Warm up

Namespace

Avoid to "using my" to access namesace easier → great Evil !

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

Namespaces are for avoid confusion and modularity.

```
namespace my
{
  namespace nested
  {
    // We are in my::nested here
  }
}
```

Access with "my::nested"

The C++ library is in *std*

Ranged-based For-Loops

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

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

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

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

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

Buffers and Pointers

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

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

instead of

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

Polymorphisms

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

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

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

is_positive accepts int or float.

```
class Scalar
{
  virtual bool is_positive() const = 0;
};

class my_int : public Scalar { \*};
class my_double : public Scalar { \*};

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.

```
bool is_positive(int i) { return i > 0; }
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.

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

means that several operator<< coexist.

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

We also have method overloading :

```
class circle : public shape
{
  circle();
  circle(float, float, float);
  float x() const;
  float& x();
}
```

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

const belongs the signature of a function.

We have parametric polymorphism

```
template <typename T> // reading: for all type T, we have

bool is_positive (T t)
{
  return t > 0;
}

void bar()
{
  int i = 1;
  if (!is_positive(i))
    return;
  float f = 1;
  if (is_positive(f))
    return;
}
```

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

```
template <unsigned n, typename T>
class vec
{
public:
  using value_type = T;
private:
  value_type data_[n]
};
```

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

```
// This code is not hand written
class vec<3, float>
{
public:
  using value_type = float;
private:
  float data_[3];
};
```

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

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

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 retreviable 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

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