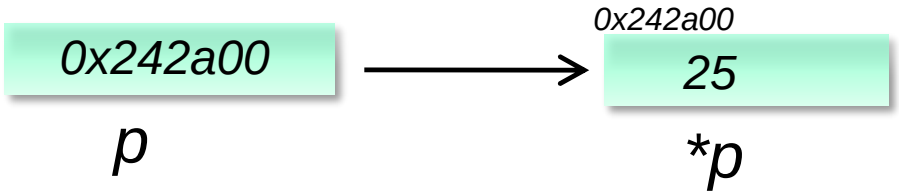
Programación Orientada a Objetos con C++

Introducción a los punteros

Punteros en C++

Qué es un **puntero**?

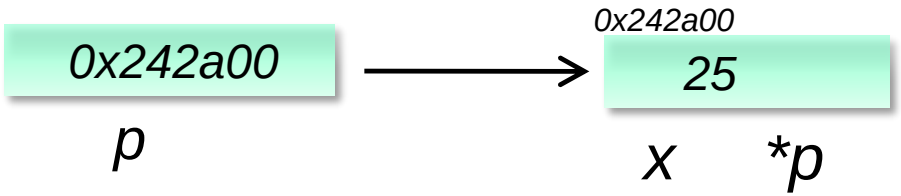
- Un puntero es una variable que contiene la ubicación física (dirección de memoria) de un elemento determinado del programa.



Punteros en C++

Cuándo se asigna una dirección de memoria al puntero?

int x=25; // variable x de tipo int
int *p; //declaración de p como puntero a un int
p= &x; // p toma la dirección de x (por ejemplo 0x242a00)



Punteros en C++

- El elemento apuntado puede constituir en C++:
  - Un dato simple
  - Una estructura de datos
  - Una función
  - Una variable de cualquier tipo
- Muchas funciones predefinidas de C++ emplean punteros como argumentos e inclusive devuelven punteros.
- Para operar con punteros C++ dispone de los operadores `&` y `*` .

Declaración de variables puntero

tipo *nombre_puntero;

Ejemplos:

```
int *p1;

float valor, *pvalor;

double *p, *q;
```

El operador de dirección o referencia &

- Todas las variables se almacenan en una posición de memoria, la cual puede obtenerse con el operador *ampersand* (**&**), que significa literalmente ***"la dirección de"***. Por ejemplo: &y;

- Hemos usado en funciones este operador para "referenciar" una dirección de memoria y definir alias de parámetros en el pasaje por referencia.

```
void funcion_nueva(int a, int b, float &r);
```

Operador de indirección *

- Usando un puntero se puede acceder directamente al valor almacenado en la variable apuntada utilizando el operador de indirección o desreferencia *asterisco* (*), que puede ser traducido literalmente como ***"valor apuntado por"***. Así, siguiendo con los valores del ejemplo previo, si se escribe:

```
int *p ;
int y=25;
p = &y;
cout<<*p<<“ se ubica en la dirección:”<<p<<endl;
```

Salida: 25 se ubica en la dirección: 0x242a00

Uso de punteros en programas

Analicemos en un esquema gráfico las variables del programa, ejecutando paso a paso las acciones propuestas en el código;:

```
int main () {
int valor1 = 5, valor2 = 15;
int *p1, *p2;
p1 = &valor1; // p1 toma dirección de valor1
p2 = &valor2; // p2 toma dirección de valor2
*p1 = 10; // valor apuntado por p1 = 10
*p2 = *p1; //valor apuntado por p2=valor apuntado por p1
p1 = p2; // p1 = p2 (asignación de punteros)
*p1 = 20; // valor apuntado por p1 = 20
cout << "valor1=" << valor1 << "/ valor2=" << valor2;
return 0;
}
```

Valor1=10 / valor2=20

Dónde se almacenan las variables de un programa?

En la memoria RAM de una computadora y durante la ejecución de un programa, existen porciones de memoria muy importantes para almacenar los elementos que utiliza el programa.

***Stack (pila):** una porción de memoria limitada donde se alojan todas las variables de cada bloque, parámetros de funciones, durante la compilación. Una variable de un bloque (alcance {..}) no puede ser accedida desde otro bloque.*

***Heap (montículo):** es una porción de memoria más extensa (será mayor cuanto más RAM tenga la computadora), donde se alojan las variables dinámicas creadas dentro del programa durante su ejecución. Estas variales se pueden acceder desde cualquier lugar del programa.*

Donde se almacenan las variables de un programa?
El Stack (la pila)

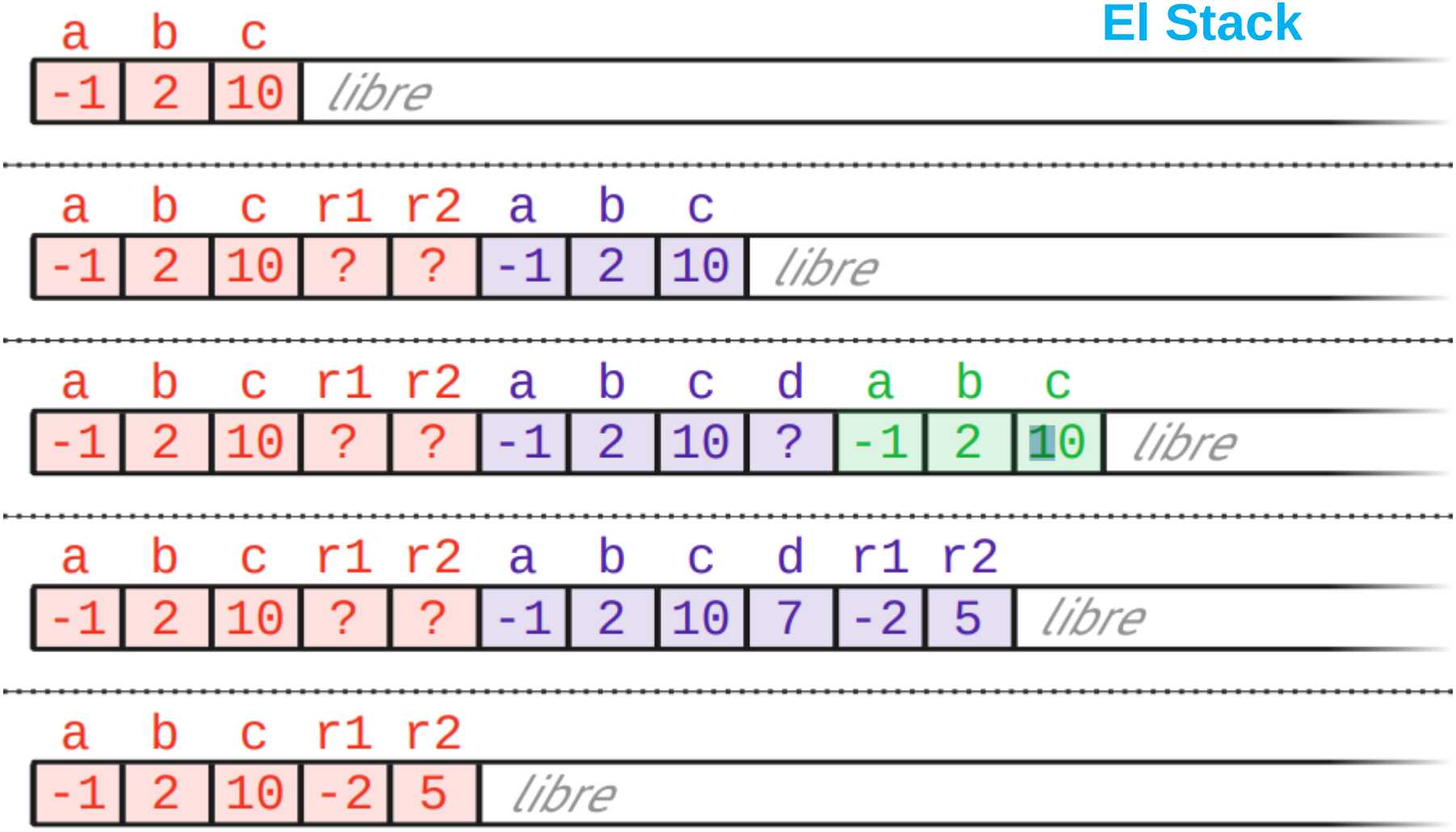
Consideremos el caso de una ecuación cuadrática $ax^2+bx+c=0$, y el código C++ para calcular las raíces r1 y r2.

```
int main () {
 float a,b,c,r1,r2;
 cin>>a>>b>>c;
 tie(r1,r2)=raíces(a,b,c);
 cout<<r1<<" "<<r2<<endl;
 return 0; }

tuple<float,float> raices(float a,float b, float c)
 {float r1=(-b+sqrt(discrim))/(2*a);
  float r2=(-b-sqrt(discrim))/(2*a);
  return make_tuple(r1,r2);}

float discrim(a,b,c)
{ return b*b-4*a*c;}
```

Dónde se almacenan las variables de un programa?



Assignación de memoria en el Stack

Cuando un programa es ejecutado, se le asigna una cantidad limitada de espacio en el **Stack**. Se trata de un fragmento de memoria donde se van **apilando** (stacking) linealmente las distintas funciones ejecutadas así como las variables locales de cada una de ellas durante **LA COMPILACION.**

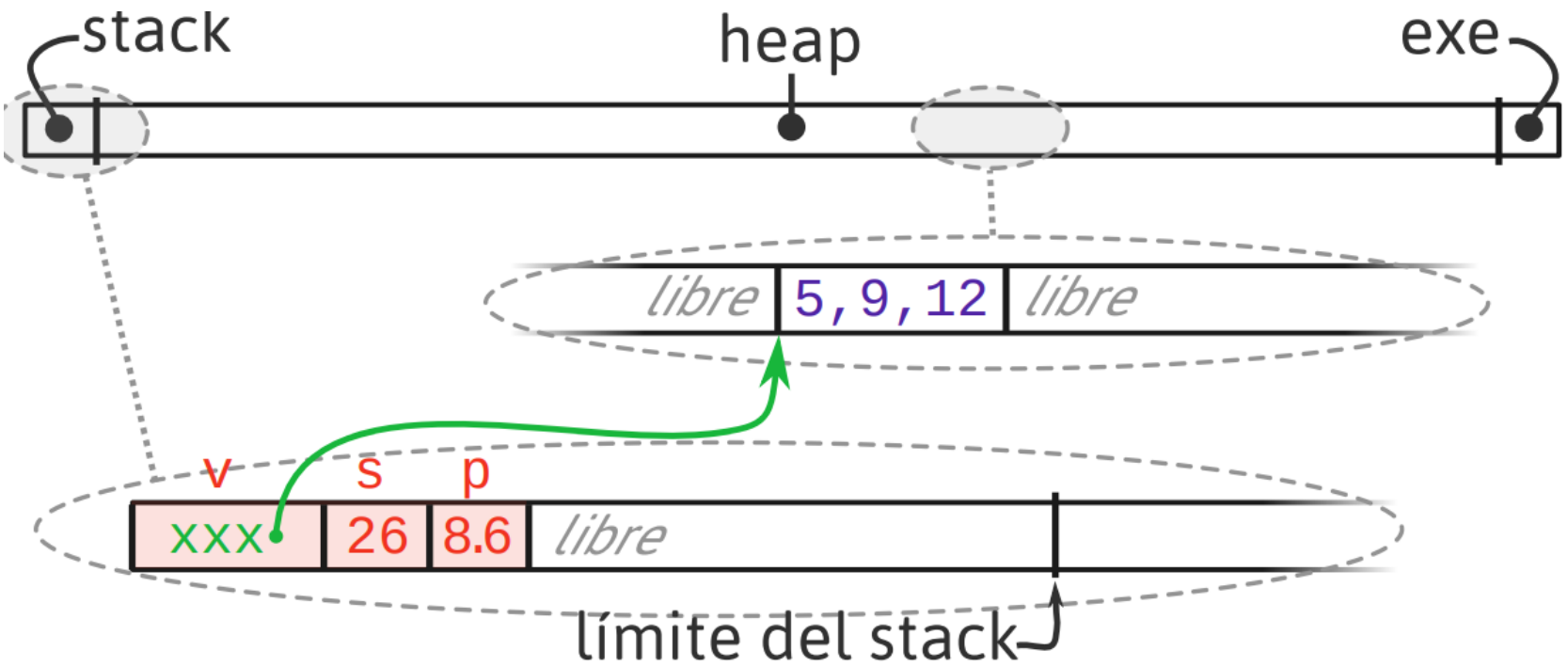
Algunas consideraciones sobre el **Stack**:

- Las variables almacenadas en el Stack (o variables automáticas) son almacenadas directamente a esta memoria.
- Su acceso es muy rápido.
- Es liberada al terminar la ejecución del bloque (scope en C++).
- Fragmentos grandes de memoria, como arreglos de gran envergadura, no deberían ser almacenados en el **Stack**, para prevenir desbordamientos del mismo (stack overflow).
- Las variables almacenadas en el **Stack** solamente son accesibles desde el bloque de código donde fueron declaradas.

Donde se almacenan las variables de un programa?
El Heap (el montículo)

Observemos un programa C++ que emplea un vector

```
int main () {
 vector <float>  v={5,9,12};
 float sum= sum_vector(v);
 float prom=sum/v.size();
 v.push_back(prom);}
```



Assignación de memoria en el HEAP

El **Heap** a diferencia del STACK, no posee ninguna estructura de asignación de espacios, y su tamaño se ve únicamente limitado por el tamaño de la memoria RAM. Es asignada durante la EJECUCIÓN del programa, y los bloques otorgados pueden no ser contiguos.

- La manipulación del **Heap** (asignación, lectura, escritura) es más lenta que la del **Stack** .
- Esta memoria se mantiene en uso hasta que se libera explícitamente por alguna acción del programa (o es liberada por el SO al terminar la ejecución del programa completo)
- Puede ser accedida desde fuera del bloque donde fue asignada.

Operaciones con punteros

a) Se puede asignar a una variable puntero la dirección de una variable no puntero.

```
float x, *p;
.....
p=&x;
```

b) A una variable puntero puede asignarse el contenido de otra variable puntero si son compatibles (ambos punteros apuntan al mismo tipo de dato).

```
int *u, *v;
.....
u=v;
```

c) A un puntero es posible asignarle el valor NULL (el puntero no apunta a dirección de memoria alguna).

```
int *p;
p=nullptr;   //Dirección nula: 0x000 en hexadecimal
```

Operaciones con punteros

d) Es posible sumar o restar una cantidad entera **n** a una variable puntero. La nueva dirección de memoria obtenida difiere de la inicial en una cantidad de bytes dada por: **n** * sizeof(tipo apuntado por el puntero).

```
int *p;
.....
p+=4; //la dir original de p se incrementó 16 bytes
p-=1; //La dir anterior de p se decrementó en 4 bytes
```

e) Es posible comparar dos variables puntero si estas son compatibles (apuntan a datos de igual tipo)

u==v **u!=v** **u==nullptr** **u<v** **u>=v**

f) Es posible operar los datos apuntados a través de la notación de punteros:

***p<*q** ***(p++)** **(*q)--** ***r=23.5**

Arreglos estáticos

```
int v[10]={25,33,17,49,51,62,11,31,29,75};
```

25	33	17	49	51	62	11	31	29	75

```
int x[10];
X[0]=19; x[1]=23; x[2]=91;
```

19	23	91							

```
int m[2][7];
```

10	24	14	89	99	20	17
93	54	65	77	87	92	98

Punteros y arreglos estáticos

El concepto de arreglo está estrechamente unido al de puntero.
De hecho, el identificador de un arreglo representa a la dirección de memoria de su primer elemento.
Por ejemplo:

```
int vector1[10]={25,33,17,49,51,62,11,31,29,75};
int *p;
```

implica que la siguiente asignación sea válida:

```
p = vector1;
```

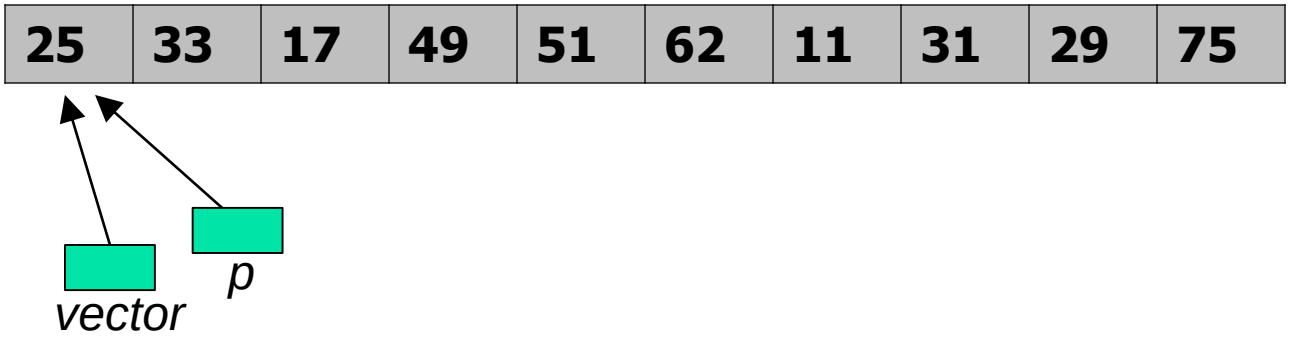
o también:

```
p = &vector1[0];
```

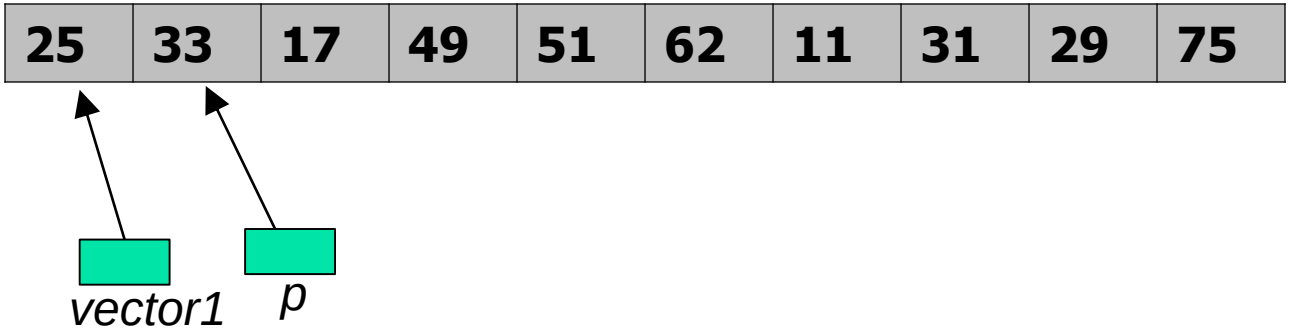
Punteros y arreglos estáticos

Siguiendo el ejemplo anterior, si se asigna

```
int *p;   p=vector1;
```

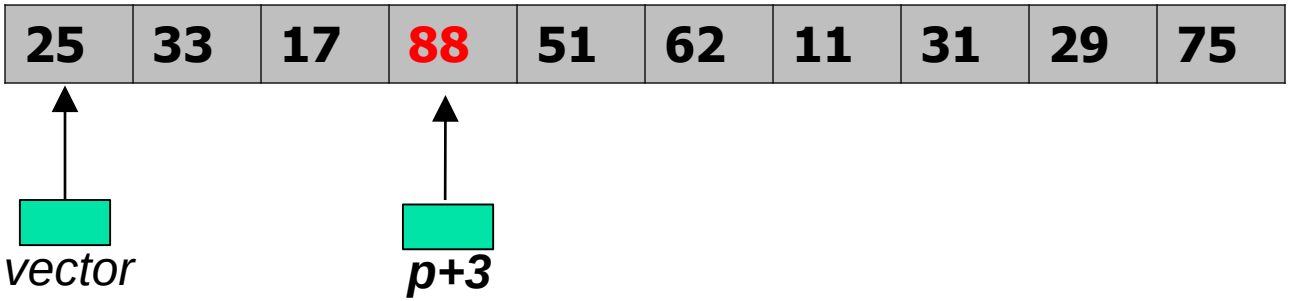



```
cout<<*p; // obtiene 25 en la salida
p++; // desplaza el puntero p 4 bytes
```



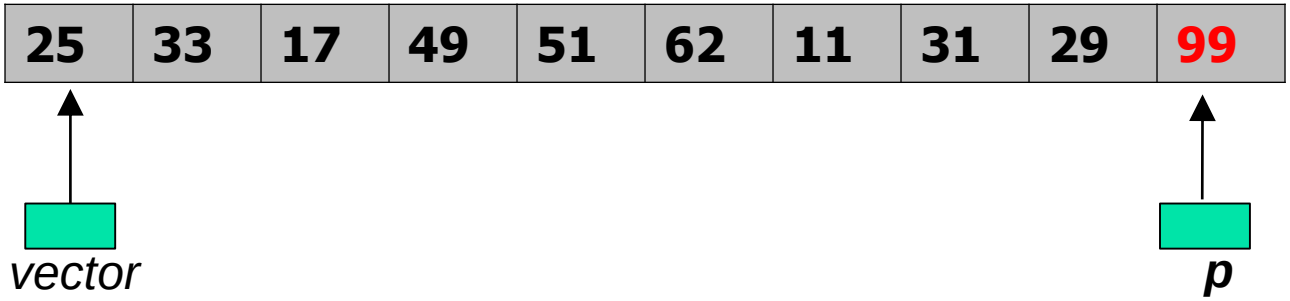
Punteros y arreglos estáticos

```
p=vector; *(p+3)= 88;
```




```
cout<<*vector<<" "<<*(vector+1);
//obtiene 25 y 33 en la salida
```

```
p= &vector[9]; *p= 99
```



Punteros y arreglos estáticos

```
// más punteros y arreglos

#include <iostream>
int main () {
  int numeros[5];
  int *p;
  p = numeros;*p = 10;
  p++; *p = 20;
  p = &numeros[2]; *p = 30;
  p = numeros + 3; *p = 40;
  p = numeros; *(p+4) = 50;
  for (int n=0; n<5; n++)
    cout << *(p+n) << " " ;
  return 0;
}
```

```
10 20 30 40 50
```

Punteros void

- Los punteros *void* pueden apuntar a cualquier tipo de dato.

- Su única limitación es que el dato apuntado no puede ser referenciado directamente (no se puede usar el operador de referencia ***** sobre ellos), dado que la longitud del dato apuntado es indeterminada.

- Por lo anterior, siempre se debe recurrir a la conversión de tipos (*type casting*) o a asignaciones para transformar el puntero *void* en un puntero de un tipo de datos concreto.

Punteros void

```
#include <iostream>

void incrementar (void *dato, int tipo);

int main () {
 char a = 5;
 short b = 9;
 int c = 12;
 incrementar (&a,sizeof(a));
 incrementar (&b,sizeof(b));
 incrementar (&c,sizeof(c));
 cout << (int)a << ", " << b << ", " << c; return 0;
}

void incrementar (void* dato, int tipo)
{ switch (tipo) {
   case sizeof(char) : (*((char*)dato))++;  break;
   case sizeof(short): (*((short*)dato))++;  break;
   case sizeof(int) : (*((int*)dato))++;  break; }
}
```

6, 10, 13

Gestión dinámica de memoria

Resuelven los siguientes problemas:

- Crear variables o estructuras de datos cuyo tamaño se desconoce en tiempo de compilación.
- Administrar grandes volúmenes de datos, aprovechando la memoria del heap (montón).
- Permite administrar (crear, eliminar) datos que no perduran durante toda la ejecución del programa.

Operadores new y new[]

Para requerir memoria dinámica existe el operador **new**. *new* va seguido por un tipo de dato y opcionalmente el número de elementos requeridos dentro de corchetes `[]`. Retorna un puntero que apunta al comienzo del nuevo bloque de memoria asignado durante la ejecución del programa.

Su forma es:

puntero **= new** *tipo*

Ejemplo:

```
int *p= new int;
*p = 25;
cout<<*p;
//solo delete elimina la variablecreada con
delete p;
```

Operadores new y new[]

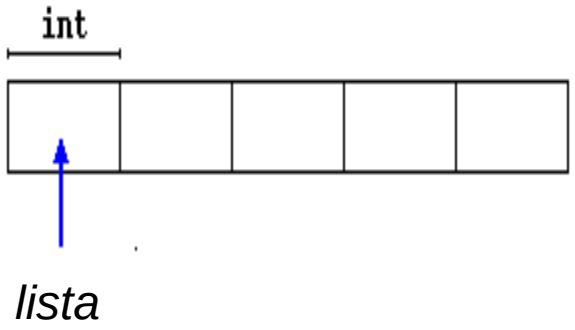
Podemos usar **new** para gestionar dinámicamente un arreglo; en este caso se debe indicar el número de elementos requeridos dentro de corchetes `[]`. Retorna un puntero que apunta al comienzo del nuevo bloque de memoria (al 1er elemento).

Su forma es:

```
puntero = new tipo [elementos]
```

Ejemplo:

```
int *p;
p= new int;
int *lista;
lista= new int[5];
```



Operadores new y new[]

Si el programa no require más hacer uso de una estructura dinámica es posible liberar los recursos que ella invlolucra y ponerlos disponibles para las aplicaciones que se están ejecutando.

```
delete puntero;
delete [] puntero;
```

Memoria dinámica

new y **delete** son operadores que nos permiten administrar variables dinámicamente (crearlas, eliminarlas), más allá de su alcance (scope).

Memoria dinámica

```
// recordador
#include <iostream.h>
#include <stdlib.h>
int main () {
  int i, n;
  int *x, dato, total = 0;
  cout <<"¿Cantidad de datos? ";
  cin>>i;
  x= new int[i];
  if (i==0) { cout<<"sin datos"; return 1; }
  for (int k=0; k<i; k++)
   {cout << "Ingrese un número: ";
    cin>> x[k];
    }
  cout <<"Usted a ingresado estos datos: ";
  for (int k=0; k<i; k++)
           cout << x[k] << ", ";
  delete[] x;
  return 0;
}
```

```
¿Cantidad de datos? 5
Ingrese un número: 75
Ingrese un número: 436
Ingrese un número: 1067
Ingrese un número: 8
Ingrese un número: 32
Usted a ingresado estos datos:
75, 436, 1067, 8, 32,
```

Puntero a estructuras

```
// punteros a estructuras estruct02.cpp
#include <iostream.h>
#include <stdlib.h>

struct pelicula {
            string titulo;
            int anio; };
int main () {
  pelicula peli;
  pelicula  *ppeli;
  ppeli = & peli;
  cout << "Ingresar titulo: ";
  getline(cin, ppeli->titulo);
  cout << "Ingresar anio: ";
  cin>> ppeli->anio ;
  cout << "\nUsted a ingresado:\n";
   cout << ppeli->titulo;
  cout << " (" << ppeli->anio << ")\n";
  return 0; }
```

```
Ingresar titulo: Matrix
Ingresar anio: 1999

Usted a ingresado:
Matrix (1999)
```

Puntero a estructuras

El operador ->. Este es un operador de acceso a miembros y es usado exclusivamente con punteros a estructuras y punteros a clases.

Nos permite eliminar el uso de paréntesis en cada referencia a un miembro de la estructura. En el ejemplo anterior:

```
ppeli->titulo
```

puede ser traducido como:

```
(*ppeli).titulo
```

ambas expresiones son válidas y significan que se está evaluando el elemento titulo de la estructura apuntada por **ppeli.**

Puntero a estructuras

Las distintas notaciones para acceder a miembros de un struct:

`peli.titulo`: *elemento titulo de la struct* ***peli***

`ppeli->titulo`: *elemento titulo del struct apuntado por* ***ppeli***

`(*ppeli).titulo`: *idem anterior*

`*peli.titulo`: *valor apuntado por el puntero titulo del struct* ***peli***

Funciones que reciben punteros

```
dosmax calcula_dosmax(int *x, int n);

void func(char *a);
```

Funciones que devuelven punteros

```
int *func1(int a[], int n);

int *func2(int *a, int n);

char *strcpy(char *c1, const char *c2)
```

Funciones que devuelven arreglos estáticos

```
include <iostream>
using namespace std;
int *func(int a[]);

int main( ) {
int x[]={23,34,45,40,67,78,89,99};
int *p;
p=func(x);// genera un arreglo p de 4 enteros

for (int i=0; i<4; i++)
   cout<<p+i<<"   "<<p[i]<<endl;
cout<<p<<"  "<<x;
return 0;
}
int *func(int a[])
{   int *q= new int[4];
    q[0]=a[4]; q[1]=a[5]; q[2]=a[6]; q[3]=a[7];
    return q;
}
```