Started on	Tuesday, 7 February 2023, 10:42 PM
State	Finished
Completed on	Tuesday, 7 February 2023, 10:50 PM
Time taken	8 mins 46 secs
Grade	**67.00** out of 68.00 (**99**%)

Information

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.

The C++ mechanism that provides notational convenience of using the same name for operations on different types is called ***overloading***. The technique is already used for basic operations in C++. That is, there is only one name for addition `+`, yet it can be used to add values of integer and floating-point types and combinations of such types. This idea is easily extended to functions defined by the programmer and is called ***function overloading***. In function overloading, functions can share the same name as long as their parameter declarations are sufficiently different.

Since function overloading eliminates the need to invent – and remember – names that exist only to help the programmer and compiler figure out which function to call, we can define a function `cube` for each specific type:

```
int cube(int n)       { return n * n * n; }
float cube(float n)   { return n * n * n; }
double cube(double n) { return n * n * n; }
long cube(long n)     { return n * n * n; }
```

and call these `cube` functions without needing to choose the right function:

```
int    i {8};
long   l {50L};
float  f {2.5F};
double d {3.14};

std::cout << cube(i) << "\n"; // ok - prints: 512
std::cout << cube(l) << "\n"; // ok - prints: 125000
std::cout << cube(f) << "\n"; // ok - prints: 15.625
std::cout << cube(d) << "\n"; // ok - prints: 30.9591
```

Now, if we decide we need to handle another data type, we simply *overload* the `cube` function to handle the new type. Clients using the `cube` library have no idea that we implement `cube` as separate functions. Each call to function `cube` has an argument type that exactly matches the parameter type of a particular definition of function `cube`. Therefore, it is straightforward for the compiler to match a function call to a specific function definition. However, the situation gets complicated when we make the following call:

```
char ch = 'a';
cube(ch);
```

Since a `cube(char)` function was not defined, the compiler will have to choose one of the previously defined `cube` functions or flag the call `cube` as an error.

Now, let's review the basics of what is deemed a function overload and what is not a function overload.

Question **1**

Correct

Mark 1.00 out of 1.00

In C++, functions declared in the same scope can share the same name as long as their ***signatures*** are different. The signature of a function consists of:

Select one:

- The function name
- The return type
- The function name, number of arguments, types of the arguments (in their respective order) and the return type
- The function name, number of arguments and the types of the arguments (in their respective order) ✔
- The types of the arguments (in their respective order)
- The number of arguments

The correct answer is: The function name, number of arguments and the types of the arguments (in their respective order)

Question **2**

Correct

Mark 1.00 out of 1.00

Do the following declarations overload function `foo`?

```
int foo(int);
void foo(int);
```

Select one:

- True
- False ✔

The correct answer is 'False'.

Question **3**

Correct

Mark 1.00 out of 1.00

Do the following declarations overload function `foo`?

```
void foo(int x, double y);
void foo(int, double);
```

Select one:

- True
- False ✔

The correct answer is 'False'.

Question **4**

Correct

Mark 1.00 out of 1.00

Do the following declarations overload function `foo`?

```
void foo();
void foo(int x, double y);
void foo(double x, int y);
void foo(int x, double y, char c);
```

Select one:

- True ✔
- False

The correct answer is 'True'.

Question **5**

Correct

Mark 1.00 out of 1.00

Do the following declarations overload function `foo`?

```
void foo(int);
void foo(int const);
```

Select one:

- True
- False ✔

The correct answer is 'False'.

Question **6**

Correct

Mark 1.00 out of 1.00

Do the following declarations overload function `foo`?

```
using INT = int;

void foo(int);
void foo(INT);
```

Select one:

- True
- False ✔

The correct answer is 'False'.

Question **7**

Correct

Mark 1.00 out of 1.00

Do the following declarations overload function `foo`?

```
typedef int INT;

void foo(int);
void foo(INT);
```

Select one:

- True
- False ✔

The correct answer is 'False'.

Question **8**

Correct

Mark 1.00 out of 1.00

Do the following declarations overload function `foo`? **Hint:** Recall the discussion in lectures [and also available in Section 2.4.3 of the text] about the meanings and functionalities of ***top-level*** `const` and ***low-level*** `const` pointers.

```
void foo(int*);
void foo(int const*);
```

Select one:

- True ✔
- False

The correct answer is 'True'.

Question **9**

Correct

Mark 1.00 out of 1.00

Do the following declarations overload function `foo`? **Hint:** Recall the discussion in lectures [and also available in Section 2.4.3 of the text] about the meanings and functionalities of ***top-level*** `const` and ***low-level*** `const` pointers.

```
void foo(int*);
void foo(int* const);
```

Select one:

- True
- False ✔

The correct answer is 'False'.

Information

Functions declared in the same scope can share the same name as long as their ***signatures*** are different. When a function `foo` is called, the C++ compiler must determine which of the functions named `foo` to invoke - that is, the compiler performs an ***Overload Resolution***. This is done by comparing the types of the actual arguments with the types of the parameters of ***all functions in scope*** called `foo`. In short, overload resolution proceeds as follows:

1. Is there an overload in scope that matches the argument type exactly: that is, the argument type matches using no or only trivial conversions [for example, array name to pointer, function name to pointer to function)? Take it; otherwise:
2. Are there overloads that match after conversion? How many?
   - 0: Error: No matching function found.
   - 1: Take it.
   - > 1: Error: ambiguous call.

As described in Section 6.6 of the text, the idea behind Overload Resolution is to invoke the best match to the argument and give a compile-time error if no function is the best match. To approximate our notion of what is reasonable, a series of criteria are tried in order by the compiler. Unfortunately, in order to cope with inherent complexities of built-in arithmetic types, the language rules are quite complicated. The simplified version is presented here:

1. Exact match; that is, match using no or only trivial conversions [for example, array name to pointer, function name to pointer to function]
2. Match using promotions; that is,
   - integral promotions [`bool` to `int`, `char` to `int`, `short` to `int`, `unsigned char` to `int`, `unsigned short` to `int`]; and
   - `float` to `double`
   - All promotions are treated as equivalent
3. Match using standard conversions; that is,
   - conversions between integral types [`bool`, `char`, `short`, `int`, `long int`, `long long int`], apart from the ones counted as promotions;
   - conversions between floating-point types [`double`, `long double`] except for `float` to `double` which is a promotion;
   - conversions between floating and integral types such as `int` to `double`, `double` to `int`;
   - conversions of integral, floating and pointer types to `bool`; and
   - conversion of integer `0` to `nullptr`
   - All standard conversions are treated as equivalent

There are other criteria but we don't need to be concerned with those.

If two matches are found at the highest level [exact match, match using promotions, or match using standard conversion] where a match is found, the call is rejected as ***ambiguous***. The resolution rules are this elaborate primarily to take into account the elaborate C and C++ rules for built-in numeric types.

Question **10**

Correct

Mark 3.00 out of 3.00

Use the following declarations and definitions to perform overload resolutions on the various calls to function `print`:

```
void print(double); // function labeled F0
void print(long);   // function labeled F1
```

`print(123.89);`	Function labeled F0 is called ✔
`print(1L);`	Function labeled F1 is called ✔
`print(123);`	Compiler error because call of overloaded function is ambiguous ✔

Your answer is correct.

The correct answer is: `print(123.89);` → Function labeled F0 is called, `print(1L);` → Function labeled F1 is called, `print(123);` → Compiler error because call of overloaded function is ambiguous

Question **11**

Partially correct

Mark 15.00 out of 16.00

Use the following declarations and definitions to perform overload resolutions on the various calls to function `print`:

```
void print(int);          // function labeled F0
void print(char const*); // function labeled F1
void print(double);       // function labeled F2
void print(long);         // function labeled F3
void print(char);         // function labeled F4

char c{'a'};
int i{1};
short s = 2;
float f{12.3F};
```

`print(static_cast<unsigned short>(s));`	Function labeled F0 is called ✔
`print(static_cast<unsigned long>(s));`	Compiler error because call of overloaded function is ambiguous ✔
`print(0);`	Function labeled F0 is called ✔
`print(&s);`	Compiler error because call of overloaded function has no match ✔
`print(c);`	Function labeled F4 is called ✔
`print('Z');`	Function labeled F4 is called ✔
`print(&c);`	Function labeled F1 is called ✔
`print(true);`	Function labeled F0 is called ✔
`print(123UL);`	Compiler error because call of overloaded function is ambiguous ✔
`print(12.1F);`	Function labeled F2 is called ✔
`print(f);`	Function labeled F2 is called ✔
`print(94);`	Run-time error ✖
`print(s);`	Function labeled F0 is called ✔
`print(i);`	Function labeled F0 is called ✔
`print("Z");`	Function labeled F1 is called ✔
`print(nullptr);`	Function labeled F1 is called ✔

The correct answer is: `print(static_cast<unsigned short>(s));` → Function labeled F0 is called, `print(static_cast<unsigned long>(s));` → Compiler error because call of overloaded function is ambiguous, `print(0);` → Function labeled F0 is called, `print(&s);` → Compiler error because call of overloaded function has no match, `print(c);` → Function labeled F4 is called, `print('Z');` → Function labeled F4 is called, `print(&c);` → Function labeled F1 is called, `print(true);` → Function labeled F0 is called, `print(123UL);` → Compiler error because call of overloaded function is ambiguous, `print(12.1F);` → Function labeled F2 is called, `print(f);` → Function labeled F2 is called, `print(94);` → Function labeled F0 is called, `print(s);` → Function labeled F0 is called, `print(i);` → Function labeled F0 is called, `print("Z");` → Function labeled F1 is called, `print(nullptr);` → Function labeled F1 is called

Information

Recall from lectures and handout that mixing overloads of reference and value parameters almost always fails. One rule to follow is that when one overload has a reference-qualified parameter, then the corresponding parameter of the other overloads should be reference-qualified as well.

Question **12**

Correct

Mark 6.00 out of 6.00

Use the following declarations and definitions to perform overload resolutions on the various calls to function `foo`:

```
void foo(int&); // function labeled F0
void foo(int);  // function labeled F1

int x{10};
double d = x;
int &rx{x};
int const cx{10};
```

`foo(rx);`	Compiler error because call of overloaded function is ambiguous ✔
`foo(d);`	Function labeled F1 is called ✔
`foo(cx);`	Function labeled F1 is called ✔
`foo(x);`	Compiler error because call of overloaded function is ambiguous ✔
`foo(10);`	Function labeled F1 is called ✔
`foo(12.34);`	Function labeled F1 is called ✔

Your answer is correct.

The correct answer is: `foo(rx);` → Compiler error because call of overloaded function is ambiguous, `foo(d);` → Function labeled F1 is called, `foo(cx);` → Function labeled F1 is called, `foo(x);` → Compiler error because call of overloaded function is ambiguous, `foo(10);` → Function labeled F1 is called, `foo(12.34);` → Function labeled F1 is called

Question **13**

Correct

Mark 7.00 out of 7.00

Use the following declarations and definitions to perform overload resolutions on the various calls to function `foo`:

```
void foo(int const&); // function labeled F0
void foo(int);        // function labeled F1

int x{10};
int const cx{11};
int const& rcx{x};
int const& crcx{cx};
double d = rcx;
```

`foo(10);`	Compiler error because call of overloaded function is ambiguous ✔
`foo(&rcx);`	Compiler error because call of overloaded function has no match ✔
`foo(x);`	Compiler error because call of overloaded function is ambiguous ✔
`foo(d);`	Compiler error because call of overloaded function is ambiguous ✔
`foo(cx);`	Compiler error because call of overloaded function is ambiguous ✔
`foo(crcx);`	Compiler error because call of overloaded function is ambiguous ✔
`foo(rcx);`	Compiler error because call of overloaded function is ambiguous ✔

Your answer is correct.

The correct answer is: `foo(10);` → Compiler error because call of overloaded function is ambiguous, `foo(&rcx);` → Compiler error because call of overloaded function has no match, `foo(x);` → Compiler error because call of overloaded function is ambiguous, `foo(d);` → Compiler error because call of overloaded function is ambiguous, `foo(cx);` → Compiler error because call of overloaded function is ambiguous, `foo(crcx);` → Compiler error because call of overloaded function is ambiguous, `foo(rcx);` → Compiler error because call of overloaded function is ambiguous

Question **14**

Correct

Mark 8.00 out of 8.00

Use the following declarations and definitions to perform overload resolutions on the various calls to function `foo`:

```
void foo(int&);       // function labeled F0
void foo(int const&); // function labeled F1

int x{11};
int &rx{x};
int const& rcx{rx};
int const cx{10};
int *px{&x};
int const *pcx{&x};
```

`foo(x);`	Function labeled F0 is called ✔
`foo(rcx);`	Function labeled F1 is called ✔
`foo(*px);`	Function labeled F0 is called ✔
`foo(*pcx);`	Function labeled F1 is called ✔
`foo(rx);`	Function labeled F0 is called ✔
`foo(cx);`	Function labeled F1 is called ✔
`foo(10);`	Function labeled F1 is called ✔
`foo(px);`	Compiler error because call to overloaded function has no match ✔

Your answer is correct.

The correct answer is: `foo(x);` → Function labeled F0 is called, `foo(rcx);` → Function labeled F1 is called, `foo(*px);` → Function labeled F0 is called, `foo(*pcx);` → Function labeled F1 is called, `foo(rx);` → Function labeled F0 is called, `foo(cx);` → Function labeled F1 is called, `foo(10);` → Function labeled F1 is called, `foo(px);` → Compiler error because call to overloaded function has no match

Question **15**

Correct

Mark 3.00 out of 3.00

Use the following declarations and definitions to perform overload resolutions on the various calls to function `foo`:

```
void foo(int&);       // function labeled F0
void foo(int const&); // function labeled F1
void foo(int);        // function labeled F2

int x{10};
int const cx{10};
int const& rcx{cx};
int const * const cpcx{&rcx};
```

`foo(x);`	Compiler error because call of overloaded function is ambiguous ✔
`foo(10);`	Compiler error because call of overloaded function is ambiguous ✔
`foo(*cpcx);`	Compiler error because call of overloaded function is ambiguous ✔

Your answer is correct.

The correct answer is: `foo(x);` → Compiler error because call of overloaded function is ambiguous, `foo(10);` → Compiler error because call of overloaded function is ambiguous, `foo(*cpcx);` → Compiler error because call of overloaded function is ambiguous

Information

We can use the overload resolution rules to select the most appropriate function when the efficiency or precision of computations differs significantly among types. In the process of choosing among overloaded functions with two or more arguments, a best match is found for each argument using the rules described earlier in this quiz [see Section 6.6 of the text].

Now, here's the difficult part: A function that is the best match for one argument and a better or equal match for all other arguments is called. If no such function exists, the call is rejected as ambiguous.

Question **16**

Correct

Mark 3.00 out of 3.00

Use the following declarations to perform overload resolution on the subsequent call to function `foo`:

```
void foo(int, double, int=2); // function labeled F0
void foo(int, double);        // function labeled F1

foo(123, 123.45); // which foo is called?
```

Select one:

- Compiler error because call of overloaded function has no match
- Run-time error
- Function labeled `F0` is called
- Function labeled `F1` is called
- Compiler error because call of overloaded function call is ambiguous ✔

The correct answer is: Compiler error because call of overloaded function call is ambiguous

Question **17**

Correct

Mark 3.00 out of 3.00

Use the following declarations and definitions to perform overload resolutions on the various calls to function `foo`:

```
void foo(int, double, int=2); // function labeled F0
void foo(int, int);           // function labeled F1
```

`foo(123, 456);`	Function labeled F1 is called ✔
`foo(123, 123.45);`	Function labeled F0 is called ✔
`foo(123, 456, 123.45);`	Function labeled F0 is called ✔

Your answer is correct.

The correct answer is: `foo(123, 456);` → Function labeled F1 is called, `foo(123, 123.45);` → Function labeled F0 is called, `foo(123, 456, 123.45);` → Function labeled F0 is called

Question **18**

Correct

Mark 5.00 out of 5.00

Use the following declarations and definitions to perform overload resolutions on the various calls to function `divide`:

```
int divide(int, int);          // function labeled F0
float divide(float, float);    // function labeled F1
double divide(double, double); // function labeled F2

short sh = 50;
int i1{50}, i2{22};
float f{22.2F};
double d1{22.2}, d2{33.3};
```

`divide(i1, i2);`	Function labeled F0 is called ✔
`divide(i1, f);`	Compiler error because call of overloaded function is ambiguous ✔
`divide(sh, d1);`	Compiler error because call of overloaded function is ambiguous ✔
`divide(d1, d2);`	Function labeled F2 is called ✔
`divide(f, d1);`	Compiler error because call of overloaded function is ambiguous ✔

Your answer is correct.

The correct answer is: `divide(i1, i2);` → Function labeled F0 is called, `divide(i1, f);` → Compiler error because call of overloaded function is ambiguous, `divide(sh, d1);` → Compiler error because call of overloaded function is ambiguous, `divide(d1, d2);` → Function labeled F2 is called, `divide(f, d1);` → Compiler error because call of overloaded function is ambiguous

Question **19**

Correct

Mark 5.00 out of 5.00

Use the following declarations and definitions to perform overload resolutions on the various calls to function `foo`:

```
void foo(unsigned short const, short const, unsigned int&); // function labeled F0
void foo(int&, unsigned int, short);                        // function labeled F1
void foo(float const, unsigned short, unsigned long&);      // function labeled F2
void foo(unsigned long, unsigned char, int const);          // function labeled F3
void foo(int const, long, unsigned const char);             // function labeled F4

long const a{};
int const& b{int()};
unsigned short const& c{0};

foo(a, b, c);
```

Select one:

- Function labeled `F2` is called
- Run-time error
- Compiler error because call of overloaded function has no match
- Function labeled `F0` is called
- Compiler error because call of overloaded function is ambiguous
- Function labeled `F3` is called ✔
- Function labeled `F4` is called
- Function labeled `F1` is called

The correct answer is: Function labeled `F3` is called

Information

As you've seen in this review, the details of Overload Resolution algorithm are quite complicated. Therefore, it is important that you don't write programs whose correctness depends on the exact details of resolving overloaded functions.

So why bother with function overloading? Consider the alternative to overloading. Often, we need similar algorithms and operations performed on objects of several types. Without overloading, we must define several functions with different names:

```
void print_int(int);
void print_char(char);
void print_string(char const*);
```

Compared to overloaded `print()`:

```
void print(int);
void print(char);
void print(char const*);
```

we've to remember several names for the same operation and remember to use those names correctly. This is tedious, defeats attempts to do generic programming [using templates], and generally encourages the programmer to focus on relatively low-level typing related issues. Because there is no overloading, all standard conversions apply to function arguments. It can also lead to errors:

```
int i;
char c;
char const *p;
double d;

print_int(i);    // ok
print_char(c);   // ok
print_string(p); // ok
print_int(c);    // problem: char argument promoted to int
print_char(i);   // problem: int argument truncated to char
print_string(i); // error
print_int(d);    // problem: double argument truncated to int
```

In the previous example, only one of the four calls with doubtful semantics is caught by the compiler. In particular, two calls rely on error-prone narrowing of information [from `int` to `char` and `double` to `int`]. On the other hand, function overloading can increase the chances that an unsuitable argument will be rejected by the compiler.

◄ Quiz 4: Review of C++ References Jump to... Quiz 6: Review of C++ Classes ►