**Question Set 2**

**The following questions are from w3schools.com and are the C++ tutorials. They come from:**

C++ Encapsulation

C++ Inheritance

C++ Polymorphism

C++ Exceptions

C++ References

C++ Pointers

**They are in order. If you start with functions and work from there, you should be fine.**

**All questions are potential quiz and exam questions.**

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.

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)