Definition of classes in C++

Programação (L.EIC009)

Eduardo R. B. Marques, DCC/FCUP

Previously ...

A class named `some_class` is a datatype declared using the `class` keyword.

```
class some_class { ... };
```

We will now see how a class is defined, covering the following notions:

- class declaration;
- `public` and `private` declarations;
- member fields, member functions, constructors and destructors; and
- `static` (global) class fields and functions.

Other topics not covered here: template classes, operator overloading and class inheritance.

Class declaration

```
namespace some_namespace {
  class some_class {
    ...
  };
}
```

A class is declared using the `class` keyword. Like other definitions (functions, structs, etc) a class may be declared in the scope of a namespace, e.g. `some_namespace` as shown above.

A class is typically declared in a header file, e.g. `some_class.hpp`. As we will see later, the header file may just declare the class while its actual implementation resides in a separate file, e.g. `some_class.cpp`. For now we will consider the joint declaration and implementation in a single header file.

Class declaration (cont.)

```
#ifndef __some_class_hpp__ // header guard section -->
#define __some_class_hpp__
namespace some_namespace {
  class some_class {
    ...
  };
}
#endif // <--
```

Header guards are a standard practice for C/C++ header files, in particular those containing class declarations. They prevent repeated includes of the same file to have any effect.

```
#include <some_class.hpp>
#include <some_class.hpp> // no effect !
```

Examples

Two examples - the code is available online at GitHub:

`fraction.hpp` (1) :

```
#ifndef __fraction_hpp__
#define __fraction_hpp__
namespace leic {
  class fraction { ... };
} ...
#endif
```

`polynomial.hpp` (2):

```
#ifndef __polynomial_hpp__
#define __polynomial_hpp__
namespace leic {
  class polynomial {  ... };
} ...
#endif
```

Examples (cont.)

```
class fraction {
  ...
};
```

`fraction`: represents fractions $\frac{n}{d}$ (rational number) in irreducible form, where $n \in \mathbb{Z}$ and $d \in \mathbb{Z}^+$.

```
class polynomial {
  ...
};
```

`polynomial`: represents polynomials $f_0 + f_1x^1 + f_2x^2 + ... + f_nx^n$ where $f_1, ..., f_n$ are `fraction` objects.

public and private visibility

A class may contain sections of `public` and `private` declarations:

```
class some_class {
public:
  ... // public declarations
private:
  ... // private declarations
};
```

`public` and `private` are keywords called **access specifiers**.

Declarations in a `public` section are accessible outside the class without restrictions. Declarations in a `private` section are accessible by the class only.

Two complementary mechanisms to be seen later: (1) `friend` classes and functions may access private declarations too; and (2) `protected` is another type of access specifier (not covered in these slides), allowing declarations to be acessed by derived classes through inheritance.

Member fields

```
class fraction {
  ...
private:
  int num, den;
  ...
};
class polynomial {
  ...
private:
  vector<fraction> coeff;
  ...
};
```

A class may declare **member fields**, also called instance fields. Each object that instantiates the class will have values for each of the member fields.

Above: `num` and `den` defined for `fraction`; `coeffs` for `polynomial`.

Member fields (cont.)

```
class fraction {
  ...
private:
  int num, den;
  ...
};
```

Fields work as in the case of `struct` types. However, unlike `structs`, they are typically `private`, hence cannot usually be manipulated directly by code outside the class . The purpose of a `private` visibility is to “hide” the internal state of objects, and allow their manipulation only indirectly through `public` constructors and member functions.

Member fields (cont.)

Access to `private` member fields from outside the class are not allowed by the compiler:

```
class fraction {
  ...
private:
  int num, den;
  ...
};
...
int main() {
  fraction f;
  f.num = 123; // NOT ALLOWED!
}
```

```
  error: ‘int leic::fraction::num’ is private
  within this context
  ...  f.num = 123;
```

Constructors and member functions

```
class fraction {
public:
  // constructors
  fraction(int n, int d=1) { ... }
  fraction() {  ... }
  fraction(const fraction& f) { ... }
  ...
private:
  int num, den;
  ...
};
```

Constructors allow an object to be created and initialized. They are typically `public`.

```
fraction a(1,2), b(3), c, d(c);
```

Constructors

```
fraction(int n, int d=1) {
  num = n;
  den = d;
  reduce(); // convert to irreducible form
            // (check the code for details)
}
fraction() {
  num = 0;
  den = 1;
}
fraction(const fraction& f) {
  num = f.num;
  den = f.den;
}
```

Constructors define the initial state of an object by assigning values to member fields.

Constructors (cont.)

Field initialisation in constructors can be expressed alternatively through **member initializer lists**. The generic syntax is as follows:

```
class_name(...) : field_1(value_1), ..., field_n(value_n) {
  // constructor body (may be empty!)
  ...
}
```

For `fraction` the previous constructors can be rewritten using initializer lists as follows:

```
fraction(int n, int d)  : num(n), den(d) { reduce(); }
fraction() : num(0), den(1) { }
fraction(const fraction& f) : num(f.num), den(f.den) { }
```

Constructors (cont.)

```
fraction(int n, int d=1)  : num(n), den(d) { reduce();}

// Delegating constructors call the first one -->
fraction() : fraction(0) { }
fraction(const fraction& f) : fraction(f.num, f.den) { }
```

Member initializer lists can also be used to call other constructors, a mechanism called **constructor delegation**. This is shown above for `fraction`.

The purpose is usually that one of the constructors provides the base functionality for initialisation, while other constructors that call it, called **delegating constructors**, just offer the conveniency of distinct types of object construction.

Constructors and member functions

```
class fraction {
public:
  // constructors
  ...
  // member functions
  int numerator() const { ... }
  int denominator() const { ... }
  void add(const fraction& f) { ... }
  void mul(const fraction& f) { ... }
  ...
};
```

After construction, member functions can be invoked on an object.

```
fraction a(1,2), b(2,3), c(1);
a.mul(b); // 1/2 * 2/3 = 1/3
a.add(c); // 1/3 + 1 = 4/3
cout << a.numerator() << ' '
     << a.denominator() << '\n'; // -> 4 3
```

Member functions

```
int numerator() const { return num; }
int denominator() const { return den; }
void add(const fraction& f) {
  num = num * f.den + f.num * den;
  den = den * f.den;
  reduce();
}
void mul(const fraction& f) {
  num = num * f.num;
  den = den * f.den;
  reduce();
}
```

Member functions allow an objet state to be queried or modified. Above: `numerator()` and `denominator()` just return the values of member fields without changing them; `add()` and `mul()` change the member fields' values.

Separate declaration and implementation

Functions can be declared within a class and their implementation may appear separately in the same source code file or a different one. For example, one could have `fraction` defined as follows:

```cpp
class fraction { // Only declarations!
public:
  fraction(int num, int den=1);
  ...
  int numerator() const;
  int denominator() const;
  ...
};
// Implementation
fraction::fraction(int n, int d) : num(n), den(d) { }
int fraction::numerator() const { return num; }
int fraction::denominator() const { return den; }
...
```

“What is `this` ?”

`this` designates a pointer to the object instance in context in a constructor or other member function.

For instance, within `fraction` ...

```
  int numerator() const { return this->num; }
```

or

```
  int numerator() const { return (*this).num; }
```

are both equivalent to

```
  int numerator() const { return num; }
```

The use of `this` is relevant only when there is a need for disambiguation between the names of fields and other variables, and in some other special cases.

const member functions

```
int numerator()   const { return num; }
int denominator() const { return den; }
```

The const modifier indicates that the member function does not change any member fields.

Changes to a member field within a const member function will result in compilation errors.

```
// Alternative definition ...
int numerator() const {
  num++; // NOT ALLOWED BY THE COMPILER!
  return num;
}
```

```
error: increment of member ‘leic::fraction::num’ in
read-only object
```

Class composition

```
class polynomial {
  ...
private:
  // Fraction coefficients.
  // Position i in the vector refers to the
  // coefficient of degree i.
  std::vector<fraction> coeff;
  ...
};
```

Class composition happens when one class is defined through member fields that are objects of other classes.

Above: `polynomial` is defined through the use of a `std::vector` field. Moreover, `std::vector` (a template class) is parameterised in this case with the `fraction` class type.

Class composition (cont.)

```
class polynomial {
public:
  polynomial(const std::vector<fraction>& c)
  : coeffs(c) {
    reduce();
  }
  ...
private:
  std::vector<fraction> coeffs;
  ...
};
```

The initialisation of `coeffs` through `coeffs(c)` in the member initialiser list means that the `vector` copy constructor is called.

Member initializer lists are specially relevant in the case of object member fields, since they ensure an adequate constructor invocation.

Class composition (cont.)

```cpp
class polynomial {
public:
  // a few public member functions
  int degree() const {
    return coeffs.size() - 1;
  }
  void add(const polynomial& p) {
    size_t n = std::min(coeffs.size(), p.coeffs.size());
    for (size_t i = 0; i < n; i++)
      coeffs[i].add(p.coeffs[i]);
    for (size_t i = coeffs.size(); i < p.coeffs.size(); i++)
      coeffs.push_back(p.coeffs[i]);
    reduce();
  }
  ...
```

Through class composition, we can combine the functionality of several classes for meaningful code.

Destructors

```
class fraction {
public:
  ...
  ~fraction() { }
  ...
};
```

A **destructor** frees up resources used by an object when the lifetime of an object ends.

Above: `~fraction()` has an empty body, hence it can be alternatively omitted; an empty destructor is defined by default if the implementation omitted.

Destructors (cont.)

```
class polynomial {
public:
  ...
  ~polynomial() { }
private:
  std::vector<fraction> coeffs;
  ...
};
```

As in `fraction`, `polynomial` has an empty destructor, thus again the destructor could be omitted.

In any case, `~vector()` (the destructor of `vector`) is invoked automatically for `coeffs` in the `polynomial` class. This always happens for object fields in a class, when a class destructor is called.

Destructors (cont.)

Now suppose `polynomial` used a dynamically allocated array for the coefficients instead of a `vector` object ...

In this case, the memory of the array would have to be released explicitly by `~polynomial()`.

```
class polynomial {
public:
  polynomial(const vector<fraction>& c) {
    coeffs = new fraction[c.size()];
    for (size_t i = 0; i < c.size(); i++) coeffs[i] = c[i];
  }
  ~polynomial() {
    delete [] coeffs;
  }
  ...
private:
  fraction* coeffs;
};
```

static class members

```
class fraction {
  ...
  // class function (static)
  static int gcd(int a, int b) {
    while (b != 0) { int tmp = a; a = b; b = tmp % b; }
    return a;
  }
  // member function (non-static)
  void reduce() {
    int g = gcd(num, den);  num /= g; den /= g;
    if (den < 0) { num = -num; den = -den; }
  } ...
};
```

The `static` modifier indicates that a function is global to the class, called a **class function**. When the function is invoked, there is no association to any particular object.

`static` class members (cont.)

We can also have **class fields** declared with `static`, e.g.,

```
class fraction {
  ...
  static const fraction ZERO;
  ...
};

// outside the class declaration
const fraction fraction::ZERO(0, 1);
```

These fields are global to the class. Quite often, `static` class fields are used to represent constants as in the example above for the `ZERO` fraction object; note that `ZERO` also has the `const` modifier. A "non-linear" aspect of C++ is that the initialisation of `static` fields must be stated outside the class declaration.

`static` class members (cont.)

Given that `ZERO` is declared as `const`, it is a class constant. It refers to an immutable object that cannot have its internal state modified after construction.

For instance, it is valid to invoke `const` member functions over `ZERO` ...

```
cout << fraction::ZERO.numerator() << ' '
     << fraction::ZERO.denominator() << '\n';
```

but not state-changing member functions ...

```
fraction a(1,2);
fraction::ZERO.add(a); // NOT ALLOWED!
```

```
    error: passing ‘const leic::fraction’ as ‘this’ argument
    discards qualifiers ...
    ...    fraction::ZERO.add(a);
```

`static` class members (cont.)

The scoping operator `::` must be used to access `static` members outside the class

`class_name::declaration_name`

e.g., `Fraction::ZERO` as in the previous slide, or `Fraction::gcd` for the `gcd` function.

friend declarations

```
class fraction {
  ...
  friend class polynomial;
  friend std::ostream&
  operator<<(std::ostream& s, fraction& f);
};
```

`friend` declarations in a class identify other external classes or functions with access to private declarations in the class. In the example above, `friend` access is granted to the `polynomial` class and an implementation of the `<<` operator (an example of operator overloading, a topic to be covered in detail later).

```
std::ostream&
operator<<(std::ostream& out, const fraction& f) {
  out << f.num;   // direct access to fields is allowed!
  if (f.den != 1) out << '/' << f.den;
  return out;
}
```