f\n"**,** result**);** //output is float 7.000

double l**=**2.2**,** m**=**4.8**;**

result **=** add**<**float**>(**l**,**m**);**

printf**(**"%f\n"**,** result**);** //output is float 7.000

**return** 0**;**

**}**

NOTE !!

What happens if we replaced the second result call with:

result **=** add**<**float**>(**in1**,**m**);**

**this will throw error as we told compiler that both function inputs have the same data types in function prototype:**

R add**(**T x**,** T y**)**

To solve this problem, we have to make another template for the second input like

template**<**typename R**,** typename T, typename U**>**

R add**(**T x**,** U y**){**

**return** x**+**y**;**

**}**

### Specialization of function templates

Templates permit developers to use the functions without the inputs and return datatypes constrains which give us generalization of the function, for example, if we implement a function to add to integers, floats, or string, we may specialize the function in this example maybe if the inputs are string the function print their addition (string concatenation) and print “this is string concatenation”, any type of inputs, the function print the addition only, so what we want exactly is to provide function specialization for string.

**Example**: funcTempSpecialtization

Make a template function to add 2 inputs but if inputs are string, the function print their addition and print “this is string concatenation”

#include<iostream>

**using** **namespace** std**;**

template**<**typename R**,** typename T**>**

R add**(**T x**,** T y**){**

cout**<<**x**+**y**<<**endl**;**

**return** x**+**y**;**

**}**

template**<>** //specialization

string add**(**string x**,** string y**){**

cout**<<**"this is string concatenation\n"**;**

cout**<<**x**+**y**<<**endl**;**

**return** x**+**y**;**

**}**

int main**(){**

float result **=** 0**;**

int in1**=**5**,**in2**=**2**;**

result **=** add**<**float**>(**in1**,**in2**);**

string x**=**"good"**,**y**=**"night"**,**out**=**""**;**

out **=** add**<**string**>(**x**,**y**);**

**return** 0**;**

**}**

We specialize the case of the inputs are strings this is full template specialization, partial template specialization is when specialize some (not all) the parameters, but C++ don’t allow partial specialization except in structs and classes

See the output Figure 57 function template specialization example output

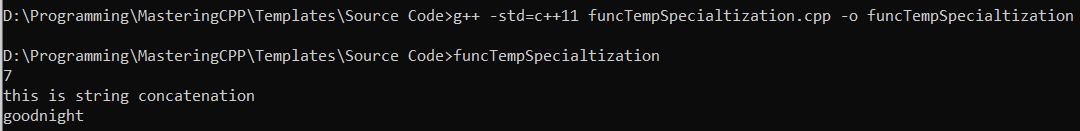


Figure 57 function template specialization example output

## Class Templates

Like in function templates, the class templates generalize the class parameters, this makes the parameter datatypes of class free and could anything not fixed like

class Animal**{**

string name**;**

int age**;**

**};**

### Template classes

The datatype of Name is string and the datatype cannot be changed

The datatype of age is int and the datatype cannot be changed (e.g to be float)

But using templates, we can use age as float for example and name as integer

template**<**typename T**,** typename U**>**

class Animal**{**

T name**;**

U age**;**

**};**

The datatype of name and age are generic could be string and float or int and int or anything !!

**Example**: classTemplates

Write class templates to use to make its parameters name and age generic and could be used for different datatypes

#include<iostream>

**using** **namespace** std**;**

template**<**typename T**,** typename U**>**

class Animal**{**

T name**;**

U age**;**

public**:**

//geters and setters for encapsulation

T getName**(){return** name**;}**

U getAge**(){return** age**;}**

void setName**(**T n**){**name **=** n**;}**

void setAge**(**U a**){**age **=** a**;}**

**};**

int main**(){**

Animal**<**string**,** float**>** Tiger**;**

Animal**<**int**,** int**>** Lion**;**

Tiger**.**setAge**(**1.5**);**

cout**<<**Tiger**.**getAge**()<<**endl**;**

Lion**.**setAge**(**1**);**

cout**<<**Lion**.**getAge**();**

**}**

### Specialization of class templates

Like in function templates specialization, we could provide specialization for classes.

In this class:

class Animal**{**

string name**;**

int age**;**

**};**

We could specialize all parameters (i.e. name and age) which means full specialization

We could specialize some parameters (i.e. name only) which means partial specialization (functions cannot partially specialized)

**Example**: fullClassSpecialization

#include<iostream>

**using** **namespace** std**;**

template**<**typename T**,** typename U**>**

class Animal**{**

T name**;**

U age**;**

public**:**

//constructor

Animal**(){**cout**<<**"datatypes: all\n"**;}**

//geters and setters for encapsulation

T getName**(){return** name**;}**

U getAge**(){return** age**;}**

void setName**(**T n**){**name **=** n**;}**

void setAge**(**U a**){**age **=** a**;}**

**};**

//full spectialization when T is string and U is float

template**<>**

class Animal**<**string**,** float**>{**

string name**;**

float age**;**

public**:**

//constructor

Animal**(){**cout**<<**"datatypes: string and float\n"**;}**

//geters and setters for encapsulation

string getName**(){return** name**;}**

float getAge**(){return** age**;}**

void setName**(**string n**){**name **=** n**;}**

void setAge**(**float a**){**age **=** a**;}**

**};**

int main**(){**

Animal**<**string**,** float**>** Tiger**;**

Animal**<**int**,** int**>** Lion**;**

**}**

The out should be:

datatypes: string and float

datatypes: all

which is the constructor calls

**Example**: partialClassSpecialization

Make the name in animal class 0 unless the user passes a string, this could be made by using the default constructor to set the to Null and specialization for string entry to set the name to a string, we only interested in name parameter not so we will make it partial specialization.

#include<iostream>

**using** **namespace** std**;**

template**<**typename T**,** typename U**>**

class Animal**{**

T name**;**

U age**;**

public**:**

//constructor

Animal**(){**name **=** 0**;**cout**<<**"name not a string\n"**;}**

//geters and setters for encapsulation

T getName**(){return** name**;}**

U getAge**(){return** age**;}**

void setName**(**T n**){**name **=** n**;}**

void setAge**(**U a**){**age **=** a**;}**

**};**

//partial spectialization when T is string and U is float

template**<**typename U**>**

class Animal**<**string**,** U**>{**

string name**;**

U age**;**

public**:**

//constructor

Animal**(){**cout**<<**"name is string\n"**;}**

//geters and setters for encapsulation

string getName**(){return** name**;}**

U getAge**(){return** age**;}**

void setName**(**string n**){**name **=** n**;}**

void setAge**(**U a**){**age **=** a**;}**

**};**

int main**(){**

Animal**<**string**,** float**>** Tiger**;**

Animal**<**int**,** int**>** Lion**;**

**}**

The output should be:

name is string *//as Tiger has string name*

name not a string *//as Lion has int name not string name*

# Standard Template Library (STL)

# Exception Handling

Just imagine calculator application, and the user divide by 0, the program will crash, if a text reader application try to read a text file that does not exists, also the application will crash, could we handle any error to prevent the program, to crash and just pop up message showing the error or something? Of course we could !

Error handling in CPP contains:

* try(something){instruction}

this will try something (e.g. dividing or opening a file), if that something finished will, the program will execute some instructions and if failed may throw an error

* throw msg

if that something failed throw a message, the message could be int or float or string etc

* catch(msg){instruction}

if an message thrown, catch will get it and execute some instruction.

For summary see Figure 58 try throw catch

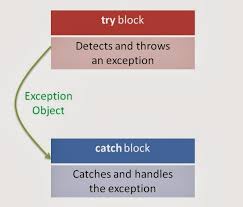


Figure 58 try throw catch

**Example**: calculator

Make simple arithmetic calculator that do these operations + - / \* the app will crash if these cases if num2 is 0: num1 / num2 and num1%num2

# File I/O

# Multithreading and Concurrency

# GDB Debugger

# Others

# Modern C++