

Apuntadors

Tema 2: Estructures de dades dinàmiques

Objectius

- Introduir estructures de dades dinàmiques
 - L'espai de memòria que utilitzen és variable (dinàmic) i s'adapta al nº d'elements reals que tenim en cada moment.
 - Conceptes bàsics:
 - Apuntadors
 - Memòria dinàmica
 - Exemples d'estructures de dades dinàmiques lineals:

Piles



Cues



Llistes



TOP TEN RICHEST AMERICANS			
1. Bill Gates (MS)	\$1.02	Microsoft	
2. Warren Buffett (BR)	\$799	Capital One	
3. Paul Allen (MS)	\$750	Microsoft	
4. Mark Zuckerberg (FB)	\$660	Meta	
5. Larry Page (GO)	\$600	Google	
6. Jeff Bezos (AM)	\$580	Amazon	
7. Elon Musk (TX)	\$450	SpaceX	
8. Michael Bloomberg (BB)	\$430	Bloomberg	
9. Mark Zuckerberg (FB)	\$420	Meta	
10. Larry Page (GO)	\$410	Google	

Limitacions estructures dades estàtiques

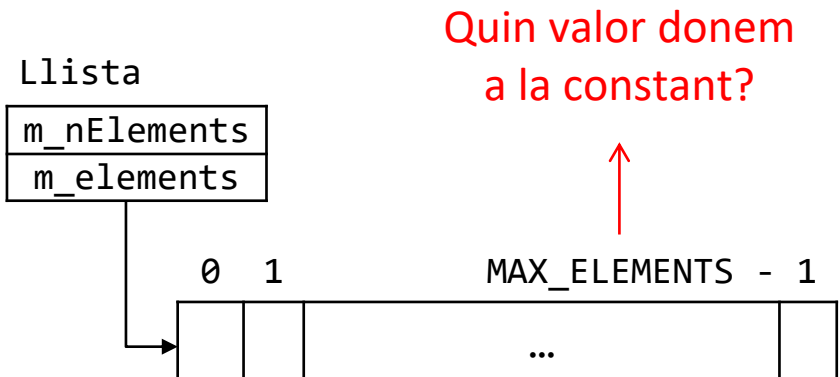
Exercici 7 – Tema 1

Volem crear una classe `Llista` que sigui el més genèrica possible per poder guardar i manipular llistes d'objectes de diferents tipus amb poques modificacions. En aquest exercici la utilitzarem per guardar objectes del tipus `Data` que hem definit a l'exercici 5, però amb petites modificacions podria fer-se servir per guardar objectes de qualsevol altre tipus.

La classe `Llista` ha de permetre guardar un conjunt qualsevol de dates (amb un màxim de 100 elements dins de la llista) i ha de permetre fer operacions per afegir, eliminar i recuperar elements de la llista, ordenar-la i llegir i escriure els elements de la llista a un fitxer.

...

```
class Llista
{
public:
    ...
private:
    static const int MAX_ELEMENTS = 100;
    Data m_elements[MAX_ELEMENTS];
    int m_nElements;
};
```



Limitacions estructures dades estàtiques

Quin valor donem a la constant?

- Quin és el màxim número d'elements que es poden haver de guardar a la llista? Com podem definir un valor genèric adequat per tots els casos?
- Què passa si s'arriba a superar el n° màxim que hem fixat amb la constant?
- Com afecta a la memòria ocupada en total si fixem un valor molt gran?
 - Executem el programa...
 - Quanta memòria ocupa abans d'afegir cap element a la llista?

Limitacions estructures dades estàtiques

- Suposem que tenim una llista que guarda dades d'objectes d'una classe Estudiant (amb atributs nom i niu) i volem afegir i eliminar elements a la llista conservant l'ordre alfabètic

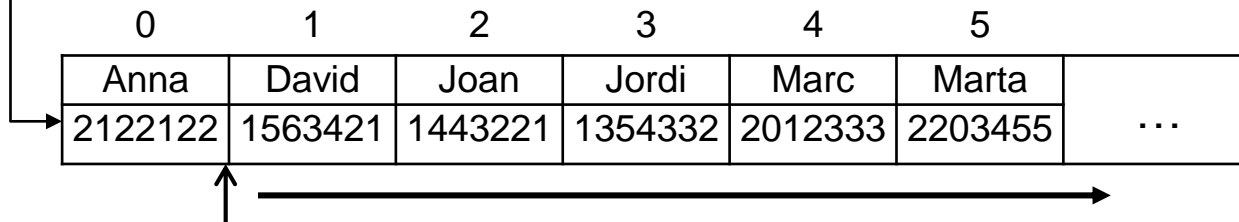
```
class Llista
{
public:
    ...
private:
    static const int MAX_ELEMENTS = 100;
    Estudiant m_elements[MAX_ELEMENTS];
    int m_nElements;
};
```

```
class Estudiant
{
public:
    ...
private:
    string m_nom;
    string m_NIU;
};
```

Llista

m_nElements
m_elements

0	1	2	3	4	5	
Anna	David	Joan	Jordi	Marc	Marta	...
2122122	1563421	1443221	1354332	2012333	2203455	



Estudiant nou: Carles
2004050

1. Buscar posició on inserir
2. Desplaçar elements de l'array

Limitacions estructures dades estàtiques

- Suposem que tenim una llista que guarda dades d'objectes d'una classe Estudiant (amb atributs nom i niu) i volem afegir i eliminar elements a la llista conservant l'ordre alfabètic

Llista

m_nElements
m_elements

0	1	2	3	4	5	6	
Anna	Carles	David	Joan	Jordi	Marc	Marta	...
2122122	2004050	1563421	1443221	1354332	2012333	2203455	

1. Buscar posició on inserir
2. Desplaçar elements de l'array
3. Inserir el nou estudiant a la posició alliberada

- Executem el programa...
- Quant temps triga a executar-se?

Estàtic vs. dinàmic

Problemes de les estructures de dades estàtiques:

1. Dificultat per preveure la memòria necessària (100 elements o 50000 elements?)
2. A l'hora d'inserir i eliminar pot ser necessari moure bona part dels elements

Alternativa: estructures de dades dinàmiques

- Necessitem poder construir estructures de dades que puguem recórrer fàcilment i on puguem afegir nous elements sense imposar-nos un nombre màxim d'elements en temps de codificació.
- Aquest tipus d'estructures s'anomenen dinàmiques:
 - Arrays dinàmics
 - Llistes
 - Piles
 - Cues
- L'eina que permet implementar estructures de dades dinàmiques són els **apuntadors**.

El tipus apuntador

Exemple:

➤ Què fa aquest programa?

```
int main()
{
    int x, y;
    int *p, *q;
    x = 5;
    y = 0;

    p = &x;
    y = *p;
    *p = 10;
    q = p;
    p = &y;
    *p = *q + 2;

    cout << "x: " << x << ", &x: " << &x << endl;
    cout << "y: " << y << ", &y: " << &y << endl;
    cout << "*p: " << *p << ", p: " << p << endl;
    cout << "*q: " << *q << ", q: " << q << endl;
    return 0;
}
```

→ Declaració de variables de tipus apuntador

El tipus apuntador

Definició:

- El tipus **apuntador** és un tipus de dades simple que permet guardar una adreça de memòria d'una variable **d'un tipus determinat** (direccionament indirecte)

Declaració:

```
<tipus>*    <nom_variable>;
```

Exemples:

```
int* pEnter;
```

```
float* pReal;
```

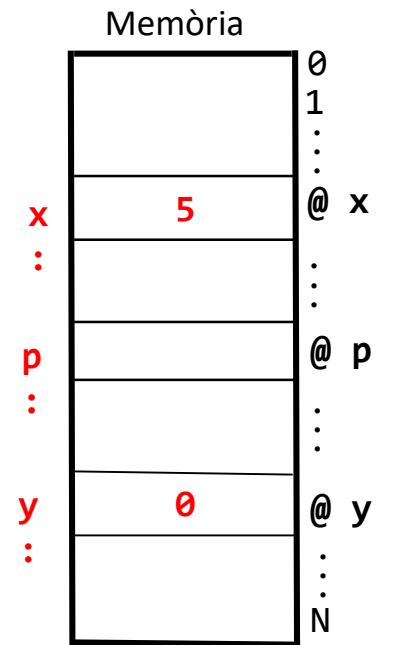
```
Estudiant* pEstudiant;
```

El tipus apuntador

Operadors

```
int x,y;  
int *p;
```

```
x = 5;  
y = 0;
```



Adreces
memòria

El tipus apuntador

Operadors

- Operador &

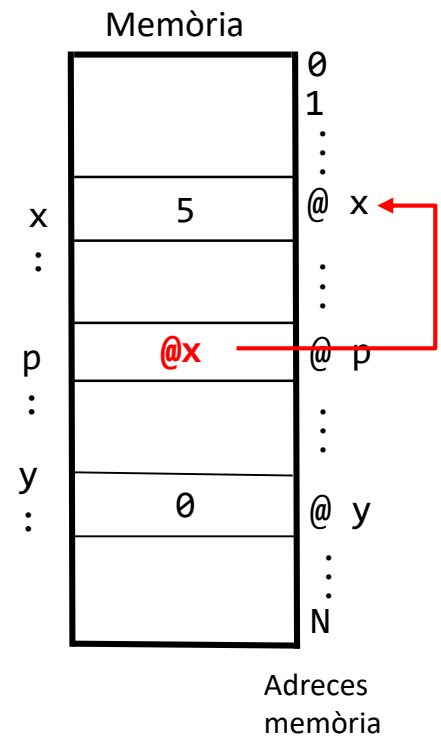
&x: retorna l'adreça de memòria de l'objecte x

```
int x,y;  
int *p;
```

```
x = 5;  
y = 0;
```

```
p = &x;
```

- Una variable de tipus **apuntador** *p* conté una adreça de memòria
- Si *p* té l'adreça de *x* (*p* = &*x*) diem que:
 - p* **apunta** a *x*
 - $p \rightarrow x$



El tipus apuntador

Operadors

■ Operador &

&x: retorna l'adreça de memòria de l'objecte x

■ Operador *

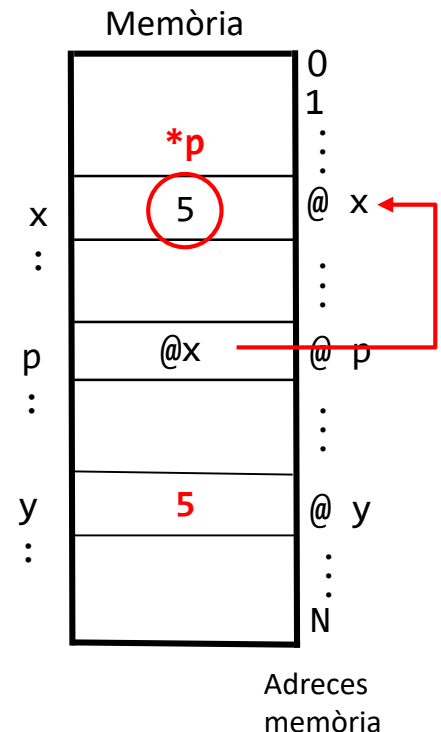
*p: retorna el valor de l'objecte que està a l'adreça de memòria apuntada per p

```
int x,y;  
int *p;
```

```
x = 5;  
y = 0;
```

```
p = &x;  
y = *p;  
*p = 10;
```

- Una variable de tipus **apuntador** *p* conté una adreça de memòria
- Si *p* té l'adreça de *x* (*p* = &*x*) diem que:
 - *p* **apunta** a *x*
 $p \rightarrow x$
 - *p* **fa referència** o **referencia** *x*
 $*p \longleftrightarrow x$



El tipus apuntador

Operadors

■ Operador &

&x: retorna l'adreça de memòria de l'objecte x

■ Operador *

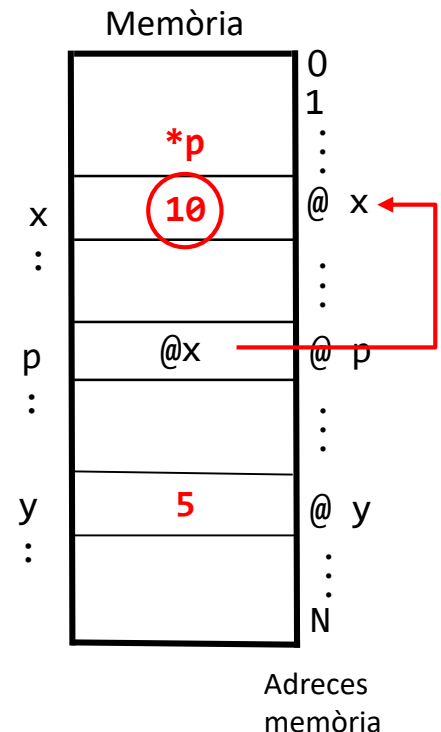
*p: retorna el valor de l'objecte que està a l'adreça de memòria apuntada per p

```
int x,y;  
int *p;
```

```
x = 5;  
y = 0;
```

```
p = &x;  
y = *p;  
*p = 10;
```

- Una variable de tipus **apuntador** *p* conté una adreça de memòria
- Si *p* té l'adreça de *x* (*p* = &*x*) diem que:
 - *p* **apunta** a *x*
 $p \rightarrow x$
 - *p* **fa referència** o **referencia** *x*
 $*p \longleftrightarrow x$



El tipus apuntador

Exemple:

➤ Què fa aquest programa?

```
int main()
{
    int x, y;
    int *p, *q;

    x = 5;
    y = 0;

    p = &x;
    y = *p;
    *p = 10;
    q = p;
    p = &y;
    *p = *q + 2;

    cout << "x: " << x << ", &x: " << &x << endl;
    cout << "y: " << y << ", &y: " << &y << endl;
    cout << "*p: " << *p << ", p: " << p << endl;
    cout << "*q: " << *q << ", q: " << q << endl;
    return 0;
}
```

	x	y	p	q
1	5	-	-	-
2	5	0	-	-
3	5	0	@x	-
4	5	5	@x	-
5	10	5	@x	-
6	10	5	@x	@x
7	10	5	@y	@x
8	10	12	@y	@x

El tipus apuntador

Operadors

- Operador &

&x: retorna l'adreça de memòria de l'objecte x

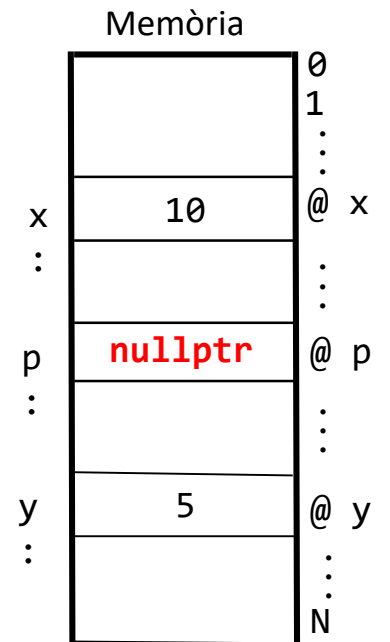
- Operador *

*p: retorna el valor de l'objecte que està a l'adreça de memòria apuntada per p

- Valor Null

Valor especial que indica que l'apuntador no apunta a cap adreça vàlida

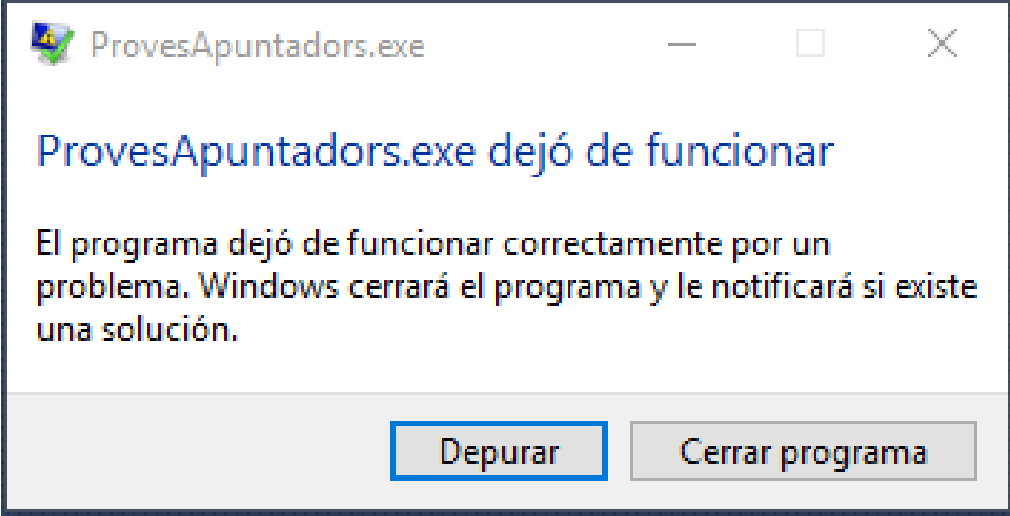
```
int x,y;  
int *p;  
  
x = 5;  
y = 0;  
  
p = &x;  
y = *p;  
*p = 10;  
p = nullptr;
```



Adreces
memòria

El tipus apuntador

Operadors

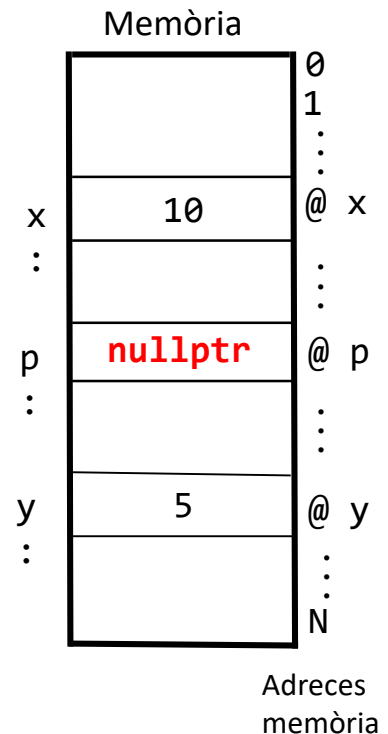
- 
- El programa dejó de funcionar correctamente por un problema. Windows cerrará el programa y le notificará si existe una solución.

No apunta enlloc !

```
int x,y;  
int *p;  
  
x = 5;  
y = 0;  
  
p = &x;  
y = *p;  
*p = 10;  
p = nullptr;  
*p = 0;
```

- Valor NULL

Valor especial que indica que l'apuntador no apunta a cap adreça vàlida



Apuntadors a objectes

Exemple:

➤ Què fa aquest programa?

```
int main()
{
    Complex numero1, numero2;
    Complex *pNumero1, *pNumero2;

    pNumero1 = &numero1;
    (*pNumero1).setReal(2.5);
    pNumero1->setImg(1.0);
    pNumero2 = pNumero1;
    pNumero1 = &numero2;
    pNumero1->setReal(pNumero2->getReal() * 2);
    pNumero1->setImg(numero1.getImg() * numero2.getReal());
    cout << "numero1: " << numero1 << endl;
    cout << "pNumero1: " << pNumero1 << endl;
    cout << "&numero1: " << &numero1 << endl;
    cout << "*pNumero1 " << *pNumero1 << endl;
    cout << "numero2: " << numero2 << endl;
    cout << "pNumero2: " << pNumero2 << endl;
    cout << "&numero2: " << &numero2 << endl;
    cout << "*pNumero2 " << *pNumero2 << endl;
    return 0;
}
```

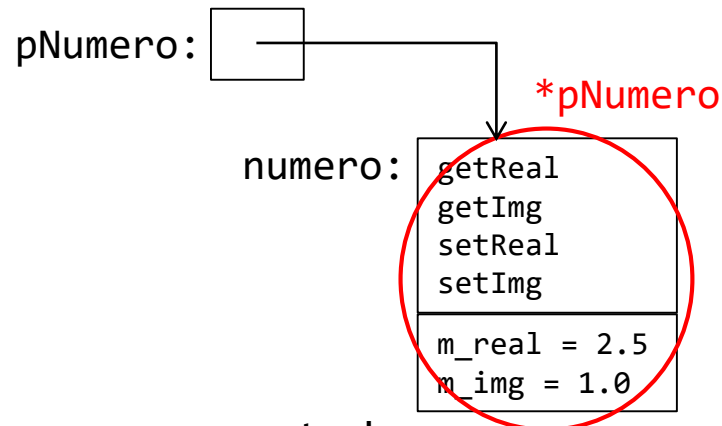
Apuntadors a objectes

- Si tenim un apuntador p a un objecte, l'accés als mètodes/atributs de l'objecte es fa amb la sintaxi habitual a partir de l'operador contingut aplicat a l'apuntador, *p

```
class Complex
{
public:
    void setReal(float);
    void setImg(float);
    float getReal();
    float getImg();
private:
    float m_real;
    float m_img;
};
```

```
Complex numero;
Complex *pNumero;

pNumero = &numero;
(*pNumero).setReal(2.5);
pNumero->setImg(1.0);
```



$(*p).nom_metode$ \longleftrightarrow

$p->nom_metode$

```
(*pNumero).getReal();
(*pNumero).getImg();
(*pNumero).setReal(0);
(*pNumero).setImg(1);
```



```
pNumero->getReal();
pNumero->getImg();
pNumero->setReal(0);
pNumero->setImg(1);
```

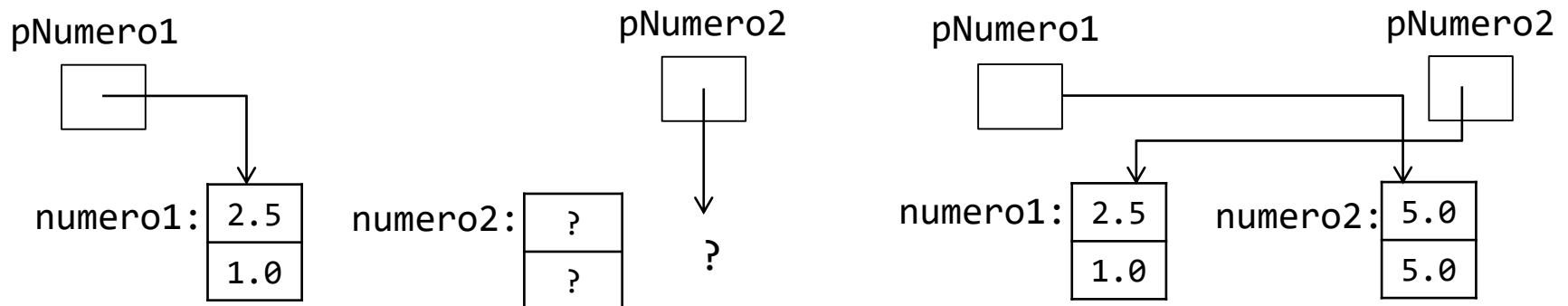
Apuntadors a objectes

Exemple:

➤ Què fa aquest programa?

```
int main()
{
    Complex numero1, numero2;
    Complex *pNumero1, *pNumero2;

    pNumero1 = &numero1;
    (*pNumero1).setReal(2.5);
    pNumero1->setImg(1.0);
    pNumero2 = pNumero1;
    pNumero1 = &numero2;
    pNumero1->setReal(pNumero2->getReal() * 2);
    pNumero1->setImg(numero1.getImg() * numero2.getReal());
    ...
}
```



Apuntadors i pas de paràmetres per referència

```
void intercanvia(int* p_x, int* p_y)
{
    int tmp;

    tmp = *p_x;
    *p_x = *p_y;
    *p_y = tmp;
}
```

➤ Què fa aquest programa?

```
int main()
{
    int x, y;

    cin >> x >> y;
    intercanvia(&x, &y);
    cout << x << " " << y;

    return 0;
}
```

Apuntadors i pas de paràmetres per referència

```
void intercanvia(int* p_x, int* p_y)
{
    int tmp;           p_x:  p_y: 

    tmp = *p_x;
    *p_x = *p_y;
    *p_y = tmp;
}
```

```
int main()
{
    int x, y;           x:  0 y:  5

    cin >> x >> y;
    intercanvia(&x, &y);
    cout << x << " " << y;

    return 0;
}
```

Apuntadors i pas de paràmetres per referència

```
void intercanvia(int* p_x, int* p_y)
{
    int tmp;

    tmp = *p_x;
    *p_x = *p_y;
    *p_y = tmp;
}
```

p_x: @x p_y: @y

p_x = @x p_y = @y

```
int main()
{
    int x, y;

    cin >> x >> y;
    intercanvia(&x, &y);
    cout << x << " " << y;

    return 0;
}
```

x: 0 y: 5

*p_x *p_y

Apuntadors i pas de paràmetres per referència

```
void intercanvia(int* p_x, int* p_y)
{
    int tmp;
    tmp = *p_x;
    *p_x = *p_y;
    *p_y = tmp;
}
```

p_x: @x p_y: @y

```
int main()
{
    int x, y;

    cin >> x >> y;
    intercanvia(&x, &y);
    cout << x << " " << y;

    return 0;
}
```

x: 0 y: 0

Apuntadors vs. referències

Utilitzant **apuntadors**

```
void intercanviaP(int* p_x, int* p_y)
{
    int tmp;

    tmp = *p_x;
    *p_x = *p_y;
    *p_y = tmp;
}
```

Declaració paràmetre

Utilització paràmetre

Utilitzant **referències**

```
void intercanviaR(int& r_x, int& r_y)
{
    int tmp;

    tmp = r_x;
    r_x = r_y;
    r_y = tmp;
}
```

Totes dues funcions realitzen el mateix, l'intercanvi del valor de les variables

```
int main()
{
    int x, y;
    cin >> x >> y;

    intercanviaP(&x, &y);
    cout << x << " " << y;

    return 0;
}
```

Pas de paràmetres a la crida a la funció

```
int main()
{
    int x, y;
    cin >> x >> y;

    intercanviaR(x, y);
    cout << x << " " << y;

    return 0;
}
```


Apuntadors vs. referències

- Tant els **apuntadors** com les **referències** permeten implementar el pas de paràmetres per referència.
- En el **llenguatge C** no existeixen ni les referències ni el pas de paràmetres per referència.
 - La utilització d'**apuntadors** permet **simular** el **pas de paràmetres per referència**, tot i que l'apuntador es passa com un paràmetre per valor.
- Les **referències** apareixen en el **llenguatge C++** per fer més eficient l'execució de constructors de còpia, i així minimitzar la còpia entre objectes en el pas de paràmetres.

Apuntadors vs. referències

- Tot i que els **apuntadors** tenen una funcionalitat similar a les **referències**, posseeixen algunes diferències:
 - Un **apuntador** és una variable que guarda una adreça de memòria, amb el propòsit d'actuar com un àlies a la variable que està en aquella adreça.
 - Una **referència** és una variable que es refereix a un objecte (un àlies)
 - Una **referència** vàlida sempre ha «d'apuntar» a un objecte. Per contra un apuntador pot apuntar a *res*, **nullptr** (el valor **nullptr** indica que no s'apunta a res)
 - La **referència** un cop definida, no se li pot re-assignar un altre valor, en canvi a l'apuntador sí

Quina diferència hi ha entre aquests dos fragments de codi?

```
int x = 0, y = 5;  
int* p;  
int* q = &y;  
p = &x;  
q = &y;  
p = q;  
*p = 10;  
p = nullptr;
```

```
int x = 0, y = 5;  
int& p = x;  
int& q = y;  
p = q;  
p = 10;
```

Exercici sessió 11 – voluntari pujar nota

Fer un programa que llegeixi tres n^ºs enters per teclat, els ordeni de més petit a més gran i els mostri ordenats per pantalla (utilitzant pas per referència amb apuntadors):

- Implementar una funció ordena, que agafi com a paràmetres 3 n^ºs enters (per referència amb apuntadors) i els retorni ordenats de més petit a més gran. Podeu utilitzar la funció intercanvia de l'exemple anterior
- Implementar el programa principal que llegeixi els 3 n^ºs enters, cridi a la funció ordena i mostri els n^ºs ordenats.

```
void intercanvia(int* p_x, int* p_y)
{
    int tmp;

    tmp = *p_x;
    *p_x = *p_y;
    *p_y = tmp;
}
```

Exercici: solució

```
void ordena(int* primer, int* segon, int* tercer)
{
    if (*primer > *tercer)
        intercanvia(primer, tercer);
    if (*segon > *tercer)
        intercanvia(segon, tercer);
    if (*primer > *segon)
        intercanvia(primer, segon);
}
```

```
int main()
{
    int x, y, z;

    cin >> x >> y >> z;
    ordena(&x, &y, &z);
    cout << x << " " << y << " " << z;

    return 0;
}
```

```
void intercanvia(int* p_x, int* p_y)
{
    int tmp;

    tmp = *p_x;
    *p_x = *p_y;
    *p_y = tmp;
}
```

Exercici

➤ Quin error hi ha en el codi d'aquest programa?

```
Complex llegeixComplex()
{
    Complex c;
    float real, img;

    cout << "Introdueix part real: ";
    cin >> real;
    c.setReal(real);
    cout << "Introdueix part imaginaria: ";
    cin >> img;
    c.setImg(img);
    return c;
}
```

```
int main()
{
    Complex c1 = llegeixComplex();

    cout << "c1: " << c1 << endl;

    int *p;

    p = &c1;
    p->setReal(c1.getReal() * 2);
    p->setImg(c1.getImg() * 2);

    cout << "c1: " << c1 << endl;
    cout << "*p: " << *p << endl;
}
```

Exercici: solució


```
int main()
{
    Complex c1 = llegeixComplex();

    cout << "c1: " << c1 << endl;

    int *p;

    p = &c1;
    p->setReal(c1.getReal() * 2);
    p->setImg(c1.getImg() * 2);

    cout << "c1: " << c1 << endl;
    cout << "*p: " << *p << endl;
}
```

 **'=': no se puede realizar la conversión de 'Complex*' a 'int*'**

- Els apuntadors contenen adreces de memòria, però associades a un determinat tipus de dades
 - Es declaren a partir d'un tipus base
 - El tipus utilitzat a la declaració de l'apuntador ha de coincidir amb el tipus de l'objecte apuntat

Exercici

➤ Quin error hi ha en el codi d'aquest programa?

```
Complex llegeixComplex()
{
    Complex c;
    float real, img;

    cout << "Introdueix part real: ";
    cin >> real;
    c.setReal(real);
    cout << "Introdueix part imaginaria: ";
    cin >> img;
    c.setImg(img);
    return c;
}
```

```
int main()
{
    Complex c1 = llegeixComplex();

    cout << "c1: " << c1 << endl;

    Complex *p;

    p->setReal(c1.getReal() * 2);
    p->setImg(c1.getImg() * 2);

    cout << "c1: " << c1 << endl;
    cout << "*p: " << *p << endl;
}
```

Exercici: solució

```
int main()
{
    Complex c1 = llegeixComplex();

    cout << "c1: " << c1 << endl;

    Complex *p;

    p->setReal(c1.getReal() * 2);
    p->setImg(c1.getImg() * 2);

    cout << "c1: " << c1 << endl;
    cout << "*p: " << *p << endl;
}
```

se utilizó la variable local 'p'
sin inicializar

- Igual que amb qualsevol altra variable és imprescindible inicialitzar correctament el valor de l'apuntador a una adreça vàlida abans d'utilitzar el seu valor (accedir al contingut de l'adreça de memòria)
 - Si no s'inicialitza correctament podem provocar errors d'accés a memòria en temps d'execució

Exercici

➤ Quin error hi ha en el codi d'aquest programa?

```
Complex* llegeixComplex()
{
    Complex c;
    float real, img;

    cout << "Introdueix part real: ";
    cin >> real;
    c.setReal(real);
    cout << "Introdueix part imaginaria: ";
    cin >> img;
    c.setImg(img);
    Complex *p = &c;
    return p;
}
```

```
int main()
{
    Complex* c1 = llegeixComplex();

    cout << "c1: " << c1 << endl;

    Complex *p = c1;

    p->setReal(c1->getReal() * 2);
    p->setImg(c1->getImg() * 2);

    cout << "c1: " << *c1 << endl;
    cout << "*p: " << *p << endl;
}
```

Exercici: solució

```
Complex* llegeixComplex()
{
    Complex c;
    float real, img;

    cout << "Introdueix part real: ";
    cin >> real;
    c.setReal(real);
    cout << "Introdueix part imaginaria: ";
    cin >> img;
    c.setImg(img);
    Complex *p = &c;
    return p;
}
```

warning C4172: devolució de
direcció de la variable
local o temporal: n

- Les variables locals d'una funció només existeixen a memòria mentre s'està executant la funció:
 - No podem retornar l'adreça de memòria d'una variable local

```
int main()
{
    Complex* c1 = llegeixComplex();

    cout << "c1: " << c1 << endl;

    Complex *p = c1;

    p->setReal(c1->getReal() * 2);
    p->setImg(c1->getImg() * 2);

    cout << "c1: " << *c1 << endl;
    cout << "*p: " << *p << endl;
}
```

Consola de depuració de Microsoft Visual St

```
Introdueix part real: 3
Introdueix part imaginaria: 4
c1: 0000002C4ECFF868
c1: -1.07374e+08 + -1.07374e+08i
*p: -1.07374e+08 + -1.07374e+08i
```

Exercici

- Quin error d'execució hi ha en el codi d'aquest programa?

```
Complex llegeixComplex()
{
    Complex c;
    float real, img;

    cout << "Introdueix part real: ";
    cin >> real;
    c.setReal(real);
    cout << "Introdueix part imaginaria: ";
    cin >> img;
    c.setImg(img);
    return c;
}
```

```
int main()
{
    Complex c1 = llegeixComplex();
    Complex c2 = llegeixComplex();

    cout << "c1: " << c1 << endl;
    cout << "c2: " << c2 << endl;

    Complex tmp;
    Complex *p_c1 = &c1;
    Complex *p_c2 = &c2;
    Complex *p_tmp = &tmp;


    p_tmp = p_c1;
    p_c1 = p_c2;
    p_c2 = p_tmp;

    cout << "c1: " << c1 << endl;
    cout << "c2: " << c2 << endl;
}
```

Exercici: solució

```
Complex llegeixComplex()
{
    Complex c;
    float real, img;

    cout << "Introdueix part real: ";
    cin >> real;
    c.setReal(real);
    cout << "Introdueix part imaginaria: ";
    cin >> img;
    c.setImg(img);
    return c;
}
```

 Consola de depuración de Microsoft Visual Studio

```
Introdueix part real: 3
Introdueix part imaginaria: 4
Introdueix part real: 1
Introdueix part imaginaria: 2
c1: 3 + 4i
c2: 1 + 2i
c1: 3 + 4i
c2: 3 + 4i
```

```
int main()
{
    Complex c1 = llegeixComplex();
    Complex c2 = llegeixComplex();

    cout << "c1: " << c1 << endl;
    cout << "c2: " << c2 << endl;

    Complex tmp;
    Complex *p_c1 = &c1;
    Complex *p_c2 = &c2;
    Complex *p_tmp = &tmp;

    *p_tmp = *p_c1;
    p_c1 = p_c2;
    *p_c2 = *p_tmp;

    cout << "c1: " << c1 << endl;
    cout << "c2: " << c2 << endl;
}
```

Exercici: solució

```
Complex llegeixComplex()
{
    Complex c;
    float real, img;

    cout << "Introdueix part real: ";
    cin >> real;
    c.setReal(real);
    cout << "Introdueix part imaginaria: ";
    cin >> img;
    c.setImg(img);
    return c;
}
```

- Hem de distingir correctament entre el valor de la variable apuntador (l'adreça de memòria) i el valor de la variable referenciada per l'apuntador (el valor guardat a l'adreça de memòria)
 - Hem d'utilitzar el que calgui convenientment en cada moment

```
int main()
{
    Complex c1 = llegeixComplex();
    Complex c2 = llegeixComplex();

    cout << "c1: " << c1 << endl;
    cout << "c2: " << c2 << endl;

    Complex tmp;
    Complex *p_c1 = &c1;
    Complex *p_c2 = &c2;
    Complex *p_tmp = &tmp;

    *p_tmp = *p_c1;
    *p_c1 = *p_c2;
    *p_c2 = *p_tmp;

    cout << "c1: " << c1 << endl;
    cout << "c2: " << c2 << endl;
}
```