Overloaded Functions

Function overloading allows you to use the same name for different functions in the same program, provided each takes a different **type or number** of arguments. The pattern of arguments received by a function—which refers only to the number and types of the arguments and not the parameter names—is called its **signature**.

Here's an example. Both the `<cmath>` header and `<cstdlib>` declare `abs()` functions for returning the absolute value of a number. Here are the three functions in `<cstdlib>`:

```
int abs(int n);
long abs(long n);
long long abs(long long n);
```

Here are the four versions of the function in `<cmath>`:

```
double abs(double x);
float abs(float x);
long double abs(long double x);
double abs(T x); // a template called when no other matches
```

There are even versions for **complex numbers** in the header `<complex>`. The only difference between these functions is the **types of the parameters**. The compiler chooses **which version to call** by looking at the types of the arguments supplied.

- Called with an `int`, the compiler **calls** the version which **takes** an `int`.
- Called with a `double`, the compiler will choose the version from `<cmath>`, which takes a `double`.

> *If you call `abs()` with an integer, and **only** include `<cmath>`, but forget `<cstdlib>`, then a special **generic version** of `abs()` that takes a type `T` parameter will be called. The difference between the generic version, and the overloaded `abs(int)` version, is that the generic version **always** returns a `double`, not an `int`.*

Overloading makes it easier for programmers to remember function names when the same operation is applied in slightly different contexts. C, which does not have overloading, requires different names for each different absolute value function: `iabs, fabs, dabs, labs, llabs, and so on.`