Classes and objects - an introduction

Programação (L.EIC009)

Eduardo R. B. Marques, DCC/FCUP

Outline

- Using classes and objects - fundaments
  - The lifecycle of an object
  - Constructors, destructor, member functions
  - Illustration using `std::string`
- Examples
  - Use of `std::string` in more detail
  - Use of `std::vector`
  - Streams for I/O and text parsing

Using classes and objects - fundaments

Classes and objects

A class `SomeClass` is a data type declared with the keyword `class`:

```
class SomeClass { ... };
```

We will see later how classes are *defined*, but first we will understand how they are *used*.

An **object** in an instance of a class. The lifecycle of an object goes through the following stages:

- When an object variable is declared, it is initialised through a special function, a **constructor**;
- We can use then use the class functionality for the object, typically through the invocation of **member functions**.
- A special function, called the **class destructor** is automatically invoked when an object goes out of scope.

Example - std::string

Class std::string, defined by header <string>, can be used to represent strings. Internally, a string object is implemented using a char array that grows dynamically in size when necessary.

```
// A few of the constructors
string (const string& str);
string (const char* s);
string (const char* s, size_t n);
. . .
// Destructor
~string();
// Some of the member functions
std::string& append(const std::string);
const char& at (size_t pos) const;
size_t length() const;
void push_back (char c);
. . .
```

Constructors

A constructor of `SomeClass` is a function with the same name `SomeClass` that can have several parameters.

```
SomeClass(); // default constructor
SomeClass(const SomeClass& other); // copy constructor
SomeClass(int a, int b); // another constructor
```

There can be several constructors. In particular, a constructor without parameters is called the **default constructor**, and a constructor that takes parameter a single `SomeClass` value or (more usually a `const`) reference parameter is called the **copy constructor**.

Use of constructors

For:

```
SomeClass();
SomeClass(const SomeClass& other);
SomeClass(int a, int b);
```

a variable for an object of type `SomeClass` is declared with the following generic syntax

```
SomeClass var1 (arg_1, . . . , arg_n);
```

The arguments are matched with a corresponding constructor, e.g.,

```
SomeClass v1; // invokes default constructor
SomeClass v2(v1);  // copy constructor
SomeClass v3 = v1; // copy constructor (variation)
SomeClass v4 { v1 }; // copy constructor (variation)
SomeClass v5(1, 2); // another constructor
```

Example - constructors in `std::string`

`std::string` defines several constructors, for instance:

```
// Default constructor (empty string)
string();
// Copy constructor.
string (const string& str);
// Constructor from C string
string (const char* s);
// Constructor from C string up to n bytes
string (const char* s, size_t n);
. . .
```

Example - constructors in `std::string`

A few `std::string` variables:

```
#include <string>
using namespace std;
. . .
string a; // empty string (default constructor)
string b("ABC"); // from C string
string c = "DEF"; // from C string (syntactic alternative)
string d("IJKL", 3); // from C string, up to 3 chars
string e(d); // use of copy constructor
string f = d; // use of copy constructor also
```

Note: `SomeClass a = v;` corresponds to the invocation of a single-argument constructor, i.e., `SomeClass obj(v);`.

Member functions

We can invoke **member functions** over an object using the `.` operator:

```
SomeClass obj(. . .);
. . .
obj.member_function_name(. . . member function arguments . .
```

A member function can:

- return or derive information related to the internal object state **without changing it** - typically these functions are declared as `const`;
- **change** the internal state of the object - not declared as `const`

Example - `std::string` member functions

A few member functions in `std::string`:

```
int length() const;
const char& at(size_t pos) const;
char& at(size_t pos);
const char* c_str() const;
std::string& append(const char*);
void push_back(char c);
```

Example use:

```
string s = "ABC";                // s <-- "ABC"
int n = s.length();              // n <-- 3
char c = s.at(2);                // c <-- 'C'
s.append("DEF");                 // s <-- "ABCDEF"
s.push_back('G');                // s <-- "ABCDEFG"
s.at(2) = '_';                   // s <-- "AB_DEFG
const char* str = s.c_str(); // str <-- "AB_DEFG"
```

Member functions and operator overloading

Some member functions can correspond to the **overloading of operators**. An example is the attribution operator (=), usually overloaded to copy state between objects (similarly to a copy constructor).

Declaration:

```
SomeClass& operator=(const SomeClass& other)
```

Use:

```
SomeClass a(. . .)
SomeClass b(. . .)
. . .
a = b; // invokes operator= member function
```

std::string - operator overloading

std::string for instance defines operators =, += e [] as member functions:

```
string& operator=(const std::string& str);
string& operator=(const char* s);
. . .
string& operator+= (const string& str);
string& operator+= (char c);
. . .
const char& operator[](size_t pos) const;
char& operator[](size_t pos);
```

Example use:

```
string a("ABC"), b("DEF");
a += b;      // a <-- "ABCDEF"
a += b[0];   // a <-- "ABCDEFD"
b = a;       // b <-- "ABCDEFD"
a = "XYZ";   // a <-- "XYZ"
```

Destructors

The **destructor** for class `SomeClass` is a function named `~SomeClass`.

```
~SomeClass(); // Destructor
```

When an object goes out of the scope, the destructor is **automatically invoked**. You should never call it explicitly.

The role of the destructor is to **free any internal resources used by the object**, in particular dynamically allocated memory segments if they are used by the object.

Destructor invocation

`~SomeClass()` is automatically invoked:

- at the end of an instruction body where an object of type `SomeClass` é declared;
- at the end of program execution if an object of type `SomeClass` is declared globally;
- when the `delete` operator over a pointer of type `SomeClass*` that refers to an object defined using `new`.

Destructors - invocation

```
SomeClass global_obj;
void f() {
  SomeClass local_obj1;
  if ( . . .) {
     SomeClass local_obj2;
     ...
     // ~SomeClass() implicitly called for local_obj2
  }
  // ~SomeClass() implicitly called for local_obj1
}
```

The constructor and destructor of `global_obj` are invoked during program initialisation and shutdown respectively.

Use of `new` e `delete`

`new` and `delete` can be used to allocate and free objects in dynamic memory. The operators work in conjunction with constructors and destructors.

`new` invokes a constructor:

```
SomeClass* p = new SomeClass(. . . arguments . . .);
```

`delete` invokes `~SomeClass()` before freeing memory:

```
delete p; // invokes ~SomeClass() automatically
```

Use of `new` e `delete` (cont.)

The `->` operator can be used (similarly to `struct` types) to invoke member functions using a pointer. `&` and `*` work as before for pointer types.

```
string* pa = new string();
string* pb = new string("abcdef");
string c("ghi"); // stack-allocated
string* pc = &c;
pa->append(*pb);
(*pa).append(*pc);
cout << *pa << ' ' << pa->length() << '\n';
delete pa;
delete pb;
```

`std::string` in more detail

`std::basic_string` and `std::string`

`string` is defined as an instantiation of a general template class for strings called `basic_string`

```
typedef std::basic_string<char> string;
```

`basic_string` is used to defined strings that have multi-byte characters, e.g., `wstring` (among others):

```
typedef basic_string<wchar_t> wstring;
```

In any case, all `basic_string<CharT>` objects work similarly: an internal `CharT` array holds the string characters, that grows dynamically in size when needed.

String member functions - brief summary

(check the documentation for reference)

Construction and assignment: several constructors are defined, as well as string assignment through `assign` and `operator=`.

Length: `length` or `size`

Element access: `operator[]`, `at`, `front`, `back`, ...

String/character operations: `operator+=`, `operator+`, `append`, `push_back`, `insert`, `erase`, `find`, `clear`, `substr`

String comparison: relational operators (`<`, `==`, `!=`, ...), `compare`

Buffer capacity: `capacity`, `reserve`, `shrink_to_fit`, ...

Range-based for loops

Strings can be used with range-based `for` loops, e.g.

```
string s = "abcde";
// Convert string to upper case
for (char& c : s) {
    c = toupper(c);
}
// Iterate characters and print them
for (char c : s) {
    cout << c;
}
```

```
ABCDE
```

(note: `toupper` used above to convert from lowercase to uppercase characters)

To discuss later in the semester: range-based `for` loops implicitly use **iterators** obtained through member functions like `begin()` and `end()`.

Example 2

Test if given string is a heterogram, i.e., contains no repeated letters.

```
bool heterogram(const string& s, string& r) {
  int count[26] = { 0 };
  for (size_t i = 0; i < s.length(); i++) {
    if (s[i] != ' ') {
      if (s[i] >= 'a' && s[i] <= 'z') count[s[i] - 'a']++;
      else count[s[i] - 'A']++;
    }
  }
  // determines repeated letters
  r.clear();
  for (char c = 'a'; c <= 'z'; c++)
    if (count[c - 'a'] > 1) r.push_back(c);
  return r.empty();
}
```

`std::vector`

Basic aspects

`vector` is a template class that is part of the C++ library known as the Standard Template Library (STL) or "containers library". The STL defines template classes for common data structures (e.g., lists, sets, and maps) and algorithms (e.g., sorting or searching).

`vector` stores a sequence of elements of a parameterised type using a dynamic array. As with strings, the internal array associated to a vector grows when needed. In fact, many member functions in `vector` have similar names and functionality to those found in `string`:

```
vector<int> iv { 1, 2, 3 };
iv.push_back(4);
vector<string> sv { "A", "B", "C" };
for (size_t i = 0; i < sv.size(); i++) cout << sv[i];
```

Example 2

```
vector<int> v;
while (true) {
  int x; cin >> x;
  if (x == 0) break;
  v.push_back(x);
 }
 sort(v.begin(), v.end());
 for (int x : v) cout << x << ' ;
```

```
5 -8 1 0
-8 1 5
```

Program fragment: reads a sequence of integers terminated by `0` onto a `vector`; sorts the vector using `sort`, and; prints the vector contents at the end using a range-based for loop.

Stream-based I/O

Overview of the stream classes

Base stream classes, defined in header `iostream`:

- `std::istream`: base input stream (the class of globally declared object `std::cin`), specialisation of template `basic_istream` for `char` character input;
- `std::ostream` base output stream (the class of `std::cout`), specialisation of the template `basic_ostream` for `char` character output;

File stream classes in header `fstream`:

- `std::ifstream`: file input stream;
- `std::ofstream`: file output stream;“

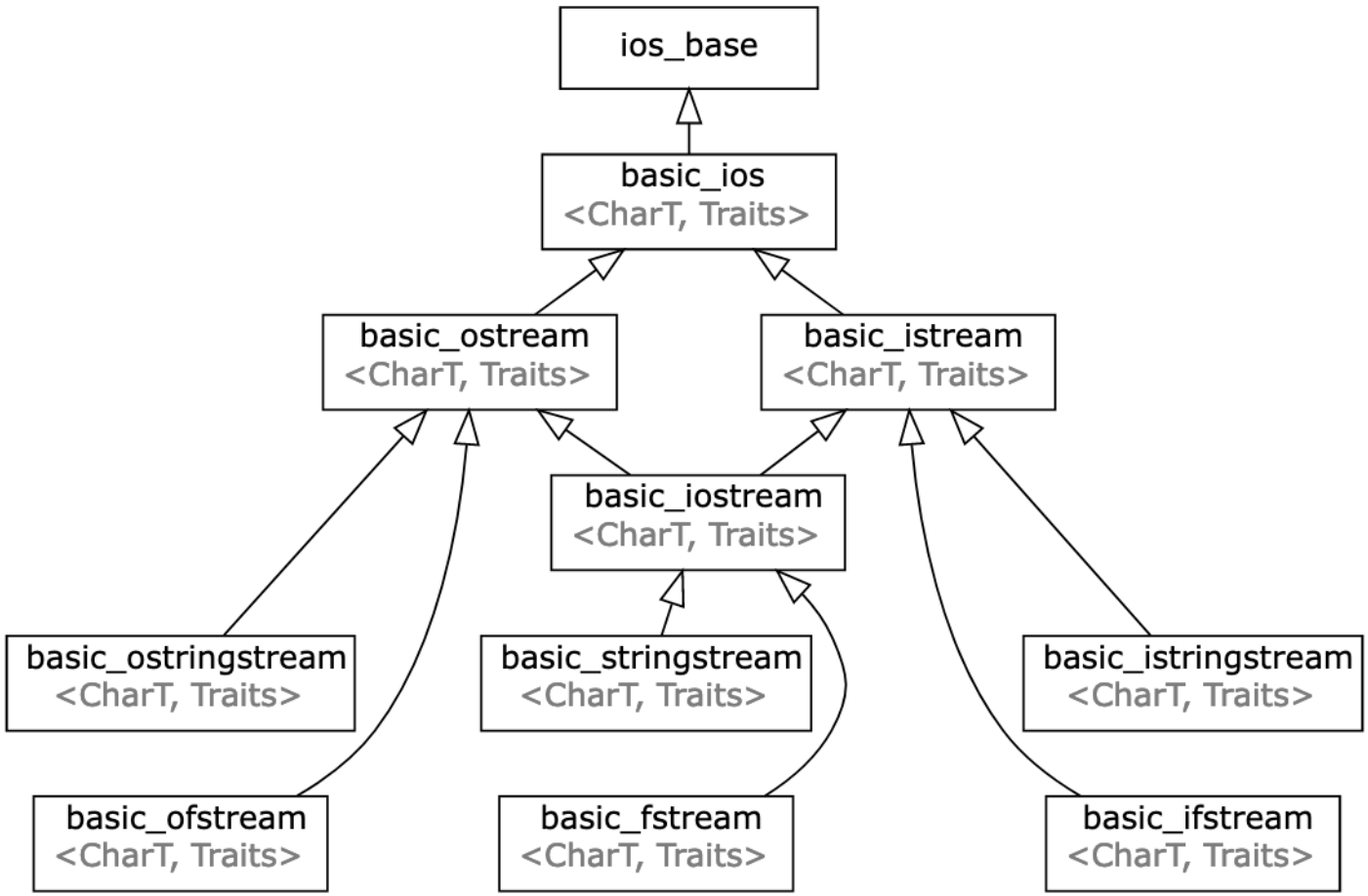
String streams (use strings as streams!), defined in header `sstream`:

- `std::istringstream`: string input stream;
- `std::ostringstream`: string output stream;

There are also stream classes that allow both input and output: `iostream`, `fstream`, and `stringstream`.

Overview of the stream classes (cont.)

The stream classes form a **class hierarchy**, a concept la later in the semester. The basic intuition is that a **class** can reuse and extend the functionality of a **parent class**.



(image from cppreference.com - "Input/Output library")

Example - `ifstream`

Read `double` values from a text file name `numbers.txt` and output their sum

```
ifstream in("numbers.txt");
double sum = 0;
while (true) {
  double x;
  in >> x;
  if (in.eof()) break;
  sum += x;
  cout << "Read " <<  x << '\n';
}
cout << "Sum: " << sum << '\n';
```

The input file can contain numbers separated by blank characters (e.g., spaces, tabs or line breaks) which are handled by the `>>` operator. `eof()` returns `true` when the end of the file is reached, and `>>` will fail in that case without assigning `x`.

Example - `ifstream` - variation

More succint code:

```
ifstream in("numbers.txt");
double sum = 0, x;
while (in >> x) {
  sum += x;
  cout << "Read " <<  x << '\n';
}
cout << "Sum: " << sum << '\n';
```

Explanation: `in >> x` returns `in` (as usual) for chained calls, which can then be evaluated as a boolean expression through `operator bool`. The result is `true` if the stream has no errors and is ready for I/O operations.

Reading line by line - example

Reads numbers line by line, and output the sum of numbers found in each line. `getline` reads an entire line onto a string, and an `istringtream` object is used to obtain the values per each line.

```
ifstream in("numbers.txt");
string line;
int line_count = 1;
while (getline(in, line)) {
  double sum = 0, x;
  istringstream iss(line);
  while(iss >> x) sum += x;
  cout << "Line " << line_count
       <<   " - Sum: " << sum << '\n';
  line_count++;
}
```

```
Line 1 - Sum: 10.2
Line 2 - Sum: 14.5
Line 3 - Sum: -2.5
```

Error recovery when reading data

Upon a read error, the input stream will be stuck at the file position where the error occurred. We can recover from the error by clearing the error flag using clear() and skipping a certain number of characters using ignore().

```
int read_int() {
  int x;
  while (true) {
    if (cin >> x)
      break;
    cin.clear(); // clear error flag
    cin.ignore(1); // skip 1 character and try again
  }
  return x;
}
```

Error recovery - variation

`ignore()` can also be used to skip all characters until a given character is found, e.g., a line break.

```
int read_int_v2() {
  int x;
  while (true) {
    if (cin >> x)
      break;
    cin.clear(); // clear error flag
    cin.ignore(std::numeric_limits<std::streamsize>::max(),
               '\n'); // skip rest of the line
  }
  return x;
}
```

Output streams

Output streams like `fstream` and `sstream` can be used just like `ostream` objects. For instance, the "sum-by-line" program variant below outputs the sum print-outs to an output file called `sums.txt`:

```
ifstream in("numbers.txt");
ofstream out("sums.txt"); // <-- File output stream
string line;
int line_count = 1;
while (getline(in, line)) {
  double sum = 0, x;
  istringstream iss(line);
  while(iss >> x) sum += x;
  out << "Line " << line_count
      <<   " - Sum: " << sum << '\n';
  line_count++;
}
```