

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>
```

- 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 also a template

Templates

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

```
template< typename T, std::size_t N>  
class array { /*...*/ };           // usage: array<double,3>
```

- Function templates
 - Global functions
 - Member functions, including constructors

```
template< typename T>  
inline T max( T x, T y) { /*...*/ }    // usage: max(p, q) or max<int>(p, q)
```

- Type alias templates [C++11]

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

- Static variable templates [C++14]

- 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 alias, variable templates: parameters specified explicitly

```
using my_type = std::array< int, 10>;  
using signed_size_t = std::make_signed_t< std::size_t>;    // std::ptrdiff_t  
constexpr bool char_is_signed = std::is_signed_v<char>;
```

- [C++17] Class template argument deduction (matching initialization to constructors)

```
std::pair x(1, 3.14);    // std::pair<int,double>
```

- 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::tuple< int, double, int, char, unsigned> v;  
x = get< 3>( v);  
// calls std::get< 3, std::tuple< int, double, int, char, unsigned>>
```

- Note: Argument-dependent lookup uses both the type arguments in <> and the types of arguments in ()

- 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
 - resolved via "argument-dependent-lookup" and "overload resolution"
 - no explicit/partial specialization possible

- Instead, the **tag-class** trick is often used:

```
std::sort(std::execution::par, k.begin(), k.end());
```

- std::execution::par is an (empty) global variable of type std::execution::parallel_policy
 - the type enforces the use of a different (parallel) implementation of std::sort

- [C++20] concepts

- a condition that enables a specific template or specialization

```
template< typename U> requires is_array_v<U> class unique_ptr< U> { /*...*/};
```

- Compiler needs hints from the programmer
 - to understand the syntax of generic code
 - **Dependent names** have unknown meaning/contents
 - i.e. names that depend on template arguments

```
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>(); }
```

- 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; }
```

```
}
```

Passing template function arguments by value/reference

- How the arguments are passed in a call to a generic function?

```
std::string a1; std::string & a2 = a1; const std::string & a3 = a1;  
f(a1); f(a2); f(a3); f(std::move(a1));
```

- It depends on the declaration of the function
 - Presence of reference type in the declaration of a2/a3 does NOT matter

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

- In this case, x is always passed by value

- If you want pass by reference, always use a “forwarding reference”

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

- “Reference collapsing rules” ensure adaptation to both lvalues and rvalues

- Rationale: plain references cannot adapt to some actual arguments:

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

- In these cases, the function cannot be called with an rvalue argument:

```
f(a1+”.txt”); f(std::move(a1));
```

```
std::vector<bool> k(100); f(k[0]); // k[0] is an object simulating bool&
```

Forwarding (universal) references

- Forwarding references may appear
 - as function arguments

```
template< typename T>
void f( T && x)
{
    // ...
}
```

- as auto variables

```
auto && x = cont.at( i);
```

- Beware, not every T && is a forwarding reference
 - It requires the ability of the compiler to select T according to the actual argument
- The use of reference collapsing tricks is (by definition) limited to T &&
 - The compiler does not try all possible T's that could allow the argument to match
 - Instead, the language defines exact rules for determining T
- Reference collapsing rules:
 - L-value reference wins over R-value reference
 - X && && -> X &&
 - X & && -> X &
 - X && & -> X &
 - X & & -> X &

Forwarding (universal) references

- In this example, T && is not a forwarding reference

```
template< typename T>
class C {
    void f( T && x) {
        // ...
    }
};

C<X> o; X lv;
o.f( lv); // error: cannot bind an rvalue reference to an lvalue
```

- The correct implementation

```
template< typename T>
class C {
    template< typename T2>
    void f( T2 && x) {
        // ...
    }
};
```

Perfect forwarding

- Problem: Forwarding a rvalue/lvalue reference to another function

```
template< typename T> void f( T && p)
{
    g( p);
}
```

```
X lv;
f( lv);
```

- If an lvalue is passed: $T = X \&$ and p is of type $X \&$
 - p appears as **lvalue** of type X in the call to g

```
f( std::move( lv));
```

- If an rvalue is passed: $T = X$ and p is of type $X \&\&$
 - p appears as **lvalue** of type X in the call to g
 - Inefficient – move semantics lost

Perfect forwarding

- Perfect forwarding

```
template< typename T> void f( T && p)
{
    g( std::forward< T>( p));
}
```

- `std::forward< T>` is simply a cast to `T &&`

```
X lv;
f( lv);
```

- `T = X &`
 - `std::forward< T>` returns `X &` due to reference collapsing
 - The argument to `g` is an `lvalue`

```
f( std::move( lv));
```

- `T = X`
 - `std::forward< T>` returns `X &&`
 - The argument to `g` is an `rvalue`
 - `std::forward< T>` acts as `std::move` in this case

Variadic templates

- Motivation
 - Allow forwarding of variable number of arguments
 - e.g. in *emplace* functions

```
template< typename ... TList>
void f( TList && ... plist)
{
    g( std::forward< TList>( plist) ...);
}
```

- Allow many dirty tricks

```
template< typename ... TList>
void print( TList && ... plist)
{
    (std::cout << ... << plist) << std::endl; // [C++17] fold expression
}
```

Variadic templates

- Template prefix
 - Allows variable number of type arguments

```
template< typename ... TList>  
class C { /* ... */ };
```

- `typename ...` declares named ***template parameter pack***

- may be combined with regular type/constant arguments

```
template< typename T1, int c2, typename ... TList>  
class D { /* ... */ };
```

- also in partial template specializations

```
template< typename T1, typename ... TList>  
class C< T1, TList ...> { /* ... */ };
```

Variadic templates

```
template< typename ... TList>
```

- template parameter pack - a list of types
- may be referenced inside the template:
 - always using the suffix ...
- as type arguments to another template:

```
X< TList ...>
```

```
Y< int, TList ..., double>
```

- in argument list of a function declaration:

```
void f( TList ... plist);
```

```
double g( int a, double c, TList ... b);
```

- this creates a named function parameter pack **b**

- in several less frequent cases, including

- base-class list:

```
class E : public TList ...
```

- number of elements in the parameter pack:

```
sizeof...(TList)
```

Variadic templates

```
template< typename ... TList>
void f( TList ... plist);
```

- named ***function parameter pack***
- may be referenced inside the function:
 - always using the suffix ...

- as parameters in a function call or object creation:

```
g( plist ...);
new T( a, plist ..., 7);
T v( b, plist ..., 8);
```

- constructor initialization section (when variadic base-class list is used)

```
E( TList ... plist)
: TList( plist) ...
{
}
```

- fold expressions (will be described later)

Variadic templates

- template/function parameter packs are passed as unpacked
 - packs in actual arguments are not required to match packs in formal arguments

```
template< typename X, typename ... YL>
    void f( X x, YL ... y1);
template< typename ... QL>
    void g( QL ... q1)
{
    f( q1 ..., 729); // x=q1[0]; y1[0..n-2]=q1[0..n-1]; y1[n-1]=729
}
```

- frequently used trick in recursive templates

```
template< typename ... TList> // master template declared but not defined
class A;

template< typename T0, typename ... TRest> // partial specialization
class A : public A< TRest...> {
    T0 data0_;
    operator T0&() { return data0_; } // conversion operator
};

template<> // explicit specialization to stop recursion
class A {};

A<int,double,char> k; (double&)k = 3.14;
```

Variadic templates

```
template< typename ... TList>
void f( TList ... plist);
```

- parameter packs may be wrapped into a type construction/expression
 - the suffix ... works as compile-time "for_each"
 - parameter pack name denotes the place where every member will be placed
 - more than one pack name may be used inside the same ... (same length required)
- the result is
 - a list of types

```
template<typename ... TList>
class A : public B<TList> ...    // different from public B<TList ...>
{}
```

```
template<typename ... TList>
void f(TList && ... plist)
{
    • a list of expressions (in a function call or object initialization)
    h( std::forward< TList>( plist) ...);    // two pack names in one ...
}
```

- iterating through variadic function arguments

- C++14 often required recursion

```
auto sum() { return 0; }
template<typename T1, typename ... T>
auto sum(T1 s, T ... ts) { return s + sum(ts ...); }
```

- [C++17] **fold expression** for any binary operator **op**

```
template<typename ... T> void f(T ... P) {
    • syntax:
    ( P op ... )           // P1 op (P2 op ... (Pn-1 op Pn))
    ( ... op P )           // ((P1 op P2) op ... Pn-1) op Pn
    ( P op ... op init )   // P1 op (P2 op ... (Pn op init))
    ( init op ... op P )   // ((init op P1) op ... Pn-1) op Pn
}
```

- examples:

```
template<typename ... T>
auto sum1(T ... s){
    return (... + s);           // undefined for empty s
}
template<typename ... T>
auto sum2(T ... s){
    return (0 + ... + s);       // zero for empty s
}
template<typename ... T> void print(T && ... args) {
    (cout << ... << args) << '\n';
}
```