¿QUÉ ES UN PUNTERO?

De manera simple, **un puntero es una dirección**. Al contrario que una variable normal, un puntero es una variable almacenada en alguna parte del espacio del programa. Siempre es mejor usar un ejemplo, por tanto, veamos un buen ejemplo de programa con punteros:

POINTER.C

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | main()                      /\* ejemplo de uso de un puntero \*/    {    int index,\*pt1,\*pt2;    index = 39;                      /\* cualquier valor numérico \*/    pt1 = &index;                   /\* la dirección de index \*/    pt2 = pt1;    printf("El valor es  %d %d %d\n",index,\*pt1,\*pt2);    \*pt1 = 13;           /\* esto cambia el valor de index \*/    printf("El valor es %d %d %d\n",index,\*pt1,\*pt2);    } |

Por el momento, ignoremos los mandatos donde definimos “index” y otros dos campos, precedidos por un asterisco,  Comúnmente se llama asterisco a este signo “\*”, pero por razones que veremos posteriormente, preferimos llamarle “estrella”. Observando el primer mandato, vemos claramente que asignamos 39 al valor de la variable “index”. Esto no es una novedad, lo hemos hecho muchas veces hasta ahora. El siguiente mandato asigna a “pt1″ un extraño valor, la variable “index” con un signo de amperson delante. En el ejemplo, pt1 y pt2 son punteros, y la variable “index” es una variable simple.

DOS IMPORTANTES REGLAS

Las siguientes reglas son muy importantes a la hora de usar punteros, y deben ser perfectamente aprendidas.

**1.** Un nombre de variable, precedido  por un signo de amperson define la **dirección de la variable**, y ocasionalmente “la apunta”. Puede leerse la línea 6 del programa como “pt1 es asignado al valor de la dirección de index”.

**2.** Un puntero con una estrella precediéndole se refiere al **valor de la variable** apuntada por el puntero. La línea 9 del programa puede leerse como: “al puntero pt1 le es asignado el valor 13″.

AYUDA PARA MEMORIZAR

**1.** Pensar en**&** como una **dirección**.

**2.** Pensar en **\*** como una estrella que señala un **almacenamiento**.

Asumimos de momento, que “pt1″ y “pt2″ son punteros (en breve trataremos la manera para definirlos). Como punteros, no contienen el valor de una variable, pero si contienen su dirección y pueden ser usados para apuntar a una variable.

La línea 6 del programa ejemplo destina el puntero “pt1″ a apuntar a la variable que hemos definido como “index”, porque hemos asignado la dirección de “index” a “pt1″. Dado que  tenemos un puntero para “index”, podemos manipular su valor llamando a la variable por su nombre o también al puntero.

En la línea 9 modificamos el valor usando el puntero. Como que “pt1″ apunta a la variable “index”, colocando una estrella en el puntero, nos referimos a la ubicación en memoria a la que apunta.

La línea 9 también asigna 13 a “index”. En cualquier lugar del programa donde sea permitido usar la variable “index” lo será también usar “\*pt1″, dado que su sentido es idéntico mientras el puntero no sea asignado a otra variable.

OTRO PUNTERO

Sólo para complicar un poco más las cosas, tenemos otro puntero en el programa, “pt2″. Dado que no ha sido asignado a ninguna variable tras su definición, no apunta a nada, sólo contiene información “de relleno”. Esto será así mientras no se le asigne una variable. Un comando asigna a “pt2″ el mismo valor que a “pt1″, por tanto “pt2″ apunta también a “index”. Continuando con la definición del párrafo anterior, en cualquier parte del programa donde sea permitido poner “index”, también lo será “\*pt2″, ya que significan lo mismo. Este hecho viene ilustrado en el printf, ya que usa las3 formas de identificar la misma variable, para imprimirla 3 veces.

HAY UNA SOLA VARIABLE

Parece que hay 3 variables, pero en realidad sólo hay una. Los dos punteros señalan a la variable simple. Esto se ilustra en el siguiente mandato, el cual asigna el valor 13 a “index”, porque ahí es donde apunta el puntero. El siguiente “printf” provoca que el nuevo valor 13 aparezca 3 veces por pantalla. En todo momento hablamos de una sola variable.

¿DÓNDE DECLARO UN PUNTERO?

Buena pregunta. Situémonos en la tercera línea del programa. Vemos una forma muy conocida para declarar variables, en concreto “index”, seguida por dos definiciones más. La segunda definición puede ser interpretada como “la dirección de almacenamiento a la cual apunta pt1 contendrá un valor entero”. Por tanto, “pt1″ es el puntero de un tipo “int”. De la misma manera, “pt2″ es otro puntero de una variable entera.

Un puntero debe definirse para apuntar algún tipo de variable. Además, no puede apuntar a un tipo de variable distinto al que se le dio en su definición, ya que cometeríamos un error de incompatibilidad de tipos. Por ejemplo, un puntero de tipo “int” no puede apuntar a una variable “float”.

EL SEGUNDO PROGRAMA CON PUNTEROS

En lo que llevamos viendo sobre punteros, hemos cubierto mucho territorio, importante territorio, pero aún nos queda mucho, veamos el siguiente ejemplo:

***POINTER2.C***

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | #include <string.h>    #include <stdio.h>    main()    {    char strg[40],\*there,one,two;    int \*pt,list[100],index;    strcpy(strg,"Esta es una cadena de caracteres.");    one = strg[0];                 /\* one y two son idénticos \*/    two = \*strg;    printf("El primer resultado es %c %c\n",one,two);    one = strg[8];                /\* one y two son idénticos \*/    two = \*(strg+8);    printf("El segundo resultado es %c %c\n",one,two);    there = strg+10;        /\* strg+10 es idéntico a strg[10] \*/    printf("El tercer resultado es %c\n",strg[10]);    printf("El cuarto resultado es %c\n",\*there);    for (index = 0;index < 100;index++)    list[index] = index + 100;    pt = list + 27;    printf("El quinto resultado es %d\n",list[27]);    printf("El sexto resultado es %d\n",\*pt);    } |

En este programa hemos definido varias variables, y 2 punteros. El primer puntero, “there” es un puntero tipo “char”, ya que apunta a una variable de tipo “char” y, el segundo, “pt” apunta a una variable “int”. Hemos definido también dos arrays, “strg” y “list”. Los usaremos para mostrar la relación entre punteros y nombres del array.

UNA VARIABLE DE CADENA ES REALMENTE UN PUNTERO

En la programación en C, una variable de cadena es definida como un puntero al empezar el programa. Esto será explicado más adelante. En el programa ejemplo, primero asignamos una constante de cadena a la variable “strg”, con lo cual ya tenemos algo para trabajar. Después, asignamos el valor del primer elemento a la variable “one”, una variable de tipo “char”. Tras esto, dado que el nombre de cadena es(según la definición de C), un puntero, podemos asignarle el mismo valor que “two” usando la estrella y el nombre de la variable,  El resultado de las dos asignaciones es que ahora “one” tiene el mismo valor que “two”, y ambos contienen el caracter “E”, el primer caracter de la cadena. Nótese que sería incorrecto escribir la línea nueve de esta forma: “two = \*strg[0];”, porque la estrella toma el lugar de los corchetes.

Para cualquier uso, “strg” es un puntero. Tiene restricciones que un auténtico puntero no tiene. No puede cambiarse, como una variable, pero debe contener siempre el valor inicial, y apuntar a una variable. Puede ser vista como un puntero constante, y en algunas aplicaciones podríamos necesitar una constante que funcionase como un puntero, y que no pudiera ser accidentalmente modificada.

Veámosla siempre como una variable que no puede ser modificada, y que se usa para referirse a otras variables, a las cuales se ha definido para apuntar.

Vayamos ahora a la línea 12, en la cual a “one” se le asigna el valor de la novena variable(ya que el índice empieza en 0), y a “two” se le asigna el mismo valor, dado que se nos permite indexar un puntero para recoger valores más lejos en la cadena. Ambas variables contienen ahora el caracter “u”.

C toma cuidado en indexar automáticamente y, ajustar el indexado al tipo de variable a la cual apunta el puntero. En el ejemplo el índice de \* es 8 transferido al puntero antes, obteniendo el resultado que esperábamos, porque el tipo char tiene un byte de longitud. Si estábamos usando un puntero de tipo “int” con una variable “int”, el índice podría doblarse y añadirse al puntero, dado que un valor “int” usa dos bytes, para almacenar el valor. Cuando lleguemos al tema estructuras veremos que una variable puede tener cientos o miles de caracteres, pero que el índice lo obtendrá automáticamente el sistema.

Ya que “there” es un puntero, puede serle asignado el valor del undécimo elemento de “strg” por el mandato de la línea 16 del programa. Recordemos que a este puntero se le puede asignar un valor a lo largo de la variable tipo “char”. Debe quedar claro que los punteros deben ser especificados correctamente, para permitir el puntero aritmético, objeto de la siguiente sección.

PUNTERO ARITMÉTICO

No todas las formas aritméticas están permitidas en un puntero. Sólo esas que tienen sentido, considerando que un puntero es una dirección de memoria. Tendría sentido añadir una constante a una dirección, moviéndola un número x de posiciones. De forma similar, se podría disminuir el valor de un puntero, atrasándolo unas posiciones. Añadir un puntero a otro no tendría sentido, ya que las direcciones absolutas de memoria no pueden sumarse. Tampoco nos es permitido multiplicar punteros, ya que nos daría un número imposible de manejar por un puntero.

UN PUNTERO “INT”

Al array “list” le son asignadas una lista de valores entre 100 y 199, para tener algunos datos con los que trabajar. Después, asignamos el valor 27 al puntero “pt”, el valor del vigésimo octavo elemento de la lista, e imprimimos el mismo valor de dos formas distintas, ilustrando que el sistema verdaderamente ajusta el índice a la variable “int”, donde quiera que esté.

Hay dos formas de tomar datos de una función, una forma es a través de un array, y la otra a través de los punteros.

Veamos el programa ***TWOWAY.C*** (Dos caminos)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | main()    {    int pecans,apples;    pecans = 100;    apples = 101;    printf("Los valores iniciales son pecans=%d apples=%d\n",pecans,apples);    /\* cuando llamamos a "fixup"     \*/    fixup(pecans,&apples);  /\* damos el valor de pecans      \*/    /\* y la dirección de apples      \*/    printf("Los valores finales son pecans=%d apples=%d\n",pecans,apples);    }    fixup(nuts,fruit)             /\* nuts es un valor entero    \*/    int nuts,\*fruit;              /\* fruit apunta a un entero   \*/    {    printf("Los valores son nuts=%d \*fruit=%d\n",nuts,\*fruit);    nuts = 135;    \*fruit = 172;    printf("Los valores son nuts=%d \*fruit=%d\n",nuts,\*fruit);    } |

RETORNO DE DATOS DE UNA FUNCIÓN MEDIANTE PUNTEROS

En ***TWOWAY.C*** hay dos variables definidas en el programa principal, “pecans” y “apples”. Nótese que ninguna de ellas es definida como un puntero. Les asignamos valores y los mostramos por pantalla. Entonces llamamos a la función “fixup” dándole ambas variables. “Pecans” es simplemente devuelta por la función, pero de “apples” sólo nos devuelve la dirección. Tenemos un problema. Los dos argumentos no son lo mismo, ya que el segundo es un puntero. Debemos avisar a la función de que es posible que reciba una variable y un puntero. Es muy sencillo de hacer: en el parámetro de declaraciones se define “nuts” como un entero, y “fruit” como un puntero tipo “int”. La llamada está ahora acorde con la cabecera de la función, y el interfaz del programa funcionará perfectamente.

En el cuerpo de la función imprimimos los dos valores dados a la función, los modificamos y los imprimimos otra vez. No tiene complicación. La sorpresa viene cuando volvemos  al programa principal e imprimimos las variables otra vez. Observamos que el valor de “pecans” es el mismo que antes de ir a la función, ya que como sabemos,            C conserva el valor de las variables, y la función trabaja con una copia de las mismas. En el caso de “apples“, hacemos una copia del puntero y la entregamos a la función. Dado que teníamos una variable original, y una copia del puntero, siempre podremos acceder a la variable y cambiarla, todo ello dentro de la función. Cuando volvemos al programa principal, vemos que “apples” ha cambiado de valor, cuando la sacamos por pantalla.

Usando un puntero en una función, accedemos a los datos de esa función, y podemos modificarlos, como hemos visto en el ejemplo. Debemos destacar que si modifica el valor de un puntero, tendrá el valor del mismo restaurado al volver al programa principal, ya que la función usa una copia, permaneciendo el puntero a salvo de modificaciones. En este programa no habría puntero en el programa principal, ya que recibíamos el puntero de la función, pero podríamos usar punteros en las llamadas a funciones. Una de las aplicaciones en las que son útiles los punteros en las llamadas a función, es cuando se necesiten datos de entrada, usando rutinas standard de E/S. Serán vistas más adelante.

COMBINANDO PUNTEROS Y ESTRUCTURAS

STRUCT3.C

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47 | main()    {    struct {    char initial;    int age;    int grade;    } kids[12],\*point;    int index;    for (index = 0;index < 12;index++) {    point = kids + index;    point->initial = 'A' + index;    point->age = index ;    point->grade = 84;    }    kids[3].age = kids[5].age = 17;    kids[2].grade = kids[6].grade = 92;    kids[4].grade = 57;    for (index = 0;index < 12;index++) {    point = kids + index;    printf("%c tiene %d años y un grado escolar de %d\n",    (\*point).initial, kids[index].age,    point->grade);    }    } |

Este programa es idéntico al anterior, excepto en que usa punteros para algunas operaciones.

La primera diferencia aparece en la definición de las variables siguientes a la definición de la estructura. En este programa definimos un puntero que apunta a la estructura. No sería correcto intentar usar el puntero con cualquier otro tipo de variable.

La siguiente diferencia está en el bucle “for”, en donde usamos el puntero para acceder a los campos de datos. Dado que “kids” es un puntero de variable que apunta a la estructura, podemos definir “pointer” en el mismo rango que “kids”. La variable “kids” es una constante, por tanto no puede ser modificada, pero “point” es una variable puntero y puede modificarse su valor según lo necesitemos. Si asignamos a “point” el valor de “kids” debería quedar claro que apuntará al primer elemento del array, una estructura conteniendo 3 campos.

PUNTERO ARITMÉTICO

Añadiendo 1 a “point” se producirá el desplazamiento al segundo elemento del array, porque así están tratados los punteros en C.

El sistema sabe que la estructura contiene 3 variables y también sabe cuanta memoria hace falta para almacenar la estructura entera. Por tanto, si le decimos que el puntero vale 1 más, el sistema aumentará en 1 unidad la cantidad de memoria necesaria para almacenar  el siguiente elemento del array. Si, por ejemplo, hubiéramos añadido 4 al puntero este avanzaría 4 elementos a través del array. Esta es la razón por la cual un puntero no puede señalar una variable de distinto tipo del que le fue definido al puntero.

A cada paso del bucle, el puntero apuntará al principio de uno de los array, cada vez. Podemos también usar el puntero para referir varios elementos de la estructura. Apuntar un elemento de la estructura con un puntero sucede muy a menudo en C, ya que es un método muy útil. Usando “point->initial” hacemos lo mismo que con”(\*point).initial”, el cual utilizamos en los dos últimos ejemplos. Recordemos que \*point es el dato de la estructura al cual debe señalar el puntero. La expresión “->” se construye con el signo menos y el signo “mayor que”.

Dado que el puntero apunta a la estructura, debemos definir otra vez cual de los elementos deseamos señalar cada vez en la estructura. Existen, como hemos visto, varios métodos para referirse a un miembro de la estructura y, en el bucle “for”, usado para imprimir y en el final del programa utilizamos 3 diferentes métodos. Podrá considerarse una practica de programación algo pobre, pero se hizo con la intención de mostrar que llevan al mismo resultado.

Instrucciones new y delete para la asignación y eliminación de datos en forma dinámica

La forma moderna de manejo de memoria dinámica es a través de las instrucciones new y delete que nos permite tanto la asignación como el borrado de memoria. Veamos un ejemplo sencillo y práctico:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83 | #include <iostream.h>    #include <conio.h>    main()    {    /\*Esta primera parte se muestra el uso de la instrucción new y delete con    enteros\*/    int \*a;    a=new(int);    //se inicializa la variable tipo entera    cout<<"Ingrese un valor entero ";    cin>>\*a;    cout<<endl<<"El valor ingresado es: "<<\*a;    delete(a);     //desaparece de la memoria la variable    getch();    //Esta segunda parte se repite el ejercicio pero con float    float \*b;    b=new(float);  //se inicializa la variable tipo float    cout<<endl<<endl<<"Ingrese un valor real ";    cin>>\*b;    cout<<endl<<"El valor ingresado es: "<<\*b;    delete(b);     //desaparece de la memoria la variable    getch();    /\*en esta tercera parte se utilizará un identificador compuesto, es decir    una estructura y veremos como inicializar el identidficador estructurado\*/    struct VarEstr {  //definición de la estructura y nuevo tipo de variable    int                           c;    float    d;    char                       e;    };    VarEstr \*g;    g=new(VarEstr); //Igualmente que antes en concepto se inicializa el identificador    cout<<endl<<"Ingreso de datos en la estructura";    cout<<endl<<"Ingrese un valor entero ";    cin>>g->c;      //esta es una forma de utilizar la estructura con punteros sin \*    cout<<endl<<"Ingrese un valor real ";    cin>>(\*g).d;      //esta es la otra forma de utilizar la estructura con punteros con \*    cout<<endl<<"Ingrese un valor char ";    cin>>(\*g).e;    cout<<endl<<endl<<"Los valores ingresados son: VALOR ENTERO= "<<(\*g).c<<" VALOR FLOAT= "    <<g->d<<" VALOR CHAR= "<<g->e;    delete(g);      //desaparece de la memoria la variable    getch();    } |

Como vemos para inicializar un identificador determinado basta con emplear la instrucción new y entre paréntesis indicar el tamaño de lo que se va a reservar. Recordemos que cuando se realiza una asignación dinámica de memoria no solo se obtiene la dirección de memoria (asignada al puntero, sino que también se establece qué tamaño se asignará, ya que no es lo mismo reservar memoria para un entero que para un float, o para una estructura, como vemos en el ejemplo).

UN INDICADOR void

A un indicador void puede asignársele el valor de cualquier otro tipo de indicador.  En el programa vemos que el indicador void  “general” se asigna a una dirección de tipo int en la línea 14 y a la dirección de un float en la línea 19, sin ninguna queja del compilador. Esto permite definir un puntero que puede usarse para indicar a datos de diferentes tipos, para transferir información a través de un programa.

ASIGNACIÓN DINÁMICA Y DESASIGNACIÓN

Algunas de las variables que hemos visto han sido automáticas, y por tanto, colocadas dinámicamente en la memoria por el sistema. Las variables de colocación dinámica no existen al cargar el programa y se crean a medida que se van necesitando. Es posible, usando estas técnicas, crear tantas variables como se necesiten, y liberar el espacio ocupado por ellas, cuando lo necesitemos para otras variables.

Veamos el programa NEWDEL.CPP como un primer ejemplo de los operadores **new** y **delete** que sirven para la asignación dinámica y la desasignación (liberación de memoria).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95 | #include <iostream.h>    struct fecha{    int m;    int d;    int a;    };    main()    {    int indice, \*punte1, \*punte2;    \*punte1 = 77;   punte1 = &indice;    punte2 = new int;    \*punte2 = 173;    cout << "Los valores son " << indice << "  " <<  punte1 = new int;                                                 \*punte1 << "  " << \*punte2 << "\n";    punte2 = punte1;    \*punte1 = 999;    cout << "Los valores son " << indice << "  " <<    \*punte1 << "  " << \*punte2 << "\n";    delete punte1;    float \*float\_punte1, \*float\_punte2 = new float;    float\_punte1 = new float;    \*float\_punte2 = 3.14159;    \*float\_punte1 = 2.4 \* (\*float\_punte2);    delete float\_punte2;    delete float\_punte1;    &nbsp;    fecha \*fecha\_punte;    fecha\_punte = new fecha;    fecha\_punte->d = 18;    fecha\_punte->m = 10;  <table cellspacing="0" cellpadding="0">  <tbody>  <tr>  <td width="42" height="31">  <table width="100%" cellspacing="0" cellpadding="0">  <tbody>  <tr>  <td>  <div>    <i>(40)</i>    <i> </i>    <i> </i>    </div></td>  </tr>  </tbody>  </table>  </td>  </tr>  </tbody>  </table>  fecha\_punte->a = 1938;    cout << fecha\_punte->d << "/" << fecha\_punte->m << "/" <<    fecha\_punte->a << "\n";    delete fecha\_punte;    char \*c\_punte;  delete c\_punte;   c\_punte = new char[37];  delete c\_punte;   c\_punte = new char[sizeof(fecha) + 133];    } |

Las líneas 13 y 14 ilustran el uso de indicadores tradicionales de C y la línea 15 ilustra el uso del nuevo operador. Este operador requiere un modificador que debe ser un tipo de variable (int, float, etc) como se ilustra aquí. El indicador “punte2” indica a la variable entera dinámicamente asignada.  La línea 17 ilustra mostrando por pantalla el valor que se asignó en la línea 16.

La línea 19 asigna otra nueva variable y la línea 20 utiliza “punte2” para referirse a la misma variable asignada dinámicamente a la que indica “punte1”.  En este caso, la referencia previa de “punte2” a otra variable se ha perdido y ya no se podrá usar ni desasignar.  Se pierde ese espacio en memoria hasta que regresemos al sistema operativo. Esta no es, obviamente, una práctica buena.

En la línea 24, “punte1” es desasignado con el operador delete, entonces, “punte2” también es desasignado porque apuntaban a la misma dirección. El indicador “punte1” en sí mismo no fue borrado, está todavía indicando a la misma posición de memoria, pero los datos originales se han borrado. Se podría intentar recuperar estos datos usando nuevamente “punte1”, pero sería una práctica terrible para un programador, ya que no se tiene ninguna garantía de lo que el sistema ha hecho con el indicador o los datos, porque ese espacio en la memoria ha sido liberado con la palabra delete.

Con esta definición del operador delete, sería legal pedir al sistema que borrara los datos indicados por un puntero NULL, pero en realidad sería desperdiciar líneas del programa, porque no se puede asegurar lo que pasará. El operador delete se utiliza únicamente para borrar datos indicados por un operador new.  Si se utiliza el delete con cualquier otro tipo de datos, la operación es indefinida y no ocurre nada.

En la línea 26, se declaran algunas variables de coma flotante sólo para recordar que las declaraciones de variables se pueden realizar en cualquier lugar del programa.

Luego, en las líneas 34 a 40 se dan ejemplos del uso de una estructura, lo que también ya se ha visto anteriormente y no necesita mayor explicación.

Finalmente, con el operador new puede destinarse un bloque de tamaño arbitrario utilizando la construcción ilustrada en la que línea 46, donde se destina un bloque de 37 datos tipo char, que serán 37 bytes . Un bloque de 133 bytes más que el tamaño de la estructura de fecha se destina en la línea 48.

LOS INDICADORES A FUNCIONES

Veamos  el programa PNT\_FUNC.CPP como un ejemplo del uso de un indicador a una función.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57 | #include <stdio.h>    void imp\_mensaje(float dato\_list);void imp\_relleno(float dato\_a\_ignorar);    void imp\_float(float dato\_a\_impr);    void (\*pnt\_funcion)(float);    main()    {    float pi = 3.14159;    float doble\_pi = 2.0 \* pi;    pnt\_funcion = imp\_relleno;   imp\_relleno(pi);    pnt\_funcion(pi);    pnt\_funcion = imp\_mensaje;    pnt\_funcion(doble\_pi);    pnt\_funcion(13.0);    pnt\_funcion = imp\_float;    pnt\_funcion(pi);    imp\_float(pi);    }    void imp\_relleno(float dato\_a\_ignorar)    {    printf("Esta es la impresión de la función relleno.\n");    }    void imp\_mensaje(float dato\_list)    {    printf("El dato a listar es %f\n", dato\_list);    }    void imp\_float(float dato\_a\_impr)    {    printf("El dato a imprimir es %f\n", dato\_a\_impr);    } |

Un indicador a una función está disponible tanto en el C++ como en el ANSI – C aunque no es usado regularmente por la mayoría de los programadores en C.

No hay nada de particular en este programa a excepción del indicador a una función declarado en la línea 6.  Se declara un indicador nulo a una función que no  devuelve nada y requiere un único parámetro formal, una variable de tipo float.

Las tres funciones declaradas en las líneas 3 a 5 se ajustan a este perfil y son por lo tanto candidatas a ser llamadas con este indicador.

En la línea 13 se llama a la función imp\_relleno () con pi como parámetro y en la línea 14 se asigna el indicador de función llamado “pnt\_funcion” al valor de imp\_relleno (), y se usa nuevamente el indicador para llamar a la misma función en la línea 15.  Las líneas 13 y 15 son idénticas en lo que realizan a causa de la línea 14. En las líneas 16 a 21 se dan más ilustraciones acerca del uso de indicadores de función.

Desde que se asigna el nombre de una función al puntero, y no hay error de asignación, el nombre de la función debe ser un indicador a esa función.  Este es exactamente el caso.  Un nombre de función es un puntero a esa función, pero es una constante y no puede cambiarse.

Como el nombre de la función es un indicador constante a esa función, se puede asignar el nombre de la función al puntero y usarlo para llamar a dicha función.  La única advertencia es que el valor de regreso y el número y los tipos de los parámetros deben ser idénticos.