Estructura de Datos (EDA)

Introducción

Mag. Ing. Nancy López

Objetivos del curso

- Presentar y analizar las estructuras de datos y algoritmos que forman la base para la resolución de problemas en computación;
- Introducir nociones de análisis de algoritmos;
- Aprender a implementar sistemas (eficientes) de porte mediano

Objetivo

- Mostar las principales características del lenguaje que será utilizado en el curso.
- A partir de esta clase van a necesitar salir a practicar bastante de C para familiarizarse. Esta clase apunta a ser una ayuda inicial.
- Este no es un curso de C/C++, es un curso de estructuras de datos y algoritmos.

Lenguaje

- Lenguaje: C*
- Es el lenguaje C, pero sumándole algunas (pocas) cosas de C++
- Es “ficticio”

Ejemplo

- hola.cpp:

```
#include <iostream>
using namespace std;
int main()
{
    printf("¡Hola, mundo!\n");
    return 0;
}
```

- main es una función especial, a partir de la cual comienza la ejecución del programa.

Compilación

- Archivos con extensión .cpp

- IDE: Zinjal

Objetos de un programa

Clasificación de los datos: según la representación de la información para su almacenamiento y trata-miento.

<table>
<tr><td rowspan="2">Datos Básicos</td><td>Numéricos</td><td colspan="3">Entero
Real</td></tr>
<tr><td colspan="4">Carácter
Lógico</td></tr>
<tr><td>Datos Derivados</td><td colspan="4">Punteros</td></tr>
<tr><td rowspan="4">Datos Estructurados</td><td rowspan="3">Internos</td><td colspan="2">Estáticos Lineales</td><td>Tabla</td></tr>
<tr><td rowspan="2">Dinámicos</td><td>Lineales</td><td>Lista
Pila
Cola</td></tr>
<tr><td>No Lineales</td><td>Árbol
Grafo</td></tr>
<tr><td>Externos</td><td colspan="3">Fichero
Base de datos</td></tr>
<tr><td>Compuesto</td><td colspan="3">Estructura de datos o registro</td></tr>
</table>

Tipos de datos elementales

Reglas para los nombres

- Pueden estar constituidos por letras y dígitos y en algunos casos el guión bajo (_)
- Deben comenzar por una letra.
- No deben contener espacios.
- El número máximo de caracteres y nombres reservados que se pueden emplear dependen del compilador utilizado.
- El nombre asignado debe tener relación con la información que contiene, pudiéndose emplear abreviaturas que sean significativas.

Tipos de datos elementales

- Entero: int
- Caracter: char
- Real: float
- Booleano: bool (de C++)

Ejemplos:

```
int i;
char c;
float f;
bool b;
i = 1;
b = false;
```

Declaración de variables

- C++ permite declarar las variables en cualquier parte.
- Recomendación: declararlas juntas al comienzo del programa.

```
int minimo(int tam, int[] arreglo) {
    int iMin = 0;
    for (int i = 1; i < tam; i++)
            if (arreglo[i] < arreglo[iMin])
                    iMin = i;
    return arreglo[iMin];
}
```

Comentarios

```
/* comentario
de
varias
lineas */
int i = 1; /* asigno 1 a i */
char c; // comentario de una linea (C++)
float f;
// otro comentario
```

Expresiones I

- Operador de asignación: =

```
int a;
int b = 2;
a = 7;
a = b;
```

La asignación retorna un valor, por lo que es válido: a = b = 9

Error común: confundir con comparación booleana

Expresiones II

- Operadores de comparación: ==, !=, <, <=, > y >=
- Operadores lógicos: &&, || y !
- Operadores aritméticos: +, -, *, / y%

Precedencia:

a+1 < b && c == 9*d || e < 7

equivale a:

(((a+1) < b) && (c == (9*d))) || (e < 7)

Expresiones III

- Incremento y decremento: ++ y --
  - ++a incrementa el valor de a y retorna su valor luego del incremento
  - a++ incrementa el valor de a y retorna su valor antes del incremento
- Análogo para decrementar

```
int a = 1;
int b, c;
b = ++a;
c = a++;
```

Valores finales:

a -> 3

b ->2

c ->2

Constantes I

- Se pueden definir utilizando define:

```c
#include <stdio.h>

#define BASE 10
#define ALTURA 5

int main() {
    int area = BASE * ALTURA;
    printf("Area: %d", area);
    return 0;
}
```

Constantes II

- O usando const:

```c
#include <stdio.h>

int main() {
    const int BASE = 10;
    const int ALTURA = 5;
    int area = BASE * ALTURA;
    printf("Area: %d", area);
    return 0;
}
```

- La diferencia es que ***define*** es un reemplazo de texto antes de compilar y ***const*** utiliza variables (y por lo tanto tiene su espacio de memoria, su tipo, etc.) que no se pueden modificar.
- Es buena práctica definir los nombres de las constantes en mayúsculas.

Estructuras de control I

- Sentencia if

```
if (6 <= valor && valor <= 12) {
    printf("Aprobado");
    cantidad_aprobados++;
} else if (valor >= 3)
    printf("Examen");
else if (valor >= 0)
    printf("Reprobado");
else
    printf("Valor incorrecto");
```

Estructuras de control II

- Selección: sentencia ***switch***

```
switch (valor) {
    case 6: case 7: case 8: case 9: case 10: case 11: case ↩
        12:
        printf("Aprobado");
        cantidad_aprobados++;
        break;
    case 3: case 4: case 5:
        printf("Examen");
        break;
    case 0: case 1: case 2:
        printf("Reprobado");
        break;
    default:
        printf("Valor incorrecto");
}
```

Estructuras de control III

- Iteración
  - Sentencia while:

    while (condicion)

      cuerpo

```
int i = 0;
while (i < 10) {
    printf("*");
    i++;
}
```

Estructuras de control IV

- Iteración
  - Sentencia for:

    for (inicio; condicion; paso)

```
for (int i = 0; i < 10; i++)
    printf("*");
```

Tipos de datos estructurados

enum

- Las enumeraciones se utilizan para sustituir los #define.

Ej. enum colores(verde, amarillo, azul);

- Si no les doy valores, toma valores desde el 0.
- Ej. VERDE=0
- AMARILLO=1
- AZUL=2

enum

```
enum mes {enero, febrero, marzo, abril, mayo, junio, julio, ↩
    agosto, setiembre, octubre, noviembre, diciembre};
mes este_mes = marzo;
```

Tipos de datos estructurados II

Estructuras : struct

```
struct fecha {
    int f_dia;
    mes f_mes;
    int f_anio;
};
```

- Se usa . para acceder a los miembros:

```
fecha hoy;

hoy.f_dia = 2;
hoy.f_mes = marzo;
hoy.f_anio = 2016;

int dia_hoy = hoy.f_dia;
```

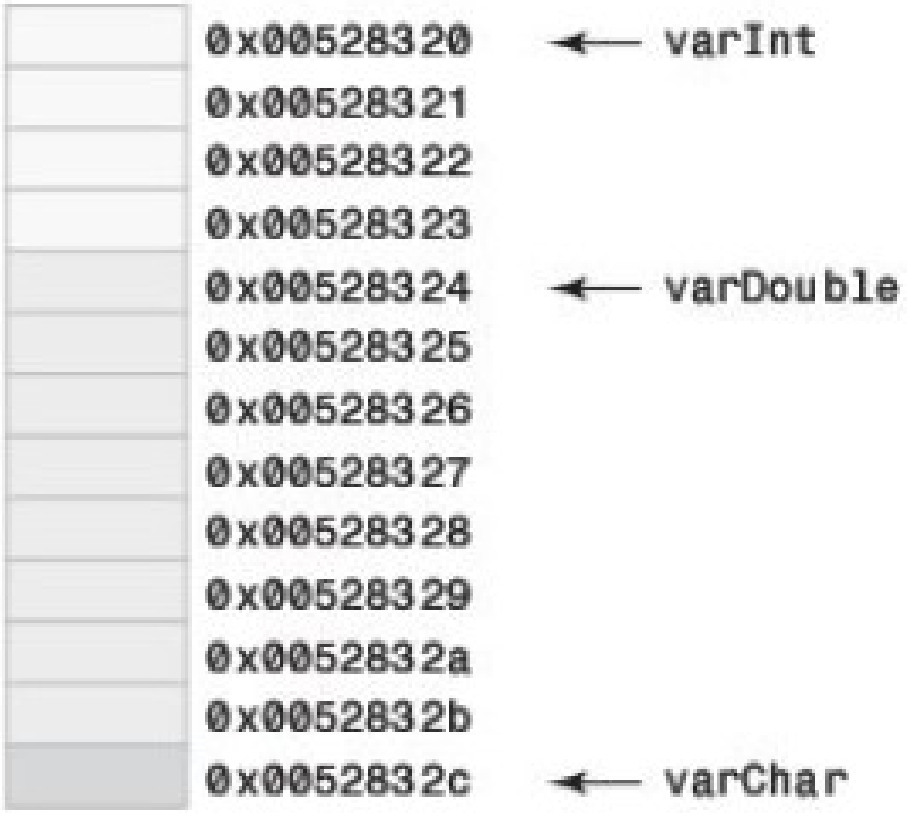
Punteros

- Al definir una variable:
- int a; double b; char pal;

1. Se reserva memoria para un valor del tipo especificado.
2. El nombre de la variable se asocia con la dirección de esa memoria.
3. La memoria se inicializa con valores procedentes (si los hay).

Punteros

- Los *char* ocupan un byte, los *int* ocupan 4 bytes y los *double* ocupan 8 bytes:



Punteros

- Se puede obtener la dirección de operador de dirección (&).
- ***&variable*** es la dirección de la variable

```
&a es 0x005208320
&b es 0x005208324
&pal es 0x00520832c
```

Punteros

- Un puntero es una variable cuyo valor es una dirección de memoria.

- Variable que contiene la dirección de memoria en la que se encuentra almacenada otra variable.

Punteros

Declaración e inicialización

- Formas:
- Tipo *variablePuntero
- Tipo *variablePuntero = direccion

- Donde *Tipo* es cualquier tipo y *dirección* es la dirección de un objeto del *Tipo* especificado.
- El operador * debe preceder a cada identificador de tipo puntero.
- En la segunda versión, la dirección de inicialización debe ser la de un objeto del mismo tipo que aquél al que apunta el puntero. Se dice que el puntero está ligado a ese tipo.

Punteros

Declaración e inicialización

A un puntero se le puede asignar:

- El **valor** de **otro puntero**, del mismo tipo.
- La **dirección de memoria** de una **variable** cuyo tipo coincida en el tipo_base del puntero.

Ejemplo:

```
int x=15, *p1=NULL, *p2=NULL;
p1 = &x; // Se le asigna la dirección de memoria de X
p2 = p1; // Se le asigna el valor de p1
```

Los dos apuntan a la misma variable

Punteros

Declaración e inicialización

- C no inicializa los punteros cuando se declaran.
- TODO PUNTERO DEBE INICIALIZARSE, ya que en caso contrario tras ser declarado apuntaría a cualquier sitio (PELIGROSO) → al usarlo puede p.ej. modificar una parte de la memoria reservada a otra variable.
- Si aún no sabemos dónde debe apuntar, se le asignará el valor NULL (nulo) →No apunta a ningún sitio en especial.
- Ejemplo: **int *p = NULL;**

Punteros

Operadores a punteros

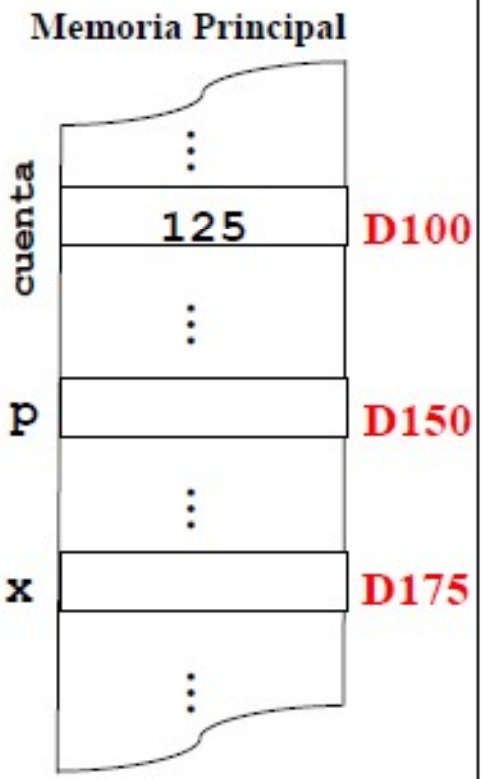
Existen dos operadores especiales de punteros: * y &.

- El operador & (operador dirección), aplicado sobre el nombre de una variable, devuelve su dirección de memoria.
- El operador * (operador indirección) aplicado sobre una variable de tipo puntero permite acceder al dato al que apunta, es decir, al valor de la variable situada en esa dirección de memoria

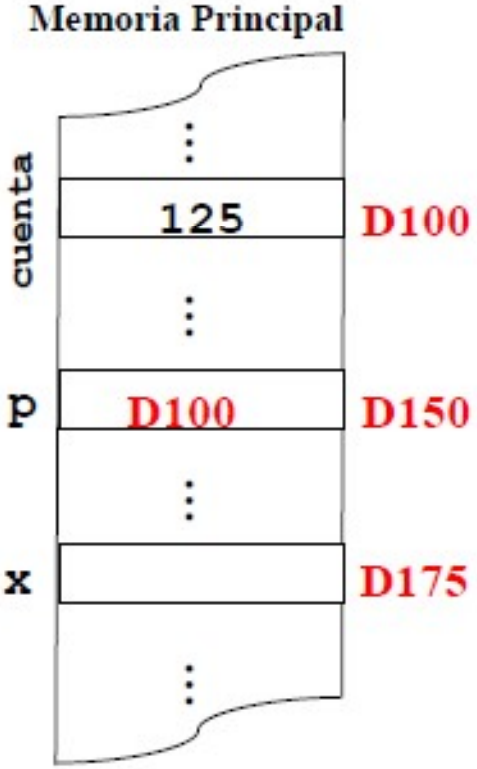
Punteros

Ejemplo

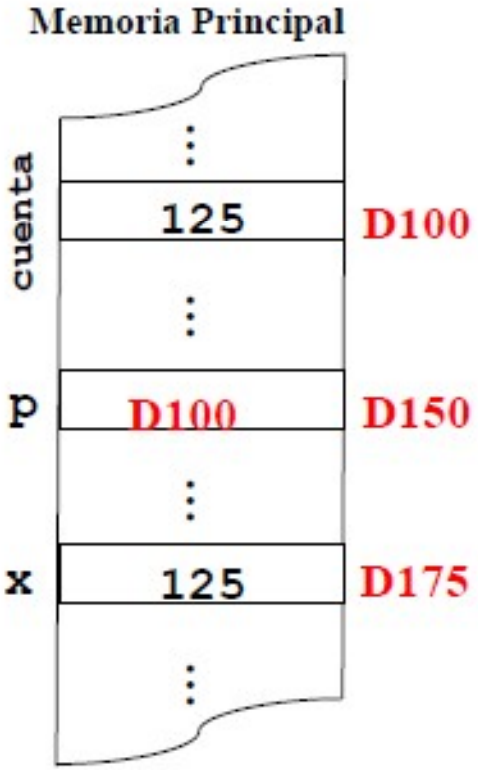
```
int cuenta = 125;
int *p;
int x;
```




```
p = &cuenta;
```




```
x = *p;
```



```
#include<iostream>
using namespace std;
int main()
{
  int x=15, *p=NULL, *q=NULL;
  p=&x;
  q=p;
  cout<< &x <<"\t"<<"Direccion de memoria de x"<<endl;
  cout<< p <<"\t"<<“Valor guardado en p"<<endl;
  cout<< q <<"\t"<<“Valor guardado en q"<<endl;
  cout<< &p <<"\t"<<"Direccion de memoria de p"<<endl;
  cout<< &q <<"\t"<<"Direccion de memoria de q"<<endl;
  cout<< *p <<"\t\t"<<"Valor de p"<<endl;
  cout<< *q <<"\t\t"<<"Valor de q"<<endl;
  cout<< x <<"\t\t"<<"Valor de x"<<endl;
  return 0;
}
```

Punteros - Ejemplo

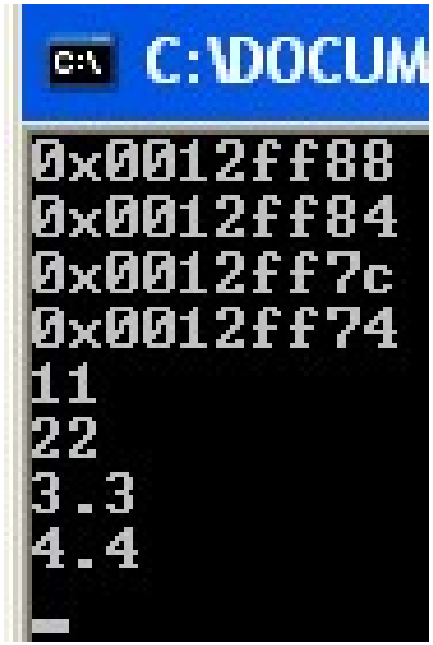
```
int main() {
    int i=100, *p, *q;
    p = &i;
    q = p;
    if (p==q) /* estamos comparando dos punteros */
            cout << "p apunta a la misma dirección de memoria que q"<<endl;
    *p = *q + 2; /* El * tiene más prioridad que el + */
    cout<<"El valor de *p es: "<<*p<<endl;
    cout<< "El valor de *q es: " << *q <<endl;
    (*q)++; /* ante la duda de prioridades usamos paréntesis */
    cout<< "El valor de *p es: " << *p <<endl;
    cout<< "El valor de *q es: " << *q <<endl;
    i--;
    cout<< "El valor de i es: "<< i;
    return 0;
}
```

```
#include<conio.h>
#include<iostream.h>
int main()
{
  int i=11;
  int j=22;
  double d=3.3, e=4.4;

  int *iPunt, *jPunt;
  double *dPunt, *ePunt;

  iPunt=&i;
  jPunt=&j;
  dPunt=&d;
  ePunt=&e;

  cout<<iPunt<<endl;
  cout<<jPunt<<endl;
  cout<<dPunt<<endl;
  cout<<ePunt<<endl;
  cout<<*iPunt<<endl;
  cout<<*jPunt<<endl;
  cout<<*dPunt<<endl;
  cout<<*ePunt<<endl;
  return 0;
}
```



Punteros

Errores comunes

- Un puntero con un valor erróneo es **MUY PELIGROSO, y el ERROR** más **DIFÍCIL DE DEPURAR**, porque en compilación **NO SE DETECTA**, y los errores tras ejecutar pueden **DESPISTAR**.
- Hay que hacer caso a los AVISOS (WARNINGS) ya que una sospecha sobre un PUNTERO puede provocar un gravísimo error en ejecución.

Punteros

Errores comunes

- Asignar punteros de distinto tipo (un float con un char, un int con un double...).
  Por ejemplo:
  int a=1 ;
  *int *p ;*
  double b=2.0 ;
  *double *q ;*
  p=&a ;
  q=&b ;
  q=p; ERROR

Punteros

Errores comunes

- *Usar punteros no inicializados.*

Por ejemplo:

```
char *p;
*p='a';
```

(FALTARÍA PONER:

```
char c='a';
p=&c; )
```

Punteros

Errores comunes

- *Asignar valores al puntero en vez de a la variable:*
- *Por ejemplo:*

```
int x;
int *p;
p=&x;
p=7                MAL (DEBERÍA SER *p=7)
```

Punteros

Errores comunes

- Hacer asignaciones a la dirección de memoria NULL (dirección de memoria cero).

Por ejemplo:

*int *p=NULL;*
**p=6;* // ERROR (NO SE PUEDE ASIGNAR NADA A LA DIRECCIÓN DE MEMORIA NULL)

Punteros

Errores comunes

```
#include <stdio.h>
void main(){
int i, *p;
i = 50;
p = i; //Error: “invalid conversion from ‘int’ to ‘int*’”
cout<<“El valor de i es: ”<<  i <<endl;
cout<<“El valor de *p es:  ” << *p;
}
```

Lo correcto sería: p=&i;

Punteros

Errores comunes

- Puntero no inicializado

```
void main(){
int i, *p;
i = 50;
*p = i;
Cout<<“El valor de i es: << i<<endl;
Cout<<“El valor de *p es << *p<<endl;
}
```

El compilador no da error.

Punteros

Relación con vectores

- Vector: conjunto de variables del mismo tipo.
- El nombre del vector es un puntero al primer elemento.

Por lo tanto:

v[i] es equivalente a **(v+i)*

&v[i] es equivalente a *(v+i)*

Punteros

Relación con vectores

```
#include<iostream.h>
#include<conio.h>
int main()
{
 const int capacidad=10;
 int i;
 int vector[capacidad];
 int *puntVector = &vector[0];
 for(i=0; i<capacidad;i++)
  {
   cout<<"Ingrese número";
   cin>>*(puntVector+i);
  }
cout<<"Con punteros" <<endl;
 for(i=0;i<capacidad;i++)
  cout<<*(puntVector+i)<<endl;

cout<<"Sin punteros" <<endl;
 for (i=0; i<capacidad; i++)
  cout<<vector[i]<<endl;

 return 0;
}
```

Punteros

Asignación dinámica de memoria

- C/C++ permite declarar variables en tiempo de ejecución.
- Asignación dinámica de memoria implica reservar memoria para estas variables llamadas “variables anónimas”.
- Heap: zona de la memoria donde se reserva espacio para asignarlo a las variables dinámicas.

Punteros

Asignación dinámica de memoria

- Puesto que esta zona tiene tamaño limitado, debemos reservar y liberar la memoria.
- Esto se realiza mediantes las funciones ***new()*** y ***delete()***.
- Ej.

```
int  *p = new  int;
delete  p;
```

Importante: delete sólo libera la memoria, no apunta a null.

Punteros

Asignación dinámica de memoria

- Si hay memoria, new devuelve un puntero al primer byte de la región de memoria reservada.
- Si no hay memoria, se produce un fallo de asignación y new devuelve un nulo.

- **typedef char *puntChar ;**
- puntChar p = new char;

Punteros

Asignación dinámica de memoria

- Siempre comprobar si el puntero devuelto no es nulo.

```
char *p= new char;
if (p == NULL)
{
// No se ha podido reservar la memoria deseada
// Tratar error
}
```

Punteros

Asignación dinámica de memoria

- También:
- **if (p)** //Si es válido, equivale a if(p!=NULL)
- **if(!p)** //Si no es nulo, equivale a if(p==NULL)

- Punteros

```
int *p3; // el asterisco puede ir junto a la variable (a la ↩
    izquierda)

int* p4, p5;
```

  - ¿p5 es un puntero? **No**, la declaración anterior es equivalente a la siguiente:

```
int* p4;
int p5;
```

  - Es como si el asterisco se pegara a la variable y no al tipo
  - Si queremos dos punteros:

```
int* p4, * p5;
```

Punteros

Punteros a estructuras

- Dos usos principales de punteros a estructuras:

1. Pasar por referencia una estructura a una función.
2. Crear listas enlazadas y otras estructuras dinámicas utilizando asignación dinámica de memoria.

Punteros

Punteros a estructuras

Primera Forma

```
struct  bal {
            float balance;
            char nombre[80];
            };
```

- bal persona; // Variable persona de tipo struct bal
- bal *p; // Puntero a una estructura de tipo struct bal
- p = &persona; // p tiene la dirección de la variable persona

Punteros

Punteros a estructuras

Segunda Forma

```
struct  bal {
            float balance;
            char nombre[80];
            };
```

- p = new bal; // p apunta a una variable anónima

- En **AMBOS CASOS**, la manera de acceder a los campos es:
- p->balance

Punteros

Punteros a estructuras

```
struct  bal {
            float balance;
            char nombre[80];
            };
int main()
{ int i=0;
  bal *p = new bal[10];
  p->balance=3.0;
  strcpy(p->nombre, "Juana");
  p++;
  p->balance=4.0;
  strcpy(p->nombre, "Julia");
  p++;
  p -=2;
  while(p!=NULL && i<2)
   {
     cout<<p->balance<<endl;
     cout<<p->nombre<<endl;
     i++;
     p++;
   }
  return 0;
}
```

Tipos de datos estructurados

- Arreglos
  - Varios objetos del mismo tipo puestos consecutivamente en memoria
  - El primer elemento está en el índice 0
  - Estáticos:

```
int arr[2]; // valores posibles: arr[0] y arr[1]

int vector[5] = {1, 2, 3, 4, 5};
int matriz[2][3] = {{1, 2, 3}, {4, 5, 6}};

int suma = 0;
for (int i = 0; i < 5; i++)
    suma += vector[i];
```

  - Dinámicos:

```
int* vector = new int[10];
delete [] vector;
```

- Arreglos

- Tener en cuenta que ni C ni C++ verifican que el índice esté dentro del rango permitido. Te deja así acceder a otra dirección de memoria, y a veces puede dar segmentation fault.

Conversión de tipos

- La mayoría de las conversiones son implícitas

```
float vf = 1.6;
int vi = 1 + vf; // vi = 2 (float se trunca)
vi = 1 + vf + vf; // vi = 4 (cast al "más grande")
vi = vi + true; // vi = 5 (true es 1)
vi = vi + false; // vi = 5 (false es 0)
vi = 'a' + 1; // vi = 98 (valor ASCII)
char vc = 'a' + 1; // vc = 'b'
vf = 1.5 + vi // vf = 99.500000
bool vb = 237; // vb = true (0 es false, otro true)
vf = 3 / 2; // vf = 1.000000
```

- Se puede hacer cast explícito

```
vf = (float)3 / 2; // vf = 1.500000
```

Conversión de tipos II

- ¿Cuál es el valor de `res`?

```
int res;
int i = 5 - 4.3;
bool b = 100.1;

if (i = 0)
    res = b + 100.9;
else
    res = b + i;
```

- ¿El resultado es 1?
- **Error común**: haber puesto = en lugar de ==. Aunque el resultado es 1, si cambiamos por == obtenemos 101.

Funciones

```
int sumar(int a, int b) {
    return a + b;
}
```

- Se puede invocar así:

```
c = sumar(3, 8);
```

Procedimientos

```
void imprimirSuma(int a, int b) {
    int suma = a + b;
    printf("La suma es: %d\n", suma);
}
```

Funciones II

- Las funciones no se puede anidar
- En C todos los parámetros se pasan por valor
  - En C++ (y C*) existe el pasaje por referencia (&)

```
void sumarEnB(int a, int & b) {
    b = a + b;
}
```

Entrada y salida de datos

- Para la entrada y salida de datos se usan flujos (o stream)
- Se usa la biblioteca iostream
- Los flujos son:

- cout(consola out): flujo de salida a la pantalla
- cin(consola in): flujo de entrada

- Se usan en conjunto con los operadores << y >>

Entrada y salida de datos

- Ejemplo:

```
#include<iostream.h>

int main()
{
  int a;
  cout<<"Ingrese un valor ";
  cin>>a;
  cout<<"El doble del valor ingresado es: "<<a*2;
  return 0;
}
```

Entrada y salida de datos

- En el ejemplo anterior se muestra texto y valores de variables en la misma línea:

  cout<<"El doble del valor ingresado es: "<<a*2;

- Los símbolos “<<“ se usan para separar texto de valores.

Formato de entrada y salida de datos

- C/C++ cuenta con algunos manipuladores de flujos (usa <iomanip.h>):
  - **endl**: se imprime un '\n' y se vacía el buffer de salida.
  - **setw(int num)**: establece la anchura mínima de campo. Necesita declarar <iomanip.h>
  - **setprecision(p)**: establece el número de cifras