Templating

Victor Eijkhout, Susan Lindsey

Fall 2023

`last formatted: November 2, 2023`

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);
};
```

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 ...
```

3. Example: function

Definition:

```
template<typename T>
void function(T var) { cout << var << end; }
```

Usage:

```
int i; function(i);
double x; function(x);
```

and the code will behave as if you had defined `function` twice, once for `int` and once for `double`.

4. 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
```

5. Class that stores one element

Code:

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

Output:

```
5
```

6. 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;};
```

7. Templated class as return

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
```

8. Class name injection

Template parameter can often be left out in methods:

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

Intermezzo: complex numbers

9. Complex

Code:

```
// complex/basic.cpp
#include <complex>
using std::complex;
//codesnippet complexdef
    /* ... */

int main() {

  cout << "Def\n";
  //codesnippet complexdef
    /* ... */
  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)
```

10. Complex literals

```
using namespace std::complex_literals;
  std::complex<double> c = 1.0 + 1i;
```

Beware: `1+1i` does not compile:

11. Example usage

Code:

```
// complex/veccomplex.cpp
  vector< complex<double> >
    vec1(N, 1.+2.5i );
  auto vec2( vec1 );
    /* ... */
  for ( int i=0; i<vec1.size(); ++i ) {
    vec2[i] = vec1[i] * ( 1.+1.i );
  }
    /* ... */
  auto sum = accumulate
    ( vec2.begin(),vec2.end(),
      complex<double>(0.) );
  cout << "result: " << sum << '\n';
```

Output:

```
result:
      (-1.5e+06,3.5e+06)
```

Newton's method

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);
  }
```

Update your Newton program with templates. If you have it working for `double`, try using `complex<double>`. Does it work? Use your complex Newton method to compute $\sqrt{2}$. Does it work?

How about $\sqrt{-2}$? Can you templatize your Newton code that used lambda expressions? Your function header would now be:

```
// newton/lambda-complex.cpp
template<typename T>
T newton_root
   ( function< T(T) > f,
     function< T(T) > fprime,
     T init) {
```

You would for instance 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';
```

Templates and headers

13. Templated declaration

Declaration of a templated class:

```
// template/example2.cpp
template< typename T >
class Store {
private:
  T stored;
public:
  Store(T v);
  T value();
  Store copy();
  Store<T> negative();
};
```