Editorial de la Universidad Tecnológica Nacional

Ciclo de Seminarios y Talleres del Área de Programación

Parámetros y Argumentos en el Lenguaje de Programación C++

Ing. Ubaldo José Bonaparte

Departamento de Sistemas de Información
Facultad Regional Tucumán
Universidad Tecnológica Nacional
U.T.N. - Argentina

Editorial de la Universidad Tecnológica Nacional - edUTecNe
http://www.edutecne.utn.edu.ar
edutecne@utn.edu.ar

2- Ciclo de Seminarios y Talleres del Area de Programación.
Parámetros y Argumentos en el lenguaje de programación C++.

Prologo:

Considerando la necesidad de una comunicación fluida entre los docentes, fundamentalmente entre profesores del Area de Programación[1], de la carrera de Ingeniería en Sistemas de Información, como integrante[2] de la cátedra de Paradigmas de Programación eleve el proyecto "Ciclos de Seminarios y Talleres del Area de Programación". Dicho proyecto contempla la exposición de temas incluidos en la curricula o bien de actualidad, y el debate entre los presentes con el fin de mejorar nuestra didáctica, contenidos teórico-prácticos y bibliografías.

Partiendo de esta base, de gran importancia hacia un futuro perfeccionado de nuestros estudiantes, y desde luego de nuestros docentes, creo indispensable que las palabras expresadas en los seminarios queden documentadas a partir de escritos de este tipo. Detallando conceptos y ejemplificándolos de modo que toda persona relacionada a esta casa de altos estudios, pueda consultar y enriquecer sus conocimientos.

En estas líneas procuro introducir al lector a un tema muy específico dentro de la programación utilizando el lenguaje C++. Simplemente apunto a lectores con conocimientos básicos en dicho lenguaje y con ganas de aclarar o bien perfeccionar sus programas haciendo un buen uso de estas herramientas denominadas parámetros o argumentos.

[1] Departamento de Ingeniería en Sistemas de Información, Facultad Regional Tucumán
[2] Profesor Asociado Ordinario, Jefe de cátedra.

Indice

4- Ciclo de Seminarios y Talleres del Area de Programación.
Parámetros y Argumentos en el lenguaje de programación C++.

Introducción:

Cuando comenzamos a vincularnos con el aprendizaje del desarrollo de programas o bien con algún lenguaje de programación en particular, incorporamos una serie de conceptos nuevos que no formaban parte de nuestros días cotidianos. Lógica, variables, constantes, estructuras de datos, archivos, subprogramas, funciones etc. etc. Entendemos la secuencialidad de la ejecución de nuestros comandos o sentencias de programas, captamos los conceptos de input y output, hacemos nuestros primeros programas, continuamos creándolos y quizá después de unos cincuenta expresamos saber programar.

Ahora bien, siempre he pensado que el arte de programar requiere de un alto porcentaje de conocimientos sobre las herramientas que nos brinda el lenguaje de programación con el que trabajamos. Eso por un lado. Por otro, nuestra creatividad debe estar motivada y sustentada sobre una base muy sólida de detalles madurados sobre lo que tenemos que programar. Así también, siempre pensamos que nuestros programas deben ser amigables con el usuario, sólidos, consistentes y robustos.

Cuantas veces consultamos con colegas o bibliografía, sobre detalles menores o muy importantes ya que no visualizamos como aplicar nuestras herramientas para un fin determinado. Desde luego aprendemos constantemente y lo mejor de todo, es que nuestros proveedores de lenguajes de programación no dejan de brindarnos tecnología nueva que nos exija dedicación, estudio y mucha práctica.

Este seminario tiene la bondad de hacer un aporte sobre el paradigma procedimental, como parte de la programación imperativa puntualmente sobre parámetros y argumentos en funciones desarrolladas en el lenguaje de programación C++. Siempre procurando trabajar sobre la formación del programador en cuanto al orden y manejo, que debe tener sobre las variables y sus ámbitos en un programa. Y tratando de evitar los efectos colaterales que produce el perder control sobre las mismas.

Los parámetros y argumentos forman parte esencial para la comunicación entre funciones o procedimientos de nuestros programas.

Ya que por paradigma, es prohibitivo el manejo de variables globales, que si podrían ser reconocidas desde procedimientos y funciones, nuestro recurso de comunicación se acrecienta a partir de parámetros y argumentos. O sea, el ámbito de nuestras variables debe ser la función donde fueron declaradas y si necesitamos dichas variables en otras funciones, haremos uso de parámetros y argumentos para disponer de ellas adecuadamente.

Entonces, a partir de parámetros y argumentos vamos a respetar y extender el ámbito de variables locales?. Si, eso trataremos de lograr y nos olvidaremos de gran parte de los efectos colaterales que las variables pueden producir.

Un ejemplo simple para analizar

Cargar los valores de un vector de números enteros, mostrar sus valores y obtener la suma de dichos valores, significaría desarrollar el siguiente programa.

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

int arreglo[5] = {0,0,0,0,0}; // variable global inicializada

void carga(){
  int i;
  for (i = 0; i < 5; i++)
   cin >> arreglo[i];
}

void muestra(){
  for (int i = 0;i < 5;i++)
    printf("%d,",arreglo[i]);
}

int suma(){
  int i,valor;
```

```
  valor = 0;
  for (i = 0;i < 5;i++)
    valor += arreglo[i];
  return(valor);
}

void main(){
  carga();
  muestra();
  printf("\nLa suma de los elementos es = %d\n",suma());
}
```

La variable arreglo, de tipo vector de enteros es global al programa. Significa que podemos disponer de ella en cualquier función, para observar u obtener sus datos o bien para modificarlos. Las variables i de tipo entero, que utilizamos para variar los subíndices, tienen sus ámbitos en cada función; y porque no, podría haber sido global. Quien expresaría que este programa esta mal escrito, que tiene efectos colaterales?. Dada su magnitud, extensión, cantidad de líneas y simplicidad, este programa no estaría mal escrito. Pero si aportaría sobre los malos hábitos de un futuro programador.

Mejoremos nuestro primer ejemplo.

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

typedef int arreglo[];   // declaración de un tipo arreglo de enteros

void carga(arreglo &a){
  for (int i = 0;i < 5;i++)
     cin>>a[i];
}

void muestra(arreglo b){
  for (int i = 0;i < 5;i++)
    printf("%d,",b[i]);
}
```

```
int suma(arreglo c){
  int valor = 0;
  for (int i = 0;i < 5;i++)
    valor += c[i];
  return(valor);
}

void main(){
  arreglo vector = {0,0,0,0,0};
  carga(vector);
  muestra(vector);
  printf("\nLa suma de los elementos es = %d\n",suma(vector));
}
```

Este programa funciona casi como el ejemplo anterior, para el usuario se diría que igual. Nosotros como programadores observamos que las cosas son muy distintas. Disponemos del tipo arreglo, que nos permite definir variables de tipo arreglos de números enteros y dimensionarlos. Definimos la variable vector local a la función main() y si queremos utilizarla en otras funciones, al invocarlas la pasamos como parámetro. Además observamos que las variables i, tienen como ámbito solo las estructuras donde las usamos.

Estamos comenzando a tener buenos hábitos, no declarando variables globales, extendiendo ámbitos de variables locales a funciones pasándolas como parámetros y declarando variables solo en ámbitos donde son necesarias.

8- Ciclo de Seminarios y Talleres del Area de Programación.
Parámetros y Argumentos en el lenguaje de programación C++.

Conceptos de Parámetros y Argumentos

Son el medio a partir del cual podemos expandir el ámbito de variables locales de funciones, hacia otras funciones y además quienes nos permiten establecer comunicaciones entre funciones. Si nos vemos ante la necesidad de visualizar o modificar el valor de una variable local en otra función que llamaremos, debemos invocar a dicha función haciendo referencia de su nombre, seguido de los parámetros o nombres de variables para las cuales, en teoría ampliaríamos su ámbito.

```
void funcion_llamada(int x){  // función que recibe un argumento
 :
}

void una_funcion(void){
 int a,b;   // variables de ámbito local
 :
 :
 funcion_llamada(a); // llamada de la función con un parámetro
 :
}
```

La expresión funcion_llamada(a); se denomina llamado a la función funcion_llamada con un parámetro a de tipo entero. Y la expresión void funcion_llamada(int x) se denomina cabecera de la función funcion_llamada con un argumento, en este caso x, también de tipo entero.

Desde luego que, sobre la base de la comunicación entre funciones y la teoría del paradigma procedimental donde aplicamos la disgregación de procesos, nos podemos encontrar con las siguientes variantes:

1. Llamado de funciones sin pasar parámetros.
2. Llamado de funciones pasando parámetros.

esta claro que dichas funciones pueden o no devolvernos valores hacia el origen de su llamado.

Entonces, pasamos como parámetros nombres de variables al llamar una función, que deben corresponderse en su tipo al ser recibidos sobre los argumentos de dicha función. Esto básicamente es correcto, pero debemos completar la idea describiendo con claridad los tipos de parámetros que nos permite manejar el lenguaje C++.

Tipos de parámetros

Los tipos de parámetros a utilizar van a depender siempre de la necesidad que se le presente al programador. De todos modos hay dos necesidades básicas que generalmente están vigentes:

1. Poder obtener el contenido de una variable.
2. Poder modificar el contenido de una variable.

Parámetros por valor

Cuando surge la necesidad de obtener el valor o contenido de una variable original o local a una función, en otra función, se utiliza parámetros por valor. En este caso se produce una copia de la variable original hacia el argumento formal de la función receptora. Dicha variable tendrá como ámbito la función receptora y al culminar esta liberará el espacio de memoria que ocupa.

```
void imprime_cantidad(int can){
  printf(“%d\n”,can);
}

void alguna_funcion(void){
 int cantidad;
 :
 imprime_cantidad(cantidad);
 :
}
```

El valor de una variable original puede pasar de una función a otra y a su vez a otras funciones, sin inconvenientes. De pronto en

nuestro desarrollo, nos podemos encontrar con una hoja de ruta de valores de variables que pasan por funciones.

Si usamos parámetros por valor, nunca podremos modificar los valores de las variables originales ya que se producen copias de las mismas hacia las variables de la función llamada y su ámbito es local.

Al utilizar parámetros por valor, éstos pueden ser constantes como por ejemplo:

```
#include <stdio.h>

void imprime_datos(char nombre[], int edad){
  printf("Nombre :%s , edad: %d\n",nombre, edad);
}

void main(void){
  char alias[35] = {"Pepe"};
  imprime_datos(alias,23);
}
```

Parámetros por referencia

La referencia indica trabajar sobre la dirección de memoria que ocupa el parámetro o variable original.

```
void ingresar_cantidad(int &can){
  cin>>can;
}

void alguna_funcion(void){
 int cantidad;
 :
 ingresar_cantidad(cantidad);
 :
}
```

Como se observa en este ejemplo, el argumento de la función receptora, presenta el operador unario & precediendo al nombre de la variable. Esto indica que se tome la dirección del parámetro hacia el argumento, produciendo un espejo o alias de la variable original cantidad. Lo cual significará, que toda alteración producida sobre el argumento can, afectará a la variable original o parámetro cantidad.

Siempre que surja la necesidad de afectar a las variables originales debemos usar parámetros por referencia. Algunos casos serían: cargar un dato o recalcular un dato.

Ya que las funciones devuelven solo un valor, es habitual usar varios parámetros por referencia cuando necesitamos que una función provoque varios resultados y no solo uno.

Si al argumento por referencia lo precedemos de la palabra const, dicha variable no podrá ser alterada en el ámbito de la función receptora, por lo tanto nunca afectará a la variable original.

```
#include <iostream.h>

void incrementa_sin_exito(int b)
 { b++;}  // no afecta a la variable original a
void incrementa_con_exito(const int &b)
  { b++;} // Error de compilación

void main(void){
  int a = 4;
  cout<<a<<"\n";
  incrementa_sin_exito(a);
  cout<<a<<"\n";
  incrementa_con_exito(a);
  cout<<a<<"\n";
}
```

Si extraemos la palabra reservada const del argumento b en la función incrementa_con_exito(), no habría error de compilación y la variable original a se vería afectada al incrementarse en uno por el b++.

Parámetros por referencia puntero

Kernighan y Ritchie[3] el su capítulo apuntadores y arreglos nos hacen notar que la definición de los argumentos char s[]; y char *s; son completamente equivalentes cuando se pasa un nombre de un arreglo de caracteres a una función. Ya que al pasar un arreglo como parámetro, lo que en realidad se esta pasando es la dirección del comienzo del mismo.

Recordemos la función carga() de nuestro ejemplo de un arreglo de números enteros.

```
void carga(int  &b){ // sin & funciona igual por que?
  for (int i = 0;i < 5;i++)
   cin>>b[i];
}
```

Si escribimos

```
void carga(int * b){
  for (int i = 0;i < 5;i++)
   cin>>b[i];
}
```

o bien

```
void carga(int * b){
  for (int i = 0;i < 5;i++)
    cin>>*b++;
}
```

en definitiva y aceptando los conceptos de Brian y Dennis la variable original de tipo arreglo de enteros se verá modificada al producirse la carga de elementos, manipulando el arreglo a partir de subíndices o bien un puntero a enteros.

[3] Brian W. Kerninghan and Dennis M. Ritchie, El Lenguaje de Programación C

También podríamos haber escrito la función carga del siguiente modo:

```
void carga(int * b){
  for (int i = 0;*b == '\0';b++){
     i++;
     cin>>*b;
  }
}
```

siempre y cuando los elementos del arreglo estén inicializados en cero, ya que de lo contrario tendríamos algunos inconvenientes.

Parámetros de tipo puntero o apuntadores siempre tomarán la dirección de la variable original, por consecuencia la referencia a la misma.

Los parámetros por referencia puntero pueden tener el mismo cometido que los por referencia, solo en el concepto que al modificar un argumento, se afectará al parámetro. Pero no en su tratamiento, donde se trabaja siempre sobre la base de la aritmética de punteros.

Completemos la idea, con un ejemplo sencillo:

```
#include <iostream.h>

void incrementa(int *b) {
 (*b)++;
}
void main(void){
  int a = 4;
  incrementa(&a);
  cout<<a<<"\n"; // imprime 5
}
```

que sería totalmente equivalente a escribir

```
#include <iostream.h>
```

```
void incrementa(int *b) {
 (*b)++;
}
void main(void){
  int a = 4; int * ptero;
  ptero = &a;
  incrementa(ptero);
  cout<<a<<"\n"; // imprime 5
}
```

ya que el parámetro debe ser una variable de tipo puntero o bien la dirección de una variable.

Ejemplo para una matriz bidimensional:

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

typedef int matriz[2][2];

void carga(int *a){  // recibe la dirección de m[0][0]
for (int i = 0;i < 2;i++)
   for (int j = 0;j < 2;j++)
     cin>>*a++;  // incrementa direcciones de la matriz
}

void muestra(matriz b){
for (int i = 0;i < 2;i++)
   for (int j = 0;j < 2;j++)
      printf("%d,",b[i][j]);
}

int suma(matriz c){
int valor=0;
for (int i = 0;i < 2;i++)
    for (int j = 0;j < 2;j++)
       valor += c[i][j];
return(valor);
```

```
}

void main(){
  matriz m = {0,0,0,0};
  cargapuntero(&m[0][0]);
  muestra(m);
  printf("\nLa suma de los elementos es = %d\n",suma(m));
}
```

Al trabajar con una variable subindizada y bidimensional cuando invocamos a la función carga(), debemos pasarle como parámetro la dirección de la variable local entera m (&m[0][0]), para tratar la matriz con punteros. El argumento de la función carga(), recibe dicha dirección sobre una variable de tipo puntero a un entero a, y manipula las direcciones de memoria de la matriz m con la aritmética de punteros aplicada a dicha variable.

Combinando tipos de parámetros

Al invocar funciones, podemos hacerlo pasando parámetros de distintos tipos e inclusive con tipos de variables diferentes.

Prototipos de funciones como ejemplos:

```
double mayor( double a, double b);
```

Para averiguar cual contenido de variable es mayor, el tipo de parámetro solo debe ser por valor.

```
double suma_uno_al_mayor(double &, double &);
```

Si deseamos incrementar al contenido de la variable mayor en uno, debemos tener en cuenta que se podrá modificar una u otra variable. De modo que el tipo de parámetro debe ser por referencia.

```
void carga_datos( cha * nombre, int & edad, int tipo);
```

Como tipo indicará alguna alternativa de ingreso de datos, el parámetro es por valor. Y ya que el interés es afectar variables originales que responden a un nombre y la edad, dichos parámetros son por referencia.

int funcion(int, int &, char *, int *, float);

Otras variantes de combinaciones de parámetros.

La función main() y sus argumentos.

Los parámetros pasados en la línea de órdenes del DOS o líneas de comandos del Windows a un programa, son recibidos por la función main() sobre sus argumentos. Existen dos variables predefinidas dentro del lenguaje que reciben los argumentos que se pasan al ejecutar un programa.

	Tipo	Finalidad
argc	entero	contiene el número de argumentos que se han introducido mas uno.
argv	array	array de punteros a caracteres.

El argv contiene los parámetros que se han pasado desde el sistema operativo al invocar el programa. La declaración de los argumentos es:

La variable argc como mínimo valdrá 1, ya que el nombre del programa se toma como primer parámetro, almacenado con argv[0], que es el primer elemento de la matriz. Cada elemento del array apunta a un parámetro de la línea de órdenes. Todos los parámetros de la línea de órdenes son cadenas.

```
#include <stdio.h>

void main(int argc, char *argv[]) {
  if(argc != 2){
```

```
   printf("Ha olvidado su nombre.\n");
   return;}
  printf("Hola %s", argv[1]);
  printf("Parámetros del usuario %d", argc);
}
```

Este programa imprime Hola y su nombre en la pantalla si se escribe directamente tras el nombre del programa. Supongamos que hemos llamado al programa saludo.cpp, entonces, al compilarlo se creará un ejecutable saludo.exe, al llamar al programa saludo con un nombre, por ejemplo:Juan, para ejecutar el programa tendrá que introducir

c:\tcwin\bin\saludo Juan

Y la salida del programa sería

Hola Juan
Parámetros del usuario 2.

Observaremos para un nuevo ejemplo, que a pesar de no ejecutarse inicialmente la función main(), ya que se ejecuta la función inicio (especificado por #pragma startup), el comportamiento de los parámetros es igual al ejemplo anterior.

```
#include <stdio.h>
void inicio(void){
  printf("Comenzo la ejecución\n");
}
void fin(void){
  printf("\nTermino la ejecución\n");
}

#pragma startup inicio
#pragma exit fin

void main(int argc, char *argv[]) {
  if(argc < 2){
     printf("Ha olvidado su nombre.\n");
```

```
    return; }
  else
    printf("Hola %s\n", argv[1]);
  printf("Parametros %d", argc);
}
```

En la mayoría de los entornos, cada parámetro de la línea de órdenes debe estar separado por un espacio o una tabulación. Las comas, puntos y comas, y similares no se consideran separadores. argv[0] contiene el path (camino) y nombre del programa que se esta ejecutando.

Main() puede no recibir parámetros si fue definida como:

```
 <tipo_de_dato> main(void) {
  :
  :
  return(tipo_de_dato);
}
```

Variedad de ejemplos usando parámetros

Vamos a observar una serie de ejemplos que utilizan los tres tipos de parámetros sobre argumentos de tipos distintos.

Ejemplo para una variable nativa simple, entera:

```
#include <iostream.h>

void incrementa_valor(int b) // por valor no afecta a la variable original a
  { b++;}
void incrementa_referencia(int &b) // afecta a la variable original a
  { b++;}
void incrementa_ref_puntero(int *b) // afecta a la variable original a
 { (*b)++;}  // la aritmética de punteros nos exige colocar los paréntesis.

void main(void){
   int a = 4;   // variable local de la función main
   cout<<a<<"\n"; // imprime 4
```

```
  incrementa_valor(a);
  cout<<a<<"\n"; // imprime 4
  incrementa_referencia(a);
  cout<<a<<"\n"; // imprime 5
  incrementa_ref_puntero(&a);
  cout<<a<<"\n"; // imprime 6
 }
```

Ejemplo para una variable de tipo Estructura:

```
#include <iostream.h>
#include <string.h>

typedef struct cliente {char nombre[35]; int codigo;} reg_cli;

void suma_par_valor(reg_cli a)
{ a.codigo++;} // no afecta a las variables originales.

void suma_referencia(reg_cli &a)
{ strcpy(a.nombre,"pepe");
 a.codigo++;} // afecta a las variables originales.

void suma_ref_puntero(reg_cli *a)
{  (*a).codigo++; // afecta a las variables originales.
  strcpy((*a).nombre,"juan");}

void main(void){                                {
 reg_cli b;
 b.codigo = 3;
 strcpy(b.nombre,"jose");
 cout<<b.codigo<<"\n";          // imprime 3
 cout<<b.nombre<<"\n";         // imprime jose
 suma_par_valor(b);
 cout<<b.codigo<<"\n";         // imprime 3
 cout<<b.nombre<<"\n";         // imprime jose
 suma_referencia(b);
 cout<<b.codigo<<"\n";        // imprime 4
 cout<<b.nombre<<"\n";       // imprime pepe
```

```
 suma_ref_puntero(&b);
 cout<<b.codigo<<"\n";       // imprime 5
 cout<<b.nombre<<"\n";     // imprime juan
}
```

Ejemplo para un tipo de estructura autoreferenciada con punteros.

Este tipo nos permite generar estructuras dinámicas de tipo lista, pila y cola, entre otras.

```
#include <iostream.h>
#include <string.h>
#include <alloc.h>
#include <conio.h>
typedef struct cliente {char nombre[35]; cliente *siguiente;} reg_cli;

void suma_par_valor(reg_cli a)
  { strcpy(a.nombre,"pepe");
    a.siguiente = NULL;
 }
void suma_referencia(reg_cli &a)
{ strcpy(a.nombre,"pepe");
   a.siguiente = NULL;
}
void suma_ref_puntero(reg_cli *a)
{ (*a).siguiente = NULL;
   strcpy((*a).nombre,"juan");
}
void main(void){
  clrscr();
  reg_cli b;
  b.siguiente = NULL;
  strcpy(b.nombre,"jose");
  cout<<b.nombre<<"\n";
  suma_par_valor(b);
  cout<<b.nombre<<"\n";
  suma_referencia(b);
  cout<<b.nombre<<"\n";
```

```
  suma_ref_puntero(&b);
  cout<<b.nombre<<"\n";
}
```

Se debe tener, como se observa, especial cuidado en la aritmética de punteros que se utiliza, en particular en las funciones que manipulan variables de tipo puntero. Siempre haciendo uso de ( ) para referenciar la variable de tipo puntero y luego, el miembro al que nos referimos en particular.

```
#include <iostream.h>
#include <string.h>
#include <alloc.h>

typedef struct cliente {char *nombre; int codigo;} reg_cli;

void suma_par_valor(reg_cli a){
   a.codigo++;} // no afecta a las variables originales.

void suma_referencia(reg_cli &a){
  strcpy(a.nombre,"pepe");
  a.codigo++;
} // afecta a las variables originales.

void suma_ref_puntero(reg_cli *a){
   (*a).codigo++; // afecta a las variables originales.
   strcpy((*a).nombre,"juan");
}

void main(){
  reg_cli b;
  b.codigo = 3;
  delete(b.nombre);
  b.nombre = (char *) malloc(sizeof("Jose"));
  strcpy(b.nombre,"jose");
  cout<<b.codigo<<"\n";          // imprime 3
  cout<<b.nombre<<"\n";        // imprime jose
  suma_par_valor(b);
```

```
  cout<<b.codigo<<"\n";         // imprime 3
  cout<<b.nombre<<"\n";         // imprime jose
  suma_referencia(b);
  cout<<b.codigo<<"\n";         // imprime 4
  cout<<b.nombre<<"\n";         // imprime pepe
  suma_ref_puntero(&b);
  cout<<b.codigo<<"\n";         // imprime 5
  cout<<b.nombre<<"\n";         // imprime juan
}
```

Observe este ejemplo para una variable de tipo puntero:

```
#include <iostream.h>

void incrementa_valor(int b) // argumento por valor
  { b++;}  // no afecta la variable original
void incrementa_entero(int *b) // por referencia
  { (*b)++;} //  afecta la variable original
void incrementa_ref_puntero_numero(int **b) // por ref. puntero
 { (**b)++;}  // afecta la variable original

void main(void){
   int *a;   // variable local de tipo puntero a un entero
   *a = 4;  // la dirección donde se almacena un entero, se inicializa
   cout<<*a<<"\n"; // imprime 4
   incrementa_valor(*a);   // parámetro de tipo entero
   cout<<*a<<"\n"; // imprime 4
   incrementa_entero(a);
   cout<<*a<<"\n"; // imprime 5
   incrementa_ref_puntero_numero(&a);
   cout<<*a<<"\n"; // imprime 6
 }
```

La variable original es de tipo puntero a un entero, con ámbito en la función main(). Al llamar la función incrementa_entero() pasando como parámetro, la variable original a , hacia un argumento del mismo tipo, se referencia la variable original. Dicha variable al sufrir modificaciones, afectará a la original a.

Ahora bien, al pasar como parámetro la dirección de la variable a, debe ser recibida por un argumento de tipo puntero a puntero de un entero. En este caso toda modificación que se produzca sobre el contenido del argumento afectará a la variable original.

Para no afectar a la variable original, se debe pasar como parámetro el contenido de la misma incrementa_valor(*a); que será recibido en un argumento de tipo entero.

Variantes de parámetros y argumentos

Ya conocidos los tipos de parámetros básicos, podemos ver algunas variantes que C++ nos brinda como alternativas validas en la especificación de argumentos de funciones.

Argumentos por defecto (omisión)

Una mejora de las funciones en C++ es que se pueden especificar los valores por defecto para los argumentos cuando se proporciona un prototipo de una función. Cuando se llama a una función se pueden omitir argumentos e inicializarlos a un valor por defecto.

Un argumento por defecto u omisión es un parámetro que el llamado a una función no va a proporcionar. Los argumentos por defecto también pueden ser parámetros opcionales. Si se pasa un valor a uno de ellos, se utiliza ese valor. Si no se pasa un valor a un parámetro opcional, se utiliza un valor por defecto como argumento.

Ejemplo:

```
#include <iostream.h>

void f(int ,int = 2); // prototipo de función

void main(){
  f(4,5);
```

```
  f(6);
}

void f(int i, int j){
  cout<<i<<","<<j<<endl;
}
```

Al ejecutar el programa, se visualizará: **4,5 6,2**

Los parámetros por defecto se pasan por valor.

Todos los argumentos por defecto deben estar situados al final del prototipo o cabecera de la función. Después del primer argumento por defecto, todos los argumentos posteriores deben incluir también valores por defecto.

Argumentos punteros a void

Un uso importante de punteros void en C++, es pasar la dirección de tipos de datos diferentes en una llamada a función, cuando no se conoce por anticipado que tipo de dato que se pasa como parámetro.

Ejemplo:

```
#include <iostream.h>

enum dato{caracter,real,entero,cadena};

void ver(void *,dato);

void main(){
  char a = 'b';
  int x = 3;
  double y = 4.5;
  char *cad = "hola";
  ver(&a,caracter);
  ver(&x,entero);
  ver(&y,real);
```

```
  ver(cad,cadena);
}

void ver( void *p, dato d){
  switch(d){
    case caracter: printf("%c\n",*(char *)p);
      break;
    case entero: printf("%d\n",*(int *)p);
      break;
    case real: printf("%ld\n",*(double *)p);
      break;
    case cadena: printf("%s\n",(char *)p);
  }
}
```

El argumento void *p recibe el parámetro enviado desconociendo el tipo, lo cual es indispensable de especificar para direccionar, como en el printf() de la función ver() *(char *)p, en el caso de ser un caracter. En este caso nos apoyamos sobre el enumerado dato, para identificar el tipo al que apunta *p.

Argumentos no especificados o variables

C++ permite declarar una función cuyo número y tipo de argumentos no son conocidos en el momento de la compilación. Esta característica se indica con la ayuda del operador especial puntos suspensivos (...) en la declaración de la función. void f1(...);

Esta declaración indica que f1 es una función que no devuelve ningún valor y que tomará un número variable de argumentos en cada llamada. Si algunos argumentos se conocen, deben situarse al principio de la lista:

```
void f2(int a,float b,...);
```

Los puntos suspensivos indican que la función se puede llamar con diferentes conjuntos de argumentos en distintas ocasiones. Este

formato de prototipo reduce la cantidad de verificación que el compilador realiza.

Así, la función predefinida **printf()**, que tiene el prototipo:

int printf(char *formato,...);

Esto significa que la función devuelve un entero y acepta un puntero a un parámetro fijo de caracteres y cualquier número de parámetros adicionales de tipo desconocido. Con este prototipo, el compilador verifica en tiempo de compilación los parámetros fijos, y los parámetros variables se pasan sin verificación de tipos.

Para entender cómo es posible crear este tipo de funciones con parámetros variables, es interesante comprender aunque sólo sea de manera superficial, cómo funciona un compilador. En especial el mecanismo de llamadas a funciones.

Cuando el compilador traduce el código fuente a código máquina utiliza una pila para realizar ciertas tareas. La pila es, básicamente, una zona de almacenamiento temporal en memoria a la cual se hace referencia con un puntero. A través de este puntero se puede conocer sus contenidos. Cuando se llama a una función, se crea una nueva zona en ella (un *frame*) donde se almacena cierta información, como por ejemplo, la dirección de retorno, las *variables automáticas* de la función, etc. A nosotros, lo que nos interesa es que en esa zona, también se copian los argumentos de la función.

En C los argumentos se pasan *por valor*, es decir, se hacen copias de los mismos, y por eso podemos usarlos como variables dentro del cuerpo de la función, sin miedo a que varíen fuera, y debemos trabajar con punteros cuando queremos el comportamiento contrario, es decir el paso de argumentos *por referencia*.

Veamos un caso sencillo. Tenemos la siguiente función y llamada:

```
void funcion(int a, int b);
funcion(10, 20);
```

En el *frame* de la pila tendremos los valores de dichas variables. Por lo tanto, si la dirección de memoria del primer parámetro es X, el contenido [X] es 10 –en adelante, usaré la convención de los corchetes para indicar el contenido de una posición de memoria-. Además, el segundo parámetro estará en XXXX+sizeof(int), por tanto [X+sizeof(int)] es 20.

Parece claro que las variables están contiguas en memoria. Si tenemos ahora una función y la llamada:

```
void funcion(char *s , ... );
funcion("Estos son dos
números: %d y %f", 10, 0.5);
```

Entonces, como vimos antes, sabemos que [X], por ser un puntero contiene la dirección de memoria donde se almacena la letra 'E'. Pero además, sabemos que de existir otro parámetro este estaría en la posición X+sizeof(char *), y que el siguiente estaría enX+sizeof(char *)+sizeof(*tipo*) -el tipo es desconocido a priori-. Es aritmética de punteros pura. De este modo, es sencillo obtener todos los argumentos que se pasen a la función, sin importar cuantos sean ni de que tipo.

En realidad, esto no tiene porque suceder de este modo tan ideal, dependerá de la arquitectura, el sistema operativo, el compilador, etc. Sin embargo, este modelo de posiciones contiguas para los argumentos es la base del sistema que usa C, y ayudará al lector a entender el mecanismo que se nos proporciona.

El archivo **stdarg.h** contiene macros que se pueden utilizar en funciones definidas por el usuario con número variable de parámetros.

En **C++** hay otro método para proporcionar argumentos a funciones que tienen listas de argumentos en número variable. Este método facilita al compilador toda la información necesaria en una lista con un número variable de argumentos, proporcionándole un conjunto de variables preescritas que le indican cuántos argumentos contiene la lista y qué tipo de datos son.

Las macros que se pueden utilizar para este propósito están definidas en un archivo de cabecera denominado **stdarg.h** y son **va_start()**, **va_arg()** y **va_end()**.

Las macros del archivo **stdarg.h** tienen dos características útiles:

- Han sido escritas para uso del programador y son bastantes versátiles, de modo que se pueden utilizar de diversas formas.

- Las macros proporcionan un medio transportables para pasar argumentos variables. Esto se debe a que están disponibles dos versiones de macros: las definidas en **stdarg.h**, que siguen el **standard** ANSI C, y las definidas en **varargs.h**, que son compatibles con UNIX**System V**.

La macro **va_list** es un tipo de dato que es equivalente a una lista de variables. Una vez que se define una variable **va_list**, se puede utilizar como un parámetro de las macros **va_start()**y **va_end()**.

La sintaxis de la macro **va_start()** es: **void va_start(va_list arg_ptr, prev_param); arg_ptr** apunta al primer argumento opcional en una lista variable de argumentos pasados a una función. Si la lista de argumentos de una función contiene parámetros que se han especificado en la declaración de la función, el argumento **prev_param** de **va_start()** proporciona el nombre del argumento especificado de la función que precede inmediatamente al primer argumento opcional de la lista de argumentos.

Cuando la macro **va_start()** se ejecuta, hace que el parámetro **arg_ptr** apunte al argumento especificado por **prev_param**.

La sintaxis de la macro **va_arg()** es: **void va_arg(va_list arg_ptr,tipo);**

La macro **va_arg()** tiene un propósito doble:

- Primero, **va_arg()** devuelve el valor del objeto apuntado por el argumento **arg_ptr**.

- Segundo, **va_arg()** incrementa **arg_ptr** para apuntar al siguiente elemento de la lista variable de argumentos de la función que se está llamando, utilizando el tamaño tipo para determinar dónde comienza el siguiente argumento.

La sintaxis de la macro **va_end()** es: **void va_end(va_list arg_ptr);**

La macro **va_end()** realiza las tareas auxiliares que son necesarias para que la función llamada retorne correctamente. Cuando todos los argumentos se leen, **va_end()** reinicializa **arg_ptr** a **NULL**.

El pseudocódigo del proceso para manejar la lista variable de argumentos sería:

```
tipo funcionVariable(last , ...) {
    va_list pa;
    tipo_X argumento_de_tipo_X;
    va_start(pa,last);
    while (quedanArgumentos)
      argumento_de_tipo_X = va_arg(pa, tipo_X);
    va_end(pa);
 }
```

El orden es innegociable. Debe definirse el puntero de tipo va_list que se debe inicializar con va_start para poder comenzar a trabajar.

Mientras haya argumentos, hay que obtenerlos con va_arg y para terminar, hay que llamar siempre a va_end.

Hay que prestar atención a unos detalles cuando usemos este tipo de construcciones.

1. La lista variable siempre va al final de los argumentos.
2. Tiene que haber al menos un argumento definido (last), para poder tener una referencia y acceder al resto de argumentos.
3. Dado que la dirección de last es utilizada por va_start no debe ser ni una variable register, ni una función, ni un array.
4. Si no hay próximo argumento, o si tipo no es compatible con el tipo del próximo argumento, se producirán errores impredecibles al llamar a va_arg.
5. Cuidado con los tipos. Tengan en cuenta que por defecto C hace promoción de los parámetros pasados, así, si pasan un float, este será promovido a double y la sentencia va_arg(pa,float) será incorrecta. Lo mismo con shorts, etc. Normalmente el compilador nos avisará de esto.
6. Las macros nos ayudan a trabajar de modo portable. No importa, por ejemplo, que un int tenga 32 o 16 bits. Las macros y el compilador se encargan de resolver la diferencia. Nosotros sólo vemos una lista de variables de distintos tipos.

Como controlar el número de argumentos

Controlar el número de argumentos es importante para evitar errores. Pongamos como ejemplo algo muy habitual cuando se nos escapa un detalle: equivocarnos en un printf. Si escribimos lo siguiente:

```
printf("%d %d %d\n", 3, 4, 5, 6);
printf("%d %d %d\n", 3, 4);   // error 3 imágenes para 2 datos
```

obtendremos la salida, donde el último número puede variar de una sistema operativo a otro.

```
3 4 5
3 4 134513729
```

Pasarse en el número de argumentos no parece muy problemático, sin embargo, quedarse corto provoca resultados extraños. El compilador está buscando un entero pero encuentra memoria sin inicializar (basura) que en este caso particular equivale a un 134513729, si se interpreta como un entero.

Hay que tener mucho cuidado. En este sencillo ejemplo, tenemos un problema visual, que no parece muy grave. ¿Y si el argumento fuese un puntero? los efectos colaterales podrían ser terribles, podríamos tener un error de segmento o uno de esos bugs casi imposibles de depurar.

De todos modos, parece que stdarg.h nos ofrece un mecanismo un poco escaso. Sería mejor disponer de algo similar a los argumentos de línea de comandos en main(int argc y char* argv[]), o al menos tener una macro va_count, o similar, que nos exprese cuántos argumentos tenemos. Por desgracia no es así y tenemos que usar otra aproximación donde, como programadores, tenemos que hacer explícito el número de los mismos.

En general, hay dos modos de hacerlo:

1. Con un **contador**... uno de los argumentos fijos es un entero que indica el número de argumentos variables que debemos esperar. No permite indicar el tipo de los argumentos a menos que usemos un artificio adicional (como indicarlos con constantes simbólicas). Suele usarse con funciones que reciben argumentos de un tipo concreto.
2. Con una **cadena de formateo**... el estilo que usa la familia de funciones printf y scanf. La función recibe un argumento que indica de alguna manera la cantidad de argumentos variables esperados y el tipo de los mismos.

Ejemplo Nº 1:

```
#include <iostream.h>
#include <stdarg.h>

int suma_enteros(int primero,...);
```

```
void main(){
  cout<<suma_enteros(2,15,-1)<<endl;
  cout<<suma_enteros(6,6,6,-1)<<endl;
  cout<<suma_enteros(8,10,1946,47,-1)<<endl;
}

int suma_enteros(int primero,...){
  int cuenta = 0,suma = 0,i = primero;
  va_list marcador;
  va_start(marcador primero);
  while (i != -1){ suma += i;
     cuenta++;
     i = va_arg(marcador,int);
   }
  va_end(marcador);
  return suma;
}
```

La función suma_enteros() recibe parámetros variables, 1 o n, sobre sus argumentos variables y enteros. El argumento primero, será la base inicial de encontrar los demás argumentos, hasta encontrar un argumento con el valor -1, punto en el cual se determina que no hay mas argumentos en la pila.

Ejemplo Nº 2:

```
#include <iostream.h>
#include <stdarg.h>

double suma_reales(int operandos, ...) {
    double resultado = 0.0;
    va_list pa;
    va_start(pa, operandos);
    while (operandos--) {
       resultado += va_arg(pa, double);
    }
    va_end(pa);
```

```
    return resultado;
 }

void pprintf(char *formato, ...) {
    char *p;
    va_list pa;
    va_start(pa, formato);
    for (p = formato; *p; p++) {
       if (*p !=  '?') {
          putchar (*p);
          continue;
       }
       switch (*++p) {
          case 'd':
             printf("{%d}", va_arg(pa, int));
             break;
          case 's':
             printf("{%s}", va_arg(pa, char *));
             break;
          default:
             putchar(*p);
             break;
          }
       }
       va_end(pa);
 }

int main (void) {
   printf("Resultado es %g\n", suma_reales(3, 0.3,0.1, 0.2));
   pprintf("Probemos a poner llaves a una ?s y a un ?d \n", "cadena", 25);
   return 0;
 }
```

La función suma_reales() muy similar a la función suma_enteros() del ejemplo anterior, recibe como parámetros un entero que indica la cantidad de argumentos variables a buscar en la pila y una lista de argumentos variables de tipos double.

La función pprintf() que recibe un puntero a una cadena como parámetro (formato), donde pa de tipo va_list apuntará al primer carácter de la misma (“P”) y una lista variable de argumentos a encontrar con va_arg(pa, tipo).

Se produce la salida de caracteres hasta encontrar el carácter ?, el cual seguido de una d establece que se busque el siguiente argumento de tipo entero y seguido de una s un argumento de tipo cadena, que nos devuelve va_arg() y se imprimen.

Conclusiones:

Los parámetros juegan un papel demasiado importante en los desarrollos, donde el programador haciendo el uso adecuado de los mismos, alimentará el buen hábito de controlar exhaustivamente la persistencia y manipulación de sus variables u objetos y como consecuencia mantendrá distancia de los efectos colaterales.

En la actualidad, la programación con lenguajes como Java, J#, .Net, C# y otros, continua sobre la base de los conceptos expuestos. De modo tal que recomendamos este artículo, especialmente a principiantes que deseen manejar cuidadosamente los parámetros y argumentos.

Bibliografía principal:

- El Lenguaje de Programación C, Brian W. Kernighan y Dennis M. Ritchie.
- El C++ Lenguaje de Programación, Bjarne Stroustrup.
- Programación y resolución de problemas en C++, Nell Dale y Chip Weems