Q.2 a. Write four advantages of using an Object Oriented Programming language. (4)

Answer:

The major advantages of OOPs are:

1. Simplicity: Software objects model real world objects, so the complexity is reduced and the program structure is very clear.
2. Modularity: Each object forms a separate entity whose internal workings are decoupled from other parts of the system.
3. Modifiability: It is easy to make minor changes in the data representation or the procedures in an OO program. Changes inside a class do not affect any other part of a program, since the only public interface that the external world has to a class is through the use of methods.
4. Extensibility: Adding new features or responding to changing operating environments can be solved by introducing a few new objects and modifying some existing ones.
5. Maintainability: Objects can be maintained separately, making locating and fixing problems easier.
6. Re-usability: Objects can be reused in different programs

b. Differentiate between an object and a class with a suitable example. (4)

Answer:

Class is blueprint means you can create different object based on one class which varies in there property. e.g. if Car is a class than Mercedes, BMW or Audi can be considered as object because they are essentially a car but have different size, shape, color and feature.

A Class can be analogous to structure in C programming language with only difference that structure doesn't contain any methods or functions, while class in Java contains both state and behavior, state is represented by field in class e.g. numberOfGears, whether car is automatic or manual, car is running or stopped etc. On the other hand behavior is controlled by functions, also known as methods in Java e.g. start() will change state of car from stopped to started or running and stop() will do opposite.

Object is also called instance in Java and every instance has different values of instance variables. e.g. in following code

c. Write an object based program to read a positive integer n, compute the sum of first n natural number and then output "Sum of first natural number is =" <actual computed sum>. (8)

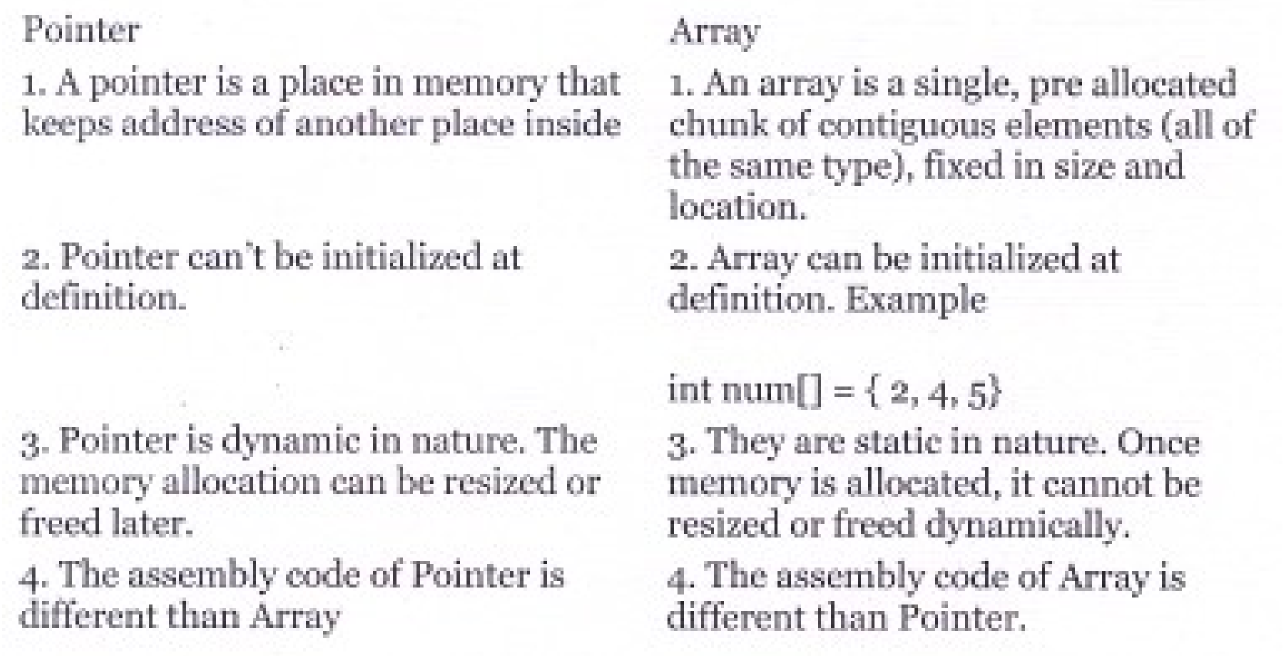
Answer:

Q.3 a. Define a class *strcmp* that accepts two strings as input and compares the two string and returns a value -1, 0 or 1 depending upon whether first string is less than or equal to or greater than the second string. (6)

Answer:

b. Give four basic differences between a pointer and an array. (4)

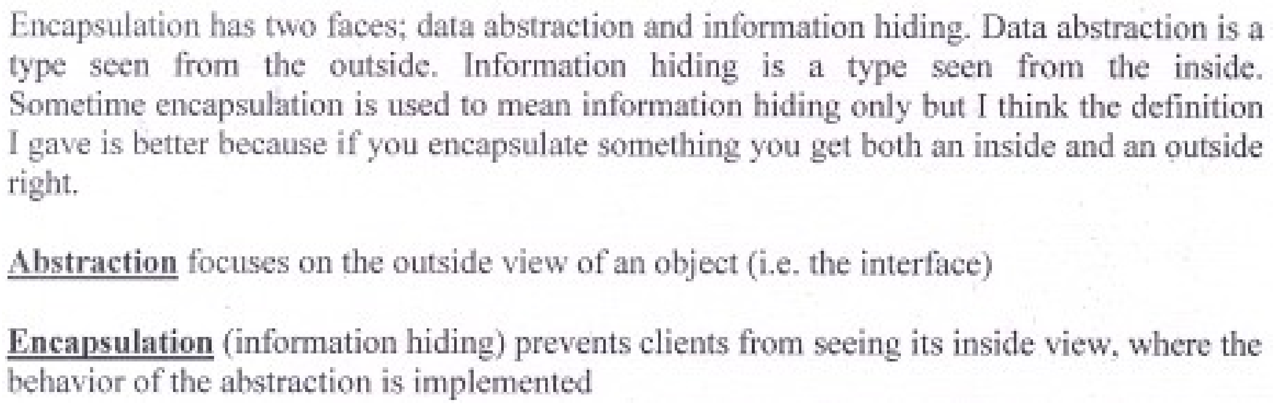
Answer:



Pointer	Array
1. A pointer is a place in memory that keeps address of another place inside	1. An array is a single, pre allocated chunk of contiguous elements (all of the same type), fixed in size and location.
2. Pointer can't be initialized at definition.	2. Array can be initialized at definition. Example int num[] = { 2, 4, 5}
3. Pointer is dynamic in nature. The memory allocation can be resized or freed later.	3. They are static in nature. Once memory is allocated, it cannot be resized or freed dynamically.
4. The assembly code of Pointer is different than Array	4. The assembly code of Array is different than Pointer.

c. What do you mean by data abstraction and encapsulation? (6)

Answer:



Encapsulation has two faces; data abstraction and information hiding. Data abstraction is a type seen from the outside. Information hiding is a type seen from the inside. Sometime encapsulation is used to mean information hiding only but I think the definition I gave is better because if you encapsulate something you get both an inside and an outside right.

Abstraction focuses on the outside view of an object (i.e. the interface)

Encapsulation (information hiding) prevents clients from seeing its inside view, where the behavior of the abstraction is implemented

Abstraction uses in order to do some works easily. For example, you have a abstract class fruit. And fruit has some methods one of which has to be abstract. So you should fill the body of this method's body for your each class who are subclass of this abstract class. The purpose of it is that to implement method according to that class.

Abstraction is the process of hiding the details and exposing only the essential features of a particular concept or object.

Encapsulation is the ability of an object to be a container (or capsule) for related properties (ie. data variables) and methods (ie. functions). Programs written in older languages did not enforce any property/method relationship. This often resulted in side effects where variables had their contents changed or reused in unexpected ways and 'spaghetti' code (branching into procedures from external points) that was difficult to unravel, understand and maintain. Encapsulation is one of three fundamental principles within object oriented approach.

Data hiding is the ability of objects to shield variables from external access. These private variables can only be seen or modified by use of object accessor and mutator methods. This permits validity checking at run time. Access to other object variables can be allowed but with tight control on how it is done. Methods can also be completely hidden from external use. Those that are made visible externally can only be called by using the object's front door (ie. there is no 'goto' branching concept).

Q.4 a. What does 'this' pointer stand for? What is the advantage of 'this' pointer? (4)

Answer:

Each time we creates a class instance in a program, C++ creates a special pointer called *this*, which contains the address of the current object instance. Each time the program invokes an instance method (e.g. `a.init()`), the compiler pre-assigns a special pointer named *this* to point to the object instance.
The value of *this* pointer changes with different instance invocations. C++ recognizes the *this* pointer only when a non-static member of the object instance is executing. The instances, in turn, use the *this* pointer to access the different methods.

Every member function of a class has an implicitly defined constant pointer called *this*. The type of *this* is the type of the class of which the function is member. It is initialized when a member function is called to the address of the class instance for which the function was called.

b. What is typecasting? What are explicit and implicit type conversions? Explain your answer with a suitable example. (6)

Answer:

Converting an expression of a given type into another type is known as *type-casting*.

Implicit conversions do not require any operator. They are automatically performed when a value is copied to a compatible type. For example:

```
short a=2000;
int b;
b=a;
```

Here, the value of a has been promoted from short to int and we have not had to specify any type-casting operator. This is known as a standard conversion. Standard conversions affect fundamental data types, and allow conversions such as the conversions between numerical types (short to int, int to float, double to int...), to or from bool, and some pointer conversions. Some of these conversions may imply a loss of precision, which the compiler can signal with a warning. This warning can be avoided with an explicit conversion.

Implicit conversions also include constructor or operator conversions, which affect classes that include specific constructors or operator functions to perform conversions. For example:

```
class A {};
class B { public: B (A a) {} };

A a;
B b=a;
```

Here, an implicit conversion happened between objects of class A and class B, because B has a constructor that takes an object of class A as parameter. Therefore implicit conversions from A to B are allowed.

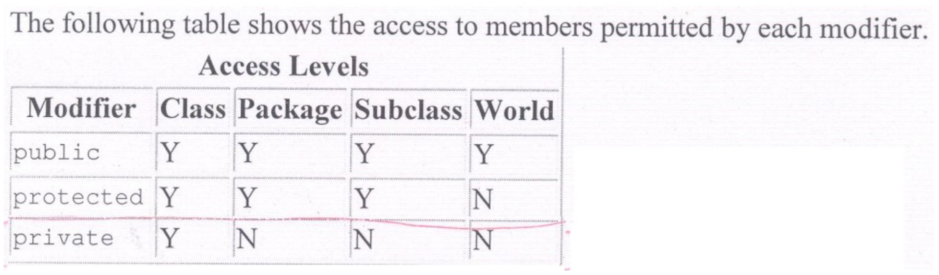
C++ is a strong-typed language. Many conversions, specially those that imply a different interpretation of the value, require an explicit conversion. We have already seen two notations for explicit type conversion: functional and c-like casting:

```
short a=2000;
int b;
b = (int) a;    // c-like cast notation
b = int (a);    // functional notation
```

The functionality of these **explicit conversion** operators is enough for most needs with fundamental data types. However, these operators can be applied indiscriminately on classes and pointers to classes, which can lead to code that while being syntactically correct can cause runtime errors.

c. **Explain the scope of private, public and protected member function. (6)**

Answer:

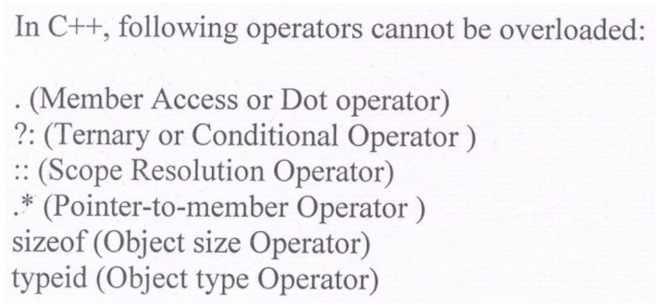

The following table shows the access to members permitted by each modifier.

Access Levels

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
private	Y	N	N	N

Q.5 a. Which operators cannot be overloaded? Write steps to overload + operator so that it can add two complex numbers. (8)

Answer:


In C++, following operators cannot be overloaded:

. (Member Access or Dot operator)
?: (Ternary or Conditional Operator)
:: (Scope Resolution Operator)
.* (Pointer-to-member Operator)
sizeof (Object size Operator)
typeid (Object type Operator)

Refer text Book of remaining part of the question.

b. **Write a program in C++ that display entered string into reverse order. (8)**

Answer:

Q.6 a. What is base class? How is it relevant in multiple inheritances? Does a constructor/destructor also inherited from base class to its derived class? (8)

Answer:

b. **What is the difference between ':' and "::" operator? Explain the concept using a suitable example. (8)**

Answer: Refer the scope section of Text Book.

Q.7 a. Define polymorphism. Write a program to demonstrate implementation of polymorphism. (8)

Answer: Refer to the prescribed text book.

b. Explain the working of the following program code: (8)

```
#include <iostream>
using namespace std;
double division(int a, int b)
{
   if( b == 0 ) {throw "Division by zero condition!";}return (a/b);
}
int main ()
{
int x = 50;
int y = 0;
double z = 0;
      try {
            z = division(x, y);
            cout << z << endl;
      } catch (const char* msg) { err << msg << endl;}
return 0;
}
```

Answer:

This is a program for exception handling when "divide by zero" situation arises.

Q.8 a. What is difference between opening a file with constructor function and with open( ) function? Explain your answer with a suitable example. (8)

Answer:

In case of constructor, the file can be accessed through methods defined in class and from outside, it can be accessed according to the defined scope of the function and data type (e.g. private, public and protected). However in case of through files opened directly through open () function, it can be accessed from anywhere in the program.

A suitable example will complete the answer.

b. What is Standard Template Library? How is it different from the C++ Standard Library? (8)

Answer:

(b) The Standard Template Library, or *STL*, is a C++ library of container classes, algorithms, and iterators; it provides many of the basic algorithms and data structures of computer science. The STL is a *generic* library, meaning that its components are heavily parameterized: almost every component in the STL is a template. You should make sure that you understand how templates work in C++ before you use the STL.

One very important question to ask about any template function, not just about STL algorithms, is what the set of types is that may correctly be substituted for the formal template parameters. Clearly, for example, `int*` or `double*` may be substituted for `find`'s formal template parameter `InputIterator`. Equally clearly, `int` or `double` may not: `find` uses the expression `*first`, and the dereference operator makes no sense for an object of type `int` or of type `double`. The basic answer, then, is that `find` implicitly defines a set of requirements on types, and that it may be instantiated with any type that satisfies those requirements. Whatever type is substituted for `InputIterator` must provide certain operations: it must be possible to compare two objects of that type for equality, it must be possible to increment an object of that type, it must be possible to dereference an object of that type to obtain the object that it points to, and so on.

`Find` isn't the only STL algorithm that has such a set of requirements; the arguments to `for_each` and `count`, and other algorithms, must satisfy the same requirements. These requirements are sufficiently important that we give them a name: we call such a set of type requirements a *concept*, and we call this particular concept **Input Iterator**. We say that a type *conforms to a concept*, or that it *is a model of a concept*, if it satisfies all of those requirements. We say that `int*` is a model of **Input Iterator** because `int*` provides all of the operations that are specified by the **Input Iterator** requirements.

Concepts are not a part of the C++ language; there is no way to declare a concept in a program, or to declare that a particular type is a model of a concept. Nevertheless, concepts are an extremely important part of the STL. Using concepts makes it possible to write programs that cleanly separate interface from implementation: the author of `find` only has to consider the interface specified by the concept **Input Iterator**, rather than the implementation of every possible type that conforms to that concept. Similarly, if you want to use `find`, you need only to ensure that the arguments you pass to it are models of **Input Iterator.** This is the reason why `find` and `reverse` can be used with `list`s, `vector`s, C arrays, and many other types: programming in terms of concepts, rather than in terms of specific types, makes it possible to reuse software components and to combine components together.

People learning C++ for the first time *do not* know this distinction, and may not notice small language differences. The entire C++ Standard Library *is* the STL, as believed some computer professionals. However there are features that were never part of the STL itself. Most vocal proponents of "the STL", in contrast, know exactly what they mean by it and refuse to believe that not everybody "gets it". Clearly, the term's usage is not uniform.

In addition, there are some STL-like libraries that are in fact implementations of the original STL, not the C++ Standard Library. Until recently, STLPort was one of them (and even there, the confusion abounds!).

Further, the C++ Standard does not contain the text "STL" anywhere, and some people habitually employ phrases like "the STL is *included* in the C++ Standard Library", which is plain incorrect.

Q.9 Write short note on any <u>TWO</u> of the followings: (8×2)

(a) Exception Handling
(b) Class template
(c) I/O Streams and its handling

Answer:

(a) *Exception handling* is a mechanism that separates code that detects and handles exceptional circumstances from the rest of your program. Note that an exceptional circumstance is not necessarily an error.

When a function detects an exceptional situation, you represent this with an object. This object is called an *exception object*. In order to deal with the exceptional situation you *throw the exception*. This passes control, as well as the exception, to a designated block of code in a direct or indirect caller of the function that threw the exception. This block of code is called a *handler*. In a handler, you specify the types of exceptions that it may process. The C++ run time, together with the generated code, will pass control to the first appropriate handler that is able to process the exception thrown. When this happens, an exception is *caught*. A handler may *rethrow* an exception so it can be caught by another handler.

The exception handling mechanism is made up of the following elements:

- try blocks
- catch blocks
- throw expressions
- Exception specifications

(b) The relationship between a class template and an individual class is like the relationship between a class and an individual object. An individual class defines how a group of objects can be constructed, while a class template defines how a group of classes can be generated.

Note the distinction between the terms *class template* and *template class*:

Class template

is a template used to generate template classes. You cannot declare an object of a class template.

Template class

is an instance of a class template.

A template definition is identical to any valid class definition that the template might generate, except for the following:

- The class template definition is preceded by
  ```
  template< template-parameter-list >
  ```
  where *template-parameter-list* is a comma-separated list of one or more of the following kinds of template parameters:
  - type
  - non-type
  - template
- Types, variables, constants and objects within the class template can be declared using the template parameters as well as explicit types (for example, int or char).

A class template can be declared without being defined by using an elaborated type specifier. For example:

```
template<class L, class T> class Key;
```

This reserves the name as a class template name. All template declarations for a class template must have the same types and number of template arguments. Only one template declaration containing the class definition is allowed.

Note:

When you have nested template argument lists, you must have a separating space between the > at the end of the inner list and the > at the end of the outer list. Otherwise, there is an ambiguity between the extraction operator >> and two template list delimiters >.

```
template<class L, class T> class Key { /* ... */};
template<class L> class Vector { /* ... */ };

int main ()
{
  class Key <int, Vector<int> > my_key_vector;
  // implicitly instantiates template
}
```

Objects and function members of individual template classes can be accessed by any of the techniques used to access ordinary class member objects and functions. Given a class template:

(c) One of the great strengths of C++ is its I/O system, IO Streams. As Bjarne Stroustrup says in his book "The C++ Programming Language", "Designing and implementing a general input/output facility for a programming language is notoriously difficult". He did an excellent job, and the C++ IOstreams library is part of the reason for C++'s success. IO streams provide an incredibly flexible yet simple way to design the input/output routines of any application.

IOstreams can be used for a wide variety of data manipulations thanks to the following features:

- A 'stream' is internally nothing but a series of characters. The characters may be either normal characters (char) or wide characters (wchar_t). Streams provide you with a universal character-based interface to any type of storage medium (for example, a file), without requiring you to know the details of how to write to the storage medium. Any object that can be written to one type of stream, can be written to all types of streams. In other words, as long as an object has a stream representation, any storage medium can accept objects with that stream representation.

- Streams work with built-in data types, and you can make user-defined types work with streams by overloading the insertion operator (<<) to put objects into streams, and the extraction operator (>>) to read objects from streams.
- The stream library.s unified approach makes it very friendly to use. Using a consistent interface for outputting to the screen and sending files over a network makes life easier. The programs below will show you what is possible.

The IO stream class hierarchy is quite complicated, so rather than introduce you to the full hierarchy at this point, I'll start with explaining the concepts of the design and show you examples of streams in action. Once you are familiar with elements of the design and how to apply those concepts to design a robust I/O system for your software, an understanding of what belongs where in the hierarchy will come naturally.

TEXT BOOK

I. Object-oriented Programmeming with C++, Poornachandra Sarang, PHI, 2004