

[Dashboard](#) / [My courses](#) / [RSE1202s23-a.sg](#) / [29 January - 4 February](#) / [Quiz 6: Review of C++ Classes](#)

Started on	Tuesday, 7 February 2023, 9:03 PM
State	Finished
Completed on	Tuesday, 7 February 2023, 9:33 PM
Time taken	29 mins 42 secs
Grade	72.00 out of 72.00 (100%)

To keep code fragments small, it may be necessary to sometimes remove headers from the code. Don't provide ***doesn't compile*** as a valid answer only because certain headers are NOT included in a code fragment. Instead, assume all necessary headers are included in the code fragment.

C++ is a multi-paradigm language, and the paradigm that is most strongly associated with C++ is Object-Oriented Programming (OOP). OOP is a style of programming focused on the design and use of classes and class hierarchies. A *class* is a user-defined type that models a concept or a thing using data members, member functions, and member types. An *object* is an instantiation of such a class that represents an initialized region of memory holding values associated with the data members of that class type.

The basic principles of OOP related to C++ are:

- Abstraction: is the design strategy that concentrates on using things through a well designed interface without knowing anything about their implementation. Classes support abstraction by providing mechanisms for implementing both interface and implementation.
- Encapsulation: denotes the hiding of implementation details. Classes provide mechanisms for enforcing separation between interface and implementation.
- Inheritance: is focused on modeling concepts using classes and class hierarchies.
- Run-time polymorphism: is the manipulation of an unknown instance of an object in a class hierarchy. The actual type of the object is not resolved in principle until runtime at each particular point of execution.

The use of classes for implementing abstraction and encapsulation are the subject of this reading quiz. You should read the handouts related to classes plus Chapter 7, Sections 13.1, 14.1-14.6 from the textbook.

All of you immediately understand what I mean by the term "alarm clock" even though there probably isn't an alarm clock nearby. Why is that? You have seen many alarm clocks in your life and realized that all alarm clocks share certain attributes such as a time, an alarm time [both displayed in terms of hours and minutes], and a designation as to whether the alarm is on or off. You also realize that all alarm clocks you've seen allow you to set their time, set their alarm time, and turn the alarm on and off. In effect, you now have a concept, called *alarm clock*, which captures the notion of data and behavior of all alarm clocks in one tidy package. This concept is known as an *abstract data type* [ADT]. Usually, we think of an ADT as having an *interface* plus an *implementation*. The interface is that portion of the ADT definition that can be directly accessed by clients. The implementation is that part of the ADT definition that its clients access only indirectly through the interface.

In C++, a concept such as alarm clock can be implemented as an ADT using keyword **class**. A class defines a new user-defined [abstract] data type. It is composed of built-in types, other user-defined types, and functions. The parts used to define the class are called *members*. A class can have zero or more members. For example:

```
// definition of class X
class X {
private:
    int m; // data member
public:
    int mf(int v) { int old = m; m = v; return old; } // member function
};
```

Members can be of various types. Most are either data members, which define the representation of an object of the class, or functions, which provide operations that will change the state of objects by manipulating their data members.

C++ enables data encapsulation through access specifiers **public**, **private**, and **protected**. The section of the class definition labeled **public**: specifies the interface that the class provides to its clients. Likewise, the section of class definition labeled **private** or **protected** specifies the implementation of the interface and is therefore not accessible to clients.

A class can have multiple public, protected, or private labeled sections. Each section remains in effect until either another section label or the closing right brace of the class body is seen. If no access specifier is provided in the class definition, it is as though the entire class definition is labeled **private**: with the consequence that the entire class definition is an implementation.

```
class A {
public:
    // public members go here

protected:
    // protected members go here

private:
    // private members go here
};
```

To recap: a public member is accessible to clients of the class; a private member is accessible only to other members of the class and inaccessible to clients of the class; a protected member is like a private member but accessible to derived class. You will learn about protected members during the discussion on inheritance and object orientation.

Excluding protected members, a class definition in HLP2 has the following general format:

```

class X {
public:
    // public members - the interface to users [accessible by all]
    // functions
    // types
private:
    // private members - the implementation details [used by members of this class only]
    // functions
    // types
    // data
};

```

Members of a class [both data and functions] can be accessed using the member access operator `..`. A public member can be accessed from outside the class anywhere within the scope of the class object. It is important to note that private and protected members cannot be accessed directly using operator `..`.

Earlier, we spoke of the concept of an alarm clock with its attendant behavior represented in C++ by an alarm clock class. The physical alarm clock you hold in your hand is an object [or instance] of the alarm clock class. An alarm clock object is said to be instantiated from the alarm clock class, while the alarm clock class is said to be the generalization of all alarm clock objects you have encountered. In C++, objects are variables of a class data types. Objects are defined in declaration statements just as one would define variables of basic types or structs. The memory requirements to represent a class object in general are the accumulated size of its data members plus any padding due to alignment constraints imposed by the CPU.

A class member function is a function that is [only] declared in the class definition or declared and defined in the class definition. A member function that is only declared in a class definition must be defined outside the class with a fully qualified name that includes the class name and scope operator `::`. Class **A** declares and defined member function **print** while class **B** only declares member function **print** in the class definition and must therefore define the function outside the class definition:

```

class A {
    void print() {
        std::cout << "Hello World!";
    }
};

class B {
    void print();
};

void B::print() {
    std::cout << "Hello World!";
}

```

Question **1**

Correct

Mark 1.00 out of 1.00

In C++, a **struct** is similar to a **class** except that members of a **struct** have **public** access by default while members of a **class** have **private** access by default.

Select one:

- ☒ True ✓
- ☐ False

The correct answer is 'True'.

Question **2**

Correct

Mark 1.00 out of 1.00

What types of data members are *always* accessible to clients of a **class**?

Select one or more:

- ☐ **protected** data members that are non-**static**
- ☐ **static** data members that are **private**
- ☐ **private** data members that are non-**static**
- ☒ **public** data members that are non-**static** ✓
- ☒ **static** data members that are **public** ✓

Your answer is correct.

Usually, we think of a class as having an interface plus an implementation. The interface is the part of the class definition that its clients access directly. The implementation is that part of the class definition that its clients access only indirectly through the interface. The public interface is identified by the label **public:** and the implementation by the label **private:**. Just like any other class member, **static** members are also bound by **public:** and **private:** labels.

The correct answers are: **public** data members that are non-**static**, **static** data members that are **public**

Question **3**

Correct

Mark 1.00 out of 1.00

Which is true of specifying **private:** and **public:** labels in a class definition?

Select one:

- ☐ The order must be **private:** label followed by **public:** label
- ☐ The order is irrelevant, but each term can only be used once
- ☐ None of the listed choices
- ☐ The order must be: **public:** label followed by **private:** label
- ☒ They can be used anywhere, as many times as desired, and in any order ✓

Your answer is correct.

The correct answer is: They can be used anywhere, as many times as desired, and in any order

Question **4**

Correct

Mark 1.00 out of 1.00

Class member functions that are declared immediately after a **private:** label can only access private data members of the class.

Select one:

- ☐ True
- ☒ False ✓

The correct answer is 'False'.

Question **5**

Correct

Mark 1.00 out of 1.00

A *constructor* is a special member function of a class that is automatically executed whenever objects of that class are defined [on the stack or by a **new** expression]. Constructors have the exact same name as the class and do not specify a return type. Constructors are necessary for automatically initializing [note the use of the term initializing rather than assigning] data members when an object is defined.

A destructor is a special member function of a class that is executed whenever a class object goes out of scope or whenever the **delete** expression is applied to a pointer of that class type. A destructor will have the exact same name as the class with a **~** prefix and it can neither return a value nor can it take any parameters. This means that a class can only declare a single destructor function. The single destructor is a counterpart to [possibly many] constructors and is necessary for automatically releasing resources such as memory, network sockets, and files when class objects go out of scope.

Given the definition of type **A**:

```
class A {  
public:  
    A() { std::cout << "A()"; } // definition of constructor  
    ~A() { std::cout << "~A()"; } // definition of destructor  
};
```

what is the output of the following code fragment:

```
int main() {  
    A a, b;  
}
```

Answer: A()A()~A()~A()



The correct answer is: A()A()~A()~A()

Question **6**

Correct

Mark 1.00 out of 1.00

The data members in a **class** definition must have the same type.

Select one:

- ☐ True
- ☒ False ✓

The correct answer is 'False'.

Question **7**

Correct

Mark 1.00 out of 1.00

Member functions in a **class** definition must be declared after a **public:** label.

Select one:

- ☐ True
- ☒ False ✓

The correct answer is 'False'.

Question **8**

Correct

Mark 1.00 out of 1.00

A **class** definition can declare more than one constructor.

Select one:

- ☒ True ✓
- ☐ False

The correct answer is 'True'.

Question **9**

Correct

Mark 1.00 out of 1.00

A **class** definition can declare multiple destructors.

Select one:

- ☐ True
- ☒ False ✓

The correct answer is 'False'.

Question **10**

Correct

Mark 1.00 out of 1.00

Both constructor and destructor functions can have parameters.

Select one:

- ☐ True
- ☒ False ✓

Constructors can be overloaded and can therefore have a range of parameters including no parameters. On the other hand, every class can have one and only one destructor (the compiler will synthesize a destructor if one is not declared in the class definition) that doesn't take any parameters.

The correct answer is 'False'.

Question **11**

Correct

Mark 1.00 out of 1.00

Certain constructors are conversion functions that can only be used to convert objects of other types to the class type.

Select one:

- ☒ True ✓
- ☐ False

The correct answer is 'True'.

Question **12**

Correct

Mark 1.00 out of 1.00

Prefixing a single-argument conversion constructor with keyword **explicit** prevents implicit type conversion to the class type using that specific constructor.

Select one:

- ☒ True ✓
- ☐ False

The correct answer is 'True'.

Question **13**

Correct

Mark 1.00 out of 1.00

The synthesized default constructor for the following class does nothing.

```
class Y {
public:
    // no constructors are declared
private:
    int i;
    std::string s;
    std::vector<int> v;
};
```

Select one:

- ☐ True
- ☒ False ✓

A constructor has an **initialization part** and a **function body**. The *initialization part* initializes each data member before the *function body* is executed, and members are initialized in the same order as they are declared in the class. The initialization part uses a *constructor member initializer list* to initialize data members. If the programmer doesn't author a constructor member initializer list, the compiler will synthesize a constructor member initializer list that will default initialize the data members. Default initialization of a data member of built-in type means the data member will have an unspecified value. Default initialization of a data member of user-defined type will take place by calling its default constructor or by synthesizing a default constructor for the data member. If the user-defined type doesn't define a default constructor and the compiler is unable to synthesize one, the compiler will flag an error.

Consider the following class **Y** that doesn't declare any constructors:

```
class Y {
public:
    // no constructors are declared
private:
    int i;
    std::string s;
    std::vector<int> v;
};
```

When an object of type **Y** is defined

```
Y y;
```

the compiler will synthesize a default constructor that will have an initialization part and a function body. The initialization part will consist of the following ordered steps:

- 1. Default initialize data member **i** which means nothing is done since **i** is a built-in data type.
- 2. Default initialize data member **s** using default constructor **std::string::string()** of class **std::string**.
- 3. Default initialize data member **v** using default constructor **std::vector<int>::vector<int>()** of class **std::vector<int>**.

The function body of the synthesized default constructor will be empty.

Now, consider the following definitions of classes **X** and **Y**:

```
class X {
public:
    X(double x);
    // no other constructors are declared
private:
    double d;
};

class Y {
public:
    // no constructors nor a destructor are declared
private:
    int i;
    std::string s;
    std::vector<int> v;
    X x;
};
```

When an object of type **Y** is defined

```
Y y;
```

the compiler will synthesize a default constructor that will have an initialization part and a function body. The initialization part will consist of the following ordered steps:

1. Default initialize data member **i** with an unspecified value since **i** is a built-in data type.
2. Default initialize data member **s** using default constructor **std::string::string()** of class **std::string**.
3. Default initialize data member **v** using default constructor **std::vector<int>::vector<int>()** of class **std::vector<int>**.
4. Default initialize data member **x** using default constructor **x::X()** of user-defined type **x**. However, class **x** doesn't declare a default constructor. In addition, the compiler is unable to synthesize a default constructor because class **x** declares a constructor **x::X(double)**.

Since the compiler is unable to default initialize data member **x**, the compiler will flag the definition

```
Y y;
```

as a compile-time error.

Main takeaway: The constructor body doesn't directly initialize the members themselves. Members are initialized as part of the initialization phase that precedes the constructor body. A constructor body executes in addition to the member-wise initialization that takes place as part of an object's initialization. In short, if a constructor body is empty, it is not true that the constructor is not doing anything.

The correct answer is 'False'.



Question **14**

Correct

Mark 1.00 out of 1.00

The synthesized destructor for the following class does nothing.

```
class Y {  
public:  
    // a destructor is not declared  
private:  
    int i;  
    std::string s;  
    std::vector<int> v;  
};
```

Select one:

- ☐ True
- ☒ False ✓

Just as a constructor has an *initialization part* and a *function body*, a destructor has a **function body** and a **destruction part**. In a constructor members are initialized before the function body is executed, and members are initialized in the same order as they're declared in the class. In a destructor, the function body is executed first and then the members are destroyed in the reverse order from the order in which they're initialized.

The function body of a destructor does whatever operations the class designer wishes to have executed subsequent to the last use of an object. Typically, the destructor frees resources an object allocated during its lifetime.

In a destructor, there is nothing akin to the constructor initializer list to control how members are destroyed; the destruction part is implicit. What happens when a member is destroyed depends on the type of the member. Members of class type are destroyed by running the member's own destructor.

If a destructor is not defined by a class, the compiler synthesizes one only if the class contains a member of a user-defined type that declares a destructor. Otherwise, the destructor is considered to be trivial and is therefore neither synthesized nor invoked in practice. The synthesized destructor has an empty function body. The members are automatically destroyed after the empty destructor body is run.

Consider the following code fragment that initializes a variable **y**:

```
class Y {  
public:  
    // a destructor is not declared  
private:  
    int i;  
    std::string s;  
    std::vector<int> v;  
};  
  
int main() {  
    Y y;  
} // y goes out of scope here ...
```

When object **y** goes out of scope, the compiler will synthesize a destructor since class **Y** doesn't declare a destructor. The synthesized destructor will consist of an empty function body followed by the destruction part that destructs the data members in the reverse order in which the constructor of class **Y** has initialized them. Therefore, data member **v** of variable **y** is destroyed using destructor **std::vector<int>::~~vector<int>()** followed by the destruction of data member **s** using destructor **std::string::~~string()** followed by the destruction of data member **i** of built-in type [which doesn't do anything].

Main takeaway: The destructor body doesn't directly destroy the members themselves. Members are destroyed as part of the implicit destruction phase that follows the destructor body. A destructor body executes in addition to the member-wise destruction that takes place as part of destroying an object. Therefore, an empty destructor body doesn't mean that the destructor is not doing anything.

The correct answer is 'False'.

Question **15**

Correct

Mark 1.00 out of 1.00

To convert a single value of a certain type to a class type, the class must implement a

Select one:

- ☐ conversion operator member function
- ☐ default constructor
- ☐ copy constructor
- ☒ constructor that can be called with a single argument ✓
- ☐ copy assignment operator function

Your answer is correct.

The correct answer is: constructor that can be called with a single argument

Question **16**

Correct

Mark 1.00 out of 1.00

Consider the definition of type **Person**:

```
struct Person {  
    char name[100];  
    int ID;  
    double weight, height;  
};
```

After reviewing Section 7.1.4 of the text, do you agree that type **Person** does not declare any constructors?

Select one:

- ☒ True ✓
- ☐ False

The correct answer is 'True'.

Question **17**

Correct

Mark 1.00 out of 1.00

Based on the following definition of type **Person**, which of the following is true about constructors when applied to type **Person**? Read Section 7.1.4 of the text before answering the question.

```
struct Person {  
    char name[100];  
    int ID;  
    double weight, height;  
};
```

Select one or more:

- ☒ The responsibility of constructor(s) for type **Person** is to initialize the data members of an object of type **Person**. ✓
- ☒ A constructor for type **Person** should be a member function. ✓
- ☒ If overloaded constructor(s) are declared for type **Person**, an appropriate constructor will be called when an object of type **Person** object is defined. ✓
- ☒ If no constructors are declared for type **Person** and if a default constructor is necessary, the compiler will synthesize a default constructor. ✓

Your answer is correct.

The correct answers are: A constructor for type **Person** should be a member function., If overloaded constructor(s) are declared for type **Person**, an appropriate constructor will be called when an object of type **Person** object is defined., The responsibility of constructor(s) for type **Person** is to initialize the data members of an object of type **Person**., If no constructors are declared for type **Person** and if a default constructor is necessary, the compiler will synthesize a default constructor.

Question **18**

Correct

Mark 1.00 out of 1.00

In the following code fragment, determine whether the compiler will synthesize a default constructor to initialize object **P**. Read Section 7.1.4 before answering this question.

```
struct Person {  
    char name[100];  
    int ID;  
    double weight, height;  
};  
  
// in function main ...  
Person P;
```

Select one:

- ☒ True ✓
- ☐ False

The C++ standard has this to say:

A *default constructor* for a class **x** is a constructor of class **x** for which each parameter has a default argument [including the case of a constructor with no parameters]. If there is no user-declared constructor for class **x**, a non-explicit constructor having no parameters is implicitly declared as **defaulted**. An implicitly-declared default constructor is an inline public member of its class.

The correct answer is 'True'.

Question **19**

Correct

Mark 1.00 out of 1.00

If not explicitly provided by the class, which functionality is synthesized by the compiler?

Select one:

- ☐ copy assignment operator overload, conversion operator overload, subscript operator overload, and accessor [that only read data members] functions
- ☒ default constructor, copy constructor, destructor, and copy assignment operator overload ✓
- ☐ no default functionality is provided
- ☐ default constructor, destructor, and conversion operator overload
- ☐ subscript operator overload, conversion operator overload, and destructor

Your answer is correct.

The correct answer is: default constructor, copy constructor, destructor, and copy assignment operator overload

Question **20**  
Correct  
Mark 1.00 out of 1.00

The *copy constructor* is a constructor which creates an object by initializing it with an object of the same class which has been created previously.

If a copy constructor is not defined in a class, the compiler itself defines "shallow" or member-wise copy constructor. This means that each member of the class individually. When classes are simple (e.g. do not contain any dynamically allocated memory), this works very well. If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a custom made copy constructor for the class.

Given the definition of type **A**:

```
class A {
    int n;
public:
    A() {n = 10;}           // default ctor
    A(A const& a) { n = a.n + 10; } // copy ctor
    int N() const { return n; }    // accessor
};
```

what is the output of the following code fragment? Write NC if the code fragment doesn't compile.

```
// in function main ...
A a, b(a);
std::cout << a.N() << ',' << b.N() << ',' << A(b).N();
```

Answer:

In the statement

```
std::cout << a.N() << ',' << b.N() << ',' << A(b).N();
```

the expression **A(b).N()** evaluates to an unnamed temporary object which calls member function **N**.

The correct answer is: 10,20,30

Question **21**  
Correct  
Mark 1.00 out of 1.00

A copy constructor must have two parameters: a reference to the class that is being copied from and a reference to the class that is being copied to.

Select one:

☐ True

☒ False

By definition, a copy constructor is a constructor requiring a single argument of its class type. The copy constructor for any class **x** is declared as:

```
X::X(X const& x);
```

It is possible for a copy constructor to take multiple arguments provided each second and subsequent argument is provided with a default value. So for any class **y**, the following is a valid copy constructor:

```
Y::Y(Y const& y, int = 0);
```

The correct answer is 'False'.

Question **22**

Correct

Mark 1.00 out of 1.00

In previous questions, we provided values to data members in the constructor's body using the assignment operator. As we know, **const** and reference variables must be initialized when they are defined. Since initialization of data members is complete by the time the constructor's body begins execution, it is not possible to initialize **const** and reference data members in the constructor's body

To solve this problem, C++ provides a method for initializing class data members [rather than assigning values to them after they are created] via a *constructor member initializer list* [often called a *constructor initialization list*].

In the constructor's definition, the member initializer list is inserted after the constructor's signature. It begins with a colon **:** and then lists each variable to initialize along with the value for that variable separated by a comma.

The order in which data members are initialized in the member initializer list doesn't matter - data members are *always* initialized in the order in which they're declared in the class.

Given definition of type **A**:

```
class A {  
    int i;  
    double d;  
    char c;  
public:  
    A() : i {1}, d(2.2), c {'c'} {}  
    void print() { std::cout << i << ',' << d << ',' << c; }  
};
```

what is the output of the following code fragment? Write NC if the code doesn't compile.

```
// in function main ...  
A *pa{new A};  
pa->print();  
delete pa;
```

Answer: 1,2.2,c



Although the statement:

```
A *pa{new A};
```

may seem to be a single operation, it is actually accomplishing two execution steps:

1. The allocation of the requested memory by evaluating **new** expression. The **new** expression is provided the size in bytes of the object: **A \*pa = new (sizeof(A));**
2. The initialization of the allocated object: **\*pa = A();**

So, after the execution of the statement, **pa** is pointing to an object of type **A** on the free store that is default constructed. The statement

```
pa->print();
```

will print the state of the object pointed to by **pa** on the free store. Following that the statement

```
delete pa;
```

will first invoke the destructor synthesized by the compiler [that doesn't do anything because data members of **A** have built-in types] followed by returning the memory back to the free store.

The correct answer is: 1,2.2,c

Question **23**

Correct

Mark 1.00 out of 1.00

Data members are initialized in the order they appear in a constructor's member initializer list.

Select one:

- ☐ True
- ☒ False ✓

There is a subtlety to note here. The order in which the list entries are executed is determined by the declaration order of the members with the class declaration, not the order within the initialization list. This apparent anomaly between initialization order and order within the initialization list can lead to the following nasty pitfall:

```
class X {
    int i, j;
public:
    // oops!!! do you see the problem?
    X(int val) : j{val}, i{j} {}
    // other stuff ...
};
```

The correct answer is 'False'.

Question **24**

Correct

Mark 1.00 out of 1.00

What kind of class data members listed below must only be initialized using a constructor member initializer list?

Select one or more:

- ☐ pointer data members
- ☐ data members of fundamental data types
- ☒ data members of user-defined types that are initialized with a set of arguments ✓
- ☒ **const** data members ✓
- ☐ data members of user-defined types that are initialized by default constructors
- ☒ reference data members ✓

Your answer is correct.

You must use the member initialization list in the following cases in order for your program to compile:

1. When initializing a reference member
2. When initializing a **const** member
3. When invoking a base or member class constructor with a set of arguments

User-defined members with default constructors are not part of the answer since such members can be implicitly initialized by the compiler without requiring them to be listed in the initialization list.

The correct answers are: reference data members, **const** data members, data members of user-defined types that are initialized with a set of arguments

Question **25**

Correct

Mark 5.00 out of 5.00

Determine the values printed to the standard output stream by the statements in function `main`. Read the explanation of default initialization in Section 2.2 of the textbook and the explanation of in-class initializers in Section 2.6.1 before answering this question.

```
int x;

namespace X {
    int x;
}

struct S {
    int i{100};
    double y;
};

int main() {
    S s;
    int x;
    std::cout << ::x; // Label 1
    std::cout << X::x; // Label 2
    std::cout << x; // Label 3
    std::cout << s.i; // Label 4
    std::cout << s.y; // Label 5
}
```

Value printed by statement labeled <b>Label 2</b> :	<input type="text" value="0"/>	✓
Value printed by statement labeled <b>Label 3</b> :	<input type="text" value="Value is uninitialized"/>	✓
Value printed by statement labeled <b>Label 5</b> :	<input type="text" value="Value is uninitialized"/>	✓
Value printed by statement labeled <b>Label 4</b> :	<input type="text" value="100"/>	✓
Value printed by statement labeled <b>Label 1</b> :	<input type="text" value="0"/>	✓

Your answer is correct.

Instead of initializing data members in the member initializer list of a constructor, C++11 introduced in-class initializers for data members:

```
class Date {
public:
    Date(int yy); // January 1 of year yy
private:
    int m{1}, d{1}, y{2001};
};
```

In the initialization part of a `Date` constructor, data members `m`, `d`, and `y` will be initialized to values `1`, `1`, and `2001`, respectively through an implicit member initializer list. Furthermore, an explicit member initializer list can be used to initialize these data members to values different from their in-class initializers. For example, the single-argument constructor can be defined to initialize data member `y` to the value specified by the constructor's parameter:

```
Date::Date(int yy) : y{yy} {}
```

The correct answer is: Value printed by statement labeled **Label 2**: → 0, Value printed by statement labeled **Label 3**: → Value is uninitialized, Value printed by statement labeled **Label 5**: → Value is uninitialized, Value printed by statement labeled **Label 4**: → 100, Value printed by statement labeled **Label 1**: → 0

Question **26**

Correct

Mark 5.00 out of 5.00

Determine the values printed to standard output stream by each of the labeled statements. If no values are printed by a statement, choose "Nothing printed". Answer this question after reading about default initialization and in-class initializers in Sections 2.2 and 2.6.1 of the textbook, respectively.

```
struct P {
    P() { std::cout << "P::P()"; }
    int x{90};
    double y;
};

struct Q {
    P p;
    float f{9.F};
    short s{10};
};

// in function main ...
Q q;           // Label 1
std::cout << q.p.x; // Label 2
std::cout << q.p.y; // Label 3
std::cout << q.f;   // Label 4
std::cout << q.s;   // Label 5
```

Value printed by statement labeled <b>Label 3</b>	Value is uninitialized	✓
Value printed by statement labeled <b>Label 5</b>	10	✓
Value printed by statement labeled <b>Label 4</b>	9	✓
Value printed by statement labeled <b>Label 1</b>	P::P()	✓
Value printed by statement labeled <b>Label 2</b>	90	✓

Your answer is correct.

The correct answer is: Value printed by statement labeled **Label 3** → Value is uninitialized, Value printed by statement labeled **Label 5** → 10, Value printed by statement labeled **Label 4** → 9, Value printed by statement labeled **Label 1** → P::P(), Value printed by statement labeled **Label 2** → 90



Question **27**

Correct

Mark 1.00 out of 1.00

Choose the most appropriate description of what is wrong with the following code fragment:

```
struct Str {
    char *c;
public:
    Str(char const *ps) : c(new char [std::strlen(ps)+1]) {
        std::cout << "Hello World";
        std::strcpy(c, ps);
    }
};

void foo(char const *ps) {
    Str s(ps);
    // other stuff of no interest to us ...
}

// in function main ...
for (int i{0}; i < 100; ++i) {
    foo("Today is a good day");
}
```

Select one:

- ☐ The code fragment doesn't compile because type **Str** doesn't declare a destructor
- ☐ The code fragment leaks the literal **"Hello World"**
- ☐ There is nothing wrong with the code fragment
- ☐ The code fragment doesn't compile because type **Str** doesn't declare a default constructor
- ☒ The code fragment leaks memory ✓

Your answer is correct.

The class designer has incorrectly not defined a destructor. The compiler will synthesize a destructor:

```
Str::~~Str() {}
```

As seen in the above definition, the synthesized version doesn't explicitly call **operator delete[]**. Therefore, every time an object of type **Str** is instantiated and then goes out of scope, it will cause a memory leak.

The correct answer is: The code fragment leaks memory

Question **28**

Correct

Mark 1.00 out of 1.00

As indicated in Section 13.1.3 of the textbook, the compiler will synthesize a destructor if a type doesn't explicitly define one. Write the definition of the synthesized destructor of type **Hello** outside the type definition and without using whitespace characters.

```
struct Hello {
    int who;
};
```

Answer:  ✓

The correct answer is: Hello::~~Hello(){}

Question **29**

Correct

Mark 5.00 out of 5.00

What is written to standard output stream by the following code fragment?

```
struct Obj {
    Obj() { std::cout << "A"; }
    ~Obj() { std::cout << "B"; }
};

struct Yolo {
    Yolo() { std::cout << "C"; }
    ~Yolo() { std::cout << "D"; }
};

int main() {
    Obj x;
    {
        Yolo y;
    }
    Yolo y;
    Obj z;
}
```

Answer: 

A simple rule of thumb for construction/destruction of objects defined on the stack is: ***Order of destruction is always the opposite of construction.***

Or said simply: ***First constructed, last destructed.***

```
class X {
    X(); // default constructor
    ~X(); // destructor
};

class Y {
    Y(); // default constructor
    ~Y(); // destructor
};

int main() {
    X x;
    Y y;
    // some other code ...
}
```

In function **main**, variable **x**'s constructor is called first followed by variable **y**'s constructor. When function **main** returns, variables **x** and **y** go out of scope. Since variable **y** was constructed after variable **x**, **y**'s destructor is called first followed by **x**'s destructor.

The correct answer is: ACDCABDB

Question **30**  
Correct  
Mark 1.00 out of 1.00

A *conversion constructor* is a single-argument constructor that is declared without the function specifier **explicit**. The compiler uses such conversion constructors to implicitly convert objects from the type of the parameter(s) to the type of the conversion constructor's class.

What is the output of the following code fragment? Write NC if the code doesn't compile.

```
class A {
    int n;
public:
    A(int m) : n(m) { std::cout << "A"; }
    int N() const { return n; }
};

// in function main ...
std::cout << A{5}.N();
```

Answer:  ✓

The correct answer is: A5

Question **31**  
Correct  
Mark 1.00 out of 1.00

What is the output of the following code fragment? Write NC if the code doesn't compile.

```
class A {
    int n;
public:
    A(int m) : n(m) { std::cout << "A"; }
    int N() const { return n; }
};

void foo(A const& r) {
    std::cout << r.N();
}

// in function main ...
foo(15);
```

Answer:  ✓

The compiler uses conversion constructor **A::A(int)** to convert **int** argument **15** in function call **foo(15)** to an unnamed **A** object on the stack. Next, the compiler will initialize parameter **r** in the definition of function **foo** as an read-only alias to this unnamed **A** object.

The correct answer is: A15

Question **32**  
Correct  
Mark 1.00 out of 1.00

After reading Section 7.5.4 of the textbook, choose statement(s) from the following list that are true:

Select one or more:

- ☐ A converting constructor is a constructor that converts a class type to a single argument.
- ☒ A converting constructor is a constructor that converts a single argument into a class type. ✓
- ☒ The compiler will only allow one class-type conversion. ✓
- ☐ The compiler will allow multiple class-type conversions.

Your answer is correct.

The correct answers are: A converting constructor is a constructor that converts a single argument into a class type., The compiler will only allow one class-type conversion.

Question **33**

Correct

Mark 1.00 out of 1.00

What is printed to standard output stream by the following code fragment? If the code doesn't compile, write NC as your answer.

```
class A {  
    int n;  
public:  
    A(int m) : n(m) { std::cout << "A"; }  
    ~A()           { std::cout << "~A"; }  
    int N() const  { return n; }  
};  
  
void foo(A const& a) {  
    std::cout << a.N();  
}  
  
int main() {  
    foo(5);  
}
```

Answer: A5~A



Consider the call to function **foo**:

```
foo(5);
```

Argument **5** in the call to **foo** will be used as the argument to constructor **A::A(int)** that will initialize an unnamed, temporary object of type **A** on the stack. This will cause character '**A**' to be printed to standard output. Further, **foo**'s reference parameter **a** is initialized to reference this unnamed, temporary object. In function **foo**, the value of data member **n** of the object referenced by **a** - which is **5** - will be printed to standard output. When function **foo** terminates, the unnamed, temporary object will go out of scope causing the automatic execution of its destructor. The destructor's body will write C-style string "**~A**" to standard output stream.

The correct answer is: A5~A

Question **34**

Correct

Mark 4.00 out of 4.00

Given the following code fragment, match each statement involving a call to function `foo` to the expected values written to standard output stream by that statement. Choose NC if a particular call to function `foo` doesn't compile.

```
struct Obj2 {
    Obj2(char*) { std::cout << "char*"; }
};

struct Obj1 {
    Obj1(int)    { std::cout << "int"; }
    Obj1(double) { std::cout << "double"; }
    Obj1(short)  { std::cout << "short"; }
    Obj1(Obj2)   { std::cout << "obj2"; }
};

void foo(Obj1 obj) {
    // empty by design ...
}
```

<code>foo(5);</code>	<input type="text" value="int"/>	✓
<code>short s{2}; foo(s);</code>	<input type="text" value="short"/>	✓
<code>char c{'a'}; foo(&amp;c);</code>	<input type="text" value="NC"/>	✓
<code>foo(5.0);</code>	<input type="text" value="double"/>	✓

Your answer is correct.

**The compiler will implicitly apply only one class-type conversion.** How does this apply here. Consider that:

- 1. Class `Obj2` defines a single-argument conversion constructor `Obj2::Obj2(char *)`;
- 2. Class `Obj1` defines a single-argument conversion constructor `Obj1::Obj1(Obj2)`;
- 3. Function `foo` takes an object of type `Obj1`.
- 4. In function `main`, `foo` is called with an argument of type `char*`, as in: `foo(&c)`;

The compiler could first transform the `char*` argument to an unnamed, temporary object of type `Obj2` [using `Obj2`'s conversion constructor]. This unnamed `Obj2` object can be transformed to an `Obj1` object [using `Obj1`'s conversion constructor]. The compiler can then use this unnamed `Obj1` object to initialize function `foo`'s parameter `obj` and successfully complete the call to `foo`.

However, the rule is only one conversion is allowed and since the process described above requires two conversions, the compiler will flag the statement `foo(&c)`; as an error.

To compile statement with `foo(&c)`; without an error, the programmer must explicitly specify `foo`'s argument `&c` [of type `char*`] to be converted to type `Obj2`:

```
foo(static_cast<Obj2>(&c));
```

Then, the rule of one conversion will kick in to convert the unnamed, temporary object of type `Obj2` to be converted to type `Obj1` using class `Obj1`'s single-argument conversion constructor `Obj1::Obj1(Obj2)`;

The correct answer is: `foo(5);` → int, `short s{2}; foo(s);` → short, `char c{'a'}; foo(&c);` → NC, `foo(5.0);` → double

Question **35**

Correct

Mark 4.00 out of 4.00

Given the following code fragment, match each statement involving a call to function `foo` to the expected values written to standard output stream by that statement. Choose NC if a particular call to function `foo` doesn't compile.

```
struct Obj2 {
    Obj2(char*) { std::cout << "char*"; }
};

struct Obj1 {
    explicit Obj1(int)    { std::cout << "int"; }
    Obj1(double)         { std::cout << "double"; }
    explicit Obj1(short) { std::cout << "short"; }
    Obj1(Obj2)           { std::cout << "obj2"; }
};

void foo(Obj1 obj) {
    // empty by design ...
}
```

<code>foo(5.0);</code>	double	✓
<code>short sh{2}; foo(sh);</code>	double	✓
<code>foo(5);</code>	double	✓
<code>char c{'c'}; foo(&amp;c);</code>	NC	✓

Your answer is correct.

Let's first consider why statement `foo(5);` writes `double` to the standard output stream. The compiler doesn't give up and flag an error when it evaluates the argument in expression `foo(5)` as type `int` and not of type `Obj1`. Instead, the compiler looks [starting from the scope of the function call and going outwards to higher scopes] for a function that can convert a value of type `int` to a value of type `Obj1`. The compiler finds a declaration of single-argument constructor `Obj1::Obj1(int);` that converts an `int` value to a value of type `Obj1`. However, since this constructor is declared `explicit`, the compiler cannot implicitly convert the `int` argument to an `Obj1` parameter. Compilers are hardworking and don't easily give up. Upon further search, the compiler finds a non-`explicit` constructor `Obj1::Obj1(double);`. The compiler will reason that expression `foo(5)` can be successfully compiled by first implicitly converting `foo`'s `int` argument `5` to type `double` [and value `5.0`] and then convert this `double` argument to an unnamed object of type `Obj1` through constructor `Obj1::Obj1(double);`.

A similar reasoning can be employed to understand why expression `foo(s)` successfully compiles.

Likewise, we now understand that expression `foo(5.0)` is evaluated by the compiler by implicitly converting argument `5.0` of type `double` to parameter of type `Obj1` using constructor `Obj1::Obj1(double);`.

Now, let's try to understand why statement `foo(&c);` will not compile. Applying the same reasoning as above, we could argue that the compiler could convert argument `&c` to an unnamed, temporary object of type `Obj2` which is then further converted to another unnamed, temporary object of type `Obj1`. However, the compiler will apply only one class-type conversion at a time. Here, two class-type conversions are required: first from `char*` to `Obj2`; and then from `Obj2` to `Obj1`. Since multiple class-type conversions are illegal, the compiler will flag the call `foo(&c)` as a compiler error.

The correct answer is: `foo(5.0);` → double, `short sh{2}; foo(sh);` → double, `foo(5);` → double, `char c{'c'}; foo(&c);` → NC

Question **36**

Correct

Mark 1.00 out of 1.00

Which of the following statements are true about type **Haha**?

```
struct Haha {  
    Haha(int = 5);  
};
```

Select one or more:

- ☒ Constructor declared in **struct Haha** is a converting constructor ✓
- ☒ An object of type **HaHa** can be defined like this: **HaHa h;** ✓
- ☒ An object of type **HaHa** can be defined like this: **Haha h(2.0f);** ✓
- ☒ Constructor declared in **struct Haha** is a default constructor ✓
- ☐ An object of type **HaHa** can be defined like this: **Haha h{2.0f};**
- ☒ An object of type **HaHa** can be defined like this: **Haha h{2};** ✓

Your answer is correct.

A **single argument conversion constructor** is a constructor that can called with only one argument. Such a constructor may declare only a single parameter or it may declare multiple parameters, with each parameter after the first having a default value.

The correct answers are: Constructor declared in **struct Haha** is a converting constructor, Constructor declared in **struct Haha** is a default constructor, An object of type **HaHa** can be defined like this: **HaHa h;**, An object of type **HaHa** can be defined like this: **Haha h(2.0f);**, An object of type **HaHa** can be defined like this: **Haha h{2};**

Question **37**

Correct

Mark 1.00 out of 1.00

A **const** member function can only access read-only [that is, constant] data members.

Select one:

- ☐ True
- ☒ False ✓

Variables are meant to be changed - that's why we call them "variables" - but some are not; that is, we have a variable representing an immutable value. That variable, we typically call *read-only*, *constant*, or just **const** variable. Consider:

```
class Date {
public:
    // assume appropriate constructors are declared
    int day(); // accessor that returns day of year
    void add_day(); // mutator to change date to tomorrow's date
};

void some_function(Date& d, Date const& start_of_term) {
    int a = d.day(); // ok
    int b = start_of_term.day(); // error but should be ok
    d.add_day();           // fine
    start_of_term.add_day(); // error
}
```

Here we intend parameter **d** to be mutable, but parameter **start\_of\_term** to be immutable; it is not acceptable for function **some\_function** to amend **start\_of\_term**. How would the compiler know that? It knows because we told it by declaring parameter **start\_of\_term** as **Date const&**. So far, so good, but then why is it not OK to read the current day of **start\_of\_term** using member function **day()**? As the definition of **Date** stands so far, the expression **start\_of\_term.day()** is in error because the compiler doesn't know that **day()** doesn't change the state of its **Date** object. We never told it, so the compiler assumes that member function **day()** may modify its **Date** object and therefore reports an error.

We can deal with this problem by classifying operations on a class as modifying and non-modifying. That's a pretty fundamental distinction that helps us to understand a class, but it also has a very practical importance: **operations that do not modify the object can be invoked for const objects**. For example:

```
class Date {
public:
    // assume appropriate constructors are declared
    int day() const; // const member function: can't modify the object
    void add_day(); // mutator to change date to tomorrow's date
};
```

We add specifier **const** right after the argument list in a member function declaration to indicate that the member function doesn't amend its object and can therefore be called for a **const** object. Once we've declared a member function **const**, the compiler holds us to our promise not to modify the object. For example, the compiler will not compile this version of the member function:

```
int Date::day() const {
    ++d; // error: attempt to change object from const member function
    return d;
}
```

Naturally, we don't usually try to cheat in this way. What the compiler provides for the class implementer is primarily protection against accident, which is particularly useful for more complex code.

The correct answer is 'False'.



Question **38**

Correct

Mark 1.00 out of 1.00

Given a type **S** that overloads operator **+** so that the following code fragment successfully compiles:

```
// in function main ...
S s;
int x = 1 + s;
```

specify which overload of operator **+** is chosen by the compiler.

Select one:

- ☒ `int operator+(int i, const S& s);` ✓
- ☐ `int int::operator+(const S& s) const;`
- ☐ `int operator+(S const& lhs, const S& rhs);`
- ☐ `int operator+(const S& s, int i);`
- ☐ `int S::operator+(int i) const;`

Your answer is correct.

**1** is a value of type **int** and **int** is a fundamental, built-in type. It is not a class and doesn't have an overloaded **operator+** to enable a call:

```
1.operator+(s)
```

Therefore, the expression **1 + s** must be evaluated as a call to a non-member function in the global scope: **operator+(1, s);**

The correct answer is: `int operator+(int i, const S& s);`

Question **39**

Correct

Mark 1.00 out of 1.00

Which of the following is *not* a reason to use operator overloading?

Select one:

- ☐ Overloaded operators put user defined types on the same footing as fundamental types
- ☐ Code with overloaded operators is easier to read and understand
- ☐ Overloaded operators are easier to remember and spell than functions
- ☒ Overloaded operators execute faster than functions ✓

Your answer is correct.

This is wrong: ~~Overloaded operators are faster than functions.~~

Overloaded operators are implemented using member or non-member functions to provide conventional notation for a type we design. Therefore, it is a misunderstanding to assume that overloaded operators are faster than functions.

The correct answer is: Overloaded operators execute faster than functions

Question **40**

Correct

Mark 1.00 out of 1.00

Consider the following type definitions:

```
struct Employee {
    std::string name;
};

struct Department {
    std::vector<Employee> members;
};
```

Suppose we are hiring a new employee and we wish to add this employee to a certain department like so:

```
Department department;
Employee new_hire {"Chris"};
department += new_hire;
```

What is the best declaration [best in terms of efficiency and for ensuring objects of type **Department** behave similar to built-in types] for **Department**'s member function that overloads operator **+=**?

Select one:

- ☐ `Department operator+=(Employee const& employee);`
- ☐ `Employee& operator+=(Employee const& employee);`
- ☐ `void operator+=(Employee const& employee);`
- ☐ `Employee operator+=(Employee const& employee);`
- ☐ `Department& operator+=(Employee const employee);`
- ☒ `Department& operator+=(Employee const& employee);` ✓

Your answer is correct.

We know that the copy assignment operator overload for class **Department** must be declared as a member function [if the class doesn't declare the function, the compiler will synthesize one]:

```
Department& Department::operator=(Department const&);
```

We've also seen that compound-assignment operators ordinarily ought to be declared as members. That is, since they're not synthesized by the compiler, they're not required to be members. However, in implementing a variant of class **std::string** in lectures, we've seen that they should be declared as member functions. To behave similar to built-in types, the compound-assignment operators should have the same form as the copy assignment operator:

```
Department& Department::operator+=(Department const&);
```

An overload of such a compound-assignment function should have the form:

```
Department& Department::operator+=(Employee const&);
```

The correct answer is: `Department& operator+=(Employee const& employee);`

Question **41**

Correct

Mark 1.00 out of 1.00

If you want to allow the client to chain calls for a member function just as with built-in types, you would specify **\*this** be returned:

Select one:

- ☐ by value
- ☐ by pointer
- ☒ by reference ✓

Your answer is correct.

Built-in types enable programmers to chain assignments like this:

```
int a = 1, b = 2, c = 3;  
(a = b) = c;
```

The second statement will assign variable **a** the value of variable **c**.

User-defined types can exhibit similar behavior only if they return a reference to **\*this**.

The correct answer is: by reference

Question **42**

Correct

Mark 1.00 out of 1.00

The compiler will issue an error if you overload operator **\*** in your class to perform division.

Select one:

- ☐ True
- ☒ False ✓

Clients have a meaning for each operator from its use on built-in types. The compiler doesn't care if there is no logical mapping from an operator's use for built-in types and an operation on a class type.

However, an overloaded operator has the same precedence and associativity as the corresponding built-in operator.

The correct answer is 'False'.

Question **43**

Correct

Mark 1.00 out of 1.00

To generate temporary objects, the compiler either uses the assignment operator overload function declared in the class definition or it synthesizes an assignment operator overload function.

Select one:

- ☐ True
- ☒ False ✓

Generating temporary objects implies the creation of objects. This occurs when:

1. Variables are defined using an **=**
2. An object is passed as an argument to a parameter of non-reference type
3. Return an object from a function that has a nonreference return type.

All of these cases are examples of copy initialization which uses the copy constructor.

The correct answer is 'False'.

Question **44**  
Correct  
Mark 1.00 out of 1.00

When using a member function to overload a binary operator, the function is implicitly invoked by

Select one or more:

- ☒ the left operand ✓
- ☐ the right operand
- ☐ either the left or the right operand
- ☐ neither operand when a friend function is used to overload the binary operator
- ☐ None of the choices are correct

Your answer is correct.

**When an overloaded operator is a member function, this is bound to the left-hand operand.** Therefore, member operator functions have one less (explicit) parameter than the number of operands.

The correct answer is: the left operand

Question **45**  
Correct  
Mark 4.00 out of 4.00

After reading Section 14.1 of the textbook, match the overloaded operator categories with their expected number of explicit parameters. Note the **this** pointer is not counted as an explicit parameter.

Binary operator overloaded as a non-member function	Two parameters	✓
Unary operator overloaded as a non-member function	One parameter	✓
Binary operator overloaded as member function	One parameter	✓
Unary operator overloaded as a member function	Zero parameters	✓

Your answer is correct.

An overloaded operator function has the same number of parameters as the operator has operands. A unary operator has one parameter; a binary has two. In a binary operator, the left-hand operand is passed to the first parameter and the right-hand operand to the second.

If an operator function is a member function, the first operand is bound to the implicit **this** pointer. Because the first operand is implicitly bound to **this**, a member operator function has one less explicit parameter than the operator has operands.

**If an operator function is a non-member function, it must have at least one user-defined type as operand.** Therefore, a non-member unary operator function must take the user-defined type as a parameter. And, a non-member binary operator function must either take two user-defined types as parameters or one user-defined type as a parameter and a fundamental type as the other parameter.

The correct answer is: Binary operator overloaded as a non-member function → Two parameters, Unary operator overloaded as a non-member function → One parameter, Binary operator overloaded as member function → One parameter, Unary operator overloaded as a member function → Zero parameters

Question **46**

Correct


Mark 1.00 out of 1.00

Write the values printed to standard output stream by the following code fragment.

```
struct Obj {
    Obj operator+(Obj const&) { std::cout << "binary+"; return Obj(); }
    Obj operator+()           { std::cout << "unary+"; return Obj(); }
};

Obj operator*(Obj const& x, Obj const& y) {
    std::cout << "binary-mul";
    return Obj();
}

// in function main ...
Obj o1, o2;
o1+o2;
+o2;
o1*o2;
```

Answer:  

The correct answer is: binary+unary+binary-mul

Question **47**

Correct

Mark 1.00 out of 1.00

Rewrite the indicated statement in the following code fragment so that the overloaded function is called directly. Write your answer without using whitespace and ensure that it is syntactically correct. Otherwise, the grader will mark your answer as incorrect.

```
struct Obj {
    Obj operator+(Obj const&) { std::cout << "binary+"; return Obj(); }
    Obj operator+()           { std::cout << "unary+"; return Obj(); }
};

Obj operator*(Obj const& x, Obj const& y) {
    std::cout << "binary-mul";
    return Obj();
}

// in function main ...
Obj o1, o2;
o1+o2; // rewrite this statement so that overloaded function is called directly
```

Answer:  

Ordinarily, we call an overloaded operator function indirectly by using the operator on arguments of the appropriate type. However, we can also call an overloaded operator function directly in the same way that we call an ordinary function. We name the function and pass an appropriate number of arguments of the appropriate type:

```
o1 + o2;           // normal expression
o1.operator+(o2);  // equivalent function call
```

The correct answer is: o1.operator+(o2);

Question **48**  
Correct  
Mark 1.00 out of 1.00

Rewrite the indicated statement in the following code fragment so that the overloaded function is called directly. Write your answer without using whitespace and ensure that it is syntactically correct. Otherwise, the grader will mark your answer as incorrect.

```
struct Obj {
    Obj operator+(Obj const&) { std::cout << "binary+"; }
    Obj operator+()          { std::cout << "unary+"; }
};

Obj operator*(Obj const& x, Obj const& y) {
    std::cout << "binary-mul";
}

// in function main ...
Obj o2;
+o2; // rewrite this statement so that overloaded function is called directly
```

Answer:  

Ordinarily, we call an overloaded operator function indirectly by using the operator on arguments of the appropriate type. However, we can also call an overloaded operator function directly in the same way that we call an ordinary function. We name the function and pass an appropriate number of arguments of the appropriate type:


```
+o2;           // normal expression
o2.operator+(); // equivalent function call
```

The correct answer is: o2.operator+();

Question **49**  
Correct  
Mark 1.00 out of 1.00

For **inline** functions to be called from any source file, they *must* be defined in a header file.

Select one:

- ☒ True 
- ☐ False

To avoid the substantial overheads of function calls, functions specified as **inline** are usually [usually because the **inline** specification is a request to the compiler] expanded "in line" at each call. In order to expand the code, the compiler will need the definition, not just the declaration, of a function. Therefore, **inline** functions must be defined in header files [since a compiler will have access to only the source file that it is currently compiling].

The correct answer is 'True'.

Question **50**

Correct

Mark 1.00 out of 1.00

A **friend** function must be declared in the class definition. Such a declaration does not make the **friend** function a class member function but it is now given the right to access all **private** and **protected** members of the class. Such a **friend** function of a class is defined outside that class' scope.

To declare a function as a **friend** of a class, precede the function declaration in the class definition with keyword **friend**:

```
// in interface file
class A {
    int n;
public:
    A() : n{340} { std::cout << "A"; }
    ~A()          { std::cout << "B"; }
    friend int N(A const& a);
};

// in implementation file
int N(A const& a) {
    return a.n;
}
```

What is the expected output of the following code fragment?

```
// in function main ...
std::cout << N(A());
```

Answer:



Argument **A()** to function **N** in statement

```
std::cout << N(A());
```

will use **A**'s default constructor to define an unnamed, temporary object of type **A** whose data member **n** is initialized to **340**. Function call **N(A())** is evaluated with this unnamed, temporary object as its argument. Function **N** will return the value **340** that is written to the standard output stream. After function **N(A())** returns value **340**, the unnamed, temporary object referenced by parameter **a** goes out of scope. This causes destructor **A::~~A()**; is executed causing **"B"** to be written to the standard output stream.

The correct answer is: A340B

Question **51**

Correct

Mark 1.00 out of 1.00

To stream an object binary operators `<<` or `>>` must be overloaded for the stream object as operators' left-hand argument and the object as the right-hand argument. The canonical way to overload `<<` for some class **A** is:

```
std::ostream& operator<<(std::ostream& os, A const& obj) {  
    // insert obj's data into os  
    return os;  
}
```

If an object has some private members that must be written to the stream object, the operator overloading must be declared as a **friend** inside the object's class definition.

```
class A {  
    int n; // private  
public:  
    A() : n{760} { }  
    friend std::ostream& operator<<(std::ostream&, A const&);  
};
```

The overloaded **friend** function is defined in the implementation file:

```
std::ostream& operator<<(std::ostream& os, A const& a) {  
    os << a.n; // stream private data member n into os  
    return os;  
}
```

What is the expected output of the following code fragment?

```
// in function main ...  
std::cout << A();
```

Answer:

✓

The expression **A()** in the statement

```
std::cout << A();
```

will use **A**'s default constructor to define an unnamed, temporary object of type **A**.

The correct answer is: 760