Templates

Victor Eijkhout, Susan Lindsey

Fall 2025

`last formatted: November 14, 2025`

1. 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>
```

2. 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
```

3. Type deduction

The compiler can deduce the template argument:

```
// 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 `machine_eps` so that the following code prints out the values of the machine precision for the `float` and `double` type respectively:

Code:

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

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

Output:

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

4. Templated Point class

```
// geom/pointtemplate.cpp
template<typename T>
class Point {
private:
  T x,y;
public:
  Point(T ux,T uy) { x = ux; y = uy; };
```

Now you can have `Point<float>` or `Point<double>`.
(With the code so far, also `Point<string>`)

Exercise: add the `distance_to_origin` and `distance` functions and test them.

5. Templated vector

The templated vector class looks roughly like:

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

Vector consists of

- small block with size information and pointer to data; on the stack
- actual data, dynamically allocated on the heap

Worked out example

6. Class that stores one element

Intended behavior:

Code:

```
// template/example1.cpp
Store<int> i5(5);
cout << i5.stored_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 stored_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.stored_value() << '\
    n';
Store<float> min314 = f314.negative
    ();
cout << min314.stored_value() << '\n
    ';
```

Output:

```
3.14
-3.14
```

Exercise: can you write this with `auto`?

9. 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 (CTAD)

Separate compilation

10. Templated declaration

Declaration of a templated class:

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

11. Template method definitions

Each method needs the `template` line:

```
// template/example2.cpp
template< typename T >
Store<T>::Store(T v) : stored(v) {};

template< typename T >
T Store<T>::value() const { return stored;};

template< typename T >
Store<T>  Store<T>::copy() const { return Store<T>(stored); };

template< typename T >
Store<T> Store<T>::negative() const { return Store<T>(-stored); };
```

12. Header-only approach

Use of templates increases compilation time
this is the case for ‘header-only’ libraries.

This can be countered by using header/implementation files if you know in advanced with what types a template will be instantiated.

13. Templated class

Suppose a templated class will only be used with certain instantiations:

Class definition:

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

Only instantiations:

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

14. Instantiation

Lines added to implementation file:

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

Now you can use header/library split

Complex numbers

15. Complex numbers

```
#include <complex>

complex<float> f;
f.re = 1.; f.im = 2.;
complex<double> d(1.,3.);

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

conj(c); exp(c);
```

Complex Newton

Exercise 2

Rewrite your Newton code where the algorithm is in the main 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)
```

Templated functions

16. Templatized Newton, first attempt

You can templatize the objective function and the 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 in the main program:

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

Exercise 4

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

How about $\sqrt{-2}$? If it doesn't work, why, and can you fix this?

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>>
  ( // 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';
```