Lesson 6

References

Const qualifier and pointers

Let me remind you that C ++ has a const keyword, which is a variable qualifier. If we create a variable and add the const qualifier to it, that variable cannot be modified.

```
int main () {
    const int x = 3;
    x = 5; /// CE
    return 0;
}
```

The const qualifier can be applied to pointers too! However, in this case there is a question: what exactly will we prohibit modifying if we add the word const: the pointer itself (address), or the value it points to ?! In fact, pointers are of four types:

- A regular pointer. You can modify both its value and the value to which it points.
- Constant pointer. Modifying its value is not possible, but it is possible to modify the value to which it points.
- Constant pointer. Modifying its value is possible, but it is not possible to modify the value to which it points
- A constant pointer to a constant. Modifying neither the value of the pointer nor the value to which it points is allowed.

```
int main () {
    int x;
    const int y = 3;

    int* ptr = &x;
    int* const const_ptr = &x;
    const int* ptr_const = &y;
    const int* const const_ptr_const = &y;

    return 0;
}
```

Const qualifier and pointers

A pointer to a const returns a const when dereferenced.

A pointer to a const cannot be assigned to an normal pointer, but vice versa is allowed.

```c
int main () {
    int x;

    const int* const_ptr = &x;
    int* ptr = &x;

    ptr = const_ptr;
    const_ptr = ptr;

    return 0;
}
```

Conclusion: any operation should not underpromote (*get rid of*) the **const** qualifier.

Swap

Let's imagine the following problem: we want to swap the values of variables.
Moreover, we want to create exactly the function that will do this.

Formally, we want to create a function that will take two variables and swap their values.
Let's try to implement such a function.

```
void swap (int x, int y) {
    int t = x;
    x = y;
    y = t;
}

int main () {
    int x = 2;
    int y = 3;
    swap(x, y);
    std::cout << x << " " << y << std::endl;
    return 0;
}
```

But it doesn't work
=(Why?

```
2 3
```

Swap

When we pass an argument to a function by value (as in the previous example), in fact, we are creating a copy of the variable, and not passing this particular variable! So, in previous implementation of the swap function, we do not work with the initial variables x and y, but with their **copies**.
But how, then, can we access the original variables x and y!? Well, for example, we can pass the addresses of these variables to the function, and not just copy their values! To do this, let's change the signature of the swap function to receive pointers to the original two variables, not copies of them!

```
void swap (int* x, int* y) {
    int t = *x;
    *x = *y;
    *y = t;
}

int main () {
    int x = 2;
    int y = 3;
    swap(&x, &y);
    std::cout << x << " " << y << std::endl;
    return 0;
}
```

And this time everything works as it should!

```
3 2
```

Swap

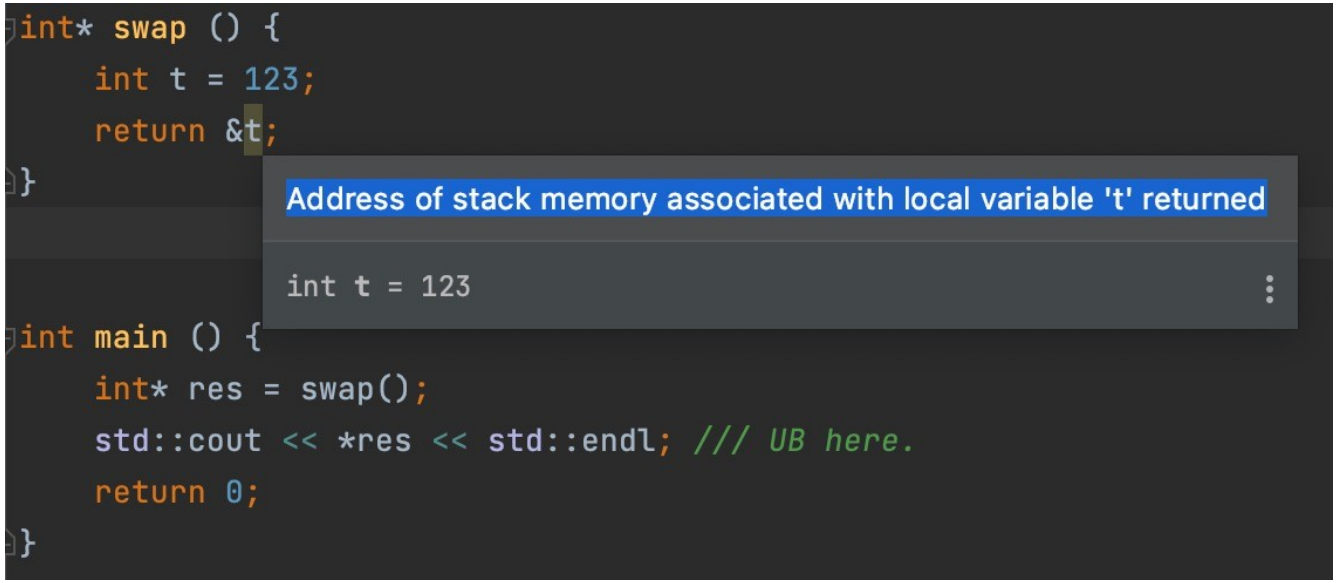
We can improve the code a bit by keeping the **const** rule, which states: everything that can be const must be const.

For example, in this problem, we can make **const** pointers (but NOT pointers to **const**).

```
void swap (int* const x, int* const y) {
    int t = *x;
    *x = *y;
    *y = t;
}
```

References

In fact, before the invention of C ++, the method I showed was the only method for passing initial variables to functions. This method is not very simple, as it requires a lot of dereferencing and address-of operations. In addition, you always need to make sure everything is fine with the pointer. For example, if you accidentally delete an object and then refer to it by the pointer, it'll be an UB:



But with the invention of C ++, everything changed and the concept of references emerged.

References

In Python, for example, if we create a list and create a variable to which we assign the value of this list, we will NOT copy the list, but create a reference to it!

```python
a = [1, 2, 3]
b = a
print(len(a), len(b))

a += [4]
print(len(a), len(b))
```

```
3 3
4 4
```

You will find similar behavior in some cases in Java, JavaScript, TypeScript, and many other programming languages. In them - the concept of a link is built into the language, but you have a way to control whether to create a reference to an object, or make a copy of it.

Similarly in C ++, we can create references to objects (variables). However, in C ++, by default, the copy is getting created, not creating a reference. This is what distinguishes C ++ from Python and the other languages listed above.

```cpp
int main () {
    int a = 2;
    int b = a;
    a += 124;
    /// b is a copy here, not a reference, so
    /// its value hasn't been changed
    std::cout << a << " " << b << std::endl;
    return 0;
}
```

```
126 2
```

References

```
int main () {
    int a = 3;
    int &b = a;
    return 0;
}
```

To create a reference to a variable in C ++, you must use the ampersand character in declaration (not to be confused with the address-of operator!)

& here denotes a modifier of type **int** (as in the case of a pointer). That is, the type of variable **b** is int **&** (not **int**!).

Now, any action with variable b will also change variable a!

```
int main () {
    int a = 3;
    int &b = a;
    b += 123;
    std::cout << a << " " << b << std::endl;
    return 0;
}
```

```
126 126
```

References

```
void swap (int& x, int &y) {
    const int t = x;
    x = y;
    y = t;
}

int main () {
    int a = 2;
    int b = 3;
    swap(a, b);
    return 0;
}
```

References can be passed to functions. In this case, we will NOT create a copy, but pass the variable itself to the function. This way we can change the value of the original variable without using pointers!

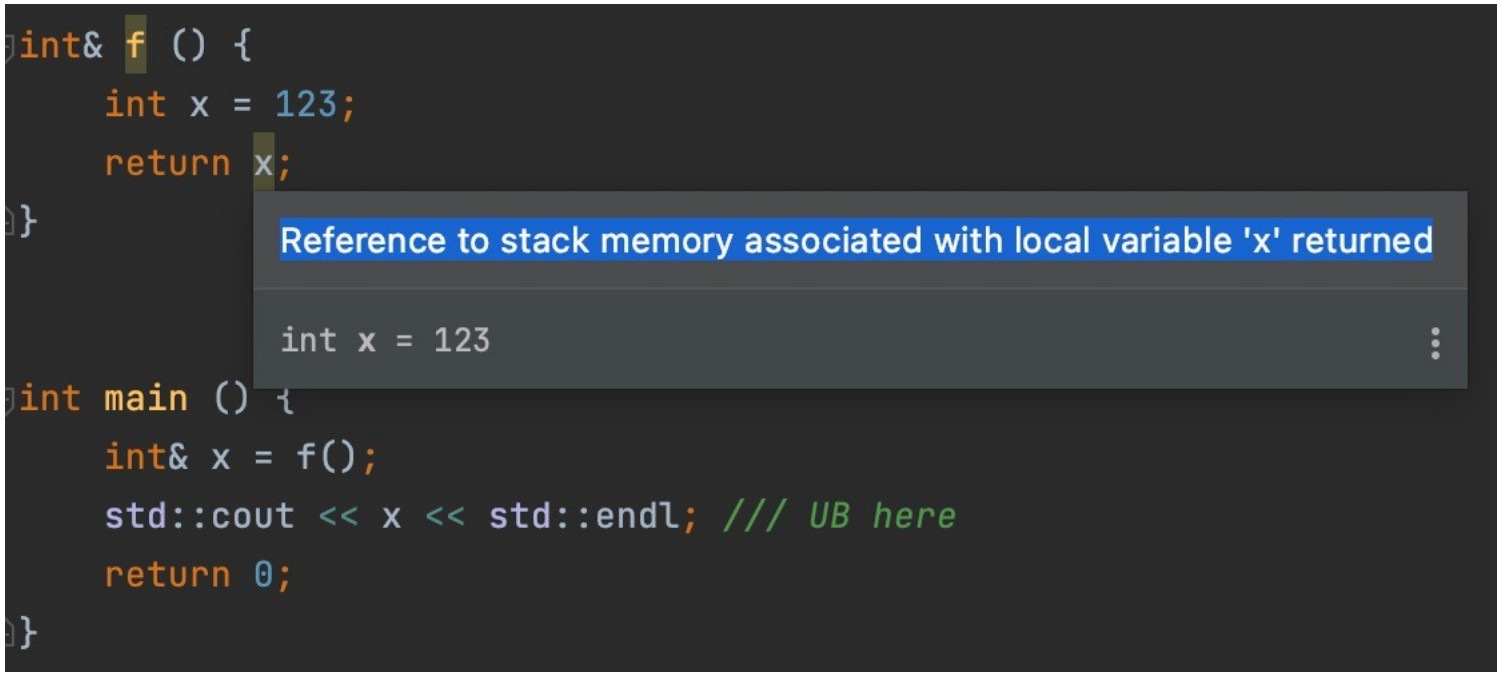
In this case, everything is fine

References

Moreover, references can not only be received by functions, but also returned from them!

However, this trick will be useless to us before we look at classes.

Moreover, there is one very serious mistake associated with this that many programmers make: returning a reference to a local object from a function



Const qualifier and references

Just like with pointers, we can create references to constants.
In this case, the rule from the first slide is also fulfilled.

```
int main () {
    const int& x = 3;
    int y = x;
    int &z = x;
    const int &t = y;
    const int tt = y;
    return 0;
}
```

Another rule: const and reference must always be initialized

```
int main () {
    int& x;
    const int y;
    return 0;
}
```

Another example:

```
int main () {
    const int *p = new int[10]; /// OK
    const int *q = new const int[10]; /// CE
    return 0;
}
```

Casts

There is a so-called C-style cast operator, which allows you to convert variables of different types. But how **exactly** does this transformation work?

In fact, the C-style cast operator isn't quite often used nowadays, because its behavior is not specific enough.

Instead, in new versions of C ++, 4 new operators are used, of which today we will discuss only 3.

```
int main () {
    int a = 123;
    double x = (double)a / 2.;
    std::cout << x << std::endl;
    float z = (float)x + 0.4f;
    std::cout << z << std::endl;

    return 0;
}
```

Casts

- static_cast: This is a compile-time conversion. If static_cast fails to convert the original variable to the correct type, we will get a compilation error.

- reinterpret cast - byte-level conversion. C ++ will simply stop treating the original object as the type it was originally, and will hang a new type on it. This is the lowest-level conversion you can do in C ++. You should be careful with it, and use it only in the most special cases. In the next lesson, I'll show you an example.

```cpp
int main () {
    int a = 123;
    double x = static_cast<double>(a) / 2.;
    std::cout << x << std::endl;
    float z = static_cast<float>(x) + 0.4f;
    std::cout << z << std::endl;

    return 0;
}
```

```cpp
int main () {
    double x = 312.52;
    unsigned long long z = *reinterpret_cast<unsigned long long*>(&x);
    std::cout << x << " " << z << std::endl;

    return 0;
}
```

```
312.52 4644205526174211768
```

Casts

- const_cast is a conversion "through" a constant. This is the only cast that violates the underpromotion rule. You have to be careful with const_cast because it can lead to an error.
- dynamic_cast is a run-time conversion that is applied to polymorphic (virtual) types. We'll be talking about it in the next semester.

```
int main () {
    int x = 312;
    const int& y = x;
    int& z = const_cast<int&>(y);
    return 0;
}
```

```
int main () {
    const int x = 312;
    const int& y = x;
    int& z = const_cast<int&>(y); /// UB
    return 0;
}
```