- C++
- parameter
  - parameter by copy
  - parameter by pointer
  - parameter by reference
- const cast
- overload
  - overload of functions
  - overload of operators
- inline function and macro
- namespace
- class
- exceptions

---

lucky777

C++

parameter

parameter by copy

```
//foo.cpp
#include <iostream>
using namespace std;

void swap(int a, int b) {
    cout << "a=" << a << ", b=" << b << endl;
    int tmp = a;
    a = b;
    b = tmp;
    cout << "a=" << a << ", b=" << b << endl;
}

int main() {
    int first = 7;
    int second = 14;
    cout << "first=" << first << ", second=" << second << endl;
    swap(first, second);
    cout << "first=" << first << ", second=" << second << endl;
    return 0;
}
```

```
+OUTPUT:
first=7, second=14
a=7, b=14
a=14, b=7
first=7, second=14
```

> `a` and `b` are **copies** of `first` and `second`. When we modifiy `a` and `b`, it won't affect `first` and `second` because they are local variables only living inside of the function `swap()`.

parameter by pointer

```
//foo.cpp
#include <iostream>
using namespace std;

void swap(int* a, int* b) {
    cout << "a=" << *a << ", b=" << *b << endl;
    int tmp = *a;
    *a = *b;
    *b = tmp;
    cout << "a=" << *a << ", b=" << *b << endl;
}

int main() {
    int first = 7;
    int second = 14;
    cout << "first=" << first << ", second=" << second << endl;
    swap(&first, &second); //addresses
    cout << "first=" << first << ", second=" << second << endl;
    return 0;
}
```

```
+OUTPUT:
first=7, second=14
a=7, b=14
a=14, b=7
first=14, second=7
```

> Here we give the addresses of the variables as parameters, so when we modifiy `a` and `b`, we are modifying directly `first` and `second` aswell. These variables point to the same place in the memory.

parameter by reference

```cpp
//foo.cpp
#include <iostream>
using namespace std;

void swap(int& a, int& b) {
    cout << "a=" << a << ", b=" << b << endl;
    int tmp = a;
    a = b;
    b = tmp;
    cout << "a=" << a << ", b=" << b << endl;
}

int main() {
    int first = 7;
    int second = 14;
    cout << "first=" << first << ", second=" << second << endl;
    swap(first, second);
    cout << "first=" << first << ", second=" << second << endl;

    int test = 777;
    int& alias = test; //this is NOT a copy of test
    cout << "alias=" << alias;
    return 0;
}
```

```
+OUTPUT:
first=7, second=14
a=7, b=14
a=14, b=7
first=14, second=7
alias=777
```

> In c++ we can precise that we are working directly with references. This code will do the same as the previous example with **parameters by pointers**, but it is easier to read and write.

```cpp
//foo.cpp
#include <iostream>
using namespace std;

int& mini(int& a, int& b) {
    return a<b ? a : b; //Returns the minimum
}

int main() {
    int first = 7;
    int second = 14;

    int result1 = mini(first, second); //This is a copy of first
    int& result2 = mini(first, second); //This is NOT a copy of first
```

```
    cout << "first=" << first;
    cout << ", result1=" << result1;
    cout << ", result2=" << result2 << endl;
    first++;
    cout << "first=" << first;
    cout << ", result1=" << result1;
    cout << ", result2=" << result2 << endl;
}
```

```
+OUTPUT:
first=7, result1=7, result2=7
first=8, result1=7, result2=8
```

> `result1` is a **copy** of `first`, but `result2` is a **reference** to `first`.

const cast

```
int& mini(const int& a, const int& b) {
    return const_cast<int&>(a<b ? a : b);
}
```

> In this code, as `a` and `b` are `const`, this return wouldn't work:
> `return a<b ? a : b;` *(error)*
> Instead we need to cast them and remove their `const` type. This is why we are using the operation `const_cast<type>`.

overload

overload of functions

```
//foo.cpp
#include <iostream>
using namespace std;

void demo() {
    cout << "Demo with no parameters" << endl;
}

void demo(int foo) {
    cout << "Demo with an integer parameter" << endl;
}

void demo(double foo) {
    cout << "Demo with a double parameter" << endl;
```

```
}

void demo(int* foo) {
    cout << "Demo with a pointer parameter" << endl;
}

void demo(const char* foo1, int foo2) {
    cout << "Demo with two parameters" << endl;
}

int main() {
    int foo = 777;
    demo();
    demo(7);
    demo(foo);
    demo(7.7);
    demo(&foo);
    demo(nullptr); //pointer null
    demo("lucky", 777);
    return 0;
}
```

```
+OUTPUT:
Demo with no parameters
Demo with an integer parameter
Demo with an integer parameter
Demo with a double parameter
Demo with a pointer parameter
Demo with a pointer parameter
Demo with two parameters
```

> When we call a function with the same name but different parameters values/number, c++ will automatically know which one to use.

overload of operators

```
//foo.cpp
#include <iostream>
using namespace std;

struct lucky_pair {
    int first;
    int second;
};

ostream& operator<<(ostream& os, lucky_pair lp) {
    os << "{" << lp.first << ", " << lp.second << "}";
    return os;
}
```

```
int main() {
    lucky_pair foo;
    foo.first = 7;
    foo.second = 14;
    cout << foo << endl;
}
```

```
+OUTPUT:
{7, 14}
```

> In this code, we precised the behavior or the operator `<<` on our struct `lucky_pair`. Now when we will call a `lucky_pair` using the operator `<<`, it will use our custom method.

inline function and macro

```
//foo.h
#ifndef foo
#define foo

//macros
#define mini1(a, b)   a<b ? a : b
#define mini2(a, b)  (a<b ? a : b)

//inline function
inline int mini3(int a, int b) {
    return a<b? a : b;
}

#endif
```

```
//foo.cpp
#include <iostream>
#include "foo.h"
using namespace std;

int main() {
    cout << "1. " << (mini1(7, 14)) << endl;
    cout << "2. " << (mini1(7, 14)+770) << endl;
    cout << "3. " << (mini1(14, 7)+770) << endl;
    cout << "4. " << (mini2(7, 14)+770) << endl;
    cout << "5. " << (mini3(7, 14)+770) << endl;
    return 0;
}
```

```
+OUTPUT
7
7
777
777
777
```

> - `MACROS`
>
> As we defined `mini1()` as an **macro**, the compilator will replace every `mini1()` call by the line `a<b ? a : b`.
> This means that when we call `mini1(7, 14)+770`, the compilator will traduce this so:
>
> ```
> a<b ? a : b+770
> ```
>
> So we will get `a` in result2, and not `mini1(a, b)+770`, this is the reason why we get the result **7** and not **777**.
> But when we swap **7** and **14** when we call `mini1()`:
>
> ```
> int result3 = mini1(14, 7) + 770;
> ```
>
> We get then `b+770`, that is **777**. *(still **NOT** mini(a, b)+770)*
>
> But in `mini2()` we added parentheses, so when wa call `mini2(7, 14)+770`, the compilator will traduce this so:
>
> ```
> (a<b ? a : b)+770
> ```
>
> This is why in both cases, we will get either `a+770` or `b+770`.
>
> > *It is recommended to use parentheses in macros!*
> >
> > ```
> > #define mini(a, b)   ((a) < (b) ? (a) : (b))
> > //is a good use
> > ```
>
> - `INLINE FUNCTION`
>
> Finally, when we use the **inline** function `mini3()`, the working will be the same as a macro, and all `mini3()` calls will be replaced by the line `a<b ? a : b` excepted that the compilator will smartly know that we don't want to interfer with our previous code, and it will give the *total result + 770*.
> This is the reason why we get the same result as with `mini2()`.

> *Don't use **inline functions** with a complex code.*
> *This is not optimal because a lot of code will be duplicated.*

namespace

```
//A.h
#ifndef A_H
#define A_H
namespace A {
    void foo();
}
#endif
```

```
//B.h
#ifndef B_H
#define B_H
namespace B {
    void foo();
}
#endif
```

```
//A.cpp
#include "A.h"
#include <iostream>

namespace A {
    void foo() {
        std::cout << "I am 'a'! blbl" << std::endl;
    }
}
```

```
//B.cpp
#include "B.h"
#include <iostream>

void B::foo() {
    std::cout << "I am 'b'! blbl" << std::endl;
}
```

```
//foo.cpp
#include <iostream>
#include "A.h"
#include "B.H"
```

```
using namespace std;

int main() {
    //foo();   nope
    A::foo();
    B::foo();
    return 0;
}
```

```
+OUTPUT:
I am 'a'! blbl
I am 'b'! blbl
```

> We defined namespaces `A` and `B` so that we can precisely choose the codes we want to call.
> `std` is a namespace too, and if we don't call `using namespace std;` at the beginning of the code, we will have to write `std::cout` instead of `cout`.
>
> > Note that:
> >
> > - in **A.cpp** we used the syntax `namespace A {...}`
> > - in **B.cpp** we used the syntax `void B::foo() {}...}`
> >
> > Both are working.

class

```
//student.h
#ifndef STUDENT
#define STUDENT

class Student {
    //Attributes
    int id;
    int bloc;

public:
    //Prototypes
    Student(int);
    int get_id() const;
    int get_bloc() const;
    void pass_bloc();
};

#endif
```

```
//student.cpp
#include "student.h"
#include <iostream>
using namespace std;

//Constructor
Student::Student(int id) : id(id), bloc(1) {}

//Getters
int Student::get_id() const {
    return this->id;
}

int Student::get_bloc() const {
    return this->bloc;
}

//Setter
void Student::pass_bloc() {
    if (this->bloc == 3) {
        cout << "Congratulation! You just graduated" << endl;
    } else {
        this->bloc++;
    }
}
```

```
//foo.cpp
#include <iostream>
#include "student.h"
using namespace std;

int main() {
    Student s(56777);
    for(int i=0; i<3; i++) {
        cout << "id=" << s.get_id() << ", bloc=" << s.get_bloc() << endl;
        s.pass_bloc();
    }
}
```

```
+OUTPUT:
id=56777, bloc=1
id=56777, bloc=2
id=56777, bloc=3
Congratulation! You just graduated
```

> This is a basic example of a **class** in c++.

exceptions

```
//foo.cpp
#include <iostream>
#include <stdexcept>
using namespace std;

double divide(int numerator, int denominator) {
    if (denominator == 0) {
        throw runtime_error("Denominator can't be null");
    }
    return ((double)numerator/(double)denominator);
}

int main() {
    std::cout << "777/111 = " << divide(777, 111) << std::endl;
    std::cout << "777/0 = " << divide(777, 0) << std::endl;
    return 0;
}
```

```
+OUTPUT:
777/111 = 7
- terminate called after throwing an instance of std::runtime_error'
-   what():  Denominator can't be null
- Abandon (core dumped)
```