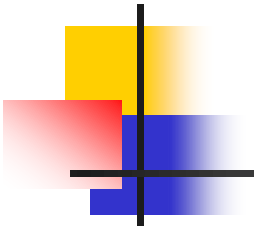
Templates

Prasanna Ghali



Outline

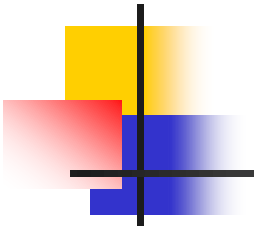
- Function templates
  - Type and non-type parameters
  - Overloading
  - Specialization
  - Template type deduction
  - Variadic templates

Function Template Overloading (1/14)

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

```
int foo(int);
double foo(double);

int i;
double d;

foo( i );    // calls foo(int)
foo( d );    // calls foo(double)
```

Function Template Overloading (2/14)



- Like POFs, function templates can also be overloaded
- Performance is common reason to overload
- But not the only reason
- We might want to overload a template to work with certain objects that don't conform to normal interface expected by generic template

Function Template Overloading (4/14)

- Let's overload `Max` template for three values, two pointers and two ordinary C-strings

```
// maximum of two values of any type
template <typename T>
inline T const& Max(T const& a, T const& b) {
  return a < b ? b : a;
}
```

Function Template Overloading (5/14)

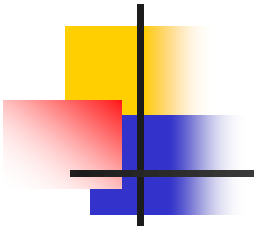
```
// maximum of two values of any type
template <typename T>
inline T const& Max(T const& a, T const& b) {
  return a < b ? b : a;
}

// maximum of three values of any type
template <typename T>
inline T const& Max(T const& a, T const& b, T const& c) {
  // error if Max(a,b) uses call-by-value
  return Max( (Max(a,b), c );
}
```

Function Template Overloading (6/14)

```
// maximum of two pointers
template <typename T>
inline T* const& Max(T* const& a, T* const& b) {
  return *a < *b ? b : a;
}

// maximum of two C-strings
inline char const* const& Max(char const* const& a,
                              char const* const& b) {
  return std::strcmp(a,b) < 0 ? b : a;
}
```

Function Template Overloading (7/14)

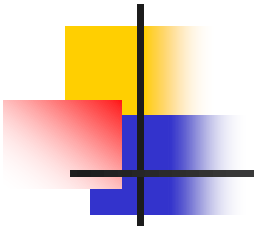
```
int a{7},b{42};
Max(a,b); // Max() for two values of type int

string s{"good"}, t{"luck"};
Max(s,t); // Max() for two values of type string

int *pa{&a}, *pb{&b};
Max(pa, pb); // Max() for two pointers
```



Function Template Specialization (1/4)

- Not same thing as function template overloading – but related
- Why specialize function templates?
- Same reasons one would overload function templates
- Allows templates to deal with special cases

Function Template Specialization (2/4)



- Sometimes, generic algorithm can work more efficiently for certain object types
- Makes sense then to *specialize* it for this case while using slower but more generic approach for all other cases

Overloading vs. Specialization (1/8)

- First, a quick review of what these terms mean
- POFs having the same name overload, and so function templates are allowed to overload too

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

- These unspecialized function templates are called the underlying function ***base templates***

Overloading vs. Specialization (2a/8)

- Further, function base templates can be ***specialized***
- Function templates can only be ***fully specialized***

```
// separate base template that overloads #a and #b
template<typename T> void f ( T* ); // #c

// full specialization of #a for int
template<> void f<int>( int ); // #d

// POF overloads with #a, #b, and #c, but not #d
void f( double ); // #e
```

Overloading vs. Specialization (2b/8)

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

// separate base template that overloads #a and #b
template<typename T> void f ( T* );         // #c

// full specialization of #a for int
template<> void f<int>( int );              // #d

// POF overloads with #a, #b, and #c, but not #d
void f( double );                           // #e
```

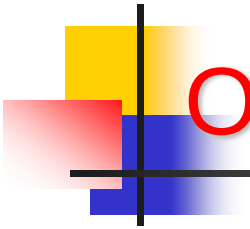
Overloading vs. Specialization (3/8)

- What are the overloading rules that determine which ones get called in different situations?
- Rules quite simple (not exactly but let's simplify) and can be expressed as a ***two-class system***

Overloading vs. Specialization (4/8)

- POFs (nontemplates) are first-class citizens
  - POF that matches parameter types as well as any function template will be selected over an otherwise just-as-good function template

Overloading vs. Specialization (5/8)



- If no suitable first-class citizens, then function ***base*** templates as second-class citizens get consulted next
- Which base template gets selected depends on which matches best
- Rules are fairly arcane

Overloading vs. Specialization (6/8)

- If there's one "best matching" function base template, that one gets used
  - If that base template is ***specialized*** for types in call, specialization will get used
  - Otherwise, base template instantiated with correct types will be used

Overloading vs. Specialization (7/8)

- Else, if there's tie for "best matching" function base template, the call is ambiguous

Overloading vs. Specialization (8/8)

- Else, if there's no function base template that can be made to match, the call is bad

Exercise: Function Template Specialization and Overloading (1/2)

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

// a separate base template that overloads #a and #b
template<typename T> void f ( T* );                 // #c

// full specialization of #a for int
template<> void f<int>( int );                      // #d

// POF overloads with #a, #b, and #c, but not #d
void f( double );                                   // #e
```

Exercise: Function Template Specialization and Overloading (2/2)

- To understand overloading rules, determine which function (#a - #e) is invoked for each of following calls:

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

f( b ); // 1
f( i, 42, d ); // 2
f( &i ); // 3
f( i );  // 4
f( d );  // 5
```

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

// separate base template overloading #a & #b
template<typename T> void f ( T* );  // #c

// full specialization of #a for int
template<> void f<int>( int );       // #d

// POF overloads with #a, #b, and #c, but not #d
void f( double );                    // #e
```

Why You Shouldn't Specialize Function Templates (1/6)

- Consider following code:

```
template <typename T> void f( T ); // #a – a base template
template <typename T>
void f( T* );  // #b – 2nd base template overloading #a

template<> void f<>( int* );   // #c – explicit specialization of #b

// …

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

- Result of call is as expected but …

Why You Shouldn't Specialize Function Templates (2/6)

- Result of call to function is as expected
- But consider the reasoning:
  - We wrote a specialization for int*, so obviously that's what is called when we pass a pointer to int
- This reasoning can lead to unwanted subtleties

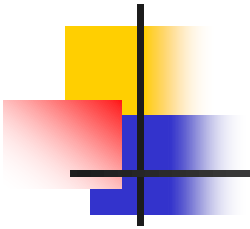
Why You Shouldn't Specialize Function Templates (3/6)

- Now, consider following code:

```
template <typename T> void f( T ); // #a – a base template
template<> void f<>( int* );   // #c – explicit specialization of #a

template <typename T>
void f( T* );  // #b – a 2nd base template overloading #a

// …

int* p;
f( p );   // calls #b!!!!
```

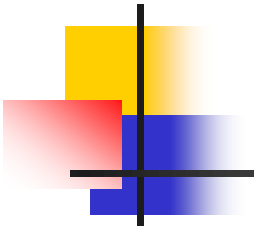
- Result of call is surprising!!!

Why You Shouldn't Specialize Function Templates (4/6)



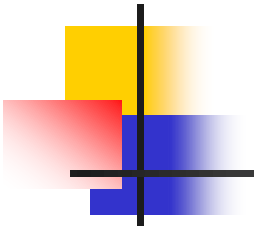
- Key to understand behavior: ***Specializations don't overload***
- Only base templates overload (along with POFs)
- What does this mean?
- Means any specializations you write will not affect which template gets used
  - Runs counter to what most people would intuitively expect

Why You Shouldn't Specialize Function Templates (5/6)



- Overload resolution only selects a POF or base template
- After *locking* which base template is to be used, the compiler will check if there is suitable specialization available
- If so, that specialization will be used

Why You Shouldn't Specialize Function Templates (6/6)



- Moral of story: Avoid specializations
- If you want to customize function base template, make it a POF, not a specialization

Class Template `initializer_list<>` (1/7)

- In `<initializer_list>`, C++11 presents class template `std::initializer_list<T>` to represent array of values of type `T`

```
initializer_list<int> x; // empty list

// y has 3 elements which are copies of initializers
initializer_list<int> y{1,2,3};

// z has 4 elements each constructed by conversion ctor:
// string(char const*)
initializer_list<string> z{"a", "b", "c", "d"};
```

Class Template initializer_list<> (2/7)

- Copying or assigning initializer_list<T> doesn't copy elements – original and copy share elements

```
initializer_list<string> z{"a","b","c"};

// z2 and z share string elements
initializer_list<string> z2(z);

initializer_list<string> z3{z2}; // error
```

Class Template `initializer_list<>` (3/7)

- Elements of `initializer_list<T>` are immutable – that is, elements are `const` values
- Elements in `initializer_list<T>` can be accessed as a sequence using member functions `begin()`, `end()`, and `size()`

Class Template initializer_list<> (4/7)

```
initializer_list<string> z{"a","b","c"};
*z.begin() = "d";   // error: changing value of const element

// use begin() and end() to iterate through sequence of elements
for (string const* it = z.begin();it != z.end(); ++it) {
  std::cout << *it << std::endl;
}

// can use range for statement
for(string const& elem : z) {
  std::cout << elem << std::endl;
}
```

Class Template initializer_list<> (5/7)

- Doesn't provide subscripting
  - We can simulate subscript operator using `begin()` and `size()` member functions

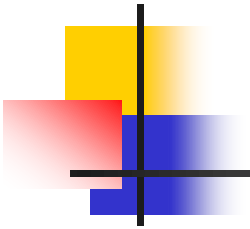
```
initializer_list<string> z{"a","b","c"};
for (size_t i=0; i<z.size(); ++i) {
 std::cout << z.begin()[i] << '\n';
}
```

Class Template `initializer_list<>` (6/7)

- We can write functions that take unknown number of arguments of same type by using `initializer_list<T>` parameter
- `initializer_list<T>` is passed by value – no overhead is imposed because object is handle to array of `T`s

Class Template initializer_list<> (7/7)

```
template <typename T>
T  adder(std::initializer_list<T>  x)
{
 T sum = T(); // default initialization
 for (T const& e : x)
  sum += e;
 return sum;
}
```



Initializer-list Constructor (1/5)

- Constructor that takes single argument of type `std::initializer_list` is called *initializer-list ctor*
  - Constructs objects using a `{}`-list as its initializer value
- Let's define an initializer-list ctor for `Str`:

Initializer-list Constructor (2/5)

```
class Str {
public:
 Str();
 Str(char const*);
 Str(Str const&);
 Str(initializer_list<char>);
// ...
private:
 size_t n;
 char* p;
};
```

Initializer-list Constructor (3/5)

```
Str::Str(initializer_list<char> x) :
n(x.size()), p(new char [n+1]) {
 char* q;
 q = copy(x.begin(), x.end(), p);
 *q = '\0';
}
```

Initializer-list Constructor (4/5)

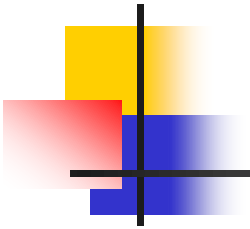
- Now, we can write:

```
Str s1{"a"}; // call Str(char const*)

Str s2{'a','b'}; // call initializer_list ctor

s1 = s2; // call op=(Str const&)

s2 = {'x'}; // call op=(initializer_list<char>)
```

Initializer-list Constructor (5/5)

- Standard-library containers have initializer-list ctors, assignments, etc

```
vector<double> v{1.2,3.4,5.6};

list<pair<string,string>> grades = {
 {"Tom","A"}, {"Dick","B"}, {"Harry","C"}
};

map<vector<string>,vector<int>> years = {
 { {"Tom","Rich","Scot"},{1960,1962,1975,1980} },
 { {"Clint","Nick"},{1953,1984,1995} },
};
```



Variadic Templates: Introduction

- For next lecture!!!!
- Template function or class that can take varying number and types of (function and template) parameters
- Useful when we know neither number nor types of arguments to be processed in call to function