Function Overloading

Let's see what function overloading is all about.

WE'LL COVER THE FOLLOWING

Definition

> **Function overloading** is the concept of affecting a function's behavior based on the **number of parameters** or their **types**.

This way, functions with different parameters can coexist with the same name. Function overloading works with different parameters. The function prototype can change and the return type changes according to parameters being returned.

Types of arguments

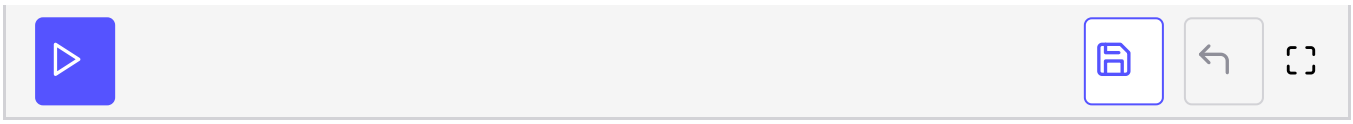
```
#include <iostream>
#include <string.h>

const char* min(const char* s, const char* t){
  return (strcmp(s,t) < 0) ? s : t;
}
float min(float x, float y){
    return (x < y) ? x : y;
}

int main() {
  const char* s = min("abc", "xyz");
```

```
  float f = min(4.45F, 1.23f);
  int f2 = min(2011, 2014);
  // float f3 = min("abc", 1.23f);

  std::cout << s << std::endl;
  std::cout << f << std::endl;
  std::cout << f2 << std::endl;
}
```



The `min` function behaves differently based on the types of arguments provided. We have defined it to handle `const char*` and `float` arguments.

In line 14, the `int` arguments are implicitly converted to `float` and an error is not thrown. The result is again converted to an `int` to match the type of `f2`.

Line 15 would not work since the function prototype is not written to handle different types of arguments (i.e., the combination of strings, floats, and integers)

Increase the number of arguments

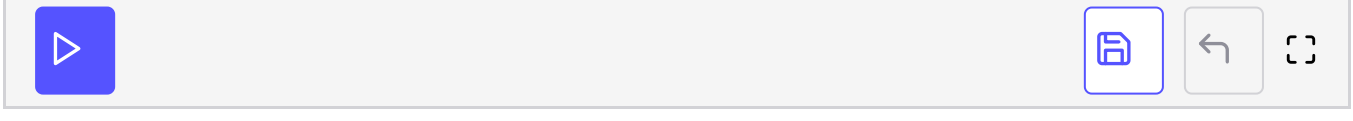
```
#include <iostream>
#include <string.h>

float min(float x, float y){
    return (x < y) ? x : y;
}

float min(float x, float y, float z){
    return x < y ? (x < z ? x : z) : (y < z ? y : z);
}

int main() {
  float f = min(4.45F, 1.23f);
  float f2 = min(4.45f, 1.23f, 0.19f);

  std::cout << f << std::endl;
  std::cout << f2 << std::endl;
}
```



Now, the function can handle three `float` arguments as well. We can extend this to as many as we want.

Rules for choosing the right function

Rules for choosing the right function

- We need to search for a function with the exact type.
- We need to apply type promotion to the arguments.
- We need to convert arguments.

Another thing to keep in mind is that the compiler ignores **references** when overloading functions. For example, `min(int x, int y)` is the same as `min(int &x, int &y)`.

Furthermore, the `const` and `volatile` qualifiers are also ignored. `min(int x, int y)` is the same as `min(volatile int x, volatile int y)`. However, `min(const int& x, const int& y)` is a different overloading function as the `const` keyword applies to `int` but not to the reference property.

Why do we overload?

In the end, we can always create functions to perform different operations. But what would happen if we were running a large scale application that needs to support a variety of similar functions?

It would be impractical to remember the names of all unique functions. By overloading, we can introduce a great deal of simplicity and readability in our code.

We only need to provide the appropriate arguments and the compiler will handle the rest of the functionality.

In the next lesson, we will be introduced to lambda functions.