S.4.5 — Enumerated types

BY ALEX ON JUNE 19TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 3RD, 2020

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** or **enum**) 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 enumerations and enumerators

Providing a name for an enumeration is optional, but common. Enums without a name are sometimes called anonymous enums. Enumeration names are often named starting with a capital letter.

Enumerators must be given names, and are typically named either using all caps (e.g. COLOR_WHITE), or prefixed with a k and intercapped (e.g. kColorWhite).

Enumerator scope

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
```

If you want to use a different integer type for enumerators, for example to save bandwidth when networking an enumerator, you can specify it at the enum declaration.

```
// Use an 8 bit unsigned integer as the enum base.
enum Color : std::uint_least8_t
{
    COLOR_BLACK,
    COLOR_RED,
    // ...
};
```

Since enumerators aren't usually used for arithmetic or comparisons, it's safe to use an unsigned integer. We also need to specify the enum base when we want to forward declare an enum.

```
enum Color; // Error
enum Color : int; // Okay

// ...

// Because Color was forward declared with a fixed base, we
// need to specify the base again at the definition.
enum Color : int
{
    COLOR_BLACK,
    COLOR_RED,
    // ...
};
```

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 can only forward declare them when you also specify a fixed base. 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) **Show Solution**

2) **Show Solution**

3) **Show Solution**

S.4.5a -- Enum classes

Index

S.4.4c -- Using a language reference

C++ TUTORIAL | PRINT THIS POST

249 comments to S.4.5 — Enumerated types

« Older Comments 1 2 3 4

AbraxasKnister
February 7, 2020 at 7:33 am · Reply

There's an occurrence of non colorized best practice and rule.

nascardriver
February 7, 2020 at 9:02 am · Reply

Thanks for pointing it out! The lessons are gradually updated to the new style, but it takes some time.

Armitage
January 21, 2020 at 6:23 am · Reply

```
enum Color : int; // Okay

// Because Color was forward declared with a fixed base, we
// need to specify the base again at the definition.
enum Color : int //? Why
{
    COLOR_BLACK,
    COLOR_RED,
    // ...
};
```

In forward declaration yes, but do we need specify base in definition? I have no error if I don't.

nascardriver
January 21, 2020 at 7:32 am · Reply

Yes, every redeclaration has to have the same base specifier as the previous declarations. If your compiler doesn't error out, you might have forgotten to disable compiler extensions (Lesson 0.10).

Armitage
January 21, 2020 at 1:00 pm · Reply

You are right. But I have another problem:

```
enum Color : int;

int main()
{
    std::cout << COLOR_BLACK;

    return 0;
}

enum Color:int
{
    COLOR_BLACK,
    COLOR_RED,
    COLOR_WHITE
```

```
15 };
```

'COLOR_BLACK': undeclared identifier. So How i must use enum forward declaration correctly?

nascardriver
January 22, 2020 at 3:59 am · Reply

You can't use an enumerator before it's defined. A forward declaration of an enum can be used to forward declare a function.

```
1  enum Color : int;
2
3  void fn(Color color);
4
5  int main() { /* ... */ }
6
7  enum Color : int
8  {
9    COLOR_BLACK,
10   COLOR_RED,
11   COLOR_WHITE
12 };
13
14 void fn(Color color) { /* ... */ }
```

In a small single-file project, this doesn't make sense. You might need it when you separate your code into multiple files to prevent circular dependencies.

Armitage
January 22, 2020 at 7:22 am · Reply

I think I get It. Thanks.
But VS2019 let you omit base type on enum declaration if it int.

nascardriver
January 22, 2020 at 7:54 am · Reply

Yep, I double checked and VS doesn't error out even with /Za and /permissive-. You'll have to pay attention yourself not the omit the base when you redeclare the enum. This is non-standard behavior, if you rely on this, your code won't compile with other compilers.

kavin
January 10, 2020 at 7:18 am · Reply

Hi, i have written a program based on 1st example. My doubt is, if its ok like the user below "Ri" has written the program without using MonsterClass enumeration parameter in the getRace() by directly accepting values as int? Without using a static_cast how will his program convert int input to compare with enumerators in MonsterRaces ?
Here's my code. Is this ok or any improvements needed ?

```
1  #include<iostream>
2  #include<string>
3
4  enum MonsterRaces
```

```
{
    MONRACE_ORCS,
    MONRACE_GLOBLINS,
    MONRACE_TROLLS,
    MONRACE_OGRES,
    MONRACE_SKELETONS,
};
int selectRace()
{
    std::cout << "Choose your race number:\n";
    std::cout << "Orcs-0\n";
    std::cout << "Globlins-1\n";
    std::cout << "Trolls-2\n";
    std::cout << "Ogres-3\n";
    std::cout << "Skeletons-4\n\n";
    int raceNum{};
    std::cin >> raceNum;
    return raceNum;
}
std::string getRace(MonsterRaces monRace)
{
    if (monRace == MONRACE_ORCS)
        return "Orcs\n";
    else if (monRace == MONRACE_GLOBLINS)
        return "Globlins\n";
    else if (monRace == MONRACE_TROLLS)
        return "You chose Trolls\n";
    else if (monRace == MONRACE_OGRES)
        return "Ogres\n";
    else if (monRace == MONRACE_SKELETONS)
        return "Skeletons\n";
    else
        return "Invalid. Choose correct race by restarting program\n";
}

int main()
{
    MonsterRaces monRace{ static_cast<MonsterRaces>(selectRace()) };

    std::cout << "The race you have selected is "<<getRace(monRace) << '\n';
    return 0;
}
```

nascardriver
January 12, 2020 at 2:10 am · Reply

enumerators of an `enum` can be compared to an `int` without a cast, though, casting the `int` first is better to avoid mixed types.

- Line 15-19: Those are magic numbers/strings. If you update `MonsterRaces`, this list is incorrect. Use your enumerators to print the selection.

kavin
January 12, 2020 at 5:42 am · Reply

If i use without a cast i get this error:
(42,40): error C2440: 'initializing': cannot convert from 'initializer list' to 'MonsterRaces'
(42,40): message : Reason: cannot convert from 'int' to 'MonsterRaces'

So i guess static_cast is compulsory in my case. I am using VS2019.

For the line 15-19 i am not sure how to do that :(Even if i do a function with something like,
if(MONRACE_ORCS==0)
std::cout<<"Orcs-0\n";
if(MONRACE_GLOBLINS==1)
std::cout<<"Globlins-1"
etc.,
it would not work properly if i change the order or add a new race in enum MonsterRaces right?
What is the code to do that?

nascardriver
January 13, 2020 at 1:03 am · Reply

> If i use without a cast i get this error
You can't initialize an `enum` with an `int`, you can only compare it to an `int`.

Your solution would skip the monster if the enumerator's value is not the same as the value in the string. You still have a magic number/string. You can print the enumerators, rather than writing a value yourself.

```
1 std::cout << "Orcs-" << static_cast<int>(MonsterRaces::MONRACE_ORCS) << '\n';
2 std::cout << "Goblins-" << static_cast<int>(MonsterRaces::MONRACE_GLOBLINS) <<
3 // ...
```

kavin
January 12, 2020 at 7:02 am · Reply

Sorry for double reply. I changed 15-19 like this but it wont print "Orcs-0". But it would out put as orcs if i input 0. I dunno why it is not getting printed ! Is this the correct way or ?

```
1  {
2  std::cout << "Choose your race number:\n";
3      if constexpr (static_cast<bool>(MONRACE_ORCS))
4              std::cout << "Orcs-0\n";
5      if constexpr (static_cast<bool>(MONRACE_GLOBLINS))
6              std::cout << "Globlins-1\n";
7      if constexpr (static_cast<bool>(MONRACE_TROLLS))
8              std::cout << "Trolls-2\n";
9      if constexpr (static_cast<bool>(MONRACE_OGRES))
10             std::cout << "Ogres-3\n";
11     if constexpr (static_cast<bool>(MONRACE_SKELETONS))
12             std::cout << "Skeletons-4\n";
13 }
```

Charan
December 16, 2019 at 8:40 am · Reply

Hey,It is stated that the enumerator can be input through keyboard. Let's say the enumerators are assigned(by us or by the system) like COLOR_BLUE = 3,COLOR_RED = 5. What if I entered a value of 7 in some integer variable x, and then static cast x into the type Color?

nascardriver
December 16, 2019 at 8:48 am · Reply

All values that are valid for an enum's underlying type (eg. int), are valid for the enum. If you let the user input a value, or create an enumerator via a cast in general, you should verify that the value is one of your enumerators.
If you manually assigned values to your enumerators, you'll have to check one after the other. If they were numbered automatically, you can check if the untrusted value is a value is in the range from your lowest enumerator to your highest enumerator.

ErwanDL
November 2, 2019 at 5:05 pm · Reply

Hey Alex, why is it still written "Because the compiler needs to know how much memory to allocate for an enumeration, you cannot forward declare enum types.", although you wrote a few lines above that forward declaration of enums is possible (since C++11 if I'm right), as long as you also forward declare the enum base ?

nascardriver
November 3, 2019 at 12:59 am · Reply

Hi!

I missed that sentence when I added the paragraph about forward declarations. Base specifications and forward declarations are indeed possible since C++11. Thanks for pointing it out! Lesson updated.

Ri
October 28, 2019 at 11:06 am · Reply

```
1  #include <iostream>
2  #include <string>
3
4  int monChoice()
5  {
6      std::cout << "Choose a Monster Type to Spawn:'\n'Type: 0 - Orc '\n'1 - Skeleton '\n'2 -
7      int monChoice{};
8      std::cin >> monChoice;
9
10     return monChoice;
11 }
12
13 enum monsters
14 {
15     MONSTER_ORC,
16     MONSTER_SKELETON,
17     MONSTER_TROLL,
18     MONSTER_GOBLIN,
19     MONSTER_OGRE,
20
21 };
22
23 std::string monName(int monToSpawn)
24 {
25     if (monToSpawn == MONSTER_ORC)
26         return std::string("an Orc");
27     else if (monToSpawn == MONSTER_SKELETON)
28         return std::string("a Skeleton");
29     else if (monToSpawn == MONSTER_TROLL)
```

```
30          return std::string("a Troll");
31      else if (monToSpawn == MONSTER_GOBLIN)
32          return std::string("a Goblin");
33      else if (monToSpawn == MONSTER_OGRE)
34          return std::string("an Ogre");
35      else
36          std::cout << "Invalid Input, Try again." << monChoice();
37
38  }
39
40  void spawnMon(int monToSpawn)
41  {
42      std::cout << "Spawning... " << monName(monToSpawn);
43
44  }
45
46  int main()
47  {
48      int monToSpawn{ monChoice() };
49      spawnMon(monToSpawn);
50
51      return 0;
52  }
```

Love the tutorials so far! Good job making them very intuitive. I tried to create a sort of spawning system for game (with what little knowledge I have!), any tips or ways to improve this code would be greatly appreciated. Many thanks.

nascardriver
October 29, 2019 at 2:28 am · Reply

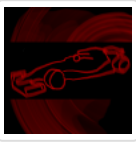
Hi!

- If your program prints anything, the last thing it prints should be a line feed ('\n').
- Line 6 uses neither `monster` nor `monName`. If you update the string, `monName` or `monsters`, your program prints wrong instructions.
- Line 26, 28, etc: You don't need to manually call `std::string`.
- Avoid abbreviations.
- `monName::monToSpawn` should be a `monster`.

Ri
October 29, 2019 at 5:41 am · Reply

Thank you for the feedback! Just to clarify, by saying "-Line 26, 28, etc: You don't need to manually call `std::string`." You mean that it would be better to use a "using std::string" to improve readability?

Thanks again!



nascardriver
October 29, 2019 at 5:42 am · Reply

No, you can

```
1  return "an Orc";
```

Ri
October 29, 2019 at 9:45 am · Reply

Ahh I see, thanks again!

Elis
November 10, 2019 at 1:09 am · Reply

Hi! I fiddled a bit with this code to accept text input as opposed to integers.
If nascardriver or anyone is around I'd be interested in hearing their opinion on the usage of the '?:' operator,
too long and convoluted?

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

enum monsters
{
    MONSTER_ORC,
    MONSTER_SKELETON,
    MONSTER_TROLL,
    MONSTER_GOBLIN,
    MONSTER_OGRE,

};

int monChoice()
{
    std::cout << "Choose a Monster Type to Spawn:\n Type in: \n Orc \n Skeleton \n"
                 " Troll \n Goblin \n or \n Ogre \n";
    std::string monChoice{};
    std::getline(std::cin, monChoice);
    int enemy{ ((monChoice == "orc") ? MONSTER_ORC :
                (monChoice == "skeleton") ? MONSTER_SKELETON :
                (monChoice == "troll") ? MONSTER_TROLL :
                (monChoice == "goblin") ? MONSTER_GOBLIN :
                (monChoice == "ogre") ? MONSTER_OGRE : 32767 )};
    return enemy;
}


void spawnPrint(int monChoice())
{
    int enemy{ monChoice() };
    std::cout << "You're facing " << ((enemy == MONSTER_ORC) ? "an orc" :
                                      (enemy == MONSTER_SKELETON) ? "a skeleton" :
                                      (enemy == MONSTER_TROLL) ? "a troll" :
                                      (enemy == MONSTER_GOBLIN) ? "a goblin" :
                                      (enemy == MONSTER_OGRE) ? "an ogre" : "nothing.
                                      << '!' ;
}

int main()
{
    spawnPrint(monChoice);
    return 0;
}
```

nascardriver
November 10, 2019 at 2:55 am · Reply

> opinion on the usage of the '?:' operator
The problem is that it's not reusable. Add `getMonsterByName` and `getNameOfMonster` with the conditionals inside, then it's ok.
"ok", because in reality you would use different techniques that you don't know about yet. Your code shows that you understood the conditional operator, that will come in handy in other quizzes.

> 32767
Magic number. There's nothing stopping this value from being valid. Add a `MONSTER_INVALID` to `monsters` and return that instead. It's obvious in the code that this is an invalid value and it won't be a duplicate.

Omri
October 12, 2019 at 9:53 pm · Reply

"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"
Is it not a postfix in this example...

nascardriver
October 12, 2019 at 11:35 pm · Reply

A prefix is at the front.
A postfix/suffix is at the end.

```
1 COLOR_RED // COLOR_ is a prefix
2 RED_COLOR // _COLOR is a suffix
3
4 ++i; // prefix
5 i++; // postfix
```

Mhz93
October 2, 2019 at 6:16 am · Reply

In the last line of the first example of this lesson, the word "red" has been printed in black color which should be blue!!
Also the same with the word "GREEN" in the second example.
Also the word "POTION" in line 8 in the 15th example.
Also the word "BACKWARDS" in line 4 in the 16th (the last) example.
and finally "SKELETON" in 7th line in the solution of first queez.
sorry I know that's not worth to mention!! but as I like this site and enjoy learning CPP from, I like to see this as well as possible.

Kris
September 19, 2019 at 8:32 am · Reply

Hi nascardriver,

No wonder there is so many comments on this topic. I found it very confusing myself, too.

I have one comment: declaring ItemType itemType = ITEMTYPE_TORCH is very confusing; why not to declare it this way: ItemType weapon = ITEMTYPE_TORCH. Just too many "itemtypes" for a novice. Couldn't make use of the function getItemName() to print out the name of the weapon.

Although it forced me to read this part of the lesson many times, play with it in Visual Studio many times until I got it! :)

Thanks.

« Older Comments 1 2 3 4