FACULTAD DE INGENIERÍA
UNIVERSIDAD NACIONAL DE MISIONES

Apunte Lenguaje C

Parte 2

Año 2007

CAPITULO 10. Punteros

Los punteros proporcionan la mayor parte de la potencia al C y C++, y marcan la principal diferencia con otros lenguajes de programación.

Una buena comprensión y un buen dominio de los punteros pondrá en tus manos una herramienta de gran potencia. Un conocimiento mediocre o incompleto te impedirá desarrollar programas eficaces.

Por eso le dedicaremos mucha atención y mucho espacio a los punteros. Es muy importante comprender bien cómo funcionan y cómo se usan.

Para entender qué es un puntero veremos primero cómo se almacenan los datos en un ordenador.

La memoria de un ordenador está compuesta por unidades básicas llamadas bits. Cada bit sólo puede tomar dos valores, normalmente denominados alto y bajo, ó 1 y 0. Pero trabajar con bits no es práctico, y por eso se agrupan.

Cada grupo de 8 bits forma un byte u octeto. En realidad el microprocesador, y por lo tanto nuestro programa, sólo puede manejar directamente bytes o grupos de dos o cuatro bytes. Para acceder a los bits hay que acceder antes a los bytes. Y aquí llegamos al quid, cada byte tiene una dirección, llamada normalmente dirección de memoria.

La unidad de información básica es la palabra, dependiendo del tipo de microprocesador una palabra puede estar compuesta por dos, cuatro, ocho o dieciséis bytes. Hablaremos en estos casos de plataformas de 16, 32, 64 ó 128 bits. Se habla indistintamente de direcciones de memoria, aunque las palabras sean de distinta longitud. Cada dirección de memoria contiene siempre un byte. Lo que sucederá cuando las palabras sean de 32 bits es que accederemos a posiciones de memoria que serán múltiplos de 4.

Todo esto sucede en el interior de la máquina, y nos importa más bien poco. Podemos saber qué tipo de plataforma estamos usando averiguando el tamaño del tipo int, y para ello hay que usar el operador "sizeof()".

Por ejemplo:

*cout << "Plataforma de " << 8*sizeof(int) << " bits";*

Ahora veremos cómo funcionan los punteros.

Un puntero es un tipo especial de variable que contiene, ni más ni menos que, una dirección de memoria. Por supuesto, a partir de esa dirección de memoria puede haber cualquier tipo de objeto: un char, un int, un float, un array, una estructura, una función u otro puntero. Seremos nosotros los responsables de decidir ese contenido.

Intentemos ver con mayor claridad el funcionamiento de los punteros. Podemos considerar la memoria del ordenador como un gran array, de modo que podemos acceder a cada celda de memoria a través de un índice. Podemos considerar que la primera posición del array es la 0 celda[0].

Si usamos una variable para almacenar el índice, por ejemplo, indice=0, entonces celda[0] == celda[indice]. Prescindiendo de la notación de los arrays, el índice se comporta exactamente igual que un puntero.

El puntero índice podría tener por ejemplo, el valor 3, en ese caso, *índice tendría el valor 'valor3'.

Las celdas de memoria existirán independientemente del valor de índice, o incluso de la existencia de indice, por lo tanto, la existencia del puntero no implica nada más que eso, pero no que el valor de la dirección que contiene sea un valor válido de memoria.

Dentro del array de celdas de memoria existirán zonas que contendrán programas y datos, tanto del usuario como del propio sistema operativo o de otros programas, el sistema operativo se encarga de gestionar esa memoria, prohibiendo o protegiendo determinadas zonas.

El propio puntero, como variable que es, ocupará ciertas direcciones de memoria.

En principio, debemos asignar a un puntero, o bien la dirección de un objeto existente, o bien la de uno creado explícitamente durante la ejecución del programa. El sistema operativo suele controlar la memoria, y no tiene por costumbre permitir el acceso al resto de la memoria.

10.1. Declaración de punteros

Los punteros se declaran igual que el resto de las variables, pero precediendo el identificador con el operador de indirección, (*), que leeremos como "puntero a".

Sintaxis:

*<tipo> *<identificador>;*

Por ejemplo:

*int *entero;*

*char *carácter;*

*struct stPunto *punto;*

Los punteros siempre apuntan a un objeto de un tipo determinado, en el ejemplo, "entero" siempre apuntará a un objeto de tipo "int".

La forma:

<tipo> <identificador>;*

con el (*) junto al tipo, en lugar de junto al identificador de variable, también está permitida.

Veamos algunos matices.

Por ejemplo:

*int *entero;*

equivale a:

int entero;*

INFORMACIÓN A TENER EN CUENTA:

Debes tener muy claro que "entero" es una variable del tipo "puntero a int", que **"*entero" NO es una variable de tipo "int"**.

Como pasa con todas las variables en C++, cuando se declaran sólo se reserva espacio para almacenarlas, pero no se asigna ningún valor inicial, el contenido de la variable permanecerá sin cambios, de modo que el valor inicial del puntero será aleatorio e indeterminado. Debemos suponer que contiene una dirección no válida.

Si "entero" apunta a una variable de tipo "int", "*entero" será el contenido de esa variable, pero no olvides que "*entero" es un operador aplicado a una variable de tipo "puntero a int", es decir "*entero" es una expresión, no una variable, que significa: ”el contenido a donde apunta el puntero” ó “ contenido de la dirección a la que apunta”.

11.2. Obtener punteros a variables

Para averiguar la dirección de memoria de cualquier variable usaremos el operador de dirección (&), que leeremos como "dirección de".

Por supuesto, los tipos tienen que ser "compatibles", no podemos almacenar la dirección de una variable de tipo "char" en un puntero de tipo "int".

Por ejemplo:

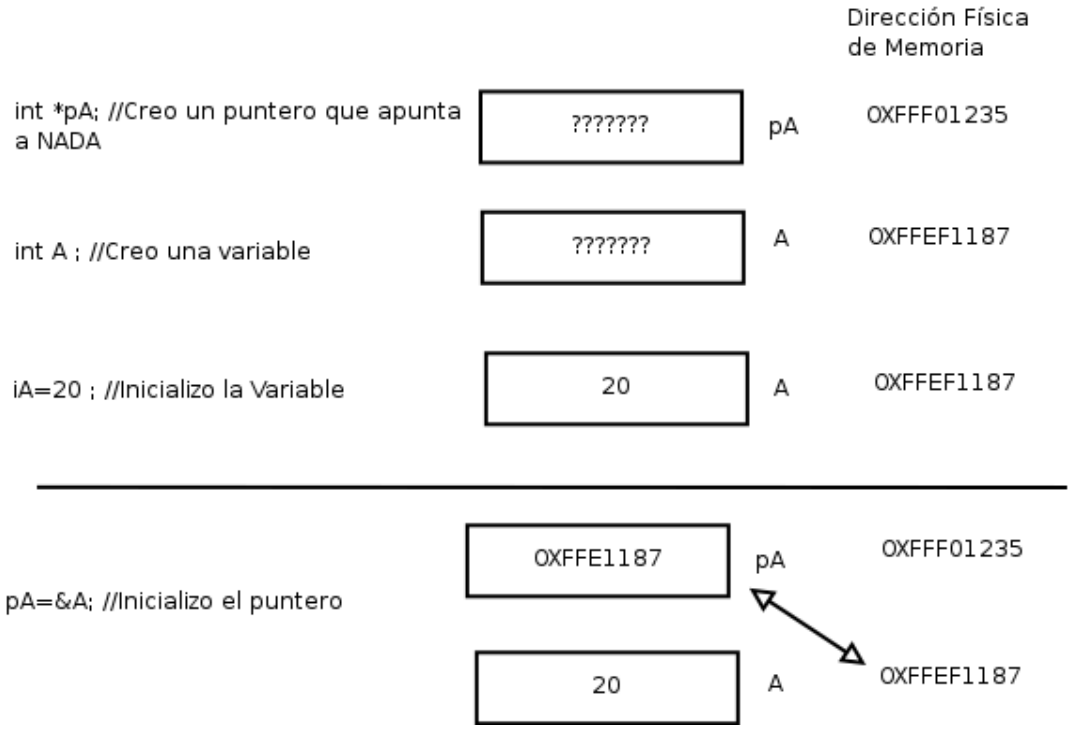
int A;

*int *pA;*

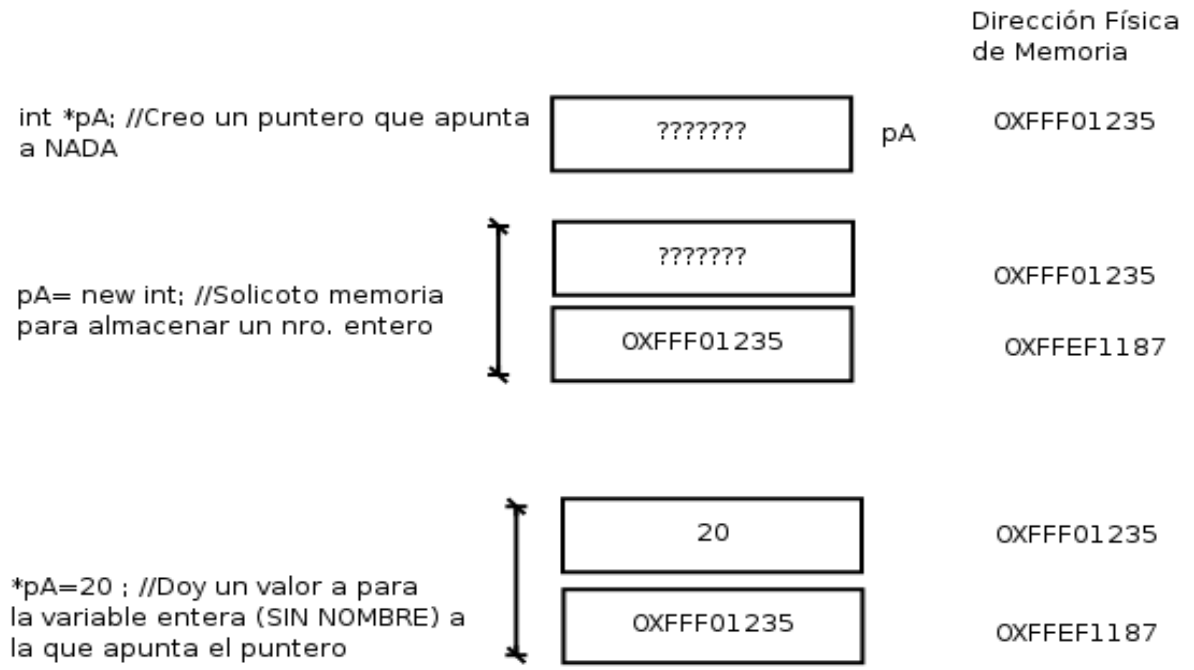
pA = &A;

Según este ejemplo, pA es un puntero a int que apunta a la dirección donde se almacena el valor del entero A.

Trataremos de graficar lo visto hasta ahora.



Veamos como sería solicitando espacio o memoria de un modo dinámico para almacenar un entero, es importante destacar que en este ejemplo **SOLO SE USA EL PUNTERO, NO EXISTE NOMBRE DE VARIABLE**.



10.3. Diferencia entre punteros y variables

Declarar un puntero no creará un objeto. Por ejemplo: `int *entero;` no crea un objeto de tipo "int" en memoria, sólo crea una variable que puede contener una dirección de memoria. Se puede decir que existe físicamente la variable "entero", y también que esta variable puede contener la dirección de un objeto de tipo "int".

Por ejemplo:

```
int A, B;
int *entero;
...
B = 213; /* B vale 213 */
entero = &A;   /* entero apunta a la
                  dirección de la variable A */
```

```
*entero = 103; /* equivale a la línea A = 103; */
B = *entero;   /* equivale a B = A; */
...
```

En este ejemplo vemos que "entero" puede apuntar a cualquier variable de tipo "int", y que podemos hacer referencia al contenido de dichas variables usando el operador de indirección (*).

Como todas las variables, los punteros también contienen "basura" cuando son declaradas. Es costumbre dar valores iniciales nulos a los punteros que no apuntan a ningún sitio concreto:

entero = NULL;

caracter = NULL;

NULL es una constante, que está definida como cero en varios ficheros de cabecera, como "cstdio" o "iostream", y normalmente vale 0.

10.4. Correspondencia entre arrays y punteros

Existe una equivalencia casi total entre arrays y punteros. Cuando declaramos un array estamos haciendo varias cosas a la vez:

- Declaramos un puntero del mismo tipo que los elementos del array, y que apunta al primer elemento del array.
- Reservamos memoria para todos los elementos del array. Los elementos de un array se almacenan internamente en el ordenador en posiciones consecutivas de la memoria.

La principal diferencia entre un array y un puntero es que el nombre de un array es un puntero constante, no podemos hacer que apunte a otra dirección de memoria. Además, el compilador asocia una zona de memoria para los elementos del array, cosa que no hace para los elementos apuntados por un puntero auténtico.

Por ejemplo:

```
int vector[10];
int *puntero;
puntero = vector;        /*  Equivale a puntero = &vector[0]; */
                         /* esto se lee como "dirección del primer de vector" */
*puntero++;              /* Equivale a vector[0]++; */
```

puntero++; */* puntero equivale a &vector[1] */*

¿Qué hace cada una de estas instrucciones?

La primera incrementa el contenido de la memoria apuntada por "puntero", que es vector[0].

La segunda incrementa el puntero, esto significa que apuntará a la posición de memoria del siguiente "int", pero no a la siguiente posición de memoria. El puntero no se incrementará en una unidad, como tal vez sería lógico esperar, sino en la longitud de un "int".

Observación: El nombre del Array es un puntero al Arreglo !!

Análogamente la operación:

puntero = puntero + 7;

No incrementará la dirección de memoria almacenada en "puntero" en siete posiciones, sino en 7*sizeof(int).

Veamos otro ejemplo:

```
#include <iostream>
using namespace std;
int main()
{//letra es un array que  tiene dimensión 2 ,"A" es distinto de 'A' !!
char letra[]="A";
// Alternativa : char letra[2]; letra[0]='A', letra[1]=0;//seria lo mismo
//letra es un puntero a un arreglo que termina con caracter nulo
//este tipo de arreglo de caracteres que termina con 0 se llama string
cout << letra << endl;
//ver que cout recibe un puntero a un string e imprime todo !!!
return 0;
}
```

10.5. Operaciones con punteros

Aunque no son muchas las operaciones que se pueden hacer con los punteros, cada una tiene sus peculiaridades.

10.5.1. Asignación

Ya hemos visto cómo asignar a un puntero la dirección de una variable.

También podemos asignar un puntero a otro, esto hará que los dos apunten a la misma posición:

```
int *q, *p;
int a;
q = &a;      /* q apunta al contenido de a */
p = q;       /* p apunta al mismo sitio, es decir, al contenido de a */
```

10.5.2. Operaciones aritméticas

También hemos visto como afectan a los punteros las operaciones de suma con enteros.

Las restas con enteros operan de modo análogo.

Pero, ¿qué significan las operaciones de suma y resta entre punteros?

Por ejemplo:

```
int vector[10];
int *p, *q;
p = vector;              /* Equivale a p = &vector[0]; */
q = &vector[4];   /* apuntamos al 5º elemento */
cout << q-p << endl;
```

El resultado será 4, que es la "distancia" entre ambos punteros.

Normalmente este tipo de operaciones sólo tendrá sentido entre punteros que apunten a elementos del mismo array.

La suma de punteros no está permitida.

10.5.3. Comparación entre punteros

Comparar punteros puede tener sentido en la misma situación en la que lo tiene restar punteros, es decir, averiguar posiciones relativas entre punteros que apunten a elementos del mismo array.

Existe otra comparación que se realiza muy frecuente con los punteros.

Para averiguar si estamos usando un puntero es corriente hacer la comparación:

if(NULL != p)

if(p)

if(NULL == p)

if(!p)

10.6. Punteros genéricos

Es posible declarar punteros sin tipo concreto:

Sintaxis:

*void *<identificador>;*

Estos punteros pueden apuntar a objetos de cualquier tipo.

Por supuesto, también se puede emplear el "*casting*" con punteros.

Sintaxis:

*(<tipo> *)<variable puntero>*

Recordemos que el cambio o conversión de tipo de dato que se realiza de forma implícita en el procesador cuando encuentra expresiones que contienen diferentes tipos de dato, pero podemos hacerlo de una forma en que programador puede forzar un cambio de tipo de forma explícita. Este cambio se llama cambio por promoción, o *casting*.

Por ejemplo:

```
#include <iostream>
using namespace std;
int main() {
  char cadena[10] = "Hola";
  char *c;
  int *n;                    //puntero a un entero
  void *v;
  c = cadena;                // c apunta a cadena
  n = (int *)cadena;         // usa casting, n también apunta a
                             // cadena
```

```
    v = (void *)cadena;           // v también
    cout << "carácter: " << *c << endl;
    cout << "entero:   " << *n << endl;
    cout << "float:    " << *(float *)v << endl;
    cin.get(); //para la ejecucion hasta que se presione una tecla
    return 0;
}
```

El resultado será:

carácter: H

entero: 1634496328

float: 2.72591e+20

Vemos que tanto "cadena" como los punteros "n", "c" y "v" apuntan a la misma dirección, pero cada puntero tratará la información que encuentre allí de modo diferente, para "c" es un carácter y para "n" un entero. Para "v" no tiene tipo definido, pero podemos hacer "casting" con el tipo que queramos, en este ejemplo con float.

INFORMACIÓN A TENER EN CUENTA:

El tipo de línea del tercer "cout" es lo que suele asustar a los no iniciados en C y C++, y se parece mucho a lo que se conoce como código ofuscado. Parece como si en C casi cualquier expresión pudiese compilar.

10.7. Punteros a Funciones

Los punteros a función son uno de los recursos más potentes y flexibles de C/C++, permitiendo técnicas de programación muy eficientes. Por ejemplo, escribir funciones que manejan diferentes tipos de datos; diseñar algoritmos muy compactos ("function dispatchers"), que pueden sustituir largas cadenas if...else o switch , o alterar el flujo de ejecución del programa, modificando el orden de llamadas a funciones en base a determinadas prioridades ("adaptive program flow"). Así mismo, resultan de gran ayuda en programas de simulación y modelado. Sin embargo, los autores están generalmente de acuerdo en que constituyen para el principiante uno de los puntos más confusos del lenguaje.

Para hablar de punteros a funciones, previamente hay que establecer que las funciones tengan "dirección". Del mismo modo que en una matriz se asume que su dirección es la del primer elemento, se asume también que la dirección de una función es la del segmento de código ("Code segment") donde comienza el código de dicha función . Es decir, la dirección de memoria a que se transfiere el control cuando se la invoca (su punto de comienzo).

Una vez establecido esto, no tiene que extrañar que puedan definirse variables de un tipo especial para apuntar a estas direcciones. Técnicamente un **puntero-a-función es una variable que guarda la dirección de comienzo de la función.** Pero como tendremos ocasión de comprobar, la mejor manera de pensar en ellos es considerarlos como una especie de "alias" de la función, aunque con una importante cualidad añadida: que pueden ser utilizados como argumentos de otras funciones.

```
#include <iostream>
using namespace std;
int suma(int,int);
int producto(int,int);
int main(){
        int (*pf[2])(int,int);  // Array de punteros a función con arg. int
        int e1,e2;
        int opc;
        pf[0] = suma; //Inicializo el 1er elemento del arreglo a suma
        pf[1] = producto; //Inicializo el 2do elemento del arreglo a
                                producto
        cout<<"Por favor Ingrese un entero : ";
        cin>>e1;
        cout<<"Por favor Ingrese otro entero : ";
        cin>>e2;
        cout<<"Ingrese 0 para sumar o 1 para multiplicar : "<<endl;
        cin>>opc;
        if((opc==0)||(opc==1)) cout<<pf[opc](e1,e2);
        else cout<<"opcion Incorrecta";
        return 0;
}
```

```
int suma(int x, int y){return (x+y);}
int producto(int x, int y){return (x*y);}
```

En este caso vemos que existe un arreglo de punteros a funciones que retornan enteros y que reciben dos enteros como argumentos. int (*pf[2]) (int,int).

Luego se inicializa cada elemento del arreglo de punteros a una función:

pf[0] = suma; //Inicializo el 1er elemento del arreglo a suma

pf[1] = producto; //Inicializo el 2do elemento del arreglo a producto

Ahora se puede usar una llamada como si fuera un argumento.

if((opc==0)||(opc==1)) cout<<pf[opc](e1,e2);

Que llamará a suma (pf[0]) o producto (pf[1]) según corresponda.

10.8. Ejemplos de uso de punteros

10.8.1. Ejemplo 1

Primero un ejemplo que ilustra la diferencia entre un array y un puntero:

```
#include <iostream>
using namespace std;
int main() {
  char cadena1[] = "Cadena 1";
  char *cadena2 = "Cadena 2";
  cout << cadena1 << endl;
  cout << cadena2 << endl;
  //cadena1++;              // Ilegal, cadena1 es constante
  cadena2++;                // Legal, cadena2 es un puntero
  cout << cadena1 << endl;
  cout << cadena2 << endl;
  cout << cadena1[1] << endl;
  cout << cadena2[0] << endl;
  cout << cadena1 + 2 << endl;
  cout << cadena2 + 1 << endl;
  cout << *(cadena1 + 2) << endl;
```

```
  cout << *(cadena2 + 1) << endl;
  cin.get();
  return 0;
}
```

Aparentemente, y en la mayoría de los casos, cadena1 y cadena2 son equivalentes, sin embargo hay operaciones que están prohibidas con los arrays, ya que son punteros constantes.

10.8.2. Ejemplo 2

```
#include <iostream>
using namespace std;
int main() {
  char Mes[][11] = { "Enero", "Febrero", "Marzo", "Abril",
    "Mayo", "Junio", "Julio", "Agosto",
    "Septiembre", "Octubre", "Noviembre", "Diciembre"};
  char *Mes2[] = { "Enero", "Febrero", "Marzo", "Abril",
    "Mayo", "Junio", "Julio", "Agosto",
    "Septiembre", "Octubre", "Noviembre", "Diciembre"};
  cout << "Tamaño de Mes: " << sizeof(Mes) << endl;
  cout << "Tamaño de Mes2: " << sizeof(Mes2) << endl;
  cout << "Tamaño de cadenas de Mes2: "
      << &Mes2[11][10]-Mes2[0] << endl;
  cout << "Tamaño de Mes2 + cadenas : "
      << sizeof(Mes2)+&Mes2[11][10]-Mes2[0] << endl;
  cin.get();
  return 0;
}
```

En este ejemplo declaramos un array "Mes" de dos dimensiones que almacena 12 cadenas de 11 caracteres, 11 es el tamaño necesario para almacenar el mes más largo (en caracteres): "Septiembre".

Después declaramos "Mes2" que es un array de punteros a char, para almacenar la misma información. La ventaja de este segundo método es que no necesitamos contar la longitud de las cadenas para calcular el espacio que necesitamos, cada puntero de Mes2 es una cadena de la longitud adecuada para almacenar cada mes.

Parece que el segundo sistema es más económico en cuanto al uso de memoria, pero hay que tener en cuenta que además de las cadenas también se almacenan los doce punteros.

El espacio necesario para almacenar los punteros lo dará la segunda línea de la salida. Y el espacio necesario para las cadenas lo dará la tercera línea.

Si las diferencias de longitud entre las cadenas fueran mayores, el segundo sistema sería más eficiente en cuanto al uso de la memoria.

10.8.3. Ejemplo 3

Ejemplo de funciones que reciben punteros como argumento:

Este código me dice cuantos caracteres tiene la cadena introducida por teclado y permite buscar la ocurrencia de un caracter dentro de la cadena.

```
#include <iostream>
#define LNG     100
#define ENTER     0x0A
using namespace std;
int LongitudCadena(char *cadena);
int BuscarCaracter(char *cadena, char c);
int main()
{
   int i;
   char texto[LNG], c;
   for(i=0; i<LNG-1; i++) {
      c = fgetc(stdin);
      texto[i] = c;
      if(c == ENTER) {
         texto[i] = 0;
                        cout << "'" << texto << "' tiene " << LongitudCadena(texto) << " caracteres" << endl;
         i=0;
      }
   }
   cout << "La cadena es muy larga!" << endl;
   return 0;
}
```

```
int LongitudCadena(char *cadena)  //calcula la longitud de la cadena
{
   int i;
   for(i=0; cadena[i]; i++)
      ;
   return i;
}
```

Como variante presentamos este código me dice cuantos caracteres tiene la cadena introducida por teclado y permite buscar la ocurrencia de un caracter dentro de la cadena.

```
#include <iostream>
#define LNG     100  //maxima logintud de la cadena
using namespace std;
int LongitudCadena(char *cadena);
int BuscarCaracter(char *cadena, char c);
int main()
{   int i;
   char texto[LNG];
   char c;
   for(;;) {
      cin >> texto;
      cout << "'"     << texto << "' posee " <<
         LongitudCadena(texto) << " caracteres" << endl;
      cout << "Buscar caracter: ";
      cin >> c;
      if((i = BuscarCaracter(texto, c)) == -1)
          cout << "La cadena no posee el caracter '" << c << "'" <<
endl;
      else
           cout << "Primer ocurrencia de '" << c << "': " << i+1 <<
endl;
   }
   return 0;
```

```
}

int LongitudCadena(char *cadena) //cuenta el lago de la cadena
{  int i;
   for(i=0; cadena[i]; i++)       ;
   return i;}
int BuscarCaracter(char *cadena, char c)  //busca las ocurrencias de un caracter en la cadena
{  int i;
   for(i=0; *cadena; i++)
      if(*cadena++ == c)
       return i;
   return -1;
}
```

10.9. Variables dinámicas

Donde mayor potencia desarrollan los punteros es cuando se unen al concepto de memoria dinámica.

Cuando se ejecuta un programa, el sistema operativo reserva una zona de memoria para el código o instrucciones del programa y otra para las variables que se usan durante la ejecución. A menudo estas zonas son la misma zona, es lomo la pila, que se usa, entre otras cosas, para intercambiar datos entre funciones. El resto, la memoria que no se usa por nin que se llama memoria local. También hay otras zonas de memoria, cogún programa es lo que se conoce como "heap" o montón. Cuando nuestro programa use memoria dinámica, normalmente usará memoria del montón, y no se llama así porque sea de peor calidad, sino porque suele haber realmente un montón de memoria de este tipo.

C++ dispone de dos operadores para acceder a la memoria dinámica, son "*new*" y "*delete*".

En C estas acciones se realizan mediante funciones de la librería estándar "*stdio.h*" que son maloc (Memory Alocation) y free.

INFORMACIÓN A TENER EN CUENTA:

Hay una regla de oro cuando se usa memoria dinámica, toda la memoria que se reserve durante el programa hay que liberarla antes de salir del programa. No seguir esta regla es una actitud muy irresponsable, y en la mayor parte de los casos tiene consecuencias desastrosas. No se fien de lo que diga el compilador, de que estas variables se liberan solas al terminar el programa, no siempre es verdad.

Veremos con mayor profundidad los operadores "new" y "delete" en otros capítulos, por ahora veremos un ejemplo.

Por ejemplo:

```
#include <iostream>
using namespace std;
int main() {
  int *a;
  char *b;
  float *c;
  struct stPunto {
    float x,y;
  } *d;
  a = new int;
  b = new char;
  c = new float;
  d = new stPunto;
  *a = 10;
  *b = 'a';
  *c = 10.32;
  d->x = 12; d->y = 15;
  cout << "a = " << *a << endl;
  cout << "b = " << *b << endl;
  cout << "c = " << *c << endl;
  cout << "d = (" << d->x << ", "
       << d->y << ")" << endl;
```

```
    delete a;
    delete b;
    delete c;
    delete d;
    cin.get();
    return 0;
}
```

Y mucho cuidado!! si pierdes un puntero a una variable reservada dinámicamente, no podrás liberarla.

Por ejemplo:

```
int main()
{
  int *a;
   a = new int;          // variable dinámica
  *a = 10;
  a = new int;          // nueva variable dinámica,
                        // se pierde el puntero a la anterior
  *a = 20;
  delete a;                   // sólo liberamos la última reservada
  return 0;
}
```

En este ejemplo vemos cómo es imposible liberar la primera reserva de memoria dinámica. Si no la necesitábamos habría que liberarla antes de reservarla otra vez, y si la necesitamos, hay que guardar su dirección, por ejemplo con otro puntero.

10.10. Ejercicios

1. Escribir un programa con una función que calcule la longitud de una cadena de caracteres. El nombre de la función será LongitudCadena, debe devolver un "int", y como parámetro de entrada debe tener un puntero a "char". En "main" probar con distintos tipos de cadenas: arrays y punteros.

2. Escribir un programa con una función que busque un carácter determinado en una cadena. El nombre de la función será BuscaCaracter, debe devolver un "int" con la posición en que fue encontrado el carácter, si no se encontró volverá con -1. Los parámetros de entrada serán una cadena y un carácter. En la función "main" probar con distintas cadenas y caracteres.

3. ¿De qué tipo es cada una de las siguientes variables?

a) `int* a,b;`

- ○ a puntero, b puntero
- ○ a puntero, b entero
- ○ a entero, b puntero
- ○ a entero, b entero

b) `int *a,b;`

- ○ a puntero, b puntero
- ○ a puntero, b entero
- ○ a entero, b puntero
- ○ a entero, b entero

c) `int *a,*b;`

- ○ a puntero, b puntero
- ○ a puntero, b entero
- ○ a entero, b puntero
- ○ a entero, b entero

d) `int* a,*b;`

- ○ a puntero, b puntero
- ○ a puntero, b puntero doble
- ○ a entero, b puntero
- ○ a entero, b puntero doble

4. Considerando las siguientes declaraciones y sentencias:

```
int array[]={1,2,3,4,5,6};
```

```
int *puntero;
puntero = array;
puntero++;
*puntero=*puntero+6;
puntero=puntero+3;
puntero=puntero-puntero[-2];
int x=puntero-array;
```

a) ¿Cuál es el valor de x?

- ○1
- ○2
- ○3
- ○4

b) ¿Cual es el valor de array[1]?

- ○2
- ○4
- ○6
- ○8

5. ¿Cuánta memoria ocupa un Puntero?

6. ¿La cantidad de memoria que ocupa el puntero depende del tipo de variable a la que apunta?

CAPITULO 11. Estructuras

11.1. Estructuras

Las estructuras son el segundo tipo de datos estructurados que veremos.

Las estructuras nos permiten agrupar varios datos, aunque sean de distinto tipo, que mantengan algún tipo de relación, permitiendo manipularlos todos juntos, con un mismo identificador, o por separado.

Las estructuras son llamadas también muy a menudo registros, o en inglés "records". Y son estructuras análogas en muchos aspectos a los registros de bases de datos. Y siguiendo la misma analogía, cada variable de una estructura se denomina a menudo campo, o "field".

Sintaxis:

```
struct [<identificador>] {
  [<tipo> <nombre_variable>[,<nombre_variable>,...]];
  .
} [<variable_estructura>[,<variable_estructura>,...];
```

El nombre de la estructura es un nombre opcional para referirse a la estructura.

Las variables de estructura son variables declaradas del tipo de la estructura, y su inclusión también es opcional. Sin bien, al menos uno de estos elementos debe existir, aunque ambos sean opcionales.

En el interior de una estructura, entre las llaves, se pueden definir todos los elementos que consideremos necesarios, del mismo modo que se declaran las variables.

Las estructuras pueden referenciarse completas, usando su nombre, como hacemos con las variables que ya conocemos, y también se puede

acceder a los elementos en el interior de la estructura usando el operador de selección (.), un punto.

También pueden declararse más variables del tipo de estructura en cualquier parte del programa, de la siguiente forma:

[struct] <identificador> <variable_estructura>
[,<variable_estructura>...];

En C++ la palabra "struct" es opcional en la declaración de variables. En C es obligatorio usarla.

Por ejemplo:

```
struct Persona {
  char Nombre[65];
  char Direccion[65];
  int AnyoNacimiento;
} Fulanito;
```

Este ejemplo declara a Fulanito como una variable de tipo Persona. Para acceder al nombre de Fulanito, por ejemplo para visualizarlo, usaremos la forma:

```
cout << Fulanito.Nombre;
```

IMPORTANTE:

Se buscará que las definiciones de la Estructuras sean GLOBALES, pero las instancias LOCALES.

11.2. Funciones en el interior de estructuras

C++, al contrario que C, permite incluir funciones en el interior de las estructuras. Normalmente estas funciones tienen la misión de manipular los datos incluidos en la estructura. Aunque esta característica se usa casi exclusivamente con las clases, como veremos más adelante, también puede usarse en las estructuras.

Dos funciones muy particulares son las de inicialización, o constructor, y el destructor. Veremos con más detalle estas funciones cuando asociemos las estructuras y los punteros.

El constructor es una función sin tipo de retorno y con el mismo nombre que la estructura. El destructor tiene la misma forma, salvo que el nombre va precedido el operador "~".

Por ejemplo:

Para ilustrar el uso de constructores:

Forma 1:

```
struct Punto {
  int x, y;
  Punto() {x = 0; y = 0;} // Constructor
} Punto1, Punto2;
```

Forma 2:

```
struct Punto {
  int x, y;
  Punto(); // Declaración del constructor
} Punto1, Punto2;
```

// Definición del constructor, fuera de la estructura

```
Punto::Punto() {
  x = 0;
  y = 0;
}
```

Si no usáramos un constructor, los valores de x e y para Punto1 y Punto2 estarían indeterminados, contendrían la "basura" que hubiese en la memoria asignada a estas estructuras durante la ejecución. Con las estructuras éste será el caso más habitual.

INFORMACIÓN A TENER EN CUENTA:

Para aquellos que usen un teclado español, el símbolo "~" se obtiene pulsando las teclas del teclado numérico 1, 2, 6, mientras se mantiene pulsada la tecla ALT, ([ALT]+126).

También mediante la combinación [Atl Gr]+[4] (la tecla [4] de la zona de las letras, no del teclado numérico).

INFORMACION COMPLEMENTARIA

Uso de espacios de trabajo

En C los espacios de trabajo reciben el nombre de *“namespace”*. Por defecto utilizamos el estándar (std) pero podemos definir tantos como sean necesarios.

Supongamos que en un curso tenemos dos alumnos que se llaman Juan Perez, cuando debamos llamarlos ¿quién responderá?

Bueno eso es incierto, algo parecido sucede en el lenguaje C, donde se pueden definir tipos de variables personales.

Sí, eso mismo ... le podemos dar el nombre que querramos. Pero ¿qué sucedería si queremos crear un tipo de datos y tenemos la mala suerte que dentro de una librería que usamos ese tipo de dato ya existe?

Es difícil saber si el nombre para el tipo de variables que queremos usar ya fué usado, en este caso el compilador tiraría un error y deberíamos cambiar el nombre, para evitar esto se definen espacios en los que se deben tener en cuenta determinados nombres. Sería algo parecido al ejemplo de Juan Perez.

Esto se soluciona ubicando explícitamente a cada alumno en una comisión distinta.
Solo con fines didácticos veremos un código en cual no compilaría pero la idea es destacar la referencia a distintos espacios de nombres.

Por ejemplo:

```
using namespace Ns1
{ int x; }
using namespace Ns2
{ int x; }
int main()
{
cout << endl <<" Ingrese el valor de x para el espacio de nombre Ns1: ";
cin >> Ns1:: x;
cout << endl <<" Ingrese el valor de x para el espacio de nombre Ns2: ";
```

```
cin >> Ns2:: x;
}
```

Mencionar aquí, sólo a título de información, que el constructor no tiene por qué ser único. Se pueden incluir varios constructores, pero veremos esto mucho mejor y con más detalle cuando veamos las clases.

Usando constructores nos aseguramos los valores iniciales para los elementos de la estructura. Veremos que esto puede ser una gran ventaja, sobre todo cuando combinemos estructuras con punteros, en capítulos posteriores.

También podemos incluir otras funciones, que se declaran y definen como las funciones que ya conocemos, salvo que tienen restringido su ámbito al interior de la estructura.

Por ejemplo:

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

struct stPareja {
  int A, B;
  int LeeA() { return A;} // Devuelve el valor de A
  int LeeB() { return B;} // Devuelve el valor de B
  void GuardaA(int n) { A = n;} // Asigna un nuevo valor a A
  void GuardaB(int n) { B = n;} // Asigna un nuevo valor a B
} Par;

int main() {
  Par.GuardaA(15);
  Par.GuardaB(63);
  cout << Par.LeeA() << endl;
  cout << Par.LeeB() << endl;
  cin.get();
  return 0;
}
```

En este ejemplo podemos ver cómo se define una estructura con dos campos enteros, y dos funciones para modificar y leer sus valores. El ejemplo es muy simple, pero las funciones de guardar valores se pueden elaborar para que no permitan determinados valores, o para que hagan algún tratamiento de los datos.

Por supuesto se pueden definir otras funciones y también constructores más elaborados y sobrecarga de operadores. Y en general, las estructuras admiten cualquiera de las características de las clases, siendo en muchos aspectos equivalentes.

Veremos estas características cuando estudiemos las clases, y recordaremos cómo aplicarlas a las estructuras.

11.3. Inicialización de estructuras

De un modo parecido al que se inicializan los arrays, se pueden inicializar estructuras, tan sólo hay que tener cuidado con las estructuras anidadas.

Por ejemplo:

```
struct A {
  int i;
  int j;
  int k;
};
struct B {
  int x;
  struct C {
    char c;
    char d;
  } y;
  int z;
};
A ejemploA = {10, 20, 30};
B ejemploB = {10, {'a', 'b'}, 20};
```

Cada nueva estructura anidada deberá inicializarse usando la pareja correspondiente de llaves "{}", tantas veces como sea necesario.

11.4. Asignación de estructuras

La asignación de estructuras está permitida, pero sólo entre variables del mismo tipo de estructura, salvo que se usen constructores, y funciona como la intuición dice que debe hacerlo.

Por ejemplo:

```
struct Punto {
  int x, y;
  Punto() {x = 0; y = 0;}
} Punto1, Punto2;
int main() {
  Punto1.x = 10;
  Punto1.y = 12;
  Punto2 = Punto1;
}
```

La línea:

```
Punto2 = Punto1;
```

equivale a:

```
Punto2.x = Punto1.x;
Punto2.y = Punto1.y;
```

11.5. Arrays de estructuras

La combinación de las estructuras con los arrays proporciona una potente herramienta para el almacenamiento y manipulación de datos.

Por ejemplo:

```
struct Persona {
  char Nombre[65];
  char Direccion[65];
  int AnyoNacimiento;
} Plantilla[200];
```

Vemos en este ejemplo lo fácil que podemos declarar el array Plantilla que contiene los datos relativos a doscientas personas.

Podemos acceder a los datos de cada uno de ellos:

cout << Plantilla[43].Direccion;

O asignar los datos de un elemento de la plantilla a otro:

Plantilla[0] = Plantilla[99];

Veamos un ejemplo de un Array de instancias 2 dimensiones o dicho de otra manera una matriz de instancias de una Estructura.-

```
#include <iostream>
using namespace std;
struct prueba {int x; char nombre[11];};
int main()
{int i,j,k=1;
prueba datos[3][3];  //Instacia de Estructura de 2 dimensiones.
for ( i=0; i < 3; i++)
{for (j=0; j < 3; j++)
        {datos[i][j].x=k;
        cout<< endl << "Ingrese Nombre: " << endl;
        cin.getline(datos[i][j].nombre,10);  //Permite ingresar hasta 10
        k++; }
}
for (i=0; i < 3; i++)
{       for (j=0; j < 3; j++)
        {cout << datos[i][j].x <<"   "<< datos[i][j].nombre <<endl;}
}
return 0;
}
```

11.6. Estructuras anidadas

También está permitido anidar estructuras, con lo cual se pueden conseguir superestructuras muy elaboradas.

Por ejemplo:

```
struct stDireccion {
  char Calle[64];
  int Portal;
  int Piso;
  char Puerta[3];
  char CodigoPostal[6];
  char Poblacion[32];
};
 struct stPersona {
  struct stNombre {
    char Nombre[32];
    char Apellidos[64];
  } NombreCompleto;
  stDireccion Direccion;
  char Telefono[10];
};
```

INFORMACIÓN A TENER EN CUENTA:

En general, no es una práctica corriente definir estructuras dentro de estructuras, ya que resultan tener un ámbito local, y para acceder a ellas se necesita hacer referencia a la estructura más externa.

Por ejemplo para declarar una variable del tipo stNombre hay que utilizar el operador de acceso (::):

stPersona::stNombre NombreAuxiliar;

Sin embargo para declarar una variable de tipo stDireccion basta con declararla:

stDireccion DireccionAuxiliar;

11.7. Operador "sizeof" con estructuras

Cuando se aplica el operador sizeof a una estructura, el tamaño obtenido no siempre coincide con el tamaño de la suma de sus campos.

Por ejemplo:

```
#include <iostream>
using namespace std;
 struct A {
   int x;
   char a;
   int y;
   char b;
};
struct B {
   int x;
   int y;
   char a;
   char b;
};
```

```
int main()
{
  cout << "Tamaño de int: "
       << sizeof(int) << endl;
  cout << "Tamaño de char: "
       << sizeof(char) << endl;
  cout << "Tamaño de estructura A: "
       << sizeof(A) << endl;
  cout << "Tamaño de estructura B: "
       << sizeof(B) << endl;
  cin.get();
  return 0;
}
```

El resultado, usando Anjuta , es el siguiente:

Tamaño de int: 4

Tamaño de char: 1

Tamaño de estructura A: 16

Tamaño de estructura B: 12

Si hacemos las cuentas, en ambos casos el tamaño de la estructura debería ser el mismo, es decir, 4+4+1+1=10 bytes. Sin embargo en el caso de la estructura A el tamaño es 16 y en el de la estructura B es 12, ¿por qué?

La explicación es algo denominado alineación de bytes (byte-aling). Para mejorar el rendimiento del procesador no se accede a todas las posiciones de memoria. En el caso de microprocesadores de 32 bits (4 bytes), es mejor si sólo se accede a posiciones de memoria múltiplos de 4, y el compilador intenta alinear las variables con esas posiciones.

En el caso de variables "int" es fácil, ya que ocupan 4 bytes, pero con las variables "char" no, ya que sólo ocupan 1.

Cuando se accede a datos de menos de 4 bytes la alineación no es tan importante. El rendimiento se ve afectado sobre todo cuando hay que leer datos de cuatro bytes que no estén alineados.

En el caso de la estructura A hemos intercalado campos "int" con "char", de modo que el campo "int" "y", se alinea a la siguiente posición múltiplo

de 4, dejando 3 posiciones libres después del campo "a". Lo mismo pasa con el campo "b".

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
x				a	vacío			y				b	vacío		

En el caso de la estructura B hemos agrupado los campos de tipo "char" al final de la estructura, de modo que se aprovecha mejor el espacio, y sólo se desperdician los dos bytes sobrantes después de "b".

0	1	2	3	4	5	6	7	8	9	10	11
x				y				a	b	vacío	

11.8. Campos de bits

Existe otro tipo de estructuras que consiste en empaquetar los campos de la estructura en el interior de enteros, usando bloques o conjuntos de bits para cada campo.

Por ejemplo, una variable char contiene ocho bits, de modo que dentro de ella podremos almacenar ocho campos de un bit, o cuatro de dos bits, o dos de tres y uno de dos, etc. En una variable int de 16 bits podremos almacenar 16 bits, etc.

Debemos usar siempre valores de enteros sin signo, ya que el signo se almacena en un bit del entero, el de mayor peso, y puede falsear los datos almacenados en la estructura.

Sintaxis:

struct [<nombre de la estructura>] {
unsigned <tipo_entero> <identificador>:<núm_de_bits>;
} [<lista_variables>];

Hay algunas limitaciones, por ejemplo, un campo de bits no puede ocupar dos variables distintas, todos sus bits tienen que estar en el mismo valor entero.

Por ejemplo:

```
struct mapaBits {
  unsigned char bit0:1;
  unsigned char bit1:1;
  unsigned char bit2:1;
  unsigned char bit3:1;
  unsigned char bit4:1;
  unsigned char bit5:1;
  unsigned char bit6:1;
  unsigned char bit7:1;
  };
 struct mapaBits2 {
  unsigned short int campo1:3;
  unsigned short int campo2:4;
  unsigned short int campo3:2;
  unsigned short int campo4:1;
  unsigned short int campo5:6;
};
 struct mapaBits3 {
  unsigned char campo1:5;
  unsigned char campo2:5;
};
```

En el primer caso se divide un valor char sin signo en ocho campos de un bit cada uno:

7	6	5	4	3	2	1	0
bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0

En el segundo caso dividimos un valor entero sin signo de dieciséis bits en cinco campos de distintas longitudes:

15	14	13	12	11	11	9	8	7	6	5	4	3	2	1	0
campo5						campo 4	campo 3		campo2				campo 1		

Los valores del campo5 estarán limitados entre 0 y 63, que son los números que se pueden codificar con seis bits. Del mismo modo, el campo4 sólo puede valer 0 ó 1, etc.

unsigned char								unsigned char							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
			campo2								campo1				

En este ejemplo vemos que como no es posible empaquetar el campo2 dentro del mismo char que el campo1, se añade un segundo valor char, y se dejan sin usar los bits sobrantes.

También es posible combinar campos de bits con campos normales.

Por ejemplo:

```
struct mapaBits2 {
  int numero;
  unsigned short int campo1:3;
  unsigned short int campo2:4;
  unsigned short int campo3:2;
  unsigned short int campo4:1;
  unsigned short int campo5:6;
  float n;
};
```

Los campos de bits se tratan en general igual que cualquier otro de los campos de una estructura. Se les puede asignar valores (dentro del rango que admitan), pueden usarse en condicionales, imprimirse, etc.

```
#include <iostream>
#include <cstdlib>
using namespace std;
struct mapaBits2 {
  unsigned short int campo1:3;
  unsigned short int campo2:4;
  unsigned short int campo3:2;
```

```
    unsigned short int campo4:1;
    unsigned short int campo5:6;
};
int main()
{
    mapaBits2 x;
    x.campo2 = 12;
    x.campo4 = 1;
    cout << x.campo2 << endl;
    cout << x.campo4 << endl;
    cin.get();
    return 0;
}
```

No es normal usar estas estructuras en programas, salvo cuando se relacionan con ciertos dispositivos físicos, por ejemplo, para configurar un puerto serie en MS-DOS se usa una estructura empaquetada en un unsigned char, que indica los bits de datos, de parada, la paridad, etc, es decir, todos los parámetros del puerto. En general, para programas que no requieran estas estructuras, es mejor usar estructuras normales, ya que son mucho más rápidas.

Otro motivo que puede decidirnos por estas estructuras es el ahorro de espacio, ya sea en disco o en memoria. Si conocemos los límites de los campos que queremos almacenar, y podemos empaquetarlos en estructuras de mapas de bits podemos ahorrar mucho espacio.

11.9. Punteros a estructuras

Los punteros también pueden apuntar a estructuras y también pueden existir miembros de la estructura que sean punteros.

En este caso, para referirse a cada elemento de la estructura se usa el operador (->), en lugar del (.).

Suponiendo que el nombre del puntero es punt y el miembro de una estructura es dia y punt esta inicializado a una instancia de esa estructura se podría acceder al miembro:

punt->dia

Veamos un Ejemplo:

Por ejemplo:

```
struct stComplejo {
   float real, imaginario;
} Complejo[10];
stComplejo *p;
p = Complejo; /* Equivale a p = &Complejo[0]; */
p++; /* p  == &Complejo[1] */
```

En este caso, al incrementar p avanzaremos las posiciones de memoria necesarias para apuntar a la siguiente instancia o conjuntos de variables del array de estructura llamado "Complejo" o si . Es decir avanzaremos sizeof(stComplejo) bytes.

Veamos otro Ejemplo:

```
#include <iostream>
using namespace std;
struct stEstructura {
        int a, b;
};   //La Estructura es Global !!
int main() {
stEstrucutra estructura, *e;  //Instancias Local !!
estructura.a = 10; //accedo con el operador punto  .
  estructura.b = 32;
  e = &estructura;
  cout << "variable" << endl;
  cout << e->a << endl; //accedo con el operado  flecha ->
  cout << e->b << endl;
  cout << "puntero" << endl;
  cout << estructura.a << endl;
  cout << estructura.b << endl;
  cin.get();
  return 0;
}
```

11.10. Ejemplo Gestion dinámica de Memoria y Lista Enlazada simple

Existe varias aplicaciones derivadas de incorporar un puntero como miembro de una estructura.. en particular mucho mas cuando el puntero apunta a una estructura del mismo tipo !! esto crea lo que se conocen como listas Enlazadas Simples, Dobles , Circulares.. etc. Veamos como son

/*Crear una lista simple de una cantidad de instancias a determinar en

tiempo de ejecución o en forma dinámica de una estructura del siguiente tipo:

```
struct dato {int tel; int legajo; struct dato *pdato;};*/
//Crear una lista simple de 5 instancias
#include <iostream>
using namespace std;
struct dato {int tel; int legajo; struct dato *pdato;};
int main()
{
int i,cantidad_instancias;
struct dato *punterodato ;
cout<< " Ingrese un nro entero para definir la cantidad de instancias"<< endl;
cin >> cantidad_instancias;
punterodato= new struct dato[cantidad_instancias];
for (i=0; i< cantidad_instancias ; i++)
{
cout << " Telefono: " << endl;
cin >> (punterodato+i)>
tel;
cout << " Legajo : " << endl;
cin >> (punterodato+i)>
legajo;
if(i==(cantidad_instancias1))(
punterodato+i)>
pdato=0;
else (punterodato+i)>
pdato=(punterodato+i+1);
```

```
}
/*for (i=0; i< cantidad_instancias ; i++)
{
cout << punterodato>
tel;
cout << punterodato>
legajo;
punterodato++;
}*/
do
{
cout << punterodato << endl;
cout << punterodato>
tel<< endl;
cout << punterodato>
legajo <<endl;
cout << punterodato>
pdato << endl;
cout << endl << endl;
punterodato++;
}while (((punterodato1)
>
pdato)!=0);
return 0;
}
```

Este tipo de Lista se puede recorrer en un solo sentido desde el principohacia el fin. Vemos otra que no tiene este inconveniente.

11.11. Ejemplo Gestion dinámica de Memoria y Lista Enlazada Doble

Cargar una estructura doblemente enlazada del siguiente tipo:
struct dato {int tel; int legajo; struct dato *pprox; struct dato *pant;};

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

```
struct dato {int tel; int legajo; struct dato *pprox; struct dato *pant;};
int main()
{
int i,cantidad_instancias;
struct dato *punterodato ;
cout<< " Ingrese un nro entero para definir la cantidad de instancias"<< endl;
cin >> cantidad_instancias;
punterodato= new struct dato[cantidad_instancias];
for (i=0; i< cantidad_instancias ; i++)
{
cout << " Telefono: " << endl;
cin >> (punterodato+i)>
tel;
cout << " Legajo : " << endl;
cin >> (punterodato+i)>
legajo;
if(i==0) (punterodato+i)>
pant=0;
else (punterodato+i)>
pant=(punterodato+i1);
if(i==(cantidad_instancias1))(
punterodato+i)>
pprox=0;
else (punterodato+i)>
pprox=(punterodato+i+1);
}
do
{
cout << " Direccion de esta instacia : ";cout << punterodato << endl;
cout << " Telefono : " << punterodato>
tel<< endl;
cout << " Legajo : ";cout << punterodato>
```

```
legajo <<endl;
cout << " Direccion de la insta. anterior: ";cout << punterodato>
pant<< endl;
cout << " Direccion de la insta. proxima: ";cout << punterodato>
pprox<< endl;
cout << endl << endl;
punterodato++;
}while (((punterodato1)
>
pprox)!=0);
return 0;
}
```

Este tipo de Lista se puede recorrer en ambos sentidos.

11.12. Ejercicios

Carga y muestra de Estructura de una sola Instancia

1- Realizar un Programa en C que cargue una Estructura llamada Alumno de una sola instancia de nombre “Electronica” del siguiente tipo: Nombre (char de 10 elementos), edad (entero), fecha de nacimiento(arreglo de 3 enteros). Luego de cargar esta estructura d esde teclado deberá mostrarla.

Carga y muestra de Estructura de 5 instancias

2 - Realizar un Programa en C que cargue una Estructura llamada Alumno de 5 instancias de nombre “Electronica” del siguiente tipo: Nombre (char de 10 elementos), edad (entero), fecha de nacimiento(arreglo de 3 enteros). Luego de cargar esta estructura desde teclado deberá mostrarla.

Carga y muestra de Estructura de 5 instancias usando puntero

3 - Realizar un Programa en C que cargue una Estructura llamada Alumno de 5 instancias de nombre “Electronica” del siguiente tipo: Nombre (char de 10 elementos), edad (entero), fecha de nacimiento(arreglo de 3 enteros). Luego de cargar esta estructura desde teclado deberá mostrarla. La carga y la muestra deberán realizarse usando punteros.

Carga y muestra de Estructura de una sola instancia usando puntero a la estructura y funciones.

4 - Realizar un Programa en C que cargue una Estructura llamada Alumno de una sola instancia de nombre “Electronica” del siguiente tipo: Nombre (char de 10 elementos), edad (entero), fecha de nacimiento(arreglo de 3 enteros). La carga de la Instancia se deberá hacer en una función llamada carga que recibe como argumento un puntero a la Estructura. La muestra

de la Instancia se deberá hacer en una función llamada muestra que recibe como argumento un puntero a la Estructura.

Carga y muestra de Estructura de 5 instancias usando puntero a la estructura y funciones.

5 - idem anterior pero para 5 instancias.

6 - Escribir un programa que almacene en un array los nombres y números de teléfono de 10 personas. El programa debe leer los datos introducidos por el usuario y guardarlos en memoria. Después debe ser capaz de buscar el nombre correspondiente a un número de teléfono y el teléfono correspondiente a una persona. Ambas opciones deben se accesibles a través de un menú, así como la opción de salir del programa. El menú debe tener esta forma, más o menos:

a) Buscar por nombre

b) Buscar por número de teléfono

c) Salir

Pulsa una opción:

Nota: No olvides que para comparar cadenas se debe usar una función, no el operador ==.

7- Para almacenar fechas podemos crear una estructura con tres campos: ano, mes y día. Los días pueden tomar valores entre 1 y 31, los meses de 1 a 12 y los años, dependiendo de la aplicación, pueden requerir distintos rangos de valores. Para este ejemplo consideraremos suficientes 128 años, entre 1960 y 2087. En ese caso el año se obtiene sumando 1960 al valor de año. El año 2003 se almacena como 43.

Usando estructuras, y ajustando los tipos de los campos, necesitamos un char para día, un char para mes y otro para año.

Diseñar una estructura análoga, llamada "fecha", pero usando campos de bits. Usar sólo un entero corto sin signo (unsigned short), es decir, un entero de 16 bits. Los nombres de los campos serán: dia, mes y anno.

8 - Basándose en la estructura de bits del ejercicio anterior, escribir una función para mostrar fechas: `void Mostrar(fecha);`. El formato debe ser: "dd de mmmmmm de aaaa", donde dd es el día, mmmmmm el mes con letras, y aaaa el año. Usar un array para almacenar los nombres de los meses.

9 - Basándose en la estructura de bits del ejercicio anterior, escribir una función `bool ValidarFecha(fecha);`, que verifique si la fecha entregada como parámetro es válida. El mes tiene que estar en el rango de 1 a 12,

dependiendo del mes y del año, el día debe estar entre 1 y 28, 29, 30 ó 31. El año siempre será válido, ya que debe estar en el rango de 0 a 127. Para validar los días usaremos un array `int DiasMes[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};`. Para el caso de que el mes sea febrero, crearemos otra función para calcular si un año es o no bisiesto: `bool Bisiesto(int);` Los años bisiestos son los divisibles entre 4, al menos en el rango de 1960 a 2087 se cumple.

Nota: los años bisiestos son cada cuatro años, pero no cada 100, aunque sí cada 400. Por ejemplo, el año 2000, es múltiplo de 4, por lo tanto debería haber sido bisiesto, pero también es múltiplo de 100, por lo tanto no debería serlo; aunque, como también es múltiplo de 400, finalmente lo fue.

10 - Seguimos con el tema de las fechas. Ahora escribir dos funciones más. La primera debe responder a este prototipo: `int CompararFechas(fecha, fecha);`. Debe comparar las dos fechas suministradas y devolver 1 si la primera es mayor, -1 si la segunda es mayor y 0 si son iguales.
La otra función responderá a este prototipo: `int Diferencia(fecha, fecha);`, y debe devolver la diferencia en días entre las dos fechas suministradas.

11 - Supongamos la declaración de la siguiente estructura:

```
struct ejemplo1 {
     unsigned char c1:7;
     unsigned char c2:6;
     unsigned char c3:3;
     unsigned char c4:4;
};:
```

a) ¿Cuántos bytes ocupa esta estructura?

☐ 1

☐ 2

☐ 3

☐ 4

b) ¿Y si en lugar de un "unsigned char" usamos un "unsigned short" de 16 bits?

☐ 1

☐ 2

☐ 3

☐ 4

c) ¿Y si en lugar de un "unsigned short" usamos un "unsigned int" de 32 bits?

☐ 1

☐ 2

☐ 3

☐ 4

12) Tenemos la siguiente estructura:

```
struct A {
     struct B {
     int x,y;
     float r;
     } campoB;
     float s;
     struct {
     int z;
     float t;
     } campoC;
} E;
```

Si tenemos la variable "E", indicar la forma correcta de acceder a las siguientes variables:

a) `x`

A.B.x

E.campoB.x

E.x

E.b.x

b) `s`

A.s

E.s

E.a.s

c) `t`

A.t

E.t

A.campoC.t

E.campoC.t

13) Considerando la siguiente declaración:

```
struct A {
   struct {
      int x;
      int y;
   } campoB;
} *estructuraA;
```

a) ¿Cómo se referenciaría el campo x de la estructura A?

○Estructura A.x

○estructura A.campoB.x

○estructura A.campoB->x

○estructura A->campoB.x

CAPITULO 12. Mas operadores

Veremos ahora más detalladamente algunos operadores que ya hemos mencionado, y algunos nuevos.

12.1. Operadores de Referencia (&) e Indirección (*)

El operador de referencia (&) nos devuelve la dirección de memoria del operando.

Sintaxis:

&<expresión simple>

El operador de indirección (*) considera a su operando como una dirección y devuelve su contenido.

Sintaxis:

**<puntero>*

Por ejemplo:

```
int A, *pA;
pA=&A;
```

Para este caso pA es un puntero, pero en el caso de que tengamos:

```
int a, b;
b=&a;
```

Vemos que “b” NO ES UN PUNTERO, en este caso se dice que b es un “alias” de o variable referenciada, es decir es la misma variable con dos nombre distinto.-

12.2. Operadores . y ->

El operador de selección (.) permite acceder a variables o campos dentro de una estructura.

Sintaxis:

<variable_estructura>.<nombre_de_variable>

El operador de selección de variables o campos se utiliza para estructuras referenciadas con punteros. (->)

Sintaxis:

<puntero_a_estructura>-><nombre_de_variable>

12.3. Operador de preprocesador

El operador "#" sirve para dar órdenes o directivas al compilador.

Por el momento este tema no forma parte del Programa de la Materia.

12.4. Operadores de manejo de memoria "new" y "delete"

Veremos su uso en el capítulo de punteros II y en mayor profundidad en el capítulo de clases y en operadores sobrecargados.

12.4.1. Operador new

El operador new sirve para reservar memoria dinámica.

Sintaxis:

[::]new [<emplazamiento>] <tipo> [(<inicialización>)]

[::]new [<emplazamiento>] (<tipo>) [(<inicialización>)]

*[::]new [<emplazamiento>] <tipo>**[**<número_elementos>**]***

*[::]new [<emplazamiento>] (<tipo>)**[**<número_elementos>**]***

El operador opcional :: está relacionado con la sobrecarga de operadores, de momento no lo usaremos. Lo mismo se aplica a emplazamiento.

La inicialización, si aparece, se usará para asignar valores iniciales a la memoria reservada con new, pero no puede ser usada con arrays.

Las formas tercera y cuarta se usan para reservar memoria para arrays dinámicos. La memoria reservada con new será válida hasta que se libere con delete o hasta el fin del programa, aunque es aconsejable liberar **siempre** la memoria reservada con new usando delete. Se considera una práctica muy *sospechosa* no hacerlo.

Si la reserva de memoria no tuvo éxito, new devuelve un puntero nulo, NULL.

12.4.1.1 Ejemplo Gestion dinámica de Memoria y Lista Enlazada Doble

Cargar una estructura doblemente enlazada del siguiente tipo:

struct dato {int tel; int legajo; struct dato *pprox; struct dato *pant;}

recorrerla por teclado de manera que con P o p se pase a la proxima instancia y con A o a se pase a la

instancia anterior , con f o F se sale.

```
#include <iostream>
using namespace std;
void menu();
void recorre ( struct dato *p);
struct dato {int tel; int legajo; struct dato *pprox; struct dato *pant;};
int main()
{
int i,cantidad_instancias;
struct dato *punterodato ;
cout<< " Ingrese un nro entero para definir la cantidad de instancias"<< endl;
cin >> cantidad_instancias;
```

```
// Solicito al sistema el espacio en memoria
punterodato= new struct dato[cantidad_instancias];
for (i=0; i< cantidad_instancias ; i++)
{
cout << " Telefono: " << endl;
cin >> (punterodato+i)>
tel;
cout << " Legajo : " << endl;
cin >> (punterodato+i)>
legajo;
if(i==0) (punterodato+i)>
pant=0;
else (punterodato+i)>
pant=(punterodato+i1);
if(i==(cantidad_instancias1))(
punterodato+i)>
pprox=0;
else (punterodato+i)>
pprox=(punterodato+i+1);
}
menu();
recorre (punterodato);
return 0;
}
void menu()
{
cout << " Para recorrer las instacias presione "<< endl;
cout << " presione P >
proxima A>
anterior F >
Finaliza";
}
void recorre(struct dato *p)
{ char opc;
```

```
do{
cin>> opc;
switch (opc)
{
case 'a':
case 'A':p=p>
pant;
cout << " Direccion de esta instacia : ";cout << p << endl;
cout << " Telefono : " << p>
tel<< endl;
cout << " Legajo : ";cout << p>
legajo <<endl;
cout << " Direccion de la insta. anterior: ";cout << p>
pant<< endl;
cout << " Direccion de la insta. proxima: ";cout << p>
pprox<< endl;
cout << endl << endl;
break;
case 'p':
case 'P':p=p>
pprox;
cout << " Direccion de esta instacia : ";cout << p << endl;
cout << " Telefono : " << p>
tel<< endl;
cout << " Legajo : ";cout << p>
legajo <<endl;
cout << " Direccion de la insta. anterior: ";cout << p>
pant<< endl;
cout << " Direccion de la insta. proxima: ";cout << p>
pprox<< endl;
cout << endl << endl;
break;
case 'f':
case 'F':cout << "Chau!";break;
```

```
defalut: cout << " Opcion no valida";break;
}
}while ( opc!='f' || opc!= 'F');
}
```

12.4.2-Operador delete

El operador delete se usa para liberar la memoria dinámica reservada con new.

Sintaxis:

[::]delete [<expresión>]

[::]delete [<expresión>]

La expresión será normalmente un puntero, el operador delete[] se usa para liberar memoria de arrays dinámicas.

Es importante liberar siempre usando delete la memoria reservada con new. Existe el peligro de pérdida de memoria si se ignora esta regla.

Cuando se usa el operador delete con un puntero nulo, no se realiza ninguna acción. Esto permite usar el operador delete con punteros sin necesidad de preguntar si es nulo antes.

INFORMACIÓN A TENER EN CUENTA:

Los operadores new y delete son propios de C++. En C se usan las funciones malloc y free para reservar y liberar memoria dinámica y liberar un puntero nulo con free suele tener consecuencias desastrosas.

Por ejemplo:

```
int main() {
  char *c;
  int *i = NULL;
  float **f;
  int n;
```

```
    c = new char[123];          // Cadena de 122 más el nulo:
   f = new float *[10]; (1)          // Array de 10 punteros a float:
                        // Cada elemento del array es un array de 10
float
   for(n = 0; n < 10; n++) f[n] = new float[10]; (2)
                        // f es un array de 10*10
   f[0][0] = 10.32;
   f[9][9] = 21.39;
   c[0] = 'a';
   c[1] = 0;
                        // liberar memoria dinámica
   for(n = 0; n < 10; n++) delete[] f[n];
   delete[] f;
   delete[] c;
   delete i;
   return 0;
}
```

INFORMACIÓN A TENER EN CUENTA:

f es un puntero que apunta a un puntero que a su vez apunta a un float. Un puntero puede apuntar a cualquier tipo de variable, incluidos los propios punteros.

Este ejemplo nos permite crear arrays dinámicos de dos dimensiones. La línea (1) crea un array de 10 punteros a float. La (2) crea 10 arrays de floats. El comportamiento final de f es el mismo que si lo hubiéramos declarado como:

```
float f[10][10];
```

CAPITULO 13. Pasaje de valores a funciones

13.1. Parámetros por valor y parámetros por referencia

Dediquemos algo más de tiempo a las funciones.

Hasta ahora siempre hemos declarado los parámetros de nuestras funciones del mismo modo. Sin embargo, éste no es el único modo que existe para pasar parámetros. La forma en que hemos declarado y pasado los parámetros de las funciones hasta ahora es la que normalmente se conoce como "por valor". Esto quiere decir que cuando el control pasa a la función, los valores de los parámetros en la llamada se copian a "variables" locales de la función, estas "variables" son de hecho los propios parámetros, pero ¿como ahgo si realmente queremos que los valores pasados se modifiquen?, la respuesta a esto es pasar los valores por referencia.

Por ejemplo:

```
#include <iostream>
using namespace std;
int extremo_intervalo(int menor, int mayor);
int main() {
        int a, b;                       /*a y b locales de main */
        cout << endl << "Ingrese el extremo de un intervalo: ";
        cin >> a;
        cout << endl << "Ingrese el extremo de un intervalo: ";
        cin >> b;
        cout << "a,b valen ->" << a << ", " << b << endl;
        cout << "llamo a la funcion control pasando a y b y retorna: ";
```

```
        if (extremo_intervalo(a, b)) cout << "Intercambiados !!";
        else cout << "NO Intercambiados !!";
        cout << endl;                    /* llamo a la función */
        cout << "a,b siguen valiendo ->" << a << ", " << b << endl;
        return 0;
}
```

/* Esta función controla que el extermo inferior del intervalo sea menor o igual que el extremo mayor */

```
int extremo_intervalo(int menor, int mayor)
{
        int tmp;
        if( menor <= mayor)
                return 0;           /* retornado 0 no intercambia */
        else
                {tmp=menor;
                menor=mayor;    /*intercambio los valores*/
                mayor=tmp;}
        return 1;                   /* retorna 1 si intercambia */
}
```

Veamos como pasar valores a funciones pero que sí puedan modificarlos.

13.1.1. Referencias a variables

Las referencias sirven para definir "alias" o nombres alternativos para una misma variable.

Para ello se usa el operador de referencia (&).

Por ejemplo:

```
&a
```

Aquí estaríamos haciendo referencia a la direción de memoria de la variable a , esta es la misma dentro y fuera de cualquier función, aunque el nombre utilizado sea cualquiera.

Sintaxis:

<tipo> &<alias> = <variable de referencia>

<tipo> &<alias>

La primera forma es la que se usa para declarar variables de referencia, la asignación es obligatoria ya que no pueden definirse referencias indeterminadas.La segunda forma es la que se usa para definir parámetros por referencia en funciones, en las que las asignaciones son implícitas.

Por ejemplo :

```
#include <iostream>
using namespace std;
int main() {
  int a;                  /* defino una variable */
  int &r = a;             /* aca vemos que la dirección de la variable r
                             es igual a la de la varaible  a => son lo
mismo!!                          pero con otro nombre */
  a = 10;                 /* le doy un valor a a */
  cout << " a vale ="<< a << endl;
  cout << " el valor de de r es ="<< r << endl;   /*muestro el valor
de r */
  cin.get();
  return 0;
}
```

En este ejemplo las variables a y r se refieren al mismo objeto, cualquier cambio en una de ellas se produce en ambas. Para todos los efectos, son la misma variable.

13.2. Paso de parámetros por referencia

Si queremos que los cambios realizados en los parámetros dentro de la función se conserven al retornar de la llamada, deberemos pasarlos por referencia. Esto se hace declarando los parámetros de la función como referencias a variables.

Por ejemplo:

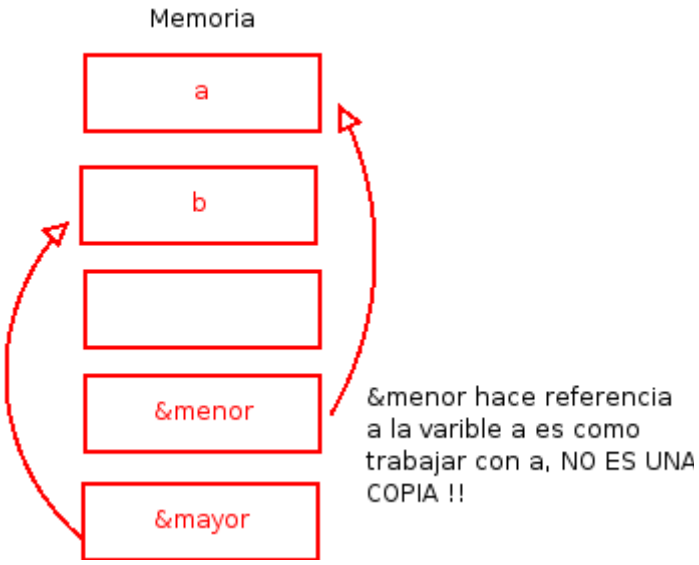
```
#include <iostream>
using namespace std;
int extremo_intervalo(int & menor, int & mayor);      /*      ver      que existe & */
int main() {
        int a, b;                                   /*a y b locales de main */
        cout << endl << "Ingreso el extremo de un intervalo: ";
        cin >> a;
        cout << endl << "Ingreso el otro extremo de un intervalo: ";
        cin >> b;
        cout << "a,b valen ->" << a << ", " << b << endl;
        cout << "llamo a la funcion extremo_intervalo pasando a, b y retorna: ";
        if (extremo_intervalo(a,b)) cout << "Intercambiados !!";
        else cout << "NO Intercambiados !!";
        cout << endl;                               /* llamo a la función */
        cout << "a,b valen ->" << a << ", " << b << endl;
        return 0;
}
```

/* esta función controla que el extermo inferior del intervalo sea menor o igual que el extremo mayor */

```
int extremo_intervalo(int &menor, int &mayor)      /*      recibe      la dirección                                de las variables */


{
        int tmp;
        if( menor <= mayor) return 0;               /*  retornado  0  no intercambia */
        else {
                tmp=menor;
```

```
            menor=mayor;            /*intercambio los valores*/
            mayor=tmp;}
        return 1;                   /* retorna 1 si intercambia */
}
```



En este caso, las variables "a" y "b" tendrán valores distintos después de llamar a la función. Cualquier cambio que realicemos en los parámetros dentro de la función, se hará también en las variables referenciadas.

INFORMACIÓN A TENER EN CUENTA:

No podemos hacer llamadas a funciones con referencia a parámetros constantes.

Por ejemplo:

```
int extremo_intervalo(int & menor, int & mayor);      /* prototipo */
extremo_intervalo(10 ,20 );                  /* ERROR !!! MAL !!! */
```

13.3. Punteros como parámetros de funciones

Cuando pasamos un puntero como parámetro por valor de una función pasa lo mismo que con las variables.

Dentro de la función trabajamos con una copia del puntero. Sin embargo, el objeto apuntado por el puntero sí será el mismo, los cambios que hagamos en los objetos apuntados por el puntero se conservarán al abandonar la función, pero no será así con los cambios que hagamos al propio puntero.

Por ejemplo:

```
#include <iostream>
using namespace std;
void funcion(int *q);
int main() {
  int a;
  int *p;
  a = 100;
  p = &a;
                // Llamamos a funcion con un puntero funcion(p);
  cout << "Variable a: " << a << endl;
  cout << "Variable *p: " << *p << endl;
                // Llamada a funcion con la dirección de "a" (constante)
  funcion(&a);                  /*Sería Equivalente poner funcion(p) */
  cout << "Variable a: " << a << endl;
  cout << "Variable *p: " << *p << endl;
  cin.get();
  return 0;
}
void funcion(int *q) {
      // Cambiamos el valor de la variable apuntada por el puntero
  *q += 50;
  q++;      /* vemos que cambio el valor de q, pero no tiene efecto
*/
}
```

Dentro de la función se modifica el valor apuntado por el puntero, y los cambios permanecen al abandonar la función.

Sin embargo, los cambios en el propio puntero son locales, y no se conservan al regresar. Con este tipo de parámetro para función pasamos el puntero por valor.

¿Y cómo haríamos para pasar un puntero por referencia?:

```
void funcion(int* &q);  /*notar que no está * si no & !!*/
```

El operador de referencia siempre se pone junto al nombre de la variable .

13.4. Funciones que devuelven referencias

Como mencionamos, para devolver referencias desde una función, basta con declarar el valor de retorno como una referencia.

Sintaxis:

```
<tipo> &<identificador_función>(<lista_parámetros>);   /* notar que no está * si no & !!  */
```

Esto nos permite trabajar con los elementos que nos pasan no con una copia.

Por ejemplo:

```
//La función mayor recibe un puntero a un array y dos indices para
// apuntar a una par de elementos del arreglo y retorna el mayor valor
// mas uno.

#include <iostream>
using namespace std;
int & mayor(int * , int , int);   /* Prototipo de función */
int main()
{
```

```
  int array [10] = {1, -3, 2, -1, 4, 5, 6, -7, 8, 9},a,b;
  cout << "Ingrese un indice del vector enter 0 y 9" << endl;
  cin >>a;
  cout << "Ingrese un otro indice del vectot enter 0 y 9" << endl;
  cin >>b;
  cout << "El mayor valor es : "<< mayor( array, a,b ) << endl;
  cin.get();
  return 0;
}

int & mayor(int *vector , int y, int x)
{
return (++((vector[x]>=vector[y])?vector[x]:vector[y]));
}
```

Esta es una potente herramienta de la que disponemos, aunque ahora no se nos ocurra ninguna aplicación interesante.

13.5. Arrays como parámetros de funciones

Cuando pasamos un array como parámetro en realidad estamos pasando un puntero al primer elemento del array, así que las modificaciones que hagamos en los elementos del array dentro de la función serán validos al retornar.

Por ejemplo:

```
#include <iostream>
using namespace std;
#define TamanioArray  10
#include <iostream>
using namespace std;
int BuscaMenor(int x[]);                    //prototipo de la función
int main()
```

```
{
int arreglo[10];
for (int i=0; i < 10; i++)
{
cout << endl << "Ingrese un numero : " << endl;
cin >> arreglo[i];
cout << "El menor valor es: " << BuscaMenor(arreglo)<< endl;
}
return 0;
}

int BuscaMenor(int x[])                    //defínción de la función
{
int menor=x[0];          // asigno como el menor al 1er elemento del array
for (int i=0; i < 10; i++)
        if(x[i]< menor) menor=x[i];
return menor;
}
```

En este caso el array es de una dimensión, sin embargo, si sólo pasamos el nombre del array de más de una dimensión no podremos acceder a los elementos del array mediante subíndices, ya que la función no tendrá información sobre el tamaño de cada dimensión.

Para tener acceso a arrays de más de una dimensión dentro de la función se debe declarar el parámetro como un array.

```
#include <iostream>
using namespace std;
#define N 3
#define M 3
void funcion(int [][M]);
int main() {
   int Tabla[N][M];              // arreglo local Tabla;
   for (int i=0; i < N; i++)for (int j=0; j<M ; j++)
   {cout << "Ingrese un valor :  " ;
```

```
        cin >> Tabla[i][j];
        }
  funcion(Tabla);                  //llamo a la funcion
    for (int i=0; i < N; i++)for (int j=0; j<M ; j++)cout << Tabla[i][j]
<< " ";
        return 0;
}

 //Esta funcion solo cambia el valor de un elemento del arreglo
 // pasado como referencia

void funcion(int arreglolocal[][M])
{
        cout << "Ingrese un valor para cambiar el valor de
arreglolocal[1][1]" << endl;
        cin >> arreglolocal [1][1] ;
}
```

INFORMACIÓN A TENER EN CUENTA:

Las estructuras también pueden ser pasadas por valor y por referencia.

Las reglas se les aplican igual que a los tipos fundamentales: las estructuras pasadas por valor no conservarán sus cambios al retornar de la función.

Las estructuras pasadas por referencia conservarán los cambios que se les hagan al retornar de la función. Por el momento no profundizaremos en esto.-

13.6 Punteros a Funciones:

Los punteros a función son uno de los recursos más potentes y flexibles de C/C++, permitiendo técnicas de programación muy eficientes. Por ejemplo, escribir funciones que manejan diferentes tipos de datos; diseñar algoritmos muy compactos ("function dispatchers"), que pueden sustituir

largas cadenas if...else o switch , o alterar el flujo de ejecución del programa, modificando el orden de llamadas a funciones en base a determinadas prioridades ("adaptive program flow"). Así mismo, resultan de gran ayuda en programas de simulación y modelado. Sin embargo, los autores están generalmente de acuerdo en que constituyen para el principiante uno de los puntos más confusos del lenguaje.

Para hablar de punteros a funciones, previamente hay que establecer que las funciones tengan "dirección". Del mismo modo que en una matriz se asume que su dirección es la del primer elemento, se asume también que la dirección de una función es la del segmento de código ("Code segment") donde comienza el código de dicha función . Es decir, la dirección de memoria a que se transfiere el control cuando se la invoca (su punto de comienzo).

Una vez establecido esto, no tiene que extrañar que puedan definirse variables de un tipo especial para apuntar a estas direcciones. Técnicamente un **puntero-a-función es una variable que guarda la dirección de comienzo de la función.** Pero como tendremos ocasión de comprobar, la mejor manera de pensar en ellos es considerarlos como una especie de "alias" de la función, aunque con una importante cualidad añadida: que pueden ser utilizados como argumentos de otras funciones.

```
#include <iostream>
using namespace std;
int suma(int,int);
int producto(int,int);
int main()
{int (*pf[2])(int,int);  // Array de punteros a función con arg. int
int e1,e2;
int opc;
pf[0] = suma; //Inicializo el 1er elemento del arreglo a suma
pf[1] = producto; //Inicializo el 2do elemento del arreglo a producto
cout<<"Por favor Ingrese un entero : ";
cin>>e1;
cout<<"Por favor Ingrese otro entero : ";
cin>>e2;
cout<<"Ingrese 0 para sumar o 1 para multiplicar : "<<endl;
cin>>opc;
```

```
if((opc==0)||(opc==1)) cout<<pf[opc](e1,e2);
else cout<<"opcion Incorrecta";
return 0;
}
int suma(int x, int y){return (x+y);}
int producto(int x, int y){return (x*y);}
```

En este caso vemos que existe un arreglo de punteros a funciones que retornan enteros y que reciben dos enteros como argumentos. int (*pf[2]) (int,int).

Luego se inicializa cada elemento del arreglo de punteros a una función:

```
pf[0] = suma; //Inicializo el 1er elemento del arreglo a suma
pf[1] = producto; //Inicializo el 2do elemento del arreglo a producto
```

Ahora se puede usar una llamada como si fuera un argumento.

```
if((opc==0)||(opc==1)) cout<<pf[opc](e1,e2);
```

Que llamará a suma (pf[0]) o producto (pf[1]) según corresponda.

13.7 Sobrecarga de Funciones:

La sobrecarga de funciones es un mecanismo de C++ que permite asignar el mismo nombre a funciones distintas. Para el compilador estas funciones no tienen nada en común a excepción del identificador, por lo que se trata en realidad de un recurso semántico del lenguaje que solo tiene sentido cuando se asigna el mismo nombre a funciones que realizan tareas similares en objetos diferentes.

Cuando se realiza la invocación de una función sobrecargada, es decir que existen otras del mismo nombre en el mismo ámbito , el compilador decide cual de ellas se utilizará mediante un proceso denominado resolución de sobrecarga ("Overload resolution") aplicando ciertas reglas para verificar cual de las declaraciones se ajusta mejor al número y tipo de los argumentos utilizados. Es decir, donde existe máxima concordancia entre los argumentos actuales y formales . El proceso sigue unas reglas denominadas de congruencia estándar de argumentos; son las siguientes (en el orden precedencia señalado):

1. Concordancia exacta en número y tipo.

2. Concordancia después de realizar promociones de los tipos asimilables a enteros , por ejemplo: char; short; bool; enum (y sus correspondientes versiones unsigned) a int. También de tipos float a double [3].

3. Concordancia después de realizar conversiones estándar. Por ejemplo: int a double; double a long double;

4. Concordancia después de realizar conversiones definidas por el usuario

Veamos un ejemplo:

```
#include <iostream>
using namespace std;
int  producto (int, int);
int  producto (int, int,int);
float  producto (int, float);

int main()
{int e1, e2, e3;
float f1;
cout<<"Por favor Ingrese un entero : ";
cin>>e1;
cout<<"Por favor Ingrese otro entero : ";
cin>>e2;
cout<<"Por favor Ingrese otro entero : ";
cin>>e3;
cout<<"Por favor Ingrese un flotante: ";
cin>>f1;
cout<<endl<<producto(e1,e2)<<endl;
cout<<endl<<producto(e1,e2,e3)<<endl;
cout<<endl<<producto(e1,f1)<<endl;

}
int producto (int x, int y){return (x*y);}
int producto(int x, int y,int z){return (x*y*z);}
```

```
float producto(int x,float z){return (x*z);}
```

Podemos ver que en este caso hay tres funciones de nombre producto, pero son distintas en cuanto la cantidad de argumentos o el tipo de argumentos.