

EDAF50 – C++ Programming

10. Templates and the standard library. <chrono>.

Sven Gestegård Robertz
Computer Science, LTH

2020



Outline

1 The standard library

- Time representation
- Algorithms
- `std::tie`
- Meta-programming

2 Templates

- Variadic templates
- Template metaprogramming
- Specialization

3 `union`

What is a value

The semantics of a value often include

- ▶ a quantity
- ▶ a number
- ▶ a unit

E.g `int` `length` = 2;

- ▶ two meters?
- ▶ two millimeters?

*Including quantity and unit in the **type** helps avoid mistakes.*

Time representation

- ▶ A “time value” can be either
 - ▶ A duration – a time interval
 - ▶ A point in time
 - ▶ relative to a particular *clock*
- ▶ Different units
 - ▶ seconds
 - ▶ milliseconds
 - ▶ nanoseconds
 - ▶ *manual conversion error prone*
- ▶ Different semantics
 - ▶ $\text{duration} + \text{duration} = \text{duration}$
 - ▶ $\text{duration} - \text{duration} = \text{duration}$
 - ▶ $\text{time_point} + \text{duration} = \text{time_point}$
 - ▶ $\text{time_point} - \text{duration} = \text{time_point}$
 - ▶ $\text{time_point} - \text{time_point} = \text{duration}$
 - ▶ $\text{time_point} + \text{time_point} = \text{error}$

Time representation

<chrono>

- ▶ Uses the type system to denote
 - ▶ if a value is a duration or a point in time
 - ▶ the unit used (seconds, milliseconds, etc.)
 - ▶ which clock a point in time is relative to
 - ▶ `system_clock` – wall clock time
 - ▶ `steady_clock` – stopwatch
- ▶ Uses compile-time computations for
 - ▶ conversions between units
 - ▶ implicit conversions when safe
 - ▶ explicit conversions when losing information
 - ▶ E.g. `duration_cast<seconds>(milliseconds)`

Time representation

<chrono>

A duration is

- ▶ an *integer value* and
- ▶ a *ratio* (the number of seconds between two values).

```
std::chrono::nanoseconds duration<signed integer (>= 64 bits),  
                                std::nano>  
std::chrono::microseconds duration<signed integer (>= 55 bits),  
                                std::micro>  
std::chrono::milliseconds duration<signed integer (>= 45 bits),  
                                std::milli>  
  
std::chrono::seconds duration<signed integer (>= 35 bits)>  
  
std::chrono::minutes duration<signed integer (>= 29 bits),  
                             std::ratio<60>>  
std::chrono::hours    duration<signed integer (>= 23 bits),  
                             std::ratio<3600>>
```

std::ratio provides compile-time rational arithmetic

Demo

The standard algorithms take function objects *by value*:

```
template< class InputIt, class UnaryFunction >  
UnaryFunction for_each(InputIt first, InputIt last, UnaryFunction f);
```

```
template< class InputIt, class UnaryPredicate >  
InputIt find_if(InputIt first, InputIt last, UnaryPredicate p);
```

How to handle *stateful function objects*?

Demo

<functional> defines helper functions `std::ref` and `std::cref`:

```
template< class T >
std::reference_wrapper<T> ref(T& t) noexcept;
```

```
template< class T >
std::reference_wrapper<const T> cref( const T& t ) noexcept;
```

that return a CopyConstructible and CopyAssignable wrapper around a reference:

```
template< class T >
class reference_wrapper {
public:
    reference_wrapper& operator=(const reference_wrapper&) noexcept;
    operator T&() const noexcept;
    T& get() const noexcept;

    template< class... ArgTypes >
    typename std::result_of<T&(ArgTypes&&...)>::type
    operator() ( ArgTypes&&... args ) const;
};
```

std::tie

Example: implementation sketch

tie for a pair<int, string>

```
std::pair<int&, string&> mytie(int& x, string& y)
{
    return std::pair<int&, string&>(x,y);    // or return {x,y};
}
```

- ▶ returns a *temporary* pair of *lvalue references*
- ▶ the assignment operator of pair assigns each member
- ▶ the references are *aliases for the variables* passed as arguments
- ▶ assigning to the references is the same as assigning to the variables

```
int i;
string s;

mytie(i,s) = t;
```

possible implementation

```
template <typename... Args>
std::tuple<Args&...> tie(Args&... args)
{
    return std::tuple<Args&...>(args...);
}
```

Iterator traits

Exemple: find

```
template <class InIt, class T>  
InIt find (InIt first, InIt last, const T& val);
```

Alternative: the compiler knows the actual value type.

With `std::iterator_traits` from `<Iterator>`

```
template <class InIt>  
InIt find (InIt first, InIt last,  
          const typename iterator_traits<InIt>::value_type& val);
```

NB! This is more restrictive on the value type

type traits

`<type_traits>` contains metafunctions for working with types. E.g.:

Type categories

`is_void` `is_scalar` `is_array` `is_class` `is_function`

Type properties

`is_const` `is_empty` `is_signed` `is_reference` `is_pointer`

Type relations

`is_same` `is_convertible` `is_base_of`

Modifiers

`add_const` `remove_const` `remove_reference` `add_lvalue_reference`
`make_signed` `make_unsigned` `remove_extent`

Variadic templates

```
void println() { base case: no arguments
    cout << endl;
}

template <typename T, typename... Tail>
void println(const T& head, const Tail&... tail)
{
    cout << head << " ";    Print the first element
    println(tail...);        recursion: print the rest
}

void test_variadic()
{
    string a{"Hello"};
    int b{10};
    double c{17.42};
    long d{100};

    println(a,b,c,d);
}
```

Template metaprogramming

- ▶ Write code that is executed *by the compiler, at compile-time*
- ▶ Common in the standard library
 - ▶ As optimization: move computations from run-time to compile-time
 - ▶ As utilities: e.g., `type_traits`, `iterator_traits`
- ▶ Metafunction: a class template containing the result
- ▶ Standard library conventions:
 - ▶ Type results: type member named `type`
 - ▶ Value results: value member named `value`

Template metaprogramming

Example of compile-time computation

```
template <int N>
struct factorial{
    static constexpr int value = N * factorial<N-1>::value;
};

template <>
struct factorial<0>{
    static constexpr int value = 1;
};

void example()
{
    display_int<factorial<5>::value>();
}
```

Result of the *meta-function call* as a compiler error:

```
In function 'void example()':
error: invalid use of incomplete type 'struct display_int<120>'
    display_int<factorial<5>::value>();
```

Template metaprogramming

Example of templates for getting values as compiler errors

- ▶ Trick: use a template that doesn't compile to get information about the template parameters through a compiler error.
- ▶ Can be useful for debugging templates.
- ▶ To get an int:

```
template <int N>  
struct display_int;    // declaration but no definition
```

- ▶ To get a type:

```
template <typename T>  
struct display_type;
```

Class Templates

Definition of function members

```
template <typename T>
void Vector<T>::print() const
{
    for(size_t i = 0; i != sz; ++i)
        cout << p[i] << " ";
    cout << endl;
}
```

- ▶ Function members in a class template are function templates
- ▶ print() works for all types with an **operator<<**
- ▶ *"Duck typing":*
if it walks like a duck and quacks like a duck, it is a duck

Class Templates

Definition of member functions

```
template <typename T>
void Vector<T>::print() const
{
    for(size_t i = 0; i != sz; ++i)
        cout << p[i] << " ";
    cout << endl;
}
```

- Works for all types with **operator<<**
- but not for elements of type

```
struct Foo{
    int x;

    Foo(int d=0) :x{d}{}
};
```

Template specialization for the type Foo:

```
template<> full specialization: no template arguments
void Vector<Foo>::print() const
{
    for(size_t i = 0; i != sz; ++i)
        cout << "Foo("<<p[i].x << " " ";
    cout << endl;
}
```

Template specialization

- ▶ Class Templates can be specialized
 - ▶ fully
 - ▶ partially
- ▶ Function templates can be specialized
 - ▶ fully
 - ▶ *but overloading is always preferable*

Templates, comments

- ▶ Templates have parameters
 - ▶ type parameters: declared with **class** or **typename**
 - ▶ value parameters: declared as usual, e.g., **int** N
- ▶ The compiler needs the template definition to instantiate
⇒ it must be in the *header file* (if used by others)
- ▶ Overloading:
 - ▶ Functions can be overloaded ⇒ function templates can be overloaded
 - ▶ Classes cannot be overloaded ⇒ class templates cannot be overloaded
- ▶ Template specialization:
 - ▶ Class templates can be specialized *partially* or *fully*
 - ▶ Function templates can only be *fully* specialized, *but*
 - ▶ Specializations are not overloaded
 - ▶ Often better/clearer to overload with a normal function (not a template) than to specialize

The size of a normal **struct** (class) is *the sum* of its members

```
struct DataS {  
    int nr;  
    double v;  
    char txt[6];  
};
```

All members in a **struct** are laid out after each other in memory.

The size of a **union** is equal to *the size of the largest member*

```
union DataU {  
    int nr;  
    double v;  
    char txt[6];  
};
```

All members in a **union** have *the same address*: only one member can be used at any time.

Example use of DataU

```
union DataU {  
    int nr;  
    double v;  
    char txt[6];  
};  
  
DataU a;  
  
a.nr = 57;  
cout << a.nr << endl;      57  
  
a.v = 12.345;  
cout << a.v << endl;      12.345  
  
strncpy(a.txt, "Tjo",6);  
cout << a.txt << endl;    Tjo
```

*The programmer is responsible for only using
the “right” member*

union

Example of wrong use

```
using std::cout;
using std::endl;

union Foo{
    int i;
    float f;
    double d;
    char c[10];
};

int main()
{
    Foo f;

    f.i = 12;
    cout << f.i << ", " << f.f << ", " << f.d << ", " << f.c << endl;

    strncpy(f.c, "Hej, du", 10);
    cout << f.i << ", " << f.f << ", " << f.d << ", " << f.c << endl;
}

12, 1.68156e-44, 5.92879e-323, ^L
745170248, 3.33096e-12, 1.90387e-306, Hej, du
```

union

encapsulate a union in a class to reduce the risk of mistakes

```
struct Bar{
    enum {undef, i, f, d, c} kind;
    Foo u;
};

void print(Bar b) {
    switch(b.kind){
    case Bar::i:
        cout << b.u.i << endl;
        break;
    case Bar::f:
        cout << b.u.f << endl;
        break;
    case Bar::d:
        cout << b.u.d << endl;
        break;
    case Bar::c:
        cout << b.u.c << endl;
        break;
    default:
        cout << "???" << endl;
        break;
    }
}
```

```
void test_kind()
{
    Bar b{};

    b.kind = Bar::i;
    b.u.i = 17;

    print(b);

    Bar b2{};
    print(b2);
}

17
???
```

union

anonymous union – removes one level

An alternative to the previous example:

```
struct FooS{  
    enum {undef, k_i, k_f, k_d, k_c} kind;  
    union{  
        int i;  
        float f;  
        double d;  
        char c[10];  
    };  
};
```

```
FooS test;
```

```
test.kind = FooS::k_c;  
strncpy(test.c, "Testing", 10);  
if(test.kind == FooS::k_c)  
    cout << test.c << endl;
```

```
Testing
```

*Exposing the tag to the users
is brittle.*

Tagged union

A class with anonymous union and access functions

```
struct FooS{
    enum {undef, k_i, k_f, k_d, k_c} kind;
    union{
        int i;
        float f;
        double d;
        char c[10];
    };
    FooS() : kind{undef} {}
    FooS(int ii) : kind{k_i}, i{ii} {}
    FooS(float fi) : kind{k_f}, f{fi} {}
    FooS(double di) : kind{k_d}, d{di} {}
    FooS(const char* ci) : kind{k_c} {strncpy(c, ci, 10);}
    int get_i() {assert(kind==k_i); return i;}
    float get_f() {assert(kind==k_f); return f;}
    double get_d() {assert(kind==k_d); return d;}
    char* get_c() {assert(kind==k_c); return c;}
    FooS& operator=(int ii) {kind=k_i; i = ii; return *this;}
    FooS& operator=(float fi) {kind=k_f; f = fi; return *this;}
    FooS& operator=(double di) {kind=k_d; d = di; return *this;}
    FooS& operator=(const char* ci){kind=k_c; strncpy(c, ci, 10);
        return *this;}
};
```

Suggested reading

References to sections in Lippman

Overloading and templates 16.4

Variadic templates 16.4

Template specialization 16.5

Union 19.6