

Transição de C para C++

Programação (L.EIC009)

Eduardo R. B. Marques, DCC/FCUP

Introdução

Antes de entrarmos na programação orientada-a-objects em C++, expomos alguns aspectos da linguagem que a diferenciam da linguagem C.

Os aspectos em causa podem ser vistos como extensões à programação imperativa em C, de que fazemos uma introdução sumária:

- “Overloading” de funções
- Uso de “namespaces”
- Referências
- “Overloading” de operadores
- Ciclos “for-each”
- Uso de “templates”
- Alocação de memória com `new` e `delete`
- Outros aspectos: `auto`, valores por omissão para parâmetros de funções.

Alguns destes tópicos serão cobertos em mais detalhe depois.

“Overloading” de funções

“Overloading” de funções

```
int f(int a) {  
    return a / 2;  
}  
  
int f(int a, int b) {  
    return a / b;  
}  
  
double f(double a, double b) {  
    return a / b;  
}
```

“Overloading” de funções: podemos definir várias funções com o mesmo nome, desde que não haja ambiguidade entre as declarações.

“Overloading” de funções (cont.)

Uso do código anterior:

```
int main(void) {  
    std::cout << f(9) << ' '  
                << f(9,2) << ' '  
                << f(9.0, 2.0) << ' '  
                << std::endl;  
  
}
```

Output:

4 4 4.5

Use de “function overloading” (cont.)

```
std::cout << f(9) << ' ' << '\n';
std::cout << f(9,2) << ' ' << '\n';
std::cout << f(9.0, 2.0) << ' ' << '\n';
std::cout << std::endl;
```

4 4 4.5

```
int f(int a) {
    return a / 2;
}

int f(int a, int b) {
    return a / b;
}

double f(double a, double b) {
    return a / b;
}
```

Uso de “namespaces”

Uso de “namespaces”

Exemplo:

```
namespace a {  
    int f() { return 1; }  
    namespace b {  
        int f() { return 2; }  
    }  
}  
  
namespace c {  
    int f() { return 3; }  
}  
  
int f() { return 4; }
```

O uso de “namespaces” permite-nos agrupar declarações em espaços de nomes distintos, evitando colisões entre declarações similares. Podemos ter namespaces dentro de outros, formando então uma hierarquia de nomes.

Uso de “namespaces” (cont.)

Uso do código anterior:

```
int main() {  
    int v = a::f()      // 1  
        + a::b::f()    // + 2  
        + c::f()       // + 3  
        + f();         // + 4  
    std::cout << v << std::endl; // 10  
    return 0;  
}
```

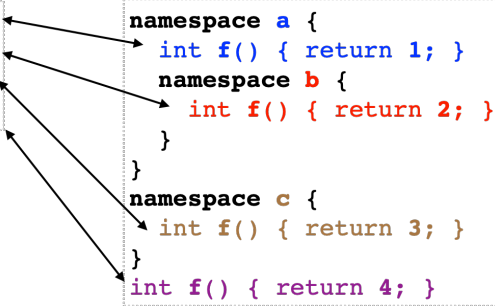
Output (1 + 2 + 3 + 4):

10

Uso de “namespaces” (cont.)

```
int v = a::f()      // 1
      + a::b::f()   // + 2
      + c::f()      // + 3
      + f();        // + 4
```

```
namespace a {
    int f() { return 1; }
    namespace b {
        int f() { return 2; }
    }
    namespace c {
        int f() { return 3; }
    }
    int f() { return 4; }
```



Referências

Referências

Em C++ podemos declarar uma variável como uma **referência**, ex.

```
int v = 10;  
int& r = v;  
r = 123; // escreve 123 em v
```

Tal como um apontador, uma referência contém o valor de um endereço de memória de determinado tipo (`int` acima). Mas as referências são mais “simples” ...

Referências (cont.)

```
int v = 1;
int& r = v;
int* p = &v;
r += 123;    // soma 123 a v
*p += 124;   // soma 124 a v
std::cout << v << ' '
          << r << ' '
          << *p << std::endl;
```

Output (p e r referem-se a v):

248 248 248

Não precisamos de usar o operador de dereferenciação (*) ou de endereço (&) com referências. Não há também operadores de aritmética em associação a referências, ao contrário de apontadores.

Referências (cont.)

Uma referência tem de ser inicializada e isso é feito apenas uma vez.

```
int v = 0;
int v2 = 2;
int& r = v;
std::cout << v << std::endl;
// Incrementa v e não o endereço guardado em r
r++;
std::cout << v << std::endl;
// Atribui a v o valor de v2
r = v2;
std::cout << v << std::endl;
```

Output:

0
1
2

Argumentos por referência

Em C ou C++ podemos usar apontadores para passagem de argumentos por referência. De aula anterior:

```
void min_max_sum
(int a, int b, int* min, int* max, int* sum) {
    *min = a < b ? a : b;
    *max = a > b ? a : b;
    *sum = a + b;
}

void some_func(void) {
    int maior, menor, s;
    min_max_sum(-123, 123, &menor, &maior, &s);
    ...
}
```

Argumentos por referência (cont.)

Em C++ (mas não em C) podemos usar referências para o mesmo efeito:

```
void min_max_sum
(int a, int b, int& min, int& max, int& sum) {
    min = a < b ? a : b;
    max = a > b ? a : b;
    sum = a + b;
}

void some_func(void) {
    int maior, menor, s;
    min_max_sum(-123, 123, menor, maior, s);
    ...
}
```


Outro exemplo - leitura com `std::cin`

O objecto `std::cin` serve para leitura de dados, de forma análoga a `stdin` em C (ex. usado implicitamente com `scanf`). Quando usamos o operador `<<` com `std::cin`, estamos a invocar uma das definições de “overloading” para o operador na classe `istream`:

```
std::cout << "Enter two numbers: " << std::endl;  
int a, b;  
std::cin >> a >> b;
```

A função invocada é a de “overloading” do **operador `>>` em `istream`** para valores de tipo `int`:

```
istream& operator>> (int& val);
```

Referências vs. apontadores - sumário

Referências:

- mais simples de usar sintacticamente
- mais seguras de usar, dadas as restrições de definição e uso

Apontadores:

- mais expressivos graças à aritmética de apontadores (que se mantém em C++)
- necessários de qualquer forma para lidarmos com arrays em muitos casos e memória alocada dinamicamente

“Overloading” de operadores

“Overloading” de operadores

Em C++ podemos definir o uso de operadores em associação a tipos de dados arbitrários. Exemplo - operadores + e * para um tipo coord2d:

```
struct coord2d {  
    int x, y;  
};  
  
coord2d operator+(const coord2d& a, const coord2d& b) {  
    coord2d r;  
    r.x = a.x + b.x;  
    r.y = a.y + b.y;  
    return r;  
}  
  
coord2d operator*(int f, const coord2d& a) {  
    coord2d r;  
    r.x = f * a.x;  
    r.y = f * a.y;  
    return r;  
}
```

“Overloading” de operadores (cont.)

Uso dos operadores anteriormente definidos:

```
#include <iostream>
int main() {
    coord2d a = { 1, 2 };
    coord2d b = { 3, 4 };

    // (1,2) + 2 * (3,4) = (7, 10)
    coord2d c = a + 2 * b;
    std::cout << c.x << ' ' << c.y << std::endl;
}
```

Output:

7 10

Ciclos “for-each”

Ciclos “for-each”

Para arrays ou objectos iteráveis (a ver depois) podemos empregar ciclos “**for-each**” (“para cada”) com a forma geral:

```
for (tipo var: iterable) {  
    loop_body  
}
```

O ciclo executa para cada elemento *v* que for parte de *iterable*.

Tipicamente conseguimos desta forma código mais sucinto e menos sujeito a erros de programação, porque dispensamos variáveis para indexação.

Ciclos “for-each” (cont.)

Exemplo:

```
int a[5] = { 1, 2, 3, 4, 5 };  
for (int v: a) {  
    std::cout << v << std::endl;  
}
```

O código acima é equivalente a:

```
int a[5] = { 1, 2, 3, 4, 5 };  
for (int i = 0; i < 5; i++) {  
    int v = a[i];  
    std::cout << v << std::endl;  
}
```


Ciclos “for-each” (cont.)

A seguinte forma geral de ciclo “for-each” permite-nos não só ler como modificar o conteúdo de `iterable` usando a referência `var`.

```
for (tipo& var: iterable) {  
    loop_body  
}
```

Em cada iteração `var` é uma referência a um elemento que pode ser modificado.

Ciclos “for-each” (cont.)

Exemplo:

```
int a[5] = { 1, 2, 3, 4, 5 };  
for (int& v: a) {  
    v++;  
}
```

O código acima é equivalente a:

```
int a[5] = { 1, 2, 3, 4, 5 };  
for (int i = 0; i < 5; i++) {  
    a[i]++;  
}
```

“Templates”

“Templates”

C++ permite a definição de código “template” através de definições do género:

```
template <typename T>  
...
```

em que T designa um tipo genérico a instanciar. O compilador gera depois código apropriado por cada instanciação concreta do “template”.

Entre outras variações, podemos também ter mais do que um tipo:

```
template <typename T, typename U>  
...
```

A seguir vamos ver exemplos simples de uso “templates” para funções e tipos de dados `struct`.

Funções “template”

Definição:

```
template <typename T>
void min_max_sum
(T a, T b, T& min, T& max, T& sum) {
    min = a < b ? a : b;
    max = a > b ? a : b;
    sum = a + b;
}
```

Uso:

```
int imin, imax, isum;
min_max_sum(-123, 123, imin, imax, isum);

double dmin, dmax, dsum;
min_max_sum(-12.5, 7.4, dmin, dmax, dsum);
```

Tipos “template”

Exemplo - declaração de `struct` para coordenadas 2D:

```
template <typename T>
struct coord2d {
    T x;
    T y;
};
```

Usamos `coord2d<t>` para declarar variáveis em que o tipo `t` é empregue para os campos `x` e `y`:

```
coord2d<int> icoords = { 1, 2 };
coord2d<double> dcoords = { 1.5, -1.3};
```

Tipos e funções template (cont.)

Em conjunto com `coord2d<T>` podemos definir funções “template”, por ex.:

```
template <typename T>
coord2d<T> midpoint(const coord2d<T> arr[], int n) {
    coord2d<T> m = { 0, 0 };
    for (int i = 0; i < n; i++) {
        m.x += arr[i].x; m.y += arr[i].y;
    }
    m.x = m.x / n; m.y = m.y / n;
    return m;
}

template <typename T>
coord2d<T> operator*(T f, const coord2d<T>& a) {
    coord2d<T> r = { f * a.x, f * a.y };
    return r;
}
```

Tipos e funções template (cont.)

Exemplo de uso do código anterior:

```
coord2d<double> dcoords = { 1.5, -1.3};  
dcoords = 2.5 * dcoords;
```

```
coord2d<int> ica[4] = { {1, 2},  
                        {2,4},  
                        {3,6},  
                        {4, 8}};  
coord2d<int> imid = midpoint(ica, 4);
```


Alocação dinâmica de memória com `new` e `delete`

new e delete

Em alternativa a `malloc` e `free`, em C++ podemos empregar os operadores `new` e `delete`. A sintaxe geral pr dados de tipo `T` ou arrays de tipo `T` é a seguinte:

new

```
type* p = new T;  
type* arr = new T[tamanho];
```

delete

```
delete p;  
delete [] arr;
```

Exemplos

```
int n = ...;
int* arr = new int[n];
coord2d* c = new coord2d;
coord2d* vc = new coord2d[n + n];
...
delete [] arr;
delete c;
delete [] vc;
```

`new / delete` vs. `malloc/free`

Devemos ter cuidados análogos ao uso de `malloc` e `free`:

- memória alocada com `new` deve ser libertada com `delete` para não termos “memory leaks”;
- após libertação com `delete`, o segmento de memória não deve ser referenciado para não termos “dangling references”;

Além disso:

- temos de usar `new` e `delete` em associação a objectos (a ver em futuras aulas), para serem respeitadas as convenções de construção e destruição de objectos.

`new / delete` vs. `malloc/free` (cont.)

- `new` e `delete` são construções de mais alto nível (operadores) por comparação a `malloc` e `free` (funções de baixo nível). Não existe no entanto operador para re-alocação de memória, i.e., que tenha funcionalidade análoga a `realloc`.
- Com `new` especificamos o tipo dos dados e no caso de arrays também a dimensão do array a criar, em vez do número de bytes a alocar com `malloc/calloc/realloc`. O compilador infere o tamanho dos segmentos a alocar.

Exemplo de programa completo

```
#include <iostream>

int main(void) {
    int n;
    std::cout << "Array size:" << std::endl;
    std::cin >> n;
    int* arr = new int[n]; // ALOCAÇÃO DE MEMÓRIA
    std::cout << "Enter values:" << std::endl;
    for (int i = 0; i < n; i++)
        std::cin >> arr[i];
    for (int i = 0; i < n; i++)
        std::cout << i << " : "
                    << arr[i] << std::endl;
    delete [] arr; // LIBERTAÇÃO DE MEMÓRIA
    return 0;
}
```

Exemplo - stack genérica

Veja `stack.hpp` para a definição de uma stack template com as operações da Ficha 4.

```
template <typename T>
struct stacknode {
    T value;
    stacknode* next;
};

template <typename T>
struct stack {
    int size;
    stacknode<T>* top;
};
```

Em próximas aulas, veremos como será mais apropriado usar classes (com `class`) em vez de `struct`.

Exemplo - stack genérica (cont.)

Alocação com new:

```
template <typename T>
stack<T>* stack_create(void) {
    stack<T>* s = new stack<T>;
    s -> size = 0;
    s -> top = NULL;
    return s;
}

template <typename T>
void stack_push(stack<T>* s, int v) {
    stacknode<T>* new_node = new stacknode<T>;
    new_node -> value = v;
    new_node -> next = s -> top;
    s -> top = new_node;
    s -> size++;
}
```


Exemplo - stack genérica (cont.)

Libertação de memória com delete:

```
template <typename T>
void stack_destroy(stack<T>* s) {
    stacknode<T>* n = s -> top;
    while (n != NULL) {
        stacknode<T>* aux = n -> next;
        delete n;
        n = aux;
    }
    delete s;
}
```

(veja também `stack_pop`)

Outros aspectos

Uso de auto

A partir do C++ 2011, a palavra-chave `auto` pode ser empregue em vez *de um tipo para uma variável*.

O compilador tem a responsabilidade de inferir o tipo apropriado.

```
auto x = 1.3; // implicitamente double
auto y = 12;  // implicitamente int
char z = 'a';
auto w = z;   // mesmo tipo que z
```

Esta facilidade é por exemplo útil quando os identificadores de tipos são verbosos (o que pode acontecer facilmente em alguns casos como veremos depois).

Uso de auto - outro exemplo

Em stack.hpp

```
template <typename T>
void stack_to_array(const stack<T>* s, T array[]) {
    auto node = s -> top; // em vez de stacknode<T>* s
    auto pos = 0; // em vez de int
    while ( node != NULL ) {
        array[pos] = node -> value;
        node = node -> next;
        pos++;
    }
}
```

Valores “default” para parâmetros

Podemos declarar valores por omissão (i.e., “default values”) para parâmetros de funções.

Por exemplo, a seguinte função `f` define o valor 1 por omissão para `b`:

```
int f(int a, int b = 1) {  
    return a * b;  
}
```

Se escrevermos

```
int v = f(0) + f(0,2);
```

isso é equivalente a termos

```
int v = f(0,1) + f(0,2);
```

Tipos `struct` e `enum`

Podemos usar `struct` e `enum` como em C, mas sem necessidade de `typedef` para código mais sucinto (o que poderá ter passado despercebido nos anteriores exemplos com `coord2d` ou `stack`!).

Definição:

```
struct coord2d {  
    int x;  
    int y;  
};  
  
enum MONTH {  
    JANUARY, FEBRUARY, ... /* etc */  
};
```

Uso (sem necessidade de `typedef`):

```
coord2d c;  
MONTH m;
```