**Functions**

1. **Built-in Functions**

#include <iostream> 🡪 cin cout

#include <cmath> 🡪 sqrt pow fabs(double) ceil floor

#include <string> 🡪 string

#include <cstdlib> 🡪 abs-(int) labs-long srand rand

#include <ctime> 🡪 time(NULL) for random number generator

Function declaration and definition is similar to C (must **declare** before int main(). Can **define** right when declared, or after int main()).

1. **Default value**

**Must always define default values in function declarations (in header file)**

1. **Function Overload**

If you want a function to accept an argument that can be either int or double; some function parameters to be optional 🡪 Function overloading

In the process of function overloading, you can *give multiple C++ functions the same name*. However, at least one of these conditions are true:

* Each has different type parameters.
* Each has a different number of parameters.

🡪 We can change the way a function behaves based on the arguments passed.

void print\_input\_twice (int num) { std::cout << num\*2; }

void print\_input\_twice (char c) { std::cout << c+c; }

1. **Local Scope**

Note that the using keyword, also has scope like declaring variables. If we use using inside a function, we would also be using the namespace inside that function only.**Inline Functions**

Using inline advises the compiler to copy the function’s body where the function call is

Note that you should ALWAYS add the inline keyword if you are inlining functions in a header (unless you are dealing with member functions, which are automatically inlined for you).

The compiler **may not perform inlining** in such circumstances as:

* If a function contains a loop. (for, while and do-while)
* If a function contains static variables.
* If a function is recursive.
* If a function return type is other than void, and return statement doesn’t exist in function body.
* If a function contains a switch or goto statement.

Recall: inlining saves the overhead of storing the address of function call, branch (subroutines), moving registers for returns… 🡪 **Can be faster for small functions**

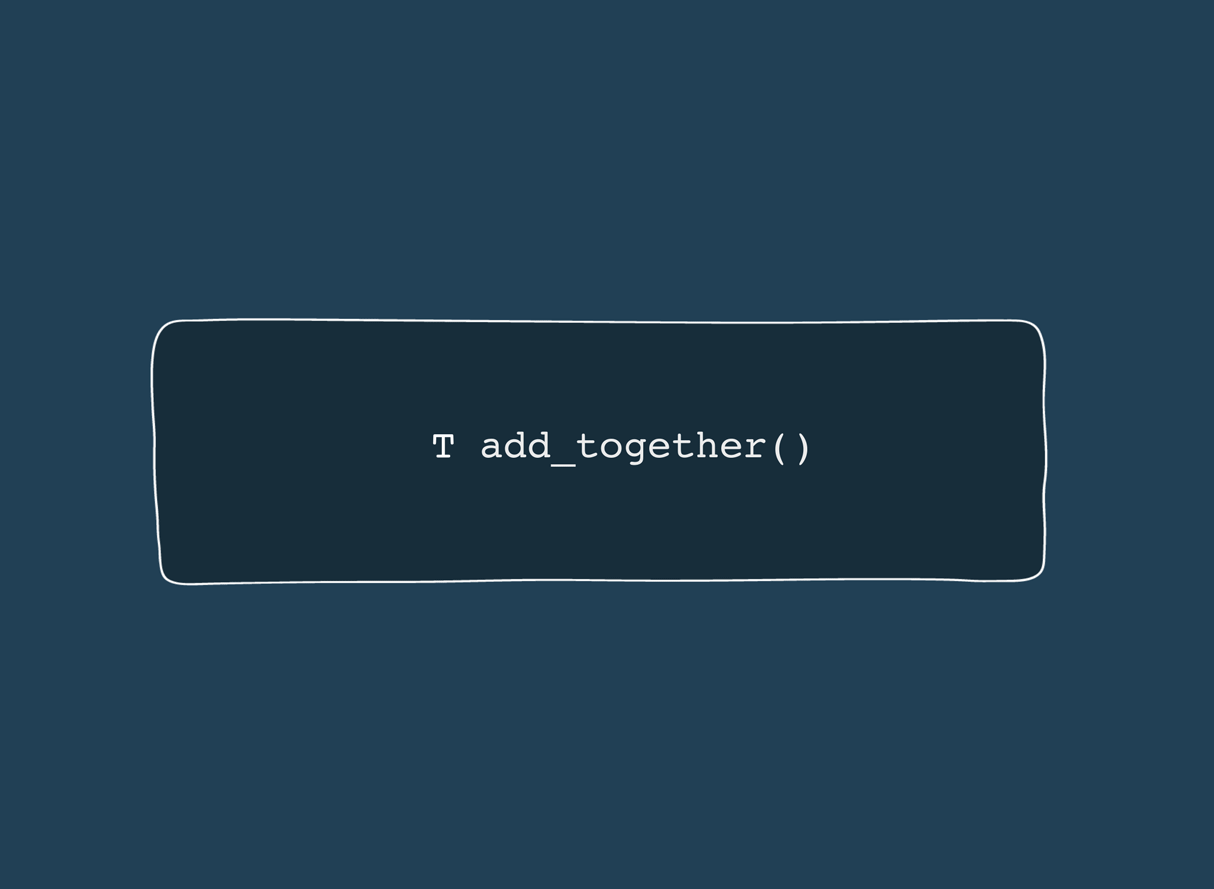
**Advantages (when usefull)**:

* Function call overhead doesn’t occur.
* It also saves the overhead of push/pop variables on the stack when a function is called.
* It also saves the overhead of a return call from a function.
* When you inline a function, you may enable the compiler to perform context-specific optimization on the body of the function. Such optimizations are not possible for normal function calls. Other optimizations can be obtained by considering the flows of the calling context and the called context.

**Disadvantages (when making it worse)**:

* The added variables from the inlined function consume additional registers, After the in-lining function if the variable number increases. So the number of registers going to be used for the variables will also get increased. So if after function inlining, variable numbers increase drastically (complex functions) then it would surely cause overhead on register variable resource utilization.
* If you use too many inline functions then the size of the binary executable file will be large, because of the duplication of the same code.
* Too much inlining can also reduce your instruction cache hit rate, thus reducing the speed of instruction fetch from that of cache memory to that of primary memory.
* The inline function may increase compile time overhead if someone changes the code inside the inline function.
* Inline functions might cause thrashing because inlining might increase the size of the binary executable file. Thrashing in memory causes the performance of the computer to degrade.

**Function Templates \*\***



If the function body is basically the same for different data types, we shouldn’t use function overload, but function template instead.

A template is a C++ tool that allows programmers to add data types as parameters.

*(In fact, std::string and std::vector are both template-based types)*.

Unlike regular functions, templates are entirely created in the header files.

Templates let us choose the type implementation right when you call the function. The type we choose may apply to the return type, a parameter type, or both.

template <typename T>

void print\_cat\_ears(T item) {

// Function body

}

Then, we can call the function for any type of parameter passed, as long as it works with the function body.

**Note:** Using templates will slow down the program’s compile time, but speed up the execution time.