Classes e objectos em C++

Programação (L.EIC009)

Eduardo R. B. Marques, DCC/FCUP

Uso de classes e objectos em C++

Uso de classes e objectos em C++

- Ciclo de vida de um objecto
  - Construção
  - Uso de funções membro
  - Destruição

Classes e objectos

Uma classe `SomeClass` é um tipo de dados declarado com a palavra chave `class`:

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

Um **objecto** é uma instância de uma classe. O ciclo de vida de um objecto compreende:

- a construção do objecto na altura da sua alocação (na stack, heap, ou memória global) usando uma função de inicialisação para a classe, designada por **construtor**;
- a invocação de **funções membro** definidas para o objecto, depois do objecto construído;
- a invocação implícita e automática de uma função especial designada por **destrutor**, quando o objecto é dealocado.

Exemplo - `std::string`

A classe `std::string` serve para representar strings, i.e. sequências de valores de tipo `char`. Internamente, um objecto `std::string` é implementado usado um array de tipo `char` que cresce dinamicamente quando necessário.

```
// Alguns dos construtores
string (const string& str);
string (const char* s);
string (const char* s, size_t n);
...
// Destrutor
~string();
// Exemplo de algumas funções membro
std::string& append(const std::string);
const char& at (size_t pos) const;
size_t length() const;
void push_back (char c);
...
```

Construtores

Um construtor de `SomeClass` é uma função com o mesmo nome que `SomeClass` e pode ter vários argumentos.

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

O construtor sem argumentos é chamado o **"default constructor"** ou **construtor por omissão**.

O construtor que toma com argumento um valor ou referência (tipicamente `const`) para `SomeClass` de dados é chamado de **"copy constructor"** ou **construtor por cópia**.

Construtores - uso

Para:

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

Uma variável de tipo `SomeClass` pode ser declarada com a sintaxe genérica

```
SomeClass var( ... argumentos de construção ...);
```

por ex.

```
// Uso de construtor SomeClass(int, int)
SomeClass v1(1, 2);
// Uso de construtor por omissão
SomeClass v1;
// Uso de construtor por cópia
SomeClass v2(v1);
SomeClass v3 = v1;
```

Exemplo - construtores de `std::string`

A classe `std::string` define vários construtores, entre os quais:

```
// 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);
...
```

Exemplo - construtores de `std::string` (cont.)

Declaração de variáveis de tipo `std::string` empregando construtores referidos anteriormente:

```
std::string a; // empty string
std::string b("ABC"); // from C string
std::string c = "DEF"; // from C string (syntactic alternative
std::string d("IJKL", 3); // from C string, up to 3 chars
std::string e(d); // copy
std::string f = e; // copy (synctatic variant)
```

Funções membro

Depois de construído um objecto, podemos invocar **funções membro** sobre o objecto.

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

Uma função membro pode:

- derivar informação do estado interno do objecto, **sem o alterar**, sendo normalmente declarada com o modificador `const` nesse caso;
- **alterar** o estado interno do objecto, não podendo ser declarada como `const`

Exemplos de funções membro em `sdt::string`

Funções:

```
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);
```

Exemplo de uso:

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

Funções membro e “operator overloading”

“Operator overloading” pode ser expresso por funções membro embora tal não seja prático ou possível em todos os casos, como veremos depois.

Um exemplo comum é operador de atribuição `=`, definido para cópia do estado de objectos, de forma análoga ao construtor por cópia:

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

Uso:

```
SomeClass a(...)
SomeClass b(..)
...
a = b;
```

Funções membro e "operator overloading" (cont.)

Em `std::string` temos por exemplo os operadores `=`, `+=` e `[]` implementados como funções membro:

```
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);
```

Exemplo de uso:

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

Destrutores

O **destrutor** para uma classe `SomeClass` é um método identificado por

```
~SomeClass();
```

O papel do destrutor é libertar recursos associados ao objecto, em particular segmentos de memória alocada dinamicamente.

`~SomeClass()` é invocado automaticamente:

- no final do corpo de instruções onde um objecto `SomeClass` é declarado;
- no final do programa para um objecto `SomeClass` globalmente;
- quando é usado o operador `delete` sobre um apontador `SomeClass*` para o objecto criado com `new`.

Destrutores - invocação

Variáveis locais e globais:

```
SomeClass global_obj;
void f() {
  SomeClass local_obj1;
  while(...) {
     SomeClass local_obj2;
     ...
     // ~SomeClass() chamado para local_obj2
  }
  // ~SomeClass() chamado para local_obj1
}
```

O constructor e destrutor de `global_obj` são invocados respectivamente na inicialisação e término do programa.

Uso de `new` e `delete`

Podemos usar `new` e `delete` em para objectos que usam memória dinâmica (a "heap"). Os operadores executam em associação à invocação de construtores e destrutores (ao contrário de `malloc` e `free`!).

Uso de `new`:

O operador `new` invoca um construtor de `SomeClass`;

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

Uso de `delete`:

O operador `delete` invoca `~SomeClass()` antes de libertar dinamicamente a memória.

```
delete p;
```

Uso de `new` e `delete` - arrays de objectos

Podemos também criar arrays de objectos com `new`:

```
int n = ...;
SomeClass* arr = new SomeClass[n];
...
delete [] arr;
```

No fragmento acima, é invocado o construtor por omissão de `SomeClass`. Não é possível invocar outro constructor para todos os elementos. Quando muito, a podemos usar "initializer list" para um nº fixo de elementos:

```
int n = ...;
SomeClass* arr = new SomeClass[] {
  SomeClass(1,2), // element 0
  SomeClass()  // element 1
};
...
delete [] arr;
```

Arrays de objectos - exemplo com `std::string`

Alocação estática (válida em C++ 11):

```
// Strings vazias, construtor por omissão
std::string a[3];
// Uso de outros construtores
std::string b[5] { // todos os elementos "A"
  std::string("A"),
  "A", // equivalente ao elemento anterior
  a[0] + 'A',
  std::string("ABC", 1),
  { "ABC", 1 } // equivalente ao elemento anterior
};
```

Arrays de objectos - exemplo com `std::string` (cont.)

Alocação dinâmica com `new` e inicialisação similar ao exemplo anterior:

```
std::string *a = new std::string[3];
std::string *b = new std::string[5] {
  std::string("A"),
  "A",
  a[0] + 'A',
  std::string("ABC", 1),
  { "ABC", 1}
};
```

Outros exemplos

Uso de `std::ifstream` para contar as linhas e número de bytes em um ficheiro (ver `ifstream_example.cpp`):

```
std::ifstream ifs(filename);
int lines = 0;
int bytes = 0;
char c;
while ( ifs.get(c) ) {
  if (c == '\n') lines++;
    bytes++;
  }
  ifs.close();
}
```

Outros exemplos (cont.)

Uso de `std::vector` (ver código completo em `vector_example.cpp`):

```
int main(int argc, char** argv) {
    std::vector<double> values;
    for (int i = 1; i < argc; i++)
      values.push_back( std::stod(argv[i]) );
    std::sort(values.begin(), values.end());
    int n = values.size();
    double median = n % 2 != 0 ?
                values.at(n / 2) :
                0.5 * ( values.at(n / 2 - 1)
                      + values.at(n/2) );
    std::cout << "Median: " << median << std::endl;
}
```

`std::vector` é uma classe "template" para armazenar sequências de elementos. Internamente, um objecto `vector` usa um array que cresce dinamicamente quando necessário.

Definição de classes

Definição de classes

- Declaração de uma classe.
- Campos e funções membro.
- Construtores e destrutores.
- Campos e funções de classe (`static`).
- Visibilidade: `public` e `private`.
- "Operator overloading"
- Compilação separada / "header" files para classes.
- Classes "template".

Definição de uma classe

Uma classe pode ser declarada fazendo uso de `class`:

```
namespace SomeNamespace {
  class SomeClass {
    ...
  };
}
```

Tal como vimos para funções, uma classe pode ser declarada no contexto de um namespace (`SomeNamespace` acima).

Exemplos

```
namespace leic {
  class fraction {
    ...
  };
  class polynomial {
    ...
  };
};
```

Iremos ver como exemplos a definição de 2 classes:

- `fraction`: para representar fracções $\frac{n}{d}$ (número racional) na forma irredutível definido por numerador $n \in \mathbb{Z}$ e denominador $d \in \mathbb{Z} - \{0\}$.
- `polynomial`: para representar polinómios $f_0 + f_1 x^1 + f_2 x^2 + ... + f_n x^n$ com coeficientes dados por fracções $f_1, \ldots, f_n$ e $f_n \neq 0$.

Campos membro

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

Uma classe pode declarar **campos membro** (também chamados de campos de instância). Cada objecto que seja instância da classe terá definidos os campos membro que forem declarados.

Exemplo acima: campos `num` e `den` para objectos `fraction`, e `coeffs` para objectos `polynomial`.

Construtores e campos membro

```
class fraction {
  int num, den;
  ...
  fraction() { num = 0; den = 1; }
  ...
};
class polynomial {
  ...
  std::vector<fraction> coeffs;
  ...
  polynomial() { ... }
  ...
};
```

Campos membro devem ser apropriadamente inicialisados por construtores. Em `polynomial()` o campo `coeffs` é inicialisado implicitamente via construtor por omissão em `std::vector`.

Construtores (cont.)

Como referido antes, uma classe pode ter vários construtores:

```
fraction() {
  num = 0;
  den = 1;
}
fraction(int n, int d=1) {
  // ver nota sobre assert nos slides a seguir
  assert(d != 0);
  num = n;
  den = d;
}
fraction(const fraction& f) {
  num = f.num;
  den = f.den;
}
```

Nota: uso de `assert`

```
#include <cassert>
...
fraction(int n, int d) {
  // ver nota sobre assert
  // no slide a seguir
  assert(d != 0);
  num = n;
  den = d;
}
```

A invocação de `assert(expression)` em C ou C++, dita uma **asserção**, aborta a execução de um programa caso `expression == 0`.

Nota: uso de `assert` (cont.)

Ao executar

```
fraction f(1, 0);
```

o programa é abortado:

```
Assertion failed: (den != 0),
function fraction, fraction.hpp, line 24.
```

Para lidar com erros é mais comum em C++ o uso de **excepções**, tópico de que falaremos em futuras aulas.

Destrutores

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

Um destrutor tem o papel de limpeza de recursos associados a um objecto. Acima `~fraction()` e `~polynomial()` têm corpo vazio (poderiam ser omitidos). O destrutor `~vector()` é invocado automaticamente para `coeffs` em qualquer caso.

Destrutores (cont.)

Considere uma variante de `polynomial` que usasse um array alocado com `new` para os coeficientes:

```
class polynomial {
  ...
  fraction* coeffs; // alocado com new
  ...
  ~polynomial() {
    delete [] coeffs;
  }
  ...
};
```

Seria neste caso necessário libertar explicitamente a memória para `coeffs` usando `delete`.

Funções membro

```
class fraction {
  int num, den;
  ...
  int numerator() const { return num; }
  int denominator() const { return den; }
  // Converte fração à forma irredutível
  void reduce() {
    int g = gcd(num, den);
    num /= g; den /= g;
    if (den < 0) {
      num = -num; den = -den;
    }
  }
  ...
};
```

Funções membro podem ler ou modificar campos membro.

Funções globais a uma classe (`static`)

```
class fraction {
  ...
  static int gcd(int a, int b) {
     while (b != 0) {
       int tmp = a;
       a = b;
       b = tmp % b;
     }
     return a;
  }
  ...
};
```

O modificador `static` indica que uma função é global à classe - não estando portanto associados a nenhum objecto instância de `fraction`.

Campos globais a uma classe (`static`)

Da mesma forma podemos definir campos globais a uma classe, ex.

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

const fraction fraction::ZERO(0,1);
```

A inicialização de campos globais (como `ZERO` acima) tem de figurar fora da declaração da classe ... um aspecto pouco "linear" de C++.

Funções membro - mais exemplos ...

Agora em `polynomial`:

```
class polynomial {
  std::vector<fraction> coeffs;
  ...
  void reduce() {
    while (coeffs.size() > 1 && coeffs.back().is_zero()) {
        coeffs.pop_back();
    }
  }
  ...
};
```

Redução à "forma normal": remove sucessivamente coeficiente de maior grau enquanto este for nulo, com excepção do coeficiente de grau 0.

Funções membro - mais exemplos ... (cont.)

```
class polynomial {
  std::vector<fraction> coeffs;
  ...
  fraction evaluate(const fraction x) const {
    fraction r(0), pow(1);
    for (const fraction& c : coeffs) {
      r += c * pow;
      pow *= x;
    }
    return r;
  }
  ...
};
```

`evaluate(x)`: calcular valor do polinómio para `x`. Faz uso de operadores `+=`, `*` e `*=` definidos em `fraction` e de um ciclo "for-each" sobre `coeffs` (aspectos a discutir em futuras aulas).

Visibilidade `public` e `private`

Uma classe pode conter secções `public:` e `private`.

```
namespace SomeNamespace {
  class SomeClass {
  private:
    ...
  public:
    ...
  };
}
```

Em uma secção `public` encontram-se declarações de campos ou funções de visibilidade **pública**: são acessíveis de fora da classe sem restrições.

Ao invés, em uma secção `private` encontram-se declarações de visibilidade **privada**: só podem ser usadas pelo código da própria classe.

Visibilidade `public` e `private` (cont.)

```
class fraction {
private:
  int num, den;
  static int gcd(int a, int b) { ... }
  ...
public:
  ...
};
```

Tipicamente campos membro têm visibilidade privada, para evitar modificação arbitrária por via externa à classe. Por exemplo, em `fraction` o acesso de fora a `num` ou `den` poderia quebrar a invariante de a fracção estar na forma irredutível.

Em uma secção `private` podem também declarar-se por ex. funções ou tipos auxiliares internos ao funcionamento da classe.

Visibilidade `public` e `private` (cont.)

```
class fraction {
private:
  int num, den;
  void reduce() { ... }
  ...
public:
  ...
  fraction() { ... }
  ...
  int numerator() const { ... }
  static const fraction ZERO;
  ...
};
```

Construtores têm tipicamente visibilidade pública. Em uma secção `public` podemos encontrar outras declarações que podem ser usadas de fora da classe, definindo o "interface" de uso externo para a classe.

Visibilidade `public` e `private` (cont.)

Acesso a membros com visibilidade `private` fora da classe

```
int x = f.num; // NÃO PERMITIDO
```

dão origem a erros de compilação:

```
error: 'num' is a private member of 'leic::fraction'
```

"Amigos" ... uso de `friend`

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

O uso de `friend` identifica funções ou classes "amigas" que podem ter acesso a declarações privadas na classe. Por exemplo, a função do operador `<<` identificada acima pode aceder aos campos privados `num` e `den` de um objecto `fraction`:

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

"What is `this` ?"

`this` designa o objecto em contexto numa função membro. Por exemplo:

```
fraction() {
  num = 0;
  den = 1;
}
```

é implicitamente equivalente a

```
fraction() {
  this -> num = 0;
  this -> den = 1;
}
```

O uso de `this` é dispensável frequentemente, mas veremos alguns casos em que o seu uso é necessário.

Mais sobre construtores

Em um construtor podemos usar listas de inicialisação para campos membro:

```
fraction() : num(0), den(1) {

}
fraction(int n, int d)  : num(n), den(d) {
  assert(den != 0);
  reduce();
}
fraction(const fraction& f) : num(f.num), den(f.den) {

}
```

Mais sobre construtores (cont.)

O uso de lista de inicialisação para membros é especialmente relevante quando alguns dos campos membros são objetos, por forma a invocar construtores para esses campos:

Possível variante de `polynomial`:

```
std::vector<fraction> coeffs;

// Chama construtor std::vector(size_t n)
// para inicialiar coeffs
polynomial(int n) : coeffs(n) {

}
```

Mais sobre construtores (cont.)

Podemos também usar a lista de inicialização para chamar outros construtores:

```
fraction() : fraction(0) {

}

fraction(const fraction& f) : fraction(f.num, f.den) {

}

fraction(int n, int d=1)  : num(n), den(d) {
  assert(den != 0);
}
```

Uso de `initializer_list`

Desde o C++ 11 podemos usar o tipo `std::initializer_list<T>` (não confundir com lista de inicialização de membros) por forma passar um número de argumentos variáveis a funções, e construtores em particular:

```
#include <initializer_list>

class  polynomial {
  ...
  polynomial(initializer_list<fraction> il) { ... }
};
```

Uma `initializer_list` é implicitamente criada com a sintaxe `{ elem1, elem2, ... }`

```
// 1/2 + 3/4 x + 0 x^2 -1 x^3
polynomial p {{ 1, 2}, {3, 4}, fraction::ZERO, { -1 }};
```

Uso de `initializer_list` (cont.)

`std::vector` pode tomar `il` como argumento no construtor:

```
polynomial(initializer_list<fraction> il) : coeffs(il) {
  reduce();
}
```

Alternativa: inicialisação de `coeffs` com contrutor por omissão seguida de adição elemento a elemento:

```
polynomial(initializer_list<fraction> il)  {
  for (const fraction& f : il ) {
       coeffs.push_back(f);
  }
  reduce();
};
```

Nota: tal como `vector` `initializer_list` é iterável (assunto a discutir em futuras aulas), portanto acima é válido o uso de um ciclo "for-each".

"Overloading" de operadores

```
T1 a = ...;
T2 b = ...;
TR r = a OP b;
```

Para um operador binário `OP` com operandos de tipo `T1` e tipo `T2`, em que `T1` é uma classe, e ainda tipo de retorno `TR`, temos duas opções para a implementação:

1 `OP` pode ser definido como função membro na classe `T1`, sendo `this` implicitamente o 1º argumento:

```
class T1 {
  ...
  TR operatorOP(T2 arg) { ... }
};
```

2 para alguns operadores é permitida a definição de `OP` fora do âmbito de qualquer classe:

```
TR operatorOP(T1 a, T2 a) { ... }
```

"Overloading" de operadores (cont.)

1 Em `fraction` podemos ter:

```
class fraction {
  ...
  // Implementado com função membro.
  bool operator==(const fraction& f) const {
    return num == f.num && den == f.den;
  }
  ...
};
```

2 Alternativa:

```
  class fraction { ... };
  // Implementado com função externa à classe.
  bool operator==(const fraction& a, const fraction& b) const
    return a.numerator() == b.numerator()
        && a.denominator() == b.denonimator();
  }
```

"Overloading" de operadores (cont.)

De forma análoga podemos ter operadores unários definidos como funções membro ou fora do âmbito da classe.

1

```
class fraction {
   ...
   // Implementado com função membro
   fraction operator-() const {
      return fraction(- num, den);
   }
};
```

2

```
class fraction { ... };
// Implementado com função membro
fraction operator-(const fraction& f) const {
   return fraction(- f.numerator(), f.denominator());
}
```

“Overloading” de operadores (cont.)

Casos comuns de “operator overloading”

Operator `==` e `!=` para teste de igualdade definidos como funções membro em `SomeClass`

```
class SomeClass {
  ...
  bool operator==(const SomeClass& other) const { ... }
  bool operator!=(const SomeClass& other) const { ... }
};
```

"Overloading" de operadores (cont.)

Casos comuns de "operator overloading"

Operador `=` para atribuição definido em `SomeClass` como:

```
class SomeClass {
  ...
  SomeClass& operator=(const SomeClass& other) {
    ... // Copia estado interno de other para this
    return *this; // Retorna referência ao próprio objecto.
  }
  ...
};
```

Implementação deve retornar `*this` para permitir encadeamento de atribuições (ex. do género `a = b = c`). De forma análoga operadores de atribuição composta (`+=`, `-=`, etc) costumam retornar `*this`.

“Overloading” de operadores (cont.)

Casos comuns de “operator overloading”

Mais exemplos de `fraction`:

```
fraction& operator=(const fraction& f) {
    num = f.num;
    den = f.den;
    return *this;
}
fraction& operator+=(const fraction& f) {
    num = num * f.den + f.num * den;
    den = den * f.den;
    reduce();
    return *this;
}
```

“Overloading” de operadores (cont.)

Outros casos comuns de “operator overloading”

- operador `[]` para indexação, ex. definido para `std::string` ou `std::vector`.
- operadores `>>` e `<<` para leitura/escrita de/para um “stream” de input/output, como função externa a classes. Tipicamente o retorno é uma referência ao stream para chamadas encadeadas (ex. como em `std::cout << a << b << ...`).

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

Compilação separada

Para programas maiores é conveniente a divisão da definição de classes:

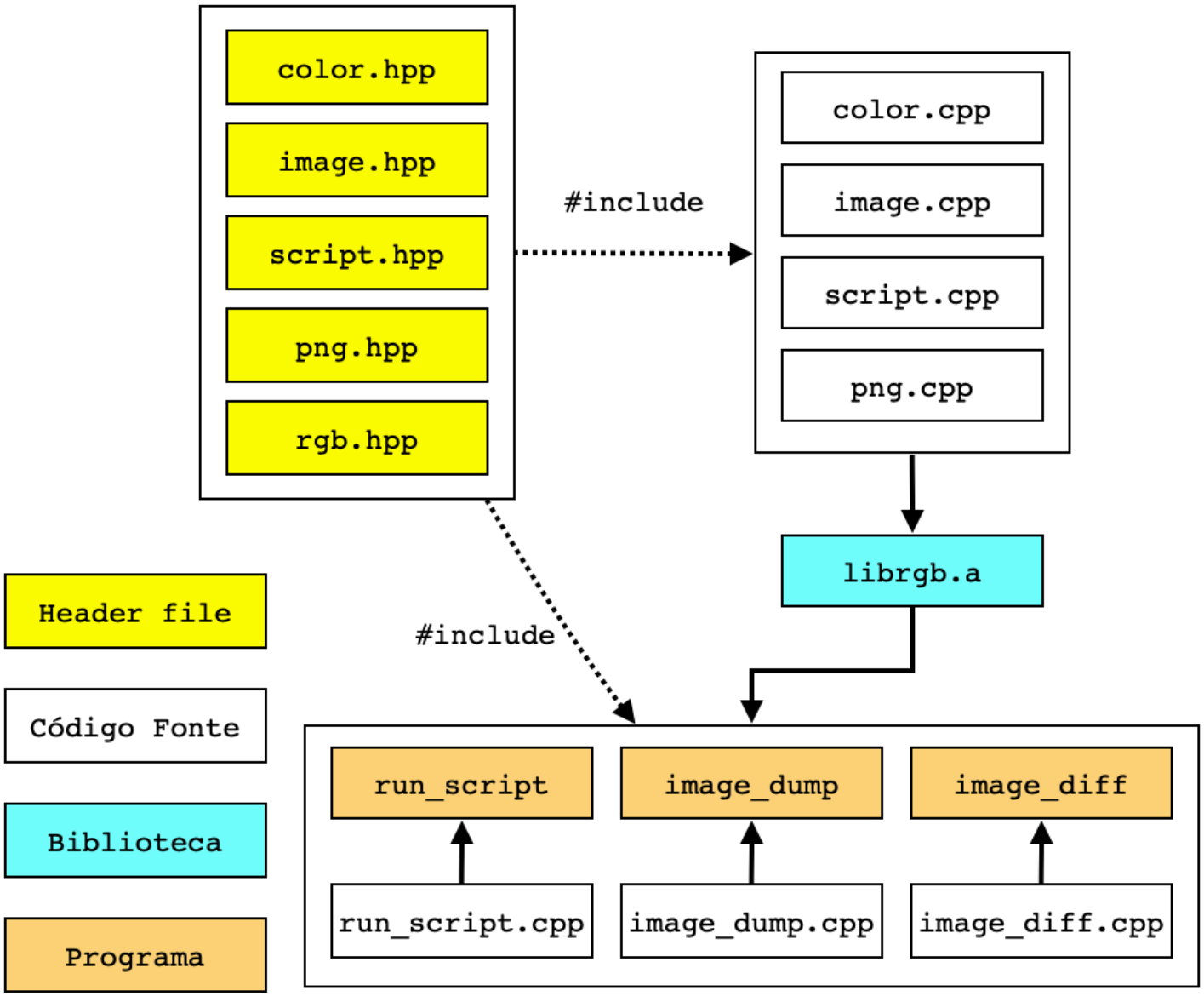
- em "header files" que definem os tipos e funções, dito o interface da classe, mas não o código de implementação;
- em ficheiros de implementação contendo a definição de funções;

Vantagens:

- apenas ficheiros que são alterados precisam de ser recompilados;
- classes podem ser parte de uma biblioteca que é depois ligada com múltiplos programas cliente;

Compilação separada - exemplo (cont.)

Estrutura parcial do projecto 1:



Compilação separada - exemplo (cont.)

Em `color.hpp`

```
...
class color {
private:
  rgb_value r;
  rgb_value g;
  rgb_value b;
public:
  static const color WHITE;
  ...
  color();
  ...
  rgb_value red() const;
  rgb_value& red();
   ...
};
```

Compilação separada - exemplo (cont.)

Em `color.cpp`:

```
#include <rgb/color.hpp>
...
const color color::WHITE(255,255,255);
...
color::color() { ... }
...
rgb_value color::red() const {
    return r;
}
rgb_value& color::red() {
    return r;
}
...
```

Classes "template"

De forma análoga ao que vimos antes para tipos `struct` e funções, podemos ter classes "template" (por ex. `std::vector` é uma classe "template").

```
namespace SomeNamespace {
  template <typename T>
  class SomeTemplateClass {
    ...
  };
}
```

Tipicamente classes template são definidas **completamente em "header files"**, incluindo **também** o código de implementação. C++ não permite compilação separada nestes casos.

Exemplo a seguir: classe template para uma fila simples implementada internamente com uma lista ligada (veja `queue.hpp`).

Classes “template” - exemplo

```
template <typename T>
class queue {
private:
  struct Node {
    T value;
    Node* next;
  };
  int q_size;
  Node *head, *tail;
public:
  queue();
  ~queue();
  bool empty() const;
  int size() const;
  void add(const T& value);
  T remove();
};
```

Classes “template” - exemplo (cont.)

Implementação (no mesmo “header file”):

```
...
template <typename T>
queue<T>::queue() : q_size(0), head(NULL), tail(NULL) { }

template <typename T>
queue<T>::~queue() {
  for (Node* node = head; node != NULL; node = node -> next)
    delete node;
}

template <typename T>
bool queue<T>::empty() const { return size() == 0; }

template <typename T>
int queue<T>::size() const { return q_size; }
...
```

Classes "template" - exemplo (cont.)

```
...
template <typename T>
void queue<T>::add(const T& value) {
  Node* node = new Node;
  node -> value = value;
  node -> next = NULL;
  if (tail != NULL) tail -> next = node;
  else head = node;
  tail = node; q_size++;
}
template <typename T>
T queue<T>::remove() {
  Node* node = head;
  T value = node -> value;
  head = node -> next; q_size--;
  delete node;
  return value;
}
```