
C++
www.postparaprogramadores.com

Contenido

C ++ Fecha y hora

Entrada / salida básica de C ++

Estructuras de datos C ++

C ++ orientado a objetos

Clases y objetos de C ++

Herencia C ++

Sobrecarga de C ++

C ++ polimorfismo

Abstracción C ++

Encapsulación C ++

Interfaces C ++

C ++ avanzado

Archivos y secuencias de C ++

Manejo de excepciones de C ++

Memoria dinámica de C ++

Espacios de nombres C ++

Plantillas C ++

Preprocesador C ++

Manejo de señal C ++

C ++ Multithreading

Programación web C ++

Descripción general de C ++

C ++ es un lenguaje de programación de forma libre, de tipo general, compilado, de propósito general, sensible a mayúsculas y minúsculas, que admite programación, orientada a objetos y genérica.

C ++ se considera un lenguaje de **nivel medio** , ya que comprende una combinación de características de lenguaje de alto y bajo nivel.

C ++ fue desarrollado por Bjarne Stroustrup a partir de 1979 en Bell Labs en Murray Hill, Nueva Jersey, como una mejora del lenguaje C y originalmente llamado C con clases, pero más tarde pasó a llamarse C ++ en 1983.

C ++ es un superconjunto de C, y que prácticamente cualquier programa legal de C es un programa legal de C ++.

Nota : Se dice que un lenguaje de programación usa la escritura estática cuando la verificación de tipo se realiza durante el tiempo de compilación en lugar del tiempo de ejecución.

Programación orientada a objetos

C ++ es totalmente compatible con la programación orientada a objetos, incluidos los cuatro pilares del desarrollo orientado a objetos:

- Encapsulación
- Ocultar datos
- Herencia
- Polimorfismo

Descarga más libros de programación GRATIS click aquí

Síguenos en Instagram para que estés al tanto de los nuevos libros de programación. Click aqui

Bibliotecas estándar

El estándar C ++ consta de tres partes importantes:

- El lenguaje principal proporciona todos los componentes básicos, incluidas las variables, los tipos de datos y los literales, etc.
- La biblioteca estándar de C ++ ofrece un amplio conjunto de funciones para manipular archivos, cadenas, etc.
- La Biblioteca de plantillas estándar (STL) que ofrece un amplio conjunto de métodos para manipular estructuras de datos, etc.

El estándar ANSI

El estándar ANSI es un intento de garantizar que C ++ sea portátil; ese código que escriba para el compilador de Microsoft se compilará sin errores, utilizando un compilador en una Mac, UNIX, un cuadro de Windows o un Alpha.

El estándar ANSI ha sido estable durante un tiempo, y todos los principales fabricantes de compiladores C ++ son compatibles con el estándar ANSI.

Aprendiendo C ++

Lo más importante al aprender C ++ es centrarse en los conceptos.

El propósito de aprender un lenguaje de programación es convertirse en un mejor programador; es decir, ser más eficaz en el diseño e implementación de nuevos sistemas y en el mantenimiento de los antiguos.

C ++ admite una variedad de estilos de programación. Puede escribir en el estilo de Fortran, C, Smalltalk, etc., en cualquier idioma. Cada estilo puede lograr sus objetivos de manera efectiva mientras mantiene el tiempo de ejecución y la eficiencia del espacio.

Uso de C ++

C ++ es utilizado por cientos de miles de programadores en prácticamente todos los dominios de aplicación.

C ++ se está utilizando mucho para escribir controladores de dispositivos y otro software que depende de la manipulación directa del hardware bajo restricciones en tiempo real.

C ++ se usa ampliamente para la enseñanza y la investigación porque es lo suficientemente limpio para la enseñanza exitosa de conceptos básicos.

Cualquiera que haya usado Apple Macintosh o una PC con Windows indirectamente ha usado C ++ porque las interfaces de usuario principales de estos sistemas están escritas en C ++.

Configuración del entorno C ++

Configuración del entorno local

Si todavía está dispuesto a configurar su entorno para C ++, debe tener los siguientes dos softwares en su computadora.

Editor de texto

Esto se usará para escribir su programa. Los ejemplos de algunos editores incluyen el Bloc de notas de Windows, el comando Editar del sistema operativo, Breve, Epsilon, EMACS y vim o vi.

El nombre y la versión del editor de texto pueden variar en diferentes sistemas operativos. Por ejemplo, el Bloc de notas se usará en Windows y vim o vi se puede usar en Windows, así como en Linux o UNIX.

Los archivos que crea con su editor se denominan archivos de origen y para C ++ generalmente se nombran con la extensión .cpp, .cp o .c.

Debe haber un editor de texto para comenzar su programación en C ++.

Compilador C ++

Este es un compilador real de C ++, que se utilizará para compilar su código fuente en el programa ejecutable final.

A la mayoría de los compiladores de C ++ no les importa qué extensión le dé a su código fuente, pero si no especifica lo contrario, muchos usarán .cpp por defecto.

El compilador más utilizado y gratuito disponible es el compilador GNU C / C ++, de lo contrario puede tener compiladores de HP o Solaris si tiene los respectivos sistemas operativos.

Instalación del compilador GNU C / C ++

Instalación de UNIX / Linux

Si está utilizando **Linux o UNIX** , compruebe si GCC está instalado en su sistema ingresando el siguiente comando desde la línea de comandos:

```
$ g++ -v
```

Si ha instalado GCC, debería imprimir un mensaje como el siguiente:

```
Using built-in specs.
Target: i386-redhat-linux
Configured with: ../configure --prefix=/usr .......
Thread model: posix
gcc version 4.1.2 20080704 (Red Hat 4.1.2-46)
```

Si GCC no está instalado, tendrá que instalarlo usted mismo utilizando las instrucciones detalladas disponibles en https://gcc.gnu.org/install/

Instalación de Mac OS X

Si utiliza Mac OS X, la forma más fácil de obtener GCC es descargar el entorno de desarrollo Xcode del sitio web de Apple y seguir las sencillas instrucciones de instalación.

Xcode está actualmente disponible en developer.apple.com/technologies/tools/ .

Instalación de ventanas

Para instalar GCC en Windows necesita instalar MinGW. Para instalar MinGW, vaya a la página de inicio de MinGW, www.mingw.org , y siga el enlace a la página de descarga de MinGW. Descargue la última versión del programa de instalación de MinGW, que debe llamarse MinGW- <versión> .exe.

Al instalar MinGW, como mínimo, debe instalar gcc-core, gcc-g ++, binutils y el tiempo de ejecución de MinGW, pero es posible que desee instalar más.

Agregue el subdirectorio bin de su instalación de MinGW a su **variable de** entorno **PATH** para que pueda especificar estas herramientas en la línea de comandos por sus nombres simples.

Cuando se complete la instalación, podrá ejecutar gcc, g ++, ar, ranlib, dlltool y varias otras herramientas GNU desde la línea de comandos de Windows.

Sintaxis básica de C ++

Cuando consideramos un programa C ++, se puede definir como una colección de objetos que se comunican invocando los métodos de cada uno. Veamos ahora brevemente qué significan una clase, un objeto, métodos y variables instantáneas.

- **Objeto** : los objetos tienen estados y comportamientos. Ejemplo: un perro tiene estados: color, nombre, raza, así como comportamientos: menear, ladrar, comer. Un objeto es una instancia de una clase.
- **Clase** : una clase se puede definir como una plantilla / modelo que describe los comportamientos / estados que admite ese objeto de su tipo.
- **Métodos** : un método es básicamente un comportamiento. Una clase puede contener muchos métodos. Es en los métodos donde se escriben las lógicas, se manipulan los datos y se ejecutan todas las acciones.
- **Variables de instancia** : cada objeto tiene su conjunto único de variables de instancia. El estado de un objeto es creado por los valores asignados a estas variables de instancia.

Estructura del programa C ++

Veamos un código simple que imprimiría las palabras *Hello World* .

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

// main() is where program execution begins.
int main() {
```

```
    cout << "Hello World"; // prints Hello World
    return 0;
}
```

Veamos las diversas partes del programa anterior:

- El lenguaje C ++ define varios encabezados, que contienen información que es necesaria o útil para su programa. Para este programa, se necesita el encabezado **<iostream>** .
- La línea que **usa el espacio de nombres estándar;** le dice al compilador que use el espacio de nombres estándar. Los espacios de nombres son una adición relativamente reciente a C ++.
- La siguiente línea ' **// main () es donde comienza la ejecución del programa.** 'es un comentario de una sola línea disponible en C ++. Los comentarios de una sola línea comienzan con // y terminan al final de la línea.
- La línea **int main ()** es la función principal donde comienza la ejecución del programa.
- La siguiente línea **cout << "Hola Mundo";** hace que el mensaje "Hola mundo" se muestre en la pantalla.
- La siguiente línea **devuelve 0;** termina la función main () y hace que devuelva el valor 0 al proceso de llamada.

Compilar y ejecutar el programa C ++

Veamos cómo guardar el archivo, compilar y ejecutar el programa. Siga los pasos que se detallan a continuación:

- Abra un editor de texto y agregue el código como se indicó anteriormente.
- Guarde el archivo como: hello.cpp
- Abra un símbolo del sistema y vaya al directorio donde guardó el archivo.
- Escriba 'g ++ hello.cpp' y presione Intro para compilar su código. Si no hay errores en su código, el símbolo del sistema lo llevará a la siguiente línea y generará un archivo ejecutable.
- Ahora, escriba 'a.out' para ejecutar su programa.
- Podrá ver 'Hello World' impreso en la ventana.

```
$ g++ hello.cpp
$ ./a.out
Hello World
```

Asegúrese de que g ++ esté en su ruta y de que lo esté ejecutando en el directorio que contiene el archivo hello.cpp.

Puede compilar programas C / C ++ usando makefile. Para más detalles, puede consultar nuestro 'Tutorial de Makefile' .

Punto y coma y bloques en C ++

En C ++, el punto y coma es un terminador de instrucciones. Es decir, cada declaración individual debe terminar con un punto y coma. Indica el final de una entidad lógica.

Por ejemplo, a continuación hay tres declaraciones diferentes:

```
x = y;
y = y + 1;
add(x, y);
```

Un bloque es un conjunto de declaraciones conectadas lógicamente que están rodeadas por llaves de apertura y cierre. Por ejemplo

```
{
   cout << "Hello World"; // prints Hello World
   return 0;
}
```

C ++ no reconoce el final de la línea como un terminador. Por esta razón, no importa dónde coloque una declaración en una línea. Por ejemplo

```
x = y;
y = y + 1;
add(x, y);
```

es lo mismo que

```
x = y; y = y + 1; add(x, y);
```

Identificadores de C ++

Un identificador de C ++ es un nombre utilizado para identificar una variable, función, clase, módulo o cualquier otro elemento definido por el usuario. Un identificador comienza con una letra A a Z o una a z o un guión bajo (_) seguido de cero o más letras, guiones bajos y dígitos (0 a 9).

C ++ no permite caracteres de puntuación como @, $ y% dentro de los identificadores. C ++ es un lenguaje de programación sensible a mayúsculas y minúsculas. Por lo tanto, **Manpower** y **manpower** son dos identificadores diferentes en C ++.

Aquí hay algunos ejemplos de identificadores aceptables:

```
mohd       zara    abc   move_name  a_123
myname50   _temp   j     a23b9      retVal
```

Palabras clave de C ++

La siguiente lista muestra las palabras reservadas en C ++. Estas palabras reservadas no se pueden usar como constantes o variables ni ningún otro nombre identificador.

asm	else	new	this

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

Trigraphs

Algunos caracteres tienen una representación alternativa, llamada secuencia trigráfica. Un trigraph es una secuencia de tres caracteres que representa un solo carácter y la secuencia siempre comienza con dos signos de interrogación.

Los trígrafos se expanden en cualquier lugar donde aparezcan, incluidos los literales de cadena y los literales de caracteres, en comentarios y en directivas de preprocesador.

Las siguientes son las secuencias trigráficas más utilizadas:

Trigraph	Reemplazo
?? =	# #
?? /	\
??	^
?? (	[
??)	]
??!	El |
?? <	{
??>	}
?? -	~

Todos los compiladores no admiten trigrafos y no se recomienda su uso debido a su naturaleza confusa.

Espacio en blanco en C ++

Una línea que contiene solo espacios en blanco, posiblemente con un comentario, se conoce como una línea en blanco, y el compilador de C ++ la ignora por completo.

Espacio en blanco es el término utilizado en C ++ para describir espacios en blanco, pestañas, caracteres de nueva línea y comentarios. El espacio en

blanco separa una parte de una declaración de otra y permite al compilador identificar dónde termina un elemento de una declaración, como int, y dónde comienza el siguiente elemento.

Declaración 1

```
int age;
```

En la declaración anterior debe haber al menos un carácter de espacio en blanco (generalmente un espacio) entre int y age para que el compilador pueda distinguirlos.

Declaración 2

```
fruit = apples + oranges;   // Get the total fruit
```

En la declaración anterior 2, no se necesitan caracteres de espacio en blanco entre la fruta y =, o entre = y las manzanas, aunque puede incluir algunos si lo desea con fines de legibilidad.

Comentarios en C ++

Los comentarios del programa son declaraciones explicativas que puede incluir en el código C ++. Estos comentarios ayudan a cualquiera que lea el código fuente. Todos los lenguajes de programación permiten algún tipo de comentarios.

C ++ admite comentarios de una o varias líneas. El compilador de C ++ ignora todos los caracteres disponibles dentro de cualquier comentario.

Los comentarios de C ++ comienzan con / * y terminan con * /. Por ejemplo

```
/* This is a comment */

/* C++ comments can also
   * span multiple lines
*/
```

Un comentario también puede comenzar con //, extendiéndose hasta el final de la línea. Por ejemplo

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

main() {
   cout << "Hello World"; // prints Hello World

   return 0;
}
```

Cuando se compila el código anterior, ignorará **// imprime Hello World** y el ejecutable final producirá el siguiente resultado:

```
Hello World
```

Dentro de un / * y * / comentario, // los caracteres no tienen un significado especial. Dentro de un // comentario, / * y * / no tienen un significado especial. Por lo tanto, puede "anidar" un tipo de comentario dentro del otro tipo. Por ejemplo

```
/* Comment out printing of Hello World:

cout << "Hello World"; // prints Hello World

*/
```

Tipos de datos C ++

Mientras escribe un programa en cualquier idioma, necesita usar varias variables para almacenar diversa información. Las variables no son más que ubicaciones de memoria reservadas para almacenar valores. Esto significa que cuando crea una variable, reserva algo de espacio en la memoria.

Es posible que desee almacenar información de varios tipos de datos como caracteres, caracteres anchos, enteros, coma flotante, doble coma flotante, booleano, etc. Según el tipo de datos de una variable, el sistema operativo asigna memoria y decide qué se puede almacenar en el memoria reservada

Tipos primitivos incorporados

C ++ ofrece al programador una gran variedad de tipos de datos integrados y definidos por el usuario. La siguiente tabla enumera siete tipos de datos básicos de C ++:

Tipo	Palabra clave
Booleano	bool
Personaje	carbonizarse
Entero	En t
Punto flotante	flotador
Doble punto flotante	doble
Sin valor	vacío
Personaje ancho	wchar_t

Varios de los tipos básicos se pueden modificar utilizando uno o más de estos modificadores de tipo:

- firmado
- no firmado
- corto
- largo

La siguiente tabla muestra el tipo de variable, cuánta memoria se necesita para almacenar el valor en la memoria y cuál es el valor máximo y mínimo que se puede almacenar en dicho tipo de variables.

Tipo	Ancho de bits típico	Rango tipico
carbonizarse	1byte	-127 a 127 o 0 a 255
char sin firmar	1byte	0 a 255
char firmado	1byte	-127 a 127
En t	4 bytes	-2147483648 a 2147483647
unsigned int	4 bytes	0 a 4294967295
int firmado	4 bytes	-2147483648 a 2147483647
int corto	2bytes	-32768 a 32767
unsigned short int	2bytes	0 a 65,535
int corto firmado	2bytes	-32768 a 32767
int largo	8bytes	-2,147,483,648 a 2,147,483,647
firmado largo int	8bytes	igual que int largo
unsigned long int	8bytes	0 a 4,294,967,295

largo largo int	8bytes	- (2 ^ 63) a (2 ^ 63) -1
unsigned long long int	8bytes	0 a 18,446,744,073,709,551,615
flotador	4 bytes	
doble	8bytes	
doble largo	12bytes	
wchar_t	2 o 4 bytes	1 personaje ancho

El tamaño de las variables puede ser diferente de las que se muestran en la tabla anterior, dependiendo del compilador y la computadora que esté utilizando.

El siguiente es el ejemplo, que producirá el tamaño correcto de varios tipos de datos en su computadora.

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

int main() {
   cout << "Size of char : " << sizeof(char) << endl;
   cout << "Size of int : " << sizeof(int) << endl;
   cout << "Size of short int : " << sizeof(short int) <<
endl;
   cout << "Size of long int : " << sizeof(long int) << endl;
   cout << "Size of float : " << sizeof(float) << endl;
   cout << "Size of double : " << sizeof(double) << endl;
   cout << "Size of wchar_t : " << sizeof(wchar_t) << endl;

   return 0;
}
```

Este ejemplo usa **endl** , que inserta un carácter de nueva línea después de cada línea y el operador << se está utilizando para pasar múltiples valores a la pantalla. También estamos utilizando el operador **sizeof ()** para obtener el tamaño de varios tipos de datos.

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado que puede variar de una máquina a otra:

```
Size of char : 1
Size of int : 4
Size of short int : 2
Size of long int : 4
```

```
Size of float : 4
Size of double : 8
Size of wchar_t : 4
```

Declaraciones typedef

Puede crear un nuevo nombre para un tipo existente usando **typedef** . La siguiente es la sintaxis simple para definir un nuevo tipo usando typedef -

```
typedef type newname;
```

Por ejemplo, lo siguiente le dice al compilador que pies es otro nombre para int -

```
typedef int feet;
```

Ahora, la siguiente declaración es perfectamente legal y crea una variable entera llamada distancia:

```
feet distance;
```

Tipos enumerados

Un tipo enumerado declara un nombre de tipo opcional y un conjunto de cero o más identificadores que pueden usarse como valores del tipo. Cada enumerador es una constante cuyo tipo es la enumeración.

Crear una enumeración requiere el uso de la palabra clave **enum** . La forma general de un tipo de enumeración es -

```
enum enum-name { list of names } var-list;
```

Aquí, el nombre de enumeración es el nombre de tipo de la enumeración. La lista de nombres está separada por comas.

Por ejemplo, el siguiente código define una enumeración de colores llamados colores y la variable c de tipo color. Finalmente, a c se le asigna el valor "azul".

```
enum color { red, green, blue } c;
c = blue;
```

Por defecto, el valor del primer nombre es 0, el segundo nombre tiene el valor 1 y el tercero tiene el valor 2, y así sucesivamente. Pero puede dar un nombre, un valor específico agregando un inicializador. Por ejemplo, en la siguiente enumeración, el **verde** tendrá el valor 5.

```
enum color { red, green = 5, blue };
```

Aquí, el **azul** tendrá un valor de 6 porque cada nombre será uno mayor que el anterior.

Tipos de variables de C ++

Una variable nos proporciona almacenamiento con nombre que nuestros programas pueden manipular. Cada variable en C ++ tiene un tipo específico, que determina el tamaño y el diseño de la memoria de la variable; el rango de

valores que se pueden almacenar dentro de esa memoria; y el conjunto de operaciones que se pueden aplicar a la variable.

El nombre de una variable puede estar compuesto de letras, dígitos y el carácter de subrayado. Debe comenzar con una letra o un guión bajo. Las letras mayúsculas y minúsculas son distintas porque C ++ distingue entre mayúsculas y minúsculas.

Existen los siguientes tipos básicos de variables en C ++ como se explica en el último capítulo:

No Señor	Tipo y descripción
1	**bool** Almacena el valor verdadero o falso.
2	**carbonizarse** Típicamente un solo octeto (un byte). Este es un tipo entero.
3	**En t** El tamaño más natural de entero para la máquina.
4 4	**flotador** Un valor de coma flotante de precisión simple.
5 5	**doble** Un valor de coma flotante de doble precisión.
6 6	**vacío** Representa la ausencia de tipo.
7 7	**wchar_t** Un amplio tipo de personaje.

C ++ también permite definir varios otros tipos de variables, que cubriremos en capítulos posteriores como **Enumeración, Puntero, Matriz, Referencia, Estructuras de datos** y **Clases** .

La siguiente sección cubrirá cómo definir, declarar y usar varios tipos de variables.

Definición variable en C ++

Una definición de variable le dice al compilador dónde y cuánto almacenamiento crear para la variable. Una definición de variable especifica un tipo de datos y contiene una lista de una o más variables de ese tipo de la siguiente manera:

```
type variable_list;
```

Aquí, **type** debe ser un tipo de datos válido de C ++ que incluya char, w_char, int, float, double, bool o cualquier objeto definido por el usuario, etc., y **variable_list** puede constar de uno o más nombres de identificadores separados por comas. Aquí se muestran algunas declaraciones válidas:

```
int    i, j, k;
char   c, ch;
float  f, salary;
double d;
```

La línea **int i, j, k;** ambos declaran y definen las variables i, j y k; que le indica al compilador que cree variables llamadas i, j y k de tipo int.

Las variables se pueden inicializar (asignar un valor inicial) en su declaración. El inicializador consiste en un signo igual seguido de una expresión constante de la siguiente manera:

```
type variable_name = value;
```

Algunos ejemplos son:

```
extern int d = 3, f = 5;    // declaration of d and f.
int d = 3, f = 5;           // definition and initializing d
and f.
byte z = 22;                // definition and initializes z.
char x = 'x';               // the variable x has the value
'x'.
```

Para la definición sin un inicializador: las variables con duración de almacenamiento estático se inicializan implícitamente con NULL (todos los bytes tienen el valor 0); El valor inicial de todas las demás variables no está definido.

Declaración variable en C ++

Una declaración de variable garantiza al compilador que existe una variable con el tipo y nombre dados, de modo que el compilador proceda a una compilación posterior sin necesidad de detalles completos sobre la variable. Una declaración de variable tiene su significado solo en el momento de la compilación, el compilador necesita una definición de variable real en el momento de la vinculación del programa.

Una declaración de variable es útil cuando está utilizando múltiples archivos y define su variable en uno de los archivos que estarán disponibles al momento de vincular el programa. Utilizará la palabra clave **extern** para declarar una variable en cualquier lugar. Aunque puede declarar una variable varias veces

en su programa C ++, solo se puede definir una vez en un archivo, una función o un bloque de código.

Ejemplo

Pruebe el siguiente ejemplo donde se ha declarado una variable en la parte superior, pero se ha definido dentro de la función principal:

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

// Variable declaration:
extern int a, b;
extern int c;
extern float f;

int main () {
   // Variable definition:
   int a, b;
   int c;
   float f;

   // actual initialization
   a = 10;
   b = 20;
   c = a + b;

   cout << c << endl ;

   f = 70.0/3.0;
   cout << f << endl ;

   return 0;
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
30
23.3333
```

El mismo concepto se aplica a la declaración de función donde se proporciona un nombre de función en el momento de su declaración y su definición real se puede dar en cualquier otro lugar. Por ejemplo

```
// function declaration
int func();
int main() {
   // function call
   int i = func();
}

// function definition
int func() {
```

```
   return 0;
}
```

Lvalues y Rvalues

Hay dos tipos de expresiones en C ++:

- **lvalue** : las expresiones que se refieren a una ubicación de memoria se llaman expresiones "lvalue". Un valor l puede aparecer como el lado izquierdo o derecho de una tarea.
- **rvalue** : el término rvalue se refiere a un valor de datos que se almacena en alguna dirección en la memoria. Un valor r es una expresión que no puede tener un valor asignado, lo que significa que un valor r puede aparecer en el lado derecho pero no en el lado izquierdo de una asignación.

Las variables son valores y, por lo tanto, pueden aparecer en el lado izquierdo de una tarea. Los literales numéricos son valores y, por lo tanto, no se pueden asignar y no pueden aparecer en el lado izquierdo. Lo siguiente es una declaración válida:

```
int g = 20;
```

Pero lo siguiente no es una declaración válida y generaría un error en tiempo de compilación:

```
10 = 20;
```

Alcance variable en C ++

Un alcance es una región del programa y, en términos generales, hay tres lugares donde se pueden declarar variables:

- Dentro de una función o un bloque que se llama variables locales,
- En la definición de parámetros de función que se llama parámetros formales.
- Fuera de todas las funciones que se llama variables globales.

Aprenderemos qué es una función y su parámetro en capítulos posteriores. Aquí déjenos explicar cuáles son las variables locales y globales.

Variables Locales

Las variables que se declaran dentro de una función o bloque son variables locales. Solo pueden ser utilizados por declaraciones que están dentro de esa función o bloque de código. Las variables locales no son conocidas por funciones fuera de la suya. El siguiente es el ejemplo usando variables locales:

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

int main () {
   // Local variable declaration:
   int a, b;
```

```
   int c;

   // actual initialization
   a = 10;
   b = 20;
   c = a + b;

   cout << c;

   return 0;
}
```

Variables globales

Las variables globales se definen fuera de todas las funciones, generalmente en la parte superior del programa. Las variables globales mantendrán su valor durante toda la vida de su programa.

Se puede acceder a una variable global mediante cualquier función. Es decir, una variable global está disponible para su uso en todo su programa después de su declaración. El siguiente es el ejemplo que usa variables globales y locales:

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

// Global variable declaration:
int g;

int main () {
   // Local variable declaration:
   int a, b;

   // actual initialization
   a = 10;
   b = 20;
   g = a + b;

   cout << g;

   return 0;
}
```

Un programa puede tener el mismo nombre para las variables locales y globales, pero el valor de la variable local dentro de una función tendrá preferencia. Por ejemplo

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

// Global variable declaration:
```

```
int g = 20;

int main () {
   // Local variable declaration:
   int g = 10;

   cout << g;

   return 0;
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
10
```

Inicializando variables locales y globales

Cuando se define una variable local, el sistema no la inicializa, debe inicializarla usted mismo. El sistema inicializa automáticamente las variables globales cuando las define de la siguiente manera:

Tipo de datos	Inicializador
En t	0 0
carbonizarse	'\ 0'
flotador	0 0
doble	0 0
puntero	NULO

Es una buena práctica de programación inicializar las variables correctamente, de lo contrario, a veces el programa produciría resultados inesperados.

Constantes / literales de C ++

Las constantes se refieren a valores fijos que el programa no puede alterar y se denominan **literales** .

Las constantes pueden ser de cualquiera de los tipos de datos básicos y pueden dividirse en números enteros, números de coma flotante, caracteres, cadenas y valores booleanos.

Una vez más, las constantes se tratan como variables regulares, excepto que sus valores no pueden modificarse después de su definición.

Literales enteros

Un literal entero puede ser una constante decimal, octal o hexadecimal. Un prefijo especifica la base o la raíz: 0x o 0X para hexadecimal, 0 para octal y nada para decimal.

Un literal entero también puede tener un sufijo que es una combinación de U y L, para unsigned y long, respectivamente. El sufijo puede ser mayúscula o minúscula y puede estar en cualquier orden.

Aquí hay algunos ejemplos de literales enteros:

```
212         // Legal
215u        // Legal
0xFeeL      // Legal
078         // Illegal: 8 is not an octal digit
032UU       // Illegal: cannot repeat a suffix
```

Los siguientes son otros ejemplos de varios tipos de literales enteros:

```
85         // decimal
0213       // octal
0x4b       // hexadecimal
30         // int
30u        // unsigned int
30l        // long
30ul       // unsigned long
```

Literales de punto flotante

Un literal de coma flotante tiene una parte entera, un punto decimal, una parte fraccionaria y una parte exponente. Puede representar literales de coma flotante en forma decimal o exponencial.

Al representar con forma decimal, debe incluir el punto decimal, el exponente o ambos, y al representar con forma exponencial, debe incluir la parte entera, la parte fraccional o ambas. El exponente con signo es introducido por e o E.

Aquí hay algunos ejemplos de literales de coma flotante:

```
3.14159       // Legal
314159E-5L    // Legal
510E          // Illegal: incomplete exponent
210f          // Illegal: no decimal or exponent
.e55          // Illegal: missing integer or fraction
```

Literales booleanos

Hay dos literales booleanos y son parte de palabras clave estándar de C ++:

- Un valor de **verdadero que** representa verdadero.
- Un valor de **falso que** representa falso.

No debe considerar el valor de verdadero igual a 1 y el valor de falso igual a 0.

Literales de personajes

Los literales de caracteres están encerrados entre comillas simples. Si el literal comienza con L (solo mayúsculas), es un literal de caracteres anchos (por ejemplo, L'x ') y debe almacenarse en el tipo de variable **wchar_t** . De lo contrario, es un literal de caracteres estrechos (por ejemplo, 'x') y se puede almacenar en una variable simple de tipo **char** .

Un literal de caracteres puede ser un carácter simple (por ejemplo, 'x'), una secuencia de escape (por ejemplo, '\ t') o un carácter universal (por ejemplo, '\ u02C0').

Hay ciertos caracteres en C ++ cuando van precedidos de una barra diagonal inversa que tendrán un significado especial y se utilizan para representar como nueva línea (\ n) o tabulación (\ t). Aquí, tiene una lista de algunos de estos códigos de secuencia de escape:

Secuencia de escape	Sentido
\\	\ personaje
\ '	' personaje
\ "	" personaje
\?	? personaje
\un	Alerta o campana
\si	Retroceso
\F	Alimentación de formulario
\norte	Nueva línea
\ r	Retorno de carro
\ t	Pestaña horizontal

\ v	Pestaña vertical
\ ooo	Número octal de uno a tres dígitos
\ xhh. . .	Número hexadecimal de uno o más dígitos.

El siguiente es el ejemplo para mostrar algunos caracteres de secuencia de escape:

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

int main() {
   cout << "Hello\tWorld\n\n";
   return 0;
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Hello   World
```

Literales de cuerda

Los literales de cadena están entre comillas dobles. Una cadena contiene caracteres que son similares a los literales de caracteres: caracteres simples, secuencias de escape y caracteres universales.

Puede dividir una línea larga en varias líneas usando literales de cadena y separarlas usando espacios en blanco.

Aquí hay algunos ejemplos de literales de cadena. Las tres formas son cadenas idénticas.

```
"hello, dear"

"hello, \

dear"

"hello, " "d" "ear"
```

Definiendo constantes

Hay dos formas simples en C ++ para definir constantes:

- Usando el preprocesador **#define** .
- Usando la palabra clave **const** .

El preprocesador #define

A continuación se muestra la forma de usar el preprocesador #define para definir una constante:

```
#define identifier value
```

El siguiente ejemplo lo explica en detalle:

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

#define LENGTH 10
#define WIDTH  5
#define NEWLINE '\n'

int main() {
   int area;

   area = LENGTH * WIDTH;
   cout << area;
   cout << NEWLINE;
   return 0;
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
50
```

La palabra clave const

Puede usar el prefijo **const** para declarar constantes con un tipo específico de la siguiente manera:

```
const type variable = value;
```

El siguiente ejemplo lo explica en detalle:

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

int main() {
   const int  LENGTH = 10;
   const int  WIDTH  = 5;
   const char NEWLINE = '\n';
   int area;

   area = LENGTH * WIDTH;
   cout << area;
   cout << NEWLINE;
   return 0;
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
50
```

Tenga en cuenta que es una buena práctica de programación definir constantes en CAPITALS.

Tipos de modificadores de C ++

C ++ permite que los tipos de datos **char, int** y **double** tengan modificadores que los preceden. Se utiliza un modificador para alterar el significado del tipo base para que se ajuste con mayor precisión a las necesidades de diversas situaciones.

Los modificadores de tipo de datos se enumeran aquí:

- firmado
- no firmado
- largo
- corto

Los modificadores con **signo, sin signo, largos** y **cortos** se pueden aplicar a tipos de base entera. Además, **firmado** y **sin firmar** se pueden aplicar a char, y **siempre** se puede aplicar al doble.

Los modificadores **firmados** y **sin firmar** también se puede utilizar como prefijo a **largas** o **cortas** modificadores. Por ejemplo, **unsigned long int** .

C ++ permite una notación abreviada para declarar enteros **sin signo, cortos** o **largos** . Simplemente puede usar la palabra **sin signo, corta** o **larga,** sin **int** . Implica automáticamente **int** . Por ejemplo, las siguientes dos declaraciones declaran variables enteras sin signo.

```
unsigned x;
unsigned int y;
```

Para comprender la diferencia entre la forma en que C ++ interpreta los modificadores de enteros con y sin signo, debe ejecutar el siguiente programa corto:

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

/* This program shows the difference between
   * signed and unsigned integers.
*/
int main() {
   short int i;           // a signed short integer
   short unsigned int j;  // an unsigned short integer

   j = 50000;

   i = j;
   cout << i << " " << j;
```

```
   return 0;
}
```

Cuando se ejecuta este programa, a continuación se muestra la salida:

```
-15536 50000
```

El resultado anterior se debe a que el patrón de bits que representa 50,000 como un entero corto sin signo se interpreta como -15,536 por un corto.

Calificadores de tipo en C ++

Los calificadores de tipo proporcionan información adicional sobre las variables que preceden.

No Señor	Calificador y significado
1	**const** Su programa no puede cambiar los objetos de tipo **const** durante la ejecución.
2	**volátil** El modificador **volátil** le dice al compilador que el valor de una variable puede cambiarse de formas no especificadas explícitamente por el programa.
3	**restringir** Un puntero calificado por **restringir** es inicialmente el único medio por el cual se puede acceder al objeto al que apunta. Solo C99 agrega un nuevo calificador de tipo llamado restringir.

Clases de almacenamiento en C ++

Una clase de almacenamiento define el alcance (visibilidad) y el tiempo de vida de las variables y / o funciones dentro de un programa C ++. Estos especificadores preceden al tipo que modifican. Existen las siguientes clases de almacenamiento, que se pueden usar en un programa C ++

- auto
- Registrarse
- estático
- externo
- mudable

La clase de almacenamiento automático

La clase de almacenamiento **automático** es la clase de almacenamiento predeterminada para todas las variables locales.

```
{
   int mount;
   auto int month;
}
```

El ejemplo anterior define dos variables con la misma clase de almacenamiento, auto solo puede usarse dentro de funciones, es decir, variables locales.

La clase de almacenamiento de registro

La clase de almacenamiento de **registros** se utiliza para definir variables locales que deberían almacenarse en un registro en lugar de RAM. Esto significa que la variable tiene un tamaño máximo igual al tamaño del registro (generalmente una palabra) y no se le puede aplicar el operador '&' unario (ya que no tiene una ubicación de memoria).

```
{
   register int  miles;
}
```

El registro solo debe usarse para variables que requieren acceso rápido, como contadores. También se debe tener en cuenta que la definición de "registro" no significa que la variable se almacenará en un registro. Significa que PODRÍA almacenarse en un registro dependiendo del hardware y las restricciones de implementación.

La clase de almacenamiento estático

La clase de almacenamiento **estático** indica al compilador que mantenga una variable local en existencia durante el tiempo de vida del programa en lugar de crearla y destruirla cada vez que entra y sale del alcance. Por lo tanto, hacer que las variables locales sean estáticas les permite mantener sus valores entre llamadas a funciones.

El modificador estático también se puede aplicar a variables globales. Cuando se hace esto, hace que el alcance de esa variable se restrinja al archivo en el que se declara.

En C ++, cuando se usa static en un miembro de datos de clase, hace que solo una copia de ese miembro sea compartida por todos los objetos de su clase.

```
#include <iostream>

// Function declaration
void func(void);

static int count = 10; /* Global variable */
```

```
main() {
   while(count--) {
      func();
   }

   return 0;
}

// Function definition
void func( void ) {
   static int i = 5; // local static variable
   i++;
   std::cout << "i is " << i ;
   std::cout << " and count is " << count << std::endl;
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
i is 6 and count is 9
i is 7 and count is 8
i is 8 and count is 7
i is 9 and count is 6
i is 10 and count is 5
i is 11 and count is 4
i is 12 and count is 3
i is 13 and count is 2
i is 14 and count is 1
i is 15 and count is 0
```

La clase de almacenamiento externo

La clase de almacenamiento **externo** se usa para dar una referencia de una variable global que es visible para TODOS los archivos de programa. Cuando usa 'extern', la variable no se puede inicializar, ya que todo lo que hace es apuntar el nombre de la variable a una ubicación de almacenamiento que se haya definido previamente.

Cuando tiene varios archivos y define una variable o función global, que también se usará en otros archivos, *extern* se usará en otro archivo para dar referencia a la variable o función definida. Solo para comprender *extern* se utiliza para declarar una variable o función global en otro archivo.

El modificador externo se usa más comúnmente cuando hay dos o más archivos que comparten las mismas variables o funciones globales como se explica a continuación.

Primer archivo: main.cpp

```
#include <iostream>
int count ;
extern void write_extern();
```

```
main() {
   count = 5;
   write_extern();
}
```

Segundo archivo: support.cpp

```
#include <iostream>

extern int count;

void write_extern(void) {
   std::cout << "Count is " << count << std::endl;
}
```

Aquí, la palabra clave *externa* se está utilizando para declarar el recuento en otro archivo. Ahora compile estos dos archivos de la siguiente manera:

```
$g++ main.cpp support.cpp -o write
```

Esto producirá un programa ejecutable de **escritura** , intente ejecutar **escritura** y verifique el resultado de la siguiente manera:

```
$./write
5
```

La clase de almacenamiento mutable

El especificador **mutable** se aplica solo a los objetos de clase, que se analizan más adelante en este tutorial. Permite que un miembro de un objeto anule la función miembro const. Es decir, un miembro mutable puede ser modificado por una función miembro const.

Operadores en C ++

Un operador es un símbolo que le dice al compilador que realice manipulaciones matemáticas o lógicas específicas. C ++ es rico en operadores integrados y proporciona los siguientes tipos de operadores:

- Operadores aritméticos
- Operadores relacionales
- Operadores logicos
- Operadores bit a bit
- Operadores de Asignación
- Operadores diversos

Este capítulo examinará los operadores aritméticos, relacionales, lógicos, bit a bit, y otros operadores uno por uno.

Operadores aritméticos

Los siguientes operadores aritméticos son compatibles con el lenguaje C ++:

Suponga que la variable A tiene 10 y la variable B tiene 20, entonces -

Mostrar ejemplos

Operador	Descripción	Ejemplo
+	Agrega dos operandos	A + B dará 30
-	Resta el segundo operando del primero	A - B dará -10
* *	Multiplica ambos operandos	A * B dará 200
/ /	Divide el numerador entre el numerador	B / A dará 2
%	Operador de módulo y resto después de una división entera	B% A dará 0
++	Operador de incremento , aumenta el valor entero en uno	A ++ dará 11
-	Operador de disminución, disminuye el valor entero en uno	A-- dará 9

Operadores relacionales

Existen los siguientes operadores relacionales compatibles con el lenguaje C ++

Suponga que la variable A tiene 10 y la variable B tiene 20, entonces -

Mostrar ejemplos

Operador	Descripción	Ejemplo
==	Comprueba si los valores de dos operandos son iguales o no, en caso afirmativo, la condición se vuelve verdadera.	(A == B) no es cierto.
! =	Comprueba si los valores de dos operandos son iguales o no, si los valores no son iguales, la condición se	(A! = B) es cierto.

	vuelve verdadera.	
>	Comprueba si el valor del operando izquierdo es mayor que el valor del operando derecho, en caso afirmativo, la condición se vuelve verdadera.	(A> B) no es cierto.
<	Comprueba si el valor del operando izquierdo es menor que el valor del operando derecho, en caso afirmativo, la condición se vuelve verdadera.	(A <B) es cierto.
> =	Comprueba si el valor del operando izquierdo es mayor o igual que el valor del operando derecho, en caso afirmativo, la condición se vuelve verdadera.	(A> = B) no es cierto.
<=	Comprueba si el valor del operando izquierdo es menor o igual que el valor del operando derecho, en caso afirmativo, la condición se vuelve verdadera.	(A <= B) es cierto.

Operadores logicos

Existen los siguientes operadores lógicos compatibles con el lenguaje C ++.

Suponga que la variable A contiene 1 y la variable B contiene 0, entonces -

Mostrar ejemplos

Operador	Descripción	Ejemplo
&&	Llamado operador lógico AND. Si ambos operandos son distintos de cero, la condición se vuelve verdadera.	(A y B) es falso.
||	Llamado Lógico O Operador. Si alguno de los dos operandos es distinto de cero, la condición se vuelve verdadera.	(A || B) es cierto.
!	Llamado operador lógico NO. Se usa para invertir el estado lógico de su operando. Si una condición es	! (A y B) es cierto.

	verdadera, entonces el operador lógico NO hará falso.	

Operadores bit a bit

El operador bit a bit trabaja en bits y realiza la operación bit a bit. Las tablas de verdad para &, | y ^ son las siguientes:

pag	q	p & q	p | q	p ^ q
0 0	0 0	0 0	0 0	0 0
0 0	1	0 0	1	1
1	1	1	1	0 0
1	0 0	0 0	1	1

Suponga que si A = 60; y B = 13; ahora en formato binario serán los siguientes:

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A | B = 0011 1101

A ^ B = 0011 0001

~ A = 1100 0011

Los operadores de Bitwise admitidos por el lenguaje C ++ se enumeran en la siguiente tabla. Suponga que la variable A tiene 60 y la variable B tiene 13, entonces -

Mostrar ejemplos

Operador	Descripción	Ejemplo
Y	El operador binario AND copia un poco al resultado si existe en ambos operandos.	(A y B) dará 12, que es 0000 1100

El |	El operador binario O copia un bit si existe en cualquiera de los operandos.	(A | B) dará 61, que es 0011 1101
^	El operador binario XOR copia el bit si está establecido en un operando pero no en ambos.	(A ^ B) dará 49, que es 0011 0001
~	El operador de complemento de binarios es unario y tiene el efecto de "voltear" los bits.	(~ A) dará -61, que es 1100 0011 en forma de complemento a 2 debido a un número binario con signo.
<<	Operador binario de desplazamiento a la izquierda. El valor de los operandos de la izquierda se mueve hacia la izquierda por la cantidad de bits especificados por el operando de la derecha.	Un << 2 dará 240, que es 1111 0000
>>	Operador binario de desplazamiento a la derecha. El valor de los operandos de la izquierda se mueve hacia la derecha por la cantidad de bits especificados por el operando de la derecha.	A >> 2 dará 15, que es 0000 1111

Operadores de Asignación

Los siguientes operadores de asignación son compatibles con el lenguaje C ++:

Mostrar ejemplos

Operador	Descripción	Ejemplo
=	Operador de asignación simple, asigna valores de operandos del lado derecho al operando del lado izquierdo.	C = A + B asignará el valor de A + B a C
+ =	Agregar operador de asignación AND, agrega el operando derecho al operando izquierdo y asigna el resultado al operando izquierdo.	C + = A es equivalente a C = C + A

- =	Restar operador de asignación AND, resta el operando derecho del operando izquierdo y asigna el resultado al operando izquierdo.	C - = A es equivalente a C = C - A
* =	Operador de multiplicación Y asignación. Multiplica el operando derecho con el operando izquierdo y asigna el resultado al operando izquierdo.	C * = A es equivalente a C = C * A
/ =	Operador Dividir Y asignar, divide el operando izquierdo con el operando derecho y asigna el resultado al operando izquierdo.	C / = A es equivalente a C = C / A
% =	Operador de asignación de módulo Y, toma módulo usando dos operandos y asigna el resultado al operando izquierdo.	C% = A es equivalente a C = C% A
<< =	Desplazamiento a la izquierda Y operador de asignación.	C << = 2 es lo mismo que C = C << 2
>> =	Desplazamiento a la derecha Y operador de asignación.	C >> = 2 es lo mismo que C = C >> 2
& =	Operador de asignación Y a nivel de bit.	C & = 2 es lo mismo que C = C & 2
^ =	Bitwise operador exclusivo y operador de asignación.	C ^ = 2 es lo mismo que C = C ^ 2
| =	Bitwise inclusivo OR y operador de asignación.	C | = 2 es lo mismo que C = C | 2

Operadores diversos

La siguiente tabla enumera algunos otros operadores que admite C ++.

No Señor	Operador y Descripción

1	**tamaño de** El operador sizeof devuelve el tamaño de una variable. Por ejemplo, sizeof (a), donde 'a' es entero, y devolverá 4.
2	**Condición? X: Y** Operador condicional (?) . Si la condición es verdadera, devuelve el valor de X; de lo contrario, devuelve el valor de Y.
3	**,** El operador de coma hace que se realice una secuencia de operaciones. El valor de toda la expresión de coma es el valor de la última expresión de la lista separada por comas.
4 4	**. (punto) y -> (flecha)** Los operadores miembros se utilizan para hacer referencia a miembros individuales de clases, estructuras y sindicatos.
5 5	**Emitir** Los operadores de conversión convierten un tipo de datos a otro. Por ejemplo, int (2.2000) devolvería 2.
6 6	**Y** Operador de puntero y devuelve la dirección de una variable. Por ejemplo & a; dará la dirección real de la variable.
7 7	*** *** El operador de puntero * es un puntero a una variable. Por ejemplo * var; apuntará a una variable var.

Precedencia de operadores en C ++

La precedencia del operador determina la agrupación de términos en una expresión. Esto afecta cómo se evalúa una expresión. Ciertos operadores tienen mayor prioridad que otros; por ejemplo, el operador de multiplicación tiene mayor prioridad que el operador de suma:

Por ejemplo x = 7 + 3 * 2; aquí, a x se le asigna 13, no 20 porque el operador * tiene mayor prioridad que +, por lo que primero se multiplica por 3 * 2 y luego se suma a 7.

Aquí, los operadores con la precedencia más alta aparecen en la parte superior de la tabla, aquellos con la más baja aparecen en la parte

inferior. Dentro de una expresión, los operadores de mayor precedencia se evaluarán primero.

Mostrar ejemplos

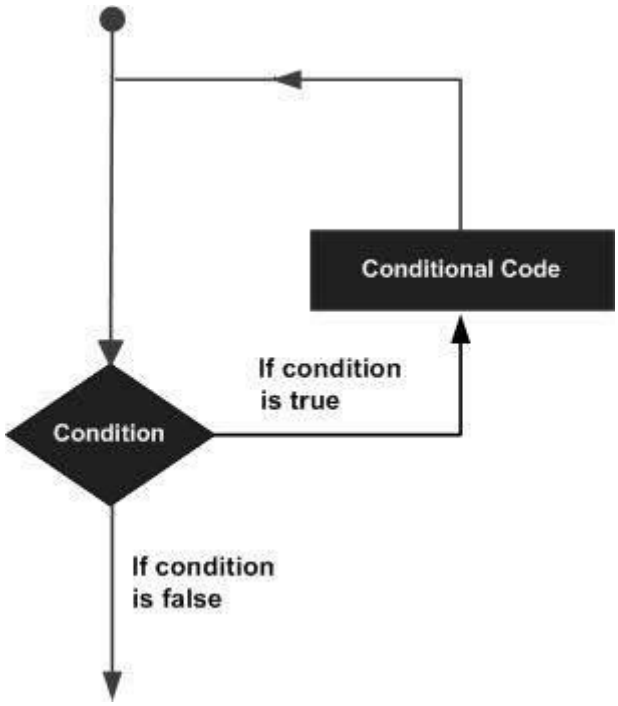
Categoría	Operador	Asociatividad
Sufijo	() [] ->. ++ - -	De izquierda a derecha
Unario	+ -! ~ ++ - - (tipo) * y sizeof	De derecha a izquierda
Multiplicativo	* /%	De izquierda a derecha
Aditivo	+ -	De izquierda a derecha
Cambio	<< >>	De izquierda a derecha
Relacional	<<=>> =	De izquierda a derecha
Igualdad	==! =	De izquierda a derecha
Bitwise Y	Y	De izquierda a derecha
Bitwise XOR	^	De izquierda a derecha
Bitwise O	El |	De izquierda a derecha
Y lógico	&&	De izquierda a derecha
O lógico	||	De izquierda a derecha
Condicional	?:	De derecha a izquierda
Asignación	= + = - = * = / =% = >> = << = & = ^ = | =	De derecha a izquierda
Coma	,	De izquierda a derecha

Tipos de bucles C ++

Puede haber una situación en la que necesite ejecutar un bloque de código varias veces. En general, las instrucciones se ejecutan secuencialmente: la primera instrucción de una función se ejecuta primero, seguida de la segunda, y así sucesivamente.

Los lenguajes de programación proporcionan diversas estructuras de control que permiten rutas de ejecución más complicadas.

Una declaración de bucle nos permite ejecutar una declaración o un grupo de declaraciones varias veces y lo siguiente es lo general de una declaración de bucle en la mayoría de los lenguajes de programación:



El lenguaje de programación C ++ proporciona el siguiente tipo de bucles para manejar los requisitos de bucle.

No Señor	Tipo de bucle y descripción
1	mientras bucle Repite una declaración o grupo de declaraciones mientras una condición dada es verdadera. Prueba la condición antes de ejecutar el cuerpo del bucle.
2	en bucle Ejecute una secuencia de declaraciones varias veces y abrevia el código que administra la variable de bucle.
3	hacer ... mientras bucle

	Como una declaración 'while', excepto que prueba la condición al final del cuerpo del bucle.
4 4	bucles anidados Puede usar uno o más bucles dentro de cualquier otro bucle 'while', 'for' o 'do.. while'.

Declaraciones de control de bucle

Las instrucciones de control de bucle cambian la ejecución de su secuencia normal. Cuando la ejecución deja un ámbito, todos los objetos automáticos que se crearon en ese ámbito se destruyen.

C ++ admite las siguientes declaraciones de control.

No Señor	Declaración de control y descripción
1	declaración de ruptura Termina la instrucción **loop** o **switch** y transfiere la ejecución a la instrucción que sigue inmediatamente al loop o switch.
2	continuar declaración Hace que el bucle omita el resto de su cuerpo e inmediatamente vuelva a probar su condición antes de reiterar.
3	ir a la declaración Transfiere el control a la declaración etiquetada. Aunque no se recomienda usar la instrucción goto en su programa.

El bucle infinito

Un bucle se convierte en bucle infinito si una condición nunca se vuelve falsa. El bucle **for** se usa tradicionalmente para este propósito. Como ninguna de las tres expresiones que forman el bucle 'for' son necesarias, puede hacer un bucle sin fin dejando vacía la expresión condicional.

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

int main () {
   for( ; ; ) {
      printf("This loop will run forever.\n");
   }
```

```
   return 0;
}
```

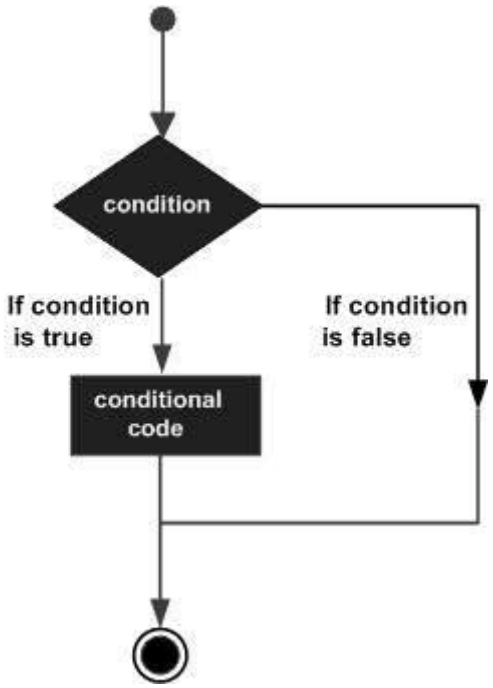
Cuando la expresión condicional está ausente, se supone que es verdadera. Es posible que tenga una expresión de inicialización e incremento, pero los programadores de C ++ usan más comúnmente la construcción 'for (;;)' para significar un bucle infinito.

NOTA - Puede terminar un ciclo infinito presionando las teclas Ctrl + C.

Declaraciones de toma de decisiones en C ++

Las estructuras de toma de decisiones requieren que el programador especifique una o más condiciones para ser evaluadas o probadas por el programa, junto con una declaración o declaraciones que se ejecutarán si se determina que la condición es verdadera, y opcionalmente, otras declaraciones que se ejecutarán si la condición se determina que es falso

A continuación se presenta la forma general de una estructura de toma de decisiones típica que se encuentra en la mayoría de los lenguajes de programación:



El lenguaje de programación C ++ proporciona los siguientes tipos de declaraciones de toma de decisiones.

No Señor	Declaración y descripción

1	si la declaración Una declaración 'if' consiste en una expresión booleana seguida de una o más declaraciones.
2	si ... otra declaración Una instrucción 'if' puede ser seguida por una instrucción opcional 'else', que se ejecuta cuando la expresión booleana es falsa.
3	declaración de cambio Una declaración 'switch' permite que una variable sea probada por igualdad contra una lista de valores.
4 4	instrucciones if anidadas Puede usar una declaración 'if' o 'else if' dentro de otra (s) declaración (s) 'if' o 'else if'.
5 5	instrucciones de cambio anidadas Puede usar una declaración 'switch' dentro de otra (s) declaración (s) 'switch'.

Los ? : Operador

Hemos cubierto operador condicional "? : " En el capítulo anterior, que se puede utilizar para reemplazar las declaraciones **if ... else** . Tiene la siguiente forma general:

```
Exp1 ? Exp2 : Exp3;
```

Exp1, Exp2 y Exp3 son expresiones. Observe el uso y la colocación del colon.

El valor de un '?' la expresión se determina así: se evalúa Exp1. Si es cierto, entonces Exp2 se evalúa y se convierte en el valor de todo el '?' expresión. Si Exp1 es falso, se evalúa Exp3 y su valor se convierte en el valor de la expresión.

Funciones de C ++

Una función es un grupo de declaraciones que juntas realizan una tarea. Cada programa C ++ tiene al menos una función, que es **main ()** , y todos los programas más triviales pueden definir funciones adicionales.

Puede dividir su código en funciones separadas. La forma en que divida su código entre diferentes funciones depende de usted, pero lógicamente la división generalmente es tal que cada función realiza una tarea específica.

Una **declaración de** función le dice al compilador sobre el nombre de una función, el tipo de retorno y los parámetros. Una **definición de** función proporciona el cuerpo real de la función.

La biblioteca estándar de C ++ proporciona numerosas funciones integradas a las que puede llamar su programa. Por ejemplo, la función **strcat ()** para concatenar dos cadenas, la función **memcpy ()** para copiar una ubicación de memoria en otra ubicación y muchas más funciones.

Una función se conoce con varios nombres como un método o una subrutina o un procedimiento, etc.

Definiendo una función

La forma general de una definición de función C ++ es la siguiente:

```
return_type function_name( parameter list ) {
   body of the function
}
```

Una definición de función C ++ consiste en un encabezado de función y un cuerpo de función. Aquí están todas las partes de una función:

- **Tipo de retorno** : una función puede devolver un valor. El **return_type** es el tipo de datos del valor devuelve la función. Algunas funciones realizan las operaciones deseadas sin devolver un valor. En este caso, return_type es la palabra clave **void** .
- **Nombre de la función** : este es el nombre real de la función. El nombre de la función y la lista de parámetros juntos constituyen la firma de la función.
- **Parámetros** : un parámetro es como un marcador de posición. Cuando se invoca una función, pasa un valor al parámetro. Este valor se conoce como parámetro o argumento real. La lista de parámetros se refiere al tipo, orden y número de parámetros de una función. Los parámetros son opcionales; es decir, una función puede no contener parámetros.
- **Cuerpo de** la función: el cuerpo de la función contiene una colección de declaraciones que definen lo que hace la función.

Ejemplo

El siguiente es el código fuente de una función llamada **max ()** . Esta función toma dos parámetros num1 y num2 y devuelve el mayor de ambos:

```
// function returning the max between two numbers

int max(int num1, int num2) {
   // local variable declaration
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

Declaraciones de funciones

Una **declaración de** función le dice al compilador sobre el nombre de una función y cómo llamarla. El cuerpo real de la función se puede definir por separado.

Una declaración de función tiene las siguientes partes:

```
return_type function_name( parameter list );
```

Para la función max definida anteriormente (), la siguiente es la declaración de función:

```
int max(int num1, int num2);
```

Los nombres de los parámetros no son importantes en la declaración de funciones, solo se requiere su tipo, por lo que a continuación también se incluye una declaración válida

```
int max(int, int);
```

La declaración de función es necesaria cuando define una función en un archivo fuente y llama a esa función en otro archivo. En tal caso, debe declarar la función en la parte superior del archivo que llama a la función.

Llamar a una función

Al crear una función C ++, usted da una definición de lo que la función tiene que hacer. Para usar una función, deberá llamar o invocar esa función.

Cuando un programa llama a una función, el control del programa se transfiere a la función llamada. Una función llamada realiza una tarea definida y cuando se ejecuta su declaración return o cuando se alcanza su llave de cierre que finaliza la función, devuelve el control del programa al programa principal.

Para llamar a una función, simplemente necesita pasar los parámetros requeridos junto con el nombre de la función, y si la función devuelve un valor, puede almacenar el valor devuelto. Por ejemplo

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

// function declaration
int max(int num1, int num2);

int main () {
   // local variable declaration:
   int a = 100;
   int b = 200;
   int ret;

   // calling a function to get max value.
   ret = max(a, b);
   cout << "Max value is : " << ret << endl;
```

```
   return 0;
}

// function returning the max between two numbers
int max(int num1, int num2) {
   // local variable declaration
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

Mantuve la función max () junto con la función main () y compilé el código fuente. Mientras ejecuta el ejecutable final, produciría el siguiente resultado:

```
Max value is : 200
```

Argumentos de funciones

Si una función va a usar argumentos, debe declarar variables que acepten los valores de los argumentos. Estas variables se llaman **parámetros formales** de la función.

Los parámetros formales se comportan como otras variables locales dentro de la función y se crean al ingresar a la función y se destruyen al salir.

Al llamar a una función, hay dos formas en que los argumentos se pueden pasar a una función:

No Señor	Tipo de llamada y descripción
1	Llamar por valor Este método copia el valor real de un argumento en el parámetro formal de la función. En este caso, los cambios realizados en el parámetro dentro de la función no tienen efecto en el argumento.
2	Llamada por puntero Este método copia la dirección de un argumento en el parámetro formal. Dentro de la función, la dirección se utiliza para acceder al argumento real utilizado en la llamada. Esto significa que los cambios realizados en el parámetro afectan el argumento.
3	Llamar por referencia Este método copia la referencia de un argumento en el parámetro formal. Dentro

	de la función, la referencia se utiliza para acceder al argumento real utilizado en la llamada. Esto significa que los cambios realizados en el parámetro afectan el argumento.

Por defecto, C ++ usa la **llamada por valor** para pasar argumentos. En general, esto significa que el código dentro de una función no puede alterar los argumentos utilizados para llamar a la función y el ejemplo mencionado anteriormente mientras se llama a la función max () se utiliza el mismo método.

Valores predeterminados para parámetros

Cuando define una función, puede especificar un valor predeterminado para cada uno de los últimos parámetros. Este valor se usará si el argumento correspondiente se deja en blanco al llamar a la función.

Esto se hace utilizando el operador de asignación y asignando valores para los argumentos en la definición de la función. Si no se pasa un valor para ese parámetro cuando se llama a la función, se usa el valor predeterminado dado, pero si se especifica un valor, este valor predeterminado se ignora y se usa el valor pasado. Considere el siguiente ejemplo:

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

int sum(int a, int b = 20) {
   int result;
   result = a + b;

   return (result);
}
int main () {
   // local variable declaration:
   int a = 100;
   int b = 200;
   int result;

   // calling a function to add the values.
   result = sum(a, b);
   cout << "Total value is :" << result << endl;

   // calling a function again as follows.
   result = sum(a);
   cout << "Total value is :" << result << endl;

   return 0;
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Total value is :300
```

```
Total value is :120
```

Números en C ++

Normalmente, cuando trabajamos con Números, utilizamos tipos de datos primitivos como int, short, long, float y double, etc. Los tipos de datos de números, sus posibles valores y rangos de números se explicaron al analizar los tipos de datos de C ++.

Definición de números en C ++

Ya ha definido números en varios ejemplos dados en capítulos anteriores. Aquí hay otro ejemplo consolidado para definir varios tipos de números en C ++:

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

int main () {
   // number definition:
   short  s;
   int    i;
   long   l;
   float  f;
   double d;

   // number assignments;
   s = 10;
   i = 1000;
   l = 1000000;
   f = 230.47;
   d = 30949.374;

   // number printing;
   cout << "short  s :" << s << endl;
   cout << "int    i :" << i << endl;
   cout << "long   l :" << l << endl;
   cout << "float  f :" << f << endl;
   cout << "double d :" << d << endl;

   return 0;
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
short  s :10
int    i :1000
long   l :1000000
float  f :230.47
double d :30949.4
```

Operaciones matemáticas en C ++

Además de las diversas funciones que puede crear, C ++ también incluye algunas funciones útiles que puede usar. Estas funciones están disponibles en C estándar y bibliotecas de C ++ y se llama **una función de** funciones. Estas son funciones que se pueden incluir en su programa y luego usar.

C ++ tiene un rico conjunto de operaciones matemáticas, que se pueden realizar en varios números. La siguiente tabla enumera algunas funciones matemáticas integradas útiles disponibles en C ++.

Para utilizar estas funciones, debe incluir el archivo de encabezado matemático **<cmath>** .

No Señor	Función y Propósito
1	**doble cos (doble);** Esta función toma un ángulo (como doble) y devuelve el coseno.
2	**doble pecado (doble);** Esta función toma un ángulo (como doble) y devuelve el seno.
3	**doble bronceado (doble);** Esta función toma un ángulo (como doble) y devuelve la tangente.
4 4	**doble registro (doble);** Esta función toma un número y devuelve el registro natural de ese número.
5 5	**doble pow (doble, doble);** El primero es un número que desea aumentar y el segundo es el poder que desea elevar t
6 6	**doble hipot (doble, doble);** Si pasa esta función a la longitud de dos lados de un triángulo rectángulo, le devolverá la longitud de la hipotenusa.
7 7	**doble sqrt (doble);** Pasa esta función un número y le da la raíz cuadrada.
8	**int abs (int);** Esta función devuelve el valor absoluto de un entero que se le pasa.

9 9	**fabs dobles (doble);** Esta función devuelve el valor absoluto de cualquier número decimal que se le pase.
10	**doble piso (doble);** Encuentra el número entero que es menor o igual que el argumento que se le pasó.

El siguiente es un ejemplo simple para mostrar algunas de las operaciones matemáticas:

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

int main () {
   // number definition:
   short  s = 10;
   int    i = -1000;
   long   l = 100000;
   float  f = 230.47;
   double d = 200.374;

   // mathematical operations;
   cout << "sin(d) :" << sin(d) << endl;
   cout << "abs(i)  :" << abs(i) << endl;
   cout << "floor(d) :" << floor(d) << endl;
   cout << "sqrt(f) :" << sqrt(f) << endl;
   cout << "pow( d, 2) :" << pow(d, 2) << endl;

   return 0;
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
sign(d)      :-0.634939
abs(i)       :1000
floor(d)     :200
sqrt(f)      :15.1812
pow( d, 2 )  :40149.7
```

Números aleatorios en C ++

Hay muchos casos en los que deseará generar un número aleatorio. En realidad, hay dos funciones que necesitará saber sobre la generación de números aleatorios. El primero es **rand ()** , esta función solo devolverá un número pseudoaleatorio. La forma de solucionar esto es llamar primero a la función **srand ()** .

El siguiente es un ejemplo simple para generar pocos números aleatorios. Este ejemplo hace uso de la función **time ()** para obtener el número de segundos en la hora de su sistema, para sembrar aleatoriamente la función rand () -

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

using namespace std;

int main () {
   int i,j;

   // set the seed
   srand( (unsigned)time( NULL ) );

   /* generate 10  random numbers. */
   for( i = 0; i < 10; i++ ) {
      // generate actual random number
      j = rand();
      cout <<" Random Number : " << j << endl;
   }

   return 0;
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Random Number : 1748144778
Random Number : 630873888
Random Number : 2134540646
Random Number : 219404170
Random Number : 902129458
Random Number : 920445370
Random Number : 1319072661
Random Number : 257938873
Random Number : 1256201101
Random Number : 580322989
```

Matrices C ++

C ++ proporciona una estructura de datos, **la matriz** , que almacena una colección secuencial de tamaño fijo de elementos del mismo tipo. Una matriz se usa para almacenar una colección de datos, pero a menudo es más útil pensar en una matriz como una colección de variables del mismo tipo.

En lugar de declarar variables individuales, como número0, número1, ... y número99, declara una variable de matriz como números y usa números [0], números [1] y ..., números [99] para representar variables individuales Se accede a un elemento específico en una matriz mediante un índice.

Todas las matrices consisten en ubicaciones de memoria contiguas. La dirección más baja corresponde al primer elemento y la dirección más alta al último elemento.

Declarar matrices

Para declarar una matriz en C ++, el programador especifica el tipo de elementos y la cantidad de elementos requeridos por una matriz de la siguiente manera:

```
type arrayName [ arraySize ];
```

Esto se llama una matriz de una sola dimensión. El **arraySize** debe ser una constante entera mayor que cero y el **tipo** puede ser cualquier tipo de datos C ++ válido. Por ejemplo, para declarar una matriz de 10 elementos llamada balance de tipo double, use esta declaración:

```
double balance[10];
```

Inicializando matrices

Puede inicializar los elementos de la matriz de C ++, uno por uno o usando una sola declaración de la siguiente manera:

```
double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

El número de valores entre llaves {} no puede ser mayor que el número de elementos que declaramos para la matriz entre corchetes []. El siguiente es un ejemplo para asignar un solo elemento de la matriz:

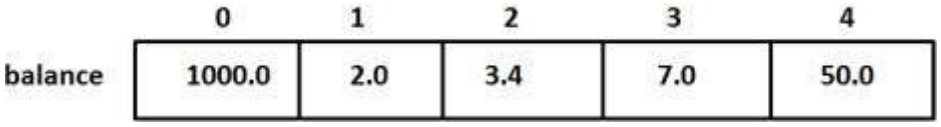
Si omite el tamaño de la matriz, se crea una matriz lo suficientemente grande como para contener la inicialización. Por lo tanto, si escribes

```
double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

Creará exactamente la misma matriz que hizo en el ejemplo anterior.

```
balance[4] = 50.0;
```

Lo anterior cesionarios declaración elemento número 5 ° en la matriz de un valor de 50,0. La matriz con el 4º índice será la 5ª , es decir, el último elemento porque todas las matrices tienen 0 como índice de su primer elemento, que también se denomina índice base. A continuación se muestra la representación gráfica de la misma matriz que discutimos anteriormente:



Acceso a elementos de matriz

Se accede a un elemento indexando el nombre de la matriz. Esto se hace colocando el índice del elemento entre corchetes después del nombre de la matriz. Por ejemplo

```
double salary = balance[9];
```

La declaración anterior se llevará a 10 ° elemento de la matriz y asignar el valor a la variable salario. El siguiente es un ejemplo, que utilizará todos los tres conceptos mencionados anteriormente, a saber. declaración, asignación y acceso a matrices -

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

#include <iomanip>
using std::setw;

int main () {

   int n[ 10 ]; // n is an array of 10 integers

   // initialize elements of array n to 0
   for ( int i = 0; i < 10; i++ ) {
      n[ i ] = i + 100; // set element at location i to i +
100
   }
   cout << "Element" << setw( 13 ) << "Value" << endl;

   // output each array element's value
   for ( int j = 0; j < 10; j++ ) {
      cout << setw( 7 )<< j << setw( 13 ) << n[ j ] << endl;
   }

   return 0;
}
```

Este programa utiliza la función **setw ()** para formatear la salida. Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Element         Value
      0           100
      1           101
      2           102
      3           103
      4           104
      5           105
      6           106
      7           107
      8           108
      9           109
```

Matrices en C ++

Las matrices son importantes para C ++ y deberían necesitar muchos más detalles. Existen algunos conceptos importantes que deben quedar claros para un programador de C ++:

No Señor	Concepto y descripción
1	Matrices multidimensionales C ++ admite matrices multidimensionales. La forma más simple de la matriz multidimensional es la matriz bidimensional.
2	Puntero a una matriz Puede generar un puntero al primer elemento de una matriz simplemente especificando el nombre de la matriz, sin ningún índice.
3	Pasar matrices a funciones Puede pasar a la función un puntero a una matriz especificando el nombre de la matriz sin un índice.
4 4	Devolver matriz de funciones C ++ permite que una función devuelva una matriz.

Cuerdas C ++

C ++ proporciona los siguientes dos tipos de representaciones de cadenas:

- La cadena de caracteres de estilo C.
- El tipo de clase de cadena introducido con Standard C ++.

La cadena de caracteres de estilo C

La cadena de caracteres de estilo C se originó en el lenguaje C y continúa siendo compatible con C ++. Esta cadena es en realidad una matriz unidimensional de caracteres que termina con un carácter **nulo** '\ 0'. Por lo tanto, una cadena terminada en nulo contiene los caracteres que comprenden la cadena seguida de un **nulo** .

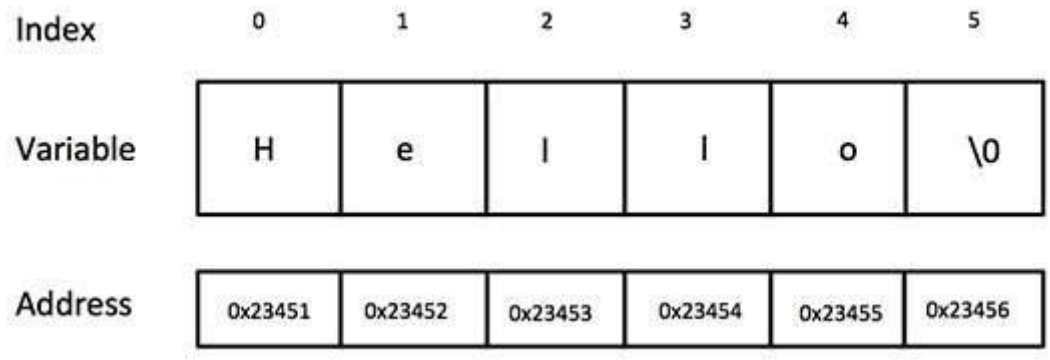
La siguiente declaración e inicialización crean una cadena que consiste en la palabra "Hola". Para mantener el carácter nulo al final de la matriz, el tamaño de la matriz de caracteres que contiene la cadena es uno más que el número de caracteres en la palabra "Hola".

```
saludo de char [6] = {'H', 'e', 'l', 'l', 'o', '\ 0'};
```

Si sigue la regla de inicialización de la matriz, puede escribir la declaración anterior de la siguiente manera:

```
char greeting [] = "Hola";
```

A continuación se presenta la memoria de la cadena definida anteriormente en C / C ++:



En realidad, no coloca el carácter nulo al final de una constante de cadena. El compilador de C ++ coloca automáticamente el '\ 0' al final de la cadena cuando inicializa la matriz. Intentemos imprimir la cadena mencionada anteriormente:

```
#include <iostream>

usando el espacio de nombres estándar ;

int main () {

   saludo de char [ 6 ] = { 'H' , 'e' , 'l' , 'l' , 'o' , '\
0' };

   cout << "Mensaje de saludo:" ;
   cout << saludo << endl ;

   devuelve 0 ; }
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Mensaje de saludo: Hola
```

C ++ admite una amplia gama de funciones que manipulan cadenas terminadas en nulo:

No Señor	Función y Propósito
1	**strcpy (s1, s2);** Copia la cadena s2 en la cadena s1.
2	**strcat (s1, s2);** Concatena la cadena s2 en el extremo de la cadena s1.
3	**strlen (s1);** Devuelve la longitud de la cadena s1.

4 4	**strcmp (s1, s2);** Devuelve 0 si s1 y s2 son iguales; menor que 0 si s1 <s2; mayor que 0 si s1> s2.
5 5	**strchr (s1, ch);** Devuelve un puntero a la primera aparición del carácter ch en la cadena s1.
6 6	**strstr (s1, s2);** Devuelve un puntero a la primera aparición de la cadena s2 en la cadena s1.

El siguiente ejemplo hace uso de algunas de las funciones mencionadas anteriormente:

```
#include <iostream> #include <cstring>


usando el espacio de nombres estándar ;

int main () {

   char str1 [ 10 ] = "Hola" ; char str2 [ 10 ] = "Mundo" ;
char str3 [ 10 ]; int   len ;




   // copia str1 en str3
   strcpy ( str3 , str1 );
   cout << "strcpy (str3, str1):" << str3 << endl ;

   // concatena str1 y str2
   strcat ( str1 , str2 );
   cout << "strcat (str1, str2):" << str1 << endl ;

   // longitud total de str1 después de la concatenación
   len = strlen ( str1 );
   cout << "strlen (str1):" << len << endl ;

   devuelve 0 ; }
```

Cuando el código anterior se compila y ejecuta, produce el resultado de la siguiente manera:

```
strcpy (str3, str1): Hola
strcat (str1, str2): HelloWorld
strlen (str1): 10
```

La clase de cadena en C ++

La biblioteca estándar de C ++ proporciona un tipo de clase de **cadena** que admite todas las operaciones mencionadas anteriormente, además de mucha más funcionalidad. Veamos el siguiente ejemplo:

```
#include <iostream> #include <string>


usando el espacio de nombres estándar ;

int main () {

   string str1 = "Hola" ; string str2 = "Mundo" ; string str3
; int   len ;




   // copia str1 en str3
   str3 = str1 ;
   cout << "str3:" << str3 << endl ;

   // concatena str1 y
   str2 str3 = str1 + str2 ;
   cout << "str1 + str2:" << str3 << endl ;

   // longitud total de str3 después de la concatenación
   len = str3 . tamaño ();
   cout << "str3.size ():" << len << endl ;

   devuelve 0 ; }
```

Cuando el código anterior se compila y ejecuta, produce el resultado de la siguiente manera:

```
str3: hola
str1 + str2: HelloWorld
str3.size (): 10
```

Punteros C ++

Los punteros C ++ son fáciles y divertidos de aprender. Algunas tareas de C ++ se realizan más fácilmente con punteros, y otras tareas de C ++, como la asignación dinámica de memoria, no se pueden realizar sin ellas.

Como sabe, cada variable es una ubicación de memoria y cada ubicación de memoria tiene su dirección definida, a la que se puede acceder mediante el operador ampersand (&) que denota una dirección en la memoria. Considere lo siguiente que imprimirá la dirección de las variables definidas:

```
#include <iostream>

using namespace std;
```

```
int main () {
   int  var1;
   char var2[10];

   cout << "Address of var1 variable: ";
   cout << &var1 << endl;

   cout << "Address of var2 variable: ";
   cout << &var2 << endl;

   return 0;
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Address of var1 variable: 0xbfebd5c0
Address of var2 variable: 0xbfebd5b6
```

¿Qué son los punteros?

Un **puntero** es una variable cuyo valor es la dirección de otra variable. Como cualquier variable o constante, debe declarar un puntero antes de poder trabajar con él. La forma general de una declaración de variable de puntero es:

```
type *var-name;
```

Aquí, **tipo** es el tipo base del puntero; debe ser un tipo de C ++ válido y **var-name** es el nombre de la variable de puntero. El asterisco que utilizó para declarar un puntero es el mismo que utiliza para la multiplicación. Sin embargo, en esta declaración, el asterisco se utiliza para designar una variable como puntero. Los siguientes son la declaración válida del puntero:

```
int    *ip;    // pointer to an integer
double *dp;    // pointer to a double
float  *fp;    // pointer to a float
char   *ch     // pointer to character
```

El tipo de datos real del valor de todos los punteros, ya sean enteros, flotantes, caracteres u otros, es el mismo, un número hexadecimal largo que representa una dirección de memoria. La única diferencia entre punteros de diferentes tipos de datos es el tipo de datos de la variable o constante a la que apunta el puntero.

Usar punteros en C ++

Hay pocas operaciones importantes, que haremos con los punteros con mucha frecuencia. **(a)** Definimos una variable de puntero. **(b)** Asigne la dirección de una variable a un puntero. **(c)** Finalmente acceda al valor en la dirección disponible en la variable de puntero. Esto se realiza mediante el uso del operador unario * que devuelve el valor de la variable ubicada en la dirección especificada por su operando. El siguiente ejemplo hace uso de estas operaciones:

```
#include <iostream>

using namespace std;

int main () {
   int  var = 20;   // actual variable declaration.
   int  *ip;        // pointer variable

   ip = &var;       // store address of var in pointer
variable

   cout << "Value of var variable: ";
   cout << var << endl;

   // print the address stored in ip pointer variable
   cout << "Address stored in ip variable: ";
   cout << ip << endl;

   // access the value at the address available in pointer
   cout << "Value of *ip variable: ";
   cout << *ip << endl;

   return 0;
}
```

Cuando el código anterior se compila y ejecuta, produce el resultado de la siguiente manera:

```
Value of var variable: 20
Address stored in ip variable: 0xbfc601ac
Value of *ip variable: 20
```

Punteros en C ++

Los punteros tienen muchos pero fáciles conceptos y son muy importantes para la programación en C ++. A continuación se detallan algunos conceptos importantes de puntero que deben quedar claros para un programador de C ++:

No Señor	Concepto y descripción
1	Punteros nulos C ++ admite puntero nulo, que es una constante con un valor de cero definido en varias bibliotecas estándar.
2	Aritmética de puntero Hay cuatro operadores aritméticos que se pueden usar en punteros: ++, -, +, -

3	Punteros vs matrices Existe una estrecha relación entre punteros y matrices.
4 4	Matriz de punteros Puede definir matrices para contener una serie de punteros.
5 5	Puntero a puntero C ++ le permite tener un puntero en un puntero y así sucesivamente.
6 6	Pasando Punteros a Funciones Al pasar un argumento por referencia o por dirección, ambos permiten que el argumento pasado se cambie en la función de llamada por la función llamada.
7 7	Indicador de retorno de funciones C ++ permite que una función devuelva un puntero a la variable local, variable estática y memoria asignada dinámicamente también.

Referencias de C ++

Una variable de referencia es un alias, es decir, otro nombre para una variable ya existente. Una vez que se inicializa una referencia con una variable, puede usarse el nombre de la variable o el nombre de referencia para referirse a la variable.

Referencias vs punteros

Las referencias a menudo se confunden con punteros, pero tres diferencias principales entre referencias y punteros son:

- No puede tener referencias NULL. Siempre debe ser capaz de asumir que una referencia está conectada a una pieza legítima de almacenamiento.
- Una vez que se inicializa una referencia a un objeto, no se puede cambiar para referirse a otro objeto. Los punteros pueden apuntar a otro objeto en cualquier momento.
- Se debe inicializar una referencia cuando se crea. Los punteros se pueden inicializar en cualquier momento.

Crear referencias en C ++

Piense en un nombre de variable como una etiqueta adjunta a la ubicación de la variable en la memoria. Luego puede pensar en una referencia como una segunda etiqueta adjunta a esa ubicación de memoria. Por lo tanto, puede acceder al contenido de la variable a través del nombre de la variable original o la referencia. Por ejemplo, supongamos que tenemos el siguiente ejemplo:

```
int i = 17;
```

Podemos declarar variables de referencia para i de la siguiente manera.

```
int& r = i;
```

Lea el & en estas declaraciones como **referencia** . Por lo tanto, lea la primera declaración como "r es una referencia entera inicializada en i" y lea la segunda declaración como "s es una referencia doble inicializada en d". El siguiente ejemplo hace uso de referencias en int y double -

```
#include <iostream>

using namespace std;

int main () {
   // declare simple variables
   int    i;
   double d;

   // declare reference variables
   int&    r = i;
   double& s = d;

   i = 5;
   cout << "Value of i : " << i << endl;
   cout << "Value of i reference : " << r  << endl;

   d = 11.7;
   cout << "Value of d : " << d << endl;
   cout << "Value of d reference : " << s  << endl;

   return 0;
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Value of i : 5
Value of i reference : 5
Value of d : 11.7
Value of d reference : 11.7
```

Las referencias se usan generalmente para listas de argumentos de funciones y valores de retorno de funciones. Los siguientes son dos temas importantes relacionados con las referencias de C ++ que deben quedar claros para un programador de C ++:

No Señor	Concepto y descripción
1	Referencias como parámetros C ++ admite referencias de paso como parámetro de función de forma más segura que los parámetros.

2	Referencia como valor de retorno Puede devolver la referencia de una función C ++ como cualquier otro tipo de datos.

C ++ Fecha y hora

La biblioteca estándar de C ++ no proporciona un tipo de fecha adecuado. C ++ hereda las estructuras y funciones para la manipulación de fecha y hora de C. Para acceder a las funciones y estructuras relacionadas con la fecha y la hora, deberá incluir el archivo de encabezado <ctime> en su programa C ++.

Hay cuatro tipos relacionados con el tiempo: **clock_t, time_t, size_t** y **tm** . Los tipos clock_t, size_t y time_t son capaces de representar la fecha y hora del sistema como una especie de número entero.

El tipo de estructura **tm** contiene la fecha y la hora en forma de una estructura C que tiene los siguientes elementos:

```
struct tm {
   int tm_sec;   // seconds of minutes from 0 to 61
   int tm_min;   // minutes of hour from 0 to 59
   int tm_hour;  // hours of day from 0 to 24
   int tm_mday;  // day of month from 1 to 31
   int tm_mon;   // month of year from 0 to 11
   int tm_year;  // year since 1900
   int tm_wday;  // days since sunday
   int tm_yday;  // days since January 1st
   int tm_isdst; // hours of daylight savings time
}
```

Las siguientes son las funciones importantes, que usamos al trabajar con fecha y hora en C o C ++. Todas estas funciones son parte de la biblioteca estándar de C y C ++ y puede verificar sus detalles haciendo referencia a la biblioteca estándar de C ++ que se proporciona a continuación.

No Señor	Función y Propósito
1	**time_t time (time_t * time);** Esto devuelve el tiempo calendario actual del sistema en número de segundos transcurridos desde el 1 de enero de 1970. Si el sistema no tiene tiempo, se devuelve .1.
2	**char * ctime (const time_t * time);** Esto devuelve un puntero a una cadena del formulario *día mes año horas: minutos: segundos año \ n \ 0* .
3	**struct tm * localtime (const time_t * time);**

	Esto devuelve un puntero a la estructura **tm** que representa la hora local.
4 4	**clock_t clock (nulo);** Esto devuelve un valor que se aproxima a la cantidad de tiempo que el programa de llamadas ha estado funcionando. Se devuelve un valor de .1 si el tiempo no está disponible.
5 5	**char * asctime (const struct tm * time);** Esto devuelve un puntero a una cadena que contiene la información almacenada en la estructura señalada por el tiempo convertida en la forma: día mes fecha horas: minutos: segundos año \ n \ 0
6 6	**struct tm * gmtime (const time_t * time);** Esto devuelve un puntero al tiempo en forma de una estructura tm. El tiempo se representa en Tiempo Universal Coordinado (UTC), que es esencialmente el Tiempo Medio de Greenwich (GMT).
7 7	**time_t mktime (struct tm * time);** Esto devuelve el equivalente de tiempo calendario del tiempo encontrado en la estructura señalada por el tiempo.
8	**doble difftime (time_t time2, time_t time1);** Esta función calcula la diferencia en segundos entre tiempo1 y tiempo2.
9 9	**size_t strftime ();** Esta función se puede usar para formatear la fecha y la hora en un formato específico.

Fecha y hora actual

Suponga que desea recuperar la fecha y hora actuales del sistema, ya sea como hora local o como hora universal coordinada (UTC). El siguiente es el ejemplo para lograr lo mismo:

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

using namespace std;

int main() {
   // current date/time based on current system
   time_t now = time(0);
```

```
   // convert now to string form
   char* dt = ctime(&now);

   cout << "The local date and time is: " << dt << endl;

   // convert now to tm struct for UTC
   tm *gmtm = gmtime(&now);
   dt = asctime(gmtm);
   cout << "The UTC date and time is:"<< dt << endl;
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
The local date and time is: Sat Jan  8 20:07:41 2011

The UTC date and time is:Sun Jan  9 03:07:41 2011
```

Formatear hora usando struct tm

La estructura **tm** es muy importante mientras se trabaja con fecha y hora en C o C ++. Esta estructura contiene la fecha y la hora en forma de una estructura C como se mencionó anteriormente. La mayoría de las funciones relacionadas hace uso de la estructura tm. El siguiente es un ejemplo que hace uso de varias funciones relacionadas con la fecha y la hora y la estructura tm:

Mientras uso la estructura en este capítulo, supongo que usted tiene una comprensión básica sobre la estructura C y cómo acceder a los miembros de la estructura usando el operador de flecha ->.

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

using namespace std;

int main() {
   // current date/time based on current system
   time_t now = time(0);

   cout << "Number of sec since January 1,1970:" << now <<
endl;

   tm *ltm = localtime(&now);

   // print various components of tm structure.
   cout << "Year" << 1900 + ltm->tm_year << endl;
   cout << "Month: "<< 1 + ltm->tm_mon<< endl;
   cout << "Day: "<<  ltm->tm_mday << endl;
   cout << "Time: "<< 1 + ltm->tm_hour << ":";
   cout << 1 + ltm->tm_min << ":";
   cout << 1 + ltm->tm_sec << endl;
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Number of sec since January 1,1970:1563027637
Year2019
Month: 7
Day: 13
Time: 15:21:38
```

Entrada / salida básica de C ++

Las bibliotecas estándar de C ++ proporcionan un amplio conjunto de capacidades de entrada / salida que veremos en capítulos posteriores. Este capítulo discutirá las operaciones de E / S muy básicas y más comunes requeridas para la programación en C ++.

La E / S de C ++ se produce en secuencias, que son secuencias de bytes. Si los bytes fluyen desde un dispositivo como un teclado, una unidad de disco o una conexión de red, etc. a la memoria principal, esto se llama **operación de entrada** y si los bytes fluyen desde la memoria principal a un dispositivo como una pantalla, una impresora, una unidad de disco , o una conexión de red, etc., esto se llama **operación de salida** .

Archivos de encabezado de biblioteca de E / S

Los siguientes archivos de encabezado son importantes para los programas C ++:

No Señor	Archivo de encabezado y función y descripción
1	**<iostream>** Este archivo define la **cin, cout, cerr** y **clog** objetos, que corresponden a la secuencia de entrada estándar, la corriente estándar de salida, la corriente de error estándar sin buffer y la corriente de error estándar tamponada, respectivamente.
2	**<iomanip>** Este archivo declara servicios útiles para realizar E / S formateadas con los denominados manipuladores de flujo parametrizados, como **setw** y **setprecision** .
3	**<fstream>** Este archivo declara servicios para el procesamiento de archivos controlados por el usuario. Discutiremos al respecto en detalle en el capítulo relacionado con Archivos y secuencias.

El flujo de salida estándar (cout)

El objeto predefinido **cout** es una instancia de la clase **ostream** . Se dice que el objeto cout está "conectado" al dispositivo de salida estándar, que generalmente es la pantalla de visualización. El **cout** se usa junto con el operador de inserción de flujo, que se escribe como <<, que son dos signos menos que los que se muestran en el siguiente ejemplo.

```
#include <iostream>

using namespace std;

int main() {
   char str[] = "Hello C++";

   cout << "Value of str is : " << str << endl;
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Value of str is : Hello C++
```

El compilador de C ++ también determina el tipo de datos de la variable que se generará y selecciona el operador de inserción de flujo apropiado para mostrar el valor. El operador << se sobrecarga a los elementos de datos de salida de los tipos incorporados entero, flotante, doble, cadenas y valores de puntero.

El operador de inserción << puede usarse más de una vez en una sola instrucción como se muestra arriba y **endl** se usa para agregar una nueva línea al final de la línea.

El flujo de entrada estándar (cin)

El objeto predefinido **cin** es una instancia de la clase **istream** . Se dice que el objeto cin está conectado al dispositivo de entrada estándar, que generalmente es el teclado. El **cin** se usa junto con el operador de extracción de flujo, que se escribe como >>, que son dos signos mayores que los que se muestran en el siguiente ejemplo.

```
#include <iostream>

using namespace std;

int main() {
   char name[50];

   cout << "Please enter your name: ";
   cin >> name;
   cout << "Your name is: " << name << endl;

}
```

Cuando el código anterior se compila y ejecuta, le pedirá que ingrese un nombre. Ingrese un valor y luego presione enter para ver el siguiente resultado:

```
Please enter your name: cplusplus
Your name is: cplusplus
```

El compilador de C ++ también determina el tipo de datos del valor ingresado y selecciona el operador de extracción de flujo apropiado para extraer el valor y almacenarlo en las variables dadas.

El operador de extracción de flujo >> puede usarse más de una vez en una sola declaración. Para solicitar más de un dato, puede usar lo siguiente:

```
cin >> name >> age;
```

Esto será equivalente a las siguientes dos declaraciones:

```
cin >> name;
cin >> age;
```

El flujo de error estándar (cerr)

El objeto predefinido **cerr** es una instancia de la clase **ostream** . Se dice que el objeto cerr está conectado al dispositivo de error estándar, que también es una pantalla de visualización, pero el objeto **cerr no** está protegido y cada inserción de flujo en cerr hace que su salida aparezca inmediatamente.

El **cerr** también se usa junto con el operador de inserción de flujo como se muestra en el siguiente ejemplo.

```
#include <iostream>

using namespace std;

int main() {
   char str[] = "Unable to read....";

   cerr << "Error message : " << str << endl;
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Error message : Unable to read....
```

La secuencia de registro estándar (obstrucción)

La **obstrucción de** objetos predefinida es una instancia de la clase **ostream** . Se dice que el objeto de obstrucción está conectado al dispositivo de error estándar, que también es una pantalla de visualización, pero la **obstrucción** del objeto está protegida. Esto significa que cada inserción para obstruir podría hacer que su salida se mantenga en un búfer hasta que el búfer se llene o hasta que el búfer se vacíe.

La **obstrucción** también se usa junto con el operador de inserción de flujo como se muestra en el siguiente ejemplo.

```
#include <iostream>

using namespace std;

int main() {
   char str[] = "Unable to read....";

   clog << "Error message : " << str << endl;
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Error message : Unable to read....
```

No podría ver ninguna diferencia en cout, cerr y clog con estos pequeños ejemplos, pero al escribir y ejecutar grandes programas la diferencia se hace evidente. Por lo tanto, es una buena práctica mostrar los mensajes de error utilizando cerr stream y al mostrar otros mensajes de registro, se debe utilizar la obstrucción.

Estructuras de datos C ++

Las matrices C / C ++ le permiten definir variables que combinan varios elementos de datos del mismo tipo, pero la **estructura** es otro tipo de datos definido por el usuario que le permite combinar elementos de datos de diferentes tipos.

Las estructuras se utilizan para representar un registro, supongamos que desea realizar un seguimiento de sus libros en una biblioteca. Es posible que desee realizar un seguimiento de los siguientes atributos sobre cada libro:

- Título
- Autor
- Tema
- ID del libro

Definiendo una Estructura

Para definir una estructura, debe usar la instrucción struct. La instrucción struct define un nuevo tipo de datos, con más de un miembro, para su programa. El formato de la declaración de estructura es este:

```
struct [structure tag] {
   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

La **etiqueta de estructura** es opcional y cada definición de miembro es una definición de variable normal, como int i; o flotador f; o cualquier otra definición de variable válida. Al final de la definición de la estructura, antes del punto y coma final, puede especificar una o más variables de estructura, pero es opcional. Así es como declararías la estructura del libro:

```
struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
} book;
```

Acceso a miembros de la estructura

Para acceder a cualquier miembro de una estructura, utilizamos el **operador de acceso de miembro (.)** . El operador de acceso de miembros se codifica como un período entre el nombre de la variable de estructura y el miembro de estructura al que deseamos acceder. Se podría utilizar **estructura** de palabras clave para definir las variables de tipo de estructura. El siguiente es el ejemplo para explicar el uso de la estructura:

```
#include <iostream>
#include <cstring>

using namespace std;

struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

int main() {
   struct Books Book1;        // Declare Book1 of type Book
   struct Books Book2;        // Declare Book2 of type Book

   // book 1 specification
   strcpy( Book1.title, "Learn C++ Programming");
   strcpy( Book1.author, "Chand Miyan");
   strcpy( Book1.subject, "C++ Programming");
   Book1.book_id = 6495407;

   // book 2 specification
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Yakit Singha");
   strcpy( Book2.subject, "Telecom");
   Book2.book_id = 6495700;

   // Print Book1 info
```

```
   cout << "Book 1 title : " << Book1.title <<endl;
   cout << "Book 1 author : " << Book1.author <<endl;
   cout << "Book 1 subject : " << Book1.subject <<endl;
   cout << "Book 1 id : " << Book1.book_id <<endl;

   // Print Book2 info
   cout << "Book 2 title : " << Book2.title <<endl;
   cout << "Book 2 author : " << Book2.author <<endl;
   cout << "Book 2 subject : " << Book2.subject <<endl;
   cout << "Book 2 id : " << Book2.book_id <<endl;

   return 0;
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Book 1 title : Learn C++ Programming
Book 1 author : Chand Miyan
Book 1 subject : C++ Programming
Book 1 id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Yakit Singha
Book 2 subject : Telecom
Book 2 id : 6495700
```

Estructuras como argumentos de función

Puede pasar una estructura como argumento de función de manera muy similar a como pasa cualquier otra variable o puntero. Accedería a las variables de estructura de la misma manera que accedió en el ejemplo anterior:

```
#include <iostream>
#include <cstring>

using namespace std;
void printBook( struct Books book );

struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

int main() {
   struct Books Book1;        // Declare Book1 of type Book
   struct Books Book2;        // Declare Book2 of type Book

   // book 1 specification
   strcpy( Book1.title, "Learn C++ Programming");
   strcpy( Book1.author, "Chand Miyan");
```

```
   strcpy( Book1.subject, "C++ Programming");
   Book1.book_id = 6495407;

   // book 2 specification
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Yakit Singha");
   strcpy( Book2.subject, "Telecom");
   Book2.book_id = 6495700;

   // Print Book1 info
   printBook( Book1 );

   // Print Book2 info
   printBook( Book2 );

   return 0;
}
void printBook( struct Books book ) {
   cout << "Book title : " << book.title <<endl;
   cout << "Book author : " << book.author <<endl;
   cout << "Book subject : " << book.subject <<endl;
   cout << "Book id : " << book.book_id <<endl;
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Book title : Learn C++ Programming
Book author : Chand Miyan
Book subject : C++ Programming
Book id : 6495407
Book title : Telecom Billing
Book author : Yakit Singha
Book subject : Telecom
Book id : 6495700
```

Punteros a estructuras

Puede definir punteros a estructuras de manera muy similar a como define puntero a cualquier otra variable de la siguiente manera:

```
struct Books *struct_pointer;
```

Ahora, puede almacenar la dirección de una variable de estructura en la variable de puntero definida anteriormente. Para encontrar la dirección de una variable de estructura, coloque el operador & antes del nombre de la estructura de la siguiente manera:

```
struct_pointer = &Book1;
```

Para acceder a los miembros de una estructura utilizando un puntero a esa estructura, debe utilizar el operador -> de la siguiente manera:

```
struct_pointer->title;
```

Permítanos reescribir el ejemplo anterior usando el puntero de estructura, espero que le sea fácil entender el concepto:

```
#include <iostream>
#include <cstring>

using namespace std;
void printBook( struct Books *book );

struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};
int main() {
   struct Books Book1;        // Declare Book1 of type Book
   struct Books Book2;        // Declare Book2 of type Book

   // Book 1 specification
   strcpy( Book1.title, "Learn C++ Programming");
   strcpy( Book1.author, "Chand Miyan");
   strcpy( Book1.subject, "C++ Programming");
   Book1.book_id = 6495407;

   // Book 2 specification
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Yakit Singha");
   strcpy( Book2.subject, "Telecom");
   Book2.book_id = 6495700;

   // Print Book1 info, passing address of structure
   printBook( &Book1 );

   // Print Book1 info, passing address of structure
   printBook( &Book2 );

   return 0;
}

// This function accept pointer to structure as parameter.
void printBook( struct Books *book ) {
   cout << "Book title : " << book->title <<endl;
   cout << "Book author : " << book->author <<endl;
   cout << "Book subject : " << book->subject <<endl;
   cout << "Book id : " << book->book_id <<endl;
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Book title : Learn C++ Programming
Book author : Chand Miyan
Book subject : C++ Programming
```

```
Book id : 6495407
Book title : Telecom Billing
Book author : Yakit Singha
Book subject : Telecom
Book id : 6495700
```

La palabra clave typedef

Hay una manera más fácil de definir estructuras o podría "tipos de alias" que cree. Por ejemplo

```
typedef struct {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
} Books;
```

Ahora, puede usar *Libros* directamente para definir variables de tipo *Libros* sin usar la palabra clave struct. El siguiente es el ejemplo:

```
Books Book1, Book2;
```

Puede usar la palabra clave **typedef** para no estructuras, así como lo siguiente:

```
typedef long int *pint32;

pint32 x, y, z;
```

x, y y z son punteros a entradas largas.

Clases y objetos de C ++

El objetivo principal de la programación en C ++ es agregar orientación de objeto al lenguaje de programación C y las clases son la característica central de C ++ que admite programación orientada a objetos y a menudo se denominan tipos definidos por el usuario.

Una clase se utiliza para especificar la forma de un objeto y combina la representación de datos y los métodos para manipular esos datos en un paquete ordenado. Los datos y funciones dentro de una clase se llaman miembros de la clase.

Definiciones de clase C ++

Cuando define una clase, define un plano para un tipo de datos. En realidad, esto no define ningún dato, pero sí define lo que significa el nombre de la clase, es decir, en qué consistirá un objeto de la clase y qué operaciones se pueden realizar en dicho objeto.

Una definición de clase comienza con la **clase de** palabra clave seguida del nombre de la clase; y el cuerpo de la clase, encerrado por un par de llaves. Una definición de clase debe ir seguida de un punto y coma o una lista

de declaraciones. Por ejemplo, definimos el tipo de datos Box usando la **clase de** palabra clave de **la** siguiente manera:

```
class Box {
   public:
      double length;   // Length of a box
      double breadth;  // Breadth of a box
      double height;   // Height of a box
};
```

La palabra clave **public** determina los atributos de acceso de los miembros de la clase que le sigue. Se puede acceder a un miembro público desde fuera de la clase en cualquier lugar dentro del alcance del objeto de clase. También puede especificar los miembros de una clase como **privados** o **protegidos** que discutiremos en una subsección.

Definir objetos C ++

Una clase proporciona los planos para los objetos, por lo que básicamente se crea un objeto a partir de una clase. Declaramos objetos de una clase con exactamente el mismo tipo de declaración que declaramos variables de tipos básicos. Las siguientes declaraciones declaran dos objetos de la clase Box:

```
Box Box1;          // Declare Box1 of type Box
Box Box2;          // Declare Box2 of type Box
```

Los objetos Box1 y Box2 tendrán su propia copia de los miembros de datos.

Acceso a los miembros de datos

Se puede acceder a los miembros de datos públicos de objetos de una clase utilizando el operador de acceso directo a miembros (.). Probemos con el siguiente ejemplo para aclarar las cosas:

```
#include <iostream>

using namespace std;

class Box {
   public:
      double length;   // Length of a box
      double breadth;  // Breadth of a box
      double height;   // Height of a box
};

int main() {
   Box Box1;         // Declare Box1 of type Box
   Box Box2;         // Declare Box2 of type Box
   double volume = 0.0;     // Store the volume of a box here

   // box 1 specification
   Box1.height = 5.0;
```

```
   Box1.length = 6.0;
   Box1.breadth = 7.0;

   // box 2 specification
   Box2.height = 10.0;
   Box2.length = 12.0;
   Box2.breadth = 13.0;

   // volume of box 1
   volume = Box1.height * Box1.length * Box1.breadth;
   cout << "Volume of Box1 : " << volume <<endl;

   // volume of box 2
   volume = Box2.height * Box2.length * Box2.breadth;
   cout << "Volume of Box2 : " << volume <<endl;
   return 0;
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

Es importante tener en cuenta que no se puede acceder directamente a los miembros privados y protegidos utilizando el operador de acceso directo a miembros (.). Aprenderemos cómo se puede acceder a miembros privados y protegidos.

Clases y objetos en detalle

Hasta ahora, tienes una idea muy básica sobre las clases y objetos de C ++. Hay otros conceptos interesantes relacionados con las clases y objetos de C ++ que discutiremos en varias subsecciones enumeradas a continuación:

No Señor	Concepto y descripción
1	Funciones del miembro de la clase Una función miembro de una clase es una función que tiene su definición o su prototipo dentro de la definición de clase como cualquier otra variable.
2	Modificadores de acceso a clases Un miembro de la clase se puede definir como público, privado o protegido. Por defecto, los miembros serían asumidos como privados.
3	Constructor y Destructor Un constructor de clase es una función especial en una clase que se llama cuando se crea un nuevo objeto de la clase. Un destructor también es una función especial que se llama cuando se elimina el objeto creado.

4 4	Copiar constructor El constructor de copia es un constructor que crea un objeto al inicializarlo con un objeto de la misma clase, que se ha creado previamente.
5 5	Funciones de amigo Una función de **amigo** tiene acceso completo a los miembros privados y protegidos de una clase.
6 6	Funciones en linea Con una función en línea, el compilador intenta expandir el código en el cuerpo de la función en lugar de una llamada a la función.
7 7	este puntero Cada objeto tiene un puntero especial **este** que apunta al objeto en sí mismo.
8	Puntero a clases de C ++ Un puntero a una clase se realiza exactamente de la misma manera que un puntero a una estructura. De hecho, una clase es realmente solo una estructura con funciones.
9 9	Miembros estáticos de una clase Tanto los miembros de datos como los miembros de función de una clase se pueden declarar como estáticos.

Herencia C ++

Uno de los conceptos más importantes en la programación orientada a objetos es el de herencia. La herencia nos permite definir una clase en términos de otra clase, lo que facilita la creación y el mantenimiento de una aplicación. Esto también brinda la oportunidad de reutilizar la funcionalidad del código y el rápido tiempo de implementación.

Al crear una clase, en lugar de escribir miembros de datos y funciones de miembros completamente nuevos, el programador puede designar que la nueva clase herede los miembros de una clase existente. Esta clase existente se denomina clase **base** , y la nueva clase se conoce como la clase **derivada** .

La idea de herencia implementa el **es una** relación. Por ejemplo, el mamífero IS-A animal, el perro IS-A mamífero, por lo tanto, el perro IS-A también es animal y así sucesivamente.

Clases Base y Derivadas

Una clase puede derivarse de más de una clase, lo que significa que puede heredar datos y funciones de múltiples clases base. Para definir una clase derivada, utilizamos una lista de derivación de clase para especificar las

clases base. Una lista de derivación de clase nombra una o más clases base y tiene la forma:

```
class derived-class: access-specifier base-class
```

Donde access-specifier es **público, protegido** o **privado** , y base-class es el nombre de una clase previamente definida. Si no se usa el especificador de acceso, entonces es privado de manera predeterminada.

Considere una **forma de** clase base y su **rectángulo de** clase derivada de la siguiente manera:

```
#include <iostream>

using namespace std;

// Base class
class Shape {
   public:
      void setWidth(int w) {
         width = w;
      }
      void setHeight(int h) {
         height = h;
      }

   protected:
      int width;
      int height;
};

// Derived class
class Rectangle: public Shape {
   public:
      int getArea() {
         return (width * height);
      }
};

int main(void) {
   Rectangle Rect;

   Rect.setWidth(5);
   Rect.setHeight(7);

   // Print the area of the object.
   cout << "Total area: " << Rect.getArea() << endl;

   return 0;
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Total area: 35
```

Control de acceso y herencia

Una clase derivada puede acceder a todos los miembros no privados de su clase base. Por lo tanto, los miembros de la clase base que no deberían ser accesibles para las funciones miembro de las clases derivadas deberían declararse privados en la clase base.

Podemos resumir los diferentes tipos de acceso según: quién puede acceder a ellos de la siguiente manera:

Acceso	público	protegido	privado
Misma clase	si	si	si
Clases derivadas	si	si	No
Clases externas	si	No	No

Una clase derivada hereda todos los métodos de la clase base con las siguientes excepciones:

- Constructores, destructores y constructores de copia de la clase base.
- Operadores sobrecargados de la clase base.
- Las funciones de amigo de la clase base.

Tipo de herencia

Al derivar una clase de una clase base, la clase base puede heredarse a través de una herencia **pública, protegida** o **privada** . El tipo de herencia se especifica mediante el especificador de acceso como se explicó anteriormente.

Apenas usamos herencia **protegida** o **privada** , pero **la** herencia **pública** se usa comúnmente. Al usar diferentes tipos de herencia, se aplican las siguientes reglas:

- **Herencia pública** : al derivar una clase de una clase base **pública** , **los** miembros **públicos** de la clase base se convierten en miembros **públicos** de la clase derivada y **los** miembros **protegidos** de la clase base se convierten en miembros **protegidos** de la clase derivada. Los miembros **privados** de una clase base nunca son accesibles directamente desde una clase derivada, pero se puede acceder a ellos mediante llamadas al **público** y miembros **protegidos** de la clase base.
- **Herencia protegida** : cuando se deriva de una clase base **protegida** , **los** miembros **públicos** y **protegidos** de la clase base se convierten en miembros **protegidos** de la clase derivada.

- **Herencia privada** : cuando se deriva de una clase base **privada** , **los** miembros **públicos** y **protegidos** de la clase base se convierten **en** miembros **privados** de la clase derivada.

Herencia múltiple

Una clase C ++ puede heredar miembros de más de una clase y aquí está la sintaxis extendida:

```
class derived-class: access baseA, access baseB....
```

Cuando el acceso sea **público, protegido** o **privado** y se otorgaría para cada clase base y se separarán por comas como se muestra arriba. Probemos con el siguiente ejemplo:

```
#include <iostream>

using namespace std;

// Base class Shape
class Shape {
   public:
      void setWidth(int w) {
         width = w;
      }
      void setHeight(int h) {
         height = h;
      }

   protected:
      int width;
      int height;
};

// Base class PaintCost
class PaintCost {
   public:
      int getCost(int area) {
         return area * 70;
      }
};

// Derived class
class Rectangle: public Shape, public PaintCost {
   public:
      int getArea() {
         return (width * height);
      }
};

int main(void) {
   Rectangle Rect;
   int area;
```

```
   Rect.setWidth(5);
   Rect.setHeight(7);

   area = Rect.getArea();

   // Print the area of the object.
   cout << "Total area: " << Rect.getArea() << endl;

   // Print the total cost of painting
   cout << "Total paint cost: $" << Rect.getCost(area) <<
endl;

   return 0;
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Total area: 35
Total paint cost: $2450
```

Sobrecarga de C ++ (operador y función)

C ++ le permite especificar más de una definición para el nombre de una **función** o un **operador** en el mismo ámbito, lo que se denomina **sobrecarga de funciones** y **sobrecarga de operadores,** respectivamente.

Una declaración sobrecargada es una declaración que se declara con el mismo nombre que una declaración previamente declarada en el mismo ámbito, excepto que ambas declaraciones tienen diferentes argumentos y obviamente una definición (implementación) diferente.

Cuando llama a una **función** u **operador** sobrecargado , el compilador determina la definición más apropiada para usar, comparando los tipos de argumentos que ha usado para llamar a la función u operador con los tipos de parámetros especificados en las definiciones. El proceso de selección de la función u operador sobrecargado más apropiado se denomina **resolución de sobrecarga** .

Sobrecarga de funciones en C ++

Puede tener varias definiciones para el mismo nombre de función en el mismo ámbito. La definición de la función debe diferir entre sí por los tipos y / o el número de argumentos en la lista de argumentos. No puede sobrecargar las declaraciones de funciones que difieren solo por tipo de retorno.

El siguiente es el ejemplo donde la misma función **print ()** se está utilizando para imprimir diferentes tipos de datos:

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

```
class printData {
   public:
      void print(int i) {
        cout << "Printing int: " << i << endl;
      }
      void print(double  f) {
        cout << "Printing float: " << f << endl;
      }
      void print(char* c) {
        cout << "Printing character: " << c << endl;
      }
};

int main(void) {
   printData pd;

   // Call print to print integer
   pd.print(5);

   // Call print to print float
   pd.print(500.263);

   // Call print to print character
   pd.print("Hello C++");

   return 0;
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Printing int: 5
Printing float: 500.263
Printing character: Hello C++
```

Sobrecarga de operadores en C ++

Puede redefinir o sobrecargar la mayoría de los operadores integrados disponibles en C ++. Por lo tanto, un programador puede usar operadores con tipos definidos por el usuario también.

Los operadores sobrecargados son funciones con nombres especiales: la palabra clave "operador" seguida del símbolo del operador que se está definiendo. Como cualquier otra función, un operador sobrecargado tiene un tipo de retorno y una lista de parámetros.

```
Box operator+(const Box&);
```

declara el operador de suma que se puede usar para **agregar** dos objetos Box y devuelve el objeto Box final. La mayoría de los operadores sobrecargados pueden definirse como funciones comunes que no son miembros o como funciones miembro de clase. En caso de que definamos la función anterior como una función no miembro de una clase, entonces tendríamos que pasar dos argumentos para cada operando de la siguiente manera:

```
Box operator+(const Box&, const Box&);
```

El siguiente es el ejemplo para mostrar el concepto de operador sobre carga utilizando una función miembro. Aquí se pasa un objeto como argumento a cuyas propiedades se accederá utilizando este objeto, se puede acceder al objeto que llamará a este operador utilizando **este** operador como se explica a continuación:

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

class Box {
   public:
      double getVolume(void) {
         return length * breadth * height;
      }
      void setLength( double len ) {
         length = len;
      }
      void setBreadth( double bre ) {
         breadth = bre;
      }
      void setHeight( double hei ) {
         height = hei;
      }

      // Overload + operator to add two Box objects.
      Box operator+(const Box& b) {
         Box box;
         box.length = this->length + b.length;
         box.breadth = this->breadth + b.breadth;
         box.height = this->height + b.height;
         return box;
      }

   private:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box
};

// Main function for the program
int main() {
   Box Box1;                // Declare Box1 of type Box
   Box Box2;                // Declare Box2 of type Box
   Box Box3;                // Declare Box3 of type Box
   double volume = 0.0;     // Store the volume of a box here

   // box 1 specification
   Box1.setLength(6.0);
   Box1.setBreadth(7.0);
   Box1.setHeight(5.0);
```

```
   // box 2 specification
   Box2.setLength(12.0);
   Box2.setBreadth(13.0);
   Box2.setHeight(10.0);

   // volume of box 1
   volume = Box1.getVolume();
   cout << "Volume of Box1 : " << volume <<endl;

   // volume of box 2
   volume = Box2.getVolume();
   cout << "Volume of Box2 : " << volume <<endl;

   // Add two object as follows:
   Box3 = Box1 + Box2;

   // volume of box 3
   volume = Box3.getVolume();
   cout << "Volume of Box3 : " << volume <<endl;

   return 0;
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400
```

Operadores sobrecargables / no sobrecargables

La siguiente es la lista de operadores que se pueden sobrecargar:

+	-	* *	/ /	%	^
Y	El |	~	!	,	=
<	>	<=	> =	++	-
<<	>>	==	! =	&&	||
+ =	- =	/ =	% =	^ =	& =
| =	* =	<< =	>> =	[]	()

->	-> *	nuevo	nuevo []	Eliminar	Eliminar []

La siguiente es la lista de operadores, que no se pueden sobrecargar:

:: ::	. *	.	?:

Ejemplos de sobrecarga del operador

Aquí hay varios ejemplos de sobrecarga de operadores para ayudarlo a comprender el concepto.

No Señor	Operadores y ejemplo
1	Sobrecarga de operadores unarios
2	Sobrecarga de operadores binarios
3	Sobrecarga de operadores relacionales
4 4	Sobrecarga de operadores de entrada / salida
5 5	++ y - Sobrecarga de operadores
6 6	Asignación de sobrecarga de operadores
7 7	Llamada a función () Sobrecarga del operador
8	Suscripción [] Sobrecarga del operador
9 9	Operador de acceso de miembro de clase -> Sobrecarga

Polimorfismo en C ++

La palabra **polimorfismo** significa tener muchas formas. Típicamente, el polimorfismo ocurre cuando hay una jerarquía de clases y están relacionadas por herencia.

El polimorfismo de C ++ significa que una llamada a una función miembro hará que se ejecute una función diferente dependiendo del tipo de objeto que invoca la función.

Considere el siguiente ejemplo donde una clase base ha sido derivada por otras dos clases:

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

class Shape {
   protected:
      int width, height;

   public:
      Shape( int a = 0, int b = 0){
         width = a;
         height = b;
      }
      int area() {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
class Rectangle: public Shape {
   public:
      Rectangle( int a = 0, int b = 0):Shape(a, b) { }

      int area () {
         cout << "Rectangle class area :" <<endl;
         return (width * height);
      }
};

class Triangle: public Shape {
   public:
      Triangle( int a = 0, int b = 0):Shape(a, b) { }

      int area () {
         cout << "Triangle class area :" <<endl;
         return (width * height / 2);
      }
};

// Main function for the program
int main() {
   Shape *shape;
   Rectangle rec(10,7);
   Triangle  tri(10,5);

   // store the address of Rectangle
   shape = &rec;
```

```
   // call rectangle area.
   shape->area();

   // store the address of Triangle
   shape = &tri;

   // call triangle area.
   shape->area();

   return 0;
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Parent class area :
Parent class area :
```

La razón de la salida incorrecta es que el compilador establece una vez la llamada del área de función () como la versión definida en la clase base. Esto se llama **resolución estática** de la llamada a la función, o **enlace estático** : la llamada a la función se repara antes de ejecutar el programa. Esto a veces también se llama **enlace temprano** porque la función area () se establece durante la compilación del programa.

Pero ahora, hagamos una ligera modificación en nuestro programa y precedamos la declaración de área () en la clase Shape con la palabra clave **virtual** para que se vea así:

```
class Shape {
   protected:
      int width, height;

   public:
      Shape( int a = 0, int b = 0) {
         width = a;
         height = b;
      }
      virtual int area() {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
```

Después de esta ligera modificación, cuando se compila y ejecuta el código de ejemplo anterior, produce el siguiente resultado:

```
Rectangle class area
Triangle class area
```

Esta vez, el compilador mira el contenido del puntero en lugar de su tipo. Por lo tanto, dado que las direcciones de los objetos de las clases tri y rec se almacenan en forma *, se llama a la función area () correspondiente.

Como puede ver, cada una de las clases secundarias tiene una implementación separada para el área de función (). Así es como se usa

generalmente el **polimorfismo** . Tiene diferentes clases con una función del mismo nombre, e incluso los mismos parámetros, pero con diferentes implementaciones.

Función virtual

Una función **virtual** es una función en una clase base que se declara usando la palabra clave **virtual** . Definir en una clase base una función virtual, con otra versión en una clase derivada, indica al compilador que no queremos un enlace estático para esta función.

Lo que queremos es que la selección de la función que se llama en cualquier punto dado del programa se base en el tipo de objeto para el que se llama. Este tipo de operación se conoce como **enlace dinámico** o **enlace tardío** .

Funciones virtuales puras

Es posible que desee incluir una función virtual en una clase base para que pueda redefinirse en una clase derivada para adaptarse a los objetos de esa clase, pero que no haya una definición significativa que pueda dar para la función en la clase base .

Podemos cambiar el área de función virtual () en la clase base a lo siguiente:

```
class Shape {
   protected:
      int width, height;

   public:
      Shape(int a = 0, int b = 0) {
         width = a;
         height = b;
      }

      // pure virtual function
      virtual int area() = 0;
};
```

El = 0 le dice al compilador que la función no tiene cuerpo y que la función virtual anterior se llamará **función virtual pura** .

Abstracción de datos en C ++

La abstracción de datos se refiere a proporcionar solo información esencial al mundo exterior y ocultar sus detalles de fondo, es decir, representar la información necesaria en el programa sin presentar los detalles.

La abstracción de datos es una técnica de programación (y diseño) que se basa en la separación de la interfaz y la implementación.

Tomemos un ejemplo de la vida real de un televisor, que puede encender y apagar, cambiar el canal, ajustar el volumen y agregar componentes externos

como altavoces, videograbadoras y reproductores de DVD, PERO no conoce sus detalles internos, que es decir, no sabe cómo recibe las señales por el aire o a través de un cable, cómo las traduce y finalmente las muestra en la pantalla.

Por lo tanto, podemos decir que un televisor separa claramente su implementación interna de su interfaz externa y puede jugar con sus interfaces como el botón de encendido, el cambiador de canales y el control de volumen sin tener ningún conocimiento de sus componentes internos.

En C ++, las clases proporcionan un gran nivel de **abstracción** de **datos** . Proporcionan suficientes métodos públicos al mundo exterior para jugar con la funcionalidad del objeto y manipular los datos del objeto, es decir, el estado sin saber realmente cómo se ha implementado la clase internamente.

Por ejemplo, su programa puede hacer una llamada a la función **sort ()** sin saber qué algoritmo usa realmente la función para ordenar los valores dados. De hecho, la implementación subyacente de la funcionalidad de clasificación podría cambiar entre versiones de la biblioteca, y mientras la interfaz permanezca igual, su llamada a la función seguirá funcionando.

En C ++, usamos **clases** para definir nuestros propios tipos de datos abstractos (ADT). Puede usar el objeto **cout** de la clase **ostream** para transmitir datos a una salida estándar como esta:

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

int main() {
   cout << "Hello C++" <<endl;
   return 0;
}
```

Aquí, no necesita comprender cómo **cout** muestra el texto en la pantalla del usuario. Solo necesita conocer la interfaz pública y la implementación subyacente de 'cout' es libre de cambiar.

Las etiquetas de acceso imponen la abstracción

En C ++, usamos etiquetas de acceso para definir la interfaz abstracta para la clase. Una clase puede contener cero o más etiquetas de acceso:

- Los miembros definidos con una etiqueta pública son accesibles a todas las partes del programa. La vista de abstracción de datos de un tipo está definida por sus miembros públicos.
- Los miembros definidos con una etiqueta privada no son accesibles para el código que usa la clase. Las secciones privadas ocultan la implementación del código que usa el tipo.

No hay restricciones sobre la frecuencia con la que puede aparecer una etiqueta de acceso. Cada etiqueta de acceso especifica el nivel de acceso de las definiciones de miembros siguientes. El nivel de acceso especificado

permanece vigente hasta que se encuentre la siguiente etiqueta de acceso o se vea el corchete derecho de cierre del cuerpo de la clase.

Beneficios de la abstracción de datos

La abstracción de datos ofrece dos ventajas importantes:

- Los componentes internos de la clase están protegidos contra errores inadvertidos a nivel de usuario, que pueden corromper el estado del objeto.
- La implementación de la clase puede evolucionar con el tiempo en respuesta a requisitos cambiantes o informes de errores sin requerir cambios en el código de nivel de usuario.

Al definir miembros de datos solo en la sección privada de la clase, el autor de la clase es libre de realizar cambios en los datos. Si la implementación cambia, solo se debe examinar el código de clase para ver qué efecto puede tener el cambio. Si los datos son públicos, cualquier función que acceda directamente a los miembros de datos de la representación anterior podría estar rota.

Ejemplo de abstracción de datos

Cualquier programa C ++ donde implemente una clase con miembros públicos y privados es un ejemplo de abstracción de datos. Considere el siguiente ejemplo:

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

class Adder {
   public:
      // constructor
      Adder(int i = 0) {
         total = i;
      }

      // interface to outside world
      void addNum(int number) {
         total += number;
      }

      // interface to outside world
      int getTotal() {
         return total;
      };

   private:
      // hidden data from outside world
      int total;
};
```

```
int main() {
   Adder a;

   a.addNum(10);
   a.addNum(20);
   a.addNum(30);

   cout << "Total " << a.getTotal() <<endl;
   return 0;
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Total 60
```

La clase anterior suma números y devuelve la suma. Los miembros públicos: **addNum** y **getTotal** son las interfaces con el mundo exterior y un usuario necesita conocerlos para usar la clase. El **total de** miembros privados es algo que el usuario no necesita saber, pero es necesario para que la clase funcione correctamente.

Estrategia de diseño

La abstracción separa el código en la interfaz y la implementación. Por lo tanto, al diseñar su componente, debe mantener la interfaz independiente de la implementación para que si cambia la implementación subyacente, la interfaz permanezca intacta.

En este caso, cualesquiera que sean los programas que utilicen estas interfaces, no se verán afectados y solo necesitarían una recopilación con la última implementación.

Encapsulación de datos en C ++

Todos los programas de C ++ están compuestos por los siguientes dos elementos fundamentales:

- **Declaraciones de programa (código)** : esta es la parte de un programa que realiza acciones y se denominan funciones.
- **Datos del programa** : los datos son la información del programa que se ve afectada por las funciones del programa.

La encapsulación es un concepto de programación orientada a objetos que une los datos y las funciones que manipulan los datos, y que los mantiene a salvo de la interferencia externa y el mal uso. La encapsulación de datos condujo al importante concepto OOP de **ocultar datos** .

La encapsulación de datos es un mecanismo para agrupar los datos, y las funciones que los utilizan y **la abstracción de datos** es un mecanismo para exponer solo las interfaces y ocultar los detalles de implementación del usuario.

C ++ admite las propiedades de encapsulación y ocultación de datos mediante la creación de tipos definidos por el usuario, llamados **clases** . Ya hemos

estudiado que una clase puede contener miembros **privados, protegidos** y **públicos** . Por defecto, todos los elementos definidos en una clase son privados. Por ejemplo

```
class Box {
   public:
      double getVolume(void) {
         return length * breadth * height;
      }

   private:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box
};
```

Las variables longitud, anchura y altura son **privadas** . Esto significa que solo otros miembros de la clase Box pueden acceder a ellos, y no cualquier otra parte de su programa. Esta es una forma de lograr la encapsulación.

Para hacer que partes de una clase sean **públicas** (es decir, accesibles a otras partes de su programa), debe declararlas después de la palabra clave **pública** . Todas las demás funciones en su programa pueden acceder a todas las variables o funciones definidas después del especificador público.

Hacer que una clase sea amiga de otra expone los detalles de implementación y reduce la encapsulación. Lo ideal es mantener tantos detalles de cada clase ocultos como sea posible de todas las demás clases.

Ejemplo de encapsulación de datos

Cualquier programa C ++ donde implemente una clase con miembros públicos y privados es un ejemplo de encapsulación de datos y abstracción de datos. Considere el siguiente ejemplo:

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

class Adder {
   public:
      // constructor
      Adder(int i = 0) {
         total = i;
      }

      // interface to outside world
      void addNum(int number) {
         total += number;
      }

      // interface to outside world
      int getTotal() {
         return total;
```

```
      };

   private:
      // hidden data from outside world
      int total;
};

int main() {
   Adder a;

   a.addNum(10);
   a.addNum(20);
   a.addNum(30);

   cout << "Total " << a.getTotal() <<endl;
   return 0;
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Total 60
```

La clase anterior suma números y devuelve la suma. Los miembros públicos **addNum** y **getTotal** son las interfaces con el mundo exterior y un usuario necesita conocerlos para usar la clase. El **total de** miembros privados es algo que está oculto del mundo exterior, pero es necesario para que la clase funcione correctamente.

Estrategia de diseño

La mayoría de nosotros hemos aprendido a hacer que los miembros de la clase sean privados por defecto a menos que realmente necesitemos exponerlos. Eso es solo una buena **encapsulación** .

Esto se aplica con mayor frecuencia a los miembros de datos, pero se aplica por igual a todos los miembros, incluidas las funciones virtuales.

Interfaces en C ++ (clases abstractas)

Una interfaz describe el comportamiento o las capacidades de una clase C ++ sin comprometerse con una implementación particular de esa clase.

Las interfaces C ++ se implementan utilizando **clases abstractas** y estas clases abstractas no deben confundirse con la abstracción de datos, que es un concepto de mantener los detalles de implementación separados de los datos asociados.

Una clase se hace abstracta declarando al menos una de sus funciones como función **virtual pura** . Una función virtual pura se especifica colocando "= 0" en su declaración de la siguiente manera:

```
class Box {
   public:
      // pure virtual function
```

```
      virtual double getVolume() = 0;

   private:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box
};
```

El propósito de una **clase abstracta** (a menudo denominada ABC) es proporcionar una clase base apropiada de la que otras clases puedan heredar. Las clases abstractas no se pueden usar para crear instancias de objetos y solo sirven como **interfaz** . Intentar instanciar un objeto de una clase abstracta provoca un error de compilación.

Por lo tanto, si una subclase de un ABC necesita ser instanciada, tiene que implementar cada una de las funciones virtuales, lo que significa que es compatible con la interfaz declarada por el ABC. No anular una función virtual pura en una clase derivada, y luego intentar instanciar objetos de esa clase, es un error de compilación.

Las clases que se pueden usar para crear instancias de objetos se denominan **clases concretas** .

Ejemplo de clase abstracta

Considere el siguiente ejemplo donde la clase padre proporciona una interfaz a la clase base para implementar una función llamada **getArea ()** -

```
#include <iostream>

using namespace std;

// Base class
class Shape {
   public:
      // pure virtual function providing interface framework.
      virtual int getArea() = 0;
      void setWidth(int w) {
         width = w;
      }

      void setHeight(int h) {
         height = h;
      }

   protected:
      int width;
      int height;
};

// Derived classes
class Rectangle: public Shape {
```

```
   public:
      int getArea() {
         return (width * height);
      }
};

class Triangle: public Shape {
   public:
      int getArea() {
         return (width * height)/2;
      }
};

int main(void) {
   Rectangle Rect;
   Triangle  Tri;

   Rect.setWidth(5);
   Rect.setHeight(7);

   // Print the area of the object.
   cout << "Total Rectangle area: " << Rect.getArea() <<
endl;

   Tri.setWidth(5);
   Tri.setHeight(7);

   // Print the area of the object.
   cout << "Total Triangle area: " << Tri.getArea() << endl;

   return 0;
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Total Rectangle area: 35
Total Triangle area: 17
```

Puede ver cómo una clase abstracta definió una interfaz en términos de getArea () y otras dos clases implementaron la misma función pero con un algoritmo diferente para calcular el área específica de la forma.

Estrategia de diseño

Un sistema orientado a objetos podría usar una clase base abstracta para proporcionar una interfaz común y estandarizada apropiada para todas las aplicaciones externas. Luego, a través de la herencia de esa clase base abstracta, se forman clases derivadas que funcionan de manera similar.

Las capacidades (es decir, las funciones públicas) que ofrecen las aplicaciones externas se proporcionan como funciones virtuales puras en la clase base abstracta. Las implementaciones de estas funciones virtuales

puras se proporcionan en las clases derivadas que corresponden a los tipos específicos de la aplicación.

Esta arquitectura también permite agregar nuevas aplicaciones a un sistema fácilmente, incluso después de que el sistema se haya definido.

Archivos y secuencias de C ++

Hasta ahora, hemos estado utilizando la biblioteca estándar **iostream** , que proporciona métodos **cin** y **cout** para leer desde la entrada estándar y escribir en la salida estándar, respectivamente.

Este tutorial le enseñará cómo leer y escribir desde un archivo. Esto requiere otra biblioteca estándar de C ++ llamada **fstream** , que define tres nuevos tipos de datos:

No Señor	Tipo de datos y descripción
1	**ofstream** Este tipo de datos representa la secuencia del archivo de salida y se utiliza para crear archivos y escribir información en los archivos.
2	**ifstream** Este tipo de datos representa la secuencia del archivo de entrada y se usa para leer información de los archivos.
3	**fstream** Este tipo de datos representa el flujo de archivos en general, y tiene las capacidades tanto de flujo de flujo como de flujo de flujo continuo, lo que significa que puede crear archivos, escribir información en archivos y leer información de archivos.

Para realizar el procesamiento de archivos en C ++, los archivos de encabezado <iostream> y <fstream> deben incluirse en su archivo fuente C ++.

Abrir un archivo

Se debe abrir un archivo antes de que pueda leerlo o escribir en él. Se puede **usar** cualquier objeto **ofstream** o **fstream** para abrir un archivo para escritura. Y el objeto ifstream se usa para abrir un archivo solo con fines de lectura.

La siguiente es la sintaxis estándar para la función open (), que es miembro de los objetos fstream, ifstream y ofstream.

```
void open(const char *filename, ios::openmode mode);
```

Aquí, el primer argumento especifica el nombre y la ubicación del archivo que se abrirá y el segundo argumento de la función miembro **open ()** define el modo en que se debe abrir el archivo.

No Señor	Indicador de modo y descripción
1	**ios :: aplicación** Modo anexar. Todos los resultados de ese archivo se agregarán al final.
2	**ios :: comió** Abra un archivo para salida y mueva el control de lectura / escritura al final del archivo.
3	**ios :: en** Abre un archivo para leer.
4 4	**ios :: fuera** Abre un archivo para escribir.
5 5	**ios :: trunc** Si el archivo ya existe, su contenido se truncará antes de abrir el archivo.

Puede combinar dos o más de estos valores **O** combinándolos. Por ejemplo, si desea abrir un archivo en modo de escritura y quiere truncarlo en caso de que ya exista, la siguiente será la sintaxis:

```
ofstream outfile;
outfile.open("file.dat", ios::out | ios::trunc );
```

De manera similar, puede abrir un archivo con fines de lectura y escritura de la siguiente manera:

```
fstream  afile;
afile.open("file.dat", ios::out | ios::in );
```

Cerrar un archivo

Cuando un programa C ++ finaliza, vacía automáticamente todas las secuencias, libera toda la memoria asignada y cierra todos los archivos abiertos. Pero siempre es una buena práctica que un programador cierre todos los archivos abiertos antes de la finalización del programa.

La siguiente es la sintaxis estándar para la función close (), que es miembro de los objetos fstream, ifstream y ofstream.

```
void close();
```

Escribir en un archivo

Mientras realiza la programación en C ++, escribe información en un archivo desde su programa usando el operador de inserción de flujo (<<) tal como lo usa para enviar información a la pantalla. La única diferencia es que usa un objeto **ofstream** o **fstream en** lugar del objeto **cout** .

Leer desde un archivo

Usted lee información de un archivo en su programa usando el operador de extracción de flujo (>>) tal como lo usa para ingresar información desde el teclado. La única diferencia es que usa un objeto **ifstream** o **fstream en** lugar del objeto **cin** .

Ejemplo de lectura y escritura

El siguiente es el programa C ++ que abre un archivo en modo lectura y escritura. Después de escribir la información ingresada por el usuario en un archivo llamado afile.dat, el programa lee la información del archivo y la muestra en la pantalla.

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

int main () {
   char data[100];

   // open a file in write mode.
   ofstream outfile;
   outfile.open("afile.dat");

   cout << "Writing to the file" << endl;
   cout << "Enter your name: ";
   cin.getline(data, 100);

   // write inputted data into the file.
   outfile << data << endl;

   cout << "Enter your age: ";
   cin >> data;
   cin.ignore();

   // again write inputted data into the file.
   outfile << data << endl;

   // close the opened file.
   outfile.close();
```

```
   // open a file in read mode.
   ifstream infile;
   infile.open("afile.dat");

   cout << "Reading from the file" << endl;
   infile >> data;

   // write the data at the screen.
   cout << data << endl;

   // again read the data from the file and display it.
   infile >> data;
   cout << data << endl;

   // close the opened file.
   infile.close();

   return 0;
}
```

Cuando el código anterior se compila y ejecuta, produce la siguiente entrada y salida de muestra:

```
$./a.out
Writing to the file
Enter your name: Zara
Enter your age: 9
Reading from the file
Zara
9
```

Los ejemplos anteriores hacen uso de funciones adicionales del objeto cin, como la función getline () para leer la línea desde afuera e ignorar () para ignorar los caracteres adicionales que dejó la instrucción de lectura anterior.

Punteros de posición de archivo

Tanto **istream** y **ostream** proporcionan funciones miembro para reposicionar el puntero de posición del fichero. Estas funciones miembro son **seekg** ("seek get") para istream y **seekp** ("seek put") para ostream.

El argumento para seekg y seekp normalmente es un entero largo. Se puede especificar un segundo argumento para indicar la dirección de búsqueda. La dirección de búsqueda puede ser **ios :: beg** (el valor predeterminado) para el posicionamiento relativo al comienzo de un flujo, **ios :: cur** para el posicionamiento relativo a la posición actual en un flujo o **ios :: end** para el posicionamiento relativo al final de un corriente.

El puntero de posición del archivo es un valor entero que especifica la ubicación en el archivo como un número de bytes desde la ubicación de inicio del archivo. Algunos ejemplos de posicionamiento del puntero de posición de archivo "get" son:

```
// position to the nth byte of fileObject (assumes ios::beg)
fileObject.seekg( n );

// position n bytes forward in fileObject
fileObject.seekg( n, ios::cur );

// position n bytes back from end of fileObject
fileObject.seekg( n, ios::end );

// position at end of fileObject
fileObject.seekg( 0, ios::end );
```

Manejo de excepciones de C ++

Una excepción es un problema que surge durante la ejecución de un programa. Una excepción de C ++ es una respuesta a una circunstancia excepcional que surge mientras se ejecuta un programa, como un intento de dividir por cero.

Las excepciones proporcionan una forma de transferir el control de una parte de un programa a otra. El manejo de excepciones de C ++ se basa en tres palabras clave: **try, catch** y **throw** .

- **throw** : un programa genera una excepción cuando aparece un problema. Esto se hace usando una palabra clave **throw** .
- **catch** : un programa detecta una excepción con un controlador de excepciones en el lugar del programa donde desea manejar el problema. La palabra clave **catch** indica la captura de una excepción.
- **try** : un bloque **try** identifica un bloque de código para el cual se activarán excepciones particulares. Es seguido por uno o más bloques de captura.

Suponiendo que un bloqueo generará una excepción, un método detecta una excepción utilizando una combinación de las palabras clave **try** y **catch** . Se coloca un bloque try / catch alrededor del código que podría generar una excepción. El código dentro de un bloque try / catch se conoce como código protegido, y la sintaxis para usar try / catch de la siguiente manera:

```
try {
   // protected code
} catch( ExceptionName e1 ) {
   // catch block
} catch( ExceptionName e2 ) {
   // catch block
} catch( ExceptionName eN ) {
   // catch block
}
```

Puede enumerar varias declaraciones **catch** para detectar diferentes tipos de excepciones en caso de que su bloque **try** provoque más de una excepción en diferentes situaciones.

Lanzando excepciones

Se pueden generar excepciones en cualquier lugar dentro de un bloque de código utilizando la instrucción **throw** . El operando de la instrucción throw determina un tipo para la excepción y puede ser cualquier expresión y el tipo del resultado de la expresión determina el tipo de excepción lanzada.

El siguiente es un ejemplo de lanzar una excepción cuando se produce una condición de división por cero:

```
double division(int a, int b) {
   if( b == 0 ) {
      throw "Division by zero condition!";
   }
   return (a/b);
}
```

Capturando excepciones

El bloque **catch** que sigue al bloque **try** detecta cualquier excepción. Puede especificar qué tipo de excepción desea capturar y esto está determinado por la declaración de excepción que aparece entre paréntesis después de la palabra clave catch.

```
try {
   // protected code
} catch( ExceptionName e ) {
  // code to handle ExceptionName exception
}
```

El código anterior detectará una excepción del tipo **ExceptionName** . Si desea especificar que un bloque catch debe manejar cualquier tipo de excepción que se lanza en un bloque try, debe colocar puntos suspensivos, ..., entre paréntesis que encierran la declaración de excepción de la siguiente manera:

```
try {
   // protected code
} catch(...) {
  // code to handle any exception
}
```

El siguiente es un ejemplo, que arroja una división por excepción cero y la capturamos en el bloque catch.

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

double division(int a, int b) {
   if( b == 0 ) {
      throw "Division by zero condition!";
   }
   return (a/b);
}

int main () {
   int x = 50;
```

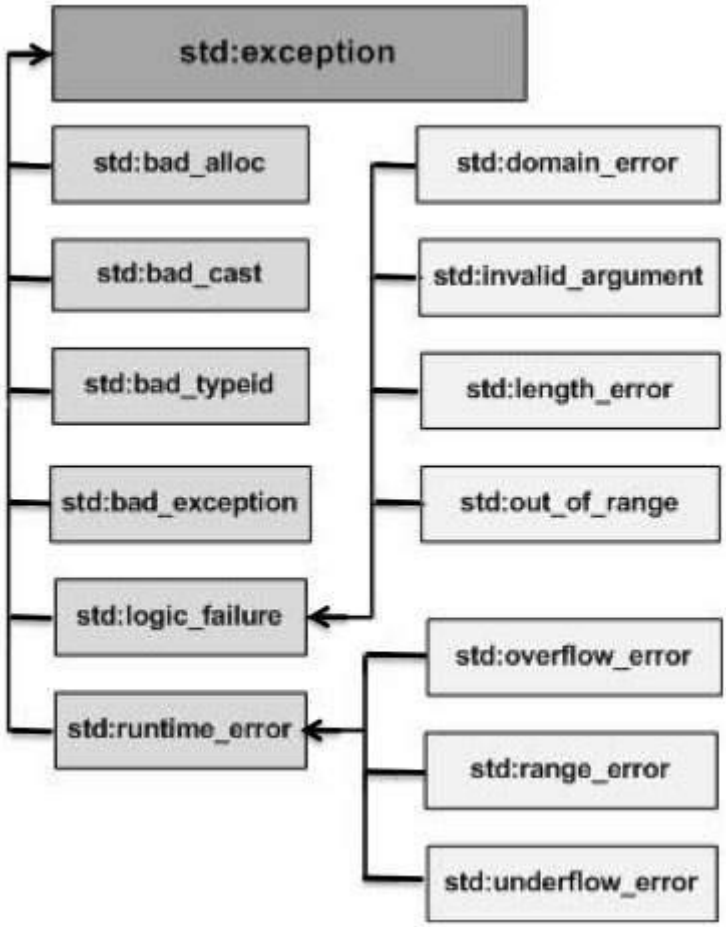
```
   int y = 0;
   double z = 0;

   try {
      z = division(x, y);
      cout << z << endl;
   } catch (const char* msg) {
     cerr << msg << endl;
   }

   return 0;
}
```

Debido a que estamos generando una excepción de tipo **const char *** , entonces, al detectar esta excepción, tenemos que usar const char * en el bloque catch. Si compilamos y ejecutamos el código anterior, esto produciría el siguiente resultado:

```
Division by zero condition!
```

Excepciones estándar de C ++

C ++ proporciona una lista de excepciones estándar definidas en **<excepción>** que podemos usar en nuestros programas. Estos se organizan en una jerarquía de clases padre-hijo que se muestra a continuación:



Aquí está la pequeña descripción de cada excepción mencionada en la jerarquía anterior:

No Señor	Excepción y Descripción
1	**std :: excepción** Una excepción y clase primaria de todas las excepciones estándar de C ++.
2	**std :: bad_alloc** Esto puede ser lanzado por **nuevo** .
3	**std :: bad_cast** Esto puede ser lanzado por **dynamic_cast** .
4 4	**std :: bad_exception** Este es un dispositivo útil para manejar excepciones inesperadas en un programa C ++.

5 5	**std :: bad_typeid** Esto puede ser lanzado por **typeid** .
6 6	**std :: logic_error** Una excepción que teóricamente se puede detectar leyendo el código.
7 7	**std :: domain_error** Esta es una excepción lanzada cuando se usa un dominio matemáticamente inválido.
8	**std :: invalid_argument** Esto se produce debido a argumentos no válidos.
9 9	**std :: length_error** Esto se produce cuando se crea una cadena std :: demasiado grande.
10	**std :: out_of_range** Esto puede ser lanzado por el método 'at', por ejemplo, std :: vector y std :: bitset <> :: operator [] ().
11	**std :: runtime_error** Una excepción que teóricamente no se puede detectar leyendo el código.
12	**std :: overflow_error** Esto se produce si se produce un desbordamiento matemático.
13	**std :: range_error** Esto ocurre cuando intenta almacenar un valor que está fuera de rango.
14	**std :: underflow_error** Esto se produce si se produce un flujo inferior matemático.

Definir nuevas excepciones

Puede definir sus propias excepciones heredando y anulando la funcionalidad de la clase de **excepción** . El siguiente es el ejemplo, que muestra cómo

puede usar la clase std :: exception para implementar su propia excepción de manera estándar:

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

struct MyException : public exception {
   const char * what () const throw () {
      return "C++ Exception";
   }
};

int main() {
   try {
      throw MyException();
   } catch(MyException& e) {
      std::cout << "MyException caught" << std::endl;
      std::cout << e.what() << std::endl;
   } catch(std::exception& e) {
      //Other errors
   }
}
```

Esto produciría el siguiente resultado:

```
MyException caught
C++ Exception
```

Aquí, **what ()** es un método público proporcionado por la clase de excepción y ha sido anulado por todas las clases de excepción secundarias. Esto devuelve la causa de una excepción.

Memoria dinámica de C ++

Una buena comprensión de cómo funciona realmente la memoria dinámica en C ++ es esencial para convertirse en un buen programador de C ++. La memoria en su programa C ++ se divide en dos partes:

- **La pila** : todas las variables declaradas dentro de la función ocuparán memoria de la pila.
- **El montón** : esta es la memoria no utilizada del programa y se puede usar para asignar la memoria dinámicamente cuando se ejecuta el programa.

Muchas veces, no sabe de antemano cuánta memoria necesitará para almacenar información particular en una variable definida y el tamaño de la memoria requerida se puede determinar en tiempo de ejecución.

Puede asignar memoria en tiempo de ejecución dentro del montón para la variable de un tipo dado utilizando un operador especial en C ++ que devuelve la dirección del espacio asignado. Este operador se llama operador **nuevo** .

Si ya no necesita memoria asignada dinámicamente, puede usar el operador de **eliminación** , que desasigna la memoria que fue asignada previamente por el nuevo operador.

Operadores nuevos y eliminados

Existe la siguiente sintaxis genérica para usar un **nuevo** operador para asignar memoria dinámicamente para cualquier tipo de datos.

```
new data-type;
```

Aquí, **el tipo de datos** podría ser cualquier tipo de datos incorporado, incluida una matriz, o cualquier tipo de datos definido por el usuario incluye clase o estructura. Comencemos con los tipos de datos integrados. Por ejemplo, podemos definir un puntero para escribir double y luego solicitar que la memoria se asigne en el momento de la ejecución. Podemos hacer esto usando el **nuevo** operador con las siguientes declaraciones:

```
double* pvalue  = NULL; // Pointer initialized with null
pvalue  = new double;   // Request memory for the variable
```

Es posible que la memoria no se haya asignado correctamente si la tienda gratuita se hubiera agotado. Por lo tanto, es una buena práctica verificar si el nuevo operador está devolviendo el puntero NULL y tomar las medidas apropiadas como se muestra a continuación:

```
double* pvalue  = NULL;
if( !(pvalue  = new double )) {
   cout << "Error: out of memory." <<endl;
   exit(1);
}
```

La función **malloc ()** de C, todavía existe en C ++, pero se recomienda evitar el uso de la función malloc (). La principal ventaja de new sobre malloc () es que new no solo asigna memoria, sino que construye objetos, que es el objetivo principal de C ++.

En cualquier momento, cuando sienta que una variable que ha sido asignada dinámicamente ya no es necesaria, puede liberar la memoria que ocupa en la tienda gratuita con el operador 'eliminar' de la siguiente manera:

```
delete pvalue;        // Release memory pointed to by pvalue
```

Pongamos los conceptos anteriores y formemos el siguiente ejemplo para mostrar cómo funcionan 'nuevo' y 'eliminar':

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

int main () {
   double* pvalue  = NULL; // Pointer initialized with null
   pvalue  = new double;   // Request memory for the variable

   *pvalue = 29494.99;     // Store value at allocated
address
```

```
   cout << "Value of pvalue : " << *pvalue << endl;

   delete pvalue;          // free up the memory.

   return 0;
}
```

Si compilamos y ejecutamos el código anterior, esto produciría el siguiente resultado:

```
Value of pvalue : 29495
```

Asignación de memoria dinámica para matrices

Considere que desea asignar memoria para una matriz de caracteres, es decir, una cadena de 20 caracteres. Usando la misma sintaxis que hemos usado anteriormente, podemos asignar memoria dinámicamente como se muestra a continuación.

```
char* pvalue  = NULL;          // Pointer initialized with
null
pvalue  = new char[20];        // Request memory for the
variable
```

Para eliminar la matriz que acabamos de crear, la declaración se vería así:

```
delete [] pvalue;              // Delete array pointed to by
pvalue
```

Siguiendo la sintaxis genérica similar del nuevo operador, puede asignar una matriz multidimensional de la siguiente manera:

```
double** pvalue  = NULL;       // Pointer initialized with
null
pvalue  = new double [3][4];   // Allocate memory for a 3x4
array
```

Sin embargo, la sintaxis para liberar la memoria para la matriz multidimensional seguirá siendo la misma que la anterior:

```
delete [] pvalue;             // Delete array pointed to by
pvalue
```

Asignación de memoria dinámica para objetos

Los objetos no son diferentes de los tipos de datos simples. Por ejemplo, considere el siguiente código donde vamos a utilizar una matriz de objetos para aclarar el concepto:

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

class Box {
   public:
```

```
      Box() {
         cout << "Constructor called!" <<endl;
      }
      ~Box() {
         cout << "Destructor called!" <<endl;
      }
};
int main() {
   Box* myBoxArray = new Box[4];
   delete [] myBoxArray; // Delete array

   return 0;
}
```

Si tuviera que asignar una matriz de cuatro objetos Box, el constructor Simple se llamaría cuatro veces y, de manera similar, al eliminar estos objetos, el destructor también se llamaría la misma cantidad de veces.

Si compilamos y ejecutamos el código anterior, esto produciría el siguiente resultado:

```
Constructor called!
Constructor called!
Constructor called!
Constructor called!
Destructor called!
Destructor called!
Destructor called!
Destructor called!
```

Espacios de nombres en C ++

Considere una situación, cuando tenemos dos personas con el mismo nombre, Zara, en la misma clase. Siempre que necesitemos diferenciarlos definitivamente tendremos que usar alguna información adicional junto con su nombre, como el área, si viven en un área diferente o el nombre de su madre o padre, etc.

La misma situación puede surgir en sus aplicaciones C ++. Por ejemplo, puede estar escribiendo algún código que tenga una función llamada xyz () y haya otra biblioteca disponible que también tenga la misma función xyz (). Ahora el compilador no tiene forma de saber a qué versión de la función xyz () se refiere dentro de su código.

Un **espacio de nombres** está diseñado para superar esta dificultad y se utiliza como información adicional para diferenciar funciones, clases, variables, etc. similares con el mismo nombre disponible en diferentes bibliotecas. Con el espacio de nombres, puede definir el contexto en el que se definen los nombres. En esencia, un espacio de nombres define un alcance.

Definiendo un espacio de nombres

Una definición de espacio de nombres comienza con el **espacio de nombres de** palabras clave seguido del nombre del espacio de nombres de la siguiente manera:

```
namespace namespace_name {
   // code declarations
}
```

Para llamar a la versión habilitada para el espacio de nombres, ya sea de función o variable, anteponga (: :) el nombre del espacio de nombres de la siguiente manera:

```
name::code;  // code could be variable or function.
```

Veamos cómo el espacio de nombres abarca las entidades, incluidas las variables y las funciones:

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

// first name space
namespace first_space {
   void func() {
      cout << "Inside first_space" << endl;
   }
}

// second name space
namespace second_space {
   void func() {
      cout << "Inside second_space" << endl;
   }
}

int main () {
   // Calls function from first name space.
   first_space::func();

   // Calls function from second name space.
   second_space::func();

   return 0;
}
```

Si compilamos y ejecutamos el código anterior, esto produciría el siguiente resultado:

```
Inside first_space
Inside second_space
```

La directiva de uso

También puede evitar anteponer espacios de nombres con la directiva **using space** . Esta directiva le dice al compilador que el código subsiguiente está

haciendo uso de nombres en el espacio de nombres especificado. El espacio de nombres está implícito para el siguiente código:

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

// first name space
namespace first_space {
   void func() {
      cout << "Inside first_space" << endl;
   }
}

// second name space
namespace second_space {
   void func() {
      cout << "Inside second_space" << endl;
   }
}

using namespace first_space;
int main () {
   // This calls function from first name space.
   func();

   return 0;
}
```

Si compilamos y ejecutamos el código anterior, esto produciría el siguiente resultado:

```
Inside first_space
```

La directiva 'using' también se puede usar para referirse a un elemento en particular dentro de un espacio de nombres. Por ejemplo, si la única parte del espacio de nombres estándar que tiene la intención de usar es cout, puede consultarlo de la siguiente manera:

```
using std::cout;
```

El código posterior puede referirse a cout sin anteponer el espacio de nombres, pero otros elementos en el **espacio de** nombres **estándar** aún deberán ser explícitos de la siguiente manera:

```
#include <iostream>
using std::cout;

int main () {
   cout << "std::endl is used with std!" << std::endl;

   return 0;
}
```

Si compilamos y ejecutamos el código anterior, esto produciría el siguiente resultado:

```
std::endl is used with std!
```

Los nombres introducidos en una directiva de **uso** obedecen las reglas de alcance normal. El nombre es visible desde el punto de la directiva de **uso** hasta el final del ámbito en el que se encuentra la directiva. Las entidades con el mismo nombre definido en un ámbito externo están ocultas.

Espacios de nombres no contiguos

Un espacio de nombres se puede definir en varias partes y, por lo tanto, un espacio de nombres se compone de la suma de sus partes definidas por separado. Las partes separadas de un espacio de nombres se pueden distribuir en varios archivos.

Entonces, si una parte del espacio de nombres requiere un nombre definido en otro archivo, ese nombre aún debe declararse. Escribir una siguiente definición de espacio de nombres define un nuevo espacio de nombres o agrega nuevos elementos a uno existente:

```
namespace namespace_name {
   // code declarations
}
```

Espacios de nombres anidados

Los espacios de nombres se pueden anidar donde puede definir un espacio de nombres dentro de otro espacio de nombres de la siguiente manera:

```
namespace namespace_name1 {
   // code declarations
   namespace namespace_name2 {
      // code declarations
   }
}
```

Puede acceder a los miembros del espacio de nombres anidado utilizando operadores de resolución de la siguiente manera:

```
// to access members of namespace_name2
using namespace namespace_name1::namespace_name2;

// to access members of namespace:name1
using namespace namespace_name1;
```

En las declaraciones anteriores, si está utilizando namespace_name1, hará que los elementos de namespace_name2 estén disponibles en el ámbito de la siguiente manera:

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

```
// first name space
namespace first_space {
   void func() {
      cout << "Inside first_space" << endl;
   }

   // second name space
   namespace second_space {
      void func() {
         cout << "Inside second_space" << endl;
      }
   }
}

using namespace first_space::second_space;
int main () {
   // This calls function from second name space.
   func();

   return 0;
}
```

Si compilamos y ejecutamos el código anterior, esto produciría el siguiente resultado:

```
Inside second_space
```

Plantillas C ++

Las plantillas son la base de la programación genérica, que implica escribir código de forma independiente de cualquier tipo en particular.

Una plantilla es un plan o fórmula para crear una clase genérica o una función. Los contenedores de la biblioteca, como los iteradores y los algoritmos, son ejemplos de programación genérica y se han desarrollado utilizando el concepto de plantilla.

Hay una única definición de cada contenedor, como **vector** , pero podemos definir muchos tipos diferentes de vectores, por ejemplo, **vector <int>** o **vector <string>** .

Puede usar plantillas para definir funciones y clases, veamos cómo funcionan:

Plantilla de funciones

La forma general de una definición de función de plantilla se muestra aquí:

```
template <class type> ret-type func-name(parameter list) {
   // body of function
}
```

Aquí, tipo es un nombre de marcador de posición para un tipo de datos utilizado por la función. Este nombre se puede usar dentro de la definición de la función.

El siguiente es el ejemplo de una plantilla de función que devuelve el máximo de dos valores:

```
#include <iostream>
#include <string>

using namespace std;

template <typename T>
inline T const& Max (T const& a, T const& b) {
   return a < b ? b:a;
}

int main () {
   int i = 39;
   int j = 20;
   cout << "Max(i, j): " << Max(i, j) << endl;

   double f1 = 13.5;
   double f2 = 20.7;
   cout << "Max(f1, f2): " << Max(f1, f2) << endl;

   string s1 = "Hello";
   string s2 = "World";
   cout << "Max(s1, s2): " << Max(s1, s2) << endl;

   return 0;
}
```

Si compilamos y ejecutamos el código anterior, esto produciría el siguiente resultado:

```
Max(i, j): 39
Max(f1, f2): 20.7
Max(s1, s2): World
```

Plantilla de clase

Así como podemos definir plantillas de funciones, también podemos definir plantillas de clase. La forma general de una declaración de clase genérica se muestra aquí:

```
template <class type> class class-name {
   .
   .
   .
}
```

Aquí, **tipo** es el nombre del tipo de marcador de posición, que se especificará cuando se instancia una clase. Puede definir más de un tipo de datos genéricos utilizando una lista separada por comas.

El siguiente es el ejemplo para definir la clase Stack <> e implementar métodos genéricos para empujar y extraer los elementos de la pila:

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <string>
#include <stdexcept>

using namespace std;

template <class T>
class Stack {
   private:
      vector<T> elems;     // elements

   public:
      void push(T const&);  // push element
      void pop();               // pop element
      T top() const;            // return top element

      bool empty() const {      // return true if empty.
         return elems.empty();
      }
};

template <class T>
void Stack<T>::push (T const& elem) {
   // append copy of passed element
   elems.push_back(elem);
}

template <class T>
void Stack<T>::pop () {
   if (elems.empty()) {
      throw out_of_range("Stack<>::pop(): empty stack");
   }

   // remove last element
   elems.pop_back();
}

template <class T>
T Stack<T>::top () const {
   if (elems.empty()) {
      throw out_of_range("Stack<>::top(): empty stack");
   }

   // return copy of last element
   return elems.back();
}
```

```
int main() {
   try {
      Stack<int>         intStack;  // stack of ints
      Stack<string> stringStack;    // stack of strings

      // manipulate int stack
      intStack.push(7);
      cout << intStack.top() <<endl;

      // manipulate string stack
      stringStack.push("hello");
      cout << stringStack.top() << std::endl;
      stringStack.pop();
      stringStack.pop();
   } catch (exception const& ex) {
      cerr << "Exception: " << ex.what() <<endl;
      return -1;
   }
}
```

Si compilamos y ejecutamos el código anterior, esto produciría el siguiente resultado:

```
7
hello
Exception: Stack<>::pop(): empty stack
```

Preprocesador C ++

Los preprocesadores son las directivas, que dan instrucciones al compilador para preprocesar la información antes de que comience la compilación real.

Todas las directivas de preprocesador comienzan con #, y solo los caracteres de espacio en blanco pueden aparecer antes de una directiva de preprocesador en una línea. Las directivas de preprocesador no son sentencias de C ++, por lo que no terminan en punto y coma (;).

Ya has visto una directiva **#include** en todos los ejemplos. Esta macro se usa para incluir un archivo de encabezado en el archivo fuente.

Hay varias directivas de preprocesador compatibles con C ++ como #include, #define, #if, #else, #line, etc. Veamos directivas importantes:

El preprocesador #define

La directiva #define preprocessor crea constantes simbólicas. La constante simbólica se llama **macro** y la forma general de la directiva es -

```
#define macro-name replacement-text
```

Cuando esta línea aparece en un archivo, todas las apariciones posteriores de macro en ese archivo serán reemplazadas por texto de reemplazo antes de compilar el programa. Por ejemplo

```
#include <iostream>
```

```
using namespace std;

#define PI 3.14159

int main () {
   cout << "Value of PI :" << PI << endl;

   return 0;
}
```

Ahora, hagamos el preprocesamiento de este código para ver el resultado asumiendo que tenemos el archivo de código fuente. Así que compílelo con la opción -E y redirija el resultado a test.p. Ahora, si marca test.p, tendrá mucha información y, en la parte inferior, encontrará el valor reemplazado de la siguiente manera:

```
$gcc -E test.cpp > test.p

...
int main () {
   cout << "Value of PI :" << 3.14159 << endl;
   return 0;
}
```

Macros con funciones similares

Puede usar #define para definir una macro que tomará argumentos de la siguiente manera:

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

#define MIN(a,b) (((a)<(b)) ? a : b)

int main () {
   int i, j;

   i = 100;
   j = 30;

   cout <<"The minimum is " << MIN(i, j) << endl;

   return 0;
}
```

Si compilamos y ejecutamos el código anterior, esto produciría el siguiente resultado:

```
The minimum is 30
```

Compilación condicional

Existen varias directivas que se pueden usar para compilar porciones selectivas del código fuente de su programa. Este proceso se llama compilación condicional.

La construcción del preprocesador condicional es muy similar a la estructura de selección 'si'. Considere el siguiente código de preprocesador:

```
#ifndef NULL
   #define NULL 0
#endif
```

Puede compilar un programa para fines de depuración. También puede activar o desactivar la depuración utilizando una sola macro de la siguiente manera:

```
#ifdef DEBUG
   cerr <<"Variable x = " << x << endl;
#endif
```

Esto hace que la instrucción **cerr** se compile en el programa si la constante simbólica DEBUG se ha definido antes de la directiva #ifdef DEBUG. Puede usar la declaración #if 0 para comentar una parte del programa de la siguiente manera:

```
#if 0
   code prevented from compiling
#endif
```

Probemos con el siguiente ejemplo:

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

#define MIN(a,b) (((a)<(b)) ? a : b)

int main () {
   int i, j;

   i = 100;
   j = 30;

#ifdef DEBUG
   cerr <<"Trace: Inside main function" << endl;
#endif

#if 0
   /* This is commented part */
   cout << MKSTR(HELLO C++) << endl;
#endif

   cout <<"The minimum is " << MIN(i, j) << endl;

#ifdef DEBUG
   cerr <<"Trace: Coming out of main function" << endl;
#endif
```

```
   return 0;
}
```

Si compilamos y ejecutamos el código anterior, esto produciría el siguiente resultado:

```
The minimum is 30
Trace: Inside main function
Trace: Coming out of main function
```

Los operadores # y

Los operadores de preprocesador # y ## están disponibles en C ++ y ANSI / ISO C. El operador # hace que un token de texto de reemplazo se convierta en una cadena rodeada de comillas.

Considere la siguiente definición de macro:

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

#define MKSTR( x ) #x

int main () {

   cout << MKSTR(HELLO C++) << endl;

   return 0;
}
```

Si compilamos y ejecutamos el código anterior, esto produciría el siguiente resultado:

```
HELLO C++
```

Veamos cómo funcionó. Es simple entender que el preprocesador de C ++ cambia la línea -

```
cout << MKSTR(HELLO C++) << endl;
```

La línea anterior se convertirá en la siguiente línea:

```
cout << "HELLO C++" << endl;
```

El operador ## se usa para concatenar dos tokens. Aquí hay un ejemplo:

```
#define CONCAT( x, y )  x ## y
```

Cuando CONCAT aparece en el programa, sus argumentos se concatenan y se usan para reemplazar la macro. Por ejemplo, CONCAT (HELLO, C ++) se reemplaza por "HELLO C ++" en el programa de la siguiente manera.

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

```
#define concat(a, b) a ## b
int main() {
   int xy = 100;

   cout << concat(x, y);
   return 0;
}
```

Si compilamos y ejecutamos el código anterior, esto produciría el siguiente resultado:

```
100
```

Veamos cómo funcionó. Es simple entender que el preprocesador C ++ se transforma:

```
cout << concat(x, y);
```

La línea anterior se transformará en la siguiente línea:

```
cout << xy;
```

Macros C ++ predefinidas

C ++ proporciona una serie de macros predefinidas que se mencionan a continuación:

No Señor	Macro y descripción
1	**__LÍNEA__** Contiene el número de línea actual del programa cuando se está compilando.
2	**__ARCHIVO__** Contiene el nombre del archivo actual del programa cuando se está compilando.
3	**__FECHA__** Contiene una cadena del formulario mes / día / año que es la fecha de la traducción del archivo fuente al código objeto.
4 4	**__HORA__** Contiene una cadena de la forma hora: minuto: segundo que es la hora a la que se compiló el programa.

Veamos un ejemplo para todas las macros anteriores:

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

int main () {
   cout << "Value of __LINE__ : " << __LINE__ << endl;
   cout << "Value of __FILE__ : " << __FILE__ << endl;
   cout << "Value of __DATE__ : " << __DATE__ << endl;
   cout << "Value of __TIME__ : " << __TIME__ << endl;

   return 0;
}
```

Si compilamos y ejecutamos el código anterior, esto produciría el siguiente resultado:

```
Value of __LINE__ : 6
Value of __FILE__ : test.cpp
Value of __DATE__ : Feb 28 2011
Value of __TIME__ : 18:52:48
```

Manejo de señal C ++

Las señales son las interrupciones entregadas a un proceso por el sistema operativo que pueden terminar un programa prematuramente. Puede generar interrupciones presionando Ctrl + C en un sistema UNIX, LINUX, Mac OS X o Windows.

Hay señales que el programa no puede detectar, pero hay una siguiente lista de señales que puede detectar en su programa y puede tomar las medidas apropiadas en función de la señal. Estas señales se definen en el archivo de encabezado C ++ <csignal>.

No Señor	Señal y descripción
1	**SIGABRT** Terminación anormal del programa, como una llamada para **abortar** .
2	**SIGFPE** Una operación aritmética errónea, como una división entre cero o una operación que produce un desbordamiento.
3	**SIGILL** Detección de una instrucción ilegal.
4 4	**SIGINT** Recepción de una señal de atención interactiva.

5 5	**SIGSEGV** Un acceso no válido al almacenamiento.
6 6	**SIGTERM** Una solicitud de terminación enviada al programa.

La función de señal ()

La biblioteca de manejo de señal de C ++ proporciona **señal de** función para atrapar eventos inesperados. A continuación se muestra la sintaxis de la función signal ():

```
void (*signal (int sig, void (*func)(int)))(int);
```

Manteniéndolo simple, esta función recibe dos argumentos: el primer argumento como un número entero que representa el número de señal y el segundo argumento como un puntero a la función de manejo de señal.

Escribamos un programa simple de C ++ donde capturaremos la señal SIGINT usando la función signal (). Cualquier señal que desee captar en su programa, debe registrar esa señal utilizando **la** función de **señal** y asociarla con un controlador de señal. Examine el siguiente ejemplo:

```
#include <iostream>
#include <csignal>

using namespace std;

void signalHandler( int signum ) {
   cout << "Interrupt signal (" << signum << ") received.\n";

   // cleanup and close up stuff here
   // terminate program

   exit(signum);
}

int main () {
   // register signal SIGINT and signal handler
   signal(SIGINT, signalHandler);

   while(1) {
      cout << "Going to sleep...." << endl;
      sleep(1);
   }

   return 0;
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
Going to sleep....
Going to sleep....
Going to sleep....
```

Ahora, presione Ctrl + c para interrumpir el programa y verá que su programa captará la señal y saldrá imprimiendo algo de la siguiente manera:

```
Going to sleep....
Going to sleep....
Going to sleep....
Interrupt signal (2) received.
```

La función raise ()

Puede generar señales mediante la función **raise ()** , que toma un número de señal entero como argumento y tiene la siguiente sintaxis.

```
int raise (signal sig);
```

Aquí, **sig** es el número de señal para enviar cualquiera de las señales: SIGINT, SIGABRT, SIGFPE, SIGILL, SIGSEGV, SIGTERM, SIGHUP. El siguiente es el ejemplo donde elevamos una señal internamente usando la función raise () de la siguiente manera:

```
#include <iostream>
#include <csignal>

using namespace std;

void signalHandler( int signum ) {
   cout << "Interrupt signal (" << signum << ") received.\n";

   // cleanup and close up stuff here
   // terminate program

   exit(signum);
}

int main () {
   int i = 0;
   // register signal SIGINT and signal handler
   signal(SIGINT, signalHandler);

   while(++i) {
      cout << "Going to sleep...." << endl;
      if( i == 3 ) {
         raise( SIGINT);
      }
      sleep(1);
   }

   return 0;
}
```

Cuando se compila y ejecuta el código anterior, produce el siguiente resultado y saldrá automáticamente:

```
Going to sleep....
Going to sleep....
Going to sleep....
Interrupt signal (2) received.
```

C ++ Multithreading

Multithreading es una forma especializada de multitarea y una multitarea es la característica que le permite a su computadora ejecutar dos o más programas al mismo tiempo. En general, hay dos tipos de multitarea: basados en procesos y basados en hilos.

La multitarea basada en procesos maneja la ejecución concurrente de programas. La multitarea se basa en subprocesos con la ejecución concurrente de piezas del mismo programa.

Un programa multiproceso contiene dos o más partes que pueden ejecutarse simultáneamente. Cada parte de dicho programa se denomina hilo, y cada hilo define una ruta de ejecución separada.

C ++ no contiene ningún soporte integrado para aplicaciones multiproceso. En cambio, depende completamente del sistema operativo para proporcionar esta función.

Este tutorial asume que está trabajando en el sistema operativo Linux y que vamos a escribir un programa C ++ multiproceso utilizando POSIX. POSIX Threads o Pthreads proporciona API que están disponibles en muchos sistemas POSIX similares a Unix, como FreeBSD, NetBSD, GNU / Linux, Mac OS X y Solaris.

Crear hilos

La siguiente rutina se usa para crear un hilo POSIX:

```
#include <pthread.h>
pthread_create (thread, attr, start_routine, arg)
```

Aquí, **pthread_create** crea un nuevo hilo y lo hace ejecutable. Esta rutina se puede llamar cualquier número de veces desde cualquier lugar dentro de su código. Aquí está la descripción de los parámetros:

No Señor	Descripción de parámetros
1	**hilo** Un identificador opaco y único para el nuevo hilo devuelto por la subrutina.
2	**attr**

	Un objeto de atributo opaco que puede usarse para establecer atributos de subproceso. Puede especificar un objeto de atributos de subproceso o NULL para los valores predeterminados.
3	**rutina_inicial** La rutina de C ++ que el subproceso ejecutará una vez que se haya creado.
4 4	**arg** Un único argumento que se puede pasar a start_routine. Se debe pasar por referencia como un molde de puntero de tipo void. NULL puede usarse si no se va a pasar ningún argumento.

El número máximo de subprocesos que puede crear un proceso depende de la implementación. Una vez creados, los hilos son pares y pueden crear otros hilos. No hay jerarquía implícita o dependencia entre hilos.

Subprocesos de terminación

Hay una siguiente rutina que usamos para terminar un hilo POSIX:

```
#include <pthread.h>
pthread_exit (status)
```

Aquí **pthread_exit** se usa para salir explícitamente de un hilo. Por lo general, la rutina pthread_exit () se llama después de que un subproceso ha completado su trabajo y ya no es necesario que exista.

Si main () termina antes de los hilos que ha creado, y sale con pthread_exit (), los otros hilos continuarán ejecutándose. De lo contrario, se terminarán automáticamente cuando finalice main ().

Ejemplo

Este código de ejemplo simple crea 5 hilos con la rutina pthread_create (). Cada hilo imprime un "¡Hola Mundo!" mensaje, y luego termina con una llamada a pthread_exit ().

```
#include <iostream>
#include <cstdlib>
#include <pthread.h>

using namespace std;

#define NUM_THREADS 5

void *PrintHello(void *threadid) {
   long tid;
   tid = (long)threadid;
   cout << "Hello World! Thread ID, " << tid << endl;
   pthread_exit(NULL);
}
```

```
int main () {
   pthread_t threads[NUM_THREADS];
   int rc;
   int i;

   for( i = 0; i < NUM_THREADS; i++ ) {
      cout << "main() : creating thread, " << i << endl;
      rc = pthread_create(&threads[i], NULL, PrintHello,
(void *)i);

      if (rc) {
         cout << "Error:unable to create thread," << rc <<
endl;
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

Compile el siguiente programa usando la biblioteca -lpthread de la siguiente manera:

```
$gcc test.cpp -lpthread
```

Ahora, ejecute su programa que da el siguiente resultado:

```
main() : creating thread, 0
main() : creating thread, 1
main() : creating thread, 2
main() : creating thread, 3
main() : creating thread, 4
Hello World! Thread ID, 0
Hello World! Thread ID, 1
Hello World! Thread ID, 2
Hello World! Thread ID, 3
Hello World! Thread ID, 4
```

Pasar argumentos a hilos

Este ejemplo muestra cómo pasar múltiples argumentos a través de una estructura. Puede pasar cualquier tipo de datos en una devolución de llamada de subproceso porque apunta a nulo como se explica en el siguiente ejemplo:

```
#include <iostream>
#include <cstdlib>
#include <pthread.h>

using namespace std;

#define NUM_THREADS 5

struct thread_data {
   int  thread_id;
```

```
   char *message;
};

void *PrintHello(void *threadarg) {
   struct thread_data *my_data;
   my_data = (struct thread_data *) threadarg;

   cout << "Thread ID : " << my_data->thread_id ;
   cout << " Message : " << my_data->message << endl;

   pthread_exit(NULL);
}

int main () {
   pthread_t threads[NUM_THREADS];
   struct thread_data td[NUM_THREADS];
   int rc;
   int i;

   for( i = 0; i < NUM_THREADS; i++ ) {
      cout <<"main() : creating thread, " << i << endl;
      td[i].thread_id = i;
      td[i].message = "This is message";
      rc = pthread_create(&threads[i], NULL, PrintHello,
(void *)&td[i]);

      if (rc) {
         cout << "Error:unable to create thread," << rc <<
endl;
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
main() : creating thread, 0
main() : creating thread, 1
main() : creating thread, 2
main() : creating thread, 3
main() : creating thread, 4
Thread ID : 3 Message : This is message
Thread ID : 2 Message : This is message
Thread ID : 0 Message : This is message
Thread ID : 1 Message : This is message
Thread ID : 4 Message : This is message
```

Unir y separar hilos

Existen dos rutinas que podemos usar para unir o separar hilos:

```
pthread_join (threadid, status)
```

`pthread_detach (threadid)`

La subrutina pthread_join () bloquea el hilo de llamada hasta que el hilo 'threadid' especificado finalice. Cuando se crea un hilo, uno de sus atributos define si se puede unir o separar. Solo los hilos que se crean como unibles se pueden unir. Si un hilo se crea como separado, nunca se puede unir.

Este ejemplo muestra cómo esperar a que se complete el subproceso utilizando la rutina de unión Pthread.

```
#include <iostream>
#include <cstdlib>
#include <pthread.h>
#include <unistd.h>

using namespace std;

#define NUM_THREADS 5

void *wait(void *t) {
   int i;
   long tid;

   tid = (long)t;

   sleep(1);
   cout << "Sleeping in thread " << endl;
   cout << "Thread with id : " << tid << "  ...exiting " <<
endl;
   pthread_exit(NULL);
}

int main () {
   int rc;
   int i;
   pthread_t threads[NUM_THREADS];
   pthread_attr_t attr;
   void *status;

   // Initialize and set thread joinable
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr,
PTHREAD_CREATE_JOINABLE);

   for( i = 0; i < NUM_THREADS; i++ ) {
      cout << "main() : creating thread, " << i << endl;
      rc = pthread_create(&threads[i], &attr, wait, (void *)i
);


      if (rc) {
         cout << "Error:unable to create thread," << rc <<
endl;
         exit(-1);
```

```
      }
   }

   // free attribute and wait for the other threads
   pthread_attr_destroy(&attr);
   for( i = 0; i < NUM_THREADS; i++ ) {
      rc = pthread_join(threads[i], &status);
      if (rc) {
         cout << "Error:unable to join," << rc << endl;
         exit(-1);
      }

      cout << "Main: completed thread id :" << i ;
      cout << "  exiting with status :" << status << endl;
   }

   cout << "Main: program exiting." << endl;
   pthread_exit(NULL);
}
```

Cuando el código anterior se compila y ejecuta, produce el siguiente resultado:

```
main() : creating thread, 0
main() : creating thread, 1
main() : creating thread, 2
main() : creating thread, 3
main() : creating thread, 4
Sleeping in thread
Thread with id : 0 .... exiting
Sleeping in thread
Thread with id : 1 .... exiting
Sleeping in thread
Thread with id : 2 .... exiting
Sleeping in thread
Thread with id : 3 .... exiting
Sleeping in thread
Thread with id : 4 .... exiting
Main: completed thread id :0  exiting with status :0
Main: completed thread id :1  exiting with status :0
Main: completed thread id :2  exiting with status :0
Main: completed thread id :3  exiting with status :0
Main: completed thread id :4  exiting with status :0
Main: program exiting.
```

Programación web C ++

¿Qué es el CGI?

- La Common Gateway Interface, o CGI, es un conjunto de estándares que definen cómo se intercambia la información entre el servidor web y un script personalizado.
- NCSA actualmente mantiene las especificaciones de CGI y NCSA define que CGI es como sigue:

- La interfaz de puerta de enlace común, o CGI, es un estándar para que los programas de puerta de enlace externos interactúen con servidores de información como los servidores HTTP.
- La versión actual es CGI / 1.1 y CGI / 1.2 está en progreso.

Buscando en la web

Para comprender el concepto de CGI, veamos qué sucede cuando hacemos clic en un hipervínculo para navegar por una página web o URL en particular.

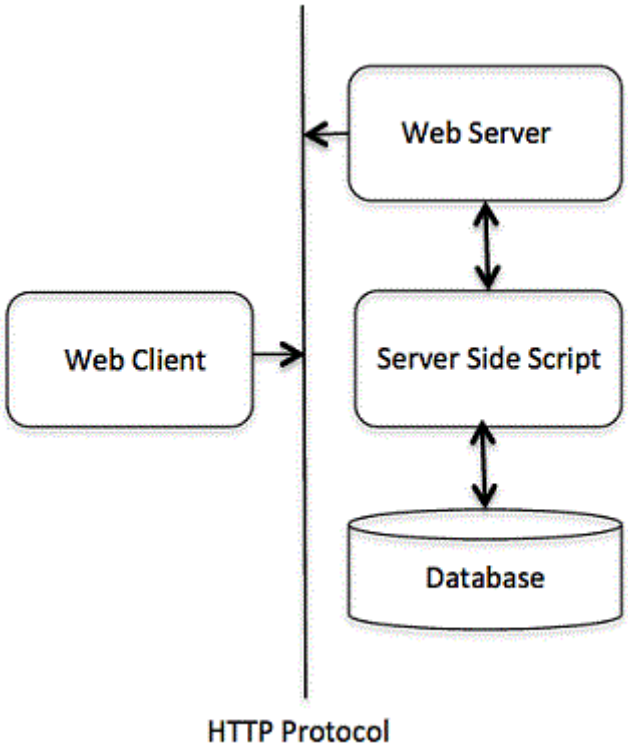
- Su navegador se pone en contacto con el servidor web HTTP y exige la URL, es decir. nombre del archivo.
- Web Server analizará la URL y buscará el nombre del archivo. Si encuentra el archivo solicitado, el servidor web lo devuelve al navegador; de lo contrario, envía un mensaje de error que indica que ha solicitado un archivo incorrecto.
- El navegador web toma la respuesta del servidor web y muestra el archivo recibido o el mensaje de error basado en la respuesta recibida.

Sin embargo, es posible configurar el servidor HTTP de tal manera que siempre que se solicite un archivo en un determinado directorio, ese archivo no se envíe de vuelta; en su lugar, se ejecuta como un programa y la salida producida por el programa se envía de vuelta a su navegador para mostrar.

La Common Gateway Interface (CGI) es un protocolo estándar para permitir que las aplicaciones (llamadas programas CGI o scripts CGI) interactúen con servidores web y con clientes. Estos programas CGI pueden escribirse en Python, PERL, Shell, C o C ++, etc.

Diagrama de arquitectura CGI

El siguiente programa simple muestra una arquitectura simple de CGI:



Configuración del servidor web

Antes de continuar con la Programación CGI, asegúrese de que su Servidor Web sea compatible con CGI y que esté configurado para manejar Programas CGI. Todos los programas CGI que debe ejecutar el servidor HTTP se guardan en un directorio preconfigurado. Este directorio se llama directorio CGI y, por convención, se denomina / var / www / cgi-bin. Por convención, los archivos CGI tendrán una extensión como **.cgi** , aunque son ejecutables en C ++.

De forma predeterminada, el servidor web Apache está configurado para ejecutar programas CGI en / var / www / cgi-bin. Si desea especificar cualquier otro directorio para ejecutar sus scripts CGI, puede modificar la siguiente sección en el archivo httpd.conf:

```
<Directory "/var/www/cgi-bin">
   AllowOverride None
   Options ExecCGI
   Order allow,deny
   Allow from all
</Directory>

<Directory "/var/www/cgi-bin">
   Options All
</Directory>
```

Aquí, supongo que tiene un servidor web funcionando correctamente y que puede ejecutar cualquier otro programa CGI como Perl o Shell, etc.

Primer programa CGI

Considere el siguiente contenido del programa C ++:

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

int main () {
   cout << "Content-type:text/html\r\n\r\n";
   cout << "<html>\n";
   cout << "<head>\n";
   cout << "<title>Hello World - First CGI
Program</title>\n";
   cout << "</head>\n";
   cout << "<body>\n";
   cout << "<h2>Hello World! This is my first CGI
program</h2>\n";
   cout << "</body>\n";
   cout << "</html>\n";

   return 0;
}
```

Compile el código anterior y nombre el ejecutable como cplusplus.cgi. Este archivo se mantiene en el directorio / var / www / cgi-bin y tiene el siguiente contenido. Antes de ejecutar su programa CGI, asegúrese de haber cambiado el modo de archivo usando el comando **chmod 755 cplusplus.cgi** UNIX para hacer que el archivo sea ejecutable.

Mi primer programa CGI

El programa C ++ anterior es un programa simple que está escribiendo su salida en el archivo STDOUT, es decir, en la pantalla. Hay una característica importante y adicional disponible que es la impresión de primera línea **Tipo de contenido: texto / html \ r \ n \ r \ n** . Esta línea se envía de vuelta al navegador y especifica el tipo de contenido que se mostrará en la pantalla del navegador. Ahora debe haber entendido el concepto básico de CGI y puede escribir muchos programas CGI complicados utilizando Python. Un programa C ++ CGI puede interactuar con cualquier otro sistema externo, como RDBMS, para intercambiar información.

Encabezado HTTP

La línea **Content-type: text / html \ r \ n \ r \ n** es una parte del encabezado HTTP, que se envía al navegador para comprender el contenido. Todo el encabezado HTTP estará en la siguiente forma:

```
HTTP Field Name: Field Content

For Example
Content-type: text/html\r\n\r\n
```

Hay algunos otros encabezados HTTP importantes, que utilizará con frecuencia en su programación CGI.

No Señor	Encabezado y descripción
1	**Tipo de contenido:** Una cadena MIME que define el formato del archivo que se devuelve. El ejemplo es Content-type: text / html.
2	**Caduca: fecha** La fecha en que la información deja de ser válida. El navegador debe utilizarlo para decidir cuándo se debe actualizar una página. Una cadena de fecha válida debe tener el formato 01 de enero de 1998 12:00:00 GMT.
3	**Ubicación: URL** La URL que debe devolverse en lugar de la URL solicitada. Puede usar este archivo para redirigir una solicitud a cualquier archivo.
4 4	**Última modificación: fecha** La fecha de la última modificación del recurso.
5 5	**Longitud del contenido: N** La longitud, en bytes, de los datos que se devuelven. El navegador utiliza este valor para informar el tiempo estimado de descarga de un archivo.
6 6	**Set-Cookie: Cadena** Establecer la cookie pasada a través de la *cadena* .

Variables de entorno CGI

Todo el programa CGI tendrá acceso a las siguientes variables de entorno. Estas variables juegan un papel importante al escribir cualquier programa CGI.

No Señor	Nombre y descripción de la variable
1	**TIPO DE CONTENIDO** El tipo de datos del contenido, utilizado cuando el cliente envía contenido adjunto

	al servidor. Por ejemplo, carga de archivos, etc.
2	**LARGANCIA DE CONTENIDO** La longitud de la información de consulta que está disponible solo para solicitudes POST.
3	**HTTP_COOKIE** Devuelve las cookies establecidas en forma de par clave y valor.
4 4	**HTTP_USER_AGENT** El campo de encabezado de solicitud de agente de usuario contiene información sobre el agente de usuario que origina la solicitud. Es un nombre del navegador web.
5 5	**RUTA_INFO** El camino para el script CGI.
6 6	**QUERY_STRING** La información codificada en URL que se envía con la solicitud del método GET.
7 7	**REMOTE_ADDR** La dirección IP del host remoto que realiza la solicitud. Esto puede ser útil para iniciar sesión o para fines de autenticación.
8	**SERVIDOR REMOTO** El nombre completo del host que realiza la solicitud. Si esta información no está disponible, entonces REMOTE_ADDR se puede usar para obtener la dirección IR.
9 9	**SOLICITUD_MÉTODO** El método utilizado para realizar la solicitud. Los métodos más comunes son GET y POST.
10	**SCRIPT_FILENAME** La ruta completa al script CGI.
11	**SCRIPT_NAME** El nombre del script CGI.

12	**NOMBRE DEL SERVIDOR** El nombre de host del servidor o la dirección IP.
13	**SERVER_SOFTWARE** El nombre y la versión del software que ejecuta el servidor.

Aquí hay un pequeño programa CGI para enumerar todas las variables CGI.

```
#include <iostream>
#include <stdlib.h>
using namespace std;

const string ENV[ 24 ] = {
   "COMSPEC", "DOCUMENT_ROOT", "GATEWAY_INTERFACE",
   "HTTP_ACCEPT", "HTTP_ACCEPT_ENCODING",
   "HTTP_ACCEPT_LANGUAGE", "HTTP_CONNECTION",
   "HTTP_HOST", "HTTP_USER_AGENT", "PATH",
   "QUERY_STRING", "REMOTE_ADDR", "REMOTE_PORT",
   "REQUEST_METHOD", "REQUEST_URI", "SCRIPT_FILENAME",
   "SCRIPT_NAME", "SERVER_ADDR", "SERVER_ADMIN",
   "SERVER_NAME","SERVER_PORT","SERVER_PROTOCOL",
   "SERVER_SIGNATURE","SERVER_SOFTWARE" };

int main () {
   cout << "Content-type:text/html\r\n\r\n";
   cout << "<html>\n";
   cout << "<head>\n";
   cout << "<title>CGI Environment Variables</title>\n";
   cout << "</head>\n";
   cout << "<body>\n";
   cout << "<table border = \"0\" cellspacing = \"2\">";

   for ( int i = 0; i < 24; i++ ) {
      cout << "<tr><td>" << ENV[ i ] << "</td><td>";

      // attempt to retrieve value of environment variable
      char *value = getenv( ENV[ i ].c_str() );
      if ( value != 0 ) {
         cout << value;
      } else {
         cout << "Environment variable does not exist.";
      }
      cout << "</td></tr>\n";
   }

   cout << "</table><\n";
   cout << "</body>\n";
   cout << "</html>\n";

   return 0;
```

```
}
```

Biblioteca C ++ CGI

Para ejemplos reales, necesitaría realizar muchas operaciones con su programa CGI. Hay una biblioteca CGI escrita para el programa C ++ que puede descargar desde ftp://ftp.gnu.org/gnu/cgicc/ y siga los pasos para instalar la biblioteca:

```
$tar xzf cgicc-X.X.X.tar.gz
$cd cgicc-X.X.X/
$./configure --prefix=/usr
$make
$make install
```

Puede consultar la documentación relacionada disponible en 'C ++ CGI Lib Documentation .

Métodos GET y POST

Debe haber encontrado muchas situaciones en las que necesita pasar información de su navegador al servidor web y, en última instancia, a su Programa CGI. Con mayor frecuencia, el navegador utiliza dos métodos para pasar esta información al servidor web. Estos métodos son el método GET y el método POST.

Pasando información usando el método GET

El método GET envía la información codificada del usuario adjunta a la solicitud de página. La página y la información codificada están separadas por el? personaje de la siguiente manera:

```
http://www.test.com/cgi-bin/cpp.cgi?key1=value1&key2=value2
```

El método GET es el método predeterminado para pasar información del navegador al servidor web y produce una cadena larga que aparece en el cuadro Ubicación del navegador. Nunca use el método GET si tiene una contraseña u otra información confidencial para pasar al servidor. El método GET tiene una limitación de tamaño y puede pasar hasta 1024 caracteres en una cadena de solicitud.

Cuando se usa el método GET, la información se pasa usando el encabezado http QUERY_STRING y estará accesible en su programa CGI a través de la variable de entorno QUERY_STRING.

Puede pasar información simplemente concatenando pares clave y valor junto con cualquier URL o puede usar etiquetas HTML <FORM> para pasar información usando el método GET.

Ejemplo de URL simple: método de obtención

Aquí hay una URL simple que pasará dos valores al programa hello_get.py usando el método GET.

/cgi-bin/cpp_get.cgi?first_name=ZARA&last_name=ALI

A continuación se muestra un programa para generar el programa CGI **cpp_get.cgi** para manejar la entrada dada por el navegador web. Vamos a usar la biblioteca C ++ CGI que hace que sea muy fácil acceder a la información pasada:

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

#include <cgicc/CgiDefs.h>
#include <cgicc/Cgicc.h>
#include <cgicc/HTTPHTMLHeader.h>
#include <cgicc/HTMLClasses.h>

using namespace std;
using namespace cgicc;

int main () {
   Cgicc formData;

   cout << "Content-type:text/html\r\n\r\n";
   cout << "<html>\n";
   cout << "<head>\n";
   cout << "<title>Using GET and POST Methods</title>\n";
   cout << "</head>\n";
   cout << "<body>\n";

   form_iterator fi = formData.getElement("first_name");
   if( !fi->isEmpty() && fi != (*formData).end()) {
      cout << "First name: " << **fi << endl;
   } else {
      cout << "No text entered for first name" << endl;
   }

   cout << "<br/>\n";
   fi = formData.getElement("last_name");
   if( !fi->isEmpty() &&fi != (*formData).end()) {
      cout << "Last name: " << **fi << endl;
   } else {
      cout << "No text entered for last name" << endl;
   }

   cout << "<br/>\n";
   cout << "</body>\n";
   cout << "</html>\n";

   return 0;
```

```
}
```

Ahora, compile el programa anterior de la siguiente manera:

```
$g++ -o cpp_get.cgi cpp_get.cpp -lcgicc
```

Genere cpp_get.cgi y colóquelo en su directorio CGI e intente acceder usando el siguiente enlace:

/cgi-bin/cpp_get.cgi?first_name=ZARA&last_name=ALI

Esto generaría el siguiente resultado:

```
First name: ZARA
Last name: ALI
```

Ejemplo de FORMA Simple: Método GET

Aquí hay un ejemplo simple que pasa dos valores usando HTML FORM y el botón de enviar. Vamos a utilizar el mismo script CGI cpp_get.cgi para manejar esta entrada.

```
<form action = "/cgi-bin/cpp_get.cgi" method = "get">
   First Name: <input type = "text" name = "first_name">  <br
/>

   Last Name: <input type = "text" name = "last_name" />
   <input type = "submit" value = "Submit" />
</form>
```

Aquí está la salida real de la forma anterior. Ingrese Nombre y Apellido y luego haga clic en el botón Enviar para ver el resultado.

Nombre de pila: Apellido: Enviar

Pasando información usando el método POST

Un método generalmente más confiable de pasar información a un programa CGI es el método POST. Esto empaqueta la información exactamente de la misma manera que los métodos GET, pero en lugar de enviarla como una cadena de texto después de un? en la URL lo envía como un mensaje separado. Este mensaje entra en el script CGI en forma de entrada estándar.

El mismo programa cpp_get.cgi también manejará el método POST. Tomemos el mismo ejemplo que el anterior, que pasa dos valores usando el formulario HTML y el botón de enviar, pero esta vez con el método POST de la siguiente manera:

```
<form action = "/cgi-bin/cpp_get.cgi" method = "post">
   First Name: <input type = "text" name = "first_name"><br
/>
   Last Name: <input type = "text" name = "last_name" />

   <input type = "submit" value = "Submit" />
```

```
</form>
```

Aquí está la salida real de la forma anterior. Ingrese Nombre y Apellido y luego haga clic en el botón Enviar para ver el resultado.

Nombre de pila: Apellido: Enviar

Pasar los datos de la casilla de verificación al programa CGI

Las casillas de verificación se usan cuando se requiere seleccionar más de una opción.

Aquí hay un código HTML de ejemplo para un formulario con dos casillas de verificación:

```
<form action = "/cgi-bin/cpp_checkbox.cgi" method = "POST"
target = "_blank">
   <input type = "checkbox" name = "maths" value = "on" />
Maths
   <input type = "checkbox" name = "physics" value = "on" />
Physics
   <input type = "submit" value = "Select Subject" />
</form>
```

El resultado de este código es la siguiente forma:

Matemáticas Física Seleccione Asunto

A continuación se muestra el programa C ++, que generará el script cpp_checkbox.cgi para manejar la entrada dada por el navegador web a través del botón de casilla de verificación.

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

#include <cgicc/CgiDefs.h>
#include <cgicc/Cgicc.h>
#include <cgicc/HTTPHTMLHeader.h>
#include <cgicc/HTMLClasses.h>

using namespace std;
using namespace cgicc;

int main () {
   Cgicc formData;
   bool maths_flag, physics_flag;

   cout << "Content-type:text/html\r\n\r\n";
```

```
   cout << "<html>\n";
   cout << "<head>\n";
   cout << "<title>Checkbox Data to CGI</title>\n";
   cout << "</head>\n";
   cout << "<body>\n";

   maths_flag = formData.queryCheckbox("maths");
   if( maths_flag ) {
      cout << "Maths Flag: ON " << endl;
   } else {
      cout << "Maths Flag: OFF " << endl;
   }
   cout << "<br/>\n";

   physics_flag = formData.queryCheckbox("physics");
   if( physics_flag ) {
      cout << "Physics Flag: ON " << endl;
   } else {
      cout << "Physics Flag: OFF " << endl;
   }

   cout << "<br/>\n";
   cout << "</body>\n";
   cout << "</html>\n";

   return 0;
}
```

Pasar datos de botón de radio al programa CGI

Los botones de radio se usan cuando solo se requiere seleccionar una opción.

Aquí hay un código HTML de ejemplo para un formulario con dos botones de opción:

```
<form action = "/cgi-bin/cpp_radiobutton.cgi" method = "post"
target = "_blank">
   <input type = "radio" name = "subject" value = "maths"
checked = "checked"/> Maths
   <input type = "radio" name = "subject" value = "physics"
/> Physics
   <input type = "submit" value = "Select Subject" />
</form>
```

El resultado de este código es la siguiente forma:

A continuación se muestra el programa C ++, que generará el script cpp_radiobutton.cgi para manejar la entrada dada por el navegador web a través de botones de radio.

```
#include <iostream>
```

```
#include <vector>
#include <string>
#include <stdio.h>
#include <stdlib.h>

#include <cgicc/CgiDefs.h>
#include <cgicc/Cgicc.h>
#include <cgicc/HTTPHTMLHeader.h>
#include <cgicc/HTMLClasses.h>

using namespace std;
using namespace cgicc;

int main () {
   Cgicc formData;

   cout << "Content-type:text/html\r\n\r\n";
   cout << "<html>\n";
   cout << "<head>\n";
   cout << "<title>Radio Button Data to CGI</title>\n";
   cout << "</head>\n";
   cout << "<body>\n";

   form_iterator fi = formData.getElement("subject");
   if( !fi->isEmpty() && fi != (*formData).end()) {
      cout << "Radio box selected: " << **fi << endl;
   }

   cout << "<br/>\n";
   cout << "</body>\n";
   cout << "</html>\n";

   return 0;
}
```

Pasar datos del área de texto al programa CGI

El elemento TEXTAREA se usa cuando el texto multilínea tiene que pasar al Programa CGI.

Aquí hay un código HTML de ejemplo para un formulario con un cuadro TEXTAREA:

```
<form action = "/cgi-bin/cpp_textarea.cgi" method = "post"
target = "_blank">
   <textarea name = "textcontent" cols = "40" rows = "4">
      Type your text here...
   </textarea>
   <input type = "submit" value = "Submit" />
</form>
```

El resultado de este código es la siguiente forma:

A continuación se muestra el programa C ++, que generará el script cpp_textarea.cgi para manejar la entrada dada por el navegador web a través del área de texto.

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

#include <cgicc/CgiDefs.h>
#include <cgicc/Cgicc.h>
#include <cgicc/HTTPHTMLHeader.h>
#include <cgicc/HTMLClasses.h>

using namespace std;
using namespace cgicc;

int main () {
   Cgicc formData;

   cout << "Content-type:text/html\r\n\r\n";
   cout << "<html>\n";
   cout << "<head>\n";
   cout << "<title>Text Area Data to CGI</title>\n";
   cout << "</head>\n";
   cout << "<body>\n";

   form_iterator fi = formData.getElement("textcontent");
   if( !fi->isEmpty() && fi != (*formData).end()) {
      cout << "Text Content: " << **fi << endl;
   } else {
      cout << "No text entered" << endl;
   }

   cout << "<br/>\n";
   cout << "</body>\n";
   cout << "</html>\n";

   return 0;
}
```

Pasar datos de cuadro desplegable al programa CGI

El cuadro desplegable se utiliza cuando tenemos muchas opciones disponibles, pero solo se seleccionarán una o dos.

Aquí hay un código HTML de ejemplo para un formulario con un cuadro desplegable:

```
<form action = "/cgi-bin/cpp_dropdown.cgi" method = "post"
target = "_blank">
   <select name = "dropdown">
      <option value = "Maths" selected>Maths</option>
      <option value = "Physics">Physics</option>
   </select>

   <input type = "submit" value = "Submit"/>
</form>
```

El resultado de este código es la siguiente forma:

A continuación se muestra el programa C ++, que generará el script cpp_dropdown.cgi para manejar la entrada dada por el navegador web a través del cuadro desplegable.

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

#include <cgicc/CgiDefs.h>
#include <cgicc/Cgicc.h>
#include <cgicc/HTTPHTMLHeader.h>
#include <cgicc/HTMLClasses.h>

using namespace std;
using namespace cgicc;

int main () {
   Cgicc formData;

   cout << "Content-type:text/html\r\n\r\n";
   cout << "<html>\n";
   cout << "<head>\n";
   cout << "<title>Drop Down Box Data to CGI</title>\n";
   cout << "</head>\n";
   cout << "<body>\n";

   form_iterator fi = formData.getElement("dropdown");
   if( !fi->isEmpty() && fi != (*formData).end()) {
      cout << "Value Selected: " << **fi << endl;
   }

   cout << "<br/>\n";
   cout << "</body>\n";
   cout << "</html>\n";
```

```
   return 0;
}
```

Uso de cookies en CGI

El protocolo HTTP es un protocolo sin estado. Pero para un sitio web comercial es necesario mantener la información de la sesión entre diferentes páginas. Por ejemplo, el registro de un usuario finaliza después de completar muchas páginas. Pero cómo mantener la información de la sesión del usuario en todas las páginas web.

En muchas situaciones, el uso de cookies es el método más eficiente para recordar y rastrear preferencias, compras, comisiones y otra información requerida para una mejor experiencia del visitante o estadísticas del sitio.

Cómo funciona

Su servidor envía algunos datos al navegador del visitante en forma de cookie. El navegador puede aceptar la cookie. Si lo hace, se almacena como un registro de texto sin formato en el disco duro del visitante. Ahora, cuando el visitante llega a otra página de su sitio, la cookie está disponible para su recuperación. Una vez recuperado, su servidor sabe / recuerda lo que estaba almacenado.

Las cookies son un registro de datos de texto sin formato de 5 campos de longitud variable:

- **Caduca** : muestra la fecha en que caducará la cookie. Si está en blanco, la cookie caducará cuando el visitante cierre el navegador.
- **Dominio** : muestra el nombre de dominio de su sitio.
- **Ruta** : muestra la ruta al directorio o página web que configura la cookie. Esto puede estar en blanco si desea recuperar la cookie de cualquier directorio o página.
- **Seguro** : si este campo contiene la palabra "seguro", la cookie solo se puede recuperar con un servidor seguro. Si este campo está en blanco, no existe tal restricción.
- **Nombre = Valor** : las cookies se configuran y recuperan en forma de pares de clave y valor.

Configurar cookies

Es muy fácil enviar cookies al navegador. Estas cookies se enviarán junto con el encabezado HTTP antes de que se presente el tipo de contenido. Suponiendo que desea establecer la identificación de usuario y la contraseña como cookies. Por lo tanto, la configuración de cookies se realizará de la siguiente manera

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

```
int main () {
   cout << "Set-Cookie:UserID = XYZ;\r\n";
   cout << "Set-Cookie:Password = XYZ123;\r\n";
   cout << "Set-Cookie:Domain =
www.postparaprogramadores.com;\r\n";
   cout << "Set-Cookie:Path = /perl;\n";
   cout << "Content-type:text/html\r\n\r\n";

   cout << "<html>\n";
   cout << "<head>\n";
   cout << "<title>Cookies in CGI</title>\n";
   cout << "</head>\n";
   cout << "<body>\n";

   cout << "Setting cookies" << endl;

   cout << "<br/>\n";
   cout << "</body>\n";
   cout << "</html>\n";

   return 0;
}
```

A partir de este ejemplo, debe haber entendido cómo configurar las cookies. Utilizamos el encabezado HTTP **Set-Cookie** para establecer cookies.

Aquí, es opcional establecer atributos de cookies como Caduca, Dominio y Ruta. Es notable que las cookies se establezcan antes de enviar la línea mágica **"Tipo de contenido: texto / html \ r \ n \ r \ n** .

Compile el programa anterior para producir setcookies.cgi e intente configurar las cookies utilizando el siguiente enlace. Establecerá cuatro cookies en su computadora:

/cgi-bin/setcookies.cgi

Recuperando Cookies

Es fácil recuperar todas las cookies establecidas. Las cookies se almacenan en la variable de entorno CGI HTTP_COOKIE y tendrán la siguiente forma.

```
key1 = value1; key2 = value2; key3 = value3....
```

Aquí hay un ejemplo de cómo recuperar cookies.

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

#include <cgicc/CgiDefs.h>
#include <cgicc/Cgicc.h>
#include <cgicc/HTTPHTMLHeader.h>
#include <cgicc/HTMLClasses.h>
```

```
using namespace std;
using namespace cgicc;

int main () {
   Cgicc cgi;
   const_cookie_iterator cci;

   cout << "Content-type:text/html\r\n\r\n";
   cout << "<html>\n";
   cout << "<head>\n";
   cout << "<title>Cookies in CGI</title>\n";
   cout << "</head>\n";
   cout << "<body>\n";
   cout << "<table border = \"0\" cellspacing = \"2\">";

   // get environment variables
   const CgiEnvironment& env = cgi.getEnvironment();

   for( cci = env.getCookieList().begin();
   cci != env.getCookieList().end();
   ++cci ) {
      cout << "<tr><td>" << cci->getName() << "</td><td>";
      cout << cci->getValue();
      cout << "</td></tr>\n";
   }

   cout << "</table><\n";
   cout << "<br/>\n";
   cout << "</body>\n";
   cout << "</html>\n";

   return 0;
}
```

Ahora, compile el programa anterior para producir getcookies.cgi e intente obtener una lista de todas las cookies disponibles en su computadora:

/cgi-bin/getcookies.cgi

Esto producirá una lista de las cuatro cookies establecidas en la sección anterior y todas las demás cookies establecidas en su computadora:

```
UserID XYZ
Password XYZ123
Domain www.postparaprogramadores.com
Path /perl
```

Ejemplo de carga de archivos

Para cargar un archivo, el formulario HTML debe tener el atributo enctype establecido en **multipart / form-data** . La etiqueta de entrada con el tipo de archivo creará un botón "Examinar".

```
<html>
   <body>
      <form enctype = "multipart/form-data" action = "/cgi-
bin/cpp_uploadfile.cgi"
         method = "post">
         <p>File: <input type = "file" name = "userfile"
/></p>
         <p><input type = "submit" value = "Upload" /></p>
      </form>
   </body>
</html>
```

El resultado de este código es la siguiente forma:

Archivo:

Upload

Nota : el ejemplo anterior se ha deshabilitado intencionalmente para evitar que las personas carguen archivos en nuestro servidor. Pero puede probar el código anterior con su servidor.

Aquí está el script **cpp_uploadfile.cpp** para manejar la carga de archivos:

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

#include <cgicc/CgiDefs.h>
#include <cgicc/Cgicc.h>
#include <cgicc/HTTPHTMLHeader.h>
#include <cgicc/HTMLClasses.h>

using namespace std;
using namespace cgicc;

int main () {
   Cgicc cgi;

   cout << "Content-type:text/html\r\n\r\n";
   cout << "<html>\n";
   cout << "<head>\n";
   cout << "<title>File Upload in CGI</title>\n";
   cout << "</head>\n";
   cout << "<body>\n";

   // get list of files to be uploaded
   const_file_iterator file = cgi.getFile("userfile");
   if(file != cgi.getFiles().end()) {
      // send data type at cout.
      cout << HTTPContentHeader(file->getDataType());
      // write content at cout.
```

```
        file->writeToStream(cout);
    }
    cout << "<File uploaded successfully>\n";
    cout << "</body>\n";
    cout << "</html>\n";

    return 0;
}
```

El ejemplo anterior es para escribir contenido en **cout** stream pero puede abrir su flujo de archivos y guardar el contenido del archivo cargado en un archivo en la ubicación deseada.

Espero que hayas disfrutado este tutorial. En caso afirmativo, envíenos sus comentarios.