4.5 — Enumerated types

BY ALEX ON JUNE 19TH, 2007 | LAST MODIFIED BY ALEX ON FEBRUARY 27TH, 2018

C++ contains quite a few built in data types. But these types aren't always sufficient for the kinds of things we want to do. So C++ contains capabilities that allow programmers to create their own data types. These data types are called **user-defined data types**.

Perhaps the simplest user-defined data type is the enumerated type. An **enumerated type** (also called an **enumeration**) is a data type where every possible value is defined as a symbolic constant (called an **enumerator**). Enumerations are defined via the **enum** keyword. Let's look at an example:

```
// Define a new enumeration named Color
enum Color
{
    // Here are the enumerators
    // These define all the possible values this type can hold
    // Each enumerator is separated by a comma, not a semicolon
    COLOR_BLACK,
    COLOR_RED,
    COLOR_BLUE,
    COLOR_GREEN,
    COLOR_WHITE,
    COLOR_CYAN,
    COLOR_YELLOW,
    COLOR_MAGENTA, // see note about trailing comma on the last enumerator below
}; // however the enum itself must end with a semicolon

// Define a few variables of enumerated type Color
Color paint = COLOR_WHITE;
Color house(COLOR_BLUE);
Color apple { COLOR_RED };
```

Defining an enumeration (or any user-defined data type) does not allocate any memory. When a variable of the enumerated type is defined (such as variable paint in the example above), memory is allocated for that variable at that time.

Note that each enumerator is separated by a comma, and the entire enumeration is ended with a semicolon.

Prior to C++11, a trailing comma after the last enumerator (e.g. after COLOR_MAGENTA) is not allowed (though many compilers accepted it anyway). However, starting with C++11, a trailing comma is allowed. Now that C++11 compilers are more prevalent, use of a trailing comma after the last element is generally considered acceptable.

Naming enums

Enum identifiers are often named starting with a capital letter, and the enumerators are often named using all caps. Because enumerators are placed into the same namespace as the enumeration, an enumerator name can't be used in multiple enumerations within the same namespace:

```
enum Color
{
RED,
BLUE, // BLUE is put into the global namespace
GREEN
};

enum Feeling
{
HAPPY,
TIRED,
BLUE // error, BLUE was already used in enum Color in the global namespace
};
```

Consequently, it's common to prefix enumerators with a standard prefix like ANIMAL_ or COLOR_, both to prevent naming conflicts and for code documentation purposes.

Enumerator values

Each enumerator is automatically assigned an integer value based on its position in the enumeration list. By default, the first enumerator is assigned the integer value 0, and each subsequent enumerator has a value one greater than the previous enumerator:

```
enum Color
{
    COLOR_BLACK, // assigned 0
    COLOR_RED, // assigned 1
    COLOR_BLUE, // assigned 2
    COLOR_GREEN, // assigned 3
    COLOR_WHITE, // assigned 4
    COLOR_CYAN, // assigned 5
    COLOR_YELLOW, // assigned 6
    COLOR_MAGENTA // assigned 7
};

Color paint(COLOR_WHITE);
std::cout << paint;
```

The cout statement above prints the value 4.

It is possible to explicitly define the value of enumerator. These integer values can be positive or negative and can share the same value as other enumerators. Any non-defined enumerators are given a value one greater than the previous enumerator.

```
// define a new enum named Animal
enum Animal
{
    ANIMAL_CAT = -3,
    ANIMAL_DOG, // assigned -2
    ANIMAL_PIG, // assigned -1
    ANIMAL_HORSE = 5,
    ANIMAL_GIRAFFE = 5, // shares same value as ANIMAL_HORSE
    ANIMAL_CHICKEN // assigned 6
};
```

Note in this case, ANIMAL_HORSE and ANIMAL_GIRAFFE have been given the same value. When this happens, the enumerations become non-distinct -- essentially, ANIMAL_HORSE and ANIMAL_GIRAFFE are interchangeable. Although C++ allows it, assigning the same value to two enumerators in the same enumeration should generally be avoided.

Best practice: Don't assign specific values to your enumerators.
Rule: Don't assign the same value to two enumerators in the same enumeration unless there's a very good reason.

Enum type evaluation and input/output

Because enumerated values evaluate to integers, they can be assigned to integer variables. This means they can also be output (as integers), since std::cout knows how to output integers.

```
int mypet = ANIMAL_PIG;
std::cout << ANIMAL_HORSE; // evaluates to integer before being passed to std::cout
```

This produces the result:

5

The compiler will *not* implicitly convert an integer to an enumerated value. The following will produce a compiler error:

```
Animal animal = 5; // will cause compiler error
```

However, you can force it to do so via a static_cast:

```
Color color = static_cast<Color>(5); // ugly
```

The compiler also will not let you input an enum using std::cin:

```
enum Color
```

```
{
    COLOR_BLACK, // assigned 0
    COLOR_RED, // assigned 1
    COLOR_BLUE, // assigned 2
    COLOR_GREEN, // assigned 3
    COLOR_WHITE, // assigned 4
    COLOR_CYAN, // assigned 5
    COLOR_YELLOW, // assigned 6
    COLOR_MAGENTA // assigned 7
};

Color color;
std::cin >> color; // will cause compiler error
```

One workaround is to read in an integer, and use a static_cast to force the compiler to put an integer value into an enumerated type:

```
int inputColor;
std::cin >> inputColor;

Color color = static_cast<Color>(inputColor);
```

Each enumerated type is considered a distinct type. Consequently, trying to assign enumerators from one enum type to another enum type will cause a compile error:

```
Animal animal = COLOR_BLUE; // will cause compiler error
```

As with constant variables, enumerated types show up in the debugger, making them more useful than #defined values in this regard.

Printing enumerators

As you saw above, trying to print an enumerated value using std::cout results in the integer value of the enumerator being printed. So how can you print the enumerator itself as text? One way to do so is to write a function and use an if statement:

```
enum Color
{
    COLOR_BLACK, // assigned 0
    COLOR_RED, // assigned 1
    COLOR_BLUE, // assigned 2
    COLOR_GREEN, // assigned 3
    COLOR_WHITE, // assigned 4
    COLOR_CYAN, // assigned 5
    COLOR_YELLOW, // assigned 6
    COLOR_MAGENTA // assigned 7
};

void printColor(Color color)
{
    if (color == COLOR_BLACK)
        std::cout << "Black";
    else if (color == COLOR_RED)
        std::cout << "Red";
    else if (color == COLOR_BLUE)
        std::cout << "Blue";
    else if (color == COLOR_GREEN)
        std::cout << "Green";
    else if (color == COLOR_WHITE)
        std::cout << "White";
    else if (color == COLOR_CYAN)
        std::cout << "Cyan";
    else if (color == COLOR_YELLOW)
        std::cout << "Yellow";
    else if (color == COLOR_MAGENTA)
        std::cout << "Magenta";
    else
        std::cout << "Who knows!";
}
```

Once you've learned to use switch statements, you'll probably want to use those instead of a bunch of if/else statements, as it's a little more readable.

Enum allocation and forward declaration

Enum types are considered part of the integer family of types, and it's up to the compiler to determine how much memory to allocate for an enum variable. The C++ standard says the enum size needs to be large enough to represent all of the enumerator values. Most often, it will make enum variables the same size as a standard int.

Because the compiler needs to know how much memory to allocate for an enumeration, you cannot forward declare enum types. However, there is an easy workaround. Because defining an enumeration does not allocate any memory, if an enumeration is needed in multiple files, it is fine to define the enumeration in a header, and #include that header wherever needed.

What are enumerators useful for?

Enumerated types are incredibly useful for code documentation and readability purposes when you need to represent a specific, predefined set of states.

For example, functions often return integers to the caller to represent error codes when something went wrong inside the function. Typically, small negative numbers are used to represent different possible error codes. For example:

```
int readFileContents()
{
    if (!openFile())
        return -1;
    if (!readFile())
        return -2;
    if (!parseFile())
        return -3;

    return 0; // success
}
```

However, using magic numbers like this isn't very descriptive. An alternative method would be through use of an enumerated type:

```
enum ParseResult
{
    SUCCESS = 0,
    ERROR_OPENING_FILE = -1,
    ERROR_READING_FILE = -2,
    ERROR_PARSING_FILE = -3
};

ParseResult readFileContents()
{
    if (!openFile())
        return ERROR_OPENING_FILE;
    if (!readFile())
        return ERROR_READING_FILE;
    if (!parsefile())
        return ERROR_PARSING_FILE;

    return SUCCESS;
}
```

This is much easier to read and understand than using magic number return values. Furthermore, the caller can test the function's return value against the appropriate enumerator, which is easier to understand than testing the return result for a specific integer value.

```
if (readFileContents() == SUCCESS)
    {
    // do something
    }
else
    {
    // print error message
    }
```

Enumerated types are best used when defining a set of related identifiers. For example, let's say you were writing a game where the player can carry one item, but that item can be several different types. You could do this:

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

enum ItemType
{
    ITEMTYPE_SWORD,
    ITEMTYPE_TORCH,
    ITEMTYPE_POTION
};

std::string getItemName(ItemType itemType)
{
    if (itemType == ITEMTYPE_SWORD)
        return std::string("Sword");
    if (itemType == ITEMTYPE_TORCH)
        return std::string("Torch");
    if (itemType == ITEMTYPE_POTION)
        return std::string("Potion");

    // Just in case we add a new item in the future and forget to update this function
    return std::string("???");
}

int main()
{
    // ItemType is the enumerated type we've defined above.
    // itemType (lower case i) is the name of the variable we're defining (of type ItemType).
    // ITEMTYPE_TORCH is the enumerated value we're initializing variable itemType with.
    ItemType itemType = ITEMTYPE_TORCH;

    std::cout << "You are carrying a " << getItemName(itemType) << "\n";

    return 0;
}
```

Or alternatively, if you were writing a function to sort a bunch of values:

```
enum SortType
{
    SORTTYPE_FORWARD,
    SORTTYPE_BACKWARDS
};

void sortData(SortType type)
{
    if (type == SORTTYPE_FORWARD)
        // sort data in forward order
    else if (type == SORTTYPE_BACKWARDS)
        // sort data in backwards order
}
```

Many languages use Enumerations to define booleans. A boolean is essentially just an enumeration with 2 enumerators: false and true! However, in C++, true and false are defined as keywords instead of enumerators.

Quiz

1) Define an enumerated type to choose between the following monster races: orcs, goblins, trolls, ogres, and skeletons.

2) Define a variable of the enumerated type you defined in question 1 and assign it the troll enumerator.

3) True or false. Enumerators can be:
3a) given an integer value
3b) not assigned a value
3c) given a floating point value

3d) negative
3e) non-unique
3f) initialized with the value of prior enumerators (e.g. COLOR_MAGENTA = COLOR_RED)

Quiz answers

1) **Hide Solution**

```
enum MonsterType
{
    MONSTER_ORC,
    MONSTER_GOBLIN,
    MONSTER_TROLL,
    MONSTER_OGRE,
    MONSTER_SKELETON
};
```

2) **Hide Solution**

```
MonsterType eMonsterType = MONSTER_TROLL;
```

3) **Hide Solution**

3a) True.
3b) True. Enumerators not assigned a value will be implicitly assigned the integer value of the previous enumerator + 1. If there is no previous enumerator, the numerator will assume value 0.
3c) False.
3d) True.
3e) True.
3f) True. Since enumerators evaluate to integers, and integers can be assigned to enumerators, enumerators can be initialized with other enumerators (though there is typically little reason to do so!).


4.5a -- Enum classes


Index


4.4b -- An introduction to std::string

Share this:

C++ TUTORIAL | PRINT THIS POST

189 comments to 4.5 — Enumerated types

« Older Comments 1 2 3

ayush
June 26, 2018 at 3:04 am · Reply

```
#include<iostream>
```