Aula 6: Memória Dinâmica e Redefinição de Operadores

Paulo Cortez

2004/2005

Técnicas de Programação I
Licenciatura em Engenharia Electrónica e Computadores

http://www.dsi.uminho.pt/disciplinas/EITPI

Objectivos da Aula

- Manipular memória dinâmica em C++;
- Redesenhar operadores em classes;

Memória dinâmica: conceitos básicos

- Em C++, são definidos dois operadores: `new` (alocar) e `delete` (libertar);
- Diferenças em relação ao `malloc` e `free`: o `new` chama o respectivo *construtor* (se existir) e o `delete` chama o respectivo *destrutor* (se existir);
- Regras: por cada `new` usar um `delete` e por cada `new[]`, usar um `delete[]` (alocação de *arrays*);

```
double *x=new double; // double, inicializa a 0
int *y=new int(3);    // int, inicializa a 3
int *z;               // array dinâmico
z=new int[20];        // com espaço para 20 inteiros
string *s=new string("ola"); // construtor da string
string *sa=new string[10]; // array com 10 strings
```

Exemplos de libertação de memória

```
delete x;
delete y;
delete[] z; // apagar um array
Object *A=new Object[10];
delete []A;// antes de apagar cada uma das 10 células,
           // activa o respectivo destruidor
```

Alargar vectores (*arrays*)

- Alocar nova memória, com um tamanho maior;
- Copiar os valores antigos para o novo *array*;
- Apagar o *array* antigo;
- Mudar o apontador antigo para o novo *array*;

Exemplo (com *strings*)

```
#include <string>
string *enlarge(string *old, int oldsize, int newsize)
{ string *tmp = new string[newsize]; // alocar novo
  for (int i = 0; i < oldsize; i++)
        tmp[idx] = old[idx]; // copiar old para tmp
  delete[] old;              // apagar old
  return tmp;                // retornar array
}
int main()
{ string *arr = new string[4]; // array com 4 strings
  arr = enlarge(arr, 4, 6);    // alargar para 6
}
```

Destrutores

- Um destrutor tem de assegurar que a memória alocada pelo objecto é libertada;
- Um destrutor é sempre activado:
  - no final de um bloco/função para um objecto local;
  - quando o programa termina para um objecto global ou estático;
  - quando um objecto dinamicamente alocado é apagado (`delete`); e
  - quando um array dinâmico de objectos é apagado (`delete[]`).
  - no caso de classes derivadas, segue-se a ordem de uma pilha (*stack*): primeiro o destrutor mais baixo, depois o destrutor da classe ascendente, etc...

Exemplo *Ages*

```
// ficheiro ages.h:
#ifndef _Ages_h_
#define _Ages_h_
class Ages
{ int *d_ages; int d_size;
  public: Ages(int size); // construtor
          ~Ages();  // destrutor
};
#endif
// ficheiro ages.cc:
Ages::Ages(int size) // construtor
{ d_size=size; d_ages=new int[d_size];}
Ages::~Ages()         // destrutor
{ delete[] d_ages;}
```

Exemplo (ficheiro *main* people.cc)

```
#include "ages.h"
int main()
{
 Ages A(5);
 Ages *B=new Ages(20);
 // ...
 delete B; // destrutor de B activado
 return 0;
} // destrutor de A activado no fim da função
```

Atribuição

- Para copiar estruturas ou classes é necessário copiar cada um dos seus atributos;
- Este processo é facilitado se for definido a respectiva função:

```
void Age::assign(Age const &A)
    { delete d_ages; // apagar a memória utilizada
      d_size = A.d_size; // copiar o conteúdo de A
      d_ages = new int[d_size];
      for(int i=0;i<d_size;i++) d_ages[i]=A.d_ages[i];
    }
```

Atribuição

- Outra alternativa consiste em gerar um construtor que copia, com a vantagem de não ter de apagar memória previamente alocada:

```
Age::Age(Age const &A)
    { d_size = A.d_size; // copiar o conteúdo de A
      d_ages = new int[d_size];
      for(int i=0;i<d_size;i++) d_ages[i]=A.d_ages[i];
    }
```

Apontador *this*

- Um método é sempre chamado no contexto de um objecto de uma classe;
- Em C++, o apontador *this* designa sempre o endereço do objecto actual;
- Embora não seja necessário o seu uso, existem situações diversas onde o uso do apontador *this* é mesmo necessário;
- Exemplo de utilização:

```
class ponto
   { int x; // igual a this->x
     ponto(int x){ this->x=x;} // construtor
   }
```

Redefinição de operadores

- O C++ possui um mecanismo para redefinição de operadores;
- Deve ser utilizado com cuidado, apenas quando:
  - um operador pode gerar efeitos negativos indesejáveis;
  - o seu significado é claro e não gera ambiguidade;
- Noutras situações é preferível definir uma função membro (método);
- Os seguintes operadores podem ser redefinidos:

```
+    -    *   /   %   ^    &    |
~    !    ,   =   <   >    <=   >=
++   --   <<  >>  ==  !=   &&   ||
+=   -=   *=  /=  %=  ^=   &=   |=
<<=  >>=  []  ()  ->  ->*  new  delete
```

Operadores + e +=

```
#include <iostream>
using namespace::std;
class Int
{ int d_x;
  public: Int(int x){d_x=x;}
          int operator+(Int y){return d_x+y.d_x;}
          void operator+=(Int y){d_x+=y.d_x;}
};
int main()
    { Int a(5); Int b(10);
      cout << ( a+b ); // 15, modo mais usado
      cout << ( a.operator+(b) ); // 15, menos usado
      a+=b;              // a.x=15
      return 0;
    }
```

Operador = e []

```
#ifndef _Ages_h_ // ages.h
#define _Ages_h_
class Ages
{ int *d_ages; int d_size;
  public: Ages(int size); // construtor
          Ages(Ages const &A); // construtor
          ~Ages();  // destrutor
          Ages const &operator=(Ages const &A);
          int size() const;
          int &operator[](int i); // primeira forma
          int operator[](int i) const; // 2a forma
  private: void boundary(int i) const;
           void copy(Ages const &A);};
#endif
```

Operador = e []

- A primeira forma permite modificar os elementos e a segunda permite aceder aos elementos;
- O operador =() deve ser utilizado sempre que uma classe aloca memória;

Exemplo (ficheiro ages.cc)

```
#include "ages.h"
Ages::Ages(int size): d_size(size) // 1o construtor
{ if(d_size<1) { cerr << "Tamanho <1!\n"; exit(1);}
  d_ages=new int[d_size];
}
Ages::Ages(Ages const &A) // 2o construtor
{ copy(A);}
Ages::~Ages() // destrutor
{ delete d_ages; }
int Ages::size()
{ return d_size;}
Ages const &Ages::operator=(Ages const &A)
{ if (this != &A)// evita auto atribuição!
    { delete d_ages;
      copy(A);
    }
    return *this; // devolve o próprio objecto
}
```

Exemplo (ficheiro ages.cc)

```
int &Ages::operator[](int i) // 1o []
    {  boundary(i);
       return d_ages[i];
    }
int Ages::operator[](int i) const // 2o []
    {  boundary(i);
       return d_ages[i];
    }
void Ages::copy(Ages const &A)
    {
       d_size = A.d_size;
       d_ages = new int [d_ages];
       for(int i=0;i<d_size;i++) d_ages[i]=A.d_ages[i];
    }
void Ages::boundary(int i) const
    { if (i >= d_size)
        { cerr << "Ages: overflow, index = " << i << "\n";
          exit(1);
        }
    }
```

Exemplo (ficheiro *main* people.cc)

```
#include <iostream>
#include "ages.h"
using namespace std; // definição do namespace std
int main()
{ Ages x(20);
  for(int i=0; i< 20; i++) x[i] = i*2; // 1o modo
  for(int i=0; i< 20; i++) // 2o modo
    cout <<"pessoa: "<< i <<": idade:"<<x[i]<< "\n";
  Ages y=x;
  for(int i=0; i< 20; i++) // 2o modo
    cout <<"pessoa: "<< i <<": idade:"<<y[i]<< "\n";
  return 0;
}
```

Operadores << e >>

- Os operadores << e >> apenas operam com tipos primitivos, mas podem ser extendidos para novos tipos;
- As respectivas funções devem ser definidas fora do objecto;
- Exemplo (acrescentar ao ficheiro `ages.h`):

```
ostream &operator<<(ostream &out, Age const &x)
{ for(int i=0;i<x.size();i++) // imprime todas
     out << "idade[" << i << "]=" << x[i];
  return out;
}
istream &operator>>(istream &in, Age &x) // lê uma idade
{ int i;
  cout << "Pessoa?";
  if(in >> i) // se conseguiu ler
     { cout << "Idade?"; in >> x[i];}
  return in;
}
```

Exemplo de uso (ficheiro *main* people2.cc)

```
#include <iostream>
#include <fstream>
#include "ages.h"
using namespace std; // definição do namespace std
int main()
{ Ages x(5);
  for(int i=0;i<5;i++) cin >> x; // ler do teclado
  cout << x; // enviar para o ecrã!
  ofstream f;
  f.open("idades.txt");
  f << x;    // enviar para ficheiro "idades.txt"!
  f.close();
 return 0;
}
```

Friends

- Os membros privados são apenas acessíveis dentro do código da própria classe;
- Contudo, a instrução `friend` permite funções externas acederem aos elementos privados;
- Nota: só deve ser aplicado em **casos excepcionais**, para evitar dependência da implementação;
- Aumenta a eficiência dos operadores `<<` e `>>`, devido ao acesso directo aos elementos;

Exemplo de *Friends*

```
class Ages
{ int *d_ages; int d_size;
  public: ... // construtor
  friend ostream &operator<<(ostream &out, Age const &x)
  ...
}
ostream &operator<<(ostream &out, Age const &x) // imprime todas
{ for(int i=0;i<x.d_size;i++)
     out << "idade[" << i << "]=" << x.d_ages[i];
  return out;
}
```

Operador ()

```
#include<iostream>
using namespace std; // definição do namespace std
int const SIN=0; int const COS=1;
class Func{ int d_tipo;
            public:
            Func(int tipo){d_tipo=tipo;}
            double operator()(double const x) const
            { if(d_tipo==SIN) return sin(x);
              else if(d_tipo==COS) return cos(x);
            }
          };
int main()  // operador() lida com objectos como
{Func f(SIN);//se fossem funções, sendo bastante
 cout << f(45.0) << "\n";         // utilizado em
 return 0;                  // algoritmo genéricos
}
```