1 Function Templates

To understand what a `template` is, it's helpful to first grasp why they are 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`:

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

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

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

And for `char` values:

```cpp
char max(char a, char 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:

```cpp
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.

```cpp
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`, as long as `T` supports the operations we perform on it. When we declare `template<class T>`, we are instructing the compiler to generate a **generic function**–a blueprint that can operate on any datatype `T`. We can explicitly specify `T` when calling the function:

```cpp
#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
```

1.1 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 demonstrate this by examining the compiler output of two source files:

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

int main() {}
```

```cpp
// 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 no calls to `abs()` are made 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 happens at compile-time when the function is called, and a unique function definition is generated 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 `char`, the compiler generates code only for those types, leaving other potential instantiations unused. This selective generation avoids unnecessary bloat, keeping both your compiled binaries smaller and your compile times faster.

1.1.1 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, 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.

```cpp
// 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)
}
```

Notice that we don't explicitly inform the compiler which version of `max()` to call. Instead, the compiler **deduces** the template parameter `T` from the function arguments. We can confirm that the compiler generates separate function instances by inspecting the assembly output:

```
Terminal
$ clang++ -std=c++23 -S implicit.cpp
$ cat implicit.s | grep "max.*:"
__Z3maxIiET_S0_S0_:                         ; @_Z3maxIiET_S0_S0_
__Z3maxIdET_S0_S0_:                         ; @_Z3maxIdET_S0_S0_
__Z3maxIcET_S0_S0_:                         ; @_Z3maxIcET_S0_S0_
```

In `implicit.cpp`, we instantiate the template function `max()` with three template arguments: `int`, `double`, and `char`. Using the `-S` flag, we produce assembly output in `implicit.s`. By searching for functions containing `"max"` using `grep`, we find three separate definitions. The compiler encodes type information into the function name through **name mangling**. The identifiers `i`, `d`, and `c` in the mangled names correspond to `int`, `double`, and `char`, respectively. This confirms that the compiler has generated a unique function instance for each type.

1.2 Explicit Instantiation