PROGRAMAÇÃO MIEIC-FEUP – NOTAS PARA EXAME

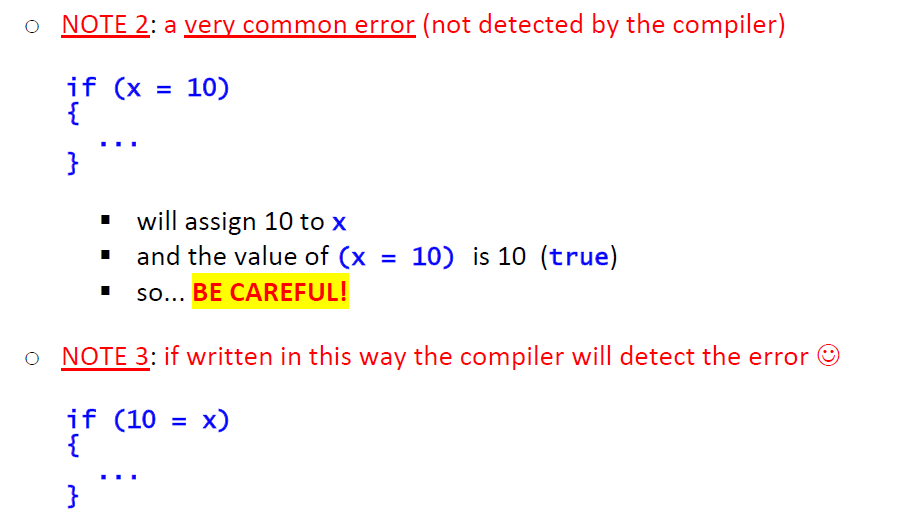
IMPORTANTE: para usar estes apontamentos no exame convém ter um conhecimento prévio da matéria, senão podem não perceber alguns destes pontos.

**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)



**S** **WITCH 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:
  + cin.eof(); //para terminar ciclo de input com CTRL-Z
  + cin.fail(); //verifica estado do buffer de entrada
  + cin.ignore(1000,’/n’); // limpa o buffer de entrada
  + cin.clear(); //permite continuar a fazer input em caso de erro

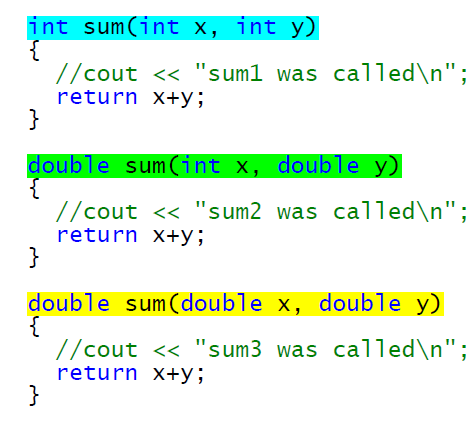
**CALL-BY-VALUE AND CALL-BY-REFERENCE**

* 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

**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()+índex)
* Útil:
  + Vec.push\_back(elem); //coloca elem no final do vetor
  + Vec.pop\_back(); //remove o ultimo elemento do vetor
  + Vec.clear(); // apaga todos os elementos do vetor
  + Vec.size(); // retorna tamanho do vetor, útil em ciclos for
  + 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:
  + Strcmp(cstr1,cstr2) //compara as strings e devolve um booleano
  + strcat(cstr1,cstr2) //concatena as strings
  + 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 <cstdlib>, malloc(num bytes), free()
* malloc aponta para primeiro byte alocado, se não houver memoria suficiente retorna um NULL pointer
* free(void \*p) retorna a memoria alocada anteriormente para o sistema, p é um apontador para essa memoria

**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:
  + while(in\_stream>>var) ou
  + 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

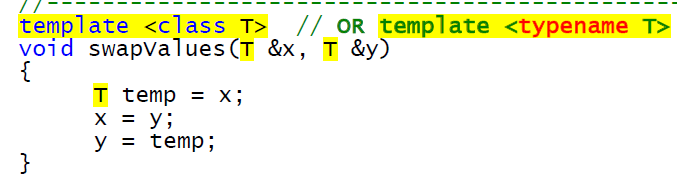
**STRINGSTREAMS**

* #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 outstr; e outstr << var1 << " " << var2 << "," << var3; para criar string, string output = outstr.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

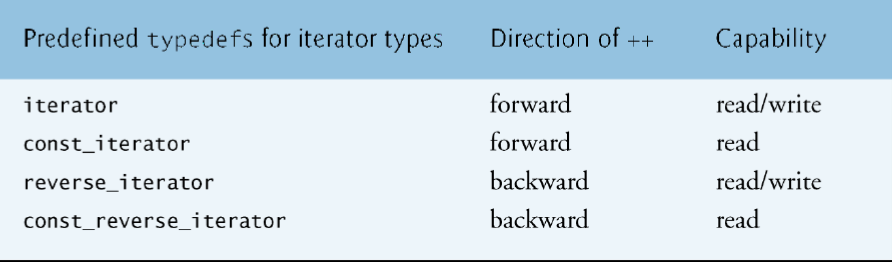
**ITERATORS**

* 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:
  + vector<int>::iterator p;

for (p = v1.begin(); p != v1.end(); p++)

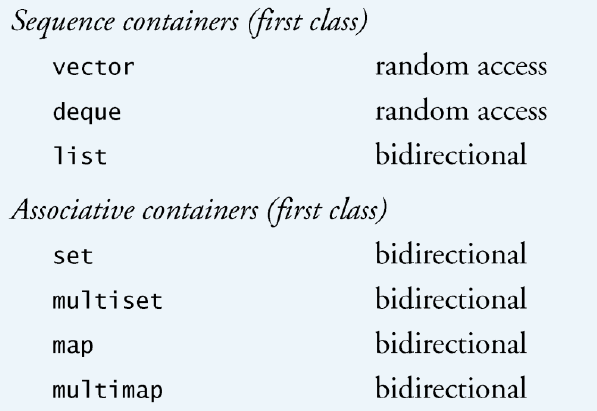
cout << \*p << endl;

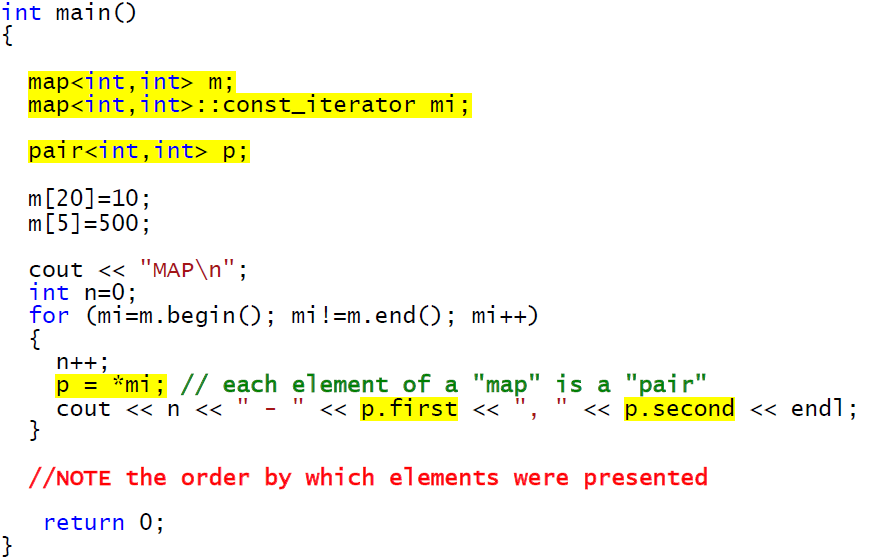
* v1.begin() - > aponta para inicio do container, v1.end() aponta para posição à frente do fim do container
* bidirectional iterator -> possível processar container na direção normal ou inversa
* random access -> igual ao anterior mas tb permite aceder a um elemento qualquer



**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))





**ALGORITHMS**

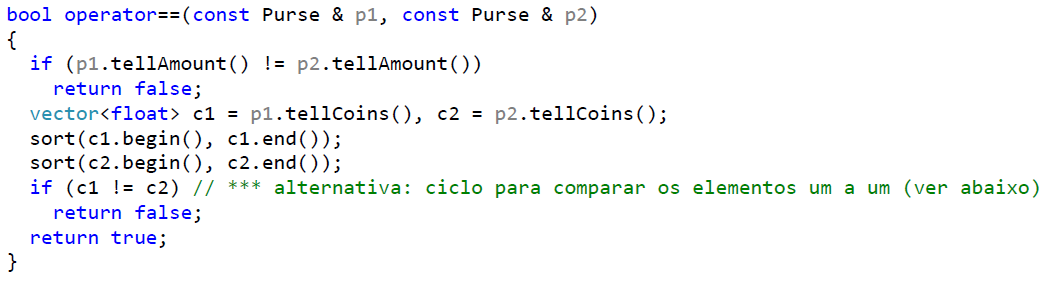
* #include <algorithm>
* Ordenar: sort(container.begin(),container.end())
* Apagar: erase e remove: numbers.erase(remove(numbers.begin(),numbers.end(),0),numbers.end()); para apagar 0s em numbers.
* Find: find(Iter first, Iter last, const T &value); retorna um iterador que aponta para o primeiro elemento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

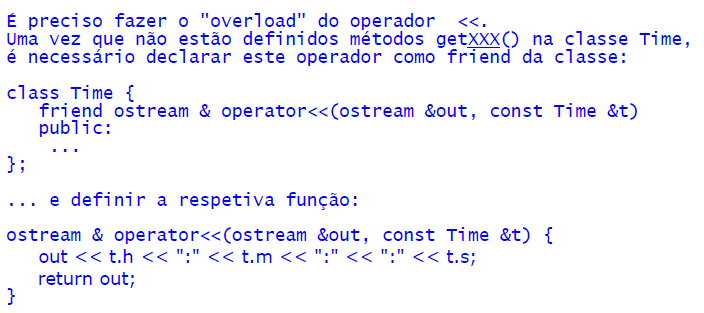
**FRIEND FUNCTIONS**

* Algumas funções são uteis 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)





**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-throw-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.

