

Templates

Templates

- ▶ A template is a generic declaration with compile-time formal arguments of these kinds:

- any type

```
template< typename T>           // also template< class T>
```

- [C++20] the type may be constrained using *Concepts*
- a list of any types (in a *variadic template*)

```
template< typename ... TL>
```

- integral number (the actual argument must be a compile-time constant)

```
template< std::size_t N>
```

- a list of integral numbers (in a *variadic template*)

```
template< int ... NL>
```

- another class template (with the given template argument list)

```
template< template< typename, std::size_t> class T>
```

- [C++17] the actual argument may also be a type template
- a pointer (the actual argument must be an address of a static variable/function)
 - almost never used, functors work better

```
template< const char * p, inf (*fp)(int,int)>
```

- ▶ A function/lambda with **auto** arguments is technically also a template

```
void f(auto && x);           template< typename T> void f(T && x);
```

Templates

▶ Template

- a generic piece of code
- parameterized by types, class/type templates, and integer constants

▶ Class templates

- Global classes
- Classes nested in other classes, including class templates

```
template< typename T, std::size_t N>  
class array { /*...*/ };
```

▶ Function templates

- Global functions
- Member functions, including constructors

```
template< typename T>  
inline T max( T x, T y) { /*...*/ }
```

▶ Type templates [C++11]

```
template< typename T>  
using array3 = std::array< T, 3>;
```

▶ Variable templates [C++14]

- Global variables and static data members
 - **inline** needed when declared in .hpp

```
template< typename T>  
inline factory_class<T> factory;
```

- Mostly used as global “constants” acting as compile-time functions on types

```
template< typename T>  
inline constexpr std::size_t my_sizeof = sizeof(T);    // usage: my_sizeof<double>
```

▶ Template instantiation

- Using the template with particular type and constant parameters
- Class, type, and variable templates: parameters specified explicitly

```
std::array< int, 10> x;
```

- [C++17] Template argument deduction: Class template parameters may be deduced from initialization

```
std::pair p = { "pi", 3.14 };
```

- Function templates: parameters specified explicitly or implicitly
 - Implicitly - derived by compiler from the types of value arguments

```
int a, b, c;
```

```
a = max( b, c);    // calls max< int>
```

- Explicitly

```
a = max< double>( b, 3.14);
```

- Mixed: Some (initial) arguments explicitly, the rest implicitly

```
std::array< int, 5> v;
```

```
x = std::get< 3>( v);    // calls std::get< 3, std::array< int, 5>>
```

Templates and compilation

- ▶ Compilers [may] check template code when defined
 - ▶ Without the knowledge of template arguments
 - ▶ Syntactic hints from the author may be required
- ▶ Compilers generate code only when templates are instantiated
- ▶ Different instantiations do not share code
 - ▶ It sometimes causes unwanted code size explosion
- ▶ There is no run-time support required for templates
 - ▶ Code generated for templates is identical to non-templated equivalents
- ▶ There is **no performance penalty** for generic code
 - ▶ Except that generic programming may encourage programmers to be lazy

▶ Implicit template instantiation

- ▶ When a class template specialization is referenced in context that requires a complete object type, or...
- ▶ ... when a function template specialization is referenced in context that requires a function definition to exist...
- ▶ ... the template is **instantiated** (the code for it is actually compiled)
 - unless the template was already explicitly specialized or explicitly instantiated
 - at link time, identical instantiations generated by different translation units are merged

▶ Instantiation of template member functions

- ▶ Instantiation of a class template doesn't instantiate any of its member functions unless they are also used
- ▶ **The definition of a template must be visible at the point of implicit instantiation**

- ▶ The definition of a template must be visible at its instantiation
 - ▶ Classes and types
 - The visibility of definition is required also in the non-template case
 - Most class and type definitions must reside in header files
 - ▶ Function templates
 - (including non-template member functions of class templates etc.)
 - The template visibility rule is equivalent to rules for *inline* functions
 - Non-inline function templates are almost unusable
 - Most **function template definitions must reside in header files** (and be **inline**)
 - If a declaration only is visible, the compiler will not complain - it will generate a call but not the code of the function called – the linker will complain
 - In rare cases, the visibility rule may be silenced by ***explicit instantiation***:
 - Applicable only if all required argument combinations may be enumerated

```
template int max<int>(int, int);    // forces instantiation of max<int>
```

```
template double max(double, double); // forces instantiation of max<double>
```

```
template class X<int>;           // forces instantiation of all member functions of X<int>
```

- ▶ Multiple templates with the same name

- ▶ Class templates:

- one "master" template

```
template< typename T> class vector { /*...*/};
```

- any number of specializations which override the master template
 - partial specialization

```
template< typename T, std::size_t n> class unique_ptr< T[n]> { /*...*/};
```

- explicit specialization

```
template<> class vector< bool> { /*...*/};
```

- ▶ Function templates:

- any number of templates with the same name
 - shared with non-templated functions

- ▶ Type and variable templates, concepts

- only one definition (which may refer to different specializations of a class template)

Writing templates

▶ Compiler needs hints from the programmer

▶ **Dependent names** have unknown meaning/contents

```
template< typename T> class X
```

```
{
```

- type names must be explicitly designated

```
using U = typename T::B;
```

```
typename U::D p; // U is also a dependent name
```

```
using Q = typename Y<T>::C;
```

```
void f() { T::D(); } // T::D is not a type
```

- explicit template instantiations must be explicitly designated

```
bool g() { return 0 < T::template h<int>(); }
```

```
int j() { return p.template k<3>(); } // type of p is a dependent name
```

```
}
```

- members inherited from dependent classes must be explicitly designated

```
template< typename T> class X : public T
```

```
{
```

```
const int K = T::B + 1; // B is not directly visible although inherited
```

```
void f() { return this->a; } // a is not directly visible
```

```
}
```