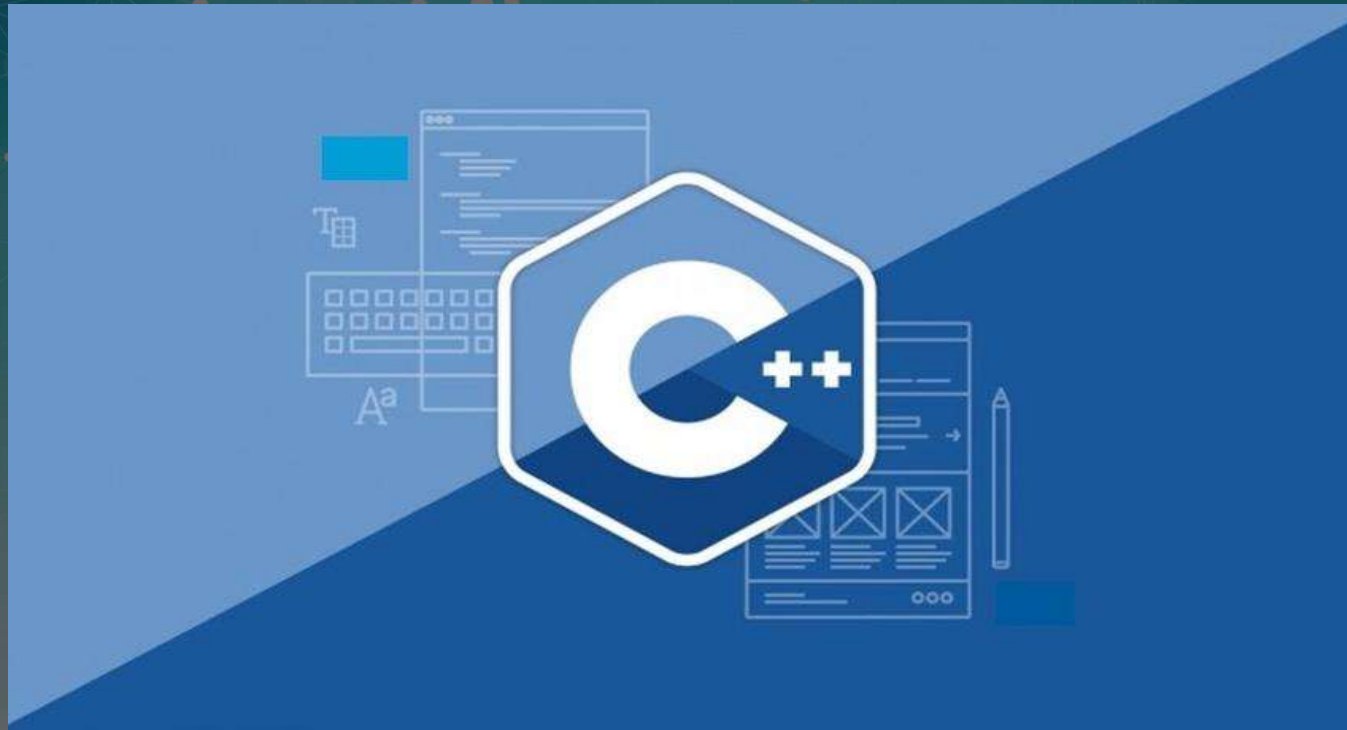


PROGRAMAÃO ORIENTADA A OBJECTOS

2ºAno – 1ºSemestre



FICHA 1

[Input-Output simples / Strings

Referências / Overloading / Parâmetros com valor por omissão]

Exercícios para [26.09.2022]: 0,1,2,3,4

FICHA 1

[Input-**Output** simples]

// Programa de impressão de texto.

#include <iostream> // permite que o programa gere saída de dados na tela

using namespace std;

// a função main inicia a execução do programa

int main()

{

cout << "Bem vindo ao C++!" << endl; // exibe a mensagem

return 0; // indica que o programa terminou com sucesso

} // fim da função main

iostream é a biblioteca com os fluxos de bytes padrão (entrada/saída)

Não tem **.h**;
Equivalente à **stdio.h**.

using namespace std;

Um **namespace** é um grupo de nomes de entidades, como classes, objetos e funções;

Neste caso, **std** é o agrupamento das entidades da biblioteca padrão C++;

Para usar os identificadores da **iostream**, precisamos indicar qual é o **namespace**;

endl

- end line
- Manipulador de fluxo
- Quebra a linha e limpa o buffer
- Nada fica acumulado no buffer

FICHA 1

[Input-**Output** simples]

A entrada e saída de dados é realizada basicamente em 3 padrões:

Entrada Padrão (cin)

Normalmente o teclado, mas pode ser redirecionado.

Saída Padrão (cout)

Normalmente o monitor, mas pode ser redirecionado.

Fluxo Padrão de Erros (cerr)

Utilizado para mostrar mensagens de erro padronizadas;

Normalmente associado ao monitor.

Todos são objetos incluídos no **namespace std**.

```
cout << "Welcome to C++! \n";
```

É uma instrução de saída

<< é o operador de inserção de fluxo

O valor à direita do operador é inserido no fluxo de saída.

```
cin >> number1;
```

É uma instrução de entrada

O objeto cin aguarda um fluxo de entrada

>> é o operador de extração de fluxo

O valor na entrada padrão é inserido na variável à direita do operador

Não utiliza formatadores.

Para no **whitespace**

```
cout << "Sum is " << sum << endl;
```

FICHA 1

[Input-**Output**: Validação da entrada]

cin.fail()

a função retorna TRUE quando o erro ocorre. No caso ao lado, deve ser um inteiro. Se o cin falhar o buffer de entrada é mantido no estado de erro.

cin.clear()

usado para limpar o buffer do estado de erro. Assegura que não se entre num loop infinito de mensagens de erro.

cin.ignore()

esta função permite ignorar o resto da linha após o primeiro erro que ocorreu.

cin.eof()

verifica o fim do ficheiro. Retorna 1, se o programa retornar 1.

```
#include<iostream>

using namespace std;

int main()
{
    int a;

    cout << "Digite um numero inteiro\n";
    cin >> a;
    while (1)
    {
        if (cin.fail())
        {
            cin.clear();
            cout << "Digitou um número errado :) " << endl;
            cin >> a;
        }
        if (!cin.fail())
            break;
    }

    cout << "O numero é : " << a << endl;
    return 0;
}
```

FICHA 1

[Argumentos de Funções por Omissão]

No C++ é possível associar valores aos parâmetros. Valores esses usados sempre que as funções são invocadas. Por exemplo, o seguinte protótipo da função sub

```
void sub(int i = 2, float = 2.3);
```

permite as seguintes invocações:

```
sub();    // utilizados os valores 2 e 2.3  
sub(5);   // utilizados os valores 5 e 2.3  
sub(5,1.1); // utilizados os valores 5 e 1.1
```

FICHA 1

[Strings e ...]

- I/O streams

Keyboard (cin) and monitor (cout)

- File streams – Contents of file are the stream of data

#include <fstream> and #include <iostream>

ifstream and ofstream objects

- Stringstreams – Contents of a string are the stream of data

#include <sstream> and #include <iostream>

stringstream objects

C++ String **getline()**

A biblioteca **string** (#include <string>) define uma função que permite a leitura completa de uma linha de texto

Exemplo ...

FICHA 1

[Strings e ...]

STRINGSTREAMS

```
#include<sstream>
using namespace std;
int main()
{
    stringstream ss;
    int num = 12345;
    ss << num;
    string strNum;
    ss >> strNum;
    return 0;
}
```

```
#include <sstream>
using namespace std;
int main()
{
    stringstream ss;
    ss << "2.0 35 a";
    double x, int y; char z;
    ss >> x >> y >> z;
    return 0;
}
```

parse (split) uma string em vários valores em variáveis separadas

Classes

Conceito

Membros Privados e Públicos

Construtor e Destrutor

Keyword `this`, `const`

Setters e Getters

Matéria aplicada (iniciada) na **Ficha de Exercício 2**

Classes (Conceito)

- Tipo de dados definido pelo utilizador (programador)
- Usadas para definir um conceito do mundo real, por exemplo, a representação de automóvel com todos os seus atributos (cor, cilindrada, ...)

Nota: A estrutura (struct) tende para ser “public” (membros), e a classe (class) tende para ser “private” (membros).

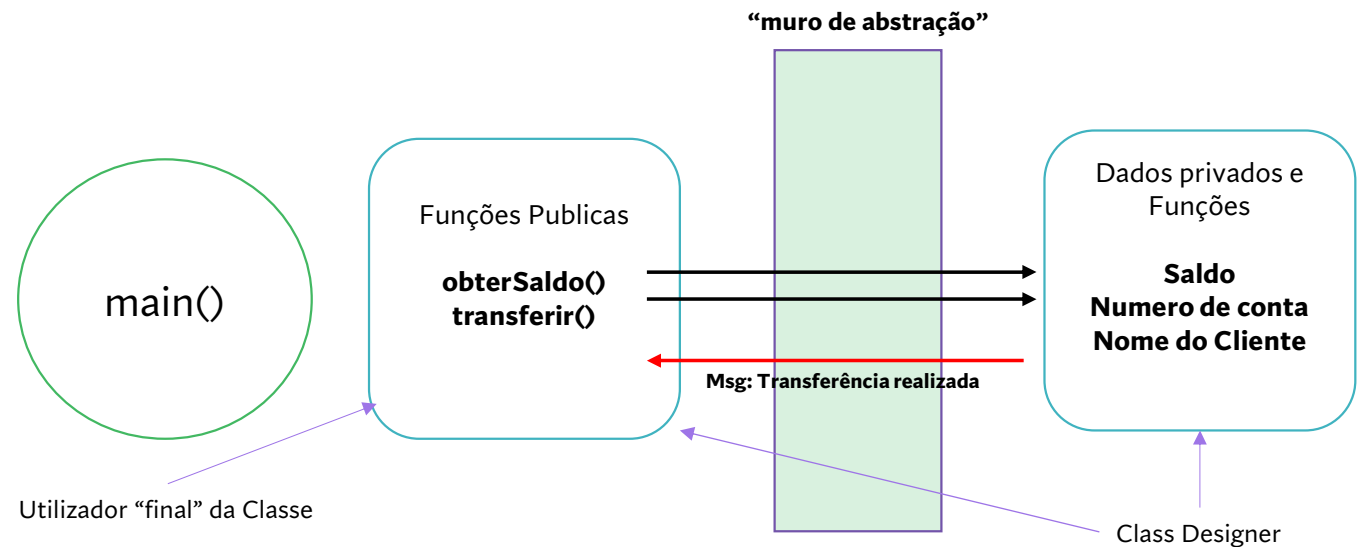
O resto é o mesmo 😊

```
11
12 class NomeClasse{
13     private:
14         {membros privados}
15         (...)
16     public:
17         {membros publicos - no contexto das classes -> métodos}
18         (...)
19 }
20
21
22
```

Classes (Encapsulamento)

Encapsulamento:

permite desenhar uma classe usando um “**muro de abstração**”. Ou seja, o utilizador final da classe não tem acesso aos dados, mas apenas quem desenvolve a classe.



Classes (o que fazer para desenhar/utilizar uma classe ?)

(iremos usar o exemplo do **automóvel**, como **conceito do mundo real** a representar)

- Entender o conceito que pretendemos representar.
atributos: cor, cilindrada, marca, modelo, combustível
funcionalidades: acelerar, travar, estacionar (...)
- Definir a classe
membros privados: usar os atributos que queremos abstrair do utilizador final da classe, por exemplo: cor, marcar ...
membros públicos: construtor, destrutor, setters e getters, outras funcionalidades a serem desenhadas (travar, acelerar) ...
- Usar a classe na função **main**
definir um objeto, e usar os métodos.

Classes (exemplo: automóvel)

(iremos usar o exemplo do **automóvel**, como **conceito do mundo real** a representar)



```
class automovel {  
    int cilindrada;  
    string cor;  
    int id; // "chave primária"  
public:  
    (...)
```

Classes (exemplo: automóvel)

// Construtor por omissão

Desenho da Classe

```
automovel() {  
  
    this->cilindrada = 1000;  
    this->cor = "verde";  
    id = ++contador;  
    cout << "Construtor por omissao ..." << "Automovel " << id << endl;  
  
}
```

Main() → Objeto

```
int main() {  
    automóvel a; // construtor por omissão  
}
```

Constructor (construtor):

- O construtor é invocado automaticamente sempre que é criado um objeto da classe
- membro-função com o mesmo nome da classe
- construtores não podem especificar tipos ou valores de retorno
- podem-se definir construtores com argumentos para receber valores a usar na inicialização
- deve ser declarado como **público**
- permite o overloading

Nota:

- keyword **this** refere-se ao objeto corrente.
- Usa → e não . (é um ponteiro).
- ponteiro "hidden" que é passado a todos os membros não estáticos

Classes (exemplo: automóvel)

construtor com parâmetros

Desenho da Classe

```
automovel(int cilind, string cor) {  
    this->cilindrada = cilind;  
    this->cor = cor;  
    id = ++contador;  
    cout << "Construtor com parametros ..." << "Automovel " << id << endl;  
}
```

Main() → Objeto

```
int main() {  
    automovel b(1200, "Amarelo");  
  
    return 0;  
}
```

Classes (exemplo: automóvel)

construtor por cópia

Objeto que queremos copiar

Desenho da Classe

```
automovel(const automovel& obj)
{
    cilindrada = obj.cilindrada;
    cor = obj.cor;
    id = ++contador;
    cout << "Construtor por copia\n";
}
```

Main() → Objeto

```
int main() {
    automovel b(1200, "Amarelo");

    return 0;
}
```

prefixo **const**:

- Especifica que a variável tem um valor constante, Diz ao compilador para prevenir a sua modificação;
- “ativa” o “read-only”

Membro-dado constante: `const automovel& obj`

declarado com prefixo const

especifica que não pode ser modificado (tem de ser inicializado)

Membro-função constante: `string getAsString() const { }`

declarado com sufixo const (a seguir ao fecho de parêntesis)

especifica que a função não modifica o objecto a que se refere a chamada

Classes (exemplo: automóvel)

Desenho da Classe

```
~automovel() { // destrutor
    // não tem argumentos, e permite fazer a "limpeza"
    contador--;
    cout << "A destruir automovel " << id << endl;
}
```

Main() → Objeto

```
int main() {

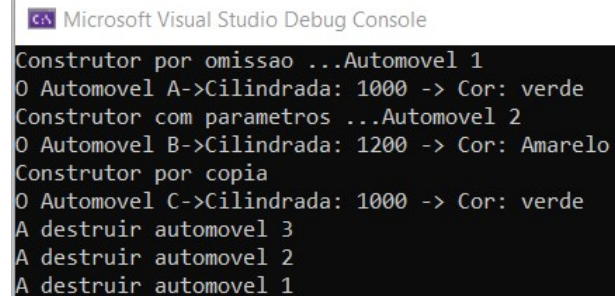
    automovel a; // construtor por omissao
    cout << "0 Automovel A->" << a.getAsString();

    automovel b(1200, "Amarelo");
    cout << "0 Automovel B->" << b.getAsString();

    automovel c(a);
    cout << "0 Automovel C->" << c.getAsString();

}
```

Output



```
Microsoft Visual Studio Debug Console
Construtor por omissao ...Automovel 1
0 Automovel A->Cilindrada: 1000 -> Cor: verde
Construtor com parametros ...Automovel 2
0 Automovel B->Cilindrada: 1200 -> Cor: Amarelo
Construtor por copia
0 Automovel C->Cilindrada: 1000 -> Cor: verde
A destruir automovel 3
A destruir automovel 2
A destruir automovel 1
```

Destructor (Destrutor):

- função que é chamada sempre que o âmbito de duração do objeto de uma classe - faz a "limpeza"
- Não podem ser chamados explicitamente pelo programador.

Classes (exemplo: automóvel)

getAsString()

Desenho da Classe

```
string getAsString() const {  
    ostringstream buffer0;  
    buffer0 << "Cilindrada: " << cilindrada << " -> " << "Cor: " << cor << "\n";  
    return buffer0.str();  
}
```

permita obter a representação textual do conteúdo da estrutura. Não se pretende necessariamente imprimir essa informação no ecrã e evite fazê-lo diretamente em código da função (“obter” ≠ “imprimir”).

Main() → Objeto

```
int main() {  
    automovel a; // construtor por omissao  
    cout << "O Automovel A->" << a.getAsString()  
  
    return 0;  
}
```

Classes (exemplo: automóvel)

getCor()

Desenho da Classe

```
string getCor() const {  
    return cor;  
}
```

setCor()

```
void setCor(string nCor) {  
    cor = ncor;  
}
```

Main() → Objeto

```
int main() {  
  
    automovel a; // construtor por omissao  
    cout << "A cor é >" << a.getCor() ;  
    a.getColor("Azul");  
  
    return 0;  
}
```

No sentido de manter o **encapsulamento**, e aceder a dados-membro privados, definimos os métodos públicos “get e “set”.

Ficheiros .h e .cpp

Arquivo .h

- Neste arquivo iremos colocar a classe, com os dados-membro e as funções-membro (mas apenas os **headers**) referentes à classe que estamos a desenhar.

Arquivo .cpp

- Neste arquivo iremos colocar as implementações referentes à classe que definimos no .h
- Normalmente, cria-se um arquivo por classe.
- No cabeçalho do arquivo colocamos a referência ao arquivo .h :

#include "arquivo.h"

main()

#include "arquivo.h"

- **Normalmente**, cria-se um arquivo por classe.

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> primeiro; // vetor vazio de inteiros
    vector<int> segundo(4, 100); // 4 inteiros com o valor 100
    vector<int> terceiro(segundo.begin(), segundo.end()); // usando a interação do segundo
    vector<int> quarto(terceiro); // copia do terceiro

    // construcao usando um array:
    int myints[] = { 16,2,77,29 };
    vector<int> quinto(myints, myints + sizeof(myints) / sizeof(int));

    // mostra o segundo
    cout << "O conteudo do segundo:";
    for (vector<int>::iterator it = segundo.begin(); it != segundo.end(); ++it)
        cout << ' ' << *it;
    cout << '\n';

    // mostra o terceiro

    // mostra o quarto

    // mostra o quinto
    cout << "O conteudo do quinto:";
    for (vector<int>::iterator it = quinto.begin(); it != quinto.end(); ++it)
        cout << ' ' << *it;
    cout << '\n';

    return 0;
}
```

Vetores

O vector é container (contentor ?) para guardar elementos, de uma forma não indexada.

Initializer list

```
#include <vector>
```

```
class MyNumber
{
private:
    vector<int> mVec;
public:
    MyNumber(const initializer_list<int>& v) {
        for (auto itm : v) {
            // a keyword auto permite que o tipo de variável seja
            // automaticamente deduzida do seu inicializador
            mVec.push_back(itm); // Colocar no vetor pela ordem ...
        }
    }
    void print() {
        for (auto itm : mVec) {
            cout << itm << " ";
        }
    }
};
```

Se usarmos o **Initializer_list** como parâmetro no construtor, iremos criar objetos (usando como entrada).

```
MyNumber m = { 1,2 }; } main()
```

To see: https://en.cppreference.com/w/cpp/utility/initializer_list

Initializer list

```
#include <vector>
```

```
class MyString
{
private:
    vector<string> mStringVec;

public:
    MyString( const initializer_list<string>& v) {
        for (auto itm : v) {
            mStringVec.push_back(itm);
        }
    }

    void print() {
        for (auto itm : mStringVec) {
            cout << itm << " ";
        }
    }
};
```

Se usarmos o **Initializer_list** como parâmetro no construtor, iremos criar objetos (usando como entrada).

```
MyString s = { "Rtp1", "Rtp2" }; } main()
```

To see: https://en.cppreference.com/w/cpp/utility/initializer_list

Composição

COMPOSIÇÃO é a mais forte de todas as associações.
Corresponde à relação → **composto por um** → “Has a”



Uma pessoa é composta por um conjunto de dedos...mas
vão-se os anéis, fiquem os dedos, diz-se...

Adaptado: <http://home.iscte-iul.pt/~mms/>

Composição

- Existe uma relação com o tempo de vida de cada um dos objetos compostos.
- Os objectos que compõem um determinado objecto são construídos depois de construído esse objecto, **ou pelo menos ao mesmo tempo**, e são destruídos antes de destruído esse objecto, **ou quando muito ao mesmo tempo**.
- A construção e destruição dos objectos que compõem o objecto é da sua exclusiva responsabilidade, embora ocasionalmente seja delegada em terceiros.

A classe representa o todo, é que vai gerir as partes, os componentes.

Esses membros não sabem da existência do 'todo'.

ordem de chamada dos construtores e dos destrutores.

=

Fora para dentro - Construtores
Dentro para fora - Destrutores

Composição (na prática)

```
1 // ComposicaoAutomovelMotor.cpp : This file contains the 'main' function. Program execution begins and ends there.
2 //
3
4 #include <iostream>
5 #include "Car.h"
6 using namespace std;
7
8 int main()
9 {
10     Car myCar;
11
12     return 0;
13 }
14
15 // Run program: Ctrl + F5 or Debug > Start Without Debugging menu
16 // Debug program: F5 or Debug > Start Debugging menu
17
18 // Tips for Getting Started:
19 // 1. Use the Solution Explorer window to add/manage files
20 // 2. Use the Team Explorer window to connect to source control
21 // 3. Use the Output window to see build output and other messages
22 // 4. Use the Error List window to view errors
23 // 5. Go to Project > Add New Item to create new code files, or Project > Add Existing Item to add existing code files to the project
24 // 6. In the future, to open this project again, go to File > Open > Project and select the .sln file
```

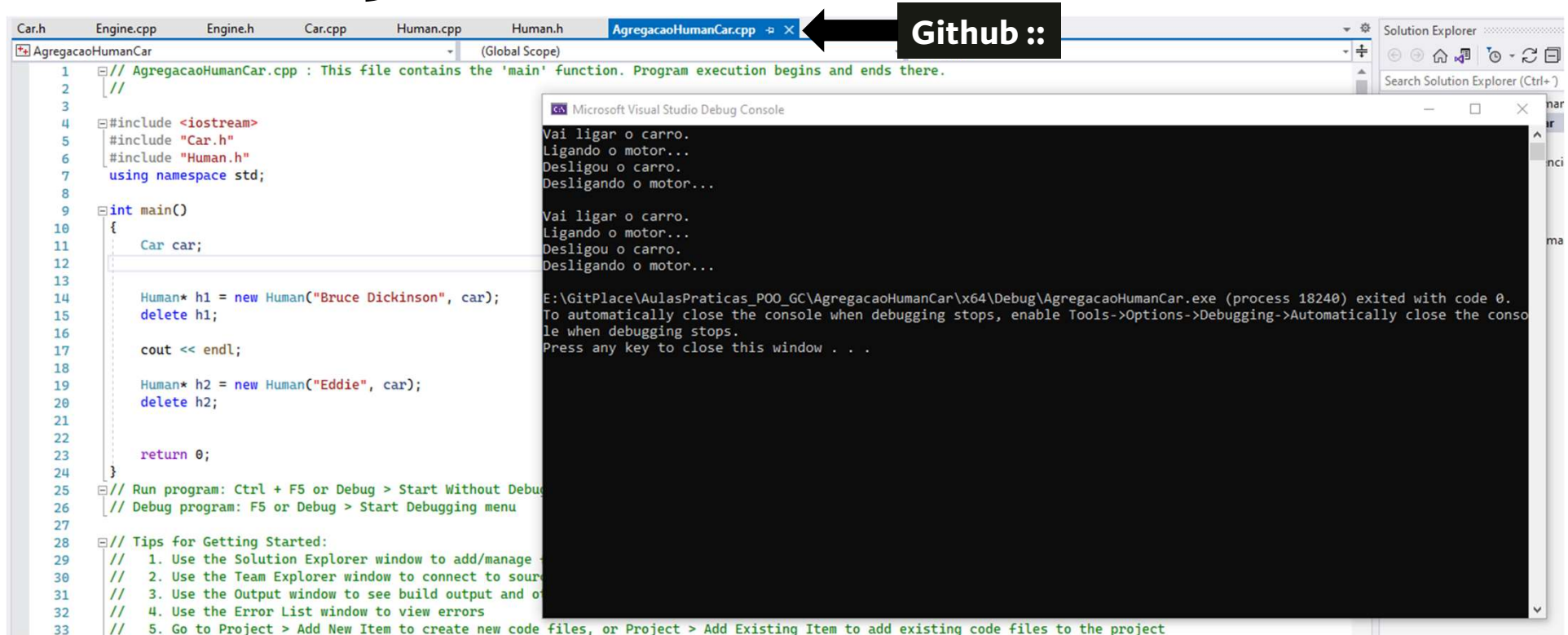


Adaptado: <http://home.iscte-iul.pt/~mms/>

Agregação

- Esses tipos de relações são chamados assim porque agregam valor para o objeto relacionado;
- Os tempos de vida de cada objeto é diferente e não dependente. Por exemplo, um professor (objeto) é adicionado a um departamento (objeto), mas entretanto o departamento é eliminado. O professor mantém-se “vivo”;
- Tipo de Relação: "tem um"
- **Resumindo:** a agregação implica um relacionamento em que o filho pode existir independentemente do pai.

Agregação (na prática)



The screenshot displays the Visual Studio IDE with the file `AgregacaoHumanCar.cpp` open. The code implements a `main` function that demonstrates aggregation by creating `Human` objects and associating them with a `Car` object. The debug console shows the program's execution, including messages for starting and stopping the car and motor, and a final exit message.

```
1 // AgregacaoHumanCar.cpp : This file contains the 'main' function. Program execution begins and ends there.
2 //
3
4 #include <iostream>
5 #include "Car.h"
6 #include "Human.h"
7 using namespace std;
8
9 int main()
10 {
11     Car car;
12
13
14     Human* h1 = new Human("Bruce Dickinson", car);
15     delete h1;
16
17     cout << endl;
18
19     Human* h2 = new Human("Eddie", car);
20     delete h2;
21
22
23     return 0;
24 }
25 // Run program: Ctrl + F5 or Debug > Start Without Debugging
26 // Debug program: F5 or Debug > Start Debugging menu
27
28 // Tips for Getting Started:
29 // 1. Use the Solution Explorer window to add/manage files
30 // 2. Use the Team Explorer window to connect to source control
31 // 3. Use the Output window to see build output and application logs
32 // 4. Use the Error List window to view errors
33 // 5. Go to Project > Add New Item to create new code files, or Project > Add Existing Item to add existing code files to the project
```

Microsoft Visual Studio Debug Console

```
Vai ligar o carro.
Ligando o motor...
Desligou o carro.
Desligando o motor...

Vai ligar o carro.
Ligando o motor...
Desligou o carro.
Desligando o motor...

E:\GitPlace\AulasPraticas_POO_GC\AgregacaoHumanCar\x64\Debug\AgregacaoHumanCar.exe (process 18240) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Adaptado: <http://home.iscte-iul.pt/~mms/>

Agregação (na prática)

Github ::



The screenshot shows a Visual Studio IDE with a C++ file named 'Ficha03_Agregacao.cpp' open. The code defines two classes: 'Professor' and 'Departamento'. The 'Professor' class has a private member 'm_name' and public methods for creation, destruction, and getting the name. The 'Departamento' class has a private member 'm_Professor' and a public method for creation. The 'main' function in the file creates a 'Professor' object named 'Pedro' and a 'Departamento' object named 'o'.

```
1 // Ficha03_Agregacao.cpp : This file contains the 'main' function. Program execution begins and ends there.
2 //
3
4 #include <iostream>
5 #include <string>
6 using namespace std;
7
8 class Professor
9 {
10 private:
11     std::string m_name{};
12 public:
13     Professor(const std::string& name) : m_name{ name } {}
14     ~Professor() {}
15     void create() { cout << "Professor Criado !!!\n"; }
16     void destroy() { cout << "Professor Destruido !!!\n"; }
17     const std::string& getName() const { return m_name; }
18 };
19
20 class Departamento
21 {
22 private:
23     const Professor& m_Professor{};
24 public:
25     Departamento(const Professor& p) : m_Professor{ p } {}
26     void create() { cout << "Departamento Criado com o " << m_Professor.getName() << endl; }
27     void destroy() {}
28 };
29
30 int main()
31 {
32     Professor p("Pedro");
33     Departamento o(p);
34     p.create();
35     o.create();
36     p.destroy();
37     o.destroy();
38 }
```

The 'Microsoft Visual Studio Debug Console' window shows the output of the program:

```
Professor Criado !!!
Departamento Criado com o Pedro
Departamento Destruido
Pedro ainda existe!
Professor Destruido !!!
```

Ficha 3

(Composição / Agregação / Vectores / Ficheiros)

Considere **pontos** de um plano representados pelas suas **coordenadas cartesianas** x e y.

```

2
3
4
5
6
7
8
9
10
11
12
13
14
class ponto
{
public:
private:
    int x, y;
};

```

ponto.h

Não deve ser possível construir objectos desta classe sem a indicação das suas coordenadas (nota: “indicação” ≠ “perguntar ao utilizador”). Qualquer valor inteiro é válido, tanto para x como para y.

```

7
8
9
10
11
ponto::ponto(int xx, int yy): x(xx), y(yy) {
    cout << "Construido o ponto(x=" << x << ", y=" << y << ")" << endl;
}

```

ponto.cpp

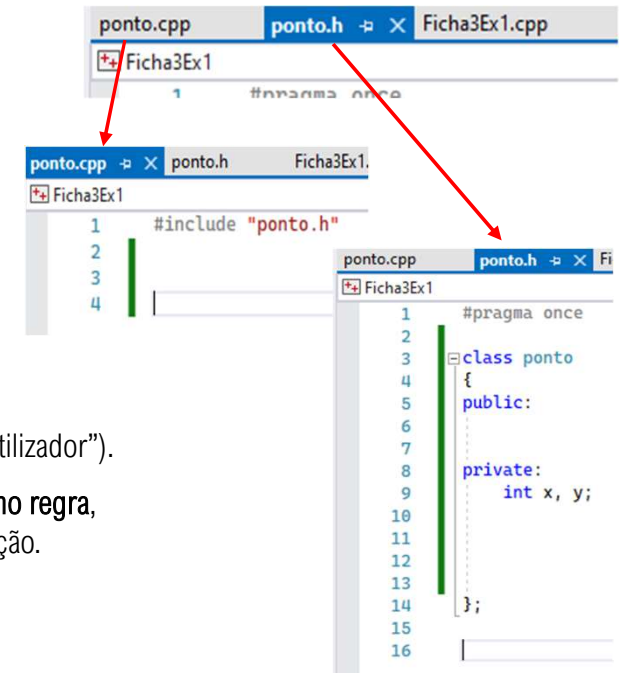
```

3
4
5
6
7
8
class ponto
{
public:
    ponto(int x=0, int y=0 );
}

```

Usar as listas de inicialização, **como regra**, ao contrário do método de atribuição.

Criar dois ficheiros .h e .cpp :



Ficha 3

(Composição / Agregação / Vectores / Ficheiros)

Deve ser possível obter e modificar cada uma das coordenadas, mas sem desrespeitar o conceito de encapsulamento. As funções que permitem **obter os dados** devem poder ser chamadas sobre **objectos Ponto** constantes, e as que modificam as coordenadas não.

```

3  class ponto
4  {
5  public:
6
7      ponto(int x=0, int y=0 );
8      ~ponto();
9      int getX() const;
10     int getY() const;
11     void setX(int x);
12     void setY(int y);
13 private:
14     int x, y;
15
16
17     ponto.h
18
19
20

```

```

int ponto::getX() const { return x; }
int ponto::getY() const { return y; }

void ponto::setX(int xx) { x = xx; }
void ponto::setY(int yy) { y = yy; }

ponto.cpp

```

```

3  class ponto
4  {
5  public:
6
7      ponto(int x=0, int y=0 );
8      ~ponto();
9      int getX() const;
10     int getY() const;
11     void setX(int x);
12     void setY(int y);
13
14     string getAsString() const;
15
16 private:
17     int x, y;
18
19     ponto.h
20

```

```

24 string ponto::getAsString() const {
25     ostringstream s;
26     s << "Pontos(" << this->x << "/" << this->y << ")" << endl;
27     return s.str();
28 }

ponto.cpp

```

Obter um **objecto string** com a **descrição textual** do seu conteúdo (formato "(x / y)")

```
string getAsString() const;
```

```
#include <string>
```

Não esquecer !!!

```
#include <sstream>
```

Não esquecer !!!

Ficha 3

(Composição / Agregação / Vectores / Ficheiros)

Obter o valor que corresponde à distância entre o ponto e um outro que é fornecido.

```
class ponto
{
public:
    ponto(int x=0, int y=0 );
    ~ponto();
    int getX() const;
    int getY() const;
    void setX(int x);
    void setY(int y);
    string getAsString() const;

    double distancia(ponto& p) const;

private:
    int x, y;
```

ponto.h

ponto.cpp

```
27
28 double ponto::distancia(ponto& p) const {
29     double dist;
30
31     dist = sqrt(((this->x - p.x) * (this->x - p.x)) + (this->y - p.y) * (this->y - p.y));
32
33     return dist;
34 }
35
36
```

```
#include <math.h>
```

Não esquecer !!!

Ficha 3

(Composição / Agregação / Vectores / Ficheiros)

Pretende-se uma classe Desenho caracterizada por um nome e um conjunto de rectângulos

```
10 class desenho
11 {
12     private:
13         string nome;
14         vector <Retangulo> figuras;
15
16     public:
17
18
```

Usamos um **vector**, temos de aprender a **adicionar** e **remover** objetos.

```
20
21 bool desenho::adicionaFigura(Retangulo r) {
22     this->figuras.push_back(r);
23     cout << r.getAsString() << endl;
24
25     return true;
26 }
27
```

```
4
5 #include <vector>
6
```

Não esquecer !!!

```
28 void desenho::removeAreaMaior(double a) {
29
30     auto i = this->figuras.begin();
31     while (i != this->figuras.end()) {
32         if (i->area() > a)
33             this->figuras.erase(i);
34         else
35             ++i;
36     }
37
38 }
```

Ficha 4

(Operadores)

Em C++, o utilizador (class designer) pode definir operadores. Significa que pode estabelecer operadores com um significado especial para um determinado tipo de dados, esta capacidade é conhecida por Operator Overloading.

Por exemplo, podemos definir um operador '+' que permite a concatenação de duas strings.

Dando assim a possibilidade de **polimorfismo** – dando assim um significado especial a um operador já existente.

Operadores que podem ser usados

+	-	*	/	%	^	&		~
!	=	<	>	+=	-=	*=	/=	%=
^=	&=	=	<<	>>	<<=	>>=	==	!=
<=	>=	&&		++	--	,	->*	->
()	[]	new	delete	new[]	delete[]			

Ficha 4

(Operadores)

overload as a member

```

20
21 // define overloaded + (plus) operator
22 complx complx::operator+ (const complx& c) const
23 {
24     complx result;
25     result.real = (this->real + c.real);
26     result.imag = (this->imag + c.imag);
27     return result;
28 }
29

```

```

1 // This example illustrates overloading the plus (+) operator.
2
3 #include <iostream>
4 using namespace std;
5
6 class complx
7 {
8     double real,
9     double imag;
10 public:
11     complx( double real = 0., double imag = 0.); // constructor
12     complx operator+(const complx&) const;      // operator+()
13 };
14
15 // define constructor
16 complx::complx( double r, double i )
17 {
18     real = r; imag = i;
19 }
20
21 // define overloaded + (plus) operator
22 complx complx::operator+ (const complx& c) const
23 {
24     complx result;
25     result.real = (this->real + c.real);
26     result.imag = (this->imag + c.imag);
27     return result;
28 }
29
30 int main()
31 {
32     complx x(4,4);
33     complx y(6,6);
34     complx z = x + y; // calls complx::operator+()
35 }

```

