Funciones, estructura del programa, modularización

15/3/2018

Resumen

En esta clase hablaremos sobre el alcance de las funciones y variables así como separar un programa en diferentes archivos para modularización.

Índice General

1 Introducción [0:10]

Al igual que otros lenguajes de programación estructurados como PASCAL, las funciones hacen que un programa complejo se pueda subdividir en tareas más sencillas, además como veremos un programa se puede dividir entre varios archivos, que además se pueden compilar por separado, permitiendo reutilizar funciones hechas por otras personas.

De hecho cada vez que incluimos un archivo .h para usar las funciones de la biblioteca estándar de C, por ejemplo `printf` de `"stdio.h"`, estamos haciendo uso de funciones, así como `main` es una función en si misma que se llama en el momento de ejecución del programa.

En el caso del primer obligatorio por ejemplo ustedes tendrán que implementar una biblioteca con algunas funciones que les serán de utilidad en los siguientes obligatorios.

2 Cuestiones básicas [0:20]

Para poder usar una función esta tiene que haber sido *declarada* en el código antes de ser invocada, pero no necesariamente implementada.

Una declaración de función consiste únicamente en decir su tipo de retorno, su nombre y los argumentos que recibe.

Su declaración puede estar separada de la implementación sobre todo si es una función hecha para utilizar en varios programas. Veremos la utilidad de esto cuando luego cuando hablemos de modularización y encapsulamiento.

NOTA: *Ponemos como ejemplo función que calcula la secuencia de fibonacci hasta un cierto indice y un main que imprime los resultados.*

Ejemplo: ejemplos/fibonacci02/fibonacci.c Función básica

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

#define MAX_IND 93

/**
 * Ejemplo de una función:
 * Calcula la secuencia de fibonacci hasta el índice n y coloca los valores en
 * en buff
 */
void fibonacci(int n, unsigned long buff[]){
        int i = 0;
        buff[0] = 0;
        buff[1] = 1;
        for (i=2; i<=n; i++) {
                buff[i] = buff[i-1] + buff[i-2];
        }
}

int main(){
        unsigned long fib[MAX_IND+1];
    int indice=30, i;
        fibonacci(indice, fib);
        for (i=0; i<=indice;i++){
                printf("fib[%d]=%lu\n",i,fib[i]);
        }
        return 0;
}
```

2.1 Declaración de funciones [0:30]

Una declaración de función tiene la forma:

`tipo-retorno nombre-función(declaración de argumentos);`

Donde la declaración de argumentos es una lista de variables que tomarán los valores de los parámetros cuando la función sea llamada. Las declaraciones de argumentos se separan por comas y consiste del tipo de variable y su nombre (opcional) separados por espacios.

En la declaración los nombres de los parámetros son opcionales pero es recomendable ponerlos para que quien la lea tenga una idea de que significa cada parámetro.

A su vez el tipo de retorno es opcional, pero también conviene explicitarlo ya que al contrario de lo que se puede suponer intuitivamente leyendo el código, cuando no se define, se asume que es `int` en lugar de `void`

Es importante comprender la necesidad de la declaración de las funciones sobre todo porque en programas extensos donde además participan varias personas, las declaraciones de las funciones son quienes definen la interfaz entre quién las usa y quien las implementa.

NOTA: *Declaramos la función antes del main sin implementación. Qué pasa?*

2.2 Implementación de funciones [0:35]

Una implementación de función tiene la misma forma que la declaración pero en lugar de `;` luego va el cuerpo que implementa la misma entre llaves, como en la función `main`.

En los casos que la función declare un valor de retorno es obligatoria la sentencia `return expresión;` para especificar el valor de retorno, si la función es declarada como `void` la sentencia `return` puede aparecer en el cuerpo de la función para salir de la misma en algún punto pero sin argumento, y además no es obligatoria ya que la función termina al llegar a la última sentencia.

NOTA: *Implementamos la función previamente declarada .*

Ejemplo: ejemplos/fibonacci02.2/fibonacci.c Función declarada e implementada aparte

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

#define MAX_IND 93

/*Ejemplo de declaración de una función*/

/**
 * calcula la secuencia de fibonacci hasta el índice n y coloca los valores en
 * en buff
 */
void fibonacci(int n, unsigned long buff[]);

int main(){
        unsigned long fib[MAX_IND+1];
        int indice=30, i;

        fibonacci(indice, fib);
        for (i=0; i<=indice;i++){
                printf("fib[%d]=%lu\n",i,fib[i]);
        }
        return 0;
}

void fibonacci(int n, unsigned long buff[]){
        int i = 0;
        buff[0] = 0;
        buff[1] = 1;
        for (i=2; i<=n; i++) {
                buff[i] = buff[i-1] + buff[i-2];
        }
}
```

3 Variables externas [0:40]

A diferencia de Pascal todas las funciones de C son globales (externas), no se pueden definir funciones dentro de otras funciones, también son globales todas las variables definidas fuera del cuerpo

de una función. Así por defecto las funciones globales son visibles en todo el alcance del programa, incluso si están definidas en fuentes separados. Cuando hacemos una llamada a una función que no está previamente declarada, el compilador va a dar un warning y asumir que su declaración es `int nombre_funcion()`.

Es **muy importante** considerar estos warnings como un error ya que si la función por ejemplo retorna otra cosa como un float, se va a convertir a entero el resultado y se pierde la parte decimal.

NOTA: *Vemos que pasa si eliminamos la declaración de la función.*

Dado que el proceso de compilación ocurre primero y luego el proceso de enlace, que es cuando el "enlazador"determina donde esta la implementación real de la función ya compilada, o la ubicación real del espacio de almacenamiento de variables, a este proceso se llama *enlazado externo* y aplica tanto a funciones como a variables. Sin embargo las variables por defecto son *internas*, es decir su visibilidad se restringe al fuente en el cual están declaradas (esto adquirirá sentido cuando veamos como se puede separar un programa en diversos fuentes luego).

En general como buena práctica de programación, debe intentar evitarse el uso de variables globales así como de variables externas, pero en casos en los cuales muchas funciones tienen que tener acceso a la misma variable se puede justificar el uso de variables globales externas para no tener que pasarlas como argumentos a todas las funciones que deben tener acceso a esa información.

También hay que tener en cuenta que cada parámetro que se pasa a una función hay que copiarlo al stack por lo que si estamos programando un sistema embebido, un procesador de escasos recursos o una función crítica que tiene un alto impacto en la eficiencia del programa hay que considerar el beneficio de tener la información que necesita ser comunicada entre las distintas rutinas en un ambiente global, partiendo de la base que ese código luego no puede llamarse en paralelo o que luego estaremos limitados a que haya una sola instancia de esa información en el programa.

NOTA: *Modificamos el programa para que el buffer sea una variable global;*

Ejemplo: ejemplos/fibonacci03/fibonacci.c hacemos del buffer una variable global

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

#define MAX_IND 93

unsigned long fib[MAX_IND+1];

void fibonacci(int n);
int main(){
        int indice=30, i;
        fibonacci(indice);
        for (i=0; i<=indice;i++){
                printf("fib[%d]=%lu\n",i,fib[i]);
        }
        return 0;
}

/**
 * calcula la secuencia de fibonacci hasta el índice n y coloca los valores en
 * en fib
 */
void fibonacci(int n){
        int i = 0;
        fib[0] = 0;
        fib[1] = 1;
        for (i=2; i<=n; i++) {
                fib[i] = fib[i-1] + fib[i-2];
        }
```

```
}
```

4 Reglas de alcance [0:50]

El alcance de un elemento (variable o función) es la porción del código que tiene acceso a ese elemento. Este alcance es desde que son declaradas hasta el final del archivo si son globales, o de la función si son variables locales.

Sin embargo para variables globales, sobre todo cuando deben ser accedidas desde distintos fuentes, la definición de la misma debe ser hecha una sola vez, pero existe la palabra clave `extern` para declarar que una variable está definida en otro fuente o más adelante en el archivo. Esta declaración de variables externas funciona como la declaración de funciones sin implementación, solo que en el caso de una variable, la implementación vendría a ser el lugar donde se define el espacio de memoria que ocupará.

Cuando una variable se define como externa el compilador no le da lugar en el stack si no que considera que su lugar va a estar definido en el proceso de enlazado, así como ocurre con las funciones.

NOTA: *Hacemos el buffer extern y lo definimos antes de la función*

Ejemplo: ejemplos/fibonacci04/fibonacci.c declaramos el buffer antes del main pero lo definimos luego

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

#define MAX_IND 93


/*declaraciones*/
extern long fib[MAX_IND+1];

void fibonacci(int n);

/**Programa principal*/
int main(){
        int indice=30, i;
        fibonacci(indice);
        for (i=0; i<=indice;i++){
                printf("fib[%d]=%lu\n",i,fib[i]);
        }
        return 0;
}

/*
 * Comienzo de la implementación
 */
long fib[MAX_IND+1];

/**
 * calcula la secuencia de fibonacci hasta el índice n y coloca los valores en
 * en fib
 */
void fibonacci(int n){
        int i = 0;
        fib[0] = 0;
        fib[1] = 1;
        for (i=2; i<=n; i++) {
                fib[i] = fib[i-1] + fib[i-2];
        }
}
```

5 Archivos de encabezado [0:55]

En general es buena idea separar las funciones accesorias que usa un programa en varios archivos, ya sea para implementar bibliotecas o porque diferentes personas estarán trabajando en la implementación de las distintas unidades. Aquí toman especial relevancia los archivos de encabezado.

Un archivo de encabezado tiene la extensión .h (de header en inglés). Y se usa para definir la interfaz a un conjunto de funciones, en general agrupadas según su funcionalidad. Existen para evitar tener que declarar las funciones en cada fuente en el cual se van a utilizar. Lo que se hace en este caso es declarar las funciones en un archivo .h, que luego se incluirá tanto en todos los fuentes .c que la usen, como en el fuente que la implementa en si mismo. Esto último es especialmente importante para que el compilador pueda detectar discrepancias entre la declaración de la función en el .h y el .c que la implementa.

El archivo de encabezado funciona como una interfaz entre utilizan un conjunto de funciones y quienes las implementan, aunque sea la misma persona es buena práctica hacer esta separación para que luego si hay que delegar la tarea de mantener el código a otra persona, ya tenga la estructura adecuada.

NOTA: *separamos la definición de la función fibonacci a otro fuente y definimos el encabezado.*

Ejemplo: ejemplos/fibonacci05/fibonacci.h archivo que declara funciones y variables globales de la función fibonacci

```
#define MAX_IND 93

extern long fib[];

void fibonacci(int n);
```

Ejemplo: ejemplos/fibonacci05/imprimir_fibonacci.c archivo con el main que llama a la función

```
#include <stdio.h>

#include "fibonacci.h"

int main(){
        int indice=30, i;
        fibonacci(indice);
        for (i=0; i<=indice;i++){
                printf("fib[%d]=%lu\n",i,fib[i]);
        }
        return 0;
}
```

Ejemplo: ejemplos/fibonacci05/fibonacci.c implementación de lo declarado en el .h

```


#include "fibonacci.h"

long fib[MAX_IND+1];

/**
 * calcula la secuencia de fibonacci hasta el índice n y coloca los valores en
 * en fib
 */
void fibonacci(int n){
        int i = 0;
        fib[0] = 0;
        fib[1] = 1;
        for (i=2; i<=n; i++) {
```

```
                fib[i] = fib[i-1] + fib[i-2];
        }
}
```

6 Variables estáticas [1:10]

La palabra clave `static` tiene dos usos posibles:

1. Cuando se aplica a una variable global o función, esto hace que la visibilidad de dicho elemento se limite al archivo que está siendo compilado, es decir que no sea visible en otro fuentes que potencialmente puedan usar el código, sirve para definir variables globales o funciones que queremos por ejemplo una biblioteca necesita internamente pero que no necesitan compartirse con el resto del código que las usa. Entre otras cosas esto es útil para evitar colisión de nombres de variables, por ejemplo si dos bibliotecas independientes usadas por el mismo programa deciden definir variables globales o funciones con el mismo nombre pero distinta declaración o implementación.

   NOTA: *Reimplementamos fibonacci de forma que no se necesite el buffer, usamos variables estáticas grobales para mantener el estado. Creamos una nueva función para implementar la secuencia: `siguiente_fibonacci`.*

   Ejemplo: ejemplos/fibonacci06.1/fibonacci.h archivo que declara funciones y variables globales de la función siguiente_fibonacci

```
#define MAX_IND 93


unsigned long siguiente_fibonacci();
```

   Ejemplo: ejemplos/fibonacci06.1/imprimir_fibonacci.c archivo con el main que llama a la función para imprimir la secuencia

```
#include <stdio.h>

#include "fibonacci.h"

int main(){
        int indice=30, i;
        for (i=0; i<=indice;i++){
                printf("fib[%d]=%lu\n",i,siguiente_fibonacci());
        }
        return 0;
}
```

   Ejemplo: ejemplos/fibonacci06.1/fibonacci.c implementación de siguiente_fibonacci

```


#include "fibonacci.h"

static unsigned long siguiente, primero = 0, segundo = 1, indice=0;


/**
 * calcula la secuencia de fibonacci hasta el índice n y coloca los valores en
 * en fib
 */
unsigned long siguiente_fibonacci(){

        if (indice <= 1) {
```

```
                return siguiente = indice++;
        }
        /*preparamos las variables para la siguiente ejecución*/

        siguiente = primero + segundo;
        primero = segundo;
        return segundo=siguiente;
}
```

2. Cuando se aplica a una variable interna de una función, esto hace que la variable sea solo visible por la función pero mantiene su valor entre distintas llamadas a la misma. Esto se logra reservando lugar para el almacenamiento de la variable fuera del stack en un lugar fijo y conocido al momento del enlazado del programa. Pero que ningún otro elemento del programa (ni siquiera dentro del mismo fuente) puede referenciar.

   NOTA: *Hacemos que las variables sean estáticas dentro de la función siguiente_fibonacci*

   Ejemplo: ejemplos/fibonacci06.2/fibonacci.c implementación con variables estáticas dentro de la función

```
#include "fibonacci.h"

/**
 * calcula la secuencia de fibonacci hasta el índice n y coloca los valores en
 * en fib
 */
unsigned long siguiente_fibonacci(){
        static unsigned long siguiente, primero = 0, segundo = 1, indice=0;

        if (indice <= 1) {
                return siguiente = indice++;
        }
        /*preparamos las variables para la siguiente ejecución*/
        siguiente = primero + segundo;
        primero = segundo;
        return segundo=siguiente;
}
```

7 Variables de registro

En la clase lo mencionamos al pasar.

C da la posibilidad de declarar que una variable sea en lo posible cargada en un registro del procesador, principalmente se usa cuando una variable numérica será muy utilizada en operaciones para hacer que el procesador no tenga que andar moviéndola de la memoria a un registro para su procesamiento en cada operación. Es mas bien una característica utilizada por temas de eficiencia.

El compilador puede elegir ignorar esta declaración si por ejemplo el procesador para el cual está compilando no tiene suficientes registros o no tiene registros del tipo de la variable.

NOTA: *Hacemos una implementación con variables de registro en la cual una sola función calcula e imprime la secuencia.*

Ejemplo: ejemplos/fibonacci07/fibonacci.h archivo que declara funciones y variables globales de la función siguiente_fibonacci

```
/**
 * imprime los numeros de fibonacci hasta el índice n
```

```
 */
void imprimir_fibonacci(int n);
```

Ejemplo: ejemplos/fibonacci07/imprimir_fibonacci.c archivo con el main que llama a la función para imprimir la secuencia

```
#include "fibonacci.h"

int main(){
        int indice=30;
        imprimir_fibonacci(indice);
        return 0;
}
```

Ejemplo: ejemplos/fibonacci07/fibonacci.c implementación de siguiente_fibonacci

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


/**
 * calcula la secuencia de fibonacci hasta el índice n e imprime los valores en
 * la salida estándar
 */
void imprimir_fibonacci(int n){
        register unsigned long siguiente, primero = 0, segundo = 1;
        int i;
        for (i=0; i<=n;i++){
                if (i <= 1) {
                        siguiente = i;
                } else {
                        /*preparamos las variables para la siguiente ejecución*/
                        siguiente = primero + segundo;
                        primero = segundo;
                        segundo = siguiente;
                }
                printf("fib[%d]=%lu\n",i, siguiente);
        }


}
```

8 Estructura de bloques

En la clase también lo mencionamos al pasar.

Los bloques de código son todos aquellos bloques delimitados entre llaves {}. Las variables definidas dentro de cada bloque sólo son visibles dentro del mismo y si se llaman igual que otras de bloques exteriores, aquellos usos dentro del bloque referirán a la variable de ese bloque enmascarando así la variable del mismo nombre del bloque que lo contiene.

En C se pueden anidar tantos bloques como se quiera, por lo general se usan para agrupar sentencias para ser usadas en una sentencia de control o también pueden usarse para simplemente agrupar un conjunto de operaciones en un bloque más grande, aunque este uso no es muy habitual.

NOTA: *vemos que pasa cuando la variable `siguiente` se enmascara con una variable siguiente dentro del bloque del for.*

Ejemplo: ejemplos/fibonacci08/fibonacci.c enmascaramos a propósito la variable siguiente en un bloque interno y vemos que pasa

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

```c
/**
 * calcula la secuencia de fibonacci hasta el índice n y coloca los valores en
 * en fib
 */
void imprimir_fibonacci(int n){
        register unsigned long siguiente, primero = 0, segundo = 1;
        int i;
        for (i=0; i<=n;i++){
                register unsigned long siguiente;
                if (i <= 1) {
                        siguiente = i;
                } else {
                        /*preparamos las variables para el siguiente ejecución*/
                        siguiente = primero + segundo;
                        primero = segundo;
                        segundo = siguiente;
                }
                printf("fib[%d]=%lu\n",i, siguiente);
        }
        printf("valor de siguiente al final:%lu\n", siguiente);


}
```

9 Inicialización de variables

También lo mencionamos al pasar

El estándar C no especifica que hacer cuando una variable es declarada, más que reservarle el lugar necesario para su almacenamiento, inicializar una varaible requiere que el procesador realice al menos una instrucción para ponerle el valor correspondiente, por lo tanto el estandar no obliga a que estas sean inicializadas con algún valor en particular, en cambio le permite al programador inicializarla explícitamente en el momento de la declaración.

A veces y según sus parámetros los compiladores pueden dar algún tipo de warning si la variable es usada sin inicializar, ya que esto es algo peligroso, dado que estaríamos usando un valor que a priori no sabemos que tiene y por lo tanto el programa no se comportará consistentemente en distintas ejecuciones. Una variable no necesita ser inicializada por ejemplo si luego se va a cargar con el valor de una expresión que se evaluará mas adelante en el código.

El estándar C define distintas formas de inicializar las variables según su tipo. Se pueden inicializar arrays, cadenas de texto (`char *`) y tipos numéricos. Más adelante veremos también los tipos punteros (`char *`) de hecho es un caso particular de punteros.

NOTA: *Ver que pasa si no inicializamos las variables primero y segundo, volver a compilar con -Wall para ver que pasa*

10 Recursión [1:30]

Las funciones en C pueden ser llamadas recursivamente, es decir una función puede llamarse a sí misma con distintos parámetros de los que fue llamada o llamar a otras que a su vez la llaman, hay muchos problemas que son mas simples de resolver usando recursión, pero es importante asegurarse que en algún punto la función termine o el programa agotará el stack y terminará con error.

NOTA: *Implementamos fibonacci de forma recursiva para ilustrar, mencioando la ineficiencia del cálculo. Prestamos especial atencion cuando le pasamos un número muy alto de índice.*

11 Preprocesador [1:40]

El preprocesador de C es algo que se ejecuta previo a la fase de compilación, y tiene como cometido interpretar directivas que facilitan la escritura de programas mediante la inclusión de otros archivos y la definición de constantes por ejemplo.

Las líneas que comienzan con # son directivas al preprocesador, la más conocida y utilizada hasta el momento es la directiva #include que incluye el contenido de otro archivo en el proceso de compilación.

En esta clase veremos además la definición de macros con argumentos

11.1 Inclusión de archivos

Hasta ahora hemos usado silenciosamente esta directiva por lo general para incluir declaraciones de funciones de la biblioteca estandar de C y en los ejemplos previos el archivo de encabezado que cumple el mismo propósito.

La directiva `#include` tiene dos formas:

- #include "ruta_archivo"
- #include <ruta_archivo>

En el primer caso nombre es relativo al directorio del archivo fuente, en el segundo caso se busca en una ruta predefinida por la implementación del compilador.

Por lo general usamos los include para incluir archivos de encabezado (.h) que declaran las funciones que se implementan luego en sus respectivos fuentes de implementación (.c)

Es una buena práctica incluir en el archivo de implementación el archivo de encabezado para así no tener discrepancias entre las declaraciones y definiciones de las funciones y variables externas.

11.2 Inclusión condicional [1:45]

A veces es deseable que cierto código se incluya en la compilación si se cumplen ciertas condiciones, para ello el preprocesador de C nos brinda la posibilidad de incluir código selectivamente mediante las siguientes directivas.

- if
- ifdef
- ifndef
- elif
- else
- endif

El uso más importante que le damos a esta posibilidad es asegurarnos de no incluir el mismo archivo de encabezamiento dos veces en la compilación, por ejemplo si dos archivos de encabezamiento se incluyen mutuamente porque necesitan declaraciones uno del otro o porque en el main se incluye uno y otro que incluye a ese uno.

Ejemplo: ejemplos/fibonacci11.2/fibonacci.h ejemplo de inclusión única

```
#ifndef FIBONACCI_H
#define FIBONACCI_H
unsigned long siguiente_fibonacci();
#endif
```

11.3 Substitución de macros [1:55]

Las macros se definen con la directiva `#define` y tiene la forma:

```
#define nombre reemplazo
```

La forma mas simple de macros ya las hemos utilizado para definir constantes, pero tienen uso también para definir operaciones que se realizan frecuentemente pero que no queremos definir como funciones para ahorrar tiempo de ejecución de llamadas a funciones, que implica reservar espacio en el stack para los parámetros y el valor de retorno entre otras cosas.

También se pueden usar para simplificar escritura de códifo tediosa, por ejemplo si queremos simplificar el for para el caso simple de iterar entre dos valores:

Ejemplo: ejemplos/fibonacci11.3/imprimir_fibonacci.c ejemplo de macro con parámetros

```
#include <stdio.h>

#include "fibonacci.h"

#define FOR(i,min,max) for(i=min; i<=max; i++)
int main(){
        int indice=30, i;
    FOR(i,0,indice){
                printf("fib[%d]=%lu\n",i,siguiente_fibonacci());
        }
        return 0;
}
```

Cuidado con los paréntesis en las macros.

Ejemplo: cuadrado

```
#define cuadrado(x) (x)*(x)
```

NOTA: *que pasa si se llama a `cuadrado(a+b)` si no están los paréntesis?*

Cuando usamos macros tenemos que ser cuidadosos dado que si se pasan como parámetros expresiones estas se ejecutan tantas veces como aparecen en la expresión de reemplazo. Ejemplo: max y ++

```
#define max(a,b) ((a)>(b)) ? (a) : (b)
```

NOTA: *qué pasa si se llama por ejemplo `max(i++,j)`?*