Function Templates: Overloading and Specialization

References:

The material in this handout is collected from the following references:

- Chapter 16 of the text book C++ Primer.
- Chapters 1 and 16 of C++ Templates: The Complete Guide.
- Chapter 10 of More Exceptional C++.

Background on function overloading

We are used to regular functions, or class methods, being overloaded - multiple functions with the same name have different parameter types. Each call invokes the function with the best match of the parameter types to the call arguments, as show in the following example:

```
void foo(int x)  { std::cout << "foo(int): "  << x << '\n'; }
void foo(long x) { std::cout << "foo(long): " << x << '\n'; }
```

If the arguments are a perfect match for one of the overloaded functions with the given name, that function is called:

```
foo(1);  // foo(int)
foo(1L); // foo(long)
```

Otherwise, the compiler considers conversions to the parameter types of the available functions. If one of the functions offers *better* conversions, that function is selected:

```
foo('a');                     // foo(int)
foo(static_cast<short>(1)); // foo(int)
```

Otherwise, the call is ambiguous:

```
foo(1.1f); // compilation error
foo(1.1);  // compilation error
```

As can be seen here, the rules for function overload resolution are arcane.

> ***Note: You're not required to know these arcane rules except for the most obvious cases.***

For example, in order to determine the best match, the compiler ranks the conversions that could be used to convert each argument to the type of its corresponding parameter. Conversions are ranked as follows:

1. An exact match.
2. Match through a `const` conversion. If `T` is a type, we can convert a pointer or a reference to `T` into a pointer or reference to `T const`, respectively:

```cpp
int i {7};
int const& rri {i};  // convert non-const to a reference to int const
int const *pri {&i}; // convert address of nonconst to address of const
```

3. Match through a promotion. The standard defines the rules of integral promotions which boil down to this: integral promotions convert small integral types to a larger integral types. The types `bool`, `char`, `signed char`, `unsigned char`, `short`, and `unsigned short` are promoted to `int` if all possible values of that type fit in an `int`. Otherwise, the value is promoted to `unsigned int`.
4. Match through an arithmetic conversion.
5. Match through a class-type conversion such as from derived to base class pointers.

In the case of multiple arguments, each argument for the chosen function must have the best conversion. There is no *voting* - if a function has three arguments, and two are an exact match for the first overload, while the third one is an exact match for the second overload, then even if the remaining arguments are implicitly convertible to their corresponding parameter types, the overloaded call is ambiguous:

```cpp
void combine(int, double, char);   // overload #1
void combine(long, double, float); // overload #2

int main() {
  // Arguments 1 and 1.2 are exact match for 1st overload while
  // argument 1.3f is an exact match for 2nd overload
  combine(1, 1.2, 1.3f); // error: ambiguous!!!
}
```

Overloading function templates

Function templates can be overloaded by other templates or by ordinary, non-template functions. That is, you can have different function definitions with the same function name so that when that name is used in a function call, a C++ compiler must decide which one of the various candidates to call. The rules for this decision are complicated without templates and become even more complicated with templates. We'll start by writing a short program that illustrates overloading function templates:

```cpp
// maximum of two int values:
int Max(int lhs, int rhs) { return lhs > rhs ? lhs : rhs; }

// maximum of two values of any type:
template<typename T>
T Max(T lhs, T rhs) { return lhs > rhs ? lhs : rhs; }

int main() {
  Max(7, 42);           // calls plain-old-function for two ints
  Max(7.0, 42.0);       // calls Max<double> [by argument deduction]
  Max('a', 'b');        // calls Max<char> [by argument deduction]
  Max<>(7, 42);         // calls Max<int> [by argument deduction]
  Max<double>(7, 42); // calls Max<double> [no argument deduction]
  Max('a', 42.7);       // calls plain-old-function for two ints
}
```

A non-template function can coexist with a function template that has the same name and can be instantiated with the same type. All other factors being equal, the overload resolution process prefers the non-template over one generated from the template.

> ***If there is a non-template function that is a near-perfect match to the call arguments, that function is selected.***

The first call falls under this rule:

```
Max(7, 42); // calls plain-old-function for two ints
```

If the template can generate a function with a better match, however, then the template is selected:

```
Max(7.0, 42.0); // calls Max<double> [by argument deduction]
Max('a', 'b');  // calls Max<char> [by argument deduction]
```

Here, the template is a better match because no conversion from `double` or `char` to `int` is required.

It is also possible to specify explicitly an empty template argument list. This syntax indicates that only templates may resolve a call, but all the template parameters should be deduced from the call arguments:

```
Max<>(7, 42);       // calls Max<int> [by argument deduction]
```

It is also possible to specify explicitly the type of `T` to prevent the compiler from attempting type deduction:

```
Max<double>(7, 42); // calls Max<double> [no argument deduction]
```

> ***Automatic type conversion is not considered for deduced template parameters but is considered for ordinary function parameters.***

Because automatic type conversion is not considered for deduced template parameters but is considered for ordinary function parameters, the last call uses the non-template function [with `'a'` and `42.7` both are converted to `int`]:

```
Max('a', 42.7); // calls plain-old-function for two ints
```

A useful example would be to overload the maximum template for pointers, C-strings, and for computing maximum of three values of any type:

```
// maximum of two C-strings [include <cstring>]:
const char* Max(char const *lhs, char const *rhs) {
  return std::strcmp(rhs, lhs) < 0 ? lhs : rhs;
}

// maximum of two values of any type:
template<typename T>
T Max(T lhs, T rhs) { return rhs < lhs ? lhs : rhs; }

// maximum of two pointers:
template<typename T>
```

```cpp
T* Max(T *lhs, T *rhs) { return *rhs < *lhs ? lhs : rhs; }

// maximum of three values of any type:
template<typename T>
T Max(T a, T b, T c) { return Max(Max(a, b), c); }

int main() {
  int a {7}, b {42};
  Max(a, b); // Max<int> [by argument deduction]

  std::string s1 {"hello"}, s2 {"world"};
  Max(s1, s2); // Max<std::string> [by argument deduction]

  int *pa {&a}, *pb {&b};
  Max(pa, pb); // Max<int*,int*> overload for pointers

  char const *pstr1 {"hello"}, *pstr2 {"world"};
  Max(pstr1, pstr2); // Max(const char*, const char*)

  Max(a, b, 33); // calls plain-old-function for two ints
}
```

There is one thing to remember:

> ***Ensure all overloaded versions of a function are declared before the function is called.***

This is because the fact that not all overloaded functions are visible when a corresponding function call is made may matter. For example, defining a three-argument version of `Max` without having seen the declaration of a special two-argument version of `Max` for `int`s causes the two-argument template to be used by the three-argument version:

```cpp
// maximum of two values of any type:
template<typename T>
T Max(T lhs, T rhs) { return rhs < lhs ? lhs : rhs; }

// maximum of three values of any type:
template<typename T>
// uses the template version even for ints because the declaration
// int Max(int, int);
// comes too late
T Max(T a, T b, T c) { return Max(Max(a, b), c); }

// maximum of two int values:
int Max(int lhs, int rhs) { return rhs < lhs ? lhs : rhs; }

int main() {
  Max(47, 11, 22); // OOPS: uses Max<T>() instead of Max(int, int)
}
```

Explicit function template specialization

It is not always possible to write a single template that is best suited for every possible template argument with which the template might be instantiated. Explicit template specialization lets function templates deal with special cases. Sometimes a generic algorithm can work much more efficiently for a certain kind of sequence, and so it makes sense to specialize it for that case while using the slower but more generic approach for all other cases. For example, given the function template:

```
template<typename T> void sort(Array<T>& v) { /*...*/ };
```

if we've a faster [or other specialized] way we want to deal specifically with arrays of `char*`s, we could explicitly specialize:

```
template<> void sort<char*>(Array<char*>&);
```

The compiler will then choose the most appropriate template:

```
Array<int>   ai;
Array<char*> apc;

sort(ai);   // calls sort<int> thro' template argument deduction
sort(apc);  // calls specialized sort<char*>
```

Performance is a common reason to specialize, but it's not the only one; for example, you might also specialize a template to work with certain objects that don't conform to the normal interface expected by the generic template. Consider the following overloads of function template `compare`:

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

// compare two string literals (include <cstring>):
template<size_t M, size_t N>
int compare(char const (&lhs)[M], char const (&rhs)[N]) {
  return std::strcmp(lhs, rhs);
}
```

The `compare` function is a good example of a function template for which neither of the two overloads are appropriate for C-strings:

```
char const *p1 = "hello", *p2 = "world";
compare(p1, p2); // calls first template
compare("hello", "bonjour"); // calls second template
```

> ***Note: In a call to `compare`, when the string literal arguments have the same type, `g++` and `clang++` are unable to resolve between the two function template overloads:***

```
compare("hello", "world"); // ambiguous call
```

I can't figure out which specific one of the overload resolution rules is causing this ambiguous behavior. The only way to solve this ambiguity is to add a third function template overload:

```cpp
// compare two string literals with same count of characters:
template<size_t M>
int compare(char const (&lhs)[M], char const (&rhs)[M]) {
return std::strcmp(lhs, rhs);
}
```

To handle C-strings, we can define a explicit template specialization of the first overload of `compare`. An explicit specialization of a function template is a separate definition of the template where we must supply arguments for every template parameter in the original template. Further, the function parameter type(s) must match the corresponding types in that original template. Since we wish to specialize the function template overload that compares any two types for C-strings, the definition of the specialization would look like this:

```cpp
// special version of compare to handle C-strings
template<>
int compare(char const* const& lhs, char const* const& rhs) {
  return std::strcmp(lhs, rhs);
}
```

Difference between overloading and specialization

It is important to remember that a function template specialization supplies every template parameter in a function template. Therefore, a specialization is not an overload but is instead a specific instantiation of a certain function template that we call a *base template*. A base template is a function template that can possibly be overloaded. This means that a specialization is a specific instance of a base template.

Whether a function is a base template or a specialization of a base template or a non-template function impacts function matching. For example, we've defined two overloads of `compare` function template, one that takes references to `T const&` and one that takes references to array parameters. These are the two base templates. Then we've specialized the first base template for C-strings. The following code fragment declares these functions:

```cpp
// base template 1:
template<typename T> int compare(T const& lhs, T const& rhs);
// base template 2:
template<size_t M, size_t N>
int compare(char const (&lhs)[M], char const (&rhs)[N]);
// specialization of base template 1:
template<> int compare(char const* const& lhs, char const* const& rhs);
```

When we call `compare` for string literals

```cpp
compare("hello", "worlds"); // calls base template 2
```

both base templates are viable. Which base template gets selected depends on which matches best and is the "most specialized" [important note: this use of "specialized" has nothing to do with template specializations; it's just an unfortunate colloquialism] according to a set of fairly arcane rules. If it's clear that there's one "most specialized" base template, that one gets used. If that base template happens to be "specialized" for the types being used, the specialization will get used, otherwise the base template instantiated with the correct types will be used. In this particular case, the second base template is considered most "specialized".

When we call `compare` for C-strings

```cpp
char const *pstr1{"hello"}, *pstr2{"world"};
compare(pstr1, pstr2); // calls specialization of base template 1
```

it is clear that the first base template best matches the template argument `char const*`. Since this base template is further specialized for `char const*`, that specialization gets chosen.

Overloading rules

Let's review what we have discussed so far. Non-template functions having the same name do overload, and so function templates are allowed to overload too. This is pretty obvious and is summarized in the following code fragment:

```cpp
// function template with two overloads
template<typename T> void foo(T); // #b
template<typename T> void foo(int, T, double); // #c
```

These unspecialized templates are also called the underlying *base templates*. Further, base templates can be *specialized*. The following code illustrates specializations of base templates:

```cpp
// a separate base template that overloads #b and #c
template<typename T> void foo(T*);   // #d
// a specialization of #b for int
template<> void foo<int>(int);       // #e
// a non-template function that overloads with #b, #c, #d, but not #e
void foo(double); // #f
```

Let's review the overloading rules for function templates to see which ones get called in different situations. The rules are pretty simple, at least at a high level, and can be expressed as a classic two-class system:

1. Non-template functions are ***first-class citizens***. A non-template function that *matches the parameter types as well as* any function template will be selected over an otherwise-just-as-good function template.
2. If there are no first-class citizens to choose from that are at least as good, then function base templates as the second-class citizens get consulted next. Which function base template gets selected depends on which matches best and is the "most specialized" [again note that this use of "specialized" has nothing to do with template specializations] according to a set of fairly arcane rules:
    1. If it's clear that there's one "most specialized" function base template, that one gets used. If that base template happens to be specialized for the types being used, the specialization will get used, otherwise the base template instantiated with the correct types will be used.
    2. Else if there's a tie for the "most specialized" function base template, the call is ambiguous because the compiler can't decide which is a better match. The programmer will have to do something to qualify the call and say which one is wanted.
    3. Else if there's no function base template that can be made to match, the call is bad and the programmer will have to fix the code.

Putting these rules together, here's a sample of what we get in terms of overload resolution:

```cpp
bool b;
int i;
double d;

foo(b);         // calls #b with T = bool
foo(i, 42, d); // calls #c with T = int
foo(&i);        // calls #d with T = int
foo(i);         // calls #e
foo(d);         // calls #f
```

Function overloading versus template specialization

Should we define a particular function as a specialization or as overloaded function template or as non-template function? The general answer is to ***avoid writing specializations***. To understand why, consider the following code:

```cpp
template<typename T> void foo(T);  // #a: base template
template<typename T> void foo(T*); // #b: base template - overloads #a
template<> void foo<>(int*); // #c: explicit specialization of #b

int *p;
foo(p); // calls #c
```

It seems intuitive that since we wrote an explicit specialization for an `int*`, that is what obviously would be called. Consider now the following code:

```cpp
template<typename T> void foo(T);  // #a: base template [as before]
template<> void foo<>(int*); // #c: explicit specialization of #a now!!!
template<typename T> void foo(T*); // #b: base template - overloads #a

int *p;
foo(p); // calls #b!!! Overload resolution ignores specializations
        // and operates on the base function templates only!!!
```

It is surprising that the call `foo(p)` will now call the second base template. The key to understanding this is simple, and here it is: *Specializations don't overload*. Only the base templates overload (along with non-template functions, of course). Consider again the salient part from the summary above of overload resolution rules, this time with specific words highlighted:

> If there are no first-class citizens to choose from that are at least as good, then ***function base templates*** as the second-class citizens get consulted next. Which ***function base template*** gets selected depends on which matches best and is the "most specialized" [...] according to a set of fairly arcane rules:
>
> ...
>
> If it's clear that there's one "most specialized" ***function base template***, that one gets used. If that ***base template*** happens to be specialized for the types being used, the specialization will get used, otherwise the base template instantiated with the correct types will be used.
>
> ... etc.

Overload resolution only selects a base template (or a non-template function, if one is available). Only after it's been decided which base template is going to be selected, and that choice is locked in, will the compiler look around to see if there happens to be a suitable specialization of that template available, and if so that specialization will get used.

If you ask the question: "Hmm. But it seems to me that I went and specifically wrote a specialization for the case when the parameter is an `int`*, and it is an* `int` which is an exact match, so shouldn't my specialization always get used?" That, alas, is a mistake: If you want to be sure it will always be used in the case of exact match, that's what a non-template function is for - so just make it a non-template function instead of a specialization.

The rationale for why specializations don't participate in overloading is simple, once explained, because the surprise factor is exactly the reverse: The standards committee felt *it would be surprising that, just because you happened to write a specialization for a particular template, that it would in any way change which template gets used*. Under that rationale, and since we already have a way of making sure our version gets used if that's what we want (we just make it a function, not a specialization), we can understand more clearly why specializations don't affect which template gets selected.

> ***If you want to customize a function base template and want that customization to participate in overload resolution (or, to always be used in the case of exact match), make it a plain old function, not a specialization. And, if you do provide overloads, avoid also providing specializations.***

Summary

It's okay to overload function templates. Overload resolution considers all base templates equally and so it works as you would naturally expect from your experience with normal C++ function overloading: Whatever templates are visible are considered for overload resolution, and the compiler simply picks the best match.

It's a lot less intuitive to specialize function templates because *specializations don't overload*. This means that any specializations you write will not affect which template gets used, which runs counter to what most people would intuitively expect. After all, if you had written a non-template function with the identical signature instead of a function template specialization, the non-template function would always be selected because it's always considered to be a better match than a template.

If you're using someone else's plain old function template and you want to write your own special-case version that should participate in overloading, don't make it a specialization; just make it an overloaded function with the same signature.