Templating

Victor Eijkhout, Susan Lindsey

Fall 2024

last formatted: October 29, 2024

1. What's the problem?

Do you have multiple vector classes?

```
class vector_of_int {
  public:
  int size();
  int at(int i);
};
```

```
class vector_of_float {
  public:
  int size();
  float at(int i);
};
```

You have already seen the solution: `vector<int>`

2. Templated type name

If you have multiple functions or classes that do 'the same' for multiple types, you want the type name to be a variable, a template parameter. Syntax:

```
template <typename yourtypevariable>
// ... stuff with yourtypevariable ...

// usually:
template <typename T>
```

3. Example: function

Definition:

```
1 // template/func.cpp
2 template <typename Real>
3 void sqrt_diff( Real x ) {
4   cout << std::sqrt(x)-1.772 << '\n';
5 };
```

We use this with a templated function:

Code:

```
1 // template/func.cpp
2 sqrt_diff<float>( 3.14f );
3 sqrt_diff<double>( 3.14 );
```

Output:

```
4.48513e-06
4.51467e-06
```

4. Type deduction

The compiler can deduce the template:

```
1 // template/func.cpp
2 sqrt_diff( 3.14f );
3 sqrt_diff( 3.14 );
```

Exercise 1

Machine precision, or 'machine epsilon', is sometimes defined as the smallest number ϵ so that $1 + \epsilon > 1$ in computer arithmetic.

Write a templated function *epsilon* so that the following code prints out the values of the machine precision for the `float` and `double` type respectively:

Code:

```
1 // template/eps.cpp
2 float float_eps =
3   epsilon<float>();
4 cout << "Epsilon float: "
5      << setw(10) <<
       setprecision(4)
6      << float_eps << '\n';
7
8 double double_eps =
9   epsilon<double>();
10 cout << "Epsilon double: "
11     << setw(10) <<
       setprecision(4)
12     << double_eps << '\n';
```

Output:

```
Epsilon float:
    ↪1.1921e-07
Epsilon double:
    ↪2.2204e-16
```

5. Templated point class

Coordinates can be `float` or `double`:

```
1 // geom/pointtemplate.cpp
2 template<typename T>
3 class Point {
4 private:
5   T x,y;
6 public:
7   Point(T ux,T uy) { x = ux; y = uy; };
```

Coordinates can also be other things, but that doesn't always make sense.

Exercise 2

Take your `Point` class from a previous exercise and templatize the class definition.

Write the `distance` function for the templated class Write a main program that tests this.

6. Templated vector

The templated vector class looks roughly like:

```
template<typename T>
class vector {
private:
  T *vectordata; // internal data
public:
  T at(int i) { return vectordata[i]  };
  int size() { /* return size of data */ };
  // much more
}
```

7. Class that stores one element

Intended behavior:

Code:

```
1 // template/example1.cpp
2 Store<int> i5(5);
3 cout << i5.value() << '\n';
```

Output:

```
5
```

8. Class definition

Template parameter is used for private data, return type, etc.

```
1 // template/example1.cpp
2 template< typename T >
3 class Store {
4 private:
5   T stored;
6 public:
7   Store(T v) : stored(v) {};
8   T value() { return stored;};
```

9. Templated class as return

Given:

```
1 // template/example1.cpp
2 Store<float> f314(3.14);
```

Methods that return a templated object:

Code:

```
1 // template/example1.cpp
2 Store<float> also314 =
        f314.copy();
3 cout << also314.value() << '\n';
4 Store<float> min314 =
        f314.negative();
5 cout << min314.value() << '\n';
```

Output:

```
3.14
-3.14
```

(easier to write with `auto`!)

10. Class name injection

Template parameter can often be left out in methods:

```
1 // template/example1.cpp
2 // spell out the template parameter
3 Store<T> copy() const { return Store<T>(stored); };
4 // using CTAD:
5 Store negative() const { return Store(-stored); };
```

'Class Template Argument Deduction'

Intermezzo: complex numbers

11. Complex

Code:

```
1 // complex/basic.cpp
2 #include <complex>
3 using std::complex;
4     /* ... */
5   complex<double> d(1.,3.);
6   cout << d << '\n';
7   complex<float> f;
8   f.real(1.); f.imag(2.);
9   cout << f << '\n';
```

Output:

```
(1,3)
(1,2)
```

12. Operations and literals

Operations on complex scalars:

Code:

```
1 // complex/basic.cpp
2 using namespace
      std::complex_literals;
3 auto e = d*2.;
4 cout << e << '\n';
5 auto g = e + 2.5i + 3.; // note
      3dot
6 cout << g << '\n';
```

Output:

```
(2,6)
(5,8.5)
```

13. Complex functions

Functions on complex numbers:

```
std::complex<T> conj( const std::complex<T>& z );
std::complex<T> exp( const std::complex<T>& z );
```

Also `abs`, `norm`, `polar`

Exercise 3

Let $x = .5 + i$ and compute

$$x + \bar{x} - e^{2\pi i}$$

which should be one. Is it actually one? How close do you get in `float` and `double` complex?

Newton's method

Exercise 4

Rewrite your Newton program so that it works for complex numbers. Here is the main; you need to write the functions:

```
1 // newton/newton-complex.cpp
2 complex<double> z{.5,.5};
3 while ( true ) {
4   auto fz = f(z);
5   cout << "f( " << z << " ) = " << fz << '\n';
6   if (std::abs(fz)<1.e-10 ) break;
7   z = z - fz/fprime(z);
8 }
```

You may run into the problem that you can not operate immediately between a complex number and a `float` or `double`. Use `static_cast`;

```
static_cast< complex<double> >(2)
```

14. Templatized Newton, first attempt

You can templatize your Newton function and derivative:

```
1 // newton/newton-double.cpp
2 template<typename T>
3 T f(T x) { return x*x - 2; };
4 template<typename T>
5 T fprime(T x) { return 2 * x; };
```

and then write

```
1 // newton/newton-double.cpp
2 double x{1.};
3 while ( true ) {
4   auto fx = f<double>(x);
5   cout << "f( " << x << " ) = " << fx << '\n';
6   if (std::abs(fx)<1.e-10 ) break;
7   x = x - fx/fprime<double>(x);
8 }
```

Exercise 5

Update your Newton program with templates. If you have it working for `double`, try using *complex*<`double`>. Does it work?

Exercise 6

Use your complex Newton method to compute $\sqrt{2}$. Does it work?

How about $\sqrt{-2}$?

Exercise 7

Write a Newton method where the objective function is itself a template parameter, not just its arguments and return type. Hint: no changes to the main program are needed.

Then compute $\sqrt{2}$ as:

```
1 // newton/lambda-complex.cpp
2 cout << "sqrt -2 = " <<
3   newton_root<complex<double>>
4   ( // objective function
5     [] (complex<double> x) -> complex<double> {
6         return x*x + static_cast<complex<double>>(2); },
7     // derivative
8     [] (complex<double> x) -> complex<double> {
9         return x * static_cast<complex<double>>(2); },
10    // initial value
11    complex<double>{.1,.1}
12    )
13     << '\n';
```

Separate compilation

15. Templated class

```
1 // namespace/instantlib.h
2 template< typename T >
3 class instant {
4  public:
5   instant() = default;
6   void out();
7 };
```

16. Use

Assume that we know what the template parameter will be:

```
1 // namespace/instant.cpp
2 instant<char> ic;
3 ic.out();
4 instant<int> ii;
5 ii.out();
```

17. Instantiation

Lines added to implementation file:

```
1 // namespace/instantlib.cpp
2 template class instant<char>;
3 template class instant<int>;
```