MTH 4300/4299: Programming and Computer Science II

Lecture 17: Templates and Generics

Jaime Abbariao

▓▓ The Problem: Type Duplication

We need a function that returns the larger of two values. So we write it for `int`:

```
int my_max(int a, int b) {
  return (a > b) ? a : b;
}
```

Then we need it for `double`, so we copy-paste and change the type:

```
double my_max(double a, double b) {
  return (a > b) ? a : b;
}
```

The bodies are identical. The only difference is the type. If we fix a bug in one, we have to fix it in the other. Sound familiar?

This is the same problem that motivated inheritance — but now across types instead of classes.

The Solution: Before and After

Before (repeated code):

```
int my_max(int a, int b) {
  return (a > b) ? a : b;
}

double my_max(double a, double b) {
  return (a > b) ? a : b;
}
```

After (one template):

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

One function works for any type. The compiler generates the int version, the double version, or any other version automatically — on demand.

What is a Template?

A template is a blueprint that the compiler fills in for a specific type when you use it.

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

Breaking it down:

- `template <typename T>` — declares a template with one type parameter named `T`
- `typename` is a keyword meaning "this is a type placeholder"
- `T` is just a name — you can use any name (`typename Value`, `typename Item`), but `T` is the convention
- Everywhere `T` appears in the function, it will be replaced by the actual type when the function is called

The compiler doesn't generate any code when it sees the template — it waits until you call `my_max<int>(...)` and generates the `int` version at that point.

Function Templates — Syntax

You can call a template function with the type written explicitly:

```
int a = my_max<int>(3, 7);            // T = int
double b = my_max<double>(2.5, 1.1);  // T = double
```

The <int> and <double> tell the compiler which version to generate.

```
int main() {
  std::cout << my_max<int>(3, 7) << std::endl;        // 7
  std::cout << my_max<double>(2.5, 1.1) << std::endl; // 2.5
  return 0;
}
```

▓▓ Template Type Deduction

Writing `<int>` explicitly is optional most of the time. The compiler can deduce the type from the arguments you pass:

```
my_max(3, 7);       // both are int    → T = int    (deduced)
my_max(2.5, 1.1);   // both are double → T = double (deduced)
```

The compiler looks at the argument types and figures out what `T` must be. You don't need to understand the full deduction rules — just know:

- If both arguments are the same type, deduction works automatically
- If they're different types (`my_max(3, 2.5)`), deduction fails — write `my_max<double>(3, 2.5)` to be explicit

Rule of thumb: omit the explicit type unless the compiler complains.

Class Templates — Motivation

The same problem applies to data structures. Recall our linked list from Lectures 6–7 — it stored `int` values. What if we want a list of `double`? A list of `std::string`?

```
IntList    — stores ints
DoubleList — stores doubles   ← copy-paste of IntList, change int → double
StringList — stores strings   ← copy-paste again
```

Every time we add a new type, we duplicate the entire class. Templates let us write the container once and use it with any type.

This is exactly how `std::vector<T>` works. There is one `vector` template in the standard library, and `std::vector<int>`, `std::vector<double>`, and `std::vector<std::string>` are all generated from it.

Class Templates — Syntax

Template syntax works the same way for classes. Here is a `Box<T>` that stores a single value of any type:

```
template <typename T>
class Box {
public:
  Box(T value) : value_(value) {}

  T get() const { return value_; }
  void set(T value) { value_ = value; }

private:
  T value_;
};
```

To create a `Box`, you specify the type in angle brackets:

```
Box<int> int_box(42);
Box<std::string> str_box("hello");

std::cout << int_box.get() << std::endl;  // 42
std::cout << str_box.get() << std::endl;  // hello
```

▓▓ Worked Example: Stack Template

A stack is a last-in, first-out container — like a pile of plates. You can only add or remove from the top.

```
template <typename T>
class Stack {
public:
  void push(T value) {
    data_.push_back(value);
  }

  void pop() {
    data_.pop_back();
  }

  T top() const {
    return data_.back();
  }

  bool empty() const {
    return data_.empty();
  }

private:
  std::vector<T> data_;
};
```

The implementation uses `std::vector<T>` — which is itself a template. One stack works for any type.

Stack Template in Action

```cpp
int main() {
  Stack<int> int_stack;
  int_stack.push(1);
  int_stack.push(2);
  int_stack.push(3);
  std::cout << int_stack.top() << std::endl;  // 3
  int_stack.pop();
  std::cout << int_stack.top() << std::endl;  // 2

  Stack<std::string> str_stack;
  str_stack.push("hello");
  str_stack.push("world");
  std::cout << str_stack.top() << std::endl;  // world

  return 0;
}
```

The same Stack class, instantiated with two different types. No code duplication.

Templates vs. Inheritance

These are two different tools for two different problems:

	Templates	Inheritance + virtual
Problem	Same logic, different types	Same interface, different behavior
When decided	Compile time	Runtime
Example	`Stack<int>` vs `Stack<std::string>`	`Circle::area()` vs `Rectangle::area()`
Key syntax	`template <typename T>`	`virtual`, `override`

Use templates when: the code is identical regardless of type — containers, algorithms, utilities.

Use inheritance when: different classes need to respond differently to the same function call — and you're working through a base class pointer or reference.

They're complementary: `Stack<T>` uses `std::vector<T>` (a template) internally. Both tools belong in your toolkit.

▓▓ Common Pitfall: Keep Templates in One File

When you define a regular class, you can split the declaration (`.h`) from the implementation (`.cpp`). You cannot do this with templates.

If you try to put a template class definition in `stack.cpp` and use it from `main.cpp`, you'll get a linker error like:

```
undefined reference to `Stack<int>::push(int)'
```

Why? The compiler needs to see the full template definition at the point where it instantiates it. If the definition is in a separate `.cpp`, the compiler can't find it when compiling `main.cpp`.

Rule: put all template code — declaration and implementation — in the same file that anyone who uses it will compile.

In this course all code is single-file, so you won't hit this. But you will in larger projects.

The Standard Library is Full of Templates

You've been using templates since Lecture 5 — you just didn't know it:

```
std::vector<int> nums;          // vector<T>    — T = int
std::vector<std::string> words; // vector<T>    — T = string
std::pair<int, double> p(1, 3.14);        // pair<T, U>
std::map<std::string, int> counts;        // map<K, V>
```

Every <int>, <std::string>, <K, V> you've written is a template instantiation. The standard library is almost entirely templates — that's how one vector works for every possible element type.

Now that you know what templates are, you can write your own general-purpose containers and utilities the same way.

Summary

- A template is a blueprint the compiler fills in for a specific type
- `template <typename T>` declares a type parameter; `T` is just a conventional name for "some type"
- Function templates work for any type: explicit `my_max<int>(3, 7)` or deduced `my_max(3, 7)`
- Class templates parameterize entire classes: `Box<T>`, `Stack<T>`
- The compiler generates separate code for each type you use — at compile time
- Templates solve type duplication; inheritance solves behavioral variation — different tools, different problems
- All template code must be visible to the compiler in the same file where it's used
- `std::vector<T>`, `std::pair<T,U>`, `std::map<K,V>` — you've been using templates all semester

Further reading: template specialization (a custom version for one specific type), C++20 concepts (constraining which types T can be).

▓▓▓ Lab: Generic Pair

Create a file called `pair.cpp` with the following:

Class template `Pair<T, U>`:

- Two private members: `T first_` and `U second_`
- Constructor that takes a `T` and a `U`
- Public `get_first()` returning `T`
- Public `get_second()` returning `U`
- Public method `swap_with(Pair<T, U>& other)` that exchanges the contents of this pair with `other`
  - After `a.swap_with(b)`: `a` holds what `b` held, and `b` holds what `a` held
  - Use a temporary variable — do not use `std::swap` on the whole pair

In `main()`:

- Create `Pair<int, std::string> a(1, "one")` and `Pair<int, std::string> b(2, "two")`
- Print both pairs, call `a.swap_with(b)`, then print both again
- Create `Pair<double, double> p(3.14, 2.71)` and print it

Lab: Expected Output

```
Before swap:
a: 1 one
b: 2 two

After swap:
a: 2 two
b: 1 one

p: 3.14 2.71
```

Hints:

- `swap_with` needs to swap both `first_` and `second_` with `other`'s members
- `Pair<double, double>` uses the same template with `T = double` and `U = double` — templates don't require the two types to differ
- No need for a `print` method — just call `get_first()` and `get_second()` directly in `main`