Manual de Programación en Lenguaje C++

Proyecto de Investigación: Métodos de Funciones de Base Radial para la Solución de EDP.

Servicio Social - DGSCA-UNAM, 2008.

Autor: Linda I. Olivares Flores

Índice general

Capítulo 1

Introducción al lenguaje de programación C++

1.1. Lenguaje C++

C++ es un lenguaje de programación, creado a mediados de 1980 por Bjarne Strousstrup, como extensión del lenguaje C. Este lenguaje abarca tres paradigmas de la programación:

1. Programación Estructurada
2. Programación Genérica
3. Programación Orientada a Objetos

En la actualidad, C++ es un lenguaje versátil, potente y general. Su éxito entre los programadores le ha llevado a ocupar el primer puesto como herramienta de desarrollo de aplicaciones, ya sea en Windows o GNU Linux, que es el sistema operativo en el cual basaremos este tutorial.

1.2. C++ en un entorno Linux

Comenzaremos diciendo que los programas se pueden escribir en cualquier editor de textos de GNU, entre ellos se encuentran *emacs, vim, kate, gedit, nan*, guardando dichos archivos con extensión *.cpp*, los cuales serán compilados en GNU/linux utilizando el compilador GNU de C++, llamado *gcc* que puede compilar C, C++, y que además se apega al estandar ANSI, permitiendo la portabilidad de estos códigos. Dicho compilador se invoca con el comando *gcc*.

Para compilar ponemos la siguiente línea en una terminal previamente ubicada en el directorio que contiene nuestro archivo:

g++ programa.cpp -o programa.out

-o indica el nombre del archivo de salida el cual será el ejecutable de nuestro proyecto.

Luego para ejecutar, escribimos sobre la línea de comandos: **./programa.out** y entonces podremos ejecutar nuestro programa.

Cuando creamos pequeños programas la compilación de éstos es muy fácil, pero cuando se trabaja con proyectos grandes, con varios archivos fuente la compilación resulta más dificil, por lo que Linux proporciona la utilería *make* de GNU, el cual busca un archivo make donde encontrará toda la información que necesita para crear el ejecutable, si encuentra el archivo busca la palabra makefile o Makefile, que son nombres predeterminados.

Los archivos make contienen información acerca de la compilación y enlace del programa, con una sintaxis muy especifica. Un makefile se define como una lista de normas y dependencias con sus correspondientes comandos para crear objetivos, utilizando dichas normas y dependencias.

Un archivo Makefile es un archivo de texto en el cual se distinguen cuatro tipos básicos de declaraciones:

1. **Comentarios**: Al igual que en los programas, contribuyen a un mejor entendimiento de las reglas definidas en el archivo. Los comentarios se inician con el caracter #, y se ignora todo lo que continúe después de ella, hasta el final de línea.

2. **Variables**: Se definen utilizando el siguiente formato:

   nombre = dato

   De esta forma, se simplifica el uso de los archivos Makefile. Para obtener el valor se emplea la variable encerrada entre paréntesis y con el caracter $ al inicio, en este caso todas las instancias de $(nombre) serán reemplazadas por datos. Por ejemplo, la siguiente definición

   SRC = main.c

   origina la siguiente línea:

   gcc $ SRC

   y será interpretada como:

   gcc main.c

   Sin embargo, pueden contener más de un elemento. Por ejemplo:

   objects = programa_1.o programa_2.o programa_3.o \ programa_4.o programa_5.o

programa: $(objects) gcc -o programa $(objects)

Hay que notar que make hace distinción entre mayúsculas y minúsculas.

3. **Reglas Explícitas**: Estas le indican a make qué archivos dependen de otros, así como los comandos requeridos para compilar un archivo en particular. Su formato es:

   archivoDestino: archivosOrigen

   Esta regla indica que, para crear archivoDestino, make debe ejecutar comandos sobre los archivos archivosOrigen. Por ejemplo:

   main: main.c funciones.h
   gcc -o main main.c funciones.h

   esto significa que, para crear el archivo de destino *main*, deben existir los archivos *main.c* y *funciones.h* y además debe ejecutar el comando:

   gcc -o main main.c funciones.h

4. **Reglas Implícitas**: Son similares a las reglas explícitas, pero no indican los comandos a ejecutar, sino que make utiliza los sufijos (extensiones de los archivos) para determinar que comandos ejecutar. Por ejemplo:

   funciones.o: funciones.c funciones.h

   origina la siguiente línea:

   $(CC) $(CFLAGS) c funciones.c funciones.h

   Existe un conjunto de variables que se emplean para las reglas implícitas, y existen dos categorías: aquellas que son nombres de programas (como CC) y aquellas que tienen los argumentos para los programas (como CFLAGS). Estas variables son provistas y contienen valores predeterminados, sin embargo, pueden ser modificadas, como se muestra a continuación:

   CC = gcc CFLAGS = -Wall -O2

   En el primer caso, se ha indicado que el compilador que se empleará es gcc y sus parámetros son -Wall -O2.

   Este código debe estar en un archivo llamado Makefile o makefile o tendriamos que usar la opción -f que nos permite ubicarnos en el directorio raíz.

Ejemplo de Makefile.

```
CPPFLAGS =
CPPLIBS =
main: Conjunto.o main.o
g++ -o main main.o Conjunto.o $(CPPLIGS)

Conjunto.o: Conjunto.cpp  Conjunto.h
g++ $(CPPFLAGS) -c -o Conjunto.o Conjunto.cpp

main.o: Conjunto.h
g++ $(CPPFLAGS) -c -o main.o main.cpp

clean:
rm *.o main
```

Para **ejecutar** este archivo, escribimos sobre la linea de comandos: **make**, esta opción nos creará el archivo de salida, que será el ejecutable.

Es aquí donde viene la verdadera prueba de fuego del programador: cuando lanza la orden de compilar y enlazar su programa.

Todos los módulos involucrados en los pasos anteriores, compilador, analizador sintáctico y enlazador pueden detectar errores en nuestro código y mostrar los mensajes correspondientes.

Estos **errores** son considerados solo de tres tipos:

1. De **tiempo de compilación**. Se engloban aquí los errores detectados por preprocesador, el analizador sintáctico y el propio compilador. Los hay meramente sintácticos.

2. De **tiempo de enlazado**. Son detectados por el enlazador. Por ejemplo una llamada a función cuya definición no aparece por ninguna parte.

3. De **tiempo de (runtime)**. Existe finalmente una última clase de errores: los que se producen cuando se ejecuta el programa; son precisamente los más difíciles de diagnosticar y verificar, sobre todo en aplicaciones grandes.

1.3. Estructura de un programa en C++

Para darnos una idea chequemos el siguiente programa

```
//Mi primer programa en C++
# include <iostream>
using namespace std;
int main (){
count << "hello World";
return 0;
}
```

El programa anterior es típico de los programadores aprendices, el resultado de su impresión en pantalla es la frase:

"hello World"

Es uno de los más simples programas que pueden estar escritos en C + +, pero ya contiene los componentes fundamentales que todos los programas escritos en C + +. Vamos a ver línea por línea en el código lo que hemos escrito:

```
//Mi primer programa en C++
```

Se trata de una línea de comentario. Todas las líneas que comiencen con dos signos barra se consideran comentarios y no tienen ningún efecto sobre el comportamiento del programa. El programador puede usar para incluir breves explicaciones o alegaciones dentro del código fuente en sí. En este caso, la línea es una breve descripción de lo que es nuestro programa.

```
# include <iostream>
```

Las líneas que comienza con un símbolo de sostenido # son directivas para el preprocesador. En este caso, la directiva # include <iostream>le dice al preprocesador que incluya el iostream estándar de archivo. Este archivo específico (iostream) incluye las declaraciones de la norma básica de entrada y salida de la biblioteca de C++.

```
using namespace std;
```

Todos los elementos del modelo de librería de C++ se declaran dentro de lo que se denomina un espacio de nombres. Por lo tanto, para poder acceder a su funcionalidad declaramos

con esta expresión que vamos a utilizar estas entidades. Esta línea es muy frecuente en los programas que utilizan la biblioteca estándar, y de hecho será incluido en la mayoría de los códigos.

```
int main ()
```

Esta línea se corresponde con el comienzo de la definición de la función principal. La función principal es el punto por donde todos los programas inician su ejecución, independientemente de su ubicación dentro del código fuente. No importa si hay otras funciones con otros nombres definidos antes o después de las instrucciones que figuran dentro de esta función, ya que por definición será siempre la primera en ser ejecutada. Por esa misma razón, es esencial que todos los programas tengan una función principal. Lo que se contiene dentro de las llaves que delimitan la función es lo que hace cuando se ejecuta.

```
count <<"hello World";
```

Esta línea es una declaración de C++, en términos simples es una expresión que produce algún efecto. De hecho, esta declaración lleva a cabo la única acción que genera un efecto visible en nuestro programa. Representa el flujo de salida y su objetivo es insertar una secuencia de caracteres ("hello World") en el flujo de salida estándar (pantalla)

```
 return 0;
```

Esta declaración hace que la función pricipal termine. Un código de retorno es 0, cuando la función principal interpreta de manera general que el programa trabajó como se esperaba, sin ningún error durante su ejecución. Esta es la forma más habitual para poner fin a un programa C++ en la consola.

1.4. Variables y Tipos de datos

Una **variable** es un espacio de memoría reservado en el ordenador para contener valores que pueden cambiar durante la ejecución de un programa. Los **tipos** que se le asignen a estas determinan cómo se manipulará la información contenida en ellas.

Cada variable necesita un identificador que la distingue de las demás. Un **identificador** válido es una secuencia de una o más letras, dígitos o guiones bajos, recordando que no deben coincidir con palabras reservadas del lenguaje, deben comenzar por una letra y además tomar en cuenta que C++ hace diferencia entre mayúsculas y minúsculas.

Las variables que se pueden presentar en un programa son de los siguientes tipos:

1. **Variables Locales** . Se definen solo en bloque en el que se vayan a ocupar, de esta manera evitamos tener variables definidas que luego no se utilizan.
2. **Variables Globales**. No son lo más recomendable, pues su existencia atenta contra la comprensión del código y su encapsulamiento.
3. **Variables estáticas**. Se tienen que inicializar en el momento en que se declaran, de manera obligatoria.

Ahora hablemos de los **tipos de datos** que reconoce C++. Estos definen el modo en que se usa el espacio (memoria) en los programas. Al especificar un tipo de datos, estamos indicando al compilador como crear un espacio de almacenamiento en particular, y también como manipular este espacio. Un tipo de dato define el posible rango de valores que una variable puede tomar al momento de ejecución del programa y a lo largo de toda la vida útil del propio programa.

Los tipos de datos pueden ser **predefinidos o abstractos**. Un tipo de dato *predefinido* es intrínsecamente comprendido por el compilador. En contraste, un tipo de datos definido por el usuario es aquel que usted o cualquier otro programador crea como una clase, que comúnmente son llamados tipos de datos *abstractos*.

Los tipos de datos más comunes en C++ son:

TipodeDato	*EspacioenMemoria*	*Rango*
unsigned char	8 bits	0 a 255
char	8 bits	-128 a 127
short int	16 bits	-32,768 a 32,767
unsigned int	32 bits	0 a 4,294,967,295
int	32 bits	-2,147,483,648 a 2,147,483,647
unsigned long	32 bits	0 a 4,294,967,295
enum	16 bits	-2,147,483,648 a 2,147,483,647
long	32 bits	-2,147,483,648 a 2,147,483,647
float	32 bits	3.4 x 10-38 a 3.4 x 10+38(6 dec)
double	64 bits	1.7 x 10-308 a 1.7*10+308(15 dec
long double	80 bits	3.4 x 10-4932 a 1.1 x 10+4932
void	sin valor	

1.4.1. Conversión de Tipos

Cuando nuestro programa contiene operaciones binarias con operandos de distintos tipos, estos se convierten a un tipo en común, en general para conversiones explícitas C++ hace uso del **casting**, lo que nos permite tener más presición en aquellos casos en los que el resultado de la operación no es un int y la variuable receptora sí lo es.

Algunas reglas que controlan estas conversiones son las siguientes:

1. Cualquier tipo entero pequeño como char o short será convertido a int o unsigned int.
2. Si algún operando es de tipo long double, el otro se convertirá a long double.
3. Si algún operando es de tipo double, el otro se convertirá a double.
4. Si algún operando es de tipo float, el otro se convertirá a float.
5. Si algún operando es de tipo unsigned long long, el otro se convertirá a unsigned long long.
6. Si algún operando es de tipo long long, el otro se convertirá a long long.
7. Si algún operando es de tipo unsigned long, el otro se convertirá a unsigned long.
8. Si algún operando es de tipo long, el otro se convertirá a long.
9. Si algún operando es de tipo unsigned int, el otro se convertirá a unsigned int.

1.5. Espacios de nombres en C++

Los **espacios de nombre** nos ayudan a evitar problemas con identificadores, nos permiten, que existan variables o funciones con el mismo nombre, declaradas en distintos espacios de nombre, además no pueden hacerse declaraciones de namespace dentro de bloques, como funciones. Un namespace se define de la siguiente manera:

```
namespace <nombre_del_namespace>
{
... //declaraciones y/o definiciones de variables, funciones, clases
}
```

aquí nombre_del_namespace es un identificador estandar C++.

El nombre del espacio funciona como un prefijo para las variables, funciones o clases declaradas en su interior, de modo que para acceder a una de esas variables se tiene que usar un especificador de ámbito (::), o activar el espacio con nombre adecuado.

Presentamos el siguiente código que genera un espacio de nombre.

```
#include <iostream>
namespace uno {
   int x;
   namespace dos {
      int x;
      namespace tres {
         int x;
      }
   }
}
using std::cout;
using std::endl;
using uno::x;
int main() {
   x = 10; // Declara x como uno::x
   uno::dos::x = 30;
   uno::dos::tres::x = 50;
   cout << x << ", " << uno::dos::x <<
      ", " << uno::dos::tres::x << endl;
   return 0;
}
```

1.6. Bloque de sentencias

Las sentencias especifican y controlan el flujo de ejecución del programa. Si no existen sentencias específicas de selección o salto, el programa se ejecuta de forma secuencial en el mismo orden en que se ha escrito el código fuente.

En C++ el concepto de bloque de sentencias se utiliza para agrupar un conjunto de sentencias dentro de un ámbito concreto dentro del programa. Un bloque de sentencias es un conjunto de instrucciones englobadas bajo llaves { }.

1.6.1. Sentencia if

La sentencia **if** elige entre varias alternativas en base al valor de una o más expresiones booleanas.

Sintaxis:

if(<expresioón booleana>)
<bloque a ejecutar cuando la expresión es verdadera>

else

<bloque a ejecutar cuando la expresión es falsa>

La sentencia else es opcional, puede utilizarse o no. En el caso de no utilizarlo, cuando la expresión evaluada sea falsa la ejecución continuará con la sentencia inmediatamente posterior al if.

1.6.2. Comparaciones en C++

En C++ las comparaciones se especifican mediante el operador ==, en primera posición la constante y como segundo mienbro de la igualdad la variable. Dicha comparación nos regresaráun booleano.

```
if ( constante == variable){ }
```

1.6.3. Asignaciones en C++

En las asignaciones se debe evitar la conversión explícita de tipos de datos. Se aconseja no hacer asignaciones múltiples, ya que estas dan a lugar a actuaciones erróneas. En general las sentencias de asignación tienen la forma:

```
tipo variable;
variable = expresión;
```

1.6.4. Sentencia Switch

En casos en los que el programa presenta varias elecciones después de checar un expresión múltiple o multialternativa, donde el valor de una expresión determina qué sentencias serán ejecutadas es mejor utilizar una sentencia **switch**.

Esta estructura ocupa la palabra reservada **break** que permite que el flujo del programa se detenga justo después de la ejecución de la sentencia anterior a ese break, pidiendo que se ejecuten las sentencias correspondientes a las siguientes alternativas de switch.

Por otro lado **default** es opcional y engloba un conjunto de sentencias que se ejecutan en caso de que ninguna de las alternativas del switch tenga un valor que coincida con el resultado de evaluar la expresión del selector.

```
switch(var int o char)
{
case const1: instrucciones;
break;

case const2: instrucciones;
break;

default: instrucciones;
};
```

1.6.5. Sentencia For

La sentencia **for** se usará para definir un ciblo en el que una variable se incrementa de manera constante en cada iteración y la finalización del ciblo se determina mediante una expresión constante. Como contador en **for** se utilizarán preferiblemente variables de un solo caracter como i, j, k, declarandolas dentro del mismo ciclo.

Su formato general es:

```
for (inicialización; condición; incremento)
{ instrucción(es); };
```

1.6.6. Sentencias While y do While

Al comenzar un ciclo **while o do...while** la expresión de control debe tener un valor claramente definido, para impedir posibles indeterminaciones o errores de funcionamiento.

La sentencia **while** se usará para definir un ciclo en el que la condición de terminación se evalúa al principio del mismo.

Su formato general es :

```
cargar o inicializar variable de condición;

while(condición)
{

grupo cierto de instrucciones;
instrucción(es) para salir del ciclo;

};
```

La sentencia **do...while** se usará para definir un ciclo en el que la condición de terminación se evaluará al final del mismo.

Su formato general es:

```
cargar o inicializar variable de condición;

do {

grupo cierto de instrucción(es);
instrucción(es) de rompimiento de ciclo;

} while (condición);
```

1.6.7. Sentencia exit y return

La sentencia **exit** finaliza la ejecución de un proceso de manera inmediata, forzando la vuelta al sistema operativo. No se aconseja su utilización en cualquier parte del código, siendo preferible controlar el flujo en el proceso mediante bucles condicionales y devolver el control mediante la sentencia return.

La sentencia **return** se utiliza para salir de una función o procedimiento, volviendo al punto en el cual se llamó a dicha función o procedimiento. En el código hay que minimizar la utilización de return, sólo tendría que aparecer una vez en cada función o procedimiento, al final del mismo, de manera que se tenga un sólo punto de entrada a la función y un solo punto de salida de la misma.

1.6.8. Algunos ejemplos

A continuación te presentamos algunos ejemplos en donde se implementan las sentencias del lenguaje C++.

1. Este programa cuenta numeros en intervalos de ocho

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

using namespace std;

int main()
{
  int numero,contador,sumador;
  sumador=contador=0;
  do
  {
  cout << "Introduzca un número mayor que 0 y menor que 500: ";
  cin >> numero;
  }while(numero < 0 || numero > 500);
  // La condición controla el intervalo establecido.

  //Controla que no entren números con diferencia inferior a ocho hasta 500 y no superior.
  if(numero<=492)
  {
    for(numero;numero<500;numero+= 8)
    {
     sumador = sumador + numero;
     contador = contador + 1;
     cout << numero << " , " ;
    }
  }
  cout << "\nEsta es la suma: " << sumador << endl;
  cout << "El número total hasta 500 separado ocho posiciones es: " <<
    contador << endl;
  system("PAUSE");
  return 0;
}
```

2. La salida de este programa es una calculadora de operaciones básicas, mediante la implemenatación de la sentencia **Switch**.

```
#include<iostream>

using namespace std;

int main(int argc, char *argv[])
{
 int a, b;
 char oper;

 cout << "Introduzca la operacion a realizar usando espacios:\n\n";
 cin >> a >> oper >> b;

 switch(oper)
 {
  case '+': cout << a << oper << b << " = " << a + b << endl;
  break;
  case '-': cout << a << oper << b << " = " << a - b << endl;
  break;
  case '*': cout << a << oper << b << " = " << a * b << endl;
  break;
  case '/': cout << a << oper << b << " = " << a / b << endl;
  break;
  case '%': cout << a << oper << b << " = " << a % b << endl;
  break;

  default: break;
 }

 return 0;
}
```

1.7. Funciones y Estructuras

1.7.1. Funciones

Una **función** es una parte de un programa (subrutina) con un nombre, que puede ser invocada (llamada a ejecución) desde otras partes tantas veces como se desee. Un bloque de código que puede ser ejecutado como una unidad funcional, opcionalmente puede recibir valores ó bien se ejecuta y devuelve un valor. Desde el punto de vista de la organización, podemos decir que una función es algo que permite un cierto orden en una maraña de algoritmos. Como resumen de lo anterior podemos concluir que el uso de funciones se justifica en dos palabras: organización y reutilización del código.

Son utilizadas para descomponer grandes problemas en tareas simples y para implementar operaciones que son comunmente utilizadas durante un programa y de esta manera reducir la cantidad de código. Cuando una función es invocada se le pasa el control a la misma, una vez que esta termina su tarea sede el control al punto desde el cual fue llamada.

Las funciones se declaran y se definen exactamente igual que en C, estas puede utilizar prototipo (prototype). Un *prototipo* es un modelo limitado de una entidad más completa que aparecerá después. En el caso de funciones, la función es la entidad completa que vendrá después, y la declaración de dicha función es el prototipo. El prototipo da un modelo de interface a la función.

La de claración de una función tiene el siguiente cuerpo:

```
tipo-de-retorno nombre-de-función(parámetros)
 {
  declaraciones
  proposiciones
 }
```

Si la función no retorna un valor el tipo de retorno es **void**. Si la función no tiene parámetros se escribe **myfunc(void)**.

1.7.2. Estructuras

Una **estructura** es un grupo de variables relacionadas de manera lógica, las cuales pueden ser de diferentes tipos y declaradas en una sola unidad, donde la unidad es la estructura. Estas son una especie de híbrido entre las estructuras de C y las clases de C++, de hecho

podrían ser sustituidas por clases definidas ya que disponen de tres tipos de modificadores de acceso: *público, privado y protegido*, y el considerarlas como tales, permite un mayor control de acceso de las estructuras.

En C++ se forma una estructura utilizando la palabra reservada **struct**, seguida por un campo etiqueta opcional, y luego una lista de miembros dentro de la estructura. La etiqueta opcional se utiliza para crear otras variables del tipo particular de la estructura.

```
struct campo_etiqueta{
  tipo_miembro miembro_1;
  tipo_miembro miembro_2;
  :
  tipo_miembro miembro_n;
};
```

Un punto y coma finaliza la definición de una estructura puesto que ésta es realmente una sentencia C++ .

1.8. Arreglos y Apuntadores

1.8.1. Arreglos

El lenguaje C++ permite construir estructuras más complejas a partir sus tipos básicos. Una de las construcciones que podemos definir son los arreglos.

Arreglo: Colección ordenada de elementos de un mismo tipo. Ordenada significa que cada elemento tiene una ubicación determinada dentro del arreglo y debemos conocerla para poder acceder a él.

Se ***definen*** de la siguiente manera:

<tipo>nombre_variable [longitud];

Con esto diremos que nombre_variable es un arreglo de longitud elementos del tipo <tipo >. Cabe destacar que longitud debe ser cualquier expresión entera constante mayor que cero.

Un arreglo se ***asigna*** de la siguiente manera:

nombre_variable[índice] = expresión del tipo <tipo >

Esta instrucción asigna el valor asociado de la expresión a la posición índice del arreglo nombre_variable. El índice debe ser una expresión del tipo entero en el rango [0, longitud-1]. Cabe destacar que C++ no chequea que el valor de la expresión sea menor a longitud, simplemente asigna el valor a esa posición de memoria como si formara parte del arreglo.

Para tener ***acceso*** al contenido de un arreglo:

nombre_variable[índice] es valor del tipo <tipo>que puede ser asignado a una variable, o pasado como parámetro, imprimirlo, etc. Aquí también vale la aclaración de que el índice debe estar dentro del rango de definición del arreglo, C++ no verificará que esto sea cierto y devolverá lo contenido en la posición de memoria correspondiente a un arreglo de mayor longitud, el dato obtenido de esta manera es basura.

1.8.2. Apuntadores

Los **apuntadores** también conocidos como *punteros*, son variables que guardan direcciones de memoria C++. Proporcionan mucha utilidad al programador para accesar y manipular datos de maneras que no es posible en otros lenguajes. También son útiles para pasarle

parámetros a las funciones de tal modo que les permiten modificar y regresar valores a la rutina que las llama.

Los punteros son un recurso que en cierta forma podría considerarse de muy bajo nivel, ya que permiten manipular directamente contenidos de memoria. Por esta razón, su utilización puede presentar algunos problemas y exigen que se preste una especial atención a aquellas secciones del código que los utilizan. Recuerde que los errores más insidiosos y difíciles de depurar que se presentan en los programas C++, están relacionados con el uso descuidado de punteros.

Los punteros, al igual que una variable común, pertenecen a un tipo (type), se dice que un puntero *apunta a* ese tipo al que pertenece. Ejemplos:

```
int* pint;            //Declara un puntero a entero
char* pchar;          //Puntero a char
fecha* pfecha;        //Puntero a objeto de clase fecha
```

Supongamos una variable de tipo entero que se llama *contenidoRAM* y otra variable que se llama *direccionRAM* que puede contener una variable de tipo entero. En C/C++ una variable precedida del operador & devuelve la dirección de la variable en lugar de su contenido. Así que para asignar la dirección de una variable a otra variable del tipo que contiene direcciones se usan sentencias como esta:

```
direccionRam = &contenidoRAM
```

El accceso al contenido de una celda cuya dirección está almacenada en la variable *direccionRAM* es tan sencillo como poner al inicio de la variable apuntador un asterisco: * *direccionRAM*. Lo que se ha hecho es eliminar la referencia directa. Por ejemplo, si se ejecutan las siguientes dos sentencias, el valor de la celda llamada *contenidoRAM* será de 20, suponiendo que este valor es el contenido de esa variable.

```
direccionRAM  = &contenidoRAM;
*direccionRAM = 20;
```

Para poner más claro todo lo antes explicado anexamos el siguiente programa, que utilizando arreglos y apuntadores nos permite encontrar el elemento más pequeño del arreglo que se le pase.

```
# include <iostream.h>

main ( )
{
 int tabla[10], minimo (int *a, int n);

cout << "Introduzca 10 enteros: \n";
for (int i=0,i<10,i++) cin>>tabla[i];
cout<<"\n el valor minimo es"
<<minimo (tabla,10)<<end;
}

int minimo (int *a, int n)
{ int menor;
   menor=*a;
 for (int i=1;i<n;i++)  if (*(a+1)<menor)
menor=*(a+i);
return menor;
}
```

1.9. Manejo de Memoría Dinámica

La **memoría dinámica** es un espacio de almacenamiento que se solicita en tiempo de ejecución. El espacio libre de almacenamiento en C++ se conoce como **almacenamiento libre**.

Además de solicitar espacios de almacenamiento, también podemos liberarlos (en tiempo de ejecución) cuando dejemos de necesitarlos.

Para realizar esta administración de la memoria dinámica, C++ cuenta con dos operadores para la gestión de memoría: *new y delete* que forman parte del lenguaje y no de un librería, de modo no se necesitan incluir archivos de encabezados para utilizarlos.

1. El operador **new** reserva memoría dinámica, su propósito es crear arrays cuyo tamaño pueda ser determinado mientras el programa corre.

   La forma de utilizar este operador es la siguiente:

   ```
   <nombre_puntero> = new <nombre>
             [<argumentos> ];
   ```

   La memoría reservada con new será válida hasta que se libere con delete o hasta el fin del programa, aunque es aconsejable liberar siempre la memoría reservada con new usando delete.

   Si new no encuentra espacio para alojar lo que se pide, devuelve 0, que significa puntero nulo, además hace una comprobación de tipos, si el puntero no es del tipo correcto lanzará un mensaje de error.

2. El operador **delete** se usa para liberar bloques de memoría dinámica reservada con new. El único inconveniente que podría ocasionar el uso del operador delete sería el utilizarlo en aquellos casos en el que el puntero a borrar realmente no ha sido reservado correctamente con la llamada a new y tiene un valor no nulo, pero el operador delete verifica que el puntero que se le pasa no sea nulo antes de destruir el objeto asociado.

   La sintaxis de delete es muy sencilla:

   ```
   delete <nombre_puntero>;
   ```

Una ejemplo de como se utilizan los dos operadores descritos anteriormente es el siguiente código:

```
#include <iostream>

using namespace std;

int main()
{
  int * xp;
  double * dp;
  int * arr_dynamic;
  int ** matriz;

  struct Punto3d
  {
    float x,y,z;
  } * punto;


  int n,m;

  cout <<" Pasame un entero y un doble"
       << endl;

  xp = new int;
  dp = new double;

  cin >> *xp >> *dp;

  cout << *xp <<" " << *dp << endl;

  cout << "Dame n y m"
       << endl;

  cin >> n >> m;

  arr_dynamic = new int[n];

  matriz = new int* [n];

  for(int i=0; i < n ; i++)
    matriz[i] = new int [m];

  srand(m);
```

```
  for(int i=0; i< n; i++)
    arr_dynamic[i] = rand()%10;

  for(int i=0; i< n; i++)
    for(int j=0; j <m; j++)
      matriz[i][j] = rand()%10;

  cout << endl;

  for (int i =0; i < n; i++)
    cout << arr_dynamic[i] <<" ";

  cout << endl;
  cout << endl;

  for(int i=0; i < n; i++){
    for(int j=0; j < m; j++)
      cout << matriz[i][j] <<" ";
    cout << endl;
  }

  punto = new Punto3d;

  punto -> x = 3;
  punto -> y = 3;
  punto -> z = 10;


  cout << punto ->x <<" "<< punto->y<<" "
       << punto ->z << endl;

  delete xp;
  delete dp;
  delete [] arr_dynamic;

  for(int i =0; i < n; i++)
    delete [] matriz[i];

  delete punto;

  return 0;

}
```

1.10. Estructura de Datos

Una **estructura de datos** nos permite organizar un conjunto de datos para manipularlos de manera más conveniente. A continuación presentamos las estructuras de datos más comunes así como su implementación en C++.

1. Listas

   Dado un dominio D, una lista de elementos de dicho conjunto es una sucesión finita de elementos del mismo. En lenguaje matemático, una lista es una aplicación de un conjunto de la forma {1,2, ... ,n} en un dominio D:

   R:{1,2, ... ,n} —>D

   Una lista se suele representar de la forma:

   <a1,a2, ... ,an>con ai = a(i)

   A *n* se le llama longitud de la lista.

   A 1,2,...,n se les llama posiciones de la lista. El elemento a(i)=ai, se dice que ocupa la posición i. Si la lista tiene *n* elementos, no existe ningún elemento que ocupe la posición n+1. Sin embargo, conviene tener en cuenta dicha posición, a la que se llama posición detrás de la última, ya que esta posición indicará el final de la lista. A *a1* se le llama primer elemento de la lista y a *an* último elemento de la lista. Si n = 0, diremos que la lista está vacía y lo representaremos como <>. Los elementos de una lista están ordenados por su posición. Así, se dice que *ai* precede a *ai+1* y que *ai* sigue a *ai-1.*

   Una lista es un conjunto de elementos del mismo tipo que: O bien es vacío, en cuyo caso se denomina lista vacía. O bien puede distinguirse un elemento, llamado cabeza, y el resto de los elementos constituyen una lista *L*, denominada resto de la lista original.

   Una propiedad importante con la que se caracteriza a las listas es que su longitud puede aumentar o disminuir, según se requiera ya que podemos insertar o eliminar elementos en cualquier posición de ella.

   Ahora presentamos un ejemplo de implementación de una Lista.

```
//Lista.cpp Crea y maneja una lista enlazada
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>


//Declaracion dato struct
struct lista_elementos
 {
   char elem[40];
   struct lista_elementos *sig;
 };
```

```
typedef struct lista_elementos nodo;

//Prototipos de funciones

int menu(void);
void crear(nodo *pt);
nodo *insertar(nodo *pt);
nodo *eliminar(nodo *pt);
void mostrar(nodo *pt);

void main()
 {
   nodo *prin;
   int eleccion;
   do
     {
       eleccion =menu(); //Llamada a la funcion menu. el valor devuelto es usado en el sig swit
       switch (eleccion)
        {
          case 1:
            prin= (nodo *) malloc(sizeof(nodo)); //Reserva dinamica de memoria
            crear(prin);   //Llamada a funcion
            printf("\n");
            mostrar(prin);
            continue;   //Ojo, no es break

         case 2:
           prin= insertar(prin);
           printf("\n");
           mostrar(prin);
           continue;

         case 3:
           prin=eliminar(prin);
           printf("\n");
           mostrar(prin);
           continue;

         default:
           printf("fin de las operaciones\n");
       }  //de switch
    }while(eleccion !=4);
  }  //de main()
```

2. Pilas

Una **Pila** es una clase especial de lista en la cual todas las inserciones y borrados tienen lugar en un extremo denominado extremo, cabeza o tope. También se les conoce como *listas FIFO* (último en entrar, primero en salir).

El modelo intuitivo de una pila es un conjunto de objetos apilados de forma que al añadir un objeto se coloca encima del último añadido, y para quitar un objeto del montón hay que quitar antes los que están por encima de él.

Ahora presentamos un ejemplo de implementación de una Pila.

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

const int TPILA=5; // TPILA es el valor maximo de elementos
                   // que puede tener nuestra pila

class PILA{
  public:
     int mipila[TPILA];   // Crea mi PILA de tamano TPILA
     int apilados;        // Numero de objetos en LA PILA
    void reset();         // Vacia LA PILA
    void push(int v);     // Agrega Valores en el tope de la PILA
     int pop();           // Retorna y elimina el tope de la PILA
};

void PILA::reset()        // Vacia la PILA
{
  apilados=0;
}

//Se Agrega(PUSH) un valor en el tope de la pila
void PILA::push(int v)
{
// Comprueba que haya espacio dentro de la pila
// para poder agregar el nuevo valor
  if(apilados<TPILA)
     {
     mipila[apilados++]=v;
     }
}

// Se Elimina (POP) el ultimo valor de la pila
// y retorna el nuevo tope
int PILA::pop()
{
  if(apilados>0)
```

```
    {
    cout<<"El valor del tope eliminado era: ";
    // Retorna el valor del tope que fue eliminado
    return(mipila[--apilados]);
    }
  else
  cout<<"No existen datos para eliminar.  ERROR ";
  return (0);
}
```

3. Colas

   Una **Cola** es otro tipo especial de lista en el cual los elementos se insertan por un extremo(el posterior) y se suprimen por el otro (el anterior o frente). Las colas se conocen también como listas FIFO (primero en entrar, primero en salir). Las operaciones para las colas son análogas a las de las pilas. Las diferencias sustanciales consisten en que las inserciones se hacen al final de la lista, y no al principio.

   Existen un caso especial de estas, estamos hablando de una ***cola de prioridad*** que se define como una estructura de datos que se utiliza en determinados problemas es los que de forma continua es necesario buscar y suprimir el menor elemento de una colección de valores. Por ejemplo, problemas en los que se mantiene una lista de tareas organizada por prioridades (sistemas operativos, procesos de simulación, etc.).

   Se requiere que los elementos de la colección se puedan ordenar mediante algún criterio (su prioridad), aunque no es necesario que los elementos pertenezcan a una colección ordenada. Para determinar la prioridad de los elementos se establece la función de prioridad.

   Ahora presentamos un ejemplo de implementación de una Cola.

```
// Fichero MCola.h
#ifndef _MCola_h_
#define _MCola_h_
#include "MCadena.h"
namespace MCola
{
  using namespace MCadena;
  typedef struct TNodo *TLista;
  struct TNodo
  { TCadena val;
     TLista sig;
  };
  struct TCola
  { TLista frente;
     TLista final;
  };
  TCola CrearCola();
```

```
  void DestruirCola(TCola &c);
  void MeterCola(TCola &c, TCadena s, bool &llena);
  void SacarCola(TCola &c, TCadena &s, bool &vacia);
  bool ColaLlena(TCola c);
  bool ColaVacia(TCola c);
}
#endif
```

4. Arboles

   El tipo de dato **Árbol** es un tipo de dato mas complejo que los tipos lineales, es decir, no existe una relación de anterior y siguiente entre los elementos que la componen (cada elemento tendrá uno anterior y otro posterior, salvo los casos de primero y último).

   Podemos considerar la estructura de árbol de manera intuitiva como una estructura jerárquica. Por tanto, para estructurar un conjunto de elementos *ei* en árbol, deberemos escoger uno de ellos *e1* al que llamaremos *raíz del árbol*. Del resto de los elementos se selecciona un subconjunto *e2,...,ek* estableciendo una relación padre-hijo entre la raíz y cada uno de dichos elementos de manera que *e1* es llamado el *padre de e2, de e3, ...ek* y cada uno de ellos es llamado un *hijo de e1*. Iterativamente, podemos realizar la misma operación para cada uno de estos elementos asignando a cada uno de ellos un número de 0 o más hijos hasta que no tengamos más elementos que insertar. El único elemento que *no tiene padre es e1*, la raíz del árbol. Por otro lado hay un conjunto de elementos que *no tienen hijos* aunque sí padre que son llamados *hojas*.

   a) Arboles Binarios

   Un **árbol binario** puede definirse como un árbol que en cada nodo puede tener como mucho grado 2, es decir, a lo más 2 hijos. Los hijos suelen denominarse hijo a la izquierda e hijo a la derecha, estableciéndose de esta forma un orden en el posicionamiento de los mismos. Número máximo de nodos por nivel en un árbol binario *en un nivel i es 2 i-1*.

   Si n es un nodo y T izq , T der son árboles binarios, entonces podemos construir un nuevo árbol binario que tenga como raíz el nodo n y como subárboles Tizq y T der (subárbol izquierdo y subárbol derecho de n, respectivamente). Un árbol vacío es un árbol binario.

Ahora presentamos un ejemplo de implementación de un Árbol Binario.

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

typedef int Tipo;

class Arbol_Bin{
   typedef struct Nodo{
     Tipo info;
     Nodo* h_izq;
     Nodo* h_der;
   }*NodoPtr;

   NodoPtr raiz;
   bool op_correcta;

   void EliminaSubArb(NodoPtr);
   NodoPtr InsertaNodo(NodoPtr,NodoPtr);
   NodoPtr EliminaNodo(Tipo,NodoPtr);
   void PrintNodo(NodoPtr);

 public:
   Arbol_Bin(void){raiz = 0;}
   ~Arbol_Bin(void){EliminaSubArb(raiz);}
   bool Inserta(Tipo);
   bool Elimina(Tipo);
   void Print(void);
 };

void Arbol_Bin::EliminaSubArb(NodoPtr nodo){

  if(nodo !=0){
    EliminaSubArb(nodo->h_izq);
    EliminaSubArb(nodo->h_der);
    delete nodo;
  }
}

Arbol_Bin::NodoPtr Arbol_Bin::InsertaNodo(NodoPtr nodo,NodoPtr arbol){
  if(arbol == 0)
    return nodo;
  if(nodo -> info <= arbol -> info)
    arbol -> h_izq = InsertaNodo(nodo,arbol->h_izq);
  else if(nodo -> info > arbol ->info)
```

```
    arbol -> h_der = InsertaNodo(nodo,arbol->h_der);
  return arbol;
}

Arbol_Bin::NodoPtr Arbol_Bin::EliminaNodo(Tipo t, NodoPtr arbol){
  if(arbol == 0)
    return arbol;
  if (t < arbol->info)
    arbol -> h_izq = EliminaNodo(t,arbol -> h_izq);
  else if(t > arbol->info)
    arbol -> h_der = EliminaNodo(t,arbol -> h_der);
  else{
    if(arbol -> h_izq == 0)
       arbol = arbol -> h_izq;
    if(arbol -> h_der == 0)
       arbol = arbol -> h_der;
    else{
      NodoPtr temp = arbol;
      arbol-> h_der = InsertaNodo(arbol->h_izq,arbol->h_der);
      arbol = arbol -> h_der;
      delete temp;
    }
    op_correcta = true;
  }
   return arbol;
}

void Arbol_Bin::PrintNodo(NodoPtr nodo){
  if(nodo!=0){
    cout << nodo ->info <<endl;
    PrintNodo(nodo->h_izq);
    PrintNodo(nodo->h_der);
  }
}

bool Arbol_Bin::Inserta(Tipo t){
  NodoPtr nodo = new Nodo;
  op_correcta = true;
  if(nodo != 0){
    nodo ->info = t;
    nodo -> h_izq = nodo -> h_der = 0;
    raiz = InsertaNodo(nodo,raiz);
  }
  return op_correcta;
}
```

```
bool Arbol_Bin::Elimina(Tipo t){
  op_correcta = true;
  raiz = EliminaNodo(t,raiz);
  return op_correcta;
}

void Arbol_Bin::Print(void){
  if(raiz != 0)
   PrintNodo(raiz);
}


void main(){
  Arbol_Bin arbol;
  int info,NumElemArb = 10;

  srand(time(0));
  for(int i=0; i<NumElemArb; i++)
    arbol.Inserta(rand()%10);
  arbol.Print();
  cout << "Proporciona el la llave del nodo a eliminar"<<endl;
  cin >> info;
  arbol.Elimina(info);
  arbol.Print();
}
```

b) Arboles BB

Un **árbol binario de búsqueda (ABB)** es un árbol binario con la propiedad de que todos los elementos almacenados en el subárbol izquierdo de cualquier nodo x son menores que el elemento almacenado en x ,y todos los elementos almacenados en el subárbol derecho de x son mayores que el elemento almacenado en x.

Una interesante propiedad es que si se listan los nodos del ABB en inorden nos da la lista de nodos ordenada.