Definición de acciones no primitivas I.

José Otero

[1] Departamento de informática
Universidad de Oviedo

13 de diciembre de 2008

1. Recursividad.
2. Tiempo de vida de una variable.
3. Modularidad y ocultación de la información.

1 Recursividad.

2 Tiempo de vida de una variable.

3 Modularidad y ocultación de la información.

- No hay restricciones a las acciones que pueden escribirse en el cuerpo de una función.
- Es posible escribir en el cuerpo de una función llamadas a la misma función.
- Una llamada a una función desde la misma función se denomina **llamada recursiva**
- Se dice que **la función es recursiva** o bien que **utiliza la recursividad**.

La recursividad es la forma más sencilla de escribir en un lenguaje de programación la definición de acciones inductivas. Ejemplo: cálculo del factorial de un número natural:

$$x! = \begin{cases} 1 \text{ si } x=0 \\ x(x-1)! \text{ si } x>0 \end{cases}$$

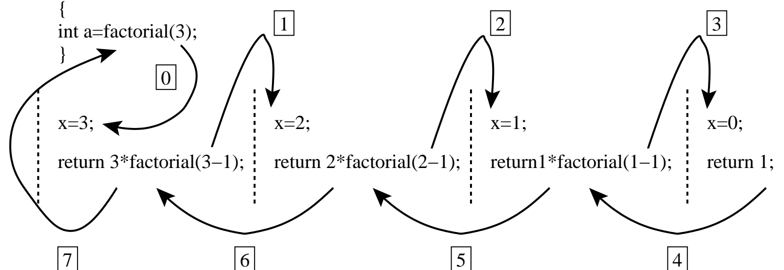
```
int factorial(int x)
{
if (x==0) return 1;
else return x*factorial(x-1);
}
```

En la diapositiva 8 se muestra un diagrama con las sucesivas llamadas recursivas a la función factorial y el retorno de las mismas. Los pasos que sigue el programa se numeran desde 0 a 7. A continuación se explica cada uno de ellos:

- Paso 0: el valor del parámetro real se copia en el parámetro formal x. Como su valor no es cero, se ejecuta la sentencia incluida en el `else`. Dicha sentencia incluye una llamada recursiva a `factorial` con el valor 2.
- Paso 1: el valor 2 se copia en x, del mismo modo que en el caso anterior, se ejecuta la sentencia incluida en el `else`, produciéndose otra llamada recursiva a `factorial`, en este caso con el valor 1.
- Paso 2: el valor 1 se copia en x, como ocurría en el caso anterior, se ejecuta la sentencia incluida en el `else`, produciéndose otra llamada recursiva a `factorial`, en este caso con el valor 0.

- Paso 3: el valor 0 se copia en x. En este caso se cumple la condición del `if`, de modo que se retorna 1 y no se efectúa ninguna llamada recursiva.
- Paso 4: el valor de retorno 1 sustituye a la llamada `factorial(1-1)`, de modo que se puede calcular la expresión `1*factorial(1-1)`, su valor es 1.
- Paso 5: el valor de retorno 1 sustituye a la llamada `factorial(2-1)`, de modo que se puede calcular la expresión `2*factorial(2-1)`, su valor es 2.
- Paso 6: el valor de retorno 2 sustituye a la llamada `factorial(3-1)`, de modo que se puede calcular la expresión `3*factorial(3-1)`, su valor es 6.
- Paso 7: el valor de retorno 6 sustituye a la llamada `factorial(3)` y se asigna a la variable `a`, el programa termina.

```
#include<iostream>
using namespace std;
int factorial(int x)
{
if (x==0) return 1;
else return x*factorial(x−1);
}
main()
```



- La forma iterativa y recursiva del factorial utilizan las mismas operaciones.
- La forma recursiva emplea mayor cantidad de memoria (el paso del parámetro es por valor).
- Además existe el coste añadido de las sucesivas llamadas,
  - Conllevan un determinado tiempo de procesador.
- Una acción no es recursiva o iterativa intrínsecamente: para cada acción recursiva existe una acción iterativa con su misma especificación, y viceversa.
- Es frecuente codificar de forma recursiva una acción cuando su versión iterativa es muy compleja.

En general, para resolver un problema *P* sobre unos datos *D*, el programador/a se pregunta:

- ¿Es posible resolver *P* sobre los datos *D* suponiendo que ya está resuelto para otros datos *D'* del mismo tipo que *D*?
- Además en un sentido bien definido,¿son más sencillos?

Se trata de encontrar una relación de recurrencia en los datos del problema que permite calcular la solución pedida recurriendo a la solución para datos más simples.

Ejemplo: calcular a^n.

- Los datos son el par *D=(a,n)* y el problema a resolver *P((a,n))=*a^n.
- Planteamiento recursivo: resolver *P* sobre unos datos *D'* más pequeños que *D*: *D'=(a,n-1)*. Si la solución de *P* sobre *D'* es conocida (*P(D')=*a^{n-1}), la solución de *P* sobre *D* es *P(D)=a·P(D')*.
- Este razonamiento que conduce a calcular *P* sobre *D* en función de *P* sobre *D'* puede aplicarse para resolver *P* sobre *D'* en función de unos datos aún más pequeños *D"*, etc.
- Se forma entonces una sucesión *D>D'>D">...* de datos cada vez más pequeños.

- Para que el razonamiento sea correcto se requiere que la sucesión sea finita.
- Se llegará entonces a unos datos D^t lo suficientemente pequeños como para resolver P directamente sin tener que resolver a soluciones de P sobre otros datos.
- Entonces D^t es un caso trivial (en el ejemplo anterior, $D^t = (a, 0)$ y $P(D^t) = 1$).
- Cuando D no es lo suficientemente simple como para ser resuelto directamente, diremos que es un caso recursivo o no trivial.

El aspecto de un programa recursivo es un reflejo de este análisis: Habrá una o más instrucciones condicionales dedicadas a separar los tratamientos correspondientes a los casos triviales de los correspondientes a los casos no triviales. Los primeros tienen el aspecto de un programa convencional, mientras que en los segundos se pueden distinguir las siguientes partes:

- Primero se calculará el sucesor *D'* de *D*. También diremos que se produce una descomposición recursiva de los datos *D* para obtener los datos más sencillos *D'*. A veces nos referiremos a *D'* diciendo que es un subproblema de *D*.
- A continuación se produce una llamada recursiva para obtener la solución del subproblema *D'*.
- Finalmente, se opera sobre los resultados obtenidos para *D'* a fin de calcular la solución *D*.

En el caso del ejemplo anterior, la solución en C++ sería:

```
int potencia(int a, int n)
{
if (n==0)
   return 1;
else
   return a*potencia(a,n-1);
}
```

- La condición `n==0` identifica el caso trivial, cuya solución se conoce.
- En el caso no trivial la función ha de recurrir al resultado de aplicar la función a datos más pequeños.
- Toda función recursiva tiene al menos un caso trivial.

1 Recursividad.

2 Tiempo de vida de una variable.

3 Modularidad y ocultación de la información.

- Por defecto, las variables declaradas dentro de un bloque son **automáticas**, se crean en el momento de la declaración y se destruyen al salir del bloque..
- Existe la posibilidad de hacer que las variables no desaparezcan tras finalizar la acción en donde se han declarado y que mantengan su valor entre llamadas a una misma función. Anteponiendo a la declaración de la variable `static`.
- Las variables estáticas se inicializan al valor por defecto (pej. las numéricas a 0)
- Es posible inicializarlas con otro valor.
- Las variables automáticas no tienen un valor inicial definido por defecto.

El siguiente ejemplo muestra por la pantalla la suma de los valores que se van pasando como argumento a la función en las sucesivas llamadas.

```
#include<iostream>
using namespace std;
void acumulado(int x)
{
static int y;
y=y+x;
cout<<y<<endl;
}
int main()
{
for (int i=1;i<=10;i++)
  acumulado(i);
}
```

Mostraría:

```
1
3
6
10
15
21
28
36
45
55
```

1. Recursividad.

2. Tiempo de vida de una variable.

3. Modularidad y ocultación de la información.

- Cuando un programa es muy extenso o está escrito por varias personas, es deseable disponer de variables, constantes, funciones, etc. que sean visibles en una parte del programa e invisibles en el resto.
- Esto permite por ejemplo, que un programador no tenga que preocuparse de llamar a sus funciones de forma distinta a las de otros programadores, y lo que es más importante, permite que el programador no necesite conocer esos nombres.
- C++ permite la definición de **módulos**, que consiste precisamente en la división de un programa en espacios de modo que todo lo que se declare en ellos es visible en su interior e invisible en el resto del programa.
- Los módulos en C++ se definen mediante la palabra reservada `namespace`.

- Si se define y declara una función dentro de un módulo, la sintaxis es la que ya se ha explicado.
- Si se declara dentro y se define fuera (por claridad), hay que identificar la función con el nombre del módulo.
- Por ejemplo, dada la declaración
  `T una_función(T1,T2,...);`
  en un módulo llamado `un_modulo`
  en la definición se escribirá
  `T un_modulo::una_funcion(T1 p1,T2 p2,...){..}`.
- El '::' se denomina **operador de resolución de ámbito**.

Si una función definida dentro de un módulo se utiliza en `main`, es necesario hacer una de las siguientes cosas:

- Denotar a la función con su nombre y el del módulo separados por '::', por ejemplo `un_modulo::una_funcion(....`
- Escribir antes de usar la función la sentencia: `using un_modulo::una_funcion`. Desde ese momento si escribimos `una_funcion` será un sinónimo de `un_modulo::una_funcion`.
- Escribir antes de usar la función la sentencia: `using namespace un_modulo`. Desde ese momento si escribimos el nombre de una función definida en `un_modulo` será un sinónimo de `un_modulo::nombre_funcion`.

En el siguiente programa se muestran ejemplos de lo comentado aquí.

```
#include<iostream>
using namespace std;
//definicion del modulo
namespace un_nombre{
   //definicion de una funcion dentro
   //del modulo
   float una_funcion(float a)
   {
   return a*a;
   }
   //declaracion de una funcion dentro
   //del modulo
   float otra_funcion(int, float);
}
//definicion de una funcion del modulo
//fuera del modulo
float un_nombre::otra_funcion(int a, float b)
{
if (a==0) return b*b;
return b*b*b;
}
namespace otro_nombre{
   float una_funcion(float a)
   {
   return a*a*a+a*a+a+1;
   }
}
```

```
int main()
{
float x;
//llamada a una_funcion de otro_nombre
x=otro_nombre::una_funcion(1.2);
//llamada a una_funcion de un_nombre
x=un_nombre::una_funcion(7.3);
//se usa otra_funcion es de un_nombre,
//no hacen falta los ::
using un_nombre::otra_funcion;
x=otra_funcion(2,3.6);
//se van a usar las funciones de un_nombre
using namespace un_nombre;
x=una_funcion(7.3);
x=otra_funcion(2,3.6);
}
```

- La razón de escribir:

  ```
  using namespace std;
  ```

  al principio de los programas que hemos visto es que `cin` y `cout` están definidos en un `namespace` que se llama `std`
- En caso contrario tendríamos que escribir siempre:

  ```
  std::cin
  std::cout
  ```

  En lugar de `cin` y `cout` respectivamente.