# Constants vs #define vs enum

When writing code in C / C++ - or anyone other language for that matter – your code often becomes littered with hard-coded values that only you, the programmer, understands.  These values are often referred to as ‘magic numbers’ as their use and derivation is a mystery to other programmers.  The use of a constant in your code allows you to assign a more textual and descriptive value to your magic number.  Which is more readable, the value 52 or the constant NUMBER\_OF\_CARDS\_IN\_DECK ?

Constants in C/ C++ can be handled a number of ways, each with their own pros and cons.  These include:

* the const qualifier for a variable
* the #definepreprocessor directive
* the enum keyword

## The Const Qualifier

Adding the const qualifier to a variable declaration tells the compiler that the variable cannot be modified by code.  Any attempt to update the value of the variable will result in a compile time error.  Declaring a variable as a constant allows the compiler to optimise the code as it knows in advance that no updates are possible.

const int cards\_in\_deck = 52;

cards\_in\_deck = 53;               < this will result in a compile time error.

The const keyword can also be used when specifying parameters to a function.  When calling a function in C / C++, all parameters are passed by value unless you pass the pointer to the variable so adding theconst keyword does is not used to prevent updates to the calling parameter outside of the function but instead to ensure that the parameter is not updated within the function.  Again, the use of the const keyword allows the compiler to optimize the code knowing that the variable is immutable.

int cards\_in\_deck = 52;

shuffleCards(cards\_in\_deck);

cards\_in\_deck = 53;                   < This will compile correctly as the cards\_in\_deck variable is not a constant.

void shuffleCards(const int cards\_to\_shuffle) {

  cards\_to\_shuffle--;                 < This will fail to compile as the parameter is defined as a constant.

}

## The #define Preprocessor Directive

Before compiling an application, the C / C++ compiler runs a preprocessing that looks for compiler directives and acts on them first. Preprocessing directives can be used to define constants and to include / exclude sections of code based on the platform the compilation is targeting or to remove debugging code from a release build.

The directives can also define simple macro functions as described in my article on debugging in the 6th issue of the Arduboy Magazine. More information on preprocessing directives can be found here <http://www.cplusplus.com/doc/tutorial/preprocessor/>

The simple #define directive instructs the preprocessor to look for the first operand and replace it within the code using the second operand. In the example below, I have created constants for three animal types, as numeric values, and three name constants as strings. One thing to note though is that the preprocessor will not check the validity of the substitutions it makes and can result in compilation errors.

#define CAT           1

#define DOG           2

#define MOUSE         3

#define CAT\_NAME      "Leo"

#define DOG\_NAME      "Fido"

#define MOUSE\_NAME    "Mickey"

int myAnimal = CAT;

String myAnimalName = CAT\_NAME;

String myOtherCat = DOG; < This will fail to compile as the variable is defined as a string but an attempt is being made to initialize it with a numeric value.

One nice thing about the #define directive is that it allows you to combine values together, like those shown below. I have used this functionality in the *Pipes* game to define the EEPROM settings used when saving and retrieving player settings where all memory positions are defined relative to the EEPROM\_STORAGE\_SPACE\_START constant. The same techniques are used in the ArduboyTones library to construct the high volume notes that add the original note value plus a TONE\_HIGH\_VOLUME constant.

#define SMALL           1

#define MEDIUM          2

#define LARGE           3

#define CAT             10

#define DOG             20

#define MOUSE           30

#define CAT\_SMALL       CAT + SMALL

#define CAT\_MEDIUM      CAT + MEDIUM

#define CAT\_LARGE       CAT + LARGE

#define DOG\_SMALL       DOG + SMALL

#define DOG\_MEDIUM      DOG + MEDIUM

#define DOG\_LARGE       DOG + LARGE

#define MOUSE\_SMALL     MOUSE + SMALL

#define MOUSE\_MEDIUM    MOUSE + MEDIUM

#define MOUSE\_LARGE     MOUSE + LARGE

int myCat = CAT;

int myLargeCat = CAT\_LARGE;

## Enumerations

In C / C++, an enumeration is a data type consisting of a set of named values of type integer. Enumerations can be named or anonymous as shown below. Note in the first example, I have nominated that the first element, cat, is assigned the numeric value 10. Dog will automatically be assigned the value of 11 and mouse, 12. In the second example where no starting number is specified, the items are numbered from zero onwards.

enum Pet {

cat = 10,

dog,

mouse

};

enum {

horse,

cow,

sheep

};

The following declarations are all valid.

int aHorse = horse;

Pet aCat = cat;

Pet aDog = Pet::dog;

Named enumerations can be used when defining parameters to a function. As you can see from the sample calls, the declaration will not prevent you from passing any other enumeration type or even an integer.

void printPetDetails(Pet thePet) {

switch (thePet) {

case Pet::cat:

Serial.println("cat");

break;

case Pet::dog:

Serial.println("dog");

break;

case Pet::mouse:

Serial.println("mouse");

break;

default:

Serial.println("other");

break;

}

}

printPetDetails(aCat); < prints “cat”

printPetDetails(aDog); < prints “dog”

printPetDetails(aHorse); < prints “other” but is illogical as it isn’t a Pet

printPetDetails(2); < prints “other” but is illogical as it isn’t a Pet

Enumerations can also include the class keyword as shown below. This overcomes the limitations in the previous example by ensuring that references to a Pet are checked in both the declarations and passing of variables to a function.

enum class Pet {

cat = 10,

dog,

mouse

};

enum class FarmAnimal {

horse,

cow,

sheep

};

Pet aCat = Pet::cat;

Pet aDog = dog; < Will not compile. The elements of Pet must be specified as Pet::{element}

FarmAnimal aHorse = FarmAnimal.horse;

printPetDetails(aCat); < prints “cat”

printPetDetails(aDog); < prints “dog”

printPetDetails(aHorse); < will not compile as the compiler cannot convert a FarmAnimal to a Pet.

printPetDetails(2); < will not compile as the compiler cannot convert an integer to a Pet.

However, enumerations declared with the class scope are not viewed as simple integers anymore. The following code illustrates this:

Serial.println(aCat); < will not compile as there is no overload Serial.println(&Pet) function.

Serial.println((int)aCat); < casting the number back to an integer allows it to compile.

You can use any or all three of these techniques in your code. I prefer to use #defines when the number actually means something, such as #define ARRAY\_SIZE 100, but would opt for enumerations where the number is irrelevant.

Use whatever you feel comfortable with!