Facultad: Ingeniería
Escuela: Computación
Asignatura: Programación II

Tema: Repaso sobre uso de Funciones, Arreglos y Punteros en C++.

Objetivo

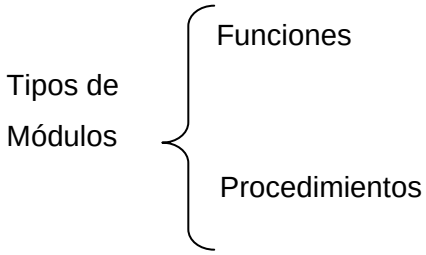
- Utilizar la sintaxis de las funciones definidas por el usuario para resolver problemas.
- Aplicar en ejemplos el paso de parámetros en funciones.
- Implementar arreglos para resolver problemas.
- Elaborar programas que utilice punteros.
- Utilizar este tipo de almacenamiento dinámico para el desarrollo de aplicaciones.

Materiales y Equipo

- Guía Número 1
- Computadora con programa Dev C++.

Introducción

Un problema complejo se puede dividir en pequeños subproblemas más sencillos. Estos subproblemas se conocen como “Módulos” y su complementación en un lenguaje se llama subprograma (procedimientos y funciones).
Un subprograma realiza las mismas acciones que un programa, sin embargo, un subprograma lo utiliza solamente un programa para un propósito especifico.
Un subprograma recibe datos de un programa y le devuelve resultados (el programa “llama” o “invoca” al subprograma, este ejecuta una tarea específica y devuelve el “control” al programa que lo llamo).



Sintaxis de una Función
tipo_devuelto **nombre_funcion**(tipo(s)_argumento(s) **nombre(s)**)
{ [argumento(s)_opcional(es)]

```
        ...
        (declaración de datos y cuerpo de la función)
        ....
        return (valor)  // valor devuelto por la función
}
```

Sintaxis de un Procedimiento

```
void nombre_funcion(tipo(s)_argumento(s) nombre(s))
{                           [argumento(s)_opcional(es)]
        ...
        (declaración de datos y cuerpo de la función)
        ....
}
```

Prototipos de las Funciones.

C++ requiere que una función se declare o defina antes de su uso. La declaración de una función se denomina prototipo. Los prototipos de una función contienen la misma cabecera de la función, con la diferencia de que los prototipos terminan con un punto y coma (;). Específicamente un prototipo consta de los siguientes elementos:

1. nombre de la función
2. lista de argumentos encerrados entre paréntesis y un punto y coma.

La inclusión del nombre de los parámetros es opcional.

Arreglos.

Los arreglos (arrays) permiten almacenar vectores y matrices. Los arreglos unidimensionales sirven para manejar vectores y los arreglos bidimensionales para matrices. Sin embargo, las matrices también se pueden almacenar mediante arreglos unidimensionales y por medio de apuntadores a apuntadores.

Sintaxis Arreglo Unidimensional:

Tipo *nombre* [longitud]; // Declaración.

Tipo puede ser: int, float, double, char, string, etc.
Longitud: es el número de posiciones o elementos que tiene el arreglo.
nombre[i]; //posición o elemento i.

El índice i puede tomar valores desde 0 hasta (longitud -1).

Sintaxis de un arreglo bidimensional o matriz:

Tipo_dato nombre[FILAS][COLUMNAS];

Punteros

Los punteros son variables que contienen direcciones de memoria como sus valores.

Sintaxis:

<tipo_de_dato> * <nombre_puntero>;

El tipo de datos de una variable de tipo puntero se corresponde con el tipo de dato de la variable a la que apunta.
Los punteros pueden ser inicializados cuando son declarados o en un enunciado de asignación. Pueden ser inicializados a 0, “NULL” ó a una dirección; un puntero con un valor NULL apunta a nada.

Para declarar un puntero se debe utilizar un tipo de dato.
int *ptrentero;
float *ptrreal;
char *ptrcdireccion;

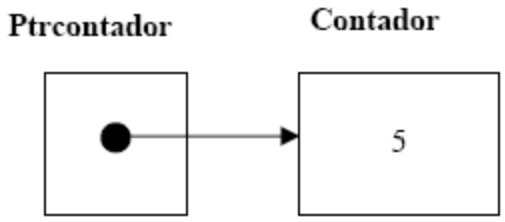
Luego no se podrá utilizar un puntero declarado **int** con variables **char** o **float**.
En un programa pueden aparecer errores en tiempo de ejecución y advertencias en tiempo de compilación cuando se define un puntero a un determinado tipo de dato y, a continuación, se utiliza para apuntar a algún otro tipo de dato.

Operadores para punteros.
El operador de dirección **(&)** es un operador unario que devuelve la dirección de memoria de su operando.
El operador de indirección (*****) regresa el valor del objeto hacia el cual se apunta.

Por ejemplo, suponga las siguientes declaraciones:
int y=5;
int *ptry;

la instrucción: **ptry=&y;**

Asigna la dirección de la variable “y” a la variable de apuntador **ptry**. Entonces, se dice que **ptry** “apunta” a **y**.
Ahora, **ptry** hace referencia de manera indirecta al valor de la variable y observe que el & de la instrucción de asignación anterior no es el mismo que el & de una declaración de variable de apuntador, el cual siempre es precedido por el nombre del tipo de dato.
En la siguiente figura se observa una representación esquemática de la definición antes descrita:



Operaciones permitidas:
Sean ptr1, ptr2 punteros a objetos del mismo tipo, y n un tipo entero o una enumeración; las operaciones permitidas y los resultados obtenidos con ellos son:

Operación	Resultado	Comentario
ptr1++	puntero	Desplazamiento ascendente de 1 elemento.

ptr1--	puntero	Desplazamiento descendente de 1 elemento.
prt1 + n	puntero	Desplazamiento ascendente n elemento.
prt1 - n	Puntero	Desplazamiento ascendente n elemento.
prt1 – ptr2	Entero	Distancia entre elementos
prt1 == NULL	booleano	Siempre se puede comprobar la igualdad o desigualdad con NULL.
prt1 != NULL	booleano	
prt1 <R> ptr2	Booleano	<R> es una expresión relacional.
Prt1 = ptr2	Puntero	Asignación
Ptr1 = void	Puntero genérico	Asignación.

La comparación de punteros solo tiene sentido entre punteros a elementos de la misma matriz; en estas condiciones los operadores relacionales (==, !=, <, >, <=, >=), funcionan correctamente.
NOTA: La suma de punteros no tiene sentido y no está permitida. La resta sólo tiene sentido cuando ambos apuntan al mismo vector y nos da la "distancia" entre las posiciones del vector (en número de elemento).

Punteros a Funciones.
En C++ se pueden declarar punteros a funciones.

Sintaxis:

tipo * identificador (lista de parámetros)

De esta forma se declara un puntero a una función que devuelve un valor de tipo <tipo> y acepta la lista de parámetros especificada. Es muy importante usar los paréntesis para agrupar el identificador, ya que de otro modo estaríamos declarando una función que devuelve un puntero al tipo especificado y que admiten la lista de parámetros indicada.
No tiene sentido declarar variables de tipo función, es decir, la sintaxis indicada, prescindiendo del '*' lo que realmente declara es un prototipo, y no es posible asignarle un valor a un prototipo, como se puede hacer con los punteros, sino que únicamente podremos definir la función.

Ejemplos:

```
int (*pfuncion1)(); (1)
void (*pfuncion2)(int); (2)
float *(*pfuncion3)(char*, int); (3)
void (*pfuncion4)(void (*)(int)); (4)
int (*pfuncion5[10])(int); (5)
```

El ejemplo 1 declara un puntero, "pfuncion1" a una función que devuelve un "int" y no acepte parámetros.
El ejemplo 2 declara un puntero, "pfuncion2" a una función que no devuelve valor y que acepte un parámetro de tipo "int".
El ejemplo 3 a una función que devuelve un puntero a "float" y admite dos parámetros: un puntero a "char" y un "int".
El ejemplo 4 declara una función "pfuncion4" que no devuelve valor y acepta un parámetro. Ese parámetro debe ser un puntero a una función que tampoco devuelve valor y admite como parámetro un "int".
El ejemplo 5 declara un array de puntero a función, cada una de ellas devuelve un "int" y admite como parámetro un "int".

Utilidad de los punteros a funciones.
La utilidad de los punteros a funciones se manifiesta sobre todo cuando se personalizan ciertas funciones de librerías. Podemos por ejemplo, diseñar una función de librería que admita como parámetro una función, que debe crear el usuario (en este caso otro programador), para que la función de librería complete su funcionamiento.
Este es el caso de la función “qsort”, declarada en “stdlib”. Si nos fijamos en su prototipo.

```
void qsort(void *base, size_t nmemb, size_t tamanyo,
   int (*comparar)(const void *, const void *));
```

Vemos que el cuarto parámetro es un puntero a una función “comparar” que devuelve un “int” y admite dos parámetros de tipo puntero genérico.
Esto permite a la librería “stdlib” definir una función para ordenar arrays independientemente de su tipo, ya que para comparar elementos del array se usa una función definida por el usuario, y “qsort” puede invocarla después.

Asignación de punteros a funciones.
Una vez declarado uno de estos punteros, se comporta como una variable cualquiera, podemos por lo tanto, usarlo como parámetro en funciones, o asignarle valores, por supuesto, del mismo tipo.

```
int funcion();
...
   int (*pf1)();  // Puntero a función sin argumentos
                  // que devuelve un int.
   pf1 = funcion; // Asignamos al puntero pf1 la
                  // función "funcion"
...
int funcion() {
   return 1;
}
```

La asignación es tan simple como asignar el nombre de la función.
Nota: Aunque muchos compiladores lo admiten, no es recomendable aplicar el operador de dirección (&) al nombre de la función *pf1 = &función;.* La forma propuesta en el ejemplo es la recomendable.

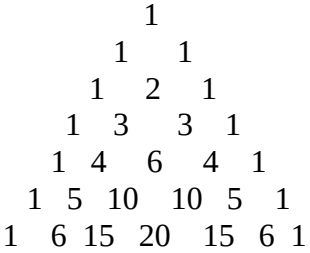
Llamadas a través de un puntero a función.
Para invocar a la función usando el puntero, sólo hay que usar el identificador del puntero como si se tratase de una función. En realidad, el puntero se comporta exactamente igual que un “alias” de la función a la que apunta.

int x = pf1 ( );

De este modo, llamamos a la función “funcion” previamente asignada a * pf1.

Procedimiento

EJEMPLO No. 1: Escribir un programa que permita visualizar el triángulo de Pascal.



En el triángulo de Pascal cada número es la suma de los dos números situados encima de él. Este problema se debe resolver utilizando arrays y funciones.

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

using namespace std;

int dibujatriangulo(int h);
int h, n, i, j;
int trian[100][100]={0};

int main()
{  system("cls");

   do
   {  cout<<"Ingrese el Grado del Triangulo[1-100] = \n";
      cin>>h;
   }
   while(h<1 || h>100);

   system("cls");
   cout<<"El Triangulo de Pascal definido es: \n ";

   dibujatriangulo(h);
   system("pause");
}

int dibujatriangulo(int h)
{
   n = (2*h)-1;
   trian[0][h-1] = 1;
   for (i = 1; i <= h; i++)
   {   for (j = h+1-i; j <= i-h+n ; j++)
       {  trian[i][j] = trian[i-1][j-1] + trian[i-1][j+1];
       }
   }
   for (I = 1;I <= h;i++)
```

```
  {   for (j = 1;j <= n;j++)
      {  if (trian[i][j] != 0)
         { if (h < 10)
           { if (trian[i][j] > 9)
               cout<<trian[i][j];
             else
               cout<<" "<<trian[i][j];
           }
           else
             if (trian[i][j] > 9)
               cout <<  trian[i][j];
             else
               cout << " " << trian[i][j];
        }
        else
        {   cout << "   ";
        }
      }
      cout << endl;
  }
}
```

EJEMPLO No. 2: Escriba un programa que contenga dos arrays con datos enteros y utilizando punteros genere un tercer arrays con la suma de los dos arrays creados con anterioridad.

```
#include<iostream>

using namespace std;

int main()
{
   int *prt_array_1, *prt_array_2, n;

   cout << "Ingrese el numero de elementos de los arreglos"<<endl;
   cin >> n;

   int array_1[n], array_2[n], array_f[n];
   prt_array_1 = array_1;
   prt_array_2 = array_2;
   cout<< "***** Llenando arreglo 1 *****"<<endl;
   for(int i = 0; i < n; i++)
   {
         cout << "Ingrese el elemento "<<i+1<<": ";
         cin >> array_1[i];
         }
   cout << endl;
   cout << "***** Llenando arreglo 2 *****"<<endl;
   for(int i = 0; i < n; i++)
   {
```

```
        cout << "Ingrese el elemento "<<i+1<<": ";
        cin >> array_2[i];
        }
    cout << endl;
    system("PAUSE");
    system("CLS");
    cout << "***** Imprimiendo la suma del arreglo 1 + arreglo 2 *****"<<endl;
    for(int i = 0; i < n; i++)
    {
        array_f[i] = *(prt_array_1 + i) + *(prt_array_2 + i);
        cout << "[" << array_f[I] << " ] ";
        }
    cout << endl;
    system("PAUSE");
    return (0);
}
```

Análisis de resultados

Ejercicio No. 1
Desarrolle con arreglos y funciones un programa que lea para N alumnos los datos siguientes: carnet, edad y nota.
A partir de estos datos realizar un menú con las siguientes acciones:
a) Solicitar los datos al usuario.
b) Calcular e imprimir la nota promedio
c) Calcular e imprimir edad promedio de los alumnos que obtuvieron una nota igual o mayor a 6.0
d) Imprimir Listado que incluya Carnet, Edad y nota correspondiente de todos los alumnos en forma ascendente (con respecto a la nota)

Bibliografía

- En la biblioteca de la UDB:
  Programación y Diseño en C++: Introducción a la programación y al diseño orientado a objetos
  Cohoon, James.
  No. Clasificación: **005.362 C678 2000**

  Programación en C, Metodología, estructurada de datos y objetos, Luis Joyanes Aguilar Ignacio Zahonero Martínez, McGrawHill.

Guía 1: Repaso sobre uso de Funciones, Arreglos y Punteros en C++.

Hoja de cotejo: 1

Alumno: Máquina No:

Docente: GL: Fecha:

EVALUACION					
	%	**1-4**	**5-7**	**8-10**	**Nota**
CONOCIMIENTO	Del 20 al 30%	Conocimiento deficiente de los fundamentos teóricos	Conocimiento y explicación incompleta de los fundamentos teóricos	Conocimiento completo y explicación clara de los fundamentos teóricos	
APLICACIÓN DEL CONOCIMIENTO	Del 40% al 60%				
ACTITUD					
	Del 15% al 30%	No tiene actitud proactiva.	Actitud propositiva y con propuestas no aplicables al contenido de la guía.	Tiene actitud proactiva y sus propuestas son concretas.	
TOTAL	100%				