Universidad Austral de Chile
Facultad de Cs. de la Ingeniería
Instituto de Informatica

Apuntes de Clases
INFO 161 : El Lenguaje de Programación C++

Prof. Luis Alberto Alvarez González

Valdivia, Marzo de 1999.-

Tabla de Contenidos

1.Introducción

1.1. Lenguajes de Computación.

- Lenguaje. Cualquier notación de representar algoritmos o estructuras de datos es considerado un lenguaje.
- Pseudolenguajes. Notación informal, pero "entendible".
- Lenguajes de Especificación. Para especificar como deben hacerse un determinado sistema. Existen formales. Z, TROLL, etc. Se estudian en Ingeniería de Software. No son implementados en computador.
- Lenguajes de Programación de Computadores. FORTRAN, COBOL, PASCAL, C, ADA, C++, Visual Basic, etc.

1.1.1. Historia de los Lenguajes

1950-55	Computadores Primitivos. Lenguaje de Máquina Lenguaje Ensamblador.	Muy primitivos números binarios o hexadecimales. mnemónicos.
1956-60	FORTRAN ALGOL COBOL LISP	Formula TRANslation ALGoritms Language COmmon Bussines Oriented Language. LISt Processing.
1961-65	SNOBOL APL	
1966-70	PL/I SIMULA BASIC	
1971-75	Pascal	
1976-80	Ada C Prolog	
1981.	C++, Smalltalk.	
1990	Visual Basic, Visual C++	

1.2. Paradigmas de Programación

Los paradigma son estilos de programación.

- Programación por Procedimiento.
- Programación Modular
- Programación orientada a listas.
- Programación orientada a objetos.

Paradigma : Estilo o forma de programar. No depende del lenguaje.

Existen lenguajes más aptos para un determinado paradigma.

Ejemplo,

Pascal para programación por procedimiento.

Lisp y Prolog, para manejo de listas.

Smaltalk, para POO.

Modula, para programación modular.

1.2.1. Programación por procedimiento o paradigma procedural.

¿ Que procedimientos se desean ? => Elegir los mejores algoritmos.

Los lenguajes que apoyan este paradigma proporcionan recursos para pasar argumentos a las funciones o procedimientos y que entregan como resultado. Este tipo de programación parte con FORTRAN, sigue ALGOL y ahora Pascal y C.

Ejemplo

```
ordenar_lista(lista)
{
    // codigo del procedimiento
}
```

1.2.2. Programación modular.

Modulo. Conjunto de procedimientos.

¿ Que módulos se necesitan ? => Dividir el programa y que los datos queden ocultos.

Muy común y usado en la construcción de grandes sistemas de computación. El lenguaje Modula apoya directamente este estilo de programación.

ejemplo en C. Un ejemplo clásico es el manejo de pilas, lo que desde un punto modular se transforma en:

1. Ofrecer al usuario una interfaz para la pila. por ejemplo funciones meter() y sacar()
2. Garantizar que se puede tener acceso a la pila a través de una interfaz con el usuario.
3. Garantizar la inicialización de la pila

ejemplo en C

```
// declaración de la interfaz para el
// modulo pila de caracteres.
void meter(char);
char sacar();
const int capacidad_pila = 100;
```

Suponiendo que la interfaz se encuentra en pila.h

```
#include “pila.h”              // emplea la interfaz
static char v(capacidad_pila);// static significa
                              // global
static char* p=v;             // al principio la pila
                              // esta vacía

void meter(char c)
{
    // verifica si la pila tiene espacio y mete c
}

char sacar()
{

    // verifica que no este vacia la pila y saca.
}
```

Note que el usuario, desde fuera del archivo/modulo tiene acceso a la pila, puesto que las variables se declaran locales respecto al módulo (`static`).

1.2.3. Programación orientada a objetos (POO).

Supongamos que una figura cualquiera tiene color y esta compuesta por puntos, de esta forma se define

```
class figura {
    punto centro;
    color col;
// ...
public :
    punto donde()  { return centro; }
    void mover(punto hacia) { centro = hacia;
                                    dibujar ();}
    virtual void dibujar();
    virtual void girar();
    //...
};
```

`void` se usa con funciones que no retornan valor
`virtual,` se usa para funciones que virtualmente serán usadas.

Algunas funciones generales

```
void girar_toda(figura v[ ], int tam, int angulo)
//
// girar "angulos" grados a todos los primeros "tam"
// miembros del arreglo v.
{
    int i=0;
    while (i<tam) {
        v[i].girar(angulo);
        i=i+1;
    }
}
```

Como definir una figura específica:

```
    class circulo : public figura {
        int radio;
    public:
        void dibujar(){ /*...  */ };
        void girar(int) {}
};
```

Se dice que `circulo` es una “clase derivada” de la “clase base” `figura`. Otra forma es decir que `circulo` es una “subclase” y `figura` es una “superclase”.

¿ Que clases se desean ?
=>
Definir relaciones funciones y relaciones entre clases

2.El lenguaje C++

2.1. Introducción

Una mejor versión de C.

Programa y salidas

ejemplo

```
#include <iostream.h>

int main()

{
   cout << “Hola INFO 161 \n”;
}
```

`#include <iostream.h>` Contiene las funciones de E/S.
`int` le dice que la salida de la función `main` es entero, no es necesario para `main.`

2.1.1. variables y aritmética

ejemplo

```
int pulgada;
```

2.1.2. tipos fundamentales

enteros	punto flotante o reales

`char`
`short`
`int`
`long`
	`float`
	`double`
	`long double`

operadores aritméticos	operadores de comparación
+	= =
-	!=
*	<
/	>
%	<=
	>=

2.1.3. Asignaciones

```
double d;
int i;
short s;
//...
d=d+i;
i=s*i;
```

2.1.4. Punteros y Arreglos

Ejemplo

```
char v[10];  // arreglo de 10 caracteres
   char *p;  // puntero a un caracter
```

`v[10]` tiene 10 elementos de `v[0]` a `v[9]`

```
      p=&v[3]     // apunta al cuarto
elemento
```

& es “dirección de “

2.1.5. Pruebas y ciclos

Se escribirá un programa que convierta pulgadas a centímetros y viceversa

```
#include <iostream.h>

int main()
{
   const float factor=2.54;
   float x,pulg,cm;
   char car=0;

   cout << "ingrese longitud: ";

   cin >> x;    // leer número
   cin >> car;  // leer sufijo

   if (car == 'p') {   //pulgadas
      pulg=x;
      cm=x*factor;
   }
   else if (car=='c') {   //centímetros
      pulg=x/factor;
      cm=x;
   }
   else
      pulg=cm=0;
   cout << pulg << " pulgadas = " << cm
<< " centímetros\n";
}
```

otra forma es reemplazar el if usando la instrucción `case`, es decir :

```
switch(car) {
case 'p':
   pulg=x;
   cm=x*factor;
   break;
case 'c':
   pulg=x/factor;
   cm=x;
   break;
default:
   pulg=cm=0;
   break;
}
```

2.2. Funciones

ejemplo

```
extern float pot(float,int);
   // pot() se define en otro lugar
int main()
{
   for(int i=0; i<10; i++) cout <<
pot(2,i)<<'\n',
}
```

aquí el compilador convierte el 2 a `float`

Una forma de definir `pot` podría ser

```
float pot(float x, int n)
{
   switch(n)  {
   case 0: return 1;
   case 1: return x;
   default: return x*pot(x,n-1);
   }
}
```

Se podrían definir dos funciones, una para enteros y otra para reales con el mismo nombre en cuyo caso el compilador elige la más apropiada,

ejemplo

```
int pot(int , int);
double pot(double, double);
//...
x=pot(2,10);     // llamar a pot(int,int)
y=pot(2.0,10.0); // llamar a
pot(double,double)
```

esto se llama sobrecarga.

2.2.1. Paso por valor y por referencia

por valor

```
void inter(int* p, int* q)
{
int t=*p;
*p=*q;
*q=t;
}
```

el operador * devuelve el valor del objeto al que apunta el puntero. Para llamarla

```
void f(int i, int j)
{
   inter(&i, &j);
}
```

Paso por referencia

```
void inter(int& r1, int& r2)
{
   int t=r1;
   r1=r2;
   r2=t;
}

void g(int i, int j)
{
   inter(i,j);
}
```

el `int&` significa una referencia a `int` y en este caso `r1` y `r2` son variables sinónimos.

Módulos

Un programa en casi simple se compone de varios módulos que se compilan en forma independiente y la declaración `extern` permite referenciar a una función u objeto definido en otro modulo,

Otra forma es creando archivos encabezados (`headers`) en cuyo caso se colocan al principio del programa y son `.h`

por ejemplo para usar `sqrt()` se debe incluir `math.h`

si los archivos a incluir se encuentran en el directorio estándar se colocan entre parántices angulares.

Si los archivos no están en el directorio estándar se encierran entre comillas y eventualmente el path.

3. Declaraciones y constantes

Se presentan los tipos fundamentales (char, int, float, etc.) y las formas básicas de derivar nuevos tipos (funciones, arreglos, punteros etc.)

Declaraciones.

Antes de usar un nombre se debe declararlo
Ejemplo

```
char car;
int cuenta = 1;
char* nombre = “Joel”;
struct complejo {float re, im;},
complejo varcom;
extern complejo sqrt(complejo);
extern int número_error;
typedef complejo punto;
float real(complejo* p) {return p->re; };
const double pi=4.14159265358979323385;
struct usuario;
template<class T> abs(T,a) {return a<0 ? -a:a;}
enum perro {Bulldog, Chihuahua, Terrier };
```

Las declaraciones pueden ser definiciones.
Las declaraciones pueden o no definir un valor por ejemplo,

```
char car;
complejo varcom;
```

no definen ningún valor

```
int cuenta = 1;
char* nombre = “Joel”;
struct complejo {float re, im;},
typedef complejo punto;
float real(complejo* p) {return p->re; };
template<class T>abs(T,a) {return a<0 ? -a : a; }
```

```
enum perro {Bulldog, Chihuahua, Terrier };
```

Las variables generalmente cambian su valor dentro del programa, sin embargo otras se definen como constantes, es decir, su valor no cambiará, por ejemplo,

```
const double pi=4.14159265358979323385;
```

Otras declaraciones no son definiciones, por ejemplo :

```
extern complejo sqrt(complejo);
extern int número_error;
struct usuario;
```

otras tipos de definiciones,

```
typedef complejo punto;
```

Permite que punto sea sinónimo de complejo; es decir

```
complejo varcom;
```

es equivalente a:

```
punto varcom;
```

```
template<class T>abs(T,a) {return a<0 ? -a : a; }
```

permite que a `abs` se le ingrese el tipo como parámetro.

```
enum perro {Bulldog, Chihuahua, Terrier };
```

es otra forma de enumerar entero.

las variables no se pueden definir más de una ves, ejemplo

```
char car;
char car;   // ¡ error !.
```

es este caso no hay error

```
extern char car;
extern char car;
```

Porque es una declaración, la definición está en otro archivo.
Si en una declaración no concuerdan los tipo, se produce un error, por ejemplo,

```
extern int número_error;
extern short número_error;
```

Alcance. (Variables Globales y locales)

Una declaración sólo es válida dentro del bloque. ejemplo

```
int x;
void f()
{
    int x;
    x=1;
    {
        int x;
        x=2;
    }
    x=3;
}
int* p=&x;
```

La primera definición es válida para todo el programa, sin embargo en los bloque siguientes se redefine `x`, y se hace válida para cada bloque.

Se recomienda no usar variables globales u ocultas con nombre como `x` o `i`, porque pueden llevar a confusión.

Se puede usar el operador :: para referirse a variables globales.

ejemplo

```
int x;
void f2()
{
    int x=1;
    :: x=2;
}
```

OBSERVACIÓN :No existe la forma de usar un nombre local desde el exterior,

Un error común es

```
void f(int x)
{
    int x;
}
```

Tiempo de vida.

Una variable nace cuando se declara y muere cuando se termina el bloque dentro del cual fue declarado.

Una forma de declarar nombre globales dentro de un bloque es usando la palabra `static` y aunque se declare dentro de una función que se invoca muchas veces, sólo se iniciará una vez.

Ejemplo,

```
int a=1;
void f()
{
    int b=1;
    static int c=a;
    cout   << “a = “ << a++
        << “b = “ << b++
        << “c = “ << c++ << ‘\n’;
}
int main()
{
    while (a<4) f();
}
```

produce las salida
a = 1 b =1 c = 1
a = 2 b =1 c = 2
a = 3 b =1 c = 3

3.1. Nombres

- El primer caracter debe ser una letra. Se puede usar _.
- No hay restricciones de largo.
- No se pueden usar nombre claves.

Ejemplos de nombre válidos

```
hola   universidad_austral_de_chile

INFO161   _Informatica ___
```

Ejemplo de nombres no válidos

```
012    universidad austral
$hola  class     num.cuenta      else
```

3.2. Tipos

Todo identificador esta asociado a un tipo.

Ejemplo

```
int numero_error;
float real(complejo* p)
```

aquí `numero_error` es un entero (`int`).

- La función real ***debe*** ser invocada pasándole un puntero de tipo `complejo` como argumento.

- `int` y `complejo` son nombres de tipos.

Se puede saber la memoria que ocupa un tipo usando la función `sizeof(tipo)`

Ejemplo.

```
int main() {
       cout << “lo que ocupa un entero
son ”<<sizeof(int)<<“ bytes\n”;
}
```

3.2.1. Tipos fundamentales

Enteros	**Reales**	**enteros sin signo**	**enteros con signo**
char	float	unsigned char	signed char
short int	double	unsigned short int	signed short int
int	long double	unsigned int	signed int
long int		unsigned long int	signed long int

3.2.2. Conversión de tipos.

Ejemplos

```
float f;
char* p;
//...
long ll=long(p);
int i=int(f);
int i1=256+255;
char car=i1;     // car == 255
int i2=car;      // i2 == ?
```

3.2.3. Tipos Derivados.

A partir de los tipos fundamentales se puede derivar otros tipos que son:

```
*   puntero
&   referencia
[ ]arreglo
( )función
```

Ejemplos,

```
char* p[20];
void f(int);
i=*p;     // usa el objeto al que apunta;
int (*p)[10];// puntero a un arreglo.
```

Void

Se comporta como un tipo fundamental, pero sirve para indicar que una función no devuelve valor.

Ejemplos

```
void f();
void* pv;
```

Punteros.

Se denota por T*, donde T es el tipo. Punteros a arreglos y funciones con más complicados.

Ejemplos

```
int* i;
char** aac;       // puntero a puntero char
int (*vp)[10];    // puntero a arreglo de 10 int
int (*fp)(char, char*)
// puntero a función que recibe como
// argumentos (char, char*) y devuelve
// un int
```

La operación fundamental de un puntero es la indirección, es decir, *hace referencia al objeto que apunta*. Es un operación unario.

Ejemplo

```
char c1='a';
char*p = &c1;    //p = dirección de c1
char c2=*p;      // c2=c1;
```

Es posible hacer algunas operaciones con punteros.

Ejemplo

```
int strlen(char* p)  {
    int i=0;
    while (*p++) i++;
    return i;
}
```

otra forma es,

```
int strlen(const char* p)  {
    char*q=p;
    while (*q++);
    return q-p-1;
}
```

Arreglos

Si T es un tipo T[n] es un arreglo de n elementos del tipo T. Los elementos se indican de 0 a n-1

Ejemplo

```
float v[3];
int a[2][5];    //matriz
char* vpc[32];
//arreglo de 32 punteros a caracter
```

un programa para desplegar el valor entero de las letras minúsculas podría ser

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

char alfa[]="abcdefghijklmnopqrestuvwxyz";

    main()
    {
        int n=strlen(alfa);
        for(int i=0; i<n; i++) {
              char car=alfa[i]
              cout<<'\''<<car<<'\''
                 <<"="<<int(car)
                 <<"=0"<<oct(car)
                 <<"=0x"<<hex(car)<< '\n';
        }
    }
```

salida

```
'a' = 97 = 0141=0x61
.
.
```

- No es necesario especificar el tamaño del arreglo,
- `oct()` y `hex()` están declaradas en `iostream.h`
- `strlen` está declarada en `string.h`
- Un string de caracteres declarado como "abcd" termina con 0.

otra forma de declarar arreglos es

```
int v1[]={1,2,3,4};
int v2[]={'a','b','c','d'};
char v3[]={1,2,3,4};
char v4[ ]={'a','b','c','d'};
```

- v4 no termina con 0, ¡ cuidado!.

NOTA: Se recomiendo no especificar arreglo de caracteres de la forma ante indicada.

- La coma (,) es un operador de secuencia,

Luego

```
int matriz[3,4]; // error
```

Lo correcto es

```
int matriz[3] [4];
```

nótese que se declara un arreglo de 4 elementos y cada elemento a su vez es un arreglo de 3 elementos.

Ejemplo

```
char v[2] [5] = {
    { ‘a’,’b’,’c’,’d’,’e’},
    {‘0’,’1’,’2’,’3’,’4’}
};
main() {
    for(int i=0;i<2;i++)  {
        for(int j=0;j<5;j++)
            cout<<“v[“<<i<<“][“<<j<<“]”
                <<v[i] [j]<<“”;
        cout<<‘\n’;
    }
}
```

Punteros y arreglos

Los punteros y arreglos están íntimamente ligados

Ejemplo

```
int main() {
    char alfa[]="abcdefghijklmnopqretuvwxyz";
    char* p=alfa;
    char car;
    while(car=*p++)
        cout<<car<<"="<<int(car)
            <<"=0"<<oct(car)<<'\n';
}
```

También pudo haberse declarado como

```
    char *p=&alfa[0];
```

Cual es el resultado de

```
main(){
    char vc[10];
    int  vi[10];
    char* pc=vc;
    int*  pi=vi;
    cout<<"char* "<<long(pc+1)-long(pc)<<'\n';
    cout<<"int*  "<<long(pi+1)-long(pi)<<'\n';
}
```

que hubiese pasado si se hubiera escrito

```
    cout<<"char* "<<int(pc+1)-int(pc)<<'\n';
    cout<<"int*  "<<int(pi+1)-int(pi)<<'\n';
```

¡Cuidado no se pueden mezclar operaciones entre punteros que apuntan a distintos arreglos !!.

ejemplo

```
int v1[10];
int v2[10];
int i=&v1[5]- &v1[3];
    i=&v1[5]- &v2[3];
int* p=v2 + 2;
    p=v2+2;
```

Estructuras

Un arreglo son varios elementos de un mismo tipo
Una estructura son varios elementos de distinto tipo

Ejemplo

```
struct domicilio {
   char* nombre;
   char* calle;
   int* numero;
   int* telefono;
}
```

Se puede accesar los elementos de un arreglo mediante el operador punto (.)

Por ejemplo

```
domicilio la;
la.nombre="Luis Alvarez"
la.numero=123;
```

También se puede usar la notación para arreglos, ejemplo

```
domicilio la={“Luis Alvarez”,”picarte”,3000,212121}
```

- Lo normal es que se tenga acceso a los elementos de una estructura usando punteros.
- Los punteros a estructuras usan el operador **->**

por ejemplo

```
void imprimir_domicilio(domicilio* p)
{
    cout<<p->nombre<<‘\n’
        <<p->calle<<‘\n’
        <<p->numero<<‘\n’
        <<p->teléfono<<‘\n’
}
```

Los objetos de una estructura se pueden asignar, pasar como argumento de una función y devolverlos como resultado

Ejemplo

```
domicilio actual;
domicilio fijar_actual(domicilio
siguiente)
{
    domicilio previo=actual;
    actual=siguiente;
    return previo;
}
```

NOTA: Las operaciones == y != no están definidas.

En un arreglo los elementos ocupan posiciones de memoria contiguas, de igual forma en los arreglos, sin embargo, podrían haber problemas al tratar de hacer `sizeof()`.

El nombre de un tipo se puede usar inclusive en la estructura, por ejemplo

```
struct dato {
   dato* predecesor;
   dato* sucesor;
};
```

No se puede declarar nuevos tipos para estructura sin haber primero terminado de declarar, ejemplo,

```
struct dato {
   dato predecesor; // ¡¡ error !!
   dato sucesor;
};
```

Dos o mas estructuras se pueden referenciar entre si, ejemplo

```
struct lista;

struct dato;
   dato* pre;
   dato* suc;
   lista* miembro_de;
};

struct lista{
   dato* cabeza;
};
```

4.Punteros.

- Los punteros son direcciones.

- El puntero es una variable que contiene una dirección de memoria.

- El puntero es una variable que apunta a otra.

Existen dos operadores especiales que son:

* : indica que la variable es un puntero
& : entrega la dirección del operando.

ejemplo

```
#include <iostream.h>

main()  {
    int *direcc, cont, val;
    cont=100;
    direcc=&cont;
    val= *direcc;
    cout << val;
}
```

Otro ejemplo

```
#include <iostream.h>

main()  {
    float x=10.1, y;
    int *p;

    p = &x;
    y = *p;
    cout << y;
}
```

4.1. Expresiones con Punteros

Asignación de Punteros.

Ejemplo

```
#include <iostream.h>

main()  {
    int x;
    int *p1, *p2;

    p1=&x;
    p2=p1;
    cout << p2;
}
```

4.2. Aritmética de Punteros.

Los únicos operadores permitidos con punteros son + y -.

Si por ejemplo un puntero `p` a entero esta apuntado a la dirección `0xABCDE` y se realiza la siguiente operación

```
p++
```

quedará apuntando a la dirección .....

También se puede hacer operaciones del tipo `p+n`, por ejemplo,

```
p+9
```

En este caso apuntará al noveno elemento, tomando como base `p`.

Comparación de punteros.

Es posible comparar punteros. Por ejemplo

```
#include <iostream.h>

main()  {
    int *p1, *p2;
    cout << "P1,P2 = " <<p1<< " , " <<p2<< '\n';
    if (p1<p2) cout << "P1 < P2\n";
        else cout << "P2 < P1\n";

}
```

Punteros y Arreglos

Ejemplo 1

```
#include <iostream.h>

main()  {
        char str[]={'a','b','c','d','e','f','g',0}, *p1;
     p1=str;

     cout<< str[4]<<'\n';
     cout<< *(p1+4);

}
```

Ejemplo 2

Versión arreglo

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

main()  {
     char str[80];

     cout <<" ingrese una palabra en mayúsculas ";
     gets(str);
     cout <<"\n palabra en minúsculas ";

     for(int i=0;str[i];i++) cout << char(tolower(str[i]));
}
```

Versión punteros

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

main()  {
     char str[80], *p;

     cout <<" ingrese una palabra en mayúsculas ";
     gets(str);
     cout <<"\n palabra en minúsculas ";

     p=str;
    while(*p) cout << char(tolower(*p++));
}
```

Indexación de Punteros.

```
#include <iostream.h>

main()  {
     int num[5]={1,2,3,4,5}, *p;
     p=num;

         for(int i=0;i<5;i++) cout << p[i]<< '\n';

}
```

NOTA: **`P[t]`** es idéntico a **`p+t.`**

Un ejemplo de Stack. Crear un stack de tal manera

```
#include <iostream.h>

int stack[50], *p1, *tope;
main()  {
     int valor;
     p1=stack;
         tope=stack;

     cout<<"ingrese enteros >s que 0 al stack \n";
     cout << " para sacarlos ingrese 0\n";
     cout << " para terminar ingrese -1 \n";

     do{
         cin >> valor;
         if(valor!=0) {
             *p1=valor;
             p1++;
               }
           else {
           if(p1==tope)
               cout<< "stack vacio";
             else {
               p1--;
               cout<< *p1;
             }
               }
     } while(valor!=-1);

}
```

Punteros y Strings

Un nombre de arreglo sin índice es un puntero al primer elemento del arreglo.

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

main()  {
    char s1[80], s2[80];
    char *p1, *p2;
    p1=s1;
    p2=s2;
    cout << "ingrese dos palabras \n";
    gets(s1);
         gets(s2);
    int flag=1;
    while(*p1&&flag)
       if(*p1-*p2) flag=0;
              else {p1++; p2++;};
    if(flag) cout << "iguales";
       else cout <<"distintos";
}
```

Obteniendo la Dirección de un Elemento de un Arreglo

Un puntero puede inicializarce con la dirección de cualquier elemento. Por ejemplo

```
p=&x[2];
```

Ejemplo

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

main() {
    char str[80], *p;
        int i,t;

    cout << "ingrese un string : ";
    gets(str);
    cout << "\n posición inicial y el tamaño ";
    cout << "de un sub string (i,t) \n";
    cin >> i;
    cin >> t;

    cout << "\n el resultado es \n";
    p=&str[i];
    for(int j=0;j<t;j++) cout << *p++;
}
```

4.3. Arreglo de Punteros

Se puede usar un arreglo de punteros a indicar otro tipo de datos. Por ejemplo

```
int *x[10];
```

De esta forma se puede asignar una variable a uno de los elementos de la siguiente forma

```
x[2]=&var;
```

Ahora `var` se puede usar como

```
*x[2]
```

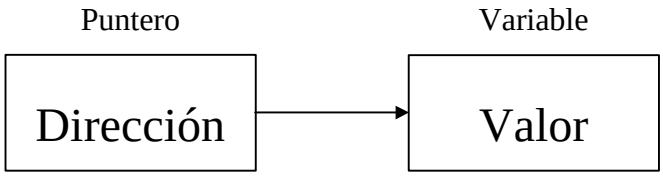
Ejemplo de arreglo de punteros.

```
#include <iostream.h>

main() {
    char *info161[]= {"Cárdenas Luis\n",
              "Chávez Pamela\n",
              "Coronado Moises\n",
              "Díaz Rodrigo\n",
              "Gallegos Walkiria\n",
              "Kruze Ivan\n",
              "Lizama Orlando\n",
              "Loaiza Pablo\n",
              "López Luis\n",
              "López Diego\n"};
   int i;

    cout << "ingrese numero de lista =";
    cin >> i;
    cout << info161[i-1];
}
```

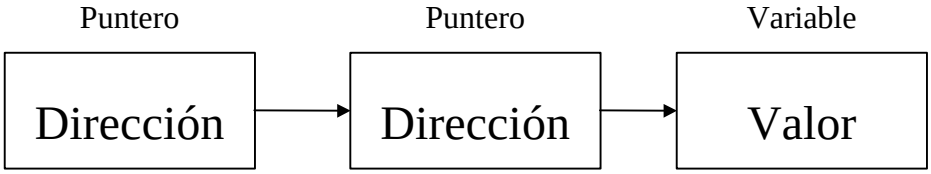
Punteros a Punteros.

Un arreglo de punteros es lo mismo que punteros punteros.

Un puntero a puntero es una forma de indirección múltiple.



Indirección simple



Indirección múltiple.

Ejemplo

```
#include <iostream.h>

main() {
   int x, *p, **q;

   x=10;
   p=&x;
   q=&p;

        cout << **q;
}
```

Inicialización de Punteros.

¡¡¡ Antes de usar un puntero se debe inicializar. !!!

Después de declarar un puntero, se debe inicializar.

Ejemplo,

```
#include <iostream.h>

main() {
    char *p="Hola. ¿ como estan ?";
         while(*p)cout << *p++;
}
```

Problemas con punteros.

Típico.
(1)

```
main() {
    int x, *p;

    x=10;
    *p=x;
}
```

(2)

```
main() {
    int x, *p;

    x=10;
    p=x;
}
```

5.Funciones

5.1. Forma General de una Función.

```
Tipo nombre_función(lista de parámetros)
Declaración de parámetros
{
Cuerpo de la función
}
```

La sentencia `return`

Usos

- Salir de la función
- retornar valores.

Regreso desde una función.

Hay dos formas:

- Hasta que encuentra el `}`
- Hasta que encuentra **`return`**

Ejemplo 1.

```
void pr_reverse(char *s) {
    for(int t=strlen(s)-1;t+1;t--) cout<<s[t];
}
```

Ejemplo 2.

```
void potencia(float base, int exponente) {
    if(exponente<0) {cout <<"no se calcular";
        return ;
    }
    float i=1;
    for(;exponente;exponente--) i=base*i;
        cout<<" La respuesta es :"<<i;
}
```

Programas completos.

Programa 1

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

void pr_reverse(char *s);

main()  {
    char str[80], *p; p=str;
    cout<<"ingrese una frase : "; gets(str);
    cout <<'\n';
    pr_reverse(p);
}
```

```
void pr_reverse(char *s) {
    for(int t=strlen(s)-1;t+1;t--) cout<<s[t];
}
```

Programa 2.

```
#include <iostream.h>

void potencia(float base, int exponente);

main()  {
     float x; int y;
     cout<<"ingrese un número : "; cin >> x;
     cout<<" ahora su exponente : ";cin >> y;
     cout <<'\n';
     potencia(x,y);
}
```

```
void potencia(float base, int exponente) {
     if(exponente<0) {cout <<"no se calcular";
          return ;
     }
     float i=1;
     for(;exponente;exponente--) i=base*i;
          cout<<" La respuesta es :"<<i;
}
```

5.1.1. Retorno de Valores.

Todas las funciones, excepto las `void`, retornan un valor.

- Una función puede ser asignada a un valor, por ejemplo `x=abs(y);`
- o no ser asignada como `swap(x,y);`

5.2. Reglas de Alcance de Funciones

Hay tres tipos de variables en funciones:

- Variables locales
- Parámetros formales
- Variables Globales.

5.2.1. Variables Locales.

Todas las variables declaradas dentro del bloque de la función son locales.

Ejemplo

```
func1() {
int x=10;
}
func2() {
int x=-199;
}
```

Otro ejemplo

```
f(){
   char ch;
   cout<<" ¿continua (s/n) ? :";
   if(ch=="s")  {
      char s[80];
      cout<<"ingrese el nombre :";
      gets(s);
      almacenar(s);
   }
}
```

Ventaja, se reserva memoria sólo si es necesario.

5.2.2. Parámetros Formales.

Si una función usa argumentos, estos deben estar asociados a variables que acepten estos argumentos. Estas variables son llamadas *parámetros formales*.

Los parámetros formales se comportan como variables locales dentro de la función.

Ejemplo

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

int esta_en(char *s, char letra);

main()  {
    char str[80], c, *p; p=str;
    cout<<"ingrese una frase : "; gets(str);
    cout <<"\n ingrese un caracter :"; cin >> c;

    cout << esta_en(p,c);
}
```

```
int esta_en(char *s, char letra){
   while(*s)
      f(*s==letra) return 1;
     else s++;
       return 0;
}
```

5.2.3. Variables Globales

Las variables globales son vistas en todo el programa y en cada parte del programa.

Ejemplo

```
#include <iostream.h>

void func1();
void func2();

int conta;

main() {
    conta=100;
    func1();
}

void func1() {
    int temp=conta;
    func2();
    cout << "conta es " << conta;
}

void func2() {
    for(int conta=1;conta<10;conta++)
cout<<".";
}
```

5.3. Argumentos de una función.

Las formas de pasar los argumentos a una función son:

- Llamado por valor, y
- llamado por referencia.

- En el llamado por valor se copia el valor como parámetro. Por lo tanto los cambios que se hacen a los parámetros de la función no producen efecto sobre las variables que la usan.
- En el llamador por referencia se copia una dirección como parámetro. Dentro de la función, la dirección es usada para accesar los parámetros actuales. Esto significa que los cambios que se hagan afectan la variable en la función.

Ejemplo

```
main() {
    int t=10;
    cout << sqrt(t)<<"  "<<t;
}

sqrt(int x) {
    x=x*x;
    return (x);
}
```

5.3.1. Creando una Llamada por Referencia.

- En una llamada por referencia se pasa una dirección. Luego se puede cambiar el valor del argumento.

- Se pueden pasar punteros a una función como si fueran valores.

Ejemplo `swap()`

```
#include <iostream.h>

void swap(int *x, int*y);

main()  {

    int a,b, *pa, *pb;
    pa=&a ; pb=&b;

    cout<< "ingrese a y b :"; cin>>a; cin>>b;
    swap(pa,pb);
    cout<<"ahora a y b son :" << a << " y "<<b;
}
```

```
void swap(int *x, int *y)  {
   int temp;

   temp=*x;
   *x=*y;
   *y=temp;
}
```

También pudo haberse usado como

```
int a,b;

    cout<< "ingrese a y b :"; cin>>a; cin>>b;
    swap(&a,&b);
```

5.3.2. Llamado de funciones con arreglos.

- Cuando se pasa un arreglo, se pasa sólo su dirección y no se copia el arreglo completo.
- Esto significa que la declaración de parámetros debe ser punteros.
- Existen tres formas de parámetros que reciben un puntero a arreglo:

1.- Se declara como un arreglo, indicando su tamaño.

```
#include <iostream.h>

void display(int num[10]);

main()  {

    int t[10];

    for(int i=0;i<10;i++) t[i]=i;
    display(t);
}
```

```
void display(int num[10]) {
    for(int i=0;i<10;i++) cout<<num[i]<<'\n';
}
```

2.- Se declara el parámetro como un arreglo sin indicar el tamaño.

```
#include <iostream.h>

void display(int num[],int n);

main()  {

    int t[10], n=10;

    for(int i=0;i<n;i++) t[i]=i;
    display(t,n);
}
```

```
void display(int num[],int n) {
    for(int i=0;i<n;i++) cout<<num[i]<<'\n';
}
```

3.- Se declara el parámetro como un puntero a un arreglo.

```
#include <iostream.h>

void display(int *n);

main()  {

    int t[10], *p;
    p=t;

    for(int i=0;i<10;i++) t[i]=i;
    display(p);
}
```

```
void display(int *n) {
    for(int i=0;i<10;i++) cout<<n[i]<<'\n';
}
```

Otro Ejemplo

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

void mayuscula(char *pstring);

main()  {
     char str[80], *p;
     p=str;

     cout <<" ingrese una palabra en minusculas ";
     gets(str);

         mayuscula(p);
}
```

```
void mayuscula(char *pstring)  {

 while(*pstring) cout << char(toupper(*pstring++));
}
```

Tarea. Construya una función `ingr_str()`, que funcione exactamente igual que `gets()`. Use sólo `cin`.

Argumentos del `main()`

Para ambientes diferentes a *Windows*, por ejemplo del tipo D.O.S o UNIX, es posible ingresar argumentos al **`main().`** Por ejemplo en el Turbo C++, se usan dos argumentos, el primero un entero que indica el número de argumentos a ingresar y el segundo en un puntero del tipo caracter a los argumentos.

5.3.3. Funciones que Retornan Valores.

De debe especificar, cuando una función retorna un tipo de valor, por ejemplo

```
#include <iostream.h>

float suma(float x, float y);

main()  {

    float a, b;
    cout<<"ingrese dos reales a,b :";cin>>a>>b;
    cout<< " la suma es : "<< suma(a,b);
}
```

```
float suma(float x, float y) {
   return x+y;
}
```

5.3.4. Retorno de Punteros

- Las funciones que retornan punteros son iguales a las demás funciones.

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

char *match(char c, char *s);

main()  {

    char letra, string[80], *p;

    cout<<"ingrese una frase "; gets(string);
    cout<<"\n ingrese una letra :"; cin>>letra;

    p=match(letra,string);

    if(*p) {
        cout<< "existe la letra a partir de ";
        cout<<p;
              }
       else cout<<" no se encuentra ";
}
```

```
char *match(char c, char *s) {

    int conta=0;
    while(c!=s[conta] && s[conta]) conta++;
    return &s[conta];
}
```

Sobre carga de funciones

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

void swap(int *x, int*y);
void swap(float *x, float *y);

main()  {
     char c;

     cout << "intercambia numerso enteros o reales (e/r) ";
     cin>>c;
     switch(c) {
        case 'e' :{
          int a,b, *pa, *pb;
          pa=&a ; pb=&b;
                cout<< "ingrese a y b :"; cin>>a; cin>>b;
          swap(pa,pb);
          cout<<"ahora a y b son :" << a << " y "<<b;
          break;
                }
        case 'r' : {
          float a, b, *pa, *pb;
          pa=&a ; pb=&b;
          cout<< "ingrese a y b :"; cin>>a; cin>>b;
          swap(pa,pb);
          cout<<"ahora a y b son :" << a << " y "<<b;
          break;
                }
     default   :
          cout <<"se equivoco ";
          break;
        }
}

void swap(int *x, int *y)  {
     int temp;

     temp=*x;
     *x=*y;
     *y=temp;
}

void swap(float *x, float *y)  {
     float temp;

     temp=*x;
     *x=*y;
     *y=temp;
}
```

Archivos de Encabezado o .h (headers)

Corresponden generalmente a un grupo de funciones similares agrupadas en una función y que se insertan a un programa cuando se necesitan.

Ejemplo

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

Tarea. Hacer funciones de encabezado.

Uso de Archivos.

En el archivo `stdio.h` existen varias funciones asociadas al manejo de archivos, como `fopen`, `fput`, etc.

Tarea : Hacer programas anteriores usando datos en archivo y de igual forma almacenando resultados en archivos.

6.Algoritmos de Ordenamiento

En general existen 3 tipos de algoritmos de ordenamiento:

- Por intercambio
- Por selección
- Por inserción

Interpretación informal. Supongamos una baraja de naipes:

Por Intercambio: Se colocan las cartas sobre la mesa y se intercambian hasta ordenarlas.

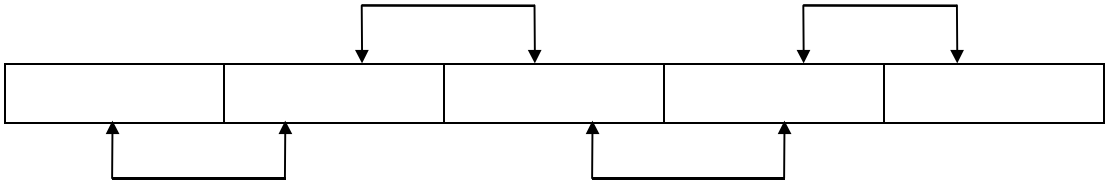
Por Selección: Se colocan las cartas sobre la mesa y se selecciona la menor (o mayor), si el orden es ascendente (o descendente) y de igual modo con las restantes.

Por Inserción: Con la baraja en la mano se saca la primera, la segunda se coloca bajo (o sobre), si el orden es ascendente (o descendente). La tercera y las restantes se van insertando de tal manera que queden ordenadas.

6.1. Ordenamiento por Intercambio

El algoritmo mas común es el de burbuja, se llama así porque en el primer paso el mayor (o menor) se va a un extremo, si el algoritmo es ascendente (o descendente).

Se compara el primer elemento con el segundo, si es menor (ascendente) se intercambian, a continuación se compara el segundo con el tercero y así sucesivamente.



Cada paso en c++ será

```
for(int i=0;i<strln(frase)-1;i++)
 if frase(frase[i]>frase[i+1]){
  char temp=frase[i];
  frase[i]=frase[i+1];
  frase[i+1]=temp ;
}
```

En el peor de los casos se deber repetir n-1 veces el paso anterior donde n = largo del arreglo. Es decir:

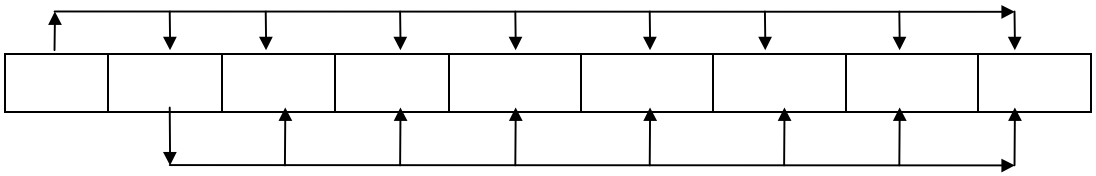
```
for(j=0;j<strln(frase)-1;j++)
 for(int i=0;i<strln(frase)-1;i++)
  if frase(frase[i]>frase[i+1]){
   char temp=frase[i];
   frase[i]=frase[i+1];
   frase[i+1]=temp ;
}
```

Sin embargo si los datos están previamente ordenados, no habrá necesidad de repetir el paso de ordenamiento.

```
int flag=1;
 while flag{
 flag=0;
 for(int i=0;i<strln(frase)-1;i++)
  if frase(frase[i]>frase[i+1]){
   char temp=frase[i];
   frase[i]=frase[i+1];
   frase[i+1]=temp;
   flag=1}
  }
```

6.2. Ordenamiento por Selección

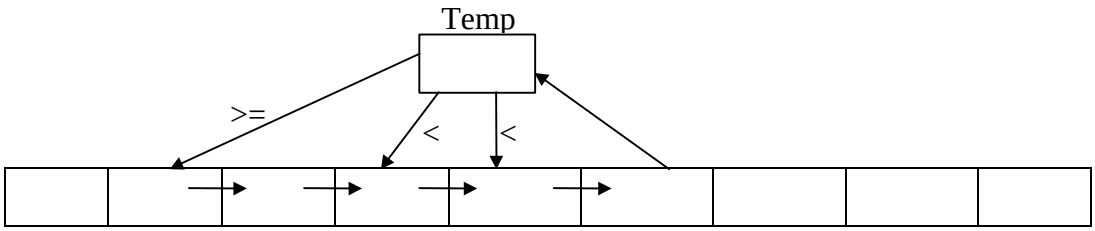
El algoritmo mas común elige el menor (o mayor) elemento si el orden es ascendente (o descendente) y lo coloca en la primera posición; repite el procedimiento con los restantes.



```
for(j=0;j<strln(frase)-1;j++)
 for(int i=j+1;i<strln(frase);i++)
  if frase(frase[j]>frase[i]){
   char temp=frase[i];
   frase[i]=frase[j];
   frase[j]=temp ;
 }
```

6.3. Ordenamiento por Inserción

En términos generales en un arreglo, los datos estarán ordenados hasta el (k-1)-ésimo elemento. De esta forma el k-ésimo elemento se va a comparar con el anterior ((k-1)-ésimo), si éste es mayor pasa a ocupar el k-ésimo lugar, luego se compara con el anterior, nuevamente el mayor se desplaza y así sucesivamente hasta que se encuentra un elemento menor o igual y allí se instala.




```
char temp=frase[i] ;
int i=k-1 ;
while (temp<frase[i]) {
        frase[i+1]=frase[i] ;
        i-- ;
}
frase[i+1]=temp ;
```

7.Estructuras

Colección de variables bajo un mismo nombre.

Ejemplo

```
struct alumno {
    char nombre[30];
    char direccion[50];
    char ciudad[20];
    int cod_area;
    long int fono;
};
```

Termina con (;) porque la definición es una sentencia.

La declaración de una variable se hace de la siguiente forma

```
struct alumno informatico;
```

Otra forma de declarar es;

```
struct alumno {
    char nombre[30];
    char direccion[50];
    char ciudad[20];
    int cod_area;
    unsigned long int fono;
} informatico, mecanico, electrico;
```

Si no sólo se va a declarar una variable, se puede omitir el nombre de la estructura,

Ejemplo

```
struct {
    char nombre[30];
    char direccion[50];
    char ciudad[20];
    int cod_area;
    long int fono;
} informatico;
```

El espacio en memoria que ocupará la estructura completa será:

nombre	30	bytes
direccion	50	bytes
ciudad	20	bytes
cod_area	2	bytes
fono	4	bytes
TOTAL	106	bytes

7.1. Referenciando Elementos de una estructura.

Se usa el operador *punto* (.).

Ejemplos

```
gets(informatico.nombre);
```

```
gets(ciudad);
if(!strcmp(ciudad,"Valdivia"))
       informatico.cod_area=63;

cout<<informatico.fono;
```

Se pueden accesar los contenidos de los elementos.

Ejemplo

```
for(int i=0; informatico.nombre[i]; i++)
       cout << informatico.nombre[i];
```

ó

```
p=informatico.nombre;
while(*p) {
    cout << *p;
    *p++
}
```

7.2. Arreglo de Estructuras

Uno de los usos más comunes de estructuras son los arreglos de estructuras.

Ejemplo

```
struct alumno informatico[100];
```

Y al igual que un arreglo se puede indicar elemento en particular.

Ejemplo

cout << informatico[2].fono;

Ejemplo de una lista de direcciones

```
/* Ejemplo de un lista de direcciones */
#include<iostream.h>
#include<string.h>
#include<stdio.h>
#include<ctype.h>
#include<process.h>
#include<stdlib.h>

#define TAMANO 100

struct alumno {
    char nombre[30];
    char direccion[50];
    char ciudad[20];
    int cod_area;
    unsigned long int fono;
} informatico[TAMANO];

void ini_lista(),ingresar();
void mostrar(), guardar(), cargar();
char menu();
```

```
main() {
    char elige;
    ini_lista();
    for(;;){
       elige=menu();
       switch(elige){
          case 'i':ingresar();
        break;
          case 'm':mostrar();
        break;
          case 'g':guardar();
        break;
          case 'c':cargar();
        break;
          case 's':exit(1);
       }
    }
}
/* inicializa arreglo */
void ini_lista() {
    for(int i=0; i<TAMANO ; i++)
       *informatico[i].nombre='\0';
}
/* menu de selección */
char menu()  {
    char s[80];
    do {
       cout<<" (I)Ingresar \n";
       cout<<" (M)Mostrar  \n";
       cout<<" (C)Cargar   \n";
       cout<<" (G)Guardar  \n";
       cout<<" (S)Salir    \n";
      gets(s);
    }while(!strrchr("imcgs",tolower(*s)));
    return char(tolower(*s));
}
```

```
/* ingreso de nombres a la lista */
void ingresar()  {
    int i;
    for(i=0; i<TAMANO; i++)
      if(!*informatico[i].nombre) break;
      if(i==TAMANO) {
        cout<<"La lista esta llena \n";
        return;
      }
      cout << " nombre : ";
      gets(informatico[i].nombre);

      cout << " direccion : ";
      gets(informatico[i].direccion);

      cout << " ciudad : ";
      gets(informatico[i].ciudad);

      cout << " código de área : ";
      cin >>informatico[i].cod_area;

      cout << " teléfono : ";
      cin >>informatico[i].fono;
}
```

```
/* Mostrar la lista */
void mostrar() {
  for(int i=0; i<TAMANO ; i++) {
    if(*informatico[i].nombre) {
      cout<<'\n'<<informatico[i].nombre;
      cout<<'\n'<<informatico[i].direccion;
      cout<<'\n'<<informatico[i].ciudad;
      cout<<'\n'<<informatico[i].cod_area;
      cout<<"-"<<informatico[i].fono;
      cout<<'\n';
      }
    }
}

/* Guardar la lista */
void guardar() {
    FILE *fp;
    if((fp=fopen("a:datos.txt","a"))==NULL){
      cout<<"no se puede abrir el archivo \n";
      return;
    }

    for(int i=0; i<TAMANO; i++)
      if(*informatico[i].nombre)
        if(fwrite(&informatico[i],sizeof(struct alumno),1,fp)!=1)
```

```
          cout<<" Error de escritura en el archivo \n";
    fclose(fp);
}

/*  Cargar el archivo  */
void cargar() {
    FILE *fp;

    if((fp=fopen("a:datos.txt","r"))==NULL){
       cout << " No se puede abrir el archivo \n";
       return;
    }

    ini_lista();
    for(int i=0; i<TAMANO; i++)
       if(fread(&informatico[i],sizeof(struct alumno),1,fp)!=1){
         if(feof(fp)) {
        fclose(fp);
        return;
         };
         cout<<"Error de lectura en el archivo \n";
           }
}
```

_fstrrchr, strrchr <STRING.H>

Finds the last occurrence of c in s

Declaration

char *strrchr(const char *s, int c);
char far * far _fstrrchr(const char far *s, int c);

Remarks

- **strrchr** scans a string in the reverse direction, looking for a specific character.
- **strrchr** finds the last occurrence of the character c in the string s.
- The null-terminator is considered to be part of the string.

Return Value

On success, strrchr returns a pointer to the last occurrence of the character c.

If c does not occur in s, strrchr returns null.

Portability

Routine	DOS	UNIX	Windows	ANSI C	C++ only
Near version	yes	yes	yes	yes	
Far version	yes		yes		

fdopen, fopen, freopen, _fsopen <STDIO.H>

- **fdopen** associates a stream with a file handle
- **fopen** opens a stream
- **freopen** associates a new file with an open stream
- **_fsopen** opens a stream with file sharing

Declaration
- FILE *fopen(const char *filename, const char *mode);

Remarks
- **fopen** and **_fsopen** open a file and associate a stream with it. Both functions return a pointer that identifies the stream in subsequent operations.

When a file is opened for update, both input and output can be done on the resulting stream. However:
- output can't be directly followed by input without an intervening **fseek** or **rewind**
- input can't be directly followed by output without an intervening **fseek** or **rewind**, or an input that encounters end-of-file

Portability

Routine	DOS	UNIX	Windows	ANSI C	C++ only
fdopen	yes	yes	yes		
fopen	yes		yes	yes	
freopen	yes	yes	yes	yes	
_fsopen	yes				

mode
The mode string used in calls to fopen,, freopen,, and _fsopen (or the type string used in calls to fdopen) is one of the following values:

String	Description

r Open for reading only
w Create for writing. If a file by that name already exists, it will be overwritten.
a Append; open for writing at end of file, or create for writing if the file does not exist.
r+ Open an existing file for update (reading and writing)
w+ Create a new file for update (reading and writing). If a file by that name already exists, it will be overwritten.
a+ Open for append; open for update at the end of the file, or create if the file does not exist.

- To specify that a given file is being opened or created in text mode, append **t** to the string (**rt**, **w+t**, etc.).
- To specify binary mode, append **b** to the string (**wb**, **a+b**, etc.).
- fopen and _fsopen also allow the **t** or **b** to be inserted between the letter and the + character in the string. For example, **rt+** is equivalent to **r+t**.

fwrite <STDIO.H>

Writes to a stream

Declaration
size_t fwrite(const void *ptr, size_t size, size_t n, FILE*stream);

Remarks
fwrite appends a specified number of equal-sized data items to an output file.

Argument **What It Is/Does**
ptr Pointer to any object; the data written begins at ptr

size	Length of each item of data
n	Number of data items to be appended
stream	Specifies output file

The total number of bytes written is (n * size)

Return Value

- On success, returns the number of items (not bytes) actually written.
- On error, returns a short count.

Portability

DOS	UNIX	Windows	ANSI C	C++ only
yes	yes	yes	yes	

feof <STDIO.H>

Macro that tests if end-of-file has been reached on a stream.

Declaration

int feof(FILE *stream);

Remarks

- feof is a macro that tests the given stream for an end-of-file indicator.
- Once the indicator is set, read operations on the file return the indicator until rewind is called, or the file is closed.
- The end-of-file indicator is reset with each input operation.

Return Value
- Returns non-zero if an end-of-file indicator was detected on the last input operation on the named stream.
- Returns 0 if end-of-file has not been reached.

Portability

DOS	UNIX	Windows	ANSI C	C++ only
yes	yes	yes	yes	

fclose <STDIO.H>

Closes a stream

Declaration
int fclose(FILE *stream);

Remarks
- fclose closes the named stream.
- All buffers associated with the stream are flushed before closing.
- System-allocated buffers are freed upon closing.
- Buffers assigned with setbuf or setvbuf are not automatically freed. (But if setvbuf is passed null for the buffer pointer, it will free it upon close.)

Return Value
- On success, returns 0
- On error, returns EOF

Portability

DOS	UNIX	Windows	ANSI C	C++ only
yes	yes	yes	yes	

7.3. Paso de Estructuras a Funciones

7.3.1. Paso de Elementos

Supongamos

```
struct alumno{
   char nombre[30];
   char direccion[50];
   char ciudad[20];
   int cod_area;
   long int fono;
} informatico;
```

Entonces

```
funcion1(informatico.cond_area);
funcion2(informatico.fono);
funcion3(&informatico.nombre);
funcion4(&informatico.direccion[2];
```

Para el caso de arreglos.

```
funcion1(informatico[5].cod_area);
funcion4(&informatico[4].direccion[2];
```

7.3.2. Paso de Estructuras Enteras a funciones

Ejemplo

```
funcion(informatico);
```

La definición será

```
void funcion(alumno cualquiera) {
...
}
```

Punteros a Estructuras.

Declaración de una puntero a estructura

```
alumno *puntero
```

Existen dos usos para punteros a estructuras

1. Para llamado por referencia a una función
2. Para crear listas enlazadas y otras estructuras dinámicas.

Se verá el primer caso.

Ejemplo,

```
struct bal {
   float balance;
   char nombre[30];
} persona;

bal *p;
```

Así,

```
p=&persona;
```

Luego para referenciar algún elemento

```
(*p).balance
```

Se usa el paréntesis porque el punto tiene mayor prioridad que el *.

lo anterior es equivalente a

```
p->balance;
```

Ejemplo

```
#include<iostream.h>

struct tm {
   int horas;
   int minutos;
   int segundos;
};

void actualizar(tm *t);
void mostrar(tm *t);
void retardo();

main() {
    tm time;
    time.horas=0;
```

```
    time.minutos=0;
    time.segundos=0;


 for(;;) {
   actualizar(&time);
   mostrar(&time);
 }
}

void actualizar(tm *p)  {

   p->segundos++;
   if((p->segundos)==60) {
      p->segundos=0;
      p->minutos++;
   };

   if((p->minutos)==60) {
      p->minutos=0;
      p->horas++;
   };

   if((p->horas==24)) p->horas=0;
   retardo();
}

void mostrar(struct tm *t)  {
   cout<<"horas :"<<t->horas;
   cout<<" minutos :"<<t->minutos;
   cout<<" segundos :"<<t->segundos;
   cout<<'\n';
}
```

```
void retardo() {
   for(long int i=0;i<5000000;i++);
}
```

Bit fields

A *bit field* is an element of a structure that is defined in terms of bits. Using a special type of struct definition, you can declare a structure element that can range from 1 to 16 bits in length.

For example, this struct

```
struct  bit_field {
  int bit_1        : 1;
  int bits_2_to_5  : 4;
  int bit_6        : 1;
  int bits_7_to_16 : 10;
} bit_var;
```

corresponds to this collection of bit fields:

16 15 14 13 12 11 10 9 8 7	6	5 4 3 2	1

8.Algunas Estructuras de Datos

Un programa consiste de:

- Algoritmos y,
- Estructuras de datos

- Un buen programa es una mescla de ambos.
- Los lenguajes entregan datos simples como `char`, `int`, `long`, `float`, etc.
- Los usuarios pueden organizar los datos en arreglos, estructuras u otros.
- El nivel final de abstracción trasciende el aspecto físico de los datos y representa mas bién la forma en que estos van a ser accesados.
- De acuerdo a esto los datos físicos son accesados por una “máquina de datos” que controla la forma en que los datos serán accesados.
- Los tipos de máquinas de datos son:

    1. Colas
    2. Stacks
    3. Listas enlazadas y
    4. Árboles binarios

- Cada método provee una solución a una clase de problemas.

- Todos los métodos tienen funciones como `guardar` y `recuperar` un ítem.

8.1. Colas

Un cola es una lista lineal de acceso FIFO (*first input first output*) .
Supongamos las funciones `guardar()` y `recuperar()`.
Un ejemplo se ve en la siguiente tabla.

Acción	Contenido de la cola
`guardar(A)`	`A`
`guardar(B)`	`A B`
`guardar(C)`	`A B C`
`recuperar()` Retorna A	`B C`
`guardar(D)`	`B C D`
`recuperar()` Retorna B	`C D`
`recuperar()` Retorna C	`D`

La función `guardar(A)` en C++ podría ser

```
void guardar(char *dato) {
   if(gpos==MAX) {
      cout<<“La cola está llena \n”;
      return;
   }
   p[gpos]=dato;
   gpos++;
}
```

y la función `recuperar()`,

```
char *recuperar() {
    if(rpos==gpos) {
       cout<<“La cola está vacía \n”;
       return NULL;
    }
    rpos++;
    return p[rpos-1];
}
```

El programa completo sería :

```
#include<iostream.h>
#include<stdio.h>
#include<string.h>
#include<ctype.h>
#include<process.h>
#include<stdlib.h>
#define MAX 100

char *p[MAX], *recupera();
int gpos, rpos;
void ingresar(), guardar(char *dato), listar(),
borrar();

main()  {
    char s[80];
    for(int i=0;i<MAX;i++) p[i]=NULL;
    gpos=rpos=0;

    for(;;) {
    cout<<"\n Ingresar, Listar, Borrar o Salir : ";
    gets(s);
    *s=toupper(*s);
       switch(*s) {
         case 'I':
            ingresar();
            break;
```

```
          case 'L':
             listar();
             break;
          case 'B':
             borrar();
             break;
          case 'S':
             exit(1);
       }
    }
}

void ingresar() {
  char s[256];
  do {
    cout<<"ingrese el dato "<<gpos+1<<" : " ;
    gets(s);
    if(*s==NULL) break;
    p[gpos]=(char *)malloc(strlen(s));
    strcpy(p[gpos],s);
    if(*s) guardar(p[gpos]);
  }while(*s);
}

void listar() {
  for(int i=rpos;i<gpos;i++)
    cout<<i+1<<".- "<<p[i]<<'\n';
}

void borrar()  {
  char *p;
  if(!(p=recupera())) return;
  cout<<"borrado "<<p<<'\n';
}

void guardar(char *dato) {
    if(gpos==MAX) {
        cout<<"La cola está llena \n";
        return;
    }
    p[gpos]=dato;
    gpos++;
}
```

```
char *recupera() {
    if(rpos==gpos) {
        cout<<"La cola está vacía \n";
        return NULL;
    }
    rpos++;
    return p[rpos-1];
}
```

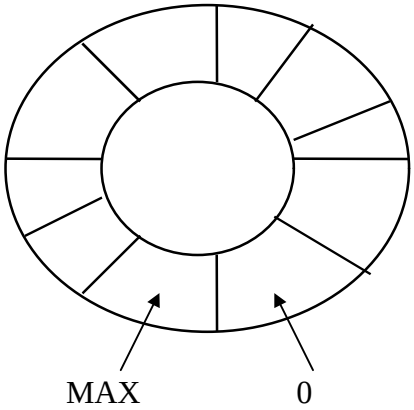
8.2. Colas Circulares

- En el caso de colas simples, cuando `gpos` llega a MAX, no se puede seguir ingresando datos.
- Una mejor es que `gpos` y `rpos` puedan volver al principio, esto se conoce como colas circulares.
- En una cola circular las funciones guardar y recuperar deben ser cambiadas.

```
void guardar(char dato) {
    if(gpos+1==rpos || (gpos+1==MAX  && !rpos)) {
        cout<<"La cola está llena \n";
        return;
    }
    p[gpos]=dato;
    gpos++;
    if(gpos==MAX) gpos=0;
}

char recuperar() {
    if(rpos==MAX) rpos=0;
    if(rpos==gpos) {
        cout<<"La cola está vacía \n";
        return NULL;
    }
    rpos++;
    return p[rpos-1];
```

```
}
```

- Osea, la cola esta llena cuando los índices `gpos` y `rpos` son iguales, de esta manera no pueden comenzar ambos en cero.



Un ejemplo es:

```
#include<iostream.h>
#include<stdio.h>
#include<conio.h>
#define MAX 80

char buf[MAX+1];
int gpos=0;
int rpos=MAX;

void guardar(char dato);
char recuperar();

int main()  {
   char ch;
   buf[80]=NULL;

   while (ch!=';') {
    if(kbhit()) {
      ch=getch();
      guardar(ch);
    }
   }
```

```
   while((ch=recuperar())!=NULL) cout<<ch;
}

void guardar(char dato) {
    if(gpos+1==rpos || (gpos+1==MAX  && !rpos)) {
        cout<<"\n La cola está llena";
        return;
    }
    buf[gpos]=dato;
    gpos++;
    if(gpos==MAX) gpos=0;
}

char recuperar() {
    if(rpos==MAX) rpos=0;
    if(rpos==gpos) {
        cout<<"\n La cola está vacía";
        return NULL;
    }
    rpos++;
    return buf[rpos-1];
}
```

8.3. *Stacks* o Pilas.

Los *stacks* son los opuestos a las colas. Usan acceso LIFO (*Last input, First output*). Piense en una pila de platos sucios, el último que se coloca en la pila es el primero que se lava.

Por razones históricas a la función `guardar()` se le llama `push()` y a la función recuperar() se le llama `pop()`.

Tabla : Uso de un *stack*.

Acción		Contenido del *Stack*
`push(A)`		`A`
`push(B)`		`B A`
`push(C)`		`C B A`
`pop()`	Saca C	`B A`
`push(F)`		`F B A`
`pop()`	Saca F	`B A`
`pop()`	Saca B	`A`
`pop()`	Saca A	*`stack`* `vacío`

Las funciones `pop()` y `push()` serían:

```
int stack[MAX];
int tos=0;       /* tope del stack */

void push( int i) {
  if(tos>=MAX)  {
    cout<<“La pila está llena \n”;
    return;
  }
stack[tos]=i;
tos++;
}
```

```
int pop() {
   if(tos==0) {
      cout<<“stack vacío \n”;
      return;
   }
   return stack[--tos];
}
```

Un buen ejemplo de stack es una calculadora con las cuatro operaciones básica:

Existen dos tipos de calculadores

1. Con notación *infix*. Por ejemplo si se suman 100 y 200, se ingresa 100, +, 200 y =.
2. Con notación *postfix*. Por ejemplo si se sumas 100 y 200, se ingresa 100, 200 y +.

La notación postfix la usan con frecuencia las calculadoras Hewlet Packard y operan de la siguiente manera:

- Ingresan los datos a un *stack* y al presionar un operador son sacados, el resultado es colocado en el *stack*.

- Para dar solución a la calculadora, se deberán modificar las funciones `pop()` y `push()`.

Ejemplo completo.

```
#include<iostream.h>
#include<stdio.h>
#include<stdlib.h>
#define MAX 100

int *p;   /* puntero a región de memoria libre */
int *tos; /* apunta al tope del stack */
int *fos; /* apunta al fondo del stack */

void push(int i);
int pop();

main() {
  int a,b;
  char s[80];

  tos=p;
  fos=p+MAX-1;

  cout<<"Calculadora de cuatro funciones \n";

  do {
     cout<<": ";
     gets(s);
     switch(*s) {
       case '+':
         a=pop();
         b=pop();
          cout<<" = "<<a+b<<'\n';
         push(a+b);
         break;
```

```
        case '-':
          a=pop();
          b=pop();
           cout<<" = "<<b-a<<'\n';
          push(b-a);
          break;
        case '*':
          a=pop();
          b=pop();
           cout<<" = "<<a*b<<'\n';
          push(a*b);
          break;
        case '/':
          a=pop();
          b=pop();
           if(a==0) {
             cout<<"división por cero \n";
             break;
            }
          cout<<" = "<<b/a<<'\n';
           push(b/a);
           break;
        case '.': /* muestra el tope del stack */
            a=pop();
            push(a);
            cout<<"El valor del tope es :"<<a<<'\n';
     break;
        default:
            push(atoi(s));
      }
  }while(*s!='q');
}

void push(int i) {
```

```
  if(p>fos)  {
    cout<<"La pila está llena \n";
    return;
  }
  *p=i;
  p++;
}

int pop() {
   p--;
   if(p<tos) {
      cout<<"stack vacío \n";
      return 0;
   }
   return *p;
}
```

8.4. Listas Enlazadas

Las colas y pilas tienen:

- reglas estrictas para hacer referencias a sus datos.
- Para recuperar un dato se deben “consumir” todos los anteriores.
- Ocupan regiones contíguas de memoria

- Una lista enlazada puede acceder a datos ubicados en cualquier parte de la lista sin tener que remover los datos anteriores.
- Cada ítem de la lista tiene un enlace a la siguiente.
- A diferencia de las colas y pilas, una lista requiere de una estructura de datos más compleja.

Las lista enlazadas se crean para dos propósitos específicos que son:

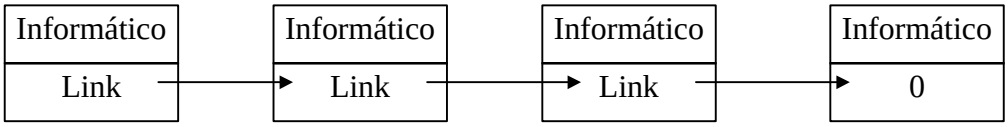
1. Crear un arreglo de tamaño desconocido.
2. Almacenamiento de bases de datos en disco.

Las listas enlazadas pueden ser:

1. Simplemente enlazada. Cada elemento contiene un enlace al siguiente.
2. Doblemente enlazadas. Cada elemento contiene un enlaces al siguiente y otro al anterior.

Listas simplemente enlazadas

- Requieren que cada item de información contenga un *link* al próximo elemento.
- Cada item consiste de una estructura que contiene campos y un puntero de enlace.



Hay dos formas de construir una lista enlazada que son:

1.Poniendo todo nuevo item al final de la lista.
2.Colocar un nuevo item en un lugar especificado.

Antes de comenzar debemos tener definida una estructura, Por ejemplo,

```
struct alumno {
    char nombre[30];
    char direccion[50];
    char ciudad[20];
    int cod_area;
    unsigned long int fono;
    alumno *proximo;
} informatico;
```

La función `guardar_ls()` coloca un nuevo elemento al final

```
void guardar_ls(alumno *i) {
    static alumno *ultimo=NULL;
    if(!ultimo) ultimo=i;
    else ultimo->proximo=i;
    i->proximo=NULL;
    ultimo=i;
}
```

Bibliografía

- Sethi, Ravi. : “Lenguajes de Programación. Conceptos y Constructores”. Eddison Wesley Iberoamericana, S.S. 1992.
- B.W.Kernighan & D.M.Richie : “ El lenguaje de Programación C”, Prentice –Hall Hispanoamericana S.A.
- Stroustrup, Bjarne. : “ C++ el lenguaje de Programación”, Segunda edición. Eddison Wesley/Díaz de Santos. 1993.
- Pratt, T. : “Diseño e implementación de lenguajes de Programación”. Prentice Hall. Segunda edición. 1987.
- Herbert Schildt. : “Using Turbo C”. Borland – Osborne/McGraw – Hill, 1988.
- Herbert Schildt. : “Advanced Turbo C”. Borland – Osborne/McGraw – Hill, 1988.