

# ANSI-C

Patricio Loncomilla  
EL710

---

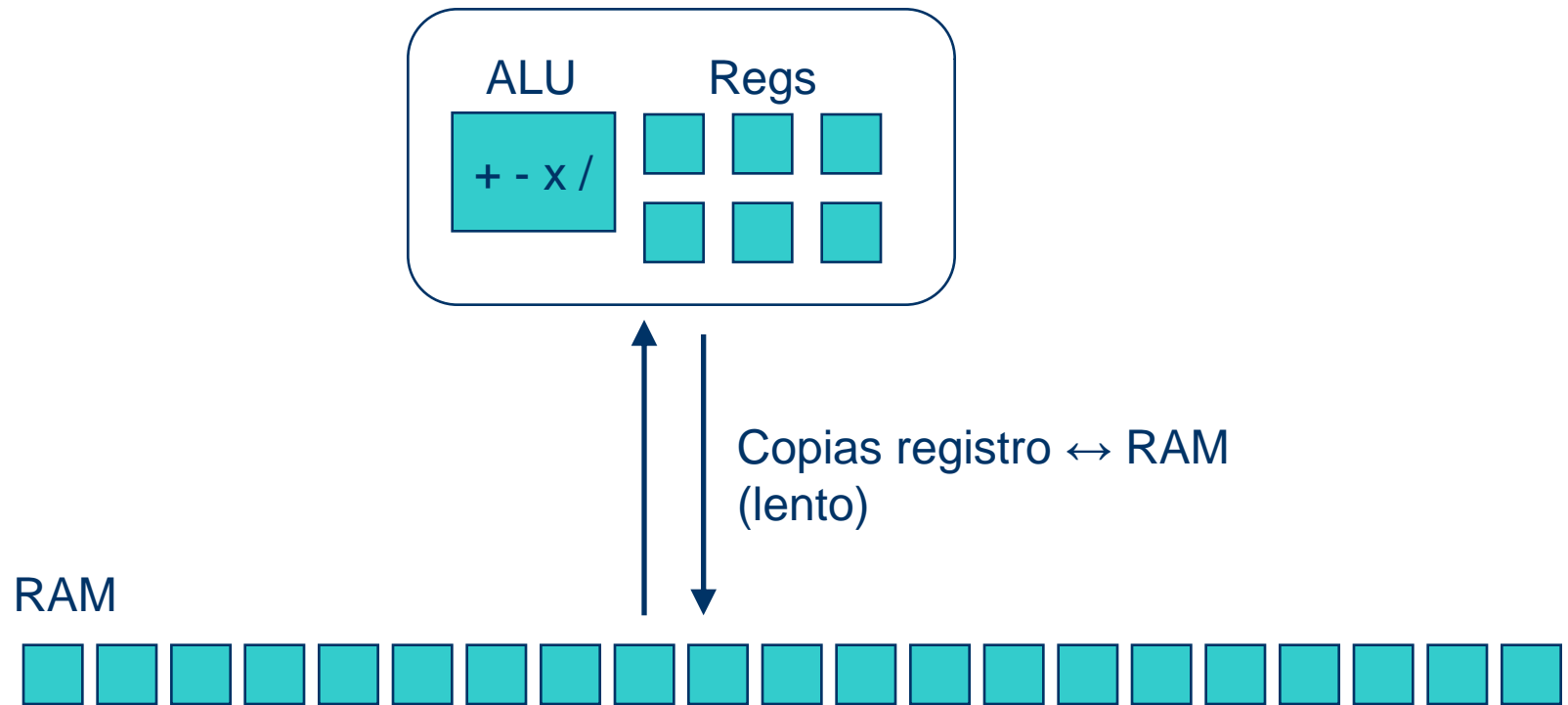
# Memoria y procesador

- El procesador ejecuta programas, que son secuencias de instrucciones que operan sobre ciertos datos
- Las instrucciones y los datos se guardan en la memoria RAM
- La memoria RAM es un conjunto de celdas consecutivas que almacenan bytes
- 1 byte = 8 bits => cabe 1 carácter

# Memoria y procesador

- RAM tiene instrucciones + datos
- Procesador tiene:
  - Registros (memoria interna pequeña, instrucciones y datos)
  - ALU (unidad aritmética-lógica), hace cálculos sobre los registros ejecutando instrucciones
- El procesador se comunica con la memoria RAM para:
  - Copiar un valor de la RAM a un registro
  - Copiar un valor de un registro a la RAM
  - Copias reg→RAM y RAM→reg también son instrucciones

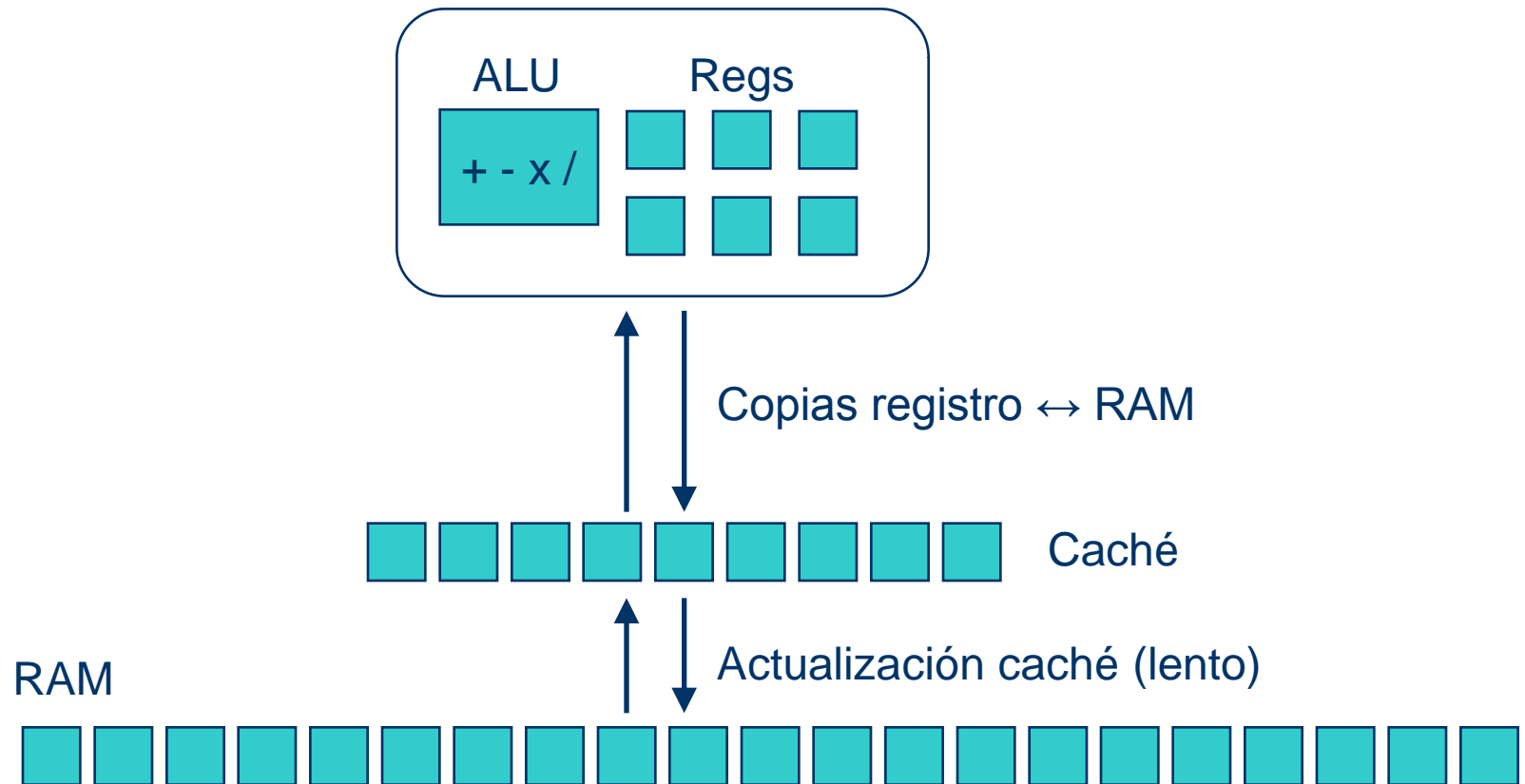
# Memoria y procesador



# Memoria y procesador

- Usualmente los procesadores tienen una memoria caché
- La memoria caché es un duplicado de las zonas de la RAM que se usan frecuentemente
- A veces dispositivos externos al procesador modifican la RAM (ej: disco duro)
- Cuando se modifica la RAM externamente, se elimina la zona del caché (el caché no se equivoca porque modifiquen la RAM)
- Los datos de la RAM (incluyendo la caché) tienen dirección de memoria, los registros no tienen

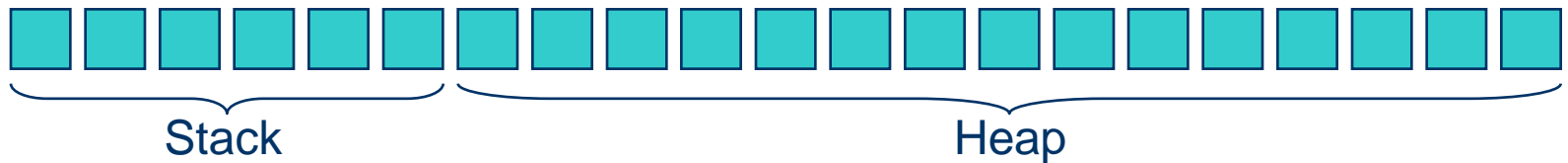
# Memoria y procesador



# Memoria y procesador

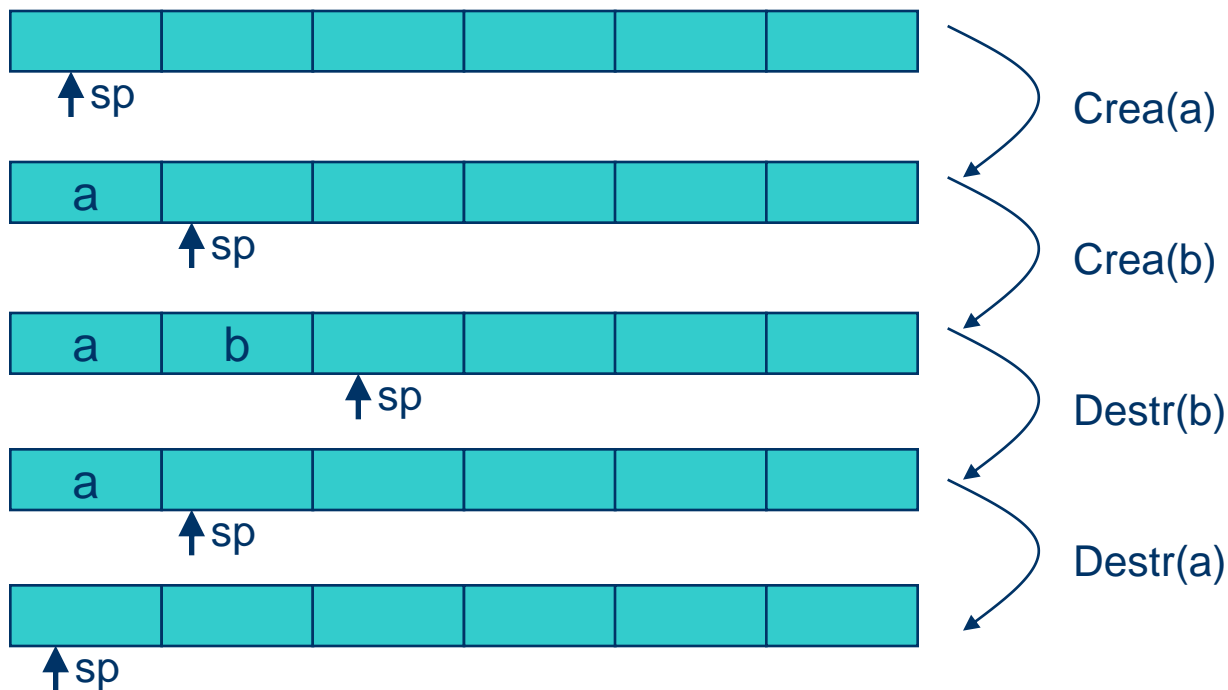
- Cuando se declara una variable, se asigna un trozo de memoria para ella
- La memoria de datos de los programas se divide en dos grandes bloques: el stack y el heap
- El stack se usa para asignar trozos de memoria a las variables del programa que tienen un tamaño fijo
- El heap se usa para asignar trozos de memoria a objetos de tamaño fijo o variable (mem. dinámica)

RAM de datos de programa



# Memoria y procesador

- El stack: Las variables de tamaño fijo se crean en un orden y se destruyen en el contrario





# Memoria y procesador

- Ejemplo 2 stack:

```
{  
    int a; // crea(a)  
    // stack: a  
    {  
        int b; // crea(b)  
        // stack: a, b  
    } // destr(b)  
    // stack: a  
    {  
        int c; // crea(c)  
        // stack: a, c  
    } // destr(c)  
    // stack: a  
} // destr(a)  
// stack:
```

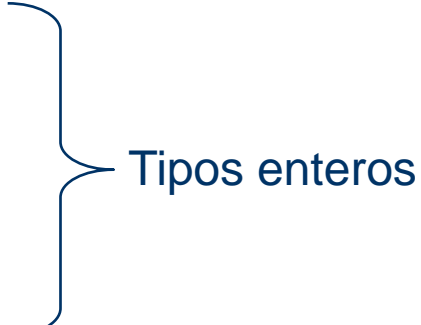
# Memoria y procesador

- En el heap, se piden trozos de memoria que existen hasta que son explícitamente liberados
- La memoria se puede pedir y liberar en cualquier orden y momento (memoria dinámica)
- Pedirla en el heap es más lento (hay que buscar)
- Ejemplo:

```
trozo = malloc(10); // Se pide un trozo de 10 bytes
...
...
free(trozo); // Se libera el trozo
```

# Tipos básicos

- Los tipos básicos son:

- char (carácter)
  - short
  - long
  - int
- 
- A blue bracket on the right side of the list groups the types char, short, long, and int. The text "Tipos enteros" is positioned to the right of the bracket.
- Tipos enteros

- 
- float
  - double
- 
- A blue bracket on the right side of the list groups the types float and double. The text "Tipos decimales" is positioned to the right of the bracket.
- Tipos decimales

# Tipos básicos

- Declaración (parten con cualquier valor):

```
{  
    int a, b;  
    a=2;  
    b=a+5;  
}
```

- Declaración e inicialización simultáneas:

```
{  
    int a=2, b=7;  
}
```

# Tipos básicos

- Declaración (parten con cualquier valor):

```
{  
    double a, b;  
    a=2.0; // Dato decimal lleva siempre punto  
    b=a+5.0;  
}
```

- Declaración e inicialización simultáneas:

```
{  
    double a=2.0, b=7.0;  
}
```

# Tipos básicos

- Declaración (parten con cualquier valor):

```
{  
    char a, b;  
    a='h'; // 'h' = 104 (porque i en ASCII es 104)  
    b=a+1; // es 'i' = 105  
}
```

- Declaración e inicialización simultáneas:

```
{  
    char a='h', b='i';  
}
```

# Tipos básicos

- ANSI-C asegura rangos para cada tipo de una forma muy poco estricta
  - char: 1 byte
    - rango {0, ..., 255} o rango {-127, ..., 126}
  - short:  $\geq 2$  bytes, rango {-32768, ..., 32767}
  - long:  $\geq 4$  bytes, rango {-16mill, ..., 16mill}
  - int  $\geq$  short (es el tipo + rápido)
  - float:  $\geq 4$  bytes, rango no definido
  - double  $\geq$  float, rango no definido

# Tipos básicos

- Casting

- Se denomina casting (moldeado) a la transformación de un tipo en otro
- Ejemplo:

```
{  
    double a = 2.3;  
    int b;  
    float c;  
    b=(int) a; // Se pierden los decimales, ahora b = 2  
    c=(float) a; // float tiene menos rango que double  
}
```



# Tipos básicos

- Operadores:

- Operador unario: +3 , -3
- Operadores binarios: +, -, \*, /
- Operador de asignación: =, devuelve valor

`a=b=c; // Es lo mismo que a=(b=c);`

- Hay otros operadores abreviados, ej:

`a+=2; // es lo mismo que a=a+2;`

`a/=2; // es lo mismo que a=a/2;`

`++a; // es lo mismo que a=a+1; (incremento)`

`--a;`

`a++; // a se incrementa al final de la instrucción`

# Tipos básicos

- La división funciona distinto con enteros y decimales
  - Decimales:  $5.0 / 2.0 = 2.5$
  - Enteros:  $5 / 2 = 2$  // La parte entera  
 $5 \% 2 = 1$  // El resto
- Regla: el resultado de una operación es el tipo más general de los argumentos
  - entero\*entero=entero
  - entero\*decimal=decimal
  - Etc.

# Tipos básicos

- Los números se pueden inicializar en base 10, base 8 o base 16
- Hay que tener cuidado con esto:
  - Prefijo 0 => base 8
  - Prefijo 0x => base 16

```
int a = 8; // a vale "8 en base 10"
```

```
int b = 010; // b vale "10 en base 8" = "8 en base 10"
```

```
int c = 0xF; // c vale "F en base 16" = "15 en base 10"
```

# Tipos básicos

- Tipos signed y unsigned
  - “char” puede tener rango  $\{-128 \dots 127\}$  o  $\{0 \dots 255\}$
  - “unsigned char” tiene rango  $\{0 \dots 255\}$
  - “signed char” tiene rango  $\{-128 \dots 127\}$
  - “short”, “int” y “long” son siempre signed
  - Hay también:
    - “unsigned short”
    - “unsigned int”
    - “unsigned long”

# Tipos básicos

- Tipos const

- Se usa la palabra “const” para indicar que la variable no debe ser modificada por el programa
- Ejemplo:

```
{  
    int a;  
    const int b=10;  
    a = 30;  
    b = a + 1; // ERROR: No deja compilar  
}
```

# Funciones

- Una función es un trozo de código que recibe varios argumentos y devuelve un valor
- Una función tiene:
  - Una declaración, para que se sepa cómo se usa
  - Una definición que indica qué es lo que hace
  - Llamadas, que son las veces que se usa la función

# Funciones

- Ejemplo:

```
double multiplica(double a, double b); // Declaración
```

```
...
```

```
double multiplica(double a, double b) // Definición
```

```
{
```

```
    return a*b; // El valor que devuelve la función
```

```
}
```

```
...
```

```
c = multiplica(3, 4); // Llamada, c va a valer 12
```

```
// La llamada hace que se ejecute la definición
```

# Funciones

- Ejemplo 2:

```
double negativo(double a); // Declaración
```

```
...
```

```
double negativo(double a) // Definición
```

```
{
```

```
    a=-a;
```

```
    return a; // El valor que devuelve la función
```

```
}
```

```
...
```

```
c = negativo(3); // Llamada, c va a valer -3
```

```
// La llamada hace que se ejecute la definición
```



# Funciones

- El tipo “void” indica que el argumento no existe (se usa en la declaración)
- Ejemplo:

```
int numeroAlAzar(void); // Decl: No recibe argumento  
c=numeroAlAzar(); // Llamada
```

```
void memoriza(int a); // Decl: No devuelve valor  
memoriza(3); // Llamada
```

```
void tarea(void); // Decl: No recibe ni devuelve  
tarea(); // Llamada
```

# Funciones

- Programación estructurada:
  - Dividir tareas grandes en pequeñas
  - Tareas pequeñas son funciones pequeñas
  - Tareas grandes son funciones que llaman otras funciones
  - Reemplazada + o - por “programación orientada a objetos”

```
void tarea(void) // C fue creado + o - para esto:
{
    subtarea1();
    subtarea2();
    subtarea3();
    return;
}
```

# Control de flujo

- Control de flujo: permite que la ejecución deje de ser estrictamente secuencial
- Para eso se evalúan condiciones lógicas
- En C:
  - “0” significa falso
  - “Número distinto de cero” significa verdadero
  - Operadores lógicos

# Control de flujo

- Operadores lógicos: devuelven 0 o 1
  - Not !:
    - “!0” vale 1
    - “!1” vale 0
    - “!2” vale 0
  - And &&:
    - “0 && 0” vale 0
    - “1 && 2.5” vale 1
  - Or ||

# Control de flujo

- Comparaciones: devuelven 0 o 1
  - Mayor que: >
  - Menor que: <
  - Mayor o igual que: >=
  - Menor o igual que: <=
  - Distinto a: !=
  - Igual a: == (tiene 2 signos igual)

# Control de flujo

- If / else

- If (condicion) instruccion1; else instruccion2;
- Si condicion es cierta se ejecuta instruccion1, sino se ejecuta instruccion2
- Ejemplo:

```
double absoluto(double x)
{
    if (x >= 0)
        return x;
    else
        return -x;
}
```

# Control de flujo

- Varias instrucciones pueden agruparse usando llaves
  - Ejemplo:

```
double absoluto(double x)
{
    if (x >= 0)
        return x;
    else
    {
        x = -x;
        return x;
    }
}
```

# Control de flujo

- While: Se ejecuta la instrucción o el bloque de instrucciones (se itera) mientras la condición sea cierta:

```
while (a < 30)
{
    a=a+1;
    b=b*a;
}
```



# Control de flujo

- For: Se parece a while, pero tiene un valor inicial, una condición para iterar y una actualización separados por ;

```
for (a=0; a < 30; a++)  
    b=b*a;
```

# Control de flujo

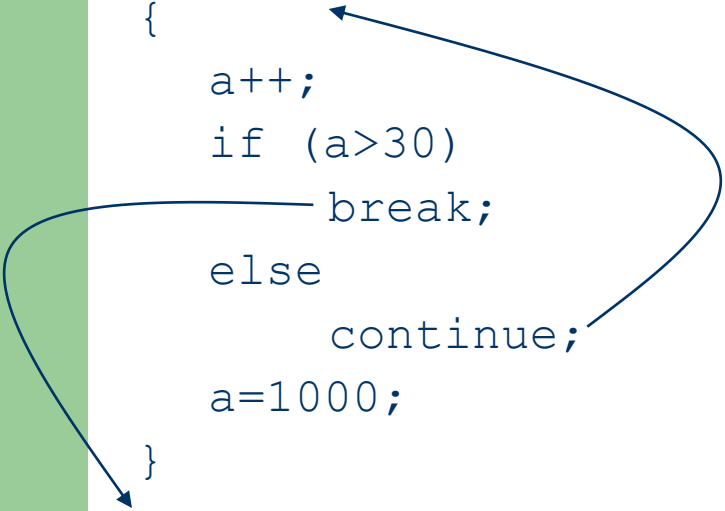
- do – while: se parece a while, pero se ejecuta siempre al menos una vez (se evalúa al final del ciclo)

```
do
{
    a++;
    b*=a;
} while (a<30);
```

# Control de flujo

- Continue y break: permiten volver al inicio del ciclo o salir del ciclo

```
while (1)
{
    a++;
    if (a>30)
        break;
    else
        continue;
    a=1000;
}
```



# Control de flujo

- Goto: es más general que continue y break

```
if (a > 10)
```

```
    goto chao;
```

```
a=0;
```

```
chao:
```

- Tiene mala fama porque

- Antes de la programación estructurada, los programas no tenían funciones y estaban llenos de gotos (ilegible)
- Se supone que el usar funciones + while + for + break + continue hace innecesario el uso de goto
- Sólo a veces es recomendable

# Control de flujo

- Switch: permite evaluar distintos casos de forma rápida, sólo para tipos enteros

```
switch(a)
{
    case 0: // Se salta acá cuando a==0
        b=1000;
        break;
    case 1: // Se salta acá cuando a==1
        b=5000;
        break;
    default:
        b=0;
}
```

# Control de flujo

- Peligro: hay que asegurarse que las variables se inicialicen (en Java esto se verifica siempre)

```
{  
    int a;  
    ...  
    if (b < 4)  
    {  
        ...  
        a=3;  
    }  
    ...  
    b=a;  
}
```

# Precompilador y bibliotecas

- A veces un programa es muy grande y conviene tener varios archivos .c

dist.c

```
int dist(int x, int y)
{
    if (x>y)
        return x-y;
    else
        return y-x;
}
```

programa.c

```
int dist(int x, int y);

int main(void)
{
    int a, b;
    a = dist(4,5);
    b = dist(a, 3)
    return b;
}
```

# Precompilador y bibliotecas

- Un archivo usa las funciones que contiene el otro
- Para eso, basta que un archivo conozca la declaración de la función que va a usar (no requiere la definición)
- Archivo .h (header) es un archivo que contiene declaraciones de un archivo .c



# Precompilador y bibliotecas

dist.c

```
int dist(int x, int y)
{
    if (x>y)
        return x-y;
    else
        return y-x;
}
```

dist.h

```
int dist(int x, int y);
```

programa.c

```
#include "dist.h"

int algo(void)
{
    int a, b;
    a = dist(4,5);
    b = dist(a, 3)
    return b;
}
```

# Precompilador y bibliotecas

- De esta manera, un par {archivo.c, archivo.h} puede ser usado por otros archivos .c sin problemas
- “#include” es una instrucción que se ejecuta antes de compilar (modifica el archivo antes de compilar)
- Las líneas que empiezan con “#” son directivas de preprocesador
- Son instrucciones para modificar el archivo antes de compilarlo

# Precompilador y bibliotecas

- Otra directiva muy usada es #define

```
// Se usa #define para definir macros
// Los nombres de macros van en mayúscula

#define CINCO 5
#define SUMA(X,Y)  ( (X) + (Y) )

int algo(void)
{
    int a=CINCO, b;
    b = SUMA(X,Y);
    return b;
}

/* Los comentarios también son como directivas */
```

# Precompilador y bibliotecas

- Se puede usar #undef para deshacer un define
- Se puede usar además #ifdef, #elif, #else, #endif

```
#define USA_DOUBLE // Esta línea se puede comentar
int areaCirculo(double r)
{
    #ifdef USA_DOUBLE
    double Pi=3.14159265358979323846;
    #else
    float Pi=3.1415926;
    #endif
    return Pi*r*r;
}
```

# Arreglos

- Los arreglos son elementos consecutivos del mismo tipo, la cantidad es fija (así se puede usar el stack)
- Ejemplo:

```
{  
    int a[3];  
    a[0]=1;  
    a[1]=2;  
    a[2]=4;  
}  
  
{  
    int a[3] = {1, 3, 4};  
}
```

# Arreglos

- La utilidad es que se pueden mezclar con “for”
- Esto permite tener un código compacto
- Ojo: `int a[10];` => `a[0]` hasta `a[9]` (10 en total)

```
{  
    int a[100];  
    for (i=0; i<100; i++) // i desde 0 hasta 99  
        a[i] = i*i;  
}
```

# Arreglos

- Los strings en C son arreglos de caracteres terminados con el carácter nulo (ASCII 0)

```
{  
    char a[5]="Hola";  
    char b[5]={ 'h', 'o', 'l', 'a', 0}  
    // a y b son el mismo string  
}
```

- Para un string de 4 letras se requieren 5 espacios

# Arreglos

- Hay arreglos multidimensionales, deben tener una cantidad fija de elementos en cada dimensión

```
{  
    int arr[10][20][30];  
    // Orden: arr[0][0][0], arr[0][0][1], etc.  
    int i, j, k;  
    for (i=0; i<10; i++)  
        for (j=0; j<20; j++)  
            for (k=0; k<30; k++)  
                arr[i][j][k]=i+j+k;  
}
```



# Arreglos

- Peligro: hay que tener cuidado para no salirse de los arreglos, o el programa va a funcionar de un modo inesperado
- No se verifican los bordes del arreglo como en Java

```
{  
    int arr[10]  
    for (i=0; i<30; i++)  
        arr[i] = 0; // Algo raro va a pasar  
}
```

# Punteros

- Un puntero es una variable que contiene una dirección de memoria (de la RAM)
- Puede guardar direcciones del stack y del heap
- Existen los operadores & y \*:
  - & (dirección): devuelve la dirección de memoria en la cual se guarda una variable
  - \* (indirección): accede al contenido que está en una cierta dirección de memoria
  - No puede usarse & con variables de tipo register

# Punteros

- Ejemplo:

```
{  
    int i, *p; // p es un puntero  
    i = 10;  
    p = &i;   // p contiene la dirección de  
              // memoria donde está i  
    *p = 20;  // Se modifica el contenido de la  
              // dirección p de memoria  
    // Ahora i vale 20  
}
```

# Punteros

- Peligro: error si se usa la dirección de una variable del stack que ya no existe (en Java no hay & ni \*)

```
{
    int *p;
    {
        int i;
        p=&i;
    }
    *p=10; // Incorrecto, pero igual compila
}
```

# Punteros

- Hay varios tipos de punteros a distintos tipos
  - `char *a`
  - `int a*`
  - `double *a`
  - `void *` (puntero genérico)
  - Etc.
- Existe la dirección especial NULL (=no válida) que es el inicio del stack. NULL es igual a `(void *)0`
- Hacer “\*NULL” hace que el programa se caiga de forma controlada (null pointer exception)

# Punteros

- Se puede hacer suma y resta con los punteros
  - Al sumarle 1 al puntero, avanza varios bytes en la memoria
  - La cantidad que avanza depende del tipo:
    - 1 byte para char
    - 2 bytes para short
    - Etc.
    - A los punteros de tipo “void \*” no se les puede sumar ni restar
- El tipo “void \*” es un tipo genérico al que se le puede hacer casting a cualquier tipo de puntero

# Punteros

```
{  
    int arr[10], *p=NULL;  
    p = arr; // Casting a puntero  
    p = &arr[0]; // Lo mismo  
    *(p+0); // Es igual a arr[0]  
    *(p+1); // Es igual a arr[1]  
    p=p+5; // p avanza 5 casillas "int"  
    *p; // Es igual a arr[5]  
    // Se pueden usar corchetes: *(p+i) = p[i]  
    p[0]; // Es igual a arr[5]  
    p[1]; // Es igual a arr[6]  
}
```

# Punteros

```
{  
    int arr[10][2], *p=NULL;  
    p = arr; // Casting a puntero  
    p = &arr[0][0]; // Lo mismo  
    *(p+0); // Es igual a arr[0][0]  
    *(p+1); // Es igual a arr[0][1]  
    p+=5; // p avanza 5 casillas "int"  
    *p; // Es igual a arr[2][0]  
    // Se pueden usar corchetes: *(p+i) = p[i]  
    p[0]; // Es igual a arr[2][0]  
    p[1]; // Es igual a arr[2][1]  
}
```



# Punteros

- Funciones que reciben punteros: pueden modificar el contenido de la memoria

```
void duplica(int *a)
{
    *a *= 2;
}

void programa(void)
{
    int a = 4;
    duplica(&a); // a puede modificarse
    // Ahora a vale 8
}
```

# Punteros

- Funciones que reciben punteros:
  - Uso de const para evitar modificación

```
int loMismo(const int *a) // No se puede modificar
{
    return *a;
}

void programa(void)
{
    int a = 4, b;
    b = loMismo(&a); // a no puede modificarse
    // Ahora b vale 4
}
```

# Punteros

- Funciones que reciben punteros

```
void haceCero(int *arr, int n)
{
    int i;
    for (i=0; i<n; i++)
        arr[i]=0;
}

void programa(void)
{
    int a[1000];
    haceCero(a, 1000); // Casting a puntero
}
```

# Memoria dinámica

- Forma de pedir y liberar memoria:

```
#include <stdlib.h> // header estándar con < >

void programa(void)
{
    int i, *a;
    // sizeof(int) es el tamaño de un int en bytes
    a = (int *)malloc(sizeof(int)*1000) // pedir
    for (i=0; i<1000; i++)
        a[i]=i;
    free(a); // liberar
}
```

# Memoria dinámica

- Peligro: usar memoria ya liberada (en Java no se puede pedir que se libere)

```
#include <stdlib.h> // header estándar con < >

void programa(void)
{
    int i, *a;
    a = (int *)malloc(sizeof(int)*1000) // pedir
    free(a); // liberar
    for (i=0; i<1000; i++)
        a[i]=i; // Incorrecto, igual compila
}
```

# Memoria dinámica

- Si malloc() no encontró un bloque de memoria lo suficientemente grande, devuelve NULL
- Es poco probable que esto ocurra, generalmente no se pregunta si devolvió NULL (aunque es lo correcto)
- Si devuelve NULL y se hace un \*NULL, el programa se va a caer de una forma “controlada”

# Bibliotecas estándar

- Hay varias bibliotecas estándar que se pueden cargar así:
  - `#include <header.h> // < >` para bibliotecas estándar
- Se mencionan las siguientes:
  - `stdlib.h`: `malloc( )`, `free( )`, `system("del *.bmp");`
  - `stdio.h`: `printf( )`, `scanf( )`, `fopen( )`, `fclose( )`, etc.
  - `string.h`: `strcpy( )`, `strcat( )`, `strlen( )`, `memcpy( )`, etc.
  - `math.h`: `sqrt( )`, `sin( )`, `cos( )`, `atan2( )`, `pow( )`, `floor( )`, etc.

# Bibliotecas estándar

- printf (escribir en pantalla) y scanf (leer teclado):

```
#include <stdio.h> // Para leer / escribir datos

void programa(void)
{
    int e;
    double r;
    printf("Ingrese un entero y un decimal:");
    scanf("%d%lf", &e, &r);
    printf("Escribió %d %lf\n", e, r);
}
```



# Bibliotecas estándar

- Especificadores de formato
  - %d : int
  - %ld: long
  - %f: float
  - %lf: double
  - %c: char
- Peligro: Es peligroso usar printf( ) y scanf( ), ya que un error en el uso de los especificadores de formato produce resultados inesperados

# Estructuras

- Son agrupaciones de variables
- Los elementos se acceden con el operador . (punto)

```
struct complejo {double re, double im};

void programa(void)
{
    struct complejo z; // Creada en el stack
    z.re=0;
    z.im=0;
}
```

# Estructuras

- Si se pasan como argumento a una función, se copian todos sus componentes
- Conviene pasar un puntero, se usa el operador ->

```
#include <math.h>

struct complejo {double re, double im};

double modulo(const struct complejo *z)
{
    return sqrt( z->re*z->re + z->im*z->im );
}

// x->y es igual a (*x).y
```

# Estructuras

- Se puede pedir memoria para las estructuras

```
struct complejo {double re, double im};

void programa(void)
{
    int i;
    struct complejo *arr;
    arr=malloc(sizeof(struct complejo)*10);
    for (i=0; i<10; i++)
    {
        arr[i].re=0;
        arr[i].im=0;
    }
    free(arr);
}
```

# Variables globales y scope

- Una variable global se crea cuando parte el programa y existe hasta que se termina
- Conviene inicializarla siempre en la declaración

```
int numero = 50

int funcion_uno(void)
{
    return numero*2;
}

int funcion_dos(void)
{
    return numero*1000;
}
```

# Variables globales y scope

- No siempre es conveniente que una variable externa se vea desde todas partes, conviene acotar su visibilidad
  - a algunos archivos .c, o
  - a algunas funciones
- La visibilidad de una variable es llamada comúnmente “scope”

# Variables globales y scope

- Acotar su visibilidad a algunos archivos:

Arch1.cpp

```
int num = 50;  
// La variable vive acá
```

Arch1.cpp

```
extern int num;  
// La variable no vive acá  
// Ahora se puede usar
```

Arch3.cpp

```
void funcion(void)  
{  
    int num;  
    num=2; // Es otro  
}
```

# Variables globales y scope

- Acotar su visibilidad a un archivo:

Arch1.cpp

```
static int num = 1;
// La variable no se ve desde otro archivo

int funcion_uno(void)
{
    num=num*2;
    return num;
}
```



# Variables globales y scope

- Acotar su visibilidad a una función:

Arch1.cpp

```
int funcion_uno(void)
{
    static int num = 1; // Sólo se ve acá dentro
    num=num*2;
    return num;
}
// Esta función va a devolver distintos valores
// Va a devolver 2, 4, 8, 16, etc
```

# Variables globales y scope

- Las variables también pueden tener una visibilidad limitada si hay varias variables con el mismo nombre dentro de otras llaves

```
int funcion (void)
{
    int a=1;
    {
        int a=3; // Es otra variable a
    }
    // Acá a vale 1
}
```

# Temas recomendados

- Operadores de bits, campos de bits
- Uniones
- Asignación condicional ? :
- Entrada/salida, archivos (stdio.h)
- Trabajo con strings (string.h)
- Listas (conviene más verlas en C++)
- Tipos register y volatile, signal.h (raros, para trabajar con hardware en bajo nivel)
- stdarg.h (para hacer funciones que reciben argumentos variables, como printf)