

Tipo de dato Puntero

Juan F. Huete

Departamento de Ciencias de la Computación e I.A.
Universidad de Granada

Metodología de la Programación, 2018

- 1 Conceptos preliminares
- 2 Gestión de Memoria
- 3 Tipo de dato puntero
 - Definición y Declaración de variables
 - Operaciones con punteros
 - Punteros y Arrays
 - Paso de punteros a funciones
 - Resumen de operaciones con punteros
 - Punteros y const
- 4 Tipo de Dato Referencia
 - Punteros a función

Declaración vs Definición

- **Declaración** nos proporciona un identificador (etiqueta) asociado un tipo, objeto (variable) o función
 - En caso de objetos, nos indica el tipo del objeto
 - en caso de funciones, se especifica el número de parámetros, el tipo de cada uno, y el valor de retorno.

Todo identificador debe ser declarado antes de utilizarlo

- **Definición** proporciona toda la información sobre un identificador, pero además si es necesario se crea (reserva memoria) el elemento
 - en caso de un tipo, proporciona información completa del tipo.
 - en caso de objeto (variable), se reserva una zona de memoria para dicho objeto.
 - en caso de una función, se proporciona el código de la función.
- En caso de objetos, en la mayoría de los casos (pero no todos), la declaración de un objeto implica su definición

Ejemplos de declaración y definición

```
1 int contador; // declara y define contador
2 extern double beta; // solo declara beta
3
4 void func() { // declara y define func
5     int n; // declara y define n
6     double x = 1.0; // declara y define x
7     // ...
8 }
9
10 bool esPar(int ); // declara esPar
11 bool esPar(int x); // declara esPar (x es ignorado)
12 bool esPar (int x){ //declara y define esPar
13     return x % 2;
14 }
15
16 struct Objeto; //declara Objeto (se definira despues)
17 struct variosElementos { //declara y define variosElementos
18     double x;
19     double y;
20     Objeto z;
21 };
```

Sobrecarga de funciones

Sobrecarga de funciones

Es posible tener dos o más funciones con el mismo nombre siempre y cuando sea posible distinguirlas mediante el tipo y número de sus parámetros formales

El compilador decide cuál de las versiones de la función usará, de acuerdo a los siguientes criterios (en orden de prioridad):

- 1 Concordancia exacta los argumentos, en el número y tipo.
Se incluyen conversiones triviales, como nombres de matriz a punteros, nombres de función a puntero a función, y tipo a const tipo. Esto último **(se entenderá a la largo del curso)**
- 2 Concordancia después de realizar promociones sencillas: de tipos asimilables a int (char , short , bool , enum) hacia int y de float a double.
- 3 Concordancia después de realizar conversiones de tipos propios del compilador: int a double , double a long double , tipo* a void*.
- 4 Concordancia después de realizar conversiones definidas por el usuario **(se entenderá a la largo del curso)**
- 5 Concordancia usando los puntos suspensivos (...) en funciones con número variable de parámetros **(no lo estudiaremos)**

Ejemplos de sobrecarga

```
1 bool mayor(int a, int b);          (1) // Declaramos
2 double mayor(double a, double b); (2)
3 int mayor(int , int , int );      (3)
4
5 .....
6 bool mayor(int a, int b){          // Definimos
7     if (a>b) return true; else return false;
8 }
9 double mayor(double a, double b){
10     if (a>=b) return a; else return b;
11 }
12 int mayor(int a, int b, int c) {
13     if (a>b && a>c) return a; ....
14 }
15
16 ....
17 int main(){
18     cout << mayor(2,3);           // (1) -> aplica criterio 1
19     cout << mayor(2.4,5.8);       // (2) -> aplica criterio 1
20     cout << mayor(45,'b');        // (1) -> aplica criterio 2
21     cout << mayor(1,2,'c');       // (3) -> aplica criterio 2
22 }
```

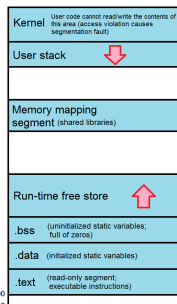
Definición implica reserva de memoria

Una definición de variables o función implica que el compilador debe encargarse de **reservar memoria** para ubicar los datos o el código, respectivamente, asociado. No todas las reservas se hacen igual.

El compilador necesita poder determinar de forma precisa:

- cuánta memoria reservar (tamaño del dato o todas las instrucciones de la función)y
- en qué zona de memoria debe hacerlo (dependiendo del tipo y ámbito de la variable).

- 1 Conceptos preliminares
- 2 Gestión de Memoria
- 3 Tipo de dato puntero
 - Definición y Declaración de variables
 - Operaciones con punteros
 - Punteros y Arrays
 - Paso de punteros a funciones
 - Resumen de operaciones con punteros
 - Punteros y const
- 4 Tipo de Dato Referencia
 - Punteros a función



- Todos los procesos que se ejecutan en el S.O. tienen asociado un espacio de memoria virtual. La arquitectura x86-64 tiene direcciones de 64 bits (en Linux usa los 48 bits menos significativos)
 - **kernel** Parte del sistema operativo que reside en memoria
 - Segmento **.text** (o código de programa) conteniendo las instrucciones ejecutables del programa
 - **.data y .bss** contienen constantes y variables estáticas (o globales) que existen durante toda la ejecución del programa, respect.
 - Pila de usuario **user stack**, empleada para la ejecución de funciones. Contiene variables locales y parámetros de funciones. El compilador aloja/libera los recursos necesarios al entrar/salir de una función (estrategia LIFO).
 - **Heap**, el área de memoria libre (run-time free store) cuyo tamaño varía en tiempo de ejecución. Es el **programador** el que se debe encargar de la **correcta gestión** de estos recursos

Atendiendo al tiempo que permanece activo en memoria distinguimos entre

- **Datos Estáticos**

- Los datos siempre ocupan la misma posición y su número no varía durante la ejecución del programa.
- Son los datos (constantes y variables) declarados fuera del ámbito de cualquier función o declarados como estáticos explícitamente.

```
int global = 0;
const double pi = 3.141592;
int main( ) {
    static char A[8]="adios";
    const static char C[10] = "constante";
}
```

- **Datos Dinámicos**

Varían durante la ejecución de un programa. Se gestionan de dos formas distintas

- **Pila (stack):**
- **montón (heap):**

Pila (stack)

- La reserva y liberación de esta memoria la realiza el sistema operativo de manera automática durante la ejecución de un programa.
- Permanecen activos desde el momento en el que se entra a la función (o bloque) en el que están declarados, y dejan de poder usarse cuando salimos de su ámbito.
- El funcionamiento de la pila sigue el esquema LIFO (Last In, First Out).

```
int funcion1(int x) {  
    int y=3;  
    // ...  
}  
  
int main(int argc, char** argv) {  
    char H[5]="hola";  
    double d = 2.36;  
    for (int i = 0; i<5; i++){  
        ....  
    }  
}
```

Errores de software (bugs) comunes en la gestión de la pila:

- Si el uso de memoria sobrepasase el tamaño máximo permitido para la pila (stack overflow) se produciría una violación de acceso (segmentation fault) y el programa aborta.

Violacion de segmento ('core' generado)

- Almacenando variables de gran tamaño (el límite de la pila en Linux es 8MB) → almacenarlo en heap
- Ejecutando algoritmos recursivos (casi) infinitos
- Escribir más allá de la cota superior de un array puede corromper la pila. Difícil de detectar en un proceso de depuración (debugging).

```
char nombre[10]="Pedro";  
nombre[25] = 'x';
```

Error evitable usando técnicas de programación del lenguaje C++ (clases bien diseñadas que controlen tales accesos como array, vector, etc.).

montón (heap)

- Es una zona de memoria donde el **programador** reserva y libera “trozos” de memoria durante la ejecución de los programas.
 - Cuando se solicita un bloque de memoria al sistema, este queda marcado como ocupado y no será usado en ninguna otra petición. El sistema devuelve la dirección de comienzo del bloque completo
 - Al liberar un bloque queda a disposición para futuras peticiones.
-
- Definimos el tipo **puntero (pointer)** como el tipo de dato cuyo dominio es el de las direcciones de memoria , esencial cuando consideramos el heap.

Lo estudiaremos en este tema ...



Gestión de Memoria: Ejemplo

```
// memoria0.cpp
int funcion1(int x) {
    int y=3;
    // ...
}
int funcion2(int &x) {
    ...
}
int global = 0;
int main(int argc, char** argv) {
    char H[5]="hola";
    static char A[8]="adios";
    const static char C[10] = "constante";
    int i1 = 2;
    double d1 = 3.2;
    double d2;
    double f[10];
    ....
}
```

Gestión de Memoria: Ejemplo

- Cada dato puede ocupar una o varias celdas (en principio consecutivas) de memoria, en función del tipo asociado.
- Los lenguajes de alto nivel gestionan los datos asociándoles un nombre y no mediante su dirección.
- Ejemplo de listado con los nombres de variables/funciones, tipos, tamaño, dirección de memoria y valor:

```
.text -----
main                :FiiPPcE 0x40117a
funcion1            :FiiE 0x400c1d
funcion2            :FiRiE 0x400d4b
.data -----
const static char C[10] :A10_c size 10 dir 0x401b40
.bss -----
global
static char A[8]     :A8_c size 8 dir 0x6030a0
user stack -----
char H[5]            :A5_c size 5 dir 0x7fff4fcb36e0 valor hola
int i1               :i size 4 dir 0x7fff4fcb367c valor 2
double d1            :d size 8 dir 0x7fff4fcb3680 valor 3.2 0x7fff4fcb3688
double d2            :d size 8 dir 0x7fff4fcb3688 valor 6.9533e-310
double f[10]         :A10_d size 80 dir 0x7fff4fcb3690 valor 0x7fff4fcb3690
user stack-----
funcion1::x i size 4 dir 0x7fff4fcb361c valor 2
funcion1::y i size 4 dir 0x7fff4fcb362c valor 3
```

- 1 Conceptos preliminares
- 2 Gestión de Memoria
- 3 Tipo de dato puntero
 - Definición y Declaración de variables
 - Operaciones con punteros
 - Punteros y Arrays
 - Paso de punteros a funciones
 - Resumen de operaciones con punteros
 - Punteros y const
- 4 Tipo de Dato Referencia
 - Punteros a función

Tipo de dato puntero: Introducción

- Es un tipo de dato diseñado para almacenar direcciones de memoria de otras variables
- Se declaran y definen de acuerdo al tipo al que apuntan.
- (+) Nos permiten realizar muchas tareas que no serían posibles con los tipos simples (trabajamos a bajo nivel)
- (-) Si son utilizados incorrectamente generan errores que son difíciles de controlar (trabajamos a bajo nivel). Por lo tanto, es **importante** utilizar una correcta Metodología de la Programación.

1 Conceptos preliminares

2 Gestión de Memoria

3 Tipo de dato puntero

- Definición y Declaración de variables
- Operaciones con punteros
- Punteros y Arrays
- Paso de punteros a funciones
- Resumen de operaciones con punteros
- Punteros y const

4 Tipo de Dato Referencia

- Punteros a función

Definición de una variable tipo puntero

Tipo de dato puntero

Tipo de dato que contiene la dirección de memoria de otro dato. Con más precisión: la dirección de memoria donde empieza a almacenarse el otro dato.

- Incluye una dirección especial llamada *dirección nula* que es el valor 0.
- El número de bytes (sizeof) utilizado para almacenar una dirección depende de la arquitectura de la CPU: 8 bytes (arqu. 64 bits) o 4 bytes (arqu. 32 bits)

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 int i=5,j; char c1 = 'a',c2; long double d1 = 2.345, d2;
2 int *pi; char *pc; double *pd;
3
4 int *a,*b,c,d; //a y b son punteros a enteros, c y de enteros
5 int* funcion(int x, int *y) // devuelve un puntero a entero
6                             // x entero, y puntero a entero
7
8 cout << "int i  :"<<"size"<< sizeof(i) <<"val" << i <<endl;
9 ...
10 cout << "int *pi  :"<<"size"<< sizeof(pi) <<"val"<< pi <<endl;
11 cout << "char *pc  :"<<"size"<< sizeof(pc) <<"val"<< pc <<endl;
12 cout << "double *pd:"<<"size"<< sizeof(pd) <<"val"<< pd <<endl;
```

```
int i           : size 4   valor 5
int j           : size 4   valor 32563
char c1         : size 1   valor a
char c2         : size 1   valor
long double d1  : size 16  valor 2.345
long double d2  : size 16  valor 5.12985e-4937
int *pi         : size 8   valor 0x10000ffff    HEXADECIMAL
char *pc        : size 8   valor 0xW#x.....   HEXADECIMAL
double *pd      : size 8   valor 0x401739       HEXADECIMAL
```

Inicialización de punteros

- Una variable puntero (como el resto de variables) NO se inicializa cuando es definida, tendrá una dirección de memoria **basura**, lo cual es la causa de muchos problemas de programación.
- Como sólo pueden almacenar direcciones de memoria, se ha definido una constante que nos permite indicar que un puntero no apunta a ningún sitio, esto es, un valor nulo.

```
1  int * pi1 = 0; // C implica conversion de int a int*
2  int * pi2 = NULL; // C implica conversion de int a int*
3                      // definida stdlib.h (C) cstdlib (C++)
4  double *pd = nullptr; // C++11 (compilar con -std=c++11)
5                      // esta es la opcion mas correcta.
```

- Podemos chequear si un puntero apunta a nulo (es nulo)

```
if (pi)
    cout << "No es nulo" ;
else
    cout << "Puntero nulo!!";
```

```
if (pd!=nullptr)
    cout << "No es nulo";
else
    cout << "Puntero nulo!!";
```

Tipo de dato: puntero a ...

- Punteros a datos de distinto tipo son tipos de datos diferentes.

Error típico



```
double *pd, d; int *p;  
p = &d;  
p = pd;
```

```
cannot convert 'double*' to 'int*' in assignment  
p=&d;  
p = pd;
```

- El tamaño (sizeof) de un puntero a ... es siempre el mismo (8 bytes en arqu. 64 bits) ya que almacenan la dirección de comienzo de objeto al que apuntan

```
struct objeto {  
    int a; double b; float c;  
};  
double *pd; int *pi; objeto *po;  
cout << sizeof(pd) << sizeof(pi) << sizeof(po); // 8 8 8
```

1 Conceptos preliminares

2 Gestión de Memoria

3 Tipo de dato puntero

- Definición y Declaración de variables
- **Operaciones con punteros**
- Punteros y Arrays
- Paso de punteros a funciones
- Resumen de operaciones con punteros
- Punteros y const

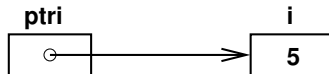
4 Tipo de Dato Referencia

- Punteros a función

Operador de dirección &

- `&<var>` devuelve la dirección de memoria asignada a la variable `<var>` (o sea, un puntero).
- El operador `&` se utiliza habitualmente para asignar valores a datos de tipo puntero.

```
int i=5;  
int *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 &

```
1 int i=5; double d1 = 2.345;
2
3 int *pi= &i;
4 double *pd= &d1;
5 float *pf =nullptr;
6
7 cout << "int i :" <<"dir"<< &i <<"val"<< i <<endl;
8 cout << "double d1:" <<"dir"<< &d1 <<"val"<< d1 <<endl;
9
10 cout << "int *pi  :" <<"dir"<< &pi <<"val"<< pi <<endl;
11 cout << "double *pd:"<<"dir"<< &pd <<"val"<< pd <<endl;
12 cout << "float *pf :"<<"dir"<< &pf <<"val"<< pf <<endl;
```

```
int i      :  dir 0x7ffc4a4196f4 valor 5           direcc. en HEXADECIMAL
double d1  :  dir 0x7ffc4a4196f8 valor 2.345

int *pi    :  dir 0x7ffc4a419700 valor 0x7ffc4a4196f4
double *pd :  dir 0x7ffc4a419708 valor 0x7ffc4a4196f8
float *pf  :  dir 0x7ffc4a419710 valor 0
```

Operador de indirección *

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

```
char c='A', *ptrc;  
.....  
// Hacemos que el puntero apunte a c  
ptrc = &c;  
// Consultamos el contenido de ptrc  
cout << *ptrc << endl;  
// Cambiamos contenido de c mediante  
    ptrc  
*ptrc = 'B'; // equivale a c = 'B'  
// Imprimimos c  
cout << c << endl;
```



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

Operador de indirección *

- Un puntero **debe estar correctamente inicializado** antes de usarse por el operador de indirección.
- Cuando se llama al operador de indirección, la aplicación intenta ir a la ubicación de la memoria que está almacenada en el puntero y recuperar el contenido de la memoria. Para ello, utiliza la información asociada al tipo al que apunta
- Si una programa intenta acceder a una ubicación de memoria no asignada por el sistema operativo, el sistema operativo puede cerrarlo.

```
1  int a = 7;  
2  int *p1 = &a, *p2=nullptr, *p3;  
3  *p1 = 10; //correcto  
4  *p2 = 20; //Error  
5  *p3 = 30; //Error
```

Violacion de segmento ('core' generado)

Los punteros en C++ son inseguros “per se” y el uso incorrecto de ellos garantiza que la aplicación se detenga



Importancia de los punteros

Aunque su uso es peligroso es necesario conocer su funcionamiento ...

Algunas razones

- 1 Los Arrays se implementan usando punteros. Los punteros se pueden usar para iterar a través de un array (alternativa a los índices). [▶ Sec. 3](#)
- 2 Se pueden usar para pasar una gran cantidad de datos a una función de manera que no implique copiar los datos, lo que es ineficiente. [▶ Sec.38](#)
- 3 Son la única forma en que puede asignar memoria dinámicamente en C++ . Este es de lejos el caso de uso más común para los punteros. [▶ siguiente tema](#)
- 4 Se pueden usar para pasar una función como parámetro a otra función [▶ Sec.1](#)
- 5 Se pueden usar para lograr polimorfismo cuando se trata de herencia (fuera del ámbito de este curso)
- 6 Se pueden usar para enlazar estructuras formando una cadena. Esto es útil en algunas estructuras de datos más avanzadas, como listas y árboles. (estudiadas a fondo en Estructuras de Datos).

1 Conceptos preliminares

2 Gestión de Memoria

3 Tipo de dato puntero

- Definición y Declaración de variables
- Operaciones con punteros
- **Punteros y Arrays**
- Paso de punteros a funciones
- Resumen de operaciones con punteros
- Punteros y const

4 Tipo de Dato Referencia

- Punteros a función

Los punteros y los arrays están estrechamente vinculados.

Al declarar un array

```
<tipo> <identif>[<n_elem>]
```

- 1 Se reserva memoria para almacenar `<n_elem>` elementos de tipo `<tipo>`.
- 2 Se crea un puntero CONSTANTE llamado `<identif>` que apunta a la primera posición de la memoria reservada.

Por tanto, el `identificador` de un array, es un puntero CONSTANTE a la dirección de memoria que contiene el primer elemento. Es decir, `v` es igual a `&(v[0])`.

Podemos usar arrays como punteros al primer elemento.

```
1 int v[5] = {2,6,3,5,3};  
2 cout << *v << endl;  
3 int *ptr = nullptr;  
4 ptr = v; // equiv. ptr = &v[0]  
5 cout << *ptr << endl;  
6 ptr = &v[2];  
7 cout << *ptr << endl;
```

	0	1	2	3	4
v	2	6	3	5	3

Podemos usar un puntero a un elemento de un array como un array que comienza en ese elemento

- De esta forma, los punteros pueden indexarse como si fuesen arrays:

$v[i]$ es equivalente a $ptr[i]$ 

Punteros y arrays: iterando sobre los elementos

- Los punteros se pueden usar para iterar (recorrer) los elementos de un array

	0	1	2	3	4
v	2	6	3	5	3

- Es una alternativa al uso de índices
- Para ello es necesario considerar algunos operadores sobre punteros:
 - Operadores aritméticos: $+$, $-$, $++$, $--$, $+=$, $-=$
 - Operadores relacionales: $<$, $>$, $<=$, $>=$

Punteros: Operadores aritméticos

- Los operadores `+`, `-`, `++`, `--`, `+=` y `-=` son aplicables a punteros.
- Al usar estos operadores, el valor del puntero (la dirección que almacena) se comporta CASI como un número entero.
- Al sumar o restar un número N al valor del puntero, éste se incrementa o decrementa un determinado número de posiciones, en función del tipo de dato apuntado, según la fórmula:

$$N * \text{sizeof}(\text{tipobase})$$

```
1  int x = 7, *p = &x;
2  cout<< p << endl;
3  cout<< p+1 << endl;
4  cout<< p+2 << endl;
```

```
1  0x7ffc0ba722d0
2  0x7ffc0ba722d4
3  0x7ffc0ba722d8
```

```
1  double d = 7, *p = &d;
2  cout<< p << endl;
3  cout<< p+1 << endl;
4  cout<< p+2 << endl;
```

```
1  0x7ffc9c43df40
2  0x7ffc9c43df48
3  0x7ffc9c43df50
```

- Permite acceder a los elementos de un array, aprovechando que todos sus elementos se almacenan en posiciones sucesivas.

Punteros: Operadores aritméticos

● Situación inicial

```
1  int v[5] = {2, 6, 3, 5, 3};
2  int *ptr = nullptr;
3  ptr = v; // equiv. ptr = &v[0]
4  cout << *ptr << endl;
5  ptr += 2; // ptr = ptr + 2;
6  cout << *ptr << endl;
7  ptr++; // ptr = ptr + 1;
8  cout << *ptr << endl;
9  *ptr = 4; // cambiamos vector
10 ptr = ptr + 10;
11 cout << *ptr << endl; //Error!!
```

	0	1	2	3	4
v	2	6	3	5	3

Operador []

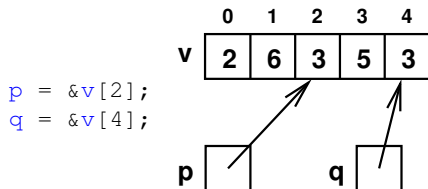
Al ejecutar `ptr[i]` lo que realmente se ejecuta es `*(ptr+i)`

```
1  ptr = v;
2  cout << ptr[2] << endl; // *(ptr+2)
3  ptr[2] = 4; // *(ptr+2)=4
```

Operadores relacionales

Operadores <, >, <=, >=

- Los operadores <, >, <= y >= tienen sentido para conocer la posición relativa de un objeto respecto a otro en la memoria.
- Sólo son útiles si los dos punteros apuntan a objetos cuyas posiciones relativas guardan relación (por ejemplo, elementos del mismo array).



<code>p==q</code>	false
<code>p!=q</code>	true
<code>*p==*q</code>	true
<code>p<q</code>	true
<code>p>q</code>	false
<code>p<=q</code>	true
<code>p>=q</code>	false

Algunos Ejemplos: Recorriendo arrays

1 Recorrer e imprimir los elementos de un array:

```
1 int v[10] = {3,5,2,7,6,7,5,1,2,5};  
2 for (int i=0; i<10; i++)  
3     cout << v[i] << endl;
```

2 Recorrer e imprimir los elementos de un array con el puntero:

```
1 int v[10] = {3,5,2,7,6,7,5,1,2,5};  
2 int *p=v;  
3 for (int i=0; i<10; i++){  
4     cout << *(p+i) << endl;  
5     cout << p[i] << endl;  
6 }
```

3 Recorrer e imprimir los elementos de un array avanzando puntero:

```
1 int v[10] = {3,5,2,7,6,7,5,1,2,5};  
2 for (int *p=v; p<v+10; p++)  
3     cout << *p << endl;
```

Punteros vs Arrays

- Es un error común en C++ es pensar que un puntero y un array son idénticos.

```
1 int vint[10] = {1,2,3,4,5,6,7,8,9,0};  
2 int *pi;  
3 pi = vint; // Correcto y posible causa del error
```

- Sin embargo son de distinto tipo, el compilador los trata diferentemente

```
1 #include <typeinfo> //informacion sobre tipos  
2 cout<<"int vint[10]:"<< typeid(vint).name() <<"size"<<  
3     sizeof(vint) <<"dir"<< &vint <<"val"<< vint;  
4 cout<<"int *pi      :"<< typeid(pi).name() <<"size"<<  
5     sizeof(pi) <<"dir"<< &pi <<"val"<< pi;
```

```
1 int vint[10]:A10_i size 40 dir 0x7ffe8e9f8e00 val 0x7ffe8e9f8e00  
2 int *pi      :Pi      size 8 dir 0x7ffef66b80d0 val 0x7ffef66b80f0
```


Problema

Al escribir en una expresión **vint** se hace una conversión implícita a **int ***:
vint es equivalente a **&vint** y a **&vint[0]** // **&(*vint)**

Paso de arrays a funciones

- Cuando una función en C++ recibe una variable, se realiza una copia de la misma. Cambiar el valor de la variable dentro de la función no cambia el valor de la variable original **paso por valor**.


```
1  double suma(double x[10000]) {  
2      double s = 0.0;  
3      for (int i=0; i<10000; i++) s+=x[i]  
4      return s;  
5  }  
6  int main() {  
7      double d[10000], v;  
8      v = suma(d);  
9  }
```

¿Qué inconveniente vemos en este ejemplo? 

Paso de arrays a funciones

- Cuando una función en C++ recibe una variable, se realiza una copia de la misma. Cambiar el valor de la variable dentro de la función no cambia el valor de la variable original **paso por valor**.

```
1  double suma(double x[10000]) {  
2      double s = 0.0;  
3      for (int i=0; i<10000; i++) s+=x[i]  
4      return s;  
5  }  
6  int main() {  
7      double d[10000], v;  
8      v = suma(d);  
9  }
```

¿Qué inconveniente vemos en este ejemplo? 

- Aunque un array es un tipo de dato normal, su paso como parámetro a una función NO se realiza como se podría esperar en C++.

Copiar arrays de gran tamaño puede ser muy ineficiente

1 Conceptos preliminares

2 Gestión de Memoria

3 Tipo de dato puntero

- Definición y Declaración de variables
- Operaciones con punteros
- Punteros y Arrays
- **Paso de punteros a funciones**
- Resumen de operaciones con punteros
- Punteros y const

4 Tipo de Dato Referencia

- Punteros a función

Paso de arrays a funciones \implies paso de punteros

- Cuando se pasa un array como argumento, como en F,

```
1 void F( double x[1000]) {
2     cout<<"En F:"<< typeid(x).name() <<"size"<< sizeof(x) <<
3     "val"<< x << endl;
4 }
5 int main() {
6     double d[1000],v; / / "d (tipo A1000_d)"
7     cout <<"En main: "<< typeid(d).name() << ;
8     cout <<"size"<< sizeof(d) <<;
9     cout <<"val"<< d << endl; //CONV. IMPLICITA a (double *)
10    F( d );      // CONVERSION IMPLICITA a (double *)
11 }
```

Tanto en la línea 8 como en 9 se hace una CONVERSIÓN IMPLÍCITA A (double *) que apunta al primer elemento del array, el compilador transforma la llamada a

```
1 F (&d)
```

```
1 En main: A10000_d size 80000 val 0x7ffd15346c80
2 En F: Pd size 8 val 0x7ffd15346c80
```

Paso de arrays a funciones \implies paso de punteros

- Por motivos de eficiencia, una función en C++ **NO PUEDE** recibir como parámetro un array, sino un puntero.

Por ello, estas tres declaraciones son equivalentes.

```
(linea 27) void F(double x[1000])
           { .... }
(linea 30) void F(double x[ ])
           { .... }
(linea 33) void F(double *x) //recomendada
           { .... }
```

```
1 prueba.cpp: In function void F(double*)':
2 prueba.cpp:30:8: error: redefinition of void F(double*)'
   void F(double x[]){
   ^
3
4
5 prueba.cpp:27:8: error: void F(double*)' previously defined here
   void F(double x[10000]){
   ^
6
7
8 prueba.cpp: In function void F(double*)':
9 prueba.cpp:33:8: error: redefinition of void F(double*)'
   void F(double *x){
   ^
10
11
12 prueba.cpp:27:8: error: void F(double*)' previously defined here
   void F(double x[10000]){
   ^
13
14
```

Paso de arrays a funciones \implies paso de punteros

- Como la función sólo recibe un puntero, será necesario indicarle también el tamaño del array

```
void pinta1(int *x, int n) {
    for (int i=0 ; i<n ; i++)
        cout << x[i] << " " ; // *(x+i)  equivalentemente
}

void pinta2(int *x, int n) {
    for (int i=0 ; i<n ; i++){
        cout << *x << " " ;
        ++x; // se avanza a la sig. direccion
    }
}

int main() {
    int x[10] = {1,2,3,4,5,6,7,8,9,0};
    pinta1(x,10); // conv. implicita de array a puntero al
                  // primer elemento del array
    pinta2(x,10); // conv. implicita de array a puntero
}
```

```
1 2 3 4 5 6 7 8 9 0
1 2 3 4 5 6 7 8 9 0
```

Paso de arrays a funciones \implies paso de punteros

- El hecho de utilizar punteros explica porqué al “cambiar un array” dentro de la función cambian los valores del valores almacenados en el mismo

El parámetro que pasamos (por valor) es la dirección del primer elemento del array, y esta NO cambia al salir de la función, pero sí el contenido

```
void cambia1(int *x, int n) {
    for (int i=0 ; i<n ; i++)
        x[i]++ ; // (*(x+i))++ equivalentes
}

void cambia2(int *x, int n) {
    for (int i=0 ; i<n ; i++){
        (*x)++; //incrementamos en 1 el valor
        ++x; // se avanza a la sig. direccion
    }
}

int main() {
    int x[10] = {1,2,3,4,5,6,7,8,9,0};
    pinta2(x,10);
    cambia1(x,10);    pinta2(x,10);
    cambia2(x,10);    pinta2(x,10);
}
```

1	2	3	4	5	6	7	8	9	0
2	3	4	5	6	7	8	9	10	1
3	4	5	6	7	8	9	10	11	2

Paso de punteros como parámetro a funciones

- Obviamente podemos pasar un puntero a cualquier dato (no necesariamente un array) y el efecto es el mismo, la llamada a la función nos permitirá modificar el valor.
- Es un comportamiento MUY INTERESANTE en muchas situaciones, aunque en C++ dispone de un mecanismo similar (aunque más simplificado para obtener el mismo resultado)

Paso de parámetros por referencia (lo veremos luego)

```
void cambiaD(double *pd) {  
    (*pd) = 3.4; // Cambiamos el elemento en  
    la direc.  
}  
  
int main() {  
    double d1=0.0;  
    cout << d1 << endl;  
    cambiaD(&d1); // Necesitamos puntero al  
    tipo de dato  
    cout << d1 << endl;  
}
```



0.0
3.4

Punteros a estructuras

- También podemos trabajar con punteros a estructuras.

Si A es una estructura con atributos a1, a2, ... y pA es un puntero a la estructura:

- `&A` es la dirección de memoria donde se almacenan los atributos (consecutivamente)
- `(*pA)` es la estructura completa
- `(*pA).a1` hace referencia al primer campo
- `pA->a1` hace referencia al primer campo

```
struct miStruct { int x; double d; } A,B;  
A.x= 1; A.d = 0.0;  
miStruct *pA = &A;  
B = (*pA)  
cout << A.x << " " << A.d << endl;  
(*pA).x = 2;  
pA->d = 3.4;  
cout << A.x << " " << A.d << endl;
```

1	0.0
2	3.4

Punteros a estructuras como parámetros de función

- Nos permite modificar los valores de la estructura dentro de la función

```
struct estr { int x; double d; };

void cambiaE(estr *p){
    (*p).x = 2;    // equiv. p->x = 2
    (*p).d = 3.4;  // equiv. p->d = 3.4
}

int main(){
    estr e1;  e1.x= 1; e1.d = 0.0;
    cout << e1.x << " " << e1.d << endl;
    cambiaE(&e1); // Necesitamos puntero al tipo de dato
    cout << e1.x << " " << e1.d << endl;
}
```

1	0.0
2	3.4

Paso de estructuras con array a funciones

Casuística con un comportamiento interesante

- Cuando un array forma parte de una estructura y esta se pasa a una función por valor, C++ NO hace la conversión implícita a puntero de dicho campo (se hace la copia completa del array)

```
struct estrArr {  
    int x[5]= {1,2,3,4,5};  
    int N = 5;  
};  
  
void cambiaE(estrArr p){//hace copia completa  
    p.x[3]= 2;    // cambiamos el valor  
    pental(p.x,p.N);  
}  
  
int main() {  
    estrArr el;  
    pental(el.x, el.N);  
    cambiaE(el);  
    pental(el.x, el.N);  
}
```

En main	:	1	2	3	4	5
En cambia	:	1	2	3	2	5
En main	:	1	2	3	4	5

Devolución de punteros

Una función podrá devolver un puntero **siempre** que nos aseguremos que la **dirección de memoria a la que apunta es correcta** (controlada por el sistema) al salir de la función

```
1  int * encuentra(int *x, int N, int valor) {
2      for (int i=0; i<N; i++) {
3          if (x[i]==valor)
4              return &x[i];
5      }
6      return nullptr;
7  }
8
9  int main() {
10     int v[10] = {1,2,3,4,5,6,7,8,9,1};
11     int *p;
12     p = encuentra(v,10,8);
13     if (p!=nullptr) {
14         cout << "esta en posicion" << p-v << endl;
15     }
16     else
17         cout << "no esta" << endl;
18 }
```

Devolución de punteros.

Errores típicos



El puntero que devolvemos está fuera del rango del array

```
1  int * encuentra(int *x, int N, int valor){
2  bool enc = false
3      for (int i=0; i<N && !enc; )
4          if (x[i]==valor) enc = true;
5          else i++;
6  return &x[i];
7  }
8  int main(){
9  int v[10] = {1,2,3,4,5,6,7,8,9,1};
10 int *p;
11 p = encuentra(v,10,8);
12 *p=5; // OK
13 p = encuentra(v,10,0);
14 *p=5; // ERROR
```

Devolución de punteros.



Errores típicos

Devolución de punteros a datos locales

Los datos locales se destruyen al terminar la función.

```
1 int * encuentra(int *x, int N, int valor) {
2   int si=0; bool enc=false;
3   for (int i=0; i<N && !enc; i++)
4     if (x[i]==valor)
5       si = x[i]
6   return &si;
7 }
8 int main() {
9   int v[10] = {1,2,3,4,5,6,7,8,9,1};
10  int *p;
11  p = encuentra(v,10,8);
12  *p = 3; // ERROR
```

1 Conceptos preliminares

2 Gestión de Memoria

3 Tipo de dato puntero

- Definición y Declaración de variables
- Operaciones con punteros
- Punteros y Arrays
- Paso de punteros a funciones
- **Resumen de operaciones con punteros**
- Punteros y const

4 Tipo de Dato Referencia

- Punteros a función

Resumen de operaciones con punteros

Expresión	equiv	Efecto
&T		obtenemos puntero (dirección) al dato de tipo T
*p		devuelve el elemento almacenado en la dirección p
p=q		asigna a p el valor de q
p[n]	*(p+n)	obtiene el elemento en la posición n (n puede ser neg)
pe->ai	(*pe).ai	atributo ai de una estructura apuntada por el puntero
p++, p--		avanza/retrocede 1 posición
p+n	n+p	puntero al elemento n posiciones adelante
p-n		puntero al elemento n posiciones hacia atrás
p+=n	p=p+n	Avanza a n posiciones hacia adelante
p-=n	p=p-n	Retrocede p n posiciones hacia atrás
p-q		distancia entre p y q (número de elementos de tipo T)
p==q		chequea si p es igual a q
p<q		chequea si p está antes que q
p>q		chequea si p está después de q
p<=q		chequea si p no esta después de q
p>=q		chequea si p no está antes que q

1 Conceptos preliminares

2 Gestión de Memoria

3 Tipo de dato puntero

- Definición y Declaración de variables
- Operaciones con punteros
- Punteros y Arrays
- Paso de punteros a funciones
- Resumen de operaciones con punteros
- Punteros y const

4 Tipo de Dato Referencia

- Punteros a función

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

- Las siguientes expresiones son equivalentes:

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

Punteros y const

Ninguno sea const	double *p;
Sólo el dato apuntado sea const	const double *p;
Sólo el puntero sea const	double *const p;
Los dos sean const	const double *const p;

```
1 int A[10], q;  
2 int *p;  
3 p = &A[4];           //OK  
4 *p = 23               // OK Asignamos 23 a A[4]  
5 cout << *p << endl; // OK accedemos al dato  
6 p++;                 // OK Avanzamos p, modificamos su valor  
7 p = &q;              // OK Cambiamos el valor
```


Punteros y const

Ninguno sea const	double *p;
Sólo el dato apuntado sea const	const double *p;
Sólo el puntero sea const	double *const p;
Los dos sean const	const double *const p;

```
1 int A[10], q;  
2 const int *p;  
3 p = &A[4];           //OK  
4 *p = 23;             // ERROR A[4] no se puede modificar  
5 cout << *p << endl; // OK accedemos al dato  
6 p++;                // OK Avanzamos p, modificamos p  
7 p = &q;             // Cambiamos el valor de p  
8 *p = 12;            // ERROR no podemos modificar el dato
```

error: assignment of read-only location '* p'

Punteros y const

Ninguno sea const	double *p;
Sólo el dato apuntado sea const	const double *p;
Sólo el puntero sea const	double *const p;
Los dos sean const	const double *const p;

```
1 int A[10], i=7, q;  
2 int * const p=&i; // OK *p es un "alias" de i  
3 int * const p1; // Error constante NO inicializada  
4 int * const pos4A = &A[4]; // pos4A es un alias de A[4]  
5  
6 p = &A[4]; // Error p no se puede modificar  
7 p++; // Error p no se puede modificar  
8 p = &q; // Error p no se puede modificar  
9 *p = 23; // OK cambiamos el valor y tambien en i  
10 cout << *p << i; // OK accedemos al dato (23,23)  
11 i = 12; // OK modificamos i  
12 cout << *p << i; // OK accedemos al dato (12,12)
```

Punteros y const

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

Puntero constante a valor constante

```
1  int  i=7;
2  const int * const p=&i;    // OK *p es un "alias" de i
3  p++;    // Error el puntero no se puede movificar
4  *p=23;  // Error el valor almacenado no se puede modificar
5
```

Conversión * a const *

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

```
1 double a = 1.0;
2 double * const p=&a; // puntero constante a double
3 double * q;
4 const double *r;
5 // puntero no constante a double
6 q = p; // OK: q puede apuntar a cualquier dato
7 p = q; // Error: p es constante
8 r = q; // OK:
9 q = r; // Error: conversion no correcta
```

```
(linea 7) ...error:assignment of read-only variable 'p'
(linea 9) ...error:invalid conversion from 'const double*' to 'double*'
```

- 1 Conceptos preliminares
- 2 Gestión de Memoria
- 3 Tipo de dato puntero
 - Definición y Declaración de variables
 - Operaciones con punteros
 - Punteros y Arrays
 - Paso de punteros a funciones
 - Resumen de operaciones con punteros
 - Punteros y const
- 4 Tipo de Dato Referencia
 - Punteros a función

Tipo de Dato Referencia

Hemos visto que C++ tiene

- Variables u objetos normales, que almacenan el valor directamente
- Punteros, que contienen la dirección de otra variable u objeto (o nullptr) y que pueden ser de-referenciados para obtener el valor es dicha dirección

Pero C++ tiene otro tipo de dato

- Referencias que actúa como un “alias” a otro objeto
- Referencia a variable normal
- Referencia constante a valor

```
int val = 2;
int &ref = value;
// ref es un alias de
// val
cout << ref;
ref=7: // eqv. val =7
```

```
int val = 2;
const int &ref = value;
// ref es un alias de
// val
cout << ref;
ref=7 // ERROR
```

- r-refs
(no las
veremos)

Referencias: Propiedades

- Internamente se suelen implementar como un puntero constante $T^* \text{ const}$
- Deben ser inicializadas a un objeto existente
- No pueden reasignarse
- No pueden contener un nullptr (al ser un $T^* \text{ const}$ no tiene sentido) lo que hace que sean mas seguras que $T^* \text{ const}$: No podemos intentar acceder a la dirección nula.

IMPORTANCIA

Se suelen utilizar como parámetros de funciones, donde actúan como un alias del argumento, permitiendo pasar la referencia (el puntero), ahorrándonos la copia del objeto (que puede ser costosa)

Paso de parámetros

Por valor o copia: ... (T x)

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

```
struct img{
    int datos[1024][1024];
    string nombre;
};

void cambia(int x, int y){
    int z;
    z = x; x = y; y = z;
}

void fA(int A, img B){
    cout << B.nombre << endl;
    B.nombre = "alhambra";
}
```

```
int a=3,b=5;
img mifoto;
cambia(a,b); // No hace nada!
fA(3,mifoto); //Ineficiente
cout << mifoto.nombre << endl;
```


Paso de parámetros

Por referencia o variable (T &x)

- No realiza una copia del parámetro actual en el formal, sino un vínculo entre ellos - **una referencia** - 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.

```
1 struct img{  
2     int datos[1024][1024];  
3     string nombre;  
4 };  
5 void cambia(int &x, int &y){  
6     int z;  
7     z = x; x = y; y = z;  
8 }  
9 void fA(int A, img &B){  
10     cout << B.nombre;  
11     B.nombre = "alhambra";  
12 }
```

```
int a=3,b=5;  
img mifoto;  
cambia(a,b); // cambia  
fA(3,mifoto); //Eficiente  
cout << mifoto.nombre << endl;
```

Paso de parámetros

Por referencia constante (`const T & x`)

- No realiza una copia del parámetro actual en el formal, sino un vínculo entre ellos - **una referencia**
- Al ser (`const T & x`) NO es posible modificar el parámetro formal. Sólo se puede consultar el valor, pero nunca puede ser modificado

```
1 struct img{
2     int datos[1024][1024];
3     string nombre;
4 };
5 void cambia(int &x, int &y){
6     int z;
7     z = x; x = y; y = z;
8 }
9 void fA(int A, const img &B){
10     cout << B.nombre;
11     B.nombre = "alhambra"; //Error
12 }
```

```
int a=3,b=5;
img mifoto;
cambia(a,b); // cambia
fA(3,mifoto); //Eficiente
```

Paso de parámetros

Paso de punteros

- Como hemos visto, hay otra forma de obtener el mismo efecto que el paso por referencia y es utilizar el paso por punteros a datos de tipo T:
 - `funcion(T * x)`
 - `funcion(const T*x)`
 - `funcion(T *const x)`
 - `funcion(const T *const x)`

con el comportamiento esperado en cada caso.

Este paso es problemático,

- errores al intentar acceder a un `nullptr`
- errores al mover el puntero fuera de la zona reservada

por lo que salvo que sea estrictamente necesario no es conveniente su uso.

En el caso de arrays estáticos, la conversión implícita a `T*` casi nos “obliga”, pero en estas situaciones se aconseja el uso del TDA `std::array`)

Normas generales

- Si un tipo, Tgrande, que se va a pasar a una función es **costoso** de copiar (estructuras, clases) pasar:
 - `funcion(const Tgrande & x)` si no se va a modificar dentro de la función
 - `funcion(Tgrande & x)` si se modifica, dentro de la función se cambian sus valores

pasar una referencia tendrá menor coste.

- Si Tsimple es un tipo simple (int, float, double, ...)
 - `funcion(Tsimple x)` si no se va a modificar dentro de la función
 - `funcion(Tsimple & x)` si se modifica, dentro de la función se cambian sus valores
- Si Tdirecc es una dirección (tipo T*), como es el caso de pasar una array a la función, esto es, `funcion(T x[N],int N)`
 - asegurarnos de que no accedemos a direcciones impropias, es necesario tener el parámetro N que indica el tamaño del array
 - chequear el caso de que el valor sea nullptr

Referencias: Devolviendo referencias en una función

Una función puede:

- devolver un valor

```
int  fA(int x, int y){  return x+y;  }  
int  z = fA(2,3);
```

- devolver un puntero (la dirección de memoria debe existir al salir de la función)

```
int * fB(int x[]){  return &x[2];  }  
int  aa[4]={1,2,3,4};  
int  *p = fB(aa);  
*p=100; *(fB(aa))=100;
```

- devolver una referencia (la referencia debe ser valida, el dato debe existir al salir de la función)

```
int & fC(array<int,10> x){  return x[2];  }  
int  aa[4]={1,2,3,4};  
int  &r = fC(aa);  
r=100;  fC(aa)=100;
```

Profundizar mas en el tipo puntero ...

Hemos visto como un puntero es otro tipo de dato, por tanto podremos ...

- Declarar un array de punteros

```
int* arrayPunts[3];  
int x,y,z;  
arrayPunts[0] = &x; arrayPunts[1]= &y; arrayPunts[2] = &z;
```

Usos posibles:

- Podemos usar un array de punteros a los elementos de otro array para ordenar sus elementos sin modificar el array original.
- Implementar matrices de varias dimensiones (siguiente tema)
- Pasar un vector de cadenas-C (veremos luego)

```
int main(int argc, char *argv[]) {  
    cout << argv[0]<< " " << argv[1] << endl;
```

- Puede ser pasado por referencia a una función, cambiamos su valor

```
void f(int *p, int x, int y) { if (x>y) p = &y; }  
int main() {  
    int a=3, b=5;  
    int *q = &a; cout << *q << endl;  
    f(q,a,b);  
    cout << *q << endl;
```

1 Conceptos preliminares

2 Gestión de Memoria

3 Tipo de dato puntero

- Definición y Declaración de variables
- Operaciones con punteros
- Punteros y Arrays
- Paso de punteros a funciones
- Resumen de operaciones con punteros
- Punteros y const

4 Tipo de Dato Referencia

- Punteros a función

Puntero a función

Contiene la dirección de memoria de una función, o sea la dirección donde comienza el código que realiza la tarea de la función apuntada.

Con estos punteros podemos hacer las siguientes operaciones:

- Usarlos como parámetro a una función. Una función recibe otra función
- Ser devueltos por una función con `return`.
- Crear arrays de punteros a funciones.
- Asignarlos a otras variables puntero a función.
- Usarlos para llamar a la función apuntada.

Volveremos a ellos al final del curso.