| | |
|---|---|
| **Started on** | Monday, 12 October 2020, 2:55 PM |
| **State** | Finished |
| **Completed on** | Monday, 12 October 2020, 2:59 PM |
| **Time taken** | 4 mins 5 secs |
| **Marks** | 10.00/10.00 |
| **Grade** | **100.00** out of 100.00 |

Question **1**
Correct
Mark 1.00 out of 1.00

What is the printout of the program shown below that uses overloaded functions for l-value references and r-value references?

```
#include <iostream>

void foo(double, int) { std::cout << "A"; }
void foo(int, double) { std::cout << "B"; }
void foo(float&, short) { std::cout << "C"; }
void foo(const int&, const long&) { std::cout << "D"; }
void foo(int&&, float&&) { std::cout << "E"; }

int main()
{
    float f = 0.0f;
    int i = 0;

    foo(1.0f, static_cast<short>(3));
    foo(i, 1L);
    foo(std::move(i), f);

}
```

Select one:

- a. NC; this program does not compile but it does not contain ambiguous calls.
- b. AEB
- c. CDD
- d. ADB ✔ Correct!
- e. NC; this program does not compile due to at least 1 ambiguous call.

The output is "ADB".

The correct answer is: ADB

Question **2**
Correct
Mark 1.00 out of 1.00

Which of the following examples is **not an *l-value* object**?

Select one:

- a. The *this* pointer in member functions ✔ Correct; *this* is a pr-value.
- b. A name of a local variable
- c. A name of an r-value reference after binding
- d. A call to a function that returns *int&*
- e. A dereferenced pointer (**ptr*)

The correct answer is: The *this* pointer in member functions

Question **3**

Correct

Mark 1.00 out of 1.00

Suppose that you are writing a class *MyVector* (similar to std::vector) that supports both the copy and the move semantics. Which of the following member functions is declared **incorrectly**?

Select one:

- a. MyVector(const MyVector& rhs);
- b. MyVector& operator=(const MyVector& rhs);
- c. MyVector& operator=(MyVector&&);
- d. MyVector& operator=(const MyVector&& rhs); ✔ This declaration is incorrect.
- e. ~MyVector();

For the move semantics, you have to be able to swap data member values from the argument. Therefore, it must not be *const*.

*MyVector(**const** MyVector**&** rhs);* // copy c-tor
*MyVector(MyVector**&&** rhs);* // move c-tor
*MyVector& operator=(**const** MyVector**&** rhs);* // copy assignment
*MyVector& operator=(MyVector**&&** rhs);* // move assignment
*~MyVector();* // d-tor

The correct answer is: MyVector& operator=(const MyVector&& rhs);

Question **4**

Correct

Mark 1.00 out of 1.00

Which of the following reference collapsing rules is **not valid**?

Select one:

- a. A& & → A&
- b. A&& & → A&& ✔ Yes, this rule is incorrect!
- c. A&& & → A&
- d. A&& && → A&&
- e. A& && → A&

The rules are as shown below:

```
T Ref → T&&
```

```
A& & → A& A&& & → A& A& && → A& A&& && → A&&
```

The correct answer is: A&& & → A&&

Question **5**

Correct

Mark 1.00 out of 1.00

Which of the following statements about references is **true**?

Select one:

- a. You can deduce the size of a reference object by calling a sizeof operator on it.
- b. You can declare an array of references.
- c. You can declare a pointer to a reference.
- d. You can legally convert l-value reference to r-value reference. ✔ Correct!
- e. You can declare a reference to a reference.

You can convert l-value references to r-value references; this is what *std::move* and *std::forward* can do.

The correct answer is: You can legally convert l-value reference to r-value reference.

Question **6**

Correct

Mark 1.00 out of 1.00

On 16th May 2018, Robert C. "Uncle Bob" Martin (*@unclebobmartin*) tweeted that this SOLID principle generalises into:

*"Don't depend on more than you need."*

Which of the principles was he writing about?

Select one:

- a. OCP
- b. ISP ✔ Correct!
- c. DIP
- d. SRP
- e. LSP

Robert C. Martin wrote:

*ISP can be seen as similar to SRP for interfaces; but it is more than that. ISP generalizes into: "Don't depend on more than you need." SRP generalizes to "Gather together things that change for the same reasons and at the same times."*

[Source]

The correct answer is: ISP

Question **7**

Correct

Mark 1.00 out of 1.00

What is the name of an STL template that provides functionality corresponding to the following code:

```
template<typename T>
T&& function_name(remove_reference_t<T>& t)
{
    return static_cast<T&&>(t);

}
```

Select one:

- a. std::decltype
- b. std::swap
- c. std::copy
- d. std::move
- e. std::forward ✔ Correct!

This function template resembles *std::forward<T>(t)*. It takes in an object by l-value reference, and returns an l-value or r-value reference to this object, depending on the template argument *T*.

The correct answer is: std::forward

Question **8**

Correct

Mark 1.00 out of 1.00

Which of the statements about function templates is **not true**?

Select one:

- a. A compiler must select a base template before it investigates template specializations.
- b. When template parameters are not explicitly indicated, a compiler searches for a non-template function with a perfect parameter match before considering function templates.
- c. A compiler is able to deduce types of some or all template function arguments based on the actual parameters used in a call.
- d. The return type of a function template indicated as T&& is an r-value reference to an object type T, assuming T is a template parameter. ✔ Correct; this statement is not true.
- e. You should avoid function template overloading and specialization at the same time due to potential conflicts with order of specialization.

The return type of a function template indicated as T&& is a universal reference if T is a template parameter; it could be either l-value or r-value reference, depending on the actual template parameters used in instantiation.

The correct answer is: The return type of a function template indicated as T&& is an r-value reference to an object type T, assuming T is a template parameter.

Question **9**

Correct

Mark 1.00 out of 1.00

Which of the following definitions **cannot** be a template in C++17?

Select one:

- a. Destructors ✔ Correct!
- b. Dependent classes
- c. Type aliases ("using")
- d. Namespace variables
- e. Global functions

"A template is a C++ entity that defines one of the following:

- a family of classes (class template), which may be nested classes,
- a family of functions (function template), which may be member functions,
- an alias to a family of types (alias template) with the using keyword,
- a family of variables (variable template)."

Source: https://en.cppreference.com/w/cpp/language/templates

The correct answer is: Destructors

Question **10**

Correct

Mark 1.00 out of 1.00

What is the behaviour of the code shown below?

```
#include <utility> // std::move

template <typename T>
void foo(T&& t)
{
    T bar{std::move(t)};
    (void)bar;

}

int main()
{
    int x = 5;
    int&& rrx = std::move(x);
    foo(rrx);

}
```

Select one:

- a. This code does not compile, even though a call to foo() passed an l-value object to a universal reference that collapses into an l-value reference. ✔ Correct!
- b. This code does not compile due to a call to foo() with an l-value object passed to an r-value reference.
- c. This code does compile, but the program may crash.
- d. This code compiles and executes correctly.
- e. This code does not compile due to a call to foo() with an r-value object passed to a universal reference that collapses into an l-value reference.

This code does not compile. foo() is invoked with an l-value object (r-value reference with a name), causing a universal reference *T&&* to resolve to *int&*. Therefore, *T* must be *int&*.

It is not possible to create an l-value reference *int& bar* and bind it to an r-value reference produced by *std::move(t)*.

The correct answer is: This code does not compile, even though a call to foo() passed an l-value object to a universal reference that collapses into an l-value reference.

◄ Quiz 2 Jump to... Assignment 3 specification ►