Física Computacional 3
mem. estática e dinâmica

1. Revisão de arrays
2. Revisão de ponteiros e ponteiros para ponteiros
3. Ponteiros para funções
4. Préprocessador: headers
5. Mais à frente: estruturas e classes
6. Mais à frente: Standard Template Library e vectors

fc.trabalhosalunos@gmail.com

Revisão de vectores

- Variáveis estáticas:
  - As variáveis estáticas são dimensionadas dentro do próprio código.
    - A **vantagem** que têm é a sintaxe que é muito simples, permitindo criar arrays (no sentido matemático podem ser vetores linha ou coluna, ou matrizes com linhas e colunas, ou tensores, por exemplo;
    - **int** a[1000]; // vetor num espaço a 1000 dimensões
    - **float** mat[3][3]; // caso se trate de um array de tipo matriz 3x3
    - **double** r[100][20][30]; // array tensorial
  - No entanto possuem duas desvantagens,
    - possuem a **desvantagem** de ocuparem memória **fixa** que não poderá ser reaproveitada. Os vetores apenas se podem apagar quando o bloco ou função onde foram definidos é terminado. Isto é feito de forma automática pelo compilador.
    - Outra **desvantagem** é termos de definir exatamente o número de elementos no código, ou seja o compilador necessita de conhecer o número de elementos, estes não podem apenas à posteriori ser definidos pelo utilizador do ficheiro executável. Por exemplo,
    - **const unsigned short** ni=3;
    - **const unsigned short** nj=3;
    - **float** m[ni][nj]; // deve ser precedido da determinação de ni e nj

- Existem vários tipos de inicialização, que podem ser realizados de forma completa numa única instrução, ou de forma incompleta,
  - **int** a[3] = {3,10,2}; //forma completai
  - **int** a[] = {3,10,2}; //forma incompleta, são assumidos 3 elementos, a[3].
- A partir do momento que o array está inicializado, os valores podem ser acedidos de várias formas, incluindo uma sintaxe parecida com a da inicialização,
  - usando a notação [i]:
  - std:: cout<< a[2]; // dá o valor do elemento 2 de a, ou seja retorna 2
  - A única forma de imprimirmos o array completo é com um loop,
  - **for**(**int** i=0; i <ni ; ++i) std::cout<<i<<" "<<a[i]<<"\n";
- Podemos alterar os valores do array usando a mesma sintaxe,
  - a[2]=7;// muda o valor do 3º elemento de 2 para 7
- Recomendamos o estudo do código exemplo diponibilizado na página da cadeira, **matrixwitharrays.cpp**

Revisão de vectores

Revisão de ponteiros

- Um ponteiro dá-nos o endereço em memória de uma variável

  int a = 4;

  int *p = &a; // equivale a escrevermos 2 comandos **int** *p; p=&a;

  O operador & (inverso de *) dá-nos o endereço de uma variável

- O valor apontado por um ponteiro pode ser acedido através do operador unário *, conhecido pelo operador de “dereferencing”

  int b = *p;

  *p = 5; //escreve , como * é o inverso de &, ficamos com a=5;

- Ponteiro nulo : 0, NULL ou (em C++11) **nullptr**
  - Não podemos desreferenciar ponteiros nulos:

    int *p = **nullptr**; // podemos fazer mas não aponta para nada

    int c = *p; //programa pode crashar
  - Tem o valor booleano **false**

Referências (notação ~ infeliz)

- Em C, usa-se ponteiros para passar variáveis por referência

```
void setZeroponteiro(int * p) { *p = 0; }
```

- Em C++ é possível utilizar referências (não confundir com o inverso de *)

```
void setZeroRef(int & a) { a = 0; } // aqui é usada no argumento
```

- Podemos também definir referências dentro de funções

```
int main() {
     int a = 5;
     setZeroRef(a);  //quando se aplica a função não se usa a sintaxe &a
     std::cout << a << std::endl; // imprime 0
     int &b = a; //& significa que b fica para sempre associado a a
    b = 7;
    std:: cout << a << std::endl; // imprime 7, b é o alias (avatar) do a
    // nota por defeito o main retorna 0
}
```

Memória dinâmica

- Podemos usar a memória de forma dinâmica. O que tem duas **vantagens** face a memória estática:
  - O compilador não precisa de saber a quantidade de memória a usar
  - Podemos apagar as variáveis quando já não são necessárias.

- No entanto, tem também algumas desvantagens, como veremos

- Em C a alocação dinâmica de memória é feita através da função malloc
  - **int** *p = (**int***) malloc(10 * **sizeof**(**int**));

    A memória deverá ser libertada usando a função free
  - free(p);

Memória dinâmica

- Em C++ a alocação de memória é feita utilizando o operador **new**
  - **int** *p = **new** int;// usa-se com *p=5 ou *p=a;
  - A memória é libertada através do operador **delete**
  - **delete** p;

- Para alocar arrays de objetos utiliza-se um ponteiro, mas com a sintaxe
  - **int** ni; // que pode ser inicializado na execução
  - **int** *row = **new** int[ni]; //aloca array de ni inteiros
  - Para libertar a memória:
  - **delete**[] row;
  -

- Nota: Os operadores **new** e **new**[] contrariamente ao malloc, não só alocam memória como também chamam os construtores do objetos alocados (a rever nas classes).

- O **delete** e o **delete**[] chamam os destrutores correspodentes.

- Para trabalharmos com matrizes, utilizamos um ponteiro para ponteiro, ou seja ponteiros para vectores:
  - **int** ni , nj ;
  - **float** ** mat= **new float***[ni];
  - **for**(int i = 0; i < ni; ++i) mat[i] = **new float**[nj];

  - Podemos anteriormente atribuir as dimensões da matriz apenas durante a execução,
    - std::cin>> ni >> nj;

  - A sintaxe para a leitura e escrita é idêntica à dos arrays

    for(**int** i=0; i <ni; ++i)

    for(**int** j=0; j <nj; ++j) std::cin>>mat[i][j];
  - Recomendamos o estudo do código exemplo disponibilizado na página da cadeira, **matrixwithpointers.cpp**

Memória dinâmica

Memória dinâmica

- Não esquecer:
  - Memória alocada com *malloc* tem de ser libertada com *free*
  - Memória alocada com **new** tem de ser libertada com **delete**
  - Memória alocada com **new**[] tem de ser libertada com **delete**[]
  - Libertar a memória quando esta deixar de ser necessária
  - Nunca libertar memória que não foi alocada dinamicamente:

```
double vec[4];
...
delete[] vec; //ERRO CRASSO!!!
```

- Não retornar ponteiros para variáveis locais

Ponteiros para funções

- Os ponteiros também podem apontar para funções com a sintaxe,
  - **float** (*f) (**int** x, **int** y) //ponteiro *f* para uma função que retorna um **float**, recebendo dois **int**
  - // o * é para termos um ponteiro, ou seja uma variável-função dinâmica
  - // ou seja desconhecida durante a compilação
  - // os () são para dizer que temos um ponteiro para uma função
  -
  - Notamos que sem () teríamos uma função que retornaria um ponteiro para float, assim a sintaxe
  - **float** *f(**int** x, **int** y)
  - estaria errada.
  -
  - O ponteiro f para função é utilizado normalmente, seja no main ou noutra função, sem nenhuma sintaxe específica, como seria usada uma função normal.

Ponteiros para funções

- Os ponteiros também podem apontar para funções com a sintaxe,
  - **float** (*f) (**int** x, **int** y) //ponteiro *f* para uma função que retorna um **float**, recebendo dois **int**
  - // o * é para termos um ponteiro, ou seja uma variável-função dinâmica
  - // ou seja desconhecida durante a compilação
  - // os () são para dizer que temos um ponteiro para uma função
  -
  - Notamos que sem () teríamos uma função que retornaria um ponteiro para float, assim a sintaxe
  - **float** *f(**int** x, **int** y)
  - estaria errada.
  -
  - O ponteiro f para função é utilizado normalmente, seja no main ou noutra função, sem nenhuma sintaxe específica, como seria usada uma função normal.

Ponteiros para funções

- Exemplo:

```
#include <iostream>
void printFunc( float (*f)(float), float x ) {
    std::cout << f(x) << std::endl;
}
float id(float x) { return x; }
float quad(float x) { return x*x; }
int main() {
    float (*f)(float) = &id;
    printFunc(f, 4); //imprime 4
    printFunc(&quad, 4); //imprime 16
}
```

- Recomendamos o estudo do código exemplo diponibilizado na página da cadeira, **functionoffunctionwithpointers.zip**
- É importante ainda aprofundar estes conceitos, por exemplo no tutorial de c++ disponível na internet,
  - http://www.cplusplus.com/doc/tutorial/

Pré-processador: biblioteca, header

- Para trabalharmos com ponteiros, ponteiros para funções, e com funcções úteis para o trabalho 3 como o gerador de números aleatórios rand, é necessário no pré-processador do código main incluir a biblioteca
  - #include <cstdlib>
- E ainda, caso os códigos respeitantes a funções estejam em ficheiros .cpp separados do ficheiro .cpp onde se inclui a função main, é conveniente criar un ficheiro header para o pré-processador, por exemplo
  - #include "myheader.h"
- que contenha os protótipos, // myheader.h, header for a main using functions: integral, function
  - **float** integral(**float** (* **function**)(float), **float** xmin, **float** xmax, **int** npoints);
  - **float** function(**float** x);
- e inclui-lo nos ficheiros a linkar, o que e automatico em Code::Blocks. mas com o comando de linha em linux ou MacOS g++, deve ser escrito por extenso,
  - g++ -o mycode.exe mycode.cpp integral.cpp fucntion.cpp myheader.h

Standard Template Library

- Para se evitar os eventuais problemas decorrentes da utilização da alocação dinâmica de memória em C++ pode-se usar os contentores da STL – *standard template library*
- Entre estes contam-se:
  - std::vector : Semelhante aos vetores mas mais seguro de usar
  - std::string : O mesmo para strings
  - std::list : *double linked list*
- Estes contentores são classes genéricas – falaremos disto mais tarde
- A STL fornece também algoritmos e iteradores para lidar com estes contentores

std::vector

- Para usar a classe std::vector: #include <vector>
- Usando ponteiros diretamente, teríamos:
  int* a = new int[10];
- Com o std::vector, fazemos:
  std::vector<int> a(10);
- Podemos aceder normalmente aos elementos de a:
  b = a[10]
  a[10] = b;
- Dimensão do vetor dada por size() :
  size_t dim = a.size();
- O vetor pode ser redimensionado dinamicamente:
  a.resize(20); //aumenta a dimensão de a para 20

std::vector e std::string

- Adicionar elementos usando push_back();
  std::vector<int> a; //ficamos com um vetor de dimensão 0
  a.push_back(4); //a fica com um elemento – 4

- Não é necessário desalocar explicitamente a memória – quando o scope de a acaba, o construtor é chamado e a memória é libertada

- Para strings, usamos o std::string
  std::string a = “abc”; //cria string com valor “abc”
  std::string b = “abc”;

- Operador == usado para comparar o conteúdo de duas strings
  std::cout << (a == b) << std::endl; //o resultado é verdadeiro

- Operador + usado para concatenar duas strings
  std::string c = a + b; //c fica com o valor “abcabc”