Constants & Typedef

Contents

- What is a constant?
- Globals
- Constant function arguments
- Typedef
- Static variables
- Extern keyword

Const

- A constant is a variable that cannot be changed after it has been initialised. That is, it is read only.
- A constant can prevent a value from being inadvertently modified

```
const float _PI = 3.1415926535;
//The following line will not compile:
_PI =_PI * _PI;
```

- This makes code a bit more readable

Global constants

- Global variables are a bad idea, especially while you are still becoming familiar with C++
- Global constants, however, are perfectly acceptable
- Since a constant cannot have its value changed, then the issues surrounding globals no longer apply.
- This is useful when all functions in a program need access to the same unchanging value (

Array sizes

- Constants can also be used to serve as array limits

```
const int MAX_SCORES = 100;
int scoreArray[MAX_SCORES];
for( int i = 0; i < MAX_SCORES; ++i )
{...}
```

- To change the size of the array only the initialisation of the constant needs to be modified

Constant Function Parameters

- Sometimes safety is important
- Recall that when we pass an array into a function we are working on the actual array, not a copy.

Const Function Parameters

- We can declare function parameters as constant if we don't want the function to modify any values that are passed into it.

```
int findSum(const int a_numbers[], int a_size)
{
	int sum = 0;
	for (unsigned int i = 0; i < a_size; ++i)
	{
		sum += a_numbers[i];
	}
	return sum;
}
```

- This functionality makes it obvious that this function will not modify the array passed into it

Const Correctness

- If the intent of a function is to only read from parameters passed into it and not modify them then they should be declared as *const*
- This makes it clear that these parameters are only for input
- The correct use of *const* improves the safety and understanding of how a program is mean to function

So what's a typedef?

- *typedef* is a C++ keyword
  - It's shorthand for *type define*
- It allows us to create new names, or alias' for types.
  - For example we could do the following… if we really wanted to…

```
typedef float currency;
currency g_CashMonies = 34.50f;
```

Oh so that's a *typedef*

- You may have already been using some type defines without realising it.
  - Take the following example:

```
typedef unsigned int size_t;
```

But how's that useful?

- *typedef* is commonly used to create abbreviations for long type names as seen in the following STL declaration for a 2D grid:

```
// create a 5x10 2D dynamic array
typedef std::vector<int> Row;
typedef std::vector<Row> Grid;

// allows the following, clear declaration
Grid grid(5, Row(10));

// rather than
std::vector<std::vector<int>> grid(5, std::vector<int>(10));
```

Clarity!

- So *typdef* can be used to add a level of clarity to functions for example

```
typedef unsigned int _error;
_error LoadTexture( const char* a_textureName, unsigned int& a_textureID );
```

- We can see here that the function *LoadTexture* returns an error code, and not a texture ID as one might assume if the return type was just an *unsigned int*

About Static

- The static keyword can be used in a number of different ways

- The behavior changes depending on how and where it is used with
  - With global variables / functions
  - With local variable / functions
  - With Class member data
  - With Class member function

Local Variables and Functions (Local Static)

- Local variables are variables declared within a function; local variables are de-allocated at the end of the function call so all values stored are not available

- Local variables declared as static are allocated in the static variable space at load time, but not initialized until the first run of the function

Local Variables and Functions (Local Static)

- Code Example

```
void Increment()
{
   static int i = 0;
   ++i;
   cout << "The value of i is : " << i << endl;
}

int main()
{
    for(int j = 0; j < 5; j++)
      Increment();
    return 0;
}
```

- What is the printout of this code?
  - Does variable i retain its value in Increment()?
  - Code this in Visual Studio and see for yourself

Declaration vs definition

- This has hopefully become more clear since learning about functions, but it bears repeating
- Declaration is the act of telling the compiler that something will exist
  - For example, a function prototype
- A definition actually creates or implements the object
  - For example a function definition
- An object must be defined once and only once. If only a declaration is provided then the code will not compile. If multiple definitions are provided then the code will not compile.

```
//declaration
double f(int, double);
```

```
//definitions
double f(int a_i, double a_d)
{
     return a_i + a_d;
}
int result;
int counter = 0;
```

Canberra Institute of Technology

aie
SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX

Extern

- We usually use multiple files of code when making a program. Often we want to be able to access an object throughout a few of these files
  - For example the global constant PI.
- However, defining an object in one file doesn't mean that another file knows about that object.
- To solve this we can put the definition of that object in a header file and include that header file in all the source files where we need to use the object.
- But this introduces other problems…

Extern

- If we use the “solution” on the previous slide we will come across a linker error that tells us the object has been multiply defined.
- What we need to do is only *define* the object in one file and *declare* it in all the others.
- To do this we use the extern keyword.

```
extern const double PI;
```

Extern

- Using the extern keyword we now have a declaration for our object.
- But we still need to define it somewhere. Make sure that this happens in one (and only one) file.

```
const double PI = 3.14159;
```

Static globals

- Making a global variable static makes the variable private to the file in which it is declared.

- Therefore static global cannot use the extern keyword.

Static Duration

- All memory allocated for global variables occurs before main() and the variables are de-allocated *after* the final brace "}" of main().

- This is also true for *most* static variables
  - Cannot control the order they are created or deleted
  - Although, generally speaking for global variables, they will be created and destroyed in the order we have defined them

Order And Flow

- Static and global memory allocated
- Static and global variables are initialised
- Main begins
- Local static variables initialized on first call to the function
- Main ends
- Static and global memory released

Summary

- This lecture covered a number of loosely related concepts
- Consts are useful when we need to use the same unchanging value multiple times, or when we need to make sure a function will not change a value that has been passed in.
- Typedef allows us to rename data types, which makes for increased readability of code.
- The static keyword has many uses – so far we've learned about creating static variables inside functions as well as sharing globals between files with extern.

References

- Brice, 2014, *Difference between static memory allocation and dynamic memory* allocation, StackOverflow
  - http://stackoverflow.com/questions/8385322/difference-between-static-memory-allocation-and-dynamic-memory-allocation
- Bronson, G 2013, *C++ for Engineers and Scientists 4th Edition*, Cengage Learning

Canberra Institute of Technology

aie
SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX