Templating

Victor Eijkhout, Susan Lindsey

Fall 2024

last formatted: November 5, 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:

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

We use this with a templated function:

Code:

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

Output:

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

4. Type deduction

The compiler can deduce the template:

```
// template/func.cpp
sqrt_diff( 3.14f );
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:

```
// template/eps.cpp
float float_eps =
  epsilon<float>();
cout << "Epsilon float: "
     << setw(10) << setprecision(4)
     << float_eps << '\n';

double double_eps =
  epsilon<double>();
cout << "Epsilon double: "
     << setw(10) << setprecision(4)
     << double_eps << '\n';
```

Output:

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

5. Templated point class

Coordinates can be `float` or `double`:

```
// geom/pointtemplate.cpp
template<typename T>
class Point {
private:
  T x,y;
public:
  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:

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

Output:

```
5
```

8. Class definition

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

```
// template/example1.cpp
template< typename T >
class Store {
private:
  T stored;
public:
  Store(T v) : stored(v) {};
  T value() { return stored;};
```

9. Templated class as return

Given:

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

Methods that return a templated object:

Code:

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

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

'Class Template Argument Deduction'

Separate compilation

11. Templated class

```
// namespace/instantlib.h
template< typename T >
class instant {
 public:
  instant() = default;
  void out();
};
```

12. Use

Assume that we know what the template parameter will be:

```
// namespace/instant.cpp
instant<char> ic;
ic.out();
instant<int> ii;
ii.out();
```

13. Instantiation

Lines added to implementation file:

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

Intermezzo: complex numbers

14. Complex

Code:

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

Output:

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

15. Operations and literals

Operations on complex scalars:

Code:

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

Output:

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

16. 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?

Templated 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:

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

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)
```

17. Templatized Newton, first attempt

You can templatize your Newton function and derivative:

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

and then write

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

18. Templatized newton root function

The final step is to have a templatized Newton function that you can instantiate with any suitable type:

```
auto x_float = newton_root<float>( 2.0f );
auto x_double = newton_root<double>( 2.0 );
auto x_complex = newton_root<complex<float>>( 2.0+2.i );
```

Exercise 5

Update your *newton_root* function with a template parameter. Test it by having a main program that computes some roots in `float`, `double`, and *complex*<`double`>.

Exercise 6

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:

```
// newton/lambda-complex.cpp
cout << "sqrt -2 = " <<
  newton_root<complex<double>>
  ( // objective function
    [] (complex<double> x) -> complex<double> {
        return x*x + static_cast<complex<double>>(2); },
    // derivative
    [] (complex<double> x) -> complex<double> {
        return x * static_cast<complex<double>>(2); },
    // initial value
    complex<double>{.1,.1}
    )
     << '\n';
```