C++
Classes

Goals

- Understand operator overloading
- Understand how to correctly manage destruction and copy

Quick reading

- Read and try to grasp the main ideas

Read

- Read and understand the explained concepts

Study

- Read, understand and remember the concepts, the rules and the principles.

Don't be afraid to try (compile, execute, modify, debug) the proposed examples!

Overloaded operators

The parameteris a reference to the second operand

```
class Fraction {
    public:
        Fraction() : Fraction{0,1} {};
        Fraction(int numerator,
                 int denominator=1)
        : m_numerator{numerator},
          m_denominator{denominator}
{};

        ~Fraction() {};

        int num() const {
         return m_numerator; }
        void num(int numerator {
         m_numerator = numerator; };
        int den() const {
         return m_denominator; }
        void den(int denominator) {
          m_denominator = denominator;
        }
```

```
Fraction& operator += (const Fraction& f) {
  int temp_numerator { f.m_numerator *
                        m_denominator };
  m_denominator *= f.m_denominator;
  m_numerator *= f.m_denominator;
  m_numerator += temp_numerator;
  return *this;
}
```

We return a reference to the object itself

```
Fraction& operator -= (const Fraction& f) {
 int temp_numerator { f.m_numerator *
                       m_denominator };
 m_denominator *= f.m_denominator;
 m_numerator *= f.m_denominator;
 m_numerator -= temp_numerator;
 return *this;
}

private:
        int m_numerator, m_denominator;
};
```

Operators as class methods

Operator overloading

```
ostream& operator << (ostream& o, const Fraction& f)
{
    o << f.num() << "/" << f.den();
    return o;
}
```

First operand (left)

```
Fraction operator+(Fraction a, const Fraction& b)
{
    return a += b;
}
```

Second operand (right)

```
Fraction operator-(Fraction a, const Fraction& b)
{
    return a -= b;
}

Fraction operator-(const Fraction& a)
{
    return { - a.num(), a.den() };
}
```


Operators as external functions

Notes

- Only valid operators can be overloaded (i.e. we can't "invent" new operators that aren't already part of the language)
  - For example, we can't overload **$** or **“**
- Unary and binary operators can be overloaded, with the following exceptions
  - Conditional test **?**
  - Scope resolution operator **::**
  - Member access operator **.**

Overloading new/delete

```
#include <iostream>
#include <cstdlib>

using namespace std;

class Example
{
public:
        void* operator new(size_t);
        void operator delete(void*);
};


void* Example::operator new(size_t size)
{
    cout << "Allocation" << endl;
    return malloc(size);
}

void Example::operator delete(void* arg)
{
       cout << "Deallocation" << endl;
       free(arg);
}
```

```
int main(void)
{
        Example* e = new Example();

        delete e;

        return 0;
}
```

Digression: object conversion

- What happens if we try to assign a variable some value of a different type?

```
Fraction f {2,3};
f = 5;
```

Conversion constructor

- In C++ a constructor with only one argument (of type different from the class itself) is considered a **conversion constructor**
  - A conversion constructor can be called implicitly by the compiler to perform the necessary type conversions

Example

```
Fraction(int numerator, int denominator=1)
               : m_numerator{numerator},
                 m_denominator{denominator} {
};
```

Conversion constructor (from int to Fraction)

```
Fraction f {2,3};
f = 5;
```

```
Fraction {5}
```

```
void process(Fraction f)
{
        // ...
}
```

```
Fraction {42}
```

```
process(42);
```

explicit

- With the **explicit** keyword (before the constructor declaration) we can prevent automatic conversions

```
Fraction(int precision)
            : m_precision{precision} {
        // ...
};
```

```
explicit Fraction(int precision)
            : m_precision{precision} {
        // ...
};
```

```
Fraction f {2,3};
f = 5;
```
Error!

Example

```
#include <iostream>

class MyClass {
    public:

        MyClass(int m) {
            arr = new int[m];
            // … do something …
            delete[] arr;
        }

};

void function(MyClass t) {

}

int main(void) {
    MyClass m{5};
    int z{4};

    // My mistake...
    function(z);
    // Error! (not reported by the
compiler)
}
```

```
#include <iostream>

class MyArray {
    public:

        explicit MyClass(int m) {
            arr = new int[m];
            // … do something …
            delete[] arr;
        }
};

void function(MyClass t) {

}

int main(void) {
    MyClass m{5};
    int z{4};

    // My mistake...
    function(z);
    // No conversion done! Compiler
error!
}
```

Conversion operator

- Conversion operators can be defined to convert a type into another:

```
operator double () {
    return (double) m_numerator / m_denominator;
}


void myfun(double d)
{
    // ...
}

Fraction x { 1, 2};
myfun(x);        0.5
```

Call resolution

- The compiler follows this order when resolving a call:

1) If there's an exact match, use it (same name, same signature)
2) Otherwise promote simple types
3) Otherwise try with conversion constructors and coversion operators
4) Otherwise try with variadic functions
5) Otherwise give up and return an error

Operator overloading

- Operator as class member

```
Fraction operator+(const
                   Fraction& b)
{
    Fraction temp = *this;
    return temp += b;
}
```

```
Fraction x { 1, 2};
Fraction y;

y = x + 4;
y = 4 + x; // Error!
```

Integer (4) has no method for adding a Fraction

- Operator as an external function

```
Fraction operator+(Fraction a,
const Fraction& b)
{
    return a += b;
}
```

```
Fraction x { 1, 2};
Fraction y;

y = x + 4;
y = 4 + x;
```

Integer (4) is converted to a Fraction and two fractions can be summed

Postfix, prefix

- **Prefix increment** (++a, --a)

```
Fraction& operator++() {
        return *this += 1;
}
```

As member

```
Fraction& operator--(Fraction& f)
{
        return f -= 1;
}
```

As function

- **postfix increment** (a++, a--)

```
Fraction operator++(int) {
        Fraction x{*this};
        *this += 1;
        return x;
}
```

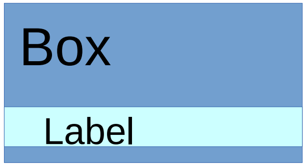
As member

```
Fraction operator--(Fraction& f, int)
{
        Fraction x{f};
        f -= 1;
        return x;
}
```

As function

Composition / Sub-objects

- An object can be composed of **sub-objects**
  - Sub-object != pointers/reference to other objects




```
#include <iostream>

using namespace std;

struct Label {
  Label(string t) : m_text{t} {
    cout << "Creating Label "
         << m_text << endl;
  }

  string m_text;
};
```

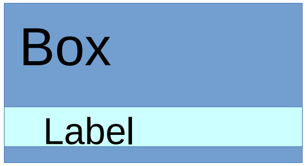
```
class Box {
       public:
         Box(string e) : m_label{e}
         {
           cout << "Creating Box" << endl;
         }

       private:
         Label m_label;
};
```

Destroying an object

- The memory used by an object is freed...
  - Explicitly, with `delete`
  - Implicitly, for object on the stack, when exiting the scope
  - Implicitly, for class members, when the object is destroyed
  - Implicitly, for **`static`** members, when the program ends
- If inside the class there are explicitly allocated resources (for example, heap allocated objects) we need to free them
  - Remember: there is no garbage collector!

Composition / Sub-objects

- An object can be composed of **sub-objects**
  - Sub-object != pointers/reference to other objects

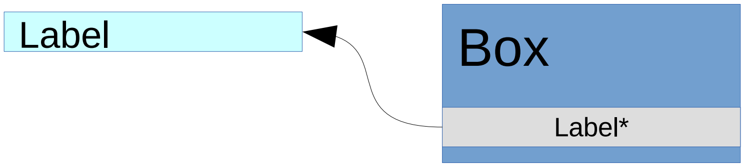



```
#include <iostream>

using namespace std;

struct Label {
  Label(string t) : m_text{t} {
    cout << "Creating Label "
         << m_text << endl;
  }

  string m_text;
};
```

```
class Box {
     public:
       Box(string e) : m_label{e}
       {
         cout << "Creating Box" << endl;
       }

     private:
       Label m_label;
};
```

When are they deleted?

Composition with pointers to heap-allocated objects




```
#include <iostream>

using namespace std;

struct Label {
  Label(string t) : m_text{t} {
    cout << "Creating Label "
         << m_text << endl;
  }

  string m_text;
};
```

```
class Box {
  public:
    Box(string e)
       : m_label{new Label{e}} {
        cout << "Creating Box" << endl;
    }

  private:
    Label* m_label;
};
```

When is it deleted?

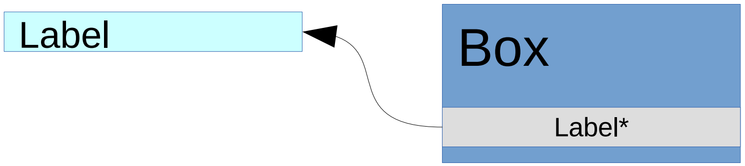
Destructors

- Before freeing memory we can execute some actions, for example to release additional resources that might be linked to the object
  - This method is called **destructor**

    `~ClassName()`

    - **`ClassName::~ClassName() { }`**
- In Java, even though memory is freed by the garbage collector, we can define a `finalize()` to achieve something similar

Composition with pointers to heap-allocated objects




```
#include <iostream>

using namespace std;

struct Label {
  Label(string t) : m_text{t} {
    cout << "Creating Label "
         << m_text << endl;
  }

  string m_text;
};
```

```
class Box {
  public:
    Box(string e)
       : m_label{new Label{e}} {
        cout << "Creating Box" << endl;
    }
    ~Box() {
      cout << "Destroying Box" << endl;
      delete m_label;
    }
  private:
    Label* m_label;
};
```

The RAII pattern

- To simplify resource management it is advisable to adopt the **RAII** (Resource Acquisition Is Initialization) pattern
  - Acquire resources (for example, allocate memory) in the constructor
  - Release acquired resources in the destructor

Copy

- When dealing with pointers, copy becomes non-trivial
- When does an object get copied?
  - During **pass by value**

```
void myfun(Box x) { ... }

Box a;
myfun(a);
```

  - During **assignments**

```
Box a;
Box b;

b = a;
```

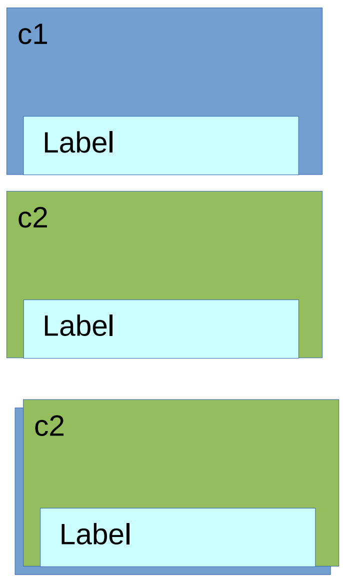
Copy

Without pointers

Box c1;

Box c2;

c1 = c2;

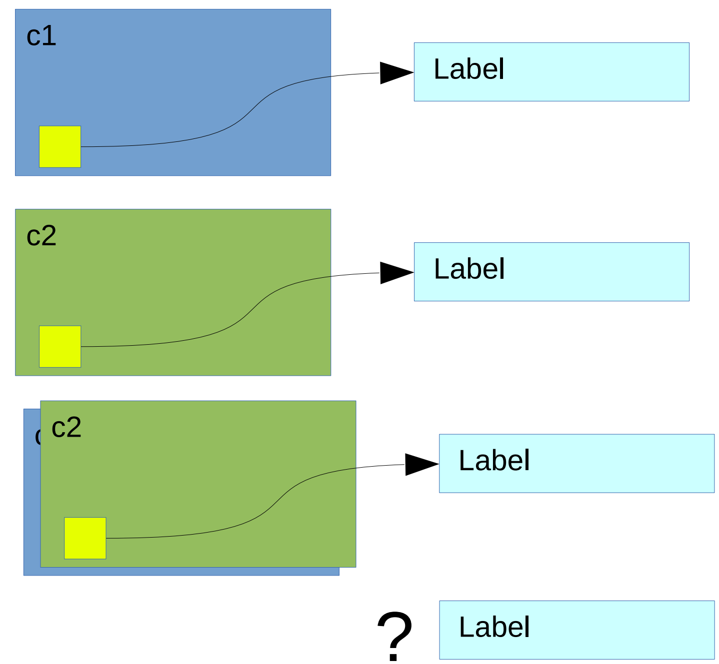


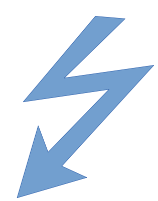
With pointers

Box c1;

Box c2;

c1 = c2;





Imminent crash

Dealing with copies

- We can control how a copy is created by defining a **copy constructor**. The signature is:

```
ClassName(const ClassName& other);
```

The origin object does not get modified

Rererence (we can't pass by value!)

What should a copy constructor do?

- It should initialize its members by copying the current state of the referenced object (passed as parameter)
  - Since objects are instances of the same class, the copy constructor can freely access **`private`** members of the referenced object
- The default copy constructor (provided by the compiler for each class) just copies the whole contents of the structure

Copy assigment operator

- When we assign a new value to an object, the **copy assignment operator** is used

```
ClassName& operator=
                (const ClassName& other);
```

- The default copy assignment operator (provided by the compiler for each class) just copies the whole contents of the structure
- Note: the operator can be overloaded (for example, to deal with different types of assignments) but only this signature is called copy assignment operator

Example

Copying label relies on the default copy constructor

```
Box(const Box& o)
        : m_label{
            new Label{*o.m_label}}
{
}
```

Copy constructor

```
Box& operator=(const Box& o)
{
        *m_label = *o.m_label;
        return *this;
}
```

Copy assignment operator

We assume that label is able to correctly copy itself

Rule of three

- The “**rule of three**” says that if a class **requires*** one of the following methods, then all of them might be required:
  - Destructor
  - Copy constructor
  - Copy assignment operator

* you can still implement a destructor if you want to do something on destruction event if its not required. In that case the rule of three might not apply (example follows in the next slides)

Example

```
class Fraction {
    public:
        Fraction() : m_numerator{0},
                m_denominator{1} {};
        Fraction(int numerator, int denominator=1)
                        : m_numerator{numerator},
                    m_denominator{denominator} {};

        ~Fraction() {};

        // ...
```

Destructor (if not specified a default one will be created)

… we would like to count the number of currently allocated objects of Type fraction...

Static members

- Static members (declared using the **static** keyword) are shared between all instances of a class

```
#include <iostream>
using namespace std;

class Fraction {
    public:
        Fraction() { instances++; }

        ~Fraction() { instances--; };

        static int getinstances() {
          return instances;
        }
    private:
        static int instances;
};
```

```
int Fraction::instances = 0;
```

Inizialization

```
int main()
{
       Fraction x, y;
       cout << Fraction::getinstances()
<<
       endl;
       {
       Fraction z;
       cout << Fraction::getinstances()
<<
       endl;
       }
       cout << Fraction::getinstances()
<<
       endl;
}
```