Aula 2: Introdução ao C++

Paulo Cortez

2005/2006

Técnicas de Programação I
Licenciatura em Engenharia Electrónica e Computadores

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

Objectivos da Aula

- Descrever as origens e vantagens do C++;
- Explicar as diferenças entre o C e o C++;
- Identificar as principais extensões do C++;

História do C++

- Primeira implementação nos anos 80, no AT&T Bell labs;
- Utilizava um pré-código, lido por um pré-compilador;
- C++ visto como um superconjunto do C, oferecendo todas as possibilidades do C, facilitando a transição do C para o C++;
- Os ficheiros tinham a extensão: `.cc`, `.C` ou `.cpp`;
- Hoje em dia os compiladores reconhecem automaticamente o código fonte escrito em C++: Borland e Microsoft utilizam a extensão `.cpp`; os compiladores GNU (por exemplo, cygwin) usam a extensão `.cc`.

Vantagens/Pretensões

- Programas novos são desenvolvidos em menos tempo, devido à reutilização de código;
- É mais fácil criar e manipular novos tipos de dados;
- A gestão de memória é mais simples e transparente;
- Os programas conterão menos erros, devido à maior exigência da verificação de erros por parte do compilador;
- Facilita o encapsulamento de dados;

A verdade é que o C++ (bem como a programação OO - Orientada por Objectos) não é uma solução universal para todos os problemas de programação!

Diferenças C/C++: Comentários, Namespaces, Apontadores NULL

- Comentários: `// comentário até ao fim da linha`
  `/* os comentários em C também são válidos */`
- Namespaces:
  - Permite definir símbolos num contexto alargado, chamado de *namespace*;
  - Utilizados para evitar conflitos entre funções;
  - Para já importa saber que a maior parte dos compiladores exige a seguinte linha de código, a seguir aos includes: `using namespace std;`
- Apontadores NULL vs 0:
  - Não existe a necessidade do termo `NULL`, basta utilizar o valor `0`, que é equivalente;

Conversão estática

- Serve para converter um tipo de dados em outro tipo de dados;
- Uso: `static_cast<tipo>(exp)` ;
- Exemplos:
  - `int a = static_cast<int>(12.45);`
  - `media = static_cast<float>(45/27);`
  - `char c = static_cast<char>(a);`

Uso de funções C

- Funções escritas em puro C devem ser declaradas do seguinte modo:
  `extern "C"função` ou `extern "C"{ /* código C */ }`
- Exemplos:
  `extern "C"{ #include "lista_ligada.h"}`
  `extern "C"void imprime(char *s);`

Definição de variáveis locais

- Em C, as variáveis locais tem de ser definidas no início de cada função. No entanto, em C++, as variáveis podem ser definidas em qualquer local!

```
int main()
{ for(int i=0;i<10;i++)
     for(int j=0;j<i;j++);
}
```

- Atenção: neste caso, as variáveis i e j só são válidas dentro dos ciclos!
- Regra: variáveis locais devem ser definidas no início das funções, após o primeiro { OU imediatamente antes de serem necessárias, mas com o devido cuidado;

Sobreposição de funções (*Function Overloading*

- É possível definir funções diferentes com os mesmos nomes;
- Contudo, as funções devem-se distinguir pelos argumentos de entrada:

```
# include <stdio.h>
void show(int val){ printf("%d\n", val);}
void show(double val){ printf("%lf\n", val);}
void show(char *val){ printf("%s\n", val);}
int main()
{ show(12); show(3.1415);show("bom dia");}
```

Regras para funções

- Deve ser evitado o uso de funções com o mesmo nome mas com acções diferentes;
- Funções com o mesmo nome não se podem distinguir pelo valor de retorno;
- A sobreposição de funções pode causar surpresas. Por exemplo, o que faz a seguinte instrução: `show(0)` ? O valor 0 pode ser interpretado como um inteiro ou apontador nulo!

Argumentos implícitos (*Default*) em funções

- É possível sugerir valores implícitos para argumentos de funções, que são activados quando o utilizador não os especifica:

```
void show(char *str="Bom dia!\n")
{ printf(str);}
int main()
{ show("Texto explicito!!!\n");
  show(); // igual a show("Bom dia!\n");
}
```

Regras para argumentos implícitos

- Funções podem ter mais do que um argumento implícito;
- Os argumentos implícitos devem ser declarados nos protótipos das funções (ficheiros `.h` ou `.hpp`):

```
// protótipo em ficheiro .h ou .hpp
void dois(int a=1, int b=4);
// código da função, em ficheiro .cc ou .cpp
void dois(int a, int b){...}
```

- É assumida uma leitura a partir da direita para a omissão:

```
void dois(int a = 1, int b = 4){ ... }
int main()
{ dois();       // argumentos: 1, 4
  dois(20);     // argumentos: 20, 4
  dois(20,10);}// argumentos: 20, 10
```

Instrução typedef

- A instrução typedef é permitida em C++, mas não é necessária antes de uma estrutura:

```
struct paciente{ int idade; char * nome; };
paciente p; p.idade=80;
```

Extensões ao C: operador de escopo

- Operador de escopo ::
  - Pode ser utilizado para distinguir variáveis locais de globais:

```
# include <stdio.h>
int conta = 50; // variável global
int main()
{   for(int conta = 1; conta <10; conta++) // local
       { printf("\%d\n", ::conta  // global
                                / conta);// local
       }
    return (0);
}
```

Entrada/Saída de dados (*Streams*)

- Existem 3 "modos" especiais: `cin` (entrada), `cout` (saída) e `cerr` (saída do erro);
- Actua-se sobre cada emissor/receptor (*stream*) utilizando os operadores: << - de inserção e >> - de extracção.

```
#include <iostream> // biblioteca de streams
using namespace std; // definição do namespace std
int main()
 { int a; char str[30];
   cout << "Valor inteiro?\n"; cin >> a; // ler a
   cout << "Texto ?"; cin >> str // ler str
   cout << "Eis o numero: " << a << endl
        << "E a palavra: " << str << endl;
 }
```

Regras com *Streams*

- É necessário incluir o `iostream`, bem como a definição do *namespace* `std`;
- Os operadores << e >> utilizam uma sobreposição de argumentos, ou seja, lidam com diferentes tipos de dados;
- O operador >> ignora todos os espaços que precedem o valor a ser extraído;
- A constante `endl` efectua a mudança de linha;
- Tanto pode ser utilizada a biblioteca `iostream` como a `stdio.h`, ou seja, pode-se até misturar `printf` e `scanf` com `cout << ` e `cin >>`;
- No entanto à vantagens no uso de *streams*:
  - É mais seguro usar os operadores << e >>, evitando-se erros;
  - Os operadores << e >> são extensíveis, ou seja, a sua funcionalidade pode ser melhorada;

Constantes

- A palavra `const` define uma constante (não pode ser alterada);
- Embora também se utilize em C, em C++é muito mais utilizada;
- Exemplos:

```
int const a=5;
double const pi=3.1415;
void imprime(char const *s);
```

Referências

- Forma mais simples para a passagem por referência em C:

```
void by_pointer(int *valp) // como em C
  { *valp += 5; }
void by_reference(int &valr) // nova forma
  { valr += 5; }
int main ()
  { int x; by_pointer(&x);  // passagem de apontador
           by_reference(x); // por referência
    return 0; }
```

Funções como parte de estruturas

- Uma função pode fazer parte de uma estrutura;
- Estas funções são designadas *membros* ou *métodos*;

```
#include <iostream>
#include <string.h>
using namespace std; // definição do namespace std
struct pessoa
  { char nome [80], morada [80];
    void print (void); // declaração da função
  };
void pessoa::print() // :: operador que designa estrutura
  { cout << "Nome:" << nome << " Morada:" << morada << endl;}
int main()
  { pessoa p; strcpy(p.nome, "Rui");
    strcpy(p.morada, "Rua Vitor, 37"); p.print();
  }
```

Novos tipos de dados

- Tipos em C: `void`, `char`, `short`, `unsigned`, `int`, `long`, `float` e `double`;
- Em C++, os mesmos mais: `bool`, `wchar_t` e `long double`:
  - `bool` - valor lógico, com os valores `true` ou `false`;

```
bool a=false;
if(a) cout << "Verdade" << endl;
else cout << "Falso" << endl;
```

  - `wchar_t` - extensão ao tipo `char`, que permite lidar com mais tipos de caracteres (até 65536 caracteres diferentes!);
  - `long double` - do tipo `double` mas com maior precisão;