# Set 2

A \_\_\_\_\_\_\_\_\_\_ is a block of code which only runs when it is called. **function**

You can pass data, known as\_\_\_\_\_\_\_\_\_\_, into a function. **parameters**

To create (often referred to as \_**declare**\_) a function, specify the name of the function, followed by parentheses **()**

A C++ function consist of two parts. One is the \_**declaration**\_ which is the function's name, return type, and parameters (if any).

A C++ function consist of two parts. One is the \_**definition**\_ which is the body of the function (code to be executed).

Is it possible to separate the declaration and the definition of the function? **Yes**

Information can be passed to functions as a \_\_**parameters**\_\_\_\_\_\_\_\_. They act as variables inside the function.

You can also use a default parameter value, by using the \_\_**=**\_\_\_\_\_\_\_\_.

If you want the function to return a value, you can use a \_\_\_**data type**\_\_\_ (such as int, string, etc.) instead of void, and use the return keyword inside the function:

Procedural programming is about writing procedures or functions that perform operations on the data, while \_**OOP**\_ is about creating objects that contain both data and functions.

Given:

|  |  |
| --- | --- |
| **Class** | **Objects** |
| Fruit | Apple, Banana, Mango |

|  |  |
| --- | --- |
| **Class** | **Objects** |
| Car | Volvo, Audi, Toyota |

|  |  |
| --- | --- |
| **Class** | **Objects** |
| Starship | Galaxy, Runabout, Shuttle |

As observed in the tables above, a \_\_**class**\_ is a template for an object.

As observed in the tables above, an object is an \_**instance**\_ of a class.

When the individual objects are created, they inherit all the variables and functions from the \_**class**\_.

C++ is an \_**OOP**\_\_\_ programming language.

Everything in C++ is associated with \_\_**classes and objects**\_\_, along with its attributes and methods.

Attributes and methods are basically **variables** and **functions** that belongs to the class. These are often referred to as \_**class members**\_.

A class is a user-defined \_\_**data type**\_ that we can use in our program.

A class works as an object \_**object constructor**\_, or a "blueprint" for creating objects.

To create a class, use the \_**class**\_ keyword.

In C++, an \_\_**object**\_ is created from a class.

Given the following example:

// -----------------------------------------------  
class MyClass {       // The class  
  public:             // Access specifier  
    int myNum;        // Attribute (int variable)  
    string myString;  // Attribute (string variable)  
};  
  
int main() {  
  MyClass **myObj**;  // Create an object of MyClass  
  
  // Access attributes and set values  
  **myObj.myNum** = 15;   
  **myObj.myString** = "Some text";  
  
  // Print attribute values  
  cout << myObj.myNum << "\n";   
  cout << myObj.myString;   
  return 0;  
}

// -----------------------------------------------  
To access the class attributes (myNum and myString), use the \_\_.\_\_ on the object.

Methods are \_**functions**\_ that belongs to the class.

There are two ways to define functions that belongs to a class: \_\_**inside class def**\_ and \_**outside class def**\_.

Given the following example:

// -----------------------------------------------  
class MyClass {        // The class  
  public:              // Access specifier  
    void myMethod() {  // Method/function defined inside the class  
      cout << "Hello World!";  
    }  
};  
  
int main() {  
  MyClass myObj;     // Create an object of MyClass  
  myObj.myMethod();  // Call the method  
  return 0;  
}// -----------------------------------------------

You access methods just like you access attributes; by creating an object of the class and by using the \_\_\_.\_\_\_\_\_\_\_.

Given the following example:

// -----------------------------------------------  
class MyClass {        // The class  
  public:              // Access specifier  
    void myMethod();   // Method/function declaration  
};  
  
// Method/function definition outside the class  
void **MyClass::myMethod()** {  
  cout << "Hello World!";  
}  
  
int main() {  
  MyClass myObj;     // Create an object of MyClass  
  myObj.myMethod();  // Call the method  
  return 0;  
}

// -----------------------------------------------  
To define a function outside the class definition, you have to declare it inside the class and then define it outside of the class. This is done by specifying the name of the class, followed by the \_\_: : \_\_, followed by the name of the function.

A \_\_\_**constructor**\_\_\_\_\_\_\_ in C++ is a **special method** that is automatically called when an object of a class is created.

To create a \_\_**constructor**\_\_, use the same name as the class, followed by parentheses ().

The constructor has the same name as the class, it is always \_**public**\_\_\_, and it does not have any return value.

Just like functions, constructors can also be defined outside the class. First, declare the constructor inside the class, and then define it outside of the class by specifying the name of the class, followed by the scope resolution :: operator, followed by the name of the \_**constructor**\_ (which is the same as the class).

Given the following example:

// -----------------------------------------------  
class MyClass {  // The class  
  **public:**          
    // class members goes here  
};

// -----------------------------------------------  
Concerning classes and referring to the above example, the public keyword is an\_**access specifier**\_\_.

Concerning classes and referring to the above example, \_**access specifier**\_\_ define how the members (attributes and methods) of a class can be accessed.

Concerning classes and referring to the above example, the members are \_\_**public**\_\_ - which means that they can be accessed and modified from outside the code.

In C++, there are three access specifiers. One is \_**public**\_\_ where the members are accessible from outside the class.

In C++, there are three access specifiers. One is \_\_**private**\_\_ where the members cannot be accessed (or viewed) from outside the class.

In C++, there are three access specifiers. One is \_**protected**\_ where the members cannot be accessed from outside the class, however, they can be accessed in inherited classes.

Given the following example:

// -----------------------------------------------  
class MyClass {  
  **public:**    // Public access specifier  
    int x;   // Public attribute  
  **private:**   // Private access specifier  
    int y;   // Private attribute  
};  
  
int main() {  
  MyClass myObj;  
  myObj.x = 25;  // Allowed (public)  
  myObj.y = 50;  // Not allowed (private)  
  return 0;  
}

// -----------------------------------------------

Referring to the above example, this program will run without error (T or F). **F**

By default, all members of a class are \_\_**private**\_ if you don't specify an access specifier.

The meaning of Encapsulation, is to make sure that \_\_**sensitive**\_\_ data is hidden from users.

To be sure data is hidden from users, you must declare class variables/attributes as \_**private**\_\_ (cannot be accessed from outside the class).

Encapsulation ensures better control of your \_**data**\_\_\_, because you (or others) can change one part of the code without affecting other parts.

In C++, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories. One is the \_\_**derived class**\_ which is the class that inherits from another class.

In C++, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories. One is the \_**base class**\_ which is the class being inherited from.

To inherit from a class, use the \_\_:\_\_\_\_\_\_\_\_ symbol.

Given the following example:

// -----------------------------------------------  
// Base class  
class Vehicle {  
  public:   
    string brand = "Ford";  
    void honk() {  
      cout << "Tuut, tuut! \n" ;  
    }  
};  
  
// Derived class  
**class Car: public Vehicle** {  
  public:   
    string model = "Mustang";  
};  
  
int main() {  
  Car myCar;  
  myCar.honk();  
  cout << myCar.brand + " " + myCar.model;  
  return 0;  
}

// -----------------------------------------------  
In which line of code are we linking the derived class to the base class?

\_\_**class Car: public Vehicle**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Why use Inheritance? \_\_**Useful code reusability: reuse attributes and methods of an existing class when you create a new class**\_\_.

Given the following example:

// -----------------------------------------------  
// Base class (parent)  
class MyClass {  
  public:   
    void myFunction() {  
      cout << "Some content in parent class." ;  
    }  
};  
  
// Derived class (child)  
class MyChild: public MyClass {  
};  
  
// Derived class (grandchild)   
class MyGrandChild: public MyChild {  
};  
  
int main() {  
  MyGrandChild myObj;  
  myObj.myFunction();  
  return 0;  
}

// -----------------------------------------------  
When we have a class derived from one class, which is already derived from another class, we call this \_\_**multilevel**\_\_\_.

What line indicates the link between the parent class and the child class? **class MyChild: public MyClass**

What line indicates the link between the child class and the grandchild class? **class MyGrandChild: public MyChild**

Given the following example:

// -----------------------------------------------  
// Base class  
class MyClass {  
  public:   
    void myFunction() {  
      cout << "Some content in parent class." ;  
    }  
};  
  
// Another base class  
class MyOtherClass {  
  public:   
    void myOtherFunction() {  
      cout << "Some content in another class." ;  
    }  
};  
  
// Derived class   
**class MyChildClass: public MyClass, public MyOtherClass** {  
};  
  
int main() {  
  MyChildClass myObj;  
  myObj.myFunction();  
  myObj.myOtherFunction();  
  return 0;  
}

// -----------------------------------------------  
A class can also be derived from more than one base class, using a comma-separated list. When we do this, it is called \_**multiple**\_.

What line links the two base classes to the one derived class? \_\_**class MyChildClass: public MyClass, public MyOtherClass**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_**polymorphism**\_\_\_\_\_\_\_\_ means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

**Inheritance** lets us inherit attributes and methods from another class. **\_\_Polymorphism\_\_** uses those methods to perform different tasks. This allows us to perform a single action in different ways.

Given the following example:

// -----------------------------------------------  
// Base class  
class Animal {  
  public:  
    void animalSound() {  
    cout << "The animal makes a sound \n" ;  
  }  
};  
  
// Derived class  
class Pig : public Animal {  
  public:  
    void animalSound() {  
    cout << "The pig says: wee wee \n" ;  
   }  
};  
  
// Derived class  
class Dog : public Animal {  
  public:  
    void animalSound() {  
    cout << "The dog says: bow wow \n" ;  
  }  
};  
  
int main() {  
  Animal myAnimal;  
  Pig myPig;  
  Dog myDog;  
  
  myAnimal.animalSound(); // uses animalSound from Animal class  
  myPig.animalSound(); // uses animalSound from Pig class  
  myDog.animalSound(); // uses animalSound from Dog class  
  return 0;  
}

// -----------------------------------------------  
This is an example of \_**polyorphism?**\_\_\_\_\_\_\_\_\_.

Note: The base class Animal and the two derived classes Pig and Dog each has a method named animalSound(). Which method will be used depends on the data type used when declaring an instance of an object. For the object myAnimal, the animalSound() method used will be from the Animal class. For the object myPig, the animalSound() method used will be from the Pig class. And for the object myDog, the animalSound() method used will be from the Dog class.

When an error occurs, C++ will normally stop and generate an error message. The technical term for this is: C++ will throw an \_**exception**\_\_\_\_\_\_\_\_\_ (throw an error).

Exception handling in C++ consist of three keywords. One, the \_**try**\_\_ statement allows you to define a block of code to be tested for errors while it is being executed.

Exception handling in C++ consist of three keywords. One, the \_**throw**\_\_ keyword throws an exception when a problem is detected, which lets us create a custom error.

Exception handling in C++ consist of three keywords. One, the \_**catch**\_\_ statement allows you to define a block of code to be executed, if an error occurs in the try block.

A reference variable is a "reference" to an existing variable, and it is created with the \_**&**\_ operator.

A pointer is a datatype that contains a memory address[[1]](#footnote-1) as its value.

The & sign is called the address operator[[2]](#footnote-2).

Given the following:

// -----------------------------------------------  
int main(){

int cars = 5;

cout << &cars << endl;

int \*carsPointer;

cout << carsPointer << endl;

return 0

}

// -----------------------------------------------  
In the code above, &cars is a reference[[3]](#footnote-3).

In the code above, \*carsPointer is a pointer[[4]](#footnote-4).

Given the following:

// -----------------------------------------------  
int main(){

string food = "Pizza";  
 string &meal = food;  
  
 cout << food << "\n";  // Outputs Pizza  
 cout << meal << "\n";  // Outputs Pizza

return 0;

}

// -----------------------------------------------

In the code above, the & was used to create a \_\_**reference**\_\_ variable.

References and Pointers are important in C++, because they give you the ability to \_**manipulate**\_ the data in the computer's memory - which can reduce the code and improve the performance.

We can get the \_**memory address**\_ of a variable by using the & operator.

A pointer variable points to a data type (like int or string) of the \_**same type**\_ , and is created with the \* operator. The address of the variable you're working with is assigned to the pointer.

Declaring a pointer can be done in three ways:

int\* p int \*p int \* p

After you declare a variable is a pointer, you do not have to use the \* again[[5]](#footnote-5).

In previous examples, we used the pointer variable to get the memory address of a variable (used together with the & reference operator). However, you can also use the pointer to get the value of the variable, by using the \_\_\*\_\_\_\_\_\_\_\_ operator (the **dereference** operator).

Given the following:

// -----------------------------------------------  
int main(){

string food = "Pizza";    
 **string\* ptr = &food;**      
  
 // Note for example. food is stored at

// memory location 0x7ffdf5798820   
 cout << ptr << "\n";  
  
 **cout << \*ptr <<** "\n**";**

return 0;

// -----------------------------------------------

In the code above, the line cout << ptr << "\n"; will display \_0x7ffdf5798820\_\_\_\_\_\_\_\_\_ .

In the code above, the line cout << \*ptr << "\n"; will display \_\_**Pizza**\_\_\_\_\_\_\_\_ .

When used in declaration (string\* ptr), it creates a\_**pointer variable**\_.

When not used in declaration, it act as a \_\_**dereference**\_ operator.

# Test 1

1. Every C++ program has a function called  **main**
2. The function main returns an **integer**
3. Built in data types:  **char, bool, int, float, double, string**
4. Before an identifier (name) can be used it must be **initialized/declared**
5. The insertion operator << is used for **output**
6. The extraction operator >> is used for **input**
7. **Endl** is the newline character
8. We use #include <**fstream**> for accessing data files
9. Concerning data files, use **open** to prepare a stream for use
10. Concerning data files, use **close** to break the connection between the stream and the variable when you are done with the stream
11. The statement intVar = int (floatVar); is an example of **explicit type coercion**
12. The body of **while loop** may never execute
13. A sentinel-controlled loop is looking for a marker called a **flag** to tell it to stop the loop
14. An **EOF** controlled loop is looking for the operating system End-Of-File marker to indicate the end of the data to be processed
15. The **for** statement is intended to simplify the writing of count controlled loops.
16. When a loop is contained inside another loop, we call this a **nested**  loop
17. The body of a **do-while** will execute at least 1 time
18. When your running program gets a **break** statement, it causes an immediate exit from the innermost switch, while, do-while, or for statement in which it appears.
19. A **function** is a block of code which only runs when it is called
20. You can receive **information** as a result from a function
21. Functions are important for **reusing code.** Define the code once and use it many times
22. There are three components to a C++ function
    1. The **prototype** is a declaration of the identifier used to name the function
    2. The **definition** contains the statements that perform that function’s task
    3. The **call** appears in the client code and is used to invoke a particular function
23. There are two types of C++ functions
    1. **Void** functions do not return any information from the function
    2. **Value returning**  functions do return information from the function
24. The function definition must immediately follow the function prototype. **F**
25. Pass by value passes the **data stored in** the variable location
26. Pass by reference passes the **address** of the variable location
27. A reference variable is a “reference to an existing variable and is created with the **&** operator
28. To pass a variable by reference, we simply declare the **variable** as references rather than as normal variables
29. Concerning a value retuning function definition, an important observation is the **output** matches the data type of the value listed in the return statement
30. A **struct** is a user defined data type. It can store data of multiple data types. Each element is called a member
31. Int values[1000]; This statement declares a **1-Dimensional Array**
32. You cannot pass arrays **by value** in C++
33. Int sample [6][4]; This statement declares a **2-Dimensional Array**
34. A **typedef** provides a level of abstraction away from the actual types being used, allowing you to focus more on the concept of just what a variable should mean
35. Typedef unsigned int score; is an example of using **typedef/user defined variables**

*Linux Review*

1. An **operating system** is a collection of programs that manage the resources of a computer
2. In UNIX and Linux, a folder is known as a **directory**
3. When you first open a terminal window you will be in your **home directory**, the location where your files will be stored
4. Filenames and directory names are **case sensitive**
5. Display current working directory **pwd**
6. List contents of a directory **ls**
7. List contents of a directory in a longer listing  **ls -l**
8. List the contents of a directory including any hidden files **ls -a**
9. Creates new directory within the current directory **mkdir**
10. Change directory **cd**
11. Switch back to parent directory of current working directory **cd –**
12. Copies source file to specified destination file **cp**
13. Removes specified file **rm**
14. Removes specified directory **rmkdir**
15. Display built in manual pages **man**

# Set 3

**Chapter 2:**

An \_\_**abstract data type** is a data type whose properties (domain and operations) are specified independently of any particular implementation.

We refer to the set of all possible values (the domain) of an encapsulated data object plus the specifications of the operations that are provided to create and manipulate the data as an \_\_\_\_**ADT**\_\_\_\_\_\_.

A \_**data structure**\_ is a collection of data elements whose organization is characterized by accessing operations that are used to store and retrieve the individual data elements; the implementation of the composite data members in an abstract data type.

A \_\_**data structure**\_\_\_\_ is defined by the logical arrangement of data elements combined with the set of operations we need to access the elements.

A \_\_**composite**\_\_ is a data type that allows a collection of values to be associated with an object of that type.

A colloquial definition of a \_\_**class**\_\_\_\_\_\_\_ is a combination of data items and the functions used to manipulate them. class

A \_\_**1-D**\_\_\_ array is a structured composite data type made up of a finite, fixed-size collection of ordered homogenous elements to which direct access is available.

In an array, \_\_**finite**\_\_ indicates that a last element is identifiable.

\_**Fixed size**\_\_ means the size of the array must be known in advance.

An \_\_**Ordered**\_\_ array means there is a first element, a second element and so on. The relative position of the elements is ordered, not necessarily the values stored there.

The elements in an array are physically \_**homogeneous**\_. That is, they are all of the same data type.

The component selection mechanism of an array is \_\_**direct access**\_ – that means we can access any element directly without first accessing the preceding elements.

In C++ arrays can only be \_\_**reference**\_\_ parameters, it’s not possible to pass an array by value.

A \_\_\_**1-D array**\_\_ is the natural structure for the storage of lists of like data elements. An array is a data structure that contains a group of \_**elements**\_\_\_ that are all of the same data type, such as an integer or string.

A two-dimensional array is commonly referred to as a \_**matrix**\_\_.

A \_\_**2-d**\_\_ array is a structured composite data type made up of a finite, fixed-size collection of homogeneous elements ordered in two dimensions and accessed by direct access.

A \_**client**\_ is software that declares and manipulates instances of objects of a particular class.

The attributes (variables) and the methods (functions) of a class are called \_\_**members**\_\_.

Although the class specification (declarations) and the implementation (definitions) can reside in the same file, the two parts of a class are usually separated into two files. The specification goes into a header file and has an \_**.h**\_ extension.

Although the class specification (declarations) and the implementation (definitions) can reside in the same file, the two parts of a class are usually separated into two files. The implementation goes into a file with the same name as the header and has an \_\_**.cpp**\_\_ extension.

The rules of C++ that governs what, where and when are called \_**scope rules**\_.

\_**class scope**\_ refers to identifiers (variables) declared within a class declaration.

\_**local scope**\_ is the scope of an identifier (variable) declared within a block (statements enclosed within { } ).

\_**global scope**\_ is the scope of an identifier (variable) declared outside all functions and classes.

All identifiers declared within a class are local to the class. This is called \_**class scope**\_.

The name of a function that is not a member of a class has global scope. Once a global function name has been declared, any subsequent function can call it. This is called \_\_\_**global scope**\_.

Given the existence of a class DateType, a client of the class DateType must have an \_**#include**\_\_ directive for the header file of the class.

System-supplied header files are enclosed in angle brackets ( <iostream> ) where user-defined header files are enclosed in \_**double quotes**\_\_\_\_\_\_\_\_\_.

Only the member functions can access the data members, so we must associate the class name with the function definitions. We do so by inserting the class name before the function name, separated by the \_**scope resolution operator (:)**\_\_.

Member functions of a class are invoked in the same way that data members of a struct are accessed – using \_**dot notation**\_\_.

An object is an \_\_**instance**\_\_\_\_\_\_\_\_ of a class type.

\_**overloading**\_ means giving the same name to several different functions (or using the same operator symbol for different operations) in a class.

The time at which a name or symbol is bound to the appropriate code is called \_**binding time**\_\_\_.

The compile-time determination of which implementation of an operation is appropriate is called \_\_**static binding**\_\_\_.

The run-time determination of which implementation of an operation is appropriate is called \_**dynamic binding**\_\_\_.

**From Pages 134-137**

Lists are very useful abstract data types (ADTs). They are members of a general category of ADTs called \_**containers**\_\_, whose purpose is to hold other objects.

Lists are provided in the \_\_**standard template library**\_\_\_\_.

A \_\_**list**\_ is a homogenous collection of elements with a linear relationship between elements.

The number of items in the list, which we will call the \_**length**\_ of the list, is a property of a list.

Lists can be \_**unsorted**\_ - their elements are placed in no particular order.

Lists can be \_**sorted**\_ - their elements are placed in a particular order.

When the elements in a sorted list are of a composite type, their logical (and often physical) order is determined by one of the members of the structure, called the \_**key**\_ member.

In general, we can group the instance methods of a class into 5 categories. They are \_\_\_\_\_\_\_\_\_\_, \_\_\_\_\_\_\_\_\_\_, \_\_\_\_\_\_\_\_\_\_, \_\_\_\_\_\_\_\_\_\_ and \_\_\_\_\_\_\_\_\_\_. Constructors, transformers, observers, iterators, destructors

A \_**constructor**\_ creates an instance of a data type.

\_**transformers**\_ are operations that change the structure in some way. They may make the structure empty, put an item into the structure or remove a specific item from a structure.

\_\_**observers**\_\_ ask true/false questions about the data type, select or access a particular item or return a property about a structure.

\_\_**iterators**\_\_\_ are used with composite data types to allow the user to process an entire structure component by component (or element by element).

\_**transformer**\_ alter the data fields of an object.

\_**observers**\_ read or otherwise get information about the data fields.

\_**constructors**\_ creates data for the ADT.

\_**iterators**\_ provide the ability to move through data fields one at a time.

A \_\_**destructor**\_ is executed when an object of it’s class goes out of scope or when the delete expression is applied to a pointer to the object of that class.

A \_\_**destructor** will have the exact same name as the class prefixed with a tilde (~). It can never take a value nor can it take any parameters. They can be used for releasing resources before coming out of a program like closing files, releasing memory, releasing printers, etc.

A \_**function prototype**\_ is a declaration.

A function heading with a body is a \_**function defintion**\_.

**Chapter 5:**



In the picture above, these are examples of \_**stack**\_.

At the logical level, a \_**stack**\_ is an ordered group of homogenous items or elements.

In a stack, the removal of existing items and the addition of new items can only take place at the \_**top**\_ of the stack.

A stack may be considered an \_**ordered**\_ group of items because elements occur in a particular sequence organized by how long they have been in the stack.

Because elements are added and removed only from the top of the stack, the last element to be added is the first to be removed. This stack behavior is referred to as \_**LIFO**\_.

The operation that adds an element to the top of the stack is usually called a \_\_**push**\_\_.

The operation that removes an element from a stack is called a \_**pop**\_.

If we need to examine the item at the top of the stack without removing the item, we can use a separate operation called \_**top**\_.

We must be able to tell whether a stack contains any elements before we pop it, so we need a Boolean operation called \_**IsEmpty**\_.

You may need to see if a stack is full before pushing. We call this operation \_\_**IsFull\_**.

A stack and an \_\_**unsorted list**\_ are two different Abstract Data Types (ADTs).

With a stack and an unsorted list can use the same \_**implementation strategy**\_ for both.

If the stack is full when we invoke Push, the resulting condition is called \_\_\_\_\_\_\_\_\_\_. A stack, queue or list can also be described as an \_\_\_\_\_\_\_\_\_\_. abstract data structure

Given an abstract data structure where an item is added at one end and removed from another. This structure is called a \_**FIFO**\_ queue.

A \_**queue**\_\_\_ is an ordered, homogenous group of elements in which one element is added at the rear and removed from the front.

To add elements to the queue, we access the rear of the queue. To remove elements we access the front of the queue. The middle elements are logically \_\_**inaccessible**\_.

Like a stack, a queue is a \_**holding structure**\_ for data we will use later.

When we add an element to the rear of a queue, this operation is called \_**enqueue**\_\_.

When we remove an element from the front of a queue, this operation is called \_**dequeue**\_.

(**True** or False) The stack operations *push* and *pop* are standard names.

(True or **False**) The queue operations *enqueue* and *dequeue* are standard names.

Computer systems must often provide a “Holding area” for messages between two processes, two programs or even two systems. This holding area, usually called \_**buffer**\_ is often implemented as a FIFO queue.

1. If question used, memory address will be the answer. – Ref. Bucky Roberts, Sr. Engineer, LH Ventures. [↑](#footnote-ref-1)
2. If question used, address operator will be the answer. – Ref. Bucky Roberts, Sr. Engineer, LH Ventures. [↑](#footnote-ref-2)
3. If question used, reference will be the answer. – Ref. Bucky Roberts, Sr. Engineer, LH Ventures. [↑](#footnote-ref-3)
4. If question used, pointer will be the answer. – Ref. Bucky Roberts, Sr. Engineer, LH Ventures. [↑](#footnote-ref-4)
5. This statement is True. – Ref. Bucky Roberts, Sr. Engineer, LH Ventures. [↑](#footnote-ref-5)