Overloading Functions

In this lesson, we'll see the same function perform different operations based on its arguments.

WE'LL COVER THE FOLLOWING

We previously learned that each function has a specific number of arguments which have to be passed when calling it. If we increase or decrease the number of arguments in the call, we'll get a compilation error:

```
#include <iostream>
using namespace std;

double product(double x, double y){
  return x * y;
}

int main() {
  cout << product(10, 20) << endl; // Works fine
  cout << product(10) << endl; // Error!
}
```

Line 10 blows up the code.

The compiler doesn't know how to handle arguments it wasn't expecting. However, one of the coolest things we can do with functions is that we can **overload** them.

What is Overloading?

> Overloading refers to making a function perform different operations

based on the nature of its arguments.

We could redefine a function several times and give it different arguments and function types. When the function is called, the appropriate definition will be selected by the compiler!

Let's see this in action by overloading the `product` function which we just wrote:

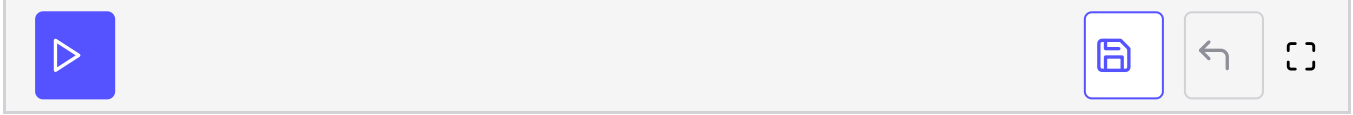
```
#include <iostream>
using namespace std;

double product(double x, double y){
  return x * y;
}

// Overloading the function to handle three arguments
double product(double x, double y, double z){
  return x * y * z;
}

// Overloading the function to handle floats
float product(float x, float y){
  return x * y;
}

int main() {
  double x = 10;
  double y = 20;
  double z = 5;
  float a = 12.5;
  float b = 4.654;
  cout << product(x, y) << endl;
  cout << product(x, y, z) << endl;
  cout << product(a, b) << endl;
  // cout << product(x) << endl;
}
```



The product function now works in three different ways!

In the code above, we see the same function behaving differently when encountering different types of inputs. We still have to define which cases it can handle. **Line 27** would crash since `product` doesn't know how to handle a single argument.

> **Note**: Functions which have no arguments and differ only in the return

types cannot be overloaded since the compiler won't be able to differentiate between their calls.

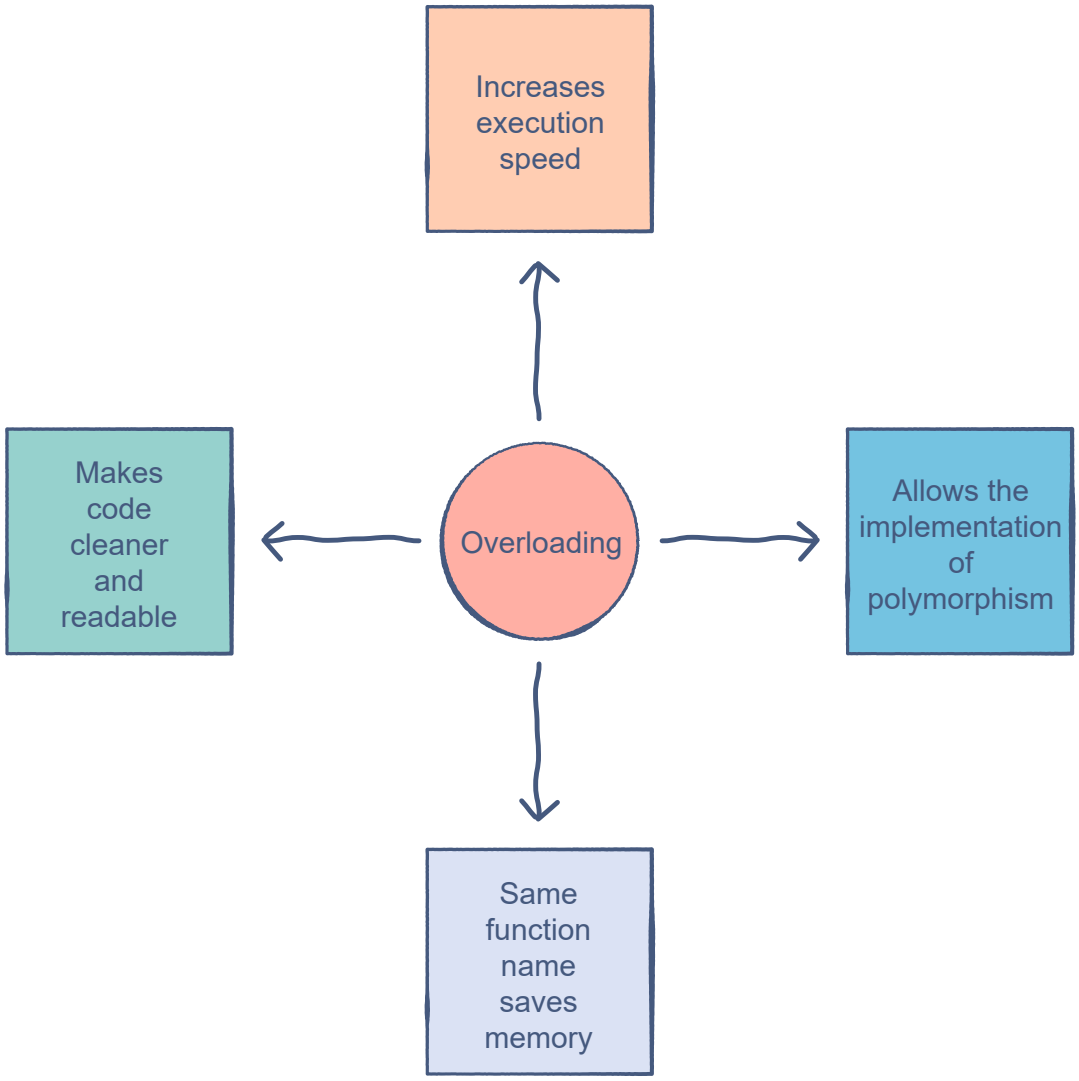
Advantages of Function Overloading

One might wonder that we could simply create new functions to perform different jobs rather than overloading the same function. However, under the hood, overloading saves us memory in the system. Creating new functions is more costly compared to overloading a single one.

Since they are memory-efficient, overloaded functions are compiled faster compared to different functions, especially if the list of functions is long.

An obvious benefit is that the code becomes simple and clean. We don't have to keep track of different functions.

Polymorphism is a very important concept in object-oriented programming. It will come up later on in the course, but function overloading plays a vital role in its implementation.



We know enough about function in order to delve deeper into the principles of OOP. In the next chapter, we will explore **pointers**.

Before that, be sure to check out the quiz and exercises ahead to test your concepts of functions. Good luck!