

Definición de acciones no primitivas I.

José Otero

¹Departamento de informática
Universidad de Oviedo

27 de noviembre de 2008

1 **Ámbito de una declaración.**

2 **Funciones y operadores.**

- Funciones.
- Operadores.
- Sobrecarga.

3 **Referencias**

4 **Punteros**

5 **Mecanismos y semántica de paso de argumentos.**

- Una variable puede declararse en cualquier parte de un programa.
 - Siempre antes de ser usada.
- La porción de código fuente en donde puede utilizarse es el ámbito de esa variable.
- Si se declara dentro de un bloque, sólo puede utilizarse dentro de ese bloque. Es *local* a ese bloque.
- Si se declara fuera de cualquier bloque, la variable es *global* y puede utilizarse desde su declaración hasta el fin del fichero fuente en donde se ha declarado.
 - Se desaconseja el uso de variables globales que no sean constantes.

```
#include<iostream>
using namespace std;
int main()
{
  int i=3;
  if (i>2)
  {
    int j;
    j=1;
    cout<<i<<" "<<j<<endl;//correcto
  }
  //cout<<j<<endl; incorrecto, j no definida
  cout<<i<<" "<<endl;//correcto
}
```

- Se pueden declarar variables de igual nombre en bloques anidados.
- La declaración más reciente ensombrece a la anterior.

```
#include<iostream>
using namespace std;
int main()
{
int i=3,j=7;
if (i>2)
{
int j;//esta declaracion ensombrece la anterior
j=1;
cout<<j<<endl;//correcto, muestra 1
}
cout<<j<<endl;//muestra 7
}
```

- Un cálculo puede ser necesario repetirlo varias veces en un mismo programa.
 - Posiblemente con distintos datos.
- Sería útil poder:
 - Implementar una sola vez el algoritmo del cálculo.
 - Especificar cuales son los datos.
 - Especificar como se devuelve el resultado.
 - Referirse al algoritmo mediante un nombre.
 - Poder utilizarlo las veces que sea necesario.
- En ocasiones es más conveniente representar un cálculo mediante un signo (+-*/*...) que se aplica a los datos.

- Una función es una porción de código fuente que realiza un cálculo.
- Antes de poder usar una función, tiene que ser declarada y definida.
- Declaración:

```
tipo_resultado nombre(tipo1, tipo2, ...);
```

 - `tipo_resultado` es el tipo del valor que produce la función.
 - `nombre` es un identificador.
 - `tipo1, tipo2`, son los tipos de los parámetros formales (datos) de la función.
 - `tipo_resultado, tipo1, tipo2` pueden ser de cualquier tipo predefinido en C++ o definido por el programador/a.
 - Además `void` denota la ausencia de resultado o dato.

- Definición:

```
tipo_resultado nombre(tipo1 p1, tipo2 p2, ...)
{
//calculos con p1, p2...
...
return expresion_de_tipo_resultado;
}
```

- `p1, p2, ...` son los nombres de los parámetros formales.
 - Dentro de la función se utilizan como si fueran variables declaradas entre las {}, es decir son locales a la función.
 - Si `tipo_resultado` no es `void` debe de haber al menos una sentencia `return` que devuelve el valor de la expresión que va a continuación.

NOTAS:

- Puede haber varias sentencias `return`.
 - La función termina cuando se ejecute una de ellas.
- Sólo se devuelve un único valor con `return`, un `int`, `float`, `char`, `vector` ... no varios de esos valores.
- Los parámetros y las variables declaradas entre las `{ }` son locales a la función y no pueden usarse fuera de ese bloque.

- La llamada a la función consiste en escribir su nombre, seguido de los valores de los argumentos (parámetros reales), entre paréntesis y separados por comas.
`nombre(valor1, valor2, ...)`
- Los tipos de `valor1, valor2, ...` deben coincidir con los de `p1, p2, ...`
- El número de parámetros reales debe coincidir con el número de parámetros formales.
- Si no se especifica nada en contra, los valores de los parámetros reales se copian en los parámetros formales.
- La llamada se puede escribir en cualquier lugar en donde se pueda escribir un valor del tipo que devuelve la función.
- Se pueden escribir cuantas llamadas se necesiten.

```
#include<iostream>
using namespace std;
//declarar, usar, definir
tipo_resultado nombre(tipo1, tipo2, ...);
int main()
{
  tipo_resultado a;
  ...
  a=nombre(valor1, valor2, ...);
  ...
}
tipo_resultado nombre(tipo1 p1, tipo2 p2, ...)
{
  //calculos con p1, p2...
  ...
  return expresion_de_tipo_resultado;
}
```

```
#include<iostream>
using namespace std;
//definir y declarar, usar
tipo_resultado nombre(tipo1 p1, tipo2 p2, ...)
{
  //calculos con nombre1, nombre2...
  ...
  return expresion_de_tipo_resultado;
}
int main()
{
  tipo_resultado a;
  ...
  a=nombre(valor1, valor2, ...);
  ...
}
```

- Cualquier cálculo imaginable puede descomponerse en llamadas a funciones, operaciones de asignación o inclusión de las llamadas a funciones en expresiones.
- Sin embargo la lista de acciones necesaria en estos casos puede ser larga, engorrosa o difícil de entender.

```
#include<iostream>    int main()
using namespace std; {
T suma(T a, T b)      T a,b,c,d,e;
{
...
T r;                  //equivale a a=b+c+d+e
//calculos con a y b a=suma(b,suma(c,suma(d,e)));
...
return r;            }
}
```

- Definir un operador es muy similar a definir una función: el nombre ha de ser obligatoriamente el operador a definir precedido de la palabra `operator`.
- En el caso de los operadores binarios, en la llamada, los parámetros se escriben a ambos lados del signo que se redefine.

```
#include<iostream>    int main()
using namespace std; {
T operator+(T a, T b) T a,b,c,d,e;
{
...
T r;                  //a=operator+(b,operator+(c,
//calculos con a y b // operator+(d,e)));
...
return r;            a=b+c+d+e;
}
}
```

- los operadores que se pueden definir son los siguientes, que ya existen en C++:

+	-	*	/	%	^	&
	~		=	<	>	+=
-=	*=	/=	%=	^=	&=	=
<<	>>	>>=	<<=	==	!=	<=
>=	&&		++	--	->*	,
->	[]	()	new	new[]	delete	delete[]

NOTA: El significado de muchos de ellos no se verá hasta más adelante o no se verá en absoluto.

NOTAS:

- Se mantienen la precedencia y la asociatividad original de los operadores, aunque se redefinan.
- Es buena práctica de programación escoger los mismos operadores para representar las mismas operaciones con nuevos tipos de datos.
 - Por ejemplo, el `*` o la `+` para representar esas operaciones con matrices, y NO `*` para la suma y `+` para el producto.

- Es posible definir funciones u operadores que realicen distintos cálculos en función del
 - tipo o del
 - número de argumentos.
- Este tipo de comportamiento se denomina **sobrecarga**.
- Las funciones y operadores definidos por el usuario pueden sobrecargarse.

NOTA: No es válido que la única diferencia sea el valor de retorno.

```

tipox nombre(tipo1 p1,      int main()
                tipo2, p2)  {
{
...//A                      tipo1 a;
}                             tipo2 b;
tipoy nombre(tipoa pa,     tipoa c;
                tipob pb)  tipob d;
{                             tipo e;
...//B                      //llama a A
}                             ...nombre(a,b)...
tipoz nombre(tipo p)      //llama a B
{                             ...nombre(c,d)...
...//C                      //llama a C
}                             ...nombre(e)...
}                             }

```

```

#include<iostream>          float suma(float a,
using namespace std;      float b)
int suma(int a, int b)    {
{
return a+b;              }
}
float suma(float a,       int main()
float b,                  {
float c)                  int a=1,b=2,c;
{                          float x=33.1,y=0.1,
return a+b+c;            z=-9.2,u;
}                          c=suma(a,b);
}                          u=suma(x,y,z);
                          u=suma(x,y);
                          }

```

Hasta el momento un objeto de almacenamiento sólo se podía modificar:

- Mediante una asignación, utilizando el nombre de la variable correspondiente.
- Mediante una operación de entrada, utilizando `cin` y el nombre de la variable correspondiente.

Además cada objeto de almacenamiento recibía un único nombre.

Es posible dar varios nombres a un mismo objeto de almacenamiento.

- Los nombres alternativos se denominan referencias.
- Se puede decir que son sinónimos.

Sintaxis:

```
tipo &sinonimo=nombre;
```

Ahora se puede utilizar `sinonimo` o `nombre` indistintamente.

NOTA: es obligatorio inicializar las referencias al declararlas.

```
#include<iostream>
using namespace std;
int main()
{
int a;
a=6;
int &c=a;
c=7;
cout<<a<<endl;
}
```

El programa anterior mostraría por la pantalla el valor 7, ya que `c` es un sinónimo de `a` y por lo tanto alterar el valor de `c` es lo mismo que alterar el valor de `a`.

- Los distintos tipos de datos permiten el almacenamiento de valores de esos tipos en la memoria del ordenador.
- Cada tipo se codifica de una forma distinta.
- Hemos visto como se accede mediante uno o varios nombres a esos valores.
- No está bajo el control del programador/a la ubicación en donde se van a almacenar esos valores.
- C++ posee mecanismos para:
 - Acceder al contenido de las posiciones de memoria del ordenador.
 - Saber en que dirección de memoria está almacenado el valor de determinada variable.

- Cada tipo de dato se codifica de distinta forma.
- Puede ocupar distinta cantidad de memoria.
- Es necesario conocer el tipo del valor para poder acceder a su valor a través de la dirección de memoria en la que se almacena.
- Tiene sentido distinguir las direcciones de memoria según el tipo de dato que se almacene en ellas.
- Si una variable almacena una dirección de memoria en donde se almacena un valor de un tipo de dato `T`, el tipo de dato de esa variable es **puntero a T**.

Una variable en donde se almacenan direcciones de memoria que ocupan valores de tipo `T` se declara:

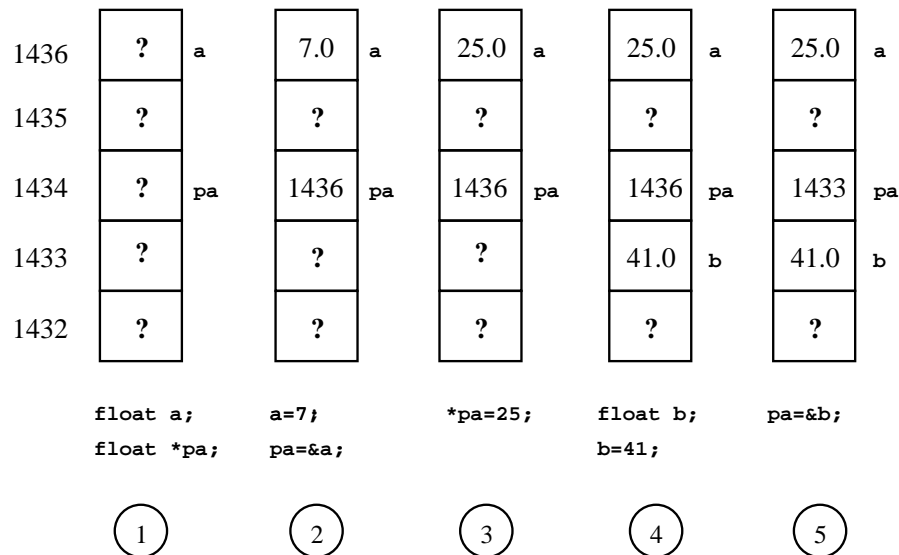
```
T *nombre;
```

Existen dos operadores relacionados con estas cuestiones.

- El **operador dirección** se representa con el signo `&`. Si `a` es el nombre de una variable, `&a` es la dirección de memoria en donde se almacena `a`.
- El operador `*` se denomina **operador indirección**. Si `pa` es una variable de tipo puntero a `T` (se ha declarado `T *pa;`), `*pa` es el valor de tipo `T` que está codificado en la dirección de memoria `pa`.

NOTA: Por el momento sólo se asignarán las variables de tipo puntero desde el resultado de aplicar `&` a variables de los tipos vistos. No se deben asignar valores arbitrarios a las variables de tipo puntero.

```
#include<iostream>
using namespace std;
int main()
{
float a;
float *pa;
a=7;
pa=&a;//*pa es sinonimo de a
*pa=25;
cout<<*pa<<' '<<a<<endl;
float b;
b=41;
pa=&b;//*pa es sinonimo de b
cout<<*pb<<' '<<b<<' '<<a<<endl;
}
```



Los parámetros formales de una función pueden ser:

- **Variables.** Los parámetros formales se declaran como una variable normal:
`tipo nombref(tipo1 p1, tipo2 p2,...){...}`
 La llamada sería:
`...nombref(exp_tipo1, exp_tipo2,...)...`
- **Referencias.** Los parámetros formales son referencias:
`tipo nombref(tipo1 &p1, tipo2 &p2,...){...}`
 La llamada sería:
`...nombref(var_tipo1, var_tipo2,...)...`
- **Punteros.** Los parámetros formales son punteros:
`tipo nombref(tipo1 *p1, tipo2 *p2,...){...}`
 La llamada sería:
`...nombref(&var_tipo1, &var_tipo2,...)...`

NOTA: Además pueden ir precedidos de `const`.

Si los parámetros formales son variables no constantes:

- Los parámetros reales se copian en los formales.
- Serán sólo de entrada pero se pueden modificar: en el cuerpo de la función pueden aparecer sentencias que modifiquen el valor de los parámetros formales (las copias) pero no tienen efecto sobre los reales.

```
#include<iostream>      int main()
using namespace std;    {
int f(int a, int b)     int x=1,y=2;
{                       cout<<f(x,y)<<' ';
a=b*2;                 cout<<x<<' ' <<y<<endl;
b=a+1;                 }
return a*b;
}
```

El programa anterior muestra 20 1 2

Si los parámetros formales son variables constantes:

- Los parámetros reales se copian en los formales.
- Serán sólo de entrada, se copian y las copias no se pueden modificar: si en el cuerpo de la función aparecen sentencias que modifiquen el valor de los parámetros formales (las copias) el programa no compila.

```
#include<iostream>      int main()
using namespace std;    {
int f(const int a,      int x=1,y=2;
        const int b)   cout<<f(x,y)<<' ';
{                       cout<<x<<' ' <<y<<endl;
a=b*2;//ERROR          }
b=a+1;//ERROR
return a*b;
}
```

El programa anterior no compila.

Si los parámetros formales son referencias no constantes:

- Los parámetros formales son sinónimos locales de los parámetros reales.
- Si se modifican los parámetros formales se modifican los parámetros reales.

```
#include<iostream>      int main()
using namespace std;    {
int f(int &a, int &b)   int x=1,y=2;
{                       cout<<f(x,y)<<' ';
a=b*2;                 cout<<x<<' ' <<y<<endl;
b=a+1;                 }
return a*b;
}
```

El programa anterior muestra 20 4 5

Si los parámetros formales son referencias constantes:

- Los parámetros formales son sinónimos locales de los parámetros reales.
- Es un error escribir sentencias que modifiquen los parámetros formales.

```
#include<iostream>      int main()
using namespace std;    {
int f(const int &a,    int x=1,y=2;
        const int &b)   cout<<f(x,y)<<' ';
{                       cout<<x<<' ' <<y<<endl;
a=b*2;//ERROR          }
b=a+1;//ERROR
return a*b;
}
```

El programa anterior no compila.

Si los parámetros formales son punteros no constantes:

- Los parámetros formales precedidos de * son sinónimos locales de los parámetros reales.
- Si se modifican los parámetros formales se modifican los parámetros reales.

```
#include<iostream>      int main()
using namespace std;    {
int f(int *a, int *b)   int x=1,y=2;
{                       cout<<f(&x,&y)<<' ';
*a=*b*2;               cout<<x<<' ';<<y<<endl;
*b=*a+1;               }
return *a*b;
}
```

El programa anterior muestra 20 4 5

Si los parámetros formales son punteros constantes:

- Los parámetros formales precedidos de * son sinónimos locales de los parámetros reales.
- Es un error escribir sentencias que modifiquen los parámetros formales.

```
#include<iostream>      int main()
using namespace std;    {
int f(const int *a,     int x=1,y=2;
        const int *b)   cout<<f(&x,&y)<<' ';
{                       cout<<x<<' ';<<y<<endl;
*a=*b*2;//ERROR        }
*b=*a+1;//ERROR
return *a*b;
}
```

El programa anterior no compila.

Resumen y recomendaciones:

- Lo que lleva `const` no se puede modificar.
- Las variables normales se copian.
- Referencias y punteros son sinónimos.
- Los tipos simples suelen pasarse como variables normales. Este modo de paso se denomina "por valor".
- Los vectores y otros tipos de datos que ocupan gran cantidad de memoria suelen pasarse como referencias por cuestiones de eficiencia. Este modo de paso se denomina "por referencia".
- Si una función produce varios resultados de distinto tipo, pueden devolverse en varios parámetros que se pasen por referencia (como suele hacerse en C++) o usando punteros (como se hacía antiguamente en C).

Advertencias:

- Las referencias son una innovación de C++ frente a C.
- Son más sencillas de utilizar:
 - El símbolo & sólo se usa en la declaración de las referencias.
 - La asignación o el paso de una variable a una referencia vincula el sinónimo al nombre original.
 - El uso del sinónimo y del nombre original es idéntico.
- En algunos casos sigue siendo imprescindible el uso de punteros. Recordar:
 - En la declaración, el * se usa para indicar que el identificador es un puntero.
 - En las expresiones, el * se usa para indicar que estamos accediendo al contenido de la dirección de memoria.
 - El & aplicado a una variable devuelve un puntero al tipo de la variable conteniendo la dirección de memoria en donde se almacena la variable.
 - La vinculación del identificador con su sinónimo se hace mediante la asignación del puntero obtenido al aplicar el operador & al identificador original a la variable de tipo puntero.
 - El sinónimo es la variable puntero precedida de *.