1 Function Templates

To understand what a `template` is, it's helpful to first grasp why they're used. Consider how you might implement a function, `max()`, that takes two arguments and returns the greater of the two. This function is general enough to work on any type that supports the comparison `operator>`. Here's an implementation for a specific type, like `int`:

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

Now, imagine you need to implement the same `max()` function for `float` values:

```
float max(float a, float b)
{
    return (a > b ? a : b);
}
```

And for `int*` values:

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

At this point, the problem becomes clear: you are duplicating the same logic for every data type. This redundancy clutters the codebase and makes it difficult to build a reusable library for generic functionality. Consider a custom class that can be compared:

```
class YourClass
{
public:
    bool operator>(const YourClass& c);
    // ...
};
```

No library in the world can create generic functionality such as `max()` that works for `YourClass`–after all, they don't even know it exists. Instead of shouting from the rooftops, pleading with the library architects to recognize the sheer brilliance of `YourClass`, and hoping they'll bless us with support in the next version, we can take matters into our own hands with a more flexible approach:

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

Behold the almighty `template`. Templates allow us to write a single function that works with any type `T`, provided `T` supports the operations we perform on it. By declaring `template<class T>`, we instruct the compiler to generate a **generic function**–a blueprint that can operate on any type `T`. We can explicitly specify `T` when calling the function:

```
#include <iostream>

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

int main()
{
    int   a { 32 },  b { 64 };
    char  p { 'p' }, q { 'q' };
    float x { 1.5 }, y { 1.0 };

    std::cout << max<int>(a, b) << std::endl;   // T = int
    std::cout << max<char>(p, q) << std::endl;  // T = char
    std::cout << max<float>(x, y) << std::endl; // T = float
}
```

Terminal

```
$ ./a.out
64
q
1.5
```

When `max()` is called, the compiler generates a specific version of the function tailored for the provided type `T`. This allows us to reuse the same logic across different types without duplicating code–an essential feature in library development, as it enables the creation of reusable code that can handle custom types.

1.1 Function Template Declaration and Definition

As with functions and classes, templates must be declared before they can be used. A template declaration specifies the template's name, the types it operates on, but it does not provide the full implementation. The syntax for a template declaration looks like this:

```
template<class T, class U>
void foo(T a, U b);
```

In this example, `T` and `U` are **template parameters**, which are placeholders representing types that will be provided when the template is instantiated. These parameters define the "family" of functions that the template describes.

The template definition is where you implement the function or class that was declared earlier. For example, continuing from the previous declaration:

```
template<class T, class U>
void foo(T a, U b)
{
    // do something...
}
```

1.1.1 `typename` Keyword

In the parameter list of a template declaration, `typename` can be used interchangeably with `class` to declare template parameters. As a result, the following two declarations are equivalent:

```
template<class T>
T max(T a, T b);
```

```
template<typename T>
T max(T a, T b);
```

For most cases, the choice between `typename` and `class` is purely a matter of style or preference. Reasons to prefer one keyword or the other will present themsevles throughout the following chapters. However, there are contexts where only `typename` is valid (and contexts where only `class` is valid)these cases will be discussed in later chapters.

1.2 Function Template Instantiation

When you declare a template function like `max()`, you are not creating a function. Instead, you're **defining a blueprint** that the compiler uses to create functions as needed. This process is called **template instantiation**. A function is only instantiated when you call it with a specific type, prompting the compiler to generate a corresponding version. We can illustrate this idea with an example comparing the compiler output of two source files:

```
// filea.cpp
template<class T>
T abs(T t) { return (t >= 0 ? t : -t); }

int main() {}
```

```
// fileb.cpp
int main() {}
```

Terminal

```
$ clang++ -S filea.cpp -o filea.s
$ clang++ -S fileb.cpp -o fileb.s
$ diff filea.s fileb.s | wc
       0       0       0
```

> In the above example, we have two source files:
>
> 1. `filea.cpp` defines a template function `abs`, but it is never called.
> 2. `fileb.cpp` is a minimal C++ program containing only `main()`.
>
> Since there are no calls to `abs()` in `filea.cpp`, the compiler does not generate any machine code for it. When we compile both files into assembly using the `-S` flag and compare the outputs with `diff`, they are identical. This confirms that template code is not instantiated until it is used.

Template instantiation occurs at compile-time when the function is called, generating a unique function definition for each combination of template parameters. The beauty of this approach lies in its efficiencyyou only pay for what you use. If your program only requires a `max()` function for `int` and `double`, the compiler generates code only for those types, leaving other potential instantiations unused. This selective generation avoids unnecessary bloat, keeping your compiled binaries smaller and your compile times faster.

1.2.1 Implicit Instantiation

When you use a template function, the compiler must create a specific version of that function for the given type. If the required instance does not already exist (or if the existence of the defintion affects the semanitcs of the program), the compiler will generate it through a process called **implicit instantiation**. In this process, the compiler deduces the template parameter `T` based on the types of the function arguments.

```
// implicit.cpp
template<class T>
T max(T a, T b) { return (a > b ? a : b); }
```

```
int main()
{
    int x { max(5, 6) };        // max<int>(int, int)
    double y { max(5.0, 6.0) }; // max<double>(double, double)
    char z { max('5', '6') };   // max<char>(char, char)
}
```

1.2.2 Template Argument Deduction

Above, notice that we don't explicitly tell the compiler which version of `max()` to call. Instead, the compiler **deduces** the template parameter `T` from the function arguments. To implicitly instantiate a function template, the compiler must be able to determine every template argument, though they don't always need to be directly specified. This deduction mechanism enables the use of template `operator` functions, as there's no valid syntax to explicitly specify types passed to arguments without rewriting them as a function call:

```
#include <iostream>

int main()
{
    // std::cout uses the template operator<< to handle
    // printing different types
    std::cout << "7" << 7 << 7.0 << std::endl;
}
```

1.2.3 Explicit Instantiation

Explicit instantiation forces the compiler to generate a specific version of a function with particular template parameters. This can be useful when you want to control exactly when and where the template code is generated. To explicitly instantiate a template function, simply follow the `template` keyword with a declaration for the function:

```
// explicit instantiation of max<int>()
template int max<int>(int, int);
```

We can verify that explicit instantiation indeed generates the function by using the `objdump` command to view the symbol table of the object file:

```
// filea.cpp
template<class T>
void foo() { /* fooing... */ }

template void foo<int>();    // instantiation of foo<int>()
template void foo<double>(); // instantiation of foo<double>()

int main() {}
```

```
// fileb.cpp
template<class T>
void foo() { /* fooing... */ }

// no explicit instantiations...

int main() {}
```

Terminal

```
$ clang++ filea.cpp -std=c++23 -c
$ clang++ fileb.cpp -std=c++23 -c
$ objdump -tC filea.o
SYMBOL TABLE:
0000000000000000 l     F __TEXT,__text ltmp0
0000000000000010 l     O __LD,__compact_unwind ltmp1
0000000000000004  w    F __TEXT,__text void foo<double>()
0000000000000000  w    F __TEXT,__text void foo<int>()
0000000000000008 g     F __TEXT,__text _main
$ objdump -tC fileb.o
SYMBOL TABLE:
0000000000000000 l     F __TEXT,__text ltmp0
0000000000000008 l     O __LD,__compact_unwind ltmp1
0000000000000000 g     F __TEXT,__text _main
```

> In this example, `foo<int>()` and `foo<double>()` are explicitly instantiated in `filea.cpp`. Running `objdump` on the object file shows entries for `void foo<int>()` and `void foo<double>()`. However, in `fileb.cpp`, since no explicit instantiations are provided, no such entries appear in the symbol table.

It might still be unclear why you would want to use explicit instantiation, given that C++ can automatically instantiate template functions for you. To demonstrate the need for explicit instantiation, consider how your project grows in complexity. As it expands, you'll likely want to break it into multiple modules, across `.cpp` and `.h` files. For example:

```
// lib.h
template<class T>
T max(T a, T b);
```

```
// lib.cpp
template<class T>
T max(T a, T b)
{
    return (a > b ? a : b);
}
```

Now, you might want to use this library in your main application:

```
// main.cpp
#include <iostream>
#include "lib.h"

int main()
{
    std::cout << max(4, 5) << std::endl;
}
```

However, when you try to compile your masterpiece, you'll encounter a linker error:

Terminal

```
$ clang++ -std=c++23 main.cpp lib.cpp
Undefined symbols for architecture arm64:
  "int max<int>(int, int)", referenced from:
      _main in main-cbf51c.o
ld: symbol(s) not found for architecture arm64
```

When the translation unit for `main.cpp` is compiled, the compiler sees a reference to `int max<int>()` and assumes the linker will be able to resolve it. However, when `lib.cpp` is compiled, the compiler finds no instantiations of `max()`, and therefore does not generate any concrete functions for it. As a result, when it's time to link the object files, the necessary function does not exist, leading to the undefined symbol error. This issue can be resolved by explicitly instantiating `max<int>()` in `lib.cpp`:

```
// lib.cpp
template<class T>
T max(T a, T b)
{
    return (a > b ? a : b);
}

template int max<int>(int, int);
```

By doing this, you ensure that the function is created, even if it's used across multiple translation units.

Like all features in programming, explicit instantiation comes with trade-offs. While it offers improved modularity by controlling when and where template code is generated, it sacrifices some level of generality because it requires explicitly defining functions for each type, limiting the flexibility to handle new or unforeseen types. This is why template libraries are often implemented entirely in header files. However, the trade-off can be worth it, especially when you know in advance which specific types your library needs to support.