

# HIGH-LEVEL PROGRAMMING 2

Function Templates: Overloading and Specialization  
by Prasanna Ghali

# Function Overloading (1 / 3)

2

- C++ lets us overload nontemplate functions, yet makes sure the right one is called:

```
int foo(int);           // 1
double foo(double);    // 2

int i;
double d;
foo(i);                // exact match with 1
foo(d);                // exact match with 2
foo('a');              // ???
foo(i+d);              // ???
foo(10.1f);            // ???
foo(10.1L);            // ???
foo(10UL);             // ???
```

# Function Overloading (2/3)

3

- Nontemplate function can coexist with function template that has same name and can be instantiated with same type

```
int Max(int lhs, int rhs) {                // 1
    return lhs > rhs ? lhs : rhs;
}

template <typename T>
T Max(T const& lhs, T const& rhs) { // 2
    return lhs > rhs ? lhs : rhs;
}

Max(7, 42);                               // ???
```

# Function Overloading (3/3)

4

- All other factors being equal, overload resolution process prefers nontemplate over one generated from template

```
int Max(int lhs, int rhs) {           // 1
    return lhs > rhs ? lhs : rhs;
}

template <typename T>
T Max(T const& lhs, T const& rhs) { // 2
    return lhs > rhs ? lhs : rhs;
}
```

```
Max(7, 42);           // ???
Max(7.0, 42.0);       // ???
Max('a', 'b');        // ???
Max<double>(7, 42);    // ???
Max<>(7, 42);          // ???
Max('a', 42.7);       // ???
```

This syntax makes it possible to specify explicitly an *empty template argument list*. This syntax indicates that only templates may resolve a call, but *template parameters must be deduced from call arguments!!!*

# Why Overload Function Templates?

## (1 / 3)

5

- Function templates can themselves be overloaded
  - ▣ Performance is common reason to overload
  - ▣ We might want to overload a template to work with certain objects that don't conform to normal interface expected by generic template

# Why Overload Function Templates?

## (2/3)

6

- `<algorithm>` defines function template `std::swap` to exchange two values
- `"stack.hpp"` defines member function to exchange with another stack

```
namespace std {  
    template <typename T>  
    void swap(T &lhs, T &rhs) {  
        T tmp{lhs};  
        lhs = rhs;  
        rhs = tmp;  
    }  
}
```

```
template <typename T>  
class Stack {  
public:  
    // public interface ....  
    void swap(Stack&);  
private:  
    size_t max_sz;  
    size_t top_idx;  
    T      *v;  
};
```

# Why Overload Function Templates?

## (3/3)?

7

- Using `std::swap` to exchange two values of type `Stack<T>` is expensive!!!
- Standard library requires every container to overload `std::swap`

```
namespace std {  
    template <typename T>  
    void swap(T &lhs, T &rhs) {  
        T tmp{lhs};  
        lhs = rhs;  
        rhs = tmp;  
    }  
}
```

```
template <typename T>  
class Stack {  
public:  
    void swap(Stack&);  
private:  
    size_t max_sz;  
    size_t top_idx;  
    T      *v;  
};  
  
template <typename T>  
void swap(Stack<T> &a, Stack<T> &b) {  
    a.swap(b);  
}
```

# Overloading Function Templates

*// 1: maximum of two values of any type*

```
template <typename T>
T Max(T lhs, T rhs) {
    return lhs > rhs ? lhs : rhs;
}
```

*// 2: maximum of two pointers*

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

*// 3: maximum of two C-strings*

```
char const* Max(char const *lhs, char const *rhs) {
    return std::strcmp(lhs, rhs) > 0 ? lhs : rhs;
}
```

*// 4: 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 i1{7}, i2{42};
int *p1{&i1}, *p2{&i2};
std::string s1{"hey"}, s2{"you"};
char const *c1{"you"}, *c2{"hey"};
char const *c3{"hey you"};
```

```
Max(i1, i2);           // ???
Max(s1, s2);           // ???
Max(p1, p2);           // ???
Max(c1, c2);           // ???
Max(c1, c2, c3);       // ???
```



# Overloading Function Templates

## (2/11)

9

- Note that in all overloads of **Max**, we pass arguments by value
- In general, good idea not to change more than necessary when overloading function templates
  - ▣ You should limit changes to number of parameters or to specifying template parameters explicitly
- Otherwise, unexpected effects may happen

# Overloading Function Templates

(3/11)

10

*// max of two values*

```
template <typename T>
```

```
T const& Max(T const& lhs, T const& rhs) {
```

```
    return lhs > rhs ? lhs : rhs;
```

```
}
```

Plain function Max overload is pass-by-value

*// incorrect version: max of two C-strings*

```
char const* Max(char const *lhs, char const *rhs) {
```

```
    return std::strcmp(lhs, rhs) > 0 ? lhs : rhs;
```

```
}
```

*// max of three values*

```
template <typename T>
```

```
T const& Max(T const& a, T const& b, T const& c) {
```

```
    return Max(Max(a, b), c);
```

```
}
```

```
char const *c1{"you"};
```

```
char const *c2{"hey"};
```

```
char const *c3{"hey you"};
```

```
Max(c1, c2, c3); // error!!! ???
```

Function templates Max have changed from pass-by-value to pass-by-const-reference

# Overloading Function Templates

(4/11)

11

```
// max of two values
template <typename T>
T const& Max(T const& lhs, T const& rhs) {
    return lhs > rhs ? lhs : rhs;
}
```


```
// correct version: max of two C-strings
char const* const&
Max(char const* const& lhs, char const* const& rhs) {
    return std::strcmp(lhs, rhs) > 0 ? lhs : rhs;
}
```

```
// max of three values
template <typename T>
T const& Max(T const& a, T const& b, T const& c) {
    return Max(Max(a, b), c);
}
```

```
char const *c1{"you"};
char const *c2{"hey"};
char const *c3{"hey you"};
```

```
Max(c1, c2, c3); // ok: ???
```

Plain function Max changed  
from pass-by-value to pass-  
by-const-reference



# Overloading Function Templates

## (5/11)

12

```
// 1: max of two values of any type
template <typename T>
T Max(T lhs, T rhs) {
    return lhs > rhs ? lhs : rhs;
}
```

Ensure all overloaded versions  
of a function are declared  
before the function is called!!!

```
// 2: max of three values of any type
template <typename T>
T Max(T a, T b, T c) {
    return Max(Max(a, b), c);
}
```

```
int i1{47}, i2{11}, i3{33};
```

*// problem when Max is called!!!*

```
Max(i1, i2, i3); // ???
```

```
// 3: max of two int values
int Max(int lhs, int rhs) {
    return lhs > rhs ? lhs : rhs;
}
```

# Overloading Function Templates

## (6/11)

13

*// 1: max of two values of any type*

```
template <typename T>
T Max(T lhs, T rhs) {
    return lhs > rhs ? lhs : rhs;
}
```

*// 2: max of two int values*

```
int Max(int lhs, int rhs) {
    return lhs > rhs ? lhs : rhs;
}
```

*// 3: max of three values of any type*

```
template <typename T>
T Max(T a, T b, T c) {
    return Max(Max(a, b), c);
}
```

Since nontemplate function overload is now declared before Max of 3 values, it is now visible to Max of 3 values!!!

```
int i1{47}, i2{11}, i3{33};
```

```
// calls nontemplate function!!!
Max(i1, i2, i3); // ???
```

# Overloading Function Templates

## (7/11)

14

- Two function templates with same name can coexist even though they may be instantiated so that both have identical parameter types

# Overloading Function Templates

## (8/11)

15

```
// 1
template <typename T>
int f(T) {
    return 1;
}

// 2
template <typename T>
int f(T*) {
    return 2;
}

int x{10}, *px{&x};

f<int*>(px); // ???
f<int>(px);  // ???
```

# Overloading Function Templates

## (9/11)

16

```
// 1
template <typename T>
int f(T) {
    return 1;
}
```

After substituting given template arguments (`<int*>` and `<int>`), overload resolution ends up picking the right function to call

```
// 2
template <typename T>
int f(T*) {
    return 2;
}
```

Two function templates with same name can coexist even though they may be instantiated so that both have identical parameter types

```
int x{10}, *px{&x};
```

```
f<int*>(px); // calls f<T>(T) == f(int*)
f<int>(px);  // calls f<T>(T*) == f(int*)
```



# Overloading Function Templates

## (10/11)

17

```
// 1
template <typename T>
int f(T) {
    return 1;
}

// 2
template <typename T>
int f(T*) {
    return 2;
}

int x{10}, *px{&x};

f(*px); // ???
f(px);  // ???
```

# Overloading Function Templates

## (11/11)

18

```
// 1
template <typename T>
int f(T) {
    return 1;
}
```

```
// 2
template <typename T>
int f(T*) {
    return 2;
}
```

```
int x{10}, *px{&x};
```

```
f(*px); // calls f<T>(T)
```

```
f(px); // can call either f<int*>(int*) or f<int>(int*)
        // special overloading rules will pick f<int>(int*)
        // because it is more specialized!!!
```

Even without explicit template arguments, template argument deduction and special overloading rules will select right function to call!!!

For expression `f(px)`, compiler can instantiate either function template 1 or 2. However, compiler will instantiate function template 2 since 2's instantiation is more *specialized* than 1 [because 2 takes fewer types than 1]!!!

# Function Template Specialization

(1 / 5)

19

- Not always possible to write single template best suited for every possible template argument with which template might be instantiated
  - ▣ In some cases, general template definition is simply wrong for a type and will not compile!!!
  - ▣ At other times, we might be able to take advantage of some specific knowledge to write more efficient code than would be instantiated from template
- When we can't or don't want to use template version, we can define ***specialized*** version of function template

# Function Template Specialization

## (2/5)

20

```
// 1: max of two values of any type
template <typename T>
const T& Max(T const& lhs, T const& rhs) {
    return lhs > rhs ? lhs : rhs;
}

// 2: max of two pointers of any type
template <typename T>
T* const& Max(T* const& lhs, T* const& rhs) {
    return *lhs > *rhs ? lhs : rhs;
}

int i1{1}, i2{2}, *pi1{&i1}, *pi2{&i2};
char const *pc1 = "San Jose", *pc2 = "Santiago";
Max(1, 2);          // ok: calls 1 with T == int
Max(pi1, pi2);      // ok: calls 2 with T == int
Max(pc1, pc2);      // error!!! calls 2 with T == char const
```

# Function Template Specialization

## (3/5)

21

- When we specialize a function template, we must supply arguments for ***every template parameter*** in that [previously defined] template

# Function Template Specialization

## (4/5)

22

*// base template 1: max of two values of any type*

```
template <typename T>
const T& Max(T const& lhs, T const& rhs) {
    return lhs > rhs ? lhs : rhs;
}
```

Function template specialization is introduced by sequence of three tokens: **template**, **<**, and **>**

*// base template 2: max of two pointers of any type*

```
template <typename T>
T* const& Max(T* const& lhs, T* const& rhs) {
    return *lhs > *rhs ? lhs : rhs;
}
```

*// 3: special version of 2 to handle pointers to char arrays*

```
template <>
const char* const& Max(char const* const& lhs, char const* const& rhs) {
    return std::strcmp(lhs, rhs) > 0 ? lhs : rhs;
}
```

```
char const *pc1 = "San Jose", *pc2 = "Santiago";
Max(pc1, pc2); // ok: calls 3 with T == char const
```

# Function Template Specialization

## (5/5)

23

- Specializations instantiate a base template, they don't overload it!!!
- As a result, specializations don't participate in function matching

# Function Templates: Simplified Overload Rules (1 / 2)

24

- Nontemplate functions are first-class citizens
  - ▣ Nontemplate function that matches parameter types as well as any function template will be selected over otherwise-just-as-good function template
- If there are no first-class citizens to choose from that are at least as good, then function base templates as second-class citizens get consulted next based on which matches best and is “most specialized” according to 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 base template instantiated with correct types will be used
  - ▣ Else if there's tie for "most specialized" function base template, call is ambiguous because compiler can't decide which is a better match; programmer will have to do something to qualify the call and say which one is wanted
  - ▣ Else if there's no function base template that can be made to match, call is bad; programmer will have to fix the code



# Function Templates: Simplified Overload Rules (2/2)

25

```
template <typename T> void f(T);           // a
template <typename T> void f(int, T, double); // b
template <typename T> void f(T*);          // c
template <> void f(int); // d [specialization of a]
void f(double);                            // e
```

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

```
// specify which function is called and template type parameter
f(b);           // ???
f(i, 42, d);    // ???
f(&i);          // ???
f(i);           // ???
f(d);           // ???
```

# Don't Specialize Function Templates!!! (1 / 2)

26

```
template <typename T> // a
void f(T) {
    std::cout << "BT 1\n";
}
```

```
template <typename T> // b
void f(T*) {
    std::cout << "BT 2\n";
}
```

```
template<> // c
void f(int*) {
    std::cout << "BT 2 "
               "specialization\n";
}
```

```
int *p;
f(p); // ???
```

```
template <typename T> // a
void f(T) {
    std::cout << "BT 1\n";
}
```

```
template<> // c
void f(int*) {
    std::cout << "BT 1"
               "specialization\n";
}
```

```
template <typename T> // b
void f(T*) {
    std::cout << "BT 2\n";
}
```

```
int *p;
f(p); // ???
```

# Don't Specialize Function Templates!!! (2/2)

27

- Key to understanding surprising behavior is this: ***Specializations don't overload; only base templates do!!!***
- Moral: If you want to customize 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 nontemplate function, not a specialization

# Review

28

- What is function template overloading?
- Why overload function templates?
- What is function template specialization?