**Enum**

# Enumeration (or enum) in C

* user defined data type in C
* mainly used to assign names to integral constants
* the names make a program easy to read and maintain
* increases comprehensibility
* alternative means for creating symbolic constants
* in practice, enumeration is used to define symbolic constants for a switch statement

# enum declaration

enum <enum\_name>{constant1, constant2, constant3, ....... };

enum flag{constant1, constant2, constant3, ....... };

# enum variable declaration

enum week{Mon, Tue, Wed};

enum day;

// Or

enum week{Mon, Tue, Wed}day;

#include<stdio.h>

enum week{Mon, Tue, Wed, Thur, Fri, Sat, Sun};

int main() {

enum week day;

day = Wed;

printf("%d",day);

return 0;

}

Output:

2

# Interesting facts about initialization of enum:

1. Two enum names can have same value

enum State {Working = 1, Failed = 0, Freezed = 0};

1. If we do not explicitly assign values to enum names, the compiler by default assigns values starting from 0
2. We can assign values to some name in any order

All unassigned names get value as value of previous name plus one

1. The value assigned to enum names must be some integeral constant (+ve or -ve)
2. All enum constants must be unique in their scope

#include<stdio.h>

enum year{Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec}; //2

enum day {sunday = 1, monday, tuesday = 5, wednesday,

thursday = 10,friday, saturday}; //3

enum state {working = 1, failed = 1}; //1

//enum state {working = 1.5, failed};// 4 CE: enumerator value for

'working' is not an integer constant

//enum result {failed, passed};//5 CE: redeclaration of enumerator'failed'

int main() {

int i;

for (i=Jan; i<=Dec; i++)

printf("%d ", i);

printf("\n%d %d %d %d %d %d %d", sunday, monday, tuesday,

wednesday, thursday, friday, saturday);

int ivar = Mar;

enum day evar = 5; **// in C++ error: invalid conversion from**

**'int' to 'day' [-fpermissive]**

printf("\nivar %d", ivar);

printf("\nevar %d", evar);

return 0;

}

0 1 2 3 4 5 6 7 8 9 10 11

1 2 5 6 10 11 12

ivar 2

evar 5

# Enum vs Macro

advantages of using enum over macro

* Enums follow scope rules
* Enum variables are automatically assigned values.

Following is simpler

enum state {Working, Failed, Freezed};

#include <stdio.h>

enum State {WORKING = 0, FAILED, FREEZED};

//enum State currState = 2; **// In C++ CE: invalid conversion from**

**'int' to 'State' [-fpermissive]**

enum State currState = FREEZED;

enum State FindState() {

return currState;

}

int main() {

(FindState() == WORKING)? printf("WORKING\n"): printf("NOT

WORKING\n");

(FindState() == 0)? printf("WORKING"): printf("NOT WORKING");

return 0;

}

Output:

NOT WORKING

NOT WORKING

# Anonymous enum or Enum hack

enums without tag names

enum {circle, trianle, rectangle};

printf("evar1 is %d\n", circle); //0

printf("evar2 is %d", rectangle); //2

# Difference between enum in C and C++

* In C++ <enum\_name> becomes new type names
* C++ does not permit an int value to automatically convert to an enum value
* In C++ enum defined within a structure (or class) is visible local to it while in C it is globally visible

enum shape{circle, trianle, rectangle};

shape evar1 = circle; // in C Compile error: unknown type name 'shape'

enum shape evar2 = rectangle;

enum shape{circle, trianle, rectangle};

enum shape evar1 = circle;

//enum shape evar2 = 2**; // in C++ compile error: invalid conversion from 'int' to 'shape'**

enum shape evar2 = (enum shape)2;

There are two distinct kinds of enumerations in C++:

* unscoped enumeration (declared with the enum) and
* scoped enumeration (declared with the enum class or enum struct).

# Unscoped enumeration

1. enum name { enumerator = constexpr , enumerator = constexpr , ... }
2. enum name : type { enumerator = constexpr , enumerator = constexpr ,

... }

1. enum name : type ;

* Declares an unscoped enumeration type whose underlying type is not fixed

(in this case, the underlying type is an implementation-defined integral type that can represent all enumerator values;

this type is not larger than int unless the value of an enumerator cannot fit in an int or unsigned int.

If the enumerator-list is empty, the underlying type is as if the enumeration had a single enumerator with value 0).

* Declares an unscoped enumeration type whose underlying type is fixed.
* Opaque enum declaration for an unscoped enumeration must specify the underlying type.

Each enumerator becomes a named constant of the enumeration's type (that is, name), visible in the enclosing scope, and can be used whenever constants are required

# Scoped enumerations

1. enum struct|class name { enumerator = constexpr , enumerator = constexpr , ... }
2. enum struct|class name : type { enumerator = constexpr , enumerator = constexpr , ... }
3. enum struct|class name ;
4. enum struct|class name : type ;

* declares a scoped enumeration type whose underlying type is int (the keywords class and struct are exactly equivalent)
* declares a scoped enumeration type whose underlying type is type
* opaque enum declaration for a scoped enumeration whose underlying type is int
* opaque enum declaration for a scoped enumeration whose underlying type is type

Each enumerator becomes a named constant of the enumeration's type (that is, name),

which is contained within the scope of the enumeration, and

can be accessed using scope resolution operator.

**There are no implicit conversions from the values of a scoped enumerator to integral types,**

**although static\_cast may be used to obtain the numeric value of the enumerator.**

The **enum classes ("new enums", "strong enums")** address three problems with traditional C++ enumerations:

1. conventional enums implicitly convert to int, causing errors when someone does not want an enumeration to act as an integer.
2. conventional enums export their enumerators to the surrounding scope, causing name clashes.
3. the underlying type of an enum cannot be specified, causing confusion, compatibility problems, and makes forward declaration impossible.

The new enums are "enum class" because they combine aspects of traditional enumerations (names values) with aspects of classes (scoped members and absense of conversions).

# what else makes enum class a better choice?

* They don't convert implicitly to int.
* They don't pollute the surrounding namespace.
* They can be forward-declared.

enum Alert{ green, yellow, orange, red }; // traditional enum

enum class Color{ red, blue }; // scoped and strongly typed enum

// no export of enumerator names into enclosing scope

// no implicit conversion to int

enum class TrafficLight{ red, yellow, green };

Alert a = 7; // error (as ever in C++)

Color c = 7; // error: no int->Color conversion

int a2 = red; // ok: Alert->int conversion

int a3 = Alert::red; // error in C++98; ok in C++11

int a4 = blue; // error: blue not in scope

int a5 = Color::blue; // error: not Color->int conversion

Color a6 = Color::blue; // ok

int a7 = static\_cast<int>(Color::blue); // ok

#include <iostream>

enum smallenum: int16\_t{a, b, c }; // enum that takes 16 bits

enum color{red, yellow, green = 20, blue };

// red-0, yellow-1, green-20, blue-21

enum class altitude: char{high='h', low='l', }; // C++11 allows the extra comma

enum {d, e, f = e + 2 }; // d 0, e1, f 3

//enumeration types (both scoped and unscoped) can have overloaded operators

std::ostream& operator<<(std::ostream& os, color c)

{

switch(c)

{

case red : os << "red"; break;

case yellow: os << "yellow"; break;

case green : os << "green"; break;

case blue : os << "blue"; break;

default : os.setstate(std::ios\_base::failbit);

}

return os;

}

std::ostream& operator<<(std::ostream& os, altitude al)

{

return os << static\_cast<char>(al);

}

int main()

{

color col = red;

altitude a;

a = altitude::low;

std::cout << "col = " << col << '\n'

<< "a = " << a << '\n'

<< "f = " << f << '\n';

}

Output:

col = red

a = l

f = 3