



Revisión de conceptos

Funciones

Pasaje de parámetros



Funciones - Bibliografía

- The Cplusplus Tutorial – Cap. 2
- Deitel & Deitel 6ta. Ed. Cap 6
- Eckel Cap. 1



Funciones - Definición

- Definición: una ***función*** es un conjunto de acciones, diseñado generalmente en forma separada y cuyo objetivo es resolver una parte del problema. Estos ***subprogramas*** pueden ser invocados desde diferentes puntos de un mismo programa y también desde otras ***funciones***.

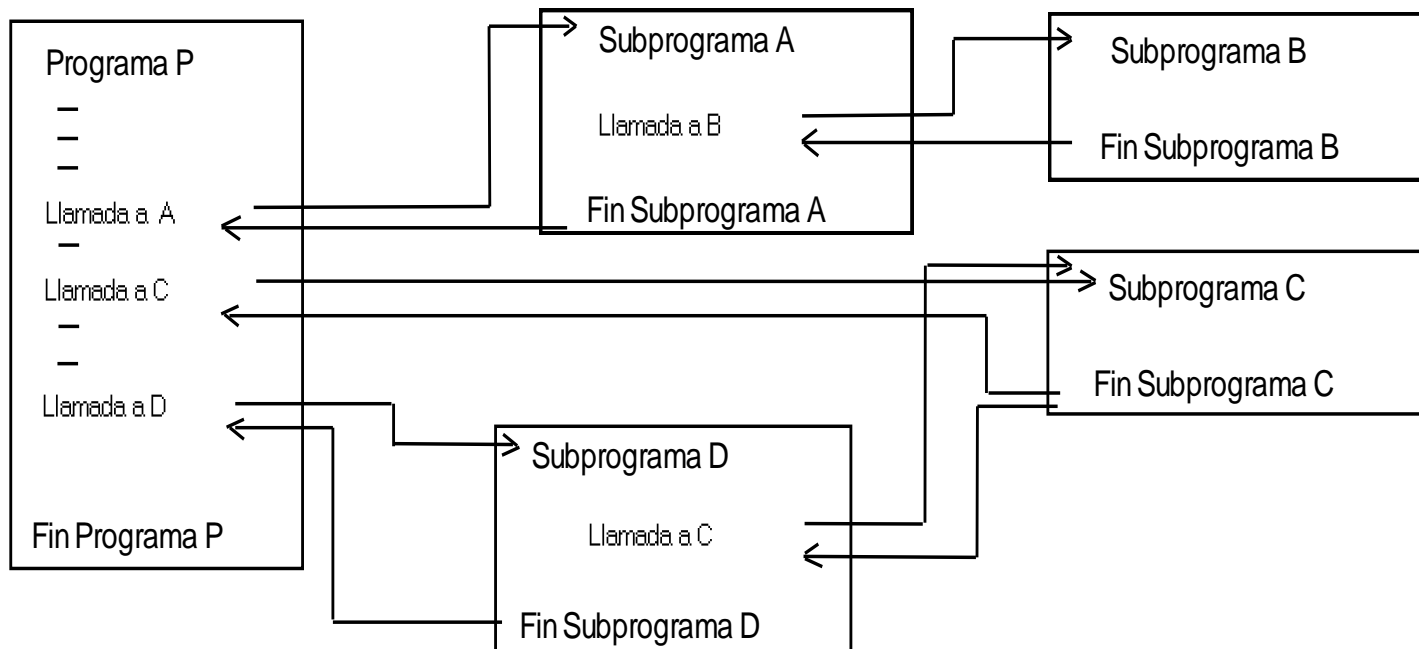


Funciones - Ventajas

La finalidad de las **funciones**, es simplificar el diseño, la codificación y la posterior depuración de los programas. Las ventajas de su empleo se destacan por:

- Reducir la complejidad del programa y lograr mayor modularidad.
- Permitir y facilitar el trabajo en equipo. Cada diseñador puede atacar diferentes módulos.
- Facilitar la prueba de un programa, ya que cada **función** puede ser probada previamente y en forma independiente.
- Optimizar el uso y administración de memoria.

Llamada a subprogramas





Formato de una función C++

*tipo nombre (argumento1,
argumento2, ...) acción*

- *tipo* es el tipo de dato que la función retorna.
- *nombre* es el nombre por el cuál se hace posible llamar a la función.
- *argumentos* (pueden especificarse tantos como sean necesarios). Cada argumento consiste en un tipo de dato seguido por su identificador, como en una declaración de variable (por ejemplo, `int x`) y que actúa dentro de la función como cualquier otra variable. Los argumentos permiten pasar parámetros a la función cuando es llamada. Los diferentes parámetros van separados por comas.
- *acción* es el cuerpo de la función. Puede ser una sola instrucción o un bloque de instrucciones, en este último caso encerrado dentro de corchetes `{ }`.



Ejemplo de función

```
#include <QCoreApplication>
#include <iostream>
#include <iomanip>

float promedio(int x,int y,int z);
```

Declaración de la función
promedio

```
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    int d1, d2, d3;
    float p;
    std::cout <<"Ingrese el primer dato:" ; std::cin >> d1;
    std::cout <<"Ingrese el segundo dato:"; std::cin >> d2;
    std::cout <<"Ingrese el tercer dato:" ; std::cin >> d3;
    p = promedio(d1, d2, d3);
```

Llamada a la función
promedio

```
    std::cout <<std::setprecision(3)<<"El promedio es:" << p << std::endl;

    return a.exec();
}
```

Definición de la función
promedio

```
float promedio(int x,int y,int z)
{ float w=(x+y+z)/3.0 ;
  return (w);
}
```



Intercambio de información con funciones

- Pasaje por valor: significa que cuando se llama a una función con parámetros, lo que se pasa en realidad son valores pero nunca las variables especificadas en sí.



Ejemplo pasaje por valor

```
int main( )
{
    .....
    float p = promedio(d1, d2, d3);
    cout << "Datos:"<<d1<<" "<<d2<<" "<<d3;
        cout << "Promedio:"<<p;  }

    float promedio(int x,int y,int z)
    {   float w=(x+y+z)/3.0 ;
        x++ ;
        return(w) ;    }
```

*d1, d2, d3: parámetros actuales
o de llamada*

*x, y , z: parámetros
formales. Son asignados en
la llamada: x=d1, y=d2,
z=d3*

*Modificación de un
parámetro formal*



Intercambio de información con funciones

- Pasaje por referencia: La referencia consiste en utilizar una variable y un alias que referencia a la misma posición de memoria. Esto significa que si modificamos el alias, la variable correspondiente también se modificará. C++ emplea el operador **&** para realizar esta referencia.



Ejemplo pasaje por referencia

```
#include <QCoreApplication>
#include <iostream>

void intercambia(int &a, int &b);

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    int d1, d2;
    std::cout << "Ingrese el primer dato:" ; std::cin >> d1;
    std::cout << "Ingrese el segundo dato:"; std::cin >> d2;

    intercambia(d1,d2);

    std::cout << "El primer dato vale:" << d1 << std::endl;
    std::cout << "El segundo dato vale:" << d2 << std::endl;

    return a.exec();
}

void intercambia(int &a, int &b)
{ int t=a;
  a=b; b=t;}
```

*Parámetros actuales
o de llamada*

*Parámetros formales a, b
alias de **d1**, **d2**
respectivamente*

*Modificación del parámetro formal a y b y
consecuente modificación de **d1** y **d2***



Parámetros por defecto

- Es posible proponer en el prototipo de la función, parámetros formales inicializados con valores. Estos valores serán asumidos por defecto en el cuerpo de la función **si no se indican parámetros actuales** para tales argumentos.

```
float promedio(int x,int y,int z=10)
```

```
.....
```

```
int main( )
```

```
{
```

```
.....
```

```
float p = promedio(d1, d2, d3);
```

```
.....
```

```
float q = promedio (d1,d2);
```

```
}
```

Llamada a la función con 3 parámetros actuales. Aquí no se empleará el valor por defecto en la función.

Llamada a la función con solo 2 parámetros actuales (el tercero de asumirá por defecto)



Sobrecarga de funciones

- Dos funciones diferentes que tienen el mismo nombre pero el prototipo de sus parámetros es distinto.

```
// sobrecarga de funciones
#include <iostream>
int dividir (int a, int b) { return (a/b); }
float dividir (float a, float b) { return (a/b); }

int main () {
    int x=5,y=2;
    float n=5.0,m=2.0;
    cout << dividir (x,y);
    cout << endl; cout << dividir (n,m);
    return 0;
}
```

2

2.5



Prototipo de funciones

- Declaración previa y corta de la definición completa, pero lo suficientemente completa como para que el compilador sepa qué parámetros necesita y el tipo de datos que devuelve.
- Su formato es:

```
tipo nombre ( tipo_de_argumento1,  
              tipo_de_argumento2, ... );
```



Prototipo de funciones - ejemplo

```
// prototipos
#include <iostream>
void impar (int a);
void par (int a);
int main () {
    int i;
    do {
        cout << "Ingrese un número: (0 to exit)";
        cin >> i; impar (i);
    }
    while (i!=0);
    return 0; }
void impar (int a) {
    if ((a%2)!=0)
        cout << "Number is impar." << endl;
    else par (a); }

void par (int a) {
    if ((a%2)==0)
        cout << "Number is par." << endl;
    else impar (a); }
```

```
Ingrese un número (0 para salir): 9
El número es impar.
Ingrese un número (0 para salir): 6
El número es par.
Ingrese un número (0 para salir):
1030
El número es par.
Ingrese un número (0 para salir): 0
El número es par.
```



Ambito de los identificadores

- El ámbito de una variable lo constituyen las partes del programa en donde dicha variable es reconocida.
- El ámbito de una variable puede ser: un bloque, una función, un archivo.
- Si una variable se declara fuera de todo bloque en un programa se define como *variable global*, y es reconocida en cualquier parte del programa.
- Una variable declarada dentro de un bloque tiene validez solamente en dicho bloque y en otros bloques más internos o anidados. Estas variables reciben el calificativo de variables **locales** o **automáticas**.
- Si se emplea el mismo identificador para una variable local y una global, se anula el acceso a la variable global en el ámbito de validez de la local, pues esta tiene prioridad en su bloque de definición.
- Los parámetros formales de las funciones, tienen validez local mientras se ejecuten las acciones de la función.

Finalizado un bloque o una función las variables locales y parámetros formales dejan de existir y su empleo es causa de error.



Constantes en C++

- C++ admite 4 tipos de constantes diferentes: literales, definidas, declaradas y enumeradas
- Constantes literales

Tipo de constante literal	Ejemplos	
Entera decimal	123 -5	Secuencia de dígitos decimales con o sin signo
Entera octal	0455	Comienza siempre con cero
Entera hexadecimal	0XF4A	Comienza siempre con 0X
Real o punto flotante	192.45 .76 -1.3e+4	Se emplea el signo decimal y/o la notación científica.
Char	'A' '\n' '\f'	Caracteres del código ASCII Secuencia de escape de nueva línea Secuencia de escape de nueva página
String	"Facultad"	Objeto de la clase string



Constantes en C++

■ Constantes definidas

Ciertas constantes pueden referenciarse en C++ a través de un nombre simbólico utilizando la directiva `#define`. También se las conoce como variables constantes.

```
#define Valor 100
```

```
#define Pi 3.14159
```

```
#define ProximaLinea '\n'
```



Constantes en C++

- Constantes declaradas: `const` y `volatile`

Al igual que otros lenguajes como Pascal y Ada, es posible declarar en C++ constantes con nombres o identificadores a través del calificador `const`.

`const` tiene el efecto de una declaración de variable, sólo que el valor asignado al identificador simbólico no puede cambiarse. Si se emplea `volatile` el valor puede ser modificado en el programa o por el hardware o software del sistema.

```
const int n=200;
```

```
const char letra='B';
```

```
const char cadena[]="Computación 2";
```

```
volatile int m=35;
```



Constantes en C++

■ Constantes enumeradas

Internamente el compilador asigna 0,1,2 .. a los valores enumerados. Esto permite usar el tipo enumerado para designar los índices de los arreglos haciendo más claro y legible el programa.

```
enum ciudad {Parana,SantaFe, Rosario, OroVerde};
```

```
enum meses {ENE,FEB,MAR .. DIC}
```

Ejemplo const01.cpp



Tipos de datos

Arreglos

Cadenas de caracteres

Punteros

Variables dinámicas

Estructuras

Arreglos

- Los arreglos son series de elementos (variables) del mismo tipo ubicados consecutivamente en la memoria y pueden ser referenciados individualmente agregando un índice a un nombre único.

Por ejemplo, un arreglo que contiene 5 valores enteros de tipo `int` llamado `comisiones` podría representarse así:



como cualquier otra variable, un arreglo debe ser declarado antes de ser usado. Una declaración típica de un arreglo en C++ es:

```
tipo nombre [cantidad_elementos];
```



Inicialización de arreglos

- Cuando se declara un arreglo de alcance local (dentro de una función), si no se especifica lo contrario, no será inicializado, así que su contenido es indeterminado hasta que se almacene algún valor dentro de él.
- Además, cuando se declara un arreglo, se tiene la posibilidad de asignar valores iniciales a cada uno de sus elementos usando llaves { }. Por ejemplo:

```
int curso[5] = { 16, 2, 77, 40, 12 };
```
- C++ incluye la posibilidad de dejar los corchetes vacíos, [], siendo el tamaño del arreglo definido por el número de valores incluidos dentro de las llaves { }:

```
int curso[] = { 16, 2, 77, 40, 12 };
```



Acceso a valores arreglo

- Siguiendo con el ejemplo previo en el cual *billy* tenía 5 elementos de tipo `int`, el nombre que se puede usar para referenciar cada elemento es:



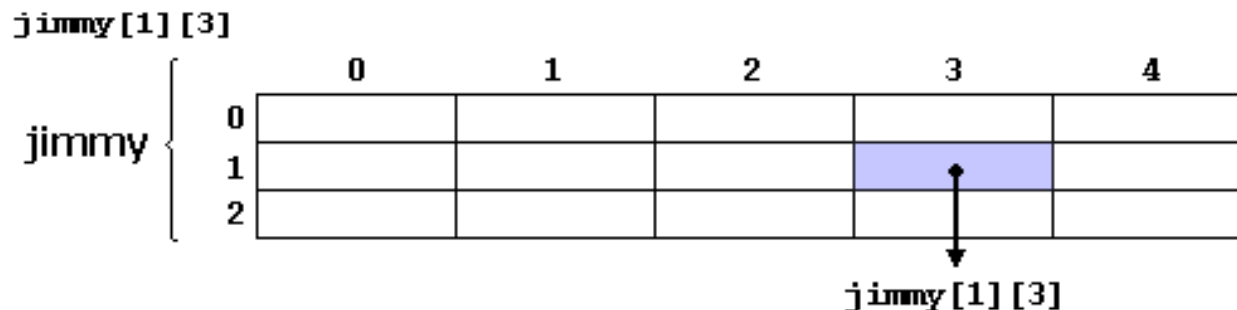
- Por ejemplo, para almacenar el valor 75 en el tercer elemento:

Arreglos multidimensionales

- Se pueden describir como arreglos de arreglos.

Por ejemplo si `jimmy` representa un arreglo bidimensional de 3 por 5 con valores de tipo `int`. La forma de declarar un arreglo de este tipo es: `int jimmy [3][5];`

- La forma de referenciar al elemento de la segunda fila (fila número 1) y cuarta columna (columna número 3) es:





Arreglos multidimensionales - Ejemplos

```
// arreglo multidimensional
#include <iostream>
const int ANCHO=5;
const int ALTO=3;
int jimmy [ALTO][ANCHO];
int n,m;

int main () {
for (n=0;n<ALTO;n++)
    for (m=0;m<ANCHO;m++) {
        jimmy[n][m]=(n+1)*(m+1); }
return 0; }
```

```
// pseudo-arreglo multidimensional
#include <iostream>
const int ANCHO=5;
const int ALTO=3;
int jimmy [ALTO * ANCHO];
int n,m;

int main () {
for (n=0;n<ALTO;n++)
    for (m=0;m<ANCHO;m++) {
        jimmy[n * ANCHO + m]=(n+1)*(m+1); }
return 0; }
```



Arreglos como parámetros

- En C++ no es posible pasar por valor un bloque de memoria completo como parámetro a una función.
- Pero está permitido pasar su dirección de memoria, lo cuál es mucho más rápido y eficiente.
- Para admitir arreglos como parámetros

```
void procedimiento (int arg[])
```



Funciones con parámetros arreglos

```
#include <QCoreApplication>
#include <iostream>

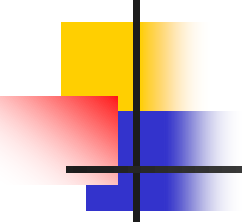
void cargaArreglo(int a[],int tam);
float calcPromedio(int a[],int tam);

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    int aMes[12]={0,0,0,0,0,0,0,0,0,0,0,0};
    cargaArreglo(aMes,12);
    std::cout << "El promedio es:" << calcPromedio(aMes,12) << std::endl;
    return a.exec();
}

void cargaArreglo(int a[],int tam){
    for(int i=0;i<tam;i++)
        a[i]=rand()%101;}

float calcPromedio(int a[],int tam){
    float prom=0;
    for (int i=0;i<tam;++i)
        prom=prom+a[i];
    return prom/tam;
}
```

Resultado 62.6667



```
#include <QCoreApplication>
#include <iostream>
//Carga un arreglo con las ventas mensuales de 5 productos
//La información se almacena en un vector bidimensional
//Debe calcular las ventas promedio de cada producto
```

```
void cargaVentas(int x[][12],int nProd);
void informaProm(int x[][12],int nProd);
```

```
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    int Ventas[5][12] ;
    cargaVentas(Ventas,5);
    informaProm(Ventas,5);
    return a.exec();}
```

```
void cargaVentas(int x[][12],int nProd){
    for (int i=0;i<12;++i)
        for (int j=0;j<nProd;++j)
            x[j][i]=rand()%1001;}
```

```
void informaProm(int x[][12],int nProd){
    float prom;
    for (int j=0;j<nProd;++j){
        prom=0;
        for (int i=0;i<12;++i)
            prom=prom+x[j][i];
        std::cout<< "El promedio del producto " << j;
        std::cout<< " es:" << prom/12 << std::endl;
    }
}
```



Punteros

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

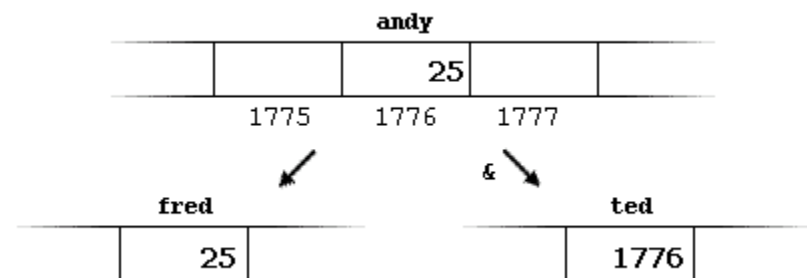
El operador de dirección &

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

```
ted = &andy;
```

- Suponiendo que la variable andy se ha ubicado en la posición de memoria 1776 las asignaciones siguientes tienen el resultado:

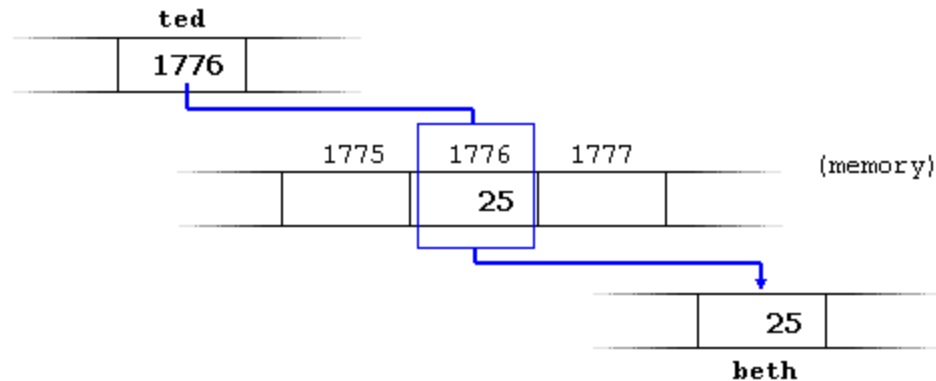
```
andy = 25;  
fred = andy;  
ted = &andy;
```



Operador de referencia *

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

```
beth = *ted;
```





Operadores de punteros

- **Operador de Dirección o Dereferencia (&)**

Se usa como prefijo de variables y puede ser traducido como **"la dirección de memoria de"**, así: `&variable1` se puede traducir como *"la dirección de memoria de variable1"*

- **Operador de Referencia (*)**

Indica que lo que debe ser evaluado es el contenido apuntado por la expresión considerada como una dirección. Puede ser traducido como **"valor apuntado por"**. `*mipuntero` puede ser traducido como *"valor apuntado por mipuntero"*.



Declaración de variables puntero

- *tipo * nombre_puntero;*
- Ejemplo:

```
// más punteros
#include <iostream>
int main () {
    int valor1 = 5, valor2 = 15;
    int *p1, *p2;
    p1 = &valor1; // p1 = dirección de valor1
    p2 = &valor2; // p2 = 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



Punteros y arreglos

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

```
int numeros [20];
```

```
int * p;
```

- implica que la siguiente asignación sea válida:

```
p = numeros;
```

```
// más punteros punteros03.cpp
#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 << numeros[n] << ", " ;
    return 0; }
```

10, 20, 30, 40, 50,



Operaciones entre 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=NULL;    //Dirección nula: 0x0000 en hexadecimal
```

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

```
int *p;  
.....  
p+=4;    //la dir original de p se ha incrementado 8 bytes  
p-=1;    //La dir anterior de p se decrementó en 2 bytes
```

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

```
*u < *v      *u >= *v      u==v      u!=v      u==NULL
```



Punteros tipos 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 asterisco * sobre ellos), dado que su longitud es siempre indeterminada, y por esta razón 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 al cual se pueda referenciar.



Punteros void - ejemplo

```
// incrementor de enteros punteros04.cpp
#include <iostream>
void incrementar (void* dato, int tipo) {
    switch (tipo) {
        case sizeof(char) : (*((char*)dato))++; break;
        case sizeof(short): (*((short*)dato))++; break;
        case sizeof(long) : (*((long*)dato))++; break; }
    }

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

6, 10, 13



Punteros a funciones

- La mayor utilidad de esto es pasar una función como parámetro a otra función, dado que éstas no pueden ser pasadas por refererencia.

```
// puntero a función punteros05.cpp
#include <iostream>
int suma (int a, int b)
{ return (a+b); }
int resta (int a, int b)
{ return (a-b); }

int (*menos)(int,int) = resta;
int operacion (int x, int y, int (*func_a_llamar)(int,int))
{ int g; g = (*func_a_llamar)(x,y); return (g); }

int main () {
int m,n; m = operacion (7, 5, &suma);
n = operacion (20, m, menos);
cout <<n; return 0; }
```



Memoria dinámica

- Resuelve problemas:
- Crear variables cuyo tamaño se desconoce en tiempo de compilación.
- Variables que no perduran durante toda la ejecución del programa.

Operadores new 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*

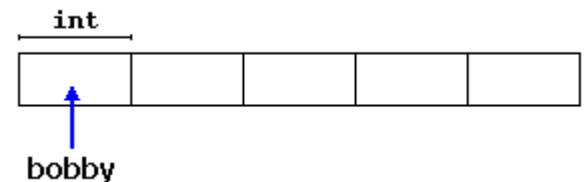
o

puntero = **new** *tipo* [*elementos*]

Por ejemplo:

```
int * uno; uno = new int;
```

```
int * bobby; bobby = new int [5];
```





Operador delete delete[]

- Una vez que no se precisa más hacer uso de la memoria debería ser liberada y ponerse disponible para las aplicaciones que se están ejecutando.

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



Memoria dinámica - ejemplo

```
// recordador
#include <iostream>
#include <stdlib>
int main () {
    char ingreso [100];
    int i, n;
    long * l, total = 0;
    cout << "¿Cuántos números va a ingresar? ";
    cin.getline (ingreso,100);
    i=atoi (ingreso);
    l= new long[i];
    if (l == NULL) exit (1);
    for (n=0; n<i; n++) {
        cout << "Ingrese un número: ";
        cin.getline (ingreso,100);
        l[n]=atol (ingreso); }
    cout << "Usted a ingresado: ";
    for (n=0; n<i; n++) cout << l[n] << ", ";
    delete[] l; return 0; }
```

¿Cuántos números va a ingresar? 5

Ingrese un número: 75

Ingrese un número: 436

Ingrese un número: 1067

Ingrese un número: 8

Ingrese un número: 32

Usted a ingresado: 75, 436, 1067, 8, 32,



Estructura de datos

- Una estructura de datos es un conjunto de diversos tipos de datos que pueden tener distintos tamaños agrupados juntos bajo una única declaración. Su forma es la siguiente:

```
struct nombre_modelo {  
    tipo1 elemento1;  
    tipo2 elemento2;  
    tipo3 elemento3;  
    . . .  
} nombre;
```

nombre_modelo es un nombre para el modelo del tipo de estructura y el parámetro opcional ***nombre*** es un identificador válido (o identificadores) para las distintas instancias de la estructura. Dentro de las llaves { } están los tipos y los sub-identificadores (campos) correspondientes a los elementos que componen la estructura.

Los miembros individuales de una estructura pueden ser de tipos simples, arrays, punteros e inclusive **struct**.



Arreglo de estructuras

```
// arreglo de estructuras estruct01
#include <iostream>
#include <stdlib.h>
const int N_PELICULAS=5;
struct pelicula_t {
    char titulo [50];
    int anio; } films [N_PELICULAS];

void imprimir_pelicula (pelicula_t pelicula);

int main () {
    char buffer [50];
    int n;
    for (n=0; n<N_PELICULAS; n++) {
        cout << "Ingresar titulo: ";
        cin.getline (films[n].titulo,50);
        cout << "Ingresar anio: ";
        cin.getline (buffer,50);
        films[n].anio = atoi (buffer); }
    cout << endl << "Usted a ingresado estas peliculas:" << endl;
    for (n=0; n<N_PELICULAS; n++) imprimir_pelicula(films[n]);
    return 0; }

void imprimir_pelicula (pelicula_t pelicula) {
    cout << pelicula.titulo; cout << " (" << pelicula.anio << ")"
    << endl; }
```

Ingresar titulo: Alien
Ingresar anio: 1979
Ingresar titulo: Blade Runner
Ingresar anio: 1982
Ingresar titulo: Matrix
Ingresar anio: 1999
Ingresar titulo: Rear Window
Ingresar anio: 1954
Ingresar titulo: Taxi Driver
Ingresar anio: 1975

Usted a ingresado estas peliculas:
Alien (1979)
Blade Runner (1982)
Matrix (1999)
Rear Window (1954)
Taxi Driver (1975)



Puntero a estructuras

```
// punteros a estructuras estruct02.cpp
#include <iostream.h>
#include <stdlib.h>
struct pelicula_t {
    char titulo [50];
    int anio; };
int main () {
    char buffer[50];
    pelicula_t apelicula;
    pelicula_t * ppelicula;
    ppelicula = & apelicula;
    cout << "Ingresar titulo: ";
    cin.getline (ppelicula->titulo,50);
    cout << "Ingresar anio: ";
    cin.getline (buffer,50);
    ppelicula->anio = atoi (buffer);
    cout << endl << "Usted a ingresado:" << endl;
    cout << ppelicula->titulo;
    cout << " (" << ppelicula->anio << ")" << endl;
    return 0; }
```

Ingresar titulo: Matrix
Ingresar anio: 1999

Usted a ingresado:
Matrix (1999)



Puntero a estructuras

- **el operador ->. Este es un operador de referencia 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:

```
ppelícula->título
```

puede ser traducido como:

```
(*ppelícula).título
```

ambas expresiones son válidas y significan que se está evaluando el elemento `título` de la estructura apuntada por `ppelícula`.



Puntero a estructuras

Expresión	Descripción	Equivalente
<code>ppelicula.titulo</code>	Elemento titulo de la estructura ppelicula	
<code>ppelicula->titulo</code>	Elemento titulo de la estructura <u>apuntada por</u> ppelicula	<code>(*ppelicula).titulo</code>
<code>*ppelicula.titulo</code>	Valor <u>apuntado por</u> el elemento titulo de la estructura ppelicula	<code>*(ppelicula.titulo)</code>