Started on	Friday, 3 March 2023, 12:21 AM
State	Finished
Completed on	Friday, 3 March 2023, 1:55 AM
Time taken	1 hour 34 mins
Grade	**57.00** out of 58.00 (**98**%)

Information

Templates aren't compiled into single entities that can handle any type. Instead, the compiler will generate different functions or classes for every type for which the template is used. Let's look at what happens when function template `compare` [defined at bottom of page 652 in Section 6.1.1 of textbook]:

```
template <typename T>
int compare(T const& lhs, T const& rhs) {
  if (lhs < rhs) return -1;
  if (rhs < lhs) return 1;
  return 0;
}
```

is called:

```
std::cout << compare(2.2, 1.1) << '\n';
```

When a function template is called, the compiler uses function arguments [`2.2` and `1.1` in our example] to implicitly deduce the ***template argument***. Since function arguments `2.2` and `1.1` evaluate to type `double`, the template argument is deduced to be concrete type `double`. The compiler generates a specific instance of function template `compare` in which template parameter `T` is replaced by the corresponding template argument `double`, as in:

```
int compare(const double &v1, const double &v2) {
  if (v1 < v2) return -1;
  if (v2 < v1) return 1;
  return 0;
}
```

After generating this specific instance, the compiler will check its code to ensure that all calls within the newly generated function are valid. Invalid calls such as unsupported function calls are flagged as compile-time errors. That is, an attempt to instantiate a template for a type that doesn't support all the operations used within it will result in a compile-time error. In fact, templates are compiled twice:

1. Without instantiation, the code associated with definition of function template `compare` is checked for correct syntax. The compiler will flag an error when it discovers a syntax error such as missing semicolon.
2. At the time of instantiation, the code associated with the newly generated instance of `compare` is checked to ensure all calls in the function are valid. If invalid calls are discovered, they're flagged as compile-time errors.

The process of replacing template parameters by concrete types is called ***instantiation***. It results in an ***instance*** of a template. After instantiation, the behavior of the specific instance is similar to ordinary functions: at run-time, function ***arguments*** are evaluated and the resulting values are used to initialize function ***parameters***.

Similarly, when the function template `compare` is called as follows:

```
std::cout << compare(200, 100) << '\n';
```

the compiler uses function arguments [`200` and `100` in our example] to implicitly conclude that the template argument must be type `int` and that during instantiation, the template parameter `T` must be `int`.

Note that during instantiation, no automatic type conversion is allowed. Since the definition of function template `compare` declares a single template parameter for both function parameters, each of the function arguments in the call to `compare` must match exactly. For example, the call

```
compare(10, 20.22);
```

will result in a compile-time error because the first `T` is `int` and the second `T` is `double`.

Question **1**

Correct

Mark 5.00 out of 5.00

Identify C++ keywords in the following function template definition:

```
template <typename T>
T cube(T value) {
  return value*value*value;
}
```

`return`	Yes - this is a keyword	✓
`T`	No - this is not a keyword	✓
`typename`	Yes - this is a keyword	✓
`cube`	No - this is not a keyword	✓
`value`	No - this is not a keyword	✓
`template`	Yes - this is a keyword	✓

Your answer is correct.

The correct answer is: `return` → Yes - this is a keyword, `T` → No - this is not a keyword, `typename` → Yes - this is a keyword, `cube` → No - this is not a keyword, `value` → No - this is not a keyword, `template` → Yes - this is a keyword

Question **2**

Correct

Mark 12.00 out of 12.00

Use the following function template declaration and variable definitions to deduce the template type parameter `T` for each of the subsequent statements involving calls to function `foo`. If template type deduction fails, choose NC as your answer.

```
template<typename T>
void foo(T);

int x{100};
const int cx{x};
int &rx{x};
int const &crx{x};
int *pi{&x};
int a[10]{10};
int const ca[10]{10,20};
int b[2][10]{};
```

`foo(a);`	int*	✓
`foo(5);`	int	✓
`foo(cx);`	int	✓
`foo(pi);`	int*	✓
`foo(&cx);`	int const*	✓
`foo(&x);`	int*	✓
`foo(rx);`	int	✓
`foo(crx);`	int	✓
`foo(x);`	int	✓
`foo(b);`	int (*)[10]	✓
`foo(ca);`	int const*	✓
`foo(&pi);`	int**	✓

Your answer is correct.

The correct answer is: `foo(a);` → int*, `foo(5);` → int, `foo(cx);` → int, `foo(pi);` → int*, `foo(&cx);` → int const*, `foo(&x);` → int*, `foo(rx);` → int, `foo(crx);` → int, `foo(x);` → int, `foo(b);` → int (*)[10], `foo(ca);` → int const*, `foo(&pi);` → int**

Question **3**

Correct

Mark 12.00 out of 12.00

Use the following function template declaration and variable definitions to deduce the template type parameter `T` for each of the subsequent statements involving calls to function `foo`. If template type deduction fails, choose NC as your answer.

```
template<typename T>
void foo(T*);

int x{100};
const int cx{x};
int &rx{x};
int const &crx{x};
int *pi{&x};
int a[10]{10};
int const ca[10]{10,20};
int b[2][10]{};
```

`foo(&x);`	int	✓
`foo(rx);`	NC	✓
`foo(a);`	int	✓
`foo(crx);`	NC	✓
`foo(b);`	int [10]	✓
`foo(&cx);`	int const	✓
`foo(pi);`	int	✓
`foo(cx);`	NC	✓
`foo(5);`	NC	✓
`foo(ca);`	int const	✓
`foo(&pi);`	int*	✓
`foo(x);`	NC	✓

Your answer is correct.

The correct answer is: `foo(&x);` → int, `foo(rx);` → NC, `foo(a);` → int, `foo(crx);` → NC, `foo(b);` → int [10], `foo(&cx);` → int const, `foo(pi);` → int, `foo(cx);` → NC, `foo(5);` → NC, `foo(ca);` → int const, `foo(&pi);` → int*, `foo(x);` → NC

Question **4**

Correct

Mark 12.00 out of 12.00

Use the following function template declaration and variable definitions to deduce the template type parameter `T` for each of the subsequent statements involving calls to function `foo`. If template type deduction fails, choose NC as your answer.

```
template<typename T>
void foo(T&);

int x{100};
const int cx{x};
int &rx{x};
int const &crx{x};
int *pi{&x};
int a[10]{10};
int const ca[10]{10,20};
int b[2][10]{};
```

`foo(&x);`	NC	✓
`foo(5);`	NC	✓
`foo(&pi);`	NC	✓
`foo(pi);`	int*	✓
`foo(ca);`	int const [10]	✓
`foo(rx);`	int	✓
`foo(x);`	int	✓
`foo(b);`	int [2][10]	✓
`foo(crx);`	int const	✓
`foo(a);`	int [10]	✓
`foo(&cx);`	NC	✓
`foo(cx);`	int const	✓

Your answer is correct.

The correct answer is: `foo(&x);` → NC, `foo(5);` → NC, `foo(&pi);` → NC, `foo(pi);` → int*, `foo(ca);` → int const [10], `foo(rx);` → int, `foo(x);` → int, `foo(b);` → int [2][10], `foo(crx);` → int const, `foo(a);` → int [10], `foo(&cx);` → NC, `foo(cx);` → int const

Question **5**

Partially correct

Mark 11.00 out of 12.00

Use the following function template declaration and variable definitions to deduce the template type parameter `T` for each of the subsequent statements involving calls to function `foo`. If template type deduction fails, choose NC as your answer.

```
template<typename T>
void foo(T const&);

int x{100};
const int cx{x};
int &rx{x};
int const &crx{x};
int *pi{&x};
int a[10]{10};
int const ca[10]{10,20};
int b[2][10]{};
```

`foo(crx);`	int	✓
`foo(&x);`	int const*	✗
`foo(ca);`	int [10]	✓
`foo(&cx);`	int const*	✓
`foo(cx);`	int	✓
`foo(&pi);`	int**	✓
`foo(5);`	int	✓
`foo(rx);`	int	✓
`foo(a);`	int [10]	✓
`foo(x);`	int	✓
`foo(b);`	int [2][10]	✓
`foo(pi);`	int*	✓

Your answer is partially correct.

You have correctly selected 11.

The correct answer is: `foo(crx);` → int, `foo(&x);` → int*, `foo(ca);` → int [10], `foo(&cx);` → int const*, `foo(cx);` → int, `foo(&pi);` → int**, `foo(5);` → int, `foo(rx);` → int, `foo(a);` → int [10], `foo(x);` → int, `foo(b);` → int [2][10], `foo(pi);` → int*

Question **6**

Correct

Mark 1.00 out of 1.00

Determine the number of function parameters specified in the definition of function template `compare`.

Select one:

- 1
- The number of function parameters cannot be determined
- 0
- [✓ 2] ✓

Your answer is correct.

The correct answer is: 2

Question **7**

Correct

Mark 1.00 out of 1.00

Provide the number of template parameters specified by the template parameter list in the definition of function template `compare`.

Select one:

- ○ 2
- ○ The number of template parameters cannot be determined
- ● 1 ✓
- ○ 0

Your answer is correct.

The correct answer is: 1

Question **8**

Correct

Mark 1.00 out of 1.00

How many times is template code compiled?

Select one or more:

- ☑ Twice if the function template is called: once to check for syntax errors in the definition of the function template itself and the second time to check for valid function calls in the instantiated function ✓
- ☐ Generic programming uses run-time polymorphism and therefore the code doesn't have to be compiled
- ☑ Only once if the function template is never called: to check for syntax errors in the definition of the function template ✓

Your answer is correct.

The correct answers are: Only once if the function template is never called: to check for syntax errors in the definition of the function template, Twice if the function template is called: once to check for syntax errors in the definition of the function template itself and the second time to check for valid function calls in the instantiated function

Question **9**

Correct

Mark 1.00 out of 1.00

Which of the following are motivations for C++ to introduce function templates?

Select one or more:

- ☑ Because we want to define a single function that can reuse the same algorithm for different types ✓
- ☑ Because it is inconvenient for programmer to maintain many similar "versions" of a functions ✓
- ☐ Because function overloading cannot figure out which function to call
- ☐ Because function templates make code look smarter

Your answer is correct.

The correct answers are: Because we want to define a single function that can reuse the same algorithm for different types, Because it is inconvenient for programmer to maintain many similar "versions" of a functions

Question **10**

Correct

Mark 1.00 out of 1.00

Write the template parameter deduced and the value printed to standard output by the following code fragment:

```
std::cout << compare(11.1f, 12.2f);
```

If the code fragment doesn't compile, write NC. Otherwise, write the template argument type deduced by the compiler followed by a comma followed by the value printed to standard output. Don't add any extraneous whitespace characters.

Answer: float,-1 ✓

The correct answer is: float,-1

◄ Midterm Test for Spring 2023

Jump to...

Lab: Doubly Linked List - Function Templates ►