https://youtu.be/bUphr3EuM8A

Templates

Templates enable generic programming by generalizing a function to apply to any class. Specifically, templates use *types* as parameters so that the same implementation can operate on different data types.

For example, you might need a function to accept many different data types. The function acts on those arguments, perhaps dividing them or sorting them or something else. Rather than writing and maintaining the multiple function declarations, each accepting slightly different arguments, you can write one function and pass the argument types as parameters. At compile time, the compiler then expands the code using the types that are passed as parameters.

```
template <typename Type> Type Sum(Type a, Type b) { return a + b; }

int main() { std::cout << Sum<double>(20.0, 13.7) << "\n"; }
```

Because `Sum()` is defined with a template, when the program calls `Sum()` with `double`s as parameters, the function expands to become:

```
double Sum(double a, double b) {
    return a+b;
}
```

Or in this case:

```
std::cout << Sum<char>('Z', 'j') << "\n";
```

The program expands to become:

```
char Sum(char a, char b) {
    return a+b;
}
```

We use the keyword `template` to specify which function is generic. Generic code is the term for code that is independent of types. It is mandatory to put the `template<>` tag before the function signature, to specify and mark that the declaration is generic.

Besides `template`, the keyword `typename` (or, alternatively, `class`) specifies the generic type in the function prototype. The parameters that follow `typename` (or `class`) represent generic types in the function declaration.

In order to instantiate a templatized class, use a templatized constructor, for example: `Sum<double>(20.0, 13.7)`. You might recognize this form as the same form used to construct a `vector`. That's because `vector`s are indeed a generic class!