

PROGRAMAÇÃO MIEIC-FEUP – NOTAS PARA EXAME

IMPORTANTE: para usar estes apontamentos no exame convém ter um conhecimento prévio da matéria, senão podem não perceber alguns destes pontos.

OPERATORS

- Divisão: Quando ambos os argumentos são inteiros, resultado é o quociente. (ex: 3/2 resulta em 1, 3.0/2.0 resulta em 1.5)
- Igual: Sinal de igual é usado para atribuir valores, para fazer comparações deve-se usar ==.

IF STATEMENT

- O valor de uma expressão é interpretado como verdade se for != 0, caso contrário é falso.
- Por isso é que se deve estar atento a usar if(x=0), por exemplo. Deve ser if(x==0)

- NOTE 2: a very common error (not detected by the compiler)

```
if (x = 10)
{
    ...
}
```

- will assign 10 to x
- and the value of (x = 10) is 10 (true)
- so... **BE CAREFUL!**

- NOTE 3: if written in this way the compiler will detect the error 😊

```
if (10 = x)
{
    ...
}
```

SWITCH CASE STATEMENT

- Valor testado deve ser um inteiro ou carater
- Cada case deve ser terminado por break;

BREAK & CONTINUE

- Para sair de um ciclo usar o comando break;
- Para continuar para a próxima iteração de um ciclo sem executar o que estiver no restante código desse ciclo, usar continue;

INVALID INPUTS

- Quando o utilizador insere por exemplo "1o " e o valor pedido é um inteiro, o programa pode entra em loop infinito, etc. Para corrigir, testar `cin.fail()`, e usar `cin.ignore(1000, '\n')` e `cin.clear()`.
- `Cin` deixa o `'\n'` no buffer, `getline` não.
- Útil:
 - o `cin.eof();` //para terminar ciclo de input com CTRL-Z
 - o `cin.fail();` //verifica estado do buffer de entrada
 - o `cin.ignore(1000, '\n');` // limpa o buffer de entrada
 - o `cin.clear();` //permite continuar a fazer input em caso de erro

CALL-BY-VALUE AND CALL-BY-REFERENCE

- Usar call-by-value se não queremos mudar as variáveis, caso contrario usar by-reference.
- Dica para performance: usar(exemplo) `const vector<int> &vec`.

STATIC STORAGE

- Existe em toda a duração do programa. Ou declarar como variável global ou usar key word `static`.
- Efeito: variável apenas é inicializada uma vez, independentemente de existir ciclos, etc.

FUNCTION OVERLOADING

- Quando há mais do que uma função com o mesmo nome MAS numero diferente de parâmetros ou tipo diferente de parâmetros e retorno

```
int sum(int x, int y)
{
    //cout << "sum1 was called\n";
    return x+y;
}

double sum(int x, double y)
{
    //cout << "sum2 was called\n";
    return x+y;
}

double sum(double x, double y)
{
    //cout << "sum3 was called\n";
    return x+y;
}
```

ARRAYS

- Largura invariável.
- Inicialização: `int a[3] = {11, 19, 12};`
- `a` retorna o endereço do primeiro elemento, tal como `&a[0]`.
- Não é possível atribuir um array a outro com um simples comando (ex `a1=a2`).

- Numa função é preciso passar sempre o número de elementos do array como outro parâmetro.
- Não é possível retornar arrays

CONST

- Garante que objeto ou variável não é modificado

VECTORS

- Declarar: `vector<int> v1;`
- `vector<double> v2(10);` //10 elementos a 0
- `vector<int> v3(5,1);` //5 elementos a 1
- `vector<int> v4 = {10,20,30};`
- fazer sempre `#include <vector>` e `using namespace std;`
- se aceder a elementos out-of-range, `at(i)` deteta o erro, `[i]` não
- Podem-se inserir elementos no meio do vetor com `v1.insert(v1.begin()+index)`
- Útil:
 - o `Vec.push_back(elem);` //coloca elem no final do vetor
 - o `Vec.pop_back();` //remove o ultimo elemento do vetor
 - o `Vec.clear();` // apaga todos os elementos do vetor
 - o `Vec.size();` // retorna tamanho do vetor, útil em ciclos for
 - o `Vec.resize(numElems);` // redimensiona o vetor

C – STRINGS

- Arrays de chars : `char s[10];`
- `#include <cstring>`
- Terminam com o carater nulo (e preciso alocar espaço para ele)
- Inicializar: `char salut[] = "Hi!";`
- Ler do teclado: `cin` apenas lê uma palavra, `cin.getline(palavra)` lê uma linha inteira ou `cin.getline(palavra, len)` só lê até len
- Para fazer assignment, usar `strcpy()`, tipo `char msg[10]; strcpy (msg, "Hello");`
- Útil:
 - o `Strcmp(cstr1,cstr2)` //compara as strings e devolve um booleano
 - o `strcat(cstr1,cstr2)` //concatena as strings
 - o `atoi(str)` //convert str para um int

STRINGS

- `#include <string>` e `using namespace std;`
- Declarar: `string name("John");` ou usar `=`, ou não atribuir valor sequer;
- Pode-se aceder aos elementos com `.at()` e `[]`
- Comparar com `==`, concatenar com `+`, tamanho com `name.length()`
- Ler linhas com `getline(cin,name);`
- Importante: `string::npos`, usado para testar se se encontram uma string ou char noutra string
- `Name.substr(pos1,pos2)` – substring

- `Name.find(str1)` retorna índice da primeira ocorrência de `str1` em `name`, ou `npos`
- `Name.find(str1, pos)` mesmo, mas procura começa em `pos`
- `Name.find_first_of(str1, pos)`
- `Name.find_first_not_of(str1, pos)`
- `string s(n, char)` cria uma string com `n` ocorrências de `char`

POINTERS

- Guarda um endereço de memória (aponta para outra variável)
- Declarar: `T *apont` (`apont` aponta para uma variável do tipo `T`)
- `&` retorna o endereço de, `*` retorna o valor do endereço dado
- Quando se incrementa o apontador, ele passa a ter a próxima posição de memória do seu tipo. EX: `apont++` e `apont=1000` e `T=int`, `apont` passa a ser `1000+4`.
- Relação próxima entre pointers e arrays.
- `char s[] = "Hello!"`; é o mesmo que `char *s = "Hello!"`; mas a segunda string não pode ser modificada

DYNAMIC MEMORY ALLOCATION C

- `#include <stdlib>`, `malloc(num bytes)`, `free()`
- `malloc` aponta para primeiro byte alocado, se não houver memória suficiente retorna um `NULL` pointer
- `free(void *p)` retorna a memória alocada anteriormente para o sistema, `p` é um apontador para essa memória

DYNAMIC MEMORY ALLOCATION C++

- `#include <new>`, `new` e `delete`
- `int *p = new int(0)`; inicializa o `int` apontado por `p` a `0`
- Alocar arrays: `int *p = new int[10]`; alocar 10 inteiros – `delete [] p`

FILES

- `#include <fstream>` e `using namespace std`;
- Abrir ficheiro: `ifstream in_stream`; -- `in_stream.open(filename)` – `if(!in_stream.fail())` – processar – `in_stream.close()`;
- Escrever ficheiro: semelhante mas com `ofstream`
- Ler de ficheiros: dependente do conteúdo, usar:
 - o `while(in_stream >> var)` ou
 - o `while(!in_stream.eof()) {`
`getline(in_stream, string);`
`vetor.push_back(string)`
`}`
- Em funções: streams devem ser passadas por referencia
- Util: `in_stream.is_open()`

FORMATTING

- `#include <iomanip>`
- Funciona para qualquer stream
- Ex.: `cout << fixed;` (ponto decimal)
- `Cout << setprecision(value)` (precisão decimal)
- `Cout << setw(value) << qqcena;` qqcena dispõe de value espaços na stream

STRINGSTREAMS

- `#include <sstream>`
- `Istringstream` para ler de stream para variáveis, `Ostringstream` para o inverso
- `istringstream instr(input);` inicializa instr com input e é possível fazer `instr >> var1 >> var2 >> var3`
- `Ostringstream ostr;` e `ostr << var1 << " " << var2 << "," << var3;` para criar string, `string output = ostr.str();`

CLASSES

- usar qualificativo `const` se método não deve alterar parâmetros `private`
- criar construtor sem e com parâmetros se necessário
- para fazer definição dos métodos `public` usar `ClassName::function()`
- `this->` quando parâmetros do construtor e `private` tem o mesmo nome
- parâmetro `private` com `static`: apenas uma cópia para todos os objetos do tipo
- deve ser definido fora da definição da classe como TIPO `ClassName::variável= valor(opcional);`
- `STATIC` apenas aparece na definição da classe
- métodos que alterem essa variável devem ter a key `static`
- método estático pode ser sempre chamado como `ClassName::método()`
- Destrutor é `~nomeDaClasse` e é usado para eliminar variáveis dinâmicas

TEMPLATES

- Quando uma função pode servir para trabalhar com vários tipos de variáveis, chama-se uma template. Basta colocar antes da definição `template <typename T>` ou se usar mais que um tipo, `template <typename T1, typename T2>`
- No corpo da função quando se referir ao tipo dos parâmetros, usar sempre `T`

- Também é possível usar template classes, é só colocar `template<typename T>` antes da definição da classe e antes dos métodos

```
//-----
template <class T> // OR template <typename T>
void swapValues(T &x, T &y)
{
    T temp = x;
    x = y;
    y = temp;
}
```

ITERATORS

- Parecidos com apontadores, são uteis para aceder aos elementos de containers da STL. Ex de declaração: `vector<int>::iterator p`; `p` é um iterador para um vetor de inteiros.
- `Const_iterator` se apenas queremos ler do container. `Reverse_iterator` se queremos ler a partir do final do container (atenção, nem todos os containers suportam)
- Para aceder a elemento do container, usar `(*p)`
- Exemplo de ciclo:
 - o `vector<int>::iterator p;`
 - o `for (p = v1.begin(); p != v1.end(); p++)`
 - o `cout << *p << endl;`
- `v1.begin()` - > aponta para inicio do container, `v1.end()` aponta para posição à frente do fim do container
- `bidirectional iterator` -> possível processar container na direção normal ou inversa
- `random access` -> igual ao anterior mas tb permite aceder a um elemento qualquer

Predefined typedefs for iterator types	Direction of ++	Capability
<code>iterator</code>	forward	read/write
<code>const_iterator</code>	forward	read
<code>reverse_iterator</code>	backward	read/write
<code>const_reverse_iterator</code>	backward	read

CONTAINERS

- `Set`, `multiset`(permite duplicados): fácil de procurar elementos, guardam elementos do mesmo tipo (ex: moedas numa carteira) e ficam ordenados automaticamente por ordem crescente
- Declarar: `#include <set> -- set<T> nome;`
- Útil: `nome.insert(elem); nome.erase(elem); nome.clear();`

- Map, multimap(permite mesma chave várias vezes): associa elementos a uma chave, que são ordenados por ordem crescente pela key. Elementos de um map são pair (#include <utility>)
- Para aceder a elementos do pair p: p.first, p.second
- Declarar: #include map – map<T1,T2> nome2;
- Aceder a elementos do map: nome2[chave] retorna o seu par
- Útil: nome2.insert(make_pair(var1,var2))

```
int main()
{
    map<int,int> m;
    map<int,int>::const_iterator mi;

    pair<int,int> p;

    m[20]=10;
    m[5]=500;

    cout << "MAP\n";
    int n=0;
    for (mi=m.begin(); mi!=m.end(); mi++)
    {
        n++;
        p = *mi; // each element of a "map" is a "pair"
        cout << n << " - " << p.first << ", " << p.second << endl;
    }

    //NOTE the order by which elements were presented

    return 0;
}
```

Sequence containers (first class)

vector	random access
deque	random access
list	bidirectional

Associative containers (first class)

set	bidirectional
multiset	bidirectional
map	bidirectional
multimap	bidirectional

ALGORITHMS

- #include <algorithm>
- Ordenar: sort(container.begin(),container.end())
- Apagar: erase e remove:
numbers.erase(remove(numbers.begin(),numbers.end(),0),numbers.end()); para apagar 0s em numbers.
- Find: find(Iter first, Iter last, const T &value); retorna um iterador que aponta para o primeiro elemento encontrado ou se não encontrar, retorna last.
- Search, binary_search

FRIEND FUNCTIONS

- Algumas funções são úteis para as classes mas não podem ser métodos (ex: operador ==)
- Atributo friend possibilita o acesso aos parâmetros private da classe
- Na definição da classe, fora da parte public, colocar atributo friend e definição da função
- A definição da função é normal, sem uso de ::, e ao ser chamada não se usa o operador '.'.

OPERATOR OVERLOADING

- Ao definir classes pode-se tornar útil fazer overload aos operadores(++ , +=, == ,etc).
- Normalmente: operadores ++, += podem ser métodos. Operadores de comparação podem ser friends, ou independentes, e os operadores de entrada e saída devem ser independentes, retornando uma stream para permitir encadeamento.
- Exemplos de definições:
- Classe& operator++(); (referencia para permitir encadeamento – Classe++++)
- Classe operator+(const Classe & left, const Classe & right);
- bool operator==(const Classe & left, const Classe & right);
- ostream& operator<<(ostream& out, const Classe & value);
- Noutras situações, para permitir encadeamento de operações no objeto, os métodos devem retornar uma referência ao próprio objeto. (return *this)

```
bool operator==(const Purse & p1, const Purse & p2)
{
    if (p1.tellAmount() != p2.tellAmount())
        return false;
    vector<float> c1 = p1.tellCoins(), c2 = p2.tellCoins();
    sort(c1.begin(), c1.end());
    sort(c2.begin(), c2.end());
    if (c1 != c2) // *** alternativa: ciclo para comparar os elementos um a um (ver abaixo)
        return false;
    return true;
}
```


É preciso fazer o "overload" do operador <<. Uma vez que não estão definidos métodos getXXX() na classe Time, é necessário declarar este operador como friend da classe:

```
class Time {
    friend ostream & operator<<(ostream &out, const Time &t)
    public:
        ...
};

... e definir a respetiva função:

ostream & operator<<(ostream &out, const Time &t) {
    out << t.h << ":" << t.m << ":" << t.s;
    return out;
}
```

THIS POINTER

- Aponta para o objeto. Exemplo de uso: (*this) ou this-> para aceder ao objeto

INHERITANCE

- Por vezes podem ser definidas classes que estão relacionadas. Por exemplo: Student e Professor são ambos FeupPerson, logo devem partilhar características. Student e Professor são classes derivadas de FeupPerson, e ambas tem as suas características próprias (cadeiras dadas ou frequentadas, etc);
- Na classe principal, usar protected em vez de private para que classes derivadas possam aceder a esses parâmetros
- Na classe derivada, definir: class Derived : public Base ...
- No construtor da classe derivada, que deve receber parâmetros da classe base, usar: Derived::Derived(parâmetrosTodos) : Base(parametrosBase) e no código apenas inicializa os parâmetros da própria classe.
- Pode-se redefinir métodos da classe base na classe derivada.
- É possível usar o assignment para atribuir a uma classe base uma classe derivada, porém ocorre perda de informação - slicing problem (ex FeupPerson p1; Student p2; p1=p2;)
- O contrário é ilegal.

EXCEPTION HANDLING

- Quando testamos a ocorrência de um erro, podemos simplesmente parar o programa (exit(1);), imprimir uma mensagem de erro, usar assert(), ou usar o método try-catch.
- Numa função, se sabemos que pode ocorrer algum erro (mau input, por exemplo), pode-se fazer: if(var<valor) throw logic_error(string)
- Na função em que é chamada, fazer try{chamar função} catch(logic_error& e){ cout << "erro " << e.what(); }
- Este método é útil para usar nos construtores e indicar a ocorrência de erros.

```
double futureValue(double initialAmount, double tax, int numYears)
{
    if (initialAmount < 0 || tax < 0 || numYears < 0)
    {
        //logic_error description("illegal futureValue parameter");
        //throw description;
        throw logic_error("illegal futureValue parameter");
    }
    return initialAmount * pow(1 + tax / 100, numYears);
}

int main()
{
    double value, amount, tax;
    int years;
    cout << "amount ? "; cin >> amount;
    cout << "tax ? "; cin >> tax;
    cout << "numYears ? "; cin >> years;
    try
    {
        value = futureValue(amount, tax, years);
        cout << "future value = " << value << endl;
    }
    catch (logic_error& e)
    {
        cerr << "Processing error: " << e.what() << "\n";
    }
}
```