HIGH-LEVEL PROGRAMMING 2

Class Templates: Specialization

by Prasanna Ghali

Class Template Specialization

- Just as with function templates, you can specialize class templates
  - Recall we specialize templates to optimize implementations for certain types or fix misbehavior of certain types for instantiations of class templates
- Unlike function templates, class templates cannot be overloaded
- This leads to two forms of specializations for class templates: full specialization and partial specialization

Full Class Template Specialization (1/3)

- Full specialization provides an implementation for a template with template parameters that are fully substituted
- We want to fully specialize our stack example:

```
// primary class template ...
template <typename T>
class Stack {
public:
  // interface ...
private:
  std::vector<T> elems;
};
```

```
// full specialization ...
template <>
class Stack<std::string> {
public:
  // interface ...
private:
  std::deque<std::string> data;
};
```

Full Class Template Specialization (2/3)

- All member functions of primary class template must be specialized ...

```cpp
// primary class template ...
class Stack {
public:
  // interface ...
  void push(T const&);
private:
  std::vector<T> elems;
};

template <typename T>
void Stack<T>::push(T const& elem) {
  elems.push_back(elem);
}
```

```cpp
// full specialization ...
template <>
class Stack<std::string> {
public:
  // interface ...
  void push(std::string const&);
private:
  std::deque<std::string> data;
};

void Stack<std::string>::
push(std::string const& elem) {
  data.push_back(elem);
}
```

Full Class Template Specialization (3/3)

- Implementation of full specialization doesn't have to be related to primary class template

```cpp
// primary class template ...
template <typename T>
struct S {
  void info() const;
};

template <typename T>
void S<T>::info() const {
  std::cout <<
    __PRETTY_FUNCTION__ << "\n";
}
```

```cpp
// full specialization ...
template <>
struct S<void> {
  void msg();
};

void S<void>::msg() {
  std::cout <<
    __PRETTY_FUNCTION__ << "\n";
}
```

Partial Class Template Specialization (1/2)

- Partial specialization provides special implementations for particular circumstances, but some template parameters must still be substituted
- Partial specialization of `Stack<T>` for pointers:

```cpp
// primary class template ...
template <typename T>
class Stack {
public:
  // interface ...
private:
  std::vector<T> elems;
};
```

```cpp
// partial specialization ...
template <typename T>
class Stack<T*> {
public:
  // interface ...
private:
  std::vector<T*> elems;
};
```

Partial Class Template Specialization (1/2)

- Specialization can provide different interface

```cpp
// primary class template ...
class Stack {
public:
  // interface ...
  void pop();
private:
  std::vector<T> elems;
};

template <typename T>
void Stack<T>::pop() {
  elems.pop_back();
}
```

```cpp
// partial specialization ...
template <typename T>
class Stack<T*> {
public:
  // interface ...
  T* pop();
private:
  std::vector<T*> elems;
};

template <typename T>
T* Stack<T*>::pop() {
 T* p = elems.back();
 elems.pop_back();
 return p;
}
```

Partial Class Template Specialization: Example (1/5)

- Primary template called C that has two template parameters

```
// primary class template ...
template <typename T, size_t N>
struct C {
  void operator()() {
    std::cout << __PRETTY_FUNCTION__ << "\n";
  }
};
```

Partial Class Template Specialization: Example (2/5)

- Partial specialization for nontype template parameter N with value 10

```
// primary class template ...
template <typename T, size_t N>
struct C {
  void operator()() {
    std::cout << __PRETTY_FUNCTION__ << "\n";
  }
};

// partial specialization 1 ...
template <typename T>
struct C<T, 10UL> {
  void operator()() {
    std::cout << __PRETTY_FUNCTION__ << "\n";
  }
};
```

Partial Class Template Specialization: Example (3/5)

- Partial specialization of `int` for type template parameter

```
// primary class template ...
template <typename T, size_t N>
struct C {
  void operator()() {
    std::cout << __PRETTY_FUNCTION__ << "\n";
  }
};

// partial specialization 2 ...
template <size_t N>
struct C<int, N> {
  void operator()() {
    std::cout << __PRETTY_FUNCTION__ << "\n";
  }
};
```

Partial Class Template Specialization: Example (4/5)

- Partial specialization of `int*` for type template parameter

```
// primary class template ...
template <typename T, size_t N>
struct C {
  void operator()() {
    std::cout << __PRETTY_FUNCTION__ << "\n";
  }
};

// partial specialization 3 ...
template <size_t N>
struct C<int*, N> {
  void operator()() {
    std::cout << __PRETTY_FUNCTION__ << "\n";
  }
};
```

Partial Class Template Specialization: Example (5/5)

- Instantiate 4 different types of C

```
int main() {
  C<char, 42>   a; a();
  C<double, 10> b; b();
  C<int, 42>    c; c();
  C<int*, 42>   d; d();
}
```

Templates: Pros

- Help avoid writing repetitive code
- Help implement generic libraries providing algorithms and types which can be used in many applications [decrease development and maintenance costs]
- Using templates can result in less and better code [development and maintenance costs are reduced]

Templates: Cons

- Complex and cumbersome syntax
- Compiler errors relating to templates are lengthy and cryptic!!!
- Increase compilation times because they're [currently] implemented entirely in headers
- Because template code is in headers, there is no information hiding
- Harder to validate because template code that is not used is not instantiated

Review

- What is full class template specialization?
- What is partial class template specialization?
- What are advantages and disadvantages of templates?