Fiuba
Facultad de Ingeniería
Universidad de Buenos Aires

Clase 2

Contenidos

1 Punteros y memoria dinámica 3

1.1 Punteros 3
1.1.1 Declaración y sintaxis 3
1.1.2 Operador sizeof() 4
1.1.3 Aritmética 4
1.1.4 Casting 5
1.1.5 Pasaje de parámetros mediante punteros 6

1.2 Memoria dinámica 6
1.2.1 Reserva de memoria 6
1.2.2 Liberación de memoria 6

1.3 Temas adicionales de punteros 7
1.3.1 Punteros a funciones 7

2 Armando del proyecto en librerías 9

3 Proceso de construcción 10

3.1 Proceso de transformación 10
3.1.1 Precompilación (Preprocesamiento) 10
3.1.1.1 Sustitución y Control de la Precompilación 11
3.1.1.2 Constantes simbólicas predefinidas 12
3.1.1.3 Expansión de macros 13
3.1.2 Compilacion 14
3.1.2.1 Parsing(Analisis Sintactico) 14
3.1.2.2 Translation(Traducción) 14
3.1.2.3 Assembler (Ensamblado) 14
3.1.3 Link Edición 15

1 Punteros y memoria dinámica

1.1 Punteros

Un puntero es un tipo de datos cuyas instancias contienen direcciones de memoria. Esto les permite "apuntar" a otras variables. El proceso de obtener la variable ubicada en la dirección contenida en el puntero se denomina desreferenciar el puntero. Como un puntero es una variable pueden definirse punteros a punteros, punteros a punteros a punteros, etc.

Son comúnmente utilizados para implementar muchas de las estructuras de datos utilizadas corrientemente, tales como listas, árboles, pilas y colas.

El lenguaje C garantiza que el puntero de valor 0[1], denominado "puntero nulo" no apunta a ningún valor válido.

1.1.1 Declaración y sintaxis

En C se declara un puntero a variables de un tipo dado de la de la misma forma que la variables de dicho tipo anteponiendo * al nombre de la variable (o agregándole * al final del tipo).

Para obtener la dirección de una variable se utiliza el operador &, mientras que para desreferenciarlo se utiliza el operador *.

```
#include <stdio.h>

int main( int argc, char *argv[])
{
    int entero = 2;
    int *ptr;

    int *ptr1, ptr2;

    ptr = &entero;

    printf("%d\n", entero);

    printf("%d\n", *ptr );

    *ptr = 3;

    printf("%d\n", entero);


    return 0;
}

/* La salida sera
2
2
3
*/
```

[1] También puede expresarse con la constante simbólica NULL.

1.1.2 Operador sizeof()

Este operador recibe tanto variables como tipos y devuelve el tamaño en bytes que éstos ocupan.

1.1.3 Aritmética

Una característica bastante particular de C es el hecho de que permite realizar ciertas operaciones aritméticas con punteros:

- Incremento y decremento (**++**, **--**): Incrementa/decrementa el valor del puntero en tantos bytes como sea el tamaño del tipo base del mismo.
- Suma/resta con un entero (**+, -, +=, -=**): Le agrega al puntero el producto del tamaño del tipo base en bytes con la cantidad entera.
- Resta de punteros (-): devuelve la cantidad de elementos del tipo base en un array que separan a los punteros (o sea resta los valores y los divide por el tamaño del tipo base del puntero).
- Comparaciones (**==, !=, <, >, <=, >=**): Compara los valores de los punteros.

```
#include <stdio.h>

int main( int argc, char *argv[])
{

    printf("char* %d\n", sizeof(char*) );
    printf("int* %d\n", sizeof(int*) );
    printf("float* %d\n", sizeof(float*) );
    printf("double* %d\n", sizeof(double*) );

    return 0;
}
/* La salida (en una máquina de 32 bits) sera
char* 4
int* 4
float* 4
double* 4
*/
```

Ejemplo de aritmética de punteros y vectores

```
#include <stdio.h>
#include <malloc.h>

int main( int argc, char *argv[])
{
    int *vector;
    int entero;
    int *ptrback;

    /* creo un vector de 10 posiciones */
    vector = (int*)malloc(10*sizeof(int));

    ptrback = vector;

    for( entero = 1 ; entero <= 10 ; entero++ )
        *(vector++) = entero;

    /* lo hago apuntar al ppio */
    vector = ptrback;

    for( entero = 1 ; entero <= 10 ; entero++ )
        printf("%d ",*(vector++));
```

```
    vector = ptrback;

    free(vector);

    return 0;
}
```

1.1.4 Casting

A diferencia de los tipos enteros, para convertir de un tipo de puntero a otro se requiere un casting explícito. Por ejemplo para convertir del puntero a entero ***p*** al puntero a carácter ***q*** se requiere de una conversión explícita: ***q = (char*)p***.

Existe un tipo especial de puntero genérico, denominado ***void****, que no apunta a ningún tipo de datos en especial puede asignarse libremente a otros tipos de puntero. Se puede utilizar, por ejemplo, para tomar arrays de datos sin tener que conoce su tipo tal como lo hacen ***fread()*** y ***fwrite()***.

```
#include <stdio.h>
#include <malloc.h>

int main(int argc, char* argv[])
{
      /* Puntero a chars */
      char* p;
      /* Puntero a ints */
      int* q;
      /* Puntero a void */
      void* r;

      /* Reservo 10 bytes */
      r = malloc(10);

      /* Se puede convertir libremente a los otros tipos de puntero */
      p = r;
      q = r;

      /* Tambien viceversa */
      r = ++p;
      r = ++q;

      /* No puedo convertir punteros a distintos tipos entre si en otros casos */
      /*p = q;*/ /* WARNING: assignment from incompatible pointer type */
      /*q = p;*/ /* WARNING: assignment from incompatible pointer type */

      /* Excepto si utilizo un cast explicito */
      p = (char*)q;
      q = (int*)(++p);

      /* Libero la memoria */
      /*free(r);*/ /* MAL, POR QUE? */
      free(((int*)r) - 1); /* POR QUE? */

      return 0;
}
```

1.1.5 Pasaje de parámetros mediante punteros

Debido a que C no permite el pasaje de parámetros por referencia, es necesario realizar en forma explícita el pasaje de las referencias. Esto se consigue pasando como parámetro a la función un puntero al valor que se desea que la función modifique. Dentro del cuerpo de la función se desreferencia el puntero para operar sobre el valor y de esta forma se consigue le efecto del pasaje por referencia.

```
#include <stdio.h>
```

```
void swap( int *a, int *b )
{
    int c;

    c = *a;
    *a = *b;
    *b = c;
}

int main( int argc, char *argv[])
{
    int a = 2, b = 3;

    printf("%d - %d\n", a, b);
    swap( &a , &b );
    printf("%d - %d\n", a, b);

    return 0;
}
```

1.2 Memoria dinámica

Se le denomina memoria dinámica a la que ha sido reservada durante la ejecución de un programa. La principal ventaja de utilizarla es la flexibilidad que esta provee para manejar demandas solo conocidas en tiempo de ejecución[2]. También permite liberar el espacio destinado a datos que ya no van a utilizarse. Los bloques de memoria reservados están en un área conocida como heap[3].

La utilización de memoria dinámica requiere el manejo de dos procedimientos: la reserva de memoria y su liberación.

1.2.1 Reserva de memoria

Para solicitarle memoria al sistema operativo en C se utiliza la función ***malloc()***. Esta recibe como argumento la cantidad en bytes de memoria que se deseen pedir y devuelve un puntero del tipo ***void**** al primer byte de la zona reservada o un puntero nulo (NULL) si no pudo reservar la memoria por alguna razón.

1.2.2 Liberación de memoria

Toda la memoria reservada con ***malloc()*** debe ser liberada utilizando la función ***free().*** Esta función recibe como parámetro el puntero al bloque de memoria a liberar.

Es muy importante realizar esta operación antes de asignarle otro valor a un puntero con la dirección de memoria reservada con ***malloc()***. Caso contrario la memoria dejará de poder ser liberada y se convertirá en una "memory leak". Esto es indeseable ya que constituye memoria no utilizada y que además no es reutilizable para otras tareas.

```
#include <stdio.h>
#include <malloc.h>

int main( int argc, char *argv[])
{

    int *vector1;
    int *vector2;

    /* correcto */
    vector1 = (int *)malloc(10*sizeof(int));
```

[2] Esto se podría manejar reservando estáticamente una cantidad de memoria adecuada al conjunto de datos más grande que se pretenda manejar. Pero es una solución más ineficiente y añade restricciones innecesarias (tamaños máximos) al programa.

[3] No confundir con la estructura de datos conocida por el mismo nombre.

```
    /* incorrecto: falta casting explicito y tamanio de int*/
    vector2 = malloc(10);

    /* correcto, se libera memoria */
    free( vector1 );

    /* incorrecto, se libera dos veces un mismo recurso */
    free( vector1 );

    return 0;
    /* nunca se libero vector2 SE PIERDE MEMORIA */

}
```

1.3 Temas adicionales de punteros

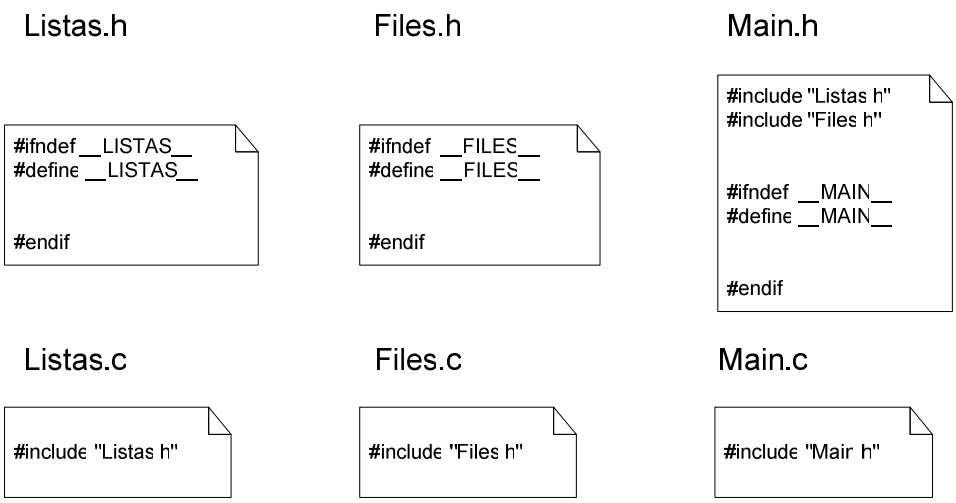
1.3.1 Punteros a funciones

```
#include <stdio.h>

int suma (int a, int b)
{
  return a+b;
}

int resta (int a, int b)
{
  return a-b;
}

int main( int argc, char *argv[])
{

    // declaramos un puntero a funciones con dos parámetros
    // enteros que devuelven un entero
    int (*funcion) (int,int);
    int x;

    funcion = suma;     // .funcion. apunta a .suma.
    x = funcion(4,3);  // x=suma(4,3)
    funcion = resta;   // .funcion. apunta a .resta.
    x = funcion(4,3);  // x=resta(4,3)

    return 0;
}
```

2 Armando del proyecto en librerías

Si bien hasta el momento veníamos trabajando sobre un único archivo .c, en la práctica nos encontraremos que por cuestiones de un mejor y más organizado desarrollo necesitaremos poder separar en diversos módulos nuestra aplicación. Esto nos lleva a la necesidad de tener diferentes archivos dentro de nuestro espacio de trabajo (workspace).

Miremos una grafica de como debería verse un proyecto dado :



Como podemos apreciar en el esquema anterior, agregamos una cláusula del precompilador para evitar la inclusión recursiva entre 2 headers dados. Para ver el por que de esto, analicemos el siguiente caso.

Supongamos que desde main incluimos files.h, y desde files.h incluimos listas.h, hasta aquí no habría problemas, ahora supongamos que desde listas.h incluimos files.h, que ocurriría?, si no tuviéramos las cláusulas ***#ifndef*** y ***#define***, se formaría un lazo que finalizaría terminando con la pre-compilación.

De aquí que debe prestarse atención desde donde se incluyen los headers.

Las cláusulas del precompilador utilizadas son :

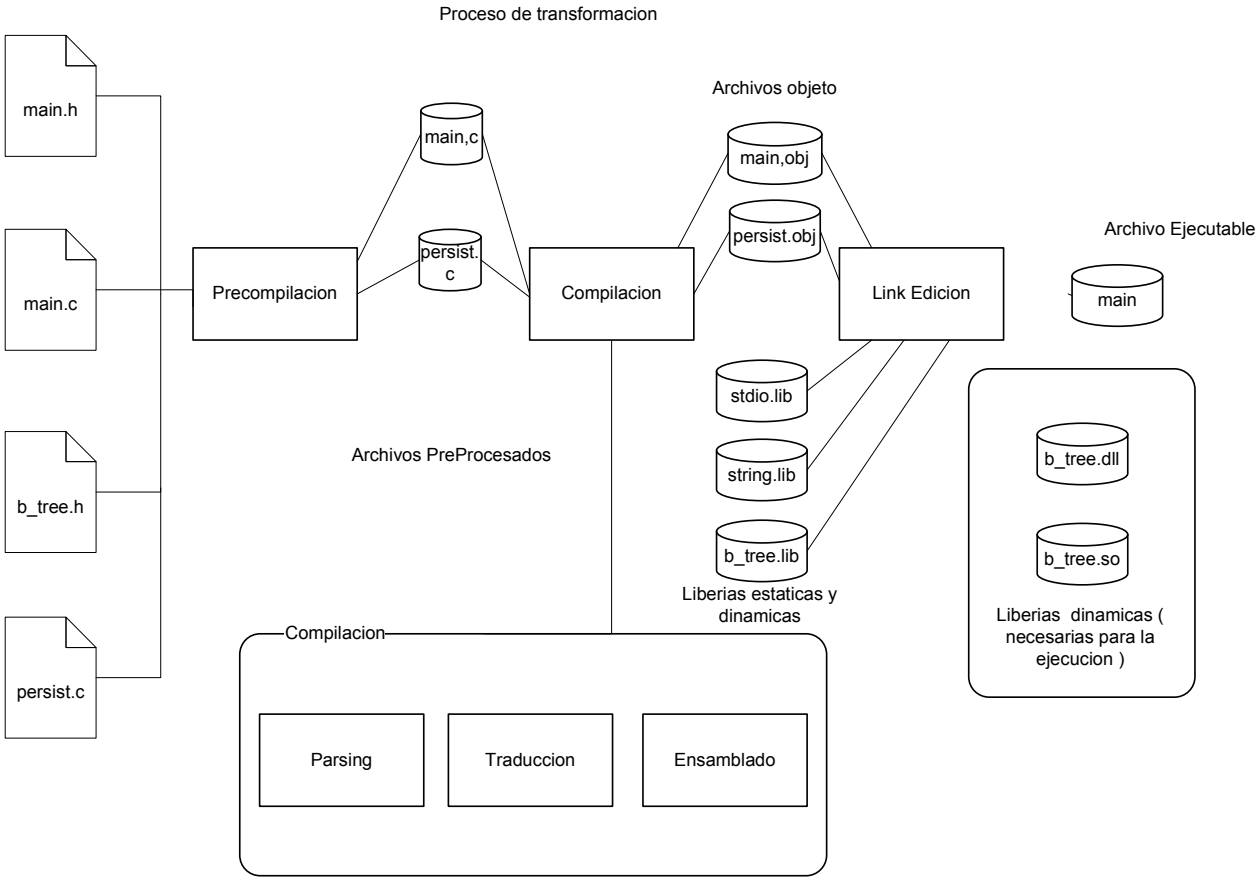
```
#define     nombre      valor_a_reemplazar

#ifndef     simbolo
      .....
      .....
#endif.

#include    “Header.h”
```

3 Proceso de construcción

3.1 Proceso de transformación

El proceso de transformación involucra partir desde los archivos fuentes, pasar por las diferentes etapas de procesamiento, y llegar a un archivo de biblioteca o ejecutable.

Podemos ver en la grafica siguiente como es el esquema sobre el cual trabajaremos :



Ahora analicemos cada una de las etapas de este proceso

3.1.1 Precompilación (Preprocesamiento)

El preprocesador es el encargado de 2 importantes tareas

- Sustitución y control de la precompilación

- Expansion de Macros.

Aunque parezcan tareas triviales, no lo son, y cada una de ellas contiene sus directivas asociadas. Antes de comenzar revisando las directivas del precompilador debemos tener presente que toda instrucción del precompilador comienza con el carácter #.

3.1.1.1 Sustitución y Control de la Precompilación

Cuando hablamos de sustitución estamos hablando hacer de la capacidad del precompilador de incluir (o dejar de hacerlo) ciertos bloques de código dentro de nuestros archivos fuente. Este proceso es en parte muy parecido a un "buscar y reemplazar" (find an replace) de un procesador de textos.

Dentro de las directivas que encontraremos tenemos una de las mas importantes y utilizadas que es la directiva ***#define*** la misma permite (entre otras cosas, luego veremos como puede ser utilizada para crear macros) definir un símbolo que será reemplazado dentro de nuestro código fuente, p.e:

```
#define     edad        12
#define     pi          3.1426
#define     max_lines   23

int main(int argc, char* argv[])
{
      .....
      .....
      printf("%i", edad);     /* Aquí se reemplazara edad por 12 */
      .....
      .....
}
```

Luego encontramos la directiva ***#include*** la misma permite incluir un archivo en el lugar donde esta la misma, que quiere decir esto, donde el precompilador encuentre un ***#include nombre_de_archivo*** el tomara el contenido del archivo y lo insertara en ese lugar.

Tenemos 2 formas diferentes de utilizar ***#include*** una es :

#include <archivo.h>

Y otra es

#include ".\ruta\archivo.h"

En la primera estaremos haciendo una inclusión de un archivo .h que se encuentra ubicado dentro de las rutas de busqueda por defaul del compilador (esto dependera de cada compilador) y en el segundo caso estaremos utilizando un path absoluto (relativo a donde estemos parados) para buscar el archivo .h. Es muy importante entender la diferencia de sintaxis para evitar inconvenientes al momento de querer procesar el archivo fuente.

Otra sentencia importante es la sentencia ***#undef*** la misma tiene el significado opuesto a ***#define*** y hace que un símbolo no tenga mas el sentido definido después de ella, p.e:

```
#define     edad        12
#define     pi          3.1426
#define     max_lines   23

int main(int argc, char* argv[])
{
      .....
      .....
      printf("%i", edad);          /* Aquí se reemplazara edad por 12 */
      .....
      Printf("%i", max_lines);     /*Aquí se reemplazara max_lines por 23 */

#undef      max_lines              /* De aquí en adelante max_lines no esta definido*/
#define     max_lines   12         /* De aquí en adelante max_lines vale 12 */



}
```

Otras de las sentencias utilizadas para el control de la precompilacion son los bloques ***#if-#else-#endif,*** y los bloques ***#ifndef-#ifdef-#if defined()-#endif***

```
#define     edad        12
#define     pi          3.1426
#define     max_lines   23

#if defined(edad)
VariableA=edad;
#else
VariableB=edad;
#endif

/* #if defined y #if ¡defined pueden ser abreviados como #ifdef #ifndef */

#if 0
printf("Este codigo pasa a ser un comentario\r\n");
#endif
```

Muchas veces es necesario separar una definición (veremos cuando veamos macros) en varias líneas para esto utilizaremos el carácter "\" al final de cada línea.

Directiva ***#assert( )*** esta directiva esta definida el el archivo de cabecera assert.h, y lo que hace es probar el valor dentro de ella y si es falso, imprimir un mensaje de error y llamar a la funcion abort de la stdlib.h deteniendo la ejecución. Muchas veces es utilizada para debug, cuando queremos ver si una variable tiene el valor correcto.

3.1.1.2 Constantes simbólicas predefinidas

Aquí encontramos las siguientes constantes definidas para ser utilizadas en nuestros programas :

__LINE__	Numero de la línea actual del código fuente (una constante entera).
__FILE__	El nombre del archivo fuente(un string).
__DATE__	La fecha de compilación del archivo fuente (una cadena de la forma "Mmm dd yyyy" como "Jan 20 1999".

__TIME__ La hora en que fue compilado el archivo fuente (una cadena de la forma "hh:mm:ss".

Asociada a estas constantes predefinidas tenemos la directiva ***#line valor_entero*** la que hace que apartir de ella las lineas de codigo fuente se renumeren, iniciando con el valor entero que se especifica después de ***#line.***

3.1.1.3 Expansión de macros

Antes de abordar como hacer un pasaje de parámetros a un define, miremos que ocurre con el siguiente bloque de código :

```
#define     numeros     1, \
                        2, \
                        3

int var[] = [ numeros ];
```

Las macros son defines que pueden recibir argumentos como si fueran una función la sintaxis se diferencia solamente por que junto al nombre del símbolo tenemos una lista de argumentos entre paréntesis.

```
#define     min(x,y)    ( (X) < (Y)? (X) : (Y) )

x=min(1,2); /* = = > x = ( (1) < (2)?(1): (2) );
```

Operador # (stringficacion)

```
#define     xstr(s)     str(s)
#define     str(s)      #s
#define     fijo        10

str(fijo)
 = = > "10"

xstr(fijo)
 = = > xstr(10)
 = = > str(10)
 = = > "10"
```

Operador ## (Concatenacion)

```
#define     COMMAND(NAME)     { #NAME, NAME ## _command }

struct command commands[] =
{
      COMMAND(quit),
      COMMAND(logon),
      COMMAND(logoff),
};

/* Tendriamos la siguiente estructura
```

```
struct command commands[] =
{
      { “quit”, quit_command},
      { “logon”, logon_command},
      { “login”, login_command}
};
*/
```

3.1.2 Compilacion

En esta etapa de un archivo fuente (o varios) obtendremos varios archivos con “*opcodes”* que son representativos y tienen sentido para el micro-procesador(controlador) . La etapa de compilación involucra varias sub-etapas. Es importante destacar que como en la ultima etapa se genera código objeto relevante para un determinado microprocesador, un compilador puede estar restringido al mismo y a una determinada plataforma, los compiladores que pueden generar código objeto para diferentes procesadores/plataformas son llamados *“cross-compilers”*

3.1.2.1 Parsing(Analisis Sintactico)

En esta sub etapa el compilador tiene la responsabilidad de armar un árbol semántica, que le permita llegar desde la sentencia final a alguno de los símbolos terminales, esta es una subetapa bastante compleja y que escapa su análisis en profundidad al objetivo de esta unidad. Existen muchas metodologías para realizar el parsing de un archivo una de las mas conocidas es ***parsing de arriba hacia abajo mediante descenso recursivo (top-down parsing by recursing descent).***

3.1.2.2 Translation(Traducción)

Una vez parseado el archivo fuente y ya desglosado en símbolos que puedan ser fácilmente traducidos, el compilador comienza a convertir de estos símbolos a instrucciones de lenguaje ensamblador, ya aquí vemos que esta etapa empieza a ser dependiente de la plataforma y procesador en la que se pretende correr el archivo ejecutable.

3.1.2.3 Assembler (Ensamblado)

La etapa de assembler consiste en tomar los archivos traducidos a lenguaje ensamblador, muchas veces llamados mnemonicos ya que son instrucciones del mas bajo nivel posible y que tienen fuerte dependencia del procesador (se compila en un ensamblador para un procesador dado) y de la plataforma (este procesador tiene una arquitectura definida). Una vez corrida esta subetapa obtendremos un código binario (llamado objeto) que en Windows suele terminar con la extensión ***.obj*** y en Linux ***.o*** este archivo si bien es un binario con código capaz de ser interpretador por un procesador aun no puede ser ejecutado, requiere de la resolución de símbolos externos, librerías, etc. Para ello necesitamos de una etapa de linkedición.

3.1.3 Link Edición

Todos los archivos objeto generados en el paso de compilación son individualmente incompletos, estos deben ser combinados de una determinada manera para que antes de que el programa sea capaz de ser ejecutado. Puede ocurrir que ciertos símbolos (Funciones o variables) no hayan sido resueltos, resolverlas y mostrar mensajes de error si estos símbolos no pueden ser hallados es parte del trabajo del linker, aquí es importante destacar 2 formas de Linkear (Enlazar), ***Estaticamente (Static Linking)*** utilizando esta forma de enlazado se copia dentro del ejecutable todo el codigo necesario para que el mismo pueda correr, osea

una vez finalizado el ejecutable no necesita nada mas que el kernel para correr, por otro lado podemos Linkear ***Dinámicamente (Dynamic Linking)*** cuando enlazamos de esta forma el linker es invocado en tiempo de ejecución y resuelve los símbolos contra las librerías presentes en el sistema, el ejecutable es mucho mas pequeño, pero tiene dependencias.