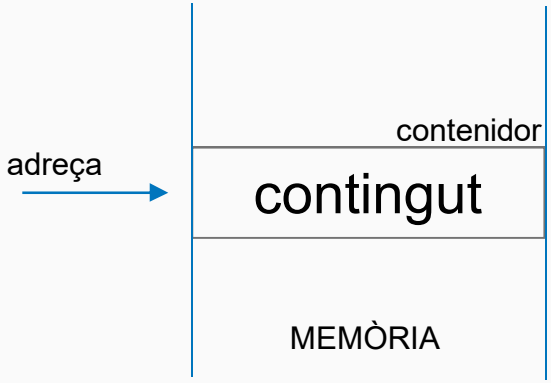
1. Punters i gestió de memòria

UPC - Videogame Design & Development - Programming II

Punters

Tota variable té:
1) El valor del seu contingut i 2) L'adreça que indica on es troba el contingut

Declaració de variables

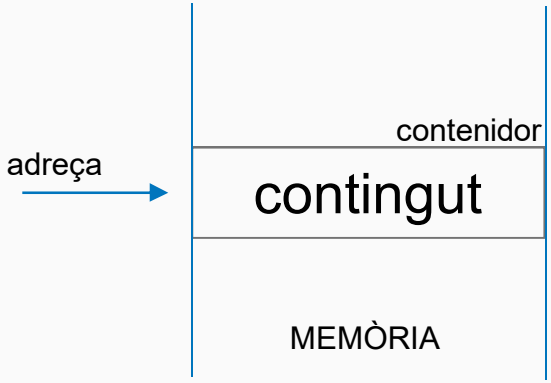


	Declaració de la variable
contenidor	`tipus nom_variable`
adreça	`tipus *nom_variable`

Punters

Tota variable té:
1) El valor del seu contingut i 2) L'adreça que indica on es troba el contingut

Ús de la variable dins el codi



Declarada com	Accés al contingut	Accés a l'adreça
`tipus nom_variable`	`nom_variable`	`&nom_variable`
`tipus *nom_variable`	`*nom_variable`	`nom_variable`

Punters

Un punter (o apuntador) és una variable que conté una adreça de memòria

```
void main() {
  int bar = 3;
  int *foo = &bar;


  printf("%d", bar);
  printf("%d", *foo);
  }
```

Els operands & i * ens permeten interactuar directament amb la memòria de l'ordinador.

& retorna l'adreça de memòria d'una variable

* retorna el contingut d'una variable tipus apuntador

Punters

Un punter (o apuntador) és una variable que conté una adreça de memòria

```
void main() {
  int bar = 3;
  int *foo;

  foo = &bar;
  printf("%d", bar);
  printf("%d", *foo);
  }
```

Els operands & i * ens permeten interactuar directament amb la memòria de l'ordinador.

& retorna l'adreça de memòria d'una variable

* retorna el contingut d'una variable tipus apuntador

Punters

Un punter (o apuntador) és una variable que conté una adreça de memòria

```
void main() {
  int bar = 3;
  int *foo;

  foo = &bar;
  printf("%d", bar);
  printf("%d", *foo);
  }
```

Un punter té un tipus especificat en la seva declaració

El punter interpretarà la dada emmagatzemada a la seva adreça d'acord al seu tipus

Amb el tipus de dada es defineix l'espai de memòria assignat a l'adreça indicada. El punter apunta al primer byte d'aquest bloc de memòria

Inicialització dels punters

Un punter (o apuntador) és una variable que conté una adreça de memòria

```
void main() {
  int bar = 3;
  int *foo = &bar;
  int *baz = nullptr;

  baz = foo;
  printf("%d", *baz);
  }
```

Inicialitzar un punter és recomanable, però no imprescindible

Si un punter no s'inicialitza, pot apuntar a qualsevol lloc de la memòria.

La referència a una adreça nul·la és "*nullptr*". Molts compiladors també accepten NULL

Arrays *(des de la perspectiva dels punters)*

```
void main() {
  int nums[4] = {0,1,2,3};
  int *p = nums;

  for(int i=0; i<4; ++i)
    printf("%d",p[i]);
  }
```

El nom de l'array retorna l'adreça de memòria del primer element

L'operador [ ] funciona tant per arrays com per punters.

a[0] és equivalent a *a

la diferència de declarar un array o un punter rau en la reserva de la memòria associada (ho veurem més endavant)

Aritmètica dels punters

La unitat bàsica dels punters està associada al tipus de dada del punter

```
void main() {
  int nums[4] = {0,1,2,3};
  int *p = nums;
  int bar = *(p++);

  bar = *(p+2); // bar=p[2]
  bar = p[3];
  p[4] = 5;
  }
```

Un punter *+ n* retorna l'adreça de memòria que apuntava inicialment *+ n * sizeof(data type)*

p. ex. Sigui v un array :
v[n] és equivalent a *(v+n)

Els operadors ++ i - - aplicats a un punter retornen les adreces de memòria posterior i anterior respectivament

Els punters com a paràmetres/arguments

```
void increment(int* num) {
   (*num)++;
   }

void main() {
   int n = 3;

   increment(&n);

   }
```

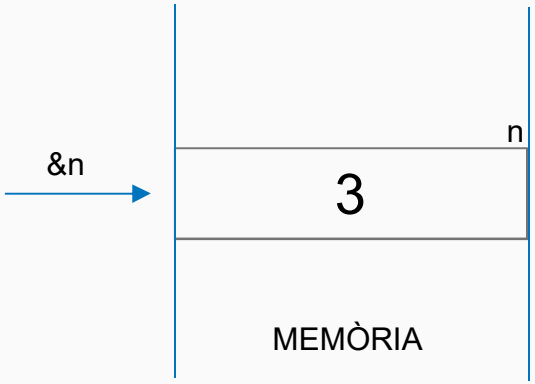
Els punters poden ser paràmetres de les funcions

El valor passat com argument és l'adreça de memòria del punter

Canviant el contingut de l'adreça de memòria, en sortir de la funció el valor modificat de la memòria es manté modificat

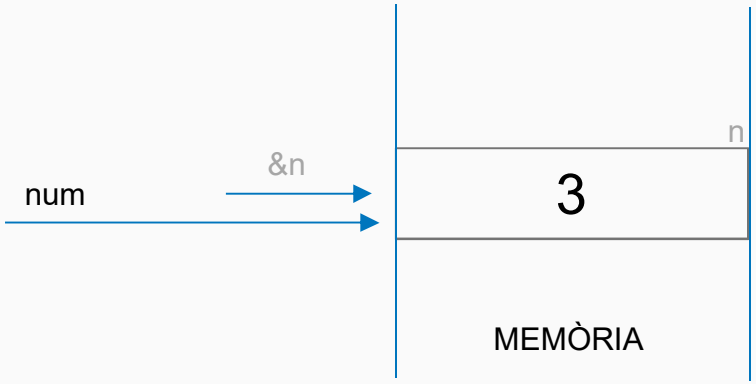
Els punters com a paràmetres/arguments

```
void increment(int* num) {
   (*num)++;
   }

void main() {
   int n = 3;
➔
   increment(&n);

   }
```



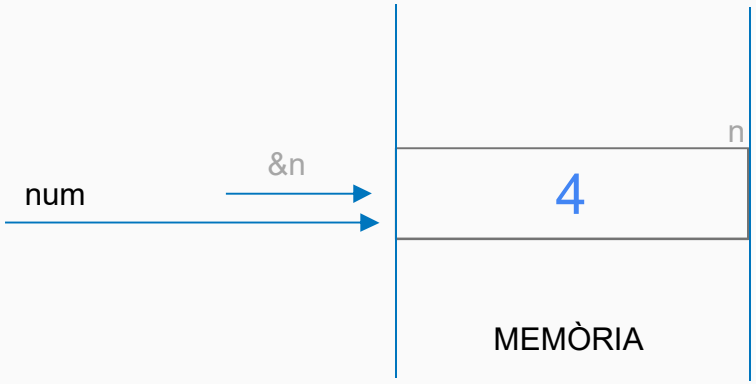
Els punters com a paràmetres/arguments

```
➔void increment(int* num) {
   (*num)++;
   }

void main() {
   int n = 3;

   increment(&n);

   }
```



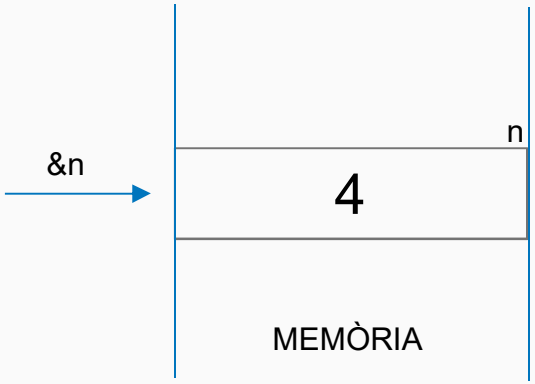
Els punters com a paràmetres/arguments

```
void increment(int* num) {
➔   (*num)++;
  }

void main() {
  int n = 3;

  increment(&n);

  }
```



Els punters com a paràmetres/arguments

```
 void increment(int* num) {
   (*num)++;
   }

void main() {
   int n = 3;

   increment(&n);
➔
   }
```



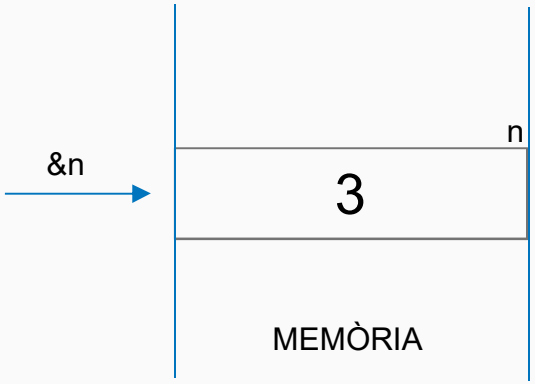
Els punters com a paràmetres/arguments

```
 void increment(int num) {
  num++;
  }

void main() {
  int n = 3;

  increment(n);

  }
```

Revisem l'efecte produït en passar el paràmetre per valor en comptes de per referència

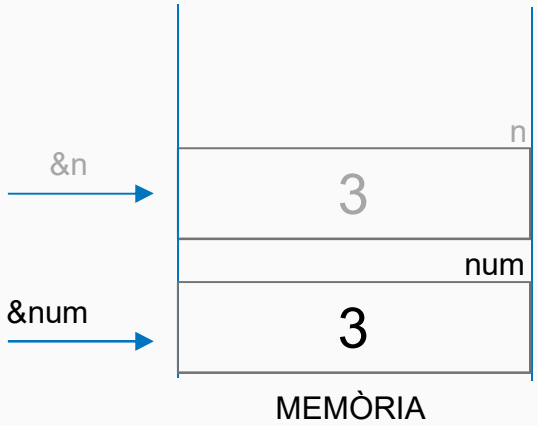
Els punters com a paràmetres/arguments

```
 void increment(int num) {
  num++;
  }

void main() {
  int n = 3;
➔
  increment(n);

  }
```



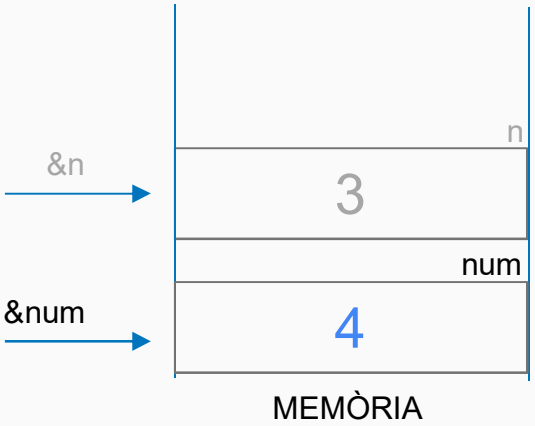
Els punters com a paràmetres/arguments

```
➔void increment(int num) {
  num++;
  }

void main() {
  int n = 3;

  increment(n);

  }
```



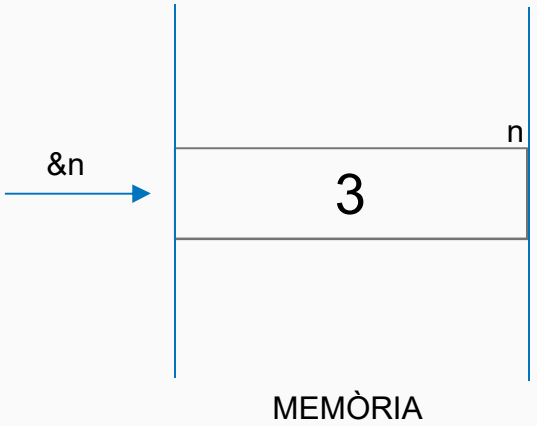
Els punters com a paràmetres/arguments

```
void increment(int num) {
➔   num++;
  }

void main() {
  int n = 3;

  increment(n);

  }
```



Els punters com a paràmetres/arguments

```
void increment(int num) {
num++;
  }

void main() {
  int n = 3;

  increment(n);
➔
  }
```



Punters a caràcters *(char)*

```
void main() {
  char* p = “Hello”;

  for(int i=0; p[i]!=‘\0’; i++)
    printf(“%c”, p[i]);
  }
```

Un punter a char es pot inicialitzar directament mitjançant un string literal (text entre dobles cometes)

De manera automàtica s’afegeix el símbol de final de string ‘\0’

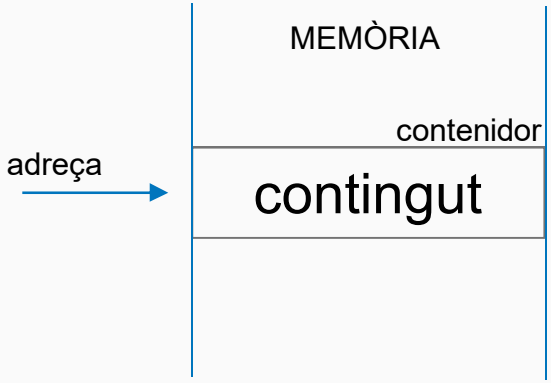
El nom del punter referència l’adreça de memòria del primer caràcter de l’string

a[n] és equivalent a *(a + n). *Per claredat del codi, procureu fer ús de la notació a[n]*

Punters a estructures (Structs)

Tota estructura (igual que qualsevol variable) té:
1) El valor del seu contingut i 2) L'adreça que indica on es troba el contingut

Declaració de variables/structs

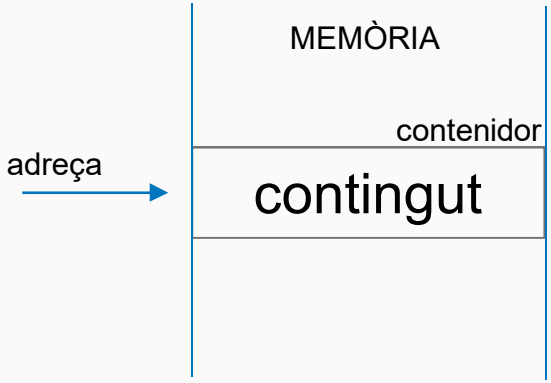


	Declaració de la variable de tipus estructura
contenidor	`struct NomStruct nom_var`
adreça	`struct NomStruct *nom_var`

Punters a estructures (Structs)

Tota estructura (igual que qualsevol variable) té:
1) El valor del seu contingut i 2) L'adreça que indica on es troba el contingut

Ús de la variable dins el codi

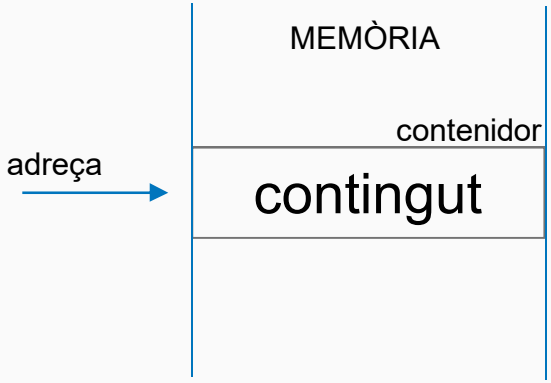


Declarada com	Accés al contingut	Accés a l'adreça
`struct NomStruct nom_var`	`nom_var`	`&nom_var`
`struct NomStruct *nom_var`	`*nom_var`	`nom_var`

Punters a estructures (Structs)

Tota estructura (igual que qualsevol variable) té:
1) El valor del seu contingut i 2) L'adreça que indica on es troba el contingut

Ús de la variable dins el codi



Declarada com	Accés al contingut d'un membre
`struct NomStruct nom_var`	`nom_var.m1`
`struct NomStruct *nom_var`	`nom_var->m1` *o bé* `(*nom_var).x`

Punters *de tipus “void”*

```
void main() {
  void* p = nullptr;
  int a = 3;

  p = &a;
  printf(“%d\n”,*((int*)p));

  float f = 3.0f;
  p = &f;
  printf(“%f\n”,*((float*)p));
  }
```

Un punter *void* pot apuntar a variables de qualsevol tipus de dades.

Quan s’inicialitza es contextualitza segons el tipus de dades d’on es fa apuntar

Punters void no es poden utilitzar directament sense inicialitzar

Compte! Fan el codi més difícil d’entendre

1. Punters i gestió de memòria

Recorregut d’arrays (exemple de punters)

UPC - Videogame Design & Development - Programming II

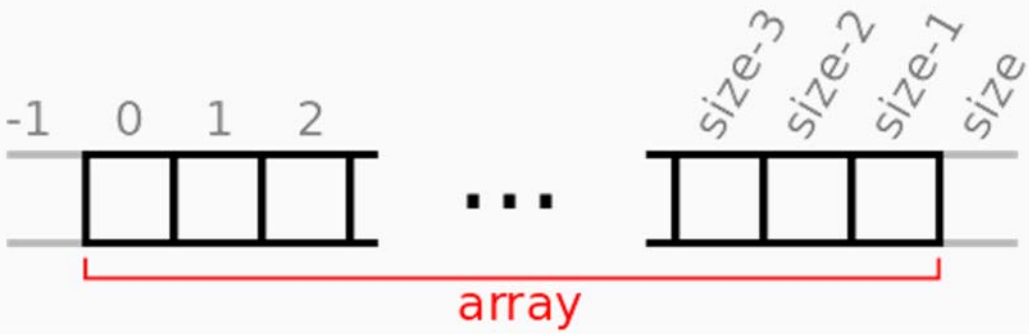
Recorregut d'arrays (1) – Utilitzant l'operador [ ]

D'esquerra a dreta

```
void traverse(int *array, int size) {
  for (int i = 0; i < size; ++i)   {
    printf("%d\n", array[i]);
  }
}
```

De dreta a esquerra

```
void traverse(int *array, int size) {
  for (int i = size-1; i >= 0; --i)   {
    printf("%d\n", array[i]);
  }
}
```



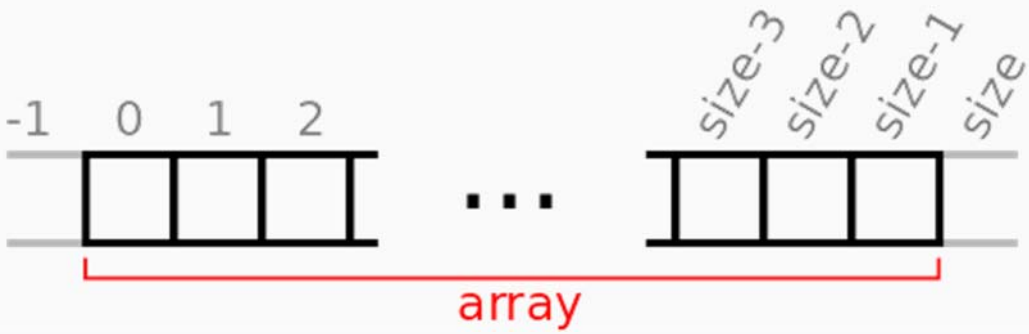
Recorregut d'arrays (2) – Utilitzant *(ptr + index)

D'esquerra a dreta

```
void traverse(int *array, int size) {
  for (int i = 0; i < size; ++i)   {
    printf("%d\n", *(array + i));
  }
}
```

De dreta a esquerra

```
void traverse(int *array, int size) {
  for (int i = size-1; i >= 0; --i)   {
    printf("%d\n", *(array + i));
  }
}
```



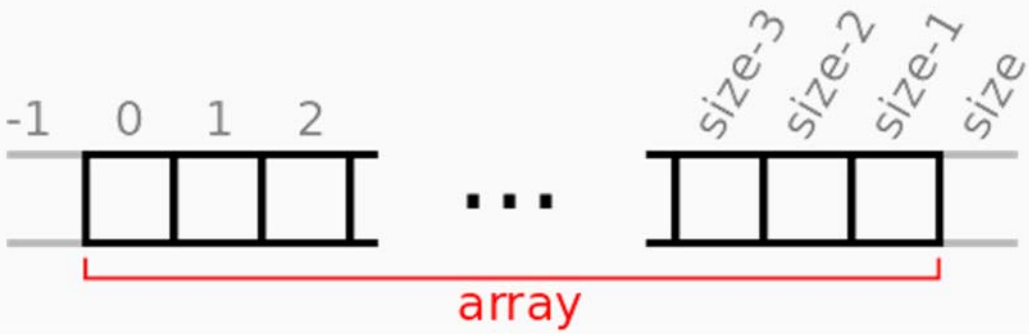
Recorregut d'arrays (3) – Incrementant punters

D'esquerra a dreta

```
void traverse(int *array, int size) {
  int *begin = array;
  int *end = array + size;
  for (int *ptr = begin; ptr != end; ++ptr) {
    printf("%d\n", *ptr);
  }
}
```

De dreta a esquerra

```
void traverse(int *array, int size) {
  int *begin = array + size;
  int *end = array;
  for (int *ptr = begin; ptr != end; --ptr) {
    printf("%d\n", *ptr);
  }
}
```



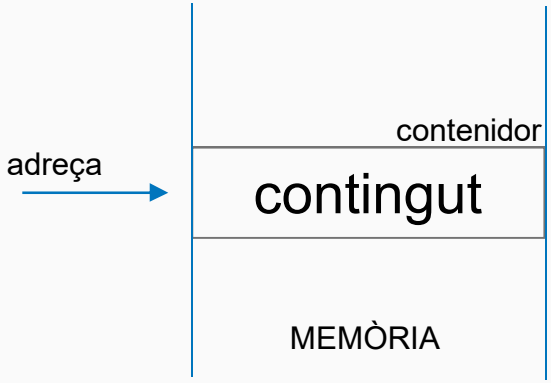
1. Punters i gestió de memòria
1.2 Constants i referències (àlies)

UPC - Videogame Design & Development - Programming II

1. Punters i gestió de memòria
1.2 Constants i referències (àlies)

Recordatori (punters): Tota variable té:
1) El valor del seu contingut i 2) L'adreça que indica on es troba el contingut

Ús de la variable dins el codi

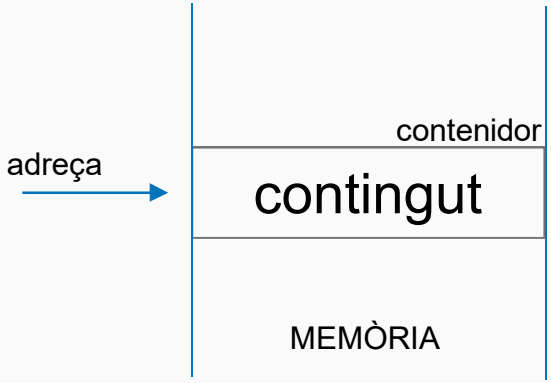


Declarada com	Accés al contingut	Accés a l'adreça
`tipus nom_variable`	`nom_variable`	`&nom_variable`
`tipus *nom_variable`	`*nom_variable`	`nom_variable`

1. Punters i gestió de memòria
1.2 Constants i referències (àlies)

Recordatori (punters): Tota variable té:
1) El valor del seu contingut i 2) L'adreça que indica on es troba el contingut

Ús de la variable dins el codi



Declarada com	Accés al contingut	Accés a l'adreça
`int x`	`x`	`&x`
`int *x`	`*x`	`x`

1. Punters i gestió de memòria
1.2 Constants i referències (àlies)

C++ permet declarar constants utilitzant dos mètodes diferents:
i) **#define** i ii) **const**

#define

El nom de la constant es substitueix pel seu valor al moment de compilar el codi. Es tracta d'una substitució del text en que s'identifica la constant i el codi compilat no manté cap referència del nom substituït (només es manté el valor)

const

Es declara una variable bloquejant el seu contingut i fent que no es pugui modificar durant l'execució del codi.

La inicialització de la variable cal fer-se al moment de declarar-la, no després.

1. Punters i gestió de memòria
1.2 Constants i referències (àlies)

C++ permet declarar constants utilitzant dos mètodes diferents:
i) **#define** i ii) **const**

```cpp
#include <iostream>
using namespace std;

#define PI 3.14159
#define NEWLINE '\n'

int main ()
{
  double r=5.0;                   // radius
  double circle;

  circle = 2 * PI * r;
  cout << circle;
  cout << NEWLINE;

}
```

```cpp
#include <iostream>
using namespace std;

const double pi = 3.14159;
const char newline = '\n';

int main ()
{
  double r=5.0;                   // radius
  double circle;

  circle = 2 * pi * r;
  cout << circle;
  cout << newline;
}
```

1. Punters i gestió de memòria
1.2 Constants i referències (àlies)

La paraula clau *cost* utilitzada en la declaració d'una variable determina que aquesta és constant i el seu valor no es pot modificar després de ser declarada

```
void main() {
  const int b = 4;
  b = 3; // Error!!
}
```

Un cop una variable es declara i s'inicialitza fent ús de la paraula clau "*const*", queda bloquejada i qualsevol intent de modificar el seu contingut generarà un error.

1. Punters i gestió de memòria
1.2 Constants i referències (àlies)

const també pot aplicar-se a apuntadors i de manera combinada als elements que apunten els apuntadors

Declaració		Significat
`tipus * nom_var`		Apuntador (variable) a un element “tipus” (variable)
`tipus const * nom_var`		Apuntador (variable) a un element “tipus” constant
`tipus * const nom_var`		Apuntador constant a un element “tipus” (variable)
`tipus const * const nom_var`		Apuntador constant a un element “tipus” constant

1. Punters i gestió de memòria
1.2 Constants i referències (àlies)

const també pot aplicar-se a apuntadors i de manera combinada als elements que apunten els apuntadors

Declaració	*Declaració equivalent*	Significat
`tipus * nom_var`		Apuntador (variable) a un element "tipus" (variable)
`tipus const * nom_var`	*`const tipus * nom_var`*	Apuntador (variable) a un element "tipus" constant
`tipus * const nom_var`		Apuntador constant a un element "tipus" (variable)
`tipus const * const nom_var`	*`const tipus * const nom_var`*	Apuntador constant a un element "tipus" constant

IMPORTANT: L'ús de *const* per definir constant l'element es pot posar tant a la dreta com a l'esquerra del tipus. Altrament, aplica al que té just a l'esquerra.

1. Punters i gestió de memòria
1.2 Constants i referències (àlies)

const també pot aplicar-se a apuntadors i de manera combinada als elements que apunten els apuntadors

Declaració	*Declaració equivalent*	Significat
`int* nom_var`		Apuntador (variable) a un element "tipus" (variable)
`int const * nom_var`	*`const int * nom_var`*	Apuntador (variable) a un element "tipus" constant
`int * const nom_var`		Apuntador constant a un element "tipus" (variable)
`int const * const nom_var`	*`const int * const nom_var`*	Apuntador constant a un element "tipus" constant

IMPORTANT: L'ús de *const* per definir constant l'element es pot posar tant a la dreta com a l'esquerra del tipus. Altrament, aplica al que té just a l'esquerra.

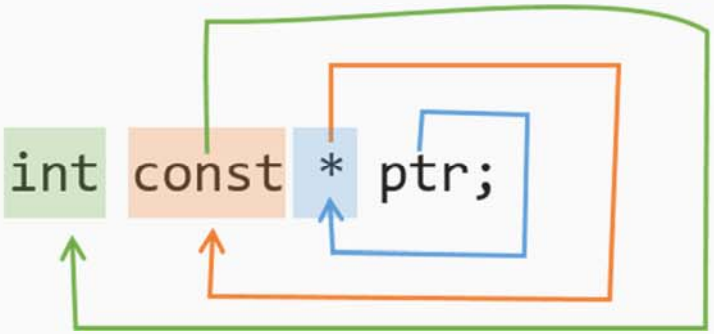
1. Punters i gestió de memòria
1.2 Constants i referències (àlies)

const també pot aplicar-se a apuntadors i de manera combinada als elements que apunten els apuntadors

ptr is a pointer to int

1. Punters i gestió de memòria
1.2 Constants i referències (àlies)

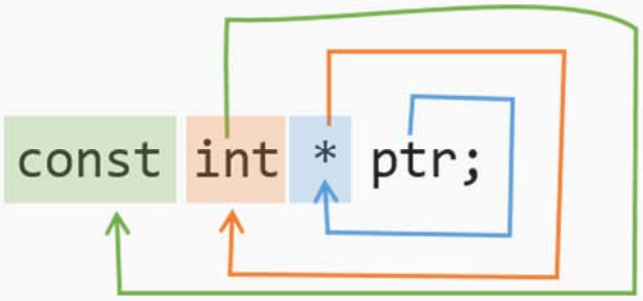
const també pot aplicar-se a apuntadors i de manera combinada als elements que apunten els apuntadors




```
ptr is a pointer to const int
```

1. Punters i gestió de memòria
1.2 Constants i referències (àlies)

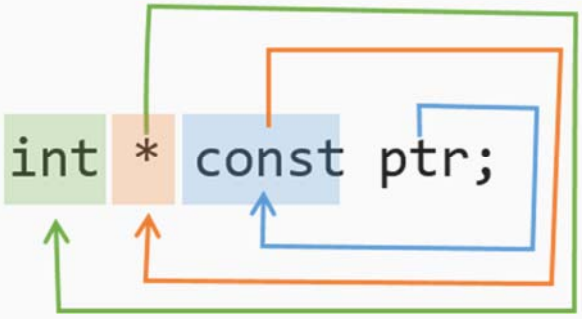
const també pot aplicar-se a apuntadors i de manera combinada als elements que apunten els apuntadors




```
ptr is a pointer to int constant (i.e. const int)
```

1. Punters i gestió de memòria
1.2 Constants i referències (àlies)

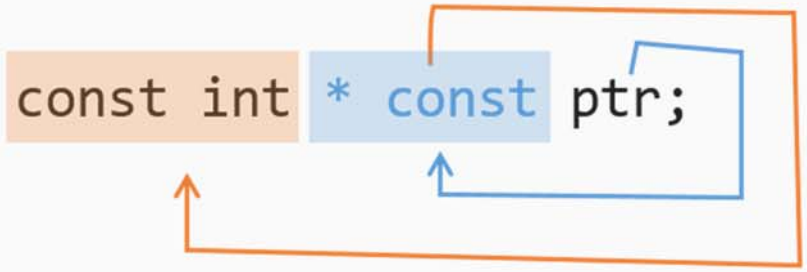
const també pot aplicar-se a apuntadors i de manera combinada als elements que apunten els apuntadors




```
ptr is a const pointer to int
```

1. Punters i gestió de memòria
1.2 Constants i referències (àlies)

const també pot aplicar-se a apuntadors i de manera combinada als elements que apunten els apuntadors




```
ptr is a constant pointer to const int
```

1. Punters i gestió de memòria
1.2 Constants i referències (àlies)

const també pot aplicar-se a apuntadors i de manera combinada als elements que apunten els apuntadors

```
int main() {
  int b = 4;
  const int *p1 = &b;          // apuntador a enter constant
  int const *p2 = &b;          // ídem
  int * const p2 = &b;         // apuntador constant a enter
  const int * const p3 = &b;   // apuntador constant a enter constant
  int const * const p3 = &b;   // ídem
  }
```

1. Punters i gestió de memòria
1.2 Constants i referències (àlies)

Referències “&”
Àlies a un objecte o una variable preexistents.

Les referències eviten duplicar o fer una còpia d’un l’element quan no és necessari.
Les referències no poden correspondre a elements NULL
Les referències NO són punters de memòria (!)
La memòria d’una variable i la de la seva referència és la mateixa. És el mateix objecte identificat amb noms diferents.
Els canvis als valors d’una variable es poden efectuar indistintament a través del nom de la pròpia variable o a través del de les seves referències.

1. Punters i gestió de memòria
1.2 Constants i referències (àlies)

Una referència és un és un àlies a un objecte o una variable i evita duplicar o fer una còpia de l'element quan s'utilitza com a paràmetre d'una funció.

```
Tipus& nom_referencia = nom_variable;

funcio (Tipus& nom_parametre) {}
funcio (nom_variable);
```

```
void increment(int& n) {
  ++n;
  }

void main() {
  int a = 3;
  int& ra = a;
  increment(a);
  }
```

Declarada com	Accés al contingut	Accés a l'adreça
`tipus nom_variable`	`nom_variable`	`&nom_variable`
`tipus *nom_variable`	`*nom_variable`	`nom_variable`
`tipus &nom_alies`	`nom_alies`	`&nom_alies`
`tipus *&nom_alies`	`*nom_alies`	`nom_alies`

1. Punters i gestió de memòria
1.2 Constants i referències (àlies)

Una referència és un és un àlies a un objecte o una variable i evita duplicar o fer una còpia de l'element quan s'utilitza com a paràmetre d'una funció.

```
Tipus& nom_referencia = nom_variable;
funcio (Tipus& nom_parametre) {}
funcio (nom_variable);
```

```
void increment(int& n) {
  ++n;
  }

void main() {
  int a = 3;
  int& ra = a;
  increment(a);
  }
```

1. Punters i gestió de memòria
1.2 Constants i referències (àlies)

Una **referència constant** és una manera eficient de passar una paràmetre a una funció quan aquest només s'ha de llegir i no modificar.

```
void foo(int n)             {printf("%d",  n);}  // Es fa una còpia de la variable que es passa com paràmtre.
void bar(int* n)            {printf("%d", *n);}  // NO es fa cap còpia de la variable a la que apunta el paràmetre

void baz(int& n)            {printf("%d",  n);}  // NO es fa cap còpia de la variable que es passa com a paràmetre
                                                    i n representa únicament un àlias d'aquesta

void supadupa(const int& n) {printf("%d",  n);}  // NO es fa cap còpia de la variable que es passa com a paràmetre
                                                    (n representa un àlias d'aquesta) i no es pot modificar dins la
                                                    funció.
```

1. Punters i gestió de memòria

1.3 Memòria dinàmica

UPC - Videogame Design & Development - Programming II

1. Punters i gestió de memòria
1.3 Memòria dinàmica

La pila (**stack**) és la memòria més propera a la CPU i el seu ús fa el codi més eficient

- Les variables declarades a les funcions s'allotgen a la pila
- L'accés a la pila és molt ràpid i la seva gestió molt eficient
- Té una mida limitada i molt més petita que la memòria principal
- La mida de les variables de la pila és fixa (no pot créixer ni disminuir)

La memòria principal (**heap**) allotja el gruix de variables i codi de l'ordinador.

- L'accés és més lent
- No és tant eficient
- Té molta més capacitat
- L'espai ocupat per les variables és dinàmic. El podem fer créixer o reduir. Cal una gestió acurada de la reserva i alliberament d'espai.

1. Punters i gestió de memòria
1.3 Memòria dinàmica

Càlcul de l'espai de memòria: ***sizeof***

	Exemple de codi	
sizeof (*tipus de variable*)	`sizeof (int);`	Retorna el nombre de bytes del tipus indicat
sizeof(*variable*)	`sizeof (x);`	Retorna el nombre de bytes d'una variable
sizeof(*estructura*)	`sizeof (struct vector);`	Retorna el nombre de bytes d'un struct
sizeof(*tipus_array*)**elems*	`sizeof (int)*n;`	Retorna l'espai de memòria que ocupa l'array

1. Punters i gestió de memòria
1.3 Memòria dinàmica

Càlcul de l'espai de memòria: ***sizeof***

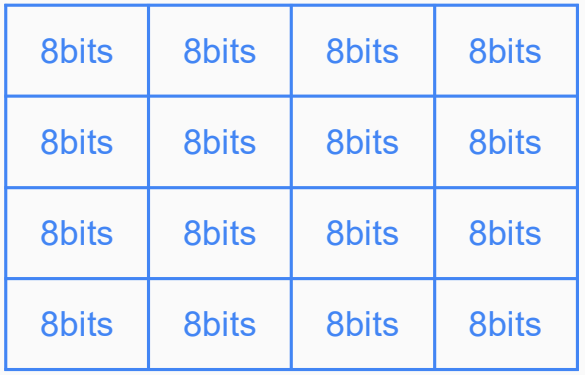
El compilador s'encarrega d'identificar l'espai de memòria que ocupen les variables del codi considerant l'arquitectura de la CPU (32 bits, 64 bits, ...).

... pel cas de les estructures, no és una solució evident.

Quin espai ocupa una variable de l'estructura X?

```
struct x x1;
```

Suposem una CPU de 32 bits (4 bytes)




```
//Exemple de codi

struct X {
 short s; // 2 bytes
 int   i; // 4 bytes
 char  c; // 1 byte
};
```

1. Punters i gestió de memòria
1.3 Memòria dinàmica

Càlcul de l'espai de memòria: ***sizeof***

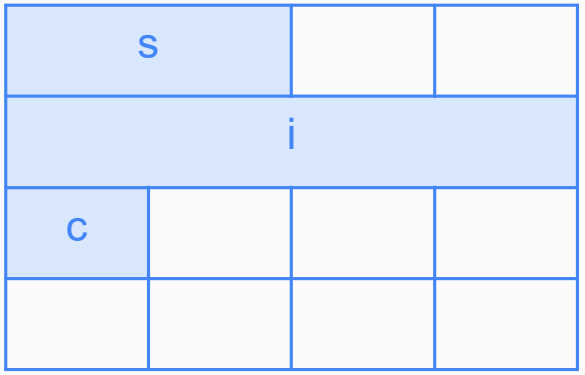
El compilador s'encarrega d'identificar l'espai de memòria que ocupen les variables del codi considerant l'arquitectura de la CPU (32 bits, 64 bits, ...).

... pel cas de les estructures, no és una solució evident.

Quin espai ocupa una variable de l'estructura X?

`struct x x1;`

Suposem una CPU de 32 bits (4 bytes)




```
//Exemple de codi

struct X {
 short s; // 2 bytes
 int   i; // 4 bytes
 char  c; // 1 byte
};
```

1. Punters i gestió de memòria
1.3 Memòria dinàmica

Càlcul de l'espai de memòria: ***sizeof***

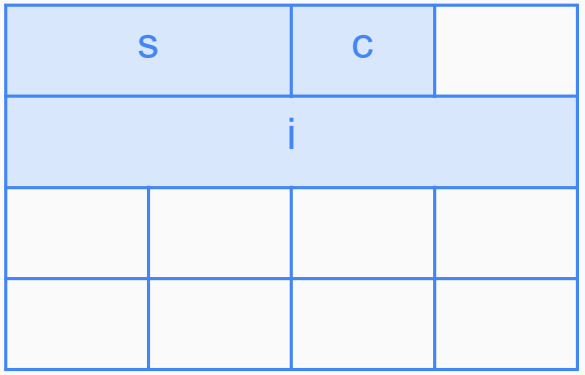
El compilador s'encarrega d'identificar l'espai de memòria que ocupen les variables del codi considerant l'arquitectura de la CPU (32 bits, 64 bits, ...).

... pel cas de les estructures, no és una solució evident.

Quin espai ocupa una variable de l'estructura X?

`struct x x1;`

Suposem una CPU de 32 bits (4 bytes)




```
//Exemple de codi

struct X {
 short s; // 2 bytes
 int   i; // 4 bytes
 char  c; // 1 byte
};
```

1. Punters i gestió de memòria
1.3 Memòria dinàmica

Càlcul de l'espai de memòria: ***sizeof***

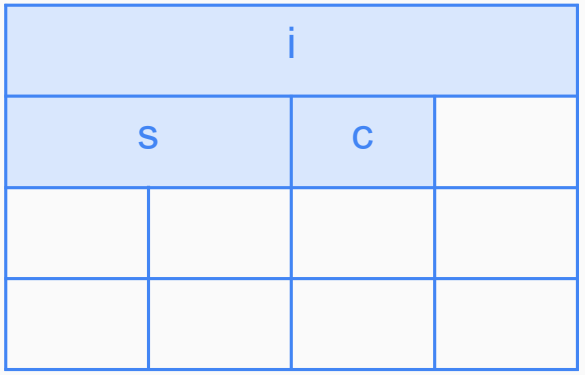
El compilador s'encarrega d'identificar l'espai de memòria que ocupen les variables del codi considerant l'arquitectura de la CPU (32 bits, 64 bits, ...).

... pel cas de les estructures, no és una solució evident.

Quin espai ocupa una variable de l'estructura X?

`struct x x1;`

Suposem una CPU de 32 bits (4 bytes)




```
//Exemple de codi

struct X {
 short s; // 2 bytes
 int   i; // 4 bytes
 char  c; // 1 byte
};
```

1. Punters i gestió de memòria
1.3 Memòria dinàmica

Càlcul de l'espai de memòria: ***sizeof***

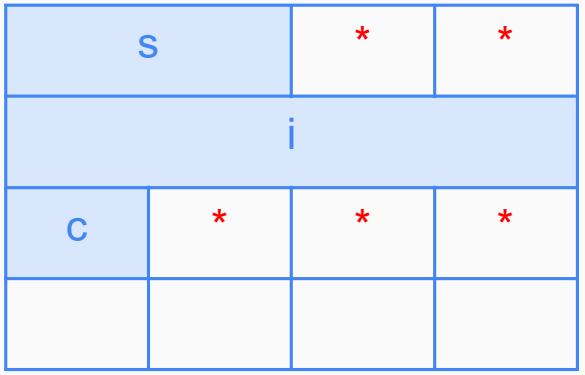
El compilador s'encarrega d'identificar l'espai de memòria que ocupen les variables del codi considerant l'arquitectura de la CPU (32 bits, 64 bits, ...).

... pel cas de les estructures, no és una solució evident.

Quin espai ocupa una variable de l'estructura X?

`struct x x1;`

Suposem una CPU de 32 bits (4 bytes)




```
//Exemple de codi

struct X {
 short s; // 2 bytes
           // +2 bytes
 int   i; // 4 bytes
 char  c; // 1 byte
           // +3 bytes
};

* Bytes de padding
```

1. Punters i gestió de memòria
1.3 Memòria dinàmica

Funcions per reservar i alliberar memòria principal (heap) de manera dinàmica amb el llenguatge C i amb C++

Llenguatge C	
malloc	Reserva un bloc de memòria de mida determinada
calloc	Reserva n blocs consecutius de memòria i els inicialitza a zero
free	Allibera la memòria reservada

Llenguatge C++	
new	Reserva la memòria necessària per allotjar el contingut d'un objecte (o variable). *És l'equivalent al malloc/calloc*
delete	Allibera la memòria d'un objecte, o el que és el mateix: destrueix la instància d'un objecte. *És l'equivalent al free de C*

malloc, calloc i new retornen punters void

1. Punters i gestió de memòria
1.3 Memòria dinàmica

Funcions per reservar i alliberar memòria principal (heap) de manera dinàmica amb el llenguatge C i amb C++

Llenguatge C

```
void main() {
 int *x;


 x = malloc(sizeof(int));
 *x = 1;

 free(x);
 }
```

Llenguatge C++

```
void main() {
 int *x;


 x = new int;
 *x = 1;

 delete x;
 }
```

1. Punters i gestió de memòria
1.3 Memòria dinàmica

Funcions per reservar i alliberar memòria principal (heap) de manera dinàmica amb el llenguatge C i amb C++

Llenguatge C

```
void main() {
 int *x;
 int y;

 x = malloc(sizeof(int));
 y = 1;
 *x= y;
 free(x);
 }
```

Llenguatge C++

```
void main() {
 int *x;
 int y;

 x = new int;
 y = 1;
 *x= y;
 delete x;
 }
```

1. Punters i gestió de memòria
1.3 Memòria dinàmica

Funcions per reservar i alliberar memòria principal (heap) de manera dinàmica amb el llenguatge C i amb C++

Llenguatge C

```
void main() {
 int *nums;
 int *more_nums;

 nums = (int*)malloc(4*sizeof(int));
 more_nums= = (int*)calloc(4,sizeof(int));

 for(int i=0; i<4; ++i) nums[i] = 0;
 free(nums);
 free(more_nums);
 nums = more_nums = nullptr;
 }
```

Llenguatge C++

```
void main() {
 int *nums;
 int *more_nums;

 nums=new int[4];
 more_nums=new int[4]();

 for(int i=0; i<4; ++i) nums[i] = 0;
 delete[] nums;
 delete[] more_nums;
 nums = more_nums = nullptr;
 }
```

1. Punters i gestió de memòria
1.3 Memòria dinàmica en C++

Reservar memòria en programació orientada a objectes vol dir crear una nova instància d'un objecte

Reserva de memòria

```
punter = new tipus
punter = new tipus [nombre_elements]
```

Alliberament de memòria

```
delete punter;
delete[] punter;
```

```
// Exemple de codi

int *n;
int *foo;

n=new int;
n=3;
foo = new int [n];
...

delete n;
delete[] foo;
```

1. Punters i gestió de memòria
1.3 Memòria dinàmica en C++

Reservar memòria en programació orientada a objectes vol dir crear una nova instància d'un objecte

Reserva de memòria

```
punter = new tipus
punter = new tipus [nombre_elements]
```

(notrow) paràmetre de new fa que en cas de no haver-hi suficient memòria a l'ordinador, en comptes de generar un error d'execució es retorna un punter a null (nullptr)

```
// Exemple de codi

double *n;
int *foo;

n=new int;
n=1000000000000000;
foo = new (notrow) int [n];
if (foo == nullptr) {
  // ERROR d'assignació de memòria
}
```

1. Punters i gestió de memòria
1.3 Memòria dinàmica en C++

Exercici

La funció *inverteix* ens canvia l'ordre dels elements d'un array d'enters. Volem utilitzar-la per invertir l'ordre d'una seqüència de nombres a introduir per l'usuari, la quantitat dels quals està indeterminada dins el codi. A l'inici del programa l'usuari indicarà la quantitat de nombres que introduirà.

Escriu el codi del programa principal utilitzant aquestes variables:

```
int *arr; // Array d'enters de longitud indeterminada
int size; // Mida que tindrà l'array
```

Cal que utilitzis les funcions de reserva i alliberament de memòria dinàmica per *arr*

```
1   #include <iostream>
2   using namespace std;
3
4   void inverteix(int *arr, int size) {
5       int aux; // variable intermedia per l'intercanvi de valors
6       int cont=size-1; // apuntador per l'intercanvi
7
8       for (int i = 0; i <size/2; i++) {
9           aux = arr[i];
10          arr[i] = arr[cont];
11          arr[cont--] = aux;
12          }
13      }
14
```

1. Punters i gestió de memòria
1.3 Memòria dinàmica en C++

Exercici

En parelles, descriviu un problema relacionat amb un videojoc i que requereixi l'ús de la gestió dinàmica de la memòria. Penseu en una possible solució, programeu-la i envieu el codi a la carpeta d'entregues. Indiqueu a la capçalera del codi l'enunciat i els membres de l'equip.

... voluntaris per a presentar l'exercici a la següent classe

1. Punters i gestió de memòria

Recorregut d’arrays

UPC - Videogame Design & Development - Programming II

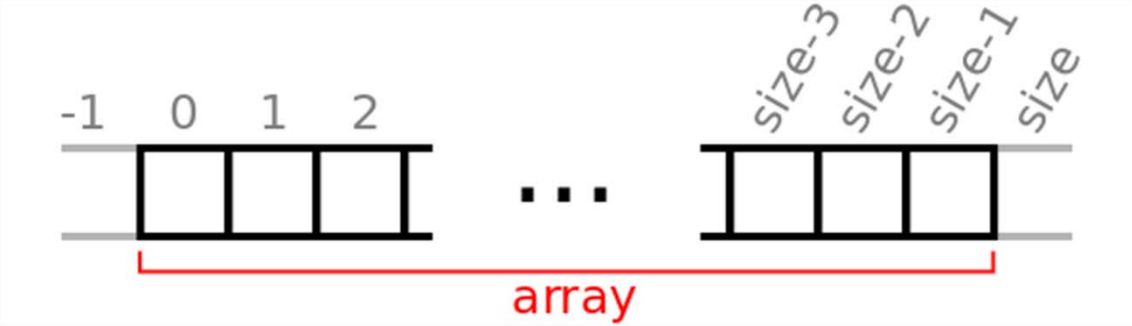
Recorregut d'arrays (1) – Utilitzant l'operador [ ]

D'esquerra a dreta

```
void traverse(int *array, int size) {
  for (int i = 0; i < size; ++i)   {
    printf("%d\n", array[i]);
  }
}
```

De dreta a esquerra

```
void traverse(int *array, int size) {
  for (int i = size-1; i >= 0; --i)   {
    printf("%d\n", array[i]);
  }
}
```



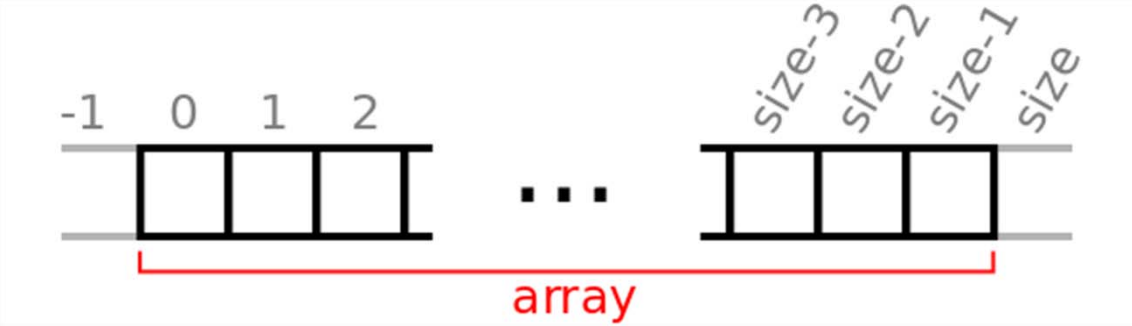
Recorregut d'arrays (2) – Utilitzant *(ptr + index)

D'esquerra a dreta

```
void traverse(int *array, int size) {
  for (int i = 0; i < size; ++i)   {
    printf("%d\n", *(array + i));
  }
}
```

De dreta a esquerra

```
void traverse(int *array, int size) {
  for (int i = size-1; i >= 0; --i)   {
    printf("%d\n", *(array + i));
  }
}
```



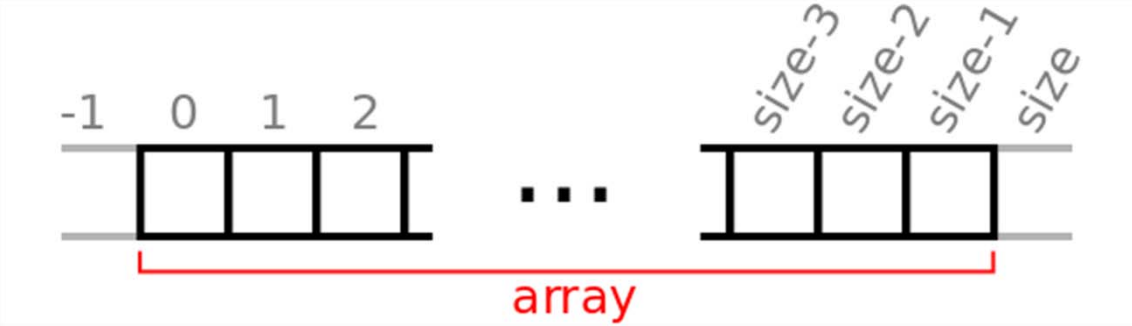
Recorregut d'arrays (3) – Incrementant punters

D'esquerra a dreta

```
void traverse(int *array, int size) {
  int *begin = array;
  int *end = array + size;
  for (int *ptr = begin; ptr != end; ++ptr) {
    printf("%d\n", *ptr);
  }
}
```

De dreta a esquerra

```
void traverse(int *array, int size) {
  int *begin = array + size;
  int *end = array;
  for (int *ptr = begin; ptr != end; --ptr) {
    printf("%d\n", *ptr);
  }
}
```



2. Programació Orientada a Objectes

UPC – Disseny i Desenvolupament de Videojocs – Programació II

2. Programació Orientada a Objectes

Característiques principals dels llenguatges Orientats a Objectes

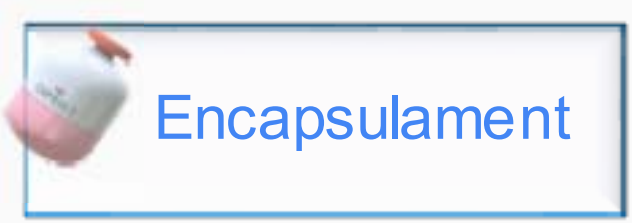
Encapsulament

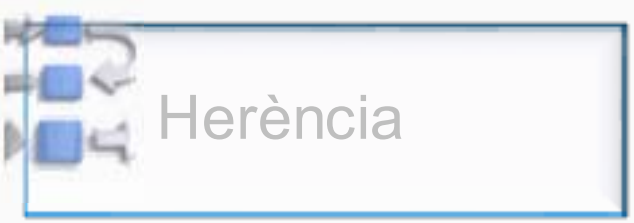
Herència

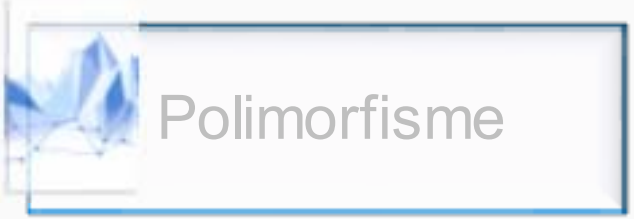
Polimorfisme

2. Programació Orientada a Objectes

Característiques principals dels llenguatges Orientats a Objectes



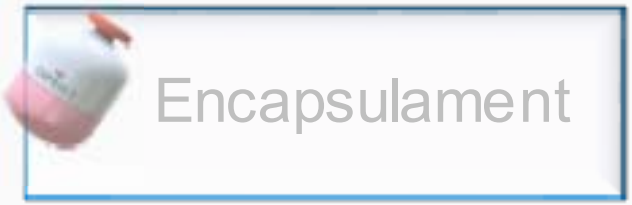


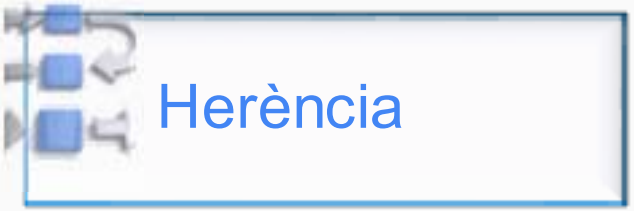


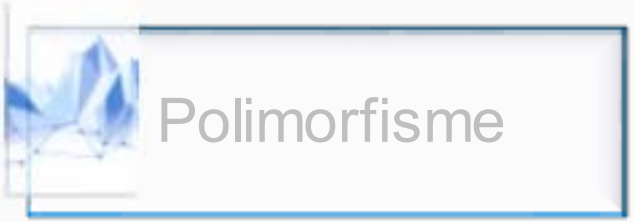
L'encapsulament es refereix a l'agrupació de dades i codi dins les classes, fent els seus elements visibles únicament dins l'àmbit en el que correspon.

2. Programació Orientada a Objectes

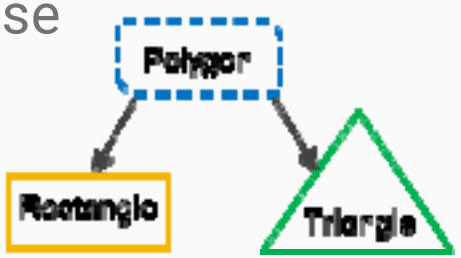
Característiques principals dels llenguatges Orientats a Objectes





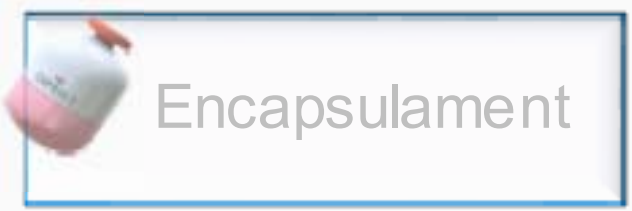


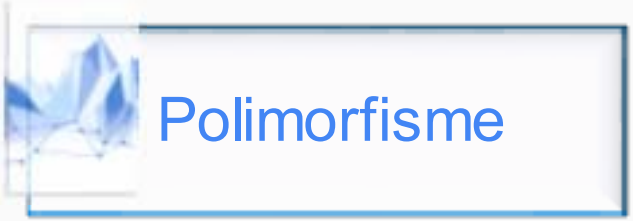
L'herència és el mecanisme a través del que una classe hereta propietats d'una altra



2. Programació Orientada a Objectes

Característiques principals dels llenguatges Orientats a Objectes

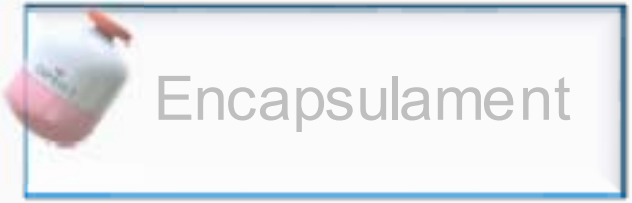




El polimorfisme és la manera en què els llenguatges Orientats a Objectes implementen el concepte de polisèmia del món real: Un únic nom per a molts significats, segons el context

2. Programació Orientada a Objectes

Característiques principals dels llenguatges Orientats a Objectes


Encapsulament


Herència


Polimorfisme

Sobrecàrrega d'operadors

Un mateix operador té comportaments diferents segons el tipus dels operands

Constructores de les classes

Diferents membres funció amb el mateix nom, activats segons els paràmetres rebuts

2. Programació Orientada a Objectes
2.1 Encapsulament

L'encapsulament és el mecanisme d'abstracció bàsic dels llenguatges Orientats a Objectes per a definir unitats de codi i dades.

Els objectes de C++ (així com de qualsevol altre llenguatge Orientat a Objectes) es poden veure com un enriquiment de les estructures *(structs)*, en que a més de definir conjunts de dades més o menys complexos, inclouen codi.

Cada objecte és responsable d'executar les tasques que li corresponen sense utilitzar codi ni dades alienes a ell.

2. Programació Orientada a Objectes
2.1 Encapsulament

L'encapsulament és el mecanisme d'abstracció bàsic dels llenguatges Orientats a Objectes per a definir unitats de codi i dades.

```
class Point {
private:
int x,y;

public:
int getx(){ return x; }
int gety(){ return y; }
}

...

Point punt1,punt2;
```

- Els objectes en C++ es defineixen amb el que s'anomena "**classe**" *(class)*.
- Un **objecte** és una instància d'una classe.
- Les classes defineixen els **membres de dades** (variables) i els **membres funció** (codi).
- Tant els uns com els altres tenen diferents nivells de visibilitat: i) "***public***", ii) "***protected***" i iii) "***private***".

2. Programació Orientada a Objectes
2.1 Encapsulament

L'encapsulament és el mecanisme d'abstracció bàsic dels llenguatges Orientats a Objectes per a definir unitats de codi i dades.

```
class Box {
public:
int llarg, ample, fons;

public:
int volum() {
  return llarg * ample * fons;
  }
};
...
Box capsa1, capsa2;
```

- Els objectes en C++ es defineixen amb el que s'anomena "**classe**" *(class)*.
- Un **objecte** és una instància d'una classe.
- Les classes defineixen els **membres de dades** (variables) i els **membres funció** (codi).
- Tant els uns com els altres tenen diferents nivells de visibilitat: i) "***public***", ii) "***protected***" i iii) "***private***".

2. Programació Orientada a Objectes
2.1 Encapsulament

L'encapsulament és el mecanisme d'abstracció bàsic dels llenguatges Orientats a Objectes per a definir unitats de codi i dades.

```
class Box {
public: // Membres de dades
int llarg, ample, fons;

public:
int volum() {
  return llarg * ample * fons;
  }
};
```

- Els objectes en C++ es defineixen amb el que s'anomena "**classe**" *(class)*.
- Un **objecte** és una instància d'una classe.
- Les classes defineixen els **membres de dades** (variables) i els **membres funció** (codi).
- Tant els uns com els altres tenen diferents nivells de visibilitat: i) "***public***", ii) "***protected***" i iii) "***private***".

2. Programació Orientada a Objectes
2.1 Encapsulament

L'encapsulament és el mecanisme d'abstracció bàsic dels llenguatges Orientats a Objectes per a definir unitats de codi i dades.

```
class Box {
public:
int llarg, ample, fons;

public: // Membres funció
int volum() {
  return height * width * depth;
  }
};
```

- Els objectes en C++ es defineixen amb el que s'anomena "**classe**" *(class)*.
- Un **objecte** és una instància d'una classe.
- Les classes defineixen els **membres de dades** (variables) i els **membres funció** (codi).
- Tant els uns com els altres tenen diferents nivells de visibilitat: i) "***public***", ii) "***protected***" i iii) "***private***".

2. Programació Orientada a Objectes
2.1 Encapsulament

L'encapsulament és el mecanisme d'abstracció bàsic dels llenguatges Orientats a Objectes per a definir unitats de codi i dades.

Accés	*Public*	*Protected*	*private*
Mateixa classe	X	X	X
Classe derivada	X	X	
Fora de la classe	X		

- Els objectes en C++ es defineixen amb el que s'anomena "**classe**" *(class)*.
- Un **objecte** és una instància d'una classe.
- Les classes defineixen els **membres de dades** (variables) i els **membres funció** (codi).
- Tant els uns com els altres tenen diferents nivells de visibilitat: i) "***public***", ii) "***protected***" i iii) "***private***".

2. Programació Orientada a Objectes
2.1 Encapsulament

Un Objecte és una instància d'una classe

```
Box capsa1;      // Declaració de capsa1 com a objecte de tipus Box
Box capsa2;      // Declaració de capsa2 com a objecte de tipus Box

capsa1.ample     // Propietat, atribut o membre de dades 'ample' de l'objecte
                    capsa1 de tipus Box

capsa1.volum()   // Mètode o membre funció 'volum' de l'objecte capsa1 que
                    calcula el seu volum
```

2. Programació Orientada a Objectes
2.1 Encapsulament

El **Constructor** és un mètode especial d'una classe que pren com a nom el mateix nom que el de la pròpia classe i serveix per inicialitzar l'objecte creat

```
class Point {
private:
int x,y;
public:
Point(int coordx, int coordy){
  x = coordx;
  y = coordy;
  }
Point(){x = 0; y = 0;}
    int getx(){return x;}
    int gety(){return y;}
}
```

Els constructors s'executen al moment en que s'instancia (es crea) un objecte de la classe.

Se'n poden declarar més d'un. En aquest cas s'executarà aquell que té definits els paràmetres tal i com s'utilitzen (polimorfisme!).

```
Point p1;
Point p2(2,3);
```

2. Programació Orientada a Objectes
2.1 Encapsulament

El **Constructor** és un mètode especial d'una classe que pren com a nom el mateix nom que el de la pròpia classe i serveix per inicialitzar l'objecte creat.

```
class Point {
private:
int x,y;
public:
Point(int coordx, int coordy){
  x = coordx;
  y = coordy;
  }
Point(){x = 0; y = 0;}
    int getx(){return x;}
    int gety(){return y;}
}
```

Els constructors s'executen al moment en que s'instancia (es crea) un objecte de la classe.

Se'n poden declarar més d'un. En aquest cas s'executarà aquell que té definits els paràmetres tal i com s'utilitzen (polimorfisme!).

```
Point p1;
Point p2(2,3);
```

2. Programació Orientada a Objectes
2.1 Encapsulament

El **Constructor** és un mètode especial d'una classe que pren com a nom el mateix nom que el de la pròpia classe i serveix per inicialitzar l'objecte creat.

```
class Point {
private:
int x,y;
public:
Point(int coordx, int coordy){
  x = coordx;
  y = coordy;
  }
Point(){x = 0; y = 0;}
    int getx(){return x;}
    int gety(){return y;}
}
```

Els constructors s'executen al moment en que s'instancia (es crea) un objecte de la classe.

Se'n poden declarar més d'un. En aquest cas s'executarà aquell que té definits els paràmetres tal i com s'utilitzen (polimorfisme!).

```
Point p1;
Point p2(2,3);
```

2. Programació Orientada a Objectes
2.1 Encapsulament

El **Constructor** és un mètode especial d'una classe que pren com a nom el mateix nom que el de la pròpia classe i serveix per inicialitzar l'objecte creat.

```
class Point {
private:
int x,y;
public:
Point(int coordx, int coordy){
  x = coordx;
  y = coordy;
  }
Point(){x = 0; y = 0;}
    int getx(){return x;}
    int gety(){return y;}
}
```

Els constructors s'executen al moment en que s'instancia (es crea) un objecte de la classe.

Se'n poden declarar més d'un. En aquest cas s'executarà aquell que té definits els paràmetres tal i com s'utilitzen (polimorfisme!).

```
Point p1;
Point p2(2,3);
```

2. Programació Orientada a Objectes
2.1 Encapsulament

El **Destructor** és un mètode especial d'una classe que pren com a nom el mateix nom que el de la pròpia classe precedit ~

```
class Point {
private:
int x,y;
public:
Point(){
  x = 0; y = 0;
  }

~Point(){
  //Destructor
  };
}
```

Els destructors s'executen al moment en que s'elimina la instància d'un objecte.

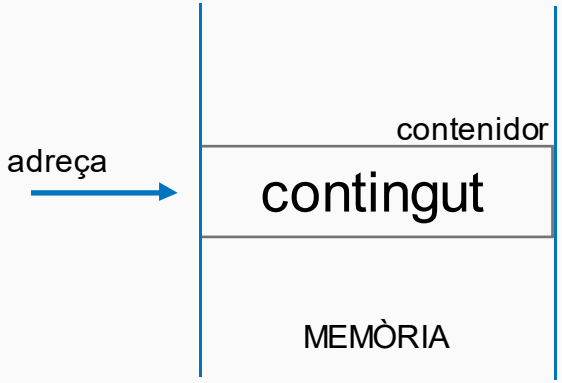
Típicament alliberen la memòria dinàmica que s'hagi pogut reservar degut a la creació de l'objecte.

```
Point p1;
Point p2(2,3);
```

2. Programació Orientada a Objectes
2.1 Encapsulament

Els objectes, igual que les variables poden ser declarats com apuntadors.
Tot objecte té: 1) El valor del seu contingut i 2) La seva adreça de memòria

Declaració

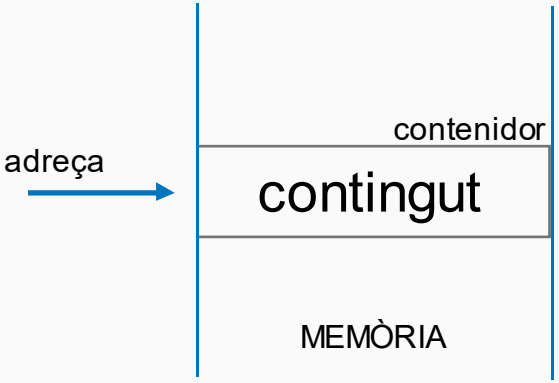


	Declaració de l'objecte
contenidor	`classe nom_objecte`
adreça	`classe *nom_objecte`

2. Programació Orientada a Objectes
2.1 Encapsulament

Els objectes, igual que les variables poden ser declarats com apuntadors.
Tot objecte té: 1) El valor del seu contingut i 2) La seva adreça de memòria

Ús de l'objecte dins el codi

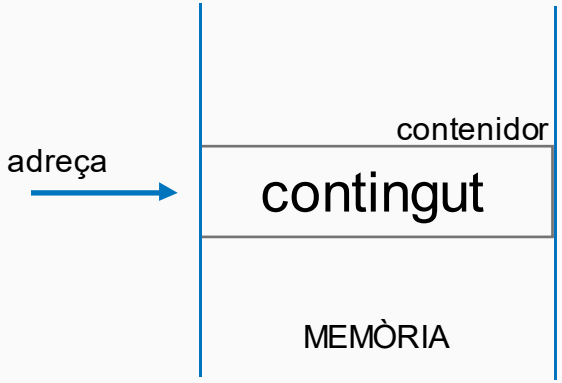


Declarat com	Accés al contingut	Accés a l'adreça
`classe nom_objecte`	`nom_objecte`	`&nom_objecte`
`classe *nom_objecte`	`*nom_objecte`	`nom_objecte`

2. Programació Orientada a Objectes
2.1 Encapsulament

Els objectes, igual que les variables poden ser declarats com apuntadors.
Tot objecte té: 1) El valor del seu contingut i 2) La seva adreça de memòria

Ús de l'objecte dins el codi



Declarat com	Accés al contingut d'un membre
`classe nom_objecte`	`nom_objecte.x`
`classe *nom_objecte`	`nom_objecte->x` *o bé* `(*nom_objecte).x`

2. Programació Orientada a Objectes
2.1 Encapsulament

Exercici: Considerant el següent codi, respondre al qüestionari

```
class Point {
public:
int x,y;
public:
Point(int coordx, int coordy){
  x = coordx;
  y = coordy;
  }
Point(){x = 0; y = 0;}
}
```

```
Point p1;
Point p2(2,3);
Point *p3;

p3=&p2;
```

Accés al qüestionari: http://www.kahoot.it

Accés pel professor

2. Programació Orientada a Objectes
2.1 Encapsulament

Exercici: Considerant el següent codi, respondre al qüestionari

```
class Point {
public:
int x,y;
public:
Point(int coordx, int coordy){
  x = coordx;
  y = coordy;
  }
Point(){x = 0; y = 0;}
}
...
Point p1;
Point p2(2,3);
Point *p3;
p3=&p2;
```

Accedir a x de p1

p1.x ✓

p1->x

2. Programació Orientada a Objectes
2.1 Encapsulament

Exercici: Considerant el següent codi, respondre al qüestionari

```
class Point {
public:
int x,y;
public:
Point(int coordx, int coordy){
  x = coordx;
  y = coordy;
  }
Point(){x = 0; y = 0;}
}
...
Point p1;
Point p2(2,3);
Point *p3;
p3=&p2;
```

Accedir a x de p2

p2.x ✓

p2->x

2. Programació Orientada a Objectes
2.1 Encapsulament

Exercici: Considerant el següent codi, respondre al qüestionari

```
class Point {
public:
int x,y;
public:
Point(int coordx, int coordy){
  x = coordx;
  y = coordy;
  }
Point(){x = 0; y = 0;}
}
...
Point p1;
Point p2(2,3);
Point *p3;
p3=&p2;
```

Accedir a x de p3

p3.x

p3->x ✓

2. Programació Orientada a Objectes
2.1 Encapsulament

Exercici: Considerant el següent codi, respondre al qüestionari

```
class Point {
public:
int x,y;
public:
Point(int coordx, int coordy){
  x = coordx;
  y = coordy;
  }
Point(){x = 0; y = 0;}
}
...
Point p1;
Point p2(2,3);
Point *p3;
p3=&p2;
```

Valor de p1.x

0 ✓

2

3

2. Programació Orientada a Objectes
2.1 Encapsulament

Exercici: Considerant el següent codi, respondre al qüestionari

```
class Point {
public:
int x,y;
public:
Point(int coordx, int coordy){
  x = coordx;
  y = coordy;
  }
Point(){x = 0; y = 0;}
}
...
Point p1;
Point p2(2,3);
Point *p3;
p3=&p2;
```

Valor de p2.x

0

2 ✓

3

2. Programació Orientada a Objectes
2.1 Encapsulament

Exercici: Considerant el següent codi, respondre al qüestionari

```
class Point {
public:
int x,y;
public:
Point(int coordx, int coordy){
  x = coordx;
  y = coordy;
  }
Point(){x = 0; y = 0;}
}
...
Point p1;
Point p2(2,3);
Point *p3;
p3=&p2;
```

Valor de p3->x

0

2 ✓

3

2. Programació Orientada a Objectes
2.1 Encapsulament

Exercici: Considerant el següent codi, respondre al qüestionari

```
class Point {
public:
int x,y;
public:
Point(int coordx, int coordy){
  x = coordx;
  y = coordy;
  }
Point(){x = 0; y = 0;}
}
...
Point p1;
Point p2(2,3);
Point *p3;
p3=&p2;
```

Assigna a y de p3
el valor de x de p2

a) p3.y=p2.x
b) p3->y=p2.x
c) *p3.y=p2.x
d) b i c són correctes ✓

2. Programació Orientada a Objectes
2.1 Encapsulament

Exercici: Considerant el següent codi, respondre al qüestionari

```
class Point {
public:
int x,y;
public:
Point(int coordx, int coordy){
  x = coordx;
  y = coordy;
  }
Point(){x = 0; y = 0;}
}
...
Point p1;
Point p2(2,3);
Point *p3;
p3=&p2;
```

Fer que p3 apunti a p1

*p3=p1

&p3=*p1

*p3=&p1

p3=&p1 ✓

2. Programació Orientada a Objectes
2.1 Encapsulament

Exercici: Considerant el següent codi, respondre al qüestionari

```
class Point {
public:
int x,y;
public:
Point(int coordx, int coordy){
  x = coordx;
  y = coordy;
  }
Point(){x = 0; y = 0;}
}
...
Point p1;
Point p2(2,3);
Point *p3;
p3=&p2;
```

2. Programació Orientada a Objectes
2.1 Encapsulament i gestió dinàmica de memòria en C++

new i ***delete*** permeten assignar i alliberar memòria a apuntadors d'objectes

```
class Point {
public:
int x,y;
public:
Point(int coordx, int coordy){
  x = coordx;
  y = coordy;
  }
Point(){x = 0; y = 0;}
}
```

```
Point p1;
Point p2(2,3);
Point *p3;

p3=new Point;  // Assigna memòria per una nova
                  instància de l'objecte Point

P3->x=p2.x;

delete p3;    //  Allibera la memòria reservada
```

2. Programació Orientada a Objectes
2.1 Encapsulament

Exemples: La classe "Box"

Declaració i ús de la classe, amb petites variants. Defineix les característiques de la capsa (ample, llarg i fons) i un mètode per calcular el volum.

Referència del codi

pr11001.cpp	Codi de referència inicial
pr11002.cpp	Definint mètodes dins la declaració de la classe
pr11003.cpp	Opcions d'inicialització diverses
pr11004.cpp	Múltiples constructors

2. Programació Orientada a Objectes
Nova notació Input/Output amb C++

Canvi de sintaxi per lectura / escriptura de teclat i pantalla

```
#include <iostream>
using namespace std;
...
cin >> var;
cout << “text”<< var << endl;
...
```

```
#include <iostream>

...
std::cin >> var;
std::cout << “text”<< var <<
  std::endl;
...
```

2. Programació Orientada a Objectes
Cadenes de caràcters amb C++

C++ disposa de l'objecte *string* per la gestió de cadenes de caràcters

Membres funció

```
int length();
string swap(string);
int find(string);
...
```

Operands

```
=
+
+=
[]
```

Ref. http://www.cplusplus.com/reference/string/string/string/

2. Programació Orientada a Objectes

UPC – Disseny i Desenvolupament de Videojocs – Programació II

2. Programació Orientada a Objectes

Característiques principals dels llenguatges Orientats a Objectes

Encapsulament

Herència

Polimorfisme

2. Programació Orientada a Objectes
2.1 Encapsulament *(resum sessió anterior)*

L'encapsulament és el mecanisme d'abstracció bàsic dels llenguatges Orientats a Objectes per a definir unitats de codi i dades.

```
class Box {
private:
int llarg, ample, fons;

public:
int volum() {
  return llarg * ample * fons;
  }
};
```

Els objectes en C++ es defineixen amb el que s'anomena "classe" *(class)*.
Un objecte és una instància d'una classe.
Les classes defineixen els membres de dades (variables) i els membres funció (codi). Tant els uns com els altres tenen diferents nivells de visibilitat: i) "*public*", ii) "*protected*" i iii) "*private*".

2. Programació Orientada a Objectes
2.1 Encapsulament *(resum sessió anterior)*

Un Objecte és una instància d'una classe

```
Box capsa1;          //Declaració de capsa1 com a objecte de tipus Box
Box capsa2;          //Declaració de capsa2 com a objecte de tipus Box

capsa1.ample         // Propietat, atribut o membre de dades 'ample' de l'objecte
                     capsa1 de tipus Box

capsa1.volum()       // Mètode o membre funció 'volum' de l'objecte capsa1 que
                     calcula el seu volum
```

2. Programació Orientada a Objectes
2.1 Encapsulament *(resum sessió anterior)*

El **Constructor** és un mètode especial d'una classe que pren com a nom el mateix nom que el de la pròpia classe i serveix per inicialitzar l'objecte creat

```
class Point {
private:
int x,y;
public:
Point(int coordx, int coordy){
  x = coordx;
  y = coordy;
  }
Point(){x = 0; y = 0;}
int getx(){return x;}
int gety(){return y;}
}
```

Els constructors s'executen al moment en que s'instancia un objecte de la classe.

Se'n poden declarar més d'un. En aquest cas s'executarà aquell que té definits els paràmetres tal i com s'utilitzen.

```
Point p1;
Point p2(2,3);
```

2. Programació Orientada a Objectes
2.1 Encapsulament *(resum sessió anterior)*

El **Destructor** és un mètode especial d'una classe que pren com a nom el mateix nom que el de la pròpia classe precedit pel símbol ~

```
class Point {
private:
int x,y;
public:
Point(){
  x = 0; y = 0;
  }

~Point(){
  //Destructor
  };
}
```

Els destructors s'executen al moment en que s'elimina la instància d'un objecte.

Típicament alliberen la memòria dinàmica que s'hagi pogut reservar degut a la creació de l'objecte.

```
Point p1;
Point p2(2,3);
```

2. Programació Orientada a Objectes
Gestió dinàmica de memòria en C++ *(recordatori)*

new i ***delete*** permeten assignar i alliberar memòria a apuntadors d'objectes

```
class Point {
private:
int x,y;
public:
Point(int coordx, int coordy){
  x = coordx;
  y = coordy;
  }
Point(){x = 0; y = 0;}
}
```

```
Point p1;
Point p2(2,3);
Point *p3;

p3=new Point;  //Assigna memòria per una nova
               instància de l'objecte Point

P3->x=p2.x;

delete p3;   // Allibera la memòria reservada
```

2. Programació Orientada a Objectes

Exercici: Escriure el codi C++ per a representar una classe de nom “polygon” i declarar un objecte de nom p1 d’aquesta classe.

Un objecte de la classe “polygon” conté:

1. un membre de dades « privat » de tipus string i de nom « mysecret »

2. Dos membres de dades « protected » de tipus enter
i de nom « width » i « height »

3. Un membre funció « public » de nom « set_values »
al que se li passen com a paràmetres dos enters per inicialitzar els membres de dades « width » i « height »

2. Programació Orientada a Objectes

Exercici: Escriure el codi C++ per a representar la classe “polygon” i declarar un objecte de nom p1 d’aquesta classe.

```
class Polygon {
  private:
    char* mysecret;
  protected:
    int width, height;
  public:
    void set_values (int a, int b) {
      width=a; height=b;
      }
 };

void main () {
  Polygon p1;
  }
```

... què li falta a aquest codi?

2. Programació Orientada a Objectes

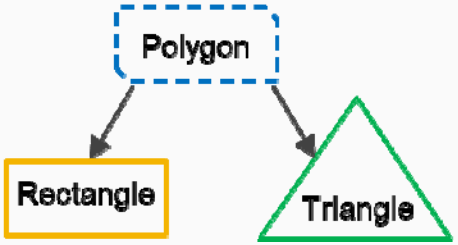
Característiques principals dels llenguatges Orientats a Objectes

Encapsulament

Herència

Polimorfisme

L'herència és el mecanisme a través del que una classe hereta propietats d'una altra

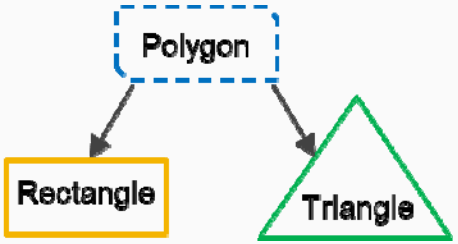


2. Programació Orientada a Objectes
2.2 Herència

L'herència és el mecanisme a través del que una classe hereta propietats d'una altra.

Les característiques o propietats comunes dels objectes poden ser declarades en una classe "base"

Les classes "derivades" hereten les propietats de la "base" i defineixen les seves pròpies, que les diferencien de la "base"

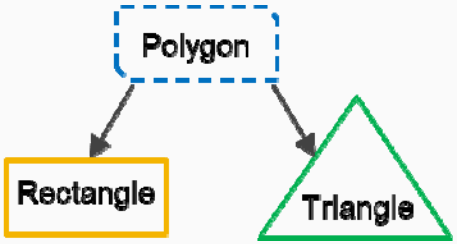


2. Programació Orientada a Objectes
2.2 Herència

L'herència és el mecanisme a través del que una classe hereta propietats d'una altra.

Raons i utilitat de l'herència

1. 1. Mantenir intacte el codi i comportament de les classes base
2. 2. Disposar de llibreries base sense accés al codi font
3. 3. Polimorfisme dels membres funció de la classe base i derivades

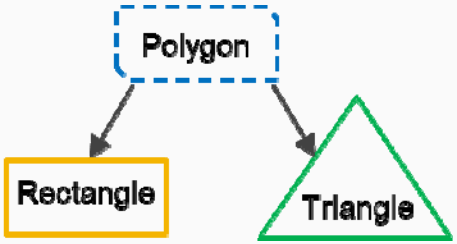


2. Programació Orientada a Objectes
2.2 Herència

L'herència és el mecanisme a través del que una classe hereta propietats d'una altra.

```
class Base {
  ...
 };

class Derived1: public Base {
  ...
 };

class Derived2: public Base {
  ...
 };
```

Accés	*Public*	*Protected*	*private*
Mateixa classe	SÍ	SÍ	SÍ
Classe derivada	SÍ	SÍ	NO
Fora de la classe	SÍ	NO	NO



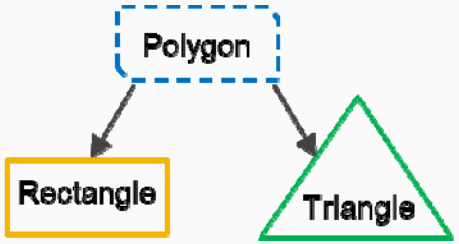
2. Programació Orientada a Objectes
2.2 Herència

L'herència és el mecanisme a través del que una classe hereta propietats d'una altra.

```
class Polygon {
  private:
       char* mysecret;

  protected:
    int width, height;

  public:
    Polygon() {
    cout << "Polygon!"; }

    void set_values (int a, int b) {
    width=a; height=b;}
 };
```

```
class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
 };

class Triangle: public Polygon {
  public:
    int area ()
      { return width * height / 2; }
  };
```

```
int main () {
  Rectangle rect;
  Triangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  cout <<rect.area()<<'\n';
  cout <<trgl.area()<<'\n';
  return 0;
}
```



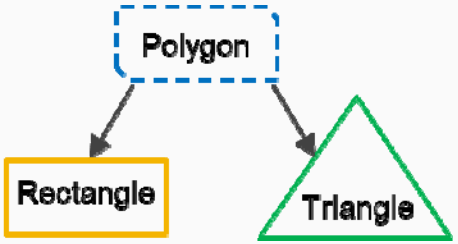
2. Programació Orientada a Objectes
2.2 Herència

L'herència és el mecanisme a través del que una classe hereta propietats d'una altra.

```
class Polygon {
  private:
       char* mysecret;

  protected:
    int width, height;

  public:
     Polygon() {
     cout << "Polygon!"; }

     void set_values (int a, int b) {
     width=a; height=b;}
 };
```

```
class Rectangle: public Polygon {
  public:
     int area ()
       { return width * height; }
 };

class Triangle: public Polygon {
  public:
     int area ()
       { return width * height / 2; }
  };
```

```
int main () {
  Rectangle rect;
  Triangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  cout <<rect.area()<<'\n';
  cout <<trgl.area()<<'\n';
  return 0;
}
```



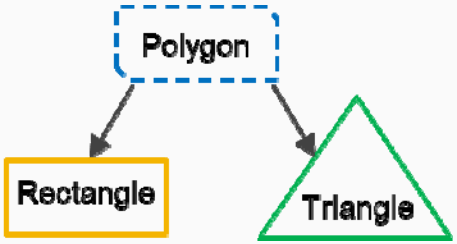
2. Programació Orientada a Objectes
2.2 Herència

L'herència és el mecanisme a través del que una classe hereta propietats d'una altra.

Elements que s'hereten
1. Mètodes i propietats "Protected"
2. Mètodes i propietats "Public"

NO s'hereta
1. Mètodes i propietats "Private"
2. Constructors i Destructors

Accés	*Public*	*Protected*	*private*
Mateixa classe	SÍ	SÍ	SÍ
Classe derivada	SÍ	SÍ	NO
Fora de la classe	SÍ	NO	NO



2. Programació Orientada a Objectes
2.2 Herència

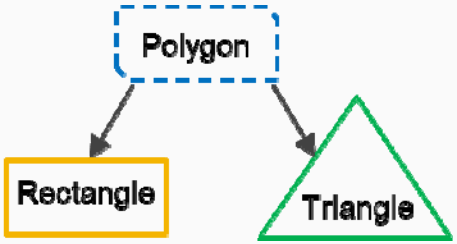
L'herència és el mecanisme a través del que una classe hereta propietats d'una altra.

Elements que s'hereten
1. Mètodes i propietats "Protected"
2. Mètodes i propietats "Public"

NO s'hereta
1. Mètodes i propietats "Private"
2. Constructors i Destructors

El constructor de la classe "pare" base és cridat automàticament en declarar un objecte de la classe derivada

Si es vol activar un altre constructor, aquest ha de ser especificat a la classe derivada



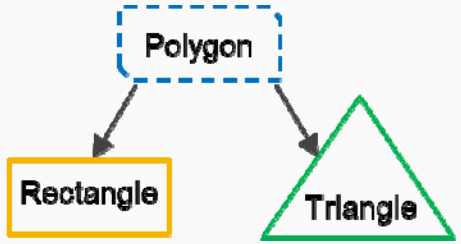
2. Programació Orientada a Objectes
2.2 Herència

L'herència és el mecanisme a través del que una classe hereta propietats d'una altra.

```
class Polygon {
  private:
       char* mysecret;

  protected:
    int width, height;

  public:
    Polygon() {
    std::cout << "Polygon!"; }

    void set_values (int a, int b) {
    width=a; height=b;}
 };
```

```
class Rectangle: public Polygon {
  public:


     int area ()
       { return width * height; }
 };

class Triangle: public Polygon {
  public:
    int area ()
      { return width * height / 2; }
  };
```

```
int main () {
  Rectangle rect;
  Triangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  cout <<rect.area()<<'\n';
  cout <<trgl.area()<<'\n';
  return 0;
}
```



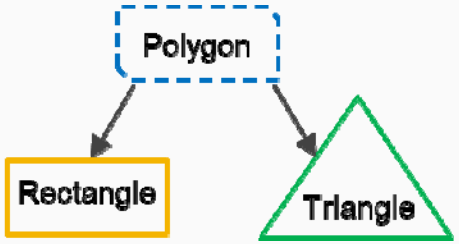
2. Programació Orientada a Objectes
2.2 Herència

L'herència és el mecanisme a través del que una classe hereta propietats d'una altra.

```
class Polygon {
  private:
       char* mysecret;

  protected:
    int width, height;

  public:
    Polygon() {
    std::cout << "Polygon!"; }

    Polygon (int a, int b) {
    width=a; height=b;}
 };
```

```
class Rectangle: public Polygon {
  public:


     int area ()
       { return width * height; }
 };

class Triangle: public Polygon {
  public:
    int area ()
      { return width * height / 2; }
  };
```

```
int main () {
  Rectangle rect(4,5);
  Triangle trgl(4,5);

  cout <<rect.area()<<'\n';
  cout <<trgl.area()<<'\n';
  return 0;
}
```



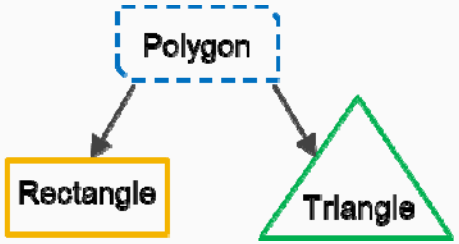
2. Programació Orientada a Objectes
2.2 Herència

L'herència és el mecanisme a través del que una classe hereta propietats d'una altra.

```
class Polygon {
  private:
       char* mysecret;

  protected:
    int width, height;

  public:
    Polygon() {
    std::cout << "Polygon!"; }

    Polygon (int a, int b) {
    width=a; height=b;}
 };
```

```
class Rectangle: public Polygon {
  public:
  Rectangle(int w, int h):Polygon(w, h){
    cout << "Rectangle!" << endl;
    }
    int area ()
      { return width * height; }
 };

class Triangle: public Polygon {
  public:
    int area ()
      { return width * height / 2; }
 };
```

```
int main () {
  Rectangle rect(4,5);
  Triangle trgl(4,5);

  cout <<rect.area()<<'\n';
  cout <<trgl.area()<<'\n';
  return 0;
}
```



2. Programació Orientada a Objectes
2.2 Herència

L'herència és el mecanisme a través del que una classe hereta propietats d'una altra.

C++ permet especificar classes derivades de més d'una classe de manera simultània.

COMPTE!: L'ús d'herència múltiple pot derivar a generar codi il·legible

2. Programació Orientada a Objectes
2.2 Herència

L'herència és el mecanisme a través del que una classe hereta propietats d'una altra.

```
class Polygon {
private:
  char* mysecret;
protected:
  int width, height;
public:
  Polygon(int w, int h) {
    width = w; height = h;
    cout << "Polygon!" << endl; }
  void set_values (int a, int b) {
    width = a; height = b;
    }
};
```

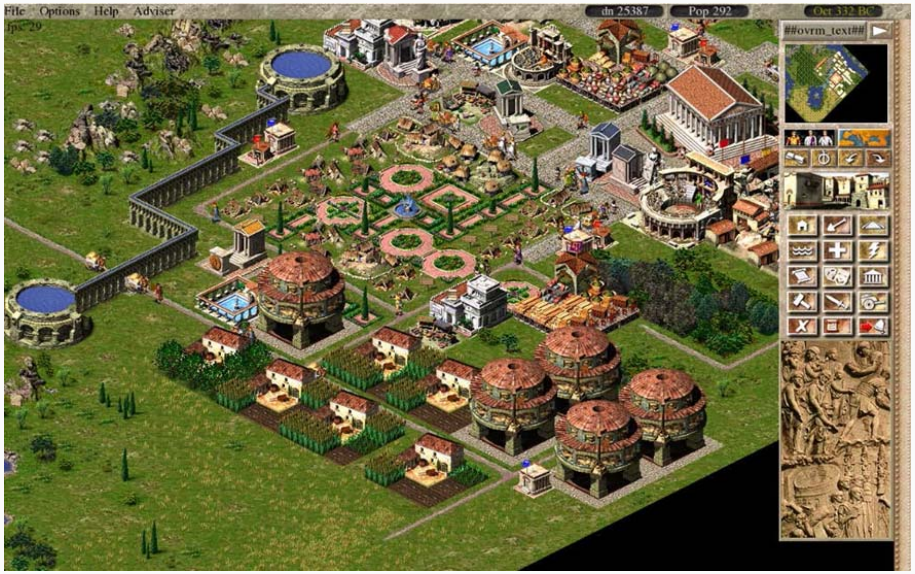
```
class Disease {
public:
  void killme() {
    cout << "why????????" << endl;
    }
  };
```

```
class Abomination : public Polygon, public Disease {
public:
  Abomination(int w, int h) : Polygon(w, h) {
    cout << "Rectangle!" << endl;
    }

int area() {
  return width * height;
  }
};
```

COMPTE!: L'ús d'herència múltiple pot derivar a generar codi il·legible

2. Programació Orientada a Objectes
2.2 Herència

Exercici: Els edificis de Caesar



Implementar la classe **Building** amb l'string ***nom*** com a membre protegit, un constructor que rep un nom de cadena i un mètode públic ***getName ()*** que retorna el nom de l'edifici.

- Implementar la classe **Warehouse** derivada de Building
- Implementar la classe **House** derivada de Building
- Implementar la classe **Temple** derivada de Building

Veure detalls per la implementació i dels elements de cada classe a l'enunciat publicat al campus

2. Programació Orientada a Objectes
2.2 Herència

Exercici: Vehicles

Implementar la classe **Vehicle** amb l'string ***nom*** com a membre protegit, un constructor que rep un nom de cadena i un mètode públic ***getNom ()*** que retorna el nom del vehicle.

- Implementar la classe **Cotxe** derivada de Vehicle
- Implementar la classe **Avió** derivada de Vehicle

Veure detalls per la implementació i dels elements de cada classe a l'enunciat publicat al campus

2. Programació Orientada a Objectes

UPC – Disseny i Desenvolupament de Videojocs – Programació II

2. Programació Orientada a Objectes
2.3 Amistat (Friendship)

L'Amistat (Friendship) és el mecanisme mitjançant el que des d'un membre funció d'una classe pot accedir a les propietats privades d'una altra classe.

```
class CustomDate {
protected:
  int day,year;
public:
...
friend class Date;
  };

Class Date {
...
  }
```

Si la classe A declara a B com amiga, llavors B pot accedir als membres protegits d'A

2. Programació Orientada a Objectes

UPC – Disseny i Desenvolupament de Videojocs – Programació II

2. Programació Orientada a Objectes

Contingut sessió

Repàs dels exercicis proposats
- Caesar (herència)
- Vehicles (herència)
- Coets (encapsulament)

Teoria: Polimorfisme en jerarquia de classes (re-escriptura) i sobrecàrrega d'operadors

Programació II - Avaluació

Calendari dels primers controls

Test 1 (10%) i Test 2 (10%)
Dimarts 19 de març. Exàmens tipus test

Examen parcial (20%)
Dimarts 26 de març. Prova pràctica (programació)

1. **Punters i gestió dinàmica de la memòria**
   Pas de paràmetres per valor vs pas de paràmetres per referència / Reserva i alliberament de memòria

2. **Programació Orientada a Objectes**
   Encapsulament, Herència, Sobrecàrrega d'operadors i Polimorfisme

3. Estructures de dades
4. Recursivitat
5. Algorismes d'ordenació

2. Programació Orientada a Objectes

Característiques principals dels llenguatges Orientats a Objectes

Encapsulament

Herència

Polimorfisme

2. Programació Orientada a Objectes

Característiques principals dels llenguatges Orientats a Objectes

Encapsulament

Herència

Polimorfisme

El polimorfisme és la manera en què els llenguatges Orientats a Objectes implementen el concepte de polisèmia del món real: **Un únic nom per a molts significats, segons el context**

2. Programació Orientada a Objectes

Característiques principals dels llenguatges Orientats a Objectes

Encapsulament

Herència

Polimorfisme

Tipus de polimorfisme

Constructores de les classes

Diferents membres funció amb el mateix nom, activats segons els paràmetres rebuts

2. Programació Orientada a Objectes

Característiques principals dels llenguatges Orientats a Objectes

Encapsulament

Herència

Polimorfisme

Tipus de polimorfisme

Reescriptura de membres funció en jerarquia de classes

Diferents implementacions d'un membre funció del mateix nom en diferents classes d'una mateixa jerarquia

2. Programació Orientada a Objectes

Característiques principals dels llenguatges Orientats a Objectes

Encapsulament

Herència

Polimorfisme

Tipus de polimorfisme

Sobrecàrrega d'operadors

Un mateix operador té comportaments diferents segons el tipus dels operands

2. Programació Orientada a Objectes
2.4 Polimorfisme

El polimorfisme és la manera en què els llenguatges O.O. implementen el concepte de polisèmia: Un únic nom per a molts significats, segons el context

Un membre funció pot tenir diferents implementacions segons els paràmetres que s'indiquin. S'activarà una o altra versió de la funció segons la manera com es cridi.

```
26  class Cabines { // Classe principal per les Cabines
27    int maxCom;    // Nombre màxim de Tripulants de tipous tècnic
28    int numCom;    // Quantitatt de Tripulants de tipus tècnic
29    Comandant **com; // Array d'apuntadors a Comandants
30
31    int maxTec;   // Nombre màxim de Tripulants de tipus tècnic
32    int numTec;   // Quantitat de Tripulants de tipus tècnic
33    Tecnic **tec; // Array d'apuntadors a tècnics
34
35    public:
36      Cabines(int maxC, int maxT); // Constructor
37      int assigna(Comandant *emp);     // Assigna Cabina a un Comandant
38      int assigna(Tecnic *emp);      // Assigna Cabina a un tècnic
39      void qui();                    // Visualitza qui ocupa cada Cabina
40      ~Cabines() {                 // Destructor
41        delete[] com;
42        delete[] tec;
43        }
44  };
```

2. Programació Orientada a Objectes
2.4 Polimorfisme

El polimorfisme és la manera en què els llenguatges O.O. implementen el concepte de polisèmia: Un únic nom per a molts significats, segons el context

Sobre-escriptura: En una jerarquia d'herència una classe derivada pot reescriure un membre funció de la classe base (overriding)

La classe base ha d'indicar que un membre funció pot ser sobreescrit

```
class Pare {
public: virtual int exemple(int a) { std::cout << "pare"; };
  };

class Filla : public Pare {
public: int exemple(int a) { std::cout << "Filla"; };
};
```

2. Programació Orientada a Objectes
2.4 Polimorfisme

Exemple: “Tripulants i Cabines”

Cas d’ús en el que veurem com el polimorfisme i la sobre-escriptura facilita el manteniment del codi i el simplifica.

2. Programació Orientada a Objectes
2.4 Polimorfisme

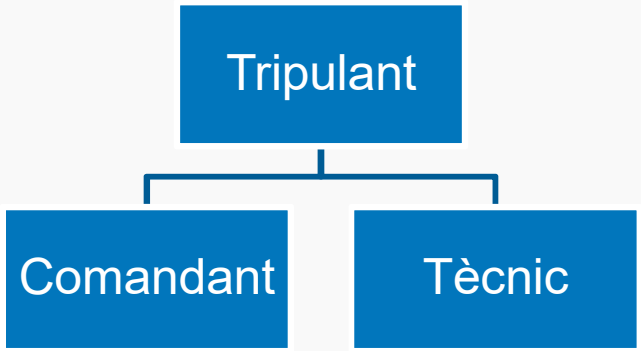
Exemple: “Tripulants i Cabines” - Descripció de l’escenari

Els Tripulants d’una nau poden ser de diferents tipus (comandaments, enginyers, tècnics, ...).

Definirem una classe base i dues derivades que heretaran les propietats de la principal

2. Programació Orientada a Objectes
2.4 Polimorfisme

Exemple: “Tripulants i Cabines” - Descripció de l’escenari



2. Programació Orientada a Objectes
2.4 Polimorfisme

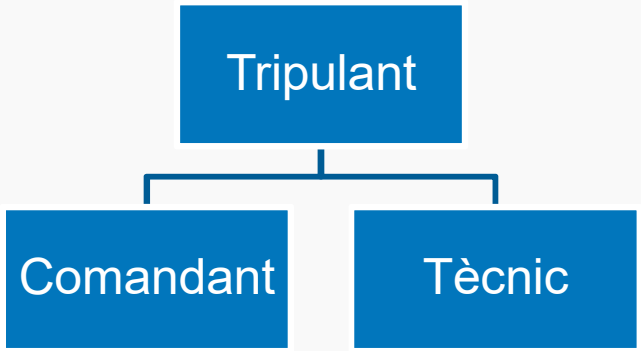
Exemple: “Tripulants i Cabines” - Descripció de l’escenari

Les Cabines poden ser ocupades pels Tripulants.

La classe Cabines tindrà un mètode funció per assignar cadascuna de les seves instàncies a un tripulant

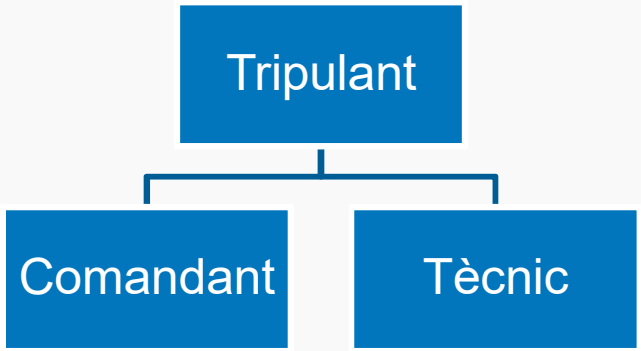
2. Programació Orientada a Objectes
2.4 Polimorfisme

Exemple: “Tripulants i Cabines” - Descripció de l’escenari



2. Programació Orientada a Objectes
2.4 Polimorfisme

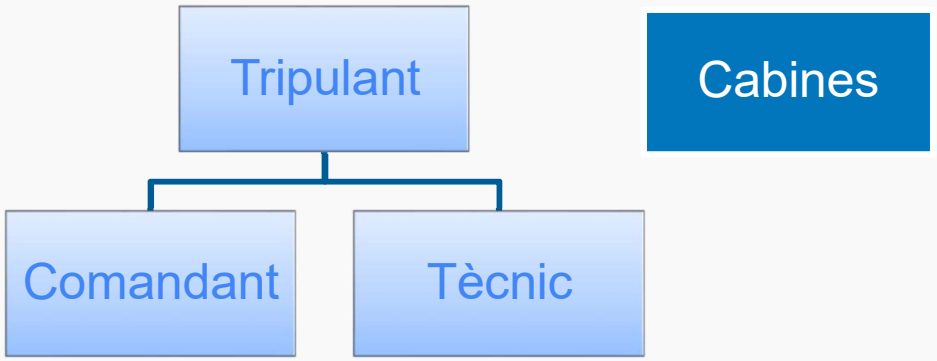
Exemple. "Tripulants i Cabines" – Codi de les classes Tripulant




```
7   class Tripulant { // Classe principal pels Tripulants
8     protected:
9       string nom; // Nom del Tripulant
10    public:
11      Tripulant(string n) { nom = n; } // Constructor
12    };
13
14  class Comandant :public Tripulant { //Subclasse pel Tripulant de tipous Comandant
15    public:
16      Comandant(string n) :Tripulant(n) {} // Constructor
17      void meuNom() { cout << "Comandant: " << nom; }
18    };
19
20  class Tecnic :public Tripulant { //Subclasse pel Tripulant de tipous tècnic
21    public:
22      Tecnic(string n) : Tripulant(n) {} //Constructor
23      void meuNom() { cout << "Tecnic: " << nom; }
24    };
25
```

2. Programació Orientada a Objectes
2.4 Polimorfisme

Exemple. "Tripulants i Cabines" – Codi de la classe Cabines




```
26  class Cabines { // Classe principal per les Cabines
27    int maxCom;    // Nombre màxim de Tripulants de tipous tècnic
28    int numCom;    // Quantitatt de Tripulants de tipus tècnic
29    Comandant **com; // Array d'apuntadors a Comandants
30
31    int maxTec;   // Nombre màxim de Tripulants de tipus tècnic
32    int numTec;   // Quantitat de Tripulants de tipus tècnic
33    Tecnic **tec; // Array d'apuntadors a tècnics
34
35    public:
36      Cabines(int maxC, int maxT); // Constructor
37      int assigna(Comandant *emp);     // Assigna Cabina a un Comandant
38      int assigna(Tecnic *emp);      // Assigna Cabina a un tècnic
39      void qui();                    // Visualitza qui ocupa cada Cabina
40      ~Cabines() {                 // Destructor
41        delete[] com;
42        delete[] tec;
43        }
44  };
```

2. Programació Orientada a Objectes
2.4 Polimorfisme

Exemple. "Tripulants i Cabines" – Codi de la classe Cabines

```
46  Cabines::Cabines(int maxC, int maxT) {
47      maxCom = maxC;
48      maxTec = maxT;
49      numCom = numTec = 0;
50      com = new Comandant *[maxCom];
51      tec = new Tecnic *[maxTec];
52      int i;
53
54      for (i = 0; i < maxCom; i++) com[i] = NULL;
55      for (i = 0; i < maxTec; i++) tec[i] = NULL;
56      }
```

```
26  class Cabines { // Classe principal per les Cabines
27    int maxCom;     // Nombre màxim de Tripulants de tipous tècnic
28    int numCom;     // Quantitatt de Tripulants de tipus tècnic
29    Comandant **com; // Array d'apuntadors a Comandants
30
31    int maxTec;   // Nombre màxim de Tripulants de tipus tècnic
32    int numTec;   // Quantitat de Tripulants de tipus tècnic
33    Tecnic **tec; // Array d'apuntadors a tècnics
34
35    public:
36      Cabines(int maxC, int maxT); // Constructor
37      int assigna(Comandant *emp);     // Assigna Cabina a un Comandant
38      int assigna(Tecnic *emp);      // Assigna Cabina a un tècnic
39      void qui();                    // Visualitza qui ocupa cada Cabina
40      ~Cabines() {                 // Destructor
41        delete[] com;
42        delete[] tec;
43        }
44  };
```

2. Programació Orientada a Objectes
2.4 Polimorfisme

Exemple. "Tripulants i Cabines" – Codi de la classe Cabines

```
62  int Cabines::assigna(Comandant *c) {
63      if (numCom == maxCom) return -1;
64      else {
65        com[numCom] = c;
66        numCom++;
67        return numCom;
68        }
69  }
70
71  int Cabines::assigna(Tecnic *t) {
72      int i;
73      if (numTec == maxTec) return -1;
74      else {
75          tec[numTec] = t;
76          numTec++;
77          return numTec;
78      }
79  }
```

```
26  class Cabines { // Classe principal per les Cabines
27    int maxCom;     // Nombre màxim de Tripulants de tipous tècnic
28    int numCom;     // Quantitatt de Tripulants de tipus tècnic
29    Comandant **com; // Array d'apuntadors a Comandants
30
31    int maxTec;   // Nombre màxim de Tripulants de tipus tècnic
32    int numTec;   // Quantitat de Tripulants de tipus tècnic
33    Tecnic **tec; // Array d'apuntadors a tècnics
34
35    public:
36      Cabines(int maxC, int maxT); // Constructor
37      int assigna(Comandant *emp);     // Assigna Cabina a un Comandant
38      int assigna(Tecnic *emp);      // Assigna Cabina a un tècnic
39      void qui();                    // Visualitza qui ocupa cada Cabina
40      ~Cabines() {                 // Destructor
41        delete[] com;
42        delete[] tec;
43        }
44  };
```

2. Programació Orientada a Objectes
2.4 Polimorfisme

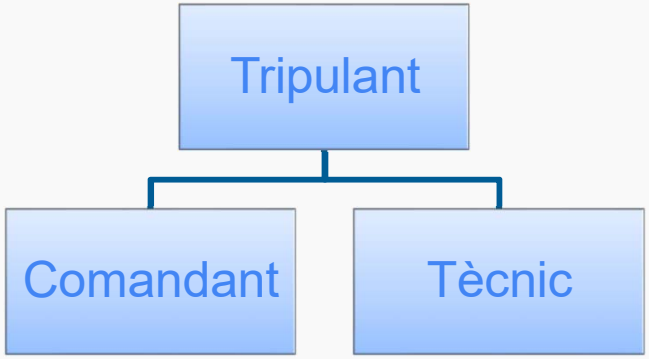
Exemple. "Tripulants i Cabines" – Codi de la classe Cabines

```
62  int Cabines::assigna(Comandant *c) {
63      if (numCom == maxCom) return -1;
64      else {
65        com[numCom] = c;
66        numCom++;
67        return numCom;
68        }
69  }
70
71  int Cabines::assigna(Tecnic *t) {
72      int i;
73      if (numTec == maxTec) return -1;
74      else {
75          tec[numTec] = t;
76          numTec++;
77          return numTec;
78      }
79  }
```

```
26  class Cabines { // Classe principal per les Cabines
27    int maxCom;    // Nombre màxim de Tripulants de tipous tècnic
28    int numCom;    // Quantitatt de Tripulants de tipus tècnic
29    Comandant **com; // Array d'apuntadors a Comandants
30
31    int maxTec;   // Nombre màxim de Tripulants de tipus tècnic
32    int numTec;   // Quantitat de Tripulants de tipus tècnic
33    Tecnic **tec; // Array d'apuntadors a tècnics
34
35    public:
36      Cabines(int maxC, int maxT); // Constructor
37      int assigna(Comandant *emp);     // Assigna Cabina a un Comandant
38      int assigna(Tecnic *emp);      // Assigna Cabina a un tècnic
39      void qui();                    // Visualitza qui ocupa cada Cabina
40      ~Cabines() {                 // Destructor
41        delete[] com;
42        delete[] tec;
43        }
44  };
```

2. Programació Orientada a Objectes
2.4 Polimorfisme

Exemple. "Tripulants i Cabines" – Codi de la classe Cabines



Cabines

Per a veure quins Tripulants hi ha a cada cabina, la classe Cabines també tindrà un mètode funció per a llistar-los

2. Programació Orientada a Objectes
2.4 Polimorfisme

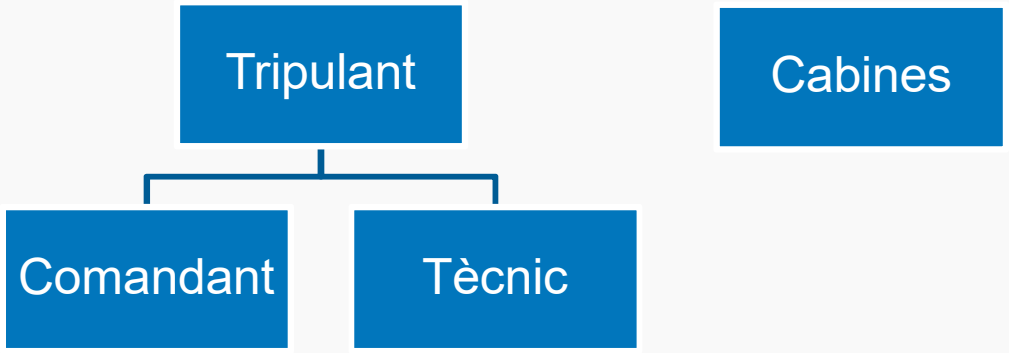
Exemple. "Tripulants i Cabines" – Codi de la classe Cabines

```
77  void Cabines::qui() {
78    int i;
79
80    if (numCom>0)
81      for (i=0;i<maxCom;i++)
82        if (com[i]!=NULL) {
83          cout <<"Cabina " << i << " ocupada per ";
84          com[i]->meuNom();
85          cout << endl;
86          }
87
88    if (numTec>0)
89        for (i = 0; i<maxTec; i++)
90            if (tec[i] != NULL) {
91                cout << "Cabina " << i << " ocupada per ";
92                tec[i]->meuNom();
93                cout << endl;
94            }
95    }
```

Per a veure quins Tripulants hi ha a cada cabina, la classe Cabines també tindrà un mètode funció per a llistar-los: `qui()`

2. Programació Orientada a Objectes
2.4 Polimorfisme

Exemple. "Tripulants i Cabines" – Codi principal: Ús de les classes

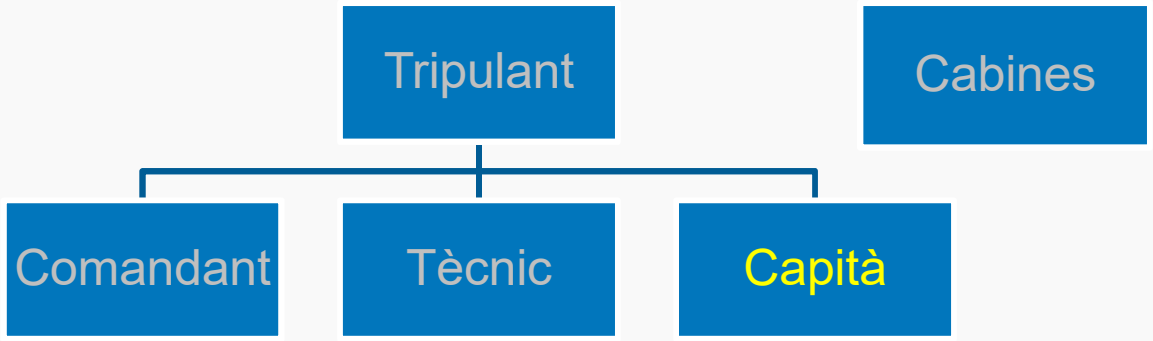



```
void main() {
Comandant c1("Joan");
Tecnic    t1("Nuria");
Comandant c2("Griselda");
Tecnic    t2("Roger");

Cabines  c(5, 5);

setlocale(LC_ALL, "");

c.assigna(&c1);
c.assigna(&t1);
c.assigna(&c2);
c.assigna(&t2);
c.qui();
system("pause");
}
```

2. Programació Orientada a Objectes
2.4 Polimorfisme

Exemple. "Tripulants i Cabines" – Exercici: Afegir nova classe de Tripulants

Que cal fer al codi (afegir/modificar) per incloure la nova classe "Capità"?



2. Programació Orientada a Objectes
2.4 Polimorfisme

Exemple. “Tripulants i Cabines” – Exercici: Afegir nova classe de Tripulants

Que cal fer al codi (afegir/modificar) per incloure la nova classe “Capità”?

1. Definir la nova classe
2. Modificar la classe de **Cabines**
   (tant els atributs com els mètodes)
3. Crear nova versió del mètode **assigna**
4. Cal també reescriure el mètode **qui**

2. Programació Orientada a Objectes
2.4 Polimorfisme

Exemple. "Tripulants i Cabines" – NOVA VERSIÓ DEL CODI

*Aprofitarem la potència del **polimorfisme en la jerarquia de classes** per afegir la possibilitat de definir un nou Tripulant sense categoria específica*

Les Cabines s'assignaran a un Tripulant, independentment del tipus que sigui

2. Programació Orientada a Objectes
2.4 Polimorfisme

Exemple. “Tripulants i Cabines” – NOVA VERSIÓ DEL CODI

Abans

```
class Cabines {    // Classe principal per les Cabines
  int maxCom;      // Nombre màxim de Tripulants de tipus Comandant
  int numCom;      // Quantitatt de Tripulants de tipus Comadandant
  Comandant **com; // Array d'apuntadors a Comandants

  int maxTec;      // Nombre màxim de Tripulants de tipus tècnic
  int numTec;      // Quantitat de Tripulants de tipus tècnic
  Tecnic **tec;    // Array d'apuntadors a tècnics

  public:
  Cabines(int, int);          // Constructor
  int assigna(Comandant *);   // Assigna Cabina a un Comandant
  int assigna(Tecnic *);      // Assigna Cabina a un tècnic
  void qui();                 // Visualitza qui ocupa cada Cabina
  ~Cabines() {                // Destructor
  delete[] com;
  delete[] tec;
  }
  };
```

Ara

```
class Cabines {      // Classe principal pels Cabines
  int maxTri;        // Nombre màxim de tripulants (de tot tipus)
  int numTri;        // Quantitatt de tripulants (de tot tipus)
  Tripulant **tri;   // Array d'apuntadors a Tripulant

  public:
  Cabines(int);               // Constructor
  int assigna(Tripulant *);   // Assigna Cabina a un Tripulant
  void qui();                 // Visualitza qui ocupa cada Cabina
  ~Cabines() {                // Destructor
    delete[] tri;
    }
  };
```

2. Programació Orientada a Objectes
2.4 Polimorfisme

Exemple. “Tripulants i Cabines” – NOVA VERSIÓ DEL CODI

Abans

```
Cabines::Cabines(int maxC, int maxT) {
  maxCom = maxC;
  maxTec = maxT;
  numCom = numTec = 0;
  com = new Comandant *[maxCom];
  tec = new Tecnic *[maxTec];
  int i;

  for (i = 0; i < maxCom; i++) com[i] = NULL;
  for (i = 0; i < maxTec; i++) tec[i] = NULL;
  }
```

Ara

```
Cabines::Cabines(int max) {
  maxTri = max;
  numTri = 0;
  tri = new Tripulant *[max];
  int i;

  for (i = 0; i < max; i++) tri[i] = NULL;
  }
```

2. Programació Orientada a Objectes
2.4 Polimorfisme

Exemple. “Tripulants i Cabines” – NOVA VERSIÓ DEL CODI

Abans

```
int Cabines::assigna(Comandant *c) {
if (numCom == maxCom) return -1;
else {
  com[numCom] = c;
  numCom++;
  return numCom;
  }
}

int Cabines::assigna(Tecnic *t) {
int i;
if (numTec == maxTec) return -1;
else {
tec[numTec] = t;
numTec++;
return numTec;
}
}
```

Ara

```
int Cabines::assigna(Tripulant *t) {
if (numTri == maxTri) return -1;
else {
  tri[numTri] = t;
  numTri++;
  return numTri;
  }
}
```

2. Programació Orientada a Objectes
2.4 Polimorfisme

Exemple. “Tripulants i Cabines” – NOVA VERSIÓ DEL CODI

Abans

```
void Cabines::qui() {
  int i;

  if (numCom>0) for (i=0;i<maxCom;i++)
  if (com[i]!=NULL) {
    cout <<"Cabina " << i << " ocupada per ";
    com[i]->meuNom();
    cout << endl;
    }

  if (numTec>0)
  for (i = 0; i<maxTec; i++)
  if (tec[i] != NULL) {
    cout << "Cabina " << i << " ocupada per ";
    tec[i]->meuNom();
    cout << endl;
    }
  }
```

Ara

```
void Cabines::qui() {
  int i;

  if (numTri>0) for (i=0;i<maxTri;i++)
  if (tri[i]!=NULL) {
    cout <<"Cabina " << i << " ocupada per ";
    tri[i]->meuNom();
    cout << endl;
    }
  }
```

2. Programació Orientada a Objectes
2.4 Polimorfisme

Exemple. "Tripulants i Cabines" – NOVA VERSIÓ DEL CODI

Abans

```
11  class Tripulant {                       // Classe principal pels Tripulants
12    protected:
13      string nom;                         // Nom del Tripulant
14    public:
15      Tripulant(string n) { nom = n; }    // Constructor
16  };
17
18  class Comandant :public Tripulant {     //Subclasse pel Tripulant de tipous Comandant
19    public:
20      Comandant(string n) :Tripulant(n) {} // Constructor
21      void meuNom() { cout << "Comandant: " << nom; }
22  };
23
24  class Tecnic :public Tripulant {        //Subclasse pel Tripulant de tipous tècnic
25    public:
26      Tecnic(string n) : Tripulant(n) {}  //Constructor
27      void meuNom() { cout << "Tecnic: " << nom; }
28  };
```

Ara

```
11  class Tripulant {                       // Classe principal pels tripula
12    protected:
13      string nom;                         // Nom del Tripulant
14    public:
15      Tripulant(string n) { nom = n; }    // Constructor
16      virtual void meuNom() { cout << "Tripulant: " << nom; }  // membre f
17  };
18
19  class Comandant :public Tripulant {     //Subclasse pel Tripulant de tip
20    public:
21      Comandant(string n) :Tripulant(n) {} // Constructor
22      void meuNom() { cout << "Comandant: " << nom; }
23  };
24
25  class Tecnic :public Tripulant {        //Subclasse pel Tripulant de tipo
26    public:
27      Tecnic(string n) : Tripulant(n) {}  //Constructor
28      void meuNom() { cout << "Tecnic: " << nom; }
29  };
```

2. Programació Orientada a Objectes
2.4 Polimorfisme

Exemple. “Tripulants i Cabines” – NOVA VERSIÓ DEL CODI

Abans

```
void main() {
Comandant c1("Joan");
Tecnic    t1("Nuria");
Comandant c2("Griselda");
Tecnic    t2("Roger");

Cabines  c(5, 5);

setlocale(LC_ALL, "");

c.assigna(&c1);
c.assigna(&t1);
c.assigna(&c2);
c.assigna(&t2);
c.qui();
system("pause");
}
```

Ara

```
void main() {
Comandant c1("Joan");
Tecnic    t1("Nuria");
Comandant c2("Griselda");
Tecnic    t2("Roger");
Tripulant tr("Maria");

Cabines c(10);

setlocale(LC_ALL, "");

c.assigna(&c1);
c.assigna(&t1);
c.assigna(&c2);
c.assigna(&t2);
c.assigna(&tr);
c.qui();
system("pause");
}
```

2. Programació Orientada a Objectes
2.4 Polimorfisme

Exemple. “Tripulants i Cabines” – NOVA VERSIÓ DEL CODI

Millores obtingudes gràcies al polimorfisme

El nombre màxim de Cabines és variable i s’ocupen només els necessaris
El constructor només té un paràmetre
El destructor es redueix a una instrucció
Només es necessita un sol mètode per assignar Cabines
Es simplifica el mètode `qui`

2. Programació Orientada a Objectes
2.4 Polimorfisme

Exemple. "Tripulants i Cabines" – Exercici: Afegir nova classe de Tripulants

Que cal fer al codi (afegir/modificar) per incloure la nova classe "Capità"?

	Abans	Ara
1. Definir la nova classe	✓	✓
2. Modificar la classe de **Cabines** *(tant els atributs com els mètodes)*	✓	✗ **NO CAL!**
3. Crear nova versió del mètode **assigna**	✓	✗ **NO CAL!**
4. Cal també reescriure el mètode **qui**	✓	✗ **NO CAL!**

2. Programació Orientada a Objectes
2.4 Polimorfisme

El polimorfisme és la manera en què els llenguatges Orientats a Objectes implementen el concepte de polisèmia del món real: Un únic nom per a molts significats, segons el context

L'ús del polimorfisme simplifica el codi i en facilita l'actualització

2. Programació Orientada a Objectes
2.4 Polimorfisme

Exemple. Nau: “Tripulants i Cabines”

Exercici

Afegir a aquesta segona versió del codi la nova classe de Tripulant: “`Capità`”

Crear un nou objecte de la classe Capità al programa principal i assignar-lo a una cabina

2. Programació Orientada a Objectes
2.4 Polimorfisme

El polimorfisme és la manera en què els llenguatges O.O. implementen el concepte de polisèmia: Un únic nom per a molts significats, segons el context

Sobrecàrrega d'operadors (overloading): En el mecanisme mitjançant el que els objectes de C++ poden actuar com operands d'una funció o expressió

Operands que poden ser usats per sobre-càrrega:

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

2. Programació Orientada a Objectes
2.4 Polimorfisme

El polimorfisme és la manera en què els llenguatges O.O. implementen el concepte de polisèmia: Un únic nom per a molts significats, segons el context

```
Date dat1{18,10,2017};
Date dat2;

dat2 = dat1+21 // afegeix 21 dies a dat1
dat2.display(); // visualitza 8/11/2017
```

Exemple 1

L'objecte *dat1* amb valor *18/10/2017* pot actuar com operador d'una expressió, utilitzant l'operand '+' sobrecarregat per a que el seu resultat retorni un altre objecte de la classe *Date*

2. Programació Orientada a Objectes
2.4 Polimorfisme

El polimorfisme és la manera en què els llenguatges O.O. implementen el concepte de polisèmia: Un únic nom per a molts significats, segons el context

```
class Comptador {
int count;

public:
Comptador(int i) {
  count = i;
  }
void Display() {
  cout << "Count: " << count<<endl;
  }
};
```

Exemple 2

Donat un objecte *cmp* de la classe *Comptador*, es vol sobreescriure l'operador ++ per a que incrementi en 1 el valor del membre de dades *count* de l'objecte.

2. Programació Orientada a Objectes
2.4 Polimorfisme

El polimorfisme és la manera en què els llenguatges O.O. implementen el concepte de polisèmia: Un únic nom per a molts significats, segons el context

```
class Comptador {
int count;

public:
Comptador(int i) {
  count = i;
  }
void Display() {
  cout << "Count: " << count<<endl;
  }
};
```

Exemple 2 (cont.)

Ús de l'operador sobrecarregat:

```
void main() {
Comptador cmp(5);
++cmp; // incrementa en un l'objecte comptador 'c'

cmp.Display();
system("pause");
}
```

2. Programació Orientada a Objectes
2.4 Polimorfisme

El polimorfisme és la manera en què els llenguatges O.O. implementen el concepte de polisèmia: Un únic nom per a molts significats, segons el context

```
class Comptador {
int count;

...

// Opció 1:
void operator++() {
  count=count+1;
  }
};
```

Exemple 2 (cont.)

Ús de l'operador sobrecarregat:

```
void main() {
Comptador cmp(5);
++cmp; // incrementa en un l'objecte comptador 'c'
       // equival a c.operator++();
c.Display();
system("pause");
}
```

2. Programació Orientada a Objectes
2.4 Polimorfisme

El polimorfisme és la manera en què els llenguatges O.O. implementen el concepte de polisèmia: Un únic nom per a molts significats, segons el context

```
class Comptador {
int count;

...

// Opció 2:
Comptador operator++() {
  count=count+1;
  return *this; }
/* El retorn és necessari si utilitzem
++cmp dins una expressió i no com
instrucció aïllada */};
```

Exemple 2 (cont.)

Ús de l'operador sobrecarregat:

```
void main() {
Comptador cmp(5);
int val;
...
val=val+(++cmp);

c.Display();}
```

2. Programació Orientada a Objectes
2.4 Polimorfisme

El polimorfisme és la manera en què els llenguatges O.O. implementen el concepte de polisèmia: Un únic nom per a molts significats, segons el context

```
class Comptador {
int count;

...

// Opció 3:(sufix). Necessari si
s'utilitza cmp++;

Comptador operator++(int) {
  Comptador cAux=*this;
  count = count + 1;
  return cAux; }};
```

Exemple 2 (cont.)

Ús de l'operador sobrecarregat:

```
void main() {
Comptador cmp(5);
cmp++; // incrementa en un l'objecte comptador 'cmp'

cmp.Display();
system("pause");
}
```

2. Programació Orientada a Objectes
2.4 Polimorfisme

El polimorfisme és la manera en què els llenguatges O.O. implementen el concepte de polisèmia: Un únic nom per a molts significats, segons el context

```
pr12001.cpp // operator +
pr12002.cpp // right and left
pr12003.cpp // operator == and <
pr12004.cpp // operator +=
pr12005.cpp // operator ++

pr12006.cpp // operator -
pPr12007.cpp // operator []
pr12008.cpp // operator ->
```

Codi de mostra

... disponible al campus

2. Programació Orientada a Objectes
2.4 Polimorfisme

Exercici: Suma de bombolles

Construir l'operador sobrecarregat suma (+) que permeti sumar objectes bombolla.

```
Bombolla b1, b2, b3;
...
b3 = b1 + b2;
```

2. Programació Orientada a Objectes
2.4 Polimorfisme

Exercici: Pokemons – dany de l'atac variable segons el tipus de pokemon

Construir membre funció polimòrfic amb comportament diferent segons el tipus de pokemon que se li passi com a paràmetre

	Electric	Grass	Fire	Water
Electric		½		2
Grass			½	2
Fire		2		½
Water		½	2