Computing at Scale

Lecture 8: Introduction to Templates

Jacob Merson

February 6, 2025

Logistics

Today's Agenda

- Function Templates
- Type Deduction and `auto`

Function Templates

Programming Principle of the Day: DRY

DRY (don’t repeat yourself) is a principle of software development aimed at reducing repetition of software patterns, replacing it with abstractions or using data normalization to avoid redundancy.

- Use (small) functions and classes to avoid repeating code.
- Use the minimum ”atomic” unit for functions.

Motivation

Write a function that adds two numbers that works with integers, floats and doubles.

- Now, write the functionality for `long` and `std::complex`.
- Did you repeat yourself? Or is this code DRY?
- Did you write a function for each type?
- What if we add a new type? How much work is this?

Template By Example

Without Templates

```
int add(int a, int b) {
  return a + b;
}
float add(float a, float b) {
  return a + b;
}
double add(double a, double b) {
  return a + b;
}

int main() {
  int x = add(1,2); // add with int
  float y = add(1.0f,2.0f); // add with float
  double y = add(1.0,2.0); // add with double
}
```

With Templates

```
template <typename T>
T add(T a, T b) {
  return a + b;
}
```

Function Templates

- Function templates are a way to write functions that can work with any type.
- The syntax is similar to a regular function, but with one or more *template parameters*.
- Each *template parameter* is a placeholder for a type.

Anatomy of A Template

- The keyword `template` followed by `<typename T>` (preferred) or `<class T>`.
- `T` is a placeholder for a type that is determined at compile time. Called a *template parameter*.
- The template parameter does not need to be named `T`, but it is a common convention.
- The function signature and body remain the same, but can use `T` as a type.

```
// one template parameter
template <typename T>
T multiply(T a, T b) {
  return a * b;
}
```

```
// three template parameters
template <typename T1, typename T2
    , typename R>
R multiply(T1 a, T2 b) {
  return a * b;
}
```

Calling Templates

Template types can be explicitly specified or deduced.

```
// one template parameter
template <typename T>
T multiply(T a, T b) {
  return a * b;
}
int main() {
  // types are placed in angle brackets
  int x = multiply<int>(1,2); // multiply<int>(1,2);
  float y = multiply(1.0f,2.0f) // multiply<float>(1.0f,2.0f);
  double y = multiply(1.0,2); // Error! Type mismatch
  double y = multiply<double>(1.0,2); // multiply<double>(1.0,2.0);
}
```

Two Phase Compilation

Phase 1: At definition time, the template is checked for correctness including template parameters.

- Syntax errors such as missing semicolons or braces.
- Unknown types or functions (that don't depend on the template parameter).

Phase 2: Template instantiation. Type deduction happens and the compiler generates code with specific types.

Compilation Sticky Points

- Compiler needs to see the template definition when it is instantiated (used).
- For now, this means we need to define all templates in header files.
- Themplates can bloat the size of your binary and drastically increase compile times if not careful.
- Explicit instantiation can help with this. (not covered in detail today)

Argument Deduction

- Template argument deduction is done when we call the template.
- Can represent all of the type or just part of it (e.g. `const T&`).
- Automatic type conversion limited during deduction.
  - When declared by reference, trivial conversions are not considered. I.e., types must match exactly.
  - When declared by value, trivial conversions that decay are considered. I.e., arrays decay to pointers, functions decay to function pointers, and `const`/`volatile` are ignored.

```
template <typename T>
T max(const T& a, const T& b) {
  return a > b ? a : b;
}
int main() {
  max(1,2); // T deduces to int (used as const int& in function)
}
```

Type Conversion Examples

```
template<typename T>
T max(T a, T b);
int i = 1;
const int c = 2;
int& ir = i;
int arr[4];
max(i, c); // T deduced as int
max(c, c); // T deduced as int
max(i, ir); // T deduced as int
max(&i, arr); // T deduced as int* (array decays to pointer)
max(4, 4.5) // Error, T could be int or double, ambiguous
std::string s = "hello";
max(s, "world"); // Error, T could be std::string or const char[6],
    ambiguous
```

Default Template Arguments

- You can provide a default template argument.
- This is useful when you have a common type that you want to use most of the time.
- Template parameters with defaults must come after those without defaults.
- Type deduction doesn't work with default function call arguments.

```
template <typename T>
void speak(T = "");
speak(1); // OK, T deduced to int
speak(); // Error, T cannot be deduced

template <typename T=std::string>
void speak(T = "");
speak(1); // OK, T deduced to int
speak(); // OK, T deduced to std::string
```

Deducing Return Types

```
template <typename T1, typename T2>
auto max(T1 a, T2 b) -> typename std::decay_t<decltype(true ? a:b)> {
  return a>b ? a : b;
}

template <typename T1, typename T2>
std::common_type_t<T1, T2> max(T1 a, T2 b) {
  return a>b ? a : b;
}

template <typename T1, typename T2, typename R=std::common_type_t<T1,
    T2>>
R max(T1 a, T2 b) {
  return a>b ? a : b;
}
```

Template Specialization / Overloading

Function templates can simply be overloaded like regular functions. The non-template version will typically be chosen over template version since it's more specific. Try the following code in https://cppinsights.io/ .

```
int max(int a, int b) {
  return a > b ? a : b;
}
template <typename T>
T max(T a, T b) {
  return a > b ? a : b;
}
```

```
int main() {
  max(7, 8); // calls non-template
       version
  max(7.0, 8.0); // calls max<double>
  max<>(7, 8); // calls max<int>
  max('a', 'b'); // calls max<char>
  max('a', 1.1); // calls non-template
       version (conversion not
       considered for template)
}
```

Hint

With overload template functions keep them consistent if you are passing by value or reference. Otherwise, you may get unexpected results.

R Value References

- R value references are a way to bind to temporary objects.
- They are used to enable move semantics.
- They are used to enable perfect forwarding.
- With concrete types R value references are written as `T&&`.

Reference Collapsing Rules

- Cannot directly define a reference to a reference.
- When combining types, reference can collapse.
- Const/volatile qualifiers from leftmost type are kept.

- T& + T& = T&
- T&& + T& = T&

```
using RI = int&;
int i = 1;
RI r = i;
R const& rr = r; // type is int&
```

- T& + T&& = T&
- T&& + T&& = T&&

```
using RCI = const int&;
RCI volatile&& r = 1; // r has
    type const int&
using RRI = int&&;
RRI const&& rr = 1; // rr has
    type int&&
```

Forwarding References

- With templates, **T&&** is a *forwarding reference*. Not an R value reference!
- They are used to enable perfect forwarding. Or, passing arguments to a function without losing their vaulue category (R vs L value).
- Forwarding references bind to both L and R value references.

```
template <typename T>
void forward(T&& p) {
  // T&& is a forwarding/universal reference
}
int main() {
  int i =0;
  const int cj = 1;
  forward(i); // argument is lvalue, T deduced as int&, p has type
      int&
  forward(cj); // argument is lvalue, T deduced as const int&, p has
      type const int&
  forward(1); // argument is rvalue, T deduced as int, p has type int
```

Perfect Forwarding Example

This technique is used for forwarding arguments to another function. Examples are `std::make_unique` and `emplace`. In practice use `std::forward` instead of `static_cast`.

```
class C{};
void g(C&);
void g(const C&);
void g(C&&); // rvalue reference

template <typename T>
void forward(T&& x) {
  g(static_cast<T&&>(x)); //
      perfect forwarding
}
```

```
int main() {
  C v;
  const C c;
  forward(v); // calls g(C&);
  forward(c); // calls g(const C
      &);
  forward(C{}); // calls g(C&&);
  forward(std::move(v)); //
      calls g(C&&);
}
```

Constraints

- By default templates match *any* type.
- In practice you want to restrict the types that can be used.
- Prior to C++20, you would use SFINAE (substitution failure is not an error) to restrict types.
- In C++20, you can use *concepts* to restrict types.
- May cover these strategies in detail later in the course.

Summary

- Function templates allow you to write functions that work with any type.
- Template parameters are placeholders for types.
- Type deduction is done at compile time.
- Default template arguments can be provided.
- Overloading and specialization work as expected.

Almost Always Auto `auto`

Auto

- `auto` allows you to declare a variable without specifying its type.
- C++ is stronly typed, so the type is still determined at compile time.
- `auto` follows the same rules as templates.

Auto Examples

```
auto x = 1; // x is int
auto y = 1.0; // y is double
auto z = 1.0f; // z is float
auto w = "hello"; // w is const char*
auto v = std::vector<int>{1,2,3}; // v is std::vector<int>
auto u = std::make_unique<int>(1); // u is std::unique_ptr<int>

auto& r = x; // r is int&
auto&& rr = x; // rr is int&
const auto& cr = x; // cr is const int&
auto* p = &x; // p is int*

// Be careful about references!
auto v1 = v; // v1 is std::vector<int>, copies v!
auto& v2 = v; // v2 is std::vector<int>&, reference to v, no copy
```

Almost Always Auto?

Some C++ folks suggest to use `auto` whenever possible. This can make code more readable and less error prone. However, there are some cases where you should not use `auto`. Essentially, when auto will not give you the correct type back or when you need to be explicit about the type. Using auto can help prevent unititialized variables and reduce duplication.

```
int x; // unititialized
auto y; // error cannot deduce type
auto f = []() { return 1; }; // anonymous types
auto g = std::make_unique<int>(1); // no repeated types
```