FEUP/LEEC/AED/2001-2002

Programação com Classes em C++ (Parte 2)

João Pascoal Faria
http://www.fe.up.pt/~jpf

Objectivo da semana

- Criar um tipo de dados (classe) Vector para representar arrays
- Sem os problemas dos arrays *built-in*:
  - o tamanho de um array tem de ser conhecido no momento da compilação (isto é, tem de ser dado por uma constante)
    - solução: usar alocação dinâmica (de forma escondida)
  - o tamanho de um array é fixo (não pode crescer nem encolher)
    - solução: usar alocação dinâmica (de forma escondida)
  - não se podem comparar arrays com operadores de comparação "==" e "!="
    - solução: usar *overloading* de operadores
  - não se podem copiar arrays com operador de atribuição "="
    - solução: usar *overloading* de operadores
  - não é controlado o acesso para fora dos limites do array
    - solução: usar excepções
- Mantendo as vantagens dos arrays *built-in*:
  - arrays de elementos de qualquer tipo
    - solução: usar *templates* de classes
  - acesso aos elementos do array com notação indexada v[i]
    - solução: usar *overloading* de operadores

Revisões sobre Apontadores

Casos de Utilização de Apontadores

- Simular passagem de argumentos por referência com apontadores
  - fundamental só em C
  - em C++ é preferível usar referências em vez de apontadores
- Percorrer eficientemente *arrays* com apontadores
  - não é fundamental
  - normalmente é mais complicado do que usar um índice
  - justifica-se mais para arrays que têm marcas especiais de fim (como é o caso das *strings* do C)
- Manipulação de *arrays* dinâmicos
  - alocados com `new`
  - fundamental para *arrays* cujo tamanho só é conhecido no momento da execução do programa
- Manipulação doutras estruturas de dados alocadas dinamicamente
  - fundamental (a ver mais tarde)
- Manipulação de estruturas de dados ligadas
  - fundamental (a ver mais tarde)

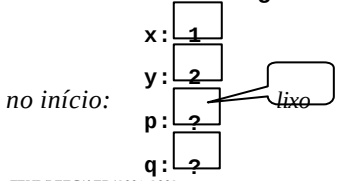
Apontadores e endereços

- Um apontador é uma variável que contém o endereço de um objecto em memória (variável, elemento de array, etc.)
- Operador de endereço: **&**
  &x = endereço do objecto x
- Operador de indireção ou desreferenciação: *****
  ***p** = objecto apontado por p (o endereço do objecto é o valor de p)
- Definição de um apontador para inteiro
  int *p; `// lógica: *p é do tipo int`

Exemplo de manipulação de apontadores

```
int x = 1, y = 2;
int *p, *q;  // p e q são apontadores para int
p = &x;   // coloca em p o endereço de x
          // p agora aponta para x
q = p;    // copia para q o conteúdo de p
          // q agora também aponta para x
y = *p;   // coloca em y o valor do objecto apontado por p
          // y agora vale 1
*p = -1;  // coloca no objecto apontado por p o valor -1
          // x agora vale -1
(*q)++;   // incrementa o valor do objecto apontado por q
          // x agora vale 0
```

*se **p** aponta para **x**, ***p** pode ocorrer em qualquer contexto em que **x** poderia ocorrer*



Apontadores com zero e por inicializar

- A única constante que faz sentido atribuir a um apontador é **`0`** (zero), para significar que não está a apontar para nada (porque é garantido que 0 não é endereço válido para dados)
  - Mesmo que `NIL` em Pascal
  - Podem-se comparar apontadores com a constante **`0`**

- Se um apontador ainda não foi inicializado, não se pode aceder ao objecto apontado (com operador *)

```
float *p2;
*p2 = 1.0; // MAL: p2 não foi inicializado
```

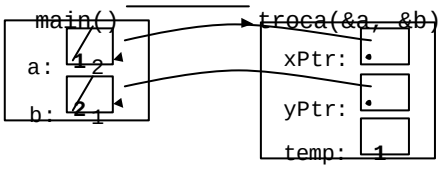
  - Não confundir com:

```
float x, *p1 = &x; // OK: inicializa p1 e não *p1
```


Simulação de argumentos por referência com apontadores

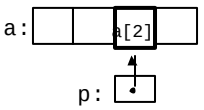
```
// Função troca com apontadores em vez de referências
void troca(int *xPtr, int *yPtr)
{
    int temp = *xPtr;
    *xPtr = *yPtr;
    *yPtr = temp;
}

main() // Testa a função anterior
{
       int a = 1, b = 2;
       troca(&a, &b); // troca valores de a e b
       cout << "a =" << a << '\n';
       cout << "b =" << b << '\n';
}
```

- Passa o endereço da variável
- Permite que a função chamada altere o valor (conteúdo) da variável
- **Pouco útil**, porque é mais simples usar referências!



Operações com apontadores para elementos de arrays

```
int a[4], *p = &a[2];
```



- Pode-se somar ou subtrair um inteiro a um apontador
  pointer + *n* : avança *n* elementos do tipo apontado por *pointer* (? *n* bytes)
  pointer - *n* : recua *n* elementos do tipo apontado por *pointer*
  p+1 dá **&a[3]**
- Pode-se incrementar ou decrementar um apontador
  p++; // **p** fica com **&a[3]**
- Podem-se subtrair apontadores do mesmo tipo (dá a distância entre eles, em nº de elementos do tipo apontado)
  p - &a[0] dá **2**
- Podem-se comparar apontadores do mesmo tipo
  p > &a[0] dá **true**
- Pode-se apontar para o elemento imediatamente a seguir ao fim de um array

Relação entre apontadores e arrays

- O nome de um array pode ser usado como apontador para o seu primeiro elemento!

```
int a[4] = {1, 2, 3, 4};
int *p = a;  // Mesmo que: int *p = &a[0];
p = a + 2;   // Mesmo que: p = &a[0] + 2;
             // ou ainda : p = &a[2];
```

? Quando se passa um array a uma função, de facto passa-se (por valor!) o endereço do seu 1º elemento

Na declaração de argumentos da função, pode-se escrever

```
void geraAposta(int ap[])
```

ou equivalentemente (tornando explícito o que se passa na realidade)

```
void geraAposta(int *ap)
```

- Pode-se usar a notação indexada com apontadores

```
p[1] = 2;  // Mesmo que: *(p+1) = 2;
```

Uso de apontadores para percorrer eficientemente arrays

```
// Devolve o índice da 1ª ocorrência de x no array
// a de tamanho n (devolve -1 se não encontrar)
int procura(int x, const int a[], int n)
{
   for (const int *p1 = a, *p2 = a + n; p1 < p2; p1++)
      if (x == *p1)
           return p1 - a;

   return -1;
}

// É mais eficiente (com um mau compilador ...) do que:
int procura(int x, const int a[], int n)
{
   for (int i = 0; i < n; i++)
      if (x ==  a[i])
           return i;

   return -1;
}
```

mesmo que **&a[0]**

mesmo que *(**&a[0] + i**)

? **envolve mais uma soma que na versão anterior!**

Apontadores e *arrays* dinâmicos

```
// Programa para inverter uma sequência de inteiros
#include <iostream.h>

main()
{
  int n;
  cout << "Quantos valores são? ";
  cin >> n;
  int *v =  new int [n] ;

  cout << "Introduza os valores:\n";
  for (int i = 0; i < n; i++)
     cin >> v[i] ;

  cout << "Valores por ordem inversa:\n";
  for (int i = n - 1; i >= 0; i--)
     cout << v[i] << '\n';

  delete [] v ;
  return 0;
}
```

O ideal era poder declarar aqui:

int v[n];

Só que isso não é possível, porque o tamanho (**n**) não é conhecido no momento da compilação!

Overloading de Operadores

Funções-operador

- Função com nome **`operator`** seguido do símbolo de um operador de C++ define o significado desse operador para objectos de uma classe (ou combinação de classes)
- Exemplo (para colmatar o facto de, por omissão, não se poderem comparar objectos de uma classe com operadores habituais de comparação):

Função amiga – está definida fora da classe mas tem acesso aos membros privados da classe

```
class Data {
  public:
    friend bool operator==(Data a, Data b);
    bool operator==(Data b)
     { return dia == b.dia && mes==b.mes && ano==b.ano; }

  private:
    int dia, mes, ano;
};

bool operator==(Data a, Data b)
{ return a.dia == b.dia && a.mes==b.mes && a.ano==b.ano; }

... Data d1, d2; ... if (d1 == d2) ...
```

Operadores que podem ser redefinidos

- Quase todos os operadores podem ser redefinidos:
  - aritméticos: `+ -` *(unário ou binário)* `* / %`
  - bit-a-bit: `^ & | ~ << >>`
  - lógicos: `! && ||`
  - de comparação: `== != > < <= >=`
  - de incremento e decremento: `++ --` *(pósfixos ou préfixos)*
  - de atribuição: `= += -= *= /= %= |= &= ^= ~= <<= >>=`
  - de alocação e libertação de memória: `new new[] delete delete[]`
  - de sequenciação: `,`
  - de acesso a elemento de array: `[]`
  - de acesso a membro de objecto apontado: `-> ->*`
  - de chamada de função: `()`
- Operadores que não podem ser redefinidos:
  - de resolução de âmbito: `::`
  - de acesso a membro de objecto: `. .*`

**Overloading* de operadores unários

- Um operador unário préfixo (`- ! ++ --`) pode ser definido por:
  1) um membro -função não estático sem argumentos, ou
  2) uma função não membro com um argumento
- Para qualquer operador unário préfixo `@`, `@x` pode ser interpretado como:
  1) `x.operator@()`, ou
  2) `operator@(x)`
- Os operadores unários pósfixos (`++ --`) são definidos com um argumento adicional do tipo `int` que nunca é usado (serve apenas para distinguir do caso préfixo):

  Para qualquer operador unário pósfixo `@`, `x@` pode ser interpretado como:
  1) `x.operator@(int)`, ou
  2) `operator@(x,int)`

Overloading de operadores binários

- Um operador binário pode ser definido por:
  1) um membro-função não estático com um argumento, ou
  2) uma função não membro com dois argumentos
- Para qualquer operador binário `@`, `x@y` pode ser interpretado como:
  1) `x.operator@(y)`, ou
  2) `operator@(x,y)`
- Os operadores `= [] ()` e `->` só podem ser definidos da 1ª forma (por membros-função não estáticos), para garantir que do lado esquerdo está um *lvalue*

Entrada e saída de dados com << e >>


```
class Data {
  public:
    friend ostream & operator<<(ostream & o, const Data & d);
    friend istream & operator>>(istream & i, Data & d);
    // ...
  private:
    int dia, mes, ano;
};

ostream & operator<<(ostream & o, const Data & d)
{
  o << d.dia << '/' << d.mes << '/' << d.ano;
  return o;
}
istream & operator>>(istream & i, Data & d)
{
  char b1, b2;
  i >> d.dia >> b1 >> d.mes >> b2 >> d.ano;
  return i;
}
```

*Exemplo com operador unário

```
class Data {
  public:
     Data & operator++();    // prefixo
     Data & operator++(int); // posfixo
     // ...
   private:
    int dia, mes, ano;
};

Data & Data::operator++()
{
   if (dia ==  numDiasMes(ano,mes)) {
       dia = 1;
       mes = mes == 12? 1 : mes+1;
   }
   else
       dia++;
   return *this;
}

inline Data & Data::operator++(int)
{ return operator++(); }
```

```
main()
{
   Data d1 (30,12,2000);
   cout << d1 << '\n';
   d1++;
   cout << d1 << '\n';
   ++d1;
   cout << d1 << '\n';
   return 0;
}
```

Nota:

```
d2 = d1++;
```

atribui a d2 o valor de d1 já incrementado!

Overloading do operador de atribuição

```
class Pessoa {// (ver construtor de cópia)
    char *nome; // alocado dinamicamente no construtor
  public:
    void setNome(const char *nm) {/*liberta, aloca e copia */}

    Pessoa & operator=(const Pessoa & p)
      { setNome(p.nome); return *this; }

    Pessoa & operator=(const char *nome)
      { setNome(nome); return *this; }

// ...

};

void teste()
{  Pessoa p1 ("Joao");
   Pessoa p2 ("Maria");
   p2 = p1;            // Agora é seguro!
   p2 = "Jose";        // Agora é possível!
}
```

Overloading de operadores de conversão

```
class Pessoa {// (ver construtor de cópia)
    char *nome; // alocado dinamicamente no construtor
  public:
    void setNome(const char *nm) {/*liberta, aloca e copia */}

    operator const char *() const {return nome; }
// ...
};

void teste()
{  Pessoa p1 ("Joao");
   Pessoa p2 ("Maria");
   const char *s = p2; // Agora é possível
}
```


Overloading do operador de função

Membro-função com o nome **operator()** permite usar um objecto como se fosse uma função (neste caso função constante com um argumento e retorno do tipo double)!

```
class Polinomio {
    double *coefs;
    int grau;
  public:
    double operator() (double x) const;
    // ...
};

// Calcula valor de polinómio num ponto x
double Polinomio::operator() (double x) const
{ double res = coefs[grau];
  for (int i = grau-1; i >= 0; i--) res = res * x + coefs[i];
  return res;
}

void teste()
{
  Polinomio pol; cout << "pol? "; cin >> pol;
  double x; cout << "x? "; cin >> x;
  cout << "pol(x)=" << pol(x) << '\n';
}
```

Tratamento de Excepções

Introdução ao tratamento de excepções

- **Tratamento de excepções** fornece uma maneira de transferir controlo e informação de um ponto na execução de um programa, para uma **rotina de tratamento da excepção** (*handler*) associada a um ponto anterior na execução (retorno automático de vários níveis)
- **`throw`** *`objecto`*`;`
  - lança uma excepção (objecto) transferindo o controlo para o *handler* mais próximo na *stack* de execução capaz de apanhar excepções (objectos) do tipo do objecto lançado
- **`try`** { ... }
  - executa bloco de instruções, sendo excepções apanhadas pelo(s) próximo(s) `catch`
- **`catch`** (*`tipo-de-objecto`* [*`nome-de-objecto`*]) `{ ...}`
  - *handler;* apanha excepções do tipo indicado lançadas com `throw` no bloco de `try` ou em funções por ele chamadas
- A mudança de controlo obriga a desfazer a *stack* e a chamar os destrutores para todos os objectos automáticos construídos desde o **`try`**

Tratamento de excepções manual *versus* automático

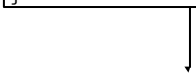
Manual

```
void f1()
{
   ...
   if (f2()==ERRO)
        goto TRATA_ERRO;
   ....
   return;

TRATA_ERRO:
   MsgBox("Erro .... ");
   ...
}
```

...

Automático

```
void f1()
{
   try
   {
      ...
      f2();
      ....
   }

    catch(ERRCOD e)
    {
       MsgBox("Erro ...");
       ...
    }
}
```



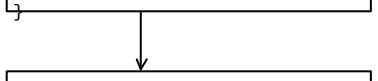
...

Tratamento de excepções manual *versus* automático (cont.)

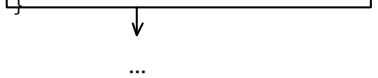
Manual

...

```
ERRCOD f2()
{
   ...
   if (f3()==ERRO)
        return ERRO;
   ....
   return OK;
}
```



```
ERRCOD f3()
{
   ...
   if (f4()==ERRO)
        return ERRO;
   ....
   return OK;
}
```

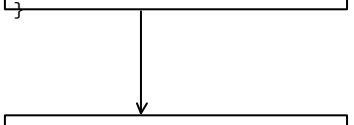


...

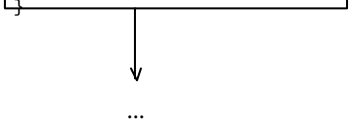
Automático

...

```
void f2()
{
   ...
   f3();
   ....
}
```



```
void f3()
{
   ...
   f4();
   ....
}
```



...

Tratamento de excepções manual *versus* automático (conc.)

Manual

```
ERRCOD f4()
{
   ...
   if (situação de erro detectada)
        return ERRO;
   ....
   return OK;
}
```

Automático

```
void f4()
{
   ...
   if (situação de erro detectada)
        throw ERRO;
   ....
}
```

Conclusões:

- Com tratamento automático, só a função que detecta o erro (`f4`) e a função que pretende reagir ao erro (`f1`) é que têm código relacionado com o erro
- As funções intermédias na *stack* de chamada (`f2`, `f3`) não têm de fazer nada
- O lançamento (**`throw`**) do erro provoca o retorno automático (**`return`**) até à função que pretende reagir ao erro (intenção sinalizada com **`try`**) e um salto automático (**`goto`**) para o bloco de tratamento do erro (**`catch`**)

Exemplo com tratamento de excepções

```
class Data {
    int dia, mes, ano;
  public:
    class DiaInvalido { }; // define classe (tipo) de excepções
    void setDia(int dia);
    //...
};

void Data::setDia(int d)
{   if (d < 1 || d > 31)
        throw DiaInvalido(); // salta fora!
    dia = d;
}

main()
{   Data d;
    try {
       d.setDia(100);
    }
    catch (Data::DiaInvalido) {
      cout << "enganei-me no dia!!\n";
    }
}
```

lança objecto da classe `DiaInvalido` criado com chamada de construtor por omissão

handler

*Agrupamento de excepções

- `catch ( ... )` - apanha uma excepção de qualquer tipo
- `catch ( T )` - apanha uma excepção do tipo `T` ou de um tipo `U` derivado publicamente de `T`
- `catch ( T * )` - apanha uma excepção do tipo `T *` ou do tipo `U *`, em que `U` é derivado publicamente de `T`
- `catch ( T & )` - apanha uma excepção do tipo `T &` ou do tipo `U *`, em que `U` é derivado publicamente de `T`

*Exemplo com agrupamento de excepções

```
class Matherr { };
class Overflow  : public Matherr {};
class Underflow : public Matherr {};
class Zerodivide: public Matherr {};

void f()
{
    try {
       // operações matemáticas
    }
    catch (Overflow) {
       // trata excepções Overflow ou derivadas
    }
    catch (Matherr) {
       // trata excepções Matherr que não são Overflow
    }
    catch (...) {
      // trata todas as outras excepções (habitual no main)
    }
}
```

Excepções com argumentos; **re-throw*

```
class Data {
    int dia, mes, ano;
  public:
    struct DiaInvalido { int dia; DiaInvalido(int d) {dia = d;} };
    void setDia(int dia);
    //...
};

void Data::setDia(int d)
{   if (d < 1 || d > 31)  throw DiaInvalido(d);
    dia = d;
}

void f()
{   Data d;
    try {
       d.setDia(100);
    }
    catch (Data::DiaInvalido x) {
      cout << "dia inválido: " << x.dia << endl;
      throw;  // sem parêntesis => re-throw
    }
}
```


*Tópicos adicionais

- Declaração de excepções lançadas por função:

```
void data::setDia(int d) throw(DiaInvalido);
void data::getDia(int d) throw(/*nenhuma*/);
void data::setDate(int d, int m, int a) throw(DiaInvalido,
  MesInvalido, Ano Invalido, DataInvalida);
```

  (por omissão: pode lançar qualquer)

- Todo o corpo de uma função pode estar dentro de `try`:

```
void f() try { /* corpo*/ } catch ( ) { /*handler*/}
```

- Excepções não apanhadas fazem abortar o programa

- Excepções *standard* (hierarquia definida em <exception>)
  - bad_alloc, bad_cast, bad_typeid, bad_exception, out_of_range, invalid_argument, overflow_error, ios_base::failure