Lecture 8

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Templates – C++ Primer Chapter 16

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Templates

- **class template**: Class definition used to define a set of type-specific classes.
  - template<typename T1, typename T2> MyClass {}
- **function template**: Definition for a function that can be used for a variety of types.
  - template<typename T> void print(T value) {}

Template Parameters I

- **template parameter**: Name specified in template parameter list that may be used inside the definition of a template.
  - Template parameters can be type or non-type parameters.
  - To use a class template, actual arguments must be specified for each template parameter.
  - The compiler uses those types or values to instantiate a version of the class in which uses of the parameter(s) are replaced by the actual argument(s).
  - When a function template is used, the compiler deduces the template arguments from the arguments in the call and instantiates a specific function using the deduced template arguments.

Template Parameters II

- **nontype parameter**: Template parameter representing value.
  - When a function template is instantiated, each nontype parameter is bound to a constant expression as indicated by the arguments used in the call.
  - When a class template is instantiated, each nontype parameter is bound to a constant expression as indicated by the arguments used in the class instantiation.
- **type parameter**: Name used in template parameter list to represent a type.
  - When the template is instantiated, each type parameter is bound to an actual type.
  - In a function template, the types are inferred from the argument types or are explicitly specified in the call.
  - Type arguments must be specified for a class template when the class is used.

Template Parameters III

- **template parameter list**: List of type or nontype parameters (separated by commas) to be used in the definition or declaration of a template.
- **template argument**: Type or value specified when using a template type, such as when defining an object or naming a type in a cast.
- Hint: keep the number of requirements placed on argument types as small as possible

Typename keyword

- In template parameter list equivalent to class
- If there is any doubt as to whether typename is necessary to indicate that a name is a type, it is a good idea to specify it
- There is no harm in specifying typename before a type, so if the typename was unnecessary it will not matter

Template Instantiation

- **Instantiation**: Compiler process whereby the actual template argument(s) are used to generate a specific instance of the template in which the parameter(s) are replaced by the corresponding argument(s).
  - Functions are instantiated automatically based on the arguments used in a call.
  - Template arguments must be provided explicitly whenever a class template is used.
- Each instantiation of a class template constitutes an independent class type

Template Argument Deduction I

- **template argument deduction**: Process by which the compiler determines which function template to instantiate.
  - The compiler examines the types of the arguments that were specified using a template parameter.
  - It automatically instantiates a version of the function with those types or values bound to the template parameters.

Template Argument Deduction II

- Multiple type parameter arguments must match exactly
- Limited conversion
  - const conversion
  - array or function to pointer conversion
- Normal conversions apply to nontemplate arguments
- When the address of a function-template instantiation is taken, the context must be such that it allows a unique type or value to be determined for each template parameter

Template Compilation models I

- **inclusion compilation model**: Mechanism used by compilers to find template definitions that relies on template definitions being included in each file that uses the template.
  - Typically, template definitions are stored in a header, and that header must be included in any file that uses the template.

Template Compilation models II

- **separate compilation model**: Mechanism used by compilers to find template definitions that allows template definitions and declarations to be stored in independent files.
  - Template declarations are placed in a header, and the definition appears only once in the program, typically in a source file.
  - The compiler implements whatever programming environment support is necessary to find that source file and instantiate the versions of the template used by the program.
- **export keyword**: Keyword used to indicate that the compiler must remember the location of the associated template definition.
  - Used by compilers that support the separate-compilation model of template instantiation.

Trailing return types (C++11)

- Problem: you do not know the exact type a function should return
- Solution: use trailing return and declare the return type after the parameter list is seen
- Example
  - template<typename It> auto fcn(It beg, It end) -> decltype(*beg) { return *beg;}

Forwarding (C++11)

- Some functions need to forward one or more of their arguments with their types unchanged to another function
- In such cases, we want to preserve also const or if it is an lvalue or rvalue
- Solution: pass the argument by std::forward
  - std::forward<Type>(arg)

Variadic Templates (C++11)

- A **variadic template** is a template function or class that can take a varying number of parameters
- Example
  - template<typename T, typename… Args> void f(const T& t,const Args& …rest);

Member templates

- **member template**: A member of a class or class template that is a function template.
  - A member template may not be virtual.

Function Template Specialization

- **template specialization**: Redefinition of a class template or a member of a class template in which the template parameters are specified.
  - A template specialization may not appear until after the class that it specializes has been defined.
  - A template specialization must appear before any use of the template for the specialized arguments is used.
- Example
  - template<> char* max<char*>(char* s1, char* s2) {}

Class Template Specialization

- When a member is defined outside the class specialization, it is not preceded by the tokens template<>
- Example
  - template<typename T> MyClass { void push(T ch); } //template
  - template<> MyClass<char*> { void push(char* ch); } // specialization
  - void MyClass<char*>::push(char* ch) {} // member definition
  - template<> void MyClass<char*>::push(char* ch) {} // specializes member function but not class MyClass

Class Template Partial Specialization

- **partial specialization**: A version of a class template in which some but not all of the template parameters are specified.
- Example
  - template<typename T1, typename T2> MyClass { }
  - template<typename T1, int> MyClass<T1,int> { }

Overloading and Function Templates I

- Steps to resolve a call to an overloaded function
  1. Build the set of candidate functions for this name, including
     - Any ordinary function with the same name
     - Any function-template instantiation for which template argument deduction finds template arguments that match the function arguments used in the call
  2. Determine which, if any, of the ordinary functions are viable. Each template instance in the candidate set is viable, because template argument deduction ensures that the function could be called

Overloading and Function Templates II

- Steps to resolve a call to an overloaded function
  - Rank the viable functions by the kinds of conversions, if any, required to make the call, remembering that the conversions allowed to call an instance of a template function are limited
    - If only one function is selected, call this function
    - If the call is ambiguous, remove any function template instances from the set of viable functions
  - Rerank the viable functions excluding the function template instantiations
    - If only one function is selected, call this function
    - Otherwise, the call is ambiguous