# TEMPLATES AND STL

---

## Generics in C++

One of the things that C++ manages to do very well is to provide generic programming to us. The *STL* is one of the best functionalities that have been ever made in the language. `Vectors`, `lists`, `for_each`, and many many more containers are available to use to shorten our code, make it more understandable, and make it independent from types! The C++ template mechanism was also Turing complete (you can compute any computable value with it), which led to template meta programming. With this tutorial we will go over what templates can do, how we can use them (and when) and then we will concentrate on the STL.

## Template programming

- Templates
- Compile-time polymorphism
- `typename`, `class`, `typedef`
- Template Meta Programming

### Templates

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

```
template<typename T>
void doProcessing(T& w)
{
    if (w.size() > 10 && w != someNastyWidget)
    {
        T temp(w);
        temp.normalize();
        temp.swap(w);
    }
}
```

There is a good chance that you will not be dealing with a lot of template code in C++ and that's okay. But it is still a good idea to be able to understand and write templates in C++. In fact, compared to Java or C#, C++ has more flexibility when it comes to writing such code (although this isn`t always good). So what are templates? They are just something that can be substituted for something else during run-time. It allows us to greatly reuse code, abstracts us from dealing with types, but at the same time, it makes executables larger, and sometimes slower.

Okay so let`s look at a template.

In the first piece of code, we have a template function that returns the larger of the two types. These types can be literally anything - they can by `int`, `double`, `char`, `someobject`, etc. As long as the type `T` has an implementation of `>`, this will compile and this will run. In the second example we can see even clearer when we say that some template must support something. We are passing `w` by reference (so maybe it's an object). The parameter uses the `size()`, `normalize()`, and `swap()` functions (okay, it's an object). We also have the operator `!=` in use, and we also have a copy constructor. If we were to just substitute `T& w` with `SpecificObject& w`, we would not worry that `w` supports all of these functionalities, because we have explicitly stated what we are passing. But since we are using a template, we are *expecting* that `T` supports them. This is the meaning of an *implicit* interface. Writing templates means that we are dealing with implicit interfaces in the first place. It's okay not to stress it that much when we are writing generic functions, which are designed to handle build-in types (such as `int` and `double`), but when we are working with user defined types, we have to be careful what the function is actually doing, and does it support the interface.

## Compile-time polymoprhism

# The STL

- Vectors and strings
- Maps and sets
- Lambda functions

# Exercises

# Resources