

Introducción a la programación con C++



Edificio de la división de
Matemáticas e Ingeniería
Oficina 10.



Universidad Nacional Autónoma de México
Facultad de Estudios Superiores Acatlán
Grupo de Cómputo Cuántico y Científico



Proyecto PAPIME
PE104919

Tabla de Contenido

Licencia	IV
I Introducción	VI
II Contenido general de C++	1
Concepto básicos	1
1. ¿C++?	1
2. Entornos de desarrollo	1
2.1. Instalación de Dev-C++ (Windows)	1
2.2. Corriendo C++ por terminal (Linux)	9
3. Compilar y Correr el código	14
1. Bases	15
1. Programación	15
2. Paradigmas de programación	15
3. Paradigma Estructurado	16
4. Empezando a programar en C++	16
5. Tipos de datos básicos	17
6. Funciones	19
6.1. Declarar y definir	21
6.2. Procedimientos	22
7. Conversiones	23
7.1. Conversiones Implícitas	23
7.2. Conversiones Explícitas	24
2. Palabras reservadas	25
1. Librerías	26
3. Operadores y secuencias de control	27
1. Operadores y su tabla de precedencia	28
1.1. Operadores aritméticos	29
1.2. Operadores unarios	30
1.3. Operadores relacionales y los booleanos	31
1.4. Operadores lógicos	32
1.5. Operadores bit a bit	33
1.6. Operadores de desplazamiento de bits o shift	35
1.7. Operadores de asignación	36
1.8. Operador ternario	37
2. Secuencias de control básicas	38
2.1. if, else	38

2.2.	for	39
2.3.	while	40
2.4.	do while	41
4.	Pensando en código	42
1.	Pseudocódigo	42
2.	Memoria	43
3.	Apuntadores	44
5.	Sección Yeyeco	47
1.	Secuencias de escape	47
2.	Buenas practicas de comentarios	48
3.	Recursividad	51
4.	Ahora Yeyeco	52
6.	Expandiendo el conocimiento	55
1.	Por diversión	56
2.	Por profesión	56
3.	Sugerencias	56
	Motivación final	57
	Bibliografía	59

Licencia

Este documento está creado con fines educativos. La información contenida está sometida a cambios y a revisiones constantes, por lo que se sugiere no imprimirlo.

Puedes compartir el documento con quien desees; sin embargo debes respetar la autoría original. Puedes citar el material y considerarlo como referencias en tus proyectos.

This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.

Esta obra está bajo una licencia [Creative Commons](#) “Reconocimiento-NoCommercial-CompartirIgual 3.0 España”.



Para citar este documento:

1. Bibtex

```
@manual{introCppGCCyC,  
  title   = "{Introducci\`on a la Programaci\`on con C++}",  
  author  = "{Aguirre-Andrade, S. U. and Aguilar-Hilario, M.  
and Orduz-Ducuara, J.A.}",  
  organization = "{Grupo de C\`omputo Cu\`antico y Cient\`ifico}",  
  month   = "{Febrero}",  
  year    = "{2020}",  
  address = "{FESAC-UNAM}",  
  note    = "Los miembros del GCCyC aporta constantemente al desarrollo  
de este proyecto",  
}
```

2. bibitem

```
\bibitem{introCppGCCyC}  
Aguirre-Andrade, S. U., Aguilar-Hilario, M.,  
Orduz-Ducuara, J.A.  
\textit{Introducci\`on a la Programaci\`on con C++}.  
Los miembros del GCCyC aporta constantemente al desarrollo  
de este proyecto  
GCCyC, FESAC-UNAM, 2020.
```

Miembros activos del GCCyC:
Salvador Uriel Aguirre Andrade^a
Miguel Aguilar Hilario^b
Miguel González Briones
Alfonso Flores Zenteno
Ana Karen Del Castillo
Diana De Luna
Leslie Valeria Vivas Lurrabaquio
Zitlalli Nayeli Avilés Palacios
Eledtih Andrea González Sánchez
Edgar Ivan Martínez Villafañe
Oscar Jair Vargas Palacios

^aMiembro que contribuyó al desarrollo de este material

^bMiembro que contribuyó al desarrollo de este material

Dr. Javier A. Orduz-Ducura
Profesor de Carrera Asociado C.
Edificio de la división de Matemáticas e Ingeniería
Oficina 10.
FES Acatlán-UNAM

Parte I: Introducción

Introducción general del curso

Aprender un lenguaje de programación es similar, en cierto modo, a aprender un nuevo idioma diferente al nativo. Un lenguaje de programación, se parece a un idioma, tiene su sintaxis (reglas del lenguaje), asimismo, en computación, existe el concepto de semántica: en pocas palabras la matemática detrás de la programación. Este conjunto de características son propias de cada lenguaje. Al final, un lenguaje de programación y un idioma son formas de expresión y de comunicación. Este aprendizaje puede implicar tanta pasión y compromiso como el discente quiera: la satisfacción por una nueva forma de expresión es propia de cada individuo. Después de incluir nuevas palabras, conceptos y reglas a la forma de expresión, seguramente, el lector se sentirá satisfecho y optará por aprender un nuevo lenguaje.

En estas notas se dejan las bases de la programación en C++. Por cuestiones de fluidez, este documento omite la semántica de la programación y expone los principios de la sintaxis del lenguaje. El propósito es compartir el conocimiento con aquellos interesados en el autoaprendizaje y el desarrollo de habilidades computacionales, más que fortalecer impartir la lógica detrás de la programación. Sin embargo, es importante que el lector comprenda que la programación tiene profundos conceptos matemáticos que no se discuten en este trabajo.

En este manual, el lector encuentra ejercicios que puede mantener ocultos hasta que dé clic sobre la palabra "solución" para mostrar el resultado y comparar su respuesta. Se anima al lector que mantener oculta la respuesta hasta proponer una solución. Además, el estudiante debe considerar que la respuesta puede tener diferentes soluciones. No se desanime.

Por otro lado, para mi (Javier Orduz) es un orgullo comentar que este trabajo ha sido desarrollado por los miembros del GCCyC (grupo que yo inicié), un grupo muy joven en la FESAc-UNAM, con miembros muy activos, que ven en sus compañeros: un equipo y un amigo de quien aprender; y con quien compartir. Esta generación aporta bastante al desarrollo del institución y del país. Es un grupo que ha nacido por iniciativa de jóvenes inquietos que me dieron la oportunidad de acompañarlos en una parte de su etapa de formación y que me brindaron su confianza para pensar en un ejercicio académico como el Grupo de Cómputo Cuántico y Científico. Espero aportar al grupo y continuar con la recepción de las enseñanzas que me brindan cada día. Agradezco mucho su apoyo.

Es posible que algunos miembros que contribuyeron al inicio de este proyecto hayan sido ignorados, pero no olvidados, así que agradeceré que me contacten para dar el respectivo reconocimiento.

Además de la siguiente división del contenido se sugiere que el usuario revise los códigos que se dejan en Github, GitLab, Jupyter y otros repositorios que están asociados a este curso, y que pueden ser consultados desde el sitio:

yeyeco¹. El contenido de este documento es: capítulo 1, hay revisión de conceptos básicos sobre C++ y encuentra una discusión sobre los paradigmas de programación; capítulo 2, se exponen las palabras reservadas, capítulo 3, se discuten los operadores implementados en el lenguaje de programación, y las secuencias (instrucciones o estructuras) de control; capítulo 4, presentación de algunos ejemplos de documentación externa para mantener el código bajo control. Capítulo 5, hay una serie de ejercicios/retos que el lector debería enfrentar con las herramientas suficientes, después de leer este material. Finalmente (cap. 6), se dejan comentarios, sugerencias y algunas referencias consultadas.

¹La dirección es <http://www.yeyeco.acatlan.unam.mx/>

Parte II: Contenido general de C++

Concepto básicos

En este documento se verán las características mas generales e introductorias del lenguaje de programación C++ para permitirle al lector adentrarse a la programación y a la solución de problemas por este medio. A continuación se explican las razones por las que se selecciona y recomienda este lenguaje, también como instalarlo y empezar a usarlo.

1. ¿C++?

C++ es un lenguaje de programación desarrollado por *Bjarne Stroustrup* basado en el lenguaje C, Se seleccionó C++ por:

- Su popularidad en aplicaciones robustas y que requieren computación ágil.
- La abundancia de fuentes, guías y manuales para aprender C++ y otras herramientas que hacen uso del lenguaje.
- Su gran capacidad y agilidad para operaciones.

2. Entornos de desarrollo

Para empezar a desarrollar nuestros proyectos necesitamos un **IDE** (*Integrated Development Environment*) que en español sería (*Entorno de Desarrollo Integrado*) quiere decir que es más completo y eficiente que un editor de texto. Hay muchos tipos de IDE, los más conocidos para c++ son CodeBlocks, Dev-C++, Visual Studio Code entre otros.

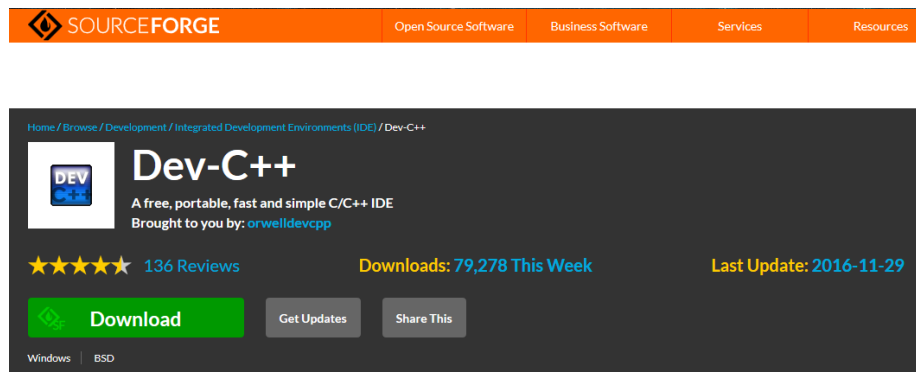
Para este curso se muestra como instalar Dev-C++, pero puedes usar el que quieras, debido que las lecciones que veremos están enfocadas en el lenguaje C++ y no depende del IDE. Solo se recomienda que el compilador de C++ pueda usar C++11 en adelante, ya que se hará mucho uso del mismo en este manual.

Si no deseas (o no puedes) instalar nada inclusive puedes usar un compilador en línea, a lo largo del manual se usa el siguiente: https://www.onlinegdb.com/online_c++_compiler. Aunque se recomienda familiarizarse con un IDE ya que estos tienen más funcionalidades.

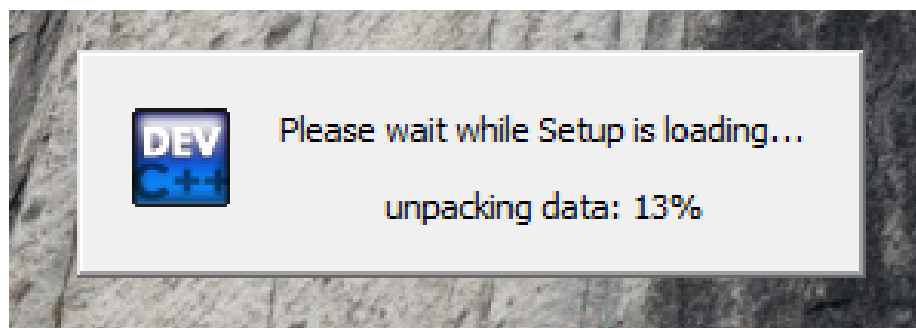
2.1. Instalación de Dev-C++ (Windows)

Entramos a nuestro navegador e ingresamos al siguiente enlace:
<https://sourceforge.net/projects/orwelldevcpp/>

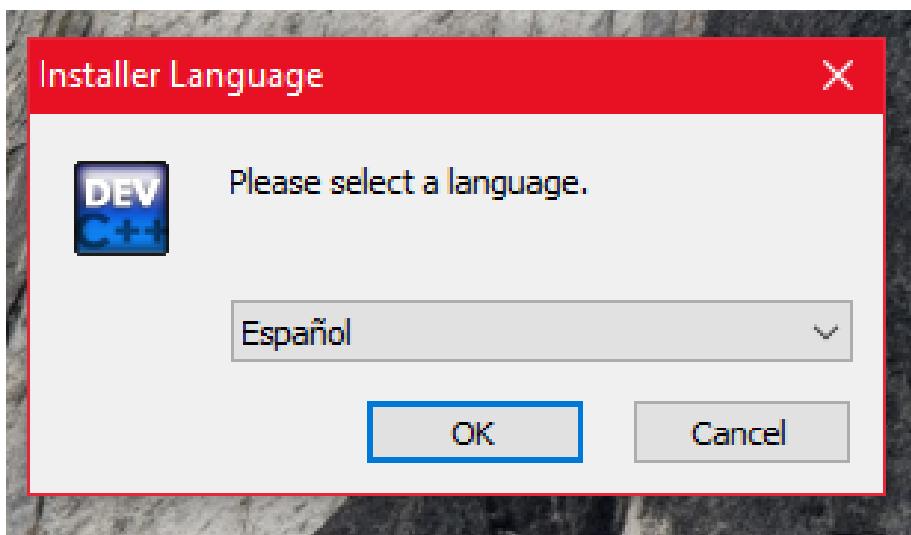
Una vez en la pagina damos *click* en el botón de *Download*



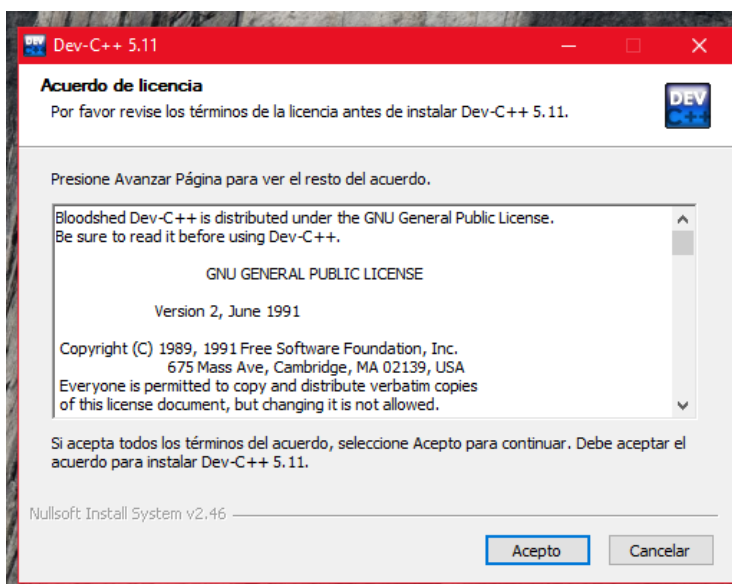
Ya descargado el programa, ubicamos donde se guardó y damos doble clic para empezar la instalación del programa.



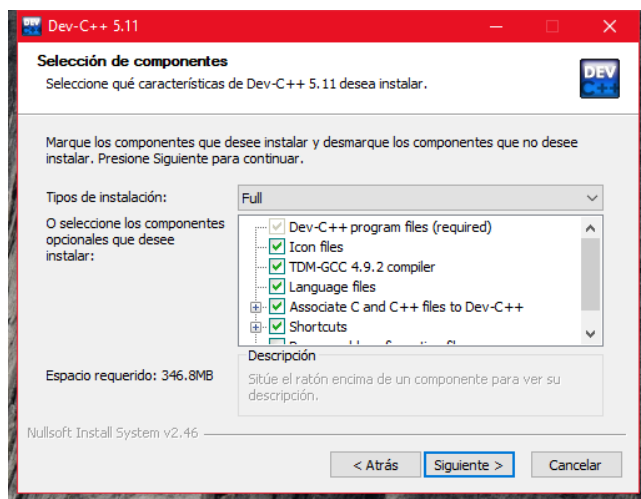
A continuación nos pide el idioma para proceder con la instalación. En este caso seleccionamos español.



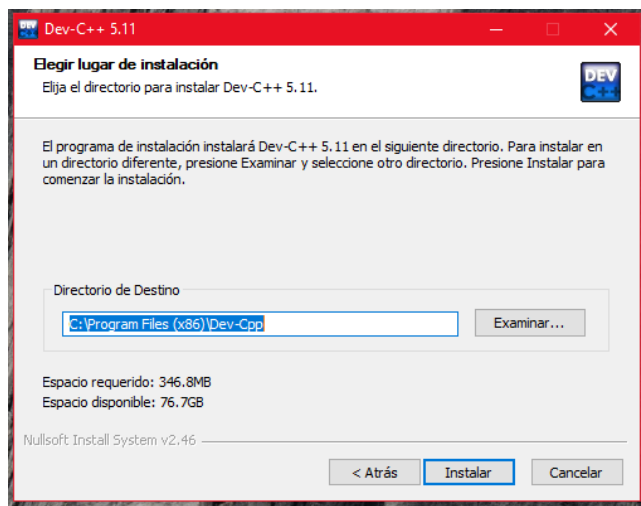
Aceptamos los términos de la licencia.



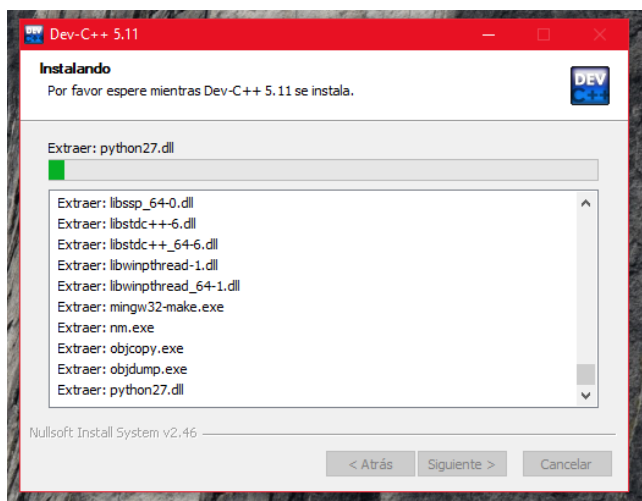
En este paso dejamos todas los componentes seleccionados (como se muestra en la imagen) para tener una buena instalación de nuestro IDE, si no aparece seleccione *Full* en la lista desplegable al lado de "Tipos de instalación".



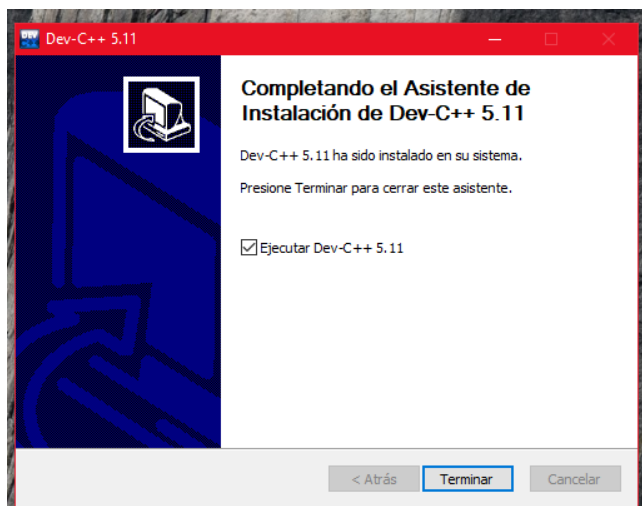
Después nos pide la ruta de donde queremos instalar el programa, en este caso se deja la ruta por defecto en el disco C:



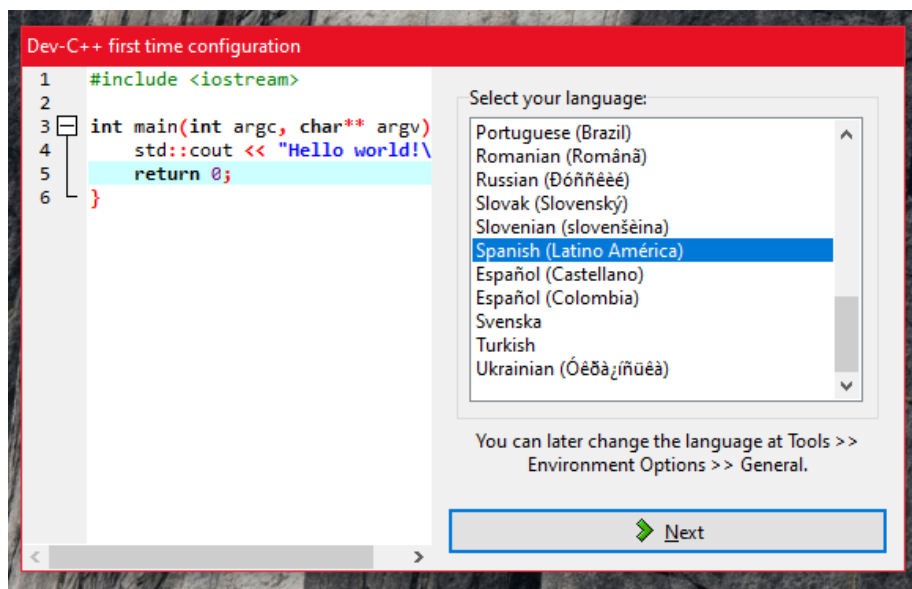
Se debe esperar a que termine el proceso de instalación de nuestro IDE.



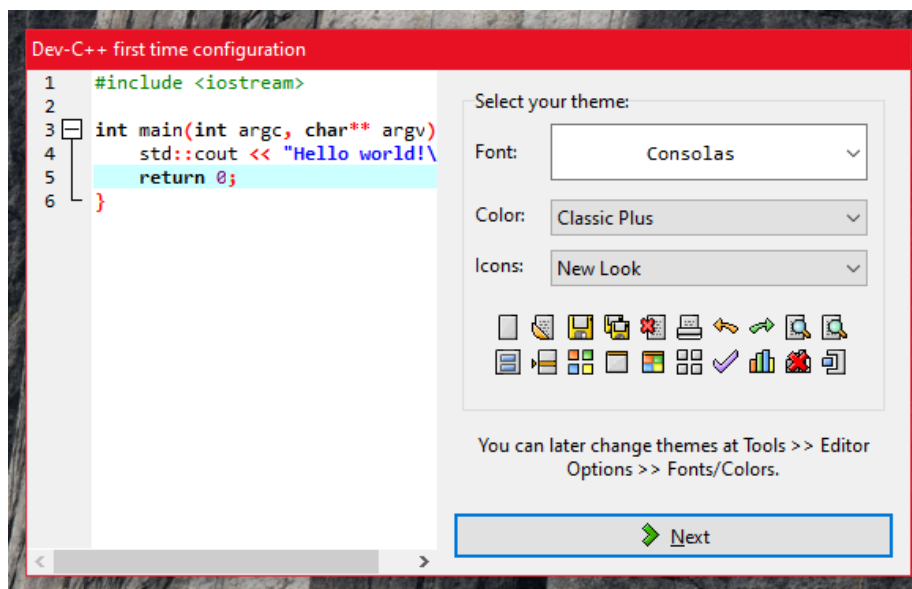
Una vez terminado nos avisa que la instalación fue un éxito y se pregunta si deseamos ejecutar de una vez Dev-C++. Dejamos seleccionada la casilla de “Ejecutar” y damos click en Terminar para empezar a programar.



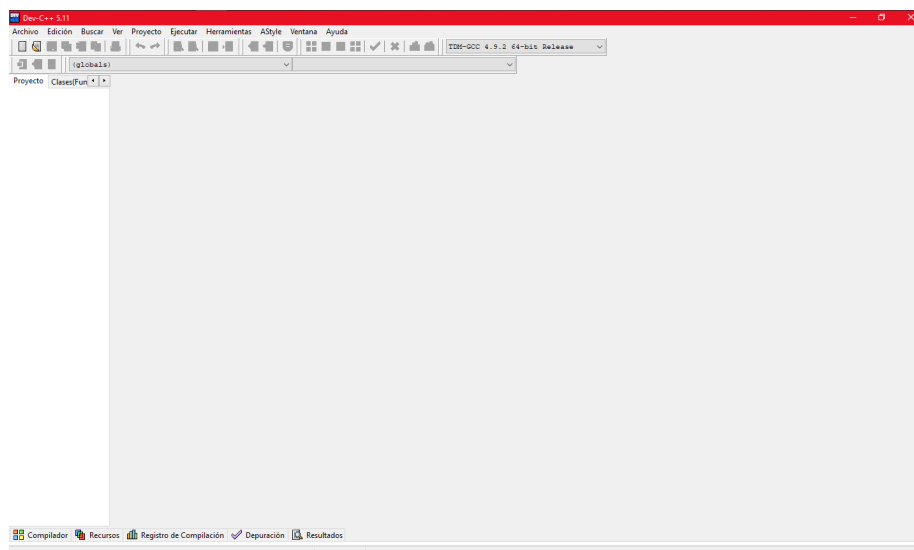
Se vuelve a pedir el lenguaje del IDE, a lo que seleccionamos otra vez español (*Spanish(Latino América)*).



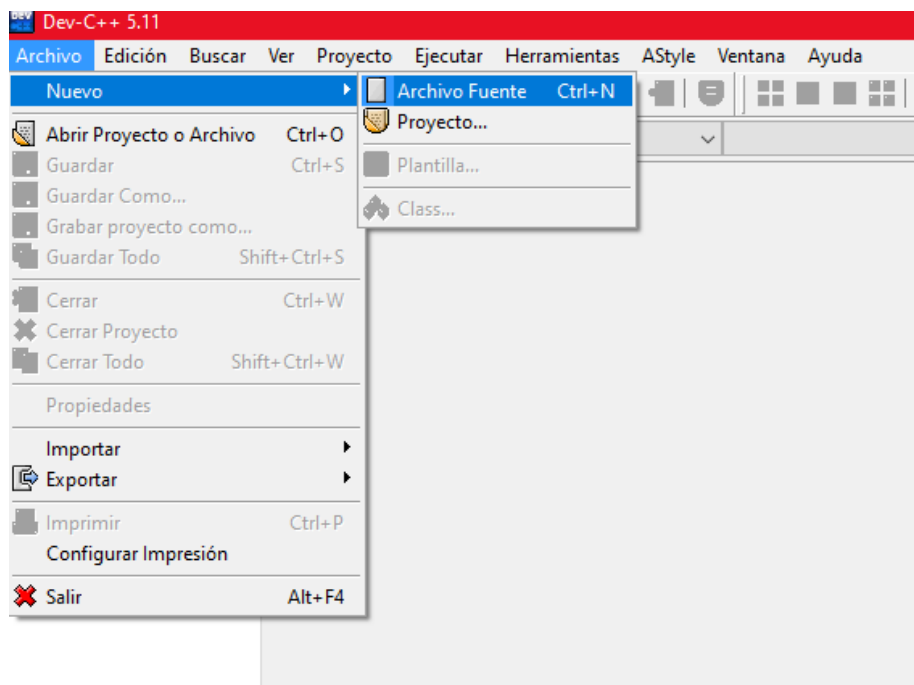
Ahora nos permite personalizar nuestro entorno de desarrollo. por el momento lo dejaremos como viene.



Una vez terminado se abre la ventana principal donde podremos empezar a programar.



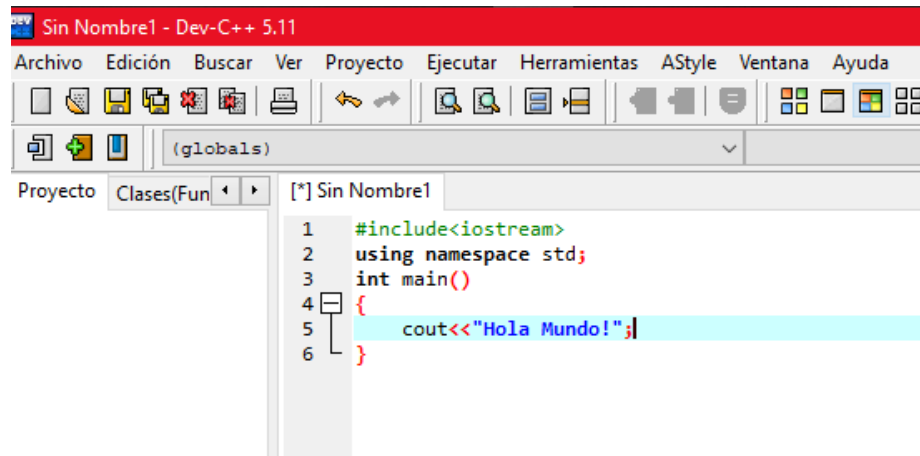
En la esquina superior izquierda de la barra de opciones se hace click en “Archivo”, “Nuevo” y posteriormente en “Archivo Fuente”. El atajo de teclado para hacer esto son los botones “Ctrl” y “N”.



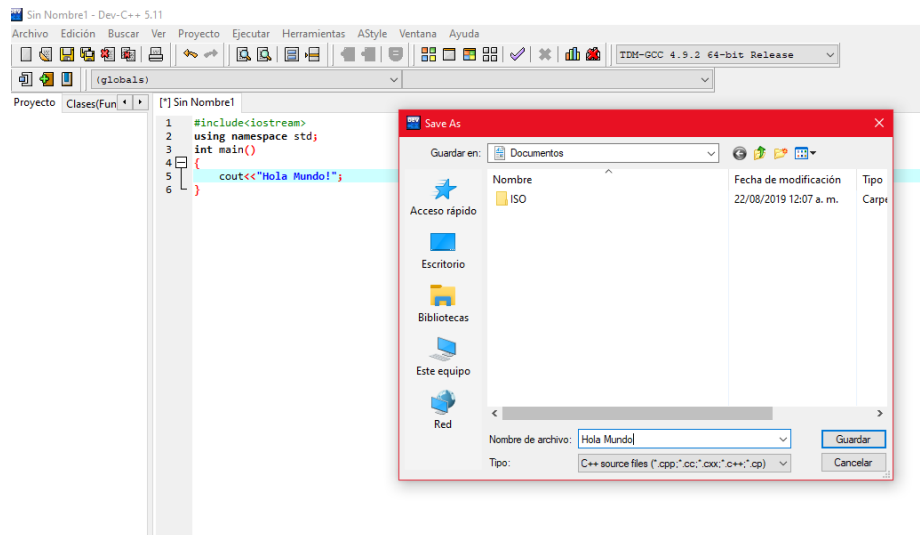
Una vez creado el archivo nos muestra un espacio para escribir nuestro código.

go.

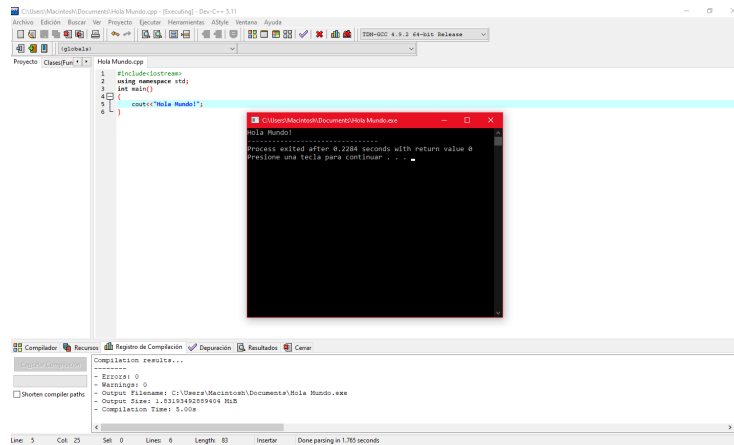
En este caso, escribiremos el clásico "Hola Mundo".



Posteriormente vamos a la barra de opciones en "Ejecutar", "Compilar y Ejecutar", ahí también nos menciona el atajo de teclado. Nos pide la ubicación donde queremos guardar nuestro código así como el ejecutable.



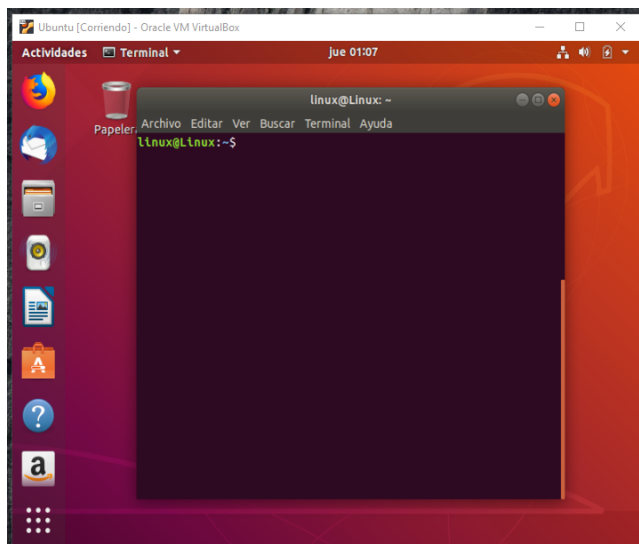
Al terminar de dar nombre y guardar el archivo, si nuestro programa esta correcto nos lanzara la consola la cual es el ejecutable, y se imprimirá el mensaje que habíamos escrito anteriormente en el código.



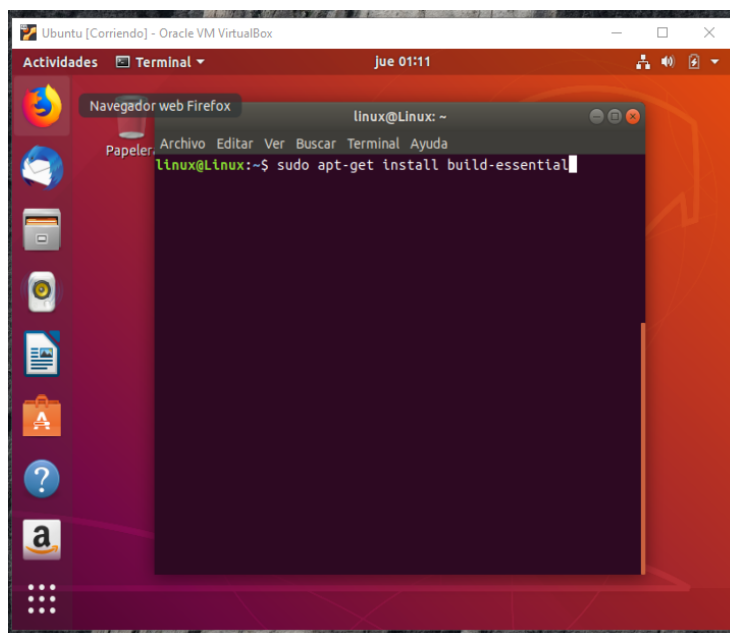
Y así ha creado su primer programa en C++!

2.2. Corriendo C++ por terminal (Linux)

Abrimos nuestra terminal (en este caso usaremos ubuntu). El proceso de instalación es el mismo para cualquier distribución, donde cambia es el comando para ejecutar la instalación: por ejemplo Sudo apt-get, pacman, etc.

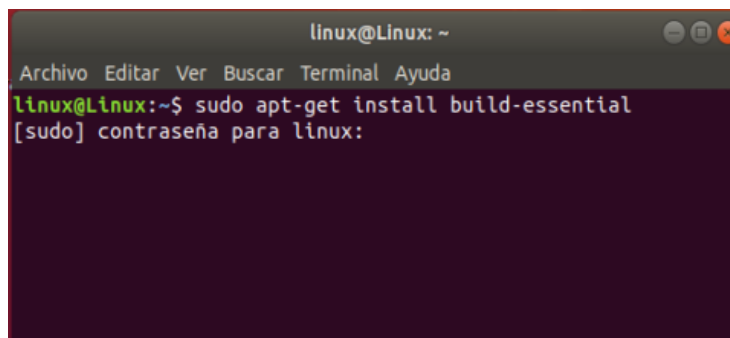


Una vez en la terminal escribimos lo siguiente:



```
1 sudo apt-get install build-essential
```

Si tienes otra forma de instalar paquetes en tu distribución tendrás que cambiar el "sudo apt-get" por el comando apropiado para instalar. Una vez escrito el comando damos Enter y se pedirá nuestra contraseña para proceder con la autorización de la instalación de los paquetes necesarios para poder compilar.



Al ingresar la contraseña correcta y dar Enter se inicia la instalación y nos pide si estamos de acuerdo con lo que se va instalar y con el espacio requerido.

```

linux@Linux: ~
Archivo Editar Ver Buscar Terminal Ayuda
Paquetes sugeridos:
  debian-keyring g++-multilib g++-7-multilib gcc-7-doc
  libstdc++6-7-dbg gcc-multilib autoconf automake libtool
  flex bison gcc-doc gcc-7-multilib gcc-7-locales
  libgcc1-dbg libgomp1-dbg libitm1-dbg libatomic1-dbg
  libasan4-dbg liblsan0-dbg libtsan0-dbg libubsan0-dbg
  libcilkrts5-dbg libmpx2-dbg libquadmath0-dbg glibc-doc
  git bzr libstdc++-7-doc make-doc
Se instalarán los siguientes paquetes NUEVOS:
  build-essential dpkg-dev fakeroot g++ g++-7 gcc gcc-7
  libalgorithm-diff-perl libalgorithm-diff-xs-perl
  libalgorithm-merge-perl libasan4 libatomic1
  libc-dev-bin libc6-dev libcilkrts5 libfakeroot
  libgcc-7-dev libitm1 liblsan0 libmpx2 libquadmath0
  libstdc++-7-dev libtsan0 libubsan0 linux-libc-dev make
  manpages-dev
Se actualizarán los siguientes paquetes:
  libdpkg-perl
1 actualizados, 27 nuevos se instalarán, 0 para eliminar y
76 no actualizados.
Se necesita descargar 27.1 MB de archivos.
Se utilizarán 117 MB de espacio de disco adicional después
de esta operación.
¿Desea continuar? [S/n] S

```

Se escribe S (como en la imagen) y presiona Enter para continuar con la instalación.

```

linux@Linux: ~
Archivo Editar Ver Buscar Terminal Ayuda
Configurando libalgorithm-diff-perl (1.19.03-1) ...
Procesando disparadores para man-db (2.8.3-2ubuntu0.1) ...
Configurando libc-dev-bin (2.27-3ubuntu1) ...
Configurando manpages-dev (4.15-1) ...
Configurando libc6-dev:amd64 (2.27-3ubuntu1) ...
Configurando libitm1:amd64 (8.3.0-6ubuntu1~18.04.1) ...
Configurando fakeroot (1.22-2ubuntu1) ...
update-alternatives: utilizando /usr/bin/fakeroot-sysv para
proveer /usr/bin/fakeroot (fakeroot) en modo automático
Configurando libgcc-7-dev:amd64 (7.4.0-1ubuntu1~18.04.1) ..
.
Configurando libstdc++-7-dev:amd64 (7.4.0-1ubuntu1~18.04.1)
...
Configurando libalgorithm-merge-perl (0.08-3) ...
Configurando libalgorithm-diff-xs-perl (0.04-5) ...
Configurando gcc-7 (7.4.0-1ubuntu1~18.04.1) ...
Configurando g++-7 (7.4.0-1ubuntu1~18.04.1) ...
Configurando gcc (4:7.4.0-1ubuntu2.3) ...
Configurando g++ (4:7.4.0-1ubuntu2.3) ...
update-alternatives: utilizando /usr/bin/g++ para proveer /
usr/bin/c++ (c++) en modo automático
Configurando build-essential (12.4ubuntu1) ...
Procesando disparadores para libc-bin (2.27-3ubuntu1) ...
linux@Linux:~$

```

Una vez terminado, ya podremos compilar y ejecutar archivos con terminación `.cpp`. Ahora nos dirigimos a la ubicación para guardar nuestro código. En este caso se guarda en Escritorio.

Para poder escribir nuestro código por la terminal tendremos que usar un editor de texto. El que viene ya instalado es *nano*. Así que para crear un archivo de

texto o de código en C++, se escribe “nano nombre_del_archivo .cpp”.

```

linux@Linux: ~/Escritorio
Archivo Editar Ver Buscar Terminal Ayuda
update-alternatives: utilizando /usr/bin/fakeroot-sysv para
proveer /usr/bin/fakeroot (fakeroot) en modo automático
Configurando libgcc-7-dev:amd64 (7.4.0-1ubuntu1-18.04.1) ..
.
Configurando libstdc++-7-dev:amd64 (7.4.0-1ubuntu1-18.04.1)
...
Configurando libalgorithm-merge-perl (0.08-3) ...
Configurando libalgorithm-diff-xs-perl (0.04-5) ...
Configurando gcc-7 (7.4.0-1ubuntu1-18.04.1) ...
Configurando g++-7 (7.4.0-1ubuntu1-18.04.1) ...
Configurando gcc (4:7.4.0-1ubuntu2.3) ...
Configurando g++ (4:7.4.0-1ubuntu2.3) ...
update-alternatives: utilizando /usr/bin/g++ para proveer /
usr/bin/c++ (c++) en modo automático
Configurando build-essential (12.4ubuntu1) ...
Procesando disparadores para libc-bin (2.27-3ubuntu1) ...
linux@Linux:~$ cd
linux@Linux:~$ ls
Descargas Escritorio Imágenes Plantillas Videos
Documentos examples.desktop Música Público
linux@Linux:~$ cd Escritorio/
linux@Linux:~/Escritorio$ ls
linux@Linux:~/Escritorio$ nano HolaMundo.cpp
linux@Linux:~/Escritorio$ nano HolaMundo.cpp
  
```

En seguida nos manda al editor, donde podemos escribir el código.

```

linux@Linux: ~/Escritorio
GNU nano 2.9.3 HolaMundo.cpp Modificado

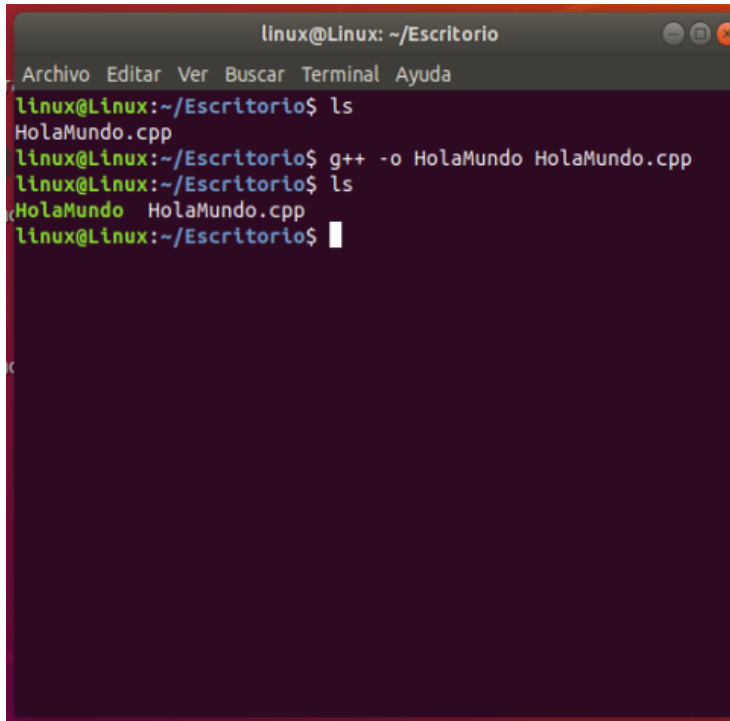
#include<iostream>
using namespace std;
int main()
{
    cout<<"Hola Mundo!";
    return 0;
}

eWriter

^G Ver ayuda ^O Guardar ^N Buscar ^K Cortar Texto
^X Salir ^R Leer fich. ^A Reemplazar ^U Pegar txt
  
```

Una vez escrito el código pulsamos las teclas “Ctrl” y “x”, y nos pregunta si queremos guardar, pulsamos “y” para hacerlo. De regreso en la terminal usamos el comando `ls` para enlistar nuestros archivos y podemos ver que nuestro archivo esta creado. Ahora compilamos el código con el siguiente comando.

```
1 g++ -o nombre_para_el_ejecutable nombre_del_codigo
```

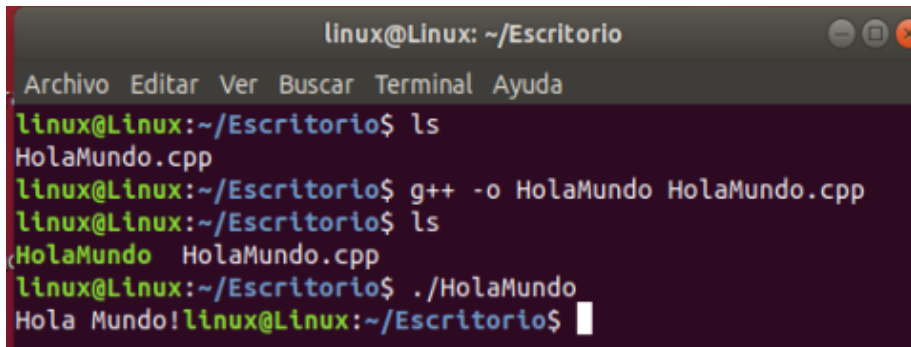


```
linux@Linux: ~/Escritorio
Archivo Editar Ver Buscar Terminal Ayuda
linux@Linux:~/Escritorio$ ls
HolaMundo.cpp
linux@Linux:~/Escritorio$ g++ -o HolaMundo HolaMundo.cpp
linux@Linux:~/Escritorio$ ls
HolaMundo  HolaMundo.cpp
linux@Linux:~/Escritorio$
```

Si el código está bien, la terminal no dará ningún mensaje adicional.

Volviendo a enlistar con `ls` se puede ver un nuevo archivo de color (en este caso aparece verde). Si te fijas bien, lleva el nombre que colocamos en *nombre_para_el_ejecutable*.

Para correr nuestro ejecutable, en la terminal escribimos “./nombre de nuestro ejecutable” y al dar Enter nuestro programa se ejecutará en la terminal, dando el siguiente mensaje:



```
linux@Linux: ~/Escritorio
Archivo Editar Ver Buscar Terminal Ayuda
linux@Linux:~/Escritorio$ ls
HolaMundo.cpp
linux@Linux:~/Escritorio$ g++ -o HolaMundo HolaMundo.cpp
linux@Linux:~/Escritorio$ ls
HolaMundo  HolaMundo.cpp
linux@Linux:~/Escritorio$ ./HolaMundo
Hola Mundo!linux@Linux:~/Escritorio$
```

Y así ha logrado hacer su primer programa en C++ en Linux!

3. Compilar y Correr el código

Para comprobar que podemos empezar a programar sin problemas, se recomienda copiar y pegar el siguiente código en el entorno de programación que se haya seleccionado:

```

1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     cout<<"Hola Mundo";
6
7 return 0;
8 }
```

En este caso veremos como hacerlo en CodeBlocks.

Abrimos nuestro IDE, y en la barra de herramientas daremos clic en File.

Debería de escribirse en la salida: *Hola Mundo*. La posición de la salida o este mensaje varían según lo que se seleccionó para programar. Independientemente de si se escribió conservando la jerarquía (espacios, tabulaciones y enters) o no.

Es importante que se haya escrito tal cuál el código, la excepción son el mensaje, la jerarquía y los espacios. Esto es porque la máquina (o computadora) necesita instrucciones precisas, al menos en c++, por lo que una mayúscula o espacios pueden cambiar por completo el sentido.

Nótese el siguiente ejemplo:

Código 1: hola mundo

```

1 #include <iostream>
2 using namespace std;
3
4 void Main(){
5     Cout<<"Hola Mundo"
6
7 return 0;
8 }
```

Este ultimo ejemplo al intentarse compilar va a generar errores, para empezar, el uso de la palabra “*void*” como tipo de “Main” no es valido, aunque algunos compiladores si lo aceptan, pero esto no es estándar y debe ser evitado por completo. El compilador se puede quejar por el **Cout** ya que al llevar mayúscula no es algo que C++ conozca, en si, no es lo mismo que con minúscula *Cout*, siendo esta ultima una palabra reservada para operar.

También *Main* debe empezar con minúscula, de lo contrario, no existe la parte principal del programa, que como su nombre en inglés lo indica, debe ser “*main*” exactamente. Y finalmente, toda oración debe terminar con punto y coma “;”, al no hacerlo el compilador marcará error, esta última delimita hasta donde corre las instrucciones, como si de un punto se tratara entre oraciones en español.

Entonces se debe tener cuidado al escribir el código, pero con la práctica se puede ir familiarizando con los errores que puedan existir, también se recomienda no confundir puntos y comas, así como números, esto es, se recomienda ser *muy* literal a la hora de escribir, ya que así es la computadora.

También hay que notar que se incluyen en este código `#include <iostream>` y `using namespace std;`, lo primero es una librería, y la segunda es una instrucción, hablaremos de ellas más adelante en la sección ?? ??, pero por el momento se recomienda colocar al inicio de cada código. **Capítulo 1**

Bases

Una vez que ya sabemos como empezar a usar el lenguaje C++, continuaremos con aprender a usarlo, entendiendo sus componentes básicos así como la forma en que pensar para dirigir nuestros pensamientos e instrucciones hacia el lenguaje.

1. Programación

¿Qué es? Actualmente se asocia con la creación de las instrucciones y materiales que usará un dispositivo para crear y correr trabajos.

Cada vez gana más popularidad la programación, por su pocos requerimientos para generar *lo que se pueda imaginar*, desde videojuegos y ocio, hasta cálculos científicos y económicos, entre otras aplicaciones.

2. Paradigmas de programación

En si los paradigmas son la forma de pensar y/o resolver los problemas que requieren de programación. Unos lenguajes pueden tomar distintos paradigmas a la vez, mientras que otros se restringen a solo uno.

De los más populares se encuentran:

- **Paradigma Estructurado.**
- **Paradigma Orientado a Objetos.**
- Paradigma Orientado a Eventos.
- Paradigma Funcional.

Los dos paradigmas marcados en negritas son populares de usar en C++, aunque según nuevas versiones del lenguaje, se pueden usar otros paradigmas más, lo que vuelve a C++ muy versátil. Dado que este manual es introductorio, solo hablaremos del Paradigma estructurado.

3. Paradigma Estructurado

Este paradigma es de los más comunes, se basa en la secuenciación de instrucciones, como si de un recetario o instructivo se tratara. Esta forma de organizar el programa sigue en sí una “estructura”, de ahí el nombre. Podemos leer e interpretar el código como una lectura en español, como los ejemplos mencionados. Esto es que el código se lee de izquierda a derecha, y de arriba a abajo. Posteriormente en [2 Secuencias de control básicas](#) se mencionaran formas de modificar un poco esta estructura, de tal manera que nos permita realizar más acciones sin escribir tanto.

4. Empezando a programar en C++

Como se vio en la sección [3 Compilar y Correr el código](#), puede usarse cualquier medio para programar, siempre que tenga la misma versión (o mayor en muchos lenguajes, incluyendo C++), para propósitos de introducción y permitirle al lector programar desde donde este (inclusive dispositivos móviles), el código mostrado en este manual ha sido compilado y ejecutado en el sitio: https://www.onlinegdb.com/online_c++_compiler.

Para empezar se mostraran las partes principales de un programa en C++ con el ejemplo clásico de “Hola Mundo”:

Código 1.1: Hola mundo

```

1  /* Este es un comentario que se contiene
2     dentro de los simbolos delimitadores,
3     sin importar cuantas lineas se tengan.
4     Los comentarios son ignorados por el
5     compilador, pero sirven para hacer
6     notas sobre el codigo.
7     */
8  //Este es un comentario de linea, funciona solo en la misma linea
9
10 #include <iostream> //esto es una biblioteca
11 using namespace std; //indica que usamos funciones con prefijo std::
12
13 //C++ siempre necesita la siguiente funcion para correr
14 int main(){ /*Entre () se tienen los datos que toma la funcion,
15     conocidos como argumentos*/
16
17     cout << "Hola Mundo"; /*Esta linea escribe(imprime) en pantalla
18         "Hola Mundo" */
19
20     return 0;

```


19 } //Los corchetes van desde dentro hasta afuera. {3{2{1}2}3}

Como se menciona en el código, los comentarios (Texto en verde) son ignorados por el compilador, lo cual es útil para que los humanos podamos dejar información sobre que hacen partes del código. Sin embargo, los acentos o caracteres fuera del abecedario simple pueden aparecer extraños o modificarse, por lo que se recomienda no usarlos.

Se considera buena práctica comentar el código que uno realiza, así puede saber para que era cada parte si se llega a olvidar posteriormente o se quiere revisar.

Se deja como ejercicio al lector compilar y ejecutar el programa eliminando los comentarios, para que pueda observar como se produce el mismo resultado que si se dejan. **Nótese** que lo que no esta de color verde (no es un comentario) lleva mayúsculas y minúsculas, así como puntos y orden, si se reemplazan, omiten o eliminan/agregan cosas, entonces el programa marcara errores. Véase la sección [3 Compilar y Correr el código](#) para mas información sobre algunos errores comunes.

La estructura `cout <<` debe cumplir con tener a la derecha un tipo de dato que pueda ser escrito, veremos la mayoría de estos en [5 Tipos de datos básicos](#). Al colocar algo entre " " estamos diciendo que todo lo de adentro es un arreglo de caracteres (incluyendo los espacios), podemos juntarle (o concatenarle) al texto otros datos con el signo (operador) `<<` que también puedan representarse en texto, por ejemplo: `cout << " Un texto y el numero "<< 3.4 << " aqui."`. Imprimirá en pantalla: *Un texto y el numero 3.4 aqui..*

Nótese que los espacios entre " " se conservaron en la frase final.

Estos textos también pueden decirse que son Strings, de las cuales hablaremos mas adelante en [2 Memoria](#).

5. Tipos de datos básicos

Para almacenar y usar datos, debemos declarar de que tipo son, para que la computadora conozca como operarlos, a continuación se presentan los mas generales y comunes en la mayoría de los lenguajes de programación:

- **bit**: unidad de almacenamiento mínima, puede valer 1 o 0.
- **byte**: unidad compuesta por 8 bits.
- **char** (carácter): puede valer 1 a 2 bytes, representa símbolos o letras.
- **int** (entero): 2 a 4 bytes, puede representar números enteros.
- **float** (punto flotante): 4 bytes, representa números reales (con decimales).

- **double** (doble punto flotante): 8 bytes, representa números reales con mayor precisión.

El lenguaje C++ incluye otros tipos de datos que brindan mayor utilidad, se mencionaran en el código que sigue pero se dejara su aplicación individual para el gusto y la necesidad del lector. Dado que los tamaños de cada dato pueden variar a veces por la arquitectura de la computadora y mas que nada por el lenguaje de programación, a continuación se muestra un ejemplo de código¹ que imprime los tamaños de cada dato básico en el lenguaje C++ que se usa en este manual para familiarizar al lector:

Código 1.2: Tipo de Datos

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     cout << "Tamanios en bytes: " << endl;
6     cout << " char: " << sizeof (char) << endl;
7     cout << " short: " << sizeof (short) << endl;
8     cout << " int: " << sizeof (int) << endl;
9     cout << " long: " << sizeof (long) << endl;
10    cout << " long long: " << sizeof (long long) << endl;
11    cout << " float: " << sizeof (float) << endl;
12    cout << " double: " << sizeof (double) << endl;
13    cout << " long double: " << sizeof (long double) << endl;
14    return 0;}
```

Ahora observe como cambia la representación de cada valor al ser impreso según su tipo de dato.

Código 1.3: Datos impresos

```
1 #include <iostream>
2 #include <iomanip> /*libreria para modificar las impresiones en
   pantalla*/
3 using namespace std;
4
5 int main(){
6     char a=42;
7     int b=42;
8     float c= (float) (41 + 0.9999998);
9     double d=41 +0.9999998;
10    cout << "char a = " << a <<endl;
11    cout << "int b = " << b <<endl;
12    cout << setprecision(10); /*especifica con que precision escribir
   numeros*/
13    cout << "float c = " <<c <<endl;
14    cout << "double d = " << d <<endl;
```

¹Código compilado y ejecutado en https://www.onlinegdb.com/online_c++_compiler

```

15
16 return 0;}

```

Del código se puede observar que usamos algo nuevo en esta línea:

```

1 float c= (float) (41 + 0.9999998);

```

Aquí (*float*) indica a la computadora que queremos convertir un tipo de dato cualquiera a un tipo *float*, esto es una conversión explícita, hablaremos a detalle de estas en [7 Conversiones](#), básicamente sirve para lograr asignar el valor y su tipo deseado a la variable *c*.

Además obsérvese como *float* recorta dígitos que en algunas situaciones nos podrían ser útiles, así que es importante saber con que propósito se usará la aplicación, para saber si vale la pena el peso extra que conlleva un *double*, o un *float* es suficiente.

Experimente un poco cambiando el valor de *a* para observar que caracteres puede llegar a escribir. También usa los otros tipos que faltan del código [1.2](#).

6. Funciones

Las funciones son una secuencia de instrucciones que se encierran con un nombre. Estas nos dan varias ventajas:

- Reusabilidad del código: podemos repetir esa secuencia de instrucciones siempre que queramos con solo llamarla.
- Modularización: separamos en partes la lógica del programa, lo cual facilita identificar en que parte se genera un problema y facilita la solución.
- Claridad: entendemos mejor que debe hacer una sección del código, se recomienda usar nombres claros y comentar las partes del código.

La estructura de una función es:

```

tipoDatoResultado nombre(tipoDato parametro1, tipoDato pm2, ...)
{
    /*Cuerpo de la función*/
    return resultado; //si tipoDatoResultado no es void
}

```

Observa el siguiente ejemplo:

Código 1.4: Funciones

```

1 #include <iostream>
2
3 using namespace std;
4
5 int nombre_funcion(int parametro)

```

```

6 {
7     return parametro*parametro;
8 }
9
10 int main(){
11     cout<< nombre_funcion(3);
12
13     return 0;
14 }

```

En este ejemplo creamos una función de tipo int (entero) que también toma un parámetro de tipo int, y regresa el mismo parámetro multiplicado por si. Este resultado que regresa se puede usar como tal, ya sea en otras funciones o partes del código como se observa en:

```

1     cout<< nombre_funcion(3);

```

Las funciones siempre deben ir declaradas antes de donde se usan, esto es, que al menos se debe conocer su nombre. Observa los siguientes códigos y prueba a ver cual de los dos funciona:

Código 1.5: Funcion 1

```

1 #include <iostream>
2
3 using namespace std;
4
5 int main(){
6     cout<< nombre_funcion(3);
7
8     return 0;
9 }
10
11 int nombre_funcion(int parametro){
12     return parametro*parametro;
13 }

```

Código 1.6: Funcion 2

```

1 #include <iostream>
2
3 using namespace std;
4
5 int nombre_funcion(int parametro);
6
7 int main(){
8     cout<< nombre_funcion(3);
9
10    return 0;

```

```
11 }
12
13 int nombre_funcion(int parametro){
14     return parametro*parametro;
15 }
```

Se recomienda usar la estructura en 1.6 para funciones extensas, así se facilita leer primero la función main (principal) y después navegar hacia las funciones específicas cuando sean llamadas.

Cabe notar, que las variables ingresadas como parámetros son **copiadas**, esto es, que hagamos lo que sea, el valor original que mandamos a la función no se vera afectado, a menos que se usen apuntadores, hablaremos más de ellos en 3 Apuntadores.

6.1. Declarar y definir

Mencionamos brevemente que era declarar, pero ahora veamos que esto se aplica tanto a funciones como a otros elementos. Observa el siguiente código e intenta identificar cual es una declaración y cual una definición.

Código 1.7: Declarar y Definir

```
1  #include <iostream>
2  using namespace std;
3
4  int funcion_a(int a){ //de
5      return a*a;
6  }
7  int funcion_b(int b); /*de*/
8
9  int main(){
10     int a; /*de*/
11     a = 3;
12     int b = 4;
13
14     cout<< funcion_a(b) << endl;
15     cout<< funcion_b(a);
16
17     return 0;
18 }
19
20 int funcion_b(int b){ return b+b; }//de
```

Como te podrás haber dado cuenta, los comentarios con estrella son declaraciones, mientras los que no la tienen son definiciones. Observa que si eliminas una definición, el programa fallara, esto es porque si bien puede identificar el elemento, no sabe que hacer con él.

En el caso de variables, se pueden declarar pero al asignarles un valor (por primera vez) es *inicializarlas*, al contrario de una definición, puedes usar una

variable si no ha sido inicializada, pero contendrá información basura, o bien pues números al azar que ya estaban en ese espacio de memoria que pidió tu programa para almacenar la variable.

6.2. Procedimientos

En la estructura de una función se menciona que se puede omitir el resultado si el tipo de la función es *void*, este tipo de dato es especial al indicar que no se espera nada, es nulo y/o vacío. Por lo que la función es en realidad un *procedimiento*, esto es, que realiza acciones sin tener un valor de salida, puede o no tomar valores. Observa el siguiente ejemplo:

Código 1.8: Funcion con Void

```
1 #include <iostream>
2
3 using namespace std;
4
5 void hola(){
6     cout << "Hola";
7 }
8
9 int main(){
10     hola();
11
12     return 0;
13 }
```

Si bien los procedimientos no regresan valores, pueden usar *return* cuando se quiere salir antes de la función, por ejemplo:

Código 1.9: Funcion con Void y return

```
1 #include <iostream>
2
3 using namespace std;
4
5 void hola(int num){
6     if(num < 6) return;
7     cout << "Numero: " << num;
8 }
9
10 int main(){
11     hola(5);
12
13     return 0;
14 }
```

El “if ” usado en la linea:

```
1  if(num < 6) return;
```

Es una secuencia de control, en si controlan el flujo del programa, hablaremos más de ellas en [2. Secuencias de control básicas](#).

7. Conversiones

Anteriormente en [5 Tipos de datos básicos](#) se menciona que se realizaba una conversión de un tipo de dato a otro. Tal como el nombre lo indica, las conversiones de tipo (*type conversion*) son el proceso de cambiar un tipo de dato en otro. En C++ este procedimiento se puede llevar a cabo de dos formas que se mencionan en las siguientes subsecciones.

7.1. Conversiones Implícitas

Como el nombre lo indica, estas conversiones se llevan a cabo de forma automática cuando es necesario por el compilador. Por ejemplo cuando se espera un tipo de dato, pero se recibe otro, o al querer evitar perder datos. Observa el siguiente programa de ejemplo e identifica cuando se realiza una conversión implícita:

Código 1.10: Conversiones Implícitas

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6      int a = 5.1;
7      float b = a;
8      cout << " a/2= " << a/2 << endl;
9      cout << " b/2= " << b/2 << endl;
10     cout << " (5/3)*b= " << (5/3)*b << endl;
11
12     return 0;
13 }
```

Ahora presta atención a esta línea:

```
1  cout << " (5/3)*b= " << (5/3)*b << endl;
```

Si corriste el código te puedes dar cuenta que la ecuación no dará el resultado esperado, esto puede complicar las cosas y llevar a errores minuciosos en ciertas ocasiones, ya que el compilador en expresiones siempre intentara hacer coincidir un valor operado con otro, de lo contrario intentara convertir el tipo del dato mas bajo en bytes al tipo del mas grande. Para permitir al programador tener la libertad de especificar que dato se espera, se pueden hacer las conversiones de a continuación.

7.2. Conversiones Explícitas

Estas son especificadas por el programador y se aseguran de tener el dato esperado. Existen varias formas de hacer este procedimiento que se mencionan a continuación:

Conversiones estilo C

Como el nombre lo indica, vienen del lenguaje original c, se realizan colocando el nombre del tipo de dato que se desea entre paréntesis y a la derecha se encuentra el valor a ser convertido. Alternativamente se puede escribir como una llamada a una función, donde el parámetro que toma es el tipo de dato a convertir. Observa el siguiente ejemplo y presta atención al orden en como se escribe:

Código 1.11: Conversiones Explícitas

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main(){
6     cout << (float)5/2 << endl;
7     cout << float(5/2) << endl;
8     cout << float(5)/2 << endl;
9
10    return 0;
11 }
```

Presta atención a como se debe ser bien específico al usar estas conversiones para evitar los mismos errores que obtendrías al omitirlos. El problema con este tipo de conversión es que no se revisan, por lo que pueden obtenerse errores difíciles de rastrear.

Conversiones con static_cast En c++ se tiene un operador más seguro llamado “*static_cast*”, el cual verifica que no se cometan cosas absurdas en la operación de un programa, este funciona mas como una función, donde toma el valor a ser convertido entre paréntesis y el tipo de dato al cual convertir entre “< >”. Observa el siguiente código:

Código 1.12: Conversiones Explícitas con static_cast

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     int a=66;
6     char b = static_cast<char>(a);
7     cout << "b = " << b << endl;
8
9     return 0;}
```


Este tipo de conversión da claridad y seguridad al código por lo que se recomienda utilizarla por lo general.

Aunque si bien existen otros tipos, estos se consideran muy específicos y fuera del objetivo de estas lecciones, por lo que se omiten.

Capítulo 2

Palabras reservadas

Hemos entendido hasta ahora algunos componentes, pero ¿Qué pasa con algunas palabras que hemos estado viendo como “include”?, estas se conocen como *palabras reservadas* del lenguaje C++, y como el nombre lo indica, tienen un propósito ya definido por el lenguaje.

Algunos que hemos utilizado son los nombres de los tipos de datos: *int*, *char*, *double*, *float*. Como se nota en el código, estas palabras se marcan de un color distintivo, en este manual se usa el color azul.

Cada palabra puede tener un significado y uso distinto, a continuación se muestran algunos de los más comunes, excluyendo los mencionados anteriormente que ya hemos visto.

- **include**: incluir, sirve para traer variables, clases, funciones entre otros elementos que se encuentran en librerías, se hablara más de estas en [1 Librerías](#).
- **using**: usando, como en los ejemplos presentados hasta ahorita, se suele usar antes de la palabra *namespace* y el nombre de la librería (en el código) para evitar escribir el nombre y el operador antes de los elementos de la librería, no se recomienda usar esto mas que en códigos cortos, y aun así se debe evitar ya que previene recordar de donde viene cada elemento, por lo que pueden causarse conflictos.
- **namespace**: permite englobar funciones, variables, y demás elementos en un nombre común con el cual se puedan usar sin interferir con otros elementos con los mismo nombres.
- **public**: publico, declara la variable o método (función) como publica, esto es que cualquier parte del programa puede acceder a ella, se asocia con el concepto de encapsulamiento que corresponde al Paradigma Orientado a Objetos, el cual no entra en el nivel de este manual.
- **private**: privada, declara la variable o método como privada, así solo la clase u objeto que lo contiene puede usarlo, se asocia con el concepto de encapsulamiento. Cuando una variable o función no tiene public, private o protected, se toma private por defecto.
- **protected**: protegido, similar a private, pero los hijos de la clase pueden acceder a esta parte, se asocia con el concepto de encapsulamiento y herencia.

- **static**: estático, establece que solo puede existir una declaración y definición de esta variable o función en el programa.
- **void**: vacío, sirve para declarar que no se espera un valor, una función que lo tiene antes indica que es un proceso, ya que realiza acciones pero no regresa un valor al terminar.
- **return**: regresar, se coloca antes del valor a regresar, el cual debe ser un tipo de dato similar al que se declaró antes del nombre de la función. Marca la salida de la función.
- **break**: termina antes la secuencia de control en la que se encuentra, se explica más sobre estas en [2 Secuencias de control básicas](#).
- **continue**: continuar, se brinca a la siguiente iteración de la secuencia de control actual.
- **auto**: automático, se usa para interpretar automáticamente el tipo de dato, su uso sale del alcance de este manual.
- **new**: nuevo, se utiliza para crear nuevos elementos como memoria dinámica, estos se deben eliminar con *delete*.
- **delete**: borrar, libera la memoria de elementos dinámicos creados con *new*.
- **try**: intenta realizar lo que le siga entre paréntesis, este y las siguientes 2 palabras son usadas para función o partes de código que pueden causar errores.
- **catch** : atrapa, sirve para declarar que hacer cuando en el código anterior dentro de try tira un error.
- **throw**: aventar, regresa un tipo de error manejable cuando sucede una condición.

1. Librerías

Las librerías son colecciones de código, donde hay tipos de datos, objetos y métodos entre otras cosas, que en lo general se engloban por su utilidad. Previamente se mencionó que la palabra reservada *using* lleva en la sintaxis el nombre de la librería, pero en realidad es su *header* (encabezado), los headers son archivos que contienen las **declaraciones** de los objetos, son como código pero sin un propósito más que para enlazar la funcionalidad de otros códigos. En el momento que se coloca el *include* y el *header* se copian todos los contenidos de este archivo al código que estamos usando, pero lo mantiene limpio para la vista del programador con una simple oración. Esto da la ventaja de tener archivos menos pesados, ya que las definiciones de los elementos de la librería se encuentran en otro archivo.

Los elementos de las librerías se suelen llamar con su *namespace* y un par de dobles puntos (el operador “::” mencionado al inicio de la tabla [3.1](#)), todas

las librerías incluidas en C++ usan el *namespace* llamado *std*, que es corto para *standard* (estándar). Existen varias librerías en C++, pero a continuación mencionamos por su *header* las más usadas:

- **iostream**: librería básica para C++, permite escribir, recibir información, entre otras cosas.
- **iomanip**: permite darle formato a la salida y entrada de datos.
- **cstdlib**: es el estándar del lenguaje C, permite usar muchas de las funciones contenidas ahí.
- **cstdio**: entrada y salida de datos en estilo C.
- **ctime**: permite usar funciones de tiempo en el estilo de C.
- **chrono**: utilidades de tiempo en C++.
- **cstring**: librería de string en estilo C.
- **array**: permite usar el tipo *array* para memoria.
- **string**: permite usar el tipo *string* para arreglos de caracteres en estilo C++.
- **vector**: permite usar el tipo *vector* para memoria dinámica.
- **map**: permite usar el tipo *map* para el mapeo de datos.
- **algorithm**: librería de funciones y operaciones con elementos, así como varios algoritmos.
- **random**: contiene utilidades para números aleatorios.

Como puedes observar hay varias librerías del lenguaje C, y no es coincidencia, ya que C++ permite usar varias funcionalidades del primero, pero es importante **no confundirlos** ya que sí contienen sus diferencias cruciales y la compatibilidad de un código escrito en C con otro en C++ y viceversa puede ser coincidencia.

Dado que ya hemos explicado las librerías y de donde vienen varias funciones que hemos estado usando, de ahora en adelante no usaremos la línea “*using namespace std;*”, para inculcar una **buena práctica de programación**, así como un nuevo reto al lector donde tendrá que identificar de donde viene cada parte del programa que hace.

Capítulo 3

Operadores y secuencias de control

Para asociar valores unos con otros, como por ejemplo sumas, relaciones, lógica, etc... , requerimos hacer operaciones tal como en matemáticas, es de esta

manera como se definen signos y palabras, conocidos como **operadores**. Esta lógica de relacionar también nos permite darle un sentido y dirección a nuestros programas, como por ejemplo tomar en cuenta casos, o inclusive hacer menús de selección en base a lo que nos de el usuario.

Este capítulo es importante por dichas razones, por lo que se le invita al lector a poner buena atención e intentar practicar lo aprendido, así como verificar que todo sea cierto, y no tomarlo por garantizado.

1. Operadores y su tabla de precedencia

En C++ para hacer operaciones se necesita un orden en el que se hagan, como en las matemáticas, así que en C++ los operadores con misma precedencia son evaluados de izquierda a derecha, excepto los de prioridad 3, 15 y 16, que a continuación se presentan en una tabla con el orden de prioridad que siguen los operadores, siendo 1 la más alta:

Orden	Operador	Descripción
1	::	Operador de resolución de alcance
2	++ -- tipo() tipo{} () () [] . -> typeid const_cast dynamic_cast reinterpret_cast static_cast	Operadores posfijos de decremento e incremento Conversión funcional Función Parentesis Subíndice de arreglo acceso a miembros Información del tipo de dato Proyección sobre constante Conversión verificando tipo Conversión de un tipo a otro Conversión revisada en compilación
3	+ + -- + - ! ~ * & (tipo) sizeof new new[] delete delete[]	Prefijos de incremento y decremento Mas y menos unarios Negación lógica Operador negación bit a bit Operador desreferencia o indirección Operador referencia o de dirección Conversión tipo C Operador de tamaño en bytes Asignación dinámica de memoria Liberación de memoria dinámica
4	->* .*	Apuntador a miembro
5	* / %	Operadores binarios de multiplicación, división y modulo

Continúa en la siguiente página. . .

Tabla 3.1 – Continuación

Orden	Operador	Descripción
6	+ -	Operadores binarios de adición
7	<< >>	Operadores binarios de shift o desplazamiento en bits
8	< > <= >=	Operadores relacionales
9	== !=	Operadores de equidad
10	&	Operador bit a bit AND (y)
11	^	Operador bit a bit OR exclusivo (xor/o exclusivo)
12		Operador bit a bit OR (o)
13	&&	Operador AND (y) logico
14		Operador OR (o) logico
15	? :	Operador ternario (note que lleva misma precedencia que los de asignación)
15	= += -= *= /= %= &= ^= = <<= >>=	Operadores de asignación
16	throw	Operador de throw o arroje
17	,	Operador coma

Tabla 3.1: Los operadores y la descripción. Las fuentes consultadas aparecen en las referencias [2] y [1].

Si bien se observan muchos operadores, en realidad necesitamos solo unos cuantos al iniciar, de los cuales ya hemos visto algunos como + y =. Para tener una buena idea de como se pueden usar véanse los siguientes temas sobre los principales operadores que se contemplan en el nivel de este manual.

1.1. Operadores aritméticos

Tal como su nombre lo indica son aquellos que conocemos por operaciones matemáticas, el único que nos puede parecer extraño es el % el cual, como se menciona en la tabla 3.1 es el operador “modulo”, en si divide un número a entre otro b y regresa el residuo de esta división. Veremos como funcionan en el siguiente ejemplo en el cual usaremos el operador = que se menciona a detalle en la sección 1.7 Operadores de asignación, por el momento podemos decir que funciona para “asignar” o dar un valor a una variable. Observe el siguiente ejemplo:

Código 3.1: Operadores aritméticos

```

1 #include <iostream>
2 //using namespace std;
3
4 int main(){

```

```

5  int a= 7, b=3; // se asigna un valor con =
6  std::cout << "a + b = " << (a + b) << std::endl; //operador binario
    suma
7  std::cout << "a - b = " << (a - b) << std::endl; //op bin resta
8  std::cout << "a * b = " << (a * b) << std::endl; //op bin
    multiplicacion
9  std::cout << "a / b = " << (a / b) << std::endl; //op bin division
10 std::cout << "a % b = " << (a % b) << std::endl; //op bin modulo
11
12 return 0;
13 }

```

Como se menciono anteriormente, las operaciones se realizan de izquierda a derecha, ahora **experimenta** agrega otra variable en las operaciones para ver como se afecta el orden de las mismas (no olvides incluir el operador entre las variables).

Estos operadores también funcionan con otros tipos de datos básicos que vimos, por ejemplo **experimenta** cambiando el int por un double y observa las salidas, también cambia los valores de a y b a negativos (usa el operador unario - como prefijo en el valor a asignar a la variable).

1.2. Operadores unarios

Son aquellos que aplican sobre una sola variable. De los operadores de incremento y decremento, es importante marcar la diferencia entre los posfijos y prefijos como se muestra en la siguiente lista:

- ++ : Operador de incremento, aumenta el valor de la variable en 1.
 - *Postfijo*: primero evalúa la variable y luego incrementa su valor.
 - *Prefijo*: primero incrementa su valor y luego evalúa la variable.
- -- : Operador de decremento, reduce el valor de la variable en 1.
 - *Postfijo*: primero evalúa la variable y luego reduce su valor.
 - *Prefijo*: primero reduce su valor y luego evalúa la variable.
- - : Da el valor negativo. Si era negativo será positivo.
- + : Da el valor aritmético. Su uso es muy raro pero puede llegar a servir como se mostrará en un ejemplo mas adelante.
- ! : Niega el valor de una variable, se usa con “booleanos” pero también funciona con números como se muestra en el ejemplo 3.2.
- ~ : Niega bit a bit los valores de una variable, este sera explicado a detalle en 1.5 Operadores bit a bit.

Código 3.2: Operadores unarios

```

1  #include <iostream>
2  //using namespace std;
3
4  int main(){
5      int a= -1, b= 0, c= 0;
6      bool d = true;
7      std::cout << "Operadores postfijos" << std::endl;
8      std::cout << "a++ = " << (a++) << std::endl;
9      std::cout << "a = " << a << std::endl;
10     std::cout << "a-- = " << (a--) << std::endl;
11     std::cout << "a = " << a << std::endl;
12     std::cout << "\nOperadores prefijos" << std::endl;
13     // el \n anterior es una secuencia de escape
14     std::cout << "b = " << b << std::endl;
15     std::cout << "++b = " << (++b) << std::endl;
16     std::cout << "b = " << b << std::endl;
17     std::cout << "--b = " << (--b) << std::endl;
18     std::cout << "b = " << b << std::endl;
19     //Observa bien el siguiente enunciado
20     c = a++ + ++b;
21     std::cout << "\nc = " << c << std::endl;
22
23     std::cout << "\nOperador unario !" << std::endl;
24     std::cout << !3 << " == " << !d << std::endl;
25
26     return 0;
27 }

```

La secuencia de escape “\n” y otras más se explican en la sección 1 [Secuencias de escape](#). El tipo de dato booleano “bool” se explica en la siguiente sección. Similarmente al ejemplo anterior, **experimenta** cambiando de lugar la variable *a* con la *b* para ver como afecta el orden a las operaciones que le dan un valor a *c*.

Experimenta cambiando el int por un double y observa las salidas, también cambia los valores de *a* y *b*.

1.3. Operadores relacionales y los booleanos

Los operadores relacionales son aquellos que analizan una relación y resuelven el si esta es verdadera o falsa.

Mencionando esta condición de verdad o falsa, también se presenta un nuevo tipo de dato: el booleano. Cuyo nombre viene del matemático *George Boole* quien creo un álgebra basada completamente en 1 y 0, o bien pues verdadero o falso, entre otros posibles nombre.

En C++ el tamaño en bytes de un dato de tipo booleano depende mucho de la implementación, esto es que varia, por lo general se marca como 1 byte, pero no se puede garantizar, ya que puede verse afectada por cosas como la arquitectura

de la computadora entre otros detalles. Lo que se mantiene cierto es que un `bool` solo puede tomar 1 valor a la vez, el cual se le puede asignar con un valor, siendo el 0 falso y el resto verdadero, o usando las palabras reservadas *true* y *false*. Se hablara más de estas palabras en la siguiente sección [4 Pensando en código](#).

Conociendo esto se explican las características de cada operador:

- `<` : verifica que el valor de la izquierda es menor que el de la derecha.
- `>` : revisa que el valor de la izquierda es mayor que el de la derecha.
- `==` : analiza si el valor de la izquierda es igual al de la derecha. Es importante **no confundir** este operador con el `=` de **asignación**.
- `!=` : checa que el valor de la izquierda es diferente al de la derecha.
- `<=` : verifica que el valor de la izquierda es menor o igual al de la derecha.
- `>=` : revisa que el valor de la izquierda es mayor o igual al de la derecha.

Observa el siguiente código para tener una mejor idea de los booleanos.

Código 3.3: Operadores relacionales

```
1 #include <iostream>
2 //using namespace std;
3
4 int main(){
5     int a= 7, b=3;
6     std::cout << " a < b = " << ( a < b ) << std::endl
7         << " a > b = " << ( a > b ) << std::endl
8         << " a == b = " << ( a == b ) << std::endl
9         << " a != b = " << ( a != b ) << std::endl
10        << " a <= b = " << ( a <= b ) << std::endl
11        << " a >= b = " << ( a >= b ) << std::endl;
12
13     return 0;
14 }
```

Como se observa, los resultados devueltos son de tipo booleano.

Ahora **experimenta** con los valores de `a` y `b` para ver como se analizan de forma diferente.

1.4. Operadores lógicos

Estos operadores son aquellos que se encargan de la lógica y las evaluaciones, vimos brevemente en [1.2 Operadores unarios](#) al operador negación `!`. Observa las tablas de a continuación para ver como se evalúan según los operandos, donde V es Verdadero y F es Falso.

a	b	
F	F	F
V	F	V
F	V	V
V	V	V

a	b	&&
F	F	F
V	F	F
F	V	F
V	V	V

a	!a
F	V
V	F

Tabla 3.2: Valores de || **Tabla 3.3:** Valores de && **Tabla 3.4:** Valores de !

Ejecuta el siguiente código para ver el comportamiento de estos operadores.

Código 3.4: Operadores lógicos

```

1 #include <iostream>
2 //using namespace std;
3
4 int main(){
5     bool a= true, b=false;
6     std::cout << " a && b = " << ( a && b ) << std::endl
7     << " a || b = " << ( a || b ) << std::endl
8     //ahora usando el operador de negacion
9     << " !b = " << ( !b ) << std::endl
10    << " a && !b = " << ( a && !b ) << std::endl
11    << " !a || b = " << ( !a || b ) << std::endl;
12
13    return 0;
14 }
```

Ahora **experimenta** cambiando los valores de a y b.

1.5. Operadores bit a bit

Estos operadores pueden parecer un poco complicados, pero su funcionamiento es similar a la de los operadores lógicos, solo que los números se operan en su representación en base binaria.

Cuando mencionamos los tamaños de bytes en [5 Tipos de datos básicos](#) de cada dato, también se mencionó que se componían de bits, estos se suelen analizar de derecha a izquierda, se cuentan empezando desde 0, y si tienen valor de 1 se eleva 2 al valor de su posición y se suma al total que vale.

Por ejemplo: 1101, de derecha a izquierda tenemos un 1 en la posición 0, un segundo 1 en la posición 2, y un ultimo 1 en la posición 3, entonces el valor de 1101 sera $2^0 + 0 + 2^2 + 2^3 = 1 + 4 + 8 = 13$. Conociendo esto se enlista como funciona cada operador, tomando en cuenta que como su nombre lo marca, operan cada **bit** de *a* por otro **bit** de *b* en la misma posición, el número en binario que obtienen al final es el que regresan.

- **&** : hace la misma operación que el operador && pero con cada bit.
- **|** : hace la misma operación que el operador || pero con cada bit.
- **^** : regresa 1 *solo si* los 2 bits que esta analizando son diferentes.

- `~` : aplica la operación de ! en cada bit de la variable..

Estas operaciones son muy útiles cuando se lidia con aspectos delicados, como ahorrar memoria al operar verdaderos y falsos sobre un solo valor, así como otras artimañas avanzadas. En el siguiente programa se muestran las operaciones y sus representaciones, no te preocupes si no entiendes la función *imprimir_binarios* ya que es algo avanzada, solo concéntrate en entender las operaciones de *main*.

Código 3.5: Operadores bit a bit

```

1 #include <iostream>
2 //using namespace std;
3
4 void imprimir_binarios(size_t tamano, void* valor);
5
6 int main(){
7     char a=5, b=13, c;
8     std::cout << " a = " ;
9     imprimir_binarios(sizeof a, &a);
10    std::cout << " b = " ;
11    imprimir_binarios(sizeof b, &b);
12    std::cout << " a & b = " << (a&b) << " = ";
13    c = a&b;
14    imprimir_binarios(sizeof c, &c );
15    std::cout << " a ^ b = " << (a^b) << " = ";
16    c = a^b;
17    imprimir_binarios(sizeof c, &c );
18    std::cout << " a | b = " << (a|b) << " = ";
19    c = a|b;
20    imprimir_binarios(sizeof c, &c );
21    std::cout << " ~a = " << ~a << " = ";
22    c = ~a;
23    imprimir_binarios(sizeof c, &c );
24
25    return 0;
26 }
27
28 void imprimir_binarios(size_t tamano, void* valor) {
29     unsigned char *allbytes = (unsigned char*) valor; //posicion en
30     bytes(referencia a direccion de memoria del valor)
31     unsigned char bit; //unsigned concentrara todo a solo los bits sin
32     importar el signo, y por tamano de 1 byte
33
34     for(int i= tamano-1;i>=0;i--){ //for para todos los bytes
35         for(int j=7;j>=0;j--){ //for para cada octeto
36             bit = (allbytes[i]>>j) & 1; //movemos el byte para comparar esa
37             posicion con 1
38             std::cout << (bit == 0 ? 0 : 1);
39         }
40     }
41 }

```

```

37     std::cout << " ";
38 }
39     std::cout << std::endl;
40 }

```

Observa que el operador \sim regreso un valor negativo, esto es por la forma en que se representan los valores negativos en binario, donde al tener una cantidad de bits limitados, se optó por dividir la cantidad de números representables a la mitad más uno para los negativos.

Como muestra la representación binaria gracias a la función *imprimir_binarios*, no cuesta más que invertir **todos** los bits que contiene el tipo de dato, para obtener el valor original en negativo menos 1 extra.

Experimenta cambiando los valores de a y b para ver como cambian los resultados, así como su tipo de dato, e intenta hacer la representación a mano de los números y las operaciones en binario para tener una mejor idea de como funcionan.

1.6. Operadores de desplazamiento de bits o shift

Estos operadores actúan sobre la representación en binario del tipo de dato que usemos, siendo cada uno un poco diferente:

- \ll : Mueve los bits hacia la izquierda agregando 0s por la derecha.
- \gg : Mueve los bits hacia la derecha agregando 0s o 1s, por la izquierda, si el valor original es positivo o negativo respectivamente.

Ve el siguiente ejemplo:

Código 3.6: Operadores de desplazamiento en bits

```

1  #include <iostream>
2  //using namespace std;
3
4  void imprimir_binarios(size_t tamano, void* valor);
5
6  int main(){
7      char a=5, b=13, c;
8      std::cout << " a = " ;
9      imprimir_binarios(sizeof a, &a);
10     std::cout << " b = " ;
11     imprimir_binarios(sizeof b, &b);
12     std::cout << " a << 2 = " << (a<<2) << " = ";
13     c = a<<2;
14     imprimir_binarios(sizeof c, &c );
15     std::cout << " b << 2 = " << (b<<2) << " = ";
16     c = b<<2;
17     imprimir_binarios(sizeof c, &c );
18     std::cout << " a >> 1 = " << (a>>1) << " = ";
19     c = a>>1;

```

```

20     imprimir_binarios(sizeof c, &c );
21     std::cout << " b >> 1 = " << (b>>1) << " = ";
22     c = b>>1;
23     imprimir_binarios(sizeof c, &c );
24
25     return 0;
26 }
27
28 void imprimir_binarios(size_t tamano, void* valor) {
29     unsigned char *allbytes = (unsigned char*) valor; //posicion en
30     //bytes(referencia a direccion de memoria del valor)
31     unsigned char bit; //unsigned concentrara todo a solo los bits sin
32     //importar el signo, y por tamano de 1 byte
33
34     for(int i= tamano-1;i>=0;i--){ //for para todos los bytes
35         for(int j=7;j>=0;j--){ //for para cada octeto
36             bit = (allbytes[i]>>j) & 1; //movemos el byte para comparar esa
37             //posicion con 1
38             std::cout << (bit == 0 ? 0 : 1);
39         }
40         std::cout << " ";
41     }
42     std::cout << std::endl;
43 }

```

Similarmente a los operadores anteriores **experimenta** con los valores para observar las distintas representaciones.

1.7. Operadores de asignación

Como su nombre lo indica, son aquellos que asignan un valor a otro. Hasta el momento hemos estado utilizando =, pero también podemos mezclar este junto la mayoría de los operadores vistos anteriormente para ahorrar pasos a la hora de escribir. Todos siguen el mismo formato donde “a *operador* = b” es lo mismo que “a = a *operador* b”.

Hablando del operador = que tanto hemos usado, este hace tal cual lo que menciona, donde hace que la variable de la izquierda tenga lo mismo que lo de la derecha. Es importante asegurarse de que es compatible el valor que queremos asignar, inclusive usando una conversión explícita en la forma de “(tipo_de_dato)”, ya que si asignamos un valor no válido, nuestro programa puede fallar.

Observa el siguiente ejemplo de operadores:

Código 3.7: Operadores de asignación

```

1 #include <iostream>
2 //using namespace std;
3
4 int main(){

```

```

5  int a=5, b=-13;
6  a += 1;
7  std::cout << " a = " << a <<std::endl;
8  a -= b;
9  std::cout << " a = " << a <<std::endl;
10 a *= b;
11 std::cout << " a = " << a <<std::endl;
12 a /= b;
13 std::cout << " a = " << a <<std::endl;
14 a %= b;
15 std::cout << " a = " << a <<std::endl;
16 a &= b;
17 std::cout << " a = " << a <<std::endl;
18 a ^= b;
19 std::cout << " a = " << a <<std::endl;
20 a |= b;
21 std::cout << " a = " << a <<std::endl;
22
23 return 0;
24 }

```

Experimenta con los valores de *a* y *b*. Se deja como ejercicio agregar los últimos dos operadores de asignación, de los cuales se debe notar que sucede al pedir que se muevan los bits en una cantidad negativa.

1.8. Operador ternario

Este operador aunque puede parecer algo confuso, es muy útil para asignar valores al crearlos, además de ser más rápido que usar su equivalente en *if* y *else* que se vera más adelante.

Funciona con la siguiente estructura: *condición* ? *a* : *b*. Lo cual se puede leer como: si *condición* es cierta, entonces regresa el valor de *a*, si no, el valor de *b*. Analiza el siguiente ejemplo:

Código 3.8: Operador ternario

```

1  #include <iostream>
2  //using namespace std;
3
4  int main(){
5      int a=5, b=-13;
6      std::cout << " El valor mayor entre a y b es: "
7      << ( a >= b ? a : b);
8
9      return 0;
10 }

```

Dado que este operador tiene la precedencia casi más baja, se debe usar paréntesis para asegurar el orden en que se realizan las operaciones. Como los ejercicios anteriores, *experimenta* con los valores de *a* y *b* para ver

como cambia el resultado.

2. Secuencias de control básicas

Como se ha ido viendo, programar es hacer instrucciones para la computadora, pero ¿como hacemos que entienda casos o repita acciones? Para esto sirven las secuencias de control. Estas nos permiten, como su nombre lo indica, darle un sentido y seriación a nuestro programa de forma menos lineal. A continuación se presentan las principales de C++ que también son comunes en otros lenguajes.

2.1. if, else

Traducido como: si, si no. *if* es una secuencia básica de control que toma verdadero o falso, siendo que si la condición entre paréntesis es cierta, se hará lo que delimita *if*, si no, se ignora o se hace lo que se coloque en un *else* después del caso de *if*. Observa el siguiente ejemplo:

Código 3.9: Secuencia de control if, else

```

1 #include <cstdlib> //contiene rand y srand para azar
2 #include <iostream>
3 #include <ctime> //con time() se obtiene el tiempo
4 //using namespace std;
5
6 int main(){
7     srand(time(nullptr));
8     int var = rand() % 6;
9     std::cout << "El numero aleatorio obtenido es "
10         << var << std::endl;
11
12     if(var > 3)
13         std::cout << var << " es mayor que 3." << std::endl;
14     else if (var == 3) //observa esta combinacion
15         std::cout << var << " es igual a 3." << std::endl;
16     else
17         std::cout << var << " es menor que 3." << std::endl;
18
19     return 0;
20 }
```

Esta secuencia se puede expandir y facilitar con un *switch* cuando se tendrían muchos *if* y *else*, este *switch* toma la condición a comparar y corre los casos bajo los que la condición sea igual a la respuesta, observa la sintaxis de esta secuencia en el ejemplo y compáralo al anterior:

Código 3.10: Secuencia de control switch

```

1 #include <cstdlib> //contiene rand y srand para azar
2 #include <iostream>
```

```
3  #include <ctime> //con time() se obtiene el tiempo
4  //using namespace std;
5
6  int main(){
7      srand(time(nullptr));
8      int var = rand() % 6 + 1;
9
10     std::cout << "El numero aleatorio obtenido es "
11               << var << std::endl;
12
13     switch(var){
14         case 6:
15             std::cout << "Que suerte tienes!";
16             break;
17         case 5:
18             std::cout << "Muy buen numero!";
19             break;
20         case 4:
21             std::cout << "Buen numero.";
22             break;
23         case 3:
24             std::cout << "Puede ser mejor.";
25             break;
26         case 2:
27             std::cout << "Algo es algo.";
28             break;
29         case 1:
30             std::cout << "Mas suerte para la proxima.";
31             break;
32         default:
33             std::cout << "Oops hay un numero no valido.";
34             break;
35     }
36
37     return 0;
38 }
```

Como podrás ver, el switch toma un valor entre paréntesis y verifica que hacer entre corchetes, siendo cada *case* un caso (tal cual la traducción), en donde realiza acciones hasta llegar a un *break*. Fíjate también como ahora nos limitamos a valores específicos, es por esto que se debe considerar bien que secuencia sirve mas en ciertos casos.

2.2. for

Traducido como: para. Se suele usar *for* para realizar operaciones desde un valor hasta otro, por ejemplo desde 0 hasta 5, o inclusive en forma descendiente, observa el siguiente ejemplo:

Código 3.11: Secuencia de control for

```
1 #include <cstdlib> //contiene rand y srand para azar
2 #include <iostream>
3 #include <ctime> //con time() se obtiene el tiempo
4 //using namespace std;
5
6 int main(){
7     srand(time(nullptr));
8     int var = rand() % 6 + 1;
9     std::cout << "El numero aleatorio obtenido es "
10         << var << std::endl;
11
12     for(int i=0; i<var; i++){
13         std::cout << i << " ";
14     }
15
16     for(int i=var; i >= 0; i--){
17         std::cout << i << " ";
18     }
19
20     return 0;
21 }
```

Como puedes ver en el ejemplo, la sintaxis de un *for* entre paréntesis es primero la declaración de un valor a usar, en medio la comparación o condición con la que se termina, y al final un aumento (o decremento) del valor que usamos, siendo separados estos elementos por punto y coma.

2.3. while

Traducido como: mientras. Hace algo mientras sea verdad el análisis o la condición dada entre paréntesis, osea que no valga 0 o *false*.

Código 3.12: Secuencia de control while

```
1 #include <cstdlib> //contiene rand y srand para azar
2 #include <iostream>
3 #include <ctime> //con time() se obtiene el tiempo
4 //using namespace std;
5
6 int main(){
7     srand(time(nullptr));
8     int var = rand() % 6;
9     std::cout << "El numero aleatorio obtenido es "
10         << var << std::endl;
11
12     while(var>1){
13         std::cout << "While repetira esto hasta ser "
14             << "falso lo que esta entre parentesis."
15             << std::endl;
16     }
```



```

16         std::cout << "Podemos salir cuando queramos de una"
17         << " secuencia de control, usando break."
18         << std::endl;
19         break;
20     }
21
22     return 0;
23 }
24

```

Observa como la palabra *break* se puede usar en otras secuencias de control, no solamente en un *switch*.

2.4. do while

Traducido como: haz mientras. Similar a while pero ejecuta las instrucciones al menos 1 vez.

Código 3.13: Secuencia de control do while

```

1  #include <cstdlib> //contiene rand y srand para azar
2  #include <iostream>
3  #include <ctime> //con time() se obtiene el tiempo
4  //using namespace std;
5
6  int main(){
7      srand(time(nullptr));
8      int var = rand() % 6;
9      std::cout << "El numero aleatorio obtenido es "
10      << var << std::endl;
11
12      int i = 0;
13      do //Los bucles/loops atorran el programa al dejarlo encerrado
14      { //haciendo las mismas operaciones
15          std::cout << "Las secuencias de control pueden causar "
16          << "bucles interminables si no se cuidan." << std::endl;
17          if(i++ > 3)
18              break;
19
20      }while(var > 1);
21
22      return 0;
23 }

```

Se invita al lector a probar varias veces los códigos para ver como se genera un valor nuevo gracias a la función *rand()*. También se deben modificar las secuencias de control para ganar un mejor entendimiento de las mismas. Como se menciona en los comentarios del código, los bucles pueden causar problemas inesperados, este caso es sencillo pero existen otros en donde ya no sabemos que pasa con la variable, por ejemplo cuando el usuario debe ingresar

el valor, y no tomamos en cuenta algo de lo que llega a escribir, es por ello que se debe tomar en cuenta un cierre para el programa siempre.

Capítulo 4

Pensando en código

Hasta el momento ya se han mostrado los componentes fundamentales del lenguaje con los que podemos hacer operaciones y acciones básicas. Ahora se mostrarán formas de expandir nuestro conocimiento con las ideas de programación en general.

1. Pseudocódigo

En este manual solo hemos estado estudiando el lenguaje de C++, pero ¿que pasa cuando queremos darle indicaciones o algoritmos a una persona que no maneja C++ o el mismo lenguaje de programación que nosotros?

Para esto se tiene el pseudocódigo como una herramienta que permite concentrarse enteramente en el procedimiento, dejando de lado aquellos detalles que sean más específicos de cada lenguaje.

El pseudocódigo considera una estructura similar al código pero usando un lenguaje más parecido a un instructivo, así el código y/o algoritmo es legible por cualquier persona sin importar que lenguaje de programación use.

Ejemplo:

```
funcionUno(tipo Algo,tipo2 otro)
    tipo extra <- Algo
    if extra = Algo
        then imprime(otro)
return Algo
```

Como se puede observar, el pseudocódigo contempla unos elementos que simplifican la sintaxis y permiten entender mejor las partes y operaciones del algoritmo:

- Usa flechas para mostrar asignación de un valor a otro, siendo la punta de la flecha el dato que recibe un valor.
- Puede omitir delimitadores del texto, por ejemplo “{ }” o “;”, entre otros, pero requiere conservar una estructura y espaciado para dar a entender el contexto y el orden de las operaciones.
- Suele usar espacios o tabulaciones para acomodar el texto y mostrar una jerarquía.
- También puede omitir el tipo de dato, especificándolo solo por el contexto.

Entre otras características que pueden ser específicas del ambiente donde se use.

2. Memoria

Parte de la programación es la información, por lo que nos interesa tanto usarla, como almacenarla, y a veces conocemos la cantidad a guardar, en otras ocasiones no, pero al menos es importante entender la naturaleza de la misma para poder administrarla.

Para esto existe:

- La memoria estática: La cual es común cuando conocemos la cantidad de información y su máximo a utilizar. Suele usarse en forma de “arreglos” (*arrays*), las cadenas de caracteres, y los conjuntos de números son los más populares. Pueden hacerse arreglos con corchetes `[]` después del tipo de variable, colocando un número para dar la cantidad máxima a usar, o dejando los corchetes pero definiendo en su momento todos los elementos que irán.
- La memoria dinámica: Esta se usa cuando desconocemos la cantidad de datos, sirve para pedir espacio de almacenamiento en el momento que lo requerimos, tiene mucho uso gracias a lo que su nombre indica. Un ejemplo que veremos son *Strings*, los cuales son un arreglo de caracteres dinámico, esto es, le podemos quitar y/o agregar sin preocuparnos de haber definido una cantidad máxima de letras.

Veamos un uso de la memoria estática con datos que ya conocemos:

Código 4.1: Memoria estática

```

1  #include <iostream>
2
3  int main(){
4      int arreglo[6] = {3,2,4,5,3,2};
5      int otro[] = {3,4,2,4,5,2};
6      int sum1=0,sum2=0;
7      for(int i= 0; i < 6; i++){
8          sum1 += arreglo[i];
9          sum2 += otro[i];
10     }
11     std::cout << "Suma del arreglo 1: "<<sum1 << std::endl;
12     std::cout << "Suma del arreglo 2: "<<sum2 ;
13
14     return 0;}
```

Observa como tenemos que usar corchetes “`[]`” y un numero entre ellos para acceder al valor del arreglo estático, mientras que al inicializarlo podemos optar entre colocar o no un número para especificarle el tamaño, solo que debemos colocar la misma cantidad de elementos **o menos** si le damos un número. **Nota** que si bien mencionamos *6* como el numero de elementos al inicializar el *int*, **empezamos con 0** y en realidad **terminamos con 5**, esto es porque el 0 es el espacio de memoria inicial, así que presta atención a como hagas las cuentas

en el futuro, ya que si intentas acceder a un espacio de memoria que no esta usando tu programa (o que bien no has declarado), tu programa fallara. Prueba esto cambiando los valores del *for* así como la cantidad de elementos del arreglo con tamaño especificado.

Una mezcla de memoria dinámica con estática es *string[] args* donde considera que requiere un número de espacios de memoria dinámica. El dato *string* esta definido en la librería “*string*”, observa el siguiente ejemplo:

Código 4.2: Memoria dinámica

```

1 #include <iostream>
2 #include <string>
3
4 int main(){
5     std::string a= "Hola";
6     std::string b[2]= {"", " esto"};
7     std::string c[] = {" ", "es", " "};
8     std::string d= "un ejemplo.";
9     std::cout<< a;
10    for(int i=0; i< 2; i++) std::cout << b[i];
11    for( auto str: c ) std::cout << str;
12    std::cout << d;
13
14    return 0;
15 }
```

También presta atención a como se usa la palabra reservada *auto* para darle una nueva estructura a la secuencia de control *for* y simplificar nuestro código.

3. Apuntadores

Dado que ya hablamos de los tipos de memoria, ahora vamos a ver que podemos realizar sobre la misma. Este tema puede ser algo complicado, por lo que se recomienda poner buena atención, ya que también trae gran potencial con sigo mismo.

Toda la información se almacena con el fin de poder usarla, aunque sea temporalmente como lo hemos estado haciendo en programas hasta el momento, y la forma en que se almacena en la memoria de las computadoras es por *direcciones*, como si de casas en la vida real se tratara. Estas direcciones son tal cual el espacio en específico donde se encuentra la información, y como tales, existen otros elementos que pueden apuntar a ellas o llevar sus datos, estos son los **apuntadores** observa el siguiente ejemplo donde al fin usamos los operadores de referencia/dirección (“&”) y desreferencia/indirección (“*”):

Código 4.3: Operadores de dirección de memoria.

```

1 #include <iostream>
2
```

```

3  int main(){
4      int a = 255;
5      std::cout << a << " esta en la direccion "
6          << &a << std::endl;
7      int* apuntador; //Se declara un apuntador
8      std::cout << "El apuntador tipo int es: " << apuntador
9          << std::endl;
10
11     apuntador = &a; //Se le asigna la direccion de a
12     std::cout << " Y ahora es (apunta a): " << apuntador
13         << ". Donde esta almacenado: " << *apuntador;
14
15     *apuntador = 3; //Se cambia el valor en esta direccion
16     std::cout << std::endl << "Ahora el valor en la direccion "
17         << &a << " vale " << a;
18
19     return 0;
20 }

```

Observa como se usa el mismo operador de indirección tanto para declarar un apuntador, como para traer la información que contiene. Este operador puede resultar ser el más complicado por estos dos usos, así como también pueden existir apuntadores de apuntadores.

Para dar mas claridad a esto, hay que entender que un apuntador es específicamente un espacio de memoria que guarda la dirección de otro espacio de memoria. Estos espacios de memoria siguen necesitando un tipo de dato, ya que como vimos en [5 Tipos de datos básicos](#), cada tipo de dato tiene un tamaño diferente en *bytes*, y este tamaño es lo que necesita conocer la computadora para guardar correctamente la información, sin chocar con la memoria de su alrededor.

El operador de referencia o dirección conserva su significado, donde siempre da la dirección en memoria del dato, ya sea un *int*, apuntador a un *int*, o apuntador de apuntador, etc.

Como sucede con los demás datos, necesitamos operar entre tipos de memoria compatibles, solo que en este caso no hay conversiones que nos salven de incompatibilidades, por lo que se debe ser específico en el tipo de dato y en que nivel de apuntador a apuntador se tiene. Analiza el siguiente ejemplo donde se usa un apuntador de un apuntador:

Código 4.4: Apuntador de apuntador.

```

1  #include <iostream>
2  //ejemplo_a asigna la direccion de b al apuntador a
3  void ejemplo_a(int **a, int *b){
4      *a = b;
5  }
6
7  int main(){

```

```

8   int var = 255;
9   int *pvar = &var;
10  int bar = 3;
11
12  std::cout << pvar << " = " << *pvar << std::endl;
13  std::cout << &bar << " = " << bar << std::endl
14      << std::endl;
15  //se manda la direccion del apuntador y el int
16  ejemplo_a(&pvar, &bar);
17  std::cout << pvar << " = " << *pvar << std::endl;
18  std::cout << &bar << " = " << bar;
19
20  return 0;
21 }

```

Por más veces que corras el código, podrás ver que las direcciones que tienen los *ints* al inicio son los mismos en el resto del programa, en cambio un apuntador puede cambiar la dirección a la que apunta, pero sigue teniendo su propio espacio dentro de la memoria, el cual obtenemos con el operador de dirección, siendo así capaces de alterar los contenidos de un apuntador.

Experimenta agregando otro operador de indirección *** en las variables *a* y *b* en la línea,

```

1   *a = b;

```

y analiza que estas afectando en esa oración.

Si realizaste lo anterior, podrás ver que sí afectamos al dato *var* dentro de la función, lo cual mencionamos que no podías hacer si lo pasabas como un parámetro normal en [6 Funciones](#). Observa algo que podemos hacer gracias a esto:

Código 4.5: Intercambio de valores.

```

1  #include <iostream>
2
3  void intercambia(int *a, int *b){
4      int temp = *b;
5      *b = *a;
6      *a = temp;
7  }
8
9  int main(){
10     int menor = 255;
11     int mayor = 3;
12
13     std::cout << menor << " < " << mayor << std::endl;
14     intercambia(&menor, &mayor);

```

```

15     std::cout << menor << " < " << mayor << std::endl;
16
17     return 0;
18 }

```

Teniendo esta capacidad de cambiar y modificar espacios de memoria, podemos realizar más acciones que necesiten orden en la memoria, así como otras alteraciones específicas de memoria como el manejo de archivos, e inclusive dispositivos, entre otras cosas.

Capítulo 5

Sección Yeyecoa

Lo aprendido hasta ahora han sido elementos básicos y necesarios para empezar a programar y encaminarse en C++. Sin embargo hay algunos temas que pueden ser de utilidad, como a continuación se presentan algunos. Si ya te sientes listo para hacer retos avanza a [4 Ahora Yeyecoa](#).

1. Secuencias de escape

Algunos caracteres que tienen una diagonal (“\”) antes tienen un significado para el compilador, a estos se les llaman “secuencias de escape”, y representan cosas distintas, como se muestra en la siguiente tabla:

Secuencia de escape	Descripción
\a	emite una campanita o alarma audible.
\b	Inserta un backspace en la posición donde está el texto.
\f	Inserta un “formfeed”, sirve para limpiar la pantalla en consola.
\n	Inserta una nueva línea o “enter”.
\r	Inserta un “retorno de carro”, solía ser para regresar el marcador del texto al inicio, pero en tiempos modernos a cambiado un poco su funcionalidad, siendo inútil o similar a un \n en algunos compiladores o entornos.
\t	Inserta una tabulación o tab (horizontal) en el texto en este punto.
\v	Inserta una tabulación o tab (vertical) en el texto en este punto.
\'	Inserta una comilla simple ' en donde se escribe la secuencia.
\"	Inserta una comilla doble " en donde se escribe la secuencia.
\\	Inserta un \ en el lugar donde se escribe la secuencia.

Continúa en la siguiente página...

Tabla 5.1 – Continuación

Secuencia de escape	Descripción
<code>\?</code>	Inserta un signo de pregunta en el texto en este punto.
<code>\nnn</code>	Donde n son dígitos, escribe un valor en representación octal.
<code>\xn</code>	Donde n es cualquier entero, escribe ese valor en hexadecimal.
<code>\unnnnn</code>	Donde n son dígitos, escribe ese valor en su representación de carácter universal.
<code>\Unnnnnnnnn</code>	Donde n son dígitos, escribe ese valor en su representación de carácter universal.

Tabla 5.1: Secuencias de escape ¹

Es de notar que estos toman significado cuando se trata de procesar texto, lo que se ha mostrado en ejemplos con “cout <<” o “std::cout <<”.

2. Buenas practicas de comentarios

Los comentarios en los códigos juegan un papel mas importante que solo dejar notas, ya que permiten hablar específicamente de partes del código sin tener que salirse de él a explicarlo, o tener que analizarlo detalladamente para entender que quiere hacer, aunque esto también es valido y puede agregar más claridad, sin embargo, esto ultimo se suele realizar en trabajos o proyectos grandes.

Entonces surgen la dudas, ¿como debería comentar mi código? y ¿cuando debería hacerlo? primero hablemos de comentarlo: No basta con solo comentar el código, también es importante entender los comentarios, estos deben ser claros y rápidos, por lo general indicando que hace la función y que regresa, y por completitud no afecta agregar explicaciones de ciertas lineas de código que puedan ser confusas en el futuro, por lo general estas son aquellas que realizan más de una acción en la misma linea, siendo la excepción el escribir con `cout <<`. Observa el siguiente ejemplo e intenta entender que hace la “función_a”, primero sin comentarios, luego compara que entiendes mejor entre las 2 versiones con comentarios:

Código 5.1: Sin comentarios

```

1 #include <iostream>
2
3 char funcion_a(char *a){
4     *a ^= 32;
5 }
6
7 int main(){
```

¹Se es similar a la tabla de `cppreference`[2].


```
8   char var = 'b';
9   std::cout<< var << std::endl;
10  funcion_a(&var);
11  std::cout<< var ;
12
13  return 0;
14 }
```

Código 5.2: Con comentarios 1

```
1  #include <iostream>
2
3  char funcion_a(char *a){ //para letras
4      *a ^= 32;
5  }
6
7  int main(){
8      char var = 'b';
9      std::cout<< var << std::endl;
10     funcion_a(&var); //mando direccion
11     std::cout<< var ;
12
13     return 0;
14 }
```

Código 5.3: Con comentarios 2

```
1  #include <iostream>
2
3  char funcion_a(char *a){ //Cambia minusculas y mayusculas
4      *a ^= 32;
5  }
6
7  int main(){
8      char var = 'b';
9      std::cout<< var << std::endl;
10     funcion_a(&var);
11     std::cout<< var ;
12
13     return 0;
14 }
```

Como puedes darte cuenta, el código 5.3 da una idea clara y rápida de lo que hace la función, es posible ser aun mas específicos con los comentarios, y en algunos lugares se usan ciertos formatos para lograr esta claridad.

Es una buena practica comentar todas las funciones y partes que no sean específicas para agregar claridad, evitando comentarios largos, con tal de entender el propósito general del código sin tener que indagarlo mucho.

Aunque claro, existe una excepción a cuando comentar, y esto es cuando la versión del código es destinada a ser enviada al cliente o publicada. Estas versiones deben ser lo mas ligeras posibles, por lo que los comentarios pueden terminar siendo un obstáculo en proyectos grandes. De todas formas no debe hacer falta una versión del código destinada al desarrollador que contenga los comentarios. Podemos resumir estos puntos de la siguiente forma:

- Siempre comenta tu código (excepto en la versión destinada a publicar).
- Realiza comentarios claros, específicos y sencillos.
- Sigue un formato para comentar, ya sea propio o decidido por el equipo.
- Se consistente en este formato (también aplica al código).

Por ultimo, se le muestra un uso útil de los comentarios en el código para pruebas:

Código 5.4: Tip

```
1 #include <iostream>
2
3 char funcion_a(char *a){ //Cambia entre minusculas y mayusculas
4     *a ^= 32;
5 }
6
7 char funcion_b(char a){ //Cambia entre minusculas y mayusculas
8     return a ^= 32;
9 }
10
11 int main(){
12     char var = 'b';
13     std::cout<< var << std::endl;
14
15     /* "/*" activa la ruta a, mientras que un "*/"
16     en su lugar activa la ruta b*/
17
18     /* //ruta a
19     funcion_a(&var);
20     std::cout<< var ;
21     /* //ruta b
22     std::cout<< funcion_b(var) ;
23     /**/
24
25     return 0;
26 }
```

Este uso es valido en otros lenguajes que acepten comentarios de la misma forma, solo recuerda no usarlo en código formal.

3. Recursividad

Cuando hablamos de funciones en [6 Funciones](#), también mencionamos que estas nos permiten reusarlas varias veces, esto es sin limite y también puede agregarse sin importar en donde sea usada. Esto ultimo nos da la gran ventaja de poder hacer funciones que se usen a si mismas, siendo conocidas como funciones “recursivas”, y brindando gran simplicidad y velocidad al código.

Estas funciones pueden causar problemas si no se cuidan, ya que pueden ocasionar bucles o errores de memoria, por lo que se deben definir claramente los siguientes aspectos:

- **Condición de paro:** esta es una revisión que nos diga cuando detener la recursividad de la función.
- **Casos:** si existen diferentes rutas o caminos a tomar, se deben especificar que sucede en aquellos casos.
- **Procedimiento:** son las acciones a tomar en cada caso, deben cuidar estar dentro de los limites del programa y su memoria asignada.

Observa el siguiente ejemplo:

Código 5.5: Cuenta regresiva

```

1  #include <iostream>
2
3  int cuenta_regresiva(int a){
4      if(a == 0) return 1;
5      std::cout << a << " ";
6      cuenta_regresiva( a-1 );
7  }
8
9  int main(){
10     int var= 10;
11     cuenta_regresiva(var);
12     std::cout << std::endl << "Despegue";
13
14     return 0;
15 }
```

Como puedes darte cuenta, realiza casi lo mismo que una secuencia de control puede lograr. Sin embargo, es importante notar que como función se realizan todas completas hasta llegar a un *return* o terminar, observa el siguiente ejemplo y diferencia entre el for y la función recursiva.

Código 5.6: Centro de una cuenta.

```

1  #include <iostream>
2
3  int centro(int a){
4      std::cout << a << " ";
```

```

5   if(a == 0) return 1;
6
7   centro( a-1 );
8   std::cout << a << " ";
9 }
10
11 int main(){
12     int var= 9;
13
14     centro(var);
15
16     std::cout << std::endl;
17     for(int i = 0; i <= var-1 ; i++){
18         std::cout << " ";
19     }
20     std::cout << "\n";
21
22     return 0;
23 }

```

Como puedes observar, esta capacidad de las funciones nos permiten realizar recursión múltiple, lo cual terminaría siendo muy complicado en el caso de usar secuencias de control. Esta técnica es muy importante al hacer operaciones con memoria o análisis que requieran operaciones múltiples. El problema es que las funciones requieren memoria, y si se hacen demasiadas por usar recursión, puede fallar el programa.

Similarmente, hay ocasiones en las que la iteración por secuencias de control es más conveniente, como por ejemplo el código 5.5 es más entendible, corto y rápido si se realiza con un simple *for*.

4. Ahora Yeyeco

Para practicar lo aprendido, realiza las siguientes actividades, si te sientes atorado en alguna revisa la respuesta, aunque no es necesario que sean idénticos los códigos para tener los mismos resultados, *inclusive puede hacer código mucho mejor!* Los ejercicios **yeyeco** son algo especiales ya que requieren pensar más en la solución, pero te permitirán perfeccionar tu conocimiento.

Práctica 1: Busca en línea los valores en ASCII de los caracteres que necesitas, para que así asignándolos a varios datos tipo char puedas imprimir “Hola Mundo!”

Respuesta 1:

```
5 char h=72, o=111, l=108, a=97, m=77,  
6     u=117, n=110, d=100, oo=111,  
7     esp=32, excl=33;  
8  
9     std::cout<< h << " " << o << l << a  
10         << esp << m << u << n  
11         << d << oo << excl;  
12  
13     return 0;  
14 }
```

Yeyecoa 2: Usando secuencias de control, imprime una pirámide hecha con
* de al menos 5 niveles.

Respuesta 2:

Práctica 3: Usando secuencias de control, calcula el factorial de un número, considera el factorial de 0 y 1 como que vale 1. *El factorial de un número es la multiplicación del mismo por el número anterior a este, así sucesivamente hasta llegar a 1.*

Respuesta 3:

```
8  
9     return 0;  
10 }
```

Práctica 4: Usando secuencias de control. Escribe en pantalla que letra denomina la calificación de un alumno, siendo que más de 90 puntos es “A”, entre 79 y 91 es “B”, entre 69 y 80 es “C”, entre 59 y 70 es “D”, menos de 60 es F. Para el valor de la calificación investiga “*std::cin <<*” y asígnale un valor a mano. Si ves muy difícil esto último inicializa manualmente la variable, y observa la solución.

Respuesta 4:

Yeyecoa 5: Calcula el Máximo Común Divisor entre 2 números. *Pista: usa el operador % y compáralo a 0 para revisar si un número es divisible, aprovecha también la secuencia de control for y el operador lógico &&. El Máximo Común Divisor (MCD) es el número más grande que puede dividir a 2 números sin dejar residuo.*

Respuesta 5:

```
12     std::cout << mcd;  
13  
14     return 0;  
15 }
```

***Yeyecoa 6:** De los primeros 20 números de la sucesión de Fibonacci, obtén cuantos de ellos son pares y cuantos impares. *Pista: usa el operador % para ver si son divisibles, y usa recursividad para obtener los números.*

Un número de la sucesión Fibonacci son aquellos obtenidos por la suma de los dos anteriores a él, siendo los dos primeros 0 y 1.

Respuesta 6:

No te detengas con estos ejercicios, busca ejemplos o casos de la vida real que se te podrían facilitar si emplearas lo aprendido de programación para realizar las operaciones y/o cálculos que veas necesarios. Si te llamaron la atención estos ejercicios revisa la sección [1 Por diversión](#) para obtener sugerencias de donde poder practicar más.

Capítulo 6

Expandiendo el conocimiento

El manual ha llegado a su fin, pero el camino de la programación apenas comienza. Puedes aprovechar lo aprendido de distintas formas que te llamen la atención, y en caso de querer conocer que más estudiar se presentan unas recomendaciones para encaminarse en la profundización sobre este lenguaje.

1. Por diversión

La programación es tanto una habilidad como un arte así como lo son el dibujar, el cantar, etc. Por lo cuál requiere de práctica y dedicación para desarrollarla. Para esto se pueden buscar mas recursos y fuentes de los cuales aprender y ponerse a prueba, por ejemplo:

- Aprender y/o practicar el inglés, ya que muchos recursos, y problemas se encuentran así, sin mencionar que las cosas básicas también están escritas en inglés.
- Resolver problemas que requieran programación, para esto existen sitios y aplicaciones como: Dcoder (App para android), projectEuler (<https://projecteuler.net/archives>), codechef (<https://www.codechef.com/>), entre otros.

2. Por profesión

Para perseguir la programación como carrera o profesión son convenientes:

- Cursos en distintos lenguajes, en la práctica siempre es útil conocer mas de un solo lenguaje de programación.
- Convertir ideas en creaciones, muchos problemas y aprendizajes se obtienen de intentar manifestar las ideas en realidades, además de que así se han generado muchas carreras, para esto es recomendable aprender diseño de software.
- Libros y guías , la editorial O'Reilly es popular por tener libros completos sobre la mayoría de lenguajes, incluyendo C++. Uno de los libros tanto de aprendizaje como de referencia más recomendados es del autor de C++ [4].

3. Sugerencias

Existen muchos medios y fuentes de los cuales aprender el lenguaje C++, solo que la mayoría están en inglés, lengua que también se sugiere aprender para poder aprovechar muchos de los recursos disponibles, algunos que se recomiendan para tener mejor entendimiento y habilidades de C++ se mencionan a continuación:

- **Programming: Principles and Practice Using C++:** es un libro muy completo (en inglés), es bueno para dar una mirada formal, correcta y amplia a los usos de C++, desde código hasta su uso en aplicaciones y juegos.
- **Tutoriales de C++:** Existen varios tutoriales sobre el lenguaje en línea, pero varios son incompletos o incorrectos, personalmente se recomienda la pagina <https://www.learncpp.com/>, por lo sencilla y clara forma en que se enseña, aunque tambien existe una gran coleccion de recomendaciones

en el siguiente foro (en ingles): <https://stackoverflow.com/questions/388242/the-definitive-c-book-guide-and-list>.

Motivación final

Estimado lector,

Felicitaciones por haber concluido este manual introductorio a la programación con C++. Programar en cualquier lenguaje es un paso valiente y grande para introducirse a la programación en general, de ahí en adelante dependerá mucho de los gustos particulares para desarrollarse, ya sean estos gustos por lo que se desea realizar, o por el lenguaje y sus herramientas.

Al final, la pasión y dedicación son los que traerán los resultados de aplicar los conocimientos y habilidades hacia las actividades, trabajos e ideas que se elaboren. Es por esto que se invita al lector a buscar metas que realizar con los conocimientos aprendidos, así podrá ir obteniendo mayor experiencia.

También se destaca que el programar y su aprendizaje es un proceso laborioso y que puede contener dificultades, pero todo esto es parte del proceso de aprendizaje, a lo cual se recomienda no rendirse al encontrarse con problemas u obstáculos, y animarse a preguntar o buscar en línea por soluciones que tal vez no le parecen obvias al inicio, pero posiblemente alguien más ya haya tenido la duda o el error al que usted se enfrente.

Sin más que agregar se le desea buena suerte.

-Grupo de Computo Cuántico y Científico de la Facultad de Estudios Superiores Acatlan.

Este documento fue creado gracias al PAPIME PE104919.

Bibliografía

- [1] Alex. learncpp. <https://www.learncpp.com/>. Recuperado el 15 de Mayo del 2019.
- [2] cppreference. cppreference. <https://en.cppreference.com/w/>. Recuperado el 16 de Mayo del 2019.
- [3] Harvey Deitel Paul Deitel. *C++ How to Program*. Deitel, 10 edition, 2017.
- [4] Bjarne Stroustrup. *Programming: Principles and Practice Using C++*. Addison-Wesley Professional, 2nd edition, 2014.