Algoritmos y Estructuras de Datos I

Guillaume Hoffmann, Walter Alini

Proyectos 4 y 5

Pasamos de la programación **funcional** (en Haskell) a la programación **imperativa** (en C).

Vamos a seguir usando **funciones**, pero van a tener una forma distinta.

Programación Imperativa

Nociones esenciales:

- variable
- instrucción
- estado
- función

Variables

En Haskell sólo tenemos constantes.

```
let x = 10 in ...
```
El nombre `x` siempre está asociado al valor 10 (en el bloque donde `x` vive).

`x` sirve como nombre para el valor 10.

En programación imperativa (como C), vamos a tener *variables*, cuyos valores pueden cambiar con el tiempo:

```
x = x + 10;
```
Una variable es el nombre de una *ubicación* donde está alojado un valor, que puede cambiar.

Instrucciones

Ejemplo de secuencia de instrucciones:

```
x = 10;
printf("Hola!\n"); /* \n indica una linea nueva */
x = f(20);
x = x + 10;
```
Forma usual de las instrucciones:

- asignación (ej: `x = 10 + 20`)
- llamada sóla de función (ej: `printf("Hola!\n")`)

Cada instrucción en C se termina por **;** .

C no *requiere* indentación, hasta podemos escribir todo en una sola línea.

Pero no sería muy legible.

Flujo de ejecución

Un programa se ejecuta una instrucción después de la otra instrucción.

Esa ejecución secuencial se llama el **flujo de ejecución**.

En cada momento, el flujo está justo antes de una instrucción precisa.

Cuando ejecutamos un programa **C**, el flujo empieza siempre al principio de la función **main**.

Estado

El **estado** es el conjunto de los valores de todas las variables vigentes en algun momento del un programa.

Es un mapeo (función) de valores a **variables**.

Función

Una función tiene un **protótipo**, seguido de una definición.

El protótipo tiene tres partes:

- el tipo del valor que la función devuelve
- el nombre de la función
- la lista de argumentos de la función con sus tipos

```
TIPO_DEVOLUCION NOMBRE( TIPO_1 ARG_1, TIPO_2 ARG_2, ... ) /* protótipo */
 {
  ...  /* definición */
 }
```

Las llaves **{** y **}** delimitan la definición de la función.

Comparación con Haskell: `TIPO_1 -> TIPO_2 -> ... -> TIPO_DEVOLUCION`.

Ejemplo de función en C

```
int funcion( int a, int b, ...)
 {
  int x;   /* declaración de variables al principio */
  int y;
  ...

  /* existen las variables a, b, ... x, y, ... */

  <instrucción>;
  <instrucción>;
  ...

  return x; /* eligimos el valor que sirve como resultado */
 }
```

Declaraciones de variables

Siempre las hacemos al *principio* de las funciones, antes de las instrucciones.

Podemos declarar una variable con un valor inicial:

```
int i = 10, j = 500;
char c = 'a';
```

El tipo `void`

Cuando una función no devuelve ningún valor, escribimos que devuelve el tipo `void`:

```
void funcion( ... )
 {
   ...
 }
```

Cuando una función no recibe ningún argumento, escribimos `void` en lugar de su lista de argumentos:

```
... funcion( void )
 {
  ...
 }
```

Llamada de función (ejemplo)

```
int f(int a)
 {
  ...
 }

int g(void)
 {
  int x;
  x = f(10);     /* siempre poner parentesis para llamar una función */
```

```
  printf("x vale %d\n", x);
 }
```

En el lenguaje **C**, la definición de una función debe estar antes de donde está llamada (acá, no se puede escribir la definición de `f` después la de `g` porque `g` llama a `f`).

Ejemplo clásico: Hello World

```
#include <stdio.h>        /* inclusión biblioteca estandar */

int main(void)            /* devuelve un int, no argumentos */
 {
    printf("Hello World!\n");

    return 0;             /* resultado de la función */
 }
```

`main` tiene que devolver un `int` para decirle al sistema operativo si el programa se ejecutó exitosamente (`0` = éxito).

Compilación

```
gcc -Wall -Werror -pedantic -o main main.c
```

- **gcc**: Compilador de C
  - **G**NU **C**ompiler **C**ollection: compilador libre del proyecto GNU
- las opciones que vienen después se llaman "flags":
  - **-Wall**: Chequea todas las advertencias
  - **-Werror**: Si hay advertencias, no compila
  - **-pedantic**: Prohibe mezclar declaraciones e instrucciones, prohibe funciones anidadas.
  - **-o NOMBRE**: Nombre del archivo ejecutable (`a.out` por defecto)
- el (o los) archivos de entrada:
  - **main.c**: Uno o más archivos de código fuente

Ejecución del programa compilado

Desde una consola:

```
$ ./main
Hello world!
$
```

Salida de datos: la función `printf`

Ejemplo:

```
int i = 0;
printf("Lo leido es: %d\n", i);
```

- Primer argumento: Texto a imprimir, puede incluir **Especificación de conversión** (%d para los `int`).
- Siguientes argumentos: un argumento por cada conversión que haya que realizar (con el valor de esa conversión)

Entrada de datos: la función `scanf`

Ejemplo:

```
int i = 0;
scanf("%d", &i); /* sintaxis rara pero correcta! */
```

El símbolo & antes del nombre de la variable `i` es porque se modifica `i` cuando llamamos a scanf.

Controlar el flujo de ejecución

Con estructuras de control de flujo:

- alternativas: **if**
- ciclos: **while**

Alternativas

```
if (<bool>) {
    <c1>;
}
if (<bool>) {
    <c1>;
} else {
    <c2>;
}
if (<bool>) {
    <c1>;
} else if (<bool>){
    <c2>;
} else {
    <c3>;
}
```

Ejemplo:

```
if (i > 4){
    printf ("Es mayor a 4\n");
}
```

Ciclos

Sintaxis:

```
while (<bool>) {
    <c1>;
}
```

Ejemplo:

```
while (i < 4){
    i = i + 1;
}
```

Operaciones Básicas

- + (suma)
- - (resta)
- * (multiplicación)
- / (división flotante)
- **%** (módulo)
- ==, !=, >, <, ...

Librerías básicas

- **stdio.h**: Provee `printf`, `scanf`, etc.
- **stdlib.h**: Provee funciones matemáticas, funciones de conversión entre charácter y enteros, etc. (ver http://es.wikipedia.org/wiki/Stdlib.h)

Librerías básicas: assert.h

Implementa "assert" para determinar el cumplimiento de suposiciones en el programa:

```
#include <assert.h>
...
assert (i != 4);
```

- "main: main.c:7: main: Assertion `i != 4' failed. Aborted"
- Puede usarse para comprobar que una precondición se satisface antes de llamar a una función
- Puede usarse para comprobar que una postcondición se satisface después de llamar a una función
- Para cualquier otra suposición que estemos seguros que se tenga que cumplir (caso contrario, el programa no tiene sentido que continúe)

Diferencia con formalismo básico: asignación múltiple

```
|[ var   a,b : Num
   { True }
   a := 4
   b := 2
```

```
    a , b := b + 1, a + 1
    { a = 3  ∧  b = 5 }
]|
```

El lenguaje C *no tiene* asignación múltiple, hay que usar variables suplementarias.

Traducción a C: bucle, guardas

Ejemplo con bucle:

```
|[ var   x : Num
   con   X : Num
   { X > 0  ∧  x = X }
   do    x < 10   →   x := x + 1
   od
   { x >= 10 }
]|
```

Traducción a “C”:

```
  int x;
  int xInput;

  printf("Ingrese un entero positivo\n");
  scanf("%d",&xInput);
  x = xInput;

  assert(xInput > 0  && x == xInput);

  while ( x < 10){
    x = x + 1;
  }
  assert( x >= 10 );
```

Cálculo de Programas, p.262:

```
|[ var   x,y  : Num
   con   X,Y  : Num
   { X > 0   ∧  Y > 0  ∧  x = X  ∧  y = Y }
   do   x > y  →  x := x - y
    □   y > x  →  y := y - x
   od
   { x = mcd.X.Y }
]|
```

Traducción a “C”:

```
  int x,y;
  int xInput,yInput;

  printf("Ingrese x\n");
  scanf("%d",&xInput);
  x = xInput;

  printf("Ingrese y\n");
  scanf("%d",&yInput);
  y = yInput;

  assert(   xInput > 0  && x == xInput
         && yInput > 0  && y == yInput);

  while ( x > y  || y > x ){
      if ( x > y ){
         x = x - y;
      } else if ( y > x ){
         y = y - x;
      }
  }
```

Parte 2: Entender la ejecución de un programa

A veces, leer el código fuente no es suficiente. Hay herramientas para ayudarnos a entender un programa y arreglar los errores posibles:

- poner `printf` en todos lados (puede ser tedioso)
- usar un debugger como GDB (más general)

GDB (GNU Debugger)

- Para usar GDB, pasar el flag **-g** a gcc:
  gcc -Wall -Werror -pedantic -g -o main main.c

- Eso le agrega al ejecutable información para deboguear el programa (código fuente)
- Para arrancar: `gdb ./main`
- `gdb` es una interfaz interactiva, donde se entran comandos

Comandos de GDB

help COMANDO	explicar lo que hace COMANDO
list	listar código
break LINEA	poner un breakpoint en la línea LINEA
continue	seguir ejecutando hasta el próximo breakpoint
run	empezar ejecución del programa
step	ejecutar una línea de código
next	ejecutar una línea de código (sin entrar a funciones)
print VARIABLE	imprimir el valor de VARIABLE
display VARIABLE	imprimir el valor de VARIABLE cada vez que para la ejecución
q o `CTRL+D`	salir

Ejemplo: collatz.c

- Bajar el programa de la página de la materia
- Usar el programa para indentar el código fuente:

```
indent collatz.c
# después de esto, mirar collatz.c
indent -kr collatz.c
# otro estilo de identación, mirar de nuevo collatz.c
```

- Identificar los bloques del programa (funciones, while, if)
- Compilar, ejecutar, probar
- Compilar con -g, gdb
- Correrlo con gdb con un breakpoint dentro del bucle while, mostrando los valores de las variables