

## Warm up

### Namespace

Avoid to “using my” to access namespace easier → great Evil !

A same namespace can be splitted in different files, or be nested.

Namespaces are for avoid confusion and modularity.

```
1 namespace my
2 {
3     namespace nested
4     {
5         // We are in my::nested here
6     }
7 }
```

Access with “my::nested”

The C++ library is in *std*

### Ranged-based For-Loops

```
1 auto v = std::vector<int>{1, 2, 4, 8};
2 for (auto i : v)
3     std::cout << i << ',';
```

displays “1,2,4,8” We can have these loops:

```
1 for (auto i : v) // by copy
```

```
1 for (const int& i : v) // by const reference
```

```
1 for (auto&& i : v) // by reference
2 for (auto& i : v) // by reference
```

### Buffers and Pointers

In C++, we prefer dynamic arrays, or other type of std containers

```
1 auto arr = std::vector<int>(n); // parentheses not braces
```

instead of

```
1 int* buf = new int[n];
```

## Polymorphisms

Polymorphism can be coercive, “inclusive”, overloaded and parametric in C++.

A routine is polymorphic if it accepts input with different types.

```
1 bool is_positive(double d) { return d > 0.; }
```

is\_positive accepts int or float.

```
1 class Scalar
2 {
3     virtual bool is_positive() const = 0;
4 };
5
6 class my_int : public Scalar { \*};
7 class my_double : public Scalar { \*};
8
9 bool is_positive(const Scalar& s { return s.is_positive(); }
```

Thanks to inheritance, is\_positive() will work for any sub-classes of Scalar, with transtyping.

```
1 bool is_positive(int i) { return i > 0; }
2 bool is_positive(float f) { return f > 0.f; }
```

Several versions of an operation (is\_positive); signature are different and not ambiguous for the client;

In C++, we can have operator overloading.

```
1 std::cout << s << c << '\n'; // with s and c different types
```

means that several operator<< coexist.

```
1 std::ostream& operator<<(std::ostream, const std::string&);
2 std::ostream& operator<<(std::ostream, const circle&);
```

We also have method overloading :

```
1 class circle : public shape
2 {
3     circle();
4     circle(float, float, float);
5     float x() const;
6     float& x();
7 }
```

but for example `circle::x() const != circle::x()`.

`const` belongs the signature of a function.

We have parametric polymorphism

```
1 template <typename T> // reading: for all type T, we have
2
3 bool is_positive (T t)
4 {
5     return t > 0;
6 }
7
8 void bar()
9 {
10     int i = 1;
11     if (!is_positive(i))
12         return;
13     float f = 1;
14     if (is_positive(f))
15         return;
16 }
```

In template `<typename T> bool foo(T t)`:

- the formal parameter `T` represents a type (**typename**)
- this kind of procedure is a description of a family of procedures
- values of `T` are not known yet
- the call `foo(i)` forces the compiler to set a value for `T`
- a *specific* procedure is then compiled for this value / this specific call

We end up with overloading because the program is transformed by the compiler.

## Parametric polymorphism

### Definition

Formal parameter is a variable attached to an entity and valued at compile-time

C++ entities that can be parameterized are:

- procedures
- methods
- classes

### Templated classes

```
1 template <unsigned n, typename T>
2 class vec
3 {
4 public:
5     using value_type = T;
6 private:
7     value_type data_[n]
8 };
```

If we use `vec<3, float>` somewhere in the program, the compiler gives:

```
1 // This code is not hand written
2 class vec<3, float>
3 {
4 public:
5     using value_type = float;
6 private:
7     float data_[3];
8 };
```

Answers for example (3/4) :

1. bar is simply fill
2. This algorithm works with types of different names (`vec<2, float>` and `std::vector<double>`)
3. This algorithm doesn't work with all types
4. 3 polymorphisms : parametric, coercion and overloading. No inclusion because no sub-classes.

## Duality 00 / Genericity

```
1 while (std::getline(std::cin, s))  
2     // ...
```

works in C++.

In C++, a concept is a list of requirements that a class should fulfill.

## A tour of std containers

### Concepts

Expressivity works when you know the language.

Key idea: learn concepts, their interface (easy), then you know a lot.

The concepts are *refined* (augmented / extended ) from top to bottom when there is a double line.

### Container | object that stores elements

#### Forward container | elements are arranged in a definite order

#### Reversible container | elements are browsable in a reverse order

#### Random Access Container | elements are retrievable without browsing

### Containers

---

vector<T> : dynamic  
array

list<T> : doubly-linked  
list

deque<T> :  
double-ended queue

---

---

stack<T> : LIFO structure

---

`queue<T>` : FIFO structure

---

---

`map<K, V>` : sorted dictionary

---

---

`unordered_map<K, V>` : hash based dictionary

---

```
1 int main()
2 {
3     auto v1 = std::vector<int>(1, 0);
4     //v1.size == 1; { 0 };
5     auto v2 = std::vector<int>{1, 0};
6     //v2.size == 2; { 1, 0 };
7 }
```