# C++ 06 - Objects are born to be alive

Feb'22

EPITA Research & Development Laboratory (LRDE)

# Recap

# Recap

**OO what we have seen so far**

- How an object comes to life
- How an object dies

**Now, more about the object life's events:**

- Objects can be copied and assigned [1]
- How to enforce coherence for the object's entire lifetime

[1] We will see an extension of this rule in unit 10.

# The special events in an object’s life

# Special functions

| | | |
|---|---|---|
| #1 | `Circle x;,` | Object creation |
| #2 | `auto y = Circle(1,2,3);` | Same but better |
| #3 | `Circle z = x;` | Object copy |
| #4 | `auto z = x;` | Same but better |
| #5 | `{ Circle c; }` | Object destruction |

# Special functions

| | | |
|---|---|---|
| #1 | `Circle x;,` | Object creation |
| #2 | `auto y = Circle(1,2,3);` | Same but better |
| #3 | `Circle z = x;` | Object copy |
| #4 | `auto z = x;` | Same but better |
| #5 | `{ Circle c; }` | Object destruction |
| #6 | `z = y;` | Replace the existing object with a copy of `y` |

# Special functions

| | | |
|---|---|---|
| #1 | `Circle x;,` | Object creation |
| #2 | `auto y = Circle(1,2,3);` | Same but better |
| #3 | `Circle z = x;` | Object copy |
| #4 | `auto z = x;` | Same but better |
| #5 | `{ Circle c; }` | Object destruction |
| #6 | `z = y;` | Replace the existing object with a copy of `y` |

```
class Circle {
  Circle();                                         // #1 Default constructor
  Circle(int x, int y, int r);                      // #2 Custom constructor
  ~Circle();                                        // #5 Destructor
  Circle(const Circle& other);                      // #3, #4 Copy constructor
  Circle& operator= (const Circle& other); // #6 Copy assignment
}
```

# The rule of 3

## The rule of 3

If you customize one of the following operations, you need to customize them all [a]

- Copy constructor
- Copy assignment
- Destructor

[a] We will see an extension of this rule in unit 10.

## The rule of 0

You should strive for classes that do not need to customize any of them.
With a good layout this should be possible in the vast majority of cases.

# Resource handling and RAII - Example 1

```
class TempFile
{
  FILE* handle_;
public:
  TempFile();  // → tmpfile()
  ~TempFile(); // → fclose
  void write(const char* data);
                // → fwrite
};
```

---

```
TempFile b;
{
  TempFile a;
  a.write("bla");
  b = a;
} // a is closed (so is b)
b.write("oups");
```

# Resource handling and RAII - Example 1

```
class TempFile
{
  FILE* handle_;
public:
  TempFile();  // → tmpfile()
  ~TempFile(); // → fclose
  void write(const char* data);
               // → fwrite
};
```

```
TempFile b;
{
  TempFile a;
  a.write("bla");
  b = a;
} // a is closed (so is b)
b.write("oups");
```

```
class TempFile
{
  FILE* handle_;
public:
  TempFile();
  ~TempFile();
  void write(const char* data);
  TempFile(const TempFile&) = delete;
  TempFile& operator=(const TempFile&) = delete;

};
```

**Restrict the behaviour**

By default:

- copy → copy each member variable

Here, customizing = *disallow such insane operations* with `=delete`.

# Resource handling and RAII - Example 2

```
class Buffer
{
  int* handle_;
public:
  Buffer();  // → malloc
  ~Buffer(); // → free
};

{
  Buffer b;
  Buffer a = b;
} // ← ✔ a is freed
  // ← ✖ double free corruption
```

# Resource handling and RAII - Example 2

```cpp
class Buffer
{
  int* handle_;
public:
  Buffer();  // → malloc
  ~Buffer(); // → free
};
```

```cpp
{
  Buffer b;
  Buffer a = b;
} // ← ✔ a is freed
  // ← ✖ double free corruption
```

```cpp
class Buffer
{
  int* handle_;
public:
  Buffer();  // → malloc
  ~Buffer(); // → free
  Buffer(const Buffer&);
  Buffer& operator=(const Buffer&);
              // → malloc + memcpy
};
```

```cpp
{
  Buffer a;
  Buffer b = a; // ← ✔ deep copy a's buffer
} // ← ✔ b is freed
  // ← ✔ a is freed
```

☞ Implementation details will come in the next course.

# C++ objects’ style of life


We have been able to customize the behavior of `a = b` with:

```
Circle& operator=(const Circle&)
```

Like python or Java, C++ allows you to customize operators for your classes:

- Adds “syntactic sugar” to your classes
- Can greatly improve user-experience
- Most common use-cases are stream formatting, assigning, accessing, addition and comparison

# Running example Polynomials

Suppose you have three instances `p1`, `p2` and `p3` of your custom class representing polynomials.

Most certainly you want to * access coefficients

Like this `std::cout << p1[1] << '\n';`

not

```
std::cout << p1.get_coeff(1) << '\n';
```

- Sum them

Like this `auto ps = p1 + p2 + p3;`

not

```
auto ps = p1;
ps.add(p2);
ps.add(p3);
```

- Print them

Like this `std::cout << p1 << '\n';`

# Running example Polynomials

Suppose you have three instances `p1`, `p2` and `p3` of your custom class representing polynomials.

Most certainly you want to * access coefficients

Like this `std::cout << p1[1] << '\n';`

not

```
std::cout << p1.get_coeff(1) << '\n';
```

- Sum them

Like this `auto ps = p1 + p2 + p3;`

not

```
auto ps = p1;
ps.add(p2);
ps.add(p3);
```

- Print them

Like this `std::cout << p1 << '\n';`

**⇒Define your own operators!**

# `operator[]`: accessing an element

`operator[](size_t idx)` is usually used to get an element from some sort of list.

poly.hpp

```cpp
class poly{
  using cvec = std::vector<float>;
  cvec coeffs_;
  float operator[](size_t idx) const;
  float& operator[](size_t idx);
}
```

main.cpp (usage)

```cpp
auto p = poly();
p[10] = 11.f;
for (size_t i = 0; i < 15; ++i)
  std::cout << i << ':' << p[i] << '\n';
```

poly.cpp

```cpp
float poly::operator[](size_t idx) const{
  if (idx < coeffs_.size())
    return coeffs_[idx];
  else
    return 0.f;
}

float& poly::operator[](size_t idx){
  coeffs_.resize(idx+1, 0.f);
  return coeffs_[idx];
}
```

Ensure coherent access of coefficients.

# `operator+` for polynomials 1

poly.hpp

```
class poly {
private:
  using cvec = std::vector<float>;
  cvec coeffs_;
  cvec add_coeffs_(const poly& c1,
                   const poly& c2);
public:
  poly() = default;
  poly(const poly& o) = default;
  poly(const cvec& c);
  poly operator+(const poly& o);
}
```

poly.cpp

```
cvec poly::add_coeffs_(const poly& p1,
                       const poly& p2)
{
  auto sz = std::max(p1.coeffs_.size(),
                     p2.coeffs_.size());
  auto cn = cvec{sz};
  for (size_t i = 0; i < sz; ++i)
    cn[i] = c1[i] + c2[i];
  return cn;
}

poly poly::operator+(const poly& o){
  return poly(add_coeffs_(*this, o));
};
```

# `operator+` for polynomials 2

Note that here, we have defined the operator as a member function.

We can define them as a free function:

```
// poly.hpp
poly operator+(const poly& p1, const poly& p2);
```

In this context `coeffs_` is however private.

The free function version is more flexible, as one can define `poly operator+(float v, const poly& p);` for instance. This is not possible using member functions.

# Formatting

- We would like to be able to write `std::cout << p;`, but how?
- `std::cout` is defined in `iostream` and has type `std::ostream`.
- We can not modify this class and add the member function `std::ostream& operator<<(const poly& p);`.

# Formatting

- We would like to be able to write `std::cout << p;`, but how?
- `std::cout` is defined in `iostream` and has type `std::ostream`.
- We can not modify this class and add the member function `std::ostream& operator<<(const poly& p);`.

**Define it as a free function**

# `operator<<`: Formatted output

**poly.hpp**

```
std::ostream& operator<<(std::ostream& os, const poly& p);
```

This has to be declared in the **same namespace** as `poly` !

**poly.cpp**

```
std::ostream& operator<<(std::ostream& os, const poly& p){
  const auto m = p.max_deg();
  os << p[0] << (m>0) ? " + " : "\n";
  for (size_t i = 1; i < m; ++i)
    os << p[i] << "x**" << i << " + ";
  if (m > 0)
    os << p[m] << "x**" << m << '\n';
  return os;
}
```

Returning the stream allows to “chain” calls.

Using `std::ostream` work for `std::cout` but also for `std::cerr`, writeable files etc.

# operator>>: Formatted input

Similarly, we can define formatted input.

Suppose we store the coefficients of a polynomial separated by a whitespace and the end of a polynomial is indicated by an ‘S’.

poly.cpp

```
std::istream& operator>>(std::istream& is, poly& p){
  float  f;
  size_t idx = 0;
  p.clear(); // A "new" polynomial
  while(is.peek() != 'S'){
    is >> f;
    p[idx++] = f;
  }
  is.get(); // Consume 'S'
  return is;
}
```

# `operator>>`: Formatted input

Reading a list of polynomials given as argument:

main.cpp argv
1
=”1 2 3S 5 6S”

```cpp
// Convert char-array to istream
auto is   = std::istringstream{argv[1]};
auto pvec = std::vector<poly>{};

while (is){
  poly p;
  is >> p;
  pvec.push_back(p);
}
```

# Conclusion

# Advantage of operators

As stated before, defining operators does not add expressivity but

- `if (o1 >= o2)` better than if `if (o1.greater_or_equal(o2))`
- Makes custom classes behave like native types
- Greatly facilitates *io* operations
- Gives your classes an algebraic touch, they behave like a *group*, which can be very intuitive to use