Templating

Victor Eijkhout, Susan Lindsey

Fall 2023

`last formatted: February 6, 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 T>
void function( T x ) {
  cout << std::sqrt(x)-1.772 << '\n';
};
```

We use this with a templated function:

Code:

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

Output:

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

4. Type deduction

The compiler can deduce the type:

```
// template/func.cpp
function( 3.14f );
function( 3.14 );
```

5. 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
}
```

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_eps);
cout << "Epsilon float: "
     << setw(10) << setprecision(4)
     << float_eps << '\n';

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

Output:

```
Epsilon float:
    1.0000e-07
Epsilon double:
    1.0000e-15
```

Hint: you may need to cast scalars to the appropriate type

6. Class that stores one element

Code:

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

Output:

```
5
```

7. 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;};
```

8. 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
```

9. Class name injection

Template parameter can often be left out in methods:

```
// template/example1.cpp
Store<T> copy() { return Store<T>(stored); };
Store negative() { return Store(-stored); };
```

Intermezzo: complex numbers

10. 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)
```

11. Operations and literals

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)
```

Newton's method

Exercise 2

Rewrite your Newton program so that it works for complex numbers:

```
// 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`; see section **??**.

12. 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);
}
```

Exercise 3

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

Exercise 4

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

How about $\sqrt{-2}$?

Exercise 5

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>>
  ( [] (complex<double> x) -> complex<double> {
        return x*x + static_cast<complex<double>>(2); },
    [] (complex<double> x) -> complex<double> {
        return x * static_cast<complex<double>>(2); },
    complex<double>{.1,.1}
    )
     << '\n';
```

Separate compilation

13. Templated class

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

14. Use

Assume that we know what the template parameter will be:

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

15. Instantiation

Lines added to implementation file:

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