CC211L

Object Oriented Programming

Laboratory 02

Introduction to Classes Objects and Member Functions I

Version: 1.0.0

Release Date: 03012023

Department of Information Technology

University of the Punjab

Lahore, Pakistan

Contents:

Learning Objectives:

- Concept of Encapsulation and Abstraction
- Getter/Setter Member Functions
- Public private access specifiers
- Constructors
- Overloaded Constructor
- Default Constructor
- Destructor

Resources Required:

- Desktop Computer or Laptop
- Microsoft ® Visual Studio 2022

General Instructions:

- In this Lab, you are **NOT** allowed to discuss your solution with your colleagues, even not allowed to ask how is s/he doing, this may result in negative marking. You can **ONLY** discuss with your Teaching Assistants (TAs) or Lab Instructor.
- Your TAs will be available in the Lab for your help. Alternatively, you can send your queries via email to one of the followings.

Teachers:		
Course Instructor	Prof. Dr. Syed Waqar ul Qounain	swjaffry@pucit.edu.pk
Teacher Assistants	Syed M.Jafar Ali Naqvi	bitf21m032@pucit.edu.pk
	M. Usman Munir	bitf19m020@pucit.edu.pk
	Aqdas Shabir	bitf21m022@pucit.edu.pk
	Ibad Nazir	bitf21m024@pucit.edu.pk
	Umer Farooq	bitf21m010@pucit.edu.pk

Background and Overview:

Abstraction:

Abstraction in C++ is the process of hiding the implementation details from the user. It allows the user to focus on the functionality of an object without worrying about its internal implementation. Abstraction is achieved in C++ through the use of classes and objects, which can be used to define data and functions that can be used to manipulate the data. By hiding the implementation details, the user is able to use the object without having to worry about its internal working. This helps make code easier to understand and maintain.

Encapsulation:

Encapsulation in C++ is the process of combining data and functions into a single unit, called a class. This process allows data and functions to be accessed and used together while keeping the data hidden from outside users. Encapsulation is a powerful concept that helps keep code organized, secure, and manageable. By using encapsulation, developers can define the interactions between different parts of a program in a clear and concise way. This helps to reduce potential errors and maintain the program's functionality over time.

Class:

A class is an important concept in Object Oriented Programming (OOP). It is a blueprint that is used to create objects, which can hold data and perform functions. A class is made up of two parts: attributes and methods. Attributes are the properties of the class that describe its characteristics, such as its name, size, and color. Methods are the functions that the class can perform, such as adding, deleting, or editing data. Classes are reusable and allow for code to be written more efficiently. They are also a key tool for creating complex applications.

Object:

Objects are instances of classes, which are templates for creating objects. Objects have properties and methods that define their state and behavior, respectively. It is a selfcontained entity that contains both data and instructions for manipulating that data. Objects are used in objectoriented programming (OOP) to represent realworld entities such as people, places, and things. Objects contain data and methods that describe the object's characteristics and behavior. The data is stored in the form of properties, while the methods are functions that can access and modify the object's data.

Comparison Between class and object:

Objects are often referred to as the "nouns" of programming, while classes are the "verbs". Objects are tangible, physical objects in the real world, while classes are abstract concepts that define how objects should behave.

Activities:

PreLab Activities:

Getter/Setter Member Functions:

Getter/Setter member functions are used to control access to class data members in C++. Getters are used to retrieve data from the class, and setters are used to assign values to the class. They allow for the encapsulation of data within a class, which makes it easier to manage class data and ensures that data is used correctly.

Example:

For example, if a class has a data member called "age", it is best to create a getter and setter function for that data member. The getter function would return the age of the class object, and the setter function would allow us to set the age of the class object. This way, we can ensure that the data member is being properly used and make sure it is not being passed invalid data.

Syntax of Getters and Setters in C++:

Getter:

The syntax of Getter method is as follows

```
Type ClassName::getPropertyName(){
   return propertyName;
}
```

Setter:

The syntax of Setter method is as follows

```
void ClassName::setPropertyName(Type propertyName){
   this>propertyName = propertyName;
}
```

Example Code:

```cpp
#include <iostream>

using namespace std;

class Student
{
private:
    string name;

public:
    void setName(string name)
    {
        this->name = name;
    }

    string getName()
    {
        return name;
    }
};

// Driver code
int main()
{
    Student student;

    // Set name
    student.setName("John Doe");

    // Get name
    cout << "Name: " << student.getName() << endl;

    return 0;
}
```

Fig. 01 (Getter and Setter Demonstration)

Output:

Fig. 02 (Output with respect to figure 1)

Some Key points about getter and setter methods:

- Getters and setters are used to access private member variables of a class from outside the class.
- Getters and setters should be used to ensure data integrity in the class by validating the input data and restricting access to certain areas of the class.
- Getters and setters should not be used to perform business logic or complex operations on private member variables.
- Getters should always return a copy of the private member variable and not the variable itself.
- Setters should always be used to modify the private member variable and not perform any other operations on the variable.
- Getters and setters should have meaningful names that clearly describe the purpose of the methods.
- Getters and setters should be declared as public methods of the class.

When to Use Getters and Setters in C++:

Getters and setters should be used when you want to maintain control over how users access and set the data members of a class. This allows you to control how the data is accessed and can be used to validate user input before setting a value or to perform calculations based on user input before setting a value. Getters and setters also make it easier to maintain the code, as the logic for how the data is accessed and set is selfcontained.

Public/Private Access Specifiers:

Access Specifiers:

Access specifiers in C++ can be used to control the access level of the data members of a class. The three access specifiers used in C++ are public, protected, and private. In this introduction, we will discuss the purpose of public and private access specifiers and how to use them in C++.

Public Access Specifiers:

Public data members can be accessed and modified from anywhere within the program. To make a member of a class public, use the keyword “public” before the member declaration.

Private Access Specifiers:

Private data members can only be accessed and modified by the member functions of the class. They cannot be accessed or modified from outside the class. To make a member of a class private, use the keyword “private” before the member declaration.

How to Use Public and Private Access Specifiers:

Example Code for Public and Private Access Specifiers:

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

// Public access specifier
class PublicClass
{
public:
    int publicVariable;
    void setPublicVariable(int variableValue)
    {
        publicVariable = variableValue;
    }
    int getPublicVariable()
    {
        return publicVariable;
    }
};

// Private access specifier
class PrivateClass
{
private:
    int privateVariable;
    void setPrivateVariable(int variableValue)
    {
        privateVariable = variableValue;
    }
    int getPrivateVariable()
    {
        return privateVariable;
    }
};

int main()
{
    PublicClass publicClassObject;
    publicClassObject.setPublicVariable(5);
    cout << publicClassObject.getPublicVariable() << endl;

    PrivateClass privateClassObject;
    // privateClassObject.setPrivateVariable(10); // Error: setPrivateVariable is private
    // cout << privateClassObject.getPrivateVariable() << endl; // Error: getPrivateVariable is private
    return 0;
}
```

Fig. 03 (Public and Private Access Specifiers)

Output:

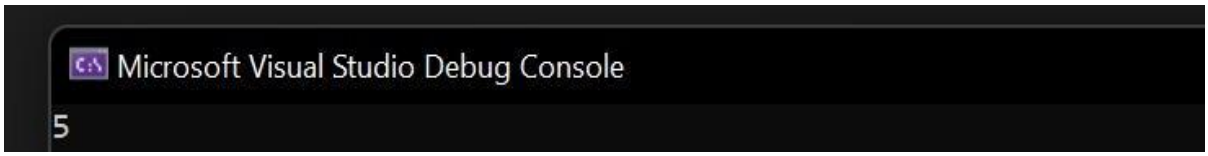


Fig. 04 (Output for figure 3)

Demonstration of Encapsulation and Abstraction using Getter/Setter functions

The demonstration of the encapsulation and abstraction using getter and setter functions is given in the code below:

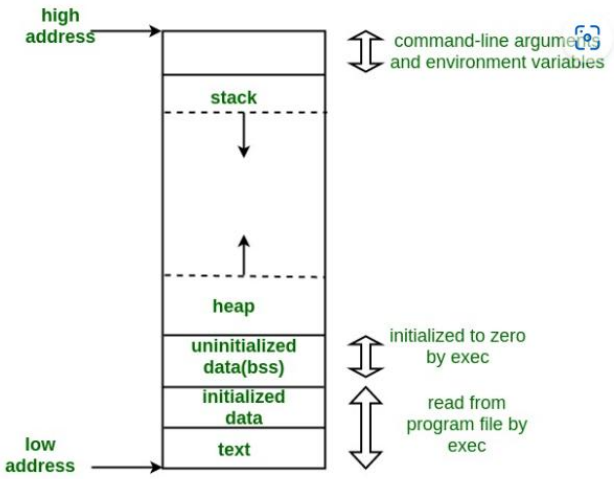
```cpp
#include<iostream>
using namespace std;

// Class Student
class Student
{
private:
    string name;
    int age;

public:
    // Getter functions
    string getName()
    {
        return name;
    }
    int getAge()
    {
        return age;
    }

    // Setter functions
    void setName(string x)
    {
        name = x;
    }
    void setAge(int y)
    {
        age = y;
    }
};

int main()
{
    Student student1;
    student1.setName("John");
    student1.setAge(15);

    cout << "Name of student1: " << student1.getName() << endl;
    cout << "Age of student1: " << student1.getAge() << endl;

    return 0;
}
```

Fig. 05 (Demonstration of Encapsulation and Abstraction using Getter/Setter functions)

Output:

```
Microsoft Visual Studio Debug Console
Name of student1: John
Age of student1: 15
```

Fig. 06 (Output for figure 5)



Writing effective code requires an awareness of stack and heap memory, making it a crucial component of learning programming. Not only that, but new programmers should also become fully acquainted with the distinctions between stack memory and heap memory in order to write effective and optimized code. This blog post will provide a comprehensive comparison of these two memory allocation techniques. We will have a thorough understanding of stack and heap memory by the conclusion of this article, allowing us to employ them effectively in our programming endeavors.

Memory allocation

Memory serves as the foundation of computer programming. It provides the space to store the data and all the commands our program requires to operate efficiently. Allocating memory can be compared to designating a specific area within our computer's memory for a distinct purpose, like accommodating variables or objects essential for our program's functionality. Memory layout and organization of a program can vary according to the operating system and architecture being used. In general, however, memory may be divided into the following segments:

- Global segment
- Code segment
- Stack
- Heap

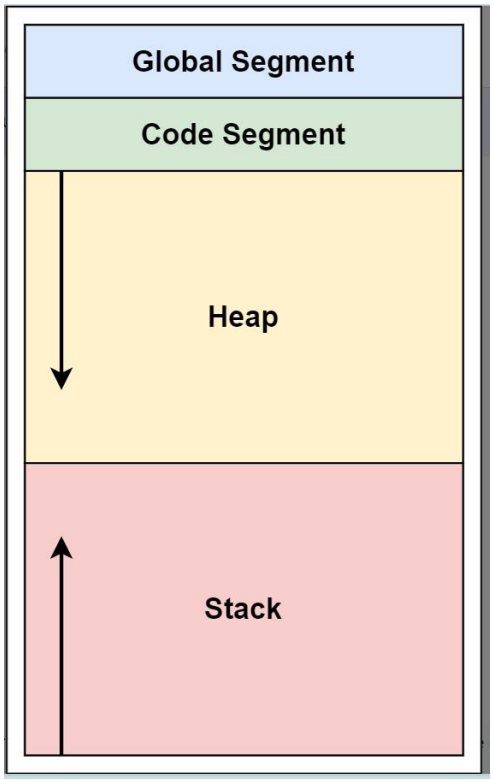
The global segment is responsible for storing global variables and static variables that have a lifetime equal to the entire duration of the program's execution.

The code segment, also known as the text segment, contains the actual machine code or instructions that make up our program, including functions and methods.

The stack segment is used for managing local variables, function arguments, and control information, such as return addresses.

The heap segment provides a flexible area for storing large data structures and objects with dynamic lifetimes. Heap memory may be allocated or deallocated during the program execution.

Note: *It is important to note that stack and heap in the context of memory allocation should not be confused with the data structures stack and heap, which have different purposes and functionalities.*



Every program has its own virtual memory layout, which is mapped onto physical memory by the operating system. The specific allocation to each segment depends on various factors, such as the following:

- Size of the program's code.
- Number and size of global variables.
- Amount of dynamic memory allocation required by the program.
- Size of the call stack used by the program.

The global variables declared outside of any function would reside in the global segment. The machine code or instructions for the program's functions and methods would be stored in the code segment. Let us see coding examples to help visualize how the global and code segments are used in memory:

```cpp
#include <iostream>

// Global Segment: Global variables are stored here
int globalVar = 42;

// Code Segment: Functions and methods are stored here
int add(int a, int b) {
    return a + b;
}

int main() {
    // Code Segment: Calling the add function
    int sum = add(globalVar, 10);

    std::cout << "Sum: " << sum << std::endl;
    return 0;
}
```

In these code examples, we have a global variable globalVar with a value of 42, which is stored in the global segment. We also have a function add, which takes two integer arguments and returns their sum; this function is stored in the code segment.
The main function (or the script in Python) calls the add function, passing the global variable and another integer value 10 as arguments.

Global Segment	Variable and its value: `globalVar = 42`
Code Segment	Complete function definition: `add(a, b)`

It's crucial to emphasize that managing the stack and heap segments plays a significant role in the performance and efficiency of our code, making it a vital aspect of programming. Therefore, programmers should understand them fully before delving into their differences.

Stack memory: the orderly storage

Think of stack memory as an organized and efficient storage unit. It uses a last in, first out (LIFO) approach, which means that the most recent data added gets removed first. The kernel, a central component of the operating system, manages stack memory automatically; we don't have to worry about allocating and deallocating memory. It just takes care of itself while our program runs.

The code instances below in different programming languages demonstrate the use of stack in various cases.

```cpp
#include <iostream>

// A simple function to add two numbers
int add(int a, int b) {
    // Local variables (stored in the stack)
    int sum = a + b;
    return sum;
}

int main() {
    // Local variable (stored in the stack)
    int x = 5;

    // Function call (stored in the stack)
    int result = add(x, 10);

    std::cout << "Result: " << result << std::endl;

    return 0;
}
```

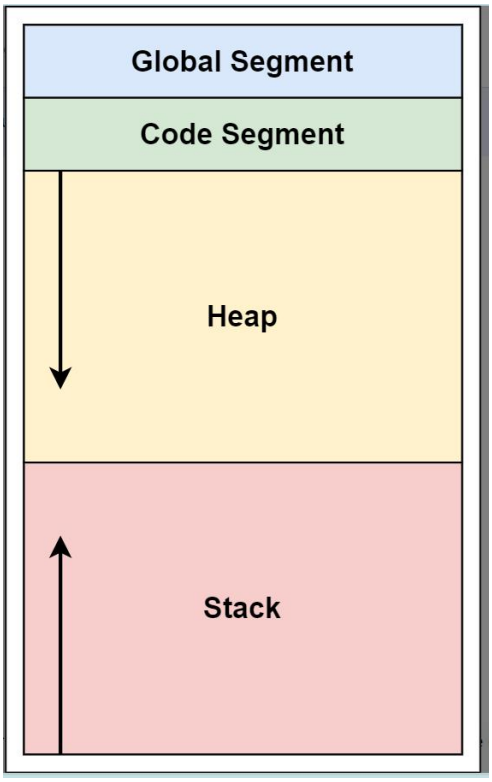
A block of memory called a stack frame is created when a function is called. The stack frame stores information related to local variables, parameters, and the return address of the function. This memory is created on the stack segment.

In the codes instances above, we created a function called add. This function takes two parameters as input integers and returns their sum. Inside the add function, we created a local variable called sum to store the result. This variable is stored in stack memory.

In the main function (or top-level script for Python), we create another local variable x and assign it the value 5. This variable is also stored in stack memory. Then, we call the add function with x and 10 as arguments. The function call and its arguments and return address are placed on the stack. Once the add function returns, the stack is popped, removing the function call and associated data, and we can print the result.

In the following explanation, we'll go over how the heap and stack change after running each important line of code. Although we're focusing on C++, the explanation for Python and Java also holds. We're only discussing the stack segment here.

```cpp
#include <iostream>

// A simple function to add two numbers
int add(int a, int b) {
    // Local variables (stored in the stack)
    int sum = a + b;
    return sum;
}

int main() {
    // Local variable (stored in the stack)
    int x = 5;

    // Function call (stored in the stack)
    int result = add(x, 10);

    std::cout << "Result: " << result << std::endl;

    return 0;
}
```



Here is the explanation of the C++ code in the order of execution:

Line 3: The function main is called, and a new stack frame is created for it.

Line 5: A local variable value on the stack frame is assigned the value 42.

Line 8: A pointer variable ptr is allocated the dynamically created memory for a single integer on the heap using the new keyword. Let's assume the address of this new memory on the heap to be 0x1000. The address of the allocated heap memory (0x1000) is stored in the pointer. ptr.

Line 11: The integer value 42 is assigned to the memory location pointed to by ptr (heap address 0x1000).

Line 12: The value stored at the memory location pointed to by ptr (42) is printed to the console.

Line 15: The memory allocated on the heap at address 0x1000 is deallocated using the delete keyword. After this line, ptr becomes a dangling pointer because it still holds the address 0x1000, but that memory has been deallocated. However, we will not discuss dangling pointers in detail for this essential discussion.

Line 17: The main function returns 0, signaling successful execution.

Line 18: The main function's stack frame is popped from the stack, and all local variables (value and ptr) are deallocated.

Key features of heap memory

Here are some notable characteristics of heap memory to keep in mind:

Flexibility in size: Heap memory size can change throughout the program's execution.

Speed trade-off: Allocating and deallocating memory in a heap is slower because it involves finding a suitable memory frame and handling fragmentation.

Storage for dynamic objects: Heap memory stores objects and data structures with dynamic lifespans, like those created with the new keyword in Java or C++.

Persistent data: Data stored in heap memory stays there until we manually deallocate it or the program ends.

Manual management: In some programming languages (for example, C and C++), heap memory must be managed manually. This can lead to memory leaks or inefficient use of resources if it's not done correctly.

Stack vs. heap: highlighting the differences

Now that we thoroughly understand how stack and heap memory allocations work, we can distinguish between them. When comparing stack and heap memory, we must consider their unique characteristics to understand their differences:

Size management: Stack memory has a fixed size determined at the beginning of the program's execution, while heap memory is flexible and can change throughout the program's lifecycle.

Speed: Stack memory offers a speed advantage when allocating and deallocating memory because it only requires adjusting a reference. In contrast, heap memory operations are slower due to the need to locate suitable memory frames and manage fragmentation.

Storage purposes: Stack memory is designated for control information (such as function calls and return addresses), local variables, and function arguments, including return addresses. On the other hand, heap memory is used for storing objects and data structures with dynamic lifespans, such as those created with the new keyword in Java or C++.

Data accessibility: Data in stack memory can only be accessed during an active function call, whereas data in heap memory remains accessible until it's manually deallocated or the program ends.

Memory management: The system automatically manages stack memory, optimizing its usage for fast and efficient memory referencing. In contrast, heap memory management is the programmer's responsibility, and improper handling can lead to memory leaks or inefficient use of resources.

This table summarizes the key differences between stack and heap memory across different aspects:

Aspect	Stack Memory	Heap Memory
Size management	Fixed size, determined at the beginning of program	Flexible size, can change during program's lifecycle
Speed	Faster, only requires adjusting a reference	Slower, involves locating suitable blocks and managing fragmentation
Storage purposes	Control info, local variables, function arguments	Objects and data structures with dynamic lifespans
Data accessibility	Accessible only during an active function call	Accessible until manually deallocated or program ends
Memory management	Automatically managed by the system	Manually managed by the programmer

Task 01: Little Humpty's Pharmacy **[Estimated 50 minutes / 40 marks]**

Little Humpty wants to start a new business. He has heard there is great scope in Pharmacy stores business. He wants to have Computerized system to handle all the medicines in a pharmacy. Being your friend he asks you for help and lucky for Humpty, you are a great programmer. Develop a system for Humpty's Pharmacy. He wants to have following info about medicines:

- Medicine id
- Medicine name
- Manufacturer
- Sail Person he bought medicine from.
- Price at which he bought
- Price to sell at
- Quantity

Humpty says he does not know how much medicine he will have but he will add them as we go along. He wants a nice little menu based system where he can add new medicine, watch the data for a medicine, watch the data of all the medicine, watch all the medicine he bought of a particular manufacturer, watch all the medicine he bought from a particular Sail Person, Total Quantity of each medicine he has at store. Humpty is a friend and not very smart, so provide him a nice userfriendly program, which will display userfriendly messages if Humpty makes any mistake. Advance Thanks from little Humpty!

In-Lab Activities:

Constructors:

Constructors in C++ are special class functions that are used to initialize objects. They are automatically called when an object is created. There are three types of constructors in C++:

- **Default Constructor:**

The default constructor is a constructor that takes no parameters, or default values may be provided. It is automatically called when an object is created.

Example Code:

```cpp
#include <iostream>

class Rectangle {
    int width, height;
public:
    Rectangle() {  // default constructor
        width = 10;
        height = 5;
    }
    int area() {
        return width * height;
    }
};

int main() {
    Rectangle rect;
    int area = rect.area();
    std::cout << "Area = " << area << std::endl;
    return 0;
}
```

Fig. 08 (Default Constructor)

Output:

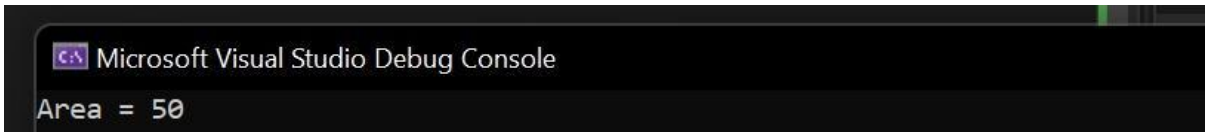


Fig. 09 (Output with respect to figure 8)

- **Parameterized Constructor:**

The parameterized constructor is a constructor that takes parameters and initializes the object with the given values.

Example Code:

```cpp
#include <iostream>

using namespace std;

class Student
{
private:
    int age;
    int id;

public:
    // Parameterized Constructor
    Student(int a, int b)
    {
        age = a;
        id = b;
    }

    void showDetails()
    {
        cout << "Age = " << age << endl;
        cout << "Id  = " << id << endl;
    }
};

int main()
{
    // Creating an object using Parameterized Constructor
    Student s1(18, 1234);
    s1.showDetails();
    return 0;
}
```

Fig. 10 (Parameterized Constructor)

Output:

```
Microsoft Visual Studio Debug Console
Age = 18
Id  = 1234
```

Fig. 11 (Output with respect to figure 10)

- **Copy Constructor:**

The copy constructor is a constructor that creates a new object by copying the values of an existing object. It is used to create a new object based on an existing object. The details are out of scope for this laboratory manual.

Overloaded Constructors:

Overloaded constructors in C++ provide developers with the ability to create multiple constructors with different parameters and different implementations. This makes it easier to create objects with different initial states.

Example:

For example, consider a Car class with the following two constructors:

Car()

Car(int numberOfDoors)

The first constructor is the default constructor which initializes the car with the default values. The second constructor takes a parameter and initializes the car with the provided number of doors. This allows a developer to create a car with the default values or one with the number of doors specified as a parameter.

Another use of overloaded constructors is to create objects with different numbers of parameters. This allows for more flexibility when creating objects. For example, a Car class could have the following three constructors:

Car()

Car(int numberOfDoors)

Car(int numberOfDoors, string color)

With these three constructors, developers can create cars with the default values, with a specified number of doors, or with a specified number of doors and color.

Overall, overloaded constructors in C++ provide developers with the ability to create multiple constructors with different parameters and different implementations. This makes it easier to create objects with different initial states, and can be a powerful tool when creating classes.

Example Code:

```cpp
#include <iostream>

using namespace std;

//Defining class
class Rectangle
{
    int length;
    int breadth;

public:
    //Default Constructor
    Rectangle()
    {
        length = 0;
        breadth = 0;
        cout << "Default Constructor called" << endl;
        cout << "Length: " << length << endl;
        cout << "Breadth: " << breadth << endl;
    }

    //Overloaded Constructor
    Rectangle(int l, int b)
    {
        length = l;
        breadth = b;
        cout << "Overloaded Constructor called" << endl;
        cout << "Length: " << length << endl;
        cout << "Breadth: " << breadth << endl;
    }
};

int main()
{
    //Default Constructor called
    Rectangle r1;
    cout << endl;

    //Overloaded Constructor called
    Rectangle r2(10, 5);

    return 0;
}
```

Fig. 11 (Overloaded Constructor)

Output:

```
Microsoft Visual Studio Debug Console
Default Constructor called
Length: 0
Breadth: 0

Overloaded Constructor called
Length: 10
Breadth: 5
```

Fig. 12 (Output with respect to figure 11)

Destructor:

A destructor is a special member function of a class that is used to release the memory held by an object of that class, just before it is destroyed. This usually happens when the object goes out of scope, or when the delete operator is used to explicitly delete the object. The destructor has the same name as the class, preceded by a tilde (~). The destructor has no return type and no parameters. Because it is not called explicitly, it is important for the destructor to be declared in the public section of the class.

Example Code:

```cpp
#include <iostream>

using namespace std;

class A
{
public:
    A() {
        cout << "Constructor called" << endl;
    }
    ~A() {
        cout << "Destructor called" << endl;
    }
};

int main()
{
    A a;   //Constructor called
    return 0;
}
```

Fig. 13 (Destructor)

Output:

```
Microsoft Visual Studio Debug Console
Constructor called
Destructor called
```

Fig. 14 (Output with respect to figure 13)

When the object a goes out of scope, the destructor ~A() is automatically called and the statement "Destructor called" is printed.

C++ Function Template

Templates are powerful features of C++ which allows us to write generic programs.

We can create a single function to work with different data types by using a template.

Defining a Function Template

A function template starts with the keyword `template` followed by template parameter(s) inside `<>` which is followed by the function definition.

```
template <typename T>
T functionName(T parameter1, T parameter2, ...)
{
    // code
}
```

In the above code, `T` is a template argument that accepts different data types (`int`, `float`, etc.), and `typename` is a keyword.

When an argument of a data type is passed to `functionName()`, the compiler generates a new version of `functionName()` for the given data type.

Calling a Function Template

Once we've declared and defined a function template, we can call it in other functions or templates (such as the `main()` function) with the following syntax

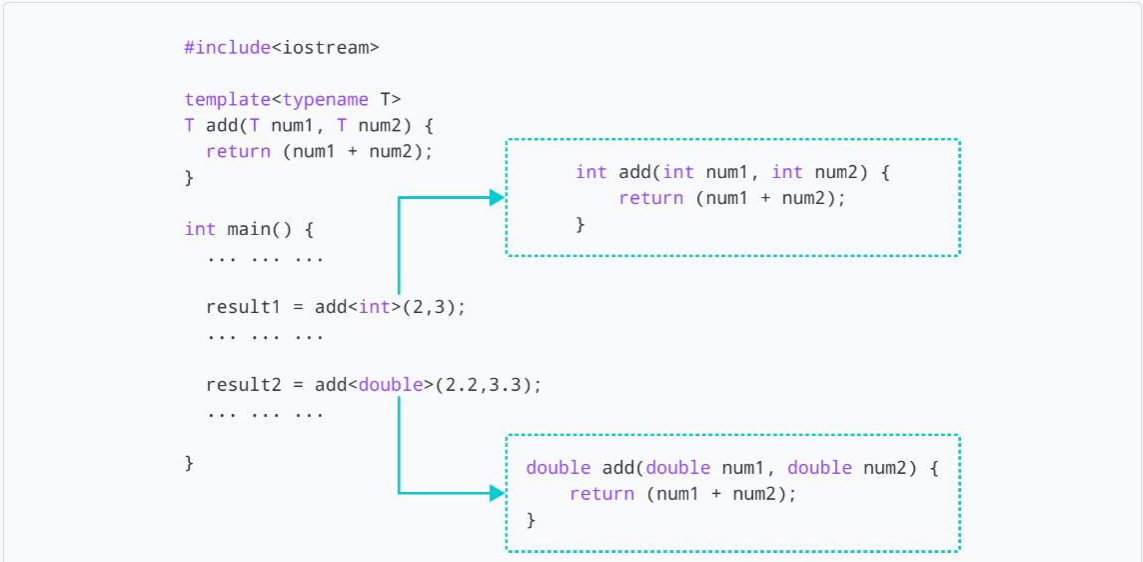
```
functionName<dataType>(parameter1, parameter2,...);
```

For example, let us consider a template that adds two numbers:

```
template <typename T>
T add(T num1, T num2)
{
    return (num1 + num2);
}
```

We can then call it in the `main()` function to add `int` and `double` numbers.

```cpp
int main()
{
    int result1;
    double result2;
    // calling with int parameters
    result1 = add<int>(2, 3);
    cout << result1 << endl;

    // calling with double parameters
    result2 = add<double>(2.2, 3.3);
    cout << result2 << endl;

    return 0;

    return 0;
}
```



Function Call based on data types

Example: Adding Two Numbers Using Function Templates

```
template <typename T>
T add(T num1, T num2)
{
    return (num1 + num2);
}

int main()
{
    int result1;
    double result2;
    // calling with int parameters
    result1 = add<int>(2, 3);
    cout << "2 + 3 = " << result1 << endl;

    // calling with double parameters
    result2 = add<double>(2.2, 3.3);
    cout << "2.2 + 3.3 = " << result2 << endl;

    return 0;
}
```

Output

```
2 + 3 = 5

2.2 + 3.3 = 5.5
```

C++ Inline Functions

In C++, we can declare a function as inline. This copies the function to the location of the function call in compile-time and may make the program execution faster.

Before following this tutorial, be sure to visit the C++ Functions.

inline Functions

To create an inline function, we use the `inline` keyword. For example,

```
inline returnType functionName(parameters) {
    // code
}
```

Notice the use of keyword `inline` before the function definition.

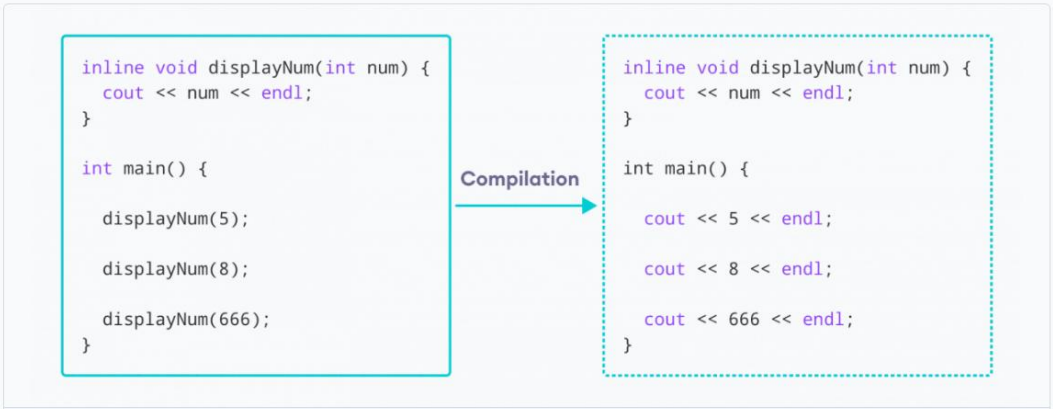
C++ Inline Function

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

inline void displayNum(int num) {
    cout << num << endl;
}

int main() {
    // first function call
    displayNum(5);

    // second function call
    displayNum(8);

    // third function call
    displayNum(666);

    return 0;
}
```

Output

```
5
8
666
```

Here is how this program works:



Working of inline functions in C++

Here, we created an inline function named `displayNum()` that takes a single integer as a parameter.

We then called the function 3 times in the `main()` function with different arguments. Each time `displayNum()` is called, the compiler copies the code of the function to that call location.

C++ Function Overloading

In C++, two functions can have the same name if the number and/or type of arguments passed is different.

These functions having the same name but different arguments are known as overloaded functions. For example:

```
// same name different arguments
int test() { }
int test(int a) { }
float test(double a) { }
int test(int a, double b) { }
```

Here, all 4 functions are overloaded functions.

Notice that the return types of all these 4 functions are not the same. Overloaded functions may or may not have different return types but they must have different arguments. For example,

```
// Error code
int test(int a) { }
double test(int b){ }
```

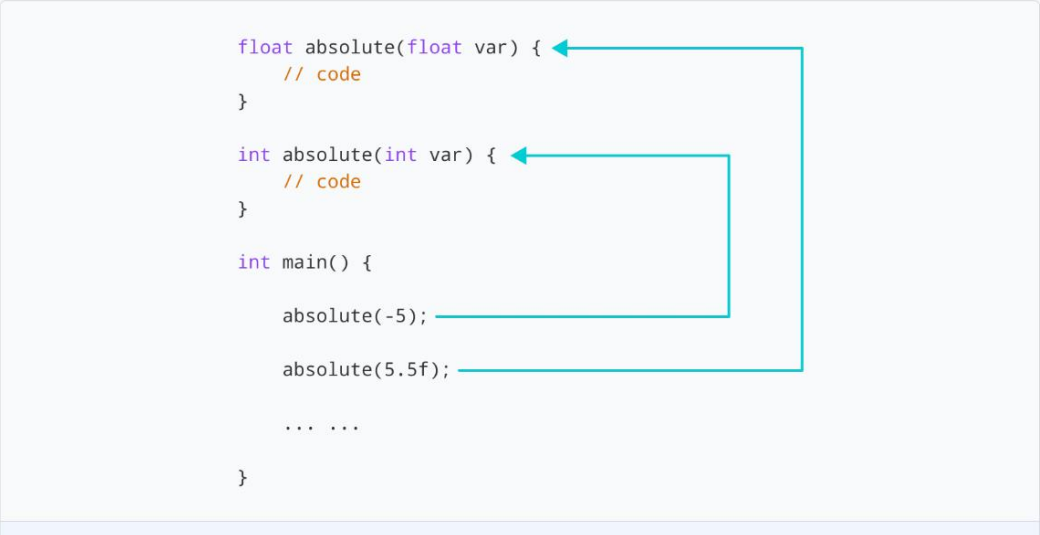
Here, both functions have the same name, the same type, and the same number of arguments. Hence, the compiler will throw an error.

Example 1: Overloading Using Different Types of Parameter

```
// function with float type parameter
float absolute(float var)
{
    if (var < 0.0)
        var = -var;
    return var;
}

// function with int type parameter
int absolute(int var)
{
    if (var < 0)
        var = -var;
    return var;
}
```

Output

```
Absolute value of -5 = 5

Absolute value of 5.5 = 5.5
```



Working of overloading for the absolute() function

In this program, we overload the `absolute()` function. Based on the type of parameter passed during the function call, the corresponding function is called.

Example 2: Overloading Using Different Number of Parameters

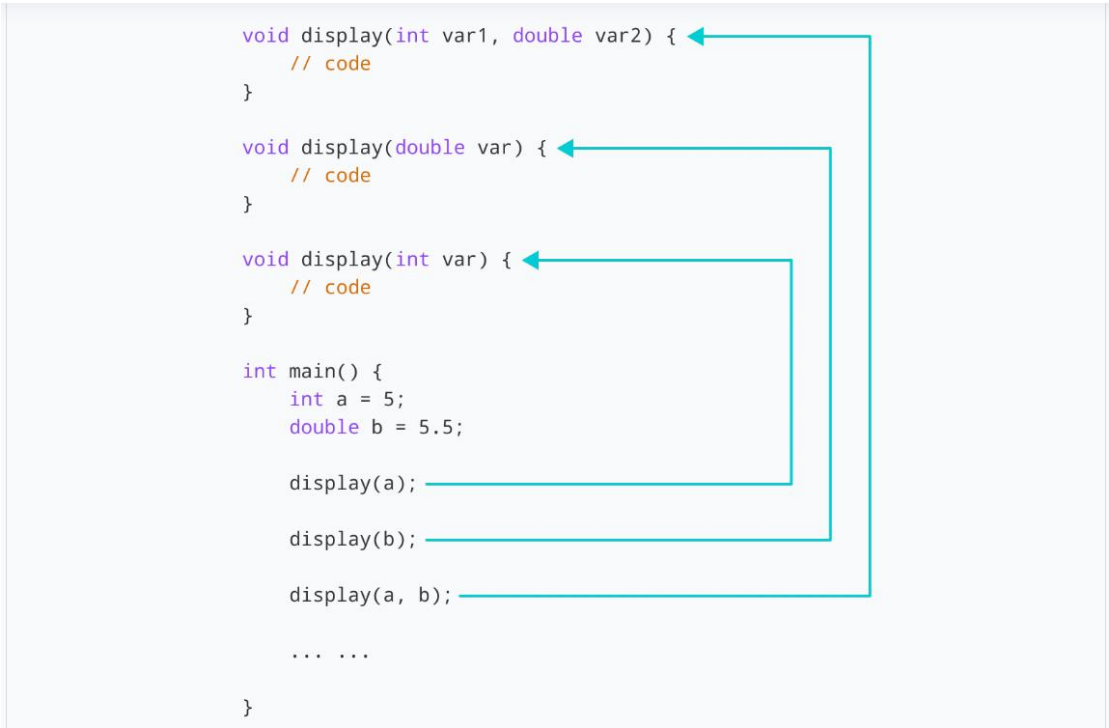
```
#include <iostream>
using namespace std;

// function with 2 parameters
void display(int var1, double var2) {
    cout << "Integer number: " << var1;
    cout << " and double number: " << var2 << endl;
}

// function with double type single parameter
void display(double var) {
    cout << "Double number: " << var << endl;
}

// function with int type single parameter
void display(int var) {
    cout << "Integer number: " << var << endl;
}
```

```
int main() {

    int a = 5;
    double b = 5.5;

    // call function with int type parameter
    display(a);

    // call function with double type parameter
    display(b);

    // call function with 2 parameters
    display(a, b);

    return 0;
}
```

Output

```
Integer number: 5

Float number: 5.5

Integer number: 5 and double number: 5.5
```

Here, the display() function is called three times with different arguments. Depending on the number and type of arguments passed, the corresponding display() function is called.



The return type of all these functions is the same but that need not be the case for function overloading.

> **Note:** In C++, many standard library functions are overloaded. For example, the `sqrt()` function can take `double`, `float`, `int`, etc. as parameters. This is possible because the `sqrt()` function is overloaded in C++.

In-Lab Tasks:

Task 01: Class Definition and Implementation [Estimated 40 minutes / 30 marks]

TEKKEN is a famous arcade game played around the world. Its last release was 10 years ago and now they want to release a new version of game. For this purpose they need an excellent programmer like you. To create a new release they need a class like the following:

class Player with the following attributes:

int ID;

string Name;

int HealthPoints;

int damagePoints;

bool isDead;

The following functionality is required.

Two players must be able to engage in a battle. When fight starts, display message:

"New Battle: {Player one} vs {Player two}"

When they fight, ask the user for number of hits for both players. Each player will do damage to other according to the his damage points.Damage points will subtract health points of the other player. If health points of a player go below zero, the player is dead. Display relative message, and show the Message:

"Winner By Knockout: {Player name}"

Task 2: Constructor Implementation [Estimated 20 minutes / 20 marks]

Mr Butcher is an old man now about 80 years old. He owns a shop where he regularly needs to add and subtract different kind of amounts for the shop daily affairs.

But Mr Butcher has a very old computer system

and with very limited storage.

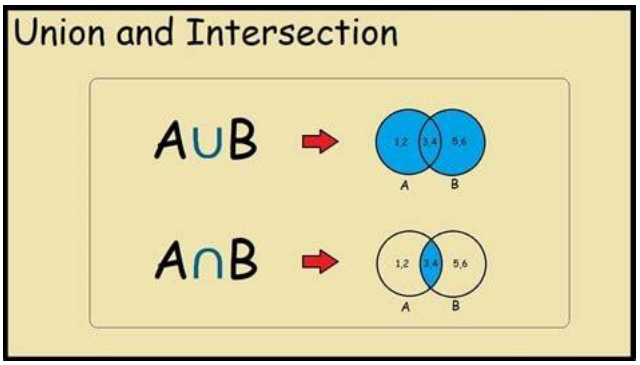
Request from the Old man:

He cant afford different functions for different data types.

Also some times he has two sum and subtract two values and sometimes three values.

Handle this functionality as well for him.

Task 3: Arrays class [Estimated 40 minutes / 20 marks]



Create a class Arrays, Attributes:

int * arrayOne;

int sizeOne;

int * arraysTwo;

int sizeTwo;

Pass values via parameterized constructor.

It will have two functions:

- Intersection: Returns a new array which will have intersection of the member arrays;
- Union: Returns a new array which will have Union of the member arrays.

Write Main function. Get result arrays, display them in main function.

Task 4: Student Class **[Estimated 30 minutes / 15 marks]**

Create a class Student with attributes:

- Roll number
- Name
- Address
- Degree Program
- Courses Array
- Number of courses

Methods:

- Provide Parameterized Constructor with default values
- Provide function to add a new course
- Provide function to Display all data for a student

Task-5: Find Errors (if any) and Remove them: (Estimated time: 5min/ Marks: 15)

```
#include <iostream>
using namespace std;
class Person{
   string ID;
   string name;
   string address;
   Person(string Id,string name, string address)
   {
      name=id;
      name=address;
      address=name;
   }
   int setID(int id)
   {
      ID=id;
   }
```

```
    int getName(string name)
    {
        name=name;
    }

    string getAddress(string address)
    {
        address=address;
    }
    void displayAllData()
    {
        cout<<"ID: "<<ID<<endl;
        cout<<"Name: "<<name<<endl;
    }
}
template <typename T>
void add(T one, T two)
{
    return one + two;
}

void add(T one, T two, T three)
{
    return one + two;
}

int main()
{
    Person person();
    person.setID(1);
    person.setName("M.Jafar Naqvi");
    person.setAddress("GRW");
```

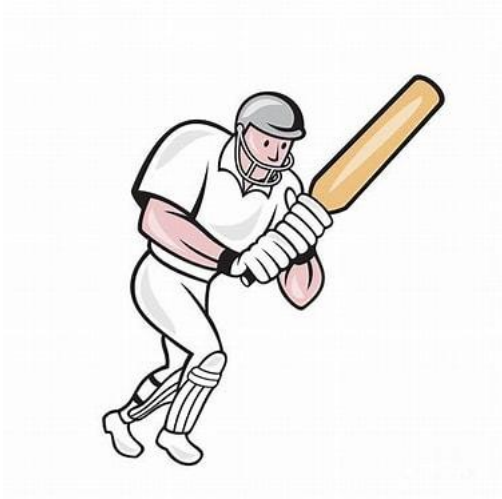
```
    person.displayAllData();
    return 0;
}
```

Post-Lab Activities:

Task 01: Cricket Stats **[Estimated 40 minutes / 40 marks]**

ICC has currently written its players match stats in registers. But with hundreds of new players coming to International arena, it is getting hard to maintain this system. Now they have decided to have a Computer based system to manage player's career stats.

Make a class Player:



Attributes:

- ID
- Name
- Team name
- No of Matches
- Total runs scored
- Total wickets
- Total catches taken

Methods:

- Default Constructor
- Parameterized Constructor
- Setters/ Getters
- Display Career Function

Main Function:

In main function, ask the user for number of Matches played. Make a team of 5 players, Input data for each player from user. Pass this team to a function, **"Data Displayer"** display the player with the Most runs scored, Player with the most wickets taken, player with most catches taken. Make a function to Display the whole team as well.

Submissions:

- For In-Lab Activity:
  - Save the files on your PC.
  - TA's will evaluate the tasks offline.
- For Pre-Lab & Post-Lab Activity:
  - Submit the .cpp file on Google Classroom and name it to your roll no.

References and Additional Material:

- Classes and Objects in C++
  https://www.programiz.com/cpp-programming/object-class

- Constructors
  https://www.programiz.com/cpp-programming/constructors

- Objects and Functions
  https://www.programiz.com/cpp-programming/pass-return-object-function

- Operator Overloading
  https://www.programiz.com/cpp-programming/operator-overloading

Lab Time Activity Simulation Log:

- Slot – 01 – 00:00 – 00:15: Class Settlement
- Slot – 02 – 00:15 – 00:30: In-Lab Task
- Slot – 03 – 00:30 – 00:45: In-Lab Task
- Slot – 04 – 00:45 – 01:00: In-Lab Task
- Slot – 05 – 01:00 – 01:15: In-Lab Task
- Slot – 06 – 01:15 – 01:30: In-Lab Task
- Slot – 07 – 01:30 – 01:45: In-Lab Task
- Slot – 08 – 01:45 – 02:00: In-Lab Task
- Slot – 09 – 02:00 – 02:15: In-Lab Task
- Slot – 10 – 02:15 – 02:30: In-Lab Task
- Slot – 11 – 02:30 – 02:45: Evaluation of Lab Tasks

Slot – 12 – 02:45 – 03:00: