Temas del Lenguaje C++

Programación Avanzada

Universidad Nacional de Entre Ríos
Facultad de Ingeniería
Bioingeniería – 2014

Tabla de Contenidos

Temas del Lenguaje C++

Universidad Nacional de Entre Ríos
Facultad de Ingeniería – Bioingeniería
Programación Avanzada

Prefacio

Este documento tiene como objetivo presentar los conceptos básicos y las características generales del lenguaje C++ que posibiliten la realización de programas para la representación de soluciones de problemas reales en una computadora.

Los temas tratados y el desarrollo que se realiza sobre ellos persiguen la enseñanza en ámbitos universitarios que necesitan conocimientos básicos de informática como una herramienta para la resolución de problemas.

Debido al fin didáctico que tiene este documento, se ha priorizado la enseñanza de una metodología de programación correcta tratando de conservar la facilidad de uso como herramienta. Se considera que lo más importante es adquirir una base de conocimientos que permita una posterior evolución hacia conocimientos más profundos de la programación.

Este documento no está mayormente centrado en la descripción teórica de conceptos o bibliotecas, sino que le da prioridad a la realización de códigos reusables con los conceptos de cada capítulo.

Los tópicos que más se destacan son:

- Conocimientos básicos sobre metodología de la programación.
- Lo más útil de la sintaxis de un lenguaje.
- El uso de bibliotecas.

Se debe tener en cuenta que los temas tratados presentan un grupo de conceptos básicos necesarios para desarrollar la más amplia variedad de soluciones. En la medida en que se dominen estos conceptos básicos aparecerán necesidades de resolver otros problemas más específicos como la programación en tiempo real o la programación para microcontroladores, para lo que se sugiere continuar el aprendizaje consultando la bibliografía específica.

Metodología de la programación

Se desarrolla una introducción a los conceptos de programación orientada a objetos, que permite adquirir "una metodología" para aplicarla a la elaboración de soluciones. Es muy valioso adquirir este conocimiento básico debido a que permitirá evolucionar hacia la solución de problemas más complejos.

Sintaxis del lenguaje

Se presenta la sintaxis del lenguaje C++ (norma internacional estándar ANSI-C++ publicada en noviembre de 1997). De la extensa sintaxis del lenguaje se utiliza la más útil. Por ejemplo, si bien existen muchos programas que utilizan *printf* (del lenguaje C) para realizar salidas por pantalla, en este documento sólo se utiliza *cout*. Otro ejemplo típico de los usuarios del lenguaje C es la utilización de punteros para crear estructuras dinámicas. En este documento se minimiza este tema porque a partir de la biblioteca estándar de plantillas se logra el mismo fin con mayor facilidad y seguridad para el programador.

Bibliotecas y utilidades de programas externos

Entre las premisas para que una herramienta sea útil está su facilidad de uso y su rápida aplicabilidad, de forma de poder dirigir todo el esfuerzo hacia el problema que se está buscando solucionar o representar en la computadora. Si bien los elementos incorporados al lenguaje estándar son muy amplios, hay casos en que se necesita un importante desarrollo que no se quiere realizar por no ser este el objetivo fundamental del trabajo. Por ejemplo, para un ingeniero que trabaja en procesamiento de señales, quizás no sea de su interés desarrollar la metodología para dibujar un gráfico en pantalla porque esto le requeriría un gran esfuerzo, su mente debería estar centrada en las técnicas de procesamiento de señales.

Las bibliotecas vienen a "complementar" el lenguaje y por lo tanto mejorar la herramienta. Para el caso anterior existe por ejemplo GNUPlot, que con un mínimo esfuerzo de codificación permite obtener excelentes gráficos. Persiguiendo los mismos objetivos existen bibliotecas de cálculo numérico, para representación y procesamiento de imágenes, etc.

De todos los temas tratados existe en Internet una profusa documentación y variedad de herramientas muchas de ellas específicas para la solución de problemas particulares. Antes de encarar un nuevo proyecto que necesite una representación informática, se aconseja la búsqueda en internet.

Estilo del documento

En la mayoría de los casos se presentan programas simples que ilustran el tema que se está tratando. Una buena forma de aprender el lenguaje es copiar estos ejemplos a un editor para luego compilarlos y analizar su comportamiento realizando modificaciones y observando los resultados.

Los ejemplos del lenguaje se realizan en un ambiente de desarrollo de "salida por consola". Si bien existen entornos con componentes visuales en ventanas, se prefirió desarrollar los temas de la primer forma por la simplicidad de los programas resultantes, su generalidad y posibilidad de transferencia directa a otros entornos. Esta simplicidad es muy útil para el alumno que está dando sus primeros pasos en programación, dado que tiene un control total de todo lo que hace su programa y puede saber exáctamente la función de cada línea del código fuente. Los entornos visuales agregan una extensa codificación a los componentes que agregan (botones, menús desplegables, etc) que en un primer acercamiento al lenguaje crea confusión.

Origen de la documentación

Este documento fue desarrollado en la Cátedra de Programación Avanzada de la Facultad de Ingeniería de la Universidad Nacional de Entre Ríos (Argentina). Existen temas que tuvieron su origen en material disponible en Internet realizándose las citas correspondientes.

Primer programa en C++

Probablemente la mejor manera de empezar a aprender un lenguaje de programación es con un programa como el que está a continuación, el que nos permite incorporar algunos conceptos iniciales.

```
// primer programa en C++
#include <iostream>
using namespace std;

int main ()
{
  cout << "Hola Mundo!";
  return (0);
}
```

```
Hola Mundo!
```

Este código debe ser poescrito con un editor de texto que no agregue caracteres especiales. Por ejemplo, no utilice procesadores de texto como Microsoft Word o Write. Se sugiere ConText para entornos Windows (www.context.com) o Scintilla (www.scintilla.org). Para entornos Linux el editor de textos Kate es el más apropiado.

El lado izquierdo muestra el código fuente para el primer programa, que se puede nombrar, por ejemplo, *"holamundo.cpp"*. El lado derecho muestra el resultado del programa una vez compilado y ejecutado. La manera de revisar y compilar un programa depende del compilador que se esté usando, de si posee una Interfaz de Desarrollo o no y de su versión.

El programa anterior es el primer código que la mayoría de los programadores aprendices escriben por primera vez, y su resultado es la impresión en pantalla de *"Hola Mundo!"*. Es uno de los programas más simples que pueden escribirse en C++, pero ya incluye algunos componentes básicos que cada programa de C++ pueden tener:

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

Ésta es una línea de comentario. Todas las líneas que empiezan con dos barras (//) son consideradas comentarios y no tienen ningún efecto en el comportamiento del programa. Los comentarios pueden ser usados por el programador para incluir explicaciones cortas u observaciones dentro del propio código fuente. En este caso, la línea es una descripción breve de lo que nuestro programa hace.

```
#include <iostream>
```

Las líneas que empiezan con un símbolo de numeral (#) son directivas para el preprocesador. No son líneas del código ejecutables pero son indicaciones para el compilador. En este caso la línea *#include <iostream>* le dice al preprocesador del compilador que debe incluir el archivo de encabezamiento estándar iostream. Este archivo específico incluye las declaraciones de la biblioteca del entrada-salida básica estándar de C++, y es incluido porque su funcionalidad se usa después en el programa para mostrar el texto por pantalla.

```
using namespace std;
```

Esta línea se explica detalladamente más adelante. Por ahora simplemente pongámosla en todos los programas.

```
int main ()
```

Esta línea corresponde a la declaración de la función *main* (principal). La función *main* es por donde todos los programas de C++ empiezan su ejecución. Es independiente de si se encuentra al principio, al final o al medio del código, su contenido siempre es el primero en ser ejecutado cuando un programa comienza. Por esta razón, es esencial que todos los programas de C++ tengan una función *main*.

La palabra *main* va seguida por un par de paréntesis "( )" porque es una función. En C++ todas las funciones son seguidas por un par de paréntesis que, opcionalmente, puede incluir argumentos dentro de ellos que son datos que se le pueden pasar al programa. El contenido de la función *main* sigue inmediatamente a su declaración formal, encerrada entre llaves "{ }", como en el ejemplo.

```
cout << "Hola Mundo!";
```

Esta instrucción hace lo más importante en este programa. *cout* es el flujo estándar de salida en C++ (normalmente a la pantalla), y la línea completa inserta una sucesión de caracteres (en este caso *"Hola Mundo!"*) en este flujo de salida, en este caso, hacia la pantalla. *cout* se declara en el archivo de encabezamiento *iostream*, así que para poder usarlo, ese archivo debe ser incluido.

Observar que la línea termina con un punto y coma ";". Este caracter tiene el significado de 'finalización de la instrucción' y debe ser incluido después de cada instrucción en cualquier programa en C++ (uno de los errores más comunes de los programadores en C++ es olvidarse de incluir un punto y coma al final de cada instrucción).

```
return (0);
```

La instrucción *return* hace que la función *main( )* termine devolviendo un código, en este caso el cero. Este es el modo más usual de terminar un programa que no ha encontrado ningún error durante su ejecución. Como se verá en los próximos ejemplos, todos los programas en C++ terminan con una línea similar a esta.

El programa se ha escrito en líneas diferentes para ser más legible, pero se podría haber escrito así:

```
int main () { cout << " Hola Mundo! "; return (0); }
```

en sólo una línea y esto habría tenido exactamente el mismo resultado porque en C++ la separación entre las instrucciones se especifica con un punto y coma al final. La división del código en diferentes líneas sirve sólo para hacerlo más legible y esquemática la lectura.

Las directivas del preprocesador (aquéllas que empiezan con #) están fuera de esta regla porque **no** son verdaderas instrucciones. Deben especificarse en su propia línea y no exigen incluir un punto y coma al final.

Comentarios

Los comentarios son pedazos del código fuente desechados por el compilador. Ellos no existen en el programa compilado. Su propósito sólo es permitir al programador insertar notas o descripciones dentro del código fuente.

C++ soporta dos maneras de insertar comentarios:

```
// comentarios de linea
/* comentarios
   de
   bloque */
```

El primero de ellos (el comentario de línea) desecha todo desde donde el par de barras (//) se encuentra hasta el extremo de esa misma línea. El segundo, el comentario del bloque, desecha todo entre los caracteres /* y la próxima aparición de los caracteres */, con la posibilidad de incluir varias líneas.

El siguiente programa agrega otros comentarios:

```
/* segundo programa en C++
   con más comentarios */

#include <iostream>
using namespace std;

int main ()
{ cout << "Hola Mundo!. ";  // dice Hola Mundo!
  cout << "Este es un programa en C++.";
          // dice que es un programa en C++
  return (0);
}
```

```
Hola Mundo!. Este es
un programa en C++.
```

Si se incluyen comentarios dentro del código fuente de los programas sin usar las combinaciones de caracteres de comentario //, /* ni */, el compilador los tomará como si fueran instrucciones de C++ y probablemente mostrará mensajes de error.

Introducción a la compilación

Compilación en consola

Este capítulo trata la compilación con el compilador g++ para el lenguaje C++ estándar de distribución gratuita con versiones para la mayoría de los sistemas operativos (para más información ver www.gnu.org), opciones para depuración, posibilidad de optimizaciones y generación de advertencias (warnings) sobre el código del programa.

Primero se presenta un programa de ejemplo para compilar con g++. Para los propósitos de este ejemplo, grabe el siguiente código con el nombre "test1.cpp":

```
#include <iostream>
using namespace std;

int main () {
  cout << "Hola, nuevo mundo!";

return (0);
}
```

La compilación de este simple programa se puede hacer de la siguiente manera:

```
g++ test1.cpp
```

Esto generará un archivo llamado "a.exe" que es el programa ejecutable (en Linux será "a.out"). Si se ejecuta este programa (en Linux por consola será "./a.out"), se tendrá por pantalla:

```
Hola, nuevo mundo!
```

A menos que se quiera siempre tener programas con el nombre "a.out", se puede utilizar la opción –o de g++, que permite cambiar el nombre del archivo ejecutable. Para utilizar esta opción, la compilación se debe hacer de la siguiente manera:

```
g++ –o test1 test1.cpp
```

Ahora el ejecutable tendrá en nombre "test1.exe" en vez de "a.out" (en Linux no tendrá extensión, es decir sólo, "test1"). Observar que –o es una opción del compilador que está indicando el nombre que tendrá el archivo resultante de la compilación (la letra 'o' es por 'output').

Advertencias

Las advertencias (warnings) son controladas por la opción –W del compilador g++. El uso más común es mostrar todas las advertencias con –Wall.

Si el código anterior es modificado así:

```
#include <iostream>
using namespace std;

int main () {
  int i; // declaración de que se va a usar una variable para números enteros
  cout << "Hola, nuevo mundo!_";

return (0);
}
```

durante la compilación se obtendrá el siguiente mensaje:

```
$ g++ –Wall –o test1 test1.cpp
test1.cpp: In function `int main()':
test1.cpp:6: warning: unused variable `int i'
```

Avisando que se declaró una variable que no se usa.

Tener en cuenta que existen opciones del compilador en mayúscula, y en minúsculas, que no son equivalentes. Además, los mensajes de advertencia aquí mostrados pueden diferir de los obtenidos con distintas versiones del g++, existiendo compiladores con mensajes en español. En programas más complejos la opción –Wall es una herramienta muy útil para prevenir errores. Se recomienda **usar siempre** esta opción y además siempre obtener compilaciones sin mensajes de advertencias. En lo posible, adoptarlo como un hábito.

Optimizaciones con g++

Una de las características más sobresalientes de los compiladores modernos como g++ es su posibilidad de obtener programas de ejecución con una optimización eficiente. El optimizador es parte del compilador y es capaz de examinar el código fuente (o el código assembler generado por el compilador), identificar aquellas áreas que se pueden optimizar, y rescribir el código que hace lo mismo pero en menor espacio o con mejor performance. En el caso de g++, además ofrece la posibilidad de configurar el grado de optimización.

Se puede habilitar la optimización mediante la opción **–O** (letra o mayúscula). Además, se puede indicar diferentes niveles de optimización: –O1 (o simplemente –O), -O2, -O3 siendo este último el nivel de mayor optimización.

La línea de comando puede quedar de la siguiente manera:

```
$ g++ –Wall –O2 –o miprograma miprograma.cpp
```

Cuando se exige que g++ realice optimizaciones, el tiempo de compilación es mayor. Por lo tanto, generalmente se prefiere desarrollar el programa sin optimizaciones y realizar éstas cuando se tiene el proyecto terminado.

Una desventaja importante de trabajar con la opción de optimización, es que las tareas de corrección de errores a través de depuradores (debuggers) podrían ser más complicadas. Esto es debido a que el

compilador puede eliminar o modificar parte del código para lograr mejor performance, entonces seguir con el debugger la traza del programa será dificultoso. Este es otro motivo para evitar optimización en tiempo de desarrollo y más aun cuando se están realizando depuraciones al proyecto.

Es importante tener en cuenta, que si bien el optimizador puede lograr resultados muy buenos, un buen programador puede lograr resultados mejores.

Código ejecutable que permite depuración

Otra de las características sobresalientes de los compiladores modernos es la posibilidad de generar símbolos internos en el programa ejecutable que permite utilizar herramientas de búsqueda de errores (debuggers). Estos depuradores le permiten al programador seguir la ejecución del programa brindando importante información. Una herramienta ampliamente utilizada es GNU Debugger (gdb). En este capítulo no se tratará el uso de depuradores pero si la forma de compilar los programas para que permitan ser analizados con éstos. Existen en Internet tutoriales y documentos de introducción al uso del gdb.

Antes de utilizar estos depuradores como gdb, es necesario realizar compilaciones que inserten una información extra en los archivos objeto (".o") y los archivos ejecutables. Esta información extra permite determinar la relación entre el código compilado y las líneas del código fuente. Sin esta información, gdb no puede determinar con que línea del código fuente se relaciona la ejecución.

Esta característica no es realizada por defecto porque se incrementa notablemente el tamaño del ejecutable. Por este motivo conviene usar esta opción sólo durante la etapa de desarrollo y evitarla en el programa final. Tener en cuenta que la generación de información para la utilización de debuggers es incompatible con la solicitación de optimización a través de la opción –O debido a la modificación o cambio de la codificación escrita en el código fuente.

La línea de comando con la opción de generación de símbolos para la depuración puede ser de la siguiente manera:

```
$ g++ –g –Wall –o miprograma miprograma.cpp
```

También se puede generar más información específica para el depurador gdb de la siguiente manera:

```
$ g++ –ggdb3 –Wall –o miprograma miprograma.cpp
```

El número 3 indica que se realizará una salida con información para el nivel 3 del gdb, que es el mayor nivel de información posible.

El intérprete de compilación: *makefile/make*

Introducción

Al trabajar con grandes proyectos se suele dividir el código en varias unidades, y se necesita alguna forma de compilar automáticamente cada archivo y que se unan todas las partes (junto con todas las bibliotecas y el código de inicio) y se conviertan en un archivo ejecutable. Anteriormente se vio, que para programas simples (sin la necesidad de bibliotecas especiales) se puede compilar mediante la siguiente línea de comando:

```
g++ –o miprograma codigo1.cpp codigo2.cpp
```

donde el código fuente está separado en los dos archivos con extensión cpp.

Aquí el problema es que el compilador compilará primero cada archivo individualmente, sin tener en cuenta si el archivo necesita ser compilado nuevamente o no. Con un proyecto que contiene muchos archivos, es ineficiente recompilar todo si únicamente se ha cambiado un solo archivo.

La solución a este problema es un programa llamado *make*. La utilidad make maneja todos los archivos individuales de un proyecto siguiendo las instrucciones de un archivo de texto llamado *makefile* (sin extensión). Cuando se ejecuta make, este programa sigue las instrucciones incluidas en el makefile para comparar las fechas en los archivos fuente con las fechas en los archivos a compilar, y si la fecha de un archivo fuente es más reciente que la del archivo a compilar, make compila el archivo más reciente. make sólo recompila aquellos archivos que sufrieron cambios, y cualquier otro archivo que haya sufrido alguna modificación por estos cambios.

Dado que make está disponible para todos los compiladores de C++ (y si no lo estuviera, existen makes disponibles gratuitamente que pueden usarse con cualquier compilador), es la herramienta más usada en la mayoría de los proyectos. Sin embargo, los compiladores comerciales (como Borland C++ Builder) también desarrollaron sus propias herramientas para "construir proyectos". Estas herramientas requieren saber cuáles son los archivos incluidos en el proyecto, y determinan la relación entre ellos. Estas herramientas utilizan algo similar a un makefile, generalmente llamado “archivo de proyecto”, pero el entorno de programación es el encargado de mantener y actualizar ese archivo, así que el programador no debe preocuparse por él. La configuración y uso de un archivo de proyecto varía de un programa a otro, así que para usarlo debe buscarse con anterioridad una documentación apropiada.

Estructura de *makefile*

El makefile permite definir variables internas, para de esa forma simplificar los cambios que se realicen con posterioridad. Las variables que pueden definirse son:

- **CXX**: nombre del compilador

g++ para C++ de GNU
gcc para C de GNU

- **OPCIONES**: opciones de compilación

Algunos ejemplos son:
-g permite depurar el ejecutable.
-Wall muestra todas las advertencias (warnings)

- **DIRECTORIOS**: permite indicar los directorios para los includes (extensión “.h”). Los directorios de los includes estándares de C++ ya están configurados. Esto sirve para cuando se quiere poner los includes de alguna biblioteca no estándar que se esté usando.

Ejemplo: -I/directorio/include (donde I es la “i” mayúscula por **i**nclude)

- **BIBLIOTECAS**: para indicar las bibliotecas precompiladas (extensión “.a”). El directorio donde se va a buscar una biblioteca precompilada se indica con *-L/directorio/lib* y la biblioteca se incluye con *-lnombre*. No es necesario especificar los directorios de las bibliotecas de C++ estándar.

Ejemplo: -lm (donde l es la “L” minúscula por **L**ibrary)

Este ejemplo agrega la biblioteca de rutinas matemáticas estándar de C++. El archivo al que se hace alusión se llama “libm.a” y está en /lib. Observar que del nombre de archivo no se coloca la extensión y además no se colocan las letras "lib", por lo tanto, la biblioteca con nombre de archivo “libm.a” se ingresa como directiva de compilación -lm.

Ejemplo: -L/directorio_x/lib –lotra

En este caso agregamos una biblioteca que se llama "libotra.a" que se encuentra en el directorio /directorio_x/lib.

Los archivos makefile son archivos de texto. En general este archivo hay que tenerlo en el mismo directorio donde está el código fuente que se desea compilar. Luego de ejecutar el comando make, este seguirá lo detallado en makefile.

Proyecto simple

A continuación se ejemplifica un archivo makefile para usar en Microsoft Windows. En este ejemplo se indica que se use el compilador g++ para compilar el código que está en "main.cpp" y genere el ejecutable "main.exe" (en Linux simplemente "main"). Se indicaron las opciones de compilación –g y –Wall. También se indica en qué directorio buscar los includes y qué bibliotecas podrá usar el programa. Finalmente, se encuentra la línea de compilación. El carácter # indica que toda la línea es un comentario. No se debe confundir con el significado del uso de // o el uso de # en el código fuente de un programa.

```
#============================================================
# Prototipo Básico de makefile para compilación con GNU C/C++
#============================================================
# Uso en Windows (cygwin)
#============================================================

CXX=g++
OPCIONES=-g -Wall
DIRECTORIOS=-I../util -I/usr/include/GL -I/bin -I/usr/include/w32api -
I/usr/include
BIBLIOTECAS=-L/lib -lglui -lglut32 -lglu32 -lopengl32 -lm

all:
   ${CXX} ${OPCIONES} ${DIRECTORIOS} main.cpp -o main ${BIBLIOTECAS}

# Para usar OpenGL, incluir en el programa las siguientes líneas
# include <GL/glut.h>
# include <glui.h>
```

Importante: la línea siguiente a "all:" deberá ser indentada con un **tab** (no puede utilizarse espacios en blanco).

Las líneas que se refieren a opengl, glut y glui son necesarias para realizar programas de manejo de gráficos con la biblioteca OpenGL.

Para compilar en el sistema operativo Linux, aparecen algunos cambios para el archivo makefile:

```
#============================================================
# Prototipo Básico de makefile para compilación con GNU C/C++
#============================================================
#  Uso en Linux
#============================================================

CXX=g++
OPCIONES=-g -Wall
DIRECTORIOS=-I../util -I/usr/include/GL -I/bin
BIBLIOTECAS=-L/lib -lm -L/usr/lib/ -lGL -lglut -lGLU

all:
   ${CXX} ${OPCIONES} ${DIRECTORIOS} main.cpp -o main ${BIBLIOTECAS}

# Para usar OpenGL, incluir en el programa la siguiente línea
# include <GL/glut.h>
```

Proyecto más complejo

La forma más fácil de tratar con pequeños bloques de código en C++ es colocando todos ellos en un único archivo y compilarlo. Una forma fácil de hacerlo es a través de los *#include*. Sin embargo, cuando el tamaño del código comienza a incrementarse, ésta forma es impráctica. Por ejemplo, es ineficiente recompilar un gran archivo cuando lo que se ha cambiado es sólo una línea de código.

Es aconsejable trabajar con módulos (o funciones y datos contenidos en archivos fuente separados) los que formarán trozos del código unidos por una relación lógica. Estos módulos (o unidades lógicas) pueden ser de un tamaño más manejable para trabajar con un editor de texto. Además, se puede simplificar la coordinación de los programadores de un equipo de desarrollo.

Para ejemplificar este punto, suponer que se tienen tres módulos como parte de un programa: io.cpp, init.cpp y calcular.cpp. El primero tiene el código para manejar la entrada y salida, el segundo la inicialización y el tercero podría suponerse que hace ciertos cálculos. Tener en cuenta que la separación no va a ser relevante para el compilador, pero si para el programador.

Para compilar todo el programa de una manera muy simple se podría usar lo siguiente:

```
g++ -Wall –o miprograma io.cpp init.cpp calcular.cpp
```

Este código compila cada ".cpp" y los coloca a todos en el ejecutable *miprograma*. Para pequeños programas, la codificación anterior es aceptable. Pero si se tiene muchos módulos (los que a su vez pueden ser también muy extensos) el menor cambio significa recompilar a todos ellos.

El próximo paso es separar la compilación en pasos. Para hacer esto, se usa la opción –c de g++. Esta opción le dice al compilador que no intente generar el código final inmediatamente, sino que simplemente genere el archivo ".o" para cada ".cpp".

```
g++ -Wall –c –o io.o io.cpp
g++ -Wall –c –o init.o init.cpp
g++ -Wall –c –o calcular.o calcular.cpp
```

Ahora que se tienen los tres archivos objeto correspondientes a cada código fuente ".cpp", se puede generar el ejecutable de la siguiente manera:

```
g++ -o miprograma io.o init.o calcular.o
```

Vamos a suponer que se ha modificado una línea en el código fuente *init.cpp*. Esto significa que hay que generar nuevamente el código objeto únicamente para este archivo:

```
g++ -Wall –c –o init.o init.cpp
```

y nuevamente generar el código ejecutable:

```
g++ -o miprograma io.o init.o calcular.o
```

Hemos encontrado el beneficio de recompilar sólo un archivo. Si el proyecto contiene ciento de archivos, esta ventaja puede ser mucho más significativa que en el ejemplo anterior. No obstante es tedioso este método de compilar lo necesario. Makefile simplifica el trabajo con el siguiente código:

```
#============================================================
# Prototipo Básico de makefile para compilación con GNU C/C++
#============================================================
#  Uso en Linux
#============================================================
#  Ejemplo más complejo de makefile
#============================================================

OBJS=io.o init.o calcular.o
EJECUTABLE=miprograma

CXX=g++
OPCIONES=-g -Wall

DIRECTORIOS=-I../util -I/bin -I/usr/include
BIBLIOTECAS=-L/lib -lm

# ==== fin de las opciones de configuración ==================

all: $(EJECUTABLE)
$(EJECUTABLE): $(OBJS)
$(CXX) –o $(EJECUTABLE) $(OBJS) ${BIBLIOTECAS}

%.o: %.cpp
        $(CXX) ${OPCIONES} ${DIRECTORIOS} –c –o $@ $<

clean:
        -rm $(OBJS) $(EJECUTABLE) *~

# ==== FIN ===================================================
```

Importante: las líneas indentadas deben ser hechas con un tab al principio (no puede utilizarse espacios en blanco).

Mediante las cuatro primeras líneas se podrá introducir los nombres de los módulos, el nombre del ejecutable que se quiere obtener, las opciones de compilación y el compilador a utilizar. En las líneas siguientes aparece una codificación que no se explicará aquí por exceder los alcances de esta sección.

Se puede probar sus ventajas modificando sólo un archivo y volviendo a ejecutar make. Deberá verse la generación del archivo ".o" de sólo el archivo modificado.

Tipos de datos simples

La utilidad de los programas del tipo *"Hola Mundo"* mostrados en la sección anterior es más que cuestionable. Nótese que se ha tenido que escribir varias líneas de código, se ha compilado, y luego se ejecutó el programa resultante para obtener simplemente una frase en la pantalla como resultado. Todo lo anterior es cierto, pero programar no está limitado sólo a imprimir textos en pantalla. Para avanzar un poco más y ser capaz de escribir programas que realizan tareas útiles se necesita introducir el concepto de *variable*.

Pensar por un momento que hay que retener en nuestra memoria el número 5, luego memorizar también el número 2. Ahora, si se pide sumar 1 al primer número, se debe retener los números 6 (5+1) y 2 en la memoria. Valores que ahora se podrían restar y obtener 4 como resultado.

Todo este proceso anterior es similar a lo que una computadora puede hacer con dos variables. Este mismo proceso puede expresarse en C++ con las siguientes instrucciones:

```
a = 5;
b = 2;
a = a + 1;
resultado = a - b;
```

Obviamente éste es un ejemplo muy simple porque se han usado sólo dos valores enteros pequeños, pero se debe considerar que la computadora puede guardar varios millones de números como éstos al mismo tiempo y realizar sofisticadas operaciones matemáticas con ellos.

Por consiguiente, se puede definir una variable como una porción de memoria donde se puede guardar un valor determinado.

Cada variable necesita un identificador que la distinga de las otras, por ejemplo, en el código anterior los identificadores de las variables fueron *a, b* y *resultado* pero se podría haber llamado a las variables con cualquier nombre que se hubiese querido inventar, siempre que fueran identificadores válidos.

Identificadores

Un identificador válido es una sucesión de una o más letras, dígitos o símbolos de subrayado "_". La longitud de un identificador no está limitada, aunque para algunos compiladores sólo los 32 primeros caracteres de un identificador son significantivos (el resto no es considerado).

Ni espacios ni letras con acentos pueden ser parte de un identificador. Sólo letras, dígitos y caracteres de subrayado son válidos. Además, los identificadores de variables siempre deben empezar con una letra. También pueden empezar con un caracter de subrayado "_", pero esto es normalmente reservado para enlaces externos. En ningún caso pueden empezar con un dígito.

Otra regla que se debe considerar cuando se inventa *identificadores* propios es que ellos no pueden ser palabras reservadas del lenguaje C++ o palabras reservadas específicas del compilador, porque pueden confundirse con éstas. Por ejemplo, las expresiones siguientes son consideradas palabras reservadas según el estándar ANSI-C++ y por consiguiente no deben usarse como identificadores:

asm, auto, bool, break, case, catch, char, class, const, const_cast, continue, default, delete, do, double, dynamic_cast, else, enum, explicit, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, operator, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_cast, struct, switch, template, this, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t .

Además, las representaciones alternativas para algunos operadores no deben ser usadas como identificadores porque son palabras reservadas bajo ciertas circunstancias:

and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq, xor, xor_eq

Algunos compiladores pueden incluir palabras reservadas específicas. Por ejemplo, muchos compiladores que generan código de 16 bits (como algunos compiladores para DOS) también incluyen **far**, **huge** y **near** como palabras reservadas.

El lenguaje C++ es "sensible a mayúsculas"; esto significa que un mismo identificador escrito en letras mayúsculas no es equivalente a otro con el mismo nombre pero escrito en letras minúsculas. Así, por ejemplo la variable *RESULTADO* no es la mismo que *resultado*, ni que *Resultado*.

Tipos de datos

Al programar se guardan las variables en la memoria de la computadora, pero la computadora debe saber qué es lo que se quiere guardar en ella, dado que no va a ocupar el mismo espacio en memoria un número entero, una letra o un número con decimales.

La memoria de la computadora está organizada en *bytes*. Un byte es la cantidad mínima de memoria que nosotros podemos manejar. Un byte puede guardar una cantidad relativamente pequeña de datos, normalmente un entero entre 0 y 255. Pero además, la computadora puede manipular tipos de datos más complejos que surgen de agrupar varios bytes, como números muy grandes o números con decimales. A continuación se tiene una lista de los tipos de datos fundamentales que existen en C++, así como el rango de valores que pueden representarse con cada uno de ellos.

Tipos de Datos

Nombre	Bytes*	Descripción	Rango*
char	1	caracter o entero de 8 bits de longitud (8 bits = 1 byte)	con signo: -128 a 127 sin signo: 0 to 255
short	2	entero de 16 bits de longitud. No permite decimales.	con signo: -32768 a 32767 sin signo: 0 a 65535
long	4	entero de 32 bits de longitud.	con signo:-2147483648 a 2147483647 sin signo: 0 a 4294967295
int	*	Entero. Su largo tradicionalmente depende del sistema, para MSDOS es 16 bits de longitud, mientras que en sistemas de 32 bit (como Windows 9x/2000/NT y sistemas que trabajan bajo el modelo protegido en x86) su longitud es de 32 bits de longitud (4 bytes).	Idem a short o long
float	4	número de punto flotante permite tener decimales.	3.4e + / - 38 (7 dígitos)
double	8	número de punto flotante de doble precisión.	1.7e + / - 308 (15 dígitos)
long double	10	número de punto flotante de doble precisión de mayor precisión.	1.2e + / - 4932 (19 dígitos)
bool	1	Valor booleano. Puede tomar uno de los siguientes dos valores: `true` o `false`. No todos los compiladores lo soportan.	`true` o `false`
wchar_t	2	Caracteres extendidos. Se designa como un tipo para guardar caracteres internacionales de un conjunto de caracteres de dos bytes. No todos los compiladores lo soportan.	caracteres extendidos

* Los valores de las columnas 'Bytes' y 'Rango' pueden variar dependiendo del sistema operativo. Los valores incluidos aquí son comúnmente los más aceptados y usados por casi todos los compiladores.

Además de estos tipos de datos fundamentales también existen los punteros y la especificación de tipo de parámetro *void* (vacío, nulo) que se presentará más adelante.

Variables

Declaración

Para usar una variable en C++, primero se debe declararla especificando el tipo de dato que va a contener. La sintaxis para declarar una nueva variable es escribir el tipo de datos (como *int, short, float*...) seguido por un identificador válido. Por ejemplo:

```
int a;
float mi_numero;
```

Son declaraciones válidas de variables. La primera declara una variable de tipo *int* con el identificador **a**. La segunda declara una variable de tipo *float* con el identificador *mi_numero*. Una vez declaradas, las variables **a** y *mi_numero* pueden usarse en el resto de su alcance en el programa.

Si se necesita declarar varias variables del mismo tipo, se puede hacer en la misma línea separando los identificadores con comas. Por ejemplo:

```
int a, b, c;
```

declara tres variables (a, b y c) de tipo int, y tiene exactamente el mismo significado si se escribe:

```
int a;
int b;
int c;
```

los tipos de datos enteros (char, short, long e int) puede ser con signo o sin signo según el rango de números que necesitamos representar. Así, para especificar un tipo de dato entero ponemos la palabra reservada signed o unsigned antes del tipo de dato. Por ejemplo:

```
unsigned short NumeroDeSonidos;
signed int LaCuentaDeMiBalance;
```

Por defecto, si no se especifica signed o unsigned se asumirá que el tipo es signed, por lo tanto, en la segunda declaración se podría haber escrito simplemente:

```
int LaCuentaDeMiBalance;
```

con exactamente el mismo significado y siendo este último el más usual.

La única excepción a esta regla es el tipo char que existe por sí mismo y es considerado un tipo diferente de signed char y unsigned char.

Finalmente, signed y unsigned también puede usarse como tipos de datos simples, significando lo mismo que signed int y unsigned int, respectivamente. Las siguientes dos declaraciones son equivalentes:

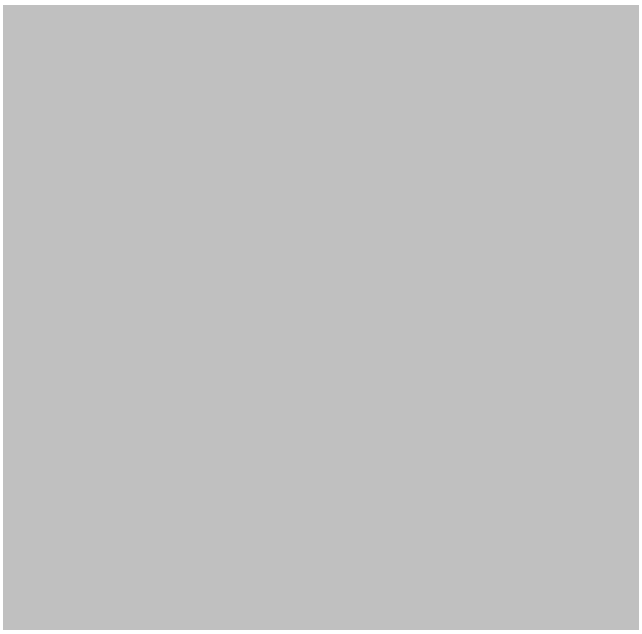
```
unsigned DiaDeMiNacimiento;
unsigned int DiaDeMiNacimiento;
```

Para ver en acción cómo se ve una declaración en un programa, a continuación está el código C++ del ejemplo sobre memoria propuesto al principio de esta sección:

```
// operando con variables

#include <iostream>
```

```
4
```

```
using namespace std;

int main ()
{
  // declarando variables:
  int a, b;
  int resultado;

  // proceso:
  a = 5;
  b = 2;
  a = a + 1;
  resultado = a - b;

  // muestra el resultado:
  cout << resultado;

  // termina el programa:
  return (0);
}
```



Inicialización

Al declarar una variable local, su valor es por defecto indeterminado. Pero se puede que una variable contenga un valor concreto desde el momento en el que se declara. Para hacer eso, se tiene que añadir un signo igual seguido por el valor que se quiere asignar a la variable:

```
tipo identificador = valor_inicial ;
```

Por ejemplo, si se quiere declarar una variable de tipo int llamada a que contenga el valor 0 desde el momento en el que se declara, se podría escribir:

```
int a = 0;
```

Además de esta manera de inicializar variables (conocido como 'la forma C'), C++ ha agregado una nueva manera de inicializar una variable que es adjuntando el valor inicial entre el paréntesis:

```
tipo identificador (valor_inicial) ;
```

Por ejemplo:

```
int a (0);
```

Ambas maneras son válidas y equivalentes en C++.

Constantes

Una constante es cualquier expresión que tiene un valor fijo. Ellos pueden ser divididos en números enteros, números de punto flotante (reales), caracteres y strings (cadenas de caracteres).

Números enteros

```
1776
```

```
707
-273
```

son constantes numéricas que identifican números enteros decimales. Nótese que para expresar una constante numérica no se necesita escribir comillas (") ni ningún caracter especial.

Además de los números en base decimal, C++ permite el uso como constantes literales de números en base octal (base 8) y números en base hexadecimal (base 16). Si se quiere expresar un número en base octal hay que precederlo con un 0 (caracter cero). Y para expresar un número en base hexadecimal hay que precederlo con los caracteres **0x** (cero, equis). Por ejemplo, las siguientes constantes literales son todas equivalente:

```
75          // decimal
0113        // octal
0x4b        // hexadecimal
```

Todas representan el mismo número: 75 (setenta y cinco) expresado en base decimal, octal y hexadecimal, respectivamente.

Nota: se puede encontrar más información sobre representaciones hexadecimal y octal en el ítem Bases Numéricas.

Números de punto flotante (reales)

Expresan números con decimales y/o exponente. Pueden incluir un punto decimal, un caracter **e** (que expresa "por diez a la Xth potencia", donde X es el valor del exponente) o ambos.

```
3.14159     // 3.14159
6.02e23     // 6.02 x 10 elevado a la 23
1.6e-19     // 1.6 x 10 elevado a la -19
3.0         // 3.0
exp(1)      // 2.71828182…
```

éstos son cuatro números válidos con decimales expresados en C++. El primer número es PI, el segundo es el número de Avogadro (donde "e" se refiere a "exponente" de base 10). El tercero es la carga eléctrica de un electrón (un número sumamente pequeño). El cuarto es el número 3 expresado como punto flotante. El último número se utiliza en los cálculos con logaritmos naturales, pero se necesita declarar la biblioteca <math.h>.

Caracteres y cadenas de caracteres

También existen constantes no-numéricas, como:

```
'z'
'p'
"Hola Mundo!"
"¿Cómo estás?"
```

Las primeras dos expresiones representan caracteres simples, y las siguientes dos representan cadenas de varios caracteres. Observe que para representar un solo caracter se debe encerrarlo entre comillas simples (') y para expresar una cadena de más de un caracter se debe encerrarla entre comillas dobles (").

Al escribir caracteres y cadenas de caracteres como constantes, es necesario poner las comillas para distinguirlos de posibles identificadores de variables o de palabras reservadas. Por ejemplo:

```
x
'x'
```

x se refiere a la variable **x**, mientras que '**x**' se refiere al caracter constante '**x**'.

Los caracteres constantes y las cadenas de caracteres constantes tienen ciertas peculiaridades, como los códigos de escape. Éstos son caracteres especiales que no pueden expresarse de otra forma en el código fuente de un programa, como el caracter de nueva línea (\n) o el de tabulación (\t). Todos ellos son precedidos por una barra invertida (\). A continuación se tiene una lista de códigos de escape:

\n	nueva línea
\r	retorno de carro
\t	tabulación
\v	tabulación vertical
\b	retroceso
\f	alimentación de página
\a	alarma (beep)
\'	comilla simple (')
\"	comilla doble (")
\\	barra invertida (\)

Por ejemplo:

```
'\n'
'\t'
"Izquierda \t Derecha"
"uno\ndos\ntres"
```

Además, se puede expresar cualquier caracter por su código ASCII numérico escribiendo una barra invertida (\) seguida por el código ASCII expresado como un número en base octal o hexadecimal. En el primer caso (octal) el número debe seguir inmediatamente la barra invertida (por ejemplo \23 o \40); en el segundo caso (hexadecimal), se debe poner una x antes del número (por ejemplo \x20 o \x4A).

Nota: consultar el ítem Código ASCII para más información sobre este tipo de código de escape.

Las cadenas de caracteres pueden extenderse por más de una sola línea del código si cada línea termina con una barra invertida (\):

```
"cadena de caracteres expresada en \
dos lineas"
```

También se puede encadenar varias cadena de caracteres constantes separándolas por uno o varios espacios en blanco, tabulaciones, caracteres de nueva línea o cualquier otro caracter en blanco válido:

```
"formamos" "una simple" "cadena" "de caracteres"
```

Constantes declaradas

Con el prefijo *const* se puede declarar constantes de un tipo de dato específico como si fuera una variable:

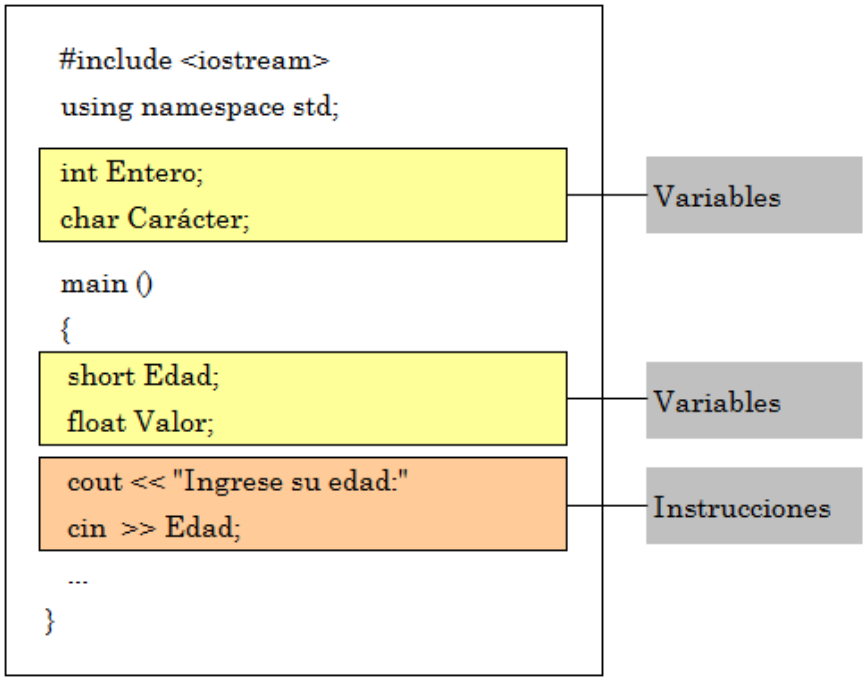
```
const int ancho = 100;
const char tab = '\t'; // si bien hay dos caracteres, es un sólo char.
const codigo_postal = 12440;
```

En caso de que el tipo no fuera especificado (como en el último ejemplo) el compilador asume que es de tipo int (pero se recomienda siempre declara el tipo).

Ámbito y namespaces

Ámbito

Pueden hacerse llamados a variables *globales* en cualquier parte en el código, dentro de cualquier función, siempre que estén después de su declaración.



El alcance de las variables *locales* se limita al nivel del código en el que se declaran. Si se declaran al principio de una función (como en main) su alcance es la función main entera hasta la próxima llave de cierre. Esto significa que si en el ejemplo de arriba, además de la función main existiera otra función, **no** podrían usarse las variables locales declaradas en main en la otra función y viceversa.

En C++, el alcance de una variable local está dado por el bloque en el que se declara (un bloque es un grupo de instrucciones agrupado dentro de los corchetes { }). Si se declara dentro de una función será una variable con alcance de función, si se declara en un ciclo su alcance será sólo el ciclo, etc.

Además de los alcances locales y globales, existe el alcance *externo*, que causa que una variable no sólo sea visible en el mismo archivo fuente sino también en todos los otros archivos adjuntos al programa.

Namespaces

Los namespaces (su traducción podría ser: alcances) permiten agrupar un conjunto de clases globales, objetos y/o funciones bajo un nombre. Para decirlo de alguna forma, sirven para separar el alcance global en sub-alcances conocidos como namespaces.

La forma de usar namespaces es:

```
namespace identificador
{
  ...
}
```

donde *identificador* es cualquier identificador válido y dentro de las llaves se encuentra el conjunto de clases, objetos y funciones incluidas dentro del namespace. Por ejemplo:

```
namespace general
{
  int a, b;
}
```

En este caso, **a** y **b** son variables normales integradas dentro del namespace *general*. Para acceder a estas variables desde fuera del namespace se tiene que hacer uso del *operador de alcance::*. Por ejemplo, para acceder a las variables previas se debería escribir:

```
general::a
general::b
```

La funcionalidad de los namespaces es especialmente útil en caso de que exista la posibilidad de que dos objetos o funciones globales puedan tener el mismo nombre, causando un error de redefinición. Esto puede suceder en grandes proyectos, que poseen varios módulos, los que quizás son realizados por varios programadores que definen cientos de variables distintas. El uso de namespace permite que cada programador tenga libertad en la elección de los identificadores de variables. Si dos programadores utilizan el identificador "edad", no habrá confusión entre las variables que representan.

A continuación se muestra un ejemplo de uso de namespace:

```
// namespaces
#include <iostream>

namespace primero
{
  int var = 5;
}

namespace segundo
{
  double var = 3.1416;
}

int main () {
  cout << primero::var << endl;
  cout << segundo::var << endl;
  return (0);
}
```

```
5
3.1416
```

En este caso existen dos variables globales con el nombre var, una definida en el namespace *primero* y otra en *segundo*. No existe ningún error de redefinición gracias a los namespaces.

Directiva *using* namespace

La directiva *using* seguida por namespace sirve para asociar el nivel de anidamiento actual con un cierto namespace ya definido y así todos los objetos y funciones de dicho namespace sean accesibles directamente como si hubieran sido definidas en el alcance global. Su prototipo es:

```
using namespace identificador;
```

Así, por ejemplo:

```
// ejemplo de uso de namespaces
#include <iostream>

namespace primero
{
  int var = 5;
}

namespace segundo
{
  double var = 3.1416;
}

int main () {
  using namespace segundo;
  cout << var << endl;
  cout << (var*2) << endl;
  return (0);
}
```

```
3.1416
6.2832
```

En este caso se ha usado *var* sin tener que precederlo con ningún operador de alcance.

La instrucción *using namespace* tiene validez sólo en el bloque en el cual es declarada (entendiendo como bloque el grupo de instrucciones entre llaves) o en todo el código si es utilizada en el alcance global. Así, por ejemplo, si se quiere usar primero objetos de un namespace y luego otros de un namespace distinto se podría hacer lo siguiente:

```
#include <iostream>

namespace primero
{
  int var = 5;
}

namespace segundo
{
  double var = 3.1416;
}

int main () {
  {
    using namespace primero;
    cout << var << endl;
  }
  {
    using namespace segundo;
    cout << var << endl;
```

```
5
3.1416
```

```
  }
  return (0);
}
```

Nuevamente aquí, el *namespace primero* puede ser un conjunto de variables, funciones, etc. desarrolladas por un programador y el *segundo* por otro.

Definiciones de *alias*

Existe la posibilidad de definir nombres alternativos para namespaces que ya existen. La forma de hacerlo es:

```
namespace nuevo_nombre = nombre_actual;
```

Namespace *std*

Uno de los mejores ejemplos que se puede encontrar acerca de namespaces es la biblioteca estándar C++ en sí. De acuerdo al estándar ANSI C++, todas las definiciones de clases, objetos y funciones de la biblioteca estándar C++ están dentro de namespace *std*.

Casi todos los compiladores, inclusive aquellos que cumplen con el estándar ANSI, permiten el uso de los archivos de encabezamiento tradicionales (como *iostream.h, stdlib.h*, etc). Sin embargo, el último estándar ANSI ha rediseñado completamente estas bibliotecas tomando ventaja de características como las plantillas y siguiendo la regla de declarar todas las funciones y variables bajo el namespace *std*.

El estándar especificó nuevos nombres para estos archivos de encabezamiento, básicamenete usando el mismo nombre para los archivos específicos de C++, pero sin la terminación *.h*. Por ejemplo, *iostream.h* se convierte en *iostream*.

Si se usa la norma ANSI-C++ para la inclusión de archivos se debe tener en cuenta que todas las funciones, clases y objetos serán declarados bajo el namespace *std*. Por ejemplo:

```
// cumpliendo el ANSI-C++ para el "hola mundo"
#include <iostream>

int main () {
  std::cout << "Hola Mundo en ANSI-C++\n";
  return (0);
}
```

```
Hola Mundo en ANSI-C++
```

Es más usual utilizar la sentencia *using namespace* y ahorrarse de tener que usar el operador de alcance **::** antes de todas las referencias a objetos estándar:

```
// cumpliendo el ANSI-C++ para el "hola mundo"
#include <iostream>
using namespace std;

int main () {
  cout << "Hola Mundo en ANSI-C++\n";
  return (0);
}
```

```
Hola Mundo en ANSI-C++
```

Importante: en las versiones más actuales de los compiladores, es obligatorio poner la instrucción *using namespace std*.

Comunicación a través de la consola

La *consola* es la forma más básica para comunicarse con las computadoras, normalmente es el conjunto compuesto por el teclado y la pantalla. El teclado es el dispositivo de *entrada* estándar y la pantalla el dispositivo de *salida* estándar.

En la biblioteca *iostream* de C++ existen los flujos de datos para las operaciones estándares de entrada y salida de un programa *cin* para la entrada y *cout* para la salida.

Nota: además, también se han implementado *cerr* y *clog*. Estos son dos flujos de salida diseñados especialmente para mostrar mensajes de error. Desde el sistema operativo pueden ser redirigidos especialmente a la salida standard o a un archivo.

Por consiguiente *cout* (el flujo de salida estándard) normalmente se dirige a la pantalla y *cin* (el flujo de entrada estándar) normalmente se asigna al teclado.

Manejando estos dos flujos se podrá interactuar con el usuario de los programas, dado que se podrán mostrar mensajes en la pantalla e ingresar peticiones desde el teclado.

Salida (*cout*)

El flujo cout se usa junto con el operador **<<** (un par de signos "menor que").

```
cout << "Salida";   // muestra la palabra 'Salida' en la pantalla
cout << 120;        // muestra el número 120 en la pantalla
cout << x;          // muestra el contenido de la variable x en la pantalla
```

El operador **<<** es conocido como *operador de inserción*, dado que inserta los datos que lo siguen en el flujo que lo precede (en este caso lo precede *cout* que por defecto es la pantalla de la computadora). En los ejemplos anteriores, insertó la cadena de caracteres *Salida*, la constante numérica 120 y la variable x en el flujo de salida cout. Se debe observar que la primera de las dos frases va entre comillas dobles (") porque es una cadena de caracteres.

El operador de inserción (<<) puede usarse más de una vez en una misma instrucción:

```
cout << "Hola, " << "soy " << "una oración";
```

esta última frase imprimiría el mensaje: *Hola, soy una oración* en la pantalla. La utilidad de repetir el operador de inserción se demuestra cuando se quiere imprimir una combinación de variables y constantes o más de una variable:

```
cout << "Hola, tengo " << edad << " años y mi código postal es " << codigo;
```

Si se supone que la variable *edad* contiene el número 24 y la variable código contiene el número 4190, la salida de la frase anterior sería:

```
 Hola, tengo 24 años y mi código postal es 4190
```

Es importante notar que *cout* no agrega saltos de línea después de la salida a menos que se indique explícitamente, por consiguiente, las frases siguientes:

```
cout << "Esto es una oración.";
```

```
cout << "Esto es otra oración.";
```

se mostrarán seguidas en la pantalla:

```
Esto es una oración.Esto es otra oración.
```

aun cuando se las escriba con dos diferentes **cout**. Así que para realizar un salto de línea en la salida, se debe explícitamente ordenarlo insertando un caracter de nueva-línea, que en C++ puede escribirse como **endl**:

```
cout << "Primera oración." << endl;
cout << "Segunda oración. "<< endl <<"Tercera oración.";
```

produce la siguiente salida:

```
Primera oración.
Segunda oración.
Tercera oración.
```

El manipulador *endl* tiene una conducta especial cuando se usa con flujos *buffered streams* (flujos que utilizan memoria temporal), dado que la vacían. Pero, sin embargo, cout es por defecto *'unbuffered'* (no utiliza memoria temporal).

En vez de endl se puede usar el caracter especial \n de la siguiente manera, pero no se recomienda su uso:

```
cout << "Primera oración.\n";
cout << "Segunda oración. \nTercera oración.";
```

Entrada (*cin*)

Se puede manejar la entrada estándar en C++ aplicando el operador de extracción (>>) en el flujo cin. Éste operador debe ir seguido por la variable que guardará los datos que van a ser leídos. Por ejemplo:

```
int edad;
cin >> edad;
```

declara la variable *edad* como un entero y luego espera por una entrada desde *cin* (teclado por defecto) para guardarlo en esta variable entera.

cin sólo puede procesar la entrada una vez que la tecla *ENTER* se ha presionado. Aun si se solicita sólo un caracter, cin no procesará la entrada hasta que el usuario pulse *ENTER* una vez que el caracter se ha introducido.

Siempre se debe considerar cuál es el *tipo* de variable que se está usando para extraer el valor desde cin. Si se pide un entero hay que introducir un entero, si se pide un caracter se debe dar un caracter y si pide una una cadena se deberá introducir una cadena de caracteres.

```
// ejemplos de entrada/salida
#include <iostream>
using namespace std;

int main ()
{
  int i;
```

```
Ingrese un valor entero:
702
El valor ingresado es 702 y
el doble de este valor es
1404.
```

```
  cout << "Ingrese un valor entero: ";
  cin >> i;
  cout << "El valor ingresado es " << i;
  cout << " y el doble de este valor es "
       << i*2 << endl;
  cin.get();
  return (0);
}
```

El usuario de un programa puede ser el principal generador de errores en los programas más simples que usan cin. Si se pide un valor entero y el usuario introduce su nombre (que es una cadena de caracteres) el resultado puede hacer que el programa no funcione, dado que no es lo que se estaba esperando del usuario. Así que cuando se usen los datos de entrada proporcionados por cin se tendrá que confiar que el usuario del programa será totalmente cooperativo y que no introducirá su nombre cuando un valor entero sea solicitado. Más adelante, cuando se vea cómo usar cadenas de caracteres, se verán posibles soluciones para los errores que pueden causar este tipo de entradas del usuario.

Nota: para ciertos casos es necesario agregar *cin.get()* para poder visualizar los últimos datos mostrados por el programa antes de que termine.

También se puede usar cin para pedir más de una entrada de datos del usuario:

```
cin >> a >> b;
```

es equivalente a:

```
cin >> a;
cin >> b;
```

En ambos casos el usuario debe ingresar dos datos, uno para la variable **a** y otro para la variable *b* que pueden ser separados por cualquier separador válido: un espacio, un caracter de tabulación o un caracter de nueva línea (tecla *enter*).

Teniendo en cuenta que un espacio es un separador válido de datos, hay que tener cuidado si se solicita, por ejemplo, ingresar el nombre y apellido de una persona para ser cargado en una variable. Para este caso se deberá usar *getline*. Por ejemplo:

```
string s;
getline (cin,s);
```

por defecto carga caracteres hasta que se pulsa la tecla *enter*.

Operadores

Una vez que se conoce la existencia de variables y constantes, se puede empezar a operar con ellos. Para este propósito, C++ proporciona los operadores, que en este lenguaje son un conjunto de palabras reservadas y signos que no son parte del alfabeto pero están disponibles en todos los teclados. Es importante conocerlos porque son la base del lenguaje C++.

Asignación

El operador de asignación (=) sirve para asignar un valor a una variable.

```
a = 5;
```

asigna el valor entero *5* a la variable *a*. La parte a la izquierda del operador ' = ' es conocido como *lvalue* (valor izquierdo) y la parte a la derecha como *rvalue* (valor derecho). El lvalue siempre debe ser una variable, mientras que el lado derecho puede ser una constante, una variable, el resultado de una operación o cualquier combinación de ellos.

Es necesario enfatizar que la operación de asignación siempre tiene lugar de derecha a izquierda y nunca a la inversa.

```
a = b;
```

asigna a la variable *a* (lvalue) el valor que contiene la variable *b* (rvalue) independientemente del valor que estaba guardado en **a** en ese momento. También considerar que se está asignando sólo el valor de *b* a **a** y que un cambio posterior de *b* no afectaría el nuevo valor de *a*.

Por ejemplo, si se toma el siguiente código (con la evolución del contenido de las variables en el comentario de la derecha):

```
int a, b;     // a:?  b:?
a = 10;       // a:10 b:?
b = 4;        // a:10 b:4
a = b;        // a:4 b:4
b = 7;        // a:4 b:7
```

nos dará como resultado que el valor contenido en **a** es 4 y el valor contenido en *b* es 7. La modificación final de *b* no afecta a *a*, aunque antes se haya declarado que *a* = *b*; (regla de-derecha-a-izquierda).

Una propiedad que posee la operación de asignación en C++ con respecto a otros lenguajes de programación es que la operación de asignación puede usarse como rvalue (o parte de un rvalue) para otra asignación. Por ejemplo:

```
a = 2 + (b = 5);
```

es equivalente a:

```
b = 5;
a = 2 + b;
```

esto significa que primero se asigna 5 a la variable *b* y luego se asigna a *a* el valor 2 más el resultado de la asignación anterior de b (ésto es, 5), quedando a con un valor final de 7. La expresión siguiente también es válida en C++:

```
a = b = c = 5;
```

asigna 5 a las tres variables *a, b* y *c*.

Operadores aritméticos

Las cinco operaciones aritméticas soportadas por el lenguaje son:

+	adición
-	sustracción
*	multiplicación
/	división
%	módulo

Las operaciones de suma, resta, multiplicación y división se corresponden literalmente con sus respectivos operadores matemáticos.

El único que puede ser desconocido es el operador *módulo*, especificado con el signo de porcentaje (%). El módulo es la operación que devuelve el resto de una división entre dos valores enteros. Por ejemplo, si se escribe *a = 11 % 3;*, la variable **a** contendrá 2 como resultado, dado que 2 es el resto de dividir 11 entre 3.

Operadores de asignación compuestos

Comprende a los siguientes operadores: *(+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)*

Una característica de la asignación en C++, que contribuye a su fama de lenguaje económico al escribir, son los operadores de asignación compuestos (+=, -=, *= y /= entre otros), que permiten modificar el valor de una variable con uno de los operadores básicos:

`valor += incremento;` es equivalente a: `valor = valor + incremento;`
`a -= 5;` es equivalente a: `a = a - 5;`
`a /= b;` es equivalente a: `a = a / b;`
`precio *= unidades + 1;` es equivalente a: `precio = precio * (unidades + 1);`

y en forma similar para todos los otros operadores.

Incremento y decremento

Otro ejemplo de economía al escribir el código de un programa son el operador de incremento (++) y el operador de decremento (--). Ellos incrementan o decrementan en 1 el valor guardado en una variable, y son equivalentes a *+=1* y *a -=1*, respectivamente. Así:

```
a++;
a+=1;
a=a+1;
```

son equivalente en su funcionalidad: los tres incrementan en 1 el valor de **a**.

Una característica de este operador es que puede ser usado tanto como prefijo o como sufijo. Esto significa que puede ser escrito tanto antes del identificador de variable (++a) o después (a++) y, aunque en expresiones simples como a++ o ++a tienen exactamente el mismo significado, en otras operaciones en las cuales el resultado del incremento o decremento es evaluado en otra expresión puede haber una diferencia importante en su significado: en caso de que el operador de incremento sea usado como prefijo

(++a) el valor es incrementado antes de que la expresión sea evaluada y por ende es el valor incrementado el que se considera en la expresión; en caso de que sea usado como sufijo (a++) el valor guardado en a es incrementado después de ser evaluado y por consiguientes el valor almacenado antes de incrementarse el que se utiliza al evaluar la expresión. Nótese la diferencia:

Ejemplo 1	Ejemplo 2

`B=3;`
`A=++B;`
`// A tiene 4, B tiene 4`
	`B=3;`
	`A=B++;`
	`// A tiene 3, B tiene 4`

En el ejemplo 1, *B* es incrementado antes de que su valor se copie a *A*, mientras que en el ejemplo 2, el valor de *B* es copiado a *A* y *B* es incrementado posteriormente.

Operadores relacionales (`==`, `!=`, `>`, `<`, `>=`, `<=`)

Para evaluar una comparación entre dos expresiones se puede utilizar los operadores relacionales. Como se especifica en el estándar ANSI-C++, el resultado de una operación relacional es un valor booleano (*bool*) que puede ser únicamente verdadero (*true*) o falso (*false*) de acuerdo al resultado de la comparación.

Se podría necesitar comparar dos expresiones, por ejemplo, para saber si son iguales o una es mayor que la otra. Aquí hay una lista de los operadores relacionales que pueden ser utilizados en C++:

==	Igualdad
!=	Diferencia
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que

Ejemplos de uso:

(7 == 5)	devuelve false.
(5 > 4)	devuelve true.
(3 != 2)	devuelve true.
(6 >= 6)	devuelve true.
(5 < 5)	devuelve false.

por supuesto, en vez de usar sólo constantes numéricas, se puede utilizar cualquier expresión válida, incluyendo variables. Suponga que a=2, b=3 y c=6:

(a == 5)	devuelve false.
(a*b >= c)	devuelve true porque (2*3 >= 6).
(b+4 > a*c)	devuelve false porque (3+4 > 2*6).
((b=2) == a)	devuelve true.

El operador = (un signo igual) no es lo mismo que el operador == (dos signos igual); el primero es un operador de asignación (asigna el lado derecho de la expresión a la variable en el lado izquierdo) y el otro (==) es un operador relacional de igualdad que compara si ambas expresiones a los lados del operador son iguales. Por consiguiente, en la última expresión ((b=2) == a), primero se asigna el valor *2* a *b* y luego se compara con *a*, que también almacena el valor *2*, así que el resultado de la operación es *true*.

En muchos compiladores previos a la publicación del estándar ANSI-C++, así como en el lenguaje C, los operadores relacionales no retornaban un valor *bool* (true o false), sino que retornaban un entero (int) como resultado con un valor de 0 para representar false y un valor diferente de 0 (usualmente 1) para representar true.

Operadores lógicos

Comprende a los operadores: `!, &&, ||`

El operador ! es equivalente a la operación booleana NOT, tiene un solo operando, localizado a la derecha, y la única cosa que hace es invertir su valor, produciendo false si su operando es true y true si su operando es false. Puede decirse que devuelve el resultado opuesto al obtenido al evaluar su operando. Por ejemplo:

!(5 == 5)	retorna false porque la expresión a la derecha (5 == 5) es true.
!(6 <= 4)	retorna true porque(6 <= 4) es false.
!true	retorna false.
!false	retorna true.

Los operadores lógicos **&&** y **||** son usados al evaluar dos expresiones para obtener un solo resultado. Se corresponden con las operaciones booleanas *AND* y *OR* respectivamente. Su resultado depende de la relación entre sus dos operandos:

Primer Operando a	Segundo Operando b	Resultado a && b	Resultado a || b
True	true	true	true
True	false	false	true
False	true	false	true
False	false	false	false

Por ejemplo:

((5 == 5) && (3 > 6)) retorna false (true && false).
((5 == 5) || (3 > 6)) retorna true (true || false).

Operador condicional (?)

El operador condicional evalúa una expresión y retorna un valor distinto de acuerdo a la expresión evaluada, dependiendo si es *verdadera* o *falsa*. Su formato es:

```
condición ? resultado_1: resultado_2
```

si condición es true la expresión retornará resultado_1, sino retornará resultado_2.

`7==5 ? 4 : 3`	retorna 3 porque 7 no es igual a 5.
`7==5+2 ? 4 : 3`	retorna 4 porque 7 es igual a 5+2.
`5>3 ? a : b`	retorna `a`, porque 5 es mayor que 3.
`a>b ? a : b`	retorna el mayor de `a` y `b`.

Operadores de bits

Los operadores de bits modifican las variables considerando los bits que representan el valor que almacenan, o sea, su representación binaria.

operador		Descripción
&	AND	AND lógico
|	OR	OR lógico
^	XOR	OR lógico exclusivo
~	NOT	Complemento a uno o negación (inversión de bits)
<<	SHL	Desplazamiento a la izquierda
>>	SHR	Desplazamiento a la derecha

Operadores de conversión explícita de tipos

Los operadores de conversión de tipos permiten convertir un tipo de datos dado en otro tipo de datos distinto. Hay varias maneras de hacer esto en C++, pero la más popular compatible con el lenguaje C es preceder la expresión que se desea convertir por el nuevo tipo encerrado entre paréntesis ( ):

```
int i;
float f = 3.14;
i = (int) f;
```

El código anterior convierte el número de punto flotante 3.14 a un valor entero (3). En este caso, el operador fue (int). Otra manera de hacer lo mismo en C++ es usando el constructor de forma: preceder la expresión que va a ser convertida con el nuevo tipo y encerrar entre paréntesis la expresión:

```
i = int ( f );
```

Ambas maneras de conversión de tipos son válidas en C++. Además, ANSI-C++ agregó nuevos operadores de tipo más específicos para la programación orientada a objetos.

Operador sizeof( )

Este operador acepta un parámetro, que puede ser tanto un tipo de variable (int, float, etc.) o la variable en sí y devuelve el tamaño en bytes de ese tipo u objeto:

```
a = sizeof (char);
```

Esta expresión dará el valor *1* a *a* porque *char* es un tipo de dato de un byte de longitud. El valor que devuelve *sizeof()* es una constante, así que siempre es determinado antes de la ejecución del programa.

Operador typeid

ANSI-C++ también define un nuevo operador llamado *typeid* que permite verificar el tipo de una expresión:

```
typeid (expresion)
```

este operador devuelve una referencia a un objeto constante de tipo *type_info* que está definido en el archivo de encabezamiento estándard <*typeinfo*>. Este valor retornado puede ser comparado con otro

usando los operadores == y != o puede servir para obtener una cadena de caracteres que representa el tipo de dato o nombre de la clase usando su función *name()*.

```
// typeid, typeinfo
#include <iostream>
#include <typeinfo>
using namespace std;

class CSolo_a { int a; };
class CSolo_b { int b; };

int main () {
CSolo_a a1,a2;
CSolo_b b1;
int i;

if (typeid(a1) == typeid(a2))
{ cout << "a1 y a2 son de igual tipo: ";
  cout << typeid(a1).name() << endl;
}

if (typeid(a1) != typeid(b1))
{ cout << "a1 y b1 son de diferentes tipos:" << endl;
  cout << "a1 es: " << typeid(a1).name() << endl;
  cout << "b1 es: " << typeid(b1).name() << endl;
}

if (typeid(a1) != typeid(i))
{ cout << "a1 y i son de diferentes tipos:" << endl;
  cout << "a1 es: " << typeid(a1).name() << endl;
  cout << "i  es: " << typeid(i).name() << endl;
}

cin.get();
return (0);
}
```

Operadores de dirección y referencia

Las variables son como celdas de memoria a las cuales se puede acceder por medio de un identificador. Pero estas variables son almacenadas en lugares completos en la memoria de la computadora. Para los programas, la memoria de la computadora es sólo una sucesión de celdas de memoria de 1 byte de longitud (el tamaño mínimo de un dato), cada uno con una dirección única.

Una comparación factible con la memoria de la computadora puede ser una calle en una ciudad. En una calle todas las casa están numeradas consecutivamente con un identificador único que nos permite decir la dirección "Belgrano Nº 50" y se podrá encontrar ese lugar, dado que debe haber sólo una casa con ese número.

De la misma forma, el sistema operativo organiza la memoria con números únicos y consecutivos, así que si se habla de la dirección 1776, se sabrá que sólo hay una dirección con ese número, que además se encuentra entre las direcciones 1775 y 1777.

Operador de dirección (&)

En el momento en que se declara una variable, ésta debe ser almacenada en un lugar concreto en esta sucesión de celdas (la memoria). Generalmente el usuario o programador no decide donde será ubicada la

variable porque afortunadamente eso es algo que realizan automáticamente el compilador y el sistema operativo en tiempo de ejecución. Pero una vez que el sistema operativo le ha asignado una dirección, hay casos en los que se podría estar interesado en saber dónde ha sido almacenada la variable.

Esto puede hacerse precediendo el identificador de la variable con el signo *ampersand (&)*, que se lee como "la dirección de". Por ejemplo:

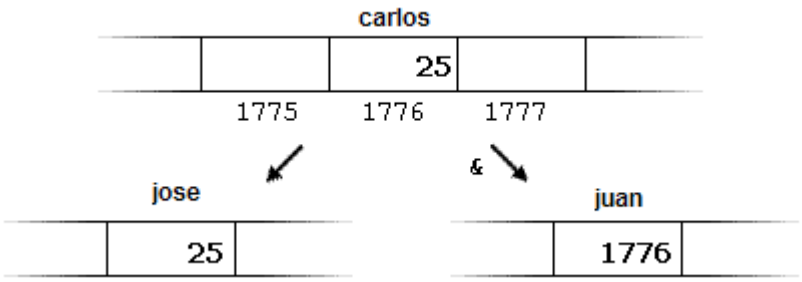
```
juan = &carlos;
```

asignaría a la variable *juan* la dirección de la variable *carlos*, porque al preceder el nombre de la variable carlos con el caracter ampersand (&) no se hace referencia al contenido de la variable sino a su dirección en memoria.

Si carlos hubiese sido ubicado en la dirección de memoria 1776 y luego se escribe lo siguiente:

```
carlos = 25;
jose = carlos;
juan = &carlos;
```

el resultado será el expuesto en el siguiente diagrama:



Se ha asignado a jose el contenido de la variable carlos, pero a juan se le asignado la dirección de memoria donde el sistema operativo almacenó el valor de carlos, que se podría imaginar que es 1776 (puede ser cualquier dirección, la anterior ha sido elegida arbitrariamente). Esto se debe a que en la asignación de juan se precedió a carlos con el caracter ampersand (&).

Por lo tanto, operador de dirección o derreferencia (&) se usa como prefijo de variables y puede ser traducido como *"la dirección de memoria de"*, así: *&variable1* se puede traducir como "la dirección de memoria de variable1"

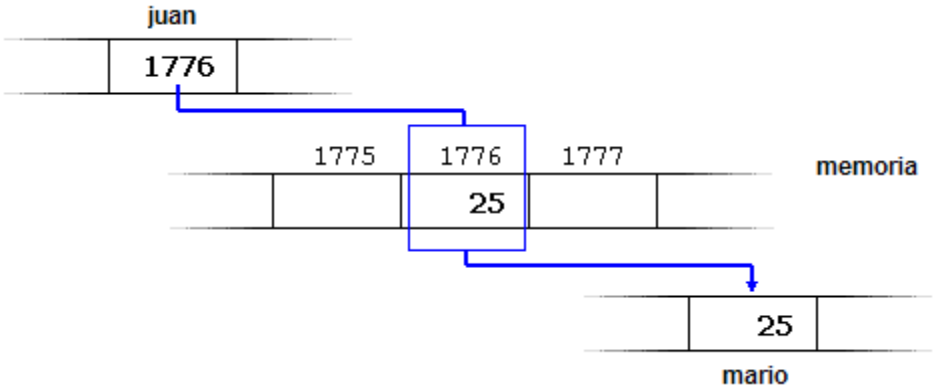
La variable que almacena la dirección de otra variable (como juan en el ejemplo previo) es lo que en C++ se denomina puntero. En C++ los punteros tienen varios usos pero cada vez se los está empleando con menos frecuencia. Más adelante se verá cómo se declara este tipo de variable.

Operador de referencia (*)

Usando un puntero se puede acceder directamente al valor almacenado en la variable apuntada (a la cual el puntero apunta) simplemente precediendo el identificador del puntero con el operador de referencia *asterisco* (*****), que puede ser traducido literalmente como *"valor apuntado por"*. Así, siguiendo con los valores del ejemplo previo, si se escribe:

```
mario = *juan;
```

(que se lee: "mario es igual al valor apuntado por juan") mario tomaría el valor 25, dado que juan contiene la dirección 1776, y el valor apuntado por 1776 es 25.



Se debe diferenciar claramente que juan almacena la dirección de memoria 1776, pero *juan (con un asterisco * antes) se refiere al valor almacenado en la dirección de memoria 1776, que es 25. Hay que notar la diferencia entre incluir o no incluir el asterisco de referencia. A continuación se encuentran comentarios explicatorios sobre cómo debe leerse cada expresión:

```
mario = juan;   // mario es igual a juan ( dirección 1776 )
mario = *juan;  // mario es igual al valor apuntado por juan ( 25 )
```

Por lo tanto, operador de referencia (*) indica que lo que debe ser evaluado es el contenido apuntado por la expresión considerada como una dirección. Puede ser traducido como "valor apuntado por". *mipuntero puede ser traducido como "valor apuntado por mipuntero".

En este punto, continuando con el mismo ejemplo donde:

```
carlos = 25;
juan = &carlos;
```

se puede ver que las siguientes expresiones son verdaderas:

```
carlos == 25
&carlos == 1776
juan == 1776
*juan == 25
```

La primera expresión es bastante clara considerando que se ha realizado la asignación carlos = 25;. La segunda expresión usa el operador & que devuelve la dirección de memoria de la variable carlos, que supusimos que era 1771. La tercera es bastante obvia dado que la segunda expresión es verdadera y la asignación de juan fue juan = &carlos;. La cuarta expresión usa el operador de referencia (*) que es equivalente al valor contenido en la dirección de memoria apuntada por juan, que es 25.

Por lo tanto mientras la dirección apuntada por juan permanezca sin cambios la siguiente expresión también dará verdadera:

```
*juan == carlos
```

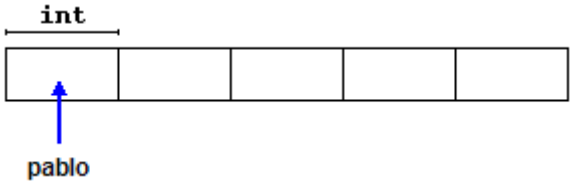
Operador new

Para requerir memoria dinámica existe el operador **new** que va seguido por un tipo de dato y opcionalmente el número de elementos requeridos dentro de corchetes [ ]. Retorna un puntero que apunta al comienzo del nuevo bloque de memoria asignado durante la ejecución del programa. Su forma es:

```
puntero = new tipo
puntero = new tipo [elementos]
```

Por ejemplo:

```
int * uno;
uno = new int;
int * pablo;
pablo = new int [5];
```



En el último caso se reservó espacio para cinco enteros que van a estar contiguos. El primer entero quedó apuntado por la variable pablo (que es del tipo *int* *, es decir, "apunta a un entero").

Operador delete

Una vez que no se precisa más hacer uso de la memoria asignada por new, debería ser liberada quedando disponible para las otras aplicaciones que se están ejecutando.

```
delete puntero;
delete [] puntero;
```

Para el ejemplo anterior sería:

```
delete uno;
delete [] pablo;
```

Memoria dinámica - ejemplo

```
// recordador
#include <iostream>
#include <cstdlib>
using namespace std;

int main () {
char ingreso [100];
int i, n;
long * l, total = 0;
cout << "¿Cuántos números va a ingresar? ";
cin.getline (ingreso,100);
i=atoi (ingreso);
l= new long[i];
if (l == NULL) exit (1);

for (n=0; n<i; n++) {
cout << "Ingrese un número: ";
cin.getline (ingreso,100);
l[n]=atol (ingreso); }
cout << "Usted a ingresado: ";
for (n=0; n<i; n++) cout << l[n] << ", ";
```

```
¿Cuántos números va a ingresar? 5
Ingrese un número: 75
Ingrese un número: 436
Ingrese un número: 1067
Ingrese un número: 8
Ingrese un número: 32
Usted a ingresado: 75, 436, 1067, 8, 32,
```

```
delete[] l; return (0); }
```

Tener en cuenta la implementación del arreglo estático donde en tiempo de ejecución hay que establecer el tamaño

```
int datos[200];
for (j=0; j<200; j++)
  cin >> datos[j];
```

Observar la particularidad del arreglo dinámico:

```
int *datos, tam;
cin >> tam;
datos = new int[tam];
for (j=0; j<tam; j++)
  cin >> datos[j];
```

donde la definición permite el uso de una variable y por lo tanto su tamaño será definido en tiempo de ejecución.

Prioridad de los operadores

Cuando se trabaja con operaciones complejas con varios operandos podemos tener algunas dudas acerca de cuál operador es evaluado primero y cuál es evaluado después. Por ejemplo, en esta expresión:

```
a = 5 + 7 % 2
```

podríamos dudar si significa realmente:

`a = 5 + (7 % 2)` que arroja el 6 como resultado, o `a = (5 + 7) % 2` que da 0

La respuesta correcta es la primera expresión, con un resultado de 6. Existe un orden establecido con la prioridad para cada operador, no sólo para los operadores aritméticos (aquellos cuya precedencia podemos conocer por las matemáticas) sino también para todos los operadores que pueden aparecer en C++. Desde la máxima prioridad hasta la mínima, el orden de precedencia es el siguiente:

Prioridad	Operador	Descripción	Asociatividad
1	::	Alcance	izquierda
2	() [ ] -> . sizeof		izquierda
3	++ --	incremento/decremento	derecha
	~	complemento a uno o negación (bits)	
	!	unario NOT	
	& *	referencia y derreferencia (punteros)	
	(type)	operador de tipos	
	+ -	signo menos unario	
4	* / %	operaciones aritméticas	izquierda
5	+ -	operaciones aritméticas	izquierda
6	<< >>	desplazamiento de bits (bitwise)	izquierda
7	< <= > >=	operadores de relación	izquierda
8	== !=	operadores de relación	izquierda
9	& ^ |	operadores de bits	izquierda

10	&& ||	operadores lógicos	izquierda
11	?:	Condicional	derecha
12	= += -= *= /= %= >>= <<= &= ^= |=	Asignación	derecha
13	,	coma, Separador	izquierda

La asociatividad define -en el caso de que haya varios operadores con el mismo nivel de prioridad- cuál de todos debe ser evaluado primero, si el que está más a la derecha o el que está más a la izquierda. Todos estos niveles de precedencia para los operadores pueden ser manipulados o hacerse más legibles usando paréntesis como en el siguiente ejemplo:

```
a = 5 + 7 % 2;
```

puede ser escrito también:

```
a = 5 + (7 % 2); ó
a = (5 + 7) % 2;
```

de acuerdo a la operación que se quiera realizar.

Al escribir una operación complicada sin estar seguro del nivel de precedencia, es mejor incluir paréntesis. Además será más legible el código.

Estructuras de control

Un programa normalmente no se limita a una sucesión lineal de instrucciones. Durante su ejecución puede bifurcarse, repetir algún código o tomar decisiones. Para esos propósitos, C++ proporciona 'estructuras de control' que sirven para especificar qué y cómo tiene que hacer sus tareas nuestro programa.

Con la introducción de secuencias de control se introduce un nuevo concepto: el *bloque de instrucciones*. Un bloque de instrucciones es un grupo de instrucciones separadas por puntos y comas (;) pero agrupadas en un bloque delimitado por corchetes: { y }.

Si se quiere que la declaración sea una sola instrucción no se necesita encerrarlo entre llaves "{ }". Si se quiere que la declaración sea más de una sola instrucción se deberá encerrar esas instrucciones entre llaves formando un bloque de instrucciones.

Estructuras condicionales: *if* y *else*

Se usa para ejecutar una instrucción o bloque de instrucciones cuando una condición se cumple. Su forma es:

```
if (condición) acción
```

donde *condición* es la expresión que se está evaluando. Si *condición* es verdadera, *acción* se ejecuta. Si es falsa, *acción* se ignora (no se ejecuta) y el programa continúa en la próxima instrucción después de la estructura condicional.

Por ejemplo, el siguiente fragmento de código imprime x es 100 sólo si el valor guardado en la variable x es, de hecho, 100:

```
if (x == 100)
  cout << "x es 100";
```

Si se quiere ejecutar más de una instrucción en caso de que esa condición sea verdadera, se debe especificar un bloque de instrucciones usando corchetes {}:

```
if (x == 100)
{
 cout << "x es ";
 cout << x;
}
```

Se puede especificar además lo que se quiere que suceda si la *condición* no es cumplida usando la palabra clave *else*. Su uso junto con *if* es:

if (*condición***)** *acción_1* **else** *acción_2*

Por ejemplo:

```
if (x == 100)
  cout << "x es 100";
else
  cout << "x no es 100";
```

imprime en la pantalla "x es 100" si de hecho x vale 100, pero si no es así (y sólo si no lo es) imprime "x no es 100".

Las estructuras *if* + *else* pueden encadenarse con la intención de verificar un rango de valores. El ejemplo siguiente muestra su uso al decir si el valor presente guardado en *x* es positivo, negativo o ninguno de los anteriores, en cuyo caso será igual a cero.

```
if (x > 0)
  cout << "x es positivo";
else if (x < 0)
  cout << "x es negativo";
else
  cout << "x es 0";
```

Recordar que en caso de que se quiera que más de una sola instrucción se ejecute se debe agruparlas en un bloque de instrucciones usando corchetes *{}*.

Estructuras repetitivas o ciclos

Las estructuras repetitivas o ciclos (loops) tienen como objetivo repetir una acción un cierto número de veces o mientras una condición se cumpla.

El ciclo *while*

Su formato es:

while (*condición***)** *acción*

y su función simplemente es repetir la *acción* mientras la *condición* es verdadera.

Por ejemplo, el siguiente programa hace una cuenta regresiva usando un ciclo:

```
// cuenta regresiva usando while
#include <iostream>
using namespace std;

int main ()
{
  int n;
  cout << "Ingresar el número de inicio > ";
  cin >> n;
  while (n>0) {
    cout << n << ", ";
    --n;
  }
  cout << "LISTO!";
  return (0);
}
```

```
Ingresar el número de inicio
> 8
8, 7, 6, 5, 4, 3, 2, 1,
LISTO!
```

Cuando el programa empieza se le solicita al usuario insertar un número de arranque para la cuenta regresiva. Entonces, el ciclo *while* comienza, y si el valor ingresado por el usuario cumple la condición n>0 (n sea mayor que 0), el bloque de instrucciones que sigue se ejecutará mientras la condición (n>0) continúe siendo verdadera (true).

Todo el proceso en el programa anterior puede interpretarse según el siguiente orden, comenzando en main():

1. El usuario le asigna un valor a n.
2. La instrucción while verifica la condicion (n>0).
   Hay dos posibilidades a esta altura:
   true: la condición es verdadera. Se ejecuta la acción (paso 3).
   false: la condición es falsa. Se omite la acción. El programa sigue en el paso 5.
3. Se ejecuta la acción:
   cout << n << ", "; (imprime n en la pantalla)
   --n; (decrementa n en 1)
4. Fin del bloque. Se retorna automáticamente al paso 2.
5. Continúa el programa después del bloque: se imprime LISTO! y termina el programa.

Se debe considerar que el ciclo tiene que acabar en algún punto, por consiguiente, dentro del bloque de instrucciones (la acción que realiza el ciclo) se debe proporcionar algún método que obligue a la condición a convertirse en falsa en algún momento, de otra forma el ciclo continuará indefinidamente, es decir que el programa nunca terminará. En este se ha incluido --n; esto causa que la condición se vuelva falsa después de algunas repeticiones del ciclo: cuando n toma el valor cero, es donde la cuenta regresiva termina.

El ciclo *do-while*

Formato:

do *acción* **while (***condición***);**

Su función es exactamente la misma que el ciclo *while* excepto que la *condición* en este ciclo *do while* es evaluada después de la ejecución de la acción y no antes, garantizando al menos una ejecución de la acción inclusive si la condición nunca se cumple. Por ejemplo, el siguiente programa repite cualquier número que se ingrese hasta el ingreso del 0.

```
// repetidor
#include <iostream>
using namespace std;

int main ()
{
  unsigned long n;
  do {
    cout << "Ingrese un número
            (0 para finalizar): ";
    cin >> n;
    cout << "Ud. ha ingresado: "
         << n << endl;
  } while (n != 0);
  return (0);
}
```

```
Ingrese un número (0 para
finalizar): 12345
Ud. ha ingresado: 12345
Ingrese un número (0 para
finalizar): 160277
Ud. ha ingresado: 160277
Ingrese un número (0 para
finalizar): 0
Ud. ha ingresado: 0
```

El ciclo do-while se usa generalmente cuando la condición que determina su finalización está dentro de la acción que se debe llevar a cabo, como en el caso anterior, donde el dato que ingresa el usuario dentro del bloque de instrucciones es lo que determina el final del ciclo. Si nunca se ingresa el valor 0 en el ejemplo anterior el ciclo nunca terminaría.

El ciclo *for*

Su formato es:

for (*inicialización; condición; incremento***)** *acción;*

y su función principal es repetir la *acción* mientras la *condición* permanece verdadera, como en el ciclo *while*. Pero además, el ciclo *for* provee lugares para especificar una instrucción de *inicialización* y una instrucción de *incremento*. Así que este ciclo está especialmente diseñado para realizar una *acción* repetitiva con un contador.

Trabaja de la siguiente manera:

1. *inicialización* se ejecuta.
   Generalmente es un valor inicial para una variable *contador*.
   Se ejecuta sólo una vez.
2. *condición* se chequea.
   Si es verdadera (true) el ciclo continúa, sino el ciclo finaliza y *acción* es obviada.
3. *acción* se ejecuta.
   Como siempre, puede ser una instrucción única o un bloque de instrucciones dentro de corchetes {}.
4. finalmente, lo que se haya especificado en *incremento* se ejecuta y el ciclo vuelve al paso 2.

Este es un ejemplo de una cuenta regresiva usando un ciclo for.

```
// cuenta regresiva usando for
#include <iostream>
using namespace std;

int main ()
{
```

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, LISTO!
```

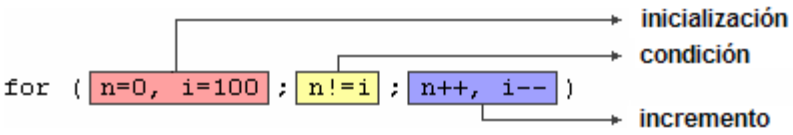
```
  for (int n=10; n>0; n--) {
    cout << n << ", ";
  }
  cout << "LISTO!";
  return (0);
}
```

La inicialización y el incremento son opcionales. Pueden ser obviados, pero no el punto y coma entre ellos. Así, por ejemplo se podría escribir: *for (;n<10;)* si no se quiere especificar ni inicialización ni incremento; o *for (;n<10;n++)* si se quiere incluir un incremento pero no una inicialización.

Opcionalmente, usando el operador coma (,) se puede especificar más de una instrucción en cualquier campo de un ciclo for, como en inicialización, por ejemplo. El operador coma (,) es un separador de instrucciones: sirve para separar más de una instrucción cuando generalmente se espera sólo una instrucción. Por ejemplo, suponga que queremos inicializar más de una variable en nuestro ciclo:

```
for ( n=0, i=100 ; n!=i ; n++, i-- )
{
  // código a ejecutar...
}
```

Este ciclo se ejecutará 50 veces si ni *n* ni *i* se modifican dentro del ciclo.

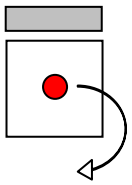


n comienza en 0 e *i* en 100, la condición es (n!=i) (que n sea distinto de i). Debido a que n se incrementa en 1 e i se decrementa en 1, la condición del ciclo se volverá falsa después de 50 iteraciones, cuando n e i sean ambas iguales a 50.

Bifurcación de control y saltos

La instrucción *break*

Usando *break* se puede salir de un ciclo aún cuando la condición para su fin no se cumpla. Puede ser usada para finalizar un ciclo indeterminado, o forzar el ciclo a finalizar antes de su fin natural. Por ejemplo, se va a detener la cuenta regresiva antes de que termine por sí misma:

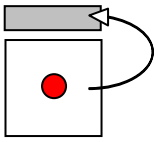


```
// ejemplo de break en un ciclo
#include <iostream>
using namespace std;

int main ()
{
  int n;
  for (n=10; n>0; n--) {
    cout << n << ", ";
    if (n==3)
    {
      cout << "cuenta regresiva abortada!";
      break;
    }
  }
  return (0);
}
```

```
10, 9, 8, 7, 6, 5, 4,
cuenta regresiva
abortada!
```

La instrución *continue*

La instrucción *continue* causa que el programa 'saltee' el resto del código del ciclo que se está ejecutando, como si se hubiera llegado al final del bloque de instrucciones de acción, causando que se pase a la siguiente iteración. Por ejemplo, se va a saltear el número 5 en la cuenta regresiva:



```
// ejemplo de continue en ciclo
#include <iostream>
using namespace std;

int main ()
{
  for (int n=10; n>0; n--) {
    if (n==5) continue;
    cout << n << ", ";
  }
  cout << "LISTO!";
  return (0);
}
```

```
10, 9, 8, 7, 6, 4, 3, 2, 1, LISTO!
```

La instrucción *goto*

Permite realizar un salto absoluto hacia otra parte del programa. Se debe ser cuidadoso al usar esta instrucción dado que su ejecución ignora cualquier tipo de limitación por anidado, por lo que hay que evitar esta instrucción siempre que sea posible.

El punto de destino se identifica con una etiqueta (label), que se usa como argumento para la instrucción *goto*. Una etiqueta se hace con un identificador válido seguido de un signo de dos puntos (:).

Esta instrucción no posee una utilidad completa en la programación orientada a objetos o la programación estructurada, excepto la que los programadores de bajo nivel puedan encontrarle. Por ejemplo, se realiza nuevamente el programa de cuenta regresiva usando goto:

```
// ejemplo de goto en un ciclo
#include <iostream>
using namespace std;

int main ()
{
  int n=10;
  loop:
  cout << n << ", ";
  n--;
  if (n>0) goto loop;
  cout << "LISTO!";
  return (0);
}
```

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
LISTO!
```

La función *exit*

El propósito de *exit* es terminar la ejecución del programa devolviendo un código específico. Su prototipo es:

```
void exit (int código_de_salida);
```

El *código_de_salida* es utilizado por ciertos sistemas operativos y puede ser usado por otros programas. Por convención, un código_de_salida de 0 significa que el programa terminó correctamente y cualquier otro valor implica que hubo un error en la ejecución.

La estructura selectiva: *switch*

Consiste en chequear varios posibles valores constantes para una expresión, algo similar a lo realizado al comienzo de esta sección al relacionar varios *if* y *else if*. Su formato es el siguiente:

```
switch (expresión) {
  case constante_1:
    bloque de instrucciónes 1
    break;
  case constante_2:
    bloque de instrucciones 2
    break;
  .
  .
  .
  default:
```

```
    bloque de instrucciones por defecto
}
```

Switch evalúa la *expresión* y chequea si es equivalente a la *constante_1*, si es así, ejecuta el bloque de *instrucciones 1* hasta que encuentra la palabra reservada *break*, momento en el cual el programa salta al final de la estructura selectiva switch.

Si la expresión no es igual a la constante_1 chequeará si es equivalente a la constante_2. De ser así, ejecutará el bloque de instrucciones 2 hasta que encuentre el break.

Finalmente, si el valor de la expresión no concuerda con ninguna de las constantes especificadas con anterioridad (puedes especificar todas condiciones que desees chequear con el case), el programa ejecutará el bloque de instrucciones por defecto incluido en la sección *default* si existe, dado que es opcional.

Los siguientes fragmentos de código son equivalentes:

ejemplo de switch	if-else equivalente

switch (x) {
  case 1:
  cout << "x es 1";
  break;
  case 2:
  cout << "x es 2";
  break;
  default:
  cout << "valor de x desconocido";
  }
	if (x == 1) {
	  cout << "x es 1";
	  }
	else if (x == 2) {
	  cout << "x es 2";
	  }
	else {
	  cout << "valor de x desconocido";
	  }

Observar la inclusión de las instrucciones break después de cada bloque. Esto es necesario porque si por ejemplo no se incluyera un break después de un bloque de instrucciones el programa no saltaría al final del bloque switch (}) y seguiría ejecutando el resto de las instrucciones hasta la primera aparición de una instrucción break o el final del bloque selectivo switch. Esto hace innecesaria la inclusión de los corchetes { } al final de cada caso, y puede ser útil también para ejecutar un mismo bloque de instrucciones para diferentes posibles valores para la expresión evaluada, por ejemplo:

```
switch (x) {
  case 1:
  case 2:
  case 3:
    cout << "x es 1, 2 o 3";
    break;
  default:
    cout << "x no es 1, 2 ni 3";
  }
```

Observar que *switch* sólo puede ser utilizado para comparar una expresión con diferentes constantes. Por ello no se puede poner variables (*case (n*2):*) o rangos (*case (1..3):*) porque no son constantes válidas.

Si se necesita verificar rangos o valores que no son constantes, utilizar una concatenación de instrucciones *if* y *else*.

Funciones

Al usar funciones se puede estructurar el programa de un modo más modular, accediendo al potencial que la programación estructurada que C++ puede ofrecer.

Una función es un bloque de instrucciones que se ejecuta cuando es llamado desde algún otro punto en el programa. Su formato es el siguiente:

```
tipo nombre ( argumento1, argumento2, ...) acción
```

donde:

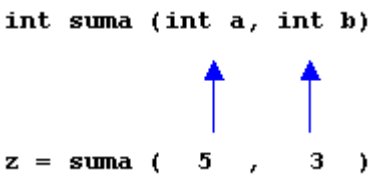
- *tipo* es el tipo de dato que la función retorna.
- *nombre* es el nombre por el cual se hace posible llamar a la función.
- *argumentos* (pueden especificarse tantos como sean necesarios). Cada argumento consiste en un tipo de dato seguido por su identificador, como en una declaración de variable (por ejemplo, int x) y que actúa dentro de la función como cualquier otra variable. Los argumentos permiten pasar parámetros a la función cuando es llamada. Los diferentes parámetros van separados por comas.
- *acción* es el cuerpo de la función. Puede ser una sola instrucción o un bloque de instrucciones, en este último caso encerrado por corchetes {}.

Primer ejemplo de función

```
// ejemplo de función
#include <iostream>
using namespace std;

int suma (int a, int b)
{
  int r;
  r=a+b;
  return (r);
}

int main ()
{
  int z;
  z = suma (5,3);
  cout <"El resultado es " << z;"
  return (0);
}
```

```
El resultado es 8
```

Se debe comenzar a examinar este código por la función main donde primero se declara la variable z del tipo int. Luego hay una llamada a la función suma. Si se presta atención se podrá ver la semejanza entre la estructura de la llamada a la función y la declaración de la función en sí algunas líneas más arriba:



Los parámetros guardan una clara correspondencia. En la función main se llama a la función suma pasándole dos valores 5 y 3 que se corresponden con los parámetros *int a* e *int b* declarados para la función suma.

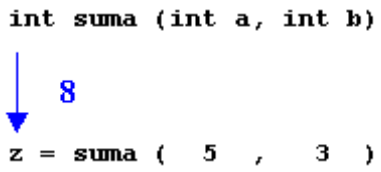
Al momento en que se llama a la función desde main, el control del programa pasa desde main a la función suma. El valor de los parámetros pasados en la llamada (5 y 3) son copiados a las variables locales int a e int b dentro de la función.

La función suma declara una nueva variable (int r;), y mediante la expresión r=a+b;, le asigna a r el resultado de la suma entre a y b. Debido a que los parámetros que se pasaron para a y b fueron 5 y 3 respectivamente, el resultado es 8.

La siguiente línea de código:

```
return (r);
```

finaliza la función suma, y le retorna el control a la función que la ha llamado (en este caso fue main) continuando el programa en el mismo punto en el que fue interrumpido por la llamada a suma . Además, return fue llamado con el contenido de la variable r (return (r);), que en ese momento era 8, así que por ello se dice que la función retorna ese valor.



El valor retornado por la función es el valor dado a la función cuando es evaluada. Por lo tanto, z guardará el valor retornado por suma(5, 3), que es 8. Para explicarlo de alguna manera, se puede imaginar que la llamada a la función (suma (5,3)) es literalmente reemplazada por el valor que la función devuelve (8).

La siguiente línea de código en main:

```
cout << "El resultado es " << z;
```

muestra el resultado en la pantalla.

Ámbito de las variables

Se debe considerar que el *alcance* de las variables declaradas dentro de una función o cualquier otro bloque de instrucciones es sólo la propia función o el propio bloque de instrucciones y no puede ser usada fuera de él. Así, en el ejemplo anterior es imposible usar las variables *a, b* o *r* directamente en la función *main* dado que eran variables *locales* de la función *suma*. Además, es imposible usar la variable **z** directamente dentro de la función *suma*, dado que esta es una variable *local* de la función *main*.

Por consiguiente, el alcance de las variables *locales* está limitado al mismo nivel de anidado en el que son declaradas. Aún así, también se puede declarar variables *globales* que son accesibles desde cualquier parte del código, dentro o fuera de cualquier función. Para declarar variables *globales* hay que situarlas fuera de cualquier función o bloque de instrucciones, o sea, directamente en el cuerpo del programa.

Segundo ejemplo de uso de funciones

```
// ejemplo de funciones
#include <iostream>
using namespace std;

int resta (int a, int b)
{
  int r;
  r=a-b;
  return (r);
}

int main ()
{
  int x=5, y=3, z;
  z = resta (7,2);
  cout << "El primer resultado es " << z << '\n';
  cout << "El segundo resultado es "
       << resta (7,2) << '\n';
  cout << "El tercer resultado es "
       << resta (x,y) << '\n';
  z= 4 + resta (x,y);
  cout << "El cuarto resultado es "
       << z << '\n';
  return (0);
}
```

```
El primer resultado es 5
El segundo resultado es 5
El tercer resultado es 2
El cuarto resultado es 6
```

En este caso se ha creado la función *resta*. Lo único que hace esta función es restar los parámetros que se pasan y devolver el resultado.

Si se examina la función main se verá que hay varias llamadas a la función resta. Como se ve, se ha usado diferentes métodos de llamada para ver otras maneras o momentos en los cuales una función puede ser llamada.

Para entender estos ejemplos se debe considerar una vez más que la llamada a una función puede perfectamente ser reemplazada por el valor que devuelve. Por ejemplo en el primer caso:

```
z = resta (7,2);
cout << "El primer resultado es " << z;
```

Si se reemplaza la llamada a la función por el valor que retorna (esto es, 5), se tendría:

```
z = 5;
cout << "El primer resultado es " << z;
```

como también

```
cout << "El segundo resultado es " << resta (7,2);
```

tiene el mismo resultado que la llamada previa, pero en este caso se realizó la llamada a resta directamente como un parámetro de cout. Simplemente imaginar que se hubiera escrito:

```
cout << "El segundo resultado es " << 5;
```

dado que 5 es el resultado de resta (7,2).

En el caso de

```
cout << "El tercer resultado es " << resta (x,y);
```

la única variante introducida es que los parámetros de resta son variables en vez de constantes, lo cual es perfectamente válido. En este caso los valores que se pasan a la función resta son los valores que poseen x e y, que son 5 y 3 respectivamente, devolviendo 2 como resultado.

El cuarto caso es más de lo mismo. Simplemente se comenta que en vez de:

```
 z = 4 + resta (x,y);
```

se podría haber puesto:

```
z = resta (x,y) + 4;
```

obteniéndose exactamente el mismo resultado. Observar que el punto y coma (;) no necesariamente debe ir a la derecha de la llamada a la función, sino que siempre va al final de la expresión completa. La explicación podría ser una vez más que la función puede ser reemplazada por su resultado:

```
z = 4 + 2;
z = 2 + 4;
```

Funciones sin tipo (*void*)

Recordando la sintaxis de la declaración de una función:

tipo *nombre* **(** *argumento1*, *argumento2* **...)** *acción*

se observó que es obligatorio que su declaración comience con un *tipo*, que es el tipo de dato que será devuelto por la función con la instrucción return. Pero ¿qué pasa si no se quiere devolver ningún valor?.

Por ejemplo, se quiere hacer una función que sólo muestre un mensaje en pantalla. Por lo tanto, no se necesita que retorne ningún valor; es más, ni siquiera recibir parámetros. Para esto fue diseñado el tipo *void*. Observar el siguiente ejemplo:

```
// ejemplo de funciones void
#include <iostream>
using namespace std;

void mensaje (void)
{
  cout << "Soy una función!";
}

int main ()
{
  mensaje ();
  return (0);
}
```

```
Soy una función!
```

Aunque en C++ no es necesario especificar el void en los parámetros, se considera que es mejor incluirlo para aclarar que es una función sin parámetros o argumentos.

Siempre se debe tener cuidado de que el formato de llamada a una función incluye la especificación de su nombre y, encerrado entre paréntesis, los argumentos. Que no haya ningún argumento no implica que no se esté obligado a poner los paréntesis, por esta razón la llamada a mensaje es

```
mensaje ();
```

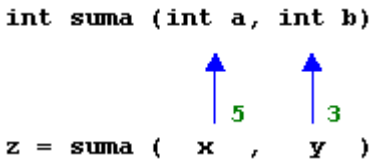
Esto comprueba que es una llamada a una función y no el nombre de una variable o algo más.

Argumentos pasados por valor y por referencia

Hasta ahora, en todas las funciones que se han visto, los parámetros pasados a las funciones han sido pasados *por valor*. Esto significa que cuando se llama a una función con parámetros, lo que se pasa en realidad son valores pero nunca las variables especificadas en sí. Por ejemplo, suponer que se llama a la función suma vista anteriormente usando el siguiente código:

```
int x=5, y=3, z;
z = suma ( x , y );
```

Lo que se ha hecho en este caso fue llamar a la función suma pasándole los valores de x e y, o sea, 5 y 3 respectivamente, no las variables en sí.



De esta forma, cuando la función suma es llamada el valor de sus variables a y b se convierte en 5 y 3 respectivamente, pero cualquier modificación de a o b dentro de la función suma no afectará los valores de x e y fuera de ella, dado que las variables x e y no fueron pasadas en sí a la función, sino sólo sus valores.

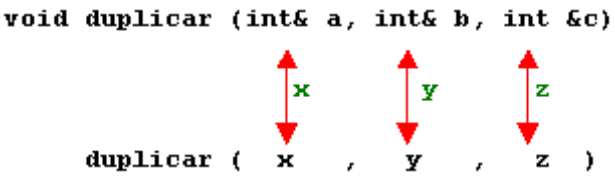
Pero podría haber ciertos casos donde se necesites manipular desde adentro de una función el valor de una variable externa. Para este propósito se tiene que usar parámetros pasados por referencia, como en la función duplicar del siguiente programa:

```
// pasando parámetros por referencia
#include <iostream>
using namespace std;

void duplicar (int& a, int& b, int& c)
{
  a=a*2;
  b=b*2;
  c=c*2;
}

int main ()
{
  int x=1, y=3, z=7;
  duplicar (x, y, z);
  cout << "x=" << x << ", y=" << y << ", z=" << z;
  return (0);
}
```

```
x=2, y=6, z=14
```

Observar que los parámetros de la función duplicar tienen un símbolo ampersand (&) luego del tipo de dato, que sirve para especificar que la variable será pasada por referencia en vez de por valor.

Cuando pasamos una variable por referencia se está pasando la variable en sí misma y cualquier modificación que se realice al parámetro dentro de la función tendrá efecto en la correspondiente variable fuera de ella.



Para expresarlo de alguna forma, se ha *asociado* a las variables a, b y c con los parámetros usados al llamar la función (**x**, **y** y **z**) y cualquier cambio que realizado sobre la variable **a** dentro de la función afectará el valor de *x* fuera de ella. Cualquier cambio que realicemos sobre *b* afectará a *y*, y lo mismo con *c* y *z*.

Es por esto que la salida del programa, que muestra los valores de **x**, **y** y **z** luego de la llamada a duplicar, muestra los valores de las tres variables de main duplicados.

Si cuando se declaró la función:

```
void duplicar (int& a, int& b, int& c)
```

se hubiera declarado:

```
void duplicar (int a, int b, int c)
```

sin el signo ampersand (&), no se hubiera pasado las variables por referencia, y sus valores, y por ende la salida en pantalla, hubieran sido los valores de **x**, **y** y **z** sin modificaciones.

Este tipo de declaración "por referencia" usando el signo ampersand (&) es exclusivo de C++. En el lenguaje C se tiene que usar punteros para hacer algo equivalente.

El pasaje por referencia es una manera efectiva para permitirle a una función retornar más de un solo valor. Por ejemplo en ejemplo que está a continuación, hay una función que retorna el valor previo y el valor siguiente del primer parámetro que se pasa.

```
// más de un valor retornado por una función
#include <iostream>
using namespace std;

void anterior_posterior (int x, int& anterior,
int& posterior)
{
  anterior  = x-1;
  posterior = x+1;
}

int main ()
{
  int x=100, y, z;
  anterior_posterior (x, y, z);
  cout << "Anterior=" << y << ", Posterior=" <<
```

```
Anterior=99, Posterior=101
```

```
z;
  return (0);
}
```

Valores por defecto en los parámetros

Cuando se declara una función se puede especificar un valor por defecto para cada parámetro. Este valor será usado si el parámetro es obviado al llamar a la función. Para hacer esto simplemente se debe asignar un valor a dicho parámetro en la declaración de la función. Si el valor para ese parámetro no se especifica cuando se llama a la función, el valor por defecto es utilizado; pero si se especifica un valor, el valor por defecto se omite y se utiliza el valor pasado.

Por ejemplo:

```
// valores por defecto en funciones
#include <iostream>
using namespace std;

int dividir (int a, int b=2)
{
  int r;
  r=a/b;
  return (r);
}

int main ()
{
  cout << dividir (12);
  cout << endl;
  cout << dividir (20,4);
  return (0);
}
```

```
6
5
```

Como se puede ver en el cuerpo del programa hay dos llamadas a la función dividir. En la primera:

```
dividir (12)
```

sólo especificamos un argumento, aunque la función dividir permite hasta dos. Así que la función dividir asume que el segundo parámetro es 2 dado que esto es lo que se ha especificado si falta el segundo parámetro (observar en la declaración de la función, que termina con int b=2). Por ende, el resultado a la llamada de la función es 6 (12/2).

En la segunda llamada:

```
dividir (20,4)
```

hay dos parámetros, así que la asignación por defecto (int b=2) es reemplazada por el parámetro pasado, que es 4, dando esta ecuación como resultado 5 (20/4).

Recordar que las variables por defecto deberán declararse de derecha a izquierda. Es incorrecto la siguiente declaración:

```
int dividir (int a=4,int b)
```

Sobrecarga de funciones

Dos funciones diferentes pueden tener el mismo nombre si el prototipo de sus parámetros es distinto, esto significa que se puede nombrar de la misma manera dos o más funciones si tienen distinta cantidad de parámetros o diferentes tipos de parámetros para que C++ pueda diferenciar a cual llamar.

Por ejemplo:

```
// sobrecarga de funciones
#include <iostream>
using namespace std;

int dividir (int a, int b)
{
  return (a/b);
}

float dividir (float a, float b)
{
  return (a/b);
}

int main ()
{
  int x=5,y=2;
  float n=5.0,m=2.0;
  cout << dividir (x,y); // llamará al de parámetros int
  cout << endl;
  cout << dividir (n,m); // llamará al de parámetros float
  return (0);
}
```

```
2
2.5
```

En este caso se ha definido dos funciones con el mismo nombre, pero una de ellas acepta parámetros de tipo int y la otra acepta parámetros de tipo float. El compilador sabe a cuál llamar en cada caso examinando los tipos al llamar la función: si es llamada con dos int como parámetros llamará a la función que tiene dos parámetros de tipo int en el prototipo, y ocurrirá lo mismo si llamamos a la función con dos parámetros de tipo float.

Por simplicidad, ambas funciones tienen el mismo código, pero esto no es estrictamente necesario. Se puede hacer dos funciones con el mismo nombre pero con comportamientos totalmente diferentes.

Funciones *inline* (obsoleto)

La directiva *inline* puede ser incluida antes de la declaración de una función para especificar que esa función debe ser compilada como código en el mismo punto en el que se encuentra la llamada a dicha función. Sus ventajas sólo se aprecian en funciones muy cortas, en las cuales el código resultante de compilar el programa puede ser más rápido, debido a que se evita llamar a esa función.

El formato de su declaración es:

inline tipo *nombre* **(** *argumentos ...* **) {** *instrucciones ...* **}**

y la llamada es exactamente igual a la de cualquier otra función. No es necesario incluir la palabra **inline** antes de cada llamada, sólo en la declaración.

```
#include <iostream>
using namespace std;

inline int dividir (int a, int b) {  return (a/b); }

int main ()
{
  int x=5,y=2;
  cout << dividir (x,y); // llamará al de parámetros int
  return (0);
}
```

Prototipo de funciones

Hasta ahora, se ha definido las funciones antes de la primera aparición de la llamada a esas funciones, que generalmente ocurre en el main, dejando la función main para el final. Si se trata de repetir algunos de los ejemplos de funciones descritos hasta ahora, pero colocando la función main antes que cualquier otra función que se llame dentro de ella, probablemente se obtendrá un mensaje de error. El motivo es que para poder llamar a una función, ésta debe haber sido declarada previamente.

Pero hay una manera alternativa de evitarnos escribir todo el código de una función antes de que sea usada en main o en otra función, y es haciendo el *prototipo de la función*. Esto consiste en hacer una declaración previa y corta de la definición completa, pero lo suficientemente completa como para que el compilador sepa qué parámetros necesita y el tipo de datos que devuelve. Su formato es:

```
tipo nombre ( tipo_de_argumento1, tipo_de_argumento2, ...);
```

Esto es idéntico a la declaración de una función, sólo que:

- No incluye la *acción* que realiza la función. Esto significa que no incluye el cuerpo con todas las instrucciones que usualmente van encerradas entre corchetes {}.
- Termina con un símbolo de punto y coma (;).

En la enumeración de parámetros es suficiente poner el tipo de cada parámetro. La inclusión de un nombre para cada parámetro como en la definición de una función estándar es opcional, aunque recomendable. Por ejemplo:

```
// prototipos
#include <iostream>
using namespace std;

void impar (int a);
void par (int a);

int main ()
{
  int i;
  do {
    cout << "Ingrese un número: (0 salir)";
    cin >> i;
    impar (i);
  } while (i!=0);
  return (0);
}
```

```
Ingrese un número (0 salir): 9
El número es impar.
Ingrese un número (0 salir): 6
El número es par.
Ingrese un número (0 salir):
1030
El número es par.
Ingrese un número (0 salir): 0
El número es par.
```

```
void impar (int a)
{
  if ((a%2)!=0) cout << "El número es
impar.\n";
  else par (a);
}

void par (int a)
{
  if ((a%2)==0) cout << "El número es
par.\n";
  else impar (a);
}
```

Este ejemplo en sí no es una muestra de efectividad pero este ejemplo ilustra como funcionan los prototipos, y en este caso el prototipo de al menos una función es necesario.

Lo primero que se identifica son los prototipos de las funciones impar y par:

```
void impar (int a);
void par (int a);
```

que permite el uso de estas funciones antes de que se definan completamente.

No obstante, la razón específica por la que este programa necesita al menos que una de las funciones tenga un prototipo, es que en la función impar hay una llamada a la función par y viceversa, razón por la cual si ninguna de las dos funciones ha sido previamente declarada, un error ocurriría, dado que impar no sería visible para par (porque todavía no hubiera sido declarada), y viceversa.

Muchos programadores recomiendan que todas las funciones sean prototipadas, mayormente en el caso que haya muchas funciones o en el caso de que sean muy largas, dado que situando el prototipo de la función en ese lugar nos puede ahorrar tiempo al consultar cómo llamarlas o facilitar la creación de un archivo de encabezamiento (archivos ".h").

Puntero a funciones

La mayor utilidad de trabajar con punteros a funciones, es la posibilidad de pasar una función como parámetro a otra función, dado que éstas no puede ser pasadas por derreferencia.

```
// punteros a función
#include <iostream>
using namespace std;

int suma(int a, int b) { return(a+b); } // función normal

int resta(int a, int b) { return(a-b); } // función normal

int operacion(int x, int y, int (*func_a_llamar)(int,int))
{ int g;
  G = (*func_a_llamar) (x,y);
  return(g);
}

int main ()
{
```

```
  int m;
  m=operacion(7,5,&suma);
  cout << m << endl;

  m=operacion(7,5,&resta);
  cout << m << endl;

  return (0);
}
```

En la función *operación* del código anterior, el último parámetro:

```
int (*func_a_llamar)(int,int))
```

significa que espera recibir la dirección de una función (que como nombre local será *func_a_llamar*), la que debe devolver un valor entero y que además, esa función pasada, deberá recibir como parámetro dos enteros.

Entonces, el siguiente llamado:

```
m=operacion(7,5,&suma);
```

es válido, porque se le está pasando la dirección de una función a través de *&suma*, ésta función *suma* devuelve un entero, y además tiene especificado dos parámetros enteros.

La salida por pantalla del código anterior es 12 y 2.

Ejemplos de aplicación

Generación de números al azar

El siguiente código utiliza números al azar de utilidad para los programas de simulación.

```
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{  srand(time(0));   // genera la semilla del rand()
   cout << "Número al azar entre 0 y " <<  RAND_MAX << " : "
        <<rand() << endl;

   float n;
   cout << "Número al azar entre 0 y 1: " << endl;
   for (int i=0; i<10 ;i++)
   {  n = (float) rand()/RAND_MAX;
      cout << setprecision(4)  << n << endl;
   }

  cin.get();
  return 0;
}
```

Uso de tiempos

El siguiente código ejemplifica el uso de funciones para trabajar con el reloj de la computadora.

```
#include <ctime>
#include <iostream>

using namespace std;

void sleep (unsigned retardo)
{ unsigned tiempo_inicial = clock();
  // En Windows la función clock() da el tiempo en milisegundos
  // desde el inicio del programa. En Linux es en microsegundos.

   while((clock() - tiempo_inicial) < retardo)  {}
}

int main()
{  unsigned retardo;

   cout << "Ingrese tiempo de espera: ";
   cin >> retardo; cin.get();

   sleep(retardo);

   cout << "Fin de la espera. " << endl;

   cout << "Realizado en: ";
   time_t actual = time(0);         // hora actual
   tm * ptr = localtime(&actual);   // se crea la estructura

   cout << asctime(ptr); // convierte a cadena tipo
                         // "Thu Aug 30:18:45 2006"

   return 0; }
```

Ordenamiento

Código de ordenamiento por el método de la burbuja.

```
// ------------------------------------------
//      Ordenamiento por burbuja
// ------------------------------------------

#include <iostream>

const int MAX=5;
int k[MAX];

void Ingresar_Valores()
{
  for (int i=0; i<MAX; i++)
   {
     cout << "Ingresar el valor " << i+1 << ": ";
     cin >> k[i];
   };
 };
```

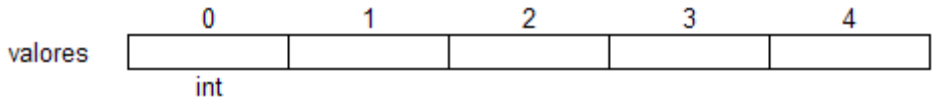
```
void Ordenar_Ascendente()
 { int aux;
   for (int j=0; j<MAX; j++)
      {
         for (int x=j+1; x<MAX; x++)
        {
           if (k[j] >= k[x])
           {
              aux=k[x];
              k[x]=k[j];
              k[j]=aux;
           };
          };
       };
};


void Ordenar_Descendente()
{  int aux;
   for (int j=0;j<MAX;j++)
      {
         for (int x=j+1;x<MAX;x++)
        {
           if (k[j] <= k[x])
           {
             aux=k[x];
             k[x]=k[j];
             k[j]=aux;
           };
         };
       };
 };


void Mostrar_Valores()
{
    cout << "Los números ingresados fueron:" << endl;
    for (int i=0; i<MAX; i++)
     cout << k[i] << "  ";

    cout << endl;
}


int main()
{
   int opcion;

   cout << "ORDENAMIENTO DE UNA LISTA DE 5 NUMEROS" << endl;
   cout << "==============================" << endl;

   bool seguir=true;

  while (seguir)
  {
     cout <<"Seleccione una opción " << endl;
     cout <<"1) Ingresar los números." << endl;
     cout <<"2) Ordenar Ascendentemente." << endl;
     cout <<"3) Ordenar Descendentemente." << endl;
```

```
      cout <<"4) Mostrar Valores." << endl;
      cout <<"5) Salir." << endl;
      cout <<"Opción: ";
      cin >> opcion;
      cout << "================================" << endl;

     switch (opcion)
     {
       case 1:  Ingresar_Valores(); break;
       case 2:  Ordenar_Ascendente(); cout << "Fin ordenamiento" << endl;
                break;
       case 3:  Ordenar_Descendente(); cout << "Fin ordenamiento" << endl;
                break;
       case 4:  Mostrar_Valores(); break;
       case 5:  seguir=false; break;
       default: cout << "Ingresar un valor de 1 a 5" << endl; break;
     };

    };  // final del while (seguir)

};
```

Arreglos y cadenas de caracteres

Arreglos y vectores

Los arreglos son series de elementos del un mismo tipo ubicados consecutivamente en la memoria y pueden ser referenciados individualmente agregando un índice a un nombre único. Por ejemplo, mediante un arreglo se puede almacenar 5 valores de tipo int sin necesidad de declarar 5 variables diferentes, cada una con un identificador distinto. Mediante un arreglo se pueden guardar estos 5 valores en un único identificador:



donde cada casillero en blanco representa un elemento del arreglo, que en este caso son valores enteros de tipo int. Estos elementos están numerados del 0 al 4 dado que en los arreglos el primer índice es siempre 0, independientemente de su longitud.

Como cualquier otra variable, un arreglo debe ser declarado antes de ser usado. Una declaración típica de un arreglo en C++ es:

```
tipo nombre[cantidad_elementos];
```

donde *tipo* es un tipo de dato válido (int, float...), nombre es un identificador válido y el campo *cantidad_elementos*, que va entre corchetes [], especifica cuántos elementos contiene el arreglo. Así, para declarar el arreglo *valores*, la sintaxis simplemente es:

```
int valores[5];
```

y para asigarle un valor:

```
valores[2]=5;
```

En la declaración de este tipo de arreglos, el campo *cantidad_elementos* debe ser un valor constante, dado que los arreglos son bloques de memoria estáticos de un tamaño determinado y el compilador necesita determinar exactamente cuánta memoria debe asignar al arreglo antes de que cualquier instrucción sea considerada.

Este tipo de estructuras se conoce como *arreglos estáticos*. Pero... ¿cómo se puede hacer para dar al usuario la posibilidad de definir el tamaño del arreglo? Evidentemente no es posible con un arreglo de este tipo debido a que la cantidad de elementos debe ser constante. Para solucionar este tipo de limitaciones se creó el *vector dinámico*, que de ahora en más llamaremos simplemente *vector*. Por ejemplo, se puede declarar:

```
vector < int > valores(5);
```

Este vector es equivalente al arreglo estático que se había declarado antes pero es toda una clase y posee varios métodos útiles. Por ejemplo, tiene la ventaja de que luego se puede cambiar su cantidad de elementos simplemente con:

```
valores.resize(nueva_cantidad_elementos);
```

Para utilizar este tipo de vectores dinámicos es necesario incluir un módulo particular mediante `#include <vector>`.

Como recomendación general, siempre que sea posible se debe utilizar un vector dinámico en lugar de un arreglo estático.

Inicialización

Cuando se declara un arreglo estático de alcance *local* (dentro de una función), si no se especifica lo contrario, no será inicializado, así que su contenido es indeterminado hasta que se almacene algún valor dentro de él.

Si se declara un arreglo *global* (afuera de toda función) su contenido será inicializado con todos sus elementos con valor 0 (cero). Así, si se declara fuera de toda función:

```
int valores [5];
```

cada elemento de *valores* será inicialmente puesto en 0:

	0	1	2	3	4
valores	0	0	0	0	0

Estas alternativas suelen confundir al programador y terminan por provocar errores muy difíciles de detectar. Como buen hábito de programación se recomienda inicializar siempre los arreglos antes de utilizarlos. Por ejemplo, simplemente con un ciclo *for*.

Cuando se declara un arreglo estático es posible asignar valores iniciales a cada uno de sus elementos usando llaves { }. Por ejemplo:

```
int valores [5] = { 16, 2, 77, 40, 12071 };
```

esta declaración crea un arreglo como el siguiente:

	0	1	2	3	4
valores	16	2	77	40	12071

El número de elementos en el arreglo que se inicializan a través de las llaves { } debe coincidir con la cantidad de elementos que se declaran para el arreglo. Por ejemplo, en el ejemplo anterior se declaró que tenía 5 elementos y en la lista de valores iniciales dentro de las llaves { } se incluyó 5 valores, uno por cada elemento.

Dado que esto puede ser considerado como una repetición inútil, C++ incluye la posibilidad de dejar los corchetes vacíos siendo el tamaño del arreglo definido por el número de valores incluidos dentro de las llaves { }:

```
int valores [] = { 16, 2, 77, 40, 12071 };
```

Nuevamente los vectores solucionan limitaciones importantes de los arreglos. En el caso de las inicialización, todos los vectores creados y los valores inicializados a cero.

Por ejemplo, en el caso del vector dinámico:

```
vector < int > valores(5);
```

se creará un vector de 5 elementos y se inicializarán sus valores a cero, independientemente de dónde sea declarado. Si se hubiese tratado de un vector de números reales, también se hubiesen inicializado todos los valores a 0.0.

Pero además existen muchas alternativas útiles en la creación de vectores.

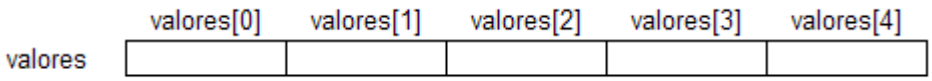
```
vector < int > a;
// vector vacío, es decir, sin ningún elemento

vector < char > b(10, 'A');
// vector de 10 caracteres todos inicializados con el carácter A

vector < int > c(b);
// el vector c será una copia idéntica del vector b
```

Acceso a los valores individuales

En cualquier punto del programa en el que el arreglo o vector sea visible, se puede acceder individualmente a cualquiera de sus valores para leerlos o modificarlos como si fueran variables normales.

El formato es: *nombre[indice]*

Siguiendo con el ejemplo previo en el cual *valores* tenía 5 elementos de tipo int, el nombre que se puede usar para referenciar cada elemento es:



Tanto si fue declarado como un arreglo estático o como un vector dinámico. Por ejemplo, para almacenar el valor 75 en el tercer elemento, la instrucción sería:

```
valores[2] = 75;
```

y para pasar el tercer elemento de *valores* a la variable **a**, se podría escribir:

```
a = valores[2];
```

Por lo tanto, para cualquier efecto, la expresión valores[2] es como cualquier otra variable de tipo int, con las mismas propiedades.

Se debe observar que el tercer elemento de *valores* es referenciado como valores [2], dado que el primero es valores[0], segundo valores[1]. Por esta misma razón, el último elemento es valores[4]. Si se escribiera valores[5], se estaría tratando de acceder al sexto elemento del arreglo y se generaría un error grave dado que se habrá excedido el tamaño del mismo.

En C++ no se generan errores de compilación al exceder el rango de índices de un arreglo o vector y esto puede causar ciertas dificultades ya que los errores resultantes pueden ser de la índole más diversa. En el mejor de los casos el programa se abortará dando un mensaje de violación de áreas de memoria. Pero es posible que ni este tipo de mensaje surja y el programa se comporte de la forma más extraña, sin seguir la lógica que hemos programado.

Es importante distinguir claramente entre las dos formas en que se utilizan los corchetes []. Realizan dos tareas diferentes: una es definir el tamaño de los arreglos estáticos al declararlos y la otra es especificar

el índice de un elemento concreto del arreglo para poder referirse a él. Simplemente se debe tener cuidado de no confundir estas dos posibilidades de uso de los corchetes [ ] en los arreglos:

```
int valores[5];    // declaración arreglo estático (comienza con un tipo)
valores[2] = 75;   // acceso a un elemento del arreglo o vector
```

Analice estas operaciones simples con arreglos y discuta los resultados:

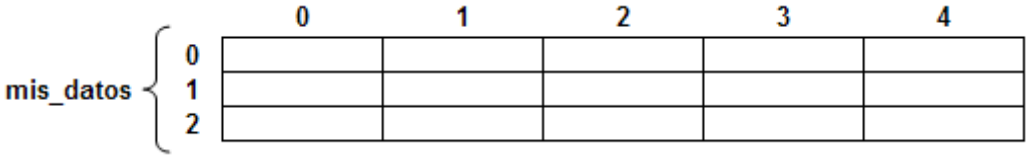
```
int a = 1;
int b;
valores[0] = a;
valores[a] = 0;
b = valores[a+2];
valores[valores[a]] = valores[2] + 5;
```

```
// ejemplo de arreglos
#include <iostream>
using namespace std;

int valores [] = {16, 2, 77, 40, 12071};
int n, resultado=0;

int main ()
{
  for ( n=0 ; n<5 ; n++ )
  {
    resultado += valores[n];
  }
  cout << resultado;
  return (0);
}
```

```
12206
```

Arreglos multidimensionales

Los arreglos multidimensionales pueden ser descritos como arreglos de arreglos. Por ejemplo, un arreglo bidimensional puede imaginarse como una tabla bidimensional de un tipo de dato uniforme. Un arreglo bidimensional de 3 por 5 con valores de tipo *int* se podría representar como una cuadrícula:

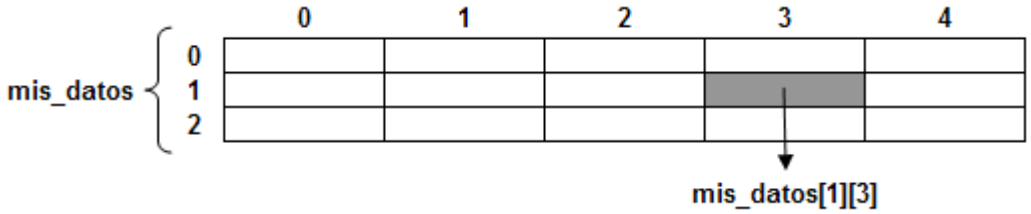


La forma de declarar un arreglo de este tipo es:

```
int mis_datos [3][5];
```

y, la forma de referenciar al elemento de la segunda fila (fila número 1) y cuarta columna (columna número 3) es:

```
mis_datos[1][3]
```



Los arreglos multidimensionales no se limitan sólo a dos índices (dos dimensiones), pueden contener tantos índices como sea necesario. Se debe tener en cuenta que la cantidad de memoria que necesitaría un arreglo con muchos índices, por ejemplo:

```
char siglo [100][365][24][60][60];
```

asigna un caracter (char) por cada segundo contenido en un siglo, que son más de 3 billones de caracteres! Esto consumiría alrededor de 3000 megabytes de memoria RAM.

Los arreglos multidimensionales no son más que una abstracción, dado que podemos obtener el mismo resultado con un arreglo simple multiplicando sus índices:

```
int mis_datos [3][5]; // es quivalente a
int mis_datos [15];   // porque 3*5=15 celdas
```

con la única diferencia que el compilador recuerda por nosotros la profundidad de cada dimensión imaginaria. A continuación se presentan dos ejemplos con exactamente el mismo resultado, uno usando un arreglo bidimensional y el otro usando un arreglo simple:

```
// arreglo multidimensional
#include <iostream>
using namespace std;

int mis_datos [3][5];
int n,m;

int main ()
{
  for (n=0;n<3;n++)
    for (m=0;m<5;m++)
    {
      mis_datos[n][m]=(n+1)*(m+1);
    }
  return (0);
}
```

```
// pseudo-arreglo multidimensional
#include <iostream>
using namespace std;

int mis_datos [3 * 5];
int n,m;

int main ()
{
 for (n=0;n<3;n++)
  for (m=0;m<5;m++)
  {
   mis_datos[n * 5 + m]=(n+1)*(m+1);
  }
  return (0);
}
```

Ninguno de los programas anteriores produce un resultado en la pantalla, pero los dos asignan valores al bloque de memoria llamado *mis_datos* de la siguiente manera:

mis_datos	0	1	2	3	4
0	1	2	3	4	5
1	2	4	6	8	10
2	3	6	9	12	15

Al igual que en el caso de los arreglos estáticos, muchas veces surge naturalmente la necesidad de cambiar las dimensiones de una matriz en cualquier parte del programa. Entonces ¿cómo podría hacerse para tener una matriz dinámica? Nuevamente vienen a nuestro auxilio los vectores dinámicos. Así como

se utilizaron arreglos de arreglos para hacer matrices estáticas, ahora se utilizarán vectores de vectores para hacer matrices dinámicas:

```
vector < vector < int >> mis_datos;
```

Pero... ¿cómo se establecen las dimensiones de esta matriz dinámica? Lo primero es especificar cuantas filas posee:

```
mis_datos.resize(3);        // se define la cantidad de filas
```

y luego, para cada una de las filas habrá que especificar la cantidad de elementos que posee:

```
for (fila=0;fila<3;fila++)      // para cada una de las filas
  mis_datos[fila].resize(5);    // se define la cantidad de columnas
```

Recordar que la cantidad de filas y columnas puede ingresarse de esta forma durante la ejecución del programa. Otra forma de dar el tamaño es mediante:

```
vector <vector <int> > mis_datos(3,5);
```

pero deberá ser especificado durante la codificación del programa.

Ahora, la matriz mis_datos puede accederse tal cual su equivalente estática pero con la gran ventaja de poder cambiar sus dimensiones de acuerdo a las necesidades del programa. Por ejemplo, para guardar un dato en la posición 1,3 sigue siendo válido hacer:

```
mis_datos[1][3]=24;
```

Pero afortunadamente (porque evita malos hábitos de programación), **no** se puede acceder a los elementos de este vector bidimensional multiplicando los dos subíndices dentro de un único corchete.

Pasaje de arreglos como parámetros de funciones

En algún momento puede ser necesario pasar como parámetro un arreglo a una función. En C++ no es posible pasar por valor un bloque de memoria completo como parámetro a una función, aún si está ordenado como un arreglo, pero está permitido pasar su dirección de memoria, lo cual es mucho más rápido y eficiente, pero aun así, no habrá sido un pasaje por valor sino por referencia.

Esta situación se simplifica notablemente cuando se utilizan vectores dinámicos en lugar de arreglos estáticos. Para el caso de los vectores el pasaje por valor y referencia se realiza tal cual fuera una variable simple como un int o un float.

Aunque es algo más complejo, a continuación se va a describir el pasaje de arreglos estáticos y luego se repetirá el mismo ejemplo pero con vectores. Para admitir arreglos como parámetros hay que declarar la función especificando en el argumento el tipo de dato que contiene el arreglo, un identificador y los corchetes vacíos [ ]. Por ejemplo, la siguiente función:

```
void procedimiento(int arg[],int largo)
```

admite un parámetro de tipo arreglo de int llamado arg. Para pasar un arreglo declarado como el siguiente a esta función:

```
int miarreglo[40];
```

la llamada debe ser: `procedimiento(miarreglo,40);`

```
// arreglo como parámetro
#include <iostream>
using namespace std;

void imprime_arreglos (int arr[], int largo)
{  for (int n=0; n<largo; n++)
     cout << arr[n] << " ";
   cout << endl;
}

int main ()
{  int arreglo_uno[] = {5, 10, 15};
   int arreglo_dos[] = {2, 4, 6, 8, 10};

   imprime_arreglos (arreglo_uno,3);
   imprime_arreglos (arreglo_dos,5);

   return (0);
}
```

```
5 10 15
2 4 6 8 10
```

Como se puede ver, el primer argumento admite un arreglo de tipo int (en realidad es la dirección de memoria a un arreglo estático), pero no es posible saber la cantidad de elementos y es por esto que se necesita un segundo parámetro que le indica a la función el largo de cada arreglo pasado como parámetro. Así, el ciclo for que muestra el arreglo en la pantalla pueda saber el largo del arreglo con el que trabaja.

En el caso de los vectores dinámicos basta con una llamada a función

```
void procedimiento(vector < int > arg)
```

Para pasar un arreglo como el siguiente:

```
vector < int > miarreglo(40);
```

sería suficiente con una llamada como esta:

```
procedimiento(miarreglo);
```

observese que no fue necesario pasar el largo del vector en este caso. Pero, ¿cómo se puede determinar el largo del vector? Los vectores entre sus muchos métodos poseen uno para obtener el tamaño:

```
miarreglo.size()
```

Utilizando un vector todo se simplifica:

```
// vectores como parámetros
#include <iostream>
#include <vector>
using namespace std;

void imprime_arreglos (vector< int > vec)
{  for (int n=0; n<vec.size(); n++)
     cout << vec[n] << " ";
   cout << endl;
}

int main ()
```

```
33 33 33
55 55 55 55 55
```

```
{ vector< int > arreglo_uno(3,33);
  vector< int > arreglo_dos(5,55);

  imprime_arreglos(arreglo_uno);
  imprime_arreglos(arreglo_dos);

  return (0);
}
```

En éste último caso, el pasaje del vector es por valor, pero si se coloca *imprime_arreglos (vector <int> **&**vec)* será por referencia con un comportamiento idéntico a los pasajes por valor y referencia de una variable normal int o float.

En la declaración de una función también es posible incluir arreglos multidimensionales. El formato para un arreglo tridimensional es:

*tipo nombre***[][***largo***][***ancho***]**

por ejemplo una función con un arreglo multidimensional como argumento sería:

```
void procedimiento (int miarreglo[][3][4])
```

Se puede notar que los primeros corchetes [] están vacíos y los siguientes no. Esto debe ser siempre así porque el compilador debe ser capaz de determinar dentro de la función cual es el tamaño de cada dimensión adicional.

Los arreglos simples o multidimensionales, pasados como parámetros de funciones son una fuente frecuente de errores entre programadores poco experimentados. Como regla general, siempre se debe tratar de utilizar un vector dinámico antes que un arreglo estático. Los vectores son más sencillos de utilizar y procian mucho menos los errores en la programación.

Más ventajas de la clase *vector*

Algunos de los métodos más útiles de la clase vector son los siguientes:

Declaración del método	Utilidad
Size_type size() const	Devuelve el tamaño actual del vector
resize(size_type num,T val = T())	Cambia el tamaño del vector agregando elementos al final. Si no se especifica val agrega ceros.
bool empty()	Devuelve true si el vector tiene cero elementos, false en otro caso.
void clear()	Elimina todos los elementos del vector, es decir, el vector queda con tamaño cero.
void push_back(const T &val)	Agrega el elemento val al final del vector. Hay que pasar como parámentro una variable del tipo que fue declarado el vector.
void pop_back()	Elimina el el último elemento del vector.

Considere los siguientes ejemplos simples para comprender como funcionan estos métodos.

```
// más sobre vectores
#include <iostream>
#include <vector>
using namespace std;

void mostrar(vector< int > vec)
{ cout << "Cantidad de elementos: "
       << vec.size() << endl;
  cout << "Valores: ";
  for (int n=0; n<vec.size(); n++)
     cout << vec[n] << " ";
  cout << endl;
}

int main ()
{ vector< int > v1(3,33);
    // tres elementos con valor 33
  vector< int > v2(v1);
    // una copia idéntica de v1

  mostrar(v1);
  v1.pop_back();
    // elimina el último elemento
  mostrar(v1);

  mostrar(v2);
  v2.push_back(88);
    // agrega un elemento al final
  mostrar(v2);

  return (0);
}
```

```
Cantidad de elementos: 3
Valores: 33 33 33
Cantidad de elementos: 2
Valores: 33 33
Cantidad de elementos: 3
Valores: 33 33 33
Cantidad de elementos: 4
Valores: 33 33 33 88;
```

Clase *string*

Una cadena de caracteres mediante un arreglo estático puede ser útil, pero es bastante limitada y tedioso su uso. Es simplemente un grupo de caracteres en la memoria, pero si se quiere hacer algo con ellos se debe tener en cuenta hasta los más pequeños detalles.

La clase *string* del estándar de C++ está diseñada para ocuparse (y ocultar) todas las manipulaciones a bajo nivel de las cadenas de caracteres. Estas manipulaciones han sido una fuente constante de errores y pérdida de tiempo desde la aparición del lenguaje C.

Para usar strings se debe incluir el archivo de encabezamiento de C++ <string>. Gracias a la sobrecarga de operadores, la sintaxis del uso de strings es bastante intuitiva como se ve en el siguiente código:

```
#include <string>
#include <iostream>
using namespace std;

int main() {
  string s1, s2;                    // creación de strings vacías
  string s3 = "Hola Mundo!";        // inicialización
```

```
  string s4("Hoy es");              // otra inicialización
  s2 = "de Marzo";                  // asignación de string
  s1 = s3 + " " + s4;               // combinando strings
  s1 += " 8 ";                      // agregando partes a un string
  cout << s1 + s2 + "!" << endl;
}
```

Los primeros dos strings, s1 y s2, comienzan vacíos, mientras que s3 y s4 muestran dos formas equivalentes para inicializar un objeto de tipo string en base a cadenas de caracteres (también se puede inicializar un string a partir de otro string).

Las asignaciones a los objetos de tipo string se realizan con '='. Esto reemplaza el contenido previo del string con lo que se encuentre del lado derecho del signo igual, y no es necesario preocuparse por lo que ocurre con el contenido previo – esto se maneja automáticamente. Para combinar strings simplemente se utiliza el operador '+' , el cual también permite combinar cadenas de caracteres con strings. Si es necesario agregar un carácter o un string a otro string, se puede usar el operador '+='. Finalmente, cabe destacar que los flujos de datos de entrada y salida (*iostreams*) ya saben qué hacer con los strings, así que únicamente es necesario enviar un string (o una expresión que produzca un string, como por ejemplo con la expresión s1 + s2 + "!") directamente al *cout* para mostrarlo en pantalla, por ejemplo.

Ejemplo de ingreso por teclado

En el código siguiente se puede observar lo simple que es su uso:

```
int main(){
string s;
cout << "Ingrese nombre y apellido";
getline(cin,s);
cout << s;
}
```

con *getline* se solicita el ingreso de una cadena de caracteres que según la cantidad ingresada tomará automáticamente la memoria necesaria.

Con cin se está indicando que será por el teclado. Por defecto terminará el ingreso cuando se pulse la tecla "enter" (esto puede ser cambiado en la llamada a getline especificando un tercer parámetro).

En el siguiente ejemplo se ingresa un número a una objeto string y se realiza una de las tantas formas que existen para almacenarlo en una variable entera.

```
int main(){
string s;
int i;
cout << "Ingrese un número entero";
getline(cin,s);
a=atoi(s.c_str());
cout << a;
}
```

Como la función atoi() transforma a entero una cadena de caracteres (no permite como parámetro un objeto string), se utilizó el método c_str() de la clase string para poder pasar el parámetro correcto.

Antiguas cadenas de caracteres (obsoleto)

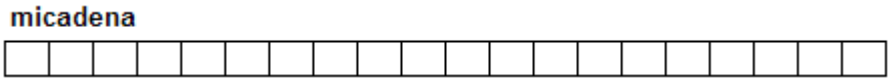
Antes, cuando la clase string no existía, la manipulación de caracteres se realizaba con las antiguas cadenas de caracteres de muy difícil uso y fuente de muchos errores y es probable que en algunos programas todavía se encuentren fragmentos de código que hagan uso de las mismas.

Las cadenas de caracteres pueden ser vistas como arreglos de tipo char, que son sucesiones de datos de tipo char. Recordar que este tipo de dato (char) es usado para almacenar un caracter, por esta razón los arreglos de este tipo se usan generalmente para hacer cadenas de caracteres.

Por ejemplo, el siguiente arreglo (o cadenas de caracteres):

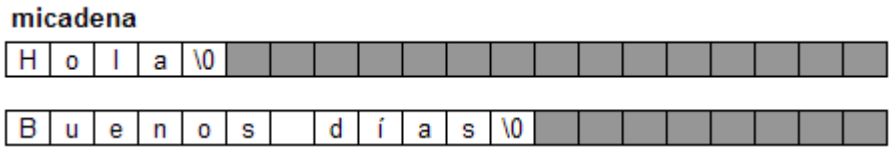
```
char micadena [20];
```

puede almacenar una cadena de hasta 20 caracteres de largo, que puede imaginarse de la siguiente manera:



Este máximo de 20 caracteres no siempre debe ser llenado completamente. Por ejemplo, micadena podría almacenar en algún momento en el programa tanto la cadena de caracteres *"Hola"* o la cadena *"Buenos días"*. Por consiguiente, dado que el arreglo de caracteres puede almacenar cadenas más cortas que su largo máximo, existe una convención para finalizar el contenido válido de una cadena con el *caracter nulo*, que puede ser escrito por el caracter **'\0'**.

Se puede almacenar en micadena (un arreglo de 20 elementos de tipo char) almacenando las cadenas de caracteres *"Hola"* y *"Buenos días"* de la siguiente manera:



Observar que después del contenido válido se incluye un caracter nulo ('\0') para indicar el fin de la cadena. Los casilleros en gris representan valores indeterminados. También se puede observar que debido a la necesidad de almacenar al final de la cadena el caracter nulo, la cantidad de caracteres útiles disponibles serán 19 en este caso, es decir, uno menos del largo declarado.

Inicialización de cadenas de caracteres

Dado que las cadenas de caracteres son arreglos comunes, siguen sus mismas reglas. Por ejemplo, inicializar una cadena de caracteres con valores predeterminados se puede hacer de la misma forma que cualquier otro arreglo:

```
char micadena [] = { 'H', 'o', 'l', 'a', '\0' };
```

En este caso declaramos una cadena de caracteres (arreglo) de 5 elementos de tipo char inicializado con los caracteres que componen la palabra “Hola” más un caracter nulo '\0'.

Sin embargo, las cadenas de caracteres tienen una manera adicional para inicializar sus valores: usando cadenas constantes.

En las expresiones usadas en ejemplos de capítulos previos ya han aparecido varias veces constantes que representan cadenas de caracteres completas. Estas se especifican entre comillas dobles ("), por ejemplo:

```
"el resultado es: "
```

es una cadena constante.

A diferencia de las comillas simples (') que permiten especificar un solo caracter como constante, las comillas dobles (") permiten especificar constantes conformadas por una sucesión de caracteres. Estas cadenas encerradas entre comillas dobles siempre tienen un caracter nulo ('\0') automáticamente adosado al final.

Así pues, se puede inicializar la cadena micadena con valores de estas dos formas:

```
char micadena [] = { 'H', 'o', 'l', 'a', '\0' };
char micadena [] = "Hola";
```

En ambos casos el arreglo o cadena de caracteres micadena es declarado con un tamaño de 5 caracteres (elementos de tipo char): los 4 caracteres que componen la palabra "Hola" más un caracter nulo al final ('\0') que especifica el final de la cadena, y es agregado automáticamente al utilizar comillas dobles (") en el segundo caso.

Antes de proseguir, notar que la asignación de múltiples constantes, como las encerradas entre comillas dobles ("), a arreglos son válidas únicamente al inicializar el arreglo, o sea, al momento de ser declarado. Expresiones como éstas:

```
micadena = "Hola";
micadena [] = "Hola";
```

no son válidas para arreglos, como tampoco serían:

```
micadena = { 'H', 'o', 'l', 'a', '\0' };
```

Recordar que se puede asignar una constante múltiple a un arreglo sólo al momento de inicializarlo. La razón será comprensible cuando se vea el capítulo dedicado a punteros, dado que en ese momento se aclarará que un arreglo es simplemente un puntero constante apuntando a un determinado bloque de memoria. Y debido a su condición de constante, al arreglo en sí no puede asignársele ningún valor, pero se puede asignar valores a cada uno de los elementos por separado.

El momento de inicializar un arreglo es un caso especial, dado que no es una asignación, aunque se use el signo igual (**=**).

Asignación de valores a cadenas

Así, dado que el *lvalue* (valor izquierdo) de una asignación puede ser únicamente un elemento del arreglo y no el arreglo entero, será válido asignar una cadena de caracteres a un arreglo de tipo char usando el siguiente método:

```
micadena[0] = 'H';
micadena[1] = 'o';
micadena[2] = 'l';
micadena[3] = 'a';
micadena[4] = '\0';
```

Pero este no parece ser un método demasiado práctico. Generalmente para asignar valores a un arreglo, y más específicamente para asignarle una cadena de caracteres, existen funciones como *strcpy* (***str**ing **cpy***) definida en la biblioteca *cstring* (string.h) y puede ser llamada de la siguiente forma:

```
strcpy (cadena1, cadena2);
```

Esta función copia el contenido de *cadena2* en *cadena1*. La *cadena2* puede ser un arreglo, un puntero o una cadena constante, así que la siguiente línea sería una forma válida para asignar la cadena constante *"Hola"* a *micadena*:

```
strcpy (micadena, "Hola");
```

Por ejemplo:

```
// asignación a cadena
#include <iostream>
#include <cstring>
using namespace std;

int main ()
{
  char MiCatedra [20];
  strcpy (MiCatedra,"Computación 2");
  cout << MiCatedra;
  return (0);
}
```

```
Computación 2
```

Observar que se tuvo que incluir el archivo <cstring> en el encabezado (header) para poder usar la función strcpy.

Aun así, siempre se puede usar una función simple como la siguiente fijar_valor con la misma utilidad que strcpy:

```
// asignar valores a cadena 2
#include <iostream>
using namespace std;

void fijar_valores (char salida [], char ingreso [])
{
  int n=0;
  do {
    salida[n] = ingreso[n];
  } while (ingreso[n++] != '\0');
}

int main ()
{
  char MiCatedra [20];
  fijar_valores (MiCatedra,"Computación 2");
  cout << MiCatedra;
  return (0);
}
```

```
Computación 2
```

Otro método frecuentemente utilizado para asignar valores a un arreglo es usar directamente el flujo de entrada (cin). En este caso el valor de la cadena es asignado por el usuario durante la ejecución del programa.

Cuando cin es utilizado con cadenas de caracteres generalmente se utiliza con el método getline, que puede ser llamado siguiendo el prototipo a continuación:

```
cin.getline ( char buffer[], int largo, char delimitador = '\n');
```

buffer es la dirección donde se va a almacenar la entrada (como un arreglo, por ejemplo), largo es el largo máximo del buffer (el tamaño del arreglo) y delimitador es el caracter utilizado para indicar la finalización de la entrada del usuario, que por defecto es el caracter de nueva línea ('\n') pero que puede ser cambiado a cualquier otro caracter. No confundir con el delimitador '\0'.

El siguiente ejemplo repite lo que se ingrese por el teclado. Es bastante sencillo pero sirve como ejemplo para usar cin.getline con cadenas:

```
// uso de cin para ingresar cadenas
#include <iostream>
using namespace std;

int main ()
{
  char MiBuffer [100];
  cout << "¿ Cuál es tu nombre ? ";
  cin.getline (MiBuffer,100);
  cout << "Hola " << MiBuffer << ".\n";
  cout << "¿Cuál es tu música favorita? ";
  cin.getline (MiBuffer,100);
  cout << "El " << MiBuffer
       << " también me gusta.\n";
  return (0);
}
```

```
¿ Cual es tu nombre ? Juan
Hola Juan.
¿Cuál es tu música favorita?
Bolero
El Bolero también me gusta.
```

En ambas llamadas a cin.getline se usa el mismo identificador MiCadena. Lo que el programa hace en la segunda llamada es simplemente sobrescribir el contenido previo de buffer por el nuevo contenido, ubicando correctamente a '\0'.

También se puede usar el operador de extracción (>>) para recibir datos directamente desde el flujo de entrada estándar. Este método también puede ser usado en vez de cin.getline con cadenas de caracteres. En el programa anterior se podría haber escrito:

```
cin >> micadena;
```

esto hubiera funcionado, pero esta manera tiene ciertas limitaciones que cin.getline no posee:

- Puede recibir sólo palabras (no oraciones completas) dado que este método usa como delimitador de finalización de entrada del usuario cualquier aparición de un caracter en blanco, incluyendo espacios, tabulaciones, caracteres de nueva línea y retorno de carro.

- No se permite especificar un tamaño para el buffer, lo que causa que el programa sea inestable en caso de que la entrada del usuario sea más larga que el arreglo que debe contenerlo.

Por estas razones es recomendable usar cin.getline en vez del operador de extracción (>>).

Conversión de cadenas a otros tipos

Debido a que una cadena de caracteres puede contener representaciones de otros tipos de datos, como por ejemplo números, puede ser útil traducir ese contenido a una variable de tipo numérico. Una cadena podría contener "1977", pero esto es una secuencia de 5 caracteres, y no es tan fácil convertirlos a un solo tipo de dato entero.

La biblioteca *cstdlib* (stdlib.h) provee tres funciones útiles para este propósito:

atoi: convierte una cadena a un tipo de dato int.
atol: convierte una cadena a un tipo de dato long.
atof: convierte una cadena a un tipo de dato float.

Todas estas funciones admiten un sólo parámetro y retornan un valor del tipo requerido (int, long o float). Estas funciones combinadas con el método getline de cin son una manera más confiable para obtener la entrada del usuario cuando se requiere el ingreso de un número que el clásico método cin >> :

```
// cin y funciones ato*
#include <iostream>
#include <cstdlib>
using namespace std;

int main ()
{
  char MiBuffer [100];
  float precio;
  int cantidad;
  cout << "Ingrese el precio unitario: ";
  cin.getline (MiBuffer,100);
  precio = atof (MiBuffer);
  cout << "Ingrese la cantidad: ";
  cin.getline (MiBuffer,100);
  cantidad = atoi (MiBuffer);
  cout << "Total: " << precio*cantidad;
  return (0);
}
```

```
Ingrese el precio unitario: 2.75
Ingrese la cantidad: 21
Total: 57.75
```

Conversión entre valores numéricos y *strings*

La biblioteca <iostream> permite convertir casi cualquier cosa en un string (y a la inversa) usando las clases *istringstream* y *ostringstream*. El siguiente ejemplo convierte un string a un número de tipo float, pero podría haberse convertido a cualquier otro tipo numérico:

```
//conversión de string a número
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

int main () {
  float valor_numerico; string valor_string;

  cout << "Ingrese el string que desea
```

```
Ingrese el string que desea
convertir a número: 4.0
La raíz cuadrada del número
es: 2
```

```
          convertir a número: ";
  cin >> valor_string;  cin.get();

  istringstream flujo;
  flujo.str(valor_string);
  flujo >> valor_numerico;

  cout << "La raíz cuadrada del número es: "
       << sqrt(valor_numerico)
       << endl;   cin.get();

  return (0);
}
```

En el ejemplo anterior, se ingresa un número desde el teclado, el cual es almacenado en un objeto de tipo string. Luego se crea un objeto de tipo *istringstream* (podría traducirse como "flujo de strings de entrada") llamado *flujo*. Mediante la función str() se envía el string que se desea convertir "hacia adentro" del flujo, y usando el operador sobrecargado >>, el flujo devuelve su contenido hacia la variable valor_numerico, convirtiendo dicho contenido automáticamente de acuerdo al tipo de variable que va a recibir lo que había dentro del flujo, en este caso, en un dato de tipo float. Para ver que el dato resultado es un número, se utiliza la función sqrt().

En el próximo ejemplo se realiza la operación inversa: se convierte un dato numérico a un objeto de tipo string, esta vez usando la clase *ostringstream*.

```
//conversión de número a string
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

int main () {

  float valor_numerico;  string valor_string;

  cout << "Ingrese el número que desea
           convertir a string: ";
  cin >> valor_numerico; cin.get();

  ostringstream flujo;
  flujo << valor_numerico;
  valor_string = flujo.str();

  cout << "El string resultante es "
       << valor_string
       << " y posee "  << valor_string.size()
       << " caracteres";
  cin.get();

  return (0);
}
```

```
Ingrese el número que desea
convertir a string: 42
El string resultante es 42 y
posee 2 caracteres
```

En este caso, el proceso es al revés: se envía un dato numérico a un flujo de tipo ostringstream (algo así como "flujo de strings de salida") y se extrae un string mediante la función str(). Para demostrar que la conversión fue exitosa, se utiliza la función size().

Funciones para cadenas

La biblioteca cstring (string.h) define funciones para realizar manipular cadenas en C (como strcpy antes vista). A continuación se muestra una lista con las más utilizadas:

strcat: char* strcat (char* destino, const char* fuente);
Agrega la cadena fuente al final de la cadena destino. Devuelve la cadena destino.

strcmp: int strcmp (const char* cadena1, const char* cadena2);
Compara las cadenas cadena1 y cadena2. Devuelve cero si ambas cadenas son iguales.

strcpy: char* strcpy (char* destino, const char* fuente);
Copia el contenido de fuente en destino. Devuelve destino.

strlen: size_t strlen (const char* cadena);
Retorna el largo de cadena.
Nota: tener en cuenta que char* es lo mismo que char[ ].

Tipos definidos por el programador

C++ permite definir nuestros tipos de datos basados en otros tipos ya existentes usando la instrucción *typedef*, cuya forma es:

```
typedef tipo_existente nuevo_tipo;
```

donde tipo_existente es un tipo fundamental de C++ o cualquier otro tipo definido y nuevo_tipo es el nombre que recibirá el nuevo tipo creado. Algunos ejemplos son:

```
typedef char C;
typedef unsigned int PALABRA;
typedef char * cadena;
typedef char campo[50];
```

En este caso se han definido 4 nuevos tipos: *C, PALABRA, cadena* y *campo* como *char, unsigned int, char** y *char[50]* respectivamente. Estos tipos se pueden usar luego como tipos válidos:

```
C un_char, otro_char, *punt_a_char1;
PALABRA mi_palabra;
cadena punt_a_char2;
campo nombre;
```

typedef puede ser útil para definir un tipo que es usado repetidamente dentro de un programa y es posible que sea necesario cambiarlo en versiones posteriores.

Estructuras de datos

Una estructura de datos es un conjunto de diversos tipos de datos agrupados juntos bajo una única declaración. Su forma es la siguiente:

```
struct nombre_modelo
{ tipo1 elemento1;
  tipo2 elemento2;
  tipo3 elemento3;
  .
  .
} nombre_instancia;
```

donde *nombre_modelo* es un nombre para el modelo del tipo de estructura y el parámetro opcional *nombre_instancia* es un identificador válido para una instancia de la estructura. Dentro de las llaves { } están los tipos y los sub-identificadores (campos) correspondientes a los elementos que componen la estructura.

Si la definición de la estructura incluye el parámetro nombre_modelo (opcional), ese parámetro se convierte en un tipo de dato válido equivalente a la estructura. Por ejemplo:

```
struct producto
{
  char nombre[30];
  float precio;
};

producto manzana;
producto naranja, melon;
```

Primero se definió el modelo de estructura producto con dos campos: nombre y precio, cada uno de un tipo diferente. Luego se usó el nombre del tipo de estructura (producto) para declarar tres instancias de ese tipo: manzana, naranja y melon.

Una vez definido, producto se ha convertido en un nuevo tipo de dato válido como los fundamentales int, char o short. Y es por esto que se ha podido declarar variables de este tipo.

El campo opcional nombre_instancia que puede ir al final de la declaración de la estructura sirve para declarar directamente instancias del tipo de la estructura. Por ejemplo, para declarar los objetos manzana, naranja y melon se podría haber escrito:

```
struct producto
{
  char nombre[30];
  float precio;
} manzana, naranja, melon;
```

Aún más, en casos como el anterior en el cual se aprovechó la declaración del modelo de estructura para declarar instancias de ese tipo, el parámetro nombre_modelo (en este caso producto) se hace opcional. Sin embargo, si nombre_modelo no es incluido no será posible declarar más instancias del mismo tipo. Es importante diferenciar claramente que modelo es el tipo de dato y que una instancia de la estructura es la variable.

Una vez declarada las tres instancias de un determinado modelo de estructura (manzana, naranja y melon) se puede operar con los campos que los conforman. Para esto se debe usar un punto (.) insertado entre el nombre de la instancia y el nombre del campo. Por ejemplo, se puede operar con cualquiera de estos elementos como si fueran variables estándar de sus respectivos tipos:

```
manzana.nombre
manzana.precio
naranja.nombre
naranja.precio
melon.nombre
melon.precio
```

cada uno de su correspondiente tipo de dato: manzana.nombre, naranja.nombre y melon.nombre son de tipo char[30], y manzana.precio, naranja.precio y melon.precio son de tipo float.

A continuación se puede ver otro ejemplo sencillo:

```
// ejemplo sobre estructuras
#include <iostream>
#include <string>
using namespace std;

struct pelicula_t
{ string titulo;
  int anio;
} mia, tuya;

void mostrar_pelicula (pelicula_t pelicula)
{ cout << pelicula.titulo
       << " ("
       << pelicula.anio
       << ")" << endl;
}
```

```
Ingrese titulo: Sexto sentido
Ingrese el anio: 1999
Mi película es: Alien (1979)
y la tuya: Sexto sentido (1999)
```

```
int main ()
{ string anio;

  mia.titulo="Alien";
  mia.anio=1979;

  cout << "Ingrese titulo: ";
  getline(cin,tuya.titulo);
  cout << "Ingrese el anio: ";
  getline(cin,anio);
  tuya.anio=atoi(anio.c_str());

  cout << "Mi película es: ";
  mostrar_pelicula(mia);
  cout << "y la tuya: ";
  mostrar_pelicula(tuya);

  return (0);
}
```

El ejemplo muestra como se pueden usar los elementos de una estructura de datos y la estructura en sí misma como variables normales. Por ejemplo mia.anio es una variable válida de tipo int, y mia.titulo es un arreglo válido de tipo string. Las estructuras *tuya* y *mia* son también tratadas como variables válidas del tipo pelicula cuando son pasadas a la función *mostrar_pelicula()*. Por lo tanto, una de las ventajas más importantes de las estructuras de datos es que se puede realizar una referencia tanto a sus elementos individuales o a toda la estructura como un bloque.

Nota: la nomenclatura cuando las estructuras de datos son usadas con punteros es la siguiente: *película->titulo;* que es idéntica a *(*película).titulo;*

Las estructuras podrían ser usadas para construir una base de datos, especialmente si se considera la posibilidad de trabajar con arreglos de estructuras.

```
// arreglo de estructuras
#include <iostream>
#include <string>
#include <vector>
using namespace std;

struct pelicula_t
{ string titulo;
  int anio;
};

void mostrar_pelicula (pelicula_t pelicula)
{
  cout << pelicula.titulo
       << " ("
       << pelicula.anio
       << ")" << endl;
}

int main ()
{ string buff; unsigned n, N;
  vector< pelicula_t > films;
```

```
Cuantas peliculas ingresara? 5
Ingresar titulo: Alien
Ingresar anio: 1979
Ingresar titulo: Blade Runner
Ingresar anio: 1982
Ingresar titulo: Matrix
Ingresar anio: 1999
Ingresar titulo: Rear Window
Ingresar anio: 1954
Ingresar titulo: Taxi Driver
Ingresar anio: 1975

Usted ha ingresado:
Alien (1979)
Blade Runner (1982)
Matrix (1999)
Rear Window (1954)
Taxi Driver (1975)
```

```
  cout << "Cuantas peliculas ingresara? ";
  getline(cin,buff);
  N=atoi(buff.c_str());
  films.resize(N);

  for (n=0; n<N; n++)
  { cout << "Ingresar titulo: ";
    getline(cin,films[n].titulo);
    cout << "Ingresar anio: ";
    getline(cin,buff);
    films[n].anio=atoi(buff.c_str());
  }

  cout << endl
       << "Usted ha ingresado:"
       << endl;
  for (n=0; n<N; n++)
    mostrar_pelicula(films[n]);

  return (0);
}
```

Estructuras anidadas

Las estructuras también pueden ser anidadas dado que un elemento válido de una estructura puede ser a su vez otra estructura:

```
struct pelicula_t
{
  string titulo;
  int anio;
};

struct amigos
{
  string nombre;
  string email;
  pelicula_t pelicula_favorita;
} jose, daniel;
```

Así, después de la declaración previa, se podría usar las siguientes expresiones:

```
jose.nombre
daniel.pelicula_favorita.titulo
jose.pelicula_favorita.anio
```

Uniones

Las uniones permiten a una misma porción de memoria ser accedida a través de diferentes tipos de datos, estando de hecho todos ellos en el mismo sitio en la memoria. Su declaración y uso es similar al de las estructuras pero su funcionamiento es totalmente diferente:

```
union nombre_modelo {
  tipo1 elemento1;
  tipo2 elemento2;
```

```
  tipo3 elemento3;
  .
  .
} nombre_variable;
```

Todos los elementos de la declaración de *union* ocupan la misma cantidad de memoria. Su tamaño corresponde al mayor elemento de la declaración. Por ejemplo:

```
union mis_tipos
{
  char c;
  int i;
  float f;
} tipo;
```

define tres elementos:

```
tipo.c
tipo.i
tipo.f
```

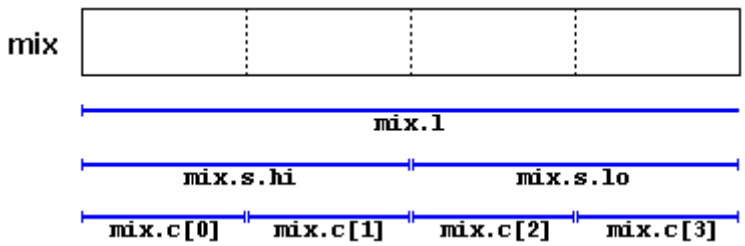
cada uno de un tipo de dato diferente. Como todos ellos se refieren a la misma dirección de memoria la modificación de uno de los elementos modificará el valor de todos ellos.

Uno de los usos que una *union* puede tener es unir un tipo elemental con un arreglo o estructura de elementos más pequeños.

Por ejemplo:

```
union mix_t
{ long l;
  struct
  { short hi;
    short lo;
  } s;
  char c[4];
} mix;
```

define tres nombres que permite acceder al mismo grupo de 4 bytes: mix.l, mix.s y mix.c, los cuales se pueden usar de acuerdo a como se quiera acceder a él, como long, short o char respectivamente. Esquemáticamente:



Uniones anónimas

En C++ se incorpora la opción de que las uniones sean anónimas. Si se incluye una unión en una estructura sin ningún nombre de objeto (el que va después de las llaves) la unión será anónima y se podrá acceder a los elementos directamente por su nombre.

Por ejemplo, observar las diferencias entre las siguientes dos declaraciones:

Estructura con unión simple	Estructura con unión anónima

struct
{ string titulo;
  string autor;
  union
  { float dolares;
  int yenes;
  } precio;
} libro;
	struct
	{ string titulo;
	  string autor;
	  union
	  { float dolares;
	  int yenes;
	  };
	} libro;

La única diferencia entre estos dos fragmentos de código es que en el primero se ha dado un nombre a las variables de la unión (precio) y en el segundo no. La diferencia está en el acceso a los miembros dolares y yenes de una variable unión. En el primer caso es:

```
libro.precio.dolares
libro.precio.yenes
```

mientras que en el segundo es:

```
libro.dolares
libro.yenes
```

Debido a que es una unión, los campos *dolares* y *yenes* ocupan el mismo sitio en memoria y no pueden ser usados para almacenar dos valores diferentes, lo cual significa que se puede incluir un precio en dólares o yenes, pero no ambos.

Enumeraciones

Las enumeraciones sirven para crear tipos de datos que contengan algo diferente que no está limitado ni a constantes numéricas o alfanuméricas ni a verdadero o falso (true y false). Su forma es la siguiente:

```
enum nombre_modelo
{ valor1,
  valor2,
  valor3,
  .
  .
} nombre_variable;
```

Por ejemplo, se puede crear un nuevo tipo de variables llamado *colores* para almacenar colores, con la siguiente declaración:

```
enum colores {negro, azul, verde, cian, rojo, violeta, amarillo, blanco};
```

Hay que observar que no se incluyó ningún tipo fundamental en la declaración. Para decirlo de alguna manera, el tipo colores es un nuevo tipo de dato que no está basado en ningún otro tipo existente. Los

únicos valores posibles que una variable de este tipo puede tomar son los que se incluyeron dentro de las llaves { }. Por ejemplo, una vez declarada la enumeración colores las siguientes expresiones serán válidas:

```
colores mi_color;
mi_color = azul;
if (mi_color == verde) mi_color = rojo;
```

El nuevo tipo de datos creado es en realidad compilado y tratado internamente como un entero y sus valores posibles son cualquier tipo de constantes enteras que se especifiquen. Si éstas no se especifican, el valor entero equivalente internamente al primer valor posible es 0 y los siguientes van aumentando de 1 en 1. Por lo tanto, en el tipo colores definido anteriormente, negro será equivalente a 0, azul será equivalente a 1, verde a 2 y así sucesivamente.

Si se especifica explícitamente un valor entero para alguno de los posibles valores del tipo enumerado (por ejemplo el primero) los siguientes valores serán incrementados a partir de éste, por ejemplo:

```
enum meses { enero=1, febrero, marzo, abril,
             mayo, junio, julio, agosto,
             septiembre, octubre, noviembre, diciembre} mes;
```

en este caso, la variable *mes* del tipo enumerado meses puede contener cualquiera de los 12 posibles valores que van desde enero a diciembre y que son equivalentes a los números desde el 1 al 12 (como normalmente se indican) y no entre 0 y 11 debido a que se ha definido enero igual a 1.

Archivos

Entrada/Salida con archivos

C++ ofrece soporte para manejo de entrada y salida de datos de archivos a través de las siguientes clases:

ofstream: clase utilizada para operaciones de escritura (derivada de *ostream*)
ifstream: clase utilizada para operaciones de lectura (derivada de *istream*)
fstream: clase utilizada para ambos tipos de operaciones (derivada de *iostream*)

Abrir un archivo

La primera operación que se realiza generalmente con un objeto de estas clases es asociarlo con un archivo real para leer y escribir en él. El archivo abierto es representado dentro del programa como un objeto utilizado para el flujo de datos (una instancia de una de estas clases) y cualquier entrada o salida de datos a través de este flujo será aplicada al archivo físico.

Para abrir un archivo usando un objeto perteneciente a una de estas clases se utiliza la función miembro *open( )*:

```
void open (const char * nombre_archivo, openmode modo);
```

donde nombre_archivo es una cadena de caracteres que representa el nombre del archivo que será abierto y `modo` es una combinación de las siguientes opciones (flags):

ios::in	Abrir archivo para lectura
ios::out	Abrir archivo para escritura
ios::ate	Posición inicial: fin del archivo (para leer o escribir)
ios::app	Cada salida se agrega al final del archivo (sólo para escribir)
ios::trunc	Si el archivo existe, se borra su contenido
ios::binary	Modo binario

Estas opciones se pueden combinar usando el operador de bits *OR*: **|**. Por ejemplo, si queremos abrir el archivo "ejemplo.bin" en modo binario para agregar datos, podríamos hacerlo llamando a la función open() así:

```
ofstream archivo;
archivo.open ("ejemplo.bin", ios::out | ios::app | ios::binary);
```

La función miembro open() de las clases ofstream, ifstream y fstream incluyen todas un modo de apertura por defecto que varía según la clase:

Clase	Modo por defecto
ofstream	ios::out | ios::trunc
ifstream	ios::in
fstream	ios::in | ios::out

El valor por defecto sólo se aplica si la función es llamada sin especificar el parámetro *modo*. Si la función es llamada con cualquier valor en ese parámetro el modo por defecto es ignorado y no se combina con el parámetro especificado.

Dado que la primera tarea que se realiza en un objeto de las clases ofstream, ifstream y fstream es frecuentemente abrir un archivo, estas tres clases incluyen un constructor que llama directamente a la función open() y que posee los mismos parámetros que ésta. De esta forma, también se podría haber declarado el objeto previo y realizar la misma apertura del archivo escribiendo:

```
ofstream archivo ("ejemplo.bin", ios::out | ios::app | ios::binary);
```

Ambas formas de abrir un archivo son válidas.

Se puede chequear si un archivo ha sido abierto correctamente llamando a la función *is_open()*:

```
bool is_open();
```

que retorna un dato de tipo bool que toma el valor true en caso de que de hecho el objeto haya sido asociado correctamente con un archivo abierto, y de otro modo toma el valor false.

Cerrando un archivo

Al terminar las operaciones de lectura, escritura o consulta en un archivo se debe cerrar para poder tenerlo a disposición nuevamente. Para hacerlo se debe llamar a la función *close()*, que se encarga de vaciar la memoria temporal (*buffers*) y cerrar el archivo. Su forma de uso es simple

```
void close ();
```

Una vez que se llama a esta función miembro, el mismo objeto puede ser usado para abrir otro archivo, y el archivo anterior está disponible para ser abierto en otros procesos.

En caso de que un objeto sea destruido mientras está asociado con un archivo abierto, el destructor automáticamente llama a la función miembro close.

Archivos en modo texto

Las clases ofstream, ifstream y fstream derivan de ostream, istream y iostream respectivamente. Por esto los objetos de tipo fstream pueden usar los miembros de estas clases padres para acceder a los datos.
Generalmente, al trabajar con archivos de texto se utilizará los mismos miembros de estas clases que usamos en comunicación a través de la consola (*cin y cout*).

Una de las funciones más útiles de la biblioteca *iostream* es la función *getline( )*, que permite leer una línea (terminada, por defecto, por un carácter de nueva línea) y pasarla a un objeto de tipo string. El primer argumento es el objeto de tipo ifstream del cual estás leyendo y el segundo argumento es un objeto de tipo string al cual se pasará el contenido de la línea. Cuando la llamada a la función termina, el objeto string contendrá dicha línea.

El siguiente ejemplo copia de una manera muy simple el contenido de un archivo de texto en otro:

```
#include <string>
#include <fstream>
using namespace std;

int main() {
  ifstream in ("Entrada.txt"); // lectura
```

```
  ofstream out ("Salida.txt"); // escritura
  string s;

  while (getline(in, s))       // mientras haya líneas para leer
    out << s << endl;          // ... pasarla al otro archivo
}
```

En el ejemplo anterior no se ha indicado directamente el llamado a las funciones miembro close(), pero al destruirse los objetos, el llamado a sus destructores cerrarán los archivos.

La condición que incluye la instrucción while es, en este caso, un tanto peculiar. Mientras la expresión entre paréntesis (en este caso, *getline(in,s)*) produzca un resultado verdadero (true), la instrucción controlada por el ciclo while continuará ejecutándose. La función getline( ) retorna verdadero (true) si otra línea ha sido leída exitosamente, y falso (false) cuando se llega al final del archivo. Por lo tanto, el ciclo while anterior lee cada línea en el archivo de entrada y la "envía" al archivo de salida.

La función getline( ) lee todos los caracteres de una línea hasta que encuentra un carácter de nueva línea (el carácter de finalización puede cambiarse). Sin embargo, la función desecha el carácter de nueva línea y no lo almacena en el string. Así, si se desea que el archivo copiado luzca igual que el original, debemos agregar el carácter de nueva línea.

Otro ejemplo interesante es la copia de un archivo entero dentro de un solo string:

```
#include <string>
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  ifstream in ("Entrada.txt");
  string s, line;

  while (getline(in, line))
     s += line + '\n'; // no funciona con +endl;

  cout << s;
}
```

Debido a la naturaleza dinámica de los strings, no es necesario preocuparse acerca de reservar espacio en memoria para almacenar string; simplemente se puede seguir agregando cosas y el string seguirá expandiéndose para contener todo lo que se ponga dentro de él.

Una de las ventajas de poner un archivo entero dentro de un string es que la clase string tiene muchas funciones de búsqueda y manipulación que permiten modificar el archivo fácilmente. Sin embargo, tiene sus limitaciones. Primero que nada, es conveniente tratar un archivo como una colección de líneas en lugar de un gran bloque de texto. Por ejemplo, si se desea numerar las líneas es mucho más fácil si se tiene cada línea como un string por separado.

Para que la función getline() lea hasta un carácter específico, se debe ingresar un tercer parámetro. Supongamos por ejemplo que se tiene el siguiente archivo de texto (sin ninguna línea en blanco):

```
Juan Perez*12
Mario Porta*34
```

Donde los datos (campos) están separados por asteriscos. El código para leer correctamente es el siguiente:

```
#include <string>
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  ifstream in ("datos.txt");
  string campo;

  while (getline(in, campo,'*')) {
    cout << "Nombre: " << campo << endl;
    getline(in, campo);
    cout << "Edad: " << campo << endl;
  }
  cin.get();
  return (0);
}
```

Problemas con la función *eof()*

La función eof(), utilizada para determinar la finalización de un archivo al leerlo, puede no devolver un valor correcto dado que hay un último intento de lectura al llegar al final del archivo. Esto es, leer el último byte desde un archivo no cambia el estado de la función eof(), lo que causa que intente leer otro byte, provocando una lectura extra que puede causar diversos errores. Por ejemplo, suponga que la entrada se realiza desde el teclado - en este caso es totalmente imposible para la función eof() predecir sin el último caracter ingresado por el usuario es el último que éste ingresará.

Por ejemplo, el fragmento de código siguiente posee un error debido a que el eof() realiza una lectura más de la que supuestamente debería realizar, lo que provoca ciertos errores, como por ejemplo que el contador exceda su valor en 1:

```
//determinar el final de un archivo
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  int numero, contador;
  ifstream mi_archivo("ejemplo.dat");
  while (mi_archivo.eof()) //incorrecto!!
   {
   mi_archivo >> numero;
   cout << numero << " ";
   contador++;
   }
  cin.get();
  return (0);
}
```

La solución es:

```
//determinar el final de un archivo
#include <iostream>
#include <fstream>
using namespace std;
```

```
int main() {
  int numero, contador;
  ifstream mi_archivo("ejemplo.dat");
  while (mi_archivo >> numero) //correcto!!
   {
    cout << numero << " ";
    contador++;
   }
  cin.get();
  return (0);
}
```

Verificación de las opciones de estado

Además de eof() existen otras funciones miembro para verificar el estado del flujo de datos (todas retornan un valor de tipo bool):

bad()	Retorna true si ocurre una falla en las operaciones de lectura o escritura. Por ejemplo en caso de que tratemos de escribir en un archivo que no está abierto para escritura.
fail()	Retorna true en los mismos casos que bad() y además en caso de que ocurra un error de formato, como tratar de leer un número entero y obtener una letra.
eof()	Retorna true si el archivo abierto ha llegado a su fin.
good()	Es el más genérico: Retorna false en los mismos casos en los que al llamar a las funciones previas hubiéramos obtenido true.

Para resetear las opciones de estado fijadas con anterioridad por las funciones previas se puede usar la función clear( ), sin parámetros.

Archivos binarios

Muchos sistemas operativos distinguen entre archivos de texto y archivos binarios. Por ejemplo, en MS-DOS, los archivos de texto sólo permiten almacenar caracteres. En otros sistemas no existe tal distinción, todos los archivos son binarios. En esencia esto es más correcto, puesto que un archivo de texto es un archivo binario con un rango limitado para los valores que puede almacenar.

En general, usaremos archivos de texto para almacenar información que pueda o deba ser manipulada con un editor de texto. Un ejemplo es un archivo fuente C++. Los archivos binarios son más útiles para guardar información cuyos valores no estén limitados. Por ejemplo, para almacenar imágenes o bases de datos. Un archivo binario permite almacenar estructuras completas en las que se mezclen datos de cadenas con datos numéricos.

En realidad no hay nada que impida almacenar cualquier valor en un archivo de texto, el problema surge cuando se almacena el valor que el sistema operativo usa para marcar el fin de archivo en un archivo de texto. En Windows ese valor es 0x1A. Si abrimos un archivo en modo de texto que contenga un dato con ese valor, no nos será posible leer ningún dato a partir de esa posición. Si lo abrimos en modo binario, ese problema no existirá.

En archivos binarios, la entrada y salida de datos con formato, como con los operadores << y >> y funciones como getline, no tiene mucho sentido, aunque son operaciones perfectamente válidas.

Los archivos de flujo incluyen dos funciones para entrada y salida de datos secuencial: *write* y *read*. La primera (write) es una función de *ostream*, también heredada por *ofstream*. Y read es una función de *istream* y es heredada por *ifstream*. Los objetos de tipo *fstream* poseen ambas funcione. Sus prototipos son:

```
write ( char * buffer, streamsize tamanio );
read  ( char * buffer, streamsize tamanio );
```

donde buffer es la dirección de un bloque de memoria donde los datos a leer están almacenados, o donde los datos desean escribirse. El parámetro tamanio es un valor entero que especifica el número de caracteres a ser leídos/escritos desde/hacia el buffer.

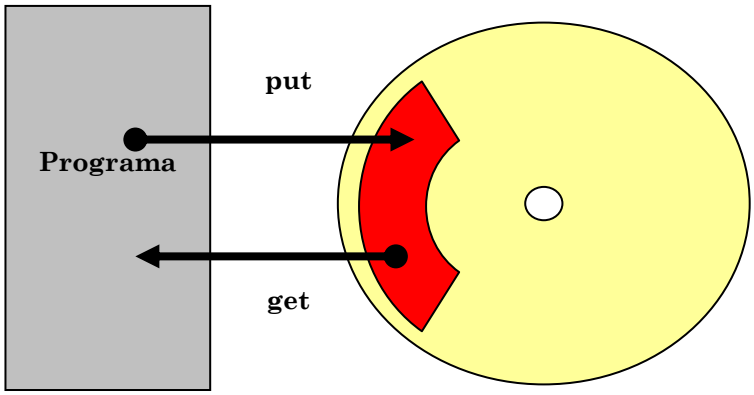
```
// Primer Ejemplo Archivos Binarios
#include <fstream>
#include <iostream>
using namespace std;

int main(void)
{ // abrir para leer y grabar.
  fstream leerGrabar("datos.dat",
                     ios::binary | ios::in |
ios::out);

  float f=2.2; int i=5;
  // se graba primero "f" y luego "i"
  leerGrabar.write((char *)&f,sizeof(f));
  leerGrabar.write((char *)&i,sizeof(i));

  f=0; i=0;
  leerGrabar.seekg(0,ios::beg);
  leerGrabar.read((char *)&f,sizeof(f));
  leerGrabar.read((char *)&i,sizeof(i));

  cout << f << "   " << i << endl;

  leerGrabar.seekg(0,ios::beg);
  f=0; i=0;
  // Error. No se lee como se grabó
  leerGrabar.read(&i,sizeof(i));
  leerGrabar.read(&f,sizeof(f));

  cout << f << "   " << i << endl;

  cin.get();
  leerGrabar.close();
  return (0);
}
```

```
2.2   5
7.00649e-45
1074580685
```

Punteros de flujo *get* y *put*

Todos los flujos de entrada y salida poseen, por lo menos, un puntero de flujo.

ifstream, como istream, tiene un puntero conocido como puntero *get* que apunta al elemento que será leido a continuación.



ofstream, como ostream, tiene un puntero llamado puntero *put* que apunta al lugar donde será escrito el siguiente elemento.

Finalmente fstream, como iostream, hereda ambos: get y put.

Estos punteros de flujo que apuntan a sitios de lectura o escritura dentro de un flujo de datos pueden ser leídos y/o manipulados usando las siguientes funciones:

tellg()* y *tellp()

Estas dos funciones no admiten parámetros y devuelven un valor de tipo long que es un número entero y representa la posición actual del puntero de flujo get (en el caso de tellg) o del puntero de flujo put (en el caso de tellp).

seekg()* y *seekp()

Este par de funciones sirve respectivamente para cambiar la posición de los punteros de flujo *get* y *put*. Ambas funciones están sobrecargadas con dos prototipos diferentes:

```
seekg ( pos_type posicion );
seekp ( pos_type posicion );
```

Usando este prototipo el puntero de flujo es cambiado a una posición absoluta contando desde el inicio del archivo. El tipo requerido es el mismo que retornan las funciones tellg y tellp.

```
seekg ( off_type desplazamiento, seekdir dirección );
seekp ( off_type desplazamiento, seekdir dirección );
```

Usando este prototipo, un desplazamiento desde un punto concreto determinado por el parámetro *dirección* puede ser especificado. Éste puede ser:

ios::beg	Desplazamiento especificado desde el principio del flujo (por defecto)
ios::cur	Desplazamiento especificado desde la posición actual del puntero
ios::end	Desplazamiento especificado desde el final del flujo

Los valores de ambos punteros get y put se cuentan de diferente forma para archivos de texto y para archivos binarios, dado que en modo texto pueden ocurrir algunas modificaciones a la apariencia de ciertos caracteres especiales. Por esta razón es aconsejable usar sólo el primer prototipo de seekg y seekp con archivos abiertos en modo texto y siempre usar sin modificar los valores devueltos por tellg o tellp.

Con archivos binarios, se puede usar libremente todas las implementaciones para estas funciones, dado que no debería ocurrir ningún comportamiento inesperado.

A continuación se encuentra un ejemplo de lectura-escritura simultánea en un archivo binario:

```
// lectura-escritura en archivos binarios
#include <fstream>
#include <iostream>
using namespace std;

int main(void)
{
  fstream LeerEscribir("valores.dat",
         ios::binary | ios::in | ios::out);

  float f=8.1; int i=4;
  LeerEscribir.write( &f,sizeof(f) );
  LeerEscribir.write( &i,sizeof(i) );

  LeerEscribir.seekg( -sizeof(i) );
  LeerEscribir.read ( &i,sizeof(i) );
  cout << i << endl;

  LeerEscribir.seekp(sizeof(f),ios::beg);
  i=10;
  LeerEscribir.write( &i,sizeof(i) );

  LeerEscribir.seekg( 0,ios::beg );
  LeerEscribir.read ( &f,sizeof(f) );
  LeerEscribir.read ( &i,sizeof(i) );
  cout << f << " " << i << endl;

  LeerEscribir.close();
  cin.get();

  return (0);
}
```

```
4
8.1 10
```

El siguiente ejemplo usa las funciones anteriores para obtener el tamaño de un archivo binario:

```
// obteniendo el tamaño de un archivo
#include <iostream>
#include <string>
#include <fstream>
using namespace std;
string nombre_archivo = "ejemplo.txt";

int main () {
  long l,m;
  ifstream file (nombre_archivo.c_str(),
                 ios::in|ios::binary);
  l = file.tellg();
  file.seekg (0, ios::end);
  m = file.tellg();
  file.close();
  cout << "El tamaño de " << nombre_archivo;
  cout << " es " << (m-l)
```

```
El tamaño de ejemplo.txt es
40 bytes.
```

```
        << " bytes." << endl;
  return (0);
}
```

Si todos los valores de un archivo binario son del mismo tipo, se pueden leer todos ellos de la siguiente manera:

```
archi.seekg(0,ios::end);
num_valores=archi.tellg()/sizeof(valor); // cantidad de valores
archi.seekg(0,ios::beg);
for (i=0;i<num_valores;i++)
{ archi.read((char*)&v,sizeof(valor));
  cout << v << endl;
}
```

Problemas con la función eof( )

En arcivos binarios también se debe tener cuidado con el uso de la función *eof( )*.

Por ejemplo, si se quiere mostrar dos veces un archivo binario así:

```
  archi.seekg(0, ios::beg);
  while(  archi.read(&valor, sizeof(valor)) )
     cout << valor << endl;

  archi.seekg(0, ios::beg);
  while(  archi.read(&valor, sizeof(valor)) )
     cout << valor << endl;
```

o usando "while(!archi.eof())" hay problemas. Con la siguiente solución funciona correctamente porque se lee exactamente lo que queda.

```
#include <iostream>
#include <fstream>

using namespace std;

int main ()
{ int i, v, N;
   fstream archi ("borrar.dat", ios::binary|ios::in|ios::out|ios::trunc);

   cout << "Graba cinco valores" << endl;
   for (i=0; i<5; i++) archi.write((char*)&i,sizeof(i));
     cout << "El puntero a get ( tellg() ) está en: "
          << archi.tellg() << endl;

   int valor;
   cout << "Leera' los cinco valores" << endl;

   archi.seekg(0,ios::end); N=archi.tellg()/sizeof(valor);
   archi.seekg(0,ios::beg);

   cout << "El puntero a get ( tellg() ) está en: "
        << archi.tellg() << endl;

   for (i=0; i<N; i++)
   { archi.read((char*)&v,sizeof(valor));
```

```
      cout << v << endl;
   }

   cout << "El puntero a get ( tellg() ) está en: "
        << archi.tellg() << endl;

   cout << "Leerá nuevamente los cinco valores" << endl;
   archi.seekg(0,ios::end); N=archi.tellg()/sizeof(valor);
   archi.seekg(0,ios::beg);
   cout << "El puntero a get ( tellg() ) está en: "
        << archi.tellg() << endl;

   for (i=0; i<N; i++)
   { archi.read((char*)&v,sizeof(valor));
      cout << v << endl;
   }

   cout << "El puntero a get ( tellg() ) esta' en: "
        << archi.tellg() << endl;
   archi.close();
   cin.get();
}
```

Errores en las rutas de acceso a archivos

Siempre debe usarse la barra inclinada hacia adelante ("/") al especificar una ruta de acceso para un archivo, inclusive en sistemas operativos que utilicen barras invertida como DOS, Windows, OS/2, etc.

Por ejemplo:

```
//especificación de la ruta de acceso
#include <fstream>
using namespace std;

int main() {
  ifstream mi_archivo("../test.dat");//correcto
  ifstream mi_archivo("..\test.dat");//incorrecto

  // ...
}
```

Debe recordarse que la barra invertida ("\") se usa para representar caracteres especiales:: "\n" es el caracter de nueva línea, "\b" es el caracter de retroceso, "\t" es una tabulación, "\a" es una "alerta", "\v" es una tabulación vertical, etc.

Por lo tanto el nombre de archivo "\version\nombre\alfabetico\beta\test.dat" se interpreta como un conjunto de caracteres sin sentido; en lugar de esto, debe usarse "/version/nombre/alfabetico/beta/test.dat", inclusive en sistemas que utilicen "\" como el separador de directorios (DOS, Windows, OS/2, etc). Esto es gracias a que las rutinas de librerías en estos sistemas operativos pueden manejar indistintamente los separadores "/" y "\".

Buffers y sincronización

Cuando operamos con archivos de flujo, estos son asociados con un buffer de tipo streambuf. Este buffer es un bloque de memoria que actúa como intermediario entre el flujo y el archivo físico. Por ejemplo, cuando tenemos un flujo de salida y llamamos repetidamente a la función put (para escribir un solo caracter), el caracter no es escrito directamente en el archivo físico con el que se asocia el flujo de datos cada vez que se llama a la función; en vez de eso el carácter se inserta en el buffer de ese flujo.

Cuando se vacía el buffer, todos los datos que contiene se escriben en el archivo físico (si es un flujo de salida), o simplemente se borran (si es un flujo de entrada). Este proceso se denomina sincronización y toma lugar bajo cualquiera de estas circunstancias:

Cuando el archivo es cerrado: antes de cerrar el archivo todos los buffers que no han sido completamente escritos o leídos son sincronizados.

Cuando el buffer está lleno: los buffers tienen un tamaño determinado. Cuando el buffer está lleno es automáticamente sincronizado.

Explícitamente con manipuladores: cuando ciertos manipuladores son usados en flujos de datos se produce una sincronización. Estos manipuladores son: *flush y endl.*

Explícitamente con la función sync(): llamando a la función sync() (sin parámetros) producimos una sincronización. Esta función retorna un valor de tipo int igual a -1 si el flujo no tiene un buffer asociado o en caso de error.

Programación orientada a objetos

Clases

Las clases son una manera lógica de organizar datos y funciones en una estructura única. Se declaran utilizando la palabra clave *class*.

Su forma es:

```
class Nombre_de_la_clase {
  etiqueta_de_permisos_1:
    atributos_1;
    métodos_1;
  etiqueta_de_permisos_2:
    atributos_2;
    métodos_2;
  } nombre_del_objeto;
```

donde el nombre_de_la_clase es el nombre de la clase (un tipo definido por el usuario) y el campo opcional nombre_del_objeto, es uno o varios identificadores de objetos válidos. El cuerpo de la declaración puede contener miembros, que pueden ser tanto declaraciones de datos (atributos) o de funciones (métodos), y opcionalmente etiquetas de permisos, que pueden ser cualquiera de las 3 palabras reservadas: *private:* , *public:* o *protected:*. Estas hacen referencias a los permisos que los miembros podrán tener:

- *private*: las funciones miembro de la clase son accesibles sólo por otros miembros de la misma clase o de sus clases amigas friendo
- *protected*: los miembros son accesibles, además de por los miembros de la misma clase y de sus clases friend, por los miembros de las clases derivadas.
- *public*: son accesibles por cualquiera para el cual la clase es visible.

Si se declaran los miembros de una clase antes de incluir cualquier etiqueta, se consideran privados, por lo que se indica que los permisos por defecto para los miembros es private.

Por ejemplo:

```
class Rectangulo {
    int x, y;
  public:
    void fijar_valores (int,int);
    int area (void);
  } rect;
```

Declara la clase Rectangulo y un objeto llamado rect de esta clase (tipo). Esta clase contiene cuatro miembros: dos variables (atributos) del tipo *int* (*x* e y) en la sección private (debido que el permiso por defecto es el privado) y dos funciones (métodos) en la sección public: fijar_valores() y area(), de las que sólo se ha incluído el prototipo.

Note la diferencia entre el nombre de la clase y el nombre del objeto: En el ejemplo anterior, Rectangulo es el nombre dela clase (tal como el nombre de un tipo definido por el usuario), mientras que rect es un objeto del tipo Rectangulo. La misma diferencia existe entre int y a en la siguiente declaración:

```
int a;
```

int es el nombre de la clase (type) y a es el nombre del objeto (variable).

En las sucesivas instrucciones del cuerpo de los programas nos referiremos a los cualquiera de los miembros públicos del objeto rect como si fueran funciones normales o variables, colocando el nombre del objeto seguido por un punto y luego el mimbro de la clase. Por ejemplo

```
rect.fijar_valores (3,4);
mi_area = rect.area();
```

pero no podremos referirnos a x ni a y porque son miembros privados (private) de la clase y pueden ser unicamente referidos por otros miembros de la misma clase.

A continuación hay un ejemplo completo de Rectangulo

```
// ejemplo de clase
#include <iostream>
using namespace std;

class Rectangulo {
    int x, y;
  public:
    void fijar_valores (int,int);
    int area (void) {return (x*y);}
};

void Rectangulo::fijar_valores (int a, int b)
{
  x = a;
  y = b;
}

int main () {
  Rectangulo rect;
  rect.fijar_valores (3,4);
  cout << "área: " << rect.area();
}
```

```
área: 12
```

Lo nuevo en este código es el operador **::** de ámbito incluido en la definición de fijar_valores(). Se utiliza para declarar un miembro de la clase fuera de ella. Observar que se ha definido el comportamiento de la función miembro area() dentro de la definición de la clase Rectangulo debido a su extremada simplicidad. Mientras que fijar_valores() tiene solamente su prototipo declarado dentro de la clase pero su definición está fuera. En esta declaración fuera de la clase es donde se utilizó el operador de ámbito **::**.

El operador de ámbito (::) permite especificar la clase a la que la función miembro que está siendo declarada pertenece, tomando exactamente las mismas propiedades de ámbito que si fuera directamente declarada dentro de la clase. Por ejemplo, en la función fijar_valores() del código anterior, hemos recurrido a las variables *x* e *y*, que son miembros de la clase Rectangulo y que son visibles solamente dentro de ella y sus miembros (debido a que son privadas).

La única razón para que se haya hecho a los miembros x e y private (recordar que por defecto todos los miembros de una clase definidos con la palabra reservada class tiene acceso privado) es porque ya se ha definido una función para introducir estos valores al objeto (fijar_valores()) y por lo tanto el resto del programa no tiene porqué acceder directamente a ellos. Si bien en ejemplos sencillos como los que se han desarrollado hasta el momento no se ve la utilidad de proteger estas dos variables, en proyectos más grandes es muy importante que los valores no puedan ser modificados de una manera inesperada (desde el punto de vista del objeto).

Una de las mayores ventajas de las clases es que se puede definir varios objetos diferentes de ellas. Por ejemplo, siguiendo con el ejemplo de la clase Rectangulo, se puede declarar el objeto rectb además del rect:

```
// ejemplo de clase
#include <iostream>
using namespace std;

class Rectangulo {
    int x, y;
  public:
    void fijar_valores (int,int);
    int area (void) {return (x*y);}
};

void Rectangulo::fijar_valores (int a, int b)
{
  x = a;
  y = b;
}

int main () {
  Rectangulo rect, rectb;
  rect.fijar_valores (3,4);
  rectb.fijar_valores (5,6);
  cout << "área rect: " << rect.area() << endl;
  cout << "área rectb: " << rectb.area() << endl;
}
```

```
área rect: 12
área rectb: 30
```

Observar que no se obtiene el mismo resultado de la llamada a rect.area() que de la llamada rectb.area(). Para explicarlo de alguna manera, cada objeto de la clase Rectangulo tiene sus propias variables x e y, y sus propias funciones fijar_valores() y area().

En esto esta basado el concepto de *objeto* y *programación orientada a objetos*. En que los datos y las funciones pertenecen al objeto, en vez del punto de vista habitual de objetos como parámetros de funciones en la programación estructurada. En esta sección y en la siguiente se discutirá sobre la ventaja de esta metodología.

En este caso concreto, la clase (tipo del objeto) del que estamos hablando es Rectangulo, del que hay dos instancias, u objetos rect y rectb, cada uno con sus propias variables y funciones miembro.

Constructores y destructores

Los objetos generalmente necesitan inicializar variables o asignar memoria dinámica durante su proceso de creación para convertirse en totalmente operativos y para evitar que retorne valores no esperados en la ejecución. Por ejemplo, ¿qué podría pasar si en el ejemplo anterior se llamara a la función area() antes de habar invocado a fijar_valores()? Probablemente se obtendría un valor indeterminado ya que a los miembros x e y nunca le fue asignado un valor.

Para evitar esto, una clase debe incluir una función especial llamada *constructor*, que puede ser declarada como una función miembro con el mismo nombre de la clase. Esta función constructor será llamada automáticamente cuando una nueva instancia de la clase sea creada (cuando declare un nuevo objeto o asigne un objeto de esta clase) y sólo en ese momento. A continuación se implementarár Rectangulo de manera de incluir un constructor.

```
// ejemplo de clase
#include <iostream>
using namespace std;

class Rectangulo {
    int ancho, largo;
  public:
    Rectangulo (int,int);
    int area (void) {return (ancho*largo);}
};

Rectangulo::Rectangulo (int a, int b) {
  ancho = a;
  largo = b;
}

int main () {
  Rectangulo rect (3,4);
  Rectangulo rectb (5,6);
  cout << "área rect: " << rect.area() << endl;
  cout << "área rectb: " << rectb.area() << endl;
}
```

```
área rect: 12
área rectb: 30
```

Como se puede ver, el resultado de los ejemplos es idéntico a los previos. En este caso solamente se ha reemplazado la función fijar_valores(), que ya no existe, por el constructor. Notar la manera en que los parámetros se pasan al constructor al momento en que se crea una instancia de la clase.

```
Rectangulo rect (3,4);
Rectangulo rectb (5,6);
```

Se puede ver también como ni en el prototipo ni más adelante en la declaración del constructor se incluye un valor de retorno, aunque no es del tipo void, esto debe ser siempre así. Un constructor jamás devuelve un valor aunque no se especifique void. Como se ha hecho en el ejemplo previo.

El *destructor* cumple la función opuesta. Esta función se llama automáticamente cuando el objeto es desalojado de la memoria, ya sea porque el ámbito de su existencia haya finalizado (por ejemplo, si fue definido como un objeto local dentro de una función y la función finaliza) o debido a que es un objeto asignado dinámicamente y el liberado usando el operador *delete*.

El destructor debe tener el mismo nombre que el de la clase con el tilde (~) como prefijo y no debe retornar ningún valor.

El uso de los destructores es especialmente adecuado cuando un objeto asigna memoria dinámica durante su vida y en el momento de ser destruido se quiere liberar la memoria que ha utilizado.

```
// ejemplo de constructores y destructores
#include <iostream>
using namespace std;

class Rectangulo {
    int *ancho, *largo;
  public:
    Rectangulo (int,int);
    ~Rectangulo ();
    int area (void) {return (*ancho * *largo);}
};
```

```
área rect: 12
área rectb: 30
```

```
Rectangulo::Rectangulo (int a, int b) {
  ancho = new int;
  largo = new int;
  *ancho = a;
  *largo = b;
}

Rectangulo::~Rectangulo () {
  delete ancho;
  delete largo;
}

int main () {
  Rectangulo rect (3,4), rectb (5,6);
  cout << "área rect: " << rect.area()
       << endl;
  cout << "área rectb: " << rectb.area()
       << endl;
  return (0);
}
```

Sobrecarga de constructores

Al igual que cualquier otra función, un constructor puede también ser sobrecargado con varias funciones que tienen el mismo nombre pero diferente tipo o cantidad de parámetros. Recordar que el compilador ejecutará la que coincida con el nombre al momento de ser invocada. En el caso de los constructores, al momento en que el objeto es declarado.

De hecho, en los casos donde se declara una clase y no se especifica ningún constructor, el compilador asume automáticamente dos constructores sobrecargados (el "constructor vacio" y "el constructor de copia"). Por ejemplo, para la clase:

```
class CEjemplo {
  public:
    int a,b,c;
    void multiplicar (int n, int m) { a=n; b=m; c=a*b; };
  };
```

que no posee constructores, el compilador automáticamente asume que tiene las siguientes funciones miembro:

Constructor vacío

Es un constructor que no tiene parámetros definido y un bloque de instrucciones vacío. Aunque no hace nada, es necesario en algunos casos como en los ejemplos que se ven más adelante.

```
CEjemplo::CEjemplo () { };
```

Constructor de copia

Este es un constructor con solo un parámetro de su mismo tipo. Asigna a todas las variables miembros no estáticas de la clase del objeto, una copia de los valores del objeto pasado como parámetro.

```
CEjemplo::CEjemplo (const CEjemplo& rv)
{
     a=rv.a;  b=rv.b;  c=rv.c;
}
```

Es importante destacar que el constructor vacío existen por defecto (es decir, se puede invocar sin haberlo codificado), sólo si ningún otro constructor es explícitamente declarado. En el caso de que algún constructor con cualquier cantidad de parámetros y/o tipo sea declarado, se debe explícitamente declarar el constructor vacío. El código puede ser idéntico al anterior si no se quiere realizar algo adicional. El contructor de copia no se "pierde", sigue existiendo con las características comentadas anteriormente.

```
// sobrecargando constructores
#include <iostream>
using namespace std;

class Rectangulo {
    int ancho, largo;
  public:
    Rectangulo ();
    Rectangulo (int,int);
    int area (void) {
      return (ancho*largo);}
};

Rectangulo::Rectangulo () {
  ancho = 5;
  largo = 5;
}

Rectangulo::Rectangulo (int a, int b) {
  ancho = a;
  largo = b;
}

int main () {
  Rectangulo rect (3,4);
  Rectangulo rectb;
  cout << "área rect: " << rect.area()
       << endl;
  cout << "área rectb: " << rectb.area()
       << endl;
}
```

```
área rect: 12
área rectb: 30
```

En este caso rectb se declaró con parámetros, por lo que será inicializado por el constructor sin parámetros, que declara tanto a ancho como a largo con el valor de 5.

Notar que si se declara un nuevo objeto y no se le quiere pasar parámetros no se debería incluir los paréntesis ( ).

```
Rectangulo rectb;   // correcto
Rectangulo rectb(); // incorrecto!
```

Vector de objetos

La clase vector puede contener objetos con la misma simplicidad que se almacenan valores numéricos. También hay que tener en cuenta que según como se creen los elementos del vector, se llamará al constructor para cada uno de los objetos. El siguiente ejemplo ilustra esta característica:

```
#include <iostream>
#include <vector>
using namespace std;

class LaClase
{ private:
    int x;
  public:
    LaClase() {x=rand()%100;};
    int devuelve_x() {return x;};
};

int main()
{
  vector <LaClase> ListaObj_1;
  for (unsigned i=0;i<5;i++)
  { LaClase obj_aux;
    ListaObj_1.push_back(obj_aux);
  }

   for (unsigned i=0;i<ListaObj_1.size();i++)
    cout << ListaObj_1[i].devuelve_x() << "   ";
  cout << endl;

  //====================================
  vector <LaClase> ListaObj_2(5);
   for (unsigned i=0;i<ListaObj_2.size();i++)
    cout << ListaObj_2[i].devuelve_x() << "   ";
  cout << endl;

  cin.get();
  return (0);
}
```

El vector ListaObj_1 el llenado llamando al constructor de cada uno de los objetos, por lo tanto, el atributo x es inicializado por medio de la función rand(), dando como resultado valores distintos de x para cada uno de los objetos. En el segundo caso, ListaObj_2 es creada de la siguiente forma: se llama al constructor del primero y luego la clase vector fue codificada internamente para que copie el mismo objeto en los siguientes elementos, de esta manera se tiene el mismo valor del atributo de x para todos los objetos. Un resultado similar a este último se hubiese logrado si se utilizaba el siguiente código:

```
vector <LaClase> ListaObj_2;
ListaObj_2.resize(5);
```

Punteros a clases

Es perfectamente válido crear punteros que apunten a objetos, para realizar esto se puede considerar que una vez declarada, la clase llega a ser un tipo válido, de manera que use el nombre de la clase como el tipo para el puntero. Por ejemplo:

```
Rectangulo * punt_a_rect;
```

Es un puntero a un objeto de la clase Rectangulo.

Como ocurre con las estructuras de datos, para referirse directamente al miembro de un objeto apuntado por un puntero debemos utilizar el operador ->. A continuación se muestra un ejemplo de algunas combinaciones posibles:

```
// punteros a clases
#include <iostream>
using namespace std;

class Rectangulo {
    int ancho, largo;
  public:
    void fijar_valores (int, int);
    int area (void) {return (ancho * largo);}
};

void Rectangulo::fijar_valores (int a, int b) {
  ancho = a;
  largo = b;
}

int main () {
  Rectangulo a, *b, *c;
  Rectangulo * d = new Rectangulo[2];
  b= new Rectangulo;
  c= &a;
  a.fijar_valores (1,2);
  b->fijar_valores (3,4);
  d->fijar_valores (5,6);
  d[1].fijar_valores (7,8);
  cout << "área de: " << a.area() << endl;
  cout << "área de *b: " << b->area() << endl;
  cout << "área de *c: " << c->area() << endl;
  cout << "área de d[0]: " << d[0].area() << endl;
  cout << "área de d[1]: " << d[1].area() << endl;
  return (0);
}
```

```
área de: 2
área de *b: 12
área de *c: 2
área de d[0]: 30
área de d[1]: 56
```

A continuación se muestra un resumen de cómo puede leer algunos operadores de clase y punteros (*, &, ., ->, [ ]) que aparecen en el ejemplo precedente:

```
*x     puede ser leído como: apuntado por x.
&x     puede ser leído como: dirección de x.
x.y    puede ser leído como: miembro y del objeto x.
(*x).  y puede ser leído como: miembro y del objeto apuntado por x.
x-y    puede ser leído como:  miembro y del objeto apuntado por x
       (equivalente al anterior).
x[0]   puede ser leído como: primer objeto apuntado por x.
x[1]   puede ser leído como: segundo objeto apuntado porx.
x[n]   puede ser leído como: (n+1)ésimo objeto apuntado por x.
```

Sobrecarga de operadores

C++ posee la opción de usar operadores estandar del lenguaje entre objetos además de poder usarlos entre tipos fundamentales. Por ejemplo:

```
int a, b, c;
a = b + c;
```

es perfectamente válido, dado que las distintas variables en la suma son todas de tipos fundamentales. No es así de directo cuando se quiere sumar objetos, porque estos pueden tener una variedad de atributos que debería indicarse cómo se realiza su suma.

Para sobrecargar un operador sólo se requiere escribir una función miembro de una clase cuyo nombre es **operator** seguido por el signo del operador que se desea sobrecargar. El prototipo es el siguiente:

tipo **operator** *signo* **(***parametros***);**

Ejemplo de operador binario

A continuación se muestra un ejemplo que incluye al operador +, que permitirá sumar dos complejos. La suma consiste en sumar las dos partes reales e imaginarias de cada objeto.

```
# include <iostream>
using namespace std;

class Complejo {
 private:
  float re, im;
 public:
  Complejo () {};
  Complejo (float a, float b) {re=a; im=b;}
  void mostrar();
  Complejo operator+ (Complejo &valor)
  { Complejo aux;
    aux.re = re + valor.re;
    aux.im = im + valor.im;
    return(aux); //devuelve el resultado de la suma
  }
};

void Complejo::mostrar()
{ cout << re;
  if (im>0) cout << "+";
  cout << im << "i" << endl << endl;
}

int main () {
  Complejo a (3,1);
  Complejo b (1,2);
  Complejo c;

  a.mostrar();
  b.mostrar();

  cout << "*** c = a+b ***** ";
  c = a+b; // ver "a+b" como si fuera "a.+(b)"
```

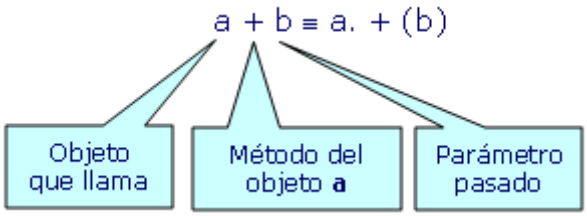
```
3+1i

1+2i

"*** c = a+b ***** 4+3i

*** c = a+b+c *** 8+6i
```

```
  c.mostrar();

  cout << "*** c = a+b+c *** ";
  c = a+b+c; // esta sintaxis también es válida
  c.mostrar();

  cin.get();
  return (0);
}
```

La función *operator*+ de la clase Complejo es la que está a cargo de sobrecargar el operador aritmético + que nos permite hacer:

```
c = a + b;
```

Esta operación puede suponerse que tiene la siguiente equivalencia:



Observar que también se incluyó el constructor vacío (sin parámetros) y se definió con un bloque sin instrucciones (vacio):

```
Complejo () { };
```

Esto es necesario, dado que ya existe otro constructor,

```
Complejo (int, int);
```

y ninguno de los constructores por defecto existirá en Complejo si no se lo declara explícitamente. De otra forma la declaración

```
Complejo c;
```

incluida en main() no sería válida.

De todas maneras, se debe advertir que un bloque vacío no es una implementación recomendada para un constructor, dado que no satisface la funcionalidad mínima que un constructor debería tener, que es la inicialización de todas las variables de la clase. En este caso este constructor deja las variables **re** e **im** indefinidas. Por lo tanto, una declaración más recomendable sería:

```
      Complejo () { re=0; im=0; };
```

Que por simplicidad no fue incluida en el código.

Así como las clases incluyen por defecto un constructor vacío y un constructor de copia, también incluyen una definición por defecto del operador de asignación (=) entre dos clases del mismo tipo. Por supuesto, se puede redefinir con cualquier otra funcionalidad que se desee para este operador, como por ejemplo, copiar sólo ciertos miembros de la clase.

El uso de *this*

this representa dentro de una clase la dirección en memoria del objeto de esa clase que está siendo ejecutado. Es un puntero cuyo valor es siempre la dirección de memoria del objeto.

Es frecuentemente usado en las funciones *operator*= que retornan objetos por referencia (evitando el uso de objetos temporarios). Siguiendo con el ejemplo de vectores podríamos haber escrito una función operator= como la siguiente:

```
CVector& CVector::operator= (const CVector& param)
{
  x=param.x;
  y=param.y;
  return *this;
}
```

Que es probablemente un código similar al por defecto generado por la clase si no incluimos ninguna función operator=.

Ejemplo de operador unario

Este ejemplo muestra el operador unario de cambio de signo (-objeto) y el de incremento (++objeto).

```
# include <iostream>
using namespace std;

class Complejo {
 private:
  float re, im;
 public:
  Complejo () {};
  Complejo (float a, float b) {re=a; im=b;}

  void mostrar();

  Complejo operator-()
  { re = -re;
    im = -im;
    return (*this); // devuelve él mismo
  }

  Complejo operator++()
  { // define ++objeto
    re++;
    im++;
    return (*this);
  }
};

void Complejo::mostrar()
{ cout << re;
  if (im>0) cout << "+";
  cout << im << "i" << endl << endl;
}

int main () {
```

```
*** b = -a ****** -3-1i

*** -a ********** 3+1i

*** ++a ********* 4+1i
```

```
  Complejo a (3,1);
  Complejo b;

  cout << "*** b = -a ****** ";
  b=-a;    // cambia signo y asigna
  b.mostrar();

  cout << "*** -a ********** ";
  -a;      // sólo cambia de signo
  a.mostrar();

  cout << "*** ++a ********* ";
  ++a;
  a.mostrar();

  cin.get();
  return (0);
}
```

Observar que no es lo mismo el operador ++*objeto* que *objeto*++. Para este último caso la codificación para la sobrecarga de este operador es con la siguiente sintaxis:

```
Complejo Complejo:: operator++(int notused)
{ // define objeto++
  re++;
  im++;
  return (*this);
}
```

Miembros estáticos

Los datos estáticos de una clase son conocidos también como *variables de clase*, porque su contenido no depende de ningún objeto. Sólo hay un único valor para todos los objetos de la misma clase.

Por ejemplo, puede ser usado para que una variable dentro de una clase pueda contener el número de objetos declarados de esa clase, como en el siguiente ejemplo:

```
// miembros estáticos en clases
#include <iostream>
using namespace std;

class Boba {
  public:
    static int n;
    Boba () { n++; };
    ~Boba () { n--; };
};

int Boba::n=0;

int main () {
  Boba a;
  Boba b[5];
  Boba *c = new Boba;
  cout << a.n << endl;
  delete c;
```

```
7
6
```

```
  cout << Boba::n << endl;
  return (0);
}
```

De hecho, los miembros estáticos tienen las mismas propiedades que las variables globales pero disfrutando del alcance de una clase. Por esta razón, y para evitar que sean declaradas varias veces, sólo podemos incluir el prototipo (declaración) en la clase pero no la definición (inicialización). Para inicializar un dato estático debemos incluir una declaración formal fuera de la clase, en el alcance global, como en el ejemplo previo.

Debido a que es una única variable para todos los objetos desde la misma clase, esto puede ser referido como un miembro de cualquier objeto de la clase o inclusive directamente por el nombre de la clase (por supuesto esto es válido únicamente para miembros estáticos).

```
cout << a.n;
cout << Boba::n;
```

estas dos llamadas incluidas en el ejemplo previo se refieren a la misma variable: la variable estática *n* dentro de la clase Boba.

Una vez más, debe recordarse que de hecho es una variable global. La única diferencia es el nombre fuera de la clase.

Como podemos incluir datos estáticos dentro de una clase también podemos incluir funciones estáticas. Representan lo mismo: son funciones globales que son llamadas como si fueran miembros de una clase determinada. Sólo pueden referirse a datos estáticos, en ningún caso a miembros no-estáticos de la clase, y tampoco permiten el uso de this, dado que hacen referencia a un objeto apuntado y estas funciones de hecho no son miembros de ningún objeto sino directamente miembros de la clase.

Relación entre clases

Funciones amigas (*friend*)

En la sección previa se vio que había tres niveles de protección interna para los diferentes miembros de una clase: public, protected y private. En el caso de los miembros *protected* y *private*, no se puede acceder a ningún miembro fuera de la misma clase en la que fueron declarados. Sin embargo, esta regla puede ser transgredida usando la palabra friend en una clase, por medio de la cual podemos permitir que una función externa obtenga acceso a los miembros protected y private de una clase.

Para permitir que una función externa tenga acceso a los miembros private y protected de una clase debemos declarar el prototipo de la función externa que obtendrá el acceso precedida de la palabra friend dentro de la declaración de la clase que compartirá sus miembros. En el siguiente ejemplo se declara la función amiga duplicar:

```
// funciones amigas
#include <iostream>
using namespace std;

class Rectangulo {
    int ancho, largo;
  public:
    void fijar_valores (int, int);
    int area (void) {return (ancho * largo);}
```

```
24
```

```
    friend Rectangulo duplicar (Rectangulo);
};
void Rectangulo::fijar_valores (int a, int b) {
  ancho = a;
  largo = b;
}
Rectangulo duplicar (Rectangulo rectparam)
{
  Rectangulo rectres;
  rectres.ancho = rectparam.ancho*2;
  rectres.largo = rectparam.largo*2;
  return (rectres);
}
int main () {
  Rectangulo rect, rectb;
  rect.fijar_valores (2,3);
  rectb = duplicar (rect);
  cout << rectb.area();
}
```

Desde dentro de la función duplicar() que es amiga de la clase Rectangulo, se accede a los miembros ancho y largo de diferentes objetos de tipo Rectangulo. Note que ni en la declaración de duplicar() ni en su posterior uso en main() se consideró a duplicar() como miembro de la clase Rectangulo, dado que no es así.

Las funciones amigas pueden servir, por ejemplo, para realizar operaciones entre dos clases diferentes. Generalmente el uso de funciones amigas está fuera de la metodología de la programación orientada a objetos, así que cuando sea posible es mejor intentar usar miembros de la misma clase para llevar a cabo un proceso. Como en el ejemplo previo, en el que hubiera sido mejor integrar a la función duplicar() dentro de la clase.

Unos de los operadores que resulta muy útil sobrecargar son los flujos de entrada y salida, de esta manera podremos ingresar y enviar a la salida los valores de los tipos de datos que hayamos definido. Consideremos el caso del operador inserción de flujo << el que debe tener un operando izquierdo del tipo ostream& (como cout de la expresión cout<<objeto de clase) por lo que según lo que sería necesario que fuera una función no miembro. De la misma manera el operador >> debe tener a la izquierda un operador de la clase istream&. Además cada una de estas funciones deberían poder acceder a los datos miembros private de la clase, por lo que deberíamos trabajar con funciones friend. A continuación se presenta un ejemplo del operador de inserción:

```
// función amiga para sobrecargar "<<" y ">>"
# include <iostream>
using namespace std;

class Complejo {
  private:
    float re, im;
  public:
    Complejo () {};
    Complejo (float a, float b) {re=a; im=b;}

    Complejo operator+ (Complejo &valor)
    { Complejo aux;
      aux.re = re + valor.re;
      aux.im = im + valor.im;
      return(aux);
    }
```

```
4+3i

3+1i   1+2i
```

```
    friend ostream & operator<<(ostream& sal, Complejo c);
    friend istream& operator>>(istream& ing, Complejo &c);
};

ostream& operator<< (ostream& sal, Complejo aux)
{ sal << aux.re;
  if (aux.im>0) sal << "+";
  sal << aux.im << "i";

  return sal;    // permite concatenar con otros "<<"
};

istream& operator>> (istream& ing, Complejo &c)
{ ing >> c.re >> c.im;
  return ing;
};

int main () {
  Complejo a (3,1);
  Complejo b (1,2);
  Complejo c;

  c = a+b;
  cout << c << endl << endl;
  cout << a << "   " << b << endl << endl;

  ofstream arch("datos.txt");
  arch << a << endl;
  arch.close();

  cout << "Ingresar parte real (enter) ";
  cout << "y parte imaginaria (enter): " << endl;
  cin >> a;
  cout << "El complejo ingresado es: " << a << endl;

  cin.get();
  return (0);
}
```

El nuevo operador <<() usa un *ostream* con el nombre *sal*. Normalmente sal se referirá al objeto cout, pero tengamos en cuenta que podría referirse a cualquier objeto stream de salida por ello estamos usando sal como alias para cualquiera sea. Respecto al objeto del tipo Complejo, podríamos haberlo pasado por valor o por referencia; aquí escogimos pasarlo por referencia porque usa menos memoria y tiempo, que el pasaje por valor.

Note que el tipo que retorna la función es ostream &. Esto significa que la función retorna una referencia a un ostream, debido a que el programa pasa una referencia a un objeto al comienzo, el efecto neto es que la función devuelve exactamente un valor igual al objeto que se le pasó.

En forma similar para ingresar datos con >> pero aquí es muy importante declarar la función amiga con el último parámetro por referencia.

Clases amigas (*friend*)

Como existe la posibilidad de declarar funciones amigas, también se puede definir una clase como amiga de otra, permitiendo que la segunda acceda a los miembros protected y private de la primera.

```
// clases amigas
#include <iostream>
using namespace std;

class Cuadrado;
class Rectangulo {
    int ancho, largo;
  public:
    int area (void)
      {return (ancho * largo);}
    void convertir (Cuadrado a);
};
class Cuadrado {
  private:
    int lado;
  public:
    void fijar_lado (int a)
      {lado=a;}
    friend class Rectangulo;
};
void Rectangulo::convertir (Cuadrado a) {
  ancho = a.lado;
  largo = a.lado;
}

int main () {
  Cuadrado sqr;
  Rectangulo rect;
  sqr.set_lado(4);
  rect.convertir(sqr);
  cout << rect.area();
  return (0);
}
```

```
16
```

En este ejemplo se declara a Rectangulo como amiga de Cuadrado para que Rectangulo pueda acceder a los miembros protected y private de Cuadrado, más concretamente a Cuadrado::lado, que define la longitud del lado del cuadrado.

Se nota también como algo nuevo la primera instrucción del programa, que es un prototipo de la clase Cuadrado, esto es necesario porque dentro de la declaración de Rectangulo se hace referencia a Cuadrado (como parámetro en convertir()). La definición de Cuadrado es incluida posteriormente, así que si no se incluye una definición previa para Cuadrado esta clase no sería visible desde adentro de la clase Rectangulo.

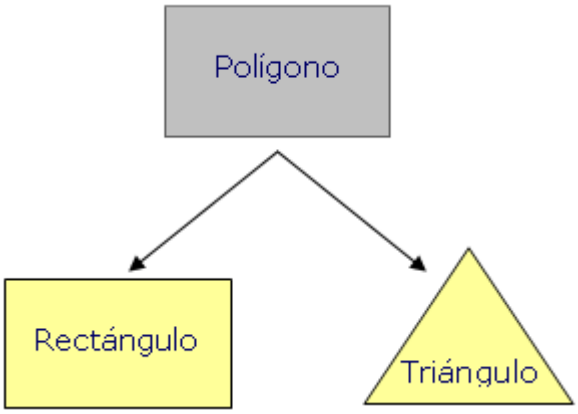
Considere que las amistades no son correspondidas si no lo especificamos explícitamente. En el ejemplo de Cuadrado, Rectangulo es considerada una clase amiga, pero Rectangulo no hace lo propio con Cuadrado, así que Rectangulo puede acceder a los miembros protected y private de Cuadrado pero no a la inversa. Aunque nada impide declarar también a Cuadrado como amiga de Rectangulo.

Herencia entre clases

Una importante propiedad de las clases es la *herencia*. Esto permite crear un objeto derivado de otro, así que éste podría incluir algunas propiedades de otros objetos más las propias. Por ejemplo, si se quiere declarar una serie de clases que describan polígonos como Rectangulo o Triangulo. Ambas poseen algunas

características en común, por ejemplo, que ambas pueden ser descritas por medio de sólo dos lados: altura y base.

Esto puede ser representado con una clase Polígono de la cual derivan las dos anteriores, Rectangulo y Triangulo.



La clase Poligono contendrá miembros que son comunes a todos los polígonos. En nuestro caso: ancho y largo. Rectangulo y Triangulo sería sus clases derivadas.

Las clases derivadas heredan todos los miembros visibles de la clase base. Esto significa que si una clase base incluye un miembro A y de ella se deriva otra clase con otro miembro llamado B, la clase derivada contendrá ambos, A y B.

Para derivar una clase de otra, se usa el operador **:** (dos puntos) en la declaración de la clase derivada de la siguiente manera:

class *nombre_clase_derivada*: **public** *nombre_clase_base { ... };*

donde nombre_clase_derivada es el nombre de la clase derivada y nombre_clase_base es el nombre de la clase en la cual está basada. public puede ser reemplazado por cualquiera de los otros especificadores de nivel de acceso como protected o private, y describe el acceso para los miembros heredados, como se ve después de este ejemplo:

```
// clases derivadas
#include <iostream>
using namespace std;

class Poligono {
  protected:
    int ancho, largo;
  public:
    void fijar_valores (int a, int b)
      { ancho=a; largo=b;}
  };
class Rectangulo: public Poligono {
  public:
    int area (void)
      { return (ancho * largo); }
  };
class Triangulo: public Poligono {
  public:
```

```
20
10
```

```
    int area (void)
      { return (ancho * largo / 2); }
  };

int main () {
  Rectangulo rect;
  Triangulo trgl;
  rect.fijar_valores (4,5);
  trgl.fijar_valores (4,5);
  cout << rect.area() << endl;
  cout << trgl.area() << endl;
  return (0);
}
```

Los objetos de las clases Rectangulo y Triangulo contienen todos los miembros de Poligono: ancho, largo y fijar_valores().

Cuando se deriva una clase, los miembros protected de la clase base pueden ser usados por otros miembros de la clase derivada, lo que no ocurre con los miembros private. Como se quiso que ancho y largo pudieran ser manipulados por miembros de las clases derivadas Rectangulo y Triangulo, y no sólo por miembros de Poligono se usó el nivel de acceso protected en vez de private.

Se puede resumir los diferentes tipos de acceso de acuerdo a quién puede acceder a ellos en la siguiente manera:

Acceso	public	Protected	private
miembros de la misma clase	sí	sí	sí
miembros de clases derivadas	sí	sí	no
no-miembros	sí	no	no

Polimorfismo

Miembros virtuales

Para declarar un método de una clase que se va a redefinir en las clases descendientes, se debe preceder con la palabra reservada *virtual*.

Observar el ejemplo siguiente:

```
// miembros virtuales
#include <iostream>
using namespace std;

class Poligono {
  protected:
    int ancho, largo;
  public:
    void fijar_valores (int a, int b)
      { ancho=a; largo=b; }
    virtual int area ()
      { return (0); }
```

```
20
10
0
```

```
  };

class Rectangulo: public Poligono {
  public:
    int area ()
      { return (ancho * largo); }
  };

class Triangulo: public Poligono {
  public:
    int area ()
      { return (ancho * largo / 2); }
  };

int main () {
  Rectangulo rect;
  Triangulo trgl;
  Poligono poly;
  Poligono * ppoly1 = &rect;
  Poligono * ppoly2 = &trgl;
  Poligono * ppoly3 = &poly;
  ppoly1->fijar_valores (4,5);
  ppoly2->fijar_valores (4,5);
  ppoly3->fijar_valores (4,5);
  cout << ppoly1->area() << endl;
  cout << ppoly2->area() << endl;
  cout << ppoly3->area() << endl;
  return (0);
}
```

Ahora las tres clases (Poligono, Rectangulo y Triangulo) tienen los mismos miembros: ancho, largo, fijar_valores() y area(). El método area() se ha definido como virtual debido a que luego se redefine en las clases descendientes.

Se puede verificar que si se elimina esta palabra reservada (virtual) del código y entonces ejecuta el programa el resultado será 0 para los tres polígonos, en vez de 20,10,0. Esto podría deberse a que en vez de llamar a la función area() correspondiente a cada objeto (Rectangulo::area(), Triangulo::area() y Poligono::area(), respectivamente), se llamará Poligono::area() para todos los casos ya que se utiliza un puntero a Poligono.

Por lo tanto es la palabra *virtual* la que indica que los miembros de clases descendientes que posean el mismo nombre que uno de la clase base, se llame adecuadamente y para que esto funcione es necesario trabajar con punteros.

Clases bases abstractas

Las clases base abstractas muchas veces son muy similares a la clase Poligono del ejemplo previo. La única diferencia es que en el ejemplo anterior se ha definido la función válida area() para los objetos de la clase Poligono, mientras que en una clase base abstracta se puede simplemente dejar sin definir esta función agregándole =0 (igual a cero) en la declaración de la función.

La clase Poligono puede entonces ser como:

```
// clase abstracta Poligono
class Poligono {
  protected:
```

```
  int ancho, largo;
public:
  void fijar_valores (int a, int b)
    { ancho=a; largo=b; }
  virtual int area () =0;
};
```

Se ha agregado **=0** a *virtual int area ()* en vez de especificar una implementación para la función. Este tipo de funciones reciben el nombre de funciones virtuales puras, y todas las clase que contienen funciones virtuales puras son consideradas clases base abstractas.

La gran diferencia de una clase base abstracta es que no pueden ser creadas instancias (objetos) de ella. Pero se puede crear punteros a ella. Por lo que una declaración como:

```
Poligono poly;
```

es incorrecta para una clase base abstracta como la declarada antes porque intentará crear el objeto 'poly'. Sin embargo los punteros:

```
Poligono * ppoly1;
Poligono * ppoly2;
```

son perfectamente válidos. Esto es debido a que las funciones virtuales puras no están definidas y es imposible crear un objeto si no tiene todos sus miembros definidos. Sin embargo un puntero a un objeto de una clase descendiente donde esta función ha sido definida es perfectamente válido, porque en la declaración no se está creando ningún objeto.

A continuación se muestra un ejemplo completo:

```
// miembros virtuales
#include <iostream>
using namespace std;

class Poligono {
  protected:
    int ancho, largo;
  public:
    void fijar_valores (int a, int b)
      { ancho=a; largo=b; }
    virtual int area () =0;
  };

class Rectangulo: public Poligono {
  public:
    int area ()
      { return (ancho * largo); }
  };

class Triangulo: public Poligono {
  public:
    int area ()
      { return (ancho * largo / 2); }
  };

int main () {
  Poligono * ppoly1 = new(Rectangulo);
  Poligono * ppoly2 = new(Triangulo);
  ppoly1->fijar_valores (4,5);
```

```
20
10
```

```
  ppoly2->fijar valores (4,5);
  cout << ppoly1->area() << endl;
  cout << ppoly2->area() << endl;
  return (0);
}
```

Revisando el programa se puede notar que se puede referir a los objetos de clases diferentes usando un tipo de puntero único (Poligono*). Esto puede ser tremendamente útil. Imaginar ahora que se puede crear una función miembro de Poligono que tiene la capacidad de imprimir en la pantalla el resultado de la función area( ) independientemente de qué clase derivada sea:

```
// ejemplo miembros virtuales
#include <iostream>
using namespace std;

class Poligono {
  protected:
    int ancho, largo;
  public:
    void fijar_valores (int a, int b)
      { ancho=a; largo=b; }
    virtual int area (void) =0;
    void imprimir_area (void)
      { cout << this->area() << endl; }
  };

class Rectangulo: public Poligono {
  public:
    int area (void)
      { return (ancho * largo); }
  };

class Triangulo: public Poligono {
  public:
    int area (void)
      { return (ancho * largo / 2); }
  };

int main () {
  Poligono * ppoly1 = new (Rectangulo);
  Poligono * ppoly2 = new (Triangulo);
  ppoly1->fijar_valores (4,5);
  ppoly2->fijar_valores (4,5);
  ppoly1->imprimir_area();
  ppoly2->imprimir_area();
  return (0);
}
```

```
20
10
```

Recordar que *this* representa un puntero al objeto cuyo código está siendo ejecutado.

Las clases abstractas y los métodos virtuales le otorgan a C++ las características polimórficas que hacen a la programación orientada a objetos un instrumento muy útil. Por supuesto, se ha visto el uso más simple de estas características, pero imaginar estas características aplicadas a arreglos de objetos u objetos asignados a través de memoria dinámica.

Clases abstractas y polimorfismo

Introducción

Los primeros contactos con las clases abstractas, funciones virtuales y el polimorfismo ocasionan un misticismo sobre lo que está sucediendo. Muy lejos de ser una realidad, las características de este concepto son utilizadas por nosotros en la construcción del conocimientos que usamos a diario.

Este documento presenta un ejemplo de uso de las funciones virtuales en C++ para mostrar algunas características de la sintaxis, sin por ello pretender ser un documento sobre los aspectos conceptuales que encierran las funciones virtuales.

Funciones virtuales puras

En el siguiente ejemplo se trabajará con "*funciones virtuales puras*", que pueden ser reconocidas por la palabra *virtual* y también porque la función es seguida por *=0*. Si se intenta realizar un objeto a partir de esta clase, el compilador va a prevenir que se realice. Se fuerza a que alguna clase derivada provea una definición de esta función.

C++ provee un mecanismo para hacer este llamado en las funciones virtuales puras. La sintaxis es la siguiente:

```
virtual void X() = 0;
```

En el ejemplo siguiente es realizará una clase abstracta Instrumento formando parte de una representación de instrumentos musicales. El objetivo de Instrumento es crear una interfaz común para todas las clases derivadas desde él y no una particular implementación, por lo tanto, crear un objeto del tipo Instrumento, no tiene sentido y debería prevenirse, por ejemplo, dando un mensaje de error.

Si se trabaja con una genuina clase abstracta (como Instrumento), los objetos de esta clase no tendrán sentido.

El siguiente ejemplo modela un conjunto de instrumentos y la posibilidad de ejecutar una nota con ellos:

```
#include <iostream>
#include <vector>
#include <string>

enum nota { DO, RE, MI, FA, SOL, LA, SIb };

class Instrumento {   // funciones virtuales puras
 public:
  Instrumento() {cout << "Constructor de Instrumento" << endl;}
  virtual void ejecutar(nota) =0;
  virtual string nombre() =0;
  virtual ~Instrumento() {cout << "Destructor de Instrumento" << endl;}
};

class Piano : public Instrumento {
 public:
  Piano()             { cout << "Constructor de Piano" << endl;}
  void ejecutar(nota) { cout << "Piano::ejecutar" << endl; }
  string nombre() { return "Piano"; }
  ~Piano() { cout << "Destructor de Piano" << endl; }
};
```

```
class Violin : public Instrumento {
 public:
  Violin()            { cout << "Constructor de Violin" << endl;}
  void ejecutar(nota) { cout << "Violin::ejecutar" << endl; }
  string nombre() { return "Violin"; }
  ~Violin() { cout << "Destructor de Violin" << endl; }
};

void EjecutaNotas(Instrumento& i) {
 i.ejecutar(DO);
 i.ejecutar(RE);
 i.ejecutar(SOL);
}

int main() {

 Instrumento* a;

 int opcion;
 cout << "[1] Piano - [2] Violin: ";
 cin >> opcion;

 switch (opcion) {
   case  1: a = new Piano; break;
   default: a = new Violin;
 }

 cin.get();  //================================================

 cout << "Instrumento a ejecutar : " << a->nombre() << endl;
 a->ejecutar(SOL);

 cin.get();  //================================================

 EjecutaNotas (*a);

 cin.get();  //================================================

 cout << "COMIENZO DESTRUCTORES" << endl;
 delete a;

 cin.get();
 return (0);
}
```

No es necesario repetir la palabra reservada "virtual" en ninguna de las redefiniciones de la función miembro en las clases derivadas porque si una función es declarada como *virtual* en la clase base, ella es virtual en todas las clases derivadas. Sin embargo, es un buen hábito de programación dejar claro (mediante la palabra "virtual") cuales son las funciones miembro virtuales y cuales no.

Polimorfismo y uso de contenedores

Aunque los *contenedores* (containers) en las STL pueden alojar objetos por valor (es decir, pueden contener completamente a un objeto) probablemente no sea conveniente en algunos diseños realizar esto.

En muchos casos de programación orientada a objetos, es necesario crear objetos con **new** y realizar *upcasting* con la clase base como fue visto anteriormente, reduciendo la complejidad del código y

otorgando extensibilidad. Este uso polimórfico de los objetos debe ser desarrollado mediante un contenedor de punteros.

El siguiente código corresponde al uso de contenedores con el ejemplo anterior. Se ejemplifica el uso de la STL vector para contener punteros a varios tipos de Instrumentos.

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

enum nota { DO, RE, MI, FA, SOL, LA, SIb };

class Instrumento {   // funciones virtuales puras
 public:
  Instrumento() {cout << "Constructor de Instrumento" << endl;}
  virtual void ejecutar(nota) =0;
  virtual string nombre() =0;
  virtual ~Instrumento() {cout << "Destructor de Instrumento" << endl;}
};

class Piano : public Instrumento {
 public:
  Piano()             { cout << "Constructor de Piano" << endl;}
  void ejecutar(nota) { cout << "Piano::ejecutar" << endl; }
  string nombre() { return "Piano"; }
  ~Piano() { cout << "Destructor de Piano" << endl; }
};

class Violin : public Instrumento {
 public:
  Violin()            { cout << "Constructor de Violin" << endl;}
  void ejecutar(nota) { cout << "Violin::ejecutar" << endl; }
  string nombre() { return "Violin"; }
  ~Violin() { cout << "Destructor de Violin" << endl; }
};

void EjecutaNotas(Instrumento& i) {
 i.ejecutar(DO);
 i.ejecutar(RE);
 i.ejecutar(SOL);
}

typedef vector<Instrumento*> Contenedor;

int main() {
 Contenedor orquesta;

 orquesta.push_back(new Piano);
 orquesta.push_back(new Violin);
 orquesta.push_back(new Violin);

 cout << orquesta[1]->nombre() << endl;
 orquesta[1]->ejecutar(SOL);

 cin.get();

 unsigned i;
 for(i = 0; i < orquesta.size(); i++)
```

```
 { cout << "Instrumento a ejecutar : " << orquesta[i]->nombre() << endl;
   EjecutaNotas (*orquesta[i]);
 }

 cout << "COMIENZO DESTRUCTORES" << endl;

 for(i = 0; i < orquesta.size(); i++)
  delete orquesta[i];

 cin.get();
 return (0);
}
```

Plantillas

Planteo de una inquietud

Hay veces que se debe escribir una función que reciba parámetros con valores enteros, pero necesariamente otra para el mismo caso pero que los parámetros sean números reales. Las siguientes tres funciones, son el código para el cálculo del promedio de tres números, pero repetida la codificación (sobrecargadas) para tres tipos de datos distintos.

```
int promedio(int a, int b, int c)
{ return (int)((a+b+c)/3); }

float promedio(float a, float b, float c)
{ return (float)((a+b+c)/3); }

double promedio(double a, double b, double c)
{ return (double)((a+b+c)/3); }
```

¿No sería mejor que pudiéramos escribir algo más genérico como:

```
?? promedio(?? a, ?? b, ?? c)
{ return (??)((a+b+c)/3); }
```

... y que el compilador se encargue de poner los tipos adecuados?

Funciones plantilla

Las plantillas permiten crear funciones genéricas que admiten tipos de datos distintos como parámetro, y devuelven valores sin tener que escribir varias veces el código de la función para sobrecargarla con todos los tipos de datos posibles.

Su prototipo puede ser:

```
template <class identificador> declaracion_de_funcion;
```

Por ejemplo, la sintaxis para crear una función plantilla que retorna el mayor de dos parámetros puede ser la siguiente:

```
template <class TipoGenerico>
TipoGenerico mayor (TipoGenerico a, TipoGenerico b) {
  return (a>b?a:b);
}
```

Como especifica la primer línea, se ha creado una plantilla para un tipo genérico de dato que hemos llamado *TipoGenerico*. Por consiguiente, en la función, *TipoGenerico* se convierte en un tipo de dato válido y es usado como tipo para sus dos parámetros a y b, y como tipo para el valor que retorna la función *mayor*.

TipoGenerico todavía no representa ningún tipo de dato concreto; cuando la función *mayor* sea llamada, se deberá llamarla con cualquier tipo de dato válido. Este tipo de dato servirá como *patrón* y reemplazará a *TipoGenerico* en la función.

El código siguiente ejemplifica el uso de las plantillas para el ejemplo anterior:

```
// function template
#include <iostream>
using namespace std;

template <class T>
T mayor (T a, T b)
{
  return (a > b ? a : b);
}

int main()
{
  cout << mayor(3,7) << endl;
  cout << mayor(3.0,7.1) << endl;
  return 0;
}
```

```
7
7.1
```

En este caso se ha llamado al tipo genérico como ***T***, en vez de *TipoGenerico*, porque es más corto y además es uno de los identificadores más usuales para plantillas, aunque es válido usar cualquier identificador.

En el ejemplo anterior se usó la misma función *mayor( )* con argumentos de tipo *int* y *float* habiéndose escrito una única implementación de la función. Lo que es lo mismo, se ha escrito una función plantilla y se llamó con dos patrones diferentes.

Para el caso que se quiera llamar *mayor(3, 7.1)* durante la compilación aparecerá una ambigüedad, de qué tipo de datos tomar. En este caso, se debe expresar explícitamente el tipo de datos con la siguiente sintaxis: *mayor <double> (3, 7.1)* donde se indica, mediante los corchetes angulares, que tipo de datos debe adoptarse.

También se puede hacer funciones plantilla que admitan más de un tipo de dato genérico. Por ejemplo:

```
template <class T, class U>
T minimo (T a, U b) {
  T resultado;
  if (a<b) resultado=a;
    else resultado=b;
  return (resultado);
}
```

En este caso, la función plantilla *minimo()* admite dos parámetros de distinto tipo y devuelve un objeto del mismo tipo que el primer parámetro pasado ***T***. Por ejemplo, después de la declaración se podrá llamar a la función de la siguiente manera:

```
int i,j;
float f;
i = minimo (j,f);
```

aún si j y f son de distinto tipo.

Clases plantilla

También existe la posibilidad de aplicar plantillas a las clases. La sintaxis es similar a lo visto anteriormente. Por ejemplo, la siguiente clase permite almacenar un par de números:

```
template <class T>
class Par {
    T x, y;
  public:
    par (T primero, T segundo)
    {
      x=primero;
      y=segundo;
    }
};
```

Por ejemplo, si se quiere declarar un objeto de esta clase para almacenar dos valores de tipo int con los valores 115 y 36, se debe escribir de la siguiente manera:

```
Par <int> mis_ints(115, 36);
```

Observar que en este caso de los objetos que se crean a partir de clases con plantillas, se debe indicar el tipo de datos con el que se quiere trabajar mediante los corchetes angulares.

El mismo ejemplo anterior, pero creando un objeto que trabaje con flotantes, la sintaxis es:

```
Par <float> mis_floats(3.0, 2.18);
```

En el caso anterior, la única función de la clase fue definida dentro de la declaración de la clase, sin embargo si esto no es así y se define una función fuera de la declaración de la clase, siempre se debe preceder esta definición con el prefijo *template* <...> como se describe en el código siguiente:

```
// clases templates
#include <iostream>
using namespace std;

template <class T>
class Par {
    T x, y;
  public:
    Par(T primero, T segundo)
      {x=primero;
       y=segundo;}
    T mayor();
};

template <class T>
T Par<T>::mayor()
{
  T resultado;
  if (x>y) resultado = x;
    else resultado = y;

  return resultado;
}

int main (){
  Par <int> p(100, 75);
  cout << p.mayor();
  return (0);
}
```

```
100
```

Observar como en la definición de la función *mayor( )*, se debió poner nuevamente *template <class T>* de la siguiente manera:

```
template <class T>
T par<T>::mayor ()
```

Todas las ***T*** que aparecen son necesarias, razón por la cual cuando se declaran funciones que pertenezcan a clases plantillas se debe seguir ésta sintaxis.

Especialización de plantillas

La especialización de plantillas permite realizar una implementación específica según sea el tipo de dato que se ingrese como patrón. Por ejemplo, suponga que la clase plantilla *par* del ejemplo anterior necesitara una función para devolver el resultado de la operación módulo entre los dos parámetros que se le pasan, pero sólo se quiere que funcione cuando el tipo de los parámetros es *float*, y para el resto de los tipos siempre devuelva 0. La sintaxis debe ser de la siguiente manera:

```
// especialización de template
#include <iostream>
#include <cmath>
using namespace std;

template <class T>
class Par {
    T x, y;
```

```
0
1.41421
```

```
  public:
    Par (T primero, T segundo)
      {x=primero; y=segundo;}
    T modulo () {return (0);}
};

// particularizado para int:
template <>
class Par <float> {
    float x, y;
  public:
    Par (float primero, float segundo)
      { x=primero; y=segundo;}
    float modulo ();   //distinto al genérico
};

template <>
float Par<float>::modulo() {
  return (sqrt(x*x+y*y));
}

int main () {
  Par <int> mis_enteros (1,1);
  Par <float> mis_floats (1.0,1.0);
  cout << mis_enteros.modulo() << endl;
  cout << mis_floats.modulo() << endl;
  return (0);
}
```

En el código la especialización es definida de la siguiente manera:

template <> class *nombre_clase* **<***tipo***>**

La especialización es una parte de la plantilla, por lo que se debe comenzar la definición con *template* <>. Y como de hecho es una especialización para un tipo concreto de dato, el tipo genérico no puede ser usado aquí, y los primeros signos **<>** deben aparecer vacíos. Luego del nombre de la clase se debe incluir el tipo que está siendo especializado encerrado entre los signos **<>**.

Cuando se quiere especializar una plantilla también se deben redefinir los miembros adecuándolos a la especialización (en el ejemplo anterior se incluyó su propio constructor, aunque es idéntico al de la plantilla genérica). La razón es que **ningún miembro es heredado de la plantilla genérica a la especialización.**

Valores de parámetros para plantillas

En la sintaxis de *template* < …. > además de los argumentos precedidos por *class* (pudiéndose usar en algunos casos *typename*) que representan un tipo genérico, las funciones plantilla y las clases plantilla pueden incluir otros parámetros que no son tipos, si no que pueden ser valores constantes, como por ejemplo tipo de datos simples. El siguiente ejemplo describe una clase que posee una plantilla que permite indicar el tipo de datos y además cuántos valores se van a almacenar.

```
#include <iostream>
#include <vector>

using namespace std;

template <class T, int N>
```

```
class Almacen {
    vector <T> datos;
  public:
    Almacen () { datos.resize(N); }
    void cargar();
        void mostrar();
};

template <class T, int N>
void Almacen<T,N>::cargar()
{   for (int i=0; i<N; i++)
     {  cout << "Ingresar valor " << i << " : ";
         cin >> datos[i];
     }
}

template <class T, int N>
void Almacen<T,N>::mostrar()
{  for (int i=0; i<N; i++)
        cout << datos[i] << endl;
}

int main ()
{   Almacen <int, 5> almacen;
     almacen.cargar();
     almacen.mostrar();
     return (0);
}
```

Herencia de Clases con Plantillas

En el siguientes ejemplo se muestra el código para realizar herencias de una clase que posee una plantilla. En este ejemplo, se realiza un código simple para que una clase maneje coordenadas 2D. Luego, por herencia se crea otra clase para tres dimensiones.

```
#include <iostream>
#include <cmath>
using namespace std;

// clase para puntos bidimensionales
template <class T>
class DosD {
  protected:
    T x, y;
  public:
    DosD (T x_aux, T y_aux);
    void mostrar() { cout << x << "  " << y << " "; }
        double modulo();
};

template <class T>
DosD<T>::DosD(T x_aux, T y_aux)
{ x=x_aux; y=y_aux; }

template <class T>
double DosD<T>::modulo()
{  return ( sqrt( (double)(x*x+y*y) )); }

// ========================================

// clase para puntos tridimensionales
template <class T>
```

```
class TresDSinTipo : public DosD <T> {
     T z;
  public:
    TresDSinTipo(T x_aux, T y_aux, T z_aux);
    void mostrar() { DosD<T>::mostrar(); cout << " " << z << " "; }
    double modulo();
};

template <class T>
TresDSinTipo<T>::TresDSinTipo(T x_aux, T y_aux, T z_aux) : DosD<T>(x_aux,y_aux)
{
   z=z_aux;
}

template <class T>
double TresDSinTipo <T>::modulo()
//{  return ( sqrt(double(this->x * this->x + this->y * this->y + z * z))); }
// También puede ser la siguiente sintaxis:
{  return (sqrt(double(DosD<T>::x * DosD<T>::x + DosD<T>::y * DosD<T>::y + z * z))); }

// ========================================

int main ()
{
     DosD <double> dosD(1.001, 1.001);
     dosD.mostrar();
     cout << "Módulo: " << dosD.modulo()<<endl;

     TresDSinTipo <int> tresdsintipo(1, 1, 0);
     tresdsintipo.mostrar();
     cout << "Módulo: " << tresdsintipo.modulo()<<endl;

     return (0);
}
```

Tambien se puede usar template por defecto. Por ejemplo, en la clase para puntos bidimensionales se puede escribir

template <class T = int>
class DosD {
....
}

Luego, si se declara un objeto

DosD < > dosD(1, 1);

Se interpretará como interos.

Ejemplo completo con números complejos

El siguiente ejemplo utiliza sobrecarga de operadores, plantilla, función amiga y autoreferencia (*this) aplicado al manejo de números Complejos. Tener en cuenta que existe una biblioteca estándar para el manejo de estos números como se comenta en el apartado siguiente.

```
=========  ARCHIVO .H ==========

#ifndef COMPLEJO_H
#define COMPLEJO_H

# include <iostream>
```

```
using namespace std;

template <class T>
class CComplejo {
  private:
      T re, im;
  public:
    CComplejo ();
    CComplejo (T a, T b);
    CComplejo operator+ (CComplejo& C);
    CComplejo operator-(CComplejo& C);
    CComplejo operator*(CComplejo& C);
    CComplejo operator-();
    CComplejo operator++();
    CComplejo operator++(int notused);
    CComplejo operator!();
    bool operator ==(CComplejo C);
    bool operator !=(CComplejo C);
    friend ostream& operator<< <>(ostream& salida, CComplejo C);
    friend istream& operator>>  <>(istream& entrada, CComplejo& C);
    ~CComplejo();
};

template<class T>
CComplejo<T>::CComplejo ()
{
   //Constructor
}

template<class T>
CComplejo<T>::CComplejo (T a, T b) //Constructor sobrecargado
{
   re=a;
   im=b;
}

template<class T>
CComplejo<T> CComplejo<T>::operator+ (CComplejo<T>& C) // operador binario
{
 CComplejo<T> aux;
 aux.re = re + C.re;
 aux.im = im + C.im;
 return aux; // devuelve el resultado de la suma
}

template<class T>
CComplejo<T> CComplejo<T>::operator-(CComplejo<T>& C) // operador binario
{
 CComplejo<T> aux;
 aux=(*this);
 aux.re -=  C.re;
 aux.im -=  C.im;
 return aux; // devuelve el resultado de la suma
}

template<class T>
CComplejo<T> CComplejo<T>::operator*(CComplejo<T>& C)
{
 CComplejo<T> aux;
 aux.re = re * C.re - im * C.im;
 aux.im = re * C.im + C.re * im;
 return aux;
}

template<class T>
```

```
CComplejo<T> CComplejo<T>:: operator-() // operador unario
{
 CComplejo<T> aux(-re,-im);
 return aux;
}

template<class T>
CComplejo<T> CComplejo<T>::operator++()
{
  re++;
  im++;
  return (*this);
}

template<class T>
CComplejo<T> CComplejo<T>::operator++(int notused)
{
   CComplejo<T> aux(re, im);
   re+=1;
   im+=1;
   return aux;
}

template<class T>
CComplejo<T> CComplejo<T>::operator!()
{
 CComplejo<T> conjugado(re, -im);
 return conjugado;
}

template<class T>
bool CComplejo<T>::operator==(CComplejo<T> C)
{
 bool son_iguales=false;
 if((re == C.re)&&(im == C.im))
   son_iguales=true;

 return son_iguales;
}

template<class T>
bool CComplejo<T>::operator!=(CComplejo<T> C)
{
 bool son_distintos=false;
 if(!((*this) == C))
   son_distintos=true;

 return son_distintos;
}

template<class T>
ostream& operator<<(ostream& salida, CComplejo<T> C)
{
 salida << C.re;
 if (C.im>=0)
   salida << "+" << C.im << "i";
 else
   salida << C.im << "i";

 return salida;    // permite concatenar con otros "<<"
}

template<class T>
istream& operator>>(istream& entrada, CComplejo<T>& C)
{
 //Entrada de datos de la forma 2-3i
```

```
 entrada >> C.re;
 entrada >> C.im;
 entrada.ignore();

 return entrada;
}

template<class T>
CComplejo<T>::~CComplejo()
{
//Destructor
}

#endif

=========  ARCHIVO .CPP ==========

#include <iostream>
#include "Complejo.h"
int main ()
{
  CComplejo<float> a (4.1, 4.0);
  CComplejo<float> b, c, d;
   //Obsérvese que ahora se puede declarar complejos de distintos
   //tipos ingreso de un complejo mediante >>

 cout << "Ingrese un número complejo de la forma a+bi: ";
  cin >> b;

  cout << "\n*** c = a+b ***** ";
  c = a+b; // ver "a+b" como si fuera "a.+(b)"
  cout << c << endl;

  cout << "*** c = a+b+c *** ";
  c = a+b+c; // esta sintaxis también es válida
  cout << c << endl;

  cout << "*** c = a*b ***** ";
  c = a*b;
  cout << c <<endl;

  cout << "*** d = -c ****** ";
  d=-c;    // cambia signo y asigna
  cout << d << endl;

  cout << "*** c *** ++c **** c  ***** ";
  cout << c ;
  cout << "******" << ++c;
  cout   << " *****" << c << "*******" << endl;

  cout << "*** c *** c++ **** c  ***** ";
  cout << c ;
  cout << "******" << c++;
  cout   << " *****" << c << "*******" << endl;

  cout << "*** cout <<  a seguido de b... << ***** ";
  cout << a << "  y  " << b << endl;

  cout << "*** (-a) + a + b ***** ";
  cout << (-a) + a + b << endl;

  cout << "*** d = !c ****** ";
  d=!c;    // conjuga y asigna
  cout << d << endl;

  cout << "*** ¿es a == a ? (0->falso, 1->verdadero)****** ";
```

```
  cout << (a==a) << endl;

  cout << "*** ¿es a != a ? (0->falso, 1->verdadero)****** ";
  cout << (a!=a) << endl;

  cout <<"=== Fin ===" << endl;
  return (0);
}
```

Estándar para números complejos

El C++ tiene una potente librería estándar para números complejos. Se puede utilizar incluyendo en la librería del programa

#include <complex>

Las funciones incluidas en la librería estándar son las siguientes:

Denotamos un número complejo $c = x+iy$

norm(c)=x^2+y^2	sqrt(c)
abs(c)=$\sqrt{(x^2+y^2)}$	pow(c1,c2)
conj(c)=$x-iy$	pow(c1,r)
exp(c)	pow(c1,i)
sin(c)	log(c)
cos(c)	arg(c)
sinh(c)	+,-,*,/
cosh(c)	== ,!=
real(c)	+=,-=,*=,/=
imag(c)	

La forma de utilizarlo es la siguiente: Supongamos que queremos declarar complejos formados por parejas de enteros, *c1* y *c2*, formados por parejas de reales en simple precisión *c3* y *c4*, y formado por parejas en doble precisión *c5* y *c6*.

complex<int> c1,c2;

complex<float>c3,c4;

complex<double>c5,c6;

Para darle un valor a c3= 3.+4.i por ejemplo,

c3 = complex <double>(3.,4.)

Damos a continuación algunos ejemplos de utilización de funciones:

```
c4 = sin(c3)*sqrt(c3);

c5=pow(c4,c3);

c6 = log(c5);

complex <double> c7 = c6/c5*c4;
```

Las funciones están definidas para el *gcc* en

```
/usr/include/g++-3/std/
```

en los ficheros *complext.h* y *complext.cc*

Biblioteca de Plantillas Estándar (STL)

Introducción

La STL (del inglés *Standard Template Library*) es una biblioteca de clases y funciones *templates* creada para estandarizar y optimizar la utilización de algoritmos y estructuras de datos en el desarrollo de software en C++. La adopción de esta biblioteca posee grandes ventajas: al ser estándar está disponible en todos los compiladores y plataformas; está libre de errores, por lo tanto se ahorrará tiempo en depurar el código; proporciona su propia gestión de memoria. En este capítulo se presenta una descripción de los componentes más importantes de la STL junto con una gama de ejemplos que permitirán visualizar su funcionamiento y la conexión que existe entre ellos.

El diseño de la Standard Template Library es el resultado de varios años de investigación dirigidos por Alexander Stepanov y Meng Lee de Hewlett-Packard, y David Musser del Rensselaer Polytechnic Institute. Su desarrollo se inspiró en otras librerías orientadas a objetos y en la experiencia de sus creadores en lenguajes de programación imperativos y funcionales, tales como Ada y Scheme.

La biblioteca presenta tres componentes básicos: *contenedores*, *iteradores* y *algoritmos*. Los contenedores son los objetos capaces de almacenar otros objetos, cada uno de una forma particular. Representan a las estructuras de datos más comúnmente utilizadas, como ser los arreglos lineales o las listas enlazadas. Además éstos presentan otras características adicionales que los hacen más potentes. Por ejemplo: pueden aumentar el número de elementos que almacenan; al ser templates, pueden alojar cualquier tipo de dato o clase; casi todos los contenedores proveen iteradores (herramientas para poder acceder a sus elementos), lo que hace que los algoritmos que se aplican a ellos sean más eficientes. Los iteradores son objetos a través de los cuales se puede acceder a los elementos del contenedor. El concepto de iterador es similar al de un puntero, sólo que al ser una clase provee mayores utilidades que éste. Pero la gran utilidad de los iteradores ocurre por el uso que de ellos hacen los algoritmos. En la biblioteca existen más de setenta algoritmos para aplicar sobre los contenedores a través de los iteradores. Hay algoritmos de búsqueda, de ordenamiento, de transformación, matemáticos, etc.

Contenedores

Son una colección de las estructuras de datos más populares utilizadas habitualmente. Un contenedor es justamente eso: un lugar en donde contener o agrupar objetos del mismo tipo. La diferencia entre un contenedor y otro está en la forma en que los objetos son alojados, en cómo se crea la secuencia de elementos y la manera en que se podrá acceder a cada uno de ellos. Éstos pueden estar almacenados en forma contigua en la memoria o enlazados a través de punteros. Esto hace que las estructuras difieran también en la forma en que se accede a los elementos, la velocidad con la cual se insertan o se eliminan estos y en la eficiencia de los algoritmos que se apliquen a ellas.

La siguiente figura proporciona una clasificación de los contenedores de la STL. Éstos se dividen en contenedores de secuencia o lineales y contenedores asociativos. Los de secuencia son *vector*, *list* y *deque*. Los asociativos son *set*, *multiset*, *map* y *multimap*. En esta sección se describirán las operaciones comunes de los contenedores y las estructuras de secuencia solamente. Las asociativas se postergarán para luego de haber visto iteradores y algoritmos.

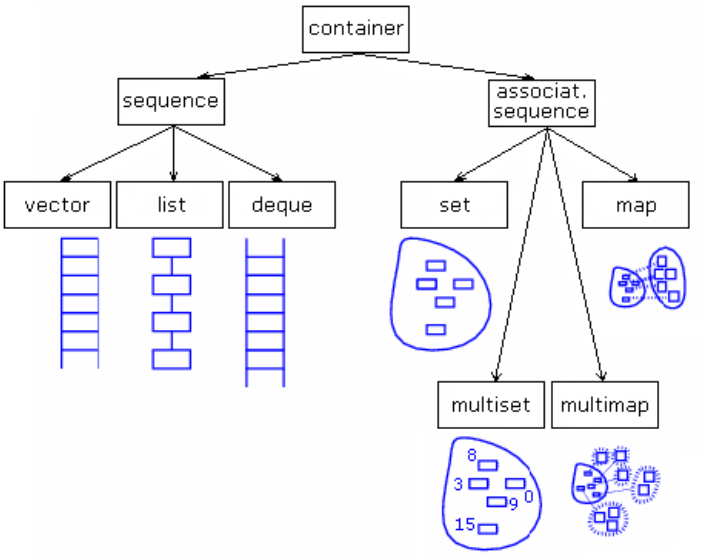


Figura 1: clasificación de los contenedores

Operaciones comunes

Antes de realizar cualquier operación con un contenedor hay que crearlo. La sintaxis utilizada para ello es la siguiente:

```
X < T >  instancia;
```

Donde X representa el tipo de contenedor que se quiere utilizar y T el tipo de dato de los elementos que almacenará la estructura. Así, por ejemplo, para crear un vector de enteros llamado "valores" se escribe:

```
vector < int >  valores;
```

Obsérvese que tanto vector como los demás contenedores son *clases template*. Por lo tanto, al hablar de clases, hablamos también de *constructores*. En la notación previa invocamos al constructor "vacío", es decir, sin parámetros. Esto hace que se cree un contenedor en memoria pero que no contiene aún ningún elemento. Esta es la sintaxis que se utilizará más a menudo, sin embargo, existirán ocasiones en que se necesite crear estructuras auxiliares que sean copias de otras preexistentes.

```
vector < int > aux (valores);
```

Aquí, "aux" es un vector de enteros y, además, es una copia exacta de "valores". En esta ocasión se utiliza el constructor de "copia". También es posible obtener el mismo resultado empleando el operador de asignación "**=**", como se sigue;

```
vector < int >  aux;
aux = valores;
```

Además de los constructores, los contenedores tienen una serie de operaciones que son comunes a todos ellos. Por otra parte existen funciones que se aplican sólo a contenedores lineales.

Tabla 1: operaciones comunes de contenedores

X::size()	Devuelve la cantidad de elementos que tiene el contenedor como un entero sin signo
X::max_size()	Devuelve el tamaño máximo que puede alcanzar el contenedor antes de requerir más memoria
X::empty()	Retorna verdadero si el contenedor no tiene elementos
X::swap(T & x)	Intercambia el contenido del contenedor con el que se recibe como parámetro
X::clear()	Elimina todos los elementos del contenedor
v == w v != w	Supóngase que existen dos contenedores del mismo tipo: *v* y *w*. Todas las comparaciones se hacen lexicográficamente y retornan un valor booleano.
v < w v > w	
v <= w v >= w	

Tabla 2: operaciones comunes de contenedores lineales

S::push_back(T & x)	Inserta un elemento al final de la estructura
S::pop_back()	Elimina un elemento del final de la estructura
S::front()	Devuelve una referencia al primer elemento de la lista
S::back()	Devuelve una referencia al último elemento de la lista

Vector

Representa al arreglo clásico de elementos, donde todos los elementos contenidos están contiguos en la memoria. Esta característica permite mayor velocidad de acceso a los elementos ya que si se quiere ver un elemento en alguna posición determinada solo se debe mover tantos lugares desde el principio del contenedor.

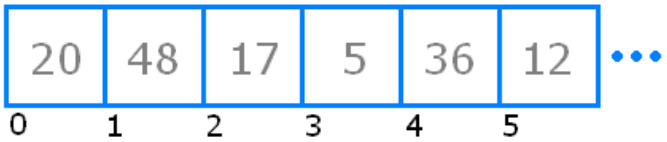


Figura 2: vector de seis elementos

La figura muestra una representación de un vector en la memoria y los índices de los elementos. En el siguiente ejemplo se muestran algunas operaciones básicas como agregar elementos, ver el contenido e insertar nuevos elementos. Se debe tener en cuenta que para poder utilizar vector se debe incluir <vector> en el programa.

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
 vector < int > datos;
 datos.push_back(20); // se insertan elementos en el final
 datos.push_back(48);
 datos.push_back(17);
 datos.push_back(5);
 datos.push_back(36);
 datos.push_back(12);

 for( unsigned i=0; i<datos.size(); ++i )
  cout << datos[i] << endl; // mostrar por pantalla

 return 0;
```

```
}
```

Como se observa en el ejemplo, se puede acceder a los elementos de vector a través de subíndices: datos[i]. "i" representa la posición en la memoria del elemento que se quiere acceder. Así, la salida en pantalla es el listado de los números contenidos en el orden en que fueron ingresados. Por medio de los subíndices también es posible modificar el contenido de un vector. Por ejemplo: datos[0] = 4, reemplaza el primer elemento (20) por 4. Tener en cuenta que en este ejemplo el arreglo es de sólo seis elementos, por lo que el índice máximo es 5. Si se trata de acceder a un elemento inexistente: datos[6], se generará un error en tiempo de ejecución.

En términos de eficiencia, los vectors son rápidos insertando o eliminando elementos al final de la estructura. También se pueden insertar elementos en una posición intermedia a través de iteradores, pero debido a que se tiene que mantener la contigüidad de los elementos en la memoria, deberán producirse desplazamientos e inserciones de manera de lograr generar el espacio para los elementos a insertar, el tiempo de inserción en posiciones intermedias es proporcional al tamaño del contenedor.

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
 vector <char> valores( 10 ); // iniciar con diez elementos

 // llenar con letras mayúsculas al azar
 for( unsigned i=0; i<valores.size(); ++i )
  valores[i] = 'A' + (rand() % 26);

 vector <char> aux( valores ); // aux es copia de valores

 // ordenar aux utilizando el método de burbujeo
 for( unsigned i=0; i<aux.size(); ++i )
  for( unsigned j=1; j<aux.size(); ++j )
   if( aux[j] < aux[j-1] )
   {
    char c = aux[j];
    aux[j] = aux[j-1];
    aux[j-1] = c;
   }

 // mostrar por pantalla
 for( unsigned i=0; i<aux.size(); ++i )
  cout << aux[i] << endl;

 return 0;
}
```

Deque

Esta es una estructura de datos que representa a una *cola* con *doble final*. Este contenedor es similar a vector ya que sus elementos también están contiguos en memoria. La diferencia principal radica en que al tener doble final se pueden insertar elementos por ambos extremos del contenedor.

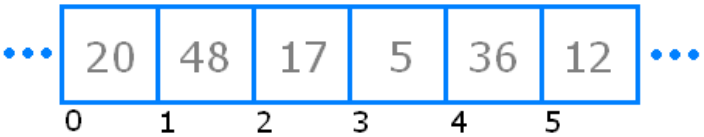


Figura 3: deque de seis elementos

Las deques tienen las mismas funcionalidades que vector, incluso se puede acceder a los elementos a través de subíndices (acceso aleatorio). Además, posee dos funciones más para insertar y eliminar elementos en la parte frontal del contenedor: *push_front(T & x)* y *pop_front()*, respectivamente.

```
#include <iostream>
#include <deque>
using namespace std;

int main()
{
 deque <char> datos;

 // cargar algunos datos
 datos.push_front('A');
 datos.push_front('B');
 datos.push_front('C');
 datos.push_back(65);
 datos.push_back('Z');

 // visualizar el contenido
 for( unsigned i=0; i<datos.size(); ++i )
  cout << datos[i];

 datos.pop_front(); // se elimina el primer elemento
 datos.push_back('C');
 cout << endl;

 for( int i=datos.size()-1; i>=0; --i )
  cout << datos[i];

 return 0;
}
```

Estos contenedores se utilizan cuando se deben insertar o eliminar varios elementos al principio o al final de la estructura. La velocidad de acceso a los elementos es rápida, aunque no tanto como vector. Si se necesitan funcionalidades típicas de arreglos es recomendable utilizar vector en vez de deque.

List

Las listas son los contenedores adecuados cuando se requieren operaciones de inserción o eliminación en *cualquier* parte de la lista. Están implementadas como *listas doblemente enlazadas*, esto es, cada elemento (nodo) contiene las direcciones del nodo siguiente y del anterior, además del valor específico almacenado.

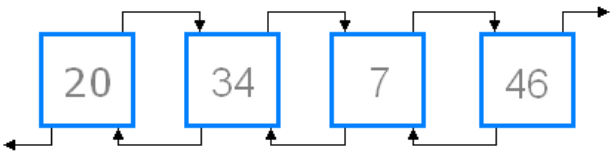


Figura 4: list de cuatro elementos

La ventaja de esta implementación es que la inserción o eliminación de un elemento se reduce a ordenar los punteros del siguiente y anterior de cada nodo. Pero la desventaja es que ya no se puede tener acceso aleatorio a los elementos, sino que se tiene un acceso secuencial en forma bidireccional. Es decir, se puede recorrer el contenedor desde el principio hasta el final o viceversa.

Para poder recorrer listas es necesario utilizar iteradotes (que más detalladamente se explicará más adelante). Como ya dijimos los iteradores son objetos similares a los punteros, que indican una posición dentro de un contenedor. Todos los contenedores proporcionan dos iteradores que establecen el *rango* del recorrido: *begin* y *end*.

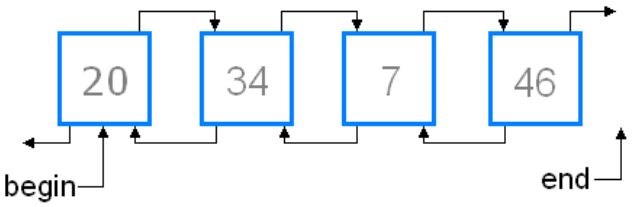


Figura 5: Iteradores de inicio y fin

El primero de ellos apunta al primer elemento de la lista y el segundo a una posición vacía (NULL) después del último elemento de la lista. El siguiente ejemplo muestra cómo se crean y utilizan estos iteradores para recorrer una lista.

```
#include <iostream>
#include <list>
using namespace std;

int main()
{
 list <int> datos;

 // llenar la lista con diez elementos aleatorios
 for( int i=0; i<10; ++i )
  datos.push_back( rand() % 10 );

 // ordenar la lista
 datos.sort();

 // crear un iterador para listas de enteros llamado p
 list <int>::iterator p;

 // hacer que p apunte al primer elemento de la lista
 p = datos.begin();

 // recorrer la lista incrementando p hasta llegar al final
 while( p != datos.end() )
 {
  // para ver el valor al que apunta p hay desrreferenciarlo
  // igual que a un puntero
  cout << *p << endl;

  // avanzar al siguiente elemento
  p++;
 }

 return 0;
}
```

Es importante notar que en este caso se accede a los elementos de la lista a través de un objeto externo al contenedor (el iterador p), el cual puede apuntar a cualquier elemento en el contenedor. Al incrementar el iterador (p++) se logra que éste apunte al elemento siguiente en la lista. Después de pasar por el último elemento, el iterador apunta a un lugar vacío, al igual que el iterador "end", lo que indica el fin del ciclo. Por otro lado, las listas proporcionan un método de ordenamiento *sort*, el cual ordena los elementos de menor a mayor. Esta funcionalidad no está implementada para vector y deque pero más adelante se verá que existe otra alternativa para estos casos.

Iteradores

Entender el concepto de iterador es la clave para comprender enteramente la estructura de la STL y hacer una mejor utilización de ella. Los algoritmos genéricos de esta biblioteca están escritos en términos de iteradores como parámetros y los contenedores proveen iteradores para que sean utilizados por los algoritmos. Estos componentes genéricos están diseñados para trabajar en conjunto y así producir un resultado óptimo.

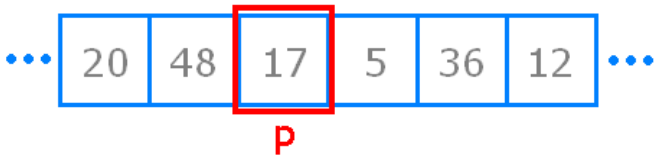


Figura 6: Esquema de un iterador "p"

Un iterador es un objeto que abstrae el proceso de moverse a través de una secuencia. El mismo permite seleccionar cada elemento de un contenedor sin la necesidad de conocer cómo es la estructura interna de ese contenedor. Esto permite crear algoritmos genéricos que funcionen con cualquier contenedor, utilizando operaciones comunes como ++, -- o *.

Ya hemos visto el empleo de iteradores con el contenedor list. La sintaxis general para crear un objeto iterador es la siguiente:

```
X::iterator instancia;
```

Donde "X" es el tipo de contenedor al cual estará asociado el iterador. Por ejemplo para crear un iterador a un deque de doubles llamado "inicio" se debería escribir:

```
deque<double>::iterator inicio;
```

En esta instancia se utiliza el constructor vacío, es decir que "inicio" no apunta a ningún elemento de ningún contenedor. Pero también es posible crear un iterador utilizando el constructor de copia:

```
deque <double> valores( 10,0 );
deque<double>::iterator inicio( valores.begin() );
deque<double>::iterator inicio2;
inicio2=valores.begin();
```

Ahora tanto inicio como inicio2 apuntan al mismo lugar que valores.begin(), es decir el primer elemento del contenedor valores.

Clasificación

Los algoritmos genéricos que se construyen empleando iteradores realizan distintas operaciones con ellos. Sin embargo, no todos los iteradores pueden soportar todas las operaciones posibles. Entonces, la clasificación que surge aquí es por la forma en que un iterador puede moverse a través de un contenedor:

Tabla 3: clasificación de iteradores

Forward iterators	Iteradores que pueden avanzar al elemento siguiente
Bidirectional iterators	Pueden avanzar al elemento siguiente o retroceder al anterior
Random acces iterators	Pueden avanzar o retroceder más de una posición de una vez

La siguiente imagen ilustra el conjunto de operaciones que se pueden realizar con estos tipos de iteradores.

Figura 7: operación con iteradores

También es importante conocer cuáles de estos iteradores proveen los contenedores antes vistos. Estas diferencias ocurren según la estructura interna de cada secuencia. En el caso de las listas doblemente enlazadas sólo se pueden realizar movimientos de avance o retroceso sobre la secuencia, por lo tanto, éstas proveen iteradores bidireccionales. Tanto vector como deque tienen sus elementos contiguos en memoria y permiten "saltar" a las diferentes posiciones sin mayor complicación. Estos contenedores proporcionan iteradores de acceso aleatorio.

```
#include <iostream>
#include <vector>
using namespace std;

// función template para mostrar los elementos
// de un contenedor por pantalla utilizando iteradores
template <class Iter>
void MostrarEnPantalla( Iter inicio, Iter final )
{
 while( inicio != final )
  cout << *inicio++ << " ";
}

int main()
{
 vector <char> letras( 20 ); // arreglo de 10 letras

 for( unsigned i=0; i<20; ++i )
  letras[i] = 'A' + (rand() % 26);

 // visualizar el contenido
 MostrarEnPantalla( letras.begin(), letras.end() );
 cout << endl;

 // visualizar el contenido en orden inverso
```

```
 MostrarEnPantalla( letras.rbegin(), letras.rend() );
 cout << endl;

 // visualizar sólo los 10 elementos del medio
 MostrarEnPantalla( letras.begin() + 5, letras.begin() + 15 );
 cout << endl;

 return 0;
}
```

Ahora se tiene un ejemplo que emplea iteradores de acceso aleatorio de un vector. La función MostrarEnPantalla recibe como parámetros los iteradores de inicio y fin del rango que se quiere mostrar en la pantalla. Lo nuevo que aparece aquí es la utilización de la función con los parámetros rbegin() y rend(). Estos nuevos iteradores son llamados iteradores inversos (*reverse iterators*), donde el primero de ellos apunta al último elemento del contenedor y el segundo a una posición antes del primero, y al incrementarlos (inicio++) se retrocede en una posición en la estructura. De esta forma se recorre el vector de atrás hacia delante. Por otro lado, al ser de acceso aleatorio se puede sumar posiciones a un iterador y de esa forma saltar a un elemento distante (6º elemento: *begin() + 5*, 16º elemento: *begin() + 15*).

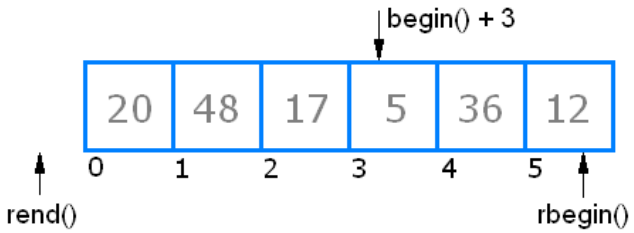


Figura 8: Iteradores inversos y aleatorios

Iteradores de entrada/salida

Existen además otros tipos de iteradores que permiten manipular objetos del tipo "streams" de entrada y salida como si fueran contenedores. Los streams más comunes son los archivos de texto y la consola. En este caso, los iteradores pueden avanzar a través de un archivo extrayendo la información que éste contiene (entrada) o escribiendo en él (salida).

Los streams_iterators son los objetos con los cuales se puede manipular estos archivos, son de un tipo similar a forward iterator y sólo pueden avanzar de a un elemento por vez desde el inicio del archivo. El siguiente ejemplo muestra cómo se utilizan para leer datos de un archivo y escribir los datos modificados en otro archivo.

```
#include <fstream> // para archivos
#include <iterator> // para streams_iterators
#include <vector>
using namespace std;

int main()
{
 // abrir el archivo para lectura
 ifstream archi( "datos.txt" );

 // crear un iterador de lectura para leer valores flotantes
 // en el constructor se indica a dónde apunta
 istream_iterator <float> p( archi );

 // crear un iterador que indique el fin del archivo
 istream_iterator <float> fin;
```

```
// crear un contenedor para almacenar lo que se lee
vector <float> arreglo;

// recorrer el archivo y guardar en memoria
while( p != fin )
{
 arreglo.push_back( *p );
 p++;
}

archi.close();

// calcular el valor medio de los elementos del contenedor
float v_medio = 0;
for( unsigned i=0; i<arreglo.size(); ++i )
 v_medio = v_medio + arreglo[i];
v_medio = v_medio/arreglo.size();

// restar el valor medio a cada elemento
for( unsigned i=0; i<arreglo.size(); ++i )
 arreglo[i] -= v_medio;

// crear un archivo para escritura
ofstream archi2( "datos_modif.txt" );

// crear un iterador de escritura para guardar los datos nuevos
ostream_iterator <float> q( archi2, "\n" );

// grabar los datos modificados en el archivo
for( unsigned i=0; i<arreglo.size(); ++i, q++ )
 *q = arreglo[i];

archi2.close();

return 0;
}
```

Algo particular para observar aquí es la forma en que se indica el rango en el cual deben actuar los iteradores. Para indicar que *p* apunta al inicio del archivo se pasa en el constructor el nombre lógico del archivo ya abierto en esa posición, esto hace que el iterador apunte a la misma posición en el archivo que el puntero de lectura del mismo. Cuando se quiere crear un iterador de *fin* sólo se debe crear uno que no apunte a nada (una dirección NULL). Esto es así debido a que cuando el iterador *p* llega al final del archivo (después de leer el último elemento) se encuentra con una dirección de memoria no asignada y por lo tanto apunta al mismo lugar que *fin* y el ciclo termina.
En el caso de los iteradores de escritura (*q*), se debe indicar en el constructor el nombre lógico del archivo en el que se quiere escribir y el caracter con el que se van a separar las respectivas escrituras de datos (en este caso "\n", un fin de línea). Cuando se incrementa el iterador (*q*++) se imprime el caracter delimitador en el flujo de datos.
Como se verá más adelante el uso de iteradores de flujo permite acelerar el proceso de lectura de datos desde archivos o consola. Esto se logrará mediante la utilización de los algoritmos definidos en la biblioteca STL.

Algoritmos

Como se mencionó anteriormente, existe una gran cantidad de algoritmos disponibles en la STL que pueden ser utilizados con los contenedores e iteradores que se explicaron hasta aquí. Hay algoritmos de ordenamiento, búsqueda, mezcla, matemáticos, etc. Estos algoritmos no son otra cosa que funciones template que operan sobre los contenedores a través de los iteradores de éstos. En esta sección se explicará la lógica utilizada en la creación de estos algoritmos y se expondrán ejemplos de aquellos más comúnmente utilizados.

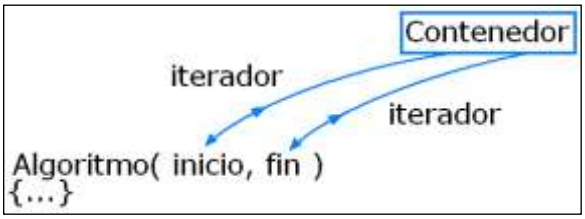


Figura 9: Relación entre los componentes

Estructura general

Para describir la estructura general que poseen los algoritmos de la biblioteca se utilizará un ejemplo.

```
template <class T>
T Max( vector <T> & v )
{
 // crear una variable para devolver el máximo
 T maximo = v[0];

 // buscar el máximo
 for( unsigned i=1; i<v.size(); ++i )
  if( maximo < v[i] )
   maximo = v[i];

 return maximo;
}
```

Aquí se tiene una función template para encontrar el elemento de máximo valor de un vector. El principal problema que tiene esta implementación es que sólo funciona con el contenedor vector y no con los demás contenedores. Otra característica es que el algoritmo busca el máximo en todo el contenedor. Sería bueno que la solución a este problema sea independiente del contenedor y además se pueda establecer un rango en el cual opere el algoritmo.

```
#include <iostream>
#include <vector>
using namespace std;

// crear un algoritmo genérico para encontrar el máximo
template <class Iter>
Iter Max( Iter inicio, Iter fin )
{
 // crear una variable para devolver el máximo
 Iter maximo = inicio;
 Iter aux = inicio;
```

```
 // buscar el máximo
 while( aux != fin )
 {
  if( (*maximo) < (*aux) )
   maximo = aux;

  ++aux;
 }

 return maximo;
}

int main()
{
 vector <int> datos;

 // llenar el vector con enteros al azar
 for( unsigned i=0; i<10; ++i )
  datos.push_back( rand() % 10 );

 // mostrar por pantalla
 for( unsigned i=0; i<datos.size(); ++i )
  cout << datos[i] << " ";

 // utilizar el algoritmo para encontrar el máximo
 cout << "\nMáximo: " << *( Max( datos.begin(), datos.end() ) );

 return 0;
}
```

Este nuevo ejemplo presenta mayores ventajas respecto del anterior. El algoritmo funciona con cualquier contenedor que proporcione iteradores; se puede cambiar el rango de búsqueda con sólo cambiar la posición de los iteradores de inicio y fin; el iterador que se devuelve en la función encapsula tanto la información (un entero en este caso) como la posición de ella en el contenedor, por lo tanto, se tiene como resultado el mayor elemento y su posición dentro de la secuencia.

Esta forma de implementación adoptada por la STL es diferente al estilo usual de objetos, donde las funciones son miembros de las clases. Los algoritmos están separados de los contenedores, lo que facilita la escritura de algoritmos genéricos que se apliquen a muchas clases de contenedores. De esta forma resulta muy fácil agregar nuevos algoritmos sin necesidad de modificar los contenedores de la STL. Además pueden operar sobre arreglos estáticos estilo C por medio de punteros.

```
#include <iostream>
#include <list>
#include <iterator>

// para los algoritmos de la STL:
#include <algorithm>

using namespace std;

int main()
{
 // generar una lista de valores al azar utilizando "generate"
 list <int> valores(10);
 generate( valores.begin(), valores.end(), rand );

 // mostrar los elementos por pantalla utilizando "copy"
 ostream_iterator<int> salida( cout, " " );
```

```
// El renglón anterior tiene:
// "salida" será el nombre del iterador que
// apunta donde va a copiar con "copy"
// Primer Parámetro: nombre lógico del archivo de texto de salida
//(por ejemplo puede ser la pantalla o un archivo en el disco).
// Aquí con cout es la pantalla
// Segundo Parámetro: Es usado para separar los datos.
// Aquí, " " indica espacios.

copy( valores.begin(), valores.end(), salida );

// buscar la primer aparición del número 5 utilizando "find"
list<int>::iterator p;
p = find( valores.begin(), valores.end(), 5 );

// analizar si el iterador quedó en el rango de búsqueda
if( p != valores.end() )
 cout << "\nEl valor " << *p << " está en la lista\n";
else
 cout << "\nNo se encontró el valor buscado\n";

 return 0;
}
```

En este ejemplo se utilizan tres algoritmos de la STL. En principio se utiliza *generate* para llenar una lista con elementos al azar. Simplemente se indica el rango de la lista y la función con la cual se generarán los elementos. Luego se emplea la función *copy* para copiar los datos almacenados en la lista a la pantalla. Este algoritmo recibe como parámetros los iteradores de inicio y fin de la *fuente* de copia, y como tercer parámetro un iterador que indique el inicio del *destino* de la copia. Cabe aclarar que el destino de la copia debe tener al menos la misma cantidad de elementos que el origen, de lo contrario se trataría de escribir en posiciones de memoria inexistentes y esto provocará un error difícil de detectar. En el caso de los streams este problema no ocurre debido a que los archivos van generando esos espacios de memoria a medida que ingresan los datos. Finalmente se procede a buscar un número particular en la secuencia con *find*, en este caso el 5. Al comparar el iterador p con end() se analiza si la búsqueda devolvió un iterador que esté dentro del rango de búsqueda. Esto es importante, ya que si se desrreferencia end() se producirá un error de acceso a memoria.

Tipos de funciones

Antes de continuar con la descripción de los algoritmos es necesario definir los diferentes tipos de funciones que existen ya que al igual que *generate* hay otros algoritmos que reciben como parámetro alguna función que establece la forma en la que éstos actúan o que opera sobre los elementos del contenedor.

Tabla 4: tipos de funciones

Tipo de función	Descripción
Función	Recibe un parámetro T y retorna void
Función Unaria	Recibe un parámetro T y devuelve un valor tipo T
Función Binaria	Recibe dos parámetros T y devuelve un valor T
Función Generadora	No recibe parámetros sólo devuelve un valor T
Función Lógica	Recibe un parámetro T y retorna un valor booleano
Función Lógica Binaria	Recibe dos parámetros T y retorna un valor booleano
Función de Comparación	Recibe dos parámetros T y retorna uno de ellos dependiendo de cómo se compare

Algoritmos de búsqueda y ordenamiento

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
using namespace std;

// Función lógica para saber si un número es par
template <class T>
bool EsPar( T val )
{
 return ( (val%2) == 0 );
}

int main()
{
 srand( (unsigned)time(0) );

 // generar una lista de valores al azar y mostrar por pantalla
 vector <int> valores(10);
 generate( valores.begin(), valores.end(), rand );
 copy( valores.begin(), valores.end(),
       ostream_iterator<int> ( cout, " " ) );

 // Buscar todos los números pares y ordenarlos en una lista
 // utilizando "find_if"
 vector <int> pares;
 cout << "\nValores pares: ";
 vector<int>::iterator p = valores.begin();
 while
 ((p=find_if( p, valores.end(), EsPar<int> )) != valores.end())
 {
  pares.push_back( *p );
  p++;
 }

 // ordenar la lista de menor a mayor utilzando "sort"
 sort( pares.begin(), pares.end() );

 // mostrarla por pantalla
 copy( pares.begin(), pares.end(),
       ostream_iterator<int> ( cout, " " ) );

 // hacer una búsqueda binaria de un valor
 if( binary_search( pares.begin(), pares.end(), 0 ) )
  cout << "\nEl valor 0 está en la lista";
 else
  cout << "\nEl valor 0 no está en la lista";

 return 0;
}
```

El algoritmo *find_if*, a diferencia del *find*, recibe como tercer parámetro una función lógica (*EsPar* en este caso) y retorna un iterador al primer elemento del rango que hace que esta función retorne un valor verdadero. De esta forma se puede realizar una búsqueda tan diversa como se quiera. Con *sort* se procede a ordenar los elementos del contenedor que están en el rango que se pasa como parámetro. La única condición para utilizarlo es que los iteradores deben ser de acceso aleatorio.

Algoritmos matemáticos

```
#include <iostream>
#include <fstream>
#include <vector>
#include <iterator>
#include <algorithm>
#include <numeric> // para accumulate

using namespace std;

// para contar la cantidad de elementos positivos
template <class T>
bool EsPositivo( T val )
{
 return val >= 0;
}

int main()
{
 // cargar una señal de archivo
 vector<float> senial;
 ifstream archi("ecg.txt");

 copy( istream_iterator<float>(archi), istream_iterator<float>(),
       back_inserter( senial ) );

 archi.close();

 // calcular los valores máximo y mínimo
 cout << "Máximo: "
      << *max_element( senial.begin(), senial.end() ) << endl;
 cout << "Mínimo: "
      << *min_element( senial.begin(), senial.end() ) << endl;

 // calcular el valor medio
 cout << "Valor medio: "
      << accumulate
         ( senial.begin(), senial.end(), 0.0 )/senial.size()
      << endl;

 // contar la cantidad de elementos positivos
 cout << count_if
         ( senial.begin(), senial.end(), EsPositivo<float> )
      << endl;

 return 0;
}
```

Los algoritmos matemáticos no modifican los elementos del contenedor sobre el que operan. Los dos primeros utilizados en el ejemplo, *max_element* y *min_element*, buscan el máximo y mínimo elemento de una secuencia respectivamente y retornan un iterador a él. *accumulate* suma los elementos que están en el rango más el tercer parámetro que es el valor inicial de la sumatoria. El cuarto algoritmo matemático es una variante de *count*. *count_if* cuenta la cantidad de elementos que satisfacen la función lógica que es pasada como parámetro.

Algo nuevo que aparece en este ejemplo es el uso de la clase *back_inserter*. Ésta recibe un contenedor como parámetro en el constructor y actúa como un iterador que cada vez que se incrementa agrega un

elemento al final de la estructura. La ventaja de su empleo es que el contenedor no tiene que tener un tamaño inicial conocido y así poder leer los elementos de un archivo con una cantidad indeterminada de elementos.

Algoritmos de transformación

```
#include <iostream>
#include <vector>
#include <iterator>
#include <algorithm>
using namespace std;

// función que pasa una letra a mayúsculas
void Mayusculas( char & c )
{
 if( ('a' <= c) && (c <= 'z') )
  c = c - ('a' - 'A');
}

// función unaria para identificar vocales
char Vocales( char c )
{
 return
 ( c=='A' || c=='E' || c=='I' || c=='O' || c=='U' )? c : '-';
}

int main()
{
 // generar un vector con letras en minúsculas
 vector<char> v;
 for( int i=0; i<10; ++i )
  v.push_back( 'a' + i );

 // desordenarlas
 random_shuffle( v.begin(), v.end() );

 copy( v.begin(), v.end(), ostream_iterator<char>(cout) );
 cout << endl;

 // aplicar a cada elemento la función Mayúsculas
 for_each( v.begin(), v.end(), Mayusculas );

 copy( v.begin(), v.end(), ostream_iterator<char>(cout) );
 cout << endl;

 // buscar las vocales
 transform( v.begin(), v.end(),
            ostream_iterator<char>(cout), Vocales );

 return 0;
}
```

En este ejemplo se tiene un contenedor de letras al cual se le aplican transformaciones. Primero se desordena la secuencia por medio de *random_shuffle*. Luego *for_each* aplica una función a cada elemento del rango. En este caso, como la función *Mayusculas* modifica el parámetro, se modifican los elementos del contenedor. El tercer algoritmo empleado es el más genérico. *transform* aplica punto a punto la función unaria, pasada por parámetro, a la secuencia, y guarda el resultado en el lugar indicado por el tercer parámetro.

Tabla 5: resumen de algoritmos

return	Algoritmo	Parámetros	Descripción
int	count	FWIter primero, FWIter ultimo, T val	Devuelve la cantidad de apariciones de val
int	count_if	FWIter primero, FWIter ultimo, PredFunc pred	Devuelve la cantidad de elementos que satisfacen la función predicado
INIter	find	INIter primero, INIter ultimo, T val	Devuelve un iterador al primer elemento igual a val, o ultimo si no existe
INIter	find_if	INIter primero, INIter ultimo, PredFunc pred	Devuelve un iterador al primer elemento que satisface la función
bool	equal	INIter1 primero1, INIter1 ultimo1, INIter2 primero2	Devuelve true si ambos tienen los mismos elementos
FWIter1	search	FWIter1 primero1, FWIter1 ultimo1, FWIter2 primero2, FWIter2 ultimo2	Localiza la aparición del segundo contenedor en el primero
FWIter	min_element	FWIter primero, FWIter ultimo	Devuelve un iterador al menor elemento
FWIter	max_element	FWIter primero, FWIter ultimo	Devuelve un iterador al mayor elemento
OUTIter	copy	INIter primero1, INIter ultimo1, OUTIter primero2	Copia el contenido del primero en el segundo
void	swap	T & p, T & q	Intercambia los valores de p y q
Func	for_each	INIter primero, INIter ultimo, Func f	Aplica la función f a cada elemento del contenedor
OUTIter	transform	INIter primero, INIter ultimo, OUTIter result, UNFunc f	Aplica f a cada elemento del contenedor y guarda el resultado en result
OUTIter	transform	INIter1 primero1, INIter1 ultimo1, INIter2 primero2, OUTIter result, BINFunc f	Aplica f a ambos contenedores y guarda el resultado en result
void	replace	FWIter primero, FWIter ultimo, T & valor_viejo, T & valor_nuevo	Reemplaza todos los valores viejos por los nuevos
void	fill	FWIter primero, FWIter ultimo, T & val	Llena el contenedor con val
FWIter	remove	FWIter primero, FWIter ultimo, T & val	Elimina todas las apariciones de val
FWIter	unique	FWIter primero, FWIter ultimo	Elimina los

			elementos contiguos repetidos
void	reverse	BDIter primero, BDIter ultimo	Invierte el orden de la lista
void	rotate	FWIter primero, FWIter centro, FWIter ultimo	Rota la lista hacia la derecha hasta que primero == centro
void	Random_shuffle	RNDIter primero, RNDIter ultimo	Reordena los elementos al azar
void	sort	RNDIter primero, RNDIter ultimo	Ordena los elementos de menor a mayor
T	accumulate	INIter primero, INIter ultimo, T val_ini	Da como resultado val_ini más la sumatoria de los elementos del contenedor
T	inner_product	INIter1 primero1, INIter1 ultimo1, INIter2 primero2, T val_ini	Producto interno entre los contenedores más val_ini

Contenedores asociativos

Hasta este momento se explicó la estructura general de la STL con los contenedores lineales, los iteradores y los algoritmos. Con esta idea general, se retoma la descripción de los contenedores faltantes, los *asociativos*. Estos contenedores son: *map*, *multimap*, *set* y *multiset*. Ellos tienen la característica de almacenar claves ordenadas que están asociadas a un valor, y en el caso de set y multiset la clave es el valor en sí mismo.

Su estructura en memoria no es secuencial como en los contenedores anteriores sino que se implementan como árboles binarios de búsqueda balanceados. Esto hace que el tiempo de búsqueda sea proporcional al logaritmo en base dos de la cantidad de elementos, en vez de ser proporcional al tamaño del contenedor, como es el caso de las listas lineales.

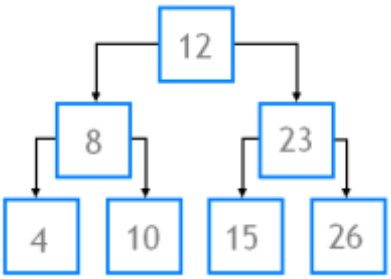


Figura 10: Representación esquemática de un árbol binario de búsqueda

La operación de comparación para el ordenamiento de las claves se establece en la instanciación, y por defecto se realiza con el operador menor (<) para un ordenamiento ascendente. Se debe tener en cuenta que los tipos de datos utilizados como clave deben "soportar" (deben tener sobrecargado) el operador que se utilice para la comparación, en caso de ser clases creadas por el programador.

Los contenedores asociativos proveen para su manipulación iteradores bidireccionales, al igual que las listas doblemente enlazadas. Por lo tanto, sólo se podrán utilizar aquellos algoritmos que requieran de estos iteradores. Sin embargo, estos contenedores proveen algunos métodos para las funciones de búsqueda y conteo, los cuales se explican a continuación.

Tabla 6: operaciones comunes en contenedores asociativos

A::insert(clave & x)	Inserta el elemento x en el contenedor
A::insert(A::iterator i, A::iterator f)	Inserta los elementos que están en el rango de los iteradores en el contenedor
A::erase(clave & x)	Borra todos los elementos que tengan la clave x
A::erase(A::iterator p)	Borra el elemento apuntado por p
A::count(clave & x)	Devuelve la cantidad de elementos que tienen la clave x
A::find(clave & x)	Devuelve un iterador al primer elemento que tenga la clave x
A::lower_bound(clave & x)	Devuelve un iterador al primer elemento que tenga la clave x
A::upper_bound(clave & x)	Devuelve un iterador al elemento siguiente al último elemento con clave x

Set y Multiset

Estos contenedores se utilizan para manipular conjuntos de elementos, especialmente en operaciones de búsqueda. Tanto en set como en multiset, la clave de ordenamiento es el valor que se quiere contener. La diferencia entre estos contenedores es que set no permite la existencia de claves repetidas, mientras que multiset sí.

```
#include <iostream>
#include <fstream>
#include <iterator>

// para set y multiset
#include <set>

using namespace std;

int main()
{
 // cargar los datos de un archivo en memoria
 set<int> valores;
 ifstream archi("datos.txt");
 istream_iterator<int> i(archi);
 istream_iterator<int> f;
 while( i != f)
  valores.insert( *i++ );
 archi.close();

 // mostrar los elementos por pantalla
 copy( valores.begin(), valores.end(),
       ostream_iterator<int>( cout, " " ) );

 // buscar un elemento en la estructura
 set<int>::iterator p = valores.find( 0 );

 if( p != valores.end() )
  cout << "\nEl valor 0 está en el archivo\n";
 else
  cout << "\nEl valor 0 no está en el archivo\n";

 // comparar con multiset
 multiset<int> multivalores;
 archi.open("datos.txt");
 istream_iterator<int> j(archi);
 while( j != f)
```

```
 multivalores.insert( *j++ );
archi.close();

// mostrar los elementos por pantalla
copy( multivalores.begin(), multivalores.end(),
      ostream_iterator<int>( cout, " " ) );

// contar las apariciones de 0
cout << "\nEl valor 0 está "
     << multivalores.count( 0 )
     << " veces en el archivo\n";

// eliminar un elemento
multivalores.erase( 2 );
copy( multivalores.begin(), multivalores.end(),
      ostream_iterator<int>( cout, " " ) );

return 0;
}
```

Aquí se tiene un ejemplo básico del empleo de estos contenedores asociativos para conocer si un valor particular se encuentra en un conjunto de elementos. Observar que con estas estructuras no se emplea la función push_back() como se hacía con las estructuras lineales; en cambio, se emplea la función *insert()*. Esto es lógico debido a que el elemento que se agregue al contenedor quedará ordenado en la estructura, y su posición no será en el final precisamente. Otra función se utiliza es *erase()*, que borra todas las claves que coincidan con el parámetro. El funcionamiento de *find()* y *count()* es el mismo que se vio anteriormente, solo que reciben como parámetro únicamente el valor a buscar o contar.

Pair

Antes de continuar con la descripción de map y multimap, se presenta una estructura que es utilizada internamente por estos contenedores. Los pairs (pares) son estructuras genéricas que contienen dos datos: *first* y *second*. Donde el primero de ellos es utilizado como clave (key) para la búsqueda de elementos y el segundo como el valor asociado a esta clave (value). Para utilizarlos se debe incluir la biblioteca *utility* de la STL.

```
template <class T1, class T2>
struct pair
{
 T1 first;
 T2 second;
 pair() {}
 pair( const T1& a, const T2& b ):
  first(a), second(b) {}
};
```

Estas estructuras se utilizan también en algunos métodos como valor de retorno, cuando se quiere devolver dos datos. Esto sucede, por ejemplo, en la función insert de los contenedores asociativos set y map, donde el resultado es un pair que tiene como primer elemento el iterador que apunta al elemento insertado y como segundo elemento un valor booleano que es verdadero si el elemento no estaba en el contenedor.

Map y Multimap

Estos contenedores mantienen la estructura de árbol al igual que set y multiset, pero almacenan los elementos de a pares: la *clave* y el *valor* asociado. Estas estructuras mantienen ordenados los pares por medio de sus claves, que por defecto lo hacen en orden ascendente. De esta manera, los maps son útiles cuando se necesita buscar determinada información en un volumen grande a partir de algún dato de búsqueda, que será la clave, o cuando se necesitan tablas de datos apareadas.

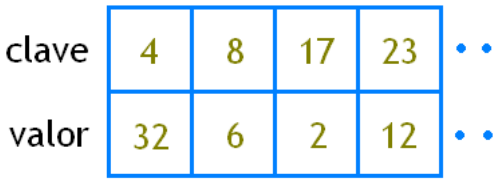


Figura 11: Representación de un map en memoria

En la figura se observa un esquema simplificado de la estructura en memoria de un map. Al igual que set y multiset, la diferencia entre map y multimap yace en que los últimos permiten que existan elementos con claves repetidas, como se puede apreciar en la siguiente figura. Obsérvese que sólo se ordenan las claves y no los valores asociados.

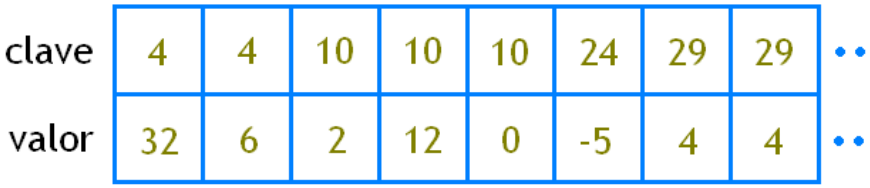


Figura 12: Multimap

En los siguientes ejemplos se expondrán las funcionalidades que estos contenedores poseen para la inserción, búsqueda y conteo de información. Para comenzar, se desea conocer la frecuencia de repetición de los valores de una secuencia de valores de punto flotantes almacenados en un archivo de texto. Se hará un mapeo de los valores y la cantidad de veces que aparece en el archivo.

```
#include <iostream>
#include <fstream>
#include <map> // para map y multimap
using namespace std;

int main()
{
 // crear un map donde la clave es el valor del archivo
 // y su valor asociado es su repetición
 map< float, unsigned > m;
 ifstream archi( "datos.txt" );
 float aux;

 while( archi >> aux )
 {
  // preguntar si ese valor no existe en la estructura
  if( m.find(aux) == m.end() )
   m[aux] = 1;
  // si ya existe, incrementar en 1 su repetición
  else
   m[aux] = m[aux] + 1;
 }
 archi.close();
```

```
    // mostrar en pantalla los datos encontrados
    map< float, unsigned >::iterator i = m.begin();
    while( i != m.end() )
    {
     cout << i->first << " " << i->second << endl;
     i++;
    }

    return 0;
   }
```

Esta resolución del problema planteado presenta una funcionalidad extra de los maps que no poseen set, multiset ni multimap: el empleo del operador []. Con él se puede insertar un par de elementos dentro de la estructura: la clave va entre corchetes y el valor asociado es el valor que se asigna. Así, por ejemplo, para insertar el par (-2.34, 5) en un map se debería escribir:

```
m[-2.34] = 5;
```

Si no se asigna ningún valor cuando se crea un par nuevo, el valor asociado será creado con su constructor por defecto. Otro detalle a tener en cuenta es el empleo de iteradores con map. En estas estructuras los iteradores apuntan a un par de elementos, por lo tanto, al desreferenciar un iterador se obtiene una estructura pair, donde el primer elemento (first) es la clave y el segundo (second) es el valor asociado.

El siguiente problema que se resolverá consiste en crear un objeto que busque información de una persona de un archivo a través de su identificación. El archivo tiene la información de personas que asistieron a una campaña de vacunación y para cada persona la información está codificada de la siguiente manera: ID (cadena de 20 caracteres), nombre (cadena de 20 caracteres), edad (char), sexo (char), domicilio (cadena de 50 caracteres) y teléfono (cadena de 10 caracteres). Para esto en el constructor de este objeto se cargará toda la información relevante de las personas en una estructura map.

```
#ifndef BUSCADOR_H
#define BUSCADOR_H

#include <fstream>
#include <string>
#include <map>
using namespace std;

class Buscador
{
 private:
  map< string, long int > map_pos;
  ifstream archi;
 public:
  Buscador( string nombre_archi );
  ~Buscador();
  void BuscarDatos( string ID );
};

Buscador::Buscador( string nombre_archi )
{ // constructor del índice del archivo
 archi.open( nombre_archi.c_str(),ios::in | ios::binary | ios::ate);
 long int pos_final = archi.tellg();
 archi.seekg( 0, ios::beg );
 char ID[20];
 long int pos;
```

```
// para saltar el nombre, edad, sexo, domicilio y teléfono
int salto=20*sizeof(char)+1*sizeof(char)+
          1*sizeof(char)+50*sizeof(char)+10*sizeof(char);

 while( archi.tellg() != pos_final )
 {
  archi.read( ID, sizeof(ID) );
  pos = archi.tellg();
  map_pos[ string(ID) ] = pos;
  archi.seekg( salto,ios::cur );
 }
}

Buscador::~Buscador()
{
 archi.close();
}

void Buscador::BuscarDatos( string ID )
{
 if( map_pos.find(ID) != map_pos.end() )
 {
  archi.seekg( map_pos[ID], ios::beg );
  char nombre[20], edad, sexo, domicilio[50], telefono[10];
  archi.read( nombre, sizeof(nombre) );
  archi.read( (char*)&edad, sizeof(edad) );
  archi.read( (char*)&sexo, sizeof(sexo) );
  archi.read( domicilio, sizeof(domicilio) );
  archi.read( telefono, sizeof(telefono) );

  cout << "Los datos de la persona con el ID N°: " << ID
       << " son:\n"
       << "Nombre: " << nombre << endl
       << "Edad: " << edad << endl
       << "Sexo: " << sexo << endl
       << "Domicilio: " << domicilio << endl
       << "Teléfono: " << telefono << endl;
 }
 else
  cout << "No se encontró a la persona con el ID " << ID << endl;
}

#endif
```

Ejemplos de uso de la STL

A través del análisis de ejemplos realizados con contenedores se puede observar la potencia y simplicidad del uso de la biblioteca.

La clase *vector*

Creación

```
vector<int> iv;            // crea un vector de int de longitud cero
vector<char> cv(20);       // crea un vector con 20 elementos char
vector<char> cv(5,  'x');  // inicializa con 5 elementos char
vector<int> iv2(iv);       // crea un vector a igual a otro
```

Operaciones básicas

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
  vector<int> v(10);
  unsigned int i;

  // muestra el tamaño original de v
  cout << "Tamaño = " << v.size() << endl;

  // asigna los elementos al vector
  for(i=0; i<10; i++) v[i] = i;

  // muestra el contenido del vector
  cout << "Contenido actual:" << endl;
  for(i=0; i<v.size(); i++)
    cout << v[i] << " ";
  cout << endl << endl;

  cout << "Expandiendo el vector" << endl;
  // poner más valores al final del vector y agrandar lo necesario
  for(i=0; i<5; i++) v.push_back(i + 10);

  // mostrar los valores actuales de v
  cout << "Tamaño actual = " << v.size() << endl;

  // cambiar el contenido del vector
  for(i=0; i<v.size(); i++) v[i] = -v[i];

  // mostrar el contenido del vector
  cout << "Contenido actual:" << endl;
  for(i=0; i<v.size(); i++)
    cout << v[i] << " ";
  cout << endl << endl;

  cin.get();
  return (0);
}
```

Uso de *pop_back()* y *empty()*

```
// Uso de pop_back() y empty().
#include <iostream>
#include <vector>
using namespace std;

int main()
{
  vector<char> v;
  unsigned int i;

  for(i=0; i<10; i++)
    v.push_back(i + 'A');

  cout << "Contenido del vector original:" << endl;
```

```
  for(i=0; i<v.size(); i++)
    cout << v[i] << " ";

  cout << endl << endl;

  do {
    v.pop_back(); // remover un elemento del final

    cout << "El vector ahora contiene:" << endl;
    for(i=0; i<v.size(); i++)
      cout << v[i] << " ";
    cout << endl;

  } while( !v.empty() );

  cin.get();
  return (0);
}
```

Acceso a través de iteradores

```
// Acceso a los elementos de un vector a través de un iterador.
#include <iostream>
#include <vector>
using namespace std;

int main()
{
  vector<int> v(10);       // crea un vector de longitud 10
  vector<int>::iterator p; // crea un iterador a un "vector <int>"
  unsigned int i;

  // asigna elementos
  p = v.begin();
  i = 0;
  while(p != v.end()) {
    *p = i; // asigna i a v a través del iterador p
    p++;    // avanza el iterador
    i++;
  }

  // cambia el contenido del vector
  p = v.begin();
  while(p != v.end()) {
    *p = *p * 2;
    p++;
  }

  // muestra el contenido del vector
  cout << "Contenido original:" << endl;
  p = v.begin();
  while(p != v.end()) {
    cout << *p << " ";
    p++;
  }
  cout << endl << endl;

  cin.get();
  return (0);
}
```

Sobrecargas para contener dos números

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>    // para usar accumulate

using namespace std;

class Muestra
{
   public:
      float tiempo;
      float valor;
      Muestra (float t, float v)  { tiempo=t; valor=v; }
      Muestra () {tiempo=0; valor=0;};
      void cargar (float t, float v)  { tiempo=t; valor=v; }
      bool operator<(Muestra m) { return (valor < m.valor ? true:false);}
      Muestra operator+(Muestra m);
      friend ostream & operator << (ostream& sal, Muestra m);
};

Muestra Muestra::operator+(Muestra m)
{ Muestra respuesta;
   respuesta.valor = valor + m.valor;
   respuesta.tiempo = tiempo;

   return respuesta;
}

ostream& operator<< (ostream& sal, Muestra m)
{ cout << m.tiempo << "  " << m.valor << endl;
   return sal;
}

//  si se quiere usar el  sort( muestras.begin(), muestras.end() );
//  de la algorithm
//  hay que sobrecargar el operador "<" en forma global de la siguiente manera:
//  bool operator< ( Muestra M1, Muestra M2 )
//  { return (M1.valor < M2.valor ? true:false); }

int main () {

  vector<Muestra> muestras;
   srand(time(0));

   for (int i=0; i<3; i++)
   {   Muestra m (rand() % 10,rand() % 10);
        muestras.push_back(m);
   }

   for (uint i=0; i<muestras.size(); i++)
     cout << muestras[i];

   // Cálculo valor máximo y mínimo
   float val_max=(*max_element(muestras.begin(),muestras.end())).valor;
   cout << "Valor Máximo: " << val_max << endl;

   float val_min=(*min_element(muestras.begin(),muestras.end())).valor;
```

```
   cout << "Valor Mínimo: " << val_min << endl;

   // Cálculo del promedio
   Muestra muestraNula (0.0,0.0); //para el tercer parámetro de accumulate
   float promedio= (accumulate(muestras.begin(),muestras.end(),
                   muestraNula ).valor ) / muestras.size();
   cout << "Promedio: " <<  promedio << endl;

   cin.get();
   return (0);
}
```

La clase *deque*

Operaciones básicas

```
#include <iostream>
#include <deque>
#include <string>
#include <algorithm>

using namespace std;

int main()
{
  deque<char> q1;
  string str = "-Hola Mundo.";

  // cargamos el texto en forma simétrica
  for(unsigned i=0; i<str.size(); i++) {
    q1.push_front(str[i]);
    q1.push_back(str[i]);
  }

  cout << "Mostramos q1 (texto simétrico):" << endl;
  for(unsigned i=0; i<q1.size(); i++)
    cout << q1[i];
  cout << endl << endl;

  // remover valores de la deque desde el principio
  for(unsigned i=0; i<str.size(); i++) q1.pop_front();

  cout << "q1 después de quitar del principio:" << endl;
  for(unsigned i=0; i<q1.size(); i++)
    cout << q1[i];
  cout << endl << endl;

  deque<char> q2(q1); // construir una copia de q1
  cout << "Contenido original de q2" << endl;
  for(unsigned i=0; i<q2.size(); i++)
    cout << q2[i];

  cout << endl << endl;

  cout << "Apuntar con un iterador a la primera ocurrencia de 'a': " << endl;
  deque<char>::iterator p = q1.begin();
  while(p != q1.end()) {
    if(*p == 'a') break;
    p++;
```

```
  }

  cout  << "*p: " << *p << endl;

  // insertar
  q1.insert(p,'x');
  cout << "Luego de q1.insert(p,'x'):  " << endl;

  for(unsigned i=0; i<q1.size(); i++)
    cout << q1[i];

  // erase
  q1.erase(q1.begin(),q1.begin()+2);
  cout << "Luego de q1.erase(q1.first(),q1.first()+2):  " << endl;
  for(unsigned i=0; i<q1.size(); i++)
    cout << q1[i];

  return (0);
}
```

La clase *list*

Operaciones básicas

```
// El programa crea una lista de enteros.
#include <iostream>
#include <list>
using namespace std;

int main()
{
  list<int> lst; // crea una lista vacía
  unsigned int i;

  for(i=0; i<10; i++) lst.push_back(i);

  cout << "Tamaño = " << lst.size() << endl;

  cout << "Contenido: ";
  list<int>::iterator p = lst.begin();
  while(p != lst.end()) {
    cout << *p << " ";
    p++;
  }
  cout << endl << endl;

  // cambia el contenido de la lista
  p = lst.begin();
  while(p != lst.end()) {  //for(p=lst.begin();p!=lst.end();p++)
    *p = *p + 100;
    p++;
  }

  cout << "Contenido modificado: ";
  p = lst.begin();
  while(p != lst.end()) {
    cout << *p << " ";
    p++;
  }
```

```
  cin.get();
  return (0);
}
```

Push_back( ) y push_front( )

```
// Muestra la diferencia entre push_back() y push_front()
#include <iostream>
#include <list>
using namespace std;

int main()
{
  list<int> lst1, lst2;
  unsigned int i;

  for(i=0; i<10; i++) lst1.push_back(i);
  for(i=0; i<10; i++) lst2.push_front(i);

  list<int>::iterator p;

  cout << "Contenido de la lista lst1: " << endl;
  p = lst1.begin();
  while(p != lst1.end()) {
    cout << *p << " ";
    p++;
  }
  cout << endl  << endl ;

  cout << "Contenido de la lista lst2:" << endl;
  p = lst2.begin();
  while(p != lst2.end()) {
    cout << *p << " ";
    p++;
  }

  cin.get();
  return (0);
}
```

Lista ordenada

```
// El siguiente programa crea una lista de números aleatorios enteros
// y los pone en una lista ordenada.
#include <iostream>
#include <list>
#include <cstdlib>
using namespace std;

int main()
{
  list<int> lst;
  unsigned int i;

  // crea una lista de enteros aleatorios
  for(i=0; i<10; i++)
    lst.push_back(rand()%50);

  cout << "Contenido original:" << endl;
```

```
  list<int>::iterator p = lst.begin();
  while(p != lst.end()) {
    cout << *p << " ";
    p++;
  }
  cout << endl  << endl ;

  // ordena la lista con un método de la clase list
  lst.sort();

  cout << "Contenido ordenado:" << endl;
  p = lst.begin();
  while(p != lst.end()) {
    cout << *p << " ";
    p++;
  }

  cin.get();
  return (0);
}
```

Lista con Plantilla

```
// Realizar una clase template que tenga una list que se mantenga ordenada.
// Si obj es un objeto de esta clase, permitir que:
//   obj ["min"] de el valor mínimos.
//   obj ["max"] de el valor máximo.
// Sobrecargar del operador << para mostrar todos los valores.

#include <iostream>
#include <list>
#include <iterator>
using namespace std;

template <class T>
class Lista {
   private:
     list <T> valores;
   public:
     void cargar (T n);
      T operator [] (string s);
      template <class U> friend ostream& operator << (ostream &sal, Lista <U> l);
      // como es friend, no pertenece a la clase,
      // así que no tiene T sino otra letra, en este caso U.
};

template <class T>
void Lista <T>::cargar(T n)
{  valores.push_back(n);
        valores.sort();
}

template <class T>
T Lista<T>::operator [] (string s)
{  T valor;
   if (s=="min") valor=*valores.begin();
   if (s=="max") valor=*(--valores.end());
   return valor;
}
```

```
template <class U>
ostream& operator << (ostream &sal, Lista <U> l)
{  typename list <U> :: iterator p=l.valores.begin();
   // typename es para que no interprete que p es un atributo de las list

   while (p != l.valores.end())
   { cout << *p << " ";
     p++;
   }

   return sal;
}

int main()
{ Lista <int> lista;

  // cargar valores
  for(unsigned i=0; i<5; i++)
  {  lista.cargar(i*2); }

     cout << "Valores Cargados: " << lista << endl;
     cout << "Mayor: " << lista["max"] << endl;
     cout << "Menor: " << lista["min"] << endl;

     return 0;
}
```

Unique( )

```
// Ejemplo de unique().
// El método unique() de la clase list remueve elementos consecutivos duplicados.
#include <iostream>
#include <list>
using namespace std;

int main()
{
  list<int> lst;
  list<int>::iterator p;

  for(int i=0; i<5; i++)
    for(int j=0; j<3; j++) lst.push_back(i);

  lst.push_back(1);  // repetido no consecutivo

  cout << "Lista original: ";
  for(p=lst.begin(); p!=lst.end(); p++)
    cout << *p << " ";

  cout << endl << endl;

  lst.unique(); // remueve elementos consecutivos duplicados

  cout << "Lista modificada: ";
  for(p=lst.begin(); p!=lst.end(); p++)
    cout << *p << " ";

  cout << endl  << endl;
```

```
  cin.get();
  return (0);
}
```

La clase *map*

Operaciones básicas

```
// El programa almacena el par de valores código_ASCII/valor.
#include <iostream>
#include <map>
using namespace std;

int main()
{
  map<char, int> m;
  int i;

  // ingreso de valores con [ ]
  m['A']=65; m['B]=66; m['C']=67; m['D']=68; m['E']=69;
  // en el código anterior observer que no se tuvo
  // previamente que reservar memoria.

  // buscamos el código ASCII de una letra
  char ch='D';
  cout << "El código de " << ch << " es " << m[ch];

  cin.get();
  return (0);
}
```

Usando iteradores

```
// Ciclo a través de map usando iteradores.
#include <iostream>
#include <map>
using namespace std;

int main()
{
  map<char, int> m;
  int i;

  // ingreso de valores sin [ ]
  m.insert(pair<char, int>('A', 65));
  m.insert(pair<char, int>('B', 66));
  m.insert(pair<char, int>('C', 67));
  m[‘D’]= 68;
  m[‘E’]= 69;

  map<char, int>::iterator p;

  // Muestra el contenido de map
  for(p = m.begin(); p != m.end(); p++) {
    cout << p->first << " le corresponde el código ASCII: ";
    cout << p->second << endl;
  }
```

```
  cin.get();
  return (0);
}
```

Ordenando objetos

```
// Almacenando Objetos en un Map
// Teniendo en cuenta que map tiene los códigos ordenados, el
// programa define el operador < para poder ordenar según algún atributo.
// Se usará map para crear un directorio telefónico.
#include <iostream>
#include <map>
#include <string>
using namespace std;

// Esta es la clase clave.
class nombre {
  string str;
public:
  nombre() { str = ""; }
  nombre(string s) { str = s; }
  string get() { return str; }
};

// se define el operador < para objetos de la clase nombre.
bool operator<(nombre a, nombre b)
{
   return a.get() < b.get();
} // observar que no pertenece a ninguna clase

// Esto es lo que se almacenará en el map.
class NumeroTelefonico {
  string str;
public:
  NumeroTelefonico() { str = ""; }
  NumeroTelefonico(string s) { str = s; }
  string get() { return str; }
};

int main()
{
  map<nombre, NumeroTelefonico> guia;

 // se ingresa los nombres y números telefónicos
  guia.insert(pair<nombre, NumeroTelefonico>(nombre("Jonhatan"),
                    NumeroTelefonico("555-4533")));
  guia.insert(pair<nombre, NumeroTelefonico>(nombre("Sole"),
                    NumeroTelefonico("555-9678")));
  guia.insert(pair<nombre, NumeroTelefonico>(nombre("Pablo"),
                    NumeroTelefonico("555-8195")));
  guia.insert(pair<nombre, NumeroTelefonico>(nombre("Patricia"),
                    NumeroTelefonico("555-0809")));

  // dado un nombre, encuentra el número
  string str;
  cout << "Ingrese un nombre: ";
  getline(cin,str);
```

```
  map<nombre, NumeroTelefonico>::iterator p;

  p = guia.find(nombre(str));
  if(p != guia.end())
    cout << "Numero telefónico: " <<  p->second.get();
  else
    cout << "El nombre no está en la guía." << endl;

  cin.get();
  return (0);
}
```

Realización de un histograma

Se tiene un archivo de texto "datos.txt" con los números:

79 74 73 84 71 71 73 73 73 84

El programa confecciona un histograma con la siguiente salida:

```
71 : **
73 : ****
74 : *
79 : *
84 : **
```

```
#include <iostream>
#include <fstream>
#include <map>

using namespace std;

int main()
{    map <int,int> histo;
     map <int, int>::iterator p;
     int numero, cantidad;

     ifstream archivo("datos.txt");

     while (archivo>>numero)
        histo [numero]+=1;
        // Observan como este código tiene en cuenta que si no ha sido aun
        // creado el nodo para la clave dada, lo crea automáticamente.

     for(p = histo.begin(); p != histo.end(); p++)
     {
        cout << p->first<< " : ";
        cantidad=p->second;
        for (int k=0; k<cantidad; k++)
           cout << "*";
        cout << endl;
      }
      // observar que los muestra ordenados

      archivo.close();
      cin.get();
      return 0;
}
```

La clase *set*

Operaciones básicas

```
// El programa almacena un conjunto de strings.
// Observar que las string son almacenadas automáticamente
// en orden ascendente. Elimina elementos repetidos.

#include <iostream>
#include <set>
#include <string>
using namespace std;

int main()
{
  set<string> s;

  // crear un conjunto de strings
  s.insert("Telescopio");
  s.insert("Casa");
  s.insert("Computadora");
  s.insert("Ratón");
  s.insert("Casa");

  set<string>::iterator p = s.begin();

  cout << "Contenido (ordenado automáticamente):" << endl;
  do {
    cout << *p << " ";  //muestra el contenido
    p++;
  } while(p != s.end());
  cout << endl;

  // busca un elemento
  p = s.find("Telescopio");
  if(p != s.end())
    cout << endl << "Encontrado Telescopio" << endl;

  cin.get();
  return (0);
}
```

Ejemplo completo (polinomios)

El siguiente ejemplo es el código para manejar polinomios. Se utiliza STL, template y sobrecarga de operadores. Permite además trabajar con coeficientes complejos.

Se muestra el código de la clase Polinomio, luego de la clase Complejo, y por último el código de ejemplo de su uso.

```
#ifndef polinomio_CPP
#define polinomio_CPP

#include <iostream>
#include <iomanip>
#include <fstream>
```

```
#include <string>
#include <map>
#include <algorithm>
using namespace std;

template <class T>
class Polinomio {
    map <int,T> map_term;
 public:
  void LeerDeDisco(string nombre);

  Polinomio <T> operator+ (Polinomio <T> &p);
  template<class U>friend ostream& operator<<(ostream& sal, Polinomio <U> p);
  template<class U>friend istream& operator>>(istream& entrada,Polinomio<U>& p);
};


template <class T>
void Polinomio <T> ::LeerDeDisco(string nombre)
{ ifstream in(nombre.c_str());
   T coeficiente;
   int exponente;

   while (in >> coeficiente >> exponente)
      map_term[exponente] = map_term[exponente] + coeficiente;
   //in >> ws;  // salta espacios en blanco y enters.
};


template <class T>
Polinomio <T> Polinomio <T> ::operator+ (Polinomio <T> &p)
{ Polinomio <T> resultado;

  // primer sumando (*this)
  resultado=(*this);

  // segundo sumando
  typename map<int,T>::iterator it;
   // el "typename" antes del iterador es para que el compilador
   // no piense que es una función estática [SIC]. Ver:
   // http://stackoverflow.com/questions/1123080/why-do-we-need-typename-here

  T coeficiente;
  int exponente;
  for (it=p.map_term.begin(); it!=p.map_term.end(); it++)
  {  coeficiente=(*it).second;
     exponente=(*it).first;

     resultado.map_term[exponente]=resultado.map_term[exponente]+coeficiente;
  }
  return resultado;
};



template <class T>
ostream& operator<< (ostream& sal, Polinomio <T> p)
{ typename map<int, T>::iterator it;
   for (it=p.map_term.begin(); it!=p.map_term.end(); it++)
     sal << setw(5) << setprecision(2)
```

```
          <<(*it).second << " * x^ " << (*it).first << endl;

  return sal;
};


template<class T>
istream& operator>> (istream& entrada, Polinomio<T> & p)
{ T coeficiente;
   int exponente;
   while (entrada >> coeficiente >> exponente)
     p.map_term[exponente] = p.map_term[exponente] + coeficiente;
   return entrada;
}

#endif



// Clase complejos
#ifndef  Complejo_CPP
#define Complejo_CPP

#include <iostream>
using namespace std;

class Complejo {
  private:
    float re, im;

  public:
    Complejo () {re=0; im=0;}
    Complejo (const Complejo &c) {re=c.re; im=c.im;}
    Complejo (float a, float b) {re=a; im=b;}

    Complejo operator+ (Complejo &valor);
    bool operator==(Complejo c);

    friend ostream & operator<< (ostream& sal, Complejo c);
    friend istream & operator>>( istream & entrada, Complejo & c );
};


Complejo Complejo::operator+ (Complejo &valor)
{ Complejo aux;
  aux.re = re + valor.re;
  aux.im = im + valor.im;
  return(aux);
}


bool Complejo::operator==(Complejo c)
{
 bool son_iguales=false;
 if((re == c.re)&&(im == c.im))
   son_iguales=true;

 return son_iguales;
}
```

```
ostream& operator<< (ostream& sal, Complejo c)
{ sal << "(" << c.re;
   if (c.im>0) sal << "+";
   sal << c.im << "i) ";
   return sal;
}


istream & operator>>( istream & entrada, Complejo & c )
{ entrada >> c.re;
   entrada >> c.im;
   return entrada;
}

#endif



#include "polinomio.cpp"
#include "complejo.cpp"
#include <iostream>
#include <fstream>

using namespace std;

int main ( )
{
     cout << "PRUEBA DE COMPLEJOS ==========================" << endl;
     Complejo a (1,1);
     Complejo b (2,2);
     Complejo c;

     cout << a << endl << b << endl;

     c = a+b;
     cout << "c = a+b : " << c << endl;

     c = a+b+c;
     cout << "c = a+b+c : " << c << endl;

     cout << "POLINOMIO DE ENTEROS  ========================" << endl;
     Polinomio <int> polin_1;
     Polinomio <int> polin_2;

     polin_1.LeerDeDisco("polin_1.dat");
     polin_2.LeerDeDisco("polin_2.dat");

     cout << "Polinomio 1: " << endl << polin_1;
     cout << "Polinomio 2: " << endl << polin_2;
     cout << "polin_1 + polin_2: " << endl << polin_1 + polin_2;

     cout << "POLINOMIO DE FLOTANTE  ========================" << endl;
     Polinomio <float> polin_3;
     Polinomio <float> polin_4;

     polin_3.LeerDeDisco("polin_3.dat");
     // otra forma de leer los datos:
     ifstream arch("polin_4.dat");  arch >> polin_4;
```

```
    cout << "Polinomio 3: " << endl << polin_3;
    cout << "Polinomio 4: " << endl << polin_4;
    cout << "polin_3 + polin_4: " << endl << polin_3 + polin_4;

    cout << "POLINOMIO DE COMPLEJOS  ======================" << endl;
    Polinomio <Complejo> polin_5;
    Polinomio <Complejo> polin_6;

    polin_5.LeerDeDisco("polin_5.dat");
    polin_6.LeerDeDisco("polin_6.dat");

    cout << "Polinomio 5: " << endl << polin_5;
    cout << "Polinomio 6: " << endl << polin_6;

    cout << "polin_5 + polin_6: " << endl << polin_5 + polin_6;

    cout << "GRABAR A DISCO UN POLINOMIO: ver archivo salida.dat ====" << endl;
    ofstream s("salida.dat");
    s << polin_1;

    cin.get();
    return 0;
}
```

Algorithm

Ejemplos de códigos
A través de los código de ejemplo que se listan a continuación, puede observarse el uso de varias algorithm útiles. Mayores detalles y otras funciones más, pueden ser consultadas en *http://www.cplusplus.com/reference/algorithm/*

```
#include <iostream>
#include <fstream>
#include <algorithm>
#include <iterator>
#include <vector>
#include <math.h>
#include <numeric>
#include <functional>
using namespace std;

// para usar en el cálculo de la desviación estándar
int promedio;
void mi_funcion (int &i) {i = (i-promedio)*(i-promedio) ;}

// para usar en find_if y en count_if
bool rango (int i) {
  return ( (i>0 && i<=10) ? true : false);
}

// para usar en unique_copy
bool comparacion (int i, int j) {
  return (i==j);
}
```

```
int main () {

   vector <int> v;

   // INGRESO DATOS DEL TECLADO CON: copy
   cout << "Ingrese número enteros (separados por espacio o enters). "
        << "Para salir pulse una letra: " << endl;
   copy(istream_iterator<int>(cin), istream_iterator<int>(), back_inserter(v));

   // SALIDA DATOS POR LA PANTALLA CON: copy
   cout << "Los datos almacenados son: ";
   copy(v.begin(), v.end(), ostream_iterator<int>(cout, " | "));
   // para mostrarlo por renglón, cambiar " | " por "\n"
   // esto no funciona con map, se deberá usar for_each().

   // GRABAR DATOS A UN ARCHIVO CON: copy
   cout << "\nLos datos se graban al archivo datos.txt: " << endl;
   ofstream arch("datos.txt");
   copy(v.begin(), v.end(), ostream_iterator<int>(arch, "\n"));
   arch.close();

   // se elimina todos los valores del vector: erase
   v.erase(v.begin(), v.end());

   // LEER LOS DATOS DESDE UN ARCHIVO CON: copy
   cout << "Se leen los datos desde el archivo datos.txt. ";
   ifstream arch2("datos.txt");
   copy(istream_iterator <int> (arch2), istream_iterator <int> (),
        back_inserter(v));  //istream_iterator<int>(); es para indicar EOF
   arch2.close();

   // SALIDA DATOS POR LA PANTALLA CON:  copy
   cout << "\nValores leidos del archivo datos.txt: ";
   copy(v.begin(), v.end(), ostream_iterator<double>(cout, " | "));

   // MAXIMO Y MINIMO ELEMENTO DE UN VECTOR: max_element y min_element
   cout << "\nValor Mínimo: " << *min_element(v.begin(), v.end());
   cout << "\nValor Máximo: " << *max_element(v.begin(), v.end()) << endl;

   // Encuentra el primer y segundo máximo: max_element
   vector<int>::iterator p;
   p = max_element(v.begin(), v.end());
   p = max_element(p, v.end());
   if(p != v.end())
    cout << "El segundo máximo es: " << *p;

   // ORDENAMIENTO DE LOS DATOS: sort
   sort(v.begin(), v.end());
   cout << "\nValores ordenados: ";
   copy(v.begin(), v.end(), ostream_iterator<int>(cout, "  | "));

   // SUMATORIA DE VALORES: accumulate
   promedio = accumulate(v.begin(), v.end(), 0) / v.size();
   cout << "\nValor Medio: " << promedio << endl;

   // APLICAR UNA FUNCION A CADA ELEMENTO: for_each
   cout << "Desviación estándar: ";
   vector <int> v_aux (v);
   for_each (v_aux.begin(), v_aux.end(), mi_funcion);
   // forma los (Xi-X_promedio)2 para luego hacer la sumatoria
```

```
   cout << accumulate (v_aux.begin(), v_aux.end(),0) / v_aux.size();

   // REALIZAR LA SUMATORIA de PRODUCTOS [(Xi)*(Xi)] : inner_product
   cout << "\nSumatoria[(Vi)*(Vi)] :" << inner_product (v.begin(),
                                        v.end(), v.begin(),0 );

   // En inner_product, las "sumas" y "productos" se pueden redefinir:
   // float mi_sumador ( float i, float j ) { return ( i-j ); }
   // float mi_producto ( float i, float j ) { return ( i+j ); }
   // cout << inner_product (a.begin(), a.end(), b.begin(),0.0,
   //                        mi_sumador,mi_producto );

   // ORDENAMIENTO AL AZAR: random_shuffle
   cout << "\nSe mostrarán dos números al azar del vector: ";
   random_shuffle ( v.begin(), v.end() );
   // reordena los valores en posiciones al azar
   cout << v[0] << " | " << v[1] << endl;

   // ENCONTRAR UN VALOR: find
   cout << "\n¿Está el 7?: ";
   p = find(v.begin(), v.end(), 7);
   if ( p==v.end() ) cout << "No existe." << endl;
      else cout << "Si existe." << endl;

   // ENCONTRAR VALORES CON find_if. Por ejemplo si hay un valor en un rango.
   p = find_if(v.begin(), v.end(), rango);
   if ( p==v.end() ) cout << "No hay un valor en el rango 1 a 10." << endl;
   else cout << "Si hay un valor en el rango 1 a 10. El valor encontrado es: "
             << *p << endl;

   // ENCONTRAR CUANTAS VECES ESTÁ UN VALORES: count_if
   int cantidad = count_if (v.begin(), v.end(), rango);
   cout << "Hay " << cantidad  << " de valores en el rango 1 a 10." << endl;

   // APLICAR UNA FUNCION A CADA ELEMENTO: transform
   cout << "Se almacenará directamente la raiz cuadrada de los valores "
        << "del vector en el archivo de texto datos3.txt." << endl;
   ofstream arch3 ("datos3.txt");
   transform(v.begin(), v.end(), ostream_iterator<double>(arch3, "\n"), sqrt);
   arch3.close();

   cout << "Valores grabados: " << endl;
   ifstream file( "datos3.txt" );  double n;
   for (uint i=0; i<v.size(); i++)
   { file >> n;
      cout << "La raiz de: " << v[i] << " es " << n << endl;
   }

   file.close();
   cout << endl;

   return 0;
}
```

Manejo de errores

Durante el desarrollo de un programa, pueden darse ciertos casos donde no se tenga la certeza de que un fragmento de código vaya a trabajar bien, ya sea porque trata de acceder a recursos que no existen o porque se sale de un rango esperado (por ejemplo división por cero).

Este tipo de situaciones anómalas se incluyen en lo que se consideran excepciones o errores y C++ posee operadores que ayudan a manejar estas situaciones: *try*, *throw* y *catch*. Su forma de uso es la siguiente:

```
try {
  // código que se intentará ejecutar
  throw posible_error;
}
catch (tipo error)
{
  // código a ser ejecutado en caso de error
}
```

El funcionamiento es el siguiente:

- El código dentro del bloque *try* es ejecutado normalmente. En caso de que ocurra un error, este código debe usar *throw* y un parámetro para devolver un error. El tipo de parámetro detalla el error y puede ser de cualquier tipo válido.
- Si un error ocurre, es decir, si se ha ejecutado una instrucción *throw* dentro del bloque *try*, el bloque *catch* es ejecutado recibiendo como parámetro el error pasado por *throw*.

Por ejemplo:

```
// excepción
#include <iostream>
using namespace std;

int main () {
 float f;
 try
 {
  cout << "Raiz cuadrada de: ";
  cin >> f;
  if (f<0) throw 1;

  cout << " es " << sqrt(f);
 }

 catch (int i)
 {  cout<<"Error "<<i<<" en ingreso
                  (valor negativo)" << endl;
 }
 return (0);
}
```

```
Raiz cuadrada de: -5
Error 1 en ingreso
(valor negativo)
```

En este ejemplo, si se ingresa un valor menor a cero *throw* es ejecutado, el bloque *try* finaliza y todos los objetos creados dentro del bloque *try* son destruidos. Después de esto, el control es pasado al bloque *catch* correspondiente (que sólo se ejecuta en caso de error). Finalmente, el programa continúa luego del bloque *catch*.

La sintaxis de *throw* es similar a la de *return*: sólo un parámetro que no necesariamente va entre paréntesis.

El bloque catch debe ir justo después del bloque try, sin incluir ningún tipo de código entre ellos. Los parámetros que catch acepta pueden ser de cualquier tipo válido. Es más, catch puede ser sobrecargado de manera que pueda aceptar diferentes tipos como parámetros. *En ese caso el bloque catch ejecutado es el que concuerda con el tipo de error enviado (el parámetro de throw):*

```
// excepciones: múltiples bloques catch
#include <iostream>
using namespace std;

void raiz(double valor)
{
 try
 {
  cout << "Raiz cuadrada de: " << valor;

  if (valor<0)  throw 1;
  if (valor>10) throw "número muy grande";

  cout << " es " << sqrt(valor) << endl;
 }

 catch (int i)
 {
  cout << " Error " << i
       << " en ingreso (valor negativo)"<< endl;
 }

 catch (char* s)
 {
  cout << "Mensaje: " << s << endl;
 }
}

int main () {

 raiz(2.0);
 raiz(-4.0);
 raiz(100.0);
 return (0);
}
```

```
Raiz cuadrada de: 2 es
1.41421
Raiz cuadrada de: -4 Error
1 en ingreso (valor
negativo)
Raiz cuadrada de: 100
Mensaje: número muy grande
```

En este caso pueden sucederse, al menos, dos diferentes errores:

- Que el valor ingresado sea menor que cero: en este caso un error es devuelto que será registrado por *catch (int i)*, dado que el parámetro es un número entero.

- Que el valor ingresado sea mayor que 10 (supongamos que por la características del programa no es admisible número mayores a 10): en este caso el error devuelto será registrado por *catch (char* s)*.

También se puede definir un bloque catch que capture todos los errores independientemente del tipo que devuelve throw. Para esto se debe escribir *tres puntos* en vez del tipo de parámetro y nombre aceptado por catch:

```
try {
  // código aquí
}
catch (...) {
  cout << "Error";
}
```

Otra posibilidad, es anidar bloques try-catch dentro de bloques try más externos. En estos casos, se tiene la posibilidad de que un bloque catch interno traspase el error recibido al nivel externo, para esto se usa la expresión throw; sin argumentos. Por ejemplo:

```
try {
  try {
      // código aquí
  }
  catch (int n) {
      throw;
  }
}
catch (...) {
  cout << "Error";
}
```

Errores estándares

Algunas funciones de la biblioteca estándar C++ envían errores que pueden ser capturados si se incluyen dentro de un bloque try. Estos errores son enviados con una clase derivada de std::error como tipo. Esta clase (std::error) es definida en el archivo de encabezamiento estándar <error> de C++ y sirve como patrón para la jerarquía estándar de errores.

Debido a que esta es una jerarquía de clases, si se incluye un bloque catch para capturar cualquiera de los errores de la jerarquía usando el argumento por referencia (agregando el ampersand & después del tipo) también se capturará todos sus derivados (reglas de herencia en C++).

El siguiente ejemplo atrapa un error de tipo *bad_typeid* (derivado de error) que es generado al requerir información acerca del tipo apuntado por un puntero nulo:

```
// excepciones estándares

#include <iostream>
#include <exception>
#include <typeinfo>
using namespace std;

class A {virtual f() {}; };

int main () {
  try {
    A * a = NULL;
    typeid (*a);
  }
  catch (std::exception& e)
  {
    cout << "Excepción: " << e.what();
  }
```

```
Excepción: Attempted typeid of NULL
pointer
```

```
  return (0);
}
```

Se puede usar las clases de errores de jerarquía estándar para devolver errores definidos por el usuario o derivar nuevas clases a partir de ellos.

Recursividad

Una función recursiva es una llamada a ella misma, o una llamada a otra función que luego llama a la original. Cada vez que se llama a una función, se reserva espacio para almacenar el conjunto completo de las nuevas variables.

Ejemplo de llamadas recursivas

En Matemáticas se definen funciones o valores a partir de la metodología de cómo se calcula. Por ejemplo, el número **pi** es definido como "la razón de la circunsferencia de un círculo a su diámetro". Esto es equivalente a establecer las siguientes instrucciones: se obtiene la circunferencia de un círculo y el diámetro, se divide el primero por el último y se llama al resultado pi. Este proceso especificado termina con un resultado definido.

Otro ejemplo, es el de la función factorial: "Dado un entero positivo n, el factorial de n se define como el producto de todos los enteros entre n y 1". Por ejemplo, 5 factorial es igual a 5*4*3*2*1 = 120. La definición la podemos escribir de la siguiente forma:

n! = 1 → si n = 0
n! =n*(n-1)*(n-2)*...*1 → si n > 0

A modo de ejemplo, se calculará el factorial de 5. Aplicando la definición anterior:

1) 5! = 5*4! -> para 4! se aplica nuevamente la definición...
2) 4! = 4*3!
3) 3! = 3*2!
4) 2! = 2*1!
5) 1! = 1*0!
6) 0!=1 (por definición)

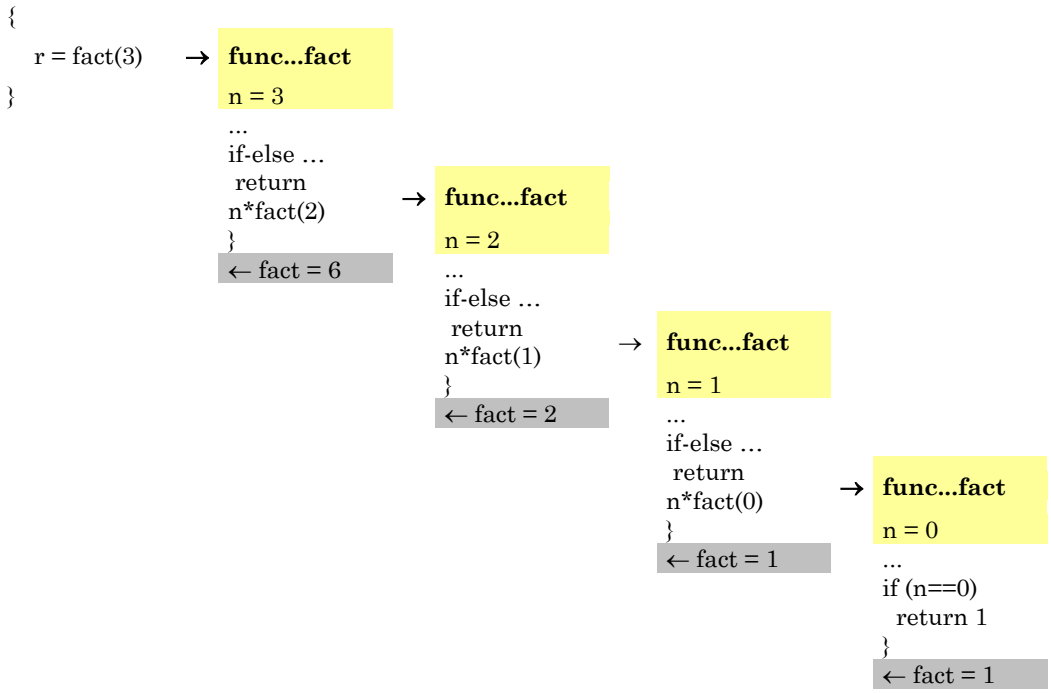
En la línea 6 se llega a un valor que es definido directamente que impone la finalización de las llamadas. "Se vuelve" de la línea 6 a la línea 5, retornando el valor obtenido y así sucesivamente.

El código para realizar este cálculo es el siguiente:

```
// recursividad - factorial
#include <iostream>
using namespace std;

int fact (int n) {
 if (n==0) return (1);
 else return (n*fract(n-1));
}
int main ( ) {
 int r, m;
 cin >> m;
 r = fact(m);
 cout << r;
 return (0);
}
```

```
3
6
```

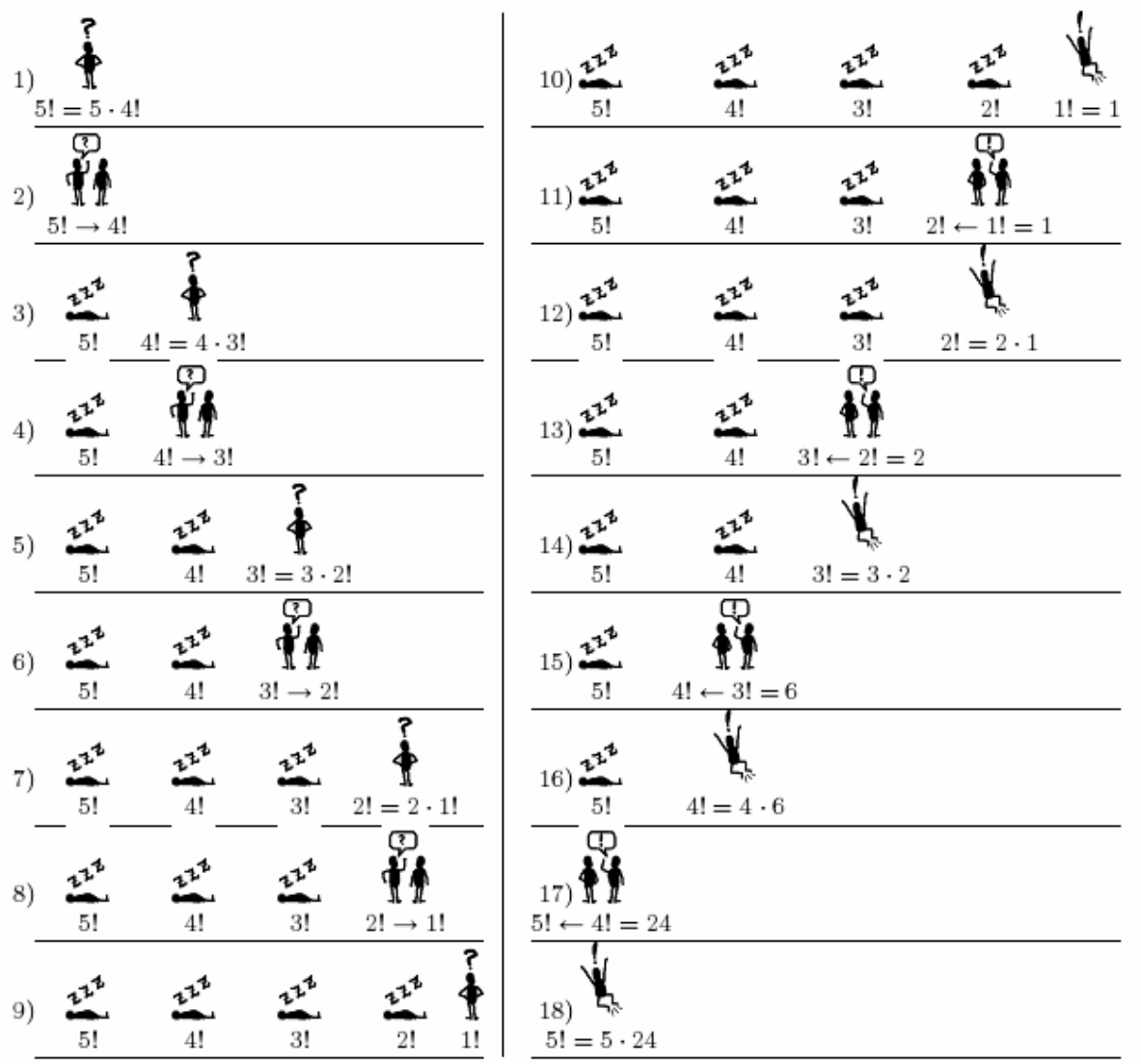
Para el cálculo del factorial de 3, esquemáticamente las llamadas a la función "fact" y el valor de las variables son las siguientes:



Cada vez que una rutina recursiva se llama a sí misma, debe ser asignada un área de datos completamente nueva para esa llamada en particular. Esta área de datos contiene todos los parámetros, variables locales y la dirección de retorno. Un área de datos está asociada no solamente con una función, sino con la llamada en particular a esa función. Cada llamada hace que se asigne una nueva área de datos, y cada referencia a un elemento en el área de datos de la subrutina se hace en el área de datos de la llamada más reciente. Igualmente, cada retorno hace que el área de datos actual sea liberada, y que el área de datos que había sido asignada inmediatamente antes se convierta en la actual. Es muy importante en este caso tener en cuenta cual es la variable **local** (aquí "n") y sus valores asignados en cada llamada de la rutina, aquí en el primer caso un 3 pero en el segundo caso un 2, los que estarán almacenados en "n" para cada una de esas llamadas. Un comportamiento muy distinto tendrá una variable declarada fuera de la función recursiva, que en caso de ser global, en un mismo lugar de memoria tendrá el último valor asignado.

Cómic explicativo del factorial

En la siguiente figura se describe paso a paso lo que ocurre al calcular el factorial de 5 con ayuda de unos muñecos.



En el paso 1, le encargamos a Amadeo que calcule el factorial de 5. El no sabe calcular el factorial de 5, sólo sabe que es 5 por el factorial del número anterior. Por lo tanto necesita que alguien le diga cuanto vale el factorial de 4.

En el paso 2, Amadeo llama a su hermano Benito, y le pide que calcule el factorial de 4 porque el no sabe hacerlo. Mientras Benito intenta resolver el problema, Amadeo se echa a dormir (paso 3).

Benito tampoco sabe resolver directamente factoriales tan complicados (sólo sabe que es 4 por el factorial del número anterior). Benito llama a Ceferino en el paso 4 y le pide que calcule el valor del factorial de 3. Mientras, Benito se echa a dormir (paso 5).

La cosa sigue de la misma manera durante un tiempo: Ceferino llama a David, y David a Eduardo.
Así legamos al paso 9 en el que Amadeo, Benito, Ceferino y David están durmiendo esperando que le traigan la respuesta de lo que pidieron, y Eduardo se pregunta cuánto valdrá el factorial de 1.

En el paso 10 vemos que Eduardo se da cuenta que el factorial de 1 es muy fácil de calcular: vale 1.

En el paso 11 Eduardo despierta a David y le comunica lo que ha averiguado: el factorial de 1! vale 1.

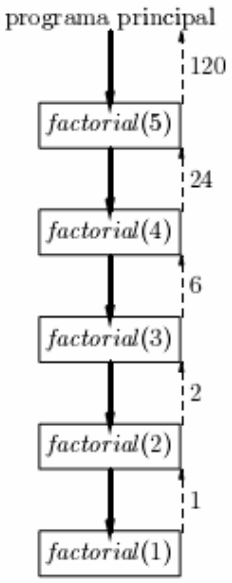
En el paso 12 Eduardo nos ha abandonado: el ya cumplió con su deber. Ahora es David el que resuelve el problema que le habían encargado: 2! se puede calcular multiplicando 2 por lo que valga 1!, y Eduardo le dijo que 1! vale 1.

En el paso 13 David despierta a Ceferino para comunicarle que 2! vale 2. En el paso 14 Ceferino averigua que 3! vale 6, pues resulta de multiplicar 3 por el valor que David le ha comunicado.

Y así sucesivamente hasta llegar al paso 17, momento en el que Benito despierta a Amadeo y le dice que 4! vale 24.

En el paso 18 sólo queda Amadeo y descubre que 5! vale 120, pues es el resultado de multiplicar por 5 el valor de 4!, que según Benito es 24.

Una forma compacta de representar la secuencia de llamadas es mediante el denominado árbol de llamadas. El árbol de llamadas para el cálculo del factorial de 5 se muestra en la siguiente figura. Los nodos del árbol de llamadas se visitan de arriba a abajo (flechas de trazo continuo) y cuando se ha alcanzado el último nodo, de abajo a arriba. Sobre las flechas de trazo discontinuo hemos representado el valor devuelto por cada llamada.



Aunque existen métodos recursivos muy diversos, los ejemplo y descripciones anteriores son una introducción general al comportamiento de estos algoritmos. A continuación se realiza una definición formal de estos método recursivos.

Definición formal de la recursividad

Se dice que un procedimiento de cálculo es recursivo, si en parte está formado por sí mismo o se define en función de sí mismo.

Los algoritmos recursivos son idóneos cuando el problema a realizar, la función a calcular o las variables a procesar están dadas en forma recursiva. En general, un programa recursivo puede expresarse como una composición **P** compuesta de un conjunto de sentencias *S* (que pueden contener a *P*) y *P*. La notación es la siguiente:

P = P[S,P]

Un requisito fundamental es el que las llamadas a todo proceso recursivo están sujetas a una condición *B* que haga terminar las llamadas. Esta condición se expresa en la siguiente notación:

funcion P = if B then P[S,P] else Fin;

Una manera práctica de asegurar la terminación consiste en asociar un parámetro, que al llamar recursivamente al proceso *P* se modifique de tal manera que en algún momento sea falsa la condición *B*.

En las aplicaciones prácticas la profundidad de la recursión debe ser finita, y además pequeña, ya que cada vez que se activa el proceso *P* por una llamada recursiva es necesario disponer de memoria para todas las variables locales. Hay variables con valores "*presentes*" y "*pendientes*" que tienen igual nombre, pero los valores son de momentos distintos, por lo tanto, de lugares distintos de memoria.

Cuando no usar la recursividad

Los algoritmos recursivos son adecuados cuando el problema o los datos están dados en forma recursiva, pero puede darse el caso de que el algoritmo recursivo no sea la mejor manera de resolver el problema. Siempre que un algoritmo pueda ser implementado iterativo, debe hacerse así.

Los problemas en los cuales es adecuado abstenerse de la recursión, presentan la composición siguiente:

Esquemas donde hay sólo una llamada P al final o al inicio de la composición, que se puede representar de la siguiente manera:

función P = if (B) S else P	función P = S; if (B) P

Estos esquemas están asociados con cálculos de valores que están asociados con funciones elementales de recurrencia.

El ejemplo de cálculo del factorial en forma recursiva puede ser realizado en forma iterativa de la siguiente forma:

```
int function fact ( int n )
 {
   int prod=1;

   if (n == 0)
     return (1);   // caso especial para n=0
   else
     while (n!=0)
      {
       prod = n*prod;      // n! = n*(n-1)*(n-2)*..*1 si n>0
       n = n-1;
      }
   return (prod);
 }
```

En general estos procesos deben transcribirse según la forma siguiente:

P = [inicializar; while (B) S]

Algoritmos de rastreo inverso

Un aspecto particularmente interesante es determinar los algoritmos para encontrar la solución de problemas sin seguir una regla específica de cálculo, sino por ensayo y error ("tanteo"). *El patrón común consiste en descomponer el proceso de tanteo en tareas parciales. Con frecuencia éstas se expresan en términos recursivos y consisten en explorar un número finito de subtareas.* Sin embargo, cuando se dan incongruencias con las exigencias del problema, el algoritmo rastrea en forma inversa, suprimiendo la parte más reciente de la solución y ensayando después otra posibilidad hasta llegar al final.

El rastreo inverso es útil en situaciones donde muchas posibilidades pueden aparecer inicialmente, pero que van disminuyendo a medida que se avanza con la estrategia de aplicación de las reglas.

La búsqueda de Juán

Veamos primero un ejemplo sencillo. Nos encontramos en una casa y estamos buscando a Juán. Para encontrarlo realizamos lo siguiente: abrir la puerta de una habitación y comprobamos si está adentro. Ocurre que dentro de esa primera habitación, que es el comedor, no lo hemos encontrado, pero vemos que hay dos puertas más, una hacia el baño y otra hacia la cocina. Decidimos abrir primero la puerta que va a la cocina. Pero en la cocina no encontramos a Juán. Vemos que hay otra puerta más, la que va hacia el patio, y vamos hacia allí, pero no lo encontramos. Como en el patio se termina la casa, volvemos hacia la cocina, y como no hay otra puerta donde buscar volvemos al comedor. En el comedor había una puerta más, la del baño, así que lo buscamos ahí también. Allí lo encontramos.

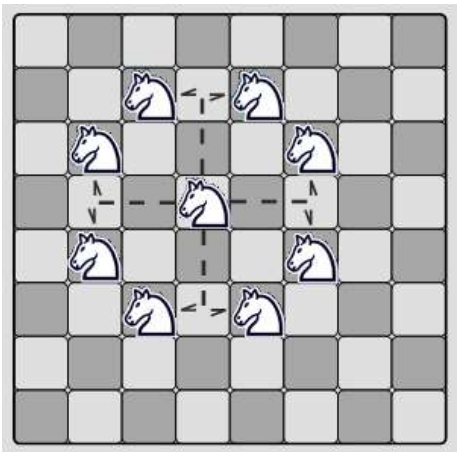
La recursividad se presenta aquí de la siguiente manera: en cada habitación se observa si está Juan, y si no está, se prosigue buscando en una nueva habitación. Pero si allí no está, como llegamos al final de la casa o porque no hay más puertas, volveremos (backtracking) a la habitación anterior, para seguir por otra puerta.

Ejemplo del salto del caballo

Para ejemplificar los conceptos anteriormente comentados, se presenta a continuación la resolución de un problema que consiste en la visita de todas las celdas de un tablero de ajedrez por medio del salto del caballo, el que realiza los movimientos permitidos para esta pieza. Además de visitar cada una de la celdas del tablero de ajedrez, debe pasar solamente una vez por cada casilla.

La simulación deberá comenzar desde un par de coordenadas iniciales cualesquiera. Deberá moverse según las reglas del ajedrez e "intentará" encontrar el itinerario que permita visitar cada una de las celdas sólo una vez.

Los ocho movimientos permitidos del caballo de ajedrez ubicado en una celda del tablero son los siguientes:



Este problema tiene un solución sencilla por medio de la recursividad. Pero para que sea más fácil de entender, plantearemos cada una de las ideas usadas, y construiremos la solución final.

Para empezar, el tablero es muy fácil de representar por medio de una matriz cuadrada. Con una matriz de 8x8 podríamos representar un tablero de ajedrez. También podemos definir que cuando esté desocupada una casilla de la matriz tenga el valor de 0, y para cualquier otro valor, quiera decir que la casilla esté ocupada.

Recordando la forma de solucionar problemas recursivamente, sabemos que debemos tener un caso base (que es el caso más sencillo que se puede tener, y que tiene una solución explícita) y además de llamadas recursivas, que solucionen el problema con una representación más pequeña del problema (como anteriormente se vió con la solución del factorial).

Primero definiremos el caso base, y este sería muy sencillo, ya que el caso base sería cuando ya no se pueda mover más el caballo. Y esto puede ser en 2 casos, cuando no se pueda mover más (todos los caminos estén tapados), en este caso no habría solución, y el más importante cuando se haya llegado al movimiento número 64, que es cuando se ha cubierto todo el tablero. Y para cada movimiento posible del caballo, tendríamos que probar si existe o no solución (esto se puede hacer usando llamadas recursivas).

En general la solución del problema quedaría de la siguiente forma:

```
1  Caballo (mov, y, x) //Definicion de la solucion recursiva
2   Si (mov=64)
3     Hay solucion, e imprimirla. [FIN]
4   Sino
5    Probar para cada movimiento del caballo (8)
6         Caballo (mov+1, ny,nx)
7         //Donde ny y nx, es la nueva posición del caballo
```

Como se puede ver esta solución general, tiene un caso base (mov=64) porque ya no se puede mover más (línea 3) y es el fin de la búsqueda. Y un caso recursivo, que consiste en llamadas a "Caballo" con otras coordenadas dentro del ciclo enunciado como "Probar" de la línea 5, y además se tiene el primer parámetro mov incrementado en 1, ya que cuando se suma 1 al movimiento, se reduce la cantidad de movimientos que restan por hacer para llegar hasta 64 (por lo tanto es una buena solución recursiva).

Ahora podremos detallar aún más el problema. Primero para saber si ya pasamos por una casilla deberemos marcar cada casilla por donde pasemos, para esto usaremos para marcar el número de

movimiento (mov), ya que además, este nos dará la oportunidad de saber como se construyó la solución (fácilmente yendo del 1 al 2, del 2 al 3, etc). Además, hay que saber si se puede mover el caballo o no. Y esto se debe de verificar antes de hacer la llamada recursiva. También hay que calcular la nueva posición del caballo cada vez que se mueva.

El problema quedaría más detallado así:

```
Caballo (mov, y, x) //Definicion de la solucion recursiva
  tablero(y,x)=mov  // Marcamos la posición del tablero
  Si (mov=64)
   Hay solucion, e imprimirla. [FIN]
  Sino
   Probar para cada movimiento del caballo (8)
    Calcular la nueva posición (ny,nx)
    Si la nueva posicion (ny,nx) es valida
     Caballo (mov+1, ny,nx)
     // Donde ny y nx, es la nueva posición del caballo
```

Ahora ya quedó un poco más detallado el problema, pero todavía falta solucionar algunos detalles. El primero, es que la solución debe regresar un valor, verdadero si es que encontró solución, y falso si es que no encontró. Además debería desmarcar la casilla si es que no se encontró solución en ese "tiro", porque sino quedaría tachada, y si hay otro camino por el cual solucionar el problema, este jamás podría ocupar esa casilla. Para solucionar estos 2 problemas, se propone lo siguiente:

```
1   Caballo (mov,y,x)     //Definición de la solución recursiva
2     tablero(y,x)=mov    // Marcamos la posición del tablero
3     Si (mov=64)
4      Hay solución (solución=true) e imprimirla. [FIN]
5      regresar verdadero.
6     Sino
7      solucion = falso
8      Probar mientras no exista solución para alguno de los 8 movimientos
9       Calcular la nueva posición (ny,nx)
10      Si la nueva posicion (ny,nx) es válida
11       solucion = Caballo (mov+1, ny,nx)
12       //La parte anterior guarda si hubo o no solución
13
14     Si no hubo solución porque intentó con los 8 movimientos
15       tablero(y,x) = 0   // Desmarcamos
16
17     regresar solucion
```

En la forma anterior, estamos especificando que si en alguno de los caminos (8 posibles movimientos) encuentra una solución, si encuentra la solución por uno de esos caminos regresará verdadero, sino, regresará falso.

Este último esbozo de solución sería nuestro pseudocódigo final para solucionar el problema. Ahora empezaremos a construir un código en C, que solucione el mismo problema.

Primero definimos el prototipo de la función.

```
bool caballo (int mov, int y, int x, int tab[8][8]);
```

Donde mov, sería el número de movimiento, x e y la posición en el tablero, y la matriz tab (de 8x8) sería el tablero.

La primera parte del pseudocódigo es muy simple de codificar, solamente tenemos que hacer lo siguiente:

```
int caballo (int mov, int y, int x, int tab[8][8])
{
  int solucion;
  tab[y][x] = mov;

  if (mov == 64)
  {
   Imprimir (tab);  // Funcion que imprima la solucion
   return 1;  // 1 verdadero
  }

  else {
   solucion = 0;  // 0 Falso
   se harán los intentos...
  }
}
```

Ahora nos encontramos con otro problema, como especificar los movimientos del caballo. Para solucionar esto, lo más simple es usar una pequeña matriz, que nos diga que movimiento vamos a hacer. La nueva posición desde una posición dada, será sumando o restando un número determinado de filas y columnas dependiendo del Movimiento que se quiera dar al caballo (según la figura anterior).

Por ejemplo para calcular la posición del movimiento 0, tenemos que restar 2 posiciones en X y 1 en Y a la posición actual del caballo. Para el movimiento 1, restamos 1 en X y 2 en Y... y así sucesivamente.

Entonces podemos definir la matriz de movimientos relativos de la siguiente forma (teniendo en cuenta que el movimiento se enumera a partir del 0, para adaptarlo a la forma en que se usa la numeración en una matriz):

	0	1	2	3	4	5	6	7
0(X)	-2	-1	+1	+2	+2	+1	-1	-2
1(Y)	-1	-2	-2	-1	+1	+2	+2	+1

Para definir esta tabla en código en C, simplemente sería de la siguiente forma

```
int salto[2][8] = {{-2,-1,+1,+2,+2,+1,-1,-2},
```

```
                  {-1,-2,-2,-1,+1,+2,+2,+1}};
// Donde salto[0], contiene las 8 sumas o restas en X
// salto[1], contiene las 8 sumas o restas en Y
```

Por ejemplo, el siguiente código quiere decir lo siguiente:

```
nx = x + salto[0][2];
ny = y + salto[1][2];
```

Si el caballo está en la coordenada [3,3], la nueva coordenadas serán:

```
nx = 3 + (+1) = 4
ny = 3 + (-2) = 1
```

Otra cosa que hay que especificar es cuando una posición del tablero es válida o no. Y la solución es muy simple. Si (0<= x < 8) y (0<=y<8), es decir que x esté dentro del tablero e y también, tomando en cuenta que la numeración en un arreglo de 8 elemento va de 0 a 7). Y además debe cumplir que tablero(y,x) sea 0. Esto fácilmente se puede expresar en un lenguaje de programación, como una condición lógica.

```
1   bool caballo (int mov, int y, int x, int tab[8][8])
2   {
3     // Se declaran las variables a usar
4     int salto[2][8] = {{-2,-1,+1,+2,+2,+1,-1,-2},
5                        {-1,-2,-2,-1,+1,+2,+2,+1}};
6     bool solucion;
7
8     int i;
9     int nx,ny;
10    tab [y][x] = mov;
11
12    if (mov == 64)
13    {
14      imprimir(tab); /// Función que imprima la solución
15      return true;
16    }
17    else {
18
19     solucion = false;
20     i = 0;    // Cuenta el movimiento en el que va.
21     while ( (!solucion) && (i<8) )  //Mientras no exista solución...
22     {
23        nx = x + salto[0][i];
24        ny = y + salto[1][i];
25        if ( (nx>=0) && (nx<8) && (ny>=0) && (ny<8) && (tab[ny][nx]==0) )
26        {
27          solucion = caballo (mov+1, ny, nx, tab);
28        }
29        i++;
30     }
31
32     if (!solucion)
33      tab[y][x] = 0;
34     return solucion;
35     }  // fin del else
36   }
```

Un ejemplo de una función principal para solucionar el problema sería el siguiente.

```
int main()
{
   int tab[8][8] = {0};
   //tablero(tab)se inicializo a 0
   caballo(1,0,0,tab);
   // Se llamó a caballo con
   // mov=1, y = 0, x = 0, y el tablero(tab)
}
```

Con esto se tiene solucionado el problema del recorrido del caballo. Aunque hay que agregar varias notas extras.

- Es importante inicializar cada celda de la matriz que representa el tablero a 0, antes de aplicar la función que soluciona el problema.

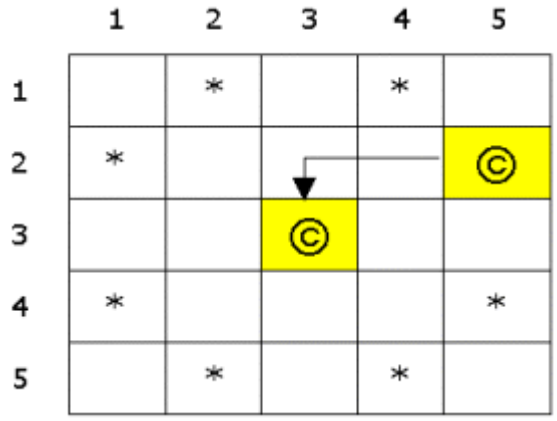
- La variable (matriz) "salto", no es conveniente que se declare dentro de la función recursiva porque sería un gasto extra de memoria, ya que de cada variable local (usada dentro de la función) se crea una copia por cada entrada recursiva. Sería preferible declararla como una variable global.

- Un error muy común, es llamar a la función desde el programa principal con un valor de mov = 0. Es conveniente que la llamada inicial se haga con un valor de mov=1 (como se mostró en el código anterior).

- Si al ejecutar este programa, se tardara demasiado y no terminara, hay que tomar en cuenta el espacio de búsqueda tan grande que hay que explorar para llegar a la solución. Por cada entrada recursiva, en el peor de los casos se pueden generar 8 llamadas, y teniendo en cuenta que el nivel de recursividad es de 64. Esto sería 8 elevado a la potencia 64. Lo cuál es un número mucho muy grande de llamadas recursivas. Para hacer las pruebas se puede buscar iniciar desde otra posición, en lugar de y=0 y x=0, por ejemplo hacerlo desde y=2 y x=3.

- Este tipo de solución por medio de recursividad, está usando una técnica llamada backtracking, que consiste en generar todas las posibilidades de un problema hasta encontrar la solución.

- Las llamadas recursivas quedan pendientes con sus variables locales. Cuando se ejecuta el fin "**}**" del método (línea 36), comienza a ejecutarse el pendiente anterior con sus variables locales, lo que significa que "retrocede" (backtracking) a la fila y columna anterior, en el lugar que quedó y continúa ejecutándose la línea siguiente de la que se realizó la llamada.



A continuación se comentará algunos ciclos de la ejecución del programa para observar como se realiza el backtracking.

Suponiendo que luego de varios saltos llegó a 3,3 a partir de 2,5. Al probar un próximo salto e intentar los 8 movimientos posibles debido a que resultaba siempre un salto fuera del tablero o `tab[ny][nx]!=0` (porque para entrar al if de la línea 25, aparte de dar un salto a adentro del tablero, debe ser también `tab[ny][nx]==0`) sale del bucle del while de la línea 21, el que llega hasta el corchete de la línea 30, debido a que el i++ de la línea 29 que intenta los 8 saltos ya llegó al valor 8 (recordar que los saltos están numerados desde 0 hasta 7).

Por lo tanto continúa en la línea 32 que tiene el código `if (!solucion)` poniendo en cero a la posición (3,3) porque en la línea 10 le había puesto anteriormente el valor del número del salto con el cual había llegado. Cuando llega al corchete de la línea 35, esa llamada en particular de la funcion se da por terminada lo que implica que vuelva a lo que tenía pendiente, es decir al ";" final de la línea 27 con las variables del procedimiento anterior, es decir, con el ny y nx que tenía en ese momento (casillero (2,5)) y el valor del contador de salto i que también tenía en ese momento, que al hacer en la línea 29 i++ intenta hacer otro salto distinto, porque por el camino (o salto) que hizo anteriormente no le fue satisfactorio. El while de la línea 21 se vuelve a ejecutar y así seguir con ese nuevo "camino" o salto a seguir.

En este ejemplo es muy importante analizar las variables que son **locales** y las que son **globales**.

Creación de bibliotecas

Uno de los conceptos más potentes que existen en los lenguajes modernos de programación es la reutilización de código.

Esto significa usar el mismo código en diferentes partes de un programa. Esto puede ser realizado con macros y también con múltiples módulos que separen el código.

Las bibliotecas, que los sistemas operativos como Linux y Windows permiten usar, son formas mucho más potentes que las anteriormente dichas.

Las bibliotecas permiten compartir código (en general ya compilado) entre programas que no están escritos en ellos. Considere, por ejemplo, la función "sumaPolinomios()". Esta función pude ser usada en miles de programas de sus sistema. En vez de tener una copia separada en cada uno de los programas que la usan, todos ellos pueden llamar directamente a la biblioteca que contiene el único código existente.

Introducción

Las bibliotecas pueden ser de dos tipos: estáticas y dinámicas. La diferencia está dada por el efecto producido cuando se unen las bibliotecas con el programa que las usa. Este enlace (link) generará el programa ejecutable final, tomando código de todos los módulos y uniéndolos en el programa ejecutable final.

Mediante bibliotecas estáticas, este proceso se realiza en el momento de la compilación del programa. El código de las bibliotecas es enlazado dentro del programa ejecutable y siempre lo acompaña. Esto significa que si la biblioteca es cambiada (por ejemplo por una nueva versión) deberá recompilarse el programa que la utiliza.

Trabajar con bibliotecas dinámicas significa que el enlace queda "preparado" en el momento de la compilación, pero realmente se realiza en el momento de la ejecución, por lo tanto el programa ejecutable final necesitará de la presencia de la biblioteca para que pueda funcionar. Esto significa que si la biblioteca es cambiada (por ejemplo, por mejoras) no es necesario recompilar el programa, sino que la actualización será automática en la nueva ejecución del programa.

Otra diferencia es que con las bibliotecas estáticas, si hay más de un programa que ejecuta la misma biblioteca, cada uno de ellos contendrá internamente a la biblioteca. Con las dinámicas hay ahorro de memoria porque sólo existe una biblioteca en memoria que es compartida por todos los programas que la utilizan.

Otra característica de las bibliotecas dinámicas es la capacidad de sobreescribir (o sobreutilizar) la característica de cualquier biblioteca que esté usando. Por ejemplo, se podría agregar nuevas características a funciones que ofrece el compilador. Esto se realiza precargando la nueva biblioteca antes que la original. De esta forma se podría reemplazar bibliotecas completamente.

Un problema del uso de las bibliotecas dinámicas, surge en los casos en que una nueva version introduce incompatibilidades con la versión previa. En el caso de las estáticas, el programador crea una hermandad (biblioteca más su código) que permanecera.

Respecto a la velocidad del proceso en el uso de bibliotecas estáticas respecto a dinámicas, para cada caso en particular deberá realizarse pruebas de desempeño debido a que las optimizaciones de los compiladores y sistemas operativos modernos pueden crear ejecuciones con resultados diferentes a los que intuitivamente parecería que podría ser respecto a los de desempeños (eficiencia en tiempo).

Creación y uso de bibliotecas estáticas

La creación de bibliotecas estáticas es muy simple. Esencialmente, se debe usar el programa "*ar*" (que generalmente se provee junto con el compilador) para combinar un numero de archivos ".o" en una única biblioteca. Luego se necesita ejecutar el programa "*ranlib*" que agrega cierta información a la biblioteca.

El ejemplo que se muestra a continuación es muy simple. Se realiza una biblioteca para sumar y restar números enteros mediante el siguiente código:

```
// === Archivo "bibli.h" =============

int Sumar(int a, int b);
int Restar(int a, int b);

=====================================


// === Archivo "bibli.cpp" ===========

int Sumar(int a, int b)
{ int resultado;

  resultado=a+b;
  return (resultado);
}

int Restar(int a, int b)
{ int resultado;

  resultado=a-b;
  return (resultado);
}

=====================================


// === Archivo "makefile" ============

## este makefile es para generar bibliotecas estaticas. Archivo ".a"

NOMBRE=libbibli
CXX=g++

# linux
OPCIONES= -g -Wall
#DIRECT= -I../util -I/bin
#BIBLIOT=-L/lib -lm -L/usr/lib/

# cygwin
#OPCIONES= -g -Wall
#DIRECT=-I../util -I/bin -I/usr/include
#BIBLIOT= -L/lib -lm

all:
        ${CXX} -c ${OPCIONES} ${DIRECT} bibli.cpp -o bibli.o ${BIBLIOT}
        ar cr ${NOMBRE}.a bibli.o
        ranlib ${NOMBRE}.a
=====================================================
```

En el archivo correspondiente al makefile, observar el uso del parÁmetro "**-c**" y el uso del programa "ar" y "ranlib".

Por convención los nombres de las bibliotecas deber ir precedidas por el prefijo "lib" y el sufijo ".a" para las estáticas.

El uso de make para ejecutar los comandos del makefile, generara la biblioteca "libbibli.a".

El ejemplo siguiente usa la biblioteca creada mediante el agregado de la linea:

```
#include "bibli.h"
```

y la existencia del archivo "libbibli.a" que estará en el mismo directorio del programa en el momento de la compilación o encontrará su ubicación mediante la nomenclatura usada en el makefile para encontrar las otras bibliotecas (por ejemplo, colocándolas en la carpeta que contine las bibliotecas del usuario: -L/usr/lib/)

```
// Ejemplo de programa de uso de la biblioteca "bibli.h"

#include <iostream>
#include "bibli.h"

int main() {

        int a,b;
        cout << "Ingresar dos numero enteros: ";
        cin >> a >> b;

        cout << "Suma:  "<< Sumar(a,b) << endl;
        cout << "Resta: "<< Restar(a,b) << endl;

        cin.get();
}
```

Para la compilacion del programa, un ejemplo de archivo makefile es el siguiente:

```
MAIN=main
CXX=g++

# linux
OPCIONES=-g -Wall
DIRECT=-I../util -I/bin
BIBLIOTECAS=-L/lib -lm -L/usr/lib/ -L. -lbibli

all:
        ${CXX} ${OPCIONES} ${DIRECT} ${MAIN}.cpp -o ${MAIN} ${BIBLIOTECAS}
```

Así puede usar la biblioteca. Se puede poner el ".a" en /usr/local/lib. También, puede simplemente colocarse en el directorio corriente. La opción "–L." le dice al linker que mire en el directorio actual, indicado por el punto. Normalmente, mira en el directorio de las bibliotecas solamente. -lbibli esta invocando a la biblioteca "libbibli.a".

Graficación con OpenGL

OpenGL es un programa de interfaz con el hardware gráfico de la computadora. Está diseñado para trabajar eficientemente, y el código desarrollado por el usuario puede ser implementado en diferentes plataformas de hardware.

En este capítulo se comentan y ejemplifican conceptos y comandos básicos, centrándose principalmente en ejemplos de códigos reusables, que sirvan como base para desarrollos más complejos o de aplicación a otros casos que el del ejemplo implementado. Se recomienda copiarlos a un editor, compilarlos y observar su comportamiento.

Para un conocimiento más completo de la biblioteca y la teoría involucrada, se sugiere consultar los manuales Open GL Red Book y el Open GL Blue Book, ambos de Silicon Graphics, como así también la extensa documentación, ejempos y foros de discusión existentes en Internet, por ejemplo, http://www.fltk.org/doc-1.3/opengl.html.

OpenGL, GLUT y FLTK

OpenGL es una biblioteca para programación gráfica, realizada para que funcione en una gran variedad de máquinas. Las rutinas para graficación fueron realizadas independientemente de un sistema operativo en particular en el que finalmente sobre una ventana se representen los dibujos. Para poder mostrar los resultados de OpenGL en una ventana gráfica, se hace uso de otras bibiliotecas específicas para los sistemas operativos, estando entre las más utilizadas en el lenguje C, la GLUT y FLTK.

Por ejemplo, se puede realizar un cubo tridimensional, con caras que no serán transparentes y de colores distintos. Si luego es rotado en los tres ejes, lo que un observador verá desde una posición determinada es un cálculo complejo. Éstos cálculos los realiza OpenGL. Luego de resuelto, es tarea de GLUT o FLTK interactuar con el sistema operativo para mostrarlo en una ventana del sistema operativo en particular en que corra la aplicación.

GLUT es una herramienta desarollada para ayudar a construir aplicaciones de OpenGL. Incluye la interconexión con el sistema operativo que permite la visualización de los dibujos en una ventana, captar eventos del mouse o del teclado. También provee un limitado paquete de interfaces gráficas, como ser botones, cajas de textos, etc. Es utilizado en muchos cursos de graficación por su facilidad de uso.

FLTK es una biblioteca más extensa que GLUT y está codificada en clases, que permite la ventaja de la Programación Orientada a Objetos en forma nativa. Además, el paquete de interfaces gráficas es más amplio. Por lo tanto, será la biblioteca empleada en este tutorial.

La forma más simple para utilizar OpenGL y mostrar por pantalla con FLTK, es realizar una subclase de Fl_Gl_Window. En esta subclass se debe implementar el método draw() que utilizará OpenGL y FLTK. El programa podría necesitar también del método redraw() en el caso de que la imágen no sea estática, o por ejemplo, deba cambiar porque el tamaño de la ventana se modificó quedando más chica, más grande u otras proporciones de las caras de la ventana.

También se puede ver en algunos códigos el uso de las funciones gl_start() y gl_finish() antes y después de los códigos en OpenGL. Esta forma es utilizada más frecuentemente con los componentes de FLTK, pero como este es un capítulo de OpenGL, no se utilizará esta forma. Además, existen otras funciones de FLTK que no serán cubiertas por este tutorial, como por ejemplo clipping, que trabajan dentro de una clase heredada de Fl_Gl_Window, es decir, todos los dibujos y transformaciones deben ser realizados exclusivamente con llamados a funciones de OpenGL.

Includes

En los códigos, se debe incluir los siguientes archivos:

```
#include <FL/Fl.H>
#include <FL/Fl_Gl_Window.H>
#include <FL/gl.h>
```

<FL/gl.h> incluirá las bibliotecas de OpenGL a través de <GL/gl.h>. Para el caso de compilar en el sistema operativo Windows, también será incluida la biblioteca <windows.h> por ejemplo, y así se incluirán a partir de los códigos anteriores, el resto de las bibliotecas necesarias.

Clase

Para trabajar orientado a objetos y además con códigos nativos de OpenGL, se debe realizar una subclase como se esquematiza a continuación:

```
class Ventana: public Fl_Gl_Window
{   void draw();
    int handle(int e);
    … etc.
    … métodos propios de la aplicación.
 }
```

El método handle() sólo es necesario si la aplicación debe recibir eventos del usuario (clicks del mouse o ingresos del teclado).

El método draw()

Este método se llamará cada vez que la ventana deba ser dibujada o redibujada en la pantalla. En este método es donde se deben realizar los dibujos con los comandos de OpenGL (no realizar llamados a funciones de <FL/fl_draw.H> o glX directamente).

En términos generales, el método draw() será:

```
void Ventana::draw( )
{
   if ( !valid( ) )
   {
       ... comandos de la proyección, viewport, etc ...
       ... el tamaño de la ventana es w( ) y h( ).
       ... valid( ) es puesto en "true" luego de terminado draw( )
   }

   glPushMatrix( );
   ... dibujo ...
   glPopMatrix( );

}
```

Los comandos de la proyección, viewport, etc. serán explicados más adelante.

Primer ejemplo de código

El siguiente código es un programa en C++ que ejemplifica el uso de los comandos anteriores para lograr la aparición en pantalla de una simple ventana, así luego, se puede escribir el código para los dibujos que se quiere realizar a través de la biblioteca OpenGL.

```
#include <FL/Fl.H>
#include <FL/Fl_Gl_Window.H>
#include <FL/gl.h>

using namespace std;

class Ventana: public Fl_Gl_Window
{   void draw();

    public:
      Ventana(int x, int y, int w, int h, const char* titulo);
};


Ventana::Ventana(int x, int y, int w, int h, const char* titulo):
    Fl_Gl_Window(x, y, w, h, titulo)
{
   // Aquí se puede cambiar algunos parámetros por defecto. Por ejemplo,
   // simple o doble buffer, color para borrar la pantalla,
   // paleta de colores (por defecto RGB), etc.
   // Además se puede realizar inicializaciones de la aplicación.
}


void Ventana::draw()
{
    // glClear(GL_COLOR_BUFFER_BIT);
    // aquí va el dibujo...
}

int main(int argc, char **argv)
{
    // Se crea un nuevo objeto "ventana" de la clase "Ventana",
    // que es hija de la clase Fl_Gl_Window.
    // Se definen las coordenadas de la posición, el alto y ancho.
    // El último parámetro es el texto que estará en la cabecera de la ventana.
    Ventana ventana (100, 100, 300, 300, "Primer Ejemplo");

    // El llamado al siguiente método, heredado de la clase Fl_Gl_Window,
    // muestra la ventana en la pantalla.
```

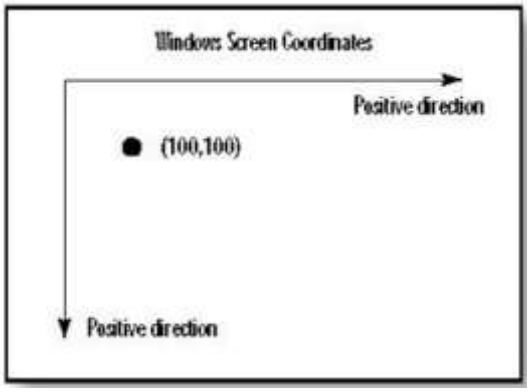
```
    ventana.show();

    // Todo programa de FLTK debe contener el siguiente método que inicia
    // los llamados a dibujar, espera eventos del usuario, etc.
    // Cuando la ventana es cerrada, termina la función Fl::run().
    return Fl::run();
}
```

Makefile

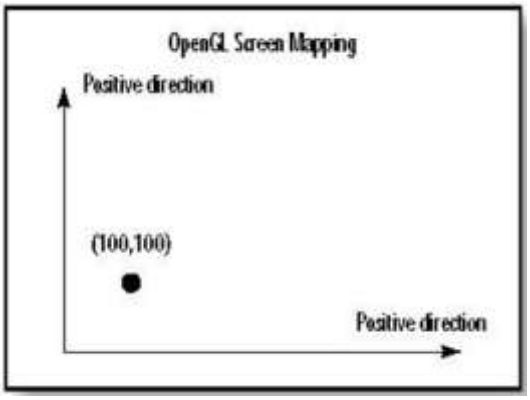
```
g++ -Wall -o main main.cpp -lfltk -lfltk_gl -lGL
```

Coordenadas

Respecto al sentido de los ejes coordenados, tanto para la ventana que administra el sistema operativo, como el interno de OpenGL, tener en cuenta los dos siguientes esquemas:



Coordenadas por defecto en Windows



Coordenadas por defecto en OpenGL y en X Window

Nota de Portabilidad: en X Window el sistema de coordenadas es el mismo que el de OpenGL. Hay que tener en cuenta esta características para llevar un programa de un sistema operativo a otro.

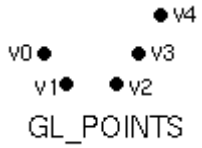
No obstante, se puede en OpenGL invertir el eje *y* de manera que hacia abajo es positivo mediante el comando *glScalef(1, -1, 1)*.

Primitivas de Dibujo

Las primitivas geométricas son usadas para realizar los dibujos. Las primitivas son: *puntos*, *líneas* y *polígonos*. Estas primitivas se describen básicamente a partir de sus *vértices* que pueden ser las coordenadas que definen al punto como tal, los extremos de los segmentos de línea o las esquinas de los polígonos.

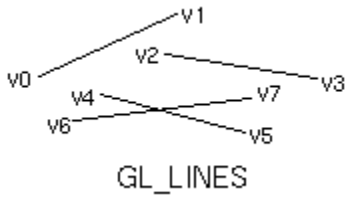
OpenGL se refiere a los puntos como un vector, manejado como un número de punto flotante, pero ello no implica que la precisión sea de un 100% en el momento de dibujar, debido a las limitaciones del *raster graphics display*, cuya unidad mínima (píxel) es mucho mayor del concepto de matemático de infinitamente pequeño (para un punto) o infinitamente delgado (para una línea).

Puntos



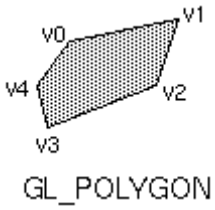
Se trata de números de punto flotante llamados *vertex*. Todos los cálculos que se involucran se realizan con vectores tridimensionales, y si no se define la coordenada Z (como en un dibujo de 2D), se asume un valor de cero como valor de su "tercera" dimensión.

Líneas



Las líneas en OpenGL son en realidad segmentos de recta, y no el concepto de una extensión al infinito como ocurre en matemáticas. Los vértices definen los puntos extremos de cada segmento.

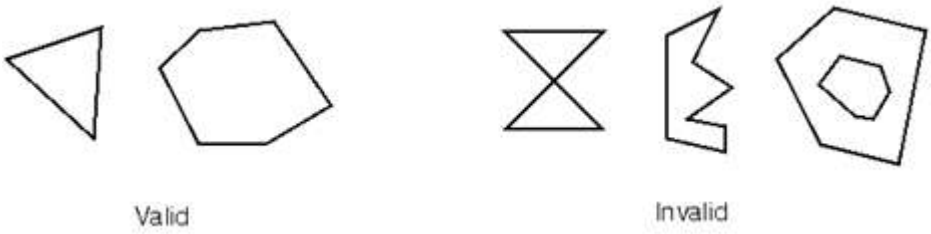
Polígonos



Son las áreas encerradas por un lazo cerrado de segmentos de recta, con los vértices indicando los extremos de los segmentos que le forman. Por omisión, los polígonos se dibujan rellenos al interior. Hay funciones que permiten tener polígonos "huecos", o definidos sólo como vértices en sus esquinas, colores distintos para una cara que para la otra, etc.

Es importante comprender que debido a la complejidad posible de los polígonos, es necesario definir restricciones:

- Las orillas no pueden intersectarse.
- Sólo es válido tener polígonos convexos, es decir, polígonos que cumplen la norma de regiones convexas (una región es convexa si dados dos puntos de su interior, todo el segmento que los une está también en el interior).



El no respetar las restricciones no garantiza que el dibujo resulte como se espera. La restricción se debe a que es más sencillo tener hardware rápido para polígonos simples.

Debido a que se trata de vértices 3D, es necesario tener cuidado ya que después de algunas operaciones, si algún vértice no está en el mismo plano, puede llegarse a resultados inesperados.

Construir imágenes basado en muchos triángulos (en vez de polígonos) garantiza que siempre se trabaja en un mismo plano.

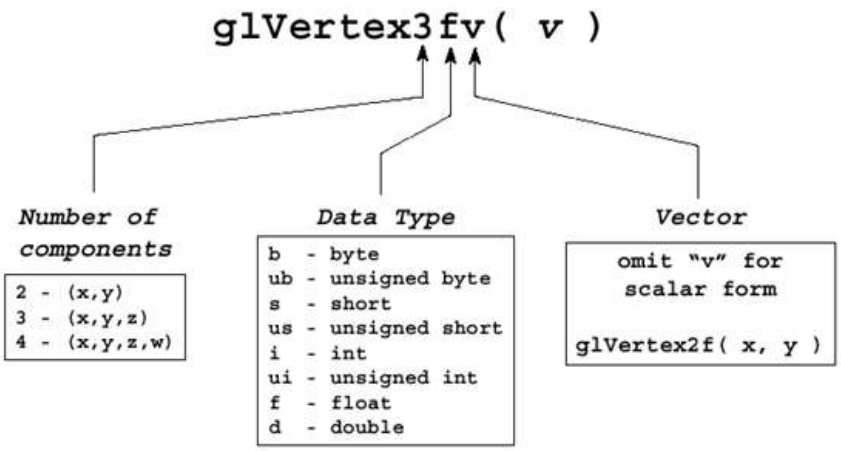
Definición de los vértices

```
glVertex{1234}{bsifd}[v](TYPE coords)
```

Donde la cantidad de coordenadas posibles son 1, 2, 3 ó 4, y los tipos de datos de las coordenadas es dado por los sufijos siguientes:

Sufijo	Tipo de datos	Tipo de Datos en C	Definición de tipo OpenGL
b	Entero 8 bits	Signed char	GLbyte
s	Entero 16 bits	Short	GLshort
i	Entero 32 bits	Long	GLint, GLsizei
f	Real 32 bits	Float	GLfloat, GLclampf
d	Real 64 bits	Double	GLdouble, GLclampd
ub	Entero sin signo 8 bits	unsigned char	GLubyte,GLboolean
us	Entero sin signo 16 bits	unsigned short	GLushort
ui	Entero sin signo 32 bits	unsigned long	GLuint,GLenum,GLbitfield

Tabla de Tipos de Datos de OpenGL



Ejemplos:

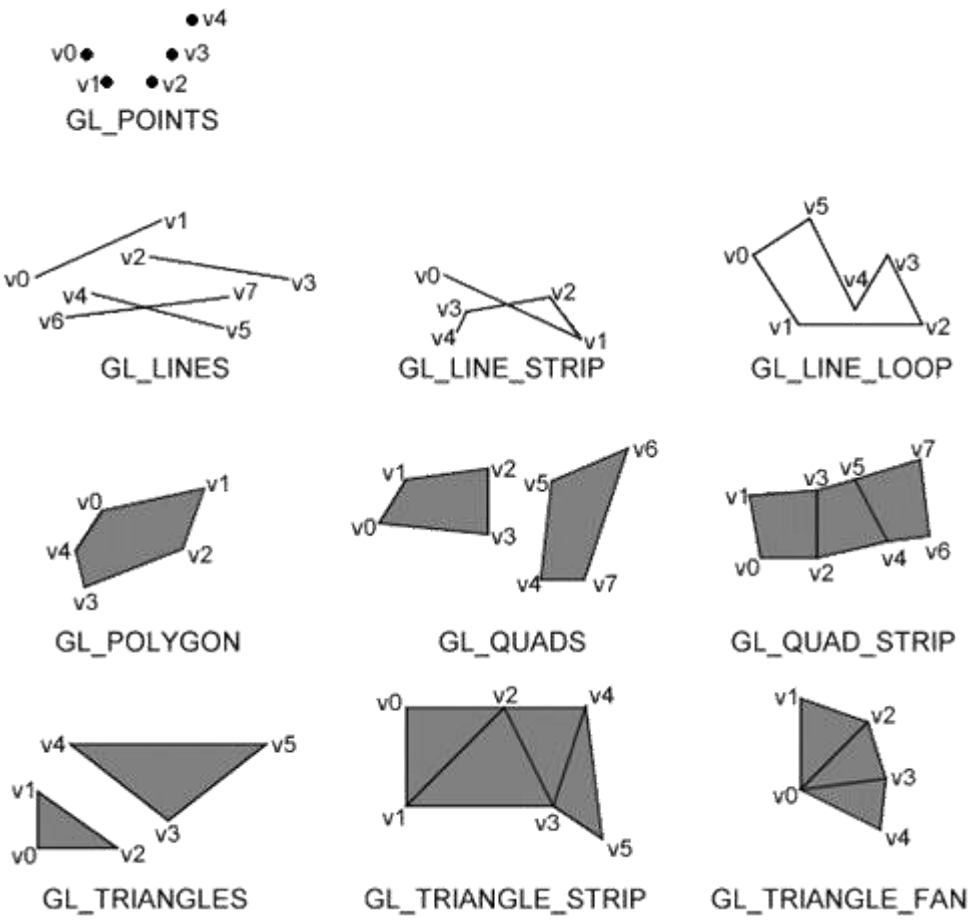
```
glVertex2s (2, 3);

GLshort a=2;
GLshort b=3;
glVertex2s (a, b);

glVertex3d (0.0, 0.0, 3.1415926536898)
GLdouble dvect[3] = {5.0, 9.0, 1992.0};
glVertex3dv (dvect);
```

Primitivas

Por ejemplo, el siguente código:

```
glBegin(GL_QUADS);
  glVertex2f ( 2,  2);
  glVertex2f (-2,  2);
  glVertex2f (-2, -2);
  glVertex2f ( 2, -2);
glEnd();
```

dibuja un rectángulo (indicado por la primitiva "GL_QUADS") cuyos vértices son las coordenadas de la función glVertex2f ( ). Otras primitivas, son las indicadas a continuación (la cantidad de vértices se corresponderá con la imagen indicada).



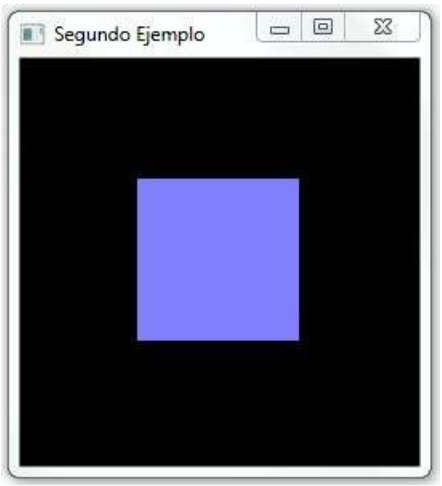
Código de draw()

La función "char Fl_Gl_Window::valid()" devuelve "cero" o "falso" cuando la ventana es creada o es cambiado su tamaño, por ejemplo, arrastrando algún lado con el mouse. Devuelve "distinto de cero" o "verdadero" luego de ejecutarse el llamado a draw().

Se utiliza esta función dentro del método draw() para evitar inicializaciones innecesarias dentro de la biblioteca OpenGL. El siguiente código esquematiza su uso:

```
void Ventana::draw()
{
 if (!valid())
 {
   glViewport(0,0,w(),h());  // Establece qué se verá en la pantalla
   glFrustum(...);  // Perspectiva en que se muestra una figura
                    // tridimensional en un plano bidimensional
   ...otras inicializaciones...
 }

 ... realizar los dibujos ...
}
```

A continuación se ejemplifica el dibujo de un un cuadrado, como el indicado en la figura siguiente. Se verá el código de un dibujo y sus transformaciones, como así también la posibilidad de cambiar el tamaño de la ventana. Dentro del código se comenta el objetivo de cada línea de comando.




```
#include <FL/Fl.H>
#include <FL/Fl_Gl_Window.H>
#include <FL/gl.h>

using namespace std;

class Ventana: public Fl_Gl_Window
{     void draw();

      // Establece en OpenGL el viewport.
      // Se debe realizar al inicio y cuando cambia el tamaño de la ventana.
      void FixViewport(int W, int H);

    public:
      Ventana(int x, int y, int w, int h, const char* titulo);
};

Ventana::Ventana(int x, int y, int w, int h, const char* titulo):
    Fl_Gl_Window(x, y, w, h, titulo)
{
    // Otras inicializaciones a las por defecto...
}

void Ventana::draw()
{
   // Fija proporciones la primera vez y cuando se cambia
   // el tamaño de la ventana.
   if (!valid())
   {  valid(1);
      FixViewport(w(),h());
   }

   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(0.5f,0.5f,1.0f);

   glPushMatrix();
   glBegin(GL_QUADS);
     glVertex2f ( 2,  2);
     glVertex2f (-2,  2);
     glVertex2f (-2, -2);
     glVertex2f ( 2, -2);
   glEnd();
   glPopMatrix();
```

```
}

void Ventana::FixViewport(int W, int H)
{  glViewport ( 0, 0, W, H );
   glMatrixMode ( GL_PROJECTION );
   glLoadIdentity ( );

   if (W>H)
      glOrtho (-5.0, 5.0*W/H, -5.0, 5.0*H/W, -5.0, 5.0);
   else
      glOrtho (-5.0*W/H, 5.0, -5.0*H/W, 5.0, -5.0, 5.0);

   glMatrixMode ( GL_MODELVIEW );
   glLoadIdentity ( );
}

int main(int argc, char **argv)
{
    Ventana ventana (100, 100, 300, 300, "Segundo Ejemplo");

    // Establece que la ventana pueda ser cambiada de tamaño arrastrando
    // con el mouse sus extremos.
    ventana.resizable(ventana);

    ventana.show();
    return Fl::run();
}
```

Los códigos anteriores son las transformaciones de la proyección, es decir, de cómo se verá el dibujo en la pantalla.

El cuadrado se realiza mendiante comandos de OpenGL de la siguiente manera:

```
    glBegin(GL_QUADS);
      glVertex2f ( 1,  1);
      glVertex2f (-1,  1);
      glVertex2f (-1, -1);
      glVertex2f ( 1, -1);
    glEnd();
```

Se puede observar que la constante para indicar cuadrados comienza con "GL_" y el comando para los vértices con "gl".

Para hacer un cuadrado o rectángulo, FLTK tiene la funcione void fl_rect(int x, int y, int w, int h) que comienzan con "fl_".

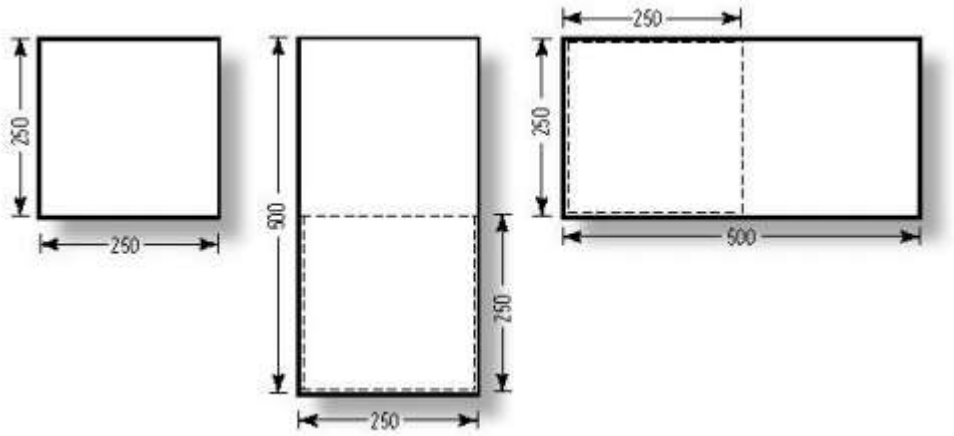
De la misma manera, en OpenGL realizaríamos un línea con el siguiente código:

```
    glBegin(GL_LINES);
      glVertex2f ( x,  y);
      glVertex2f ( x1,  y1);
    glEnd();
```

Con FLTK existe la siguiente función equivalente para la línea anterior: void fl_line(int x, int y, int x1, int y1).

Si bien en estos ejemplos se observa una simplificación con el uso de las funciones de FLTK, éstas no se extienden a sistemas tridimensionales. En este tutorial se empleará para los dibujos los comandos de OpenGL, que permite una mayor compatibilidad y extensión a dibujos tridimensionales.

Observar como estirando con el mouse en cualquier sentido los lados de la ventana, el cuadrado siempre se verá como cuadrado.



Como se ve un cuadrado en tres tamaños de ventanas.

Esto se logra con la codificación de FixViewport(int W, int H).

Orden de las transformaciones y eventos

El método handle()

Será necesario implementar este método, si se quiere que la aplicación interactue con el mouse y el teclado. Estas funciones no deben llamarse explicitamente en el programa, sino que **al pulsar en el botón del mouse, el sistema operativo procesa este evento y llama al método handle()**.

El código siguiente esquematiza la captura de algunos eventos del mouse (pulsar, pulsar y arrastrar, soltar, etc.) y del teclado, que permite una codificación o acción en respuesta.

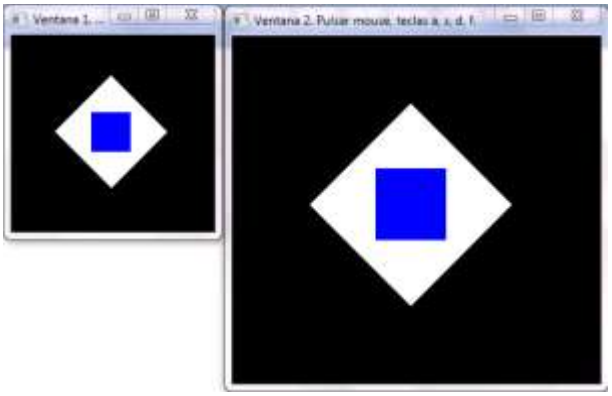
```
int Ventana::handle(int event) {
 switch(event) {
 case FL_PUSH:
        ... cuando se pulsa una tecla del mouse ...
        ... las coordenadas se obtienen llamando a Fl::event_x() and Fl::event_y()
       ... código para leer una archivo, mostrar un texto, rotar la imagen, etc.
       Idem para el resto de los eventos.
        return 1;
 case FL_DRAG:
        ... cuando se mueve y mantiene pulsado el mouse ...
        return 1;
 case FL_RELEASE:
        ... cuando se suelta la tecla del mouse ...
        return 1;
 case FL_FOCUS :
 case FL_UNFOCUS :
        ... return 1 si se quiere ingreso por el teclado, 0 en caso contrario.
 case FL_KEYBOARD:
```

```
            ... una tecla es pulsada. La tecla se conoce mediante Fl::event_key(), y su
            código ascii con Fl::event_text()
            ... return 1 si la tecla es reconocida, 0 en caso contrario...
        default:
            // pasa el tratamiento del evento a la clase madre...
            return Fl_Gl_Window::handle(event);
    }
}
```

Cuando el método handle() es llamado, al final del método debe llamarse a redraw(). No se debe realizar llamados a funciones de dibujo de OpenGL dentro del método handle().

En el siguiente ejemplo se dibujan dos cuadrados. Uno de ellos se rota pulsando el mouse, y con la tecla 'a' se aumenta sus lados, con 's' se diminuye, y con 'd' y 'f' se lo traslada en distintos sentidos.

El programa crea también dos objetos de la misma clase, apareciendo por lo tanto dos ventanas, cada uno con un control independiente de los eventos.



Código:

```
#include <FL/Fl.H>
#include <FL/Fl_Gl_Window.H>
#include <FL/gl.h>

using namespace std;

class Ventana: public Fl_Gl_Window
{   void draw();
    float angulo;     // ángulo de rotación
    float lado;       // tamaño de los lados
    float distancia;  // traslación respecto al origen

    int handle(int e);
    void FixViewport(int W, int H);

    public:
      Ventana(int x, int y, int w, int h, const char* titulo=0);
};


Ventana::Ventana(int x, int y, int w, int h, const char* titulo):
    Fl_Gl_Window(x, y, w, h, titulo)
{
    angulo=45;
    lado=4;
```

```
    distancia=0;
}


void Ventana::draw()
{
   if (!valid())
   { valid(1);
     FixViewport(w(),h());
   }

   glClear(GL_COLOR_BUFFER_BIT);

   // Dibujo 1
   glPushMatrix();
   // Traslado y Rotación:
   glTranslatef(distancia,distancia,0.0f);//Desplazar próximos dibujos a derecha
   glRotatef(angulo,0.0f,0.0f,1.0f);      // Rota próximos dibujo respecto eje Z
   glColor3f(1,1,1);

   glBegin(GL_QUADS);
    glVertex2f ( lado,  lado);
    glVertex2f (-lado,  lado);
    glVertex2f (-lado, -lado);
    glVertex2f ( lado, -lado);
   glEnd();
   glPopMatrix();

   // Dibujo 2
   glPushMatrix();
   glColor3f(0,0,1);
   glRectf(-2,-2,2,2); // otra forma de hacer un rectángulo.
   glPopMatrix();
}


int Ventana::handle(int e)
{
  char tecla;
  switch(e)
  {
   case FL_PUSH:
     angulo=angulo+5; break; // pulsando el mouse esblece otro
                             // ángulo de rotación de los dibujos
   case FL_KEYBOARD:
   {
     tecla=Fl::event_key();
     switch(tecla)
     {
       case 'a': lado=lado+1;  break;   // mayor tamaño del cuadrado
       case 's': lado=lado-1;  if (lado<0) lado=0 ; break; // menor tamaño
       case 'd': distancia++ ; break;     // corre uno de los cuadrados
       case 'f': distancia-- ; break;     // corre uno de los cuadrados
     }
     break;
   }
   default:
    // pasa el evento que no tiene "case" al gestor de eventos de la clase madre.
    return Fl_Gl_Window::handle(e);
  }
```

```
  redraw();
  return (0);
}

void Ventana::FixViewport(int W, int H)
{  glViewport ( 0, 0, W, H );
   glMatrixMode ( GL_PROJECTION );
   glLoadIdentity ( );

   if (W>H)
      glOrtho (-10.0, 10.0*W/H, -10.0, 10.0*H/W, -10.0, 10.0);
   else
      glOrtho (-10.0*W/H, 10.0, -10.0*H/W, 10.0, -10.0, 10.0);

   glMatrixMode ( GL_MODELVIEW );
   glLoadIdentity ( );
}


int main(int argc, char **argv)
{

 // creación de dos objetos a partir de la misma clase.
 // cada ventana, tendrá el control independiente de los eventos.
 Ventana ventana1(100,100,300,300, "Ventana 1. Pulsar mouse, teclas a,s,d,f.");
 Ventana ventana2(450,100,450,450, "Ventana 2. Pulsar mouse, teclas a,s,d,f.");

 ventana1.resizable(ventana1);
 ventana2.resizable(ventana2);

 ventana1.show();
 ventana2.show();

 return Fl::run();
}
```

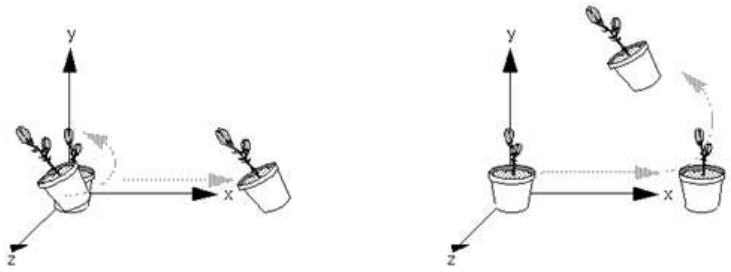
Mediante la función **glTranslatef (x,y,z)** se indica cuanto se trasladará la imagen que se dibuje a continuación. La distancia indicada en el primer parámetro, se corresponderá con el desplazamientos en x, en el segundo respecto al eje y, y en el tercero respecto al eje z.

La rotación la dibujo que estará a continuación, se realizará a través de **glRotate (angulo, x, y, z)**. El ángulo de rotación debe expresarse en grados, y el eje de rotación será respecto a un vector formado por las coordenadas [x,y,z].

Será glRotatef o glRotated según el tipo de datos que se utilice en los parámetros de la función. Los tipos de datos para 'f' o 'd' están definidos en la "Tabla de Tipos de Datos de OpenGL".

glLoadIdentity(); se encarga de limpiar las transformaciones sobre el "modelo" o dibujo.

Es interesante observar lo que se logra cuando es comentada la linea que posee a *glLoadIdentity()*; dentro de la función *draw()*. También observar lo que sucede si primero se traslada y luego rotar, y al revés.



Matrices

Se verá a continuación una descripción más detallada de algunas códigos de los ejemplos anteriores.

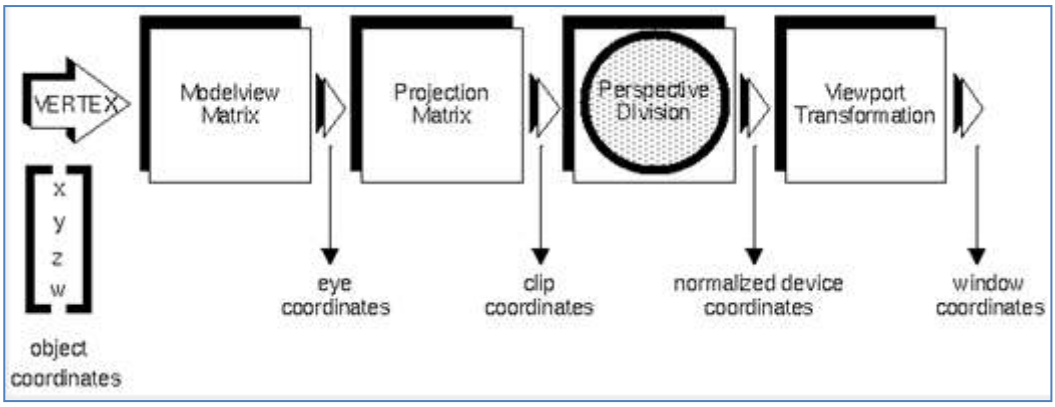
Los cálculos internos para realizar las transformaciones se simplifican gracias a las matrices. Cada una de las transformaciones pueden conseguirse multiplicando una matriz que contenga los vértices por una matriz que describa la transformación. Por tanto todas las transformaciones pueden describirse como la multiplicación de dos o más matrices.

Para realizar todas las transformaciones, deben modificarse dos matrices: **la matriz del Modelador y la matriz de Proyección**. OpenGL proporciona funciones de alto nivel que hacen muy sencillo la construcción de matrices para transformaciones. **Éstas se aplican sobre la matriz que este activa en ese instante**. Para activar una de las dos matrices, utilizamos la función glMatrixMode. Hay dos parámetros posibles:

```
glMatrixMode(GL_PROJECTION): activa la matriz de proyección.
glMatrixMode(GL_MODELVIEW): activa la del modelador.
```

Es necesario especificar con cual matriz se trabajará, para poder aplicar las transformaciones necesarias en función de lo que se desee hacer.

El camino que va desde los datos “en bruto” de los vértices hasta la coordenadas en pantalla es el siguiente.



1) “Vertex”: los vértice se convierten en una matriz 1x4 en la que los tres primeros valores son las coordenadas x,y,z. El cuarto número (llamado parámetro w) es un factor de escala que luego comentaremos. Por ejemplo, estas coordenadas las hemos generado en los códigos anteriores cuando hicimos los dibujos con glVertex3f().
2) “Modelview”: los vértices de los dibujos se multiplican por la matriz del modelador, para obtener las coordenadas oculares. Por ejemplo, esto lo hicimos cuando trasladamos y rotamos con glTranslatef(x,y,z); y glRotatef(angulo, x,y,z) a los vértices. También existe en este grupo el escalamiento de una imagen con glScalef (x,y,z) y que comentaremos luego.

3) "Projection": se debe multiplicar por la matriz de proyección para conseguir las coordenadas que luego se utilizarán para mostrar en la pantalla (parte, todo, etc.). Con esto eliminamos todos los datos que estén fuera del volumen de proyección. Por ejemplo, esto lo hemos estado haciendo con el código glOrtho (x1,x2,y1,y2,z1,z2). Existe otra más que es glFrustum( ).
4) Estas coordenadas de trabajo se dividen por el parámetro w del vértice, para hacer el escalado relativo.
5) "Viewport": finalmente, las coordenadas resultantes se mapean en un plano 2D que es lo que aparecerá en la pantalla de la computadora (que es bidimensional), mediante la transformación de la vista.

Matriz del modelador (Modelview)

La matriz del modelador es una matriz 4x4 que representa el sistema de coordenadas transformado que se está usando para colocar y orientar los objetos. Si se multiplica la matriz de los vértices (de tamaño 1x4) por ésta, obtenemos otra matriz 1x4 con los vértices transformados sobre ese sistema de coordenadas.

OpenGL proporciona funciones de alto nivel para conseguir matrices de translación, rotación y escalado, y además la multiplican por la matriz activa en ese instante.

Traslación
Por ejemplo, imaginar que se quiere dibujar un cubo realizándose todos los glVertex necesarios, unidos por líneas. Ahora pensemos que queremos moverlo 10 unidades hacia la derecha (es decir, 10 unidades en el sentido positivo del eje de las x). Para ello tendríamos que construir una matriz de transformación y multiplicarla por la matriz del modelador. OpenGL ofrece la función glTranslate, que **crea la matriz de transformación y la multiplica por la matriz que esté activa en ese instante** (en este caso debería ser la del modelador, GL_MODELVIEW).

Entonces el código quedaría de la siguiente manera:

```
glTranslatef(5.0f, 0.0f, 0.0f);
glBegin(GL_LINES);   // dibujo del cubo
   glVertex3f( );
   glVertex3f( );
   …
glEnd ( );
```

La "f" añadida a la función indica que usaremos flotantes. Los parámetros de glTranslate son las unidades a desplazar en el eje x, y y z, respectivamente. Pueden ser valores negativos, para trasladar en el sentido contrario.

Rotación
En forma similar a lo comentado anteriormente, se realiza una rotación. glRotate lleva como parámetros el ángulo a rotar (en grados, sentido horario), y después x, y y z del vector sobre el cual queremos rotar el objeto. Una rotación simple, sobre el eje y, de 10º sería: glRotatef(**10**, 0.0f, **1.0**f, 0.0f). Tener en cuenta que los águlos en las funciones trigonométricas en C++ deben ser en radianes, y aquí los ángulos deben ser en grados.

Escalado
También el escalado lo hace en forma idéntica a las dos transformaciones anteriores. Una transformación de escala incrementa el tamaño de nuestro objeto expandiendo todos los vértices a lo largo de los tres ejes por los factores especificados. La función glScale lleva como parámetros la escala en x, y y z, respectivamente. El valor 1.0f es la referencia de la escala, de tal forma que glScalef(1.0f, 1.0f, 1.0f); no modificaría el objeto en absoluto. Un valor de 2.0f sería el doble, y 0.5f sería la mitad. Por ejemplo, para que un cuadrado se muestre alargado en el sentido de las y, se debe codificar lo siguiente:

```
void display (void)
```

```
{   glScalef (1.0, 2.0, 1.0);
    …. Dibujar un cuadrado de 1x1
}
```

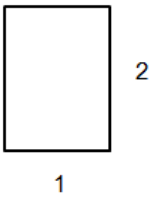
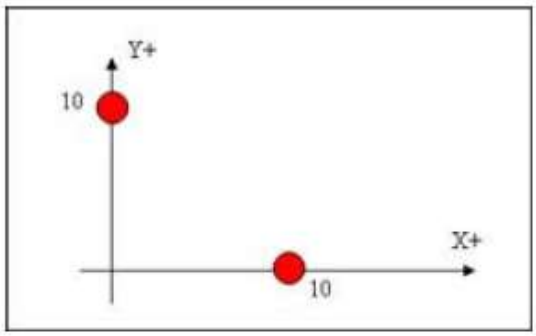


Imagen en la Pantalla

La matriz identidad

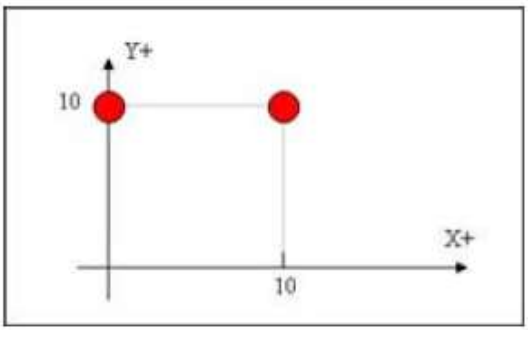
El "problema" del uso de las funciones de transformación de la vista del modelo, surge cuando se tiene más de un objeto en la escena. Estas funciones tienen efectos acumulativos. Es decir, hay que tener cuidado si se quiere dibujar dos esferas como en la figura siguiente, donde se realizar una en el centro y luego se la desplaza, y luego se dibuja otra en el centro y luego también se la desplaza.



Por ejemplo, se quiere una esfera (de radio 3) centrada en (0,10,0) y otra centrada en (10,0,0). Se puede escribir el siguiente código, que es incorrecto:

```
glTranslatef(0.0f, 10.0f, 0.0f);
dibujar_una_esfera;
glTranslate(10.0f, 0.0f, 0.0f);
dibujar_una_esfera;
```

En este código, dibujamos primero una esfera en (0,10,0) como queríamos. Pero después, estamos multiplicando la matriz del modelador que teníamos (que ya estaba transformada para dibujar la primera esfera) por otra matriz de transformación que nos desplaza 10 unidades hacia la derecha. La segunda matriz la dibujaría como está en la ilustración siguiente, en (10,10,0), y no en (10,0,0), como pretendíamos.



Para solventar este problema debemos **reiniciar la matriz del modelador** a un estado más conocido, en este caso, centrada en el origen de nuestro sistema de coordenadas oculares. Para ello se carga en la matriz del modelador la matriz identidad (una matriz 4x4 llena de ceros excepto en la diagonal, que contiene unos). Esto se consigue gracias a la función glLoadIdentity(), que no lleva parámetros.

Simplemente carga la matriz identidad en la matriz activa en ese instante. El código correcto para el ejemplo anterior quedaría de la siguiente manera:

```
glTranslatef(0.0f, 10.0f, 0.0f);
dibujar_una_esfera;
glLoadIdentity();
glTranslate(10.0f, 0.0f, 0.0f);
dibujar_una_esfera;
```

Las pilas de matrices
No siempre es deseable reiniciar la matriz del modelador con la identidad antes de colocar cada objeto. Puede suceder que se quiera almacenar el estado actual de transformación y entonces recuperarlo después de haber colocado varios objetos. Para ello, OpenGL mantiene una pila de matrices para el modelador (GL_MODELVIEW) y otra para la proyección (GL_PROJECTION). La profundidad varía dependiendo de la plataforma y puede obtenerse mediante las siguientes líneas:

```
glGet(GL_MAX_MODELVIEW_DEPTH);
glGet(GL_MAX_PROJECTION_DEPTH);
```

Por ejemplo, para la implementación en Windows, éstos valores son de 32 para GL_MODELVIEW y 2 para GL_PROJECTION.

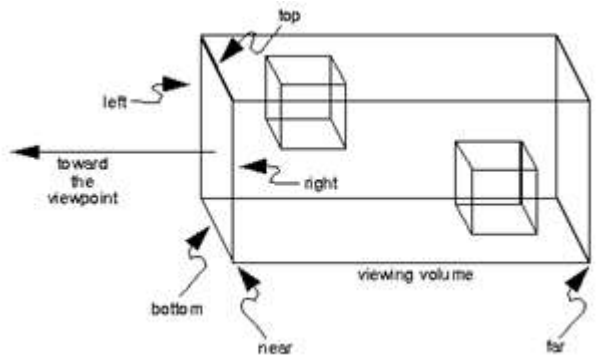
Para cargar una matriz en su pila correspondiente se usa la función **glPushMatrix** (sin parámetros), y para sacarla **glPopMatrix** (sin parámetros tampoco). Estas funciones deben encerar siempre a los dibujos, como se ejemplifica en los códigos anteriores.

La matriz de proyección (Projection)

La matriz de proyección especifica el tamaño y la forma del volumen de visualización. El volumen de visualización es todo lo que saldrá en la pantalla de la computadora. Éste está delimitado por una serie de planos de trabajo. De estos planos, los más importantes son los planos de corte, que son los que acotan el volumen de visualización por delante y por detrás. En el plano más cercano a la cámara (znear: plano en z más cercano), es donde se proyecta la escena para luego pasarla a la pantalla. Todo lo que este más adelante del plano de corte más alejado de la cámara (zfar: plano en z más lejano) no se representa.

Veremos los distintos volúmenes de visualización de las dos proyecciones más usadas: **ortográficas y perspectivas.**

Proyecciones ortográficas
Una proyección ortográfica es cuadrada en todas sus caras. Esto produce una proyección paralela, útil para aplicaciones de tipo CAD o dibujos arquitectónicos, o también para tomar medidas, ya que las dimensiones de lo que representan no se ven alteradas por la proyección.



Proyección Ortográfica (glOrtho ( ))

Una aproximación menos técnica pero más comprensible de esta proyección, es imaginar que tenemos un objeto fabricado con un material deformable, y lo aplastamos literalmente como una pared. Obtendríamos el mismo objeto, pero plano, liso. Pues eso es lo que veríamos por pantalla.

Para definir la matriz de proyección ortográfica y multiplicarla por la matriz activa (que debería ser en ese momento la de proyección, GL_PROJECTION), utilizamos la función glOrtho, que se define de la siguiente forma:

glOrtho(limiteIzquierdo, limiteDerecho, limiteAbajo, limiteArriba, znear, zfar)

siendo todos flotantes. Con esto simplemente acotamos lo que será nuestro volumen de visualización (un cubo). Cuando se trabaja exclusivamente en 2D, también puede utilizarse *gluOrtho2D(limiteIzquierdo, limiteDerecho, limiteAbajo, limiteArriba)*; observar que el prefijo en la primer función es "gl" y en esta última es "glu".

Proyecciones perspectivas
Una proyección en perspectiva ejecuta una división en perspectiva para reducir y estirar los objetos más alejados del observador.

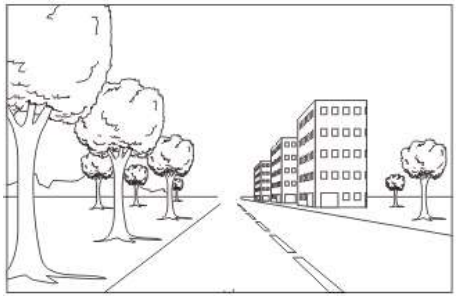


Imagen en Perspectiva

Es importante saber que las medidas de la proyección no tienen por qué coincidir con las del objeto real, ya que han sido deformadas. El volumen de visualización creado por una perspectiva se llama frustum. Un frustum es una sección piramidal, vista desde la parte afilada hasta la base.

Los párametros de la función glFrustum son los siguientes:

```
void glFrustum(   GLdouble   left,
                  GLdouble   right,
                  GLdouble   bottom,
                  GLdouble   top,
                  GLdouble   nearVal,
                  GLdouble   farVal);
```

Parámetros:

left, right
Especifica las coordenadas izquierda y derecha de los planos de corte.

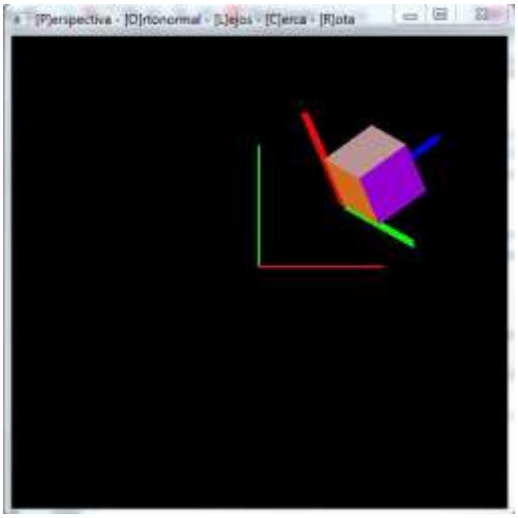
bottom, top
Especifica las coordenadas inferior y superior de los planos de corte.

nearVal, farVal
Especifica la distancia más cercana y más lejana de los planos de corte. Ambas distancias deben ser números positivos.

Se debe asumir que el ojo está localizado en (0, 0, 0) y que la imagen que se mostrará es la comprendida dentro de todos los planos de corte y estará "proyectada" en un plano ubicado a un metro de distancia del ojo.

Ejemplo de transformaciones y proyecciones.

El el siguiente código se tiene unos ejes cartesianos que quedan fijos y otros que tiene un cubo que se mueven. Pulsando "r" rota, "c" y "l" para mover respecto del eje Z. Pulsando "o" o "p" permite cambiar de proyecciones entre glOrtho y glFrustum.



```
#include <FL/Fl.H>
#include <FL/Fl_Gl_Window.H>
#include <FL/gl.h>

using namespace std;

class Ventana: public Fl_Gl_Window
{
    float angulo, mover;
    int tipoProyeccion;

    void dibujaEjes(void);
    void dibujaCubo(void);
    void draw();
    int handle(int e);
    void FixViewport(int W, int H);

    public:
      Ventana(int x, int y, int w, int h, const char* titulo=0);
};

Ventana::Ventana(int x, int y, int w, int h, const char* titulo):
    Fl_Gl_Window(x, y, w, h, titulo)
{
   angulo = 45.0f;     // rotación
   mover = 0;          // movimiento en el eje z.
   tipoProyeccion = 1; // 1:glOrtho  0:glFrustrum
}
```

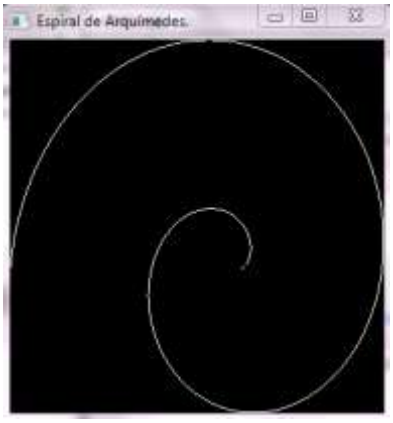
```
void Ventana::dibujaEjes(void)
{
  // eje x
  glBegin(GL_LINES);
    glColor3f(1,0,0); glVertex3f(0,0,0); glVertex3f(10,0,0);
  glEnd();

  // eje y
  glBegin(GL_LINES);
    glColor3f(0,1,0);   glVertex3f(0,0,0); glVertex3f(0,10,0);
  glEnd();

  // eje z
  glBegin(GL_LINES);
    glColor3f(0,0,1);   glVertex3f(0,0,0); glVertex3f(0,0,10);
  glEnd();
}


void Ventana::dibujaCubo(void)
{
   //cara plano xy
   glBegin(GL_QUADS);
    glColor3f(0.82, 0.41, 0.12);
    glVertex3f(0,0 ,0 );      glVertex3f(0, 5, 0);
    glVertex3f(5, 5, 0);      glVertex3f(5, 0, 0);
   glEnd();

   //cara plano yz
   glBegin(GL_QUADS);
    glColor3f(1, 0, 0);
    glVertex3f(0, 0, 0);      glVertex3f(0, 0, 5);
    glVertex3f(0, 5, 5);      glVertex3f(0, 5, 0);
   glEnd();

   //cara plano zx
   glBegin(GL_QUADS);
    glColor3f(0, 1, 0);
    glVertex3f(0, 0, 0);      glVertex3f(5, 0, 0);
    glVertex3f(5, 0, 5);      glVertex3f(0, 0, 5);
   glEnd();

   //cara paralela al plano xy
   glBegin(GL_QUADS);
    glColor3f(0, 0, 1);
    glVertex3f(0, 0, 5);      glVertex3f(0, 5, 5);
    glVertex3f(5, 5, 5);      glVertex3f(5, 0, 5);
   glEnd();

   //cara paralela al plano yz
   glBegin(GL_QUADS);
    glColor3f(0.73, 0.58, 0.58);
    glVertex3f(5, 0, 5);      glVertex3f(5, 5, 5);
    glVertex3f(5, 5, 0);      glVertex3f(5, 0 ,0 );
   glEnd();

   //cara paralela al plano zx
   glBegin(GL_QUADS);
    glColor3f(0.58, 0, 0.82);
    glVertex3f(0, 5, 0);      glVertex3f(5, 5, 0);
```

```
    glVertex3f(5, 5, 5);      glVertex3f(0, 5, 5);
   glEnd();
}


void Ventana::draw()
{  if (!valid())
   { valid(1);
     FixViewport(w(),h());
     glEnable(GL_DEPTH_TEST);  // no muestra lo que queda oculto en 3D
   }

   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glLoadIdentity();

   // Para una mejor visión con la proyección Frustrum.
   // Es equivalente a gluLookAt (especifica una posición distinta del ojo)
   // Se realiza una transformación general de toda la imagen.
   if (!tipoProyeccion)
   {
     glTranslatef( 0, 0, -10 );
     glRotatef ( 0, 0, 0, 0 );
     // coordenadas del ojo (los objetos se mueven en sentido opuesto)
     // a donde mira.
   }

   glLineWidth(2);
   dibujaEjes();    // estos ejes quedrán fijos. No hay un glTranslate antes.

   glRotatef(angulo, 0, 1, 0);  // los próximos dibujos rotarán.
   glTranslatef(0.0f, 0.0f, mover);
   glScalef(1,1,1);  // Con (2,1,1) el cubo estará más alargado en el eje x.

   glPushMatrix();
   dibujaCubo();
   glLineWidth(8);   // establece lineas más gruesas
   dibujaEjes();     // estos ejes serán móviles por el glTranslate anterior
   glPopMatrix();

   glFlush(); // se ejecutran todos los comandos en espera.
}


int Ventana::handle(int e)
{ char tecla;
  switch(e)
  { case FL_KEYBOARD:
    { tecla=Fl::event_key();
      switch(tecla)
      {   case 'o': // proyección glOrtho
                    valid(0);  // para que llame a FixViewport()
                    tipoProyeccion=1; break;

          case 'p': // proyección glFrustrum
                    valid(0);  // para que llame a FixViewport();
                    tipoProyeccion=0; break;

          case 'l': // mueve la imagen "L"ejos
                    mover=mover-3; break;
```

```
          case 'c': // mueve la imagen "C"erca
                    mover=mover+3; break;

          case 'r': // "R"ota la imagen
                    angulo+=5.0f; break;

          case  27: // escape: cierra la ventana
                    exit(0); break;
        }
        break;
      }
     default:
       return Fl_Gl_Window::handle(e);
  }
  redraw();
  return (0);
}


void Ventana::FixViewport(int W, int H)
{  glViewport ( 0, 0, W, H );
   glMatrixMode ( GL_PROJECTION );
   glLoadIdentity ( );

   if (tipoProyeccion)
   { if (W>H)
       glOrtho (-20, 20*W/H, -20, 20*H/W, -20, 20);
     else
       glOrtho (-20*W/H, 20, -20*H/W, 20, -20, 20);

     glMatrixMode ( GL_MODELVIEW );
     glLoadIdentity ( );
   }
   else
   {
     glFrustum(-8, 8, -8, 8, 1, 20);
     glMatrixMode ( GL_MODELVIEW );
     glLoadIdentity ( );
   }
}


int main(int argc, char **argv)
{
 Ventana ventana ( 50, 50, 500, 500, "[P]erspectiva - [O]rtonormal -
                   [L]ejos - [C]erca - [R]ota");

 ventana.resizable(ventana);
 ventana.show();
 return Fl::run();
}
```

Código de una curva

El siguiente código corresponde a la graficación de una curva, la ecuación representará en coordenadas polares a la “espiral de Arquímedes”.




```
#include <FL/Fl.H>
#include <FL/Fl_Gl_Window.H>
#include <FL/gl.h>
#include <cfloat>   // máximo float=FLT_MAX
#include <cmath>
using namespace std;

class Ventana: public Fl_Gl_Window
{  float x_min, x_max, y_min, y_max;  // máximos y mínimos de la imagen
   float xg, yg; // coordenadas gráficas
   int vueltas;  // vueltas de la espiral
   float r;      // radio ecuaciones coordenadas polares

   void draw ();
   int  handle (int e);
   void FixViewport (int W, int H);
   void inicializaMaxMin ( );
   void espiral (float &x, float &y, float t);  // ecuación espiral de Arquimedes
   void CalculaMaxMin ();   // obtiene el máximo y mínimo de la imagen

   public:
   Ventana(int x, int y, int w, int h, const char* titulo=0);
};

void Ventana::FixViewport(int W, int H)
{  glViewport ( 0, 0, W, H );
   glMatrixMode ( GL_PROJECTION );
   glLoadIdentity ( );

   if (W>H) glOrtho (x_min, x_max*W/H, y_min, y_max*H/W, -1, 1);
   else     glOrtho (x_min*W/H, x_max, y_min*H/W, y_max, -1, 1);

   glMatrixMode ( GL_MODELVIEW );
   glLoadIdentity ( );
}

void Ventana::inicializaMaxMin()
{  x_min=FLT_MAX;
   x_max=-FLT_MAX;
   y_min=FLT_MAX;
   y_max=-FLT_MAX;
}

void Ventana::espiral(float &x, float &y, float t)
{   r=0.5+1.2*t;  // espiral en polares
```

```
    x=r*cos(t);
    y=r*sin(t);
};

void Ventana::CalculaMaxMin ()
{  float x,y;
   for ( float ang=0.0; ang<vueltas*M_PI; ang=ang+0.01 )
   { espiral (x, y, ang);

     if (x<x_min) x_min=x;
     if (x>x_max) x_max=x;
     if (y<y_min) y_min=y;
     if (y>y_max) y_max=y;
   }
}

Ventana::Ventana(int x, int y, int w, int h, const char* titulo):
                 Fl_Gl_Window(x, y, w, h, titulo)
{  vueltas=3;
   r=1;
   inicializaMaxMin();
   CalculaMaxMin ();
}

void Ventana::draw()
{  if (!valid())
   { valid(1);
     FixViewport(w(),h());
   }

   glClear(GL_COLOR_BUFFER_BIT);
   glPushMatrix();
   glBegin(GL_LINE_STRIP);
   for ( float ang=0.0; ang<vueltas*M_PI; ang=ang+0.01 )
   { espiral (xg, yg, ang);
     glVertex2f (xg, yg);
   }
   glEnd();
   glPopMatrix();
}


int Ventana::handle(int e)
{   switch(e)
    { case FL_PUSH:
            vueltas=vueltas+1; break; // aumenta las vueltas de la espiral
      default:  Fl_Gl_Window::handle(e);
    }

    // Cambió la espiral, hay que calcular máximos y mínimos de la imagen
    inicializaMaxMin();
    CalculaMaxMin ();
    FixViewport(w(),h());
    redraw();
    return (0);
}


int main(int argc, char **argv)
{  Ventana ventana ( 100, 100, 300, 300, "Espiral de Arquímedes.");
```

```
   ventana.resizable(ventana);
   ventana.show();

   return Fl::run();
}
```

Código de una imagen

Una imagen es una matriz de valores donde a cada valor se le llama pixel.

Suponga que los pixeles están representados por valores de 1 byte sin signo (unsigned char), y que además se tiene que cada imagen está almacenada en un archivo binaro que contiene:

- cantidad de filas de la imagen (entero sin signo),
- cantidad de columnas de la imagen (entero sin signo) y
- los pixeles de la imagen (unsigned chars).

Para está situación, se tendría:

- **Atributos:**

Para guarder la imagen: vector <vector <unsigned char> > imagen;

Los extremos de la imagen: float min_x, max_x, min_y, max_y; para utilizar en: glOrtho(min_x, max_x, min_y, max_y, -1.0f, 1.0f);

- **Método draw() tendría:**

```
glPushMatrix();
glBegin(GL_POINTS);
  for(unsigned fil=0; fil<imagen.size(); fil++)
   for(unsigned col=0; col<imagen[0].size(); ++col)
     {  glColor3ub(imagen[fil][col], imagen[fil][col], imagen[fil][col]);
        glVertex3i(col, fil, 0);
     }
glEnd();
glPopMatrix();
```

- **Lectura de datos de la imagen:**

```
ifstream archi("imagen.bin", ios::in|ios::binary);
unsigned filas, colum;
archi.read((char*)&filas, sizeof(filas));
archi.read((char*)&colum, sizeof(colum));

imagen.resize(filas);
unsigned char pixel;
for(unsigned i=0; i<filas; ++i)
{
   imagen[i].resize(colum);
   for(unsigned j=0; j<colum; ++j)
   {
      archi.read((char*)&pixel, sizeof(pixel));
      imagen[i][j] = pixel;
   }
}
archi.close();
```

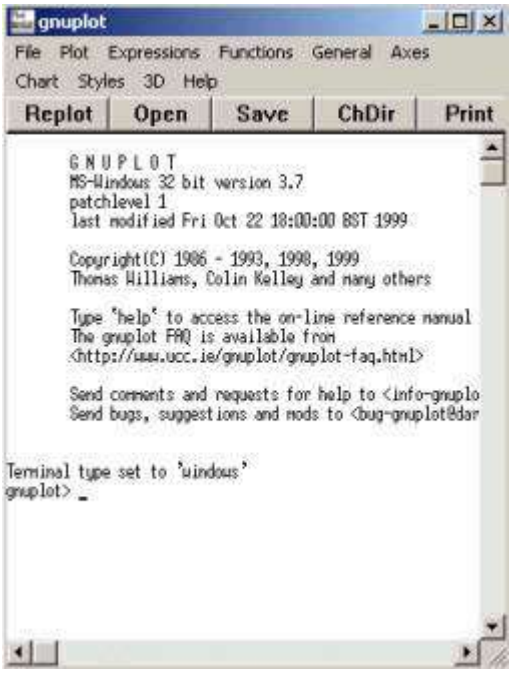
Graficación con GNUPlot

Introducción

Gnuplot es un programa que produce gráficos en 2D y 3D a partir de ecuaciones o valores. Puede ser ejecutado desde la línea de comando y trabajar en forma interactiva dentro de una ventana en un entorno como el mostrado en la figura siguiente.

Lo último que aparece es el prompt del programa desde donde se puede ejecutar comandos. Por ejemplo, se puede acceder a la ayuda en línea mediante el comando help seguido opcionalmente por un comando u opción para consultar.

Se sale del programa con el comando quit o tipeando Ctrl-D.



Pero también puede ser llamado desde la línea de comando, pasándole un archivo que posea la lista de instrucciones indicando lo que gnuplot deberá graficar. Teniendo en cuenta esta última modalidad, puede llamarse desde un programa en C y graficar en base a un archivo construido también desde C mediante:

```
system("gnuplot graf.plt");
```

se está invocando desde un programa en C al sistema operativo para que ejecute *gnuplot*, donde *graf.plt* tiene las instrucciones de lo que se quiere graficar. Si se trabaja en Windows, el comando es *wgnuplot*.

Gnuplot (www.gnuplot.info) es de libre distribución y existe en varias versiones de sistema operativo.

Se mostrará las funcionalidades más usadas en la graficación de funciones a partir de programas en C. Mayor información sobre el uso del programa puede obtenerse en: manual de Gnuplot (http://www.ucc.ie/gnuplot/gnuplot.html), página de demos de Gnuplot (http://www.gnuplot.vt.edu/gnuplot/gpdocs/all2.html) y el tutorial de la Universidad de Iowa (http://www.cs.uni.edu/Help/gnuplot/). También se puede consultar en el newsgroup *comp.graphics.apps.gnuplot*.

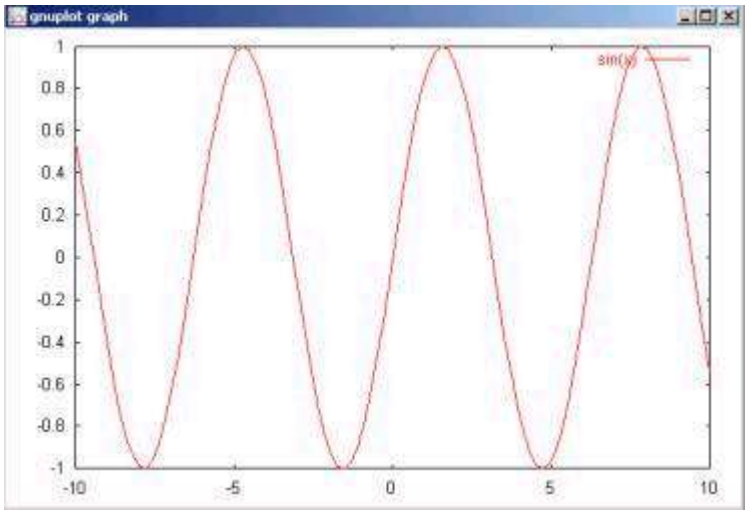
Primer ejemplo

Los ejemplos que se verán a continuación contemplan el llamado a GnuPlot desde un programa escrito en C. En el siguiente ejemplo[1] "../" está puesto porque el ejecutable gnuplot, en este ejemplo, está instalado en el directorio superior al que se está corriendo en programa. Si estuviese en el mismo directorio que el ejecutable de este ejemplo entonces se debe colocar "./". Si está en cualquier otro directorio, se debe colocar el camino completo.

Si el directorio donde está instalado wgnuplot se encuentre en el path del "autoexec.bat", no es necesario colocar "../".

[1] Se debe verificar que en el directorio "c:\cygwin\bin" exista el archivo "sh.exe". Si no existe, hacer una copia de "bash.exe" y renombrarlo "sh.exe". No se debe borrar el archivo "bash.exe".

```
#include <cstdlib>
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
// generación del archivo script
   ofstream  graf("graf.plt");

   graf << "plot sin(x)" << endl;
   graf << "pause -1 'Pulsar para finalizar' " << endl;
   graf.close();

// llamado a gnuplot con el archivo generado
 system("../wgnuplot graf.plt"); // en Linux es "gnuplot" y no se coloca "../"
}
```



La primera parte del programa genera el archivo que va a procesar gnuplot. Luego, `plot sin(x)` indica que se grafique la función seno(x) y `pause -1 'Pulsar para finalizar'` indican que realice una pausa mostrando un botón a pulsar con el mensaje indicado. En este ejemplo, Gnuplot eligió automáticamente el rango de las escalas.

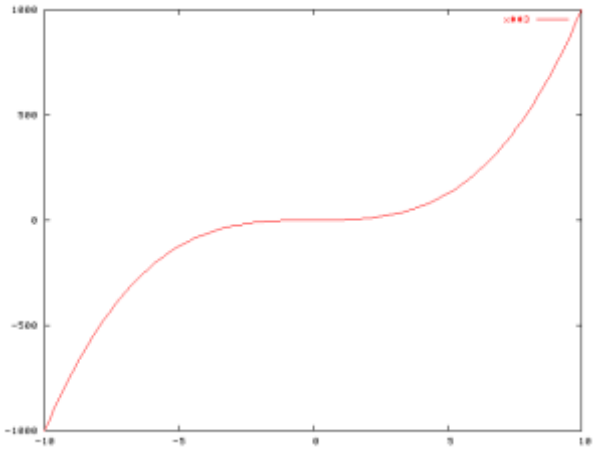
En general, son válidas las expresiones matemáticas aceptadas por C, por ejemplo: abs(x), acos(x), asin(x), atan(x), cos(x), cosh(x), erf(x), exp(x), inverf(x), invnorm(x), log(x), log10(x), norm(x), rand(x), sign(x), sin(x), sinh(x), sqrt(x), tan(x) y tanh(x).

El operador de exponente es "**" . Además posee operadores binarios y unarios. Incluye también otras funciones matemáticas como Bessel, gamma, ibeta, igamma, lgamma, funciones con argumentos complejos.

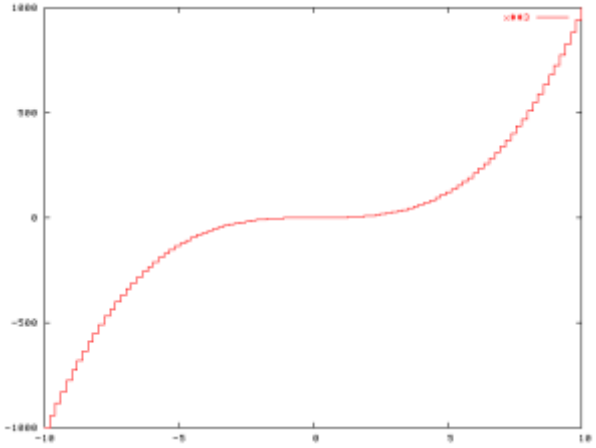
El archivo de texto con las instrucciones que debe procesar GnuPlot puede ser generado con un editor, pero en estos ejemplos se supone que se está integrando la graficación en un programa más completo.

Otros ejemplos:

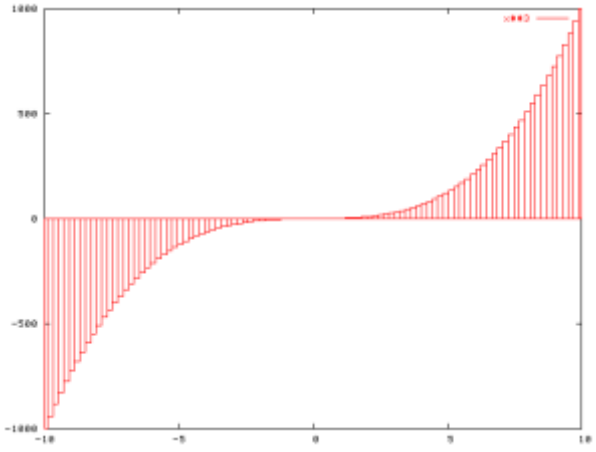
plot x**3



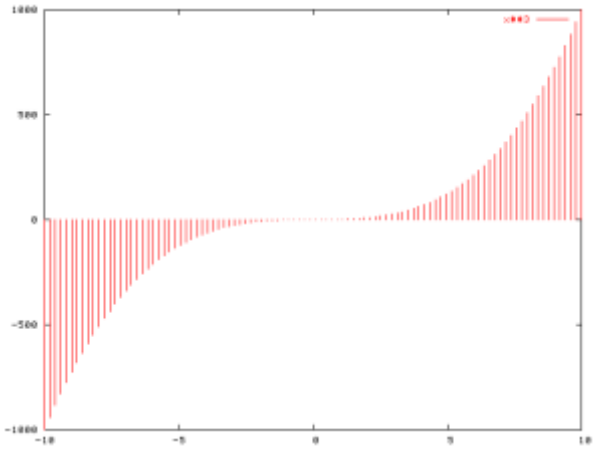
plot x**3 with steps



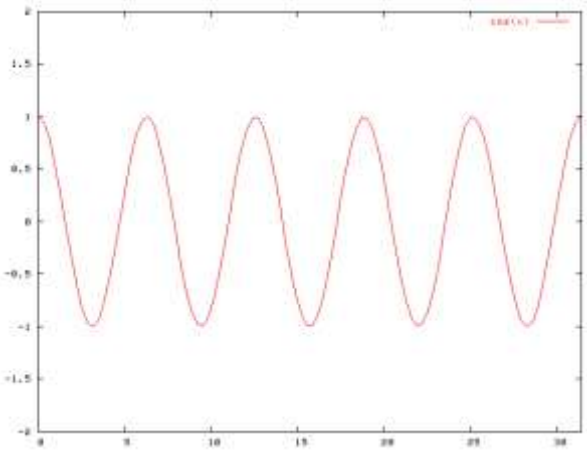
plot x**3 with boxes



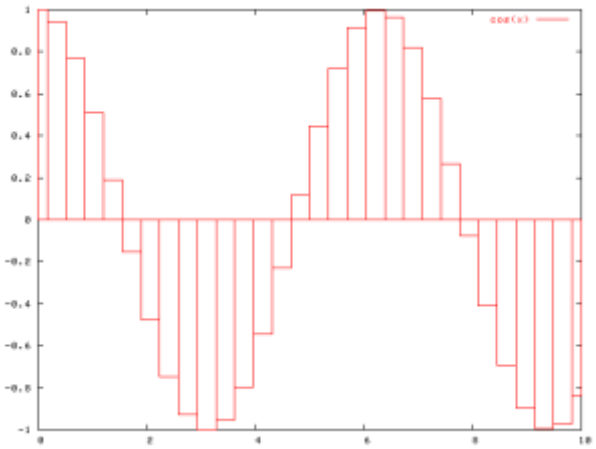
plot x**3 with impulses



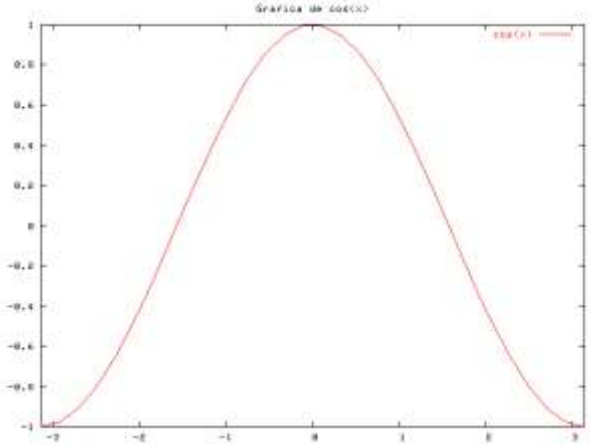
plot [0:10*pi][-2:2] cos(x)



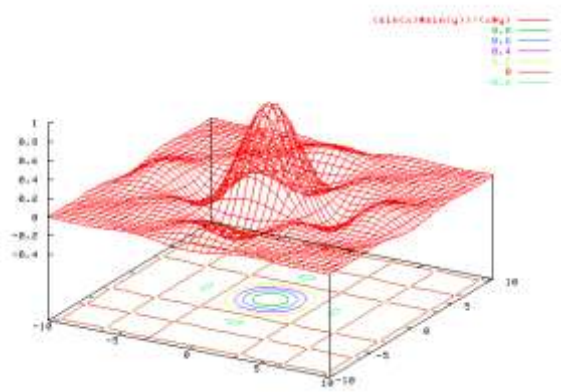
set samples 30
plot [0:10] cos(x) with boxes



set title "Gráfica de cos(x)"
plot [-pi:pi] cos(x)



set contour base
set isosamples 40,40
splot (sin(x)*sin(y))/(x*y)



Los comandos *plot* y *splot*

plot y *splot* son los comandos de graficación. plot es usado para graficar funciones y tabla de datos en 2D, mientras que splot es para superficies y datos en 3D.

Sintaxis:

```
plot {[rangos]}
     {[función] | {"[archivodedatos]" { archivodedatos -modificadores}}}
     {ejes [ejes] } { [titulo-spec] } {with [estilo] }
     {, {definiciones,} [función] ...}
```

donde [función] o el nombre del archivo de datos deberá estar entre comillas. En los siguientes ejemplos se verá con mayor claridad.

Segundo ejemplo

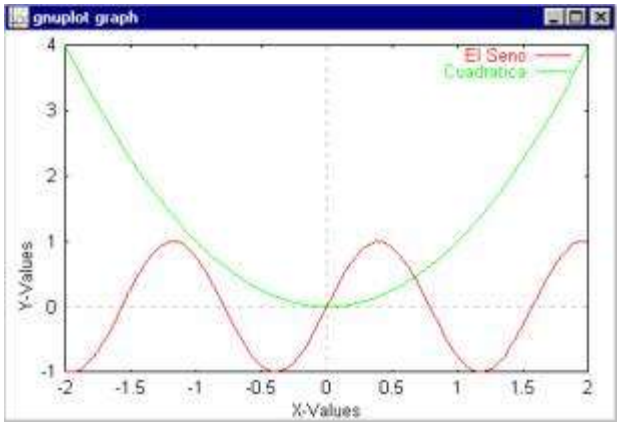
En este ejemplo se grafican varias curvas (en el comando plot se separan las curvas por una coma) y se muestran otras características que permite gnuplot.

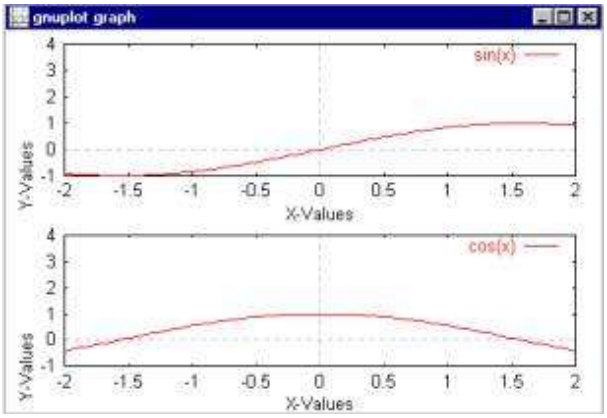
```
ofstream  graf("graf.plt");
graf << "set xrange [-2:2]" << endl
     << "set yrange [-1:4]" << endl
     << "set xlabel 'X-Values' " << endl
     << "set ylabel 'Y-Values' 0, -10 " << endl
     << "set zeroaxis"  << endl
     << "plot sin(x*4) title 'El Seno', x**2 title 'Cuadratica' " << endl
     << "pause -1 'Pulsar para terminar' " << endl;
```

Con *xrange* e *yrange* se establece el rango de los ejes. Mediante *xlabel* e *ylabel* se muestran los carteles de los ejes. Con *zeroaxis* se indica que se debe dibujar los ejes para x=0 e y=0. Por último, se muestra la graficación de dos funciones en un mismo sistema de ejes, cada una con un título personalizado.

También puede realizarse gráficos con distintos ejes en una misma ventana. Utilizando la instrucción *multiplot*. Con *nomultiplot* se vuelve a establecer la condición de que todos los gráficos van en los mismos ejes de coordenadas.

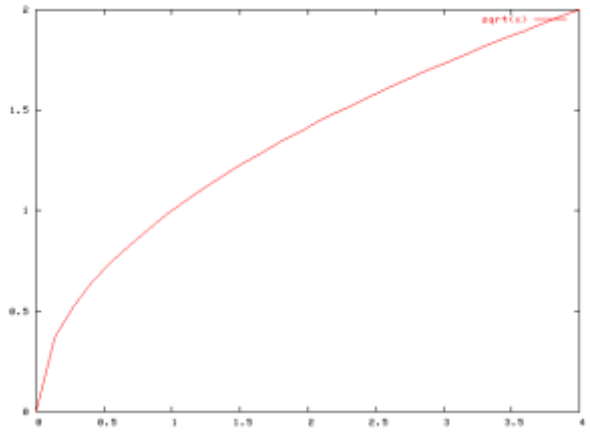
```
graf << "set multiplot" << endl
     << "set size 1,0.5" << endl
     << "set origin 0.0,0.5;  plot sin(x)" << endl
     << "set origin 0.0,0.0;  plot cos(x)" << endl
     << "set nomultiplot" << endl
     << "pause -1 'Salir' " << endl;
```



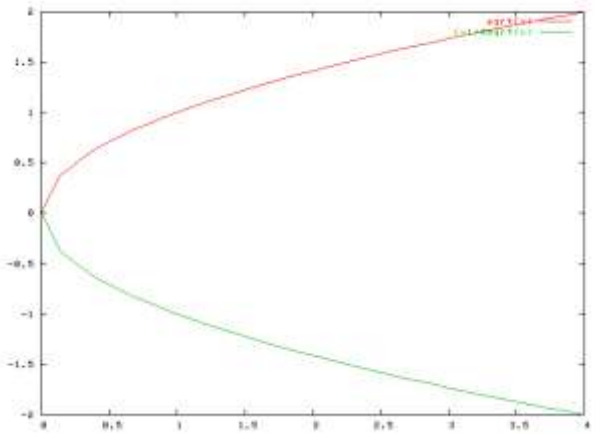


También se puede "agregar" curvas a un gráfico mostrado a través del comando replot (repite el último comando plot usado):

plot [0:4] sqrt(x)



replot (-1)*sqrt(x)



Coordenadas polares

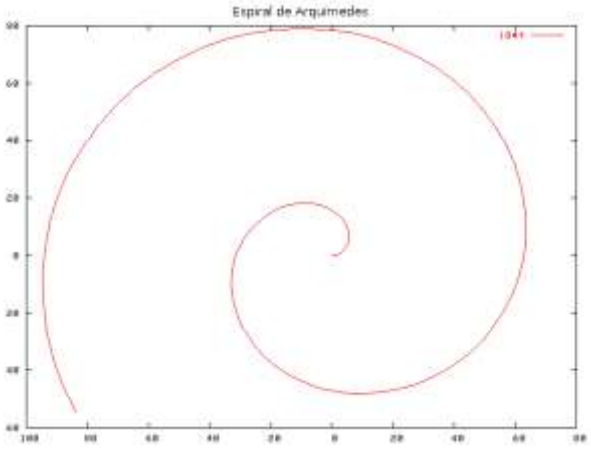
Para representar funciones en coordenadas polares se tiene que activar con la opción: `set polar`.

En coordenadas polares la variable t hace referencia al ángulo. Por defecto se grafica entre 0 y 2*pi, pero esto puede cambiarse con el comando `set trange[t1:t2]` donde t1 t2 son los valores del ángulo entre los que está definida la función. Por ejemplo `set trange[0:pi]`

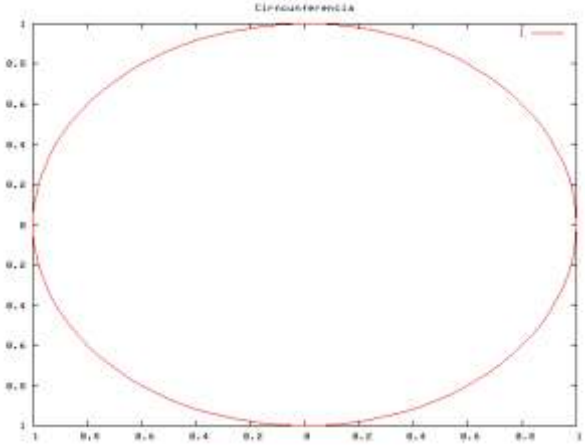
gnuplot trabaja por defecto con radiantes, pero pueden utilizarse en grados mediante el comando `set angles negrees`. Para volver a radianes utilizar `set angles radians`

Ejemplos:

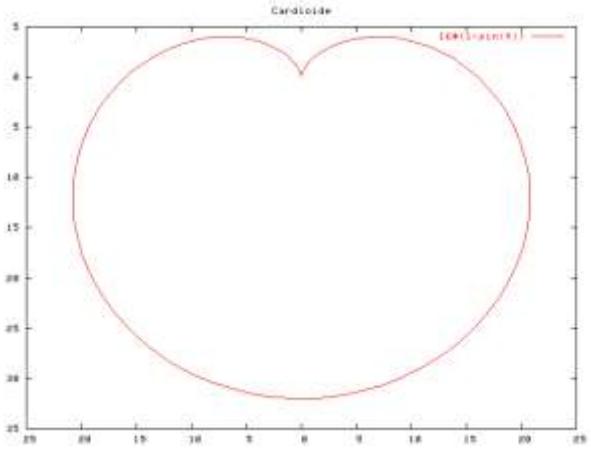
```
set title "Espiral de Arquímedes"
plot [0:10] 10*t
```



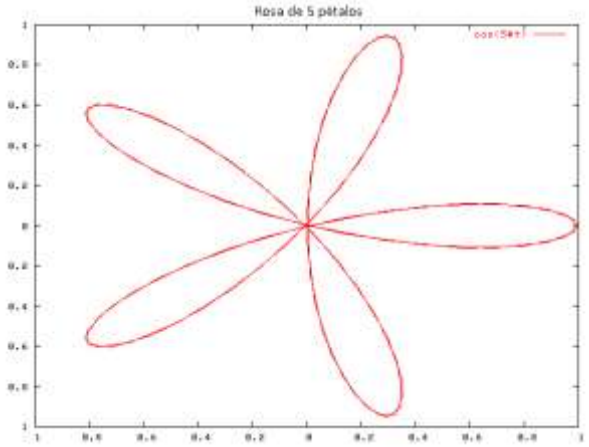

```
set title "Circunferencia"
plot 1
```




```
set title "Cardioide"
plot 16*(1-sin(t))
```



set title "Rosa de 5 pétalos"
plot cos(5*t)



Graficación de puntos

Un conjunto de datos contenidos en un archivo también pueden ser graficados mediante *plot* o *splot*. Los datos deben estar en columnas separados por espacios en blanco o tabulaciones únicamente, no está permitido separar por comas. Las líneas que comienzan con el caracter # son tratadas como un comentario e ignoradas por gnuplot. Esta característica hace que sea posible poner encabezados en los archivos para saber qué es lo que esos datos representan.

Por ejemplo se tiene los siguientes tres archivos de texto:

```
# Este archivo es "datos1.dat"
# Datos de Flexión de una viga
# Flexión
   0.000
   0.001
   0.002
   0.003
   0.0031
   0.004
   0.0041
   0.005
   0.010
   0.020
```

```
# Este archivo es "datos2.dat"
# Datos de Flexión y Fuerza
# aplicada a una viga y una barra.
# Flexión    Fuerza-barra
  0.000            0
  0.001          104
  0.002          202
  0.003          298
  0.0031         290
  0.004          289
  0.0041         291
  0.005          310
  0.010          311
  0.020          280
```

```
# Este archivo es "datos3.dat"
# Datos de Flexión y Fuerza
# aplicada a una viga y una barra.
# Flexión  Fuerza-barra   Fuerza-Viga
  0.000          0              0
  0.001        104             51
  0.002        202            101
  0.003        298            148
  0.0031       290            149
  0.004        289            201
  0.0041       291            209
  0.005        310            250
  0.010        311            260
  0.020        280            240
```

El primer archivo corresponde a valores de flexión de una barra o viga de acero. El segundo archivo posee los valores correspondientes a la fuerza aplicada a la barra para obtener la flexión consignada. En el tercer archivo se agregó la fuerza que se necesitaría en una viga para lograr las mismas flexiones.

Para mostrar el primer archivo sólo se agrega el nombre del archivo y el título. El eje x muestra el número de orden del dato. Para el segundo archivo, como se tiene dos columnas, se interpreta la primera como eje x y la segunda como eje y. Para el tercer archivo se debe especificar qué columnas se graficarán siendo la primer columna indicada lo correspondiente al eje x y la otra al eje y.

Con en el siguiente código se muestran los gráficos:

```
ofstream  graf("graf.plt");
graf << "plot 'datos1.dat' title 'Flexión'" << endl
     << "pause -1 'Próximo gráfico' " << endl;

// segundo gráfico
graf << "set grid" << endl
```

```
     << "plot 'datos2.dat' title 'Flexión y Fuerza' with lines" << endl
     << "pause -1 'Próximo gráfico' " << endl;

      // tercer gráfico
graf << "set nogrid" << endl
     << "plot 'datos3.dat' using 1:2 title 'Flexión y Fuerza-Barra',\
        'datos3.dat' using 1:3 smooth csplines title 'Flexión y Fuerza-Viga' "
     << endl
     << "pause -1 'Próximo gráfico' " << endl;

      // gráfico con más detalles
graf  << "set title 'Fuerza de Flexión en una Columna y Viga'" << endl
      << "set xlabel 'Flexión (metros)'" << endl
      << "set ylabel 'Fuerza (kN)'" << endl
      << "set key 0.01,100" << endl    // coordenas para los 'titles'
      << "set label 'Punto de Inflexión' at 0.003,260" << endl
      << "set arrow from 0.0028,250 to 0.003,280" << endl   // dibujo de una
         flecha
      << "set xr [0.0:0.022]" << endl
      << "set yr [0:325]" << endl
      << "plot 'datos3.dat' using 1:2 title 'Flexión y Fuerza-Barra' with
         linespoints,\
        'datos3.dat' using 1:3 title 'Flexión y Fuerza-Viga' with points" << endl
      << "pause -1 'Salir' " << endl;
```

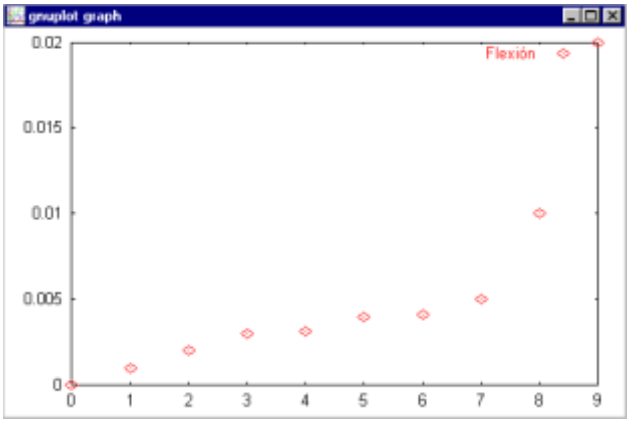


Gráfico 1

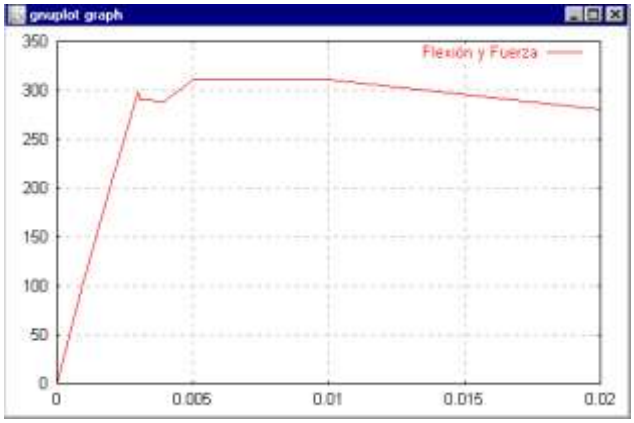


Gráfico 2

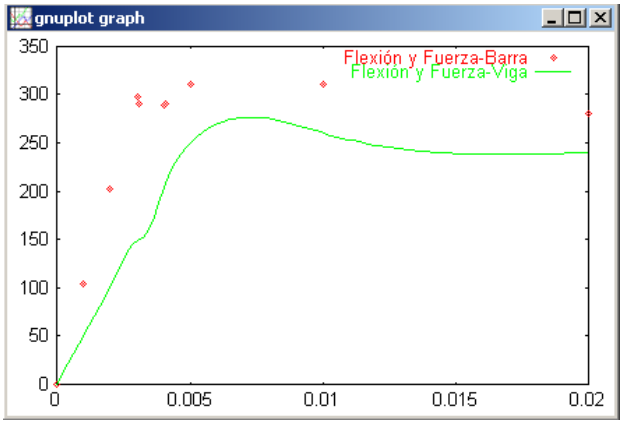


Gráfico 3

Gráfico 4

En el segundo gráfico se indicó que los puntos sean unidos con líneas mediante *with lines*. Una línea en blanco en el archivo de los datos implica una interrupción en las líneas que los unen. Otras alternativas a lines son:

`points, linespoints, impulses, dots, steps, fsteps, histeps, errorbars, xerrorbars, yerrorbars, xyerrorbars, boxes, boxerrorbars, boxxyerrorbars, financebars, candlesticks o vector.`

Además, los nombres *using*, *title* y *with* pueden ser abreviados mediante *u*, *t* y *w*. También puede graficarse en una escala logarítimica con *set logscale*.

No se debe tipear ningún espacio en blanco a continuación del caracter "\", utilizado para cortar una línea de código en C.

Resumen de algunas personalizaciones

Las personalizaciones realizadas con el comando **set**, vistas en los ejemplos anteriores, están resumidas a continuación (los valores numéricos son arbitrarios). Cuando se establece una personalización, se afectarán todos los gráficos que se realicen a continuación. Si se quiere aplicar sobre un gráfico ya dibujado, se deberá hacer *replot*. También puede utilizarse lo detallado anteriormente para quitar personalizaciones realizadas para un gráfico previo.

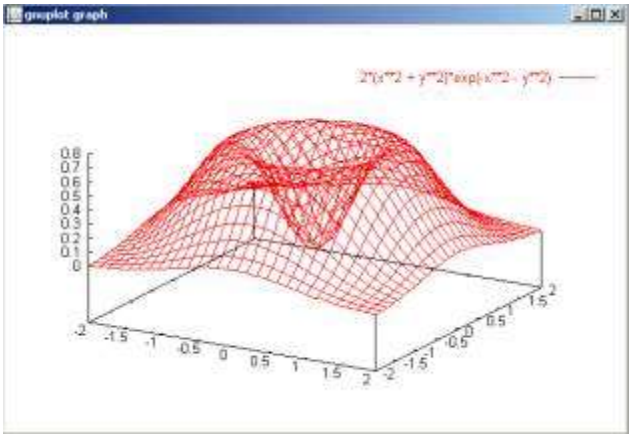
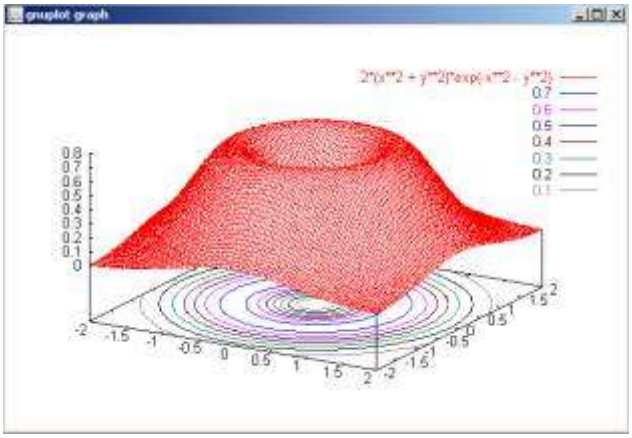
Crear el título	Set title "Fuerza de Flexión "
Poner una etiqueta en el eje x	Set xlabel "Flexión (metros)"
Poner una etiqueta en el eje y	Set ylabel "Fuerza (kN)"
Cambiar el rango del eje x	Set xrange [0.001:0.005]
Cambiar el rango del eje y	Set yrange [20:500]
Rangos automáticos	Set autoscale
Mover la referencia	Set key 0.01,100
Eliminar la referencia	Set nokey
Poner una etiqueta	Set label "Punto inflexión" at 0.003, 260
Eliminar todas las etiquetas	Set nolabel
Utilizar escala logarítmica en x	Set logscale
Utilizar escala logarítmica en y	Set nologscale; set logscale y
Posición de las divisiones en x	Set xtics (0.002,0.004,0.006,0.008)
Divisiones por omisión	Set noxtics; set xtics

Otras características para usar con el comando **set** son: arrow, border, clip, contour, grid, mapping, polar, surface, time, view, y muchas otras más.

Gráficos de superficie

El siguiente ejemplo muestra el gráfico de una superficie utilizando *splot*.

```
ofstream  graf("graf.plt");
graf << "set isosamples 30, 30" << endl
     << "splot [-2:2] [-2:2] 2*(x**2 + y**2)*exp(-x**2 - y**2)" << endl

     << "pause -1 'Continuar' " << endl
     << "set isosamples 100, 100" << endl
     << "set hidden3d" << endl
     << "set contour base" << endl
     << "replot" << endl
     << "pause -1 'Salir' " << endl;
```

Las personalizaciones realizadas son:

- *Isosamples*: especifica el número de muestras a graficar. Observar que a mayor argumento, se obtiene una superficie más definida.
- *Hidden3d*: indica que la superficie no será transparente.
- *Contourn base*: muestra curvas de nivel en la base del gráfico.

Muchas otras personalizaciones están disponibles para superficies, como distintas vistas y zoom (set view horizontal_angle, vertical_angle, zoom).

Funciones

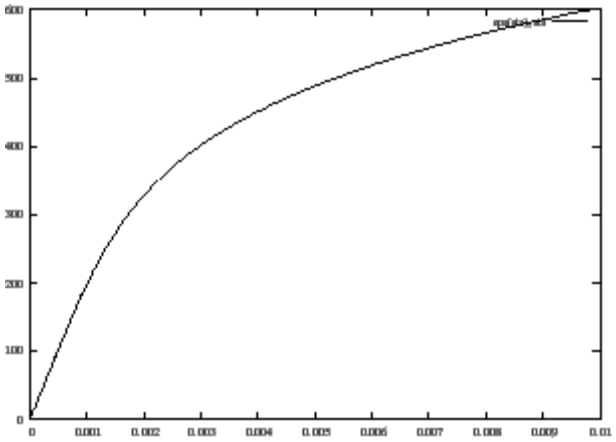
Con gnuplot se pueden definir constantes y funciones, suponer la siguiente expresión:

$$\epsilon = \frac{\sigma}{E} + \left(\frac{\sigma}{K'}\right)^{\left(\frac{1}{n'}\right)}$$

Se quiere graficar la función con ϵ (llamada eps) en el eje x y σ (llamada sts) sobre el eje y. La ecuación es válida en el rango σ=[0:600]. Primero, se debe usara el comando *reset* para iniciar los valores, luego se ponen las condiciones necesarias en gnuplot para graficar paramétricamente (dando valores a algunos parámetros), entonces se cambia la variable independiente t para que sea sts. Como se conoce el rango válido para sts se puede cambiar eso también. La función no está escrictamente definida en 0, así que se puede poner un valor muy pequeño en el límite inferior.

Los comandos para gnuplot quedan de la siguiente manera:

```
gnuplot> reset
gnuplot> set parametric
    dummy variable is t for curves, u/v for surfaces
gnuplot> set dummy sts
gnuplot> set trange [1.0e-15:600]
gnuplot> eps(sts)=sts/E+(sts/Kp)**(1.0/np)
gnuplot> E = 206000.0
gnuplot> Kp = 1734.7
gnuplot> np = 0.2134
gnuplot> plot eps(sts), sts
```



En cualquier momento se puede saber qué funciones se han definido usando:

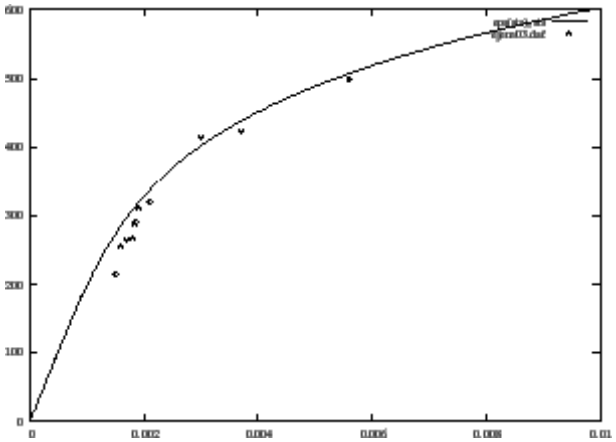
```
gnuplot> show functions

        User-Defined Functions:
        eps(sts)=sts/E+(sts/Kp)**(1.0/np)
```

También se puede saber el valor de las variables o constantes definidas:

```
gnuplot> show variables

        Variables:
        pi = 3.14159265358979
        E = 206000.0
        Kp = 1734.7
        np = 0.2134
```

Frecuentemente se deben comparar unos datos aislados con los puntos que definen alguna función. Este problema se resuelve muy bien dibujando la curva con líneas y los datos con puntos como en el siguiente ejemplo:

```
gnuplot> set xrange [0:0.01]
gnuplot> plot eps(sts), sts with lines, \
> 'ejem03.dat' with points
```



Nota: la barra "\" es para seguir una línea de comando en el renglón siguiente.

Salida de gráfico a archivo

De una manera muy simple se puede crear archivos gráficos, por ejemplo, imágenes en formato gif. El siguiente código muestra la creación de una imagen de la función seno grabada en el archivo "seno.gif".

```
graf << "set out 'seno.gif' " << endl
     << "set size 1.0, 0.5" << endl
     << "set terminal gif size 640,480" << endl
     << "plot sin(x)" << endl;
```

Cálculo numérico

Introducción

Primer ejemplo para comenzar

El ejemplo que se detalla a continuación es para realizar una introducción a los temas de cálculo numérico que más formalmente se desarrollarán en este capítulo. Se analiza la solución analítica matemática y su resolución mediante un código.

Enunciado del problema

Se está testeando una herramienta para detección de diabetes en sangre. Un primer experimento con 100 personas que tienen diabetes arroja un resultado de 85 casos de aciertos para la herramienta. Luego de mejorar la herramienta, la cual se tiene certeza que detectará el total de los casos presentados, se quiere saber cuántos exámenes más sobre los anteriores se deberían realizar para obtener un porcentaje de acierto del 90%.

Planteo del algoritmo del problema

El cálculo comienza con los 85 aciertos de la herramienta que fueron obtenidos en los 100 ensayos. Por lo tanto, en esta situación, el porcentaje de aciertos es 85/100, es decir, del 85%.

Luego de corregida la herramienta, se realiza una nueva detección. Ahora se tiene seguridad que los resultados de la existencia, o no de diabetes, es correcta.

Esta nueva detección implica el siguiente porcentaje: 86/101 =85.1485%

Análogamente al cálculo anterior, para la próxima determinación se tendrá:
87/102 =85.2941%

Y así hay que seguir calculando hasta que el resultado de 90%. El objetivo del problema es encontrar ¿cuántas detecciones más hay que realizar?.

Solución Matemática

Partiendo del planteo del algoritmo realizado anteriormente, la ecuación que generaliza el cálculo es la siguiente:

$$\frac{85 + x}{100 + x} = \frac{90}{100}$$

En esta ecuación está el resultado esperado, que consiste en encontrar el valor de x que satisface la igualdad. Despejando la única variable de la ecuación, se obtiene x=50. Es decir que para obtener un 90% de efectividad (según el planteo del problema) hay que realizar 50 pruebas más.

Solución mediante un programa

Nuevamente aquí, se parte del planteo del algoritmo del problema. En el planteo numérico del problema, se describió la evolución del cálculo en la búsqueda del objetivo, que era según se había planteado, conseguir que la división entre los aciertos y el total dé como resultado 90%.

En un programa de computación esto puede ser planteado en forma similar a como se describió anteriormente el algoritmo. El código para obtener el resultado puede ser el siguiente:

```
int main()
 {
   int x=0;
   float cuenta=85.0/100.0;

   while (cuenta < 90.0/100.0  )
   {
     cout<< " Detección: " << x << " -- Porcentaje: "<< cuenta << endl;
     x++;
     cuenta = (85.0+x) / (100.0+x);
   }
}
```

Si se ejecuta el código anterior, podrá obtener el mismo resultado obtenido con la solución matemática anterior. En este ejemplo la solución analítica es relativamente simple, pero no en todos los casos sucede así. Hay resultados que se obtienen fácilmente mediante un código o que para distintas varibles de planteo del problema, se vuelve a llamar a la función o método que obtiene el resultado, sin necesidad de una y otra vez encontrar la solución analítica. Esta última situación es ampliamente útil para los casos de simulación.

Ecuaciones No Lineales

Suponga que usted está realizando un examen y encuentra que la corriente que circulará por un paciente depende de la impedancia de su sistema en la siguiente forma:

$$I_P(z) = e^{\text{sen}(z)} - z \qquad \text{ec.1.1}$$

Su misión es evitar que el paciente esté sometido a tal circulación de corriente. Es decir, deberá encontrar el valor de z que hace IP = 0. Así obtenemos:

$$0 = e^{\text{sen}(z)} - z \qquad \text{ec.1.2}$$

Es muy probable que usted haya pensado en despejar z en la ecuación 1.2 (ec.1.2) esto no es tan trivial como parece.

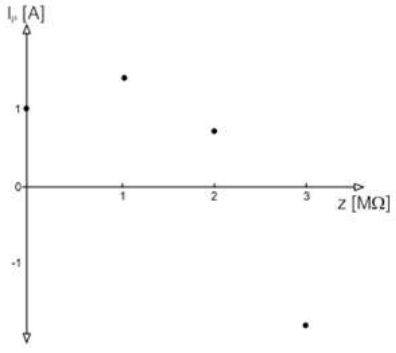


Figura 13: Primera aproximación gráfica a la solución de la ecuación del ejemplo introductorio.

A continuación se podría pensar en encontrar una *primera aproximación* a la solución. Para comenzar se sabe que las impedancias no pueden ser negativas. Por otro lado un análisis de nuestro sistema también nos podría dar el orden de magnitud que tiene el z buscado lo cual nos acota aun más la solución.

Una buena idea es graficar la función para algunos valores y observar cuando cambia de signo. Simplemente probando con la calculadora podríamos obtener la gráfica que se encuentra en la figura anterior.

Hemos encontrado que el valor de impedancia requerido está entre 2 y 3 MΩ pero podemos ver que ésta aproximación no es suficiente. Si tomásemos uno de estos valores de impedancia la corriente sería del orden del ampere y esto sería mortal para el paciente.

Bien, sabemos que el valor buscado para z está entre 2 y 3 MΩ y que debemos precisar más este resultado. Y ahora... ¿qué hacemos?.

Quizás usted pensó en evaluar la función reiteradas veces para z en el intervalo [2,3]. Esto de *probar* es una muy buena idea pero, ¿cómo vamos a seleccionar los valores de z para las reiteradas pruebas? ¿Le parece una buena alternativa probar con valores de z al azar?, recuerde que entre 2 y 3 hay infinitos valores reales. Vemos que es conveniente encontrar algunas *reglas* de prueba, alguna forma más inteligente de llevar a cabo las pruebas. Algunas ideas útiles en estos casos son: dividir el intervalo en partes iguales y buscar en cual cambia de signo la función, en el intervalo que se efectúa el cambio de signo efectuar otra subdivisión y proceder de igual forma; otra idea es partir de uno de los extremos e incrementar los valores de prueba en una cantidad fija y pequeña, en el sentido en que la función decrezca. Lo invito a que verifique estas ideas más algunas que usted proponga. Veremos con mayor detalle estas y otras formas de seleccionar los puntos de prueba más adelante.

Ahora la pregunta es ¿hasta cuándo seguimos probando?. Para el caso del ejemplo se conoce que la máxima circulación de corriente admisible en el paciente es del orden de los 10mA. Por lo tanto podemos considerar que cuando encontremos un valor de z que haga I_P<10mA habremos resulto el problema y no será necesario continuar probando. O sea estamos considerando una *tolerancia* o *error* de 10mA en el cálculo. Más adelante le daremos un significado más preciso a estos términos.

Supongamos que poseemos una computadora a nuestra disposición. Podemos hacer que ella haga todas las pruebas por nosotros. Ella podrá hacer miles de operaciones en cuestión de segundos y además podremos reducir mucho los errores por redondeo que teníamos en la calculadora. De esto se trata precisamente la resolución de ecuaciones no lineales mediante métodos numéricos. Aprenderemos a decirle a la computadora como realizar este trabajo por nosotros.

Es interesante observar en este ejemplo lo siguiente:

- Buscar los valores de la variable independiente que satisface la ecuación es el *objetivo* que perseguimos. No pierda de vista esto.
- El hecho de no poder obtener una solución analítica para la ecuación es generalmente el principal *motivo* por el cual recurrimos a los métodos numéricos para la resolución.
- Es importante que siempre hagamos una *primera aproximación* a la solución, como ejemplo mediante el análisis del problema, a través de tablas o bien con una gráfica aproximada.
- La rutina de *prueba y error* es la base de todos los métodos que veremos.
- Las distintas *reglas* mediante las cuales guiamos las pruebas son el origen de los diversos métodos de resolución.

Las metas que deberían perseguir son:

- Aprehender la estructura algorítmica común que fundamenta los métodos de resolución numérica de ecuaciones no lineales.
- Comprender los conceptos que dan origen a varios métodos numéricos sencillos para la resolución de ecuaciones no lineales.
- Conocer las condiciones de aplicabilidad de cada método.
- Valorar las ventajas y desventajas de cada método; haciendo una verificación práctica de éstas.
- Implementar los métodos analizados.

- Hacer uso de rutinas ya implementadas y disponibles en bibliotecas de procedimientos y funciones.

Características comunes a todos los métodos

Vamos a ver un algoritmo común para todos los métodos mediante el cual podremos tener una visión general de la metodología de resolución de ecuaciones no lineales con métodos numéricos.

inicialización
repetir
  aplicar la **fórmula de recurrencia**
  realizar la **redefinición de los parámetros**
  comprobar los **criterios de finalización** para evaluar
    las condiciones de finalización.
hasta verificar una **condición de finalización**

Para comenzar realizaremos una apropiada *inicialización*. En esta asignaremos a las variables a utilizar los valores con los que encontraremos la primera aproximación a la raíz.

En estos métodos encontramos que se repiten ciertos cálculos hasta que se encuentra un resultado válido o se verifica que el método no sirve para encontrarlo. De esta forma pensamos en un *ciclo* que se efectúa hasta que se cumpla una *condición de finalización.*

Dentro de este ciclo estará la regla que una vez tras otra usaremos para ir acercándonos al resultado deseado. A la expresión matemática de esta regla la denominamos *fórmula de recurrencia.*

Cada vez que apliquemos la fórmula de recurrencia deberemos hacer una actualización de las variables que toman parte en los cálculos. De esta forma se usarán en la próxima iteración. Esto será la *redefinición de los parámetros.* Veremos que en algunos casos ésta redefinición de parámetros está incluida en la fórmula de recurrencia.

Por último haremos los cálculos para poder controlar el funcionamiento del método. En base a la *verificación de criterios de finalización* preestablecidos sabremos construir la *condición de finalización*. Existen tres motivos básicos para terminar la ejecución del ciclo:

- *Se encontró el resultado buscado* : esto ocurre cuando el error o tolerancia es menor que el que especifica el usuario de la rutina.
- *Se alcanzó del límite de iteraciones* : Este control se puede llevar a cabo incrementando un contador dentro del ciclo y verificando que sea menor o igual a el límite impuesto previamente. Es de destacar que una finalización del ciclo por límite de iteraciones no implica la convergencia adecuada del método, o sea, el resultado puede no ser válido.
- *El método diverge* : en cada método podemos verificar determinadas características de la evolución que no sólo nos indicarán que no nos aproximamos al resultado buscado sino que "nos estamos alejando peligrosamente".

Es bueno puntualizar aquí dos conceptos:

- Llamamos *error* a la diferencia entre el valor encontrado para la raíz y el valor verdadero de esta.
- Llamamos *tolerancia* al máximo valor admitido para el valor absoluto de la función evaluada en el valor de la aproximación encontrada para la raíz.

En ciertos casos es posible usar como criterio de finalización ambos parámetros tanto el error máximo admisible como la tolerancia. Sin embargo veremos que en la mayor parte de los casos sólo es posible utilizar la tolerancia como criterio de finalización.

En las siguientes secciones veremos cómo son las diferentes partes de este sencillo algoritmo. Cada método nos dará una versión diferente de cada una de ellas. Lo importante que se debe rescatar de esta sección es que todos los métodos usan este algoritmo, aún más, gran parte de los métodos numéricos

hacen uso de esta estructura algorítmica básica. Tenga esto presente aún cuando esté estudiando métodos numéricos para la resolución de otros problemas.

Bisección

La idea

Este es uno de los métodos más sencillos que se conocen. Dado un intervalo que contiene una raíz, lo dividimos en dos partes iguales y tomamos para la próxima iteración el subintervalo que contenga la raíz. Posteriormente dividimos nuevamente en dos el intervalo obtenido antes y nos quedamos con el que contenga la raíz. Esto lo realizamos hasta el intervalo que nos quede sea tan pequeño que el error sea despreciable.

El método

Suponga que debemos encontrar una raíz de la función $f(x)$ que se muestra en la figura 2, en el intervalo $[x_0 , x_1]$. Partimos este intervalo en otros dos iguales, $[x_0 , x_2]$ y $[x_2 , x_1]$. Como la raíz se encuentra en el intervalo $[x_2 , x_1]$ ahora tomaremos este para realizar nuestra próxima división. En este caso encontramos el nuevo punto x_3 que nos determina los intervalos $[x_2 , x_3]$ y $[x_3 , x_1]$. Nuevamente nos quedamos con el intervalo que contiene la raíz para realizar la próxima subdivisión. Seguiremos subdividiendo hasta que obtengamos un intervalo tan pequeño que cualquiera de sus dos extremos pueda ser considerado una raíz de la función con un error despreciable a nuestros fines prácticos.

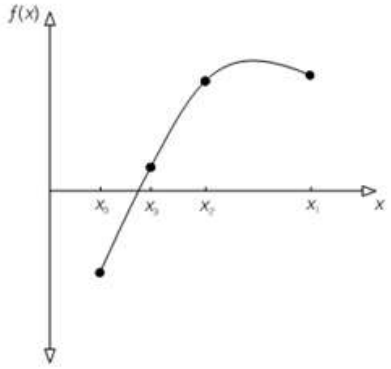


Figura 14: Aplicación del método de bisección.

Podemos realizar ciertas generalizaciones de este proceso. Para comenzar supongamos que el intervalo en el que nos encontramos es el $[x_n , x_{n+1}]$. Necesitamos calcular el punto x_{n+2} que se encuentra en el medio del intervalo. Este punto lo encontramos en:

$$x_{n+2} = \frac{x_n + x_{n+1}}{2} \qquad ec.3.1$$

Esta es la fórmula de recurrencia del método. Con esta expresión obtenemos el próximo punto de prueba del proceso.

Por otro lado debemos resolver otro inconveniente para poder realizar la próxima aproximación: ¿en cuál de los dos nuevos intervalos se encuentra la raíz? Esto se puede determinar utilizando un poco de la información que la función nos provee. Es claro que en el intervalo en que la función cambie de signo, cruzará el eje x. Una propuesta para verificar este cambio de signo en la implementación es realizar el producto de la función evaluada en los dos extremos del intervalo en cuestión. Si este producto es negativo o cero entonces es este el intervalo que contiene la raíz y por lo tanto se vuele a aplicar el cálculo. Podemos resumir esto como:

$$\text{¿}\, f(x_{n+2}) \cdot f(x_{n+1}) \leq 0\,? \quad \begin{array}{l} \text{si} \rightarrow \text{la raíz está en } [\, x_{n+1}, x_{n+2}\,] \\ \text{no} \rightarrow \text{la raíz está en } [\, x_{n}, x_{n+2}\,] \end{array}$$

Los inconvenientes

En este apartado comenzaremos con ver algunos casos en los que el método no es aplicable. Para esto voy a requerir de su colaboración. Será usted quien encontrará los inconvenientes que presenta la aplicación del método de bisección en dos casos sencillos. Las funciones que nos interesan se pueden ver en la figuras siguientes y el intervalo inicial es el [1, 3]. Usted deberá intentar aplicar las reglas anteriormente tratadas a estos casos particulares.

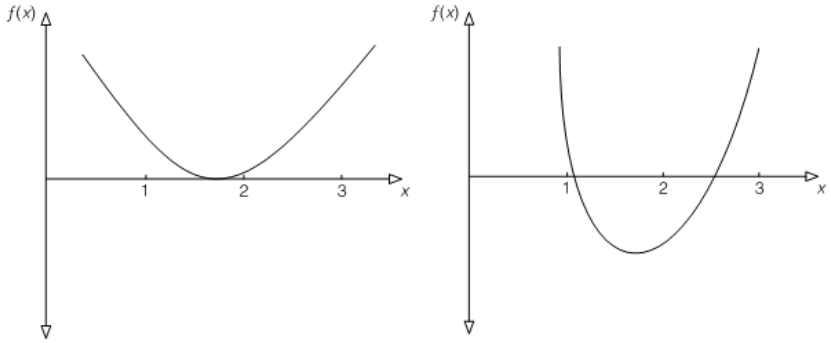


Figura 15: ¿Cuales son los inconvenientes al aplicar bisección a estas funciones en el intervalo [1,3] ?.

Si encuentra algún tipo de inconveniente trate de explicar su origen y de generalizar mediante reglas de "no aplicación del método".

Vamos a ver otros dos casos interesantes. El primero trata de funciones que en el intervalo dado presentan tres raíces. Podemos ver esta situación en la figura siguiente. Cuando intentemos aplicar el método veremos que encerraremos sólo una de las tres raíces presentes. Podemos ver en la figura los sucesivos valores que toma el x_n para n entre 0 y 4. Como usted podrá apreciar el método encontró solo una raíz de las tres existentes en el intervalo dado. Esto nos habla de la importancia de realizar un análisis previo de la función a tratar. Por otro lado veremos más adelante que esta característica, de encontrar sólo una raíz entre las muchas que pueden existir, es común a estos métodos. Le propongo que encuentre un algoritmo, basado en este método, mediante el cual podamos encontrar N raíces en un intervalo dado. Tenga en cuenta que los datos con que contará serán: intervalo inicial y cantidad de raíces a encontrar.

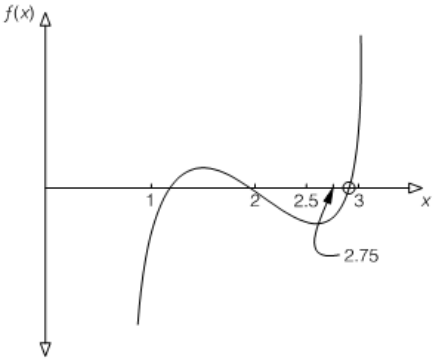


Figura 16: Evolución del método de bisección aplicado a una función con un número impar de raíces en el intervalo inicial. Los valores para x_n son 2, 2.5 y 2.75 para n igual a 2, 3 y 4 respectivamente.

El segundo caso de interés es el de la función que vemos en la figura siguiente. En este caso la función tiene una singularidad en el intervalo inicial. Le sugiero que aplique el algoritmo del método a esta función para descubrir lo que sucede... Se habrá dado cuenta que el método encuentra la singularidad como si fuese una raíz. Esto puede ser visto como una desventaja o como una ventaja en el método. Si estábamos buscando singularidades tenemos un método que las encuentra. Si estábamos buscando raíces el método está dando como resultado la ubicación de una raíz cuando en realidad en ese lugar hay una singularidad. Vemos aquí nuevamente la importancia de que hagamos un análisis previo del problema al que nos enfrentamos. Por ejemplo mediante una gráfica aproximada.

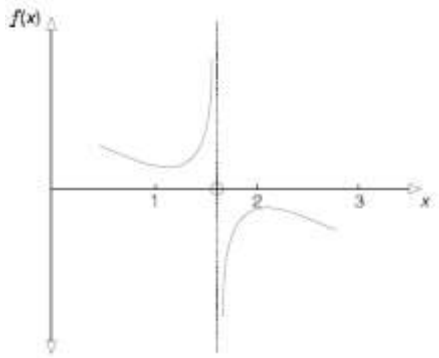


Figura 17: El caso de la detección de singularidades. ¿Una ventaja o una desventaja?.

Veamos ahora una forma interesante de estimar el error cometido después de n iteraciones. Primeramente pensemos que si tenemos un intervalo que contiene una raíz y tomamos uno de sus extremos como aproximación a la raíz.

El error máximo que podemos cometer es igual al ancho de el intervalo que contiene a la raíz seleccionado. De hecho, lo peor que nos podría suceder es que la raíz se encuentre en el otro extremo... Finalmente concluimos que el error en un intervalo dado es igual a la distancia entre sus extremos x_n - x_{n+1}.

Pero ¿podremos encontrar el error cometido después de *n* iteraciones antes haberlas realizado? Si, preste atención al siguiente razonamiento. En la primera iteración el ancho del intervalo resultante será la mitad del intervalo inicial por lo tanto:

$$\varepsilon_1 = \frac{x_0 + x_1}{2} \qquad ec.3.2$$

En la próxima iteración el ancho del intervalo será la mitad del intervalo anterior, o sea:

$$\varepsilon_2 = \frac{\frac{x_0 + x_1}{2}}{2} = \frac{x_0 + x_1}{4} = \frac{x_0 + x_1}{2^2} \qquad ec.3.3$$

Nuevamente en la iteración siguiente el ancho del intervalo será la mitad:

$$\varepsilon_3 = \frac{\frac{x_0 + x_1}{4}}{2} = \frac{x_0 + x_1}{8} = \frac{x_0 + x_1}{2^3} \qquad ec.3.4$$

Seguramente usted ya ha descubierto la regla que rige estas expresiones. El número de la iteración, subíndice del error ε, es coincidente con el exponente de la potencia que afecta al 2 en el denominador. Ahora podemos generalizar la expresión del error en la iteración *n* de la siguiente forma:

$$\varepsilon_n = \frac{x_0 + x_1}{2^n} \qquad ec.3.5$$

De esta expresión podemos despejar *n* y saber cuántas iteraciones necesitamos realizar con este método para satisfacer un error requerido dado.

Finalmente trataremos una particularidad importante de este método que, como en el caso de la detección de singularidad, puede ser vista tanto una ventaja como una desventaja. El método de bisección utiliza poca información de la función a analizar. Utilizamos sólo el signo de la función en los extremos del intervalo, no utilizamos por ejemplo su valor, el de sus derivadas u otra información que ésta nos ofrece. Decimos que esto es una ventaja ya que el método es independiente de las características de la función lo que lo hace más "robusto", aplicable con una mayor generalidad y con más probabilidades de converger hacia el resultado. Por otro lado decimos que es una desventaja por que, como veremos luego, podemos utilizar información de la función para llegar más rápidamente a la raíz buscada, es decir, acelerar el método. A continuación comenzaremos a utilizar un poco de esta información que nos brinda la función para encontrar un método que en algunos casos es más rápido.

Código

En siguiente código implementa la búsqueda de la raíz de la ecuación que se pone en la función **float** funcion**(float** x**).**

```
/* ***********************************
   CALCULO NUMERICO - Método Bisección
*************************************/

#include <iostream>
#include "math.h"
using namespace std;

float funcion(float x)
{ // Valor de la función en el punto x
  // Sus raices son 1, -2 y 3
  return( x*x*x - 2*x*x - 5*x + 6);
}

int main(void)
{
  float prod;
  float Xmedio, Xmedio_nuevo;

  float a=2;
  float b=5;

  float tolerancia=0.0001;
  float error=tolerancia*1.1; // error calculado en cada iteración.
                              // Iniciar por el while siguiente.

  int iteracion=0;
  int iteracion_max=10000;

  cout << "Iteracion - a - b - raiz - error" << endl;
  cout << "---------------------------------" << endl;

  Xmedio = ( a+b )/2;

  while ( (iteracion <= iteracion_max) && (error > tolerancia)  )
  {
  prod = funcion(a)*funcion(Xmedio);
    if (prod == 0)
      error=0;
    else
    {
     if (prod < 0)
           b = Xmedio;
     else
       a = Xmedio;

      Xmedio_nuevo = ( a+b )/2;
      iteracion=iteracion+1;
      error = fabs((Xmedio_nuevo-Xmedio)/Xmedio);
      Xmedio = Xmedio_nuevo;
      cout<<iteracion<<"  "<<a<<"  "<<b<<"  "<<Xmedio<<"  "<<error<< endl;
    }
 }
```

```
 if ( iteracion < iteracion_max )
  {
   cout<<"-----------------------------------------" << endl;
   cout<<"La Raiz de la Ecuación es: " << Xmedio << endl;
   cout<<"El valor de función en la raiz es: " << funcion(Xmedio) << endl;
   cout<<"Se encontró en " << iteracion << " Iteraciones." << endl;
  }
  else
  {
   cout<<"No se encontró raiz en " << iteracion << " Iteraciones.";
   cout<<"El valor de función en la raiz es: " << funcion(Xmedio) << endl;
  }

  cin.get();
  return 0;
}
```

Regla falsa

La idea

Y... ¿si no dividimos el intervalo en partes iguales? No sería mejor que el punto de división del intervalo estuviese más cerca de la raíz. Podemos suponer que la raíz estará más cerca del extremo donde la función alcance un menor valor. La propuesta en este caso es trazar una recta que intercepte a la función en ambos extremos del intervalo y tomar como punto de división el cruce de esta con el eje de las abscisas.

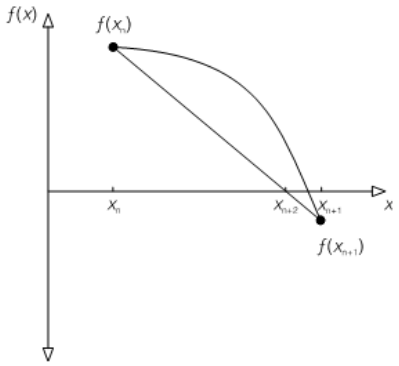


Figura 18: Aplicación del método de la regla falsa en un intervalo genérico que contiene la raíz. Obsérvese como el punto de división de intervalo se encuentra más cerca de la raíz que en el caso de bisección.

El método

En este método, también llamado de la posición falsa, lo primero que debemos hacer es encontrar la ecuación de la recta que pasa por los puntos (x_n , $f(x_n)$) y (x_{n+1} , $f(x_{n+1})$):

$$f(x) = m.x + b \qquad ec.4.1$$

donde:

$$m = \frac{f(x_{n+1}) - f(x_n)}{x_{n+1} - x_n} \qquad ec.4.2$$

y

$$b = \frac{x_{n+1}.f(x_n) - x_n.f(x_{n+1})}{x_{n+1} - x_n} \qquad ec.4.3$$

Luego buscamos la intersección de esta recta con el eje x, es decir hacemos $f(x)$=0 en la ecuación 4.1 y luego despejamos x.

$$0 = m.x + b \qquad ec.4.4$$

y luego

$$x = -\frac{b}{m} \qquad ec.4.5$$

reemplazando b y m según las ecuaciones 4.2 y 4.3 obtenemos:

$$x_{n+2} = \frac{x_{n+1}.f(x_n) - x_n.f(x_{n+1})}{f(x_{n+1}) - f(x_n)} \qquad ec.4.6$$

o bien:

$$x_{n+2} = x_{n+1} - \frac{x_{n+1} - x_n}{f(x_{n+1}) - f(x_n)} \cdot f(x_{n+1}) \qquad ec.4.7$$

Esta expresión constituye nuestra buscada fórmula de recurrencia para el método. Nótese que a pesar de que las ecuaciones 4.6 y 4.7 son equivalentes, en la ecuación 4.7 tenemos un producto y una evaluación de la función menos lo que se reflejará directamente en el tiempo de cálculo. Por otro lado la ecuación 4.7 puede ser vista como la corrección que debemos hacer a x_{n+1} para obtener x_{n+2}.

Por último para decidir cuál es el intervalo que contiene la raíz aplicaremos la misma idea utilizada en el método de bisección.

Los inconvenientes

Primeramente vemos que en general el ancho del intervalo que contiene a la raíz no tiende a cero. Esto es debido a que las funciones poseen concavidades o convexidades en las cercanías de la raíz y, mientras un extremo queda fijo, el otro avanza cada vez mas lentamente hacia la raíz.

Por otra parte: ¿Es verdad que siempre la raíz está más cerca del extremo en el que la función tomó el menor valor?. Analice la función que se encuentra en la figura siguiente. Compare la velocidad de convergencia de la regla falsa con la de bisección para este caso particular. Observe en la figura siguiente cuanto más cerca estamos de la raíz con bisección para el mismo número de iteraciones. Es evidente que hay que conocer muy bien la función a la que le aplicaremos este método. A continuación veremos como una variante del método de la regla falsa soluciona este problema.

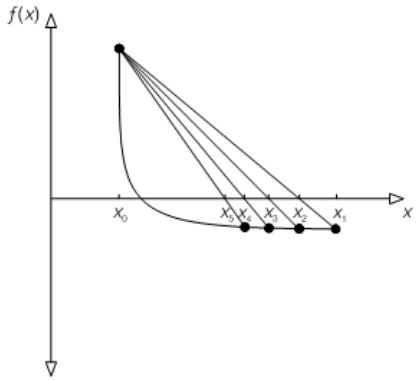


Figura 19: Cuatro iteraciones con el método de la regla falsa. Compárese con las cuatro iteraciones del método de bisección para igual intervalo y función.

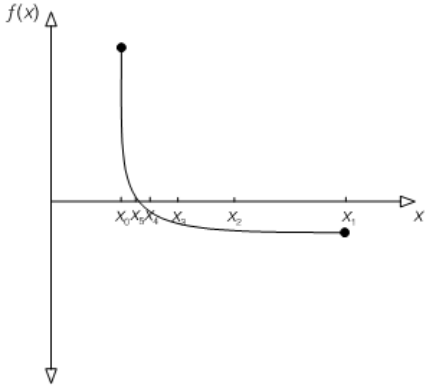


Figura 20: Cuatro iteraciones con el método de bisección.

Regla falsa modificada

La idea

Se propone construir la recta con la mitad del valor de la función en el extremo que se repita más de dos veces seguidas. Es decir, al construir la recta tomaremos el nuevo extremo, calculado en la iteración anterior, y el valor de la función en él para obtener el primer punto de la recta. El segundo punto de la recta será el otro extremo, el que se repite por segunda vez, y la mitad del valor de la función en este.

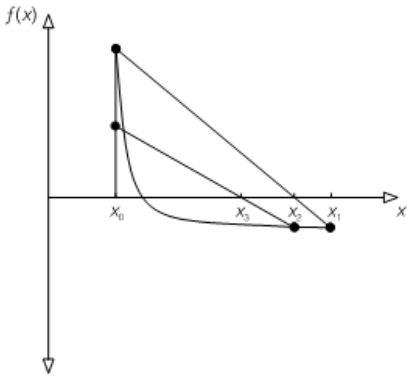


Figura 21: Modificación del método de la regla falsa para obtener una convergencia más rápida y segura.

El método

Al igual que en el método de la regla falsa debemos construir una recta y luego encontrar su intersección con el eje de las abscisas. Esta intersección será nuestro nuevo punto de división para el intervalo. Como el único cambio que hicimos fue tomar la mitad del valor de la función en el primer extremo del intervalo, podemos modificar la fórmula de recurrencia obtenida anteriormente solamente dividiendo por dos en donde aparezca $f(x_n)$, cuando este extremo se halla repetido dos veces. Existe una sola excepción a la regla: en la primera iteración no es necesario esperar más dos repeticiones para efectuar la división. Puede verse un sencillo ejemplo de esta modificación en la figura anterior. La forma en que vamos seleccionando el intervalo con el que seguiremos el análisis sigue siendo la misma que utilizamos en los métodos de bisección y regla falsa: el producto de la función evaluada en los extremos para verificar el cambio de signo.

Los inconvenientes

Luego de los cambios que realizamos sobre el método original de bisección podemos ver que seguimos teniendo los mismos problemas y restricciones que nos impone el encerrar las raíces con un intervalo. Recuerde lo que sucedía cuando teníamos un número par de raíces dentro del intervalo. O también el problema de la raíz en el mínimo o máximo local. Todos estos problemas provienen del proceso de selección del intervalo: el análisis mediante la verificación de los signos de la función en los extremos de éste.

En adelante nos olvidaremos de los intervalos que contienen a la raíz. Trataremos con uno o dos puntos ubicados cuidadosamente y con derivadas o pendientes fijas.

Secantes

La idea

Siguiendo con la recta y su intersección con el eje x, ahora cambiaremos las condiciones para la selección de los puntos que la definen. Estos ya no deben encerrar necesariamente una raíz de la función que analizamos. Trazamos una recta entre estos dos puntos iniciales y encontramos la intersección con el eje de las abscisas. Con este nuevo punto y el segundo de los anteriormente dados, trazamos la próxima recta.

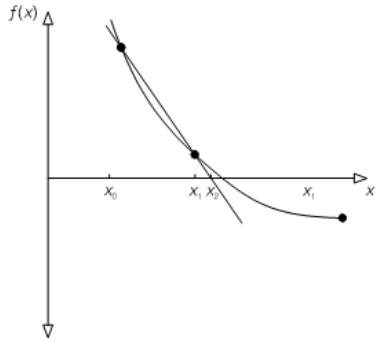


Figura 22: Primera iteración en el método de la secante. Observe como los valores iniciales dados, x_0 y x_1, no deben encerrar necesariamente a la raíz.

El método

Dados los puntos $(x_n , f(x_n))$ y $(x_{n+1} , f(x_{n+1}))$ encontraremos una recta que pase por ellos de igual forma que lo hicimos en el método de la regla falsa. Luego buscamos la intersección de esa recta con el eje de las abscisas de la forma:

$$x_{n+2} = x_{n+1} - \frac{x_{n+1} - x_n}{f(x_{n+1}) - f(x_n)} \cdot f(x_{n+1}) \qquad ec.6.1$$

Esta ecuación es exactamente igual a la ecuación 4.7 en el método de la regla falsa. A partir de ahora viene lo nuevo en este método. La próxima recta la trazaremos entre (x_{n+1} , $f(x_{n+1})$) y (x_{n+2} , $f(x_{n+2})$). Es decir, el próximo punto será una extrapolación y no una interpolación como lo era en la regla falsa. Ya no preguntamos por los signos de la función para seleccionar uno de los puntos dados, simplemente tomamos el inmediato anterior y trazamos la nueva recta. Para el caso de la figura 10 la próxima recta se trazaría por los puntos (x_1 , $f(x_1)$) y (x_2 , $f(x_2)$).

Los inconvenientes

Suponga que la función y los valores iniciales son los que se muestran en la figura siguiente. ¿Donde se encontrarán la recta y el eje x? Desde otro punto de vista: dado que $f(x_1) \cong f(x_2)$, ¿qué sucederá al evaluar la ecuación 6.1? Este es un ejemplo fehaciente de divergencia y constituye la principal desventaja del método. Como el ejemplo lo sugiere, las regiones de pendiente cercana a cero son un verdadero peligro para el algoritmo y uno deberá ser muy cuidadoso al seleccionar los dos puntos de partida.

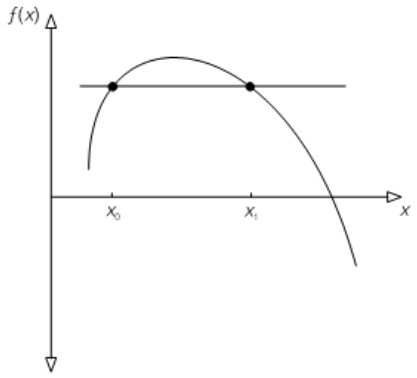


Figura 23: Una mala selección de las aproximaciones iniciales para el método de las secantes.

Otro problema es que si la distancia entre x_n y x_{n+1} es pequeña entonces $f(x_n) \cong f(x_{n+1})$ y si bien la ecuación 15 podrá ser evaluada computacionalmente hablando, los errores por redondeo pueden ser significativos. Cabe aclarar que llamamos una distancia pequeña cuando el orden de magnitud de esta es cercano el del truncamiento que presupone el tipo de datos que usamos en el caso particular. Veamos dos formas de solucionar este inconveniente.

Ante todo debemos colocar un umbral, es decir un valor β fijo que indicará la mínima distancia admisible entre $f(x_n)$ y $f(x_{n+1})$. Cuando el valor absoluto de $f(x_n)$ -$f(x_{n+1})$ sea menor que β entonces en la fórmula de recurrencia, ecuación 6.1, "saltearemos" un punto trazando la recta entre (x_{n-1} , $f(x_{n-1})$) y (x_{n+1} , $f(x_{n+1})$).

Es importante recalcar que si bien este es uno de los métodos más rápidos computacionalmente hablando, también es muy sensible a los valores iniciales, podría no converger en absoluto con una mala selección de estos.

Observe que este método hace uso de más información de la función. No sólo usa los valores que toma esta sino también hace uso de una estimación de la velocidad de cambio de la función cuando traza la secante entre dos puntos cercanos. Esto puede ser visto también como una aproximación a la pendiente o derivada de la función en la zona de trabajo.

El próximo método que veremos usa de una forma explícita la derivada de la función en el punto.

Newton - Raphson

La idea

Partiendo de un único punto trazaremos una recta que pase por éste y que tenga pendiente igual a la de la función en este punto. Luego encontramos la nueva aproximación a la raíz en la intersección de la recta y el eje de las abscisas. También podemos ver este proceso como la utilización de aproximaciones a la función en forma de series de Taylor. Seguimos el comportamiento de la función con otras aproximadas que nos llevarán poco a poco hacia la raíz.

El método

Como en las otras secciones en este apartado nos encargamos de encontrar la fórmula de recurrencia del método. El desarrollo de Taylor para la función $f(x)$ en el punto x_n se puede escribir de la siguiente forma:

$$f(x) = f(x_n) + f'(x_n) \cdot (x - x_n) + R_2 \qquad ec.7.1$$

donde R_n es el resto o residuo de segundo orden que proviene de despreciar los términos a partir del orden 2 en la serie. Vamos a dejar de lado este residuo y aceptar la aproximación de primer orden para el método.

$$g(x) = f(x_n) + f'(x_n) \cdot (x - x_n) \qquad ec.7.2$$

Encontrar la raíz de la función aproximada no es nada difícil en las condiciones en que lo planteamos:

$$0 = f(x_n) + f'(x_n) \cdot (x - x_n) \qquad ec.7.3$$

luego:

$$x = x_n - \frac{f(x_n)}{f'(x_n)} \qquad ec.7.4$$

De esta forma obtuvimos una aproximación a la raíz que puede ser generalizada para obtener la fórmula de recurrencia de la siguiente forma:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \qquad ec.7.5$$

Podemos ver la ecuación 7.2 como una recta que pasa por el punto (x_n , $f(x_n)$) y posee la misma pendiente que la función $f(x)$ en este punto. Las ecuaciones 7.3 y 7.4 constituyen la búsqueda de la intersección de la recta con el eje x. Puede observar este enfoque alternativo del método en la figura 12.

Existen dos alternativas para el costoso trabajo del cálculo de la derivada de la función. La primera consiste en tener la expresión analítica de ésta y evaluarla cuando sea necesario. La otra opción es utilizar una aproximación a ésta de la siguiente forma:

$$f'(x_n) = \frac{f(x_n + h) - f(x_n)}{h} \qquad ec.7.6$$

El valor de h debe ser elegido según la dinámica de la función y puede estar, por ejemplo, en el orden de 10^{-3}.

El método puede ser desarrollado mediante una aproximación de segundo orden pero el cálculo de las derivadas primera y segunda hace que el método converja muy lentamente.

Los inconvenientes

Este método posee un problema similar al de las secantes. Cuando nos encontremos en una región de pendiente cero o cercana a cero evaluar la ecuación 7.5 se hace computacionalmente imposible. En otras palabras la recta cortaría el eje x en el infinito (o cerca de aquel lugar). Analice las dos iteraciones que se muestran en la figura siguiente.

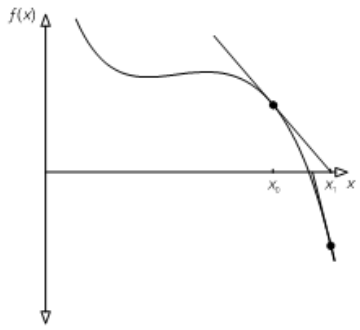


Figura 24: Aplicación del método de Newton-Raphson con un correcto punto de partida.

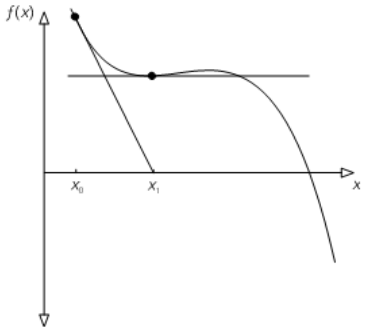


Figura 25: Divergencia en el método de Newton-Raphson

Por otro lado este método requiere más cálculos por cada iteración. El cálculo de la derivada, como ya vimos, insume demasiado tiempo siendo esta la razón fundamental por la que no es tan aplicable el método en segundo orden.

En la próxima sección veremos cómo podemos solucionar el problema de la divergencia presentado en la figura anterior.

Código

En siguiente código implementa la búsqueda de la raíz del polinomio especificado en `float` ***`funcion`***`(float x)`. Se agrega además la función correspondiente al cálculo de la derivada: `float` ***`deriva_funcion`***`(float x, float h)` que no hay necesidad de modificarla para otras funciones de búsqueda de la raiz. También se puede observar la estructura general de este programa y el de bisección, los que en términos generales siguen los criterios comentados anteriormente en "Características comunes a todos los métodos".

```
/* ************************************
   CALCULO NUMERICO - Newton Raphson
**************************************/

#include <iostream>
#include "math.h"
using namespace std;

float funcion(float x)
{ // Valor de la función en el punto x
  // Sus raices son 1, -2 y 3
  return( x*x*x - 2*x*x - 5*x + 6);
}

float deriva_funcion (float x, float h)
{ // Valor de la derivada de la funcion
  return ( (funcion (x+h) - funcion (x)) / h );
}

int main()
{
 float h=0.00001;  // intervalo para las derivadas

 float x=-100.0;   // inicio de la iteración.
 float x_nuevo;

 int iteracion=0;
 int iteracion_max=10000;

 float tolerancia=0.0001;
 float error=tolerancia*1.1; // error calculado en cada iteración.
                             // Iniciar por el while siguiente

 cout << "Iteracion - x - f(x) - error" << endl;
 cout << "----------------------------" << endl;

 while ( (iteracion <= iteracion_max) && (error > tolerancia)  )
 {
    // aplicación de la fórmula de Newton-Raphson
    x_nuevo = x - funcion (x) / deriva_funcion (x, h);

    error = fabs( x_nuevo - x );
    x=x_nuevo;
    iteracion++;

    cout << iteracion << "   " << x << "   " <<  funcion (x) << " " << error <<
endl;
 };

 cout << "=======================================" << endl;
 cout << "La raiz es: " << x << endl;
 cout << "Se obtuvo en " << iteracion << " iteraciones." << endl;
 cout << "El error fue de: " << error << endl;
 cout << "El valor de la función en la raiz da: " << funcion (x);

 cin.get();
 return 0;
}
```

Von Mises

La idea

Este método surge como una alternativa interesante para solucionar el problema de la divergencia en el método de Newton-Raphson. La propuesta es trazar todas las rectas con la misma pendiente. Esta pendiente puede ser fijada de antemano o se puede tomar la pendiente de la función en el punto de partida.

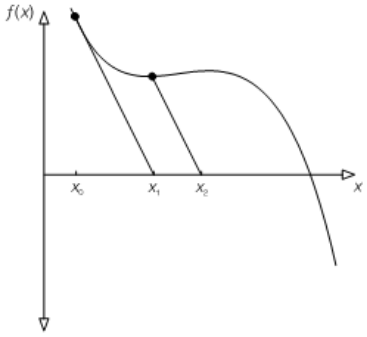


Figura 26: Aplicación del método de Von Mises al mismo caso en que el de Newton-Raphson divergía.

El método

En este caso construiremos la recta con una pendiente fija $m = f'(x_0)$ con lo que la ecuación de la recta nos quedará:

$$g(x) = f(x_n) + f'(x_0) \cdot (x - x_n) \qquad ec.8.1$$

Para encontrar la intersección con el eje x igualamos la ecuación 8.1 a cero:

$$0 = f(x_n) + f'(x_0) \cdot (x - x_n) \qquad ec.8.2$$

y luego despejamos x para obtener la fórmula de recurrencia:

$$x = x_n - \frac{f(x_n)}{f'(x_0)} \qquad ec.8.3$$

Observe en la figura anterior el resultado de dos iteraciones aplicando este método al mismo caso en el que el de Newton-Raphson divergía.

Los inconvenientes

Hemos reducido notablemente la cantidad de información que tomamos de la función lo que hace más lento al método. Por otro lado también se redujeron notablemente los cálculos que deben realizarse en cada iteración. Haciendo un balance en la mayoría de las funciones se verifica que el algoritmo en conjunto es más lento que el de Newton-Raphson. Pero no olvidemos que es más seguro por ser menos sensibles a las características de la función.

Sustituciones sucesivas

La idea

Este método da una gran flexibilidad en la selección de la fórmula de recurrencia. El esquema consiste en obtener una fórmula de recurrencia operando matemáticamente sobre la función a la que le buscamos la raíz. Una vez obtenida esta fórmula de recurrencia, que deberá satisfacer algunos requisitos, la utilizamos en el algoritmo común a todos lo métodos expuesto.

El método

Supongamos que buscamos raíces a la función $f(x)$. Por lo tanto nuestro planteo expresado mediante una ecuación:

$$f(x) = 0 \qquad ec.9.1$$

Reordenaremos esta ecuación, para obtener otra de la forma:

$$x = g(x) \qquad ec.9.2$$

De esta ecuación podemos obtener la siguiente aproximación para un algoritmo iterativo:

$$x_{n+1} = g(x_n) \qquad ec.9.3$$

Ecuación que constituye una expresión genérica para una fórmula de recurrencia. En cada iteración obtendremos un nuevo valor de x_{n+1} a partir del x_n obtenido anteriormente. Cuando se verifique que $x_{n+1} = x_n$ se verificará la ecuación 9.2 para ese valor de x. Como este x se verifica la ecuación 9.2 también verificara la ecuación 9.1, no olvidemos que ambas difieren sólo en un reordenamiento matemático. ¡Hemos encontrado la raíz que buscábamos!. Resumamos este razonamiento:

$$x_{n+1} = x_n \;\Rightarrow\; x_n = g(x_n) \;\Rightarrow\; f(x_n) = 0 \;\Rightarrow\; x_n \text{ es la raíz buscada}$$

ec.9.3 ec.9.2 ec.9.1

Un análisis algo más práctico nos lleva a la idea de considerar que la igualdad $x_{n+1} = x_n$ es poco aplicable en el ámbito computacional. En consecuencia deberemos considerar un margen de error δ de forma de considerar por satisfecha la igualdad cuando:

$$\left| x_{n+1} - x_n \right| \leq \delta \qquad ec.9.4$$

Veamos un ejemplo para aclarar el concepto. Suponga que:

$$f(x) = x - e^{\frac{1}{x}} \qquad ec.9.5$$

en este caso buscamos una x tal que satisfaga:

$$x - e^{\frac{1}{x}} = 0 \qquad ec.9.6$$

podemos reordenar esta expresión de la siguiente manera:

$$x = e^{\frac{1}{x}} \qquad ec.9.7$$

Si luego llamamos:

$$g(x) = e^{\frac{1}{x}} \qquad ec.9.8$$

obtuvimos la ecuación buscada:

$$x = g(x) = e^{\frac{1}{x}} \qquad ec.9.9$$

que iteraremos de la forma:

$$x_{n+1} = e^{\frac{1}{x_n}} \qquad ec.9.10$$

Observe que en la ecuación 9.6 podríamos haber sumado 2.x en cada miembro para obtener:

$$2x = x + e^{\frac{1}{x}} \qquad ec.9.11$$

y luego:

$$x = g(x) = \frac{x + e^{\frac{1}{x}}}{2} \qquad ec.9.12$$

que evidentemente es diferente a la ecuación 9.9 pero la obtuvimos mediante un reordenamiento matemáticamente lícito. Conclusión: podemos obtener "infinitas" expresiones diferentes para la fórmula de recurrencia. Pero... ¿son todas válidas?, aún más, ¿cuál es mejor?. En el siguiente apartado trataremos de contestar estas preguntas.

Los inconvenientes

Para comprender mejor el proceso vamos a ver una forma de representar el evolución iterativa de la ecuación 9.3. En un par de ejes coordenados graficaremos las funciones $y_A=g(x)$ e $y_B=x$. El punto en que se encuentran y_A e y_B tenemos que $g(x)=x$. Esta es precisamente la condición que pretendemos satisfacer, recordemos que cuando esto suceda se verificará $f(x)=0$. En consecuencia diremos que la intersección de las funciones $y_A=g(x)$ e $y_B=x$ representa la solución de nuestro problema. Puede todo esto representado en la figura siguiente.

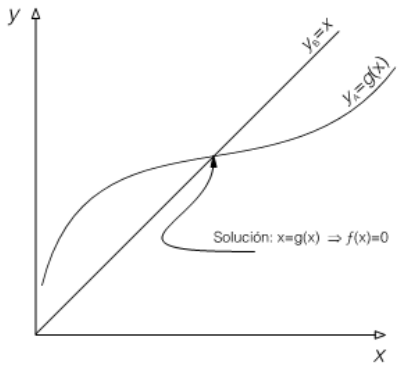


Figura 27: Representación gráfica de la fórmula de recurrencia y la solución a la ecuación no lineal.

Ahora veremos como se evoluciona el proceso iterativo. Suponga que la primera aproximación a la raíz es x_0. Lo próximo es evaluar $x_1=y_1=g(x_0)$. Este x_1 lo encontramos gráficamente trazando una recta vertical

desde el eje x en x_1 hasta encontrarnos con $g(x)$, luego trazamos otra recta horizontal hasta encontrar la recta $y=x$ que en este caso es $x_1=y_1$. Lo siguiente es usar este x_1 para encontrar $x_2=y_2=g(x_1)$.

Gráficamente significa trazar una nueva vertical hasta encontrarnos con $g(x)$ y luego una horizontal hasta llegar a la recta x=y donde tendremos $x_2=y_2$. El proceso continua aproximándose a la solución como muestra la figura siguiente.

Analicemos otras alternativas de $g(x)$. Siga los procesos mostrados en las figuras siguientes.

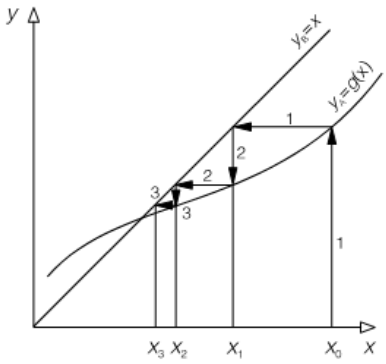


Figura 28: Aplicación del método de sustituciones sucesivas. Observe como los sucesivos valores de x se acercan a la solución.

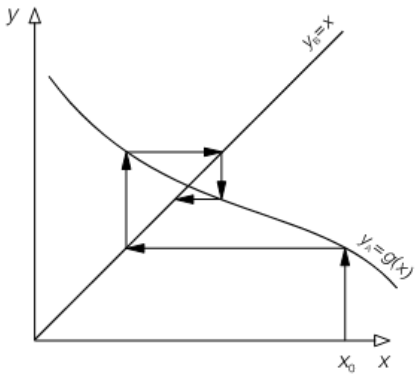


Figura 29: Otro caso en el cual los valores de x de acercan progresivamente hacia la solución.

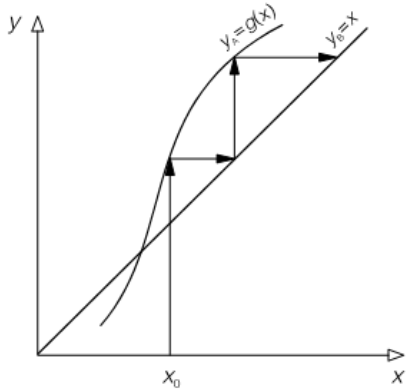


Figura 30: En este caso los valores de x se alejan progresivamente se la raíz

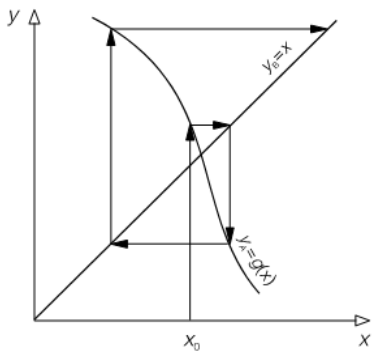


Figura 31: Aún seleccionando una aproximación inicial cercana a la solución, los siguientes valores de x se alejan cada vez más de ella.

Las dos últimas figuras nos muestran que bajo ciertas circunstancias el método es divergente. ¿Podría usted descubrir cuál es la condición que debe cumplir $g(x)$ para asegurar la convergencia del método...?

Podemos observar en la figura 28 que en un entorno de la raíz $g(x)$ posee una pendiente menor que la de la recta de pendiente 1. De forma similar, el la figura 29 la pendiente de $g(x)$ en un entorno de la solución está entre -1 y 0. Por otro lado, en las figuras 30 y 31 el valor absoluto de la pendiente de $g(x)$ es mayor a que 1.

Hemos encontrado así que la condición de convergencia para el método:

$$\textit{el método es convergente} \Leftrightarrow |g'(x)|<1 \textit{ en la vecindad de la raíz}$$

Integración Numérica

En este capítulo se verá algunos métodos básicos para realizar el cálculo de integrales por métodos numéricos.

Definiciones

Si se considera la siguiente integral

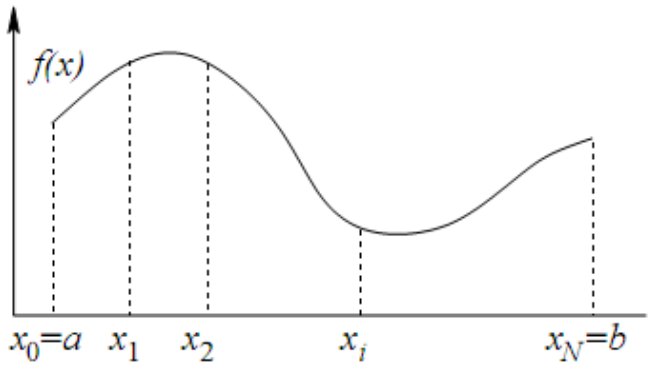
$$I = \int_a^b f(x)\,dx$$

La estrategia para estimar I es evaluar f(x) en unos cuantos puntos, y ajustar una curva simple a través de estos puntos. Primero, se realiza la subdivisión del intervalo *[a,b]* en *N* subintervalos.

Se definen los puntos xi de la siguiente manera:

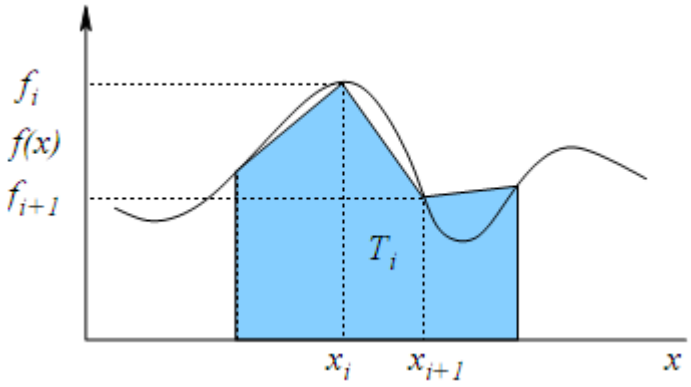
$$x_0 = a, x_N = b, x_0 < x_1 < x_2 < \cdots < x_{N-1} < x_N$$

La función f(x) sólo se evaluará en estos puntos, ver la siguiente figura



Regla trapezoidal

El método más simple es el de la regla trapezoidal. Se conectan los puntos f_i con líneas rectas como en la figura siguiente, resultando éstas líneas una nueva curva aproximante.



La integral de esta función aproximante es fácil de calcular, debido a que será la sumatoria de las áreas de trapezoides. El área de un trapezoide es

$$T_i = \frac{1}{2}(x_{i+1} - x_i)(f_{i+1} + f_i)$$

La integral es entonces calculada como la suma de las N áreas de los trapezoides:

$$I \simeq I_T = T_0 + T_1 + \cdots + T_{N-1}$$

Las expresiones quedará más simple si todos los x_i son equidistantes, por ejemplo una distancia constante h. Si la cantidad de puntos es N, los subintervalo serán N-1, el valor de h será h = (b-a) / (N-1):

Además, se tiene que $x_i = a+ih$. El área de un trapezoide se puede escribir de la siguiente manera:

$$T_i = \frac{1}{2}h(f_{i+1} + f_i)$$

La expresión del cálculo aproximado de la integral queda de la siguiente manera:

$$\begin{aligned} I_T(h) &= \frac{1}{2}hf_0 + hf_1 + hf_2 + \cdots + hf_{N-1} + \frac{1}{2}hf_N \\ &= \frac{1}{2}h(f_0 + f_N) + h\sum_{i=1}^{N-1} f_i \end{aligned}$$

Ejemplo de código

El código siguiente, ejemplifica el cálculo de la integral de la función 2.0/sqrt(M_PI))*exp(-x*x); en el rango [0, 1] para 10.000 puntos y para 5 puntos.

```
#include <iostream>
#include <vector>
#include "math.h"

using namespace std;

double funcion(double x)
{ return (2.0/sqrt(M_PI))*exp(-x*x); }


int main()
{
   double a=0, b=1,x;
   int n;
   double h;
   double I, suma=0;

   cout << "Integral función (2.0/sqrt(M_PI))*exp(-x*x)" << endl;
   cout << "Número de puntos: " ;   cin >> n;

   h = (b-a)/(n-1);

   for ( int i=2; i<n; i++)
   {
      x = a+(i-1)*h;
      suma+=funcion(x);
   }

   I = h*((funcion(a)+funcion(b))/2.0 + suma);
   cout << "Valor aproximado entre cero y 1: " << I << endl;
   cout << "Valor exacto entre cero y 1: 0.842701" << endl;
}
```

Resultados:

```
Primer ejemplo con 1000 subtintervalos:
Integral función (2.0/sqrt(M_PI))*exp(-x*x)
Número de puntos: 10000
Valor aproximado entre cero y 1: 0.842701
Valor exacto entre cero y 1: 0.842701

Primer ejemplo con 5 subtintervalos:
Integral función (2.0/sqrt(M_PI))*exp(-x*x)
Número de puntos: 5
Valor aproximado entre cero y 1: 0.838368
Valor exacto entre cero y 1: 0.842701
```

Regla de Simpson

El resultado que se conoce como regla de Simpson, utilizado para aproximar el valor de una integral definida en un intervalo *[x_0, x_2]*, cuyo punto medio es *x_1*, es el siguiente:

$$\int_{x_0}^{x_2} f(x)dx = \frac{h}{3}[f(x_0)+4f(x_1)+f(x_2)]-\frac{h^5}{90}f^{(4)}(\xi)$$

Donde

$$h = x_2 - x_1 = x_1 - x_0 \ ,$$

ξ es algún número en el intervalo de integración, el cual no conocemos, pero que nos puede servir para poder estimar una cota máxima del término de error, el cual es proporcional a la quinta potencia de h y a la cuarta derivada f(4)(ξ). Para estimar el término de error necesitaríamos conocer alguna cota para el valor de la cuarta derivada de la función f en el intervalo total de integración.

Una manera de obtener la regla de Simpson es integrar en el intervalo [x0, x2] el segundo polinomio de Lagrange, con los nodos $x_0 = a$, $x_2 = b$ y $x_1 = a + h$, donde $h = (a + b)/2$.

Recordar que el segundo polinomio de Lagrange es de segundo grado.

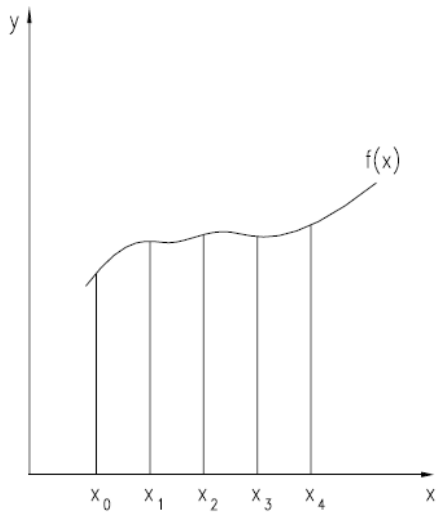
Otra manera de derivar la regla de Simpson es expandir la función f en la forma del polinomio de grado 3 de Taylor, alrededor del punto medio del intervalo, x_1.

Se omite la deducción de la regla de Simpson, pudiéndose obtener de la amplia bibliografía cita en internet.

Cuando el intervalo de integración es grande, para lograr una mejor aproximación al valor de la integral, lo dividimos en un número par de subintervalos. Por ejemplo, supongamos que dividimos el intervalo total de integración [a, b] en dos pares de subintervalos. Con $x_0 = a$ y $x_4 = b$, el primer par de subintervalos será, por ejemplo, [x_0, x_1] y [x_1, x_2], y el segundo, [x_2, x_3] y [x_3, x_4].

Aplicando la regla de Simpson a los dos pares de subintervalos, para completar el intervalo total de integración, nuestra aproximación al valor de la integral será, despreciando el término del error,

$$\int_{x_0}^{x_4} f(x)dx \approx \frac{h}{3}[f(x_0)+4f(x_1)+2f(x_2)+4f(x_3)+f(x_4)]$$



División del intervalo [x_0, x_4] en dos pares de subintervalos.

En caso de dividir del intervalo de integración en tres pares de subintervalos, la aplicación de la regla de Simpson nos conduciría a la siguiente aproximación:

$$\int_{x_0}^{x_6} f(x)dx \approx \frac{h}{3}[f(x_0)+4f(x_1)+2f(x_2)+4f(x_3)+2f(x_4)+4f(x_5)+f(x_6)]$$

Regla de tres octavos de Simpson

Si en vez de dividir nuestro intervalo total de integración en pares de subintervalos, decidiéramos utilizar tríadas de subintervalos, tendríamos que aplicar la regla de tres octavos de Simpson, Para cada tríada,

$$\int_{x_0}^{x_3} f(x)dx = \frac{3h}{8}[f(x_0)+3f(x_1)+3f(x_2)+f(x_3)]+\frac{3h^5}{80}f^{(4)}(\xi)$$

donde el término de error es directamente proporcional a la cuarta derivada de algún número ξ, del cual sólo sabemos que $x_0 < \xi < x_3$.

Para el caso de tener dos tríadas, nuestra aproximación sería

$$\int_{x_0}^{x_6} f(x)dx \approx \frac{3h}{8}[f(x_0)+3f(x_1)+3f(x_2)+2f(x_3)+3f(x_4)+3f(x_5)+f(x_6)]$$

Ejemplo de código

El código siguiente, ejemplifica el cálculo de la integral de la función 2.0/sqrt(M_PI))*exp(-x*x); en el rango [0, 1] calculada también con el método de los trapecios.

```
#include <iostream>
#include <vector>
#include "math.h"

using namespace std;

double funcion(double x)
{  return (2.0/sqrt(M_PI))*exp(-x*x);  }

int main()
{
  double a=0, b=1;
  int n;
  double h;
  double I;

  cout << "Integral función (2.0/sqrt(M_PI))*exp(-x*x)" << endl;

   // fórmula de simpson 5 términos:
   n=5;
   h = (b-a)/(n-1);
   I =  h/3*(funcion(a)+4*funcion(a+h)+2*funcion(a+2*h)
          +4*funcion(a+3*h)+funcion(a+4*h));
   cout << "Valor aproximado entre cero y 1 con 5 términos: " << I << endl;

   // fórmula de simpson 7 términos:
   n=7;
   h = (b-a)/(n-1);
   I =  h/3*(funcion(a)+4*funcion(a+h)+2*funcion(a+2*h)
          +4*funcion(a+3*h)+2*funcion(a+4*h)+4*funcion(a+5*h)+funcion(a+6*h));
   cout << "Valor aproximado entre cero y 1 con 7 términos: " << I << endl;

   // fórmula de simpson - regla 3/8:
   n=7;
   h = (b-a)/(n-1);
   I =  3.0/8.0*h*(funcion(a)+3*funcion(a+h)+3*funcion(a+2*h)+2*funcion(a+3*h)
              +3*funcion(a+4*h)+3*funcion(a+5*h)+funcion(a+6*h));
   cout << "Valor aproximado entre cero y 1 con regla 3/8: " << I << endl;

   cout << "Valor exacto entre cero y 1: 0.842701" << endl;
}
```

Resultados:

```
Integral función (2.0/sqrt(M_PI))*exp(-x*x)
Valor aproximado entre cero y 1 con 5 términos: 0.842736
Valor aproximado entre cero y 1 con 7 términos: 0.842708
Valor aproximado entre cero y 1 con regla 3/8: 0.842717
Valor exacto entre cero y 1: 0.842701
```

Regresión lineal simple

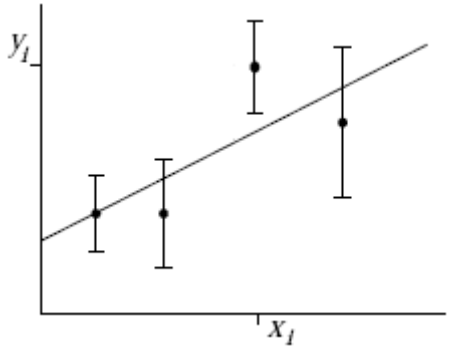
Hay ocasiones en que se requiere analizar la relación entre dos variables cuantitativas. Los dos objetivos fundamentales de este análisis serán, por un lado, determinar si las variables están asociadas y por otro, estudiar si los valores de una variable pueden ser utilizados para predecir el valor de la otra.

Se analiza el caso más sencillo en el que se considera únicamente la relación entre dos variables y se busca la modelización a través de una recta.

La recta de regresión

Consideremos una variable dependiente ***Y*** relacionada con otra variable que llamaremos independiente ***X***. A partir de una muestra de n pares de datos para los que se dispone de los valores de ambas variables, $\{(X_i,Y_i), i = 1,...n\}$, se puede visualizar gráficamente la relación existente entre ambas mediante un gráfico de dispersión, en el que los valores de la variable X se disponen en el eje horizontal y los de Y en el vertical.

Se busca encontrar una recta que ajuste todos estos puntos, y que pueda ser utilizada para predecir los valores de Y a partir de los de X. La ecuación general de la recta de regresión será entonces de la forma: $Y = a + bX$.



El problema radica en encontrar aquella recta que mejor ajuste a los datos. Tradicionalmente se ha recurrido al método de mínimos cuadrados, que elige como recta de regresión a aquella que minimiza las distancias verticales de las observaciones a la recta.

Por lo tanto, el cálculo consiste en encontrar los valores de a y b de la recta. En bibliografías específicas de Estadística, se podrá encontrar las deducciones de las siguientes fórmulas que dan estos dos valores:

$$\hat{b} = \frac{\sum_{i=1}^{n}(X_i - \overline{X})(Y_i - \overline{Y})}{\sum_{i=1}^{n}(X_i - \overline{X})^2} = \frac{s_{xy}}{s_{xx}} \quad ; \quad \hat{a} = \overline{Y} - \hat{b}\overline{X}$$

Ejemplo de cálculo

La siguiente muestra los datos de 69 pacientes de los que se conoce su edad y una medición de su tensión sistólica. Si estamos interesados en estudiar la variación en la tensión sistólica en función de la edad del individuo, deberemos considerar como variable dependiente la tensión y como variable independiente a la edad.

N	Tensión	Edad	N	Tensión	Edad	N	Tensión	Edad	N	Tensión	Edad
1	114	17	19	150	38	36	156	47	53	170	59
2	134	18	20	120	39	37	159	47	54	185	60
3	124	19	21	144	39	38	130	48	55	154	61
4	128	19	22	153	40	39	157	48	56	169	61
5	116	20	23	134	41	40	142	50	57	172	62
6	120	21	24	152	41	41	144	50	58	144	63
7	138	21	25	158	41	42	160	51	59	162	64
8	130	22	26	124	42	43	174	51	60	158	65
9	139	23	27	128	42	44	156	52	61	162	65
10	125	25	28	138	42	45	158	53	62	176	65
11	132	26	29	142	44	46	174	55	63	176	66
12	130	29	30	160	44	47	150	56	64	158	67
13	140	33	31	135	45	48	154	56	65	170	67
14	144	33	32	138	45	49	165	56	66	172	68
15	110	34	33	142	46	50	164	57	67	184	68
16	148	35	34	145	47	51	168	57	68	175	69
17	124	36	35	149	47	52	140	59	69	180	70
18	136	36									

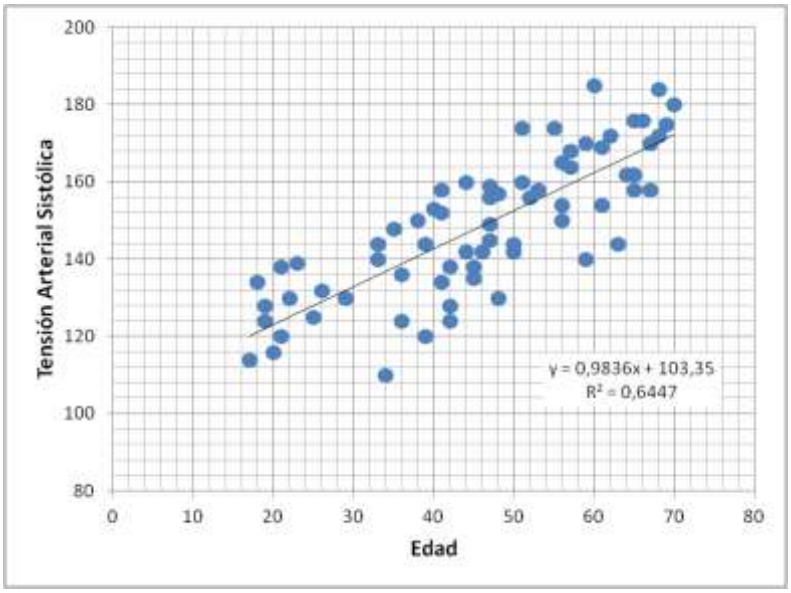
Aplicando los cálculos anteriores a este caso, resultaría:

$$\overline{X} = 46{,}13 \qquad S_{xx} = \sum_{i=}^{n} X_i^2 - \left(\sum_{i=1}^{n} X_i\right)^2 / n = 15470$$

$$\overline{Y} = 148{,}72 \qquad S_{xy} = \sum_{i=1}^{n} X_i Y_i - \left(\sum_{i=1}^{n} X_i \sum_{i=1}^{n} Y_i\right) / n = 15215 \Rightarrow \quad \hat{b} = 0.98 \Rightarrow \quad \hat{a} = 103.35$$

En el gráfico siguiente se muestran los datos y la recta de regresión de mínimos cuadrados correspondientes.

La relación $Y = a + bX$ no va a cumplirse exactamente, sino que existirá un error e que representa la variación de Y en todos los datos con un mismo valor de la variable independiente.



Coeficiente de correlación

El coeficiente de correlación indica el grado de dependencia entre las variables *x* e *y*. El coeficiente de correlación *r* es un número que se obtiene mediante la fórmula:

$$r = \frac{n\left(\Sigma x_i y_i\right) - \left(\Sigma x_i\right)\left(\Sigma y_i\right)}{\sqrt{\left[n\left(\Sigma x_i^2\right) - \Sigma\left(x_i\right)^2\right]\left[n\left(\Sigma y_i^2\right) - \left(\Sigma y_i\right)^2\right]}}$$

Su valor puede variar entre 1 y -1.
Si r = -1 todos los puntos se encuentran sobre la recta existiendo una correlación que es perfecta e inversa.
Si r = 0 no existe ninguna relación entre las variables.
Si r = 1 todos los puntos se encuentran sobre la recta existiendo una correlación que es perfecta y directa.

Ejemplo de código

Los valores del ejemplo anterior con la tensión sistólica y la edad se encentran almacenados en un archivo de texto. El código siguiente ejemplifica los cálculos para obtener los parámetros *a* y *b* de la recta, como así también el coeficiente de correlación *r*.

```
#include <string>
#include <iostream>
#include <fstream>
#include <vector>
#include <math.h>   // por sqrt

using namespace std;

vector <float> x;
vector <float> y;
float x_prom, y_prom;

void cargar_datos(string archi)
{ float i, j;
  ifstream datos (archi.c_str());
  while (datos>>i)
    { datos >> j;
      x.push_back(i);
      y.push_back(j);
    }
  datos.close();
}

void promedios ()
{ float sumaX=0.0;  for (uint i=0; i<x.size(); i++)  sumaX=sumaX+x[i];
  float sumaY=0.0; for (uint i=0; i<y.size(); i++)  sumaY=sumaY+y[i];
  x_prom=sumaX/x.size(); y_prom=sumaY/y.size();
}

float suma_difX_por_difX()  // Sum (Xi-Xprom)*(Xi-Xprom)
{  float suma=0.0;
   for (uint i=0; i<x.size(); i++)
     suma=suma+((x[i]-x_prom)*(x[i]-x_prom));
   return suma;
}
```

```
float suma_difX_por_difY()  //  Sum (Xi-Xprom)*(Yi-Yprom)
{  float suma=0.0;
   for (uint i=0; i<x.size(); i++)
     suma=suma+( (x[i]-x_prom)*(y[i]-y_prom));
   return suma;
}

float suma_X()  // Sum (Xi)
{  float suma=0.0;
   for (uint i=0; i<x.size(); i++)  suma=suma+x[i];
   return suma;
}

float suma_Y()  // Sum (Yi)
{  float suma=0.0;
   for (uint i=0; i<y.size(); i++)  suma=suma+y[i];
   return suma;
}

float suma_X_por_Y()  //  Sum (Xi)*(Yi)
{  float suma=0.0;
   for (uint i=0; i<x.size(); i++)  suma=suma+(x[i] * y[i]);
   return suma;
}

float suma_X_por_X()  //  Sum (Xi)*(Xi)
{  float suma=0.0;
   for (uint i=0; i<x.size(); i++)  suma=suma+(x[i] * x[i]);
   return suma;
}

float suma_Y_por_Y()  //  Sum (Yi)*(Yi)
{  float suma=0.0;
   for (uint i=0; i<y.size(); i++)  suma=suma+(y[i] * y[i]);
   return suma;
}

void Calcular_parametros (float &a, float &b, float &r, string archi)
{  cargar_datos(archi);
   promedios();
   b=suma_difX_por_difY() / suma_difX_por_difX();
   a=y_prom - b* x_prom;

   float numerador_r = x.size()*suma_X_por_Y() - suma_X()*suma_Y()  ;
   float denominador_r = sqrt ( (x.size()*suma_X_por_X()-
     suma_X()*suma_X())*(y.size()*suma_Y_por_Y()-suma_Y()*suma_Y()));
   r = numerador_r/denominador_r;
}


int main()
{  float a, b, r;

    Calcular_parametros (a, b, r, "datos.txt");

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "r = " << r << endl;
    cout << "r2= " << r*r << endl;
```

```
}
```

La ejecución de este código, dará como resultado lo siguiente:

```
a = 103.353
b = 0.983559
r = 0.802952
r2= 0.644732
```

Para este tipo de código, donde principalmente se realizan multiplicaciones, la function *inner_product()* (se debe agregar #include <numeric>) de la STL permite una simplificación.

Si por ejemplo se tiene:

```
vector <int> a;
a.push_back(1); a.push_back(2); a.push_back(3);
vector <int> b;  b=a;

cout << inner_product (a.begin(), a.end(), b.begin(), 0);
```

en forma simbólica inner_product es equivalente a realizar un bucle con *acc=acc+(*i1) * (*i2)* es decir (una "sumatoria" de los "productos")

en este caso el cálculo es:

```
0+(1*1)+(2*2)+(3*3)=0+1+4+9=14
```

Estas operaciones de "suma de los productos" pueden ser cambiadas. El siguiente código ejemplifica este uso.

```
int mi_sumador(int i, int j) {return (i-j);}
int mi_producto(int i, int j) {return (i+j);}
cout << inner_product (a.begin(), a.end(), b.begin(), 0,
                       mi_sumador,mi_producto);
```

en este nuevo ejemplo se tiene que el acumulador es *acc = binary_op1 (acc binary_op2((*i1,*i2))* donde la "sumatoria" se reemplaza por *binary_op1*, y los "productos" por *binary_op2*.

Entonces en vez de "productos" se tendrán sumas, y el "sumador" realizará restas, obteniendose el siguiente resultado:

```
0-(1+1)-(2+2)-(3+3) = 0-2-4-6 = -12
```

En vez de pasar la dirección a las funciones *mi_sumador()* y *mi_producto()*, también existe la posibilidad de pasar un objeto que tenga sobrecargado el operador "()"

El nuevo ejemplo de código de cálculo de *a*, *b* y *r*, utilizando *inner_product()* es el siguiente:

```
using namespace std;

vector <float> x;
vector <float> y;
float x_prom, y_prom;

// funciones para usar en inner_product( )
float mi_sumador (float i, float j) {return (i+j);}
```

```
//  para Sum (Xi)*(Yi), Sum (Xi)*(Xi) y Sum (Yi)*(Yi)
float mi_producto (float i, float j) {return (i*j);}

// para  Sum (Xi-Xprom)*(Xi-Xprom)
float mi_productoX_prom (float i, float j)
{ return ( (i-x_prom)*(j-x_prom) ); }

// para  Sum (Xi-Xprom)*(Yi-Yprom)
float mi_productoXY_prom (float i, float j)
{ return ( (i-x_prom)*(j-y_prom) ); }


void cargar_datos(string archi)
{ float i, j;
  ifstream datos (archi.c_str());
  while (datos>>i)
   { datos >> j;
     x.push_back(i);
     y.push_back(j);
   }
  datos.close();
}

void promedios ()
{  x_prom=accumulate (x.begin(), x.end(),0.0)/x.size();
   y_prom=accumulate (y.begin(), y.end(),0.0)/y.size();
}

float suma_difX_por_difX()  // Sum (Xi-Xprom)*(Xi-Xprom)
{ return (inner_product(x.begin(),x.end(),
          x.begin(),0.0,mi_sumador,mi_productoX_prom)); }

float suma_difX_por_difY()  //  Sum (Xi-Xprom)*(Yi-Yprom)
{ return (inner_product(x.begin(),
          x.end(),y.begin(),0.0,mi_sumador,mi_productoXY_prom));}

float suma_X()  // Sum (Xi)
{ return (accumulate (x.begin(), x.end(),0.0)); }

float suma_Y()  // Sum (Yi)
{ return (accumulate (y.begin(), y.end(),0.0)); }

float suma_X_por_Y()  //  Sum (Xi)*(Yi)
{ return (inner_product(x.begin(),
          x.end(),y.begin(),0.0,mi_sumador,mi_producto)); }

float suma_X_por_X()  //  Sum (Xi)*(Xi)
{ return (inner_product(x.begin(),
          x.end(),x.begin(),0.0,mi_sumador,mi_producto)); }

float suma_Y_por_Y()  //  Sum (Yi)*(Yi)
{ return (inner_product(y.begin(),
          y.end(),y.begin(),0.0,mi_sumador,mi_producto)); }

void Calcular_parametros (float &a, float &b, float &r, string archi)
{
  cargar_datos(archi);
  promedios();
  b=suma_difX_por_difY() / suma_difX_por_difX();
  a=y_prom - b* x_prom;
```

```
  float numerador_r = x.size()*suma_X_por_Y() - suma_X()*suma_Y();
  float denominador_r= sqrt (  (x.size()*suma_X_por_X()-
                       suma_X()*suma_X())*(y.size()*suma_Y_por_Y()-
                       suma_Y()*suma_Y())  );
  r=   numerador_r/denominador_r;
}

int main()
{    float a, b, r;

      Calcular_parametros (a, b, r, "datos.txt");

      cout << "a = " << a << endl;
      cout << "b = " << b << endl;
      cout << "r = " << r << endl;
      cout << "r2= " << r*r << endl;
}
```

La ejecución de este código, dará como resultado lo siguiente:

```
a = 103.353
b = 0.983559
r = 0.802952
r2= 0.644732
```

ANEXOS

Introducción a HTML

Esta guía ha sido creada para dar una visión inicial en la creación de documentos en HTML (HyperText Markup Language). Es una forma fácil de publicar resultados de la ejecución de programas en la World Wide Web[2].

Directivas

Las *directiva* (en inglés *tags*) en HTML proporcionan información adicional al cliente que visualiza el documento. Un documento sin ninguna directiva no resaltará ninguna parte del texto. Éstas son códigos especiales que están contenidos en un documento y se usan caracteres especiales para delimitarlas. Por ejemplo, la directiva para iniciar un párrafo es: `<P>`

Por lo general, se agrupan por pares: uno inicia la acción, mientras su par la finaliza. La directiva para finalizar el párrafo es: `</P>`. Como se puede ver, el par de final de acción es el mismo que el de inicio incluyendo el caracter /

HTML no es sensible a mayúsculas; esto es, la directiva puede escribirse como `<P>` o `<p>`. Los caracteres < y > son códigos reservados. Si se quieren mostrar dichos caracteres dentro del texto, se debe utilizar el concepto de entidades que se verá en el siguiente punto.

Caracteres especiales

Son especificados utilizando el siguiente formato: al inicio de la entidad se marca con el caracter "&", mientras que el final de la entidad se marca con el caracter ";". Entre los dos caracteres citados se especificará el nombre de una entidad.

Los más comunes son:

Carater	Código
<	&lt;
>	&gt;
&	&amp;
á	&aacute;
Á	&Aacute;
ñ	&ntilde;
Ñ	&Ntilde;

[2] De internet se puede obtener una gran variedad de documentación referida a la codificación de archivos en formato html. Este capítulo ha sido desarrollado a partir del publicado en http://www.eis.uva.es/GuiaHTML/introHTML.html

Estructura de un documento HTML

Los documentos HTML se dividen en las siguientes partes como se puede ver en el siguiente código:

```
<HTML>
  <HEAD>
  </HEAD>

  <BODY>
  </BODY>
</HTML>
```

La primera línea indica que es un documento HTML (no visible).	<HTML>
La segunda y tercera línea delimitan la cabecera (no visible).	<HEAD> </HEAD>
La cuarta y quinta línea delimitan la parte principal, o cuerpo, y es la que el usuario ve.	<BODY> </BODY>
La última línea indica que finaliza el documento HTML.	</HTML>

Un documento inicial

Título

No es visible dentro de la página web . Se hará dentro de la directiva HEAD y utilizando la directiva <TITLE>...</TITLE>

```
...
  <HEAD>
    <TITLE>Introduccion a HTML</TITLE>
  <HEAD>
...
```

Cabeceras

Las cabeceras permiten dividir lógicamente el documento en secciones, subsecciones, etc. HTML prevee hasta seis niveles de cabeceras.

Las cabeceras aparecen solamente en el cuerpo del documento. Para especificar un texto cabecera de nivel requerido se puede escribir:

```
...
<H1>Cabecera de nivel 1</H1>       Más general
<H2>Cabecera de nivel 2</H2>           .
<H3>Cabecera de nivel 3</H3>           .
...                                    .
<H6>Cabecera de nivel 6</H6>       Más específico
```

El usuario verá:

Cabecera de nivel 1
Cabecera de nivel 2
Cabecera de nivel 3
Cabecera de nivel 6

Párrafos

La mayor parte del texto que se escribe forma parte de un párrafo de texto. Para crear un párrafo, basta con escribir el texto que lo forma dentro de la directiva <P>...</P>. Además, en numerosas ocasiones es necesario forzar un final de línea en el documento formateado (un simple retorno de carro no es suficiente). Para ello, es suficiente con el uso de la directiva simple
 (no necesita la directiva homóloga de cierre)

```
...
<P>Este es mi primer párrafo.
   aunque escriba este texto en otra linea,
   seguirá perteneciendo al mismo párrafo.</P>
<P>Este es un segundo párrafo que origina una separación.
   Luego sigue en la<BR>
   otra línea.</P>
```

El usuario verá:

Este es mi primer párrafo. aunque escriba este texto en otra línea, seguirá perteneciendo al mismo párrafo.

Este es un segundo párrafo que origina una separación. Luego sigue en la
otra línea.

Texto con formato

Dentro de un documento HTML se puede indicar que un texto tenga un estilo o tipografía especial. Se utiliza <B> para negrita y <I> para italica. El siguiente ejemplo, muestra una forma de enfatizar texto:

Por ejemplo:

```
<P>A continuación <B>negrita</B> y luego <I>italica</I>.</P>
```

Se verá:

A continuación **negrita** y luego *italica*.

Para centrar un texto (u otro objeto, imagen, tabla, ...) se utiliza la directiva
<CENTER> ... </CENTER>

Por ejemplo:

```
<CENTER>Esta línea la hemos centrado </CENTER>
```

Se verá:

Esta línea la hemos centrado

Divisiones horizontales

Las divisiones horizontales son usadas para formatear un documento en secciones.

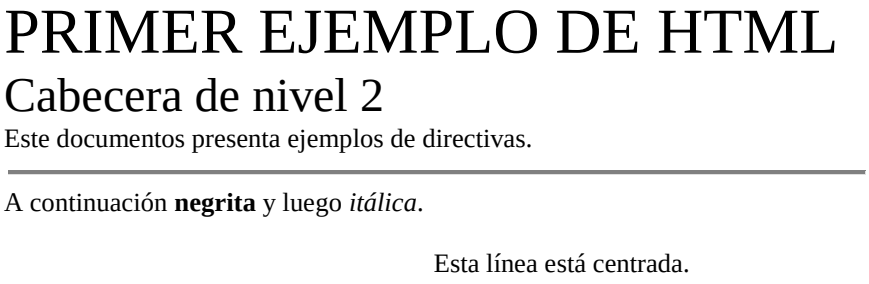
Se debe evitar el uso de imágenes para crear el efecto de divisiones horizontales porque pueden no verse correctamente. La inserción de una división horizontal es muy simple. Se realizará insertando la directiva <HR> en el lugar donde se desee que aparezca la línea horizontal (esta directiva no necesita su homóloga de cierre).

Primer ejemplo completo

El siguiente listado es un ejemplo completo de lo visto hasta aquí. Copie el siguiente código a un archivo de texto y grabándolo con la extensión "html" se podrá ver con un navegador. Desde un programa en C, puede generar estas líneas a un archivo de texto.

```
<HTML>

  <HEAD>
    <TITLE>Código del primer ejemplo</TITLE>
  <HEAD>

  <BODY>
    <H1>PRIMER EJEMPLO DE HTML</H1>
    <H2>Cabecera de nivel 2</H2>

      <P>Este documentos presenta ejemplos de directivas.</P>
      <HR>
      <P>A continuación <B>negrita</B> y luego <I>itálica</I>.</P>

      <CENTER>Esta línea está centrada. </CENTER>
  </BODY>
</HTML>
```

Se verá:



Pruebe modificarlo para observar los efectos producidos.

Listas

Una lista para HTML no es más que una agrupación o enumeración de elementos de información. Estas listas pueden anidarse.

Los miembros de la mesa de trabajo son:

- Sr. Chema Pamundi
- Sr. German Tequilla
- Sr. Carmelo Coton

Para hacer una lista como la anterior, se debe utilizar la directiva `<UL>...</UL>`; mientras que, para cada elemento de la lista debe utilizarse la directiva `<LI>...</LI>`:

```
<UL>
<LI>Sr. Chema Pamundi</LI>
<LI>Sr. German Tequilla</LI>
<LI>Sr. Carmelo Coton</LI>
</UL>
```

Para que la lista se vea enumerada, en vez de la directiva `<UL>...</UL>` utilizar `<OL>...</OL>`.

Imágenes

Una de las funcionalidades más llamativas en HTML es la posibilidad de incluir imágenes dentro de un documento. Algunos formatos gráficos tienen soporte en modo nativo (son visualizados directamente por el navegador), mientras que otros requieren del concurso de programas externos.

No todos los archivos que contienen gráficos siguen la misma convención de almacenamiento. Existen varios formatos que permiten, entre otras cosas, comprimir en distinto grado la información. Los formatos más extendidos son: GIF (Graphics Interchange Format), JPEG (Joint Photographic Experts Group bitmap) y sus variantes (JPG, BMP, MP, XBM), TIFF (Tagged Image File Format), EPS (Encapsulated PostScript), o PCX (de Paintbrush).

Solo el formato GIF es soportado directamente por todos los visualizadores, mientras que el JPEG lo es por la mayoría. El formato GIF se basa en el sistema de compresión LZW, un algoritmo muy simple y que, por ello, no alcanza unas altas cotas de reducción. Este formato trabaja con un máximo de 256 colores (8 bits); para simular colores fuera de la paleta utiliza la técnica de dithering (aproximación del color por los vecinos más próximos).

El formato JPEG utiliza un algoritmo de compresión muchos más complicado que el utilizado por el GIF: los archivos resultantes son más pequeños, pero necesitan más tiempo para su descompresión. A diferencia del anterior formato, JPEG trabaja con 16.7 millones de colores.

Como norma general, diremos que se utilizará el formato GIF para iconos e imágenes pequeñas y JPEG para imágenes grandes o de calidad.

Para insertar una imagen en un documento HTML se utilizará la directiva simple

```
<IMG>
<IMG src="/icons/network.gif">
```

El usuario verá una imagen:

Tablas

Al igual que las listas, las tablas son componentes dedicados a mejorar la visualización de datos tabulados.

Las tablas se especificarán siempre por filas; es decir, primero se escribirá la fila 1, despues la fila 2, etc. La directiva que se utiliza para delimitar una tabla es `<TABLE>...</TABLE>`.

Cada fila se especifica con la directiva `<TR>...</TR>` y, dentro de ella, cada celda se especifica con la directiva `<TD>...</TD>`

La presencia de bordes en la tabla se especifica con el atributo *border* en la directiva `<TABLE>`. Con ello se logrará un borde de dimensión la unidad; si deseamos hacer el borde más grueso deberemos dar un valor numérico al atributo: *border=espesor*.

El título de la tabla es una cadena decaracteres delimitado por la directiva `<CAPTION>...</CAPTION>`.

Por último, cada cabecera de columna se especifica con la directiva `<TH>...</TH>`

Las directivas TR, TD y TH admiten dos atributos de centrado: VALIGN para el centrado vertical y ALIGN para el horizontal; donde los valores que pueden tomar son, TOP (superior), BOTTOM (inferior), MIDDLE (centrado vertical), RIGHT (derecha), LEFT (izquierda) y CENTER (centrado horizontal). La directiva TD admite WIDTH=ancho de la columna.

Por ejemplo:

```
<TABLE border>
<CAPTION> Ejemplo de tabla</CAPTION>
<TR><TH>Primera columna</TH><TH>Segunda columna</TH><TH>Tercera columna</TH></TR>
<TR><TD WIDTH=215>100,3</TD><TD>1,8</TD><TD>313,1</TD></TR>
<TR ALIGN=RIGHT><TD>22,7</TD><TD>200,8</TD><TD>23,1</TD></TR>
<TR ALIGN=CENTER><TD>8100,3</TD><TD>1300,5</TD><TD>4100,1</TD></TR>
</TABLE>
```

El usuario verá:

Ejemplo de tabla

Primera columna	Segunda columna	Tercera columna
100,3	1,8	313,1
22,7	200,8	23,1
8100,3	1300,5	4100,1

Colores

Además de las imágenes, el color hace a las páginas más vistosas. Para especificar colores dentro de una página, se utiliza el código RGB (rojo-verde-azul) con el cual podemos especificar distintas tonalidades de colores. Para especificar un color se utiliza el caracter # seguido de un valor hexadecimal de 6 dígitos; los dos primeros para el rojo, los dos centrales para el verde y los dos finales para el azul.

Podemos ver algunos ejemplos:

Código	#FF0000	#0000FF	#00FF00	#FFFF00	#9933CC	#FFFFFF	#666666	#000000
Color	**Rojo**	**Azul**	**Verde**	**Amarillo**	**Morado**	**Blanco**	**Gris**	**Negro**

Existen tres características a las que podemos aplicar color: el fondo de un documento, al texto de un documento y a texto específico dentro del documento:

Lugar	Ejemplos	Resultado
Fondo de un documento	`<BODY BGCOLOR=#9933CC>`	Fondo de color morado
Texto de un documento	`<BODY TEXT=#00FF00>`	Texto de color verde
Texto específico	`<FONT COLOR=#FF0000>`	Texto en rojo

Segundo ejemplo completo

Ahora se ejemplifica un código en C++ que genera un archivo html.

```
#include<iostream>
#include<fstream>
#include <iomanip>

int main() {
ofstream salida("salida.html");

salida << "<HTML>" << endl;
salida << "<BODY>" << endl;
salida << "<H1>FUNCION SENO</H1>" << endl;
salida << "<HR>" << endl;

salida << "<TABLE border>" << endl;
salida << "<CAPTION> <B>Resultado</B></CAPTION>" << endl;
salida << "<TR><TH>X</TH><TH>Seno (X)</TH></TR>" << endl;

float x = 0;
while (x < 3.14) {
  salida << "<TR TR ALIGN=CENTER><TD WIDTH=100>" << x
         << "</TD><TD WIDTH=215>" << setprecision(4)
         << sin(x) << "</TD></TR>" << endl;
  x = x + 0.15;
}

salida << "</TABLE>" << endl;
salida << "</BODY>" << endl;
salida << "</HTML>" << endl;

salida.close();
}
```

Directivas del preprocesador

Las directivas del preprocesador son órdenes que se incluyen dentro del código de los programas que no son instrucciones para el programa en sí, sino que son para el preprocesador. El preprocesador es ejecutado automáticamente por el compilador cuando se compila un programa en C++ y está a cargo de realizar las primeras verificaciones del código del programa.

Cada una de estas directivas debe ser especificada en una sola línea de código y no deben incluir punto y coma ; al final.

Constantes definidas (#define) – (obsoleto)

Se pueden definir nombres propios para constantes que se usan a menudo sin tener que recurrir a variables, simplemente usando la directiva de preprocesador **#define**. Éste es su formato:

#define *identificador valor*

Por ejemplo:

```
#define PI 3.14159265
#define NUEVALINEA '\n'
#define ANCHO 100
```

definen tres nuevas constantes. Observar que no se dice de que tipo de datos son. Por este motivo, se recomienda **no** utilizar esta forma de declarar constantes y utilizar "const" como se muestra a continuación. Una vez declaradas, se pueden usar en el resto del código como si fueran cualquier otra constante, por ejemplo:

```
circulo = 2 * PI * r;
cout << NUEVALINEA;
```

De hecho lo único que el compilador hace cuando encuentra la directiva #define es reemplazar literalmente cualquier ocurrencia de ellos (en el ejemplo anterior, PI, NUEVALINEA o ANCHO) por el código que los define (3.14159265, ' \n ' y 100, respectivamente). Por esta razón, las constantes definidas con #define son consideradas *constantes macro*.

La directiva #define no es una instrucción de código, es una directiva para el preprocesador, razón por la cual asume la línea entera como directiva y no se requiere un punto y coma (;) al final. Si se incluye un punto y coma (;) al final, también se agregará cuando el preprocesador sustituya cualquier ocurrencia de la constante definida dentro del cuerpo del programa.

#undef

#undef realiza la función inversa que *#define*. Lo que hace es eliminar de la lista de constantes definidas la que posea el nombre pasado como parámetro a #undef:

```
#define ANCHO_MAX 100
char str1[ANCHO_MAX];
#undef ANCHO_MAX
#define ANCHO_MAX 200
char str2[ANCHO_MAX];
```

#ifdef, #ifndef, #if, #endif, #else y #elif

Estas directivas permiten descargar parte del código de un programa si una cierta condición no es cumplida.

#ifdef - #ifndef

#ifdef

Permite que una sección de un programa sea compilada sólo si la constante definida que se especifica como parámetro ha sido definida, independientemente de su valor. Su uso es:

```
#ifdef nombre
// código
#endif
```

Por ejemplo:

```
#ifdef ANCHO_MAX
char str[ANCHO_MAX];
#endif
```

En este caso, la línea char str[ANCHO_MAX]; es sólo considerada por el compilador si la constante definida ANCHO_MAX ha sido previamente definida, independientemente de su valor. Si no ha sido definida, el código encerrado no será incluido en el programa.

#ifndef

Rearealiza lo opuesto: el código entre la directiva **#ifndef** y la directiva **#endif** es compilado sólo si el nombre constante que ha sido especificado no ha sido definido. Por ejemplo:

```
#ifndef ANCHO_MAX
#define ANCHO_MAX 100
#endif
char str[ANCHO_MAX];
```

En este caso, si al llegar a esta parte del código la *constante definida* ANCHO_MAX no ha sido definida, será definida con un valor de 100. Si ya existe, mantendrá el valor que actualmente posee (debido a que la directiva #define no sería ejecutada).

Las directivas #if, #else y #elif (elif = else if) sirven para que la porción de código que las sigue sea compilada sólo si se cumple la condición específica. La condición únicamente sirve para evaluar expresiones constantes. Por ejemplo:

```
#if ANCHO_MAX>200
#undef ANCHO_MAX
#define ANCHO_MAX 200

#elsif ANCHO_MAX<50
#undef ANCHO_MAX
#define ANCHO_MAX 50

#else
#undef ANCHO_MAX
```

```
#define ANCHO_MAX 100
#endif

char str[ANCHO_MAX];
```

observar como la estructura de directivas encadenadas #if, #elsif y #else terminan con #endif.

#line

Cuando se compila un programa y ocurre un error durante el proceso de compilación, el compilador muestra el error que ha ocurrido precedido por el nombre del archivo y la línea dentro de ese archivo donde el error ha ocurrido.

La directiva *#line* permite controlar ambas cosas, los números de línea dentro de los archivos y el nombre del archivo que se quiere que aparezca cuando un error ocurre. Se utiliza de la siguiente manera:

#line *numero* "*nombre_archivo*"

donde *numero* es el nuevo número de línea que será asignado a la siguiente línea de código. El número de línea de la líneas sucesivas será incrementado en uno a partir de ésta.

Nombre_archivo es un parámetro opcional que sirve para reemplazar el nombre de archivo que será mostrado en caso de error hasta que otra directiva lo cambie nuevamente o hasta llegar al final del archivo. Por ejemplo:

```
#line 1 "asignando variables"
int a?;
```

este código generará un error que será mostrado como error en el archivo "asignando variables", línea 1.

#error

Esta directiva interrumpe el proceso de compilación cuando es encontrada devolviendo el error especificado como parámetro:

```
#ifndef __constanteDeError
#error Se requiere compilador C++
#endif
```

Este ejemplo interrumpe el proceso de compilación si la constante definida *__constanteDeError* no está definida.

#include

Esta directiva también ha sido usada asiduamente en otras secciones. Cuando el preprocesador encuentra una directiva *#include* la reemplaza por el contenido total del archivo especificado. Existen dos maneras de especificar un archivo para ser incluido:

#include "archivo"
#include <*archivo***>**

La única diferencia entre ambas expresiones es en las carpetas en que el compilador va a buscar los archivos. En el primer caso, en el que el archivo es especificado entre comillas, el compilador buscará por los archivos en al misma carpeta en la que se encuentra el archivo que se está utilizando, y sólo en caso

de que no esté allí el compilador buscará en las carpetas por defecto (donde se encuentran los archivos de encabezamiento estándar).

En caso que se encierre el nombre del archivo entre signos de **<>** se busca al archivo directamente en las carpetas por defecto.

#pragma

Esta directiva se usa para especificar diversas opciones para el compilador. Estas opciones son específicas de la plataforma y el compilador que se está utilizando. Se deberá consultar el manual o la referencia del compilador para más información acerca de los posibles parámetros a definir con #*pragma*.

Bases numéricas

Estamos acostumbrados a usar números decimales para expresar cantidades. Esta nomenclatura que parece tan lógica puede no serlo para un romano de la Roma clásica. Para ellos cada símbolo que ponían para expresar un número siempre representaba el mismo valor:

```
I       1
II      2
III     3
IV      4
V       5
```

Si se presta atención todos los signos I siempre representan el valor 1 (uno) dondequiera que se sitúen, como el signo V siempre representa nuestro 5 (cinco). Sin embargo esto no ocurre en nuestro sistema decimal. Cuando escribimos el símbolo decimal 1 no siempre estamos hablando acerca del valor 1 (I en números romanos).

Por ejemplo:

```
  1     I
 10     X
100      C
```

En estos casos nuestro símbolo 1 no siempre posee el valor 1. Por ejemplo, en el segundo caso, el símbolo 1 representa el valor 10 y en el tercero, 1 representa el valor 100.

Otro ejemplo:

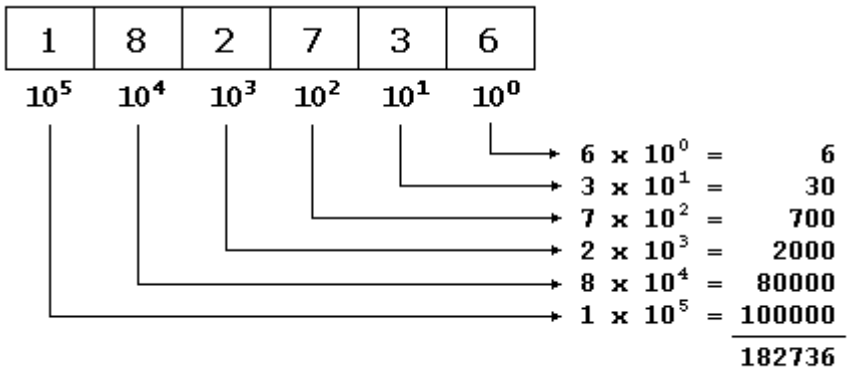
275

no es equivalente a 2+7+5, sino que puede ser descompuesto como 200+70+5:

```
 200
+ 70
   5
 ---
 275
```

por lo tanto, el primer símbolo 2 es equivalente a 200 (2 x 100), el segundo símbolo, 7 es equivalente a 70 (7 x 10) mientras que el último signo corresponde al valor 5 (5 x 1).

Toda la introducción previa puede ser representada matemáticamente en una forma muy simple. Por ejemplo, para representar el valor 182736 podemos asumir que cada dígito es el producto de sí mismo multiplicado por 10 elevado a la potencia que corresponde a su ubicación, comenzando desde la derecha con 10^0, siguiendo con 10^1, 10^2, y así:



Números octales (base 8)

Como nuestros números "normales" son en base 10 porque tenemos 10 dígitos diferentes (del 0 al 9):

```
0123456789
```

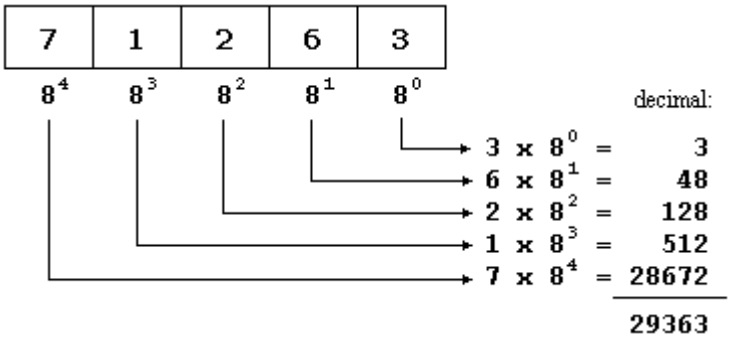
los números octales incluyen sólo dígitos del 0 al 7:

```
01234567
```

y por lo tanto, su base matemática es 8. En C++ los números octales tienen la peculiaridad de que siempre comienzan con un dígito 0. Veamos como escribiríamos los primeros números en base octal:

octal	decimal	
0	0	*(cero)*
01	1	*(uno)*
02	2	*(dos)*
03	3	*(tres)*
04	4	*(cuatro)*
05	5	*(cinco)*
06	6	*(seis)*
07	7	*(siete)*
010	8	*(ocho)*
011	9	*(nueve)*
012	10	*(diez)*
013	11	*(once)*
014	12	*(doce)*
015	13	*(trece)*
016	14	*(catorce)*
017	15	*(quince)*
020	16	*(dieciseis)*
021	17	*(diecisiete)*

podemos aplicar el esquema que vimos previamente con los números decimales simplemente considerando a 8 como base. Por ejemplo tomando el número octal 071263:



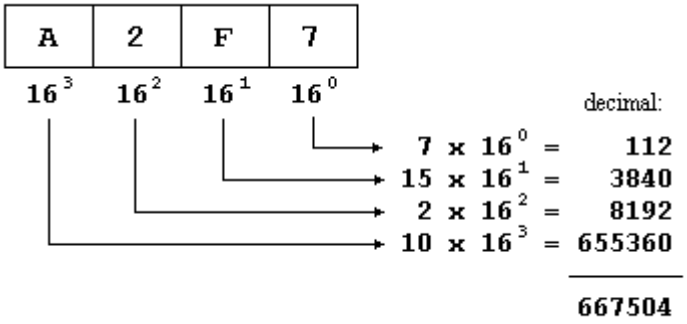
por lo tanto el número octal 071263 se expresa como 29363 en números decimales.

Números hexadecimales (base 16)

Como los números decimales tienen 10 distintos dígitos para representarse (0123456789) y los números octales tienen 8 (01234567), los números hexadecimales tiene 16: números del 0 al 9 y las letras A, B, C, D, E y F que juntos sirven para representar los 16 diferentes símbolos que necesitamos:

hexadecimal	decimal	
0	0	*(cero)*
0x1	1	*(uno)*
0x2	2	*(dos)*
0x3	3	*(tres)*
0x4	4	*(cuatro)*
0x5	5	*(cinco)*
0x6	6	*(seis)*
0x7	7	*(siete)*
0x8	8	*(ocho)*
0x9	9	*(nueve)*
0xA	10	*(diez)*
0xB	11	*(once)*
0xC	12	*(doce)*
0xD	13	*(trece)*
0xE	14	*(catorce)*
0xF	15	*(quince)*
0x10	16	*(dieciseis)*
0x11	17	*(diecisiete)*

Una vez más podemos usar el mismo método para pasar un número de una base a otra:



Representaciones binarias

Los números en base octal y hexadecimal tienen una considerable ventaja sobre los números en base decimal en el mundo de los bits, y es que sus bases (8 y 16) son potencias de 2 (2^3 y 2^4) lo que permite hacer conversiones más fáciles a números binarios que desde números decimales (cuya base es 2x5). Por ejemplo, si quisiéramos pasar la siguiente secuencia binaria a otras bases:

```
110011111010010100
```

para pasar esta secuencia a un número decimal necesitaríamos realizar una operación matemática similar a las hechas anteriormente para convertir desde hexadecimal o octal, lo cual nos daría el número decimal 212628.

Sin embargo, para pasar esta secuencia a un número octal sólo nos llevaría algunos segundos y lo podemos hacer sólo mirando: dado que 8 es 2^3, separaremos el valor binario en grupos de 3 números:

```
110 011 111 010 010 100
```

y ahora sólo tenemos que traducir a un número en base octal cada grupo por separado:

```
110 011 111 010 010 100
 6   3   7   2   2   4
```

resultando el número 637224. el mismo proceso puede realizarse a la inversa para pasar desde octal a binario.

Para realizar esta operación con números hexadecimales debemos realizar el mismo proceso pero separando el valor binario en grupos de 4 números ($16 = 2^4$):

```
11 0011 1110 1001 0100
3    3    E    9    4
```

por lo tanto, la expresión binaria 110011111010010100 puede ser representada en C++ como 212628 (decimal), 0637224 (octal) o 0x33e94 (hexadecimal).

El código hexadecimal es especialmente interesante en computación dado que hoy en día las computadoras están basadas en bytes compuestos de 8 bits binarios y por lo tanto cada byte es igual al rango que 2 números hexadecimales pueden representar. Por esta razón es el tipo más usado al representar valores traducidos desde binario

Código ASCII

Como probablemente sabrás, al nivel más bajo las computadoras sólo manejan 0s (ceros) y 1s (unos). Usando secuencias de 0s y 1s una computadora puede manejar números en formato binario. Sin embargo no existe una forma evidente para representar letras con 0s y 1s. Para este propósito se usa el código ASCII (*Código Standard Americano para Intercambio de Información*).

El código ASCII es una tabla o lista que contiene todas las letras del alfabeto más una variedad de caracteres adicionales. En este código, cada caracter se representa con un número, que siempre es el mismo. Por ejemplo, el código ASCII para representar la letra A es siempre representada por el número 65, que es fácilmente representable usando 0s y 1s en notación binaria (1000001).

El código ASCII standard define 128 códigos de caracteres (de 0 a 127). Los primeros 32 son códigos de control (no imprimibles), y los otros 96 son caracteres representables

*	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	TAB	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	A	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	Q	r	s	t	u	v	w	x	y	z	{	|	}	~	

* Este panel está organizado para ser leído fácilmente en hexadecimal: los número de las filas representan el primer dígito y los números de las columnas representan el segundo dígito. Por ejemplo, el caracter A se localiza en la 4º fila y la 1º columna, así que sería representado como un número hexadecimal como 0x41 (65).

Además de los 128 códigos ASCII standard (los listados arriba en el rango de 0 a 127), muchas máquinas tienen otros 128 códigos extra cuyo formato se conoce como código ASCII extendido (en el rango de 129 a 255). Este conjunto de caracteres ASCII extendido depende de la plataforma, lo que significa que puede variar de una máquina a otra, o entre sistemas operativos.

Los conjuntos de caracteres extendidos ASCII más usados son OEM y ANSI.

El conjunto de caracteres OEM se incluye en todas las computadoras PC-compatibles como el conjunto de caracteres por defecto cuando el sistema bootea antes de cargar cualquier sistema operativo y bajo MS-DOS. Incluye algunos signos extranjeros, algunos caracteres acentuados y también piezas para dibujar paneles simples. Desafortunadamente es redefinido usualmente para configuraciones regionales específicas para incluir símbolos locales.

OEM Extended ASCII

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	Ç	ü	é	â	ä	à	å	ç	ê	ë	è	ï	î	ì	Ä	Å
9	É	æ	Æ	ô	ö	ò	û	ù	ÿ	Ö	Ü	¢	£	¥	₧	ƒ
A	á	í	ó	ú	ñ	Ñ	ª	º	¿	⌐	¬	½	¼	¡	«	»
B	░	▒	▓	│	┤	╡	╢	╖	╕	╣	║	╗	╝	╜	╛	┐
C	└	┴	┬	├	─	┼	╞	╟	╚	╔	╩	╦	╠	═	╬	╧
D	╨	╤	╥	╙	╘	╒	╓	╫	╪	┘	┌	█	▄	▌	▐	▀
E	α	ß	Γ	π	Σ	σ	µ	τ	Φ	Θ	Ω	δ	∞	ø	∈	∩
F	≡	±	≥	≤	⌠	⌡	÷	≈	°	∙	·	√	ⁿ	²	■	

cplusplus.com

El conjunto de caracteres estándars de ANSI lo incorporan los sistemas como Windows, algunas plataformas UNIX y varias otras aplicaciones. Incluye otros simbolos y letras acentuada que pueden ser usadas sin necesidad de ser redefinidas en otros lenguajes:

ANSI Extended ASCII (Windows)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	□	□	‚	ƒ	„	…	†	‡	ˆ	‰	Š	‹	Œ	□	□	□
9	□	‘	’	“	”	•	–	—	˜	™	š	›	œ	□	□	Ÿ
A		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
B	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
C	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

cplusplus.com

Lógica booleana

Operaciones AND, OR, XOR y NOT

Un bit es la mínima cantidad de información que podemos imaginar, dado que sólo almacena valores de 1 o 0, que representan SI o NO, activado o desactivado, verdadero o falso, etc… esto es: dos estados posibles, cada uno opuesto al otro, sin posibilidad de intermedios.

Muchas operaciones pueden ser realizadas con bits, tanto en conjunto con otros bits o sin ellos. Estas operaciones reciben el nombre de *operaciones booleanas***,** una palabra que viene del nombre de uno de los matemáticos que más contribuyó en este campo:: George Boole (1815-1864).

Todas estas operaciones tienen un comportamiento predeterminado y todas ellas pueden ser aplicadas a cualquier bit cualquiera sea el valor que contenga (**0** o **1**). A continuación tienes una lista de las operaciones booleanas básicas y una tabla con el comportamiento de esa operación con todas las posibles combinaciones de bits.

AND (`&`)

Esta operación se realiza entre dos bits distintos (a y b). El resultado es 1 si ambos a y b son iguales a 1, si alguno de ellos es igual a 0 el resultado es 0

a	b	a&b
0	0	0
0	1	0
1	0	0
1	1	1

OR (`|`)

Esta operación se realiza entre dos bits distintos (a y b). El resultado es 1 si alguno de ellos a o b es 1. Si ninguno es 1 el resultado es 0

a	b	a|b
0	0	0
0	1	1
1	0	1
1	1	1

XOR (Or exclusivo: ^)

Esta operación se realiza entre dos bits distintos (a y b). El resultado es 1 si alguno de ellos a o b es 1, excepto si ambos son 1. Si ninguno de los dos o ambos son iguales a 1 el resultado es 0

a	b	a^b
0	0	0
0	1	1
1	0	1
1	1	0

NOT (~)

Esta operación se realiza en un sólo bit. Su resultado es la inversión del valor actual del bit: si contenía 1 pasa a contener 0 y si contenía 0 pasa a contener 1

a	~a
0	1
1	0

Estas son las cuatro operaciones básicas (**AND**, **OR**, **XOR** y **NOT**). Combinando estas operaciones cualquier resultado deseado puede ser obtenido.

En C++, estas operaciones pueden ser usadas entre dos variables de cualquier tipo entero; la operación lógica se realiza entre los bits de las dos variables. Por ejemplo, suponiendo dos variables: a y b, ambas de tipo char, a conteniendo 195 (11000011 en binario) y b conteniendo 87 (o 01010111 en binario). Si escribimos el siguiente código:

```
char a = 195;
char b = 87;
char c;
c = a&b;
```

Lo que hicimos fue una operación de bits **AND** entre **a** y **b**. El resultado (el contenido de **c**) sería el siguiente:

```
a :   1 1 0 0 0 0 1 1
b :   0 1 0 1 0 1 1 1  &
      -----------------
c :   0 1 0 0 0 0 1 1
```

01000011, que es 67.

Archivos de encabezamiento estándar

En ANSI-C++ la forma de incluir archivos de encabezamiento desde la biblioteca standard ha cambiado.

El standard especifica la siguiente modificación a la manera C de incluir archivos de encabezamiento standard:

Los archivos de encabezamiento no mantienen la extensión .h típica del lenguaje C y de los compiladores C++ previos al standard, como en el caso de stdio.h, stdlib.h, iostream.h, etc. Esta extensión .h simplemente desaparece y los archivos antes conocidos como iostream.h se convierten en iostream (sin .h).

Los archivos de encabezamiento que vienen del lenguaje C ahora deben ser precedidos por un caracter c para distinguirlos de los nuevos archivos de encabezamiento exclusivos de C++ que tienen el mismo nombre. Por ejemplo stdio.h se convierte en cstdio, o stdlib.h en cstdlib.

Todas las clases y funciones definidas en las bibliotecas standard están bajo el namespace **std** en vez de ser globales. Estos no se aplican a las macros C que permanecen igual.

La siguiente lista corresponde a los archivos de encabezamiento standard de C++:

```
<algorithm>   <bitset>   <deque>   <exception>   <fstream>   <functional>
<iomanip>   <ios>   <iosfwd>   <iostream>   <istream>   <iterator>   <limits>
<list>   <locale>   <map>   <memory>   <new>   <numeric>   <ostream>   <queue>
<set>   <sstream>   <stack>   <stdexcept>   <streambuf>   <string>   <typeinfo>
<utility>   <valarray>   <vector>
```

Y aquí hay una lista de los archivos de encabezamiento standard de C incluidos en el ANSI-C++ con el nuevo nombre y sus equivalentes en ANSI-C:

ANSI-C++	ANSI-C
<cassert>	<assert.h>
<cctype>	<ctype.h>
<cerrno>	<errno.h>
<cfloat>	<float.h>
<ciso646>	<iso646.h>
<climits>	<limits.h>
<clocale>	<locale.h>
<cmath>	<math.h>
<csetjmp>	<setjmp.h>
<csignal>	<signal.h>
<cstdarg>	<stdarg.h>
<cstddef>	<stddef.h>
<cstdio>	<stdio.h>
<cstdlib>	<stdlib.h>
<cstring>	<string.h>
<ctime>	<time.h>

<cwchar>	<wchar.h>
<cwtype>	<wtype.h>

Desde ahora las clases y funciones de las bibliotecas se localizan dentro del namespace std y debemos usar la directiva C++ using para que estos se vuelvan útiles en la misma manera que lo eran antes. Por ejemplo, para poder utilizar todas las clases standard de iostream tendríamos que incluir algo similar a esto:

```
#include <iostream>
using namespace std;
```

que sería equivalente a la vieja expresión

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

previa al estándard.

Sin embargo para compatibilidad con ANSI-C, el uso de la forma name.h para incluir archivos de encabezamiento C está permitida. Por lo tanto, los siguientes ejemplos son válidos y equivalentes en un compilador que cumple con las especificaciones ANSI-C++:

```
// Ejemplo ANSI C++

#include <cstdio>
using namespace std;
int main ()
{
  printf ("Hola Mundo!");
  return (0);
}
```

```
// Ejemplo pre-ANSI C++
// también válido bajo ANSI C++

#include <stdio.h>
int main ()
{
  printf ("Hola Mundo!");
  return (0);
}
```

Comentarios sobre los Entornos de Desarrollo

Este anexo presenta comentarios breves sobre el compilador y entornos de desarrollos más usados en la cátedra.

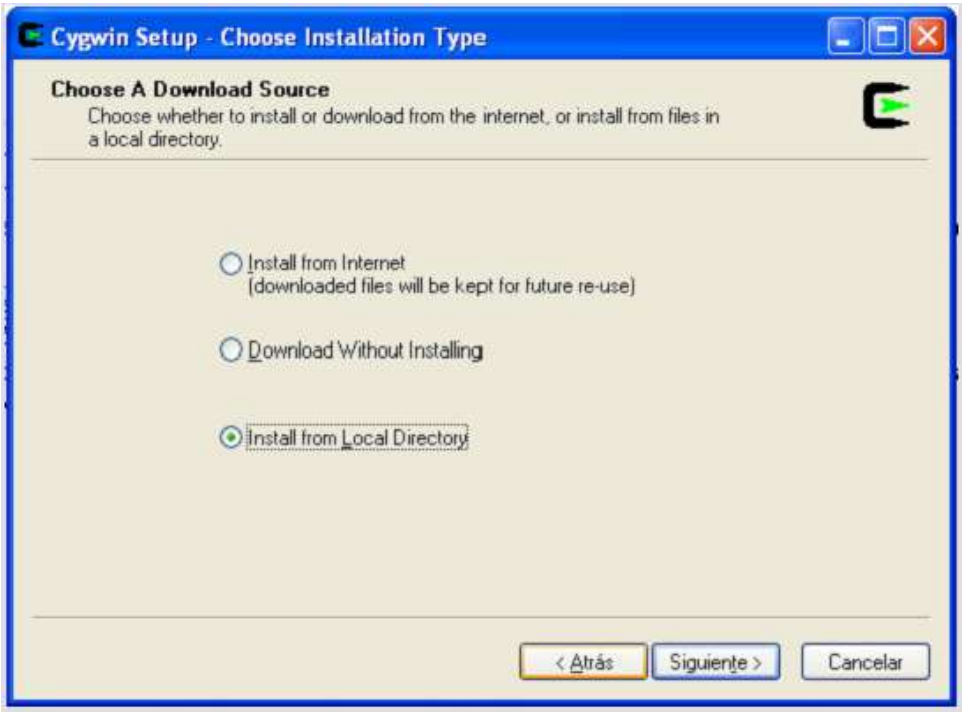
Instructivo instalación Cygwin v1.7.4

El compilador de c++ utilizado por la cátedra es cygwin. Se descarga del sitio www.cygwin.com.

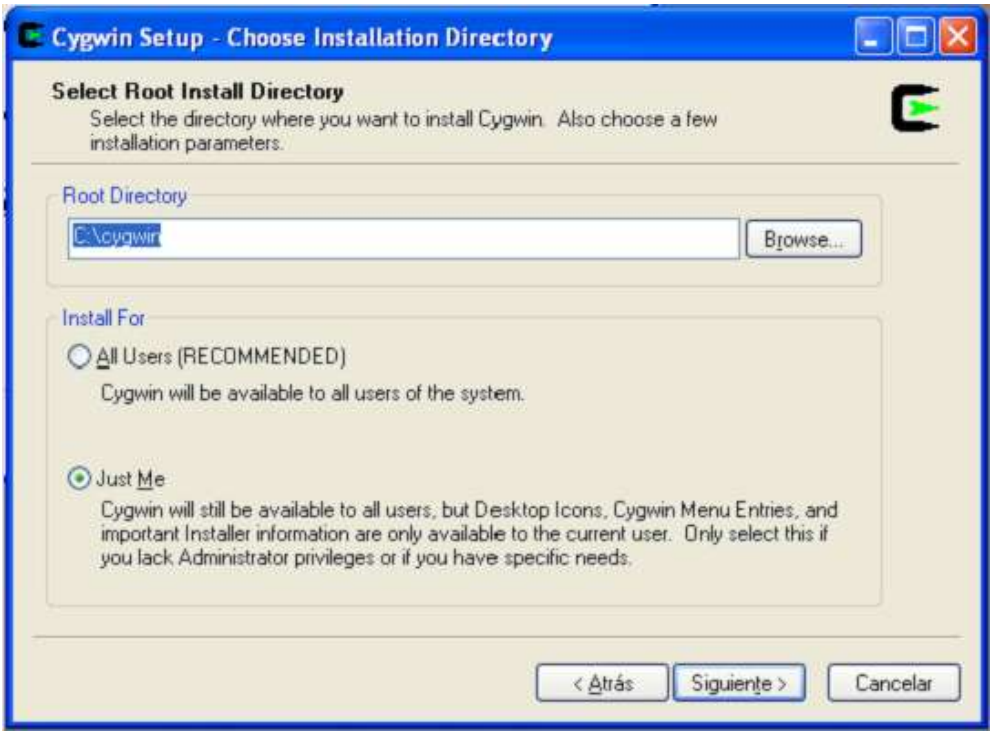
Primero se descarga el archivo setup.exe (a través del link Install or update now!). Si es este el caso, deberá tener conexión a internet durante la instalación para poder bajar los paquetes necesarios. Si utiliza el material provisto por la cátedra esto no es necesario ya que éste contiene tanto al instalador como a los paquetes.

Ejecutar el archivo setup.exe para comenzar la instalación de cygwin y hacer clic en el botón Siguiente en la ventana que se abre.

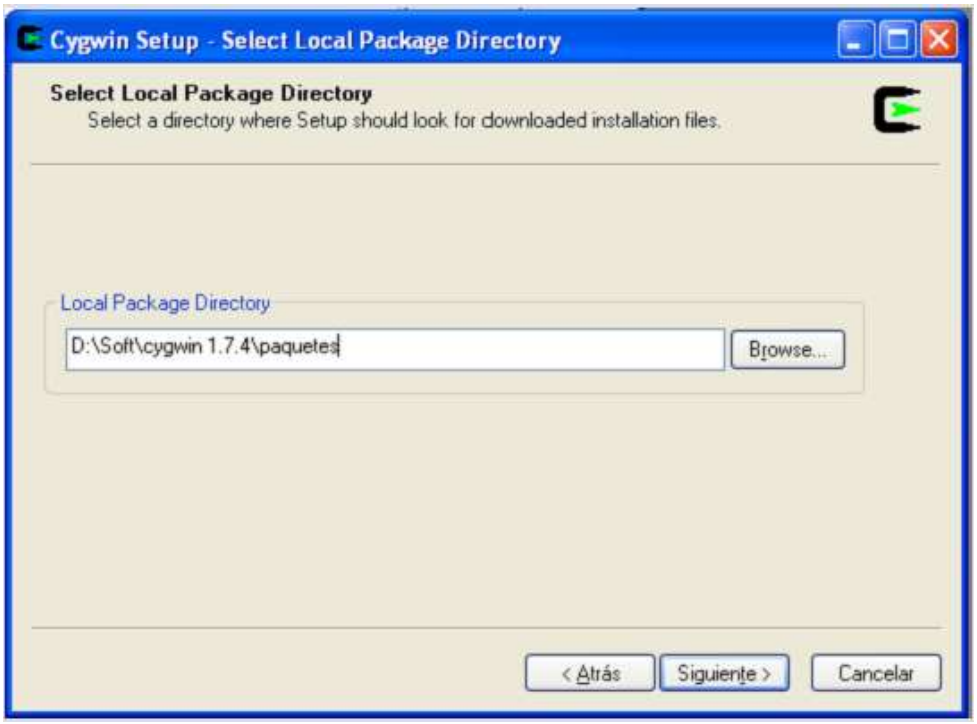
En la ventana "Cygwin Setup - Choose Installation Type" seleccionar la opción “Install from Local Directory” y hacer clic en “Siguiente”.



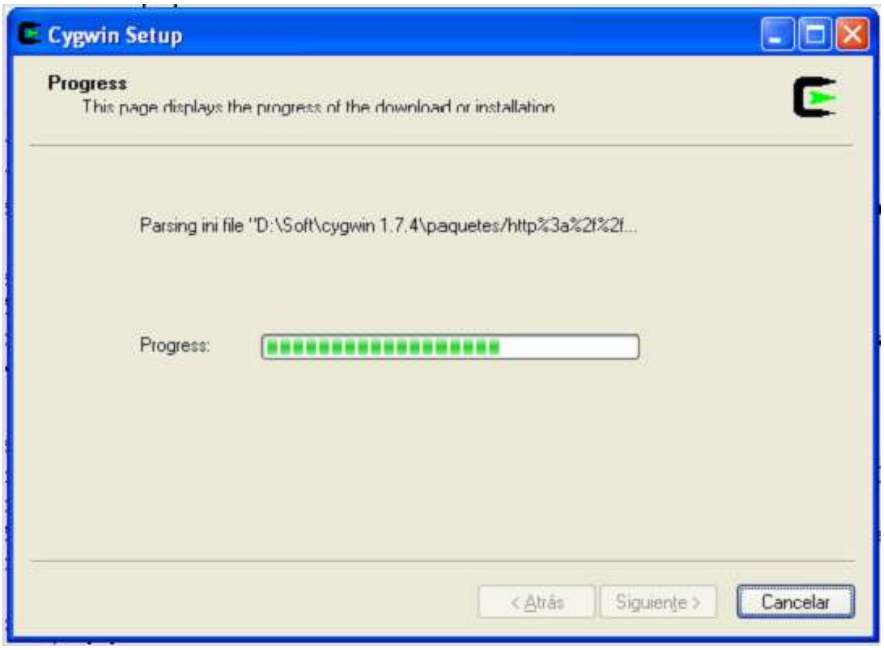
En la ventana "Cygwin Setup - Choose Installation Directory" establecer en el panel “Root Directory” la dirección C:\cygwin (si ya sale esa dirección dejarla como está). En el panel “Install For” seleccionar la opción “Just me” para instalar el programa sólo para el usuario actual y de esa forma evitar problemas con los mecanismos de seguridad de Windows.



En la ventana "Cygwin Setup – Select Local Package Directory" introducir la dirección en la que se encuentra la carpeta paquetes ubicada en el mismo directorio que el archivo setup.exe. En el ejemplo de la figura, dicha carpeta está en la ruta D:\Soft\cygwin 1.7.4, por lo que se debe introducir la dirección D:\Soft\cygwin 1.7.4\paquetes. Dicha dirección puede ser escrita o también puede ser seleccionada en la ventana que se abre al hacer clic en el botón "Browse...".

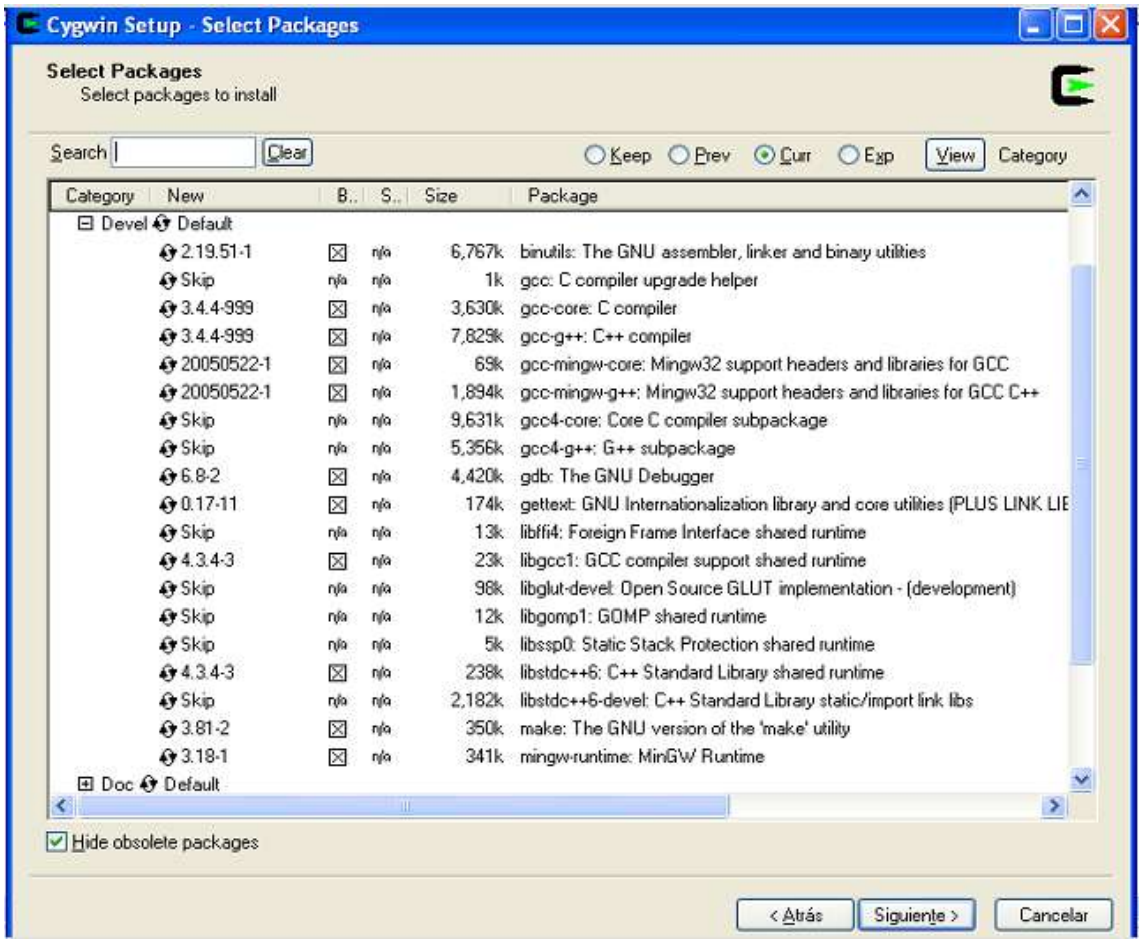


A continuación se debe esperar unos instantes hasta que se termine de organizar el archivo de configuración de los paquetes.



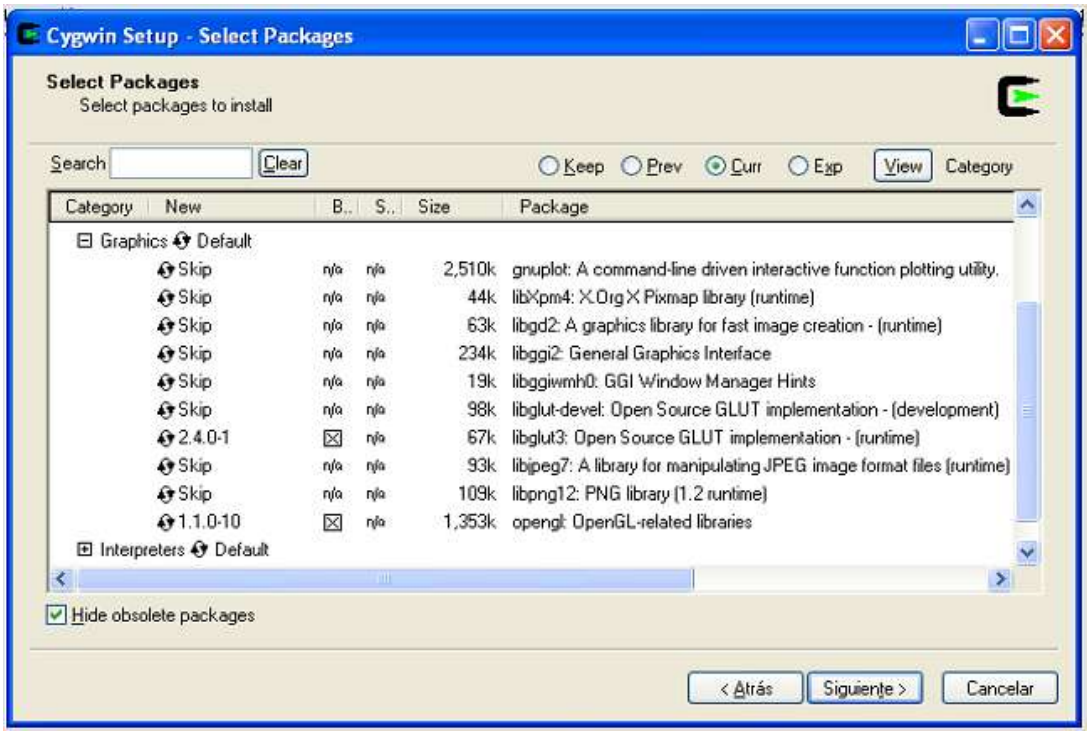
Una vez finalizado este proceso, aparece una ventana titulada "Cygwin Setup – Select Packages" en la que se deben seleccionar los paquetes que se desea instalar. Los paquetes aparecen agrupados en categorías. Por ejemplo, la categoría Admin agrupa los paquetes de administración y la categoría Devel agrupa los paquetes de desarrollo. Si se despliega una categoría haciendo clic en signo + que se encuentra a la izquierda de su nombre aparecen los paquetes de dicha categoría que se encuentran disponibles para ser instalados.

Para instalar los paquetes correspondientes al compilador de C++ y algunas utilidades para el desarrollo, se debe desplegar la categoría Devel y seleccionar los paquetes gcc-g++: C++ compiler, gdb: The GNU debugger y make: the GNU version of the 'make' utility. Para seleccionar un paquete se debe hacer clic donde dice "Skip" en el renglón correspondiente. Al seleccionar algunos paquetes puede ocurrir que automáticamente se seleccionen muchos otros; esto es normal y significa que se instalan automáticamente las dependencias de un paquete determinado. En la figura siguiente se muestra cómo debe quedar dicha categoría una vez seleccionados los paquetes necesarios.

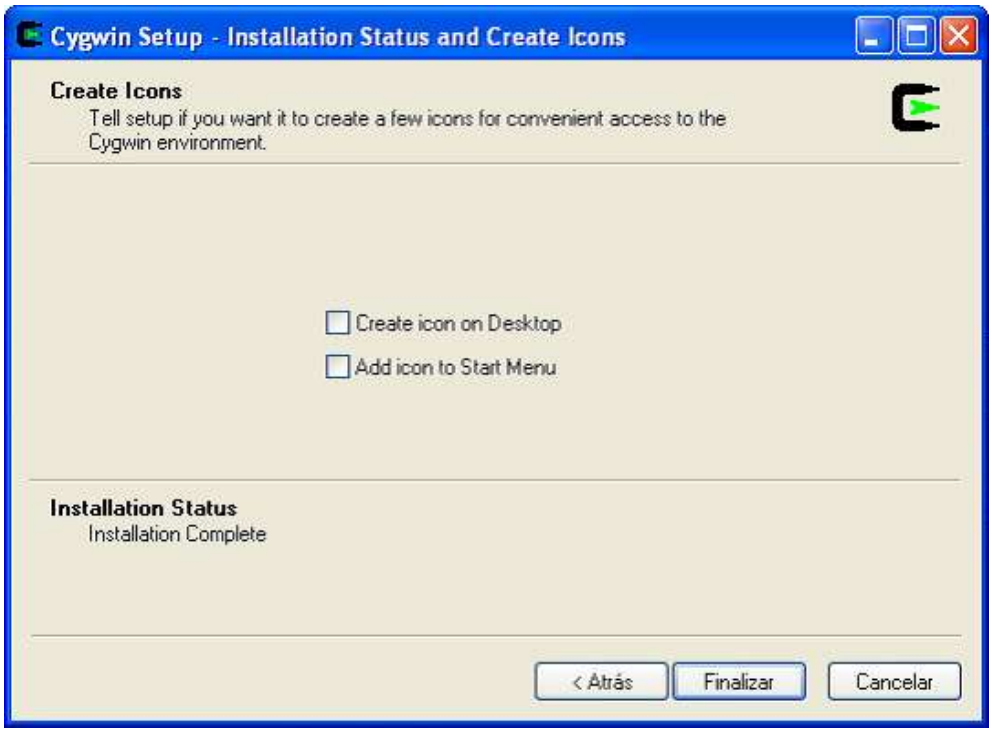


Para instalar los paquetes necesarios para la compilación de programas que utilicen OpenGL y la GLUT deben instalarse los respectivos paquetes. Para ello se debe desplegar la categoría Graphics y seleccionar en ella los paquetes opengl y libglut3. En la figura siguiente se muestra cómo debe quedar dicha categoría una vez seleccionados los paquetes de gráficos.

Es importante NO seleccionar el paquete libglut-devel, ya que produce problemas a la hora de encadenar las bibliotecas de OpenGL.



Una vez seleccionados los paquetes, se hace clic en el botón "Siguiente" y se espera hasta que se terminen de instalar los paquetes. Una vez terminado este proceso, se abre una última ventana en la que se seleccionan los accesos directos a Cygwin que se desea crear y luego se hace clic en "Finalizar", con lo cual termina la instalación del programa.



La primera vez que se ejecuta Cygwin, el programa pasa un instante configurando una serie de archivos. Esto ocurre únicamente al ejecutarlo por primera vez.

Si la instalación se realizó en Windows XP, se deben tener en cuenta los siguientes comentarios:

- Se notará que el archivo "autoexec.bat" no existe enWindows XP, por lo que no se puede modificar la variable PATH de esta manera. Para realizar los cambios necesarios, deben seguirse los siguientes pasos:

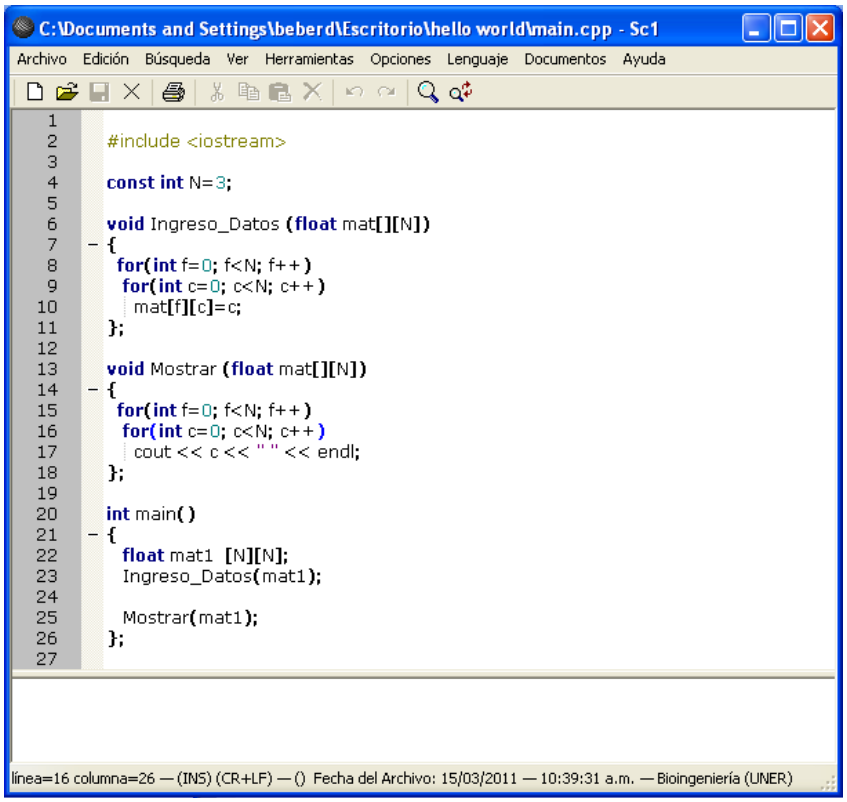
  - Hacer click con el botón derecho del mouse en el ícono de "Mi PC".
  - Seleccionar el ítem "Propiedades".
  - Seleccionar la solapa "Opciones Avanzadas".
  - Hacer click en el botón "Variables de Entorno".
  - En el cuadro "Variables de Sistema", seleccionar "PATH", y hacer click sobre el botón "Modificar".
  - En el campo "Valor de la Variable", agregar AL FINAL la siguiente línea (sin las comillas y no debe haber un espacio en blanco al final): ";C:\cygwin\bin".
  - Guardar los cambios y reiniciar la PC.
  - Instalar el editor de texto que se va a utilizar.

Herramienta de desarrollo Scintilla

Esta herramienta es un editor de texto que fácilmente puede linkearse con el compilador cygwin. Como el resto de los entornos de desarrollo, ofrece la posibilidad de que la salida por consola se realice en una ventana en el mismo entorno, sin necesidad de pasar a ejecutar el código en una ventana del sistema operativo.

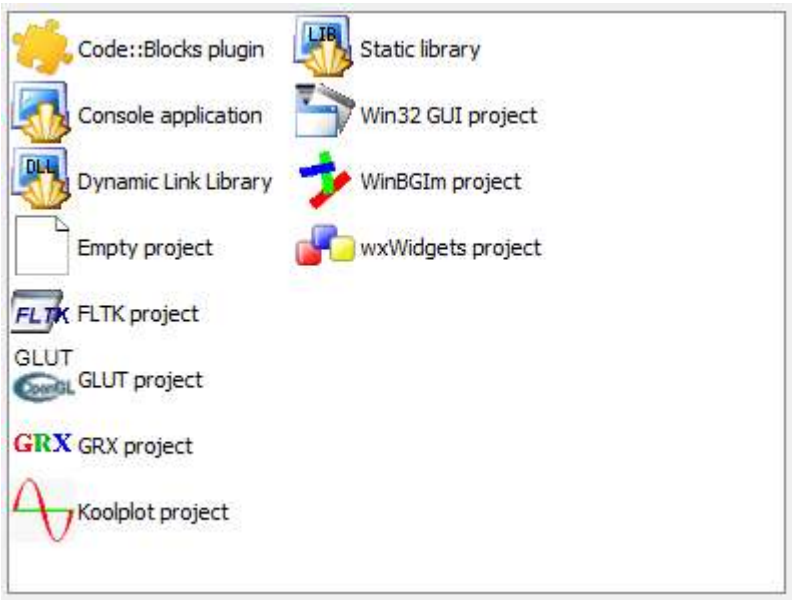
La descarga se realiza desde http://www.scintilla.org/

Su principal característica es la simplicidad, facilidad de uso y además que no necesita ser instalado.



Herramiento de Desarrollo CodeBlocks. Portable con FLTK

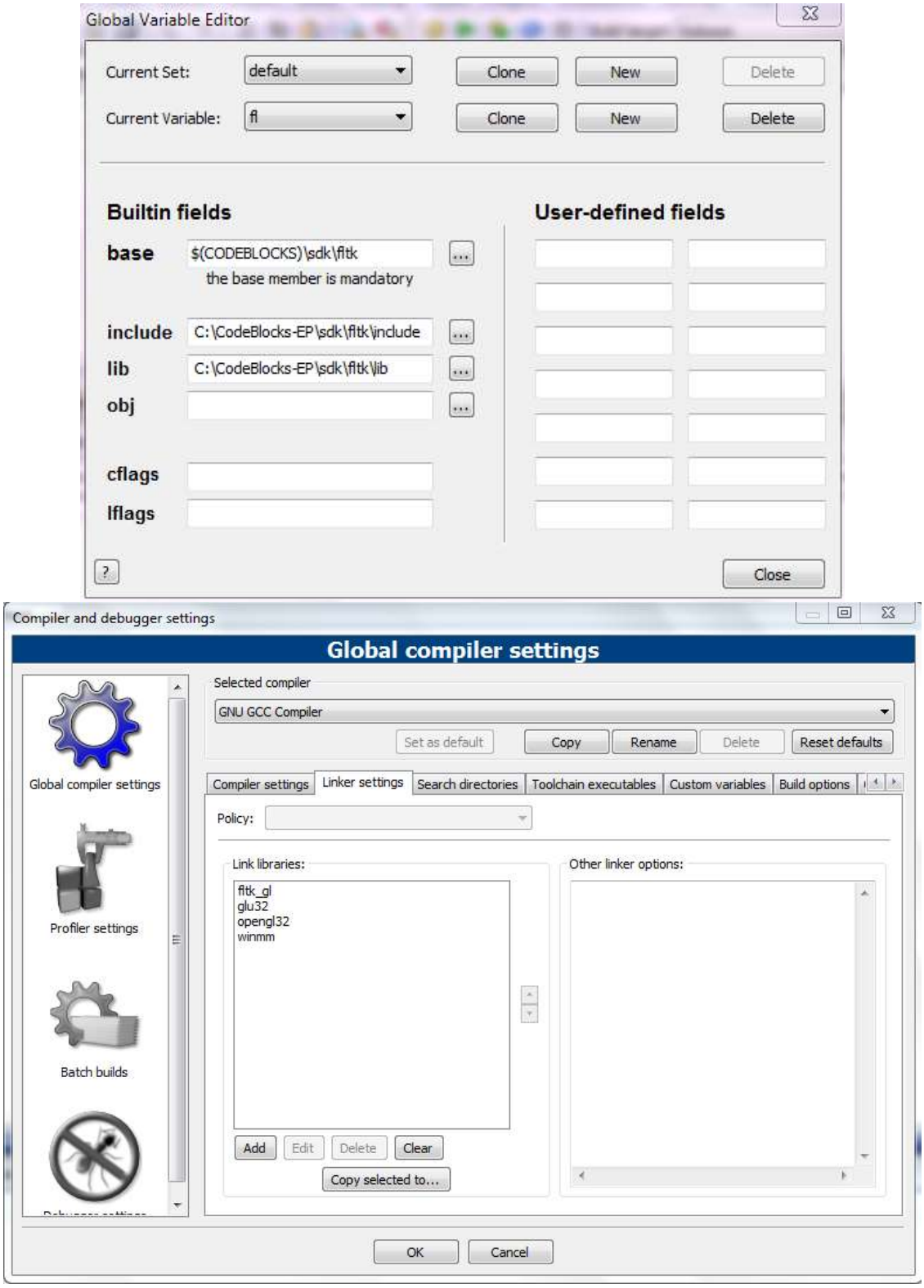
En la dirección http://codeblocks.codecutter.org/ se puede descargar una versión del entorno de desarrollo Codeblocks. Posee un editor, el compilador c++, fltk, glut, graficador de funciones matemáticas, etc.

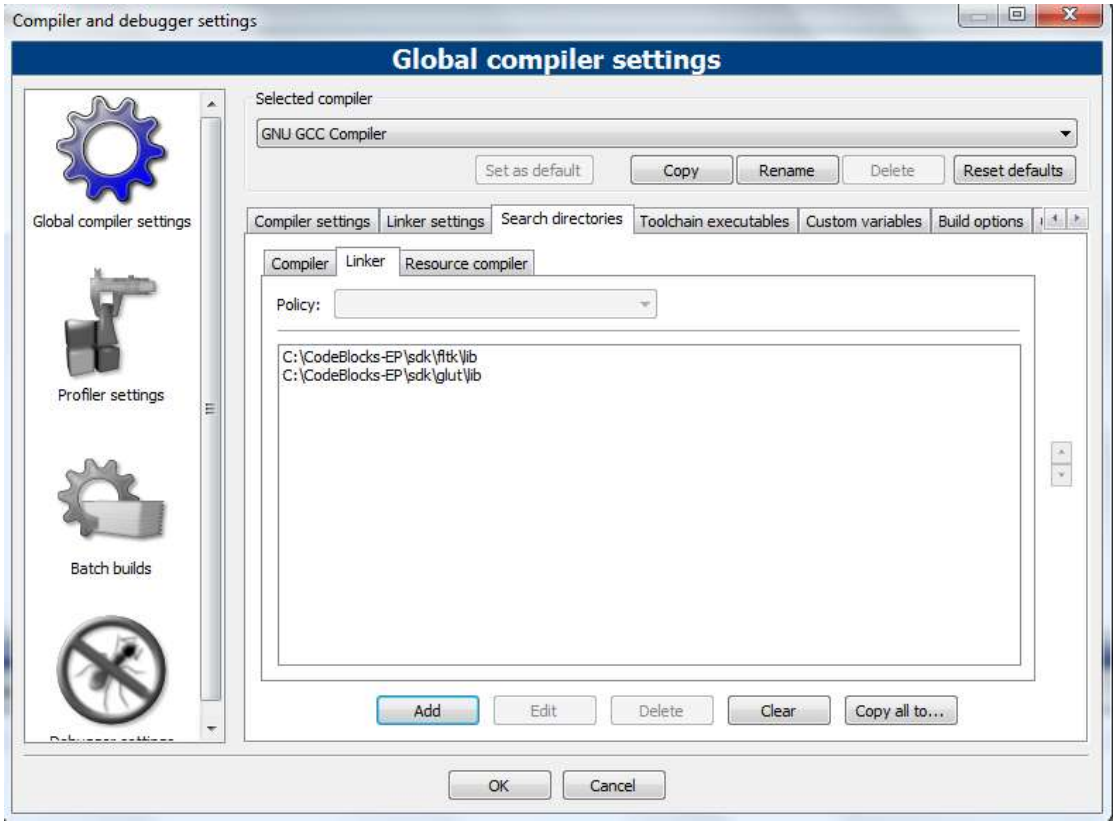


Luego de bajar el .zip, se debe descargar todo a un directorio a C:/CodeBlocks-EP. **No es necesario realizar instalación** porque es un programa "portable", por lo que no interfiere en ninguna configuración de la máquina. Para eliminarlo sólo hay que borrar este directorio. Además, puede ser copiado y ejecutado desde un pendrive.

Si se realiza un acceso directo desde el escritorio o el menú inicio, hay que realizarlo a "CbLancher.exe" y no a "codeblocks.exe".

Para trabajar con FLTK (OpenGL) hay que realizar las siguientes configuraciones:


Global Variable Editor
Current Set:
default
Clone
New
Delete
Current Variable:
fl
Clone
New
Delete
Builtin fields
User-defined fields
base
$(CODEBLOCKS)\sdk\fltk
the base member is mandatory
include
C:\CodeBlocks-EP\sdk\fltk\include
lib
C:\CodeBlocks-EP\sdk\fltk\lib
obj
cflags
lflags
Close
Compiler and debugger settings
Global compiler settings
Selected compiler
GNU GCC Compiler
Set as default
Copy
Rename
Delete
Reset defaults
Compiler settings
Linker settings
Search directories
Toolchain executables
Custom variables
Build options
Policy:
Link libraries:
fltk_gl
glu32
opengl32
winmm
Other linker options:
Add
Edit
Delete
Clear
Copy selected to...
Global compiler settings
Profiler settings
Batch builds
OK
Cancel

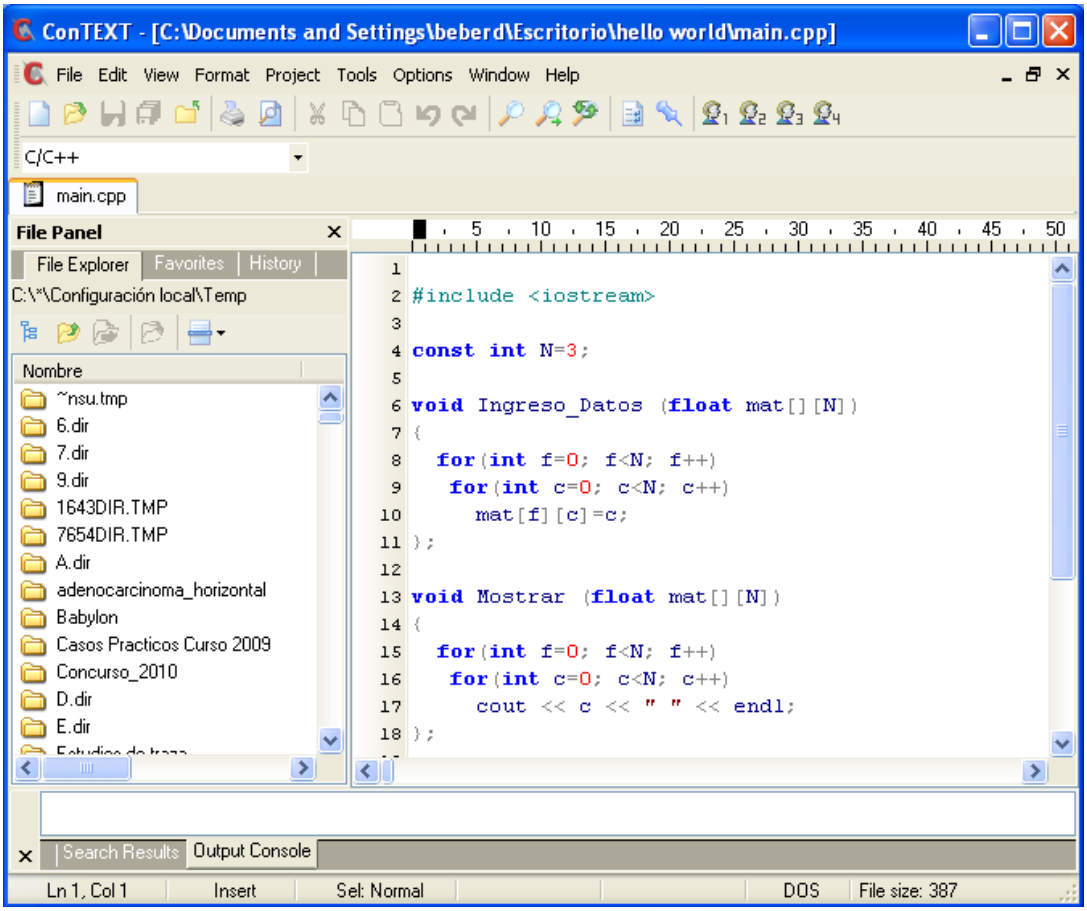


Herramienta de desarrollo ConText

Esta herramienta es un editor de texto que fácilmente puede linkearse con el compilador cygwin.

La descarga se realiza desde http://www.contexteditor.org/

Aunque trae un compilador por defecto, se recomienda el uso del cygwin



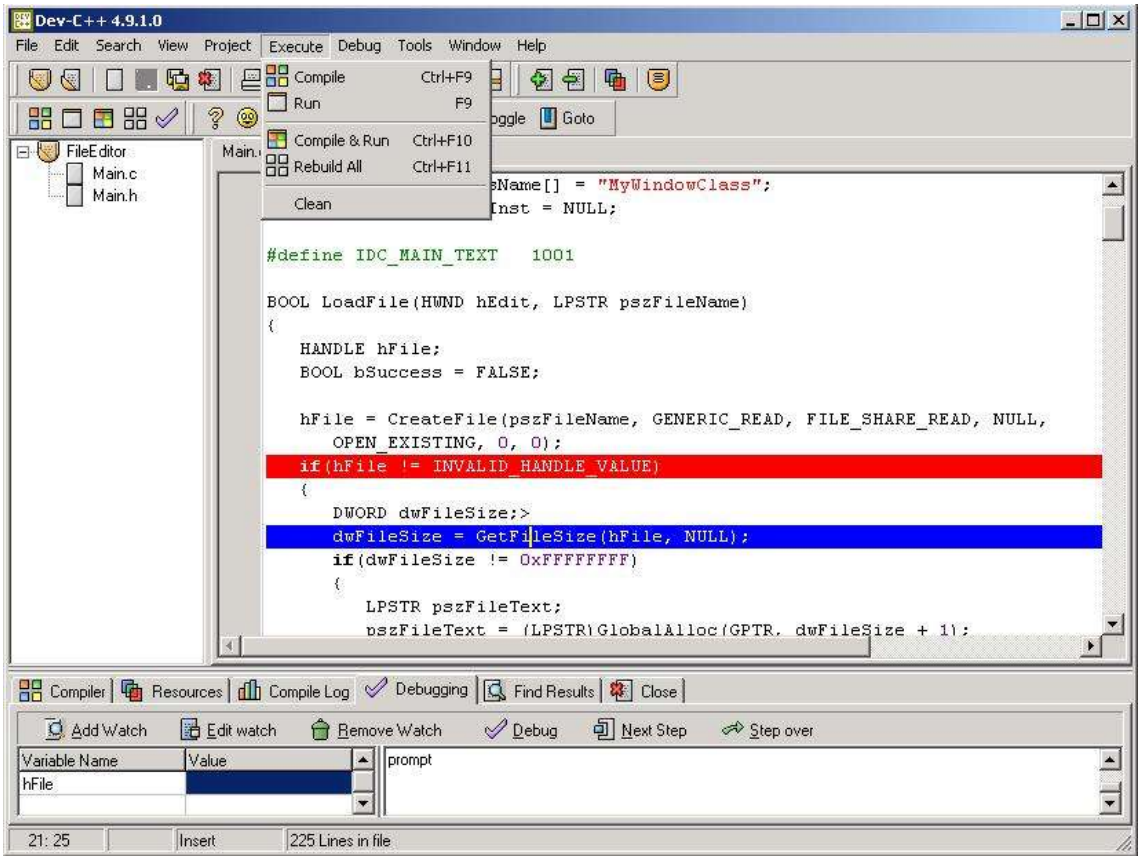
Herramienta de desarrollo DevC++

A diferencia de los anteriores que prácticamente se los puede usar como editores de texto, esta herramienta es un paso hacia los entornos de desarrollo.

También trae un compilador por defecto pero se aconseja realizar un link con el cygwin como a continuación se explica.

Este entorno de desarrollo tiene la ventaja de detectar y mostrar automáticamente las clases, sus propiedades y sus método para una fácil identificación y acceso. Esta característica es muy útiles en el desarrollo de programas largos y complejos.

Se descarga de http://www.bloodshed.net/devcpp.html

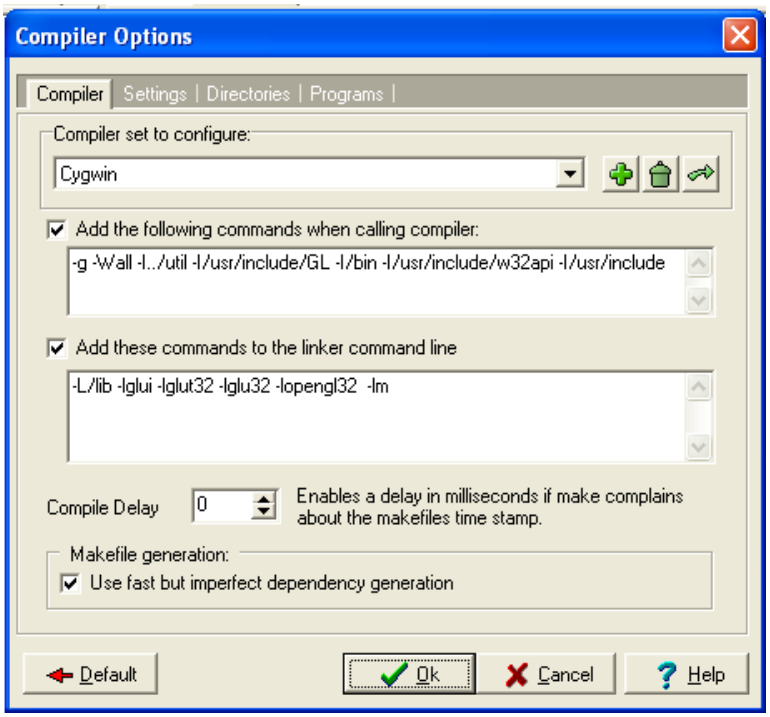


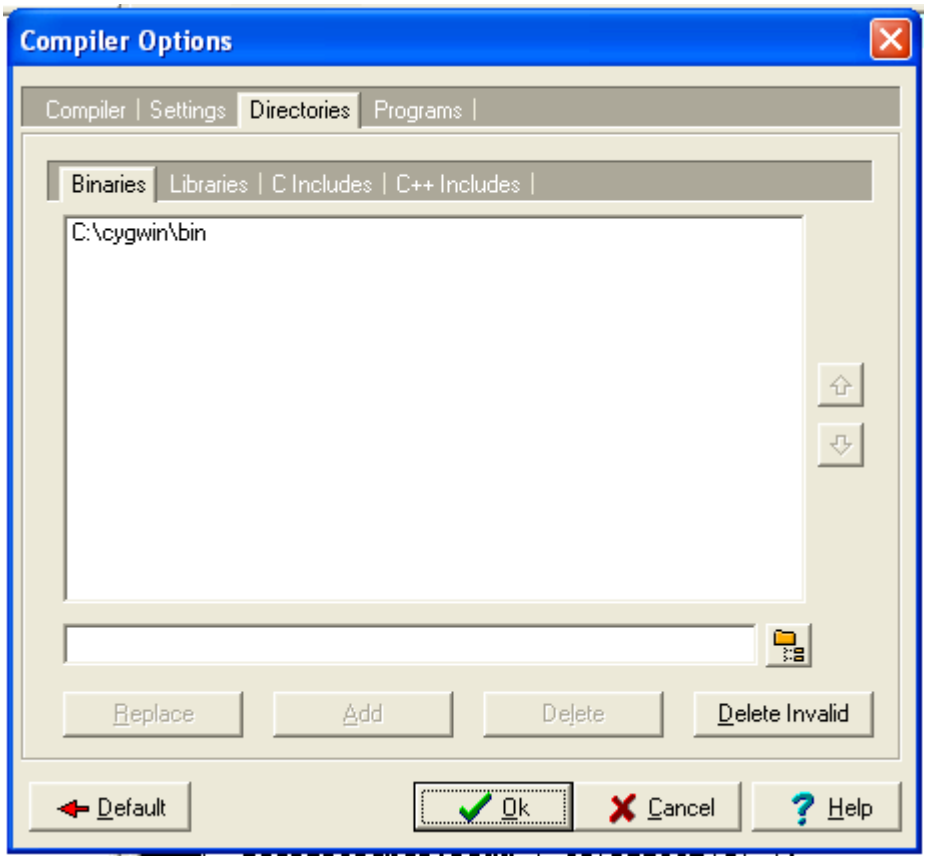
A continuación se describe como configurarlo para que compile con el cygwin.

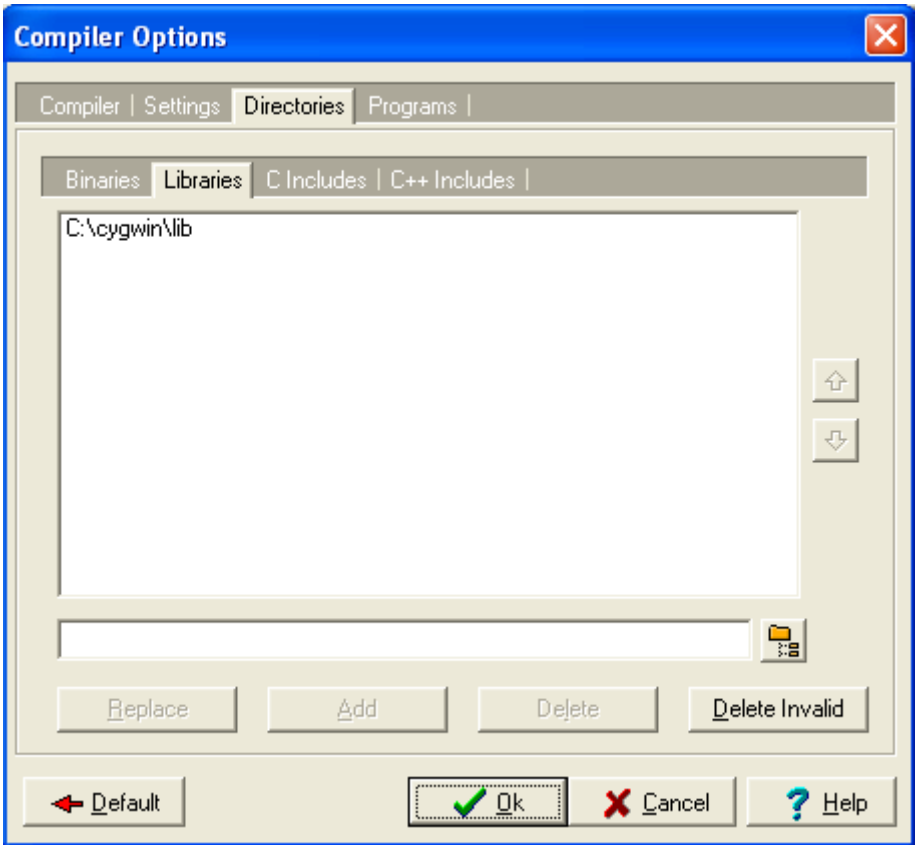
Se arma a partir del makefile siguiente:

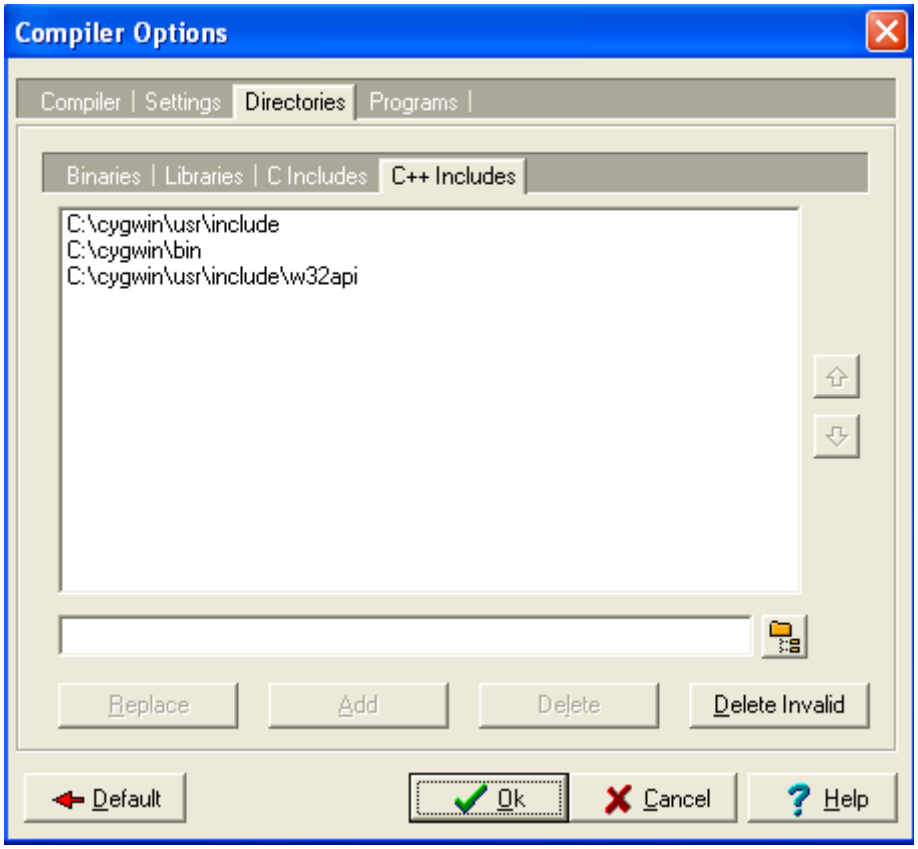
```
CXX=g++
OPCIONES=-g  -Wall  -I../util  -I/usr/include/GL  -I/bin  -I/usr/include/w32api  -
I/usr/include
BIBLIOTECAS=-L/lib -lglui -lglut32 -lglu32 -lopengl32  -lm
all:
      ${CXX} ${OPCIONES} main.cpp -o main ${BIBLIOTECAS}
```

Las siguientes pantallas muestras las configuraciones necesarias

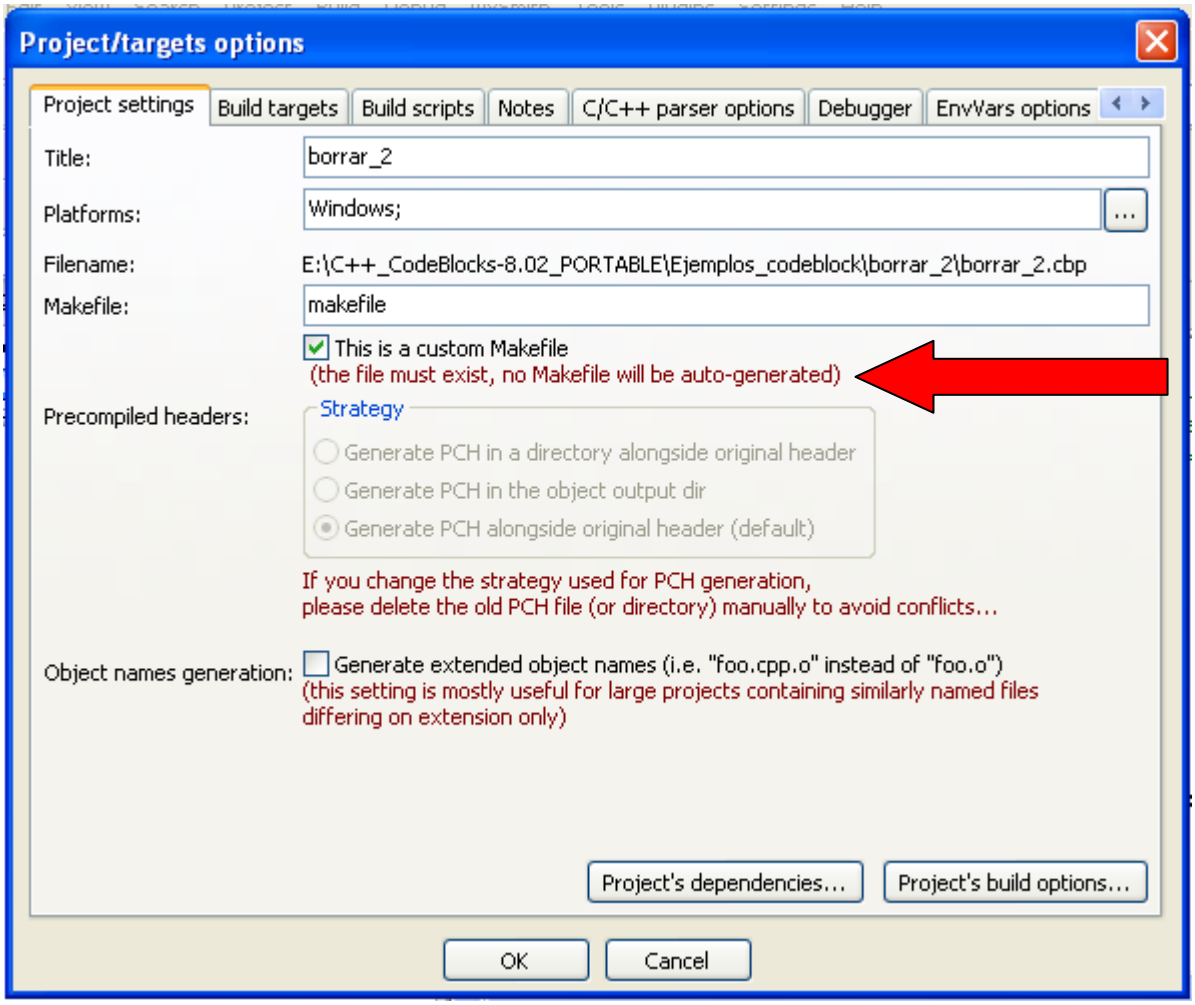

Compiler Options
Compiler
Settings
Directories
Programs
Compiler set to configure:
Cygwin
Add the following commands when calling compiler:
-g -Wall -I../util -I/usr/include/GL -I/bin -I/usr/include/w32api -I/usr/include
Add these commands to the linker command line
-L/lib -lglui -lglut32 -lglu32 -lopengl32 -lm
Compile Delay
0
Enables a delay in milliseconds if make complains about the makefiles time stamp.
Makefile generation:
Use fast but imperfect dependency generation
Default
Ok
Cancel
Help


Compiler Options
Compiler
Settings
Directories
Programs
Binaries
Libraries
C Includes
C++ Includes
C:\cygwin\bin
Replace
Add
Delete
Delete Invalid
Default
Ok
Cancel
Help





La hoja C Includes no interesa.


Project/targets options
Project settings
Build targets
Build scripts
Notes
C/C++ parser options
Debugger
EnvVars options
Title:
borrar_2
Platforms:
Windows;
Filename:
E:\C++_CodeBlocks-8.02_PORTABLE\Ejemplos_codeblock\borrar_2\borrar_2.cbp
Makefile:
makefile
This is a custom Makefile
(the file must exist, no Makefile will be auto-generated)
Precompiled headers:
Strategy
Generate PCH in a directory alongside original header
Generate PCH in the object output dir
Generate PCH alongside original header (default)
If you change the strategy used for PCH generation,
please delete the old PCH file (or directory) manually to avoid conflicts...
Object names generation:
Generate extended object names (i.e. "foo.cpp.o" instead of "foo.o")
(this setting is mostly useful for large projects containing similarly named files
differing on extension only)
Project's dependencies...
Project's build options...
OK
Cancel