C++: FUNCTIONS AND TEMPLATES

C FUNCTIONS WERE QUITE SIMPLE

```c
int max(int a, int b)
{
    return a > b ? a : b;
}
```

In C, each function name is associated to a specific signature

C++ FUNCTION OVERLOADING

```
int max (int a, int b);

int max (int a, int b, int c);

long max (long a, long b);
```

In C++, each function is defined by a name and a list of typed parameters

Two C++ functions cannot have the same parameters and different return types

C++ OPTIONAL FUNCTION PARAMETERS

```
void log_message(std::string message, bool as_error = false);

std::vector<int> get_fibonnaci_sequence(
    size_t values,
    size_t starting_position = 0
);
```

Parameters can be turned optional by specifying a default value

Optional parameters cannot be followed by non-optional parameters

THE ABI PROBLEM

IN C

```c
int max(int, int);
```

In the library file:

```
max
```

THE ABI PROBLEM

IN C++

```
int max(int, int);
```

In the library file compiled by GCC:

```
_Z3maxii
```

In the library file compiled by MSVC:

```
?max@@YAHHH@Z
```

This is called name mangling

THE ABI PROBLEM

THE C++ WAY TO GENERATE C ABI FUNCTIONS

```cpp
extern "C" int max(int, int);
```

extern "C" instructs the C++ compiler to not mangle the functions names in the binary files. This is required only on the function declaration (typically in the header file)

THE ABI PROBLEM

THE C++ WAY TO GENERATE C ABI FUNCTIONS

```
extern "C" {
    int max(int, int);
    int min(int, int);
}
```

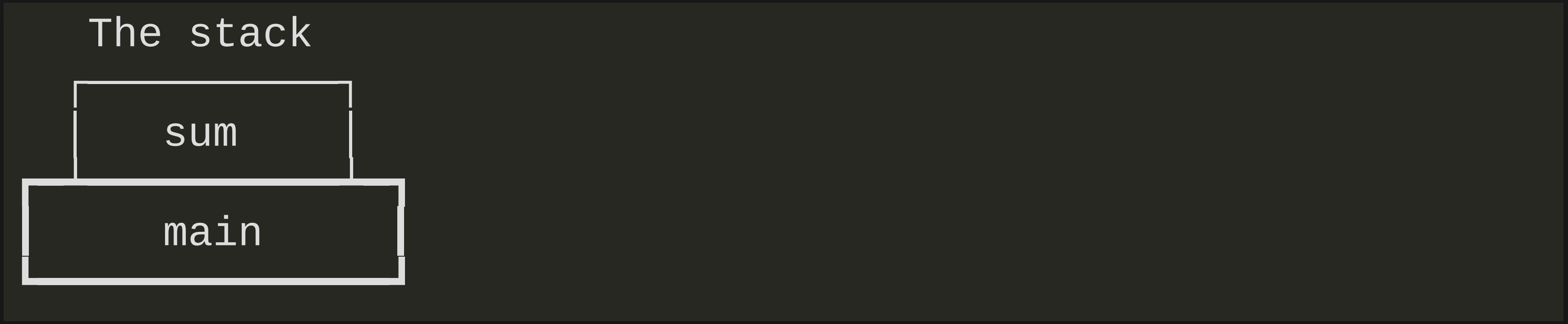
This special instruction can be followed by a block containing multiple functions declarations

Functions declared that way cannot be overloaded

A REMINDER ABOUT FUNCTIONS

Function aren't some special language abstraction; they are not equivalent to having the function code copy-pasted in place of the function call

Calling a functions means - for the CPU - creating a new memory environment stacked on the current one.



A REMINDER ABOUT FUNCTIONS

Calling a function has a resource cost, an overhead

Sometimes we just write function as small chunks of code that we don't want to duplicate in our codebase, but we don't want the overhead either

Introducing ...

INLINE FUNCTIONS

INLINE FUNCTIONS

```
inline int max(int a, int b)
{
    return a > b ? a : b;
}
```

The inline keyword declares that the following function should not be compiled as a function, but as a chunk of code that's going to be written in place of every function call

The definition of a function declared as inline can be written directly in the header file

INLINE FUNCTIONS

However, the inline keyword is just an hint for the compiler.

The compiler may decide to not inline a function declared as inline, or to inline a regular function if it considers that it would be a better optimisation.

TEMPLATES

A function template defines how to build a function depending on compile-time parameters

```
template <int A, int B>
int max ()
{
    return A > B ? A : B;
}
```

```
std::cout << max <3, 4> () << std::endl;
```

The template parameters are defined inside angle brackets and should be defined at compile-time

TEMPLATES

The template function definition is not a function definition.

It just describes what would be the code of that function depending on compile-time parameters.

A template definition always takes place in the header file.

TEMPLATES

```
template <int A, int B>
int max ()
{
    return A > B ? A : B;
}
```

This code compiles as nothing.

TEMPLATES

```
template <int A, int B>
int max ()
{
    return A > B ? A : B;
}
```

```
int x = max <3, 4> ();
```

This code generates and compiles only the version of the max function with 3 and 4.

TEMPLATES

```
template <int A, int B>
int max ()
{
    return A > B ? A : B;
}
```

```
int x = max <3, 4> () + max <10, -10> ();
```

This code generates and compiles both specified versions of the max function.

TEMPLATES

```
template <long X>
long factorial ()
{
    return X * factorial<X-1>();
}
```

```
template <>
long factorial<0>()
{
    return 1;
}
```

```
long x = factorial<10>();
```

This code generates and compiles at compile time all required versions of the function, recursively.

TEMPLATES

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

Template parameters can be typenames.

This allows generic programing.

The calls to the generic templated function will compile as long as the code would compile with the specified type.