

Metodología de la Programación

Tema 1. Funciones (ampliación)

Andrés Cano Utrera
(acu@decsai.ugr.es)

Departamento de Ciencias de la Computación e I.A.



Curso 2020-2021

Contenido del tema

- 1 La función main
- 2 Paso de parámetros a funciones por valor y referencia
- 3 Paso de objetos a funciones
- 4 Referencias
- 5 Parámetros con valor por defecto
- 6 Sobrecarga de funciones
- 7 Funciones inline
- 8 Variables locales static
- 9 Funciones recursivas
 - Introducción a la recursividad
 - Ejemplos de funciones recursivas
 - Recursivo versus iterativo

Contenido del tema

- 1 La función main
- 2 Paso de parámetros a funciones por valor y referencia**
- 3 Paso de objetos a funciones
- 4 Referencias
- 5 Parámetros con valor por defecto
- 6 Sobrecarga de funciones
- 7 Funciones inline
- 8 Variables locales static
- 9 Funciones recursivas
 - Introducción a la recursividad
 - Ejemplos de funciones recursivas
 - Recursivo versus iterativo

Ejercicio:

Construir una función que intercambie el valor de dos variables

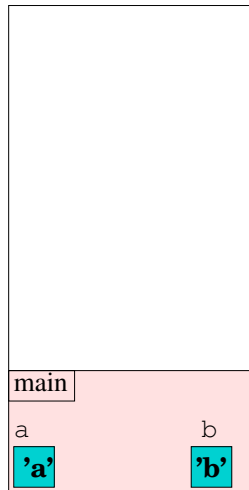
```
1 #include <iostream>
2 using namespace std;
3
4 void Swap(char c1, char c2){
5     char aux=c1;
6     c1=c2;
7     c2=aux;
8 }
9
10 int main(){
11     char a='a', b='b';
12
13     Swap(a,b);
14     cout << "a=" << a
15           << " y b=" << b << endl;
16 }
```

PILA

Ejercicio:

Construir una función que intercambie el valor de dos variables

```
1 #include <iostream>
2 using namespace std;
3
4 void Swap(char c1, char c2){
5     char aux=c1;
6     c1=c2;
7     c2=aux;
8 }
9
10 int main(){
11     char a='a', b='b';
12
13     Swap(a,b);
14     cout << "a=" << a
15           << " y b=" << b << endl;
16 }
```



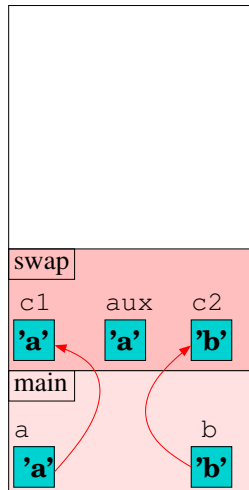
Ejercicio:

Construir una función que intercambie el valor de dos variables

```

1 #include <iostream>
2 using namespace std;
3
4 void Swap(char c1, char c2){
5     char aux=c1;
6     c1=c2;
7     c2=aux;
8 }
9
10 int main(){
11     char a='a', b='b';
12
13     Swap(a,b);
14     cout << "a=" << a
15           << " y b=" << b << endl;
16 }

```



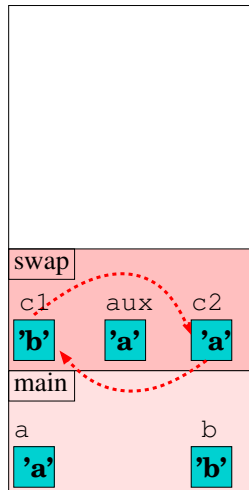
Ejercicio:

Construir una función que intercambie el valor de dos variables

```

1  #include <iostream>
2  using namespace std;
3
4  void Swap(char c1, char c2){
5      char aux=c1;
6      c1=c2;
7      c2=aux;
8  }
9
10 int main(){
11     char a='a', b='b';
12
13     Swap(a,b);
14     cout << "a=" << a
15           << " y b=" << b << endl;
16 }

```



Ejercicio:

Construir una función que intercambie el valor de dos variables

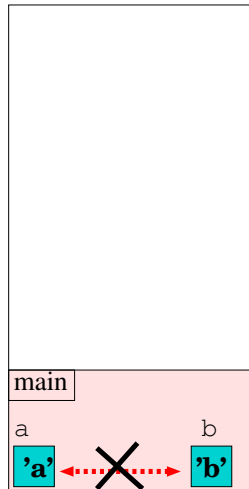
```
1 #include <iostream>
2 using namespace std;
3
4 void Swap(char c1, char c2){
5     char aux=c1;
6     c1=c2;
7     c2=aux;
8 }
9
10 int main(){
11     char a='a', b='b';
12
13     Swap(a,b);
14     cout << "a=" << a
15          << " y b=" << b << endl;
16 }
```



Ejercicio:

Construir una función que intercambie el valor de dos variables

```
1 #include <iostream>
2 using namespace std;
3
4 void Swap(char c1, char c2){
5     char aux=c1;
6     c1=c2;
7     c2=aux;
8 }
9
10 int main(){
11     char a='a', b='b';
12
13     Swap(a,b);
14     cout << "a=" << a
15          << " y b=" << b << endl;
16 }
```



Ejercicio:

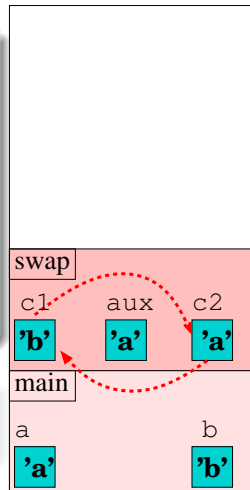
Construir una función que intercambie el valor de dos variables

Análisis

- Los valores de las variables a y b no se han modificado.
- Los que se intercambiaron fueron sus copias c1 y c2.
- El problema es que se necesita extender el ámbito de a y b para que sean manipulables en el entorno de Swap.

Solución

Paso de parámetros por referencia



Ejercicio:

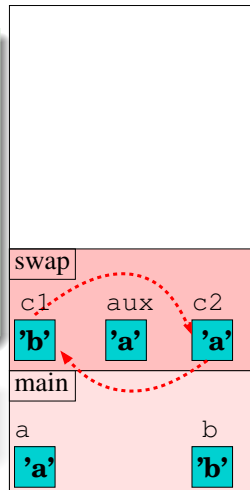
Construir una función que intercambie el valor de dos variables

Análisis

- Los valores de las variables a y b no se han modificado.
- Los que se intercambiaron fueron sus copias c1 y c2.
- El problema es que se necesita extender el ámbito de a y b para que sean manipulables en el entorno de Swap.

Solución

Paso de parámetros por referencia



Ejercicio:

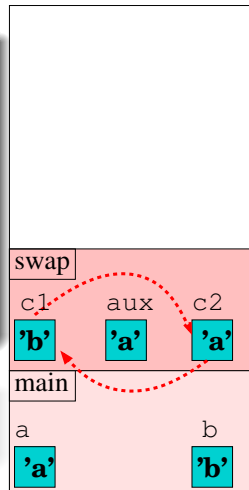
Construir una función que intercambie el valor de dos variables

Análisis

- Los valores de las variables a y b no se han modificado.
- Los que se intercambiaron fueron sus copias c1 y c2.
- El problema es que se necesita extender el ámbito de a y b para que sean manipulables en el entorno de Swap.

Solución

Paso de parámetros por referencia



Ejercicio:

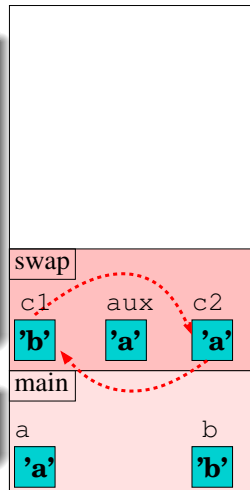
Construir una función que intercambie el valor de dos variables

Análisis

- Los valores de las variables a y b no se han modificado.
- Los que se intercambiaron fueron sus copias c1 y c2.
- El problema es que se necesita extender el ámbito de a y b para que sean manipulables en el entorno de Swap.

Solución

Paso de parámetros por referencia



Ejercicio:

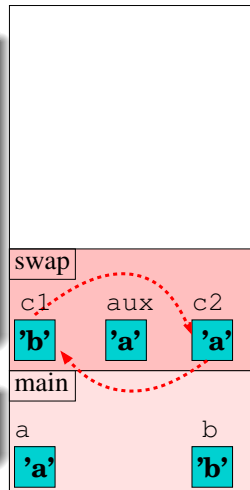
Construir una función que intercambie el valor de dos variables

Análisis

- Los valores de las variables a y b no se han modificado.
- Los que se intercambiaron fueron sus copias c1 y c2.
- El problema es que se necesita extender el ámbito de a y b para que sean manipulables en el entorno de Swap.

Solución

Paso de parámetros por referencia



Paso de parámetros

Por valor o copia

- Es el paso de argumentos por defecto.
- Durante la llamada se realiza una copia del parámetro actual en el parámetro formal.
- De esta forma, el módulo invocado trabaja con una copia y no con el valor original.

Para resolver el problema del ejercicio anterior tenemos que trabajar con los datos originales y no con las copias.

Paso de parámetros

Por valor o copia

- Es el paso de argumentos por defecto.
- Durante la llamada se realiza una copia del parámetro actual en el parámetro formal.
- De esta forma, el módulo invocado trabaja con una copia y no con el valor original.

Para resolver el problema del ejercicio anterior tenemos que trabajar con los datos originales y no con las copias.

Paso de parámetros

Por valor o copia

- Es el paso de argumentos por defecto.
- Durante la llamada se realiza una copia del parámetro actual en el parámetro formal.
- De esta forma, el módulo invocado trabaja con una copia y no con el valor original.

Para resolver el problema del ejercicio anterior tenemos que trabajar con los datos originales y no con las copias.

Paso de parámetros

Por valor o copia

- Es el paso de argumentos por defecto.
- Durante la llamada se realiza una copia del parámetro actual en el parámetro formal.
- De esta forma, el módulo invocado trabaja con una copia y no con el valor original.

Para resolver el problema del ejercicio anterior tenemos que trabajar con los datos originales y no con las copias.

Paso de parámetros

Por referencia o variable

- No realiza una copia del parámetro actual en el formal, sino un vínculo entre ellos, de tal forma que una modificación en el parámetro formal, conlleva la misma modificación en el parámetro actual.
- Se usa `&` entre el tipo y el identificador del argumento para indicar que el paso se realiza por referencia.

Ejemplos

```
1 void Swap (char &c1, char &c2);  
2 void Division (int dividendo, int divisor,  
3               int &coc, int &resto);  
4 void ElegirOpcion (char &opcion);
```

Paso de parámetros

Por referencia o variable

- No realiza una copia del parámetro actual en el formal, sino un vínculo entre ellos, de tal forma que una modificación en el parámetro formal, conlleva la misma modificación en el parámetro actual.
- Se usa **&** entre el tipo y el identificador del argumento para indicar que el paso se realiza por referencia.

Ejemplos

```
1 void Swap (char &c1, char &c2);  
2 void Division (int dividendo, int divisor,  
3               int &coc, int &resto);  
4 void ElegirOpcion (char &opcion);
```

Paso de parámetros

Por referencia o variable

- No realiza una copia del parámetro actual en el formal, sino un vínculo entre ellos, de tal forma que una modificación en el parámetro formal, conlleva la misma modificación en el parámetro actual.
- Se usa **&** entre el tipo y el identificador del argumento para indicar que el paso se realiza por referencia.

Ejemplos

```
1 void Swap (char &c1, char &c2);  
2 void Division (int dividendo, int divisor,  
3               int &coc, int &resto);  
4 void ElegirOpcion (char &opcion);
```

Solución del ejercicio

Construir una función que intercambie el valor de dos variables

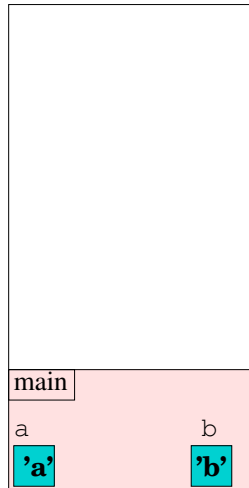
```
1 #include <iostream>
2 using namespace std;
3
4 void Swap(char &c1, char &c2){
5     char aux=c1;
6     c1=c2;
7     c2=aux;
8 }
9
10 int main(){
11     char a='a', b='b';
12
13     Swap(a,b);
14     cout << "a=" << a
15          << " y b=" << b << endl;
16 }
```

PILA

Solución del ejercicio

Construir una función que intercambie el valor de dos variables

```
1 #include <iostream>
2 using namespace std;
3
4 void Swap(char &c1, char &c2){
5     char aux=c1;
6     c1=c2;
7     c2=aux;
8 }
9
10 int main(){
11     char a='a', b='b';
12
13     Swap(a,b);
14     cout << "a=" << a
15          << " y b=" << b << endl;
16 }
```



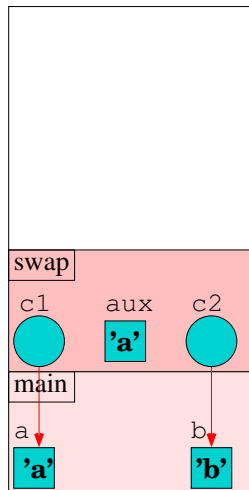
Solución del ejercicio

Construir una función que intercambie el valor de dos variables

```

1  #include <iostream>
2  using namespace std;
3
4  void Swap(char &c1, char &c2){
5      char aux=c1;
6      c1=c2;
7      c2=aux;
8  }
9
10 int main(){
11     char a='a', b='b';
12
13     Swap(a,b);
14     cout << "a=" << a
15          << " y b=" << b << endl;
16 }

```



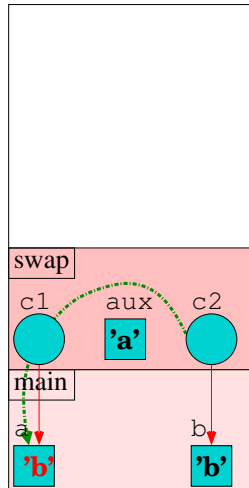
Solución del ejercicio

Construir una función que intercambie el valor de dos variables

```

1  #include <iostream>
2  using namespace std;
3
4  void Swap(char &c1, char &c2){
5      char aux=c1;
6      c1=c2;
7      c2=aux;
8  }
9
10 int main(){
11     char a='a', b='b';
12
13     Swap(a,b);
14     cout << "a=" << a
15          << " y b=" << b << endl;
16 }

```



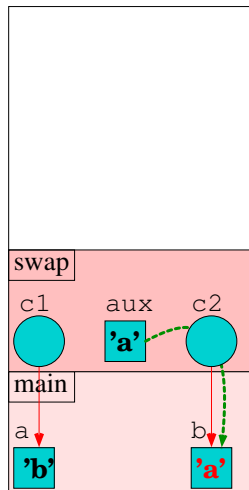
Solución del ejercicio

Construir una función que intercambie el valor de dos variables

```

1  #include <iostream>
2  using namespace std;
3
4  void Swap(char &c1, char &c2){
5      char aux=c1;
6      c1=c2;
7      c2=aux;
8  }
9
10 int main(){
11     char a='a', b='b';
12
13     Swap(a,b);
14     cout << "a=" << a
15          << " y b=" << b << endl;
16 }

```



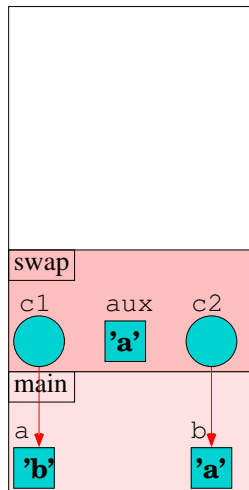
Solución del ejercicio

Construir una función que intercambie el valor de dos variables

```

1  #include <iostream>
2  using namespace std;
3
4  void Swap(char &c1, char &c2){
5      char aux=c1;
6      c1=c2;
7      c2=aux;
8  }
9
10 int main(){
11     char a='a', b='b';
12
13     Swap(a,b);
14     cout << "a=" << a
15           << " y b=" << b << endl;
16 }

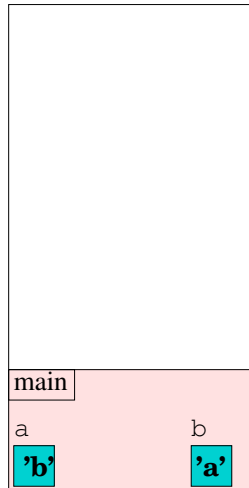
```



Solución del ejercicio

Construir una función que intercambie el valor de dos variables

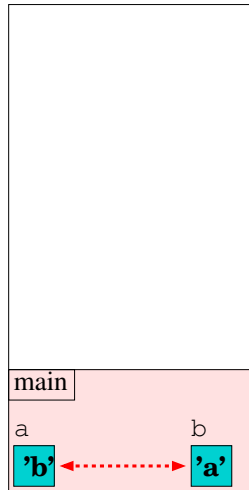
```
1 #include <iostream>
2 using namespace std;
3
4 void Swap(char &c1, char &c2){
5     char aux=c1;
6     c1=c2;
7     c2=aux;
8 }
9
10 int main(){
11     char a='a', b='b';
12
13     Swap(a,b);
14     cout << "a=" << a
15          << " y b=" << b << endl;
16 }
```



Solución del ejercicio

Construir una función que intercambie el valor de dos variables

```
1 #include <iostream>
2 using namespace std;
3
4 void Swap(char &c1, char &c2){
5     char aux=c1;
6     c1=c2;
7     c2=aux;
8 }
9
10 int main(){
11     char a='a', b='b';
12
13     Swap(a,b);
14     cout << "a=" << a
15          << " y b=" << b << endl;
16 }
```



Valor/Referencia versus Entrada/Salida

La identificación de la información que aporta un argumento en una función nos indica la forma en la que debe ser pasado dicho argumento.

- Si el argumento es usado como vehículo para obtener la solución, entonces nos encontramos ante un **parámetro de entrada**.

PASO POR VALOR

- Si el argumento es usado para almacenar la solución o parte de ella, entonces nos encontramos ante un **parámetro de salida**.
- Si el argumento es tanto vehículo para obtener la solución como parte de la misma, entonces nos encontramos con un **parámetro de entrada/salida**.

PASO POR REFERENCIA

Valor/Referencia versus Entrada/Salida

La identificación de la información que aporta un argumento en una función nos indica la forma en la que debe ser pasado dicho argumento.

- Si el argumento es usado como vehículo para obtener la solución, entonces nos encontramos ante un **parámetro de entrada**.

PASO POR VALOR

- Si el argumento es usado para almacenar la solución o parte de ella, entonces nos encontramos ante un **parámetro de salida**.
- Si el argumento es tanto vehículo para obtener la solución como parte de la misma, entonces nos encontramos con un **parámetro de entrada/salida**.

PASO POR REFERENCIA

Valor/Referencia versus Entrada/Salida

La identificación de la información que aporta un argumento en una función nos indica la forma en la que debe ser pasado dicho argumento.

- Si el argumento es usado como vehículo para obtener la solución, entonces nos encontramos ante un **parámetro de entrada**.

PASO POR VALOR

- Si el argumento es usado para almacenar la solución o parte de ella, entonces nos encontramos ante un **parámetro de salida**.
- Si el argumento es tanto vehículo para obtener la solución como parte de la misma, entonces nos encontramos con un **parámetro de entrada/salida**.

PASO POR REFERENCIA

Valor/Referencia versus Entrada/Salida

La identificación de la información que aporta un argumento en una función nos indica la forma en la que debe ser pasado dicho argumento.

- Si el argumento es usado como vehículo para obtener la solución, entonces nos encontramos ante un **parámetro de entrada**.

PASO POR VALOR

- Si el argumento es usado para almacenar la solución o parte de ella, entonces nos encontramos ante un **parámetro de salida**.
- Si el argumento es tanto vehículo para obtener la solución como parte de la misma, entonces nos encontramos con un **parámetro de entrada/salida**.

PASO POR REFERENCIA

Valor/Referencia versus Entrada/Salida

La identificación de la información que aporta un argumento en una función nos indica la forma en la que debe ser pasado dicho argumento.

- Si el argumento es usado como vehículo para obtener la solución, entonces nos encontramos ante un **parámetro de entrada**.

PASO POR VALOR

- Si el argumento es usado para almacenar la solución o parte de ella, entonces nos encontramos ante un **parámetro de salida**.
- Si el argumento es tanto vehículo para obtener la solución como parte de la misma, entonces nos encontramos con un **parámetro de entrada/salida**.

PASO POR REFERENCIA

Valor/Referencia versus Entrada/Salida

La identificación de la información que aporta un argumento en una función nos indica la forma en la que debe ser pasado dicho argumento.

- Si el argumento es usado como vehículo para obtener la solución, entonces nos encontramos ante un **parámetro de entrada**.

PASO POR VALOR

- Si el argumento es usado para almacenar la solución o parte de ella, entonces nos encontramos ante un **parámetro de salida**.
- Si el argumento es tanto vehículo para obtener la solución como parte de la misma, entonces nos encontramos con un **parámetro de entrada/salida**.

PASO POR REFERENCIA

Valor/Referencia versus Entrada/Salida

La identificación de la información que aporta un argumento en una función nos indica la forma en la que debe ser pasado dicho argumento.

- Si el argumento es usado como vehículo para obtener la solución, entonces nos encontramos ante un **parámetro de entrada**.

PASO POR VALOR

- Si el argumento es usado para almacenar la solución o parte de ella, entonces nos encontramos ante un **parámetro de salida**.
- Si el argumento es tanto vehículo para obtener la solución como parte de la misma, entonces nos encontramos con un **parámetro de entrada/salida**.

PASO POR REFERENCIA

Valor/Referencia versus Entrada/Salida

Debes tener en cuenta que

cuando el paso de parámetros es por valor, el argumento actual puede ser una expresión, una constante o una variable.

Sin embargo

cuando el paso de parámetros es por referencia, el argumento actual debe ser obligatoriamente una variable.

Valor/Referencia versus Entrada/Salida

Debes tener en cuenta que

cuando el paso de parámetros es por valor, el argumento actual puede ser una expresión, una constante o una variable.

Sin embargo

cuando el paso de parámetros es por referencia, el argumento actual debe ser obligatoriamente una variable.

Contenido del tema

- 1 La función main
- 2 Paso de parámetros a funciones por valor y referencia
- 3 Paso de objetos a funciones**
- 4 Referencias
- 5 Parámetros con valor por defecto
- 6 Sobrecarga de funciones
- 7 Funciones inline
- 8 Variables locales static
- 9 Funciones recursivas
 - Introducción a la recursividad
 - Ejemplos de funciones recursivas
 - Recursivo versus iterativo

Paso de objetos a funciones

Los objetos (clases y estructuras) se comportan en C++ como si fueran tipos de datos básicos cuando se utilizan como argumento de las funciones:

- Se pueden pasar **por valor**:

```
double calculaDistancia(Punto punto1,Punto punto2);
```

- Se pueden pasar **por referencia**:

```
void leerPunto(Punto &punto);
```

- Se pueden pasar **por referencia constante**:

```
double calculaDistancia(const Punto &punto1,const Punto  
&punto2);
```

- Las funciones pueden **devolver objetos**:

```
Punto puntoMedio(const Punto &punto1,const Punto &punto2);
```


Diferencia entre paso por referencia constante y paso por valor

- Paso por valor:

```
void imprimePunto(Punto punto){
    ....
    punto.x=5.2; //Permitido: se modifica punto que es una copia
```

- Paso por referencia constante:

```
void imprimePunto(const Punto &punto){
    ....
    punto.x=5.2; //NO Permitido
```

El paso por referencia constante, pasa una referencia sobre el dato con el que se quiere trabajar (con lo que evitamos una copia que puede ocupar mucha memoria) pero lo protege para que no se pueda modificar el dato original.

Contenido del tema

- 1 La función main
- 2 Paso de parámetros a funciones por valor y referencia
- 3 Paso de objetos a funciones
- 4 Referencias**
- 5 Parámetros con valor por defecto
- 6 Sobrecarga de funciones
- 7 Funciones inline
- 8 Variables locales static
- 9 Funciones recursivas
 - Introducción a la recursividad
 - Ejemplos de funciones recursivas
 - Recursivo versus iterativo

Referencias

Referencia

Es una especie de alias a otro dato u objeto. Se usa en:

- Paso de parámetros por referencia en una función o método
- Referencias como alias a otras variables
- Devolución por referencia desde una función

Referencias como alias a otras variables

Referencias como alias a otras variables

Una variable referencia es un alias a otra variable:

```
<tipo> & <identificador> = <iniciador> ;
```

Las variables referencia deben **inicializarse** en su declaración y **no pueden reasignarse** como alias a otras variables.

- **Ejemplo 1:**

```
int a=0;
int &ref=a;
ref=5;
cout<<a<<endl;
```

- **Ejemplo 2:**

```
int v[5]={1,2,3,4,5};
int &ref=v[3];
ref=0;
cout<<v[3]<<endl;
```

Referencias como alias a otras variables

Referencias como alias a otras variables

Una variable referencia es un alias a otra variable:

`<tipo> & <identificador> = <iniciador> ;`

Las variables referencia deben **inicializarse** en su declaración y **no pueden reasignarse** como alias a otras variables.

- **Ejemplo 1:**

```
int a=0;
int &ref=a;
ref=5;
cout<<a<<endl;
```

- **Ejemplo 2:**

```
int v[5]={1,2,3,4,5};
int &ref=v[3];
ref=0;
cout<<v[3]<<endl;
```

Referencias como alias a otras variables

Referencias como alias a otras variables

Una variable referencia es un alias a otra variable:

`<tipo> & <identificador> = <iniciador> ;`

Las variables referencia deben **inicializarse** en su declaración y **no pueden reasignarse** como alias a otras variables.

- **Ejemplo 1:**

```
int a=0;
int &ref=a;
ref=5;
cout<<a<<endl;
```

- **Ejemplo 2:**

```
int v[5]={1,2,3,4,5};
int &ref=v[3];
ref=0;
cout<<v[3]<<endl;
```

Paso de parámetros por referencia en una función o método

Por referencia o variable

- Debe usarse un **lvalue** en el parámetro actual (en la llamada a la función o método).
- No realiza una copia del parámetro actual en el formal, sino un vínculo entre ellos.
- Una modificación en el parámetro formal, conlleva la misma modificación en el parámetro actual.

Paso de parámetros por referencia en una función o método

Por referencia o variable

- Debe usarse un **lvalue** en el parámetro actual (en la llamada a la función o método).
- No realiza una copia del parámetro actual en el formal, sino un vínculo entre ellos.
- Una modificación en el parámetro formal, conlleva la misma modificación en el parámetro actual.

Paso de parámetros por referencia en una función o método

Por referencia o variable

- Debe usarse un **lvalue** en el parámetro actual (en la llamada a la función o método).
- No realiza una copia del parámetro actual en el formal, sino un vínculo entre ellos.
- Una modificación en el parámetro formal, conlleva la misma modificación en el parámetro actual.

Paso de parámetros por referencia en una función o método

Ejemplo:

Función que intercambia el valor de dos variables

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

void Swap(char &c1, char &c2){
    char aux=c1;
    c1=c2;
    c2=aux;
}

int main(){
    char a='a', b='b';

    Swap(a,b);
    cout << "a=" << a
         << " y b=" << b << endl;
}
```

Paso de parámetros por referencia constante

Paso por referencia constante

Habitualmente se usa para pasar objetos de gran tamaño que no van a modificarse en la función o método

```
double calcularMedia(const VectorSD& v){
    double suma = 0.0;
    for(int i=0; i< v.nElementos(); i++){
        suma += v.getDatos(i);
    }
    return suma/v.nElementos();
}

int main(){
    VectorSD miVector;
    for(int i = 0; i < 1000000; i++)
        miVector.aniadir(uniforme(0,50));
    cout << calcularMedia(miVector);
}
```

Paso de parámetros por referencia en una función o método

Paso por referencia constante: ¿se puede llamar con un rvalue?

Podemos usar un lvalue y también una expresión (un **rvalue**) para llamar a una función o método que espera un argumento por referencia constante.

```
void mostrar(const double& dato){  
    cout << "Dato: " << dato << endl;  
}  
  
int main(){  
    int a = 3.0;  
    mostrar(a);    // Llamada con un lvalue  
    mostrar(a+2);  // ¿Es válida esta llamada?  
}
```

Llamada a funciones o métodos con lvalues o rvalues

Llamada a funciones con parámetros actuales lvalue o rvalue

Según sea el parámetro formal, podremos llamar a la función o método con parámetros actuales lvalue o rvalue.

- **Paso por valor:** argumento actual puede ser una expresión, una constante o una variable.
- Paso por referencia: argumento actual solo puede ser un lvalue.
- Paso por referencia constante: argumento actual puede ser una expresión, una constante o una variable.

Llamada a funciones o métodos con lvalues o rvalues

Llamada a funciones con parámetros actuales lvalue o rvalue

Según sea el parámetro formal, podremos llamar a la función o método con parámetros actuales lvalue o rvalue.

- **Paso por valor:** argumento actual puede ser una expresión, una constante o una variable.
- **Paso por referencia:** argumento actual solo puede ser un lvalue.
- **Paso por referencia constante:** argumento actual puede ser una expresión, una constante o una variable.

Llamada a funciones o métodos con lvalues o rvalues

Llamada a funciones con parámetros actuales lvalue o rvalue

Según sea el parámetro formal, podremos llamar a la función o método con parámetros actuales lvalue o rvalue.

- **Paso por valor:** argumento actual puede ser una expresión, una constante o una variable.
- **Paso por referencia:** argumento actual solo puede ser un lvalue.
- **Paso por referencia constante:** argumento actual puede ser una expresión, una constante o una variable.

Devolución por referencia

Función con devolución por referencia

Una función puede devolver una referencia a un dato u objeto

```
int& valor(int v[], int i){  
    return v[i];  
}
```

La referencia puede usarse en el lado derecho de una asignación

```
int main(){  
    int v[]={3,5,2,7,6};  
    int a=valor(v,3);  
}
```

Pero también en el lado izquierdo de la asignación

```
int main(){  
    int v[]={3,5,2,7,6};  
    valor(v,3)=0;  
}
```


Devolución por referencia

Función con devolución por referencia

Una función puede devolver una referencia a un dato u objeto

```
int& valor(int v[], int i){  
    return v[i];  
}
```

La referencia puede usarse en el lado derecho de una asignación

```
int main(){  
    int v[]={3,5,2,7,6};  
    int a=valor(v,3);  
}
```

Pero también en el lado izquierdo de la asignación

```
int main(){  
    int v[]={3,5,2,7,6};  
    valor(v,3)=0;  
}
```

Devolución por referencia

Función con devolución por referencia

Una función puede devolver una referencia a un dato u objeto

```
int& valor(int v[], int i){  
    return v[i];  
}
```

La referencia puede usarse en el lado derecho de una asignación

```
int main(){  
    int v[]={3,5,2,7,6};  
    int a=valor(v,3);  
}
```

Pero también en el lado izquierdo de la asignación

```
int main(){  
    int v[]={3,5,2,7,6};  
    valor(v,3)=0;  
}
```

Devolución por referencia

Devolución de referencias a datos locales

La devolución de referencias a datos locales a una función es un error típico: Los datos locales se destruyen al terminar la función.

```
#include <iostream>
using namespace std;
int& funcion()
{
    int x=3;
    return x; //Aviso al compilar: devolución referencia a variable local
}
int main()
{
    int y=funcion(); // Error de ejecución
    cout << y << endl;
}
```

Devolución por referencia

Devolución de referencia constante

Una función puede devolver una referencia constante: significa que el dato referenciado es constante.

```
const int &valor(const int v[], int i){
    return v[i];
}

int main(){
    int v[3]={0,1,2};
    v[2]=3*5; // Correcto
    valor(v,2)=3*5 // Error compilación, pues la referencia es const
    int res=valor(v,2)*3; // Correcto
}
```

Devolución por referencia

Devolución de puntero constante

Lo mismo ocurre cuando una función devuelve un puntero: podemos hacer que éste sea `const`: significa que el dato apuntado es constante.

```
const int *valor(int *v, int i){
    return v+i;
}

int main(){
    int v[3];
    v[2]=3*5; // Correcto
    *(valor(v,2))=3*5; // Error, pues el puntero devuelto es const
    int res=*(valor(v,2))*3; // Correcto
}
```

Contenido del tema

- 1 La función main
- 2 Paso de parámetros a funciones por valor y referencia
- 3 Paso de objetos a funciones
- 4 Referencias
- 5 Parámetros con valor por defecto**
- 6 Sobrecarga de funciones
- 7 Funciones inline
- 8 Variables locales static
- 9 Funciones recursivas
 - Introducción a la recursividad
 - Ejemplos de funciones recursivas
 - Recursivo versus iterativo

Parámetros con valor por defecto

Parámetros con valor por defecto

Una función o método puede tener parámetros con un valor por defecto

- Deben ser los últimos de la función.
- En la llamada a la función, si solo se especifican un subconjunto de ellos, deben ser los primeros.

```
void funcion(char c, int i=7){  
    ...  
}  
  
int main(){  
    funcion('a',8);  
    funcion('z');  
}
```

Parámetros con valor por defecto

Parámetros con valor por defecto

Una función o método puede tener parámetros con un valor por defecto

- Deben ser los últimos de la función.
- En la llamada a la función, si solo se especifican un subconjunto de ellos, deben ser los primeros.

```
void funcion(char c, int i=7){  
    ...  
}  
  
int main(){  
    funcion('a',8);  
    funcion('z');  
}
```


Parámetros con valor por defecto: Ejemplo

```
#include <iostream>
using namespace std;
int volumenCaja(int largo=1, int ancho=1, int alto=1);
int main()
{
    cout << "Volumen por defecto: " << volumenCaja() << endl;
    cout << "El volumen de una caja (10,1,1) es: " <<
        volumenCaja(10) << endl;
    cout << "El volumen de una caja (10,5,1) es: " <<
        volumenCaja(10,5) << endl;
    cout << "El volumen de una caja (10,5,2) es: " <<
        volumenCaja(10,5,2) << endl;
    return 0;
}
int volumenCaja( int largo, int ancho, int alto )
{
    return largo * ancho * alto;
}
```

Metodología de la Programación

Tema 3. Punteros y memoria dinámica

Andrés Cano Utrera
(acu@decsai.ugr.es)

Departamento de Ciencias de la Computación e I.A.



Curso 2020-2021

Parte I

Tipo de Dato Puntero

Contenido del tema

1 Definición y Declaración de variables

2 Operaciones con punteros

- Operador de dirección &
- Operador de indirección *
- Asignación e inicialización de punteros
- Operadores relacionales
- Operadores aritméticos

3 Punteros y arrays

4 Punteros y cadenas

5 Punteros, struct y class

6 Punteros y funciones

7 Punteros a punteros

8 Punteros y const

9 Arrays de punteros

10 Punteros a funciones

11 Errores comunes con punteros

12 Estructura de la memoria

13 Gestión dinámica de la memoria

14 Objetos Dinámicos Simples

15 Objetos dinámicos compuestos

16 Arrays dinámicos

- Arrays dinámicos de datos de tipo primitivo
- Arrays dinámicos de objetos

17 Clases que contienen datos en memoria dinámica

18 Matrices dinámicas

19 Lista de celdas enlazadas

Definición de una variable tipo puntero

Tipo de dato puntero

Tipo de dato que contiene la dirección de memoria de otro dato.

- Incluye una dirección especial llamada *dirección nula* que es el valor 0.
- En C esta dirección nula se suele representar por la constante NULL (definida en `stdlib.h` en C o en `cstdlib` en C++).

Sintaxis

```
<tipo> *<identificador>;
```

- <tipo> es el tipo de dato cuya dirección de memoria contiene <identificador>
- <identificador> es el nombre de la variable puntero.

Definición de una variable tipo puntero

Tipo de dato puntero

Tipo de dato que contiene la dirección de memoria de otro dato.

- Incluye una dirección especial llamada *dirección nula* que es el valor 0.
- En C esta dirección nula se suele representar por la constante NULL (definida en `stdlib.h` en C o en `cstdlib` en C++).

Sintaxis

`<tipo> *<identificador>;`

- `<tipo>` es el tipo de dato cuya dirección de memoria contiene `<identificador>`
- `<identificador>` es el nombre de la variable puntero.

Ejemplo: Declaración de punteros

```
1
2 .....
3
4 // Se declara variable de tipo entero
5 int i=5;
6
7 // Se declara variable de tipo char
8 char c='a';
9
10 // Se declara puntero a entero
11 int * ptri;
12
13 // Se declara puntero a char
14 char * ptrc;
15
16 .....
17
```

Ejemplo: Declaración de punteros

1012	
1011	
1010	
1009	
1008	
1007	
1006	
1005	
1004	
1003	
1002	
1001	

// Se declara la variable de tipo entero

```
int i=5;
```

// Se declara la variable de tipo char

```
char c='a';
```

// Se declara puntero a entero

```
int * ptri;
```

// Se declara el puntero a char

```
char * ptrc;
```


Ejemplo: Declaración de punteros

1012	
1011	
1010	
1009	
1008	
1007	
1006	
1005	
1004	
1003	
1002	
1001	

// Se declara la variable de tipo entero

```
int i=5;
```

// Se declara la variable de tipo char

```
char c='a';
```

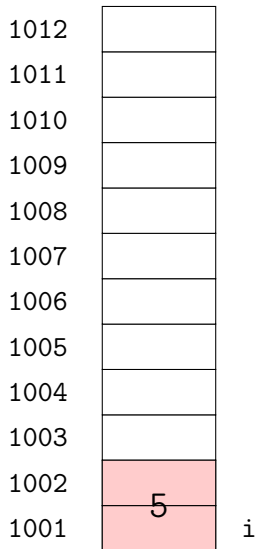
// Se declara puntero a entero

```
int * ptri;
```

// Se declara el puntero a char

```
char * ptrc;
```

Ejemplo: Declaración de punteros



// Se declara la variable de tipo entero

```
int i=5;
```

// Se declara la variable de tipo char

```
char c='a';
```

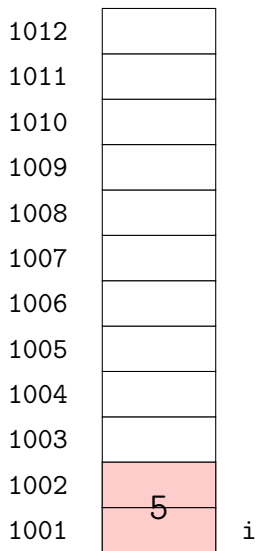
// Se declara puntero a entero

```
int * ptri;
```

// Se declara el puntero a char

```
char * ptrc;
```

Ejemplo: Declaración de punteros



```
// Se declara la variable de tipo entero
```

```
int i=5;
```

```
// Se declara la variable de tipo char
```

```
char c='a';
```

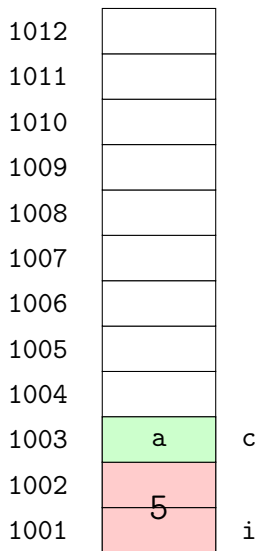
```
// Se declara puntero a entero
```

```
int * ptri;
```

```
// Se declara el puntero a char
```

```
char * ptrc;
```

Ejemplo: Declaración de punteros



// Se declara la variable de tipo entero

```
int i=5;
```

// Se declara la variable de tipo char

```
char c='a';
```

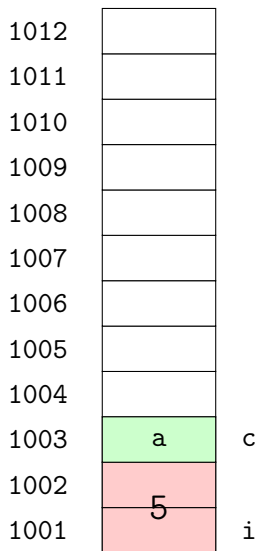
// Se declara puntero a entero

```
int * ptri;
```

// Se declara el puntero a char

```
char * ptrc;
```

Ejemplo: Declaración de punteros



// Se declara la variable de tipo entero

```
int i=5;
```

// Se declara la variable de tipo char

```
char c='a';
```

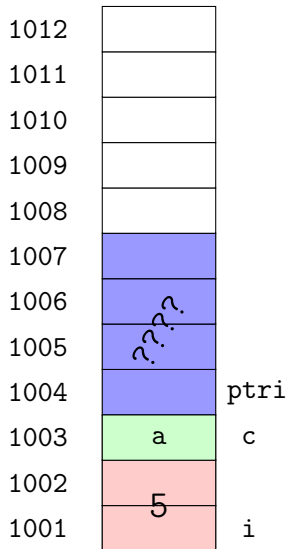
// Se declara puntero a entero

```
int * ptri;
```

// Se declara el puntero a char

```
char * ptrc;
```

Ejemplo: Declaración de punteros



```
// Se declara la variable de tipo entero
```

```
int i=5;
```

```
// Se declara la variable de tipo char
```

```
char c='a';
```

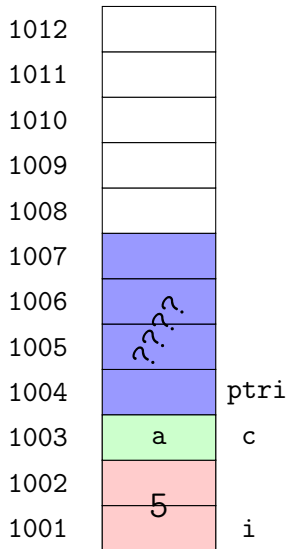
```
// Se declara puntero a entero
```

```
int * ptri;
```

```
// Se declara el puntero a char
```

```
char * ptrc;
```

Ejemplo: Declaración de punteros



```
// Se declara la variable de tipo entero
```

```
int i=5;
```

```
// Se declara la variable de tipo char
```

```
char c='a';
```

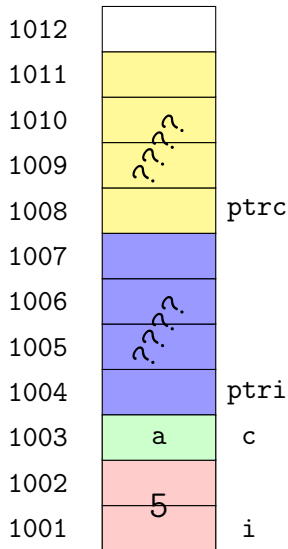
```
// Se declara puntero a entero
```

```
int * ptri;
```

```
// Se declara el puntero a char
```

```
char * ptrc;
```

Ejemplo: Declaración de punteros



```
// Se declara la variable de tipo entero
```

```
int i=5;
```

```
// Se declara la variable de tipo char
```

```
char c='a';
```

```
// Se declara puntero a entero
```

```
int * ptri;
```

```
// Se declara el puntero a char
```

```
char * ptrc;
```


Se dice que

- ptri es un *puntero a enteros*
- ptrc es un *puntero a caracteres*.

¡Nota!

Cuando se declara un puntero se reserva memoria para albergar la dirección de memoria de un dato, no el dato en sí.

¡Nota!

El tamaño de memoria reservado para albergar un puntero es el mismo independientemente del tipo de dato al que 'apunte' (será el espacio necesario para albergar una dirección de memoria, 32 ó 64 bits, dependiendo del tipo de procesador usado).

Se dice que

- ptri es un *puntero a enteros*
- ptrc es un *puntero a caracteres*.

¡Nota!

Cuando se declara un puntero se reserva memoria para albergar la dirección de memoria de un dato, no el dato en sí.

¡Nota!

El tamaño de memoria reservado para albergar un puntero es el mismo independientemente del tipo de dato al que 'apunte' (será el espacio necesario para albergar una dirección de memoria, 32 ó 64 bits, dependiendo del tipo de procesador usado).

Contenido del tema

- 1 Definición y Declaración de variables
- 2 Operaciones con punteros
 - Operador de dirección &
 - Operador de indirección *
 - Asignación e inicialización de punteros
 - Operadores relacionales
 - Operadores aritméticos
- 3 Punteros y arrays
- 4 Punteros y cadenas
- 5 Punteros, struct y class
- 6 Punteros y funciones
- 7 Punteros a punteros
- 8 Punteros y const
- 9 Arrays de punteros
- 10 Punteros a funciones
- 11 Errores comunes con punteros
- 12 Estructura de la memoria
- 13 Gestión dinámica de la memoria
- 14 Objetos Dinámicos Simples
- 15 Objetos dinámicos compuestos
- 16 Arrays dinámicos
 - Arrays dinámicos de datos de tipo primitivo
 - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas

Contenido del tema

- 1 Definición y Declaración de variables
- 2 Operaciones con punteros
 - Operador de dirección &
 - Operador de indirección *
 - Asignación e inicialización de punteros
 - Operadores relacionales
 - Operadores aritméticos
- 3 Punteros y arrays
- 4 Punteros y cadenas
- 5 Punteros, struct y class
- 6 Punteros y funciones
- 7 Punteros a punteros
- 8 Punteros y const
- 9 Arrays de punteros
- 10 Punteros a funciones
- 11 Errores comunes con punteros
- 12 Estructura de la memoria
- 13 Gestión dinámica de la memoria
- 14 Objetos Dinámicos Simples
- 15 Objetos dinámicos compuestos
- 16 Arrays dinámicos
 - Arrays dinámicos de datos de tipo primitivo
 - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas

Operador de dirección &

Operador de dirección &

`&<var>` devuelve la dirección de la variable `<var>` (o sea, un puntero).

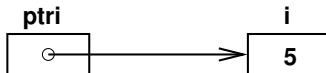
Operador de dirección &

Operador de dirección &

&<var> devuelve la dirección de la variable <var> (o sea, un puntero).

- El operador & se utiliza habitualmente para asignar valores a datos de tipo puntero.

```
int i = 5, *ptri;  
ptri = &i;
```



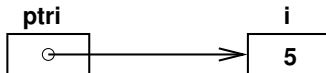
Operador de dirección &

Operador de dirección &

&<var> devuelve la dirección de la variable <var> (o sea, un puntero).

- El operador & se utiliza habitualmente para asignar valores a datos de tipo puntero.

```
int i = 5, *ptri;  
ptri = &i;
```



- i es una variable de tipo entero, por lo que la expresión &i es la dirección de memoria donde comienza un entero y, por tanto, puede ser asignada al puntero ptri.

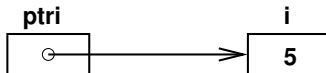
Operador de dirección &

Operador de dirección &

`&<var>` devuelve la dirección de la variable `<var>` (o sea, un puntero).

- El operador `&` se utiliza habitualmente para asignar valores a datos de tipo puntero.

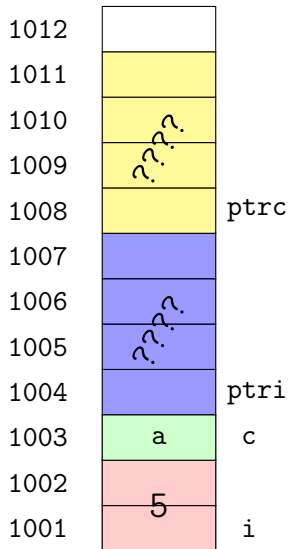
```
int i = 5, *ptri;  
ptri = &i;
```



- `i` es una variable de tipo entero, por lo que la expresión `&i` es la dirección de memoria donde comienza un entero y, por tanto, puede ser asignada al puntero `ptri`.

Se dice que `ptri` *apunta* o *referencia* a `i`.

Operador de dirección &



```
// Se declara la variable de tipo entero
```

```
int i=5;
```

```
// Se declara la variable de tipo char
```

```
char c='a';
```

```
// Se declara puntero a entero
```

```
int * ptri;
```

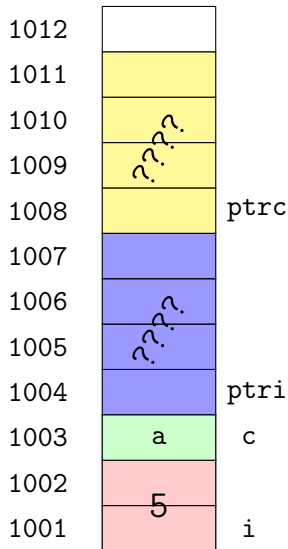
```
// Se declara el puntero a char
```

```
char * ptrc;
```

```
// ptri apunta a la variable i
```

```
ptri=&i;
```

Operador de dirección &



```
// Se declara la variable de tipo entero
```

```
int i=5;
```

```
// Se declara la variable de tipo char
```

```
char c='a';
```

```
// Se declara puntero a entero
```

```
int * ptri;
```

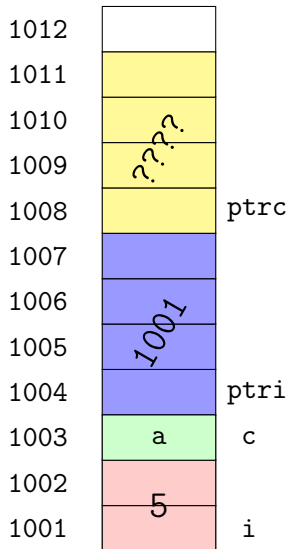
```
// Se declara el puntero a char
```

```
char * ptrc;
```

```
// ptri apunta a la variable i
```

```
ptri=&i;
```

Operador de dirección &



```
// Se declara la variable de tipo entero
```

```
int i=5;
```

```
// Se declara la variable de tipo char
```

```
char c='a';
```

```
// Se declara puntero a entero
```

```
int * ptri;
```

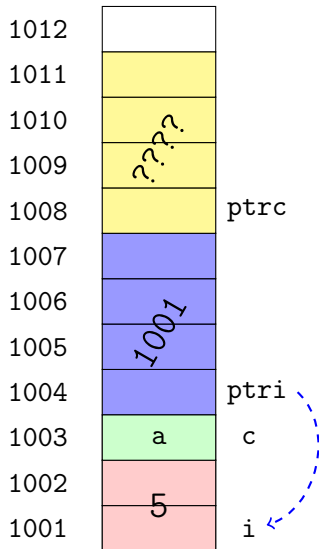
```
// Se declara el puntero a char
```

```
char * ptrc;
```

```
// ptri apunta a la variable i
```

```
ptri=&i;
```

Operador de dirección &



```
// Se declara la variable de tipo entero
```

```
int i=5;
```

```
// Se declara la variable de tipo char
```

```
char c='a';
```

```
// Se declara puntero a entero
```

```
int * ptri;
```

```
// Se declara el puntero a char
```

```
char * ptrc;
```

```
// ptri apunta a la variable i
```

```
ptri=&i;
```

Contenido del tema

- 1 Definición y Declaración de variables
- 2 Operaciones con punteros
 - Operador de dirección &
 - **Operador de indirección ***
 - Asignación e inicialización de punteros
 - Operadores relacionales
 - Operadores aritméticos
- 3 Punteros y arrays
- 4 Punteros y cadenas
- 5 Punteros, struct y class
- 6 Punteros y funciones
- 7 Punteros a punteros
- 8 Punteros y const
- 9 Arrays de punteros
- 10 Punteros a funciones
- 11 Errores comunes con punteros
- 12 Estructura de la memoria
- 13 Gestión dinámica de la memoria
- 14 Objetos Dinámicos Simples
- 15 Objetos dinámicos compuestos
- 16 Arrays dinámicos
 - Arrays dinámicos de datos de tipo primitivo
 - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas

Operador de indirección *

Operador de indirección *

*<puntero> devuelve el valor del objeto apuntado por <puntero>.

- Ejemplo:

```
char c, *ptrc;
```

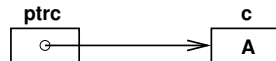
```
.....
```

```
// Hacemos que el puntero apunte a c
```

```
ptrc = &c;
```

```
// Cambiamos contenido de c mediante ptrc
```

```
*ptrc = 'A'; // equivale a c = 'A'
```



Operador de indirección *

Operador de indirección *

*<puntero> devuelve el valor del objeto apuntado por <puntero>.

- Ejemplo:

```
char c, *ptrc;
```

```
.....
```

```
// Hacemos que el puntero apunte a c
```

```
ptrc = &c;
```

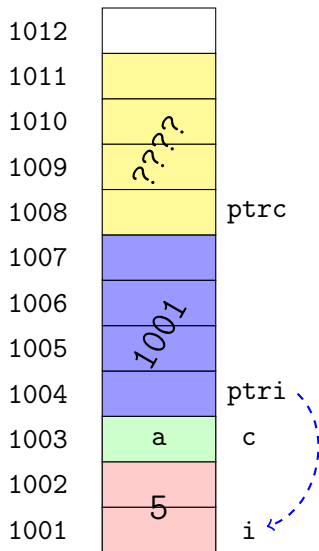
```
// Cambiamos contenido de c mediante ptrc
```

```
*ptrc = 'A'; // equivale a c = 'A'
```



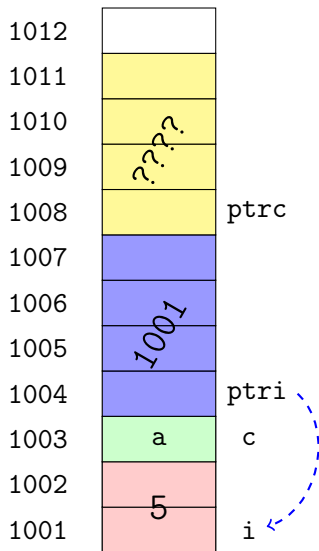
- ptrc es un puntero a carácter que contiene la dirección de c, por tanto, *ptrc es el objeto apuntado por el puntero, es decir, c.

Operador de indirección *



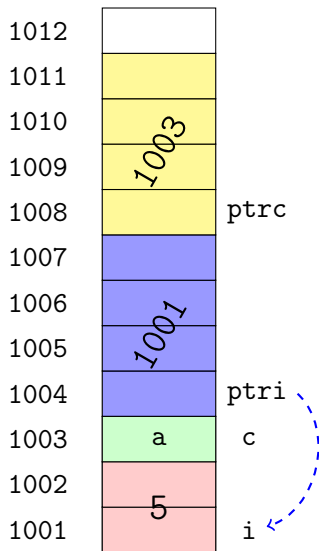
```
// Se declara la variable de tipo entero
int i=5;
// Se declara la variable de tipo char
char c='a';
// Se declara puntero a entero
int * ptri;
// Se declara el puntero a char
char * ptrc;
// ptri apunta a la variable i
ptri=&i;
// ptrc apunta a c
ptrc=&c;
//cambia contenido con ptrc
*ptrc='A';
```


Operador de indirección *



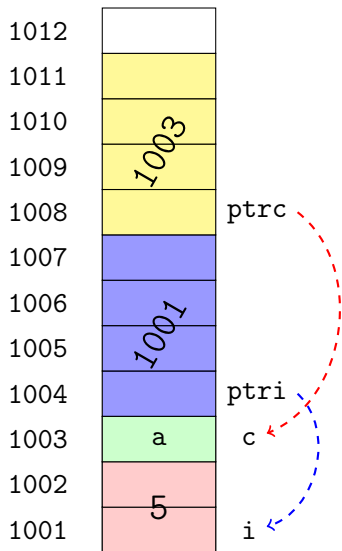
```
// Se declara la variable de tipo entero
int i=5;
// Se declara la variable de tipo char
char c='a';
// Se declara puntero a entero
int * ptri;
// Se declara el puntero a char
char * ptrc;
// ptri apunta a la variable i
ptri=&i;
// ptrc apunta a c
ptrc=&c;
//cambia contenido con ptrc
*ptrc='A';
```

Operador de indirección *



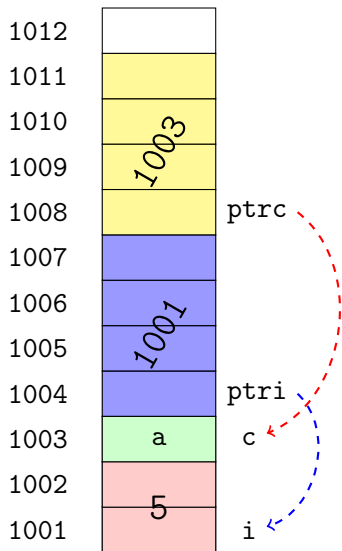
```
// Se declara la variable de tipo entero
int i=5;
// Se declara la variable de tipo char
char c='a';
// Se declara puntero a entero
int * ptri;
// Se declara el puntero a char
char * ptrc;
// ptri apunta a la variable i
ptri=&i;
// ptrc apunta a c
ptrc=&c;
//cambia contenido con ptrc
*ptrc='A';
```

Operador de indirección *



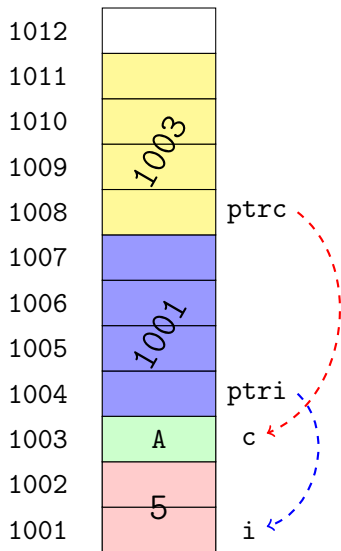
```
// Se declara la variable de tipo entero
int i=5;
// Se declara la variable de tipo char
char c='a';
// Se declara puntero a entero
int * ptri;
// Se declara el puntero a char
char * ptrc;
// ptri apunta a la variable i
ptri=&i;
// ptrc apunta a c
ptrc=&c;
//cambia contenido con ptrc
*ptrc='A';
```

Operador de indirección *



```
// Se declara la variable de tipo entero
int i=5;
// Se declara la variable de tipo char
char c='a';
// Se declara puntero a entero
int * ptri;
// Se declara el puntero a char
char * ptrc;
// ptri apunta a la variable i
ptri=&i;
// ptrc apunta a c
ptrc=&c;
//cambia contenido con ptrc
*ptrc='A';
```

Operador de indirección *



```
// Se declara la variable de tipo entero
int i=5;
// Se declara la variable de tipo char
char c='a';
// Se declara puntero a entero
int * ptri;
// Se declara el puntero a char
char * ptrc;
// ptri apunta a la variable i
ptri=&i;
// ptrc apunta a c
ptrc=&c;
//cambia contenido con ptrc
*ptrc='A';
```

Contenido del tema

- 1 Definición y Declaración de variables
- 2 Operaciones con punteros
 - Operador de dirección &
 - Operador de indirección *
 - **Asignación e inicialización de punteros**
 - Operadores relacionales
 - Operadores aritméticos
- 3 Punteros y arrays
- 4 Punteros y cadenas
- 5 Punteros, struct y class
- 6 Punteros y funciones
- 7 Punteros a punteros
- 8 Punteros y const
- 9 Arrays de punteros
- 10 Punteros a funciones
- 11 Errores comunes con punteros
- 12 Estructura de la memoria
- 13 Gestión dinámica de la memoria
- 14 Objetos Dinámicos Simples
- 15 Objetos dinámicos compuestos
- 16 Arrays dinámicos
 - Arrays dinámicos de datos de tipo primitivo
 - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas

Asignación e inicialización de punteros

Inicialización de un puntero

Un puntero se puede inicializar con la dirección de una variable.

```
int a;  
int *ptri = &a;
```

Asignación de punteros

A un puntero se le puede asignar una dirección de memoria de otra variable. La única dirección de memoria que se puede asignar directamente (valor literal) a un puntero es la dirección nula.

```
int *ptr = 0;
```

```
int *ptr = nullptr; // Válido desde C++ 11
```

Asignación e inicialización de punteros

Inicialización de un puntero

Un puntero se puede inicializar con la dirección de una variable.

```
int a;  
int *ptri = &a;
```

Asignación de punteros

A un puntero se le puede asignar una dirección de memoria de otra variable. La única dirección de memoria que se puede asignar directamente (valor literal) a un puntero es la dirección nula.

```
int *ptr = 0;
```

```
int *ptr = nullptr; // Válido desde C++ 11
```


Asignación e inicialización de punteros

Inicialización de un puntero

Un puntero se puede inicializar con la dirección de una variable.

```
int a;  
int *ptri = &a;
```

Asignación de punteros

A un puntero se le puede asignar una dirección de memoria de otra variable. La única dirección de memoria que se puede asignar directamente (valor literal) a un puntero es la dirección nula.

```
int *ptr = 0;
```

```
int *ptr = nullptr; // Válido desde C++ 11
```

Asignación e inicialización de punteros

- La asignación solo está permitida entre punteros de igual tipo.

```
int a=7;  
int *p1=&a;  
char *p2=&a; //ERROR: char *p2 = reinterpret_cast<char*>(&a);  
int *p3=p1;
```

```
asignacionPunteros.cpp: En la función 'int main()':  
asignacionPunteros.cpp:8:14: error: no se puede convertir 'int*' a 'char*' en la inicialización
```



Asignación e inicialización de punteros

- Un puntero debe estar correctamente inicializado antes de usarse

```
int a=7;  
int *p1=&a, *p2;  
*p1 = 20;  
*p2 = 30; // Error
```

Violación de segmento ('core' generado)



- Es conveniente inicializar los punteros en la declaración, con el puntero nulo: `nullptr`

```
int *p2 = nullptr;
```

Asignación e inicialización de punteros

- Un puntero debe estar correctamente inicializado antes de usarse

```
int a=7;  
int *p1=&a, *p2;  
*p1 = 20;  
*p2 = 30; // Error
```

Violación de segmento ('core' generado)

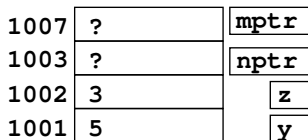


- Es conveniente inicializar los punteros en la declaración, con el puntero nulo: `nullptr`

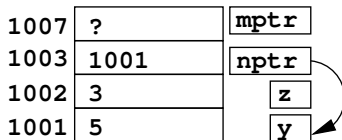
```
int *p2 = nullptr;
```

Ejemplo

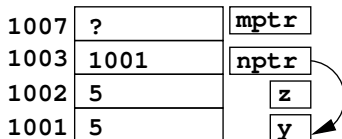
```
int main() {
    char y = 5, z = 3;
    char *nptr;
    char *mptr;
```



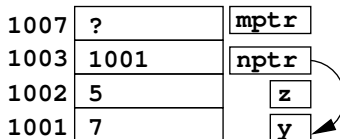
```
nptr = &y;
```



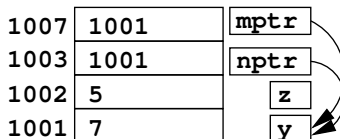
```
z = *nptr;
```



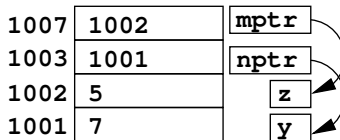
```
*nptr = 7;
```



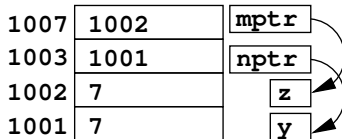
```
mptr = nptr;
```



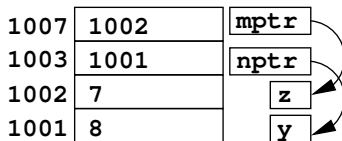
```
mptr = &z;
```



```
*mptr = *nptr;
```



```
y = (*mptr) + 1;  
}
```



Ejemplo anterior animado

1012	
1011	
1010	
1009	
1008	
1007	
1006	
1005	
1004	
1003	
1002	
1001	

```
char y = 5, z = 3;  
char * nptr;  
char * mptr;  
nptr = &y;  
z = *nptr;  
*nptr=7;  
mptr = nptr;  
mptr = &z;  
*mptr = *nptr;  
y = (*mptr)+1;
```


Ejemplo anterior animado

1012	
1011	
1010	
1009	
1008	
1007	
1006	
1005	
1004	
1003	
1002	
1001	

```
char y = 5, z = 3;
```

```
char * nptr;
```

```
char * mptr;
```

```
nptr = &y;
```

```
z = *nptr;
```

```
*nptr=7;
```

```
mptr = nptr;
```

```
mptr = &z;
```

```
*mptr = *nptr;
```

```
y = (*mptr)+1;
```

Ejemplo anterior animado

1012		
1011		
1010		
1009		
1008		
1007		
1006		
1005		
1004		
1003		
1002	3	z
1001	5	y

```
char y = 5, z = 3;
```

```
char * nptr;
```

```
char * mptr;
```

```
nptr = &y;
```

```
z = *nptr;
```

```
*nptr=7;
```

```
mptr = nptr;
```

```
mptr = &z;
```

```
*mptr = *nptr;
```

```
y = (*mptr)+1;
```

Ejemplo anterior animado

1012		
1011		
1010		
1009		
1008		
1007		
1006		
1005		
1004		
1003		
1002	3	z
1001	5	y

```
char y = 5, z = 3;
```

```
char * nptr;
```

```
char * mptr;
```

```
nptr = &y;
```

```
z = *nptr;
```

```
*nptr=7;
```

```
mptr = nptr;
```

```
mptr = &z;
```

```
*mptr = *nptr;
```

```
y = (*mptr)+1;
```

Ejemplo anterior animado

1012		
1011		
1010		
1009		
1008		
1007		
1006		
1005		
1004		
1003		nptr
1002	3	z
1001	5	y

```
char y = 5, z = 3;
```

```
char * nptr;
```

```
char * mptr;
```

```
nptr = &y;
```

```
z = *nptr;
```

```
*nptr=7;
```

```
mptr = nptr;
```

```
mptr = &z;
```

```
*mptr = *nptr;
```

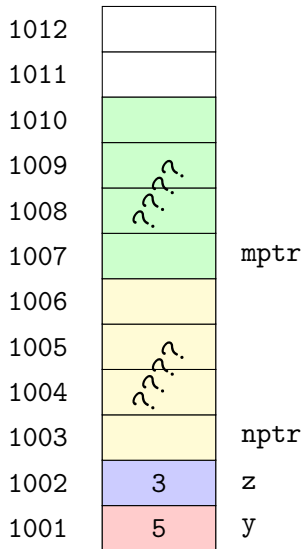
```
y = (*mptr)+1;
```

Ejemplo anterior animado

1012		
1011		
1010		
1009		
1008		
1007		
1006		
1005		
1004		
1003		nptr
1002	3	z
1001	5	y

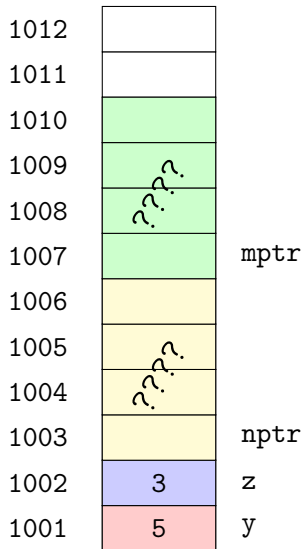
```
char y = 5, z = 3;  
char * nptr;  
char * mptr;  
nptr = &y;  
z = *nptr;  
*nptr=7;  
mptr = nptr;  
mptr = &z;  
*mptr = *nptr;  
y = (*mptr)+1;
```

Ejemplo anterior animado



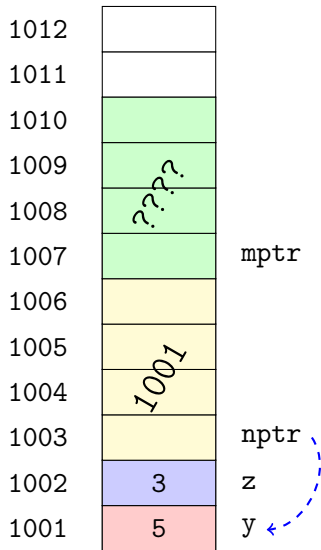
```
char y = 5, z = 3;  
char * nptr;  
char * mptr;  
nptr = &y;  
z = *nptr;  
*nptr=7;  
mptr = nptr;  
mptr = &z;  
*mptr = *nptr;  
y = (*mptr)+1;
```

Ejemplo anterior animado



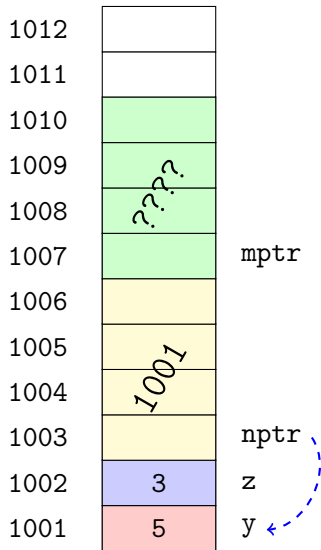
```
char y = 5, z = 3;  
char * nptr;  
char * mptr;  
nptr = &y;  
z = *nptr;  
*nptr=7;  
mptr = nptr;  
mptr = &z;  
*mptr = *nptr;  
y = (*mptr)+1;
```

Ejemplo anterior animado



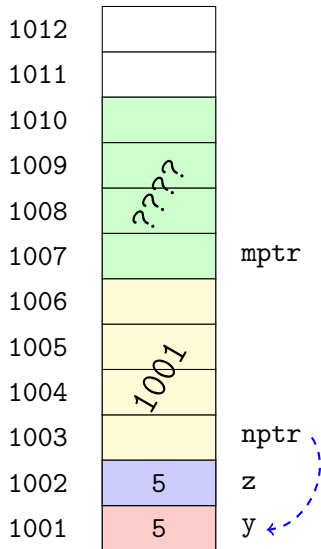
```
char y = 5, z = 3;  
char * nptr;  
char * mptr;  
nptr = &y;  
z = *nptr;  
*nptr=7;  
mptr = nptr;  
mptr = &z;  
*mptr = *nptr;  
y = (*mptr)+1;
```


Ejemplo anterior animado



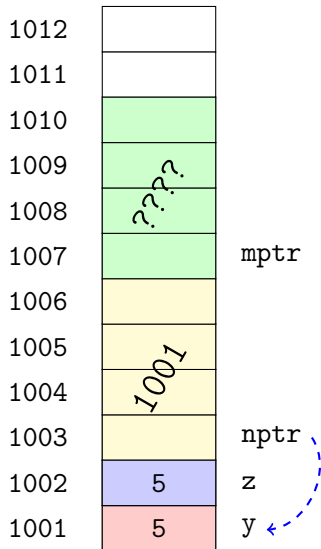
```
char y = 5, z = 3;  
char * nptr;  
char * mptr;  
nptra = &y;  
z = *nptra;  
*nptra=7;  
mptra = nptra;  
mptra = &z;  
*mptra = *nptra;  
y = (*mptra)+1;
```

Ejemplo anterior animado



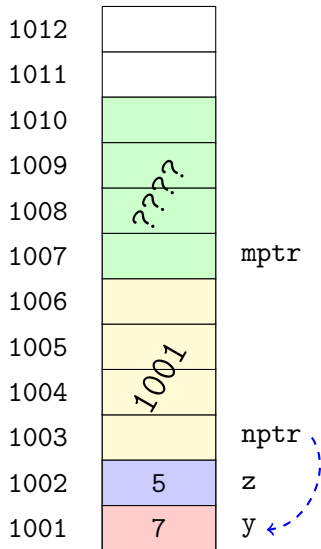
```
char y = 5, z = 3;  
char * nptr;  
char * mptr;  
nptr = &y;  
z = *nptr;  
*nptr=7;  
mptr = nptr;  
mptr = &z;  
*mptr = *nptr;  
y = (*mptr)+1;
```

Ejemplo anterior animado



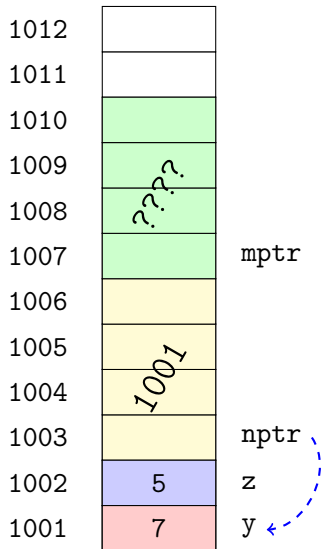
```
char y = 5, z = 3;  
char * nptr;  
char * mptr;  
nptr = &y;  
z = *nptr;  
*nptr=7;  
mptr = nptr;  
mptr = &z;  
*mptr = *nptr;  
y = (*mptr)+1;
```

Ejemplo anterior animado



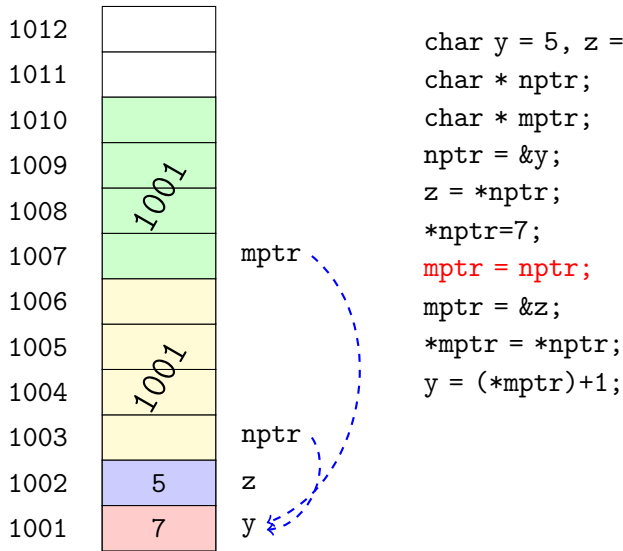
```
char y = 5, z = 3;  
char * nptr;  
char * mptr;  
nptra = &y;  
z = *nptra;  
*nptra=7;  
mptra = nptra;  
mptra = &z;  
*mptra = *nptra;  
y = (*mptra)+1;
```

Ejemplo anterior animado



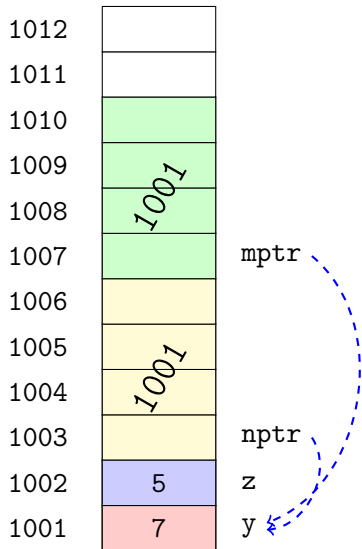
```
char y = 5, z = 3;  
char * nptr;  
char * mptr;  
nptr = &y;  
z = *nptr;  
*nptr=7;  
mptr = nptr;  
mptr = &z;  
*mptr = *nptr;  
y = (*mptr)+1;
```

Ejemplo anterior animado



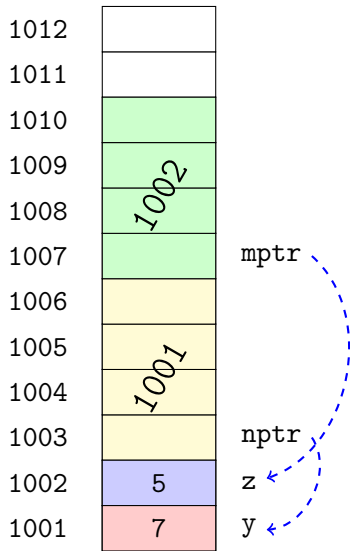
```
char y = 5, z = 3;  
char * nptr;  
char * mptr;  
nptr = &y;  
z = *nptr;  
*nptr=7;  
mptr = nptr;  
mptr = &z;  
*mptr = *nptr;  
y = (*mptr)+1;
```

Ejemplo anterior animado



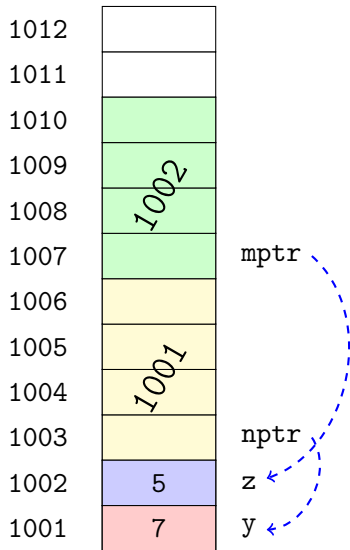
```
char y = 5, z = 3;  
char * nptr;  
char * mptr;  
nptra = &y;  
z = *nptra;  
*nptra=7;  
mptra = nptra;  
mptra = &z;  
*mptra = *nptra;  
y = (*mptra)+1;
```

Ejemplo anterior animado



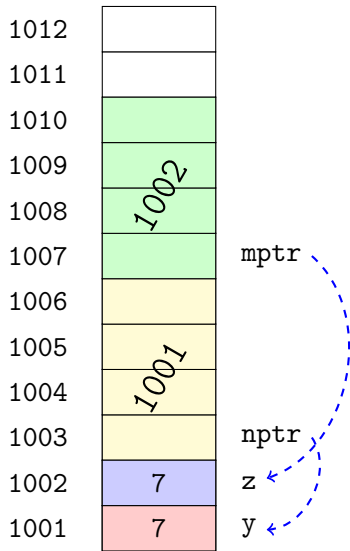
```
char y = 5, z = 3;  
char * nptr;  
char * mptra;  
nptr = &y;  
z = *nptr;  
*nptr=7;  
mptra = nptr;  
mptra = &z;  
*mptra = *nptr;  
y = (*mptra)+1;
```


Ejemplo anterior animado



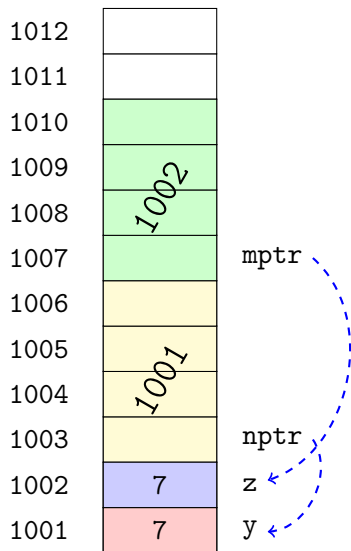
```
char y = 5, z = 3;  
char * nptr;  
char * mptr;  
nptr = &y;  
z = *nptr;  
*nptr=7;  
mptr = nptr;  
mptr = &z;  
*mptr = *nptr;  
y = (*mptr)+1;
```

Ejemplo anterior animado



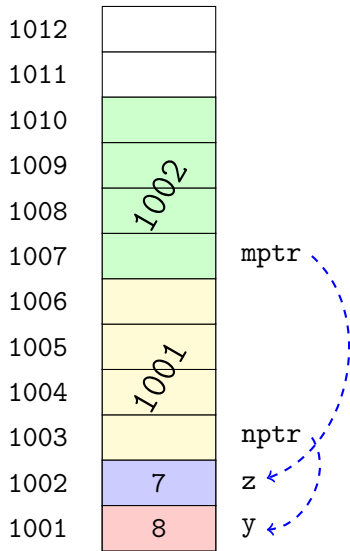
```
char y = 5, z = 3;  
char * nptr;  
char * mptr;  
nptr = &y;  
z = *nptr;  
*nptr=7;  
mptr = nptr;  
mptr = &z;  
*mptr = *nptr;  
y = (*mptr)+1;
```

Ejemplo anterior animado



```
char y = 5, z = 3;  
char * nptr;  
char * mptr;  
nptr = &y;  
z = *nptr;  
*nptr=7;  
mptr = nptr;  
mptr = &z;  
*mptr = *nptr;  
y = (*mptr)+1;
```

Ejemplo anterior animado



```
char y = 5, z = 3;  
char * nptr;  
char * mptr;  
nptr = &y;  
z = *nptr;  
*nptr=7;  
mptr = nptr;  
mptr = &z;  
*mptr = *nptr;  
y = (*mptr)+1;
```

Contenido del tema

- 1 Definición y Declaración de variables
- 2 Operaciones con punteros
 - Operador de dirección &
 - Operador de indirección *
 - Asignación e inicialización de punteros
 - Operadores relacionales
 - Operadores aritméticos
- 3 Punteros y arrays
- 4 Punteros y cadenas
- 5 Punteros, struct y class
- 6 Punteros y funciones
- 7 Punteros a punteros
- 8 Punteros y const
- 9 Arrays de punteros
- 10 Punteros a funciones
- 11 Errores comunes con punteros
- 12 Estructura de la memoria
- 13 Gestión dinámica de la memoria
- 14 Objetos Dinámicos Simples
- 15 Objetos dinámicos compuestos
- 16 Arrays dinámicos
 - Arrays dinámicos de datos de tipo primitivo
 - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas

Contenido del tema

- 1 Definición y Declaración de variables
- 2 Operaciones con punteros
 - Operador de dirección &
 - Operador de indirección *
 - Asignación e inicialización de punteros
 - Operadores relacionales
 - Operadores aritméticos
- 3 Punteros y arrays
- 4 Punteros y cadenas
- 5 Punteros, struct y class
- 6 Punteros y funciones
- 7 Punteros a punteros
- 8 Punteros y const**
- 9 Arrays de punteros
- 10 Punteros a funciones
- 11 Errores comunes con punteros
- 12 Estructura de la memoria
- 13 Gestión dinámica de la memoria
- 14 Objetos Dinámicos Simples
- 15 Objetos dinámicos compuestos
- 16 Arrays dinámicos
 - Arrays dinámicos de datos de tipo primitivo
 - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas

Punteros y const

- Cuando tratamos con punteros manejamos dos datos:
 - El dato puntero.
 - El dato que es apuntado.
- Pueden ocurrir las siguientes situaciones:

Ninguno sea const	<code>double *p;</code>
Solo el dato apuntado sea const	<code>const double *p;</code>
Solo el puntero sea const	<code>double *const p;</code>
Los dos sean const	<code>const double *const p;</code>

- Las siguientes expresiones son equivalentes:

<code>const double *p;</code>	<code>double const *p;</code>
-------------------------------	-------------------------------



Punteros const y no const

Es posible asignar un puntero no const a uno const, pero no al revés (en la asignación se hace una conversión implícita).

```
double a = 1.0;
double * const p=&a; // puntero constante a double
double * q;         // puntero no constante a double
q = p;              // BIEN: q puede apuntar a cualquier dato
p = q;              // MAL: p es constante
```

Error de compilación:

...error: asignación de la variable de solo lectura 'p'

p ha quedado asignado en la declaración de la constante y no admite cambios posteriores (como buena constante.....)

Punteros const y no const

Es posible asignar un puntero no const a uno const, pero no al revés (en la asignación se hace una conversión implícita).

```
double a = 1.0;
double * const p=&a; // puntero constante a double
double * q;         // puntero no constante a double
q = p;              // BIEN: q puede apuntar a cualquier dato
p = q;              // MAL: p es constante
```

Error de compilación:

...error: asignación de la variable de solo lectura 'p'

p ha quedado asignado en la declaración de la constante y no admite cambios posteriores (como buena constante.....)

Puntero a dato no const

Un puntero a dato no const no puede apuntar a un dato const.

Ejemplo 1

El siguiente código da error ya que `&f` devuelve un `const double *`

```
double *p;  
const double f=5.2;  
p = &f;    // INCORRECTO, ya que permitiría cambiar el  
*p = 5.0;  // valor de f a través de p
```

Error de compilación:

...error: conversión inválida de 'const double*' a 'double*'[-fpermissive]

Nota: observad que de permitirse la operación se permitiría cambiar el valor de `f`, que fue declarada como constante.

Puntero a dato no const

Un puntero a dato no const no puede apuntar a un dato const.

Ejemplo 1

El siguiente código da error ya que `&f` devuelve un `const double *`

```
double *p;  
const double f=5.2;  
p = &f;      // INCORRECTO, ya que permitiría cambiar el  
*p = 5.0;    // valor de f a través de p
```

Error de compilación:


...error: conversión inválida de 'const double*' a 'double*'
[-fpermissive]

Nota: observad que de permitirse la operación se permitiría cambiar el valor de `f`, que fue declarada como constante.

Ejemplo 2

El siguiente código da error ya que *p devuelve un const double

```
const double *p;  
double f;  
p = &f;    // (const double *) = (double *)  
*p = 5.0;  // ERROR: no se puede cambiar el valor
```



Error de compilación:

...error: asignación de la ubicación de solo lectura '*p'

Ejemplo 3

El siguiente código da error ya que `&(vocales[2])` devuelve un `const char *`

```
const char vocales[5]={'a','e','i','o','u'};  
char *p;  
p = &(vocales[2]); // ERROR de compilación
```

Error de compilación:

...error: conversión inválida de 'const char*' a 'char*' [-fpermissive]

Punteros, funciones y const

Funciones con parámetro puntero a dato const

Podemos llamar a una función que espera un puntero a dato const con uno a dato no const.

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

void EscribirEntero(const int *p){
    cout << *p;
}

int main(){
    const int a = 1;
    int b=2;
    HacerCero(&a);           // ERROR
    EscribirEntero(&a);      // CORRECTO
    EscribirEntero(&b);      // CORRECTO
}
```



Error de compilación:

...error: conversión inválida de 'const int*' a 'int*' [-fpermissive]

Punteros, arrays y const

Array de constantes y puntero a dato const

Dada la estrecha relación entre arrays y punteros, podemos usar un array de constantes como un puntero a constantes, y al contrario:

```
const int matConst[5]={1,2,3,4,5};  
int mat[3]={3,5,7};  
const int *pconst;  
int *p;  
pconst = matConst;  // CORRECTO  
pconst = mat;        // CORRECTO  
p = mat;             // CORRECTO  
p = matConst;        // ERROR
```



Contenido del tema

- 1 Definición y Declaración de variables
- 2 Operaciones con punteros
 - Operador de dirección &
 - Operador de indirección *
 - Asignación e inicialización de punteros
 - Operadores relacionales
 - Operadores aritméticos
- 3 Punteros y arrays
- 4 Punteros y cadenas
- 5 Punteros, struct y class
- 6 Punteros y funciones
- 7 Punteros a punteros
- 8 Punteros y const
- 9 Arrays de punteros
- 10 Punteros a funciones
- 11 Errores comunes con punteros
- 12 Estructura de la memoria
- 13 Gestión dinámica de la memoria
- 14 Objetos Dinámicos Simples
- 15 Objetos dinámicos compuestos
- 16 Arrays dinámicos
 - Arrays dinámicos de datos de tipo primitivo
 - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas

Algunos errores comunes

- Asignar puntero de distinto tipo

```
int a=10, *ptri;
double b=5.0, *ptrf;
```

```
ptri = &a;
ptrf = &b;
ptrf = ptri; // Error en compilación
```

- Uso de punteros no inicializados

```
char y=5, *nptr;
*nptr=5; // ERROR
```

- Asignación de valores al puntero y no a la variable.

```
char y=5, *nptr =&y;
nptr = 9; // Error de compilación
```

Algunos errores comunes

- Asignar puntero de distinto tipo

```
int a=10, *ptri;  
double b=5.0, *ptrf;
```

```
ptri = &a;  
ptrf = &b;  
ptrf = ptri; // Error en compilación
```

- Uso de punteros no inicializados

```
char y=5, *nptr;  
*nptr=5; // ERROR
```

- Asignación de valores al puntero y no a la variable.

```
char y=5, *nptr =&y;  
nptr = 9; // Error de compilación
```

Algunos errores comunes

- Asignar puntero de distinto tipo

```
int a=10, *ptri;  
double b=5.0, *ptrf;
```

```
ptri = &a;  
ptrf = &b;  
ptrf = ptri; // Error en compilación
```

- Uso de punteros no inicializados

```
char y=5, *nptr;  
*nptr=5; // ERROR
```

- Asignación de valores al puntero y no a la variable.

```
char y=5, *nptr =&y;  
nptr = 9; // Error de compilación
```

Parte II

Gestión Dinámica de Memoria

Contenido del tema

- 1 Definición y Declaración de variables
- 2 Operaciones con punteros
 - Operador de dirección &
 - Operador de indirección *
 - Asignación e inicialización de punteros
 - Operadores relacionales
 - Operadores aritméticos
- 3 Punteros y arrays
- 4 Punteros y cadenas
- 5 Punteros, struct y class
- 6 Punteros y funciones
- 7 Punteros a punteros
- 8 Punteros y const
- 9 Arrays de punteros
- 10 Punteros a funciones
- 11 Errores comunes con punteros
- 12 Estructura de la memoria
- 13 Gestión dinámica de la memoria
- 14 Objetos Dinámicos Simples**
- 15 Objetos dinámicos compuestos
- 16 Arrays dinámicos
 - Arrays dinámicos de datos de tipo primitivo
 - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas

El operador new

Operador new

Reserva una zona de memoria en el Heap del tamaño adecuado para almacenar un dato del tipo *tipo* (**sizeof**(*tipo*) bytes), devolviendo la dirección de memoria donde empieza la zona reservada.

```
<tipo> *p;  
p = new <tipo>;
```



- Si new no puede reservar espacio (p.e. no hay suficiente memoria disponible), se provoca una excepción y el programa termina.
- Por ahora supondremos que siempre habrá suficiente memoria.

Otra opción (no recomendable)

```
<tipo> *p;  
p = new (nothrow) <tipo>;
```

En caso de que no se haya podido hacer la reserva devuelve el puntero nulo (nullptr).

El operador new

Operador new

Reserva una zona de memoria en el Heap del tamaño adecuado para almacenar un dato del tipo *tipo* (**sizeof**(*tipo*) bytes), devolviendo la dirección de memoria donde empieza la zona reservada.

```
<tipo> *p;  
p = new <tipo>;
```

- Si new no puede reservar espacio (p.e. no hay suficiente memoria disponible), se provoca una excepción y el programa termina.
- Por ahora supondremos que siempre habrá suficiente memoria.

Otra opción (no recomendable)

```
<tipo> *p;  
p = new (nothrow) <tipo>;
```

En caso de que no se haya podido hacer la reserva devuelve el puntero nulo (nullptr).

El operador new

Operador new

Reserva una zona de memoria en el Heap del tamaño adecuado para almacenar un dato del tipo *tipo* (**sizeof**(*tipo*) bytes), devolviendo la dirección de memoria donde empieza la zona reservada.

```
<tipo> *p;  
p = new <tipo>;
```

- Si new no puede reservar espacio (p.e. no hay suficiente memoria disponible), se provoca una excepción y el programa termina.
- Por ahora supondremos que siempre habrá suficiente memoria.

Otra opción (no recomendable)

```
<tipo> *p;  
p = new (nothrow) <tipo>;
```

En caso de que no se haya podido hacer la reserva devuelve el puntero nulo (nullptr).

El operador new

Operador new


Reserva una zona de memoria en el Heap del tamaño adecuado para almacenar un dato del tipo *tipo* (**sizeof**(*tipo*) bytes), devolviendo la dirección de memoria donde empieza la zona reservada.

```
<tipo> *p;
p = new <tipo>;
```

- Si **new** no puede reservar espacio (p.e. no hay suficiente memoria disponible), se provoca una excepción y el programa termina.
- Por ahora supondremos que siempre habrá suficiente memoria.

Otra opción (no recomendable)

```
<tipo> *p;
p = new (nothrow) <tipo>;
```



En caso de que no se haya podido hacer la reserva devuelve el puntero nulo (nullptr).

Ejemplo

```
int main(){  
    int *p;  
  
    p = new int;  
    *p = 10;  
}
```



Notas:

- Observad que **p** se declara como un puntero más.
- Se pide memoria en el Heap para guardar un dato **int**. Si hay espacio para satisfacer la petición, **p** apuntará al principio de la zona reservada por **new**. Asumiremos que siempre hay memoria libre para asignar.
- Se trabaja, como ya sabemos, con el objeto referenciado por **p**.

Ejemplo

```
int main(){  
    int *p;  
  
    p = new int;  
    *p = 10;  
}
```

Notas:

- Observad que **p** se declara como un puntero más.
- Se pide memoria en el Heap para guardar un dato **int**. Si hay espacio para satisfacer la petición, **p** apuntará al principio de la zona reservada por **new**. Asumiremos que siempre hay memoria libre para asignar.
- Se trabaja, como ya sabemos, con el objeto referenciado por **p**.

Ejemplo

```
int main(){  
    int *p;  
  
    p = new int;  
    *p = 10;  
}
```

Notas:

- Observad que **p** se declara como un puntero más.
- Se pide memoria en el Heap para guardar un dato **int**. Si hay espacio para satisfacer la petición, **p** apuntará al principio de la zona reservada por **new**. Asumiremos que siempre hay memoria libre para asignar.
- Se trabaja, como ya sabemos, con el objeto referenciado por **p**.

El operador delete

Operador delete

Libera la memoria del Heap que previamente se había reservado y que se encuentra referenciada por un puntero.

```
delete puntero;
```

Ejemplo

```
int main(){  
    int *p, q=10;  
  
    p = new int;  
    *p = q;  
    .....  
    delete p;  
}
```

Notas:

- El objeto referenciado por **p** deja de ser “operativo” y la memoria que ocupaba está disponible para nuevas peticiones con **new**.

El operador delete

Operador delete

Libera la memoria del Heap que previamente se había reservado y que se encuentra referenciada por un puntero.

```
delete puntero;
```

Ejemplo

```
int main(){  
    int *p, q=10;  
  
    p = new int;  
    *p = q;  
    .....  
    delete p;  
}
```



Notas:

- El objeto referenciado por **p** deja de ser “operativo” y la memoria que ocupaba está disponible para nuevas peticiones con **new**.

Contenido del tema

- 1 Definición y Declaración de variables
- 2 Operaciones con punteros
 - Operador de dirección &
 - Operador de indirección *
 - Asignación e inicialización de punteros
 - Operadores relacionales
 - Operadores aritméticos
- 3 Punteros y arrays
- 4 Punteros y cadenas
- 5 Punteros, struct y class
- 6 Punteros y funciones
- 7 Punteros a punteros
- 8 Punteros y const
- 9 Arrays de punteros
- 10 Punteros a funciones
- 11 Errores comunes con punteros
- 12 Estructura de la memoria
- 13 Gestión dinámica de la memoria
- 14 Objetos Dinámicos Simples
- 15 Objetos dinámicos compuestos
- 16 Arrays dinámicos**
 - Arrays dinámicos de datos de tipo primitivo
 - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas

Arrays dinámicos

Motivación

- Hasta ahora, solo podíamos crear un array conociendo *a priori* el número máximo de elementos que podría llegar a tener. P.e.
`int vector[20];`
- Esa memoria está ocupada durante la ejecución del módulo en el que se realiza la declaración.

Array dinámico

Usando memoria dinámica, podemos crear arrays dinámicos que tengan **justo el tamaño necesario**.

Podemos, además, crearlos **justo en el momento** en el que lo necesitamos y destruirlos cuando dejen de ser útiles.

Arrays dinámicos

Motivación

- Hasta ahora, solo podíamos crear un array conociendo *a priori* el número máximo de elementos que podría llegar a tener. P.e.
`int vector[20];`
- Esa memoria está ocupada durante la ejecución del módulo en el que se realiza la declaración.

Array dinámico

Usando memoria dinámica, podemos crear arrays dinámicos que tengan **justo el tamaño necesario**.

Podemos, además, crearlos **justo en el momento** en el que lo necesitamos y destruirlos cuando dejen de ser útiles.

Contenido del tema

- 1 Definición y Declaración de variables
- 2 Operaciones con punteros
 - Operador de dirección &
 - Operador de indirección *
 - Asignación e inicialización de punteros
 - Operadores relacionales
 - Operadores aritméticos
- 3 Punteros y arrays
- 4 Punteros y cadenas
- 5 Punteros, struct y class
- 6 Punteros y funciones
- 7 Punteros a punteros
- 8 Punteros y const
- 9 Arrays de punteros
- 10 Punteros a funciones
- 11 Errores comunes con punteros
- 12 Estructura de la memoria
- 13 Gestión dinámica de la memoria
- 14 Objetos Dinámicos Simples
- 15 Objetos dinámicos compuestos
- 16 **Arrays dinámicos**
 - **Arrays dinámicos de datos de tipo primitivo**
 - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas

Arrays dinámicos

Operador new[]

Reserva una zona de memoria en el Heap para almacenar `num` datos de tipo `<tipo>`, devolviendo la dirección de memoria inicial.

`num` es un entero estrictamente mayor que 0.

```
<tipo> *p;  
p = new <tipo> [num];
```



Operador delete[]

Libera (pone como disponible) la zona de memoria **previamente reservada** por una orden `new[]`, zona referenciada por puntero.

```
delete [] puntero;
```

Arrays dinámicos

Operador new[]

Reserva una zona de memoria en el Heap para almacenar `num` datos de tipo `<tipo>`, devolviendo la dirección de memoria inicial.

`num` es un entero estrictamente mayor que 0.

```
<tipo> *p;
```

```
p = new <tipo> [num];
```

Operador delete[]

Libera (pone como disponible) la zona de memoria **previamente reservada** por una orden `new[]`, zona referenciada por puntero.

```
delete [] puntero;
```

Arrays dinámicos: Ejemplo I

Ejemplo de creación y destrucción de array dinámico

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     int *v = nullptr, n;
6
7     cout << "Número de casillas: ";
8     cin >> n;
9     // Reserva de memoria
10    v = new int [n];
```



Ejemplo I

```
1  for (int i= 0; i<n; i++) {    // Lectura del vector dinámico
2      cout << "Valor en casilla " << i << ": ";
3      cin >> v[i];
4  }
5  cout << endl;
6
7  for (int i= 0; i<n; i++) // Escritura del vector dinámico
8      cout << "En la casilla " << i
9          << " guardo: " << v[i] << endl;
10
11  delete [] v; // Liberar memoria
12  v = nullptr;
13 }
```



Ejemplo

Una función que devuelve una copia de un array automático (o dinámico) en un array dinámico.

```
1 #include <iostream>
2 using namespace std;
3
4 int *copia_vector(const int v[], int n){
5     int *copia = new int[n];
6     for (int i=0; i<n; i++)
7         copia[i]=v[i];
8     return copia;
9 }
10 int main(){
11     int v1[30], *v2 = nullptr, m;
12     cout << "Número de casillas: ";
13     cin >> m;
```



```
14  for (int i=0; i<m; i++) { // Rellenar el vector
15      cout << "Valor en casilla " << i << ": ";
16      cin >> v1[i];
17  }
18  cout << endl;
19
20  // Copiar en v2 (dinámico) el vector v1
21  v2 = copia_vector(v1,m);
22
23  for (int i=0; i<m; i++) // Escribir vector v2
24      cout << "En la casilla " << i
25          << " guardo: " << v2[i] << endl;
26
27  delete [] v2; // Liberar memoria
28  v2 = nullptr;
29 }
```


¡Cuidado!

Un **error** muy común a la hora de construir una función que copie un array es el siguiente:

```
int *copia_vector(const int v[], int n){  
    int copia[100];  
    for (int i=0; i<n; i++)  
        copia[i]=v[i];  
    return copia;  
}
```

¡Cuidado!

Al ser copia una variable local no puede ser usada fuera del ámbito de la función en la que está definida.

¡Cuidado!

Un **error** muy común a la hora de construir una función que copie un array es el siguiente:

```
int *copia_vector(const int v[], int n){  
    int copia[100];  
    for (int i=0; i<n; i++)  
        copia[i]=v[i];  
    return copia;  
}
```

¡Cuidado!

Al ser copia una variable local no puede ser usada fuera del ámbito de la función en la que está definida.

Ejemplo: Ampliación del espacio ocupado por un array dinámico

```
void redimensionar (int* &v, int& tama, int aumento){  
    if(tama+aumento > 0){  
        int *v_ampliado = new int[tama+aumento];  
  
        for (int i=0; (i<tama) && (i<tama+aumento); i++)  
            v_ampliado[i] = v[i];  
        delete[] v;  
        v = v_ampliado;  
        tama=tama+aumento;  
    }  
}
```

Cuestiones a tener en cuenta:

- v y tama se pasan por referencia porque se van a modificar.
- Es necesario liberar v antes de asignarle el valor de v_ampliado.
- El aumento de tamaño puede ser positivo o negativo.

Ejemplo: Ampliación del espacio ocupado por un array dinámico

```
void redimensionar (int* &v, int& tama, int aumento){
    if(tama+aumento > 0){
        int *v_ampliado = new int[tama+aumento];

        for (int i=0; (i<tama) && (i<tama+aumento); i++)
            v_ampliado[i] = v[i];
        delete[] v;
        v = v_ampliado;
        tama=tama+aumento;
    }
}
```

Cuestiones a tener en cuenta:

- v y tama se pasan por referencia porque se van a modificar.
- Es necesario liberar v antes de asignarle el valor de v_ampliado.
- El aumento de tamaño puede ser positivo o negativo.

Contenido del tema

- 1 Definición y Declaración de variables
- 2 Operaciones con punteros
 - Operador de dirección &
 - Operador de indirección *
 - Asignación e inicialización de punteros
 - Operadores relacionales
 - Operadores aritméticos
- 3 Punteros y arrays
- 4 Punteros y cadenas
- 5 Punteros, struct y class
- 6 Punteros y funciones
- 7 Punteros a punteros
- 8 Punteros y const
- 9 Arrays de punteros
- 10 Punteros a funciones
- 11 Errores comunes con punteros
- 12 Estructura de la memoria
- 13 Gestión dinámica de la memoria
- 14 Objetos Dinámicos Simples
- 15 Objetos dinámicos compuestos
- 16 Arrays dinámicos**
 - Arrays dinámicos de datos de tipo primitivo
 - **Arrays dinámicos de objetos**
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas

Array dinámico de objetos

Array dinámico de objetos

Usando el operador `new[]` y `delete[]` podemos crear y destruir también arrays dinámicos de objetos `struct` y `class`

- Operador `new[]`:
 - **Reserva la memoria** necesaria para almacenar todos y cada uno de los objetos del array.
 - Y llama al **constructor** para cada objeto del array.
- Operador `delete[]`:
 - Llama al **destructor** de la clase con cada objeto del array.
 - Y después **libera la memoria** ocupada por el array de objetos.

Array dinámico de objetos

Ejemplo con class

```
class Estudiante {  
    string nombre;  
    int nAsignaturasMatricula;  
    vector<int> codigosAsignaturasMatricula;  
public:  
    Estudiante();  
    Estudiante(string name);  
  
    void setNombre(string nuevoNombre);  
    string getNombre() const;  
    void insertaAsignatura(int codigo);  
    int getNumeroAsignaturas() const;  
    int getCodigoAsignatura(int index) const;  
    ...  
};
```



Array dinámico de objetos

```
int main() {  
    Estudiante* arrayEstudiantes;  
    arrayEstudiantes=new Estudiante[50];  
    arrayEstudiantes[0].setNombre("Ramón Rodríguez Ramírez");  
    arrayEstudiantes[0].insertaAsignatura(302);  
    arrayEstudiantes[0].insertaAsignatura(307);  
    arrayEstudiantes[0].insertaAsignatura(205);  
    ...  
    delete[] arrayEstudiantes;  
}
```


Contenido del tema

- 1 Definición y Declaración de variables
- 2 Operaciones con punteros
 - Operador de dirección &
 - Operador de indirección *
 - Asignación e inicialización de punteros
 - Operadores relacionales
 - Operadores aritméticos
- 3 Punteros y arrays
- 4 Punteros y cadenas
- 5 Punteros, struct y class
- 6 Punteros y funciones
- 7 Punteros a punteros
- 8 Punteros y const
- 9 Arrays de punteros
- 10 Punteros a funciones
- 11 Errores comunes con punteros
- 12 Estructura de la memoria
- 13 Gestión dinámica de la memoria
- 14 Objetos Dinámicos Simples
- 15 Objetos dinámicos compuestos
- 16 Arrays dinámicos
 - Arrays dinámicos de datos de tipo primitivo
 - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica**
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas

Clases que contienen datos en memoria dinámica

Clases que contienen datos en memoria dinámica

Una clase puede contener datos miembro punteros que pueden usarse para alojar datos en memoria dinámica. Para ello:

- Los **constructores** pueden reservar la memoria dinámica al crear los objetos.
- Otros métodos podrían aumentar o disminuir el tamaño de la memoria dinámica necesaria.
- El **destructor** liberará automáticamente la memoria dinámica que contenga el objeto.
Lo veremos en tema 4. Por ahora, lo haremos explícitamente usando un método `liberar()`.

Clases que contienen datos en memoria dinámica

Ejemplo: clase Poligono

Contiene un array dinámico con los vértices (objetos Punto).

```
class Punto{
    double x;
    double y;
public:
    Punto(){x=0; y=0;};
    Punto(int x, int y){this->x=x; this->y=y};
    double getX(){return x;} const;
    double getY(){return y;} const;
    double setXY(int x, int y){this->x=x; this->y=y};
};
```

Clases que contienen datos en memoria dinámica

```
class Poligono{
    int nVertices;
    Punto* vertices;
public:
    Poligono();
    ~Poligono(); // destructor (lo veremos en tema 4)
    Poligono(const Poligono& otro); // constructor copia (tema 4)
    Poligono& operator=(const Poligono& otro); // op asignación (tema 4)
    int getNumeroVertices() const;
    Punto getVertice(int index) const;
    void addVertice(Punto v);
    ...
};
```

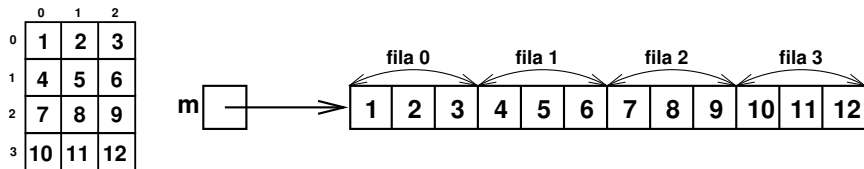
Clases que contienen datos en memoria dinámica

```
int main() {  
    Punto punto;  
    Poligono poligono;  
    punto.setXY(10,10);  
    poligono.addVertice(punto);  
    ...  
    poligono.destruir();  
}
```

Contenido del tema

- 1 Definición y Declaración de variables
- 2 Operaciones con punteros
 - Operador de dirección &
 - Operador de indirección *
 - Asignación e inicialización de punteros
 - Operadores relacionales
 - Operadores aritméticos
- 3 Punteros y arrays
- 4 Punteros y cadenas
- 5 Punteros, struct y class
- 6 Punteros y funciones
- 7 Punteros a punteros
- 8 Punteros y const
- 9 Arrays de punteros
- 10 Punteros a funciones
- 11 Errores comunes con punteros
- 12 Estructura de la memoria
- 13 Gestión dinámica de la memoria
- 14 Objetos Dinámicos Simples
- 15 Objetos dinámicos compuestos
- 16 Arrays dinámicos
 - Arrays dinámicos de datos de tipo primitivo
 - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas**
- 19 Lista de celdas enlazadas

Matriz 2D usando un array 1D




- Creación de la matriz:

```
int *m;
int nfil, ncol;
m = new int[nfil*ncol];
```

- Acceso al elemento f,c:

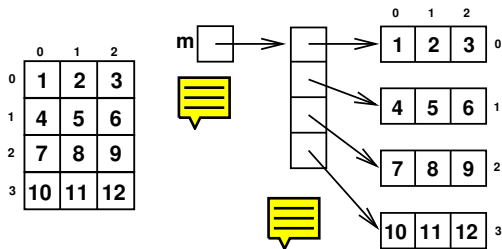
```
int a;
a = m[f*ncol+c];
```



- Liberación de la matriz:

```
delete[] m;
```

Matriz 2D usando un array 1D de punteros a arrays 1D



- Creación de la matriz:

```
int **m;
int nfil, ncol;
m = new int*[nfil];
for (int i=0; i<nfil;++i)
    m[i] = new int[ncol];
```

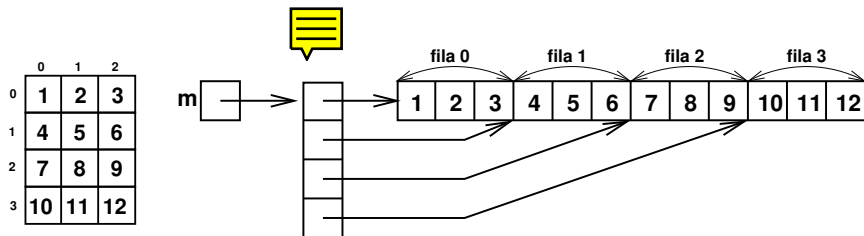
- Acceso al elemento f,c:

```
int a;
a = m[f][c];
```

- Liberación de la matriz:

```
for(int i=0;i<nfil;++i)
    delete[] m[i];
delete[] m;
```


Matriz 2D usando un array 1D de punteros a un único array



- Creación de la matriz:

```
int **m;
int nfil, ncol;
m = new int*[nfil];
m[0] = new int[nfil*ncol];
for (int i=1; i<nfil;++i)
    m[i] = m[i-1]+ncol;
```

- Acceso al elemento f, c :

```
int a;
a = m[f][c];
```

- Liberación de la matriz:

```
delete[] m[0];
delete[] m;
```

Contenido del tema I

- 1 Introducción
 - Abstracción funcional
 - Abstracción de datos
- 2 Clases con datos dinámicos
- 3 Los constructores
- 4 Los métodos de la clase
 - Métodos const
 - Métodos inline
 - Métodos modificadores de la interfaz básica
 - Métodos adicionales en la interfaz de la clase
 - Puntero this
- 5 Funciones y clases friend
- 6 El destructor
- 7 Constructores y destructores en clases con datos miembro de otras clases
- 8 El constructor de copia
 - El constructor de copia por defecto
 - Creación de un constructor de copia
 - Llamadas al constructor de copia
- 9 Llamadas a constructores y destructores
 - Llamadas al declarar variables locales y globales
 - Llamadas explícitas a un constructor

Contenido del tema II

- Listas de inicialización en constructores
- Arrays de objetos
- Creación/destrucción de objetos en memoria dinámica
- Uso de constructores para hacer conversiones implícitas

Contenido del tema

- 1 **Introducción**
 - Abstracción funcional
 - Abstracción de datos
- 2 Clases con datos dinámicos
- 3 Los constructores
- 4 Los métodos de la clase
 - Métodos const
 - Métodos inline
 - Métodos modificadores de la interfaz básica
 - Métodos adicionales en la interfaz de la clase
 - Puntero this
- 5 Funciones y clases friend
- 6 El destructor
- 7 Constructores y destructores en clases con datos miembro de otras clases
- 8 El constructor de copia
 - El constructor de copia por defecto
 - Creación de un constructor de copia
 - Llamadas al constructor de copia
- 9 Llamadas a constructores y destructores
 - Llamadas al declarar variables locales y globales
 - Llamadas explícitas a un constructor
 - Listas de inicialización en constructores
 - Arrays de objetos
 - Creación/destrucción de objetos en memoria dinámica
 - Uso de constructores para hacer conversiones implícitas

Ejemplo (media de un array): usando Abstracción Funcional

```
// Arraydin_int.h
```

```
#ifndef _ARRAYDIN_INT_H
```

```
#define _ARRAYDIN_INT_H
```

```
void inicializar(int * &arrayint, int &nElementos);
```

```
void redimensionar (int* &arrayint, int& nElementos, int aumento);
```

```
void liberar(int * &arrayint, int &nElementos);
```

```
bool leer(int * &arrayint, int &nElementos);
```

```
void mostrar(const int *arrayint, int nElementos):
```

```
double sumar(const int *arrayint, int nElementos);
```

```
#endif
```

Ejemplo (media de un array): usando Abstracción Funcional

```
// Arraydin_int.cpp

#include <iostream>
using namespace std;

void inicializar(int * &arrayint, int &nElementos){
    arrayint=0;
    nElementos=0;
}

void redimensionar (int* &arrayint, int& nElementos, int aumento){
    if(nElementos+aumento > 0){
        int *v_ampliado = new int[nElementos+aumento];

        for (int i=0; (i<nElementos) && (i<nElementos+aumento); i++)
            v_ampliado[i] = arrayint[i];
        delete[] arrayint;
        arrayint = v_ampliado;
        nElementos+=aumento;
    }
}
```

Ejemplo (media de un array): usando Abstracción Funcional

```
void liberar(int * &arrayint, int &nElementos){
    delete[] arrayint;
    arrayint = 0;
    nElementos = 0;
}

bool leer(int * &arrayint, int &nElementos){
    int n;
    cin >> n;
    if(n<=0)
        return false;
    redimensionar(arrayint, nElementos, n);
    for(int i=0; i<nElementos; i++)
        cin >> arrayint[i];
    return true;
}
```

Ejemplo (media de un array): usando Abstracción Funcional

```
void mostrar(const int *arrayint, int nElementos){  
    for(int i=0; i<nElementos; i++)  
        std::cout << arrayint[i] << " ";  
    cout << endl;  
}  
  
double sumar(const int *arrayint, int nElementos){  
    double suma=0.0;  
    for(int i=0; i<nElementos; i++)  
        suma+=arrayint[i];  
    return suma;  
}
```


Ejemplo (media de un array): usando Abstracción Funcional

```
#include <iostream>
#include "Arraydin_int.h"

using namespace std;

int main(int argc, char* argv[]){
    int *arrayint;
    int nElementos;
    double suma;

    inicializar(arrayint, nElementos);
    if(leer(arrayint, nElementos)){
        mostrar(arrayint, nElementos);
        suma=sumar(arrayint, nElementos);
        cout << "Media=" << suma/nElementos << endl;
        liberar(arrayint, nElementos);
    }
    else
        cerr << "Array no válido" << endl;
}
```

Contenido del tema

- 1 **Introducción**
 - Abstracción funcional
 - **Abstracción de datos**
- 2 Clases con datos dinámicos
- 3 Los constructores
- 4 Los métodos de la clase
 - Métodos const
 - Métodos inline
 - Métodos modificadores de la interfaz básica
 - Métodos adicionales en la interfaz de la clase
 - Puntero this
- 5 Funciones y clases friend
- 6 El destructor
- 7 Constructores y destructores en clases con datos miembro de otras clases
- 8 El constructor de copia
 - El constructor de copia por defecto
 - Creación de un constructor de copia
 - Llamadas al constructor de copia
- 9 Llamadas a constructores y destructores
 - Llamadas al declarar variables locales y globales
 - Llamadas explícitas a un constructor
 - Listas de inicialización en constructores
 - Arrays de objetos
 - Creación/destrucción de objetos en memoria dinámica
 - Uso de constructores para hacer conversiones implícitas

Ejemplo (media de un array): usando Abstracción de Datos

Cambio de la representación del Tipo de Dato ArrayDinamico

Si alguna vez, cambiamos la representación de los datos del ArrayDinamico, **no será necesario** cambiar los proyectos que lo usan ya que los parámetros de las funciones del módulo no necesitan cambiarse:

```
#ifndef _ARRAYDIN_INT_H
#define _ARRAYDIN_INT_H

struct ArrayDinamico{
    int *arrayint;
    int nElementos;
    int reservados;
};

void inicializar(ArrayDinamico &arrayint);
void redimensionar (ArrayDinamico &arrayint, int aumento);
void liberar(ArrayDinamico &arrayint);
bool leer(ArrayDinamico &arrayint);
void mostrar(const ArrayDinamico &arrayint):
double sumar(const ArrayDinamico &arrayint);
#endif
```

Contenido del tema

- 1 Introducción
 - Abstracción funcional
 - Abstracción de datos
- 2 **Clases con datos dinámicos**
- 3 Los constructores
- 4 Los métodos de la clase
 - Métodos const
 - Métodos inline
 - Métodos modificadores de la interfaz básica
 - Métodos adicionales en la interfaz de la clase
 - Puntero this
- 5 Funciones y clases friend
- 6 El destructor
- 7 Constructores y destructores en clases con datos miembro de otras clases
- 8 El constructor de copia
 - El constructor de copia por defecto
 - Creación de un constructor de copia
 - Llamadas al constructor de copia
- 9 Llamadas a constructores y destructores
 - Llamadas al declarar variables locales y globales
 - Llamadas explícitas a un constructor
 - Listas de inicialización en constructores
 - Arrays de objetos
 - Creación/destrucción de objetos en memoria dinámica
 - Uso de constructores para hacer conversiones implícitas

La clase Polinomio

TDA Polinomio

- Construiremos una clase Polinomio para poder trabajar con polinomios del tipo:

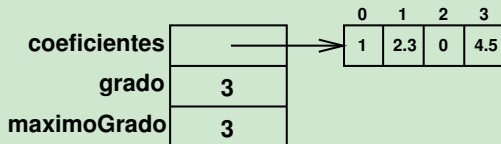
$$4,5 \cdot x^3 + 2,3 \cdot x + 1$$

- El número de coeficientes es desconocido a priori: usaremos **memoria dinámica**.
- Los datos miembro que usaremos para representar este tipo de dato son:
 - Grado del polinomio (0 si $p(x)=0$)
 - Coeficientes: lista con los coeficientes de cada monomio.
- Algunas operaciones que podrían definirse son: Suma, Multiplicación, Derivada, ...

La clase Polinomio

Implementación del TDA Polinomio: datos miembro (parte interna)

- **coeficientes**: *array dinámico* con los coeficientes, que permite polinomios de cualquier grado. Se requiere siempre que tenga una dimensión de **al menos un elemento**.
- **grado**: grado del polinomio. Es necesario para conocer qué parte del array dinámico estamos usando.
 - Vale 0 para un polinomio nulo.
- **maximoGrado**: indica el **máximo grado** posible (capacidad del array), es decir, el tamaño concreto del array de coeficientes.



La clase Polinomio

```

#ifndef POLINOMIO
#define POLINOMIO
#include <assert.h>
class Polinomio {
    private:
        // Array con los coeficientes del polinomio
        float *coeficientes;

        //Grado del polinomio
        int grado;

        //Máximo grado posible: limitación debida a la implementación
        //de la clase: el array de coeficientes tiene un tamaño limitado
        int maximoGrado;

    public:
        .....
};

#endif

```

La clase Polinomio

Implementación del TDA Polinomio: métodos (interfaz de la clase)

Conviene seguir el siguiente orden al definir los métodos de una clase:

- Constructores
- Operaciones naturales sobre los objetos de la clase (deberían ser métodos públicos)
- Otros métodos auxiliares que resulten convenientes (bien por la forma en que se ha hecho la implementación, bien por seguir el principio de descomposición modular....). A menudo, estos métodos serán privados.

Asumimos que cada vez que se agregue un método debe incorporarse a la declaración de la clase, en el archivo **Polinomio.h**.

Contenido del tema

- 1 Introducción
 - Abstracción funcional
 - Abstracción de datos
- 2 Clases con datos dinámicos
- 3 Los constructores**
- 4 Los métodos de la clase
 - Métodos const
 - Métodos inline
 - Métodos modificadores de la interfaz básica
 - Métodos adicionales en la interfaz de la clase
 - Puntero this
- 5 Funciones y clases friend
- 6 El destructor
- 7 Constructores y destructores en clases con datos miembro de otras clases
- 8 El constructor de copia
 - El constructor de copia por defecto
 - Creación de un constructor de copia
 - Llamadas al constructor de copia
- 9 Llamadas a constructores y destructores
 - Llamadas al declarar variables locales y globales
 - Llamadas explícitas a un constructor
 - Listas de inicialización en constructores
 - Arrays de objetos
 - Creación/destrucción de objetos en memoria dinámica
 - Uso de constructores para hacer conversiones implícitas

Los constructores de la clase Polinomio

Constructores de una clase

Los constructores se encargan de inicializar de forma conveniente los datos miembro.

En clases como **Polinomio**, deben además reservar la memoria dinámica que sea necesaria.

Constructor por defecto

Es el constructor sin parámetros. Una clase lo puede tener mediante:

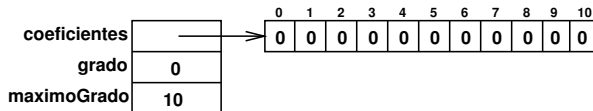
- El compilador lo crea **implícitamente** cuando la clase no define ningún constructor.
 - Tal constructor no inicializa los datos miembro de la clase (un dato miembro no inicializado probablemente contendrá un valor basura).
 - Solo llama al constructor por defecto de cada dato miembro que sea un objeto de otra clase.
- Definiéndolo **explícitamente** en la clase.

Los constructores de la clase Polinomio

Constructor por defecto de la clase Polinomio

Crea espacio para un polinomio de hasta grado 10. Cabe plantearse qué valores dar a los datos miembro:

- 10 para el grado máximo: entendemos que correspondería a un polinomio donde la variable apareciese elevada a 10 (x^{10})
- 0 para el grado
- los coeficientes deberían inicializarse todos a cero



Constructores: constructor por defecto

```
/**
 * Constructor por defecto de la clase. El trabajo de este
 * constructor se limita a crear un objeto nuevo, con
 * capacidad máxima para guardar once coeficientes
 */
Polinomio::Polinomio(){
    // Se inicializan los datos miembro maximoGrado y grado
    maximoGrado=10;
    grado=0;

    // Se reserva espacio para el array de coeficientes
    coeficientes=new float[maximoGrado+1];

    // Se inicializan todos los coeficientes a 0
    for(int i=0; i<=maximoGrado; i++){
        coeficientes[i]=0.0;
    }
}
```

```
#include <iostream>
#include "Polinomio.h"
using namespace std;

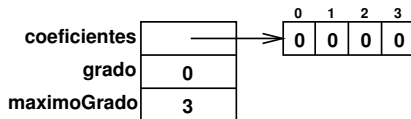
int main(){
    Polinomio pol1; // Polinomio creado con constructor sin parámetros
    ...
}
```

Los constructores de la clase Polinomio

Polinomio: Constructor con un parámetro que indica el grado máximo

Crea espacio para un polinomio con tamaño justo para que quepa un polinomio del grado máximo indicado.

Como en el constructor previo, el dato miembro grado se inicializa a 0 y los coeficientes toman también este valor



Constructores: constructor con valor de máximo grado

```

/**
 * Constructor de la clase indicando el máximo grado posible
 * @param maximoGrado valor del grado máximo
 */
Polinomio::Polinomio(int maximoGrado){
    // Si máximo grado es negativo se hace que el programa finalice
    assert(maximoGrado>=0);

    // Si el valor de maximoGrado es correcto, se asigna su
    // valor al dato miembro
    this->maximoGrado=maximoGrado;

    // Se inicializa a 0 el valor de grado
    grado=0;

    // Se reserva espacio para el array de coeficientes
    coeficientes=new float[maximoGrado+1];

    // Se inicializan a valor 0
    for(int i=0; i <= maximoGrado; i++){
        coeficientes[i]=0.0;
    }
}

```



```
#include <iostream>
#include "Polinomio.h"
using namespace std;

int main(){
    Polinomio pol1; // Polinomio creado con constructor sin parámetros
    Polinomio pol2(20); // Polinomio creado con constructor Polinomio(int)
    ...
}
```

Constructores

Código común en constructores

A menudo, varios constructores comparten un trozo de código, cuya repetición puede evitarse con un **método auxiliar** que habitualmente será privado.

```
Polinomio::Polinomio(){
    maximoGrado=10;
    grado=0;

    coeficientes=new float[maximoGrado+1];

    for(int i=0; i<=maximoGrado; i++){
        coeficientes[i]=0.0;
    }
}
```

```
Polinomio::Polinomio(int maximoGrado){
    assert(maximoGrado>=0);
```

```
    this->maximoGrado=maximoGrado;
```

```
    grado=0;
```

```
    coeficientes=new float[maximoGrado+1];
```

```
    for(int i=0; i <= maximoGrado; i++){
        coeficientes[i]=0.0;
    }
}
```

Constructores

Método auxiliar usado en los constructores de Polinomio

Añadimos el método privado inicializar() a la clase Polinomio.

```
/**
 * Método privado para inicializar el valor de grado y para
 * crear array de coeficientes de tamaño dado por el valor
 * de maximoGrado (más uno), poniéndolos todos a cero
 */
void Polinomio::inicializar() {
    // Se inicializa a 0 el valor de grado
    grado = 0;

    // Se reserva espacio para el array de coeficientes
    coeficientes = new float[maximoGrado + 1];

    // Se inicializan a valor 0
    for (int i = 0; i <= maximoGrado; i++) {
        coeficientes[i] = 0.0;
    }
}
```

Constructores

Los constructores quedarían ahora:

```
Polinomio::Polinomio(){
    maximoGrado=10;

    inicializar();
}
```

```
Polinomio::Polinomio(int maximoGrado){
    assert(maximoGrado>=0);

    this->maximoGrado=maximoGrado;

    inicializar();
}
```

```
#include <iostream>
#include "Polinomio.h"
using namespace std;

int main(){
    Polinomio pol1; // Polinomio creado con constructor sin parámetros
    Polinomio pol2(20); // Polinomio creado con constructor Polinomio(int)
    ...
}
```

Constructores

De momento, la declaración de la clase contendría:

```
#ifndef POLINOMIO
#define POLINOMIO
#include <assert.h>
class Polinomio {
    private:
        float *coeficientes; // Array con los coeficientes del polinomio
        int grado; //Grado del polinomio

        // Máximo grado posible: limitación debida a la implementación
        //de la clase: el array de coeficientes tiene un tamaño limitado
        int maximoGrado;

        // Método inicializar para facilitar la programación de los constructores
        void inicializar();

    public:
        Polinomio(); // Constructor por defecto
        Polinomio(int gradoMaximo); // Constructor indicando el grado máximo
};

#endif
```

Constructores

Definición de constructores usando parámetros por defecto

Observando los dos constructores, se aprecia que la única diferencia consiste en la asignación explícita de valor al dato miembro `maximoGrado`:

- Podemos usar un **parámetro por defecto** para definir los dos constructores con uno solo.

```
.....
class Polinomio{
    .....
    /**
     * Constructor indicando el máximo grado posible
     * @param maximoGrado valor del grado máximo
     */
    Polinomio(int maximoGrado = 10);
    .....
}
```

Constructores

Definición de constructores usando parámetros por defecto

Observando los dos constructores, se aprecia que la única diferencia consiste en la asignación explícita de valor al dato miembro `maximoGrado`:

- Podemos usar un **parámetro por defecto** para definir los dos constructores con uno solo.

```
.....
class Polinomio{
    .....
    /**
     * Constructor indicando el máximo grado posible
     * @param maximoGrado valor del grado máximo
     */
    Polinomio(int maximoGrado = 10);
    .....
}
```

Constructores

```
/**
 * Constructor de la clase indicando el máximo grado posible
 * @param maximoGrado valor del grado máximo
 */
Polinomio::Polinomio(int maximoGrado) {
    // Si máximo grado es negativo se hace que el programa
    // finalice
    assert(maximoGrado >= 0);

    // Si el valor de maximoGrado es correcto, se asigna su
    // valor al dato miembro
    this->maximoGrado = maximoGrado;

    // Se inicializan los demás datos miembro
    inicializar();
}
```


Contenido del tema

- 1 Introducción
 - Abstracción funcional
 - Abstracción de datos
- 2 Clases con datos dinámicos
- 3 Los constructores
- 4 Los métodos de la clase
 - Métodos const
 - Métodos inline
 - Métodos modificadores de la interfaz básica
 - Métodos adicionales en la interfaz de la clase
 - Puntero this
- 5 Funciones y clases friend
- 6 El destructor
- 7 Constructores y destructores en clases con datos miembro de otras clases
- 8 El constructor de copia
 - El constructor de copia por defecto
 - Creación de un constructor de copia
 - Llamadas al constructor de copia
- 9 Llamadas a constructores y destructores
 - Llamadas al declarar variables locales y globales
 - Llamadas explícitas a un constructor
 - Listas de inicialización en constructores
 - Arrays de objetos
 - Creación/destrucción de objetos en memoria dinámica
 - Uso de constructores para hacer conversiones implícitas

Métodos del interfaz básico

Métodos del interfaz básico

- Deberían ser **pocos**: definen la funcionalidad básica.
- Suelen utilizar directamente los datos miembro de la clase.
- En el conjunto de operaciones distinguimos entre **métodos consultores** y **métodos modificadores**.
IMPORTANTE: No tiene por qué haber un método consultor y modificador por cada dato miembro.

Métodos del interfaz adicional

Métodos del interfaz adicional

- Facilitan el uso del tipo de dato abstracto.
- No deberían extenderse demasiado.
- Aunque se implementen como métodos, no es conveniente que accedan directamente a los datos miembro de la clase, ya que un cambio en la representación del TDA supondría cambiar todos los métodos adicionales.

Contenido del tema

- 1 Introducción
 - Abstracción funcional
 - Abstracción de datos
- 2 Clases con datos dinámicos
- 3 Los constructores
- 4 **Los métodos de la clase**
 - **Métodos const**
 - Métodos inline
 - Métodos modificadores de la interfaz básica
 - Métodos adicionales en la interfaz de la clase
 - Puntero this
- 5 Funciones y clases friend
- 6 El destructor
- 7 Constructores y destructores en clases con datos miembro de otras clases
- 8 El constructor de copia
 - El constructor de copia por defecto
 - Creación de un constructor de copia
 - Llamadas al constructor de copia
- 9 Llamadas a constructores y destructores
 - Llamadas al declarar variables locales y globales
 - Llamadas explícitas a un constructor
 - Listas de inicialización en constructores
 - Arrays de objetos
 - Creación/destrucción de objetos en memoria dinámica
 - Uso de constructores para hacer conversiones implícitas

Métodos const

Métodos const

Los métodos consultores no modifican el objeto sobre el que se llaman, por lo que se declararán de forma especial para remarcar esta característica: **métodos const**.

- Esto impide que accidentalmente incluyamos en tales métodos alguna sentencia que modifique algún dato miembro de la clase.
- Además, permite que sean utilizados con objetos declarados como *constantes*.

Métodos const

```
/**
 * Obtiene el grado del objeto
 * @return grado
 */
int Polinomio::getGrado() const {
    return grado;
}

/**
 * Permite acceder a los coeficientes del objeto.
 * @param indice asociado al coeficiente
 * @return coeficiente solicitado
 */
float Polinomio::getCoeficiente(int indice) const {
    float salida = 0.0;
    // Se comprueba si el índice es menor o igual que el grado
    if (indice >= 0 && indice <= grado){
        salida = coeficientes[indice];
    }
    return salida;
}
```

Métodos const

```
#include <iostream>
#include "Polinomio.h"
using namespace std;

int main(){
    Polinomio pol1; // Polinomio creado con constructor sin parámetros
    ...
    cout << "Grado del polinomio: " << pol1.getGrado() << endl;
    cout << "Coeficientes del polinomio: "
    for(int i=0; i<=pol1.getGrado(); i++){
        cout << i << ": " << pol1.getCoeficiente(i) << " ";
    }
    cout << endl;
}
```

Contenido del tema

- 1 Introducción
 - Abstracción funcional
 - Abstracción de datos
- 2 Clases con datos dinámicos
- 3 Los constructores
- 4 **Los métodos de la clase**
 - Métodos const
 - Métodos inline
 - **Métodos modificadores de la interfaz básica**
 - Métodos adicionales en la interfaz de la clase
 - Puntero this
- 5 Funciones y clases friend
- 6 El destructor
- 7 Constructores y destructores en clases con datos miembro de otras clases
- 8 El constructor de copia
 - El constructor de copia por defecto
 - Creación de un constructor de copia
 - Llamadas al constructor de copia
- 9 Llamadas a constructores y destructores
 - Llamadas al declarar variables locales y globales
 - Llamadas explícitas a un constructor
 - Listas de inicialización en constructores
 - Arrays de objetos
 - Creación/destrucción de objetos en memoria dinámica
 - Uso de constructores para hacer conversiones implícitas

Métodos modificadores de la interfaz básica

Métodos modificadores

Modifican el valor de alguno de los datos miembro de un objeto.

Modificadores de la clase Polinomio

No incluiremos un método para modificar el grado, pues este se determinará en base a los coeficientes del polinomio.

Incluiremos un método que permita asignar valores a los coeficientes:

- **setCoeficiente(int indice, float coeficiente)**: permite asignar el coeficiente asociado a un determinado término.

Métodos modificadores de la interfaz básica

Analizando qué debe hacer este método de asignación de coeficientes, encontramos cuatro situaciones diferentes:

- Si el índice pasado como argumento es mayor que el máximo grado, hay que **reservar más espacio de memoria** para los coeficientes, al excederse la capacidad de almacenamiento previo.
- Si el índice es mayor al actual grado y el nuevo coeficiente no es cero, hay que actualizar el grado.
- Si el índice coincide con el máximo grado y el nuevo coeficiente es cero, entonces hay que **determinar el nuevo grado** del polinomio analizando los coeficientes.
- En la situación normal, basta con asignar el correspondiente coeficiente.

Métodos modificadores de la interfaz básica

```
void Polinomio::setCoeficiente(int i, float c){
    if(i>=0){ // Si el índice del coeficiente es válido
        if(i>maximoGrado){ // Si necesitamos más espacio
            float *aux=new float[i+1]; // Reservamos nueva memoria
            for(int j=0;j<=grado;++j) // Copiamos coeficientes a nueva memoria
                aux[j]=coeficientes[j];
            delete[] coeficientes; // Liberamos memoria antigua
            coeficientes=aux; // Reasignamos puntero de coeficientes
            for(int j=grado+1;j<=i;++j) //Hacemos 0 el resto de nuevos coeficientes
                coeficientes[j]=0.0;
            maximoGrado=i; // Asignamos el nuevo número máximo grado del polinomio
        }
        coeficientes[i]=c; // Asignamos el nuevo coeficiente

        // actualizamos el grado
        if(c!=0.0 && i>grado){ //Si coeficiente!=0 e índice coeficiente>antiguo grado
            grado=i; // lo actualizamos al valor i
        }
        else if(c==0.0 && i==grado){ //Si coeficiente==0.0 e índice coeficiente==grado
            while(coeficientes[grado]==0.0 && grado>0){ //Actualizamos grado con el
                grado--; //primer término cuyo coeficiente no sea cero
            }
        }
    }
}
```

Métodos modificadores de la interfaz básica

```
#include <iostream>
#include "Polinomio.h"
using namespace std;

int main(){
    Polinomio pol1(2); // Polinomio creado con constructor Polinomio(int)
    pol1.setCoeficiente(0, 1.0);
    pol1.setCoeficiente(1, 2.3);
    pol1.setCoeficiente(3, 4.5); // requiere redimensionamiento
    ...
    cout << "Grado del polinomio: " << pol1.getGrado() << endl;
    cout << "Coeficientes del polinomio: "
    for(int i=0; i<=pol1.getGrado(); i++){
        cout << i << ": " << pol1.getCoeficiente(i) << " ";
    }
    cout << endl;
}
```

Contenido del tema

- 1 Introducción
 - Abstracción funcional
 - Abstracción de datos
- 2 Clases con datos dinámicos
- 3 Los constructores
- 4 **Los métodos de la clase**
 - Métodos const
 - Métodos inline
 - Métodos modificadores de la interfaz básica
 - Métodos adicionales en la interfaz de la clase
 - **Puntero this**
- 5 Funciones y clases friend
- 6 El destructor
- 7 Constructores y destructores en clases con datos miembro de otras clases
- 8 El constructor de copia
 - El constructor de copia por defecto
 - Creación de un constructor de copia
 - Llamadas al constructor de copia
- 9 Llamadas a constructores y destructores
 - Llamadas al declarar variables locales y globales
 - Llamadas explícitas a un constructor
 - Listas de inicialización en constructores
 - Arrays de objetos
 - Creación/destrucción de objetos en memoria dinámica
 - Uso de constructores para hacer conversiones implícitas

Puntero this

Puntero this

Desde los métodos (o constructores) de una clase, disponemos de un puntero que apunta al objeto usado para la llamada: **puntero this**.

- Puede usarse para:
 - Referenciar un dato miembro del objeto apuntado.
 - Llamar a un método de instancia del objeto apuntado.
- Solo es necesario usarlo en caso de que un dato miembro haya sido ocultado por un parámetro de un método. En otros casos es opcional.

Puntero this

```

class Polinomio{
    ...
public:
    Polinomio sumar(const Polinomio &pol) const;
}

Polinomio Polinomio::sumar(const Polinomio &pol) const{
    int gmax=(this->grado>pol.grado)?this->grado:pol.grado;
    int gmin=(this->grado<pol.grado)?this->grado:pol.grado;
    Polinomio res(gmax);
    for(int i=0;i<=gmin;++i) // asignar suma de coeficientes comunes
        res.setCoeficiente(i,this->coeficientes[i]+pol.coeficientes[i]);
    for(int i=gmin+1;i<=gmax;++i) // asignar resto de coeficientes
        res.setCoeficiente(i,
            (this->grado<pol.grado)?pol.coeficientes[i]:this->coeficientes[i]);
    return res;
}

int main(){
    Polinomio p1(3), p2;
    p1.setCoeficiente(3,4.5);
    p1.setCoeficiente(1,2.3);
    ...
    Polinomio p3=p1.sumar(p2);
}

```

The diagram illustrates the state of memory during the execution of the `sumar` method. It shows two frames: **Dentro de sumar()** and **Dentro de main()**.

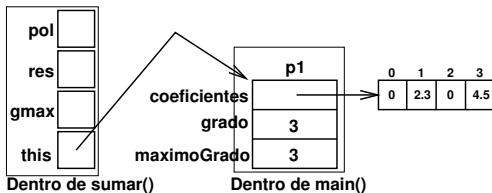
- Dentro de sumar():** Contains local variables `pol`, `res`, `gmax`, and `this`. The `this` pointer points to the `p1` object in the `main` frame.
- Dentro de main():** Contains the `p1` object and the `maximoGrado` variable. The `p1` object has three fields: `coeficientes` (an array), `grado` (3), and `maximoGrado` (3). The `coeficientes` array is shown with values [0, 2.3, 0, 4.5] for indices 0, 1, 2, and 3 respectively.

Puesto que sumar() puede considerarse un método de la interfaz adicional, es mejor que no acceda directamente a los datos miembro de la clase.

```
class Polinomio{
    ...
public:
    Polinomio sumar(const Polinomio &pol) const;
}

Polinomio Polinomio::sumar(const Polinomio &pol) const{
    int gmax=(this->getGrado()>pol.getGrado())?
        this->getGrado():pol.getGrado();
    Polinomio res(gmax);
    for(int i=0;i<=gmax;++i){
        res.setCoeficiente(i,
            this->getCoeficiente(i) + pol.getCoeficiente(i));
    }
    return res;
}

int main(){
    Polinomio p1(3),p2;
    p1.setCoeficiente(3,4.5);
    p1.setCoeficiente(1,2.3);
    ...
    Polinomio p3=p1.sumar(p2);
}
```



Destrucción automática de objetos locales

Destrucción automática de variables locales

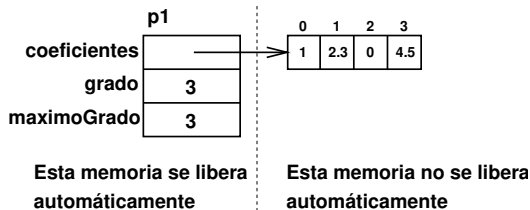
Las variables locales se destruyen automáticamente al finalizar la función en la que se definen, según se ha visto anteriormente.

Ejemplo

En el siguiente código, `p1` es una variable local de `main()`: se destruirá automáticamente al acabar `funcion()`.

Pero, ¿qué ocurre con la memoria dinámica reservada por el constructor?

```
int main() {
    Polinomio p1(3);
    ...
    p1.imprimir();
}
```



¿Cómo liberar la memoria dinámica del objeto?

Una mala solución para liberar la MD de un objeto

Hacer un método para liberar la memoria dinámica del objeto y llamarlo explícitamente antes de que se destruya el objeto.

```
class Polinomio{
    ...
public:
    ...
    void liberar();
};
...
void Polinomio::liberar(){
    delete[] coeficientes;
    coeficientes = 0;
    grado=0;
    maximoGrado=-1;
}

int main() {
    Polinomio p1(3);
    ...
    p1.imprimir();
    ...
    p1.liberar();
}
```

El destructor de la clase

Destructor de una clase

Automatiza el proceso de destrucción.

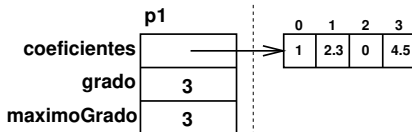
- El destructor es único, no lleva parámetros y no devuelve nada.
- Se ejecuta de forma automática, en el momento de destruir un objeto de la clase:
 - Los objetos locales a una función o trozo de código, justo antes de acabar la función o trozo de código.
 - Los objetos variable global, justo antes de acabar el programa.
 - Los objetos pasados por valor a una función, justo antes de acabar la función.

El destructor de la clase

Destructor de la clase Polinomio

```
class Polinomio {
    ...
public:
    ...
    ~Polinomio();
};
Polinomio::~~Polinomio()
{
    delete[] coeficientes;
}
```

```
int main() {
    Polinomio p1(3);
    ...
    p1.imprimir();
    ...
} // Aquí se destruirá automáticamente el
  // objeto p1 (MD incluida)
```



Esta memoria se libera automáticamente

Esta memoria la libera el destructor

Ejemplo de llamadas al destructor

Al ejecutar el siguiente ejemplo, puede verse en qué momento se llama al constructor y destructor de la clase.

```
#include <iostream>
using namespace std;
class Cotilla{
public:
    Cotilla();
    ~Cotilla();
};
Cotilla::Cotilla(){
    cout<<"Constructor"<<endl;
}
Cotilla::~Cotilla(){
    cout<<"Destructor"<<endl;
}
void funcion(){
    Cotilla local;
    cout<<"funcion()"<<endl;
}
Cotilla varGlobal;
int main(){
    cout<<"Comienza main()"<<endl;
    Cotilla ppal;
    cout<<"Antes de llamar a funcion()"<<endl;
    funcion();
    cout<<"Después de llamar a funcion()"<<endl;
    cout<<"Termina main()"<<endl;
}
```

En la traza se han agregado comentarios para aclarar en qué momento se genera cada línea.

```
Constructor // Construcción objeto varGlobal
Comienza main() // Inicio ejecucion main()
Constructor // Construcción objeto ppal
Antes de llamar a funcion()
Constructor // Construcción objeto local de funcion()
funcion() // Ejecución de funcion()
Destructor // Se destruye objeto local (en el ámbito de funcion() )
Después de llamar a funcion() // De vuelta en main()
Termina main() // Finaliza ejecución main()
Destructor // Se destruye objeto ppal
Destructor // Se destruye objeto varGlobal
```

Contenido del tema

- 1 Introducción
 - Abstracción funcional
 - Abstracción de datos
- 2 Clases con datos dinámicos
- 3 Los constructores
- 4 Los métodos de la clase
 - Métodos const
 - Métodos inline
 - Métodos modificadores de la interfaz básica
 - Métodos adicionales en la interfaz de la clase
 - Puntero this
- 5 Funciones y clases friend
- 6 El destructor
- 7 Constructores y destructores en clases con datos miembro de otras clases
- 8 El constructor de copia
 - El constructor de copia por defecto
 - Creación de un constructor de copia
 - Llamadas al constructor de copia
- 9 Llamadas a constructores y destructores
 - Llamadas al declarar variables locales y globales
 - Llamadas explícitas a un constructor
 - Listas de inicialización en constructores
 - Arrays de objetos
 - Creación/destrucción de objetos en memoria dinámica
 - Uso de constructores para hacer conversiones implícitas

Contenido del tema

- 1 Introducción
 - Abstracción funcional
 - Abstracción de datos
- 2 Clases con datos dinámicos
- 3 Los constructores

- 4 Los métodos de la clase
 - Métodos const
 - Métodos inline
 - Métodos modificadores de la interfaz básica
 - Métodos adicionales en la interfaz de la clase
 - Puntero this

- 5 Funciones y clases friend
- 6 El destructor

- 7 Constructores y destructores en clases con datos miembro de otras clases
- 8 **El constructor de copia**
 - **El constructor de copia por defecto**
 - Creación de un constructor de copia
 - Llamadas al constructor de copia
- 9 Llamadas a constructores y destructores
 - Llamadas al declarar variables locales y globales
 - Llamadas explícitas a un constructor
 - Listas de inicialización en constructores
 - Arrays de objetos
 - Creación/destrucción de objetos en memoria dinámica
 - Uso de constructores para hacer conversiones implícitas

Contenido del tema

- 1 Introducción
 - Abstracción funcional
 - Abstracción de datos
- 2 Clases con datos dinámicos
- 3 Los constructores
- 4 Los métodos de la clase
 - Métodos const
 - Métodos inline
 - Métodos modificadores de la interfaz básica
 - Métodos adicionales en la interfaz de la clase
 - Puntero this
- 5 Funciones y clases friend
- 6 El destructor
- 7 Constructores y destructores en clases con datos miembro de otras clases
- 8 **El constructor de copia**
 - El constructor de copia por defecto
 - **Creación de un constructor de copia**
 - Llamadas al constructor de copia
- 9 Llamadas a constructores y destructores
 - Llamadas al declarar variables locales y globales
 - Llamadas explícitas a un constructor
 - Listas de inicialización en constructores
 - Arrays de objetos
 - Creación/destrucción de objetos en memoria dinámica
 - Uso de constructores para hacer conversiones implícitas

Creación de un constructor de copia

Creación de un constructor de copia

Es posible crear nuestro propio constructor de copia que haga una **copia correcta** de un objeto de la clase en otro.

```
class Polinomio {  
    ...  
    public:  
        ...  
        Polinomio(const Polinomio &p);  
};
```

- Al ser un constructor, tiene el mismo nombre que la clase.
- No devuelve nada y tiene como único parámetro, constante y por referencia, el objeto de la clase que se quiere copiar.
- Copia el objeto que se pasa como parámetro en el objeto que construye el constructor.
- Se llama automáticamente al hacer un paso por valor para copiar el parámetro actual en el parámetro formal.

Creación de un constructor de copia

Solución correcta: implementación del constructor de copia en Polinomio

```
class Polinomio {  
    ...  
public:  
    ...  
    Polinomio(const Polinomio &p);  
};  
  
Polinomio::Polinomio(const Polinomio &p){  
    maximoGrado=p.maximoGrado;  
    grado=p.grado;  
    coeficientes=new float[maximoGrado+1];  
    for(int i=0; i<=maximoGrado; ++i)  
        coeficientes[i]=p.coeficientes[i];  
}
```

Contenido del tema

- 1 Introducción
 - Abstracción funcional
 - Abstracción de datos
- 2 Clases con datos dinámicos
- 3 Los constructores
- 4 Los métodos de la clase
 - Métodos const
 - Métodos inline
 - Métodos modificadores de la interfaz básica
 - Métodos adicionales en la interfaz de la clase
 - Puntero this
- 5 Funciones y clases friend
- 6 El destructor
- 7 Constructores y destructores en clases con datos miembro de otras clases
- 8 El constructor de copia
 - El constructor de copia por defecto
 - Creación de un constructor de copia
 - Llamadas al constructor de copia
- 9 **Llamadas a constructores y destructores**
 - Llamadas al declarar variables locales y globales
 - Llamadas explícitas a un constructor
 - Listas de inicialización en constructores
 - Arrays de objetos
 - **Creación/destrucción de objetos en memoria dinámica**
 - Uso de constructores para hacer conversiones implícitas

Creación/destrucción de objetos en memoria dinámica

Creación/destrucción de objetos en memoria dinámica

Según vimos en el tema 2:

- Operador `new`:
 - **Reserva la memoria** necesaria para almacenar todos y cada uno de los datos del objeto.
 - Y llama al **constructor** de la clase para inicializar los datos del objeto.
- Operador `delete`:
 - Llama al **destructor** de la clase.
 - Después **libera la memoria** de todos los datos del objeto.

```
int main(){
    Polinomio *pol;
    pol = new Polinomio; // Reserva de MD para un objeto Polinomio
                        // y llamada al constructor por defecto
    ...
    delete pol; // Llamada al destructor y liberación de la MD
}
```

Creación/destrucción de objetos en memoria dinámica

Uso de otros constructores con new

Podemos usar el constructor deseado.

```
int main(){
    Polinomio *pol1, *pol2;
    pol1 = new Polinomio(3); // Reserva de MD para un objeto Polinomio
                           // y llamada al constructor Polinomio(int)
    ...
    pol2 = new Polinomio(*pol1); // Reserva de MD para un objeto Polinomio
                                // y llamada al constructor de copia
    ...
    delete pol1; // Llamada al destructor y liberación de la MD
    delete pol2;
}
```

Creación/destrucción de objetos en memoria dinámica

Array dinámico de objetos

También vimos en el tema 2, que usando los operadores `new[]` y `delete[]` podemos crear y destruir arrays dinámicos de objetos.

- Operador `new[]`:
 - **Reserva la memoria** necesaria para almacenar todos y cada uno de los objetos del array.
 - Y llama al **constructor** para cada objeto del array.
- Operador `delete[]`:
 - Llama al **destructor** de la clase con cada objeto del array.
 - Y después **libera la memoria** ocupada por el array de objetos.

Creación/destrucción de objetos en memoria dinámica

```
int main(){
    Polinomio *arrayP;
    arrayP = new Polinomio[100]; // Reserva de MD para 100 polinomios y
                                // llamada al constructor por defecto
                                // para cada uno

    ...
    delete[] arrayP; // Llamada al destructor para cada uno de los
                    // 100 polinomios y liberación de la MD ocupada
                    // por los 100 polinomios
}
```

Metodología de la Programación

Tema 5. Clases II: Sobrecarga de operadores

Andrés Cano Utrera
(acu@decsai.ugr.es)
Departamento de Ciencias de la Computación e I.A.



Curso 2020-2021

Contenido del tema

- 1 **Introducción a la sobrecarga de operadores**
- 2 Mecanismos de sobrecarga de operadores
 - Sobrecarga como función externa: Ejemplo operator+
 - Sobrecarga como función miembro: Ejemplo operator+
 - Sobrecarga de operadores como función miembro o externa
- 3 El operador de asignación
- 4 La clase mínima
- 5 Operadores << y >>
 - Sobrecarga del operador <<
 - Sobrecarga del operador >>
 - Sobrecarga del operador << con una función amiga
- 6 Operador de indexación
- 7 Operadores de asignación compuestos
- 8 Operadores relacionales
- 9 Operadores de incremento y decremento
- 10 Operador de llamada a función

Introducción a la sobrecarga de operadores

- C++ permite **sobrecargar** casi todos sus operadores en nuestras propias clases, para que podamos usarlos con los objetos de tales clases.
- Para ello, definiremos un método o una función cuyo nombre estará compuesto de la palabra `operator` junto con el operador correspondiente. Ejemplo: `operator+()`.
- Esto permitirá usar la siguiente sintaxis para hacer cálculos con objetos de nuestras propias clases:

```
Polinomio p, q, r;  
// ...  
r= p+q;
```

- No puede modificarse la sintaxis de los operadores (número de operandos, precedencia y asociatividad).
- No deberíamos tampoco modificar la semántica de los operadores.

Operadores que pueden sobrecargarse

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

- Los operadores que no pueden sobrecargarse son:

.	.*	::	?:	sizeof
---	----	----	----	--------

- Al sobrecargar un operador no se sobrecargan automáticamente operadores relacionados.

Por ejemplo, al sobrecargar + no se sobrecarga automáticamente +=, ni al sobrecargar == lo hace automáticamente !=.

Contenido del tema

- 1 Introducción a la sobrecarga de operadores
- 2 Mecanismos de sobrecarga de operadores
 - Sobrecarga como función externa: Ejemplo operator+
 - Sobrecarga como función miembro: Ejemplo operator+
 - Sobrecarga de operadores como función miembro o externa
- 3 El operador de asignación
- 4 La clase mínima
- 5 Operadores << y >>
 - Sobrecarga del operador <<
 - Sobrecarga del operador >>
 - Sobrecarga del operador << con una función amiga
- 6 Operador de indexación
- 7 Operadores de asignación compuestos
- 8 Operadores relacionales
- 9 Operadores de incremento y decremento
- 10 Operador de llamada a función

Sobrecarga como función externa

Sobrecarga como función externa

Consiste en añadir una función externa a la clase, que recibirá dos objetos (o uno para operadores unarios) de la clase y devolverá el resultado de la operación.

Polinomio `operator+(const Polinomio &p1, const Polinomio &p2);`

- Cuando el compilador encuentre una expresión tal como `p+q`, la interpretará como una llamada a la función `operator+(p,q)`
- Incluso podríamos sobrecargar el operador aunque los dos operandos sean de tipos distintos:

- Suma de Polinomio con float: `pol+3.5`

Polinomio `operator+(const Polinomio &p1, float f);`

- Suma de float con Polinomio: `3.5+pol`

Polinomio `operator+(float f, const Polinomio &p1);`

Sobrecarga como función externa

```
Polinomio operator+(const Polinomio &p1,const Polinomio &p2){
    int gmax=(p1.getGrado()>p2.getGrado())?
        p1.getGrado():p2.getGrado();
    Polinomio resultado(gmax);
    for(int i=0;i<=gmax;++i){
        resultado.setCoeficiente(i,
            p1.getCoeficiente(i)+p2.getCoeficiente(i));
    }
    return resultado;
}

int main(){
    Polinomio p1, p2, p3;
    ... // dar valores a coeficientes de p2 y p3
    p1 = p2 + p3; // equivalente a p1 = operator+(p2,p3);
}
```

Contenido del tema

- 1 Introducción a la sobrecarga de operadores
- 2 Mecanismos de sobrecarga de operadores
 - Sobrecarga como función externa: Ejemplo operator+
 - Sobrecarga como función miembro: Ejemplo operator+
 - Sobrecarga de operadores como función miembro o externa
- 3 El operador de asignación
- 4 La clase mínima
- 5 Operadores << y >>
 - Sobrecarga del operador <<
 - Sobrecarga del operador >>
 - Sobrecarga del operador << con una función amiga
- 6 Operador de indexación
- 7 Operadores de asignación compuestos
- 8 Operadores relacionales
- 9 Operadores de incremento y decremento
- 10 Operador de llamada a función

Sobrecarga como función miembro

Sobrecarga como función miembro

Consiste en añadir un método a la clase, que recibirá un objeto (o ninguno para operadores unarios) de la clase y devolverá el resultado de la operación.

```
Polinomio Polinomio::operator+(const Polinomio &p) const;
```

- Cuando el compilador encuentre una expresión tal como `p+q` la interpretará como una llamada al método `p.operator+(q)`
- También podríamos sobrecargar así el operador con un operando de tipo distinto:
 - Suma de Polinomio con float: `pol+3.5`

```
Polinomio Polinomio::operator+(float f) const;
```
 - Sin embargo no es posible definir así el operador para usarlo con expresiones del tipo: `3.5+pol`

Sobrecarga como función miembro

```
Polinomio Polinomio::operator+(const Polinomio &pol) const{
    int gmax=(this->getGrado()>pol.getGrado())?
        this->getGrado():pol.getGrado();
    Polinomio resultado(gmax);
    for(int i=0;i<=gmax;++i){
        resultado.setCoeficiente(i,
            this->getCoeficiente(i)+pol.getCoeficiente(i));
    }
    return resultado;
}

int main(){
    Polinomio p1, p2, p3;
    ... // dar valores a coeficientes de p2 y p3
    p1 = p2 + p3; // equivalente a p1 = p2.operator+(p3);
}
```

Contenido del tema

- 1 Introducción a la sobrecarga de operadores
- 2 Mecanismos de sobrecarga de operadores
 - Sobrecarga como función externa:
Ejemplo operator+
 - Sobrecarga como función miembro:
Ejemplo operator+
 - Sobrecarga de operadores como función miembro o externa
- 3 El operador de asignación
- 4 La clase mínima
- 5 Operadores << y >>
 - Sobrecarga del operador <<
 - Sobrecarga del operador >>
 - Sobrecarga del operador << con una función amiga
- 6 Operador de indexación
- 7 Operadores de asignación compuestos
- 8 Operadores relacionales
- 9 Operadores de incremento y decremento
- 10 Operador de llamada a función

Sobrecarga como función miembro o externa

- La sobrecarga de un operador con una **función miembro** puede hacerse si tenemos acceso al código fuente de la clase y el primer operando es del tipo de la clase.

Ejemplo: Para sumar dos polinomios, podemos sobrecargar `operator+` en la clase `Polinomio` con una función miembro, pues tenemos acceso a su implementación.

```
Polinomio Polinomio::operator+(const Polinomio &pol) const{
    ...
}
int main(){
    Polinomio p1, p2, p3;
    ... // dar valores a coeficientes de p2 y p3
    p1 = p2 + p3; // equivalente a p1 = p2.operator+(p3);
}
```

- El lenguaje obliga a que los operadores `()`, `[]`, `->` y el operador de asignación, sean implementados como **funciones miembro**.

Sobrecarga como función miembro o externa

- Si el primer operando debe ser un objeto de una clase diferente, debemos sobrecargarlo como **función externa**.

Ejemplo: El operador + para concatenar un string con un Polinomio lo implementaremos con una función externa.

```
string operator+(const string& cadena, const Polinomio& pol){  
    ...  
}
```

```
int main(){  
    Polinomio p;  
    string s1="Polinomio: ", s2;  
    ...  
    s2 = s1 + p; // equivale a s2 = operator+(s1, p);  
}
```

- También, si el primer operando debe ser un dato de un tipo primitivo, debemos sobrecargarlo como **función externa**.

```
Polinomio operator+(int i, const Polinomio& pol){  
    ...  
}  
  
int main(){  
    Polinomio p1, p2;  
    int i;  
    ... // dar valores a coeficientes de p1 y p2  
    p1 = i + p2; // equivale a p1 = operator+(i, p2);  
}
```

Sobrecarga como función miembro o externa

Directrices para elegir entre miembro y no-miembro

Según el libro de Rob Murray, C++ Strategies & Tactics , Addison Wesley, 1993, página 47.

Operador	Uso recomendado
Todos los operadores unarios	miembro
= () [] -> ->*	debe ser miembro
+= -= /= *= ^= &= = %= >>= <<=	miembro
El resto de operadores binarios	no miembro

Contenido del tema

- 1 Introducción a la sobrecarga de operadores
- 2 Mecanismos de sobrecarga de operadores
 - Sobrecarga como función externa: Ejemplo operator+
 - Sobrecarga como función miembro: Ejemplo operator+
 - Sobrecarga de operadores como función miembro o externa
- 3 El operador de asignación**
- 4 La clase mínima
- 5 Operadores << y >>
 - Sobrecarga del operador <<
 - Sobrecarga del operador >>
 - Sobrecarga del operador << con una función amiga
- 6 Operador de indexación
- 7 Operadores de asignación compuestos
- 8 Operadores relacionales
- 9 Operadores de incremento y decremento
- 10 Operador de llamada a función

El operador de asignación: segunda aproximación

```
Polinomio& operator=(const Polinomio &pol);
```

- Recordemos que el operador de asignación puede usarse de la siguiente forma: `p=q=r=s;`.
- C++ evalúa la expresión anterior de derecha a izquierda, de forma que lo primero que realiza es `r=s`.
- El resultado de esta última expresión (`r=s`) es el objeto que queda a la izquierda (`r`), que se usa para evaluar el siguiente operador de asignación (asignación a `q`).
- Por tanto `operator=` debe devolver el mismo tipo de la clase (`Polinomio` en este caso).
- Para que la llamada a `r.operator=(s)` devuelva el objeto `r` es necesario que la devolución sea por referencia.

El operador de asignación: implementación final

```
Polinomio& operator=(const Polinomio &pol);
```

- En el caso de realizar una asignación del tipo $p=p$ nuestro operador de asignación no funcionaría bien.
- En tal caso, dentro del método `operator=`, `*this` y `pol` son el mismo objeto.

```
Polinomio& Polinomio::operator=(const Polinomio &pol){
    if(&pol!=this){
        delete[] this->coeficientes;
        this->maximoGrado=pol.maximoGrado;
        this->grado=pol.grado;
        this->coeficientes=new float[this->maximoGrado+1];
        for(int i=0; i<=maximoGrado; ++i)
            this->coeficientes[i]=pol.coeficientes[i];
    }
    return *this;
}
```

El operador de asignación: esquema genérico

Esquema genérico del operador de asignación

En una clase que tenga datos miembro que usen memoria dinámica, el esquema genérico del operador de asignación (`operator=`) sería el siguiente:

```
CLASE& CLASE::operator=(const CLASE &p)
{
    if (&p!=this) { // Si no es el mismo objeto
        // Si *this tiene memoria dinámica -> liberarla
        // Copiar p en *this (reservar nueva memoria y copiar)
    }
    return *this; // Devolver referencia a *this
}
```

Contenido del tema

- 1 Introducción a la sobrecarga de operadores
- 2 Mecanismos de sobrecarga de operadores
 - Sobrecarga como función externa: Ejemplo operator+
 - Sobrecarga como función miembro: Ejemplo operator+
 - Sobrecarga de operadores como función miembro o externa
- 3 El operador de asignación
- 4 La clase mínima
- 5 Operadores << y >>**
 - Sobrecarga del operador <<
 - Sobrecarga del operador >>
 - Sobrecarga del operador << con una función amiga
- 6 Operador de indexación
- 7 Operadores de asignación compuestos
- 8 Operadores relacionales
- 9 Operadores de incremento y decremento
- 10 Operador de llamada a función

Contenido del tema

- 1 Introducción a la sobrecarga de operadores
- 2 Mecanismos de sobrecarga de operadores
 - Sobrecarga como función externa:
Ejemplo operator+
 - Sobrecarga como función miembro:
Ejemplo operator+
 - Sobrecarga de operadores como función miembro o externa
- 3 El operador de asignación
- 4 La clase mínima
- 5 Operadores << y >>
 - Sobrecarga del operador <<
 - Sobrecarga del operador >>
 - Sobrecarga del operador << con una función amiga
- 6 Operador de indexación
- 7 Operadores de asignación compuestos
- 8 Operadores relacionales
- 9 Operadores de incremento y decremento
- 10 Operador de llamada a función

Sobrecarga del operador <<

Operador << (operador de salida)

Se usa para enviar el contenido de un objeto a un flujo de salida (por ej. `cout`)

- Podemos sobrecargar el operador << para mostrar un objeto usando la sintaxis `cout << p` (equivalente a `cout.operator<<(p)`).
- Puesto que no podemos añadir un método a la clase `ostream` (a la que pertenece `cout`), usamos una **función externa**.

```
ostream& operator<<(ostream& flujo, const Polinomio& p){
    flujo << p.getCoeficiente(p.getGrado());//Mostrar término grado mayor
    if(p.getGrado()>1)
        flujo << "x^" << p.getGrado();
    else if (p.getGrado()==1)
        flujo << "x";
    for(int i=p.getGrado()-1;i>=0;--i){//Recorrer resto de términos
        if(p.getCoeficiente(i)!=0.0){//Si el coeficiente no es 0.0
            if(p.getCoeficiente(i)>0.0) //imprimir coeficiente positivo
                flujo << " + " << p.getCoeficiente(i);
            else //imprimir coeficiente negativo
                flujo << " - " << -p.getCoeficiente(i);
            if(i>1)
                flujo << "x^" << i;
            else if (i==1)
                flujo << "x";
        }
    }
    return flujo;
}
```

Contenido del tema

- 1 Introducción a la sobrecarga de operadores
- 2 Mecanismos de sobrecarga de operadores
 - Sobrecarga como función externa:
Ejemplo operator+
 - Sobrecarga como función miembro:
Ejemplo operator+
 - Sobrecarga de operadores como función miembro o externa
- 3 El operador de asignación
- 4 La clase mínima
- 5 Operadores << y >>
 - Sobrecarga del operador <<
 - Sobrecarga del operador >>
 - Sobrecarga del operador << con una función amiga
- 6 Operador de indexación
- 7 Operadores de asignación compuestos
- 8 Operadores relacionales
- 9 Operadores de incremento y decremento
- 10 Operador de llamada a función

Sobrecarga del operador >>

Operador >> (operador de entrada)

Se usa para leer el contenido de un objeto desde un flujo de entrada (por ej. `cin`).

- Podemos sobrecargar el operador >> para leer un objeto usando la sintaxis `cin >> p` (equivalente a `cin.operator>>(p)`).
- De nuevo, puesto que no podemos añadir un método a la clase `istream` (a la que pertenece `cin`), sobrecargaremos este operador con una **función externa**.

Sobrecarga del operador >>

```
istream& operator>>(std::istream &flujo, Polinomio &p){
    int g;
    float v;

    p.clear();
    do{
        flujo>> v >> g; //Introducir coeficientes en la forma "coeficiente grado"
        if(g>=0){ // Se introduce grado<0 para terminar
            p.setCoeficiente(g,v);
        }
    }while(g>=0);
    return flujo;
}

void Polinomio::clear(){
    if(coef)
        delete[] coef;
    grado=0;
    max_grado=10;
    coef=new float[max_grado+1];
    for(int i=0; i<=max_grado; ++i)
        coef[i]=0.0;
}
```

Contenido del tema

- 1 Introducción a la sobrecarga de operadores
- 2 Mecanismos de sobrecarga de operadores
 - Sobrecarga como función externa: Ejemplo operator+
 - Sobrecarga como función miembro: Ejemplo operator+
 - Sobrecarga de operadores como función miembro o externa
- 3 El operador de asignación
- 4 La clase mínima
- 5 Operadores << y >>
 - Sobrecarga del operador <<
 - Sobrecarga del operador >>
 - Sobrecarga del operador << con una función amiga
- 6 Operador de indexación**
- 7 Operadores de asignación compuestos
- 8 Operadores relacionales
- 9 Operadores de incremento y decremento
- 10 Operador de llamada a función

Operador de indexación

Operador de indexación

La función `operator[]` permite sobrecargar el operador de indexación.

- Debe realizarse usando un método de la clase con un parámetro (índice) que podría ser de cualquier tipo.
- De esta forma podremos cambiar la sintaxis:

```
x = p.getCoeficiente(i);
```

por esta otra:

```
x = p[i];
```

Operador de indexación

- La versión final de la implementación de este operador quedaría como:

```
float& Polinomio::operator[](int i) {
    assert(i>=0); assert(i<=grado);
    return coeficientes[i];
}

const float& Polinomio::operator[](int i) const{
    assert(i>=0); assert(i<=grado);
    return coeficientes[i];
}

int main(){
    Polinomio p1;
    float x;
    ...
    const Polinomio p2=p1;
    x=p2[j]; // Usa const float& Polinomio::operator[](int i) const
    p1[i]=x; // Usa float& Polinomio::operator[](int i)
}
```

Contenido del tema

- 1 Introducción a la sobrecarga de operadores
- 2 Mecanismos de sobrecarga de operadores
 - Sobrecarga como función externa: Ejemplo operator+
 - Sobrecarga como función miembro: Ejemplo operator+
 - Sobrecarga de operadores como función miembro o externa
- 3 El operador de asignación
- 4 La clase mínima
- 5 Operadores << y >>
 - Sobrecarga del operador <<
 - Sobrecarga del operador >>
 - Sobrecarga del operador << con una función amiga
- 6 Operador de indexación
- 7 Operadores de asignación compuestos**
- 8 Operadores relacionales
- 9 Operadores de incremento y decremento
- 10 Operador de llamada a función

Operadores de asignación compuestos

Operadores de asignación compuestos

Son los operadores `+=`, `-=`, `*=`, `/=`, `%=`, `^=`, `&=`, `|=`, `»=`, `«=`

- Tener implementado el operador `+` y el operador `=` no supone la existencia automática del operador `+=`, y así con el resto: debemos implementarlo de **forma explícita**.
- Estos operadores deben devolver una referencia al objeto usado en la llamada, para así poder hacer por ejemplo:

```
p3 = (p1 += p2);
```

- Implementación como función miembro:

```
Polinomio& Polinomio::operator+=(const Polinomio& pol){
    *this = *this + pol;
    return *this;
}
```

- Es posible también implementarlos como función externa:

```
Polinomio& operator+=(Polinomio& pol1, const Polinomio& pol2){
    pol1 = pol1 + pol2;
    return pol1;
}
```

Contenido del tema

- 1 Introducción a la sobrecarga de operadores
- 2 Mecanismos de sobrecarga de operadores
 - Sobrecarga como función externa: Ejemplo operator+
 - Sobrecarga como función miembro: Ejemplo operator+
 - Sobrecarga de operadores como función miembro o externa
- 3 El operador de asignación
- 4 La clase mínima
- 5 Operadores << y >>
 - Sobrecarga del operador <<
 - Sobrecarga del operador >>
 - Sobrecarga del operador << con una función amiga
- 6 Operador de indexación
- 7 Operadores de asignación compuestos
- 8 Operadores relacionales**
- 9 Operadores de incremento y decremento
- 10 Operador de llamada a función

Operadores relacionales

Operadores relacionales

Son los operadores binarios $==$, $!=$, $<$, $>$, $<=$ y $>=$, que devuelven un valor booleano.

- Se usan cuando es necesario establecer una *relación de orden* entre los objetos de la clase.
- El definir una parte de los operadores no implica que los demás lo estén de forma automática.

Ejemplo: si definimos el operador $==$, el operador $!=$ no estará definido de forma automática.

Operadores relacionales

Ejemplo: operador $<$ en Polinomio

$\text{pol1} < \text{pol2}$ si pol1 tiene grado menor que pol2 , o, si son del mismo grado, su coeficiente máximo es menor que el de pol2 .

```
bool Polinomio::operator<(const Polinomio& pol) const{
    bool menor = this->getGrado()<pol.getGrado() ? true : false;
    if (!menor){
        bool iguales = this->getGrado()==pol.getGrado() ? true : false;
        if(iguales){
            menor = coeficientes[this->getGrado()]<
                pol.coeficientes[this->getGrado()] ? true:false;
        }
    }
    return menor;
}
```

Contenido del tema

- 1 Introducción a la sobrecarga de operadores
- 2 Mecanismos de sobrecarga de operadores
 - Sobrecarga como función externa: Ejemplo operator+
 - Sobrecarga como función miembro: Ejemplo operator+
 - Sobrecarga de operadores como función miembro o externa
- 3 El operador de asignación
- 4 La clase mínima
- 5 Operadores << y >>
 - Sobrecarga del operador <<
 - Sobrecarga del operador >>
 - Sobrecarga del operador << con una función amiga
- 6 Operador de indexación
- 7 Operadores de asignación compuestos
- 8 Operadores relacionales
- 9 Operadores de incremento y decremento
- 10 Operador de llamada a función

Operador de llamada a función

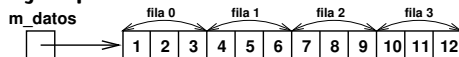
Operador de llamada a función

Es la función `operator()` que obligatoriamente se implementará como función miembro.

Puede implementarse con cualquier número de parámetros (podemos tener varias versiones de este operador).

Operador de llamada a función

Ejemplo:



```
class Matriz {
    double* m_datos;
    int m_filas, m_columnas;
public:
    Matriz(int nf, int nc){
        m_filas=nf;
        m_columnas=nc;
        m_datos = new double[m_filas*m_columnas];
    }
    double& operator() (int fila, int columna){
        assert(fila>=0 && fila<m_filas && columnas >=0 && columna<m_columnas);
        return m_datos[fila*m_columnas + columna];
    }
    const double& operator() (int fila, int columna) const{
        assert(fila>=0 && fila<m_filas && columnas >=0 && columna<m_columnas);
        return m_datos[fila*m_columnas + columna];
    }
}
```

Operador de llamada a función

```
int main(){  
    Matriz m(4,3);  
    ...  
    cout<<m(3,2)<<endl;  
    m(3,2) = 7.4;  
}
```