INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORE MONTERREY

LABORATORIO DE SISTEMAS OPERATIVOS

Práctica 5

Programación en C.

Conceptos:

*Etapas de compilación: preprocesamiento, compilación, ensamblado, ligado

*Ligado estático vs dinámico.

*Preprocesamiento

- Lenguaje de programación C.
- Elementos de un programa sencillo de C
- Compilar programas con gcc

Lenguaje de programación C[1]

C

es un lenguaje de programación creado en 1972 por Ken Thompson y Dennis M. Ritchie en los Laboratorios Bell como evolución del anterior lenguaje B, a su vez basado en BCP

Al igual que B, C es un lenguaje orientado a la implementación de Sistemas Operativos incluyendo por supuesto a Unix. C es apreciado por la eficiencia del código que produce y es el lenguaje de programación más popular para crear software de sistemas (compiladores, linkers, utilities...), aunque también se utiliza para crear otro tipo de aplicaciones.

Se trata de un lenguaje débilmente tipificado de medio nivel pero con muchas características de bajo nivel. Dispone de las estructuras típicas de los lenguajes de alto nivel pero, a su vez, dispone de construcciones del lenguaje que permiten un control a muy bajo nivel. Los compiladores de C suelen ofrecer extensiones al lenguaje que posibilitan mezclar código en ensamblador con código C o acceder directamente a memoria o dispositivos periféricos.

La primera estandarización del lenguaje C fue en ANSI, con el estándar X3.159-1989. El lenguaje que define este estándar fue conocido como ANSI C. Posteriormente, en 1990 fue ratificado como estándar ISO (ISO/IEC 9899:1990). La adopción de este estándar es muy amplia por lo que, si los programas creados lo siguen, el código es portátil entre plataformas y/o arquitecturas. En la práctica, los programadores suelen usar elementos no-portátiles dependientes del compilador o del sistema operativo.

...

C se desarrolló originalmente (conjuntamente con el sistema operativo Unix, con el que ha estado asociado mucho tiempo) por programadores para programadores. Sin embargo, ha alcanzado una popularidad enorme, y se ha usado en contextos muy alejados de la programación de sistemas, para la que se diseñó originalmente.

Elementos de un programa sencillo de C

Estructura de un programa en C:

La estructura básica de un programa en lenguaje C se muestra a continuación (basic.c):

```
/* <--SIMBOLO PARA COMENTARIOS */
#include  <stdio.h>        /* Encabezado */

int x;       /* Variables globales*/
```

```
int funcion(int entrada)           /* Declaración de una función */
{
       return 1;
}

int main(int argc, char* argv[])         /* Función principal */
{
       …       /* Código */
       return 0;
}
```

Encabezado : Aquí se definen los nombres de las bibliotecas de funciones que se incluirán en el programa.

Variables Globales : Variables cuyo alcance es todo el programa, también se definen constantes.

Declaración de funciones : Aquí se definen las subrutinas del programa.

Programa principal : Esta es la declaración de la función principal del programa.

Código : Este es el área del programa principal.

argc (argument count) especifica el número de argumentos con que se ha invocado la función principal.

argv* [ ] (argument values) es un arreglo ([ ]) de apuntadores o punteros (*) a los caracteres (char) que forman cada argumento.

Tipos de datos básicos:

int	enteros (números enteros positivos y negativo
char	caracteres (letras)
float	números en punto flotante (números reales)
double	números en punto flotante de doble precisión
void	No-tipo (se emplea con punteros)

Se pueden construir tipos de datos más elaborados a partir de estos tipos básicos:

- Arreglos y matrices
- Punteros
- Tipos estructurados (estructuras)

La sintaxis para declarar variables es:

tipo nombre_de_la_variable ;

Por ejemplo:

```
char letra;
```

Arreglos y matrices.

Se pueden crear variables que sean conjuntos de elementos del mismo tipo (arreglos o matrices), con la siguiente sintaxis:
tipo nombre_del_arreglo [dimensión] ;
Ejemplo:

```
int arreglo [5] ;       /* Crea un array de cinco enteros */
```

Los elementos de un array empiezan en *cero* y terminan en *dimensión-1*. Para acceder al elemento i de un arreglo, se utiliza la expresión:
arreglo [i]. Entonces en la declaración anterior, arreglo[5] queda fuera del arreglo.

Múltiples dimensiones.

Se pueden declarar matrices de dos o más dimensiones, según esta sintaxis:
tipo matriz [dimensión1] [dimensión2] ... ;
Ejemplo:

```
int matriz [3][8] ;
```

Se accede a los elementos con esta expresión:

matriz [i][j]

Operadores.

Los operadores *aritméticos* se muestran en la siguiente tabla:

Operador	Significado
+	Suma
-	Resta
*	producto
/	división
%	módulo (resto de la división

La forma de dar valor a una variable es:
variable = expresión ;

Por ejemplo:

```
contador = contador + 1;
```

Los operadores *lógicos* son:

Operador	Resultado
A == B	1 ("true") si A es igual a B; 0 ("false" en caso contrario
A != B	1 si A es distinto de B
A > B	1 si A es mayor que B
A < B	1 si A es menor que B
A >= B	1 si A es mayor o igual que B

if (A=B) ... es un error común. La expresión **no** está comparando A con B para ver si son iguales. ¿Cual es el efecto de esa expersión?

Y los *booleanos*:

Expresión	Resultado
A1 && A2	Cierta si A1 y A2 son ciertas (AND)
A1 || A2	Cierta si A1 o A2 son ciertas (OR)
! A	Cierta si A es falsa; falsa si A es cierta (N

También existen operadores *avanzados*:

Operadores	Significado
A++, ++A	Incrementa en 1 el valor de A (A=A
A--, --A	Disminuye en 1 el valor de A (A=A-1
A+=x	A=A+x
A-=x	A=A-x
A*=x	A=A*x
A/=x	A=A/x

Entrada y salida de datos:

Salida a pantalla se realiza con la función printf, se utiliza según este formato:

printf (“cadena de formato”, arg1, arg2, ... argN);

Para usar printf, hay que escribir al principio del programa el header o encabezado:

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

Algunos formatos de printf son:

%d Entero decimal
%u Entero decimal con signo
%x Entero hexadecimal
%c Carácter
%s String o arreglo de caracteres.
%f Punto flotante (float)

Ejemplo:

```
int numero = 1234;
char letra = ‘h’;
printf( “un numero %d y una letra %c”, numero, letra);
```

La *lectura de datos* desde el teclado se hace con la función: scanf, la lectura de datos se hace desde el teclado. La sintaxis es:
scanf (formato, & arg1, & arg2, ...);

En formato se especifica qué tipo de datos se quieren leer. Se utilizan los mismos format que en printf. También hay que incluir el encabezado: <stdio.h>. Ejemplo:

```
int x;
char y;
scanf ( “%d”, &x );
scanf ( “%c”, &y );
```

Comandos para control de flujo.

A continuación se muestra un resumen de estos comandos.
El comando **if** sirve para ejecutar código sólo si la condición es cierta:

```
if  ( condición ) {
      sentencia si cierto
} else {
      sentencia si falso
}
```

Ejemplo:

```
int x = 5;
if ( x >= 0 ){
      printf (“la variable x es positiva”);
} else {
      printf (“x es negativa”);
}
```

El comando **switch** se utiliza para ejecutar acciones diferentes según el valor de la expresión. La sintaxis es:

```
      switch ( expresión ) {
            case valor1:
              ... sentenciasA...
            break;

            case valor2:
              ... sentenciasB ...
            break;

            case valor3:
            case valor4:
              ... sentenciasC ...
```

```
            break;

            default:
              ... sentenciasD ...
      }
```

Las *sentenciasA* se ejecutarán si expresión adquiere el *valor1*.
Las *sentenciasB* se ejecutarán si adquiere el *valor2*.
Las *sentenciasC* se ejecutarán si adquiere el *valor3* o el *valor4*, indistintamente.
Cualquier otro valor de expresión conduce a la ejecución de las *sentenciasD*; eso viene indicado por la palabra reservada **default**.

Ejemplo de switch:

```
int opcion;
printf("Escriba 1 si desea continuar; 2 si desea terminar: ");
scanf ( "%d", &opcion );

switch ( opcion ) {
      case 1:
            printf ("OK, continuamos");
   break;

      case 2:
            salir = 1;
            printf(“Bye, salimos”);
            break;
      default:
            printf ("opción no reconocida");
}
```

Comandos de repetición

El comando **while** se utiliza para ejecutar el mismo código varias veces. La sentencia se ejecuta una y otra vez mientras la condición sea cierta. La sintaxis es:

```
while ( condición ) {
            sentencia
}
```

Ejemplo:

```
int x=1;

while ( x < 100 ) {
      printf(“Línea número %d\n”,x);
      x++;
}
```

El commando **do...while** es similar al ciclo while, pero itera al menos una vez. La sintaxi

sintaxis es:

char nombre_de_cadena [longitud + 1];

Ejemplo:

```
char hola [5];
```

Toda cadena ha de terminar con el carácter especial '\0' (cero), es por eso que el tamaño del arreglo debe ser *longitud + 1* para poder albergar la cadena de caracteres más el carácter especial de fin de cadena '\0'.

También se puede inicializar una cadena en su declaración, el carácter de fin de cadena se agrega automáticamente, ejemplo:

```
char saludo[11] = "hola mundo";
```

La biblioteca estándar de C también incluye algunas funciones específicas para el manejo de cadenas. Una de ellas es la función **strlen**, que sirve para calcular la longitud de una cadena. Su sintaxis es:

```
#include <string.h>
int strlen(const char *cadena); /*declaración de la función strlen dentro de
string.h*/
```

Ejemplo:

```
$ cat ejemploStrings.c

#include <stdio.h>
#include <string.h>

int main()
{
      int longitud = 0;
      const char *cadena = "Esta es una cadena de longitud desconocida"; /*
char*: apuntador a la cadena */
      longitud = strlen(cadena);
      printf ("Hay %d caracteres en: %s\n", longitud, cadena);
      return 0;
}

$ gcc -o ejemploStrings ejemploStrings.c
$ ./ejemploStrings
Hay 42 caracteres en: Esta es una cadena de longitud desconocida
```

Puedes leer más sobre las funciones para el manejo de strings en **man string.h**.

Manejo de archivos básico con stdio.h

La biblioteca STDIO contiene funciones para trabajar con archivos: básicamente leer y escribir datos de diferentes formas. Antes de trabajar con un archivo, hay que abrirlo. Cuando se abre un archivo, se devuelve un puntero a una estructura de tipo FILE (definido en STDIO). Esta estructura, llamada descriptor de archivo , servirá para manipular posteriormente el archivo.

Al terminar con su uso hay que cerrar el archivo.

Abrir un archivo
fd = fopen (nombre, modo);

Devuelve NULL si no se pudo abrir correctamente el archivo.
nombre es el nombre del archivo.
fd es un descriptor de archivo.
modo es una cadena donde se define el modo de apertura:
r sólo lectura
w escritura
a append (escribir después del final)
Ejemplo:

```
fd = fopen ( "archivo.txt", "rw" );
```

Cerrar un archivo

fclose (fd);

donde *fd* es un descriptor de archivo ya abierto.

Leer una cadena desde un archivo

fgets (cadena, N, fd);

Lee una línea de texto.
cadena es el arreglo de caracteres donde se almacenará la línea.
N es el máximo número de caracteres para leer. Si la línea ocupa más de *N* caracteres, sólo se lee esa cantidad.
fd es el descriptor del archivo.

Cuando se abre un archivo, se comienza a leer desde la primera línea. Llamadas consecutivas a **fgets** van leyendo línea por línea. Si se cierra el archivo y se vuelve a abrir, se lee otra vez desde el principio.
Internamente existe un puntero del archivo que apunta a la posición relativa al archivo donde se va a leer el siguiente dato.

Escribir una cadena en un archivo

fputs (cadena, fd);

cadena es el texto que se escribe. fputs incluye además un salto de línea.
fd es el descriptor del archivo.
Las líneas escritas se van añadiendo una detrás de otra. Se comienza por el principio del archivo (w) o por el final (a). También existe internamente un

puntero de archivo para escritura.

fprintf (fd, formato, arg1, arg2, ...);

Escribe un texto con formato en un archivo. La sintaxis es idéntica a printf, salvo que el primer argumento es un descriptor de archivo.

Detectar el final de archivo

x = feof (fd);

Devuelve un 1 si se ha llegado al final del archivo. Un cero en caso contrario. Útil para saber cuándo dejar de leer información, por ejemplo.

Ejemplo del manejo de archivos:

Archivo fuente *ejemploArchivos.c* :

```
#include <stdio.h>

int main ()
{
      FILE* fd_in; /* variable que apunta al descriptor de archivo */
      FILE* fd_out;/* variable que apunta al descriptor de archivo */
      int contador = 1;
      static int maximo = 1023;
      char linea[1024];

      /* abre el archivo en modo lectura */
      fd_in = fopen ( "entrada.txt", "r" );
      /* abre el archivo en modo escritura */
      fd_out = fopen ( "salida.txt", "w" );

      while ( feof(fd_in) != 1 ) {
            fgets ( linea, maximo, fd_in );
            fprintf ( fd_out, "%d - %s", contador, linea );
            contador++;
      }

      fclose (fd_in);         /* cierra el archivo */
      fclose (fd_out);  /* cierra el archivo */

      return 0;
}
```

Compilado:

```
$ gcc -o ejemploArchivos ejemploArchivos.c
$ ls
ejemploArchivos   ejemploArchivos.c
```

Ejecución:

```
$ cat entrada.txt
esta es la linea uno
esta es la linea dos
esta es la linea tres
esta es la linea cuatro
esta es la linea cinco
$ ./ejemploArchivos
$ cat salida.txt
1 - esta es la linea uno
2 - esta es la linea dos
3 - esta es la linea tres
4 - esta es la linea cuatro
5 - esta es la linea cinco
6 - esta es la linea cinco
```

Procesamiento de los argumentos de línea de comando:

Cuando se ejecuta un programa en la línea de comandos, el shell se encarga de pasar a dicho programa los argumentos que usuario tecleó, además del nombre del programa. Un ejemplo común es el listar los archivos contenidos en algún directorio, especificando la opción de mostrar archivos ocultos usando el siguiente comando: `ls -a` , donde **ls** es el nombre de programa y **–a** es un argumento que recibe dicho programa. En este caso "ls" es un programa que ya viene precompilado en Unix.

El en caso del lenguaje de programación C, el programador es el encargado de procesar dichos argumentos. Los argumentos se manejan a través de dos variables, cuya declaración se realiza al principio del bloque *main*, por ejemplo:

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

La primer variable (de tipo entero): *argc*, contiene el número de argumentos que el shell pasó al programa, incluyendo el nombre del programa; por ejemplo, al ejecutar el siguiente programa:

```
$ ./concatena  uno  dos  tres
```

la variable *argc* tendrá el valor de: 4, es decir, son tres argumentos más el nómbre del programa.

La segunda variable (de tipo doble apuntador a caracter): *argv* , contiene un arreglo de strings con la transcripción de cada uno de los argumentos. El primer elemento de este arreglo, el elemento 0, es el nombre del programa, y el último elemento, argc - 1, es el último argumento. Por ejemplo, al ejecutar el siguiente programa:

```
$ ./concatena  uno  dos  tres
```

la variable *argv* almacena los siguientes valores:

```
argv[0]=    .  /  c  o  n  c  a  t  e  n  a  '\0'
argv[1]=    u  n  o  '\0'
argv[2]=    d  o  s  '\0'
argv[3]=    t  r  e  s  '\0'
```

El siguiente programa *argumentos.c* muestra un ejemplo del manejo de los argumentos
línea de comandos.

argumentos.c :

```
#include <stdio.h>

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

      if (argc == 1) {
            printf("Error, debes teclear al menos un argumento\n");
            return -1;
      }

      printf("Hay %d argumentos: \n", argc);

      for(i=0; i<argc; ++i) {
            printf("argumento (%d) = %s \n", i, argv[i]);
      }

      return 0;
}
```

La ejecución de dicho código se muestra en estas capturas de pantalla:

```
$ ./argumentos
Error, debes teclear al menos un argumento
$ ./argumentos uno dos tres
Hay 4 argumentos:
argumento (0) = ./argumentos
argumento (1) = uno
argumento (2) = dos
argumento (3) = tres
```

Compilar programas con GCC[2]

GNU Compiler Collection -Colección de Compiladores GNU- es un conjunto de compiladores creados por el proyecto GNU. GCC es software libre y lo distribuye la FSF bajo la licencia GPL. Estos compiladores se consideran estándar para los sistemas operativos derivados de UNIX, de código abierto o también de propietarios, como Mac OS X. GCC requiere el conjunto de aplicaciones conocido como binutils para realizar tareas