C++: Gerenciamento de Memória

INF1761 – Computação Gráfica

Waldemar Celes
celes@inf.puc-rio.br

Departamento de Informática, PUC-Rio

Arrays

Alocação dinâmica de *arrays*

```
int n = ...;
...
float* v = new float[n];
v[0] = ...;
v[1] = ...;
v[n-1] = ...;
...
delete [] v;
```

Template

Implementação comum a diferentes tipos

```
template<typename T> void swap (T& a, T& b)
{
  T t = a;
  a = b;
  b = t;
}
```

Template

Implementação comum a diferentes tipos

```
template<typename T> void swap (T& a, T& b)
{
  T t = a;
  a = b;
  b = t;
}
```

- Uso: tipo explícito ou inferido do contexto

```
int main ()
{
  float a = 1.2f;
  float b = 2.3f;
  swap<float>(a,b);   // ou simplesmente: swap(a,b);
  std::cout << a << " " << b << std::endl;
  return 0;
}
```

Template

Implementação comum a diferentes tipos

```
template<typename T> void swap (T& a, T& b)
{
  T t = a;
  a = b;
  b = t;
}
```

- Uso: tipo explícito ou inferido do contexto

```
int main ()
{
  float a = 1.2f;
  float b = 2.3f;
  swap<float>(a,b);   // ou simplesmente: swap(a,b);
  std::cout << a << " " << b << std::endl;
  return 0;
}
```

- Classes também podem ser templates

```
template <typename T> class A {
  T m_member;
  ...
};
```

Classe `std::vector`

Implementa *array* dinâmico

- Template do tipo do elemento armazenado

```
#include <vector>
...
  std::vector<float> v;
  v.push_back(1.2f);
  v.push_back(2.3f);
...
```

  - Pode ser inicializado *a la* array local

```
std::vector<float> v {1.2f, 2.3f};
...
v.push_back(7.4f);
...
```

  - Note que o `vector` é implementado como código regular
    - Sem privilégios de acesso

Classe `std::vector`

Formas de iterar sobre os elementos do vetor

- Usando `for` de índices convencional

```
for (int i=0; i<v.size(); ++i)
  std::cout << v[i] << std::endl;
```

- Usando `for` com iteradores

```
for (std::vector<float>::iterator it = v.begin() ;
     it != v.end();
     ++it
    )
  std::cout << *it << std::endl;
```

- Usando *range-based* `for`

```
for (float x : v)
  std::cout << x << std::endl;
```

"Tipo" `auto`

Variável do "tipo" `auto`

- Tipo inferido pelo contexto
  - Revendo laço com iteradores

```
for (auto it = v.begin(); it != v.end(); ++it)
  std::cout << *it << std::endl;
```

  - Revendo laço *range-based*

```
for (auto x : v)
  std::cout << x << std::endl;
```

Herança e Polimorfismo

Exemplo: coleção de figuras

- Classe contendo um container de `Figure`
  - Arquivo "collection.h"

```
#ifndef COLLECTION_H
#define COLLECTION_H

#include <vector>

class Figure;

class Collection
{
  std::vector<Figure*> m_figs;
public:
  Collection ();
  ~Collection ();
  void AddFigure (Figure* fig);
  float ComputeTotalArea () const;
};
```

Herança e Polimorfismo

Exemplo: coleção de figuras

- Classe contendo um container de `Figure`
  - Arquivo "collection.cpp"

```
#include "collection.h"
#include "figure.h"

Collection::Collection ()
: m_figs() {
}
Collection::~Collection () {
}
void Collection::AddFigure (Figure* fig) {
  m_figs.push_back(fig);
}
float Collection::ComputeTotalArea () const {
  float area = 0.0f;
  for (Figure* fig : m_figs) {
    area += fig->ComputeArea();
  }
  return area;
}
```

Herança e Polimorfismo

Exemplo: coleção de figuras

- Cliente

```
#include "square.h"
#include "circle.h"
#include "collection.h"
#include "color.h"

#include <iostream>

int main ()
{
  Color* color = new Color("red");
  Figure* s = new Square(2.0f,color);
  Figure* c = new Circle(1.0f,color);

  Collection* set = new Collection();
  set->AddFigure(s);
  set->AddFigure(c);
  float a_t = set->ComputeTotalArea();
  std::cout << "total area: " << a_t << std::endl;
  return 0;
}
```

Herança e Polimorfismo

Exemplo: coleção de figuras

- Introduzindo inicializador de lista

Arquivo “collection.h”: alteração no construtor

- Note use de valor defautl do parâmetro

```
#include <initializer_list>
...
  Collection (std::initializer_list<Figure*> figs={});
```

Herança e Polimorfismo

Exemplo: coleção de figuras

- Introduzindo inicializador de lista

Arquivo “collection.h”: alteração no construtor

- Note use de valor defautl do parâmetro

```
#include <initializer_list>
...
  Collection (std::initializer_list<Figure*> figs={});
```

Arquivo “collection.cpp”

```
Collection::Collection(std::initializer_list<Figure*> figs)
: m_figs(figs)
{
}
```

Cliente

```
Collection* set = new Collection({s,c});
```

Gerência de memória

Monitoramento de destrutores de cada classe

```
Figure::~Figure ()
{
  std::cout << "figure deleted" << std::endl;
}
```

Gerência de memória

Monitoramento de destrutores de cada classe

```
Figure::~Figure ()
{
  std::cout << "figure deleted" << std::endl;
}
```

- A execução do programa teste não chama nenhum destrutor

Gerência de memória

Monitoramento de destrutores de cada classe

```
Figure::~Figure ()
{
  std::cout << "figure deleted" << std::endl;
}
```

- A execução do programa teste não chama nenhum destrutor

De quem é a responsabilidade de liberar memória?

- Do cliente?
- Do *collection*?

Gerência automática de memória

Entendendo o mecanismo

- Gerência explícita da memória

```
#include <iostream>

class Test {
public:
  Test () {}
  ~Test () {
    std::cout << "test destructor" << std::endl;
  }
};

int main ()
{
  Test* t = new Test();
  delete t;
  return 0;
}
```

- Saída:

```
test destructor
```

Gerência automática de memória

- Gerência implícita de memória

```
#include <iostream>

template <typename T>
class Ptr {
  T* m_p;
public:
  Ptr (T* t) : m_p(t) {}
  ~Ptr () { delete m_p;}
};

class Test {
public:
  Test () {}
  ~Test () {
    std::cout << "test destructor" << std::endl;
  }
};
int main ()
{
  Ptr<Test> p(new Test());
  return 0;
}
```

Gerência automática de memória

Mecanismos de C++

```
#include <memory>
```

- Ponteiros únicos: `std::unique_ptr<T>`

- Ponteiros compartilhados: `std::shared_ptr<T>`

Gerência automática de memória

Mecanismos de C++

```
#include <memory>
```

- Ponteiros únicos: `std::unique_ptr<T>`
  - Só pode existir uma referência para o ponteiro
  - Atribuição via `std::move`, difícil de gerenciar
  - Destrutor chamado ao fim do escopo da única referência
- Ponteiros compartilhados: `std::shared_ptr<T>`

Gerência automática de memória

Mecanismos de C++

```
#include <memory>
```

- Ponteiros únicos: `std::unique_ptr<T>`
  - Só pode existir uma referência para o ponteiro
  - Atribuição via `std::move`, difícil de gerenciar
  - Destrutor chamado ao fim do escopo da única referência
- Ponteiros compartilhados: `std::shared_ptr<T>`
  - Implementa mecanismo de *reference counter*
  - Destrutor chamado ao fim do escopo da última referência

Gerência automática de memória

Mecanismos de C++

```
#include <memory>
```

- Ponteiros únicos: `std::unique_ptr<T>`
  - Só pode existir uma referência para o ponteiro
  - Atribuição via `std::move`, difícil de gerenciar
  - Destrutor chamado ao fim do escopo da única referência
- Ponteiros compartilhados: `std::shared_ptr<T>`
  - Implementa mecanismo de *reference counter*
  - Destrutor chamado ao fim do escopo da última referência
  - Ponteiro "fraco": `std::weak_ptr<T>`
    - Guarda referência sem aumentar *counter*
    - Uso para impedir ciclos de referências compartilhadas

Gerência automática de memória no nosso exemplo

- Definição de "tipo ponteiro": evitar tipos complexos no código
- Fábrica no lugar de construtor: evitar ponteiros soltos
  - Uso de método `static`

```
#include <memory>
class Circle;
typedef std::shared_ptr<Circle> CirclePtr;

#ifndef CIRCLE_H
#define CIRCLE_H
#include "figure.h"
class Circle : public Figure
{
  float m_radius;
protected:
  Circle (float radius, ColorPtr color=nullptr);
public:
  static CirclePtr Make (float radius, ColorPtr c=nullptr);
  virtual ~Circle ();
  float ComputeArea () const;
};
#endif
```

Gerência automática de memória no nosso exemplo

Aplicação no nosso exemplo

- Arquivos ".cpp"

```
...
CirclePtr Circle::Make (float radius, ColorPtr color) {
  return CirclePtr(new Circle(radius,color));
}
...
```

Gerência automática de memória no nosso exemplo

- `Figure` guarda uma referência "fraca" para `Collection`
  - Evitando ciclo de referências

```
#include <memory>
class Figure;
typedef std::shared_ptr<Figure> FigurePtr;

#ifndef FIGURE_H
#define FIGURE_H
#include "color.h"
#include "collection.h"
class Figure {
  ColorPtr m_color;
  std::weak_ptr<Collection> m_parent;
protected:
  Figure (ColorPtr color=nullptr);
public:
  virtual ~Figure();
  void SetParent (CollectionPtr parent);
  ColorPtr GetColor () const;
  CollectionPtr GetParent () const;
  virtual float ComputeArea () const = 0;
};
#endif
```

Gerência automática de memória no nosso exemplo

- `Collection` deve derivar de `std::enable_shared_from_this<Collection>`
  - Permitir criar referência compartilhada a partir do valor `this`

```
class Collection
: public std::enable_shared_from_this<Collection>
{
  std::vector<FigurePtr> m_figs;
protected:
  Collection (std::initializer_list<FigurePtr> figs);
public:
  static CollectionPtr Make
            (std::initializer_list<FigurePtr> figs={});
  ~Collection ();
  void AddFigure (FigurePtr fig);
  float ComputeTotalArea () const;
};
```

Gerência automática de memória no nosso exemplo

- Métodos de `Collection`

```
CollectionPtr Collection::Make
                  (std::initializer_list<FigurePtr> figs)
{
  auto col = CollectionPtr(new Collection(figs));
  for (auto f : figs)
    f->SetParent(col);
  return col;
}

void Collection::AddFigure (FigurePtr fig)
{
  m_figs.push_back(fig);
  fig->SetParent(shared_from_this());
}
```

Gerência automática de memória no nosso exemplo

- Arquivos cliente

```
ColorPtr color = Color::Make("red");
FigurePtr s = Square::Make(2.0f,color);
FigurePtr c = Circle::Make(1.0f,color);
```

Gerência automática de memória no nosso exemplo

- Arquivos cliente

```
ColorPtr color = Color::Make("red");
FigurePtr s = Square::Make(2.0f,color);
FigurePtr c = Circle::Make(1.0f,color);
```

- ou:

```
auto color = Color::Make("red");
auto s = Square::Make(2.0f,color);
auto c = Circle::Make(1.0f,color);
```

Gerência automática de memória no nosso exemplo

- Arquivo cliente completo

```
int main ()
{
  ColorPtr color = Color::Make("red");
  FigurePtr s = Square::Make(2.0f,color);
  FigurePtr c = Circle::Make(1.0f,color);
  float a_s = s->ComputeArea();
  float a_c = c->ComputeArea();
  std::cout << "square area (" << s->GetColor()->GetName() <
  std::cout << "circle area (" << s->GetColor()->GetName() <

  CollectionPtr set = Collection::Make({s,c});

  float a_t = set->ComputeTotalArea();
  std::cout << "total area: " << a_t << std::endl;
  return 0;
}
```

Gerência automática de memória no nosso exemplo

- Saída

```
square area (red): 4
circle area (red): 3.14159
total area: 7.14159
collection deleted
circle deleted
figure deleted
square deleted
figure deleted
color deleted
```