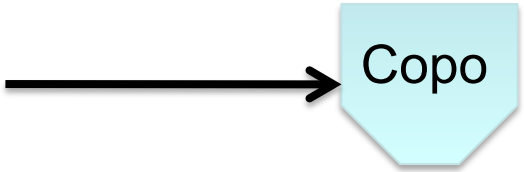
Universidade do Minho

Aula 6: Apontadores, Vectores, Memória Dinâmica e Polimorfismo

Objectivos desta aula:

- **Manipular** apontadores e vectores em C++;
- **Aplicar** o conceitos de memória dinâmica em C++;
- **Aplicar** polimorfismo em C++;

Apontadores



Universidade do Minho

- Em C/C++, existe o conceito de **apontador**, tipo de dados que "aponta" para outro tipo de dados (uma posição de memória);

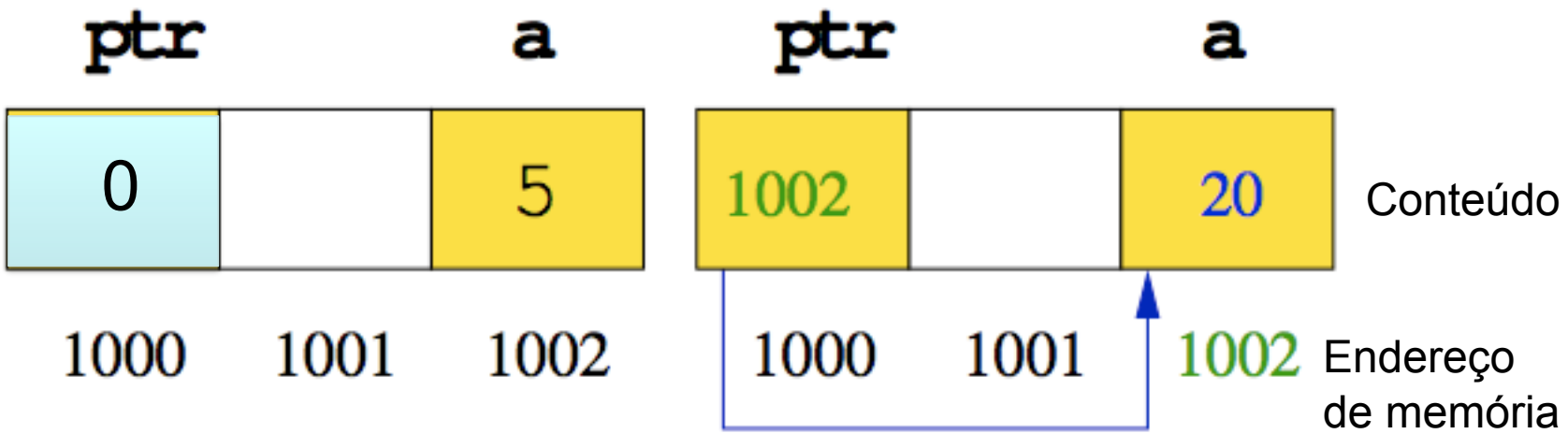
- Declaração: **tipo *var;**
- Valor apontado: ***var**
- Endereço de uma variável: **&var**
- Campo/função (**x**) de valor apontado:
  (*var).x
  var->x

- **Ver: http://www.cs.stanford.edu/cslibrary/PointerFunCpp.avi**

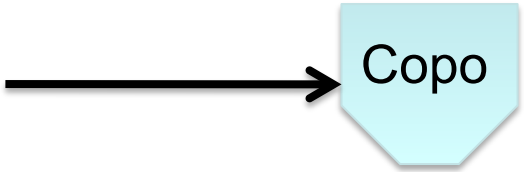
Apontadores: Exemplo

Universidade do Minho

```
int a=5;
int *ptr=0; // inicializar ptr a NULL (vazio)
ptr=&a; // ptr aponta para a, assumir que &a=1002
cout << a << *ptr; // 5 5
*ptr=20; // altera o valor apontado de ptr
cout << a; // 20
```



Apontadores

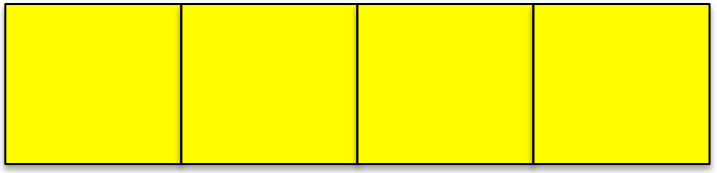


Universidade do Minho

- Podem aplicar operadores a apontadores:
+,-,<,>,<=,>=,==,!=,=,++,--,+=,-=
- É bom hábito inicializar sempre um apontador: **int *ptr=0;**
- É possível ter um apontador de apontadores: **int **ptr;**
- Podem ser usados em funções: **int*** converte(**char *c**);
- void* é genérico (aponta para qualquer coisa):

```
int a=5;
void *ptr=(void *)&a; // conversão de (int*) para (void *)
cout << *((int *)ptr); // conversão de (void*) para (int*)
```

Vectores (arrays)

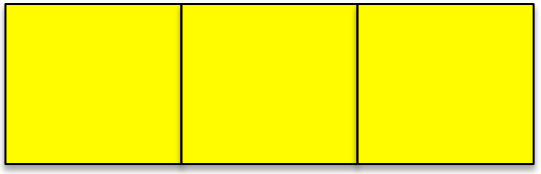


Universidade do Minho

- **Conjunto** de elementos do mesmo **tipo**;
- Elementos **indexados**, entre 0 e n-1 (n=comprimento do vector);
- Declaração: **tipo** var**[n]**;
- Inicialização automática: tipo var[n]**= {v1,...,vn}**;
- Acesso ao elemento i: **var[i]**;

```
int a[4]= {1,1,1,1}; // declarar e inicializar a
a[0]=7; a[1]=6; a[2]=3; a[3]=4; // alterar a
```

Indice	0	1	2	3
Valor	7	6	3	4



Vectores: manipulação

Universidade do Minho

- Temos de saber o tamanho do array (n)**: int func(int v[], int n);**
- Percorrer todo array, ciclo de 0 a n-1: **for(i=0;i<n;i++) {v[i]…}**

- Algoritmos fundamentais: **Somatório**

```
int sum=0; int i;
for(i=0;i<n;i++) sum=sum+v[i];
```

Vectores: manipulação

Universidade do Minho

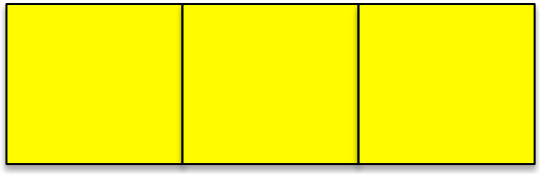
```
int maximo(int v[], int n)
{ int max;
  
  return max;
}
```

Q1: Preencher __? (In-Class Teams):

-Grupos de 3 elementos (2 min.)

-Descubram quem é último em termos da ordem alfabética → Representante

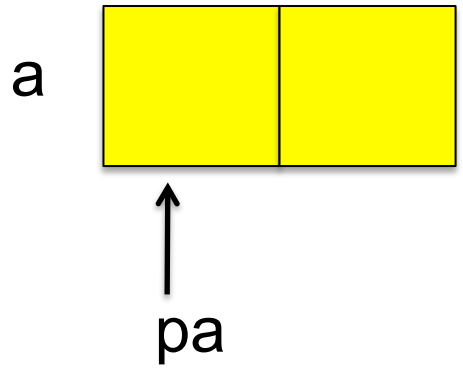
Vectores e Classes



Universidade do Minho

```
#include <iostream> // organizer2.cc ou organizer2.cpp
#include "person.h" // para ter acesso a classe
using namespace std; // definicao do namespace std
int main()
{
Person p1("Rui",23);
Person p2("Ivo",24);
Person v[2]; Person *vp1=&p1;
v[0]=p1; v[1]=p2;
cout << "Nome de v[1]:" << v[1].name() << "\n"; // Ivo
cout << “Nome de vp1:” << vp1->name() << ”\n";
return 0;
}
```

Vectores e apontadores

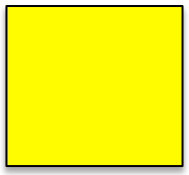


Universidade do Minho

- Usar um * por dimensão: **int a[2]; int *pa;**
- Declaração: **tipo v[n]; tipo *ptr;**
- Inicialização: **v==&v[0]; ptr=&v[0];**
- Acesso ao elemento i: **v[i]; *(ptr+i);**
- Funções: **func(int v[i], int n); func(int *v, int n);**

v	ptr	endereço de início do array
v[0]	*ptr	1º elemento
&v[1]	ptr+1	endereço do 2º elemento
v[1]	*(ptr+1)	2º elemento

Memória dinâmica

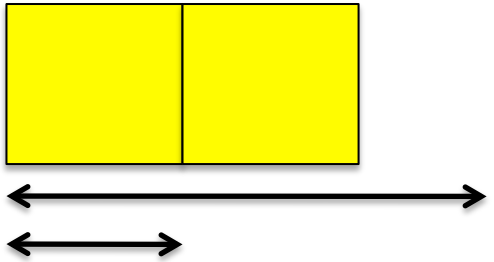


Universidade do Minho

- Em C++, são definidos dois operadores:
  - **new –** reservar memória, chama o respectivo construtor (se existir)
  - **delete** – libertar memória, chama o destrutor (se existir);
- **Regra**: por cada new usar um delete;

```
double *x=new double; // double, inicializa a 0
*x=3.1415; cout << (*x); // 3.1415
delete x; // liberta a memoria de x
int *y=new int(3); // int, inicializa a 3
cout << (*y); // 3
delete y; // liberta a memoria de y
string *s=new string("ola"); // construtor da string
delete s; // liberta a memoria de s
```

Memória dinâmica: vectores



Universidade do Minho

- Em vectores, usa-se: **new[]** e **delete[]**;

```
int *z; // array dinamico
int a; cin >> a; z=new int[a]; // reservar espaço
// usar z
delete[] z; // libertar a memória de z, liberta a*sizeof(int) bytes

string *a=new string[10]; // array com espaço para 10 strings
for(int i=0;i<10;i++) a[i]="ola";
delete[] a; // apagar um array

Person *a=new Person[2]; // array com espaço para 2 pessoas
a[0]=p1; a[1]=p2;
delete[] a; // libertar a
```

Memória dinâmica: alargar vectores

Universidade do Minho

```
#include <string> // exemplo para array de strings
using namespace std;
string *enlarge(string *old, int oldsize, int newsize)
{ string *res = new string[newsize]; // reservar novo
for (int i = 0; i < oldsize; i++)
    res[i] = old[i]; // copiar old para res
delete[] old; // apagar old
return res; // retornar array
}
int main()
{ string *arr = new string[4]; // array com 4 strings
  arr = _____; // alargar para 6
}
```

Q2: Preencher __? (In-Class Teams):

-Grupos de 3 elementos (1 min.)

-Descubram quem tem cabelos mais compridos → Representante

Memória dinâmica dentro de classes

Universidade do Minho

- Usa-se do mesmo modo, contudo:
  - O **construtor** tem de reservar a memória;
  - É necessário definir o **destrutor** para libertar a memória;
- Um **destrutor** é sempre activado:
  - no final de um bloco/função para um objecto local;
  - quando o programa termina para um objecto global;
  - quando um objecto é apagado via delete;
  - quando um array dinâmico é apagado via **delete[]**;
  - no caso de classes derivadas, segue-se a ordem de uma pilha (*stack*): primeiro o destrutor da classe mais abaixo, depois o destrutor da classe ascendente, etc...

Exemplo:


```
// ficheiro ages.h:
class Ages
{ int *d_ages; // vector dinamico
  int d_size; // comprimento actual de d_ages
public:
  Ages(int size); // construtor
~Ages(); // destrutor
};
```

```
// ficheiro ages.cc:
Ages::Ages(int size) // construtor
{ d_size=size; d_ages=new int[d_size];}
Ages::~Ages() // destrutor
{ delete[] d_ages;}
```

Exemplo:


```
#include "ages.h" // programa com a main
int main()
{
 Ages A(5); // A tem capacidade para 5 idades
 Ages *B=new Ages(20); // B com 20 idades
 // ...
 delete B; // destrutor de B activado
return 0;
} // destrutor de A activado no fim da funçao
```

Copia de classes:

- Para copiar classes com atributos dinâmicos, é necessário copiar cada um dos seus atributos;
- Este processo é facilitado se for definido o respectivo método ou se for criado um construtor;

```
// age.cc ou age.cpp
Age::Age(Age const &A)
{ d_size = A.d_size; // copiar o conteudo de A
  d_ages = new int[d_size];
 for(int i=0;i<d_size;i++)  d_ages[i]=A.d_ages[i]; }

void Age::copy(Age const &A)
{ delete[] d_ages; // apagar a memoria utilizada
  d_size = A.d_size; // copiar o conteudo de A
  d_ages = new int[d_size];
  for(int i=0;i<d_size;i++) d_ages[i]=A.d_ages[i];
}
```

Polimorfismo

- O objectivo é ter um **apontador** que pode apontar para diferentes objectos;
- Cada **objecto comporta-se conforme o seu tipo**;
- É possível usar o mesmo método mas com comportamentos diferentes conforme o objecto;
- Em C++, obtem-se isso utilizando apontadores e **métodos virtuais**;
- Se um método for **virtual**, significa que pode ser "redefinido" numa classe descendente;
- Em C++, existe a classe **typeid**, que ajuda a verificar em tempo real de execução qual o tipo de um dado objecto;

Polimorfismo: exemplo completo

Universidade do Minho

```
#include <iostream>
#include <typeinfo> // biblioteca para acesso ao typeid
using namespace std;

class Figura // classe base, mais generica
{
 double d_largura;
 double d_altura;
public:
Figura(){} // const. vazio
Figura(double largura,double altura):d_largura(largura),d_altura(altura){}
virtual double area() const { cerr << "Erro.\n"; return -1.0;}
double largura() const{ return d_largura;}
double altura() const{ return d_altura;}
};
```

Polimorfismo: exemplo completo


```
class Quadrado: public Figura // classe descendente
{
public:
Quadrado(){}
Quadrado(double lado):Figura(lado,lado){}
virtual double area() const { return largura()*altura();}
};

class Triangulo: public Figura // classe descendente
{
public:
Triangulo(){}
Triangulo(double largura, double altura):Figura(largura,altura){}
virtual double area() const { return largura()*altura()/2;}
};
```

Polimorfismo: exemplo completo


```
int main() // eis a 1a versao do exemplo
{
  Figura a(4,2); Quadrado b(4); Triangulo c(4,4);
  cout << typeid(a).name() << "\n";
  cout << typeid(b).name() << "\n";
  cout << typeid(c).name() << "\n";

  Figura *p=&a; // apontador para figura
  cout << p->area() << '\n';
  cout << (typeid(*p)==typeid(Figura)) << '\n';
  p = &b;
  cout << p->area() << '\n';
  cout << (typeid(*p)==typeid(Quadrado)) << '\n';
}
```

```
6Figura
8Quadrado
9Triangulo
Erro.
-1
1
16
1
```

Polimorfismo: exemplo completo


```
int main() // eis a 2a versao do exemplo,  mais elaborada
{
  Figura **v=new Figura*[2]; // vector de apontadores para Figura
  v[0]=new Quadrado(4); v[1]=new Triangulo(4,4);
  for(int i=0;i<2;i++)
  {
   cout << typeid(*v[i]).name() << "\n";
   cout << v[i]->area() << "\n";
   cout << (typeid(*v[i])==typeid(Quadrado)) << "\n";
   cout << (typeid(*v[i])==typeid(Triangulo)) << "\n";
  }
  for(int i=0;i<2;i++) delete v[i]; // apagar conteudo v
  delete[] v; // apagar v
}
```

```
8Quadrado
16
1
0
9Triangulo
8
0
1
```