Introducción al lenguaje C

Herramientas de programación para procesamiento de señales

Índice

- Conceptos básicos
- Conceptos avanzados

Índice (primera parte)

- Conceptos básicos
  - Estructura de un programa C
  - Proceso de compilación
  - Tipo de datos
  - Operadores
  - Control de flujo
  - Funciones
  - Variables
  - Arreglos

Índice (segunda parte)

- Conceptos avanzados
  - Programación modular
  - Punteros
  - Relación entre arreglos y punteros
  - String: cadenas de caracteres
  - Estructuras
  - Uniones

Clasificación de lenguajes

- Nivel
  - **alto nivel: C**, C++, Pascal, FORTRAN
  - bajo nivel: ensamblador, código de máquina.
- Compilación
  - **compilados: C**, C++, Pascal, FORTRAN
  - pre-compilados: Java (bytecode)
  - interpretados: Matlab, Perl, bash, javascript
- Otros: Tipeo (fuerte, débil), etc.

Primer programa: primer.c

primer.c

```c
#include <stdio.h>

void main(void) {
    printf("Hola mundo!!\n");
}
```

`#include<stdio.h>`- archivo de encabezado (stdlib.h, unistd.h, math.h).

`main()` – main es una función, el punto de entrada al programa.

`{}` - paréntesis, son para indicar el comienzo y el fin de un bloque, como el cuerpo de una función

`printf()`- es el llamado a la función imprimir...

Otro programa: segundo.c

segundo.c

```
#include <stdio.h>
#define NUM 2
void main(void) {
   int a, b, suma;
   a = NUM;
   b = 2;
   suma = a + b;
   printf("suma vale %d", suma);
}
```

`#define NUM 2`– directiva al preprocesador, sustituye NUM por 2.

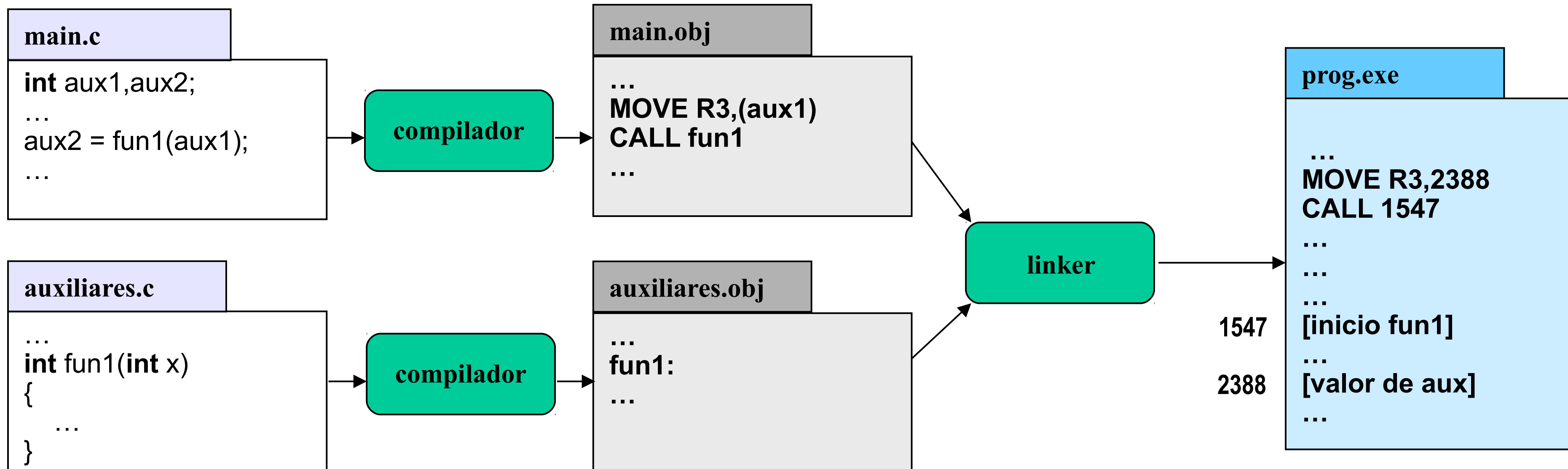
`int` `a, b, suma`- declara las variables enteras: a, b y suma.

`a = 1; ...` –define las variables y les asigna valor.

`suma = a + b;` realiza la suma.

`printf()`- imprime el resultado.

Proceso de compilación



Atributos de los objetos

Atributo	Descripción
Tipo	char, int, unsigned int, etc.
Nombre	Identificador para acceder el objeto.
Valor	Datos contenidos en el objeto.
Dirección	La ubicación en memoria donde reside el objeto.
Alcance	El código fuente donde el nombre del objeto es reconocido.
Tiempo de vida	Cuando el objeto está disponible (dependiendo de cuando el objeto es creado y destruido).

Declaración/Definición

- Declaración: especifica al compilador que un variable o función existe.

```
int optimo(int a, int b);
int i, j;
int aux;
```

- Definición: la variable o función misma.

```
int optimo(int a, int b){
        int aux = a;
        /* sigue... */
        return aux;
}
```

Variables: tipos de variables

- Tipo de variables fundamentales:
  - enteros: `int`, `char`.
  - flotantes: `float`, `double`.
- Modificadores (adjetivo):
  - `short`, `long`, `unsigned`, `signed`
  - No todas las combinaciones de tipos y modificadores son válidas.

Tipos de datos enteros

Tipo de dato		Tamaño	Rango
unsigned	char	8 bits	0 a 255
	short int	16 bits	0 a 65,535
	int	16 bits o 32 bits	Idem unsigned short int; Idem unsigned long int.
	long int	32 bits	0 a 4,294,967,295
signed	char	8 bits	-128 a +127
	short int	16 bits	-32,768 a +32,767
	int	16 bits o 32 bits	Idem signed short int; Idem signed long int.
	long int	32 bits	-2,147,483,648 a +2,147,483,647

- Atención:
  - `int` tamaño del bus del CPU.
  - `short int` menor o igual a `int`.
  - `long int` mayor o igual a `int`.

Operadores: asignación

- Operadores: =, +=, -=, *=, /=
  - Ejemplos:

```c
int i = 1;
i += 2;            //equivalente a: i = i + 2;
i *= 10;           //equivalente a: i = i * 10;
```

Operadores: aritméticos

- Operadores: +, -, *, /, %
  - Ejemplos:

```
int a,b,n;
a = 10;
b = 3;
n = a%b;// resultado: n = 1;
```

- Operadores pos, pre-incremento: ++, --
  - Ejemplos:

```
int i = 10;
i++;    // equivalente a: i = i + 1;
```

Operadores: relacionales

- Operadores: ==, != , <, <=, >, >=
  - Ejemplo:

```
int x, y, b;
x = 10;
y = 3;
b = (x==y);     // resultado: b = 0;
```

  - Observación:
    - No existe el tipo boolean: resultado entero
      - false: `0`
      - true: `!0`
    - `a = b` no es lo mismo que `a == b`

Operadores: lógicos

- Operadores: `&&` (AND), `||` (OR), `!` (NOT).
  - Ejemplos:

```
int x, b;
// sigue: x = ...
b = (x < 0) && (x >10);
```

  - Observación:
    - Expresiones con operadores lógicos pueden ser corto circuitadas (algunos compiladores).
      Por ejemplo: si `x < 0` es falso, toda la expresión será falsa.

Operadores: manipulación de bits

- Permiten la manipulación de bits individuales de una variable.
- Operadores:
  - | (OR), & (AND), ~ (NOT), ^ (XOR), >> (RRA), << (RLA)
  - Ejemplos:

```
mask = (1<<7);        // mask = 0x08.
bits = bits | mask;   // bits = bits OR mask.
                      // setea el bit 7 de bits.
```

Operadores lógicos y “bitwise”

<table>
<tr><th>Operación</th><th>Operador lógico</th><th>Operador “bitwise “</th></tr>
<tr><td>AND</td><td>&&</td><td>&</td></tr>
<tr><td>OR</td><td>||</td><td>|</td></tr>
<tr><td>XOR</td><td>No definido</td><td>^</td></tr>
<tr><td>NOT</td><td>!</td><td>~</td></tr>
</table>

- Ejemplo
  - `(5 || !(3)) && 6 = 1`
  - `(5 | ~(3)) & 6 = 4`

Operadores: conversión de tipos

- Conversión de tipos: “casting”
- Promoción: no hay pérdida de precisión
  - Ejemplos:

```
int i = 1;
float f = i;    // sigue: x = ...
```

- Degradación: hay pérdida de precisión
  - Ejemplos:

```
float f = 4.3;  // sigue: x = ...
int i = f;
```

Control de flujo

- Secuencia
  - `{...}`
- Condicionales
  - `if-else`, `switch-case`
- Iteraciones
  - `while`, `do-while`, `for`

Control de flujo

- Condicionales: `if(cond)-else`
  - Ejemplos:

```
int x, y;
if (x < 0){
    y = x;
} else {
    y = x*10;
}
```

Control de flujo

- Condicionales: `switch-case`
  - Ejemplo:

```
char c;
//...
if (c == 'a'){
  // sentencias si 'a'
} else if (c == 'b'){
  // sentencias si 'b'
} else if (c == 'c'){
  // sentencias si 'c'
} else {
  // otras letras
}
```

```
char c;
//...
switch ( c ) {
case 'a':
   // sentencias si 'a'
   break;
case 'b':
   // sentencias si 'b'
   break;
case 'c':
   // sentencias si 'a'
   break;
default:
   // otras letras
}
```

Control de flujo

- Iteración: `while(cond)`, `do-while(cond)`
  - Ejemplos:

```
int cond;
// cond = ...; Se determina si entra a la iterac.
while (cond){
  // sentencias a repetir y modificación de cond
}

int cond; // siempre se entra una vez a la iterac.
do {
  // sentencias a repetir y modificación de cond
while (cond);
```

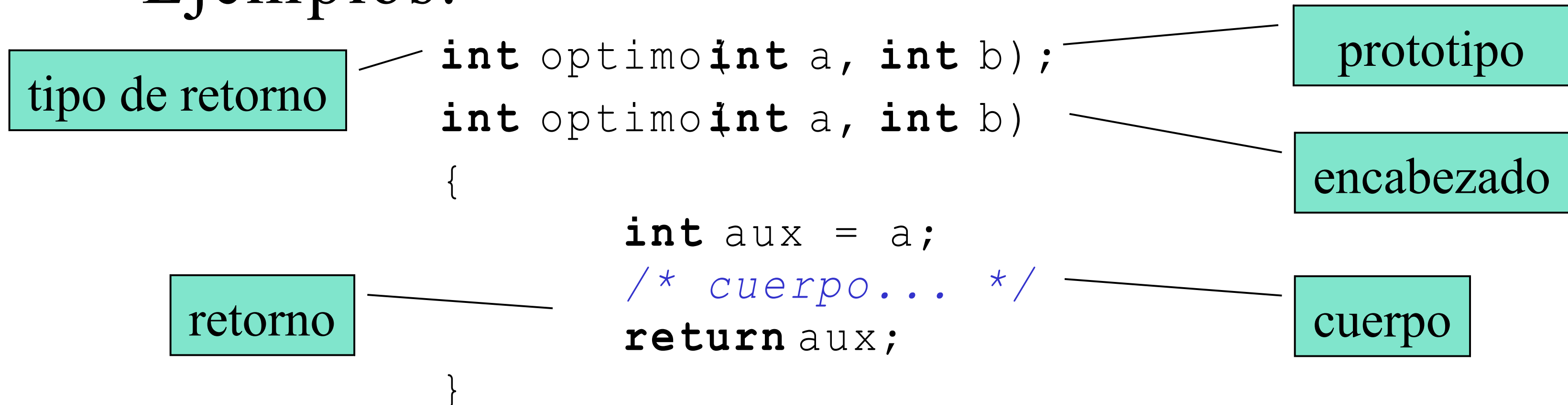
Control de flujo

- Iteración: `for(ini;cond;modif)`
  - Ejemplos:

```
int i;
// cond = ...; Se determina si entra a la iterac.
for(i = 0; i<MAX;i++){
  // sentencias a repetir MAX veces.
  // i pude ser utilizado como índice.
}
for(;;){
  // bucle infinito.
}
```

Funciones

- Declaración y definición
  - declaración: prototipo (interfaz), función abstracta.
  - definición: la función misma.
- Ejemplos:



Funciones

- Prototipo, definición y llamada
  - Ejemplos:

```
int optimo(int a, int b);

main(){
    /* codigo... */
    c = optimo(a,b);
}
int optimo(int a, int b)
{
    /* cuerpo... */
    return aux;
}
```

prototipo

llamada

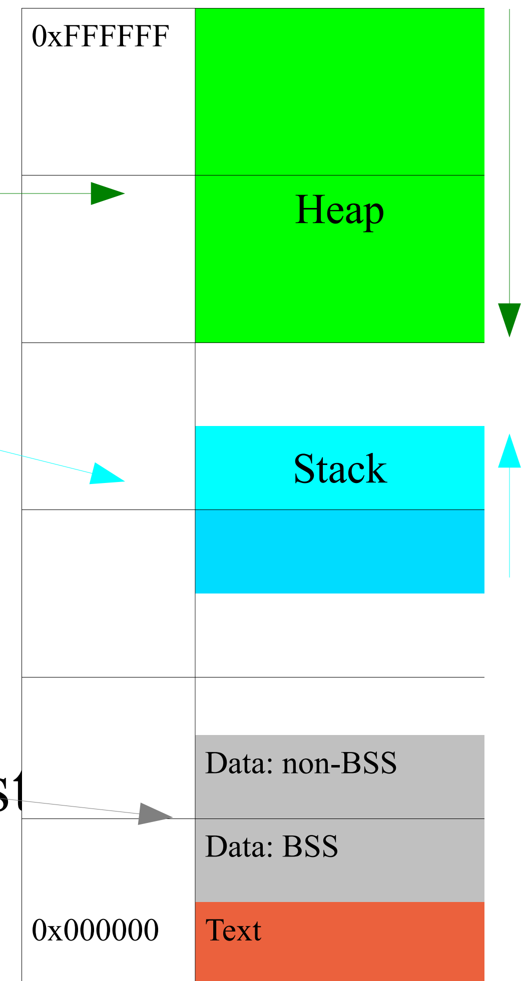
definición

Variables

- local vs. global (visibilidad o alcance)
  - local: definida dentro de una bloque (típicamente una función), alcance el propio bloque (función).
  - global: definida fuera de toda función, accesible desde todas las funciones del archivo.
- auto vs. static (vida y almacenamiento)
  - auto: dura mientras dure la función, guardada en el stack o también registros (locales por omisión).
  - static: dura siempre, guardada en un lugar fijo de memoria (globales o locales declaradas static).

Modelo de memoria

- Heap:
  - Memoria dinámica (alloc)
- Stack
  - Memoria estática (local)
  - Variables locales
  - Parámetros de funciones
- Data
  - Memoria estática (global, static, const



Variables

- "Cualificadores": `const`, `volatile`.
  - `const` - guarda la variale en un área constante de memoria y hace que la misma sea inmodificable.
  - `volatile` - indica al compilador que este valor puede ser modificado fuera del control del programa (evita optimizaciones)
  - `register` - sugiere al compilador que ubique la variable en un registro del CPU

Variables

– Ejemplos:

```
int var1;
void func(int a, int b){
    int tempo1;
    static inttempo1;
    var1 = tempo;
   /* codigo... */
}
```

global y static

local y auto

local y static

– Observaciones:

- global siempre son static.
- local puede ser auto (por omisión) o static.

Variables

main.c

```
int fun1();
int fun2();
int global;
void main(void) {
   int local = 1;
   local += fun1();
}
int fun1(){
   int local = 2;
   return (global+local);
}
```

global y static

local y auto

local y auto
diferente local de main

funciones.c

```
extern int global;
int fun2(){
   return global++;
}
```

extern: accede a variable
global de otro archivo

Variables: tipo de almacenam.

- `extern`- hace que la variable especificada acceda a la variable del mismo nombre de otro archivo (da acceso global de archivo).
- `static`- hace que la variable o función tenga alcance solamente de archivo.
  - Ejemplos:

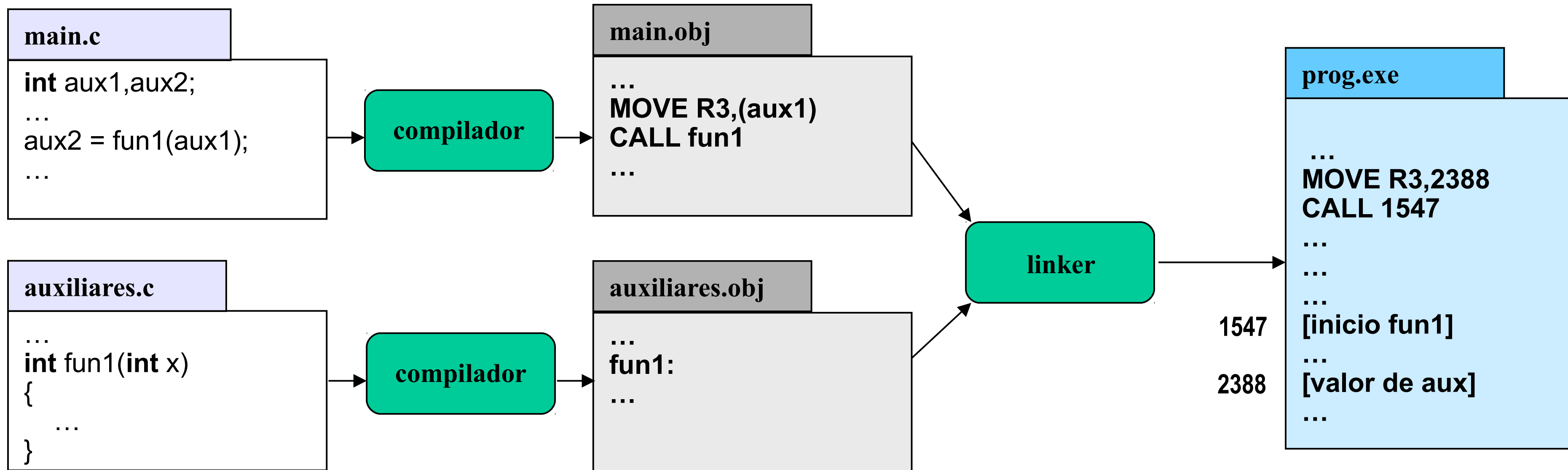
```
extern int varglobal;
static int funcion();
```

Arreglos

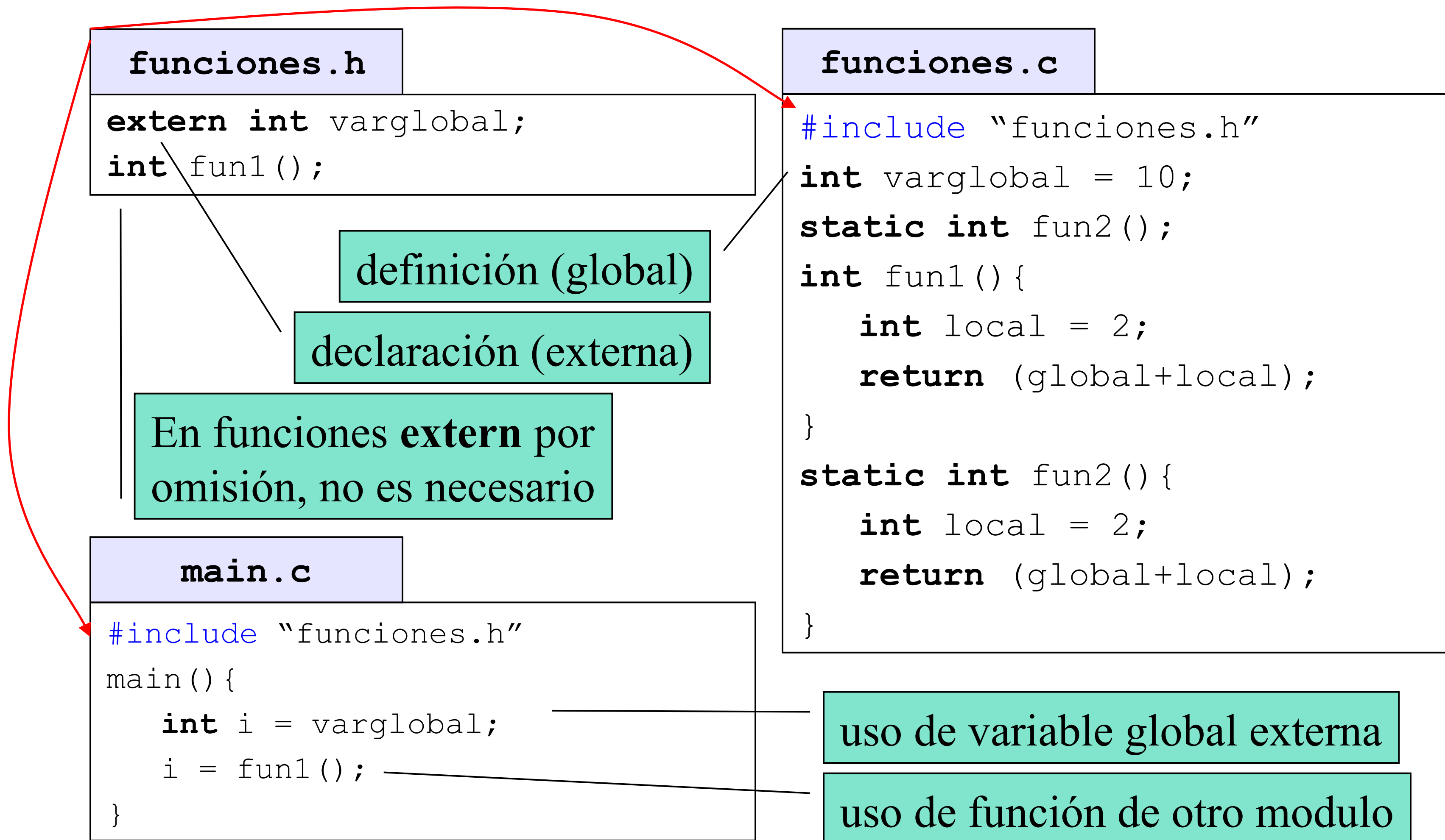
- Arreglo: colección de elementos del mismo tipo (i.e. `char`, `int`, etc.)
  - ubicados en bloques contiguos de memoria
  - referenciados con nombre único e índice.
  - Ejemplos:

```
#define TAM 10
int ai1[]= {1, 2, 3, 4};

int ai2[TAM];
ai2[0] =1;

char frase[] = "alarma";
```

Proceso de compilación



Módulo



Módulo

- archivo.c - implementación del módulo.
  - definición de variables globales, locales y de acceso archivo (static)
  - definición de funciones públicas y privadas (acceso archivo: static)
  - incluye su propio encabezado para verificación
- archivo.h - interfaz pública del módulo.
  - declaración variables externas (globales)
  - funciones públicas (no declaradas static en .c)

Preprocesador

- Sustitución de texto
  - #define X 2: Cambia etiqueta por valor
  - #include "funcion.h": Sustituye por el contenido del archivo

funciones.h

```
extern int varglobal;
int fun1();
```

main.c

```
#include "funciones.h"
main(){
   int i = varglobal;
   i = fun1();
}
```

main.i

```
extern int varglobal;
int fun1();
main(){
   int i = varglobal;
   i = fun1();
}
```

Índice (segunda parte)

- Conceptos más avanzados
  - Programación modular
  - Punteros
  - Relación entre arreglos y punteros
  - String: cadenas de caracteres
  - Estructuras
  - Uniones

Punteros

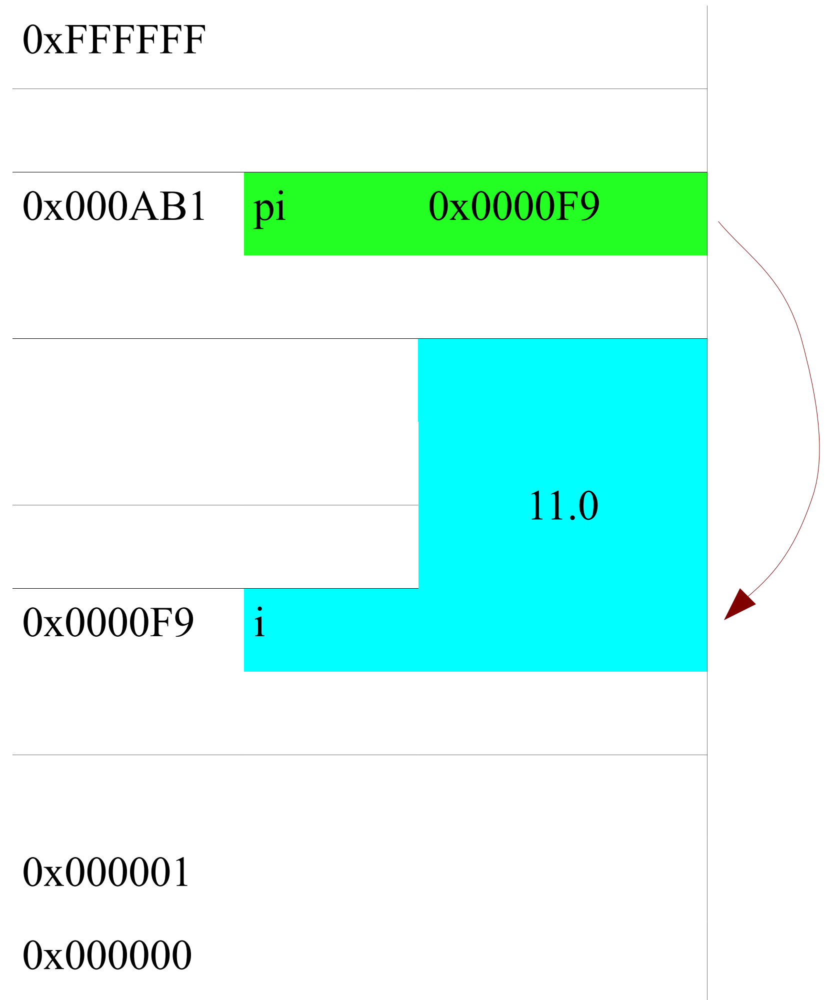
- Puntero: es una variable
  - guarda la dirección de memoria de otra variable (específica)
  - cuando se declara no guarda espacio para la variable.
- Operadores
  - * (referencia): accede al contenido de la dirección de memoria guardado por el puntero.
  - & (dereferencia): obtiene la dirección de memoria de una variable.
  - Ejemplos:

```
int i = 10;
int* pi; // equivalente a int *pi
// pi es una var. int* (puntero a int) o *pi es int
pi = &i;          // pi contiene la dirección de i
 printf("i vale: %d\n",*pi); //mostramos el valor de i
*pi = 10;// escribe 10 en la dirección guardada por pi
```

Punteros

Ejemplos

```
float i = 10.0;
float* pi; // equivalente a float *pi
   // pi es una varde tipo float* (puntero
   a float)
   // o  *pi es float
 pi = &i;  // pi contiene la dirección de i
 printf("i vale: %f\n",*pi); //mostramos el
   valor de i
 *pi = 11.0; // escribe 10 en la dirección
   guardada por pi
```



Punteros

- Pasaje por valor
  - void mod1 (int a)
  - Adentro de la función se copia a
  - Cuando se retorna a tiene el mismo valor.
- Pasaje por referencia
  - void mod2 (int *pa)
  - Se le pasa la dirección de a
  - Se puede modificar
- Ejemplo:

```
int a = 10;    // definimos un entero cualquiera
printf("a %d\n",a);         // imprimimos valor
mod1(a);   // tratamos de modificarlo
printf("a1 %d\n",a);        // imprimimos valor
mod2(&a);  // modificamos valor
printf("a2 %d\n",a);        //imprimimos valor
```

Relación entre arreglos y punteros

- El nombre de un arreglo es un puntero al inicio del bloque de memoria del arreglo

```
int ai[] = {1, 2, 3, 4, 5};
int* pi;

pi = ai;        // equivalente a pi = &ai[0];
```

- Aritmética de punteros: útil para recorridas
  - Ejemplos (continuación):

```
*pi = 10;       // equivalente a ai[0] = 10;
 pi++;
*pi = 12;       // equivalente a ai[1] = 12;
```

String: cadena de caracteres

- No está soportado directamente por C.
- Strings:
  - arreglo de caracteres
  - deben terminar en carácter nulo (`NULL`), que indica fin de `string`
- Ejemplos:

```
char* frase = {'H', 'o', 'l', 'a', '\0'}; //equivalente a:
char  frase[] = "Hola";  // compilador agrega caract. '\0'
```

- La biblioteca `string.h` provee funciones para manipulación de cadenas:
  - Ejemplos:

```
char *strcpy(char *dest, const char *src);
int strcmp(const char *s1, const char *s2);
int atoi(const char *nptr);
```

Estructuras

- Estructura
  - colección de items de diferente tipo.
- En general definiremos tipos con `typedef`
- Ejemplos:

```
char* str = "este es el menasaje";
typedef struct {
char tipo;
char payload[TAM];
} mensaje; // equivalente a:
mensaje m;
m.tipo = '0';
strcpy(m.payload,str); //¿parám. pasados correctamente?
```

Uniones

- Uniones: declaración y uso igual a las estructuras.
  - se utiliza un solo miembro
  - los miembros comparten la memoria
- Ejemplos:

```
typedef struct {
   unsigned char tipo;
   union {
      int entero;
      float flotante;
   } contenido;
} numero;
numero n;
//...
if (n.tipo == ENTERO)
n.contenido = 1;
else
n.contenido = 1.0;
```