**Item 2: Prefer consts, enums, and inlines to #defines.**

#define ASPECT\_RATIO 1.653

Prefer to replace the macro with a constant

const double AspectRatio = 1.653;

But why?

Macro is processed by pre-processor and expended code contains value 1.653 rather than ASPECT\_RATIO which is more meaningful. The symbolic name ASPECT\_RATIO may never be seen by compilers; it may be removed by the preprocessor before the source code ever gets to a compiler.

This can be confusing if we get an error during compilation involving the use of the constant, because the error message may refer to 1.653, not ASPECT\_RATIO.

There’s no way to create a class-specific constant using a #define, because #defines don’t respect scope.

class GamePlayer {

private:

static const int NumTurns = 5; // constant declaration

int scores[NumTurns]; // use of constant

};

As of now it has only declaration for NumTurns, not a definition.Usually, C++ requires that you provide a definition for anything you use, but class-specific constants that are static and of integral type (e.g., integers, chars, bools) are an exception.

As long as we don’t take their address, you can declare them and use them without providing a definition. If you do take the address of a class constant, or if your compiler incorrectly insists on a definition even if you don’t take the address, you provide a separate definition like this:

const int GamePlayer::NumTurns; //definition of NumTurns see below for why no value is

//given

// call f with the maximum of a and b

#define CALL\_WITH\_MAX(a, b) f((a) > (b) ? (a) : (b))

int a = 5, b = 0;

CALL\_WITH\_MAX(++a, b); // a is incremented twice

CALL\_WITH\_MAX(++a, b+10); // a is incremented once

We can get all the efficiency of a macro plus all the predictable behavior and type safety of a regular function by using a template for an inline function

template<typename T>

inline void callWithMax(const T& a, const T& b) {

a > b ? a : b);

}

This template generates a whole family of functions, each of which takes two objects of the same type and calls f with the greater of the two objects. There’s no need to parenthesize parameters inside the function body, no need to worry about evaluating parameters multiple times, etc. Furthermore, because callWithMax is a real function, it obeys scope and access rules. For example, it makes perfect sense to talk about an inline function that is private to a class. In general, there’s just no way to do that with a macro.

**Things to Remember**

✦ For simple constants, prefer const objects or enums to #defines.

✦ For function-like macros, prefer inline functions to #defines.

**Item 3: Use const whenever possible.**

you can specify whether the pointer itself is const, the data it points to is const, both, or neither:

char greeting[] = "Hello";

char \*p = greeting; // non-const pointer, non-const data

const char \*p = greeting; // non-const pointer, const data

char \* const p = greeting; // const pointer, non-const data

const char \* const p = greeting; // const pointer, const data

If the word const appears to the left of the asterisk, what’s pointed to is constant; if the word const appears to the right of the asterisk, the pointer itself is constant; if const appears on both sides, both are constant.

void f1(const Widget \*pw); // f1 takes a pointer to a constant Widget object

void f2(Widget const \*pw); // so does f2

**Const-ness in STL**

std::vector<int> vec;

const std::vector<int>::iterator iter = vec.begin() // iter acts like a T\* const

\*iter = 10; // OK, changes what iter points to

++iter; // error! iter is const

std::vector<int>::const\_iterator cIter = vec.begin() // cIter acts like a const T\*

\*cIter = 10; // error! \*cIter is const

++cIter; // fine, changes cIter

Function with return const type:

Having a function return a constant value is generally inappropriate, but sometimes doing so can reduce the incidence of client errors without giving up safety or efficiency.

class Rational { ... };

const Rational operator\*(const Rational& lhs, const Rational& rhs);

For Example: Why should the result of operator\* be a const object? Because if it weren’t, clients would be able to commit atrocities like this:

Rational a, b, c;

...

(a \* b) = c; // invoke operator= on the result of a\*b

if (a \* b = c) // oops, meant to do a comparison!

**Const Member Functions used for:**

It’s important to know which functions may modify an object and which may not.

They make it possible to work with const objects.

class TextBlock {

public:

...

const char& operator[](std::size\_t position) const // operator[] for const objects

{

return text[position];

//Here return type also const required, because this method

//returns reference of text string and you can’t modify this later point of

//time also.

}

char& operator[](std::size\_t position){ // operator[] for non-const objects

return text[position];

}

std::string text;

};

TextBlock’s operator [] can be used like this:

TextBlock tb("Hello");

std::cout << tb[0]; // calls non-const

const TextBlock ctb("World");

std::cout << ctb[0]; // calls const

What does it mean for a member function to be const?

There are two prevailing notions:

1. *Bitwise const-ness* (also known as *physical const-ness*) and
2. *logical const-ness*

***Bitwise constness***

Bitwise constness state that if any member function is declared as const then it doesn’t modify any of the bits inside the object (excluding those that are static).

C++’s definition of bitwise constness, a const member function isn’t allowed to modify any of the non-static data members of the object on which it is invoked.

***logical constness***

See the below program:

Suppose we have a TextBlock-like class that stores its data as a char\* instead of a string, because it needs to communicate through a C API that doesn’t understand string objects.

class CTextBlock {

public:

...

char& operator[](std::size\_t position) const // inappropriate (but bitwise const)

// declaration of operator[]

{

return pText[position];

}

private:

char \*pText;

};

This class (inappropriately) declares operator[] as a const member function, even though that function returns a reference to the object’s internal data .Operator[]’s implementation doesn’t modify pText in any way. As a result, compilers will happily generate code for operator[]; it is, after all, bitwise const, and that’s all compilers check for. But look what it allows to happen:

const CTextBlock cctb("Hello"); // declare constant object

char \*pc = &cctb[0]; // call the const operator[] to get a

// pointer to cctb’s data

\*pc = ’J’; // cctb now has the value “Jello”

Surely there is something wrong when we create a constant object with a particular value and invoke only const member functions on it, yet we can still change its value.

***This leads to the notion of logical constness***

Modify member data inside const member function using *mutable* keyword.

class CTextBlock {

public:

...

std::size\_t length() const;

private:

char \*pText;

mutable std::size\_t textLength; // these data members may always be

// modified, even in

mutable bool lengthIsValid; // const member functions

};

std::size\_t CTextBlock::length() const

{

if (!lengthIsValid) {

textLength = std::strlen(pText); // now fine

lengthIsValid = true; // also fine

}

return textLength;

}

**Avoiding Duplication in const and Non-const Member Functions.**

We have same method for const and non const version. For example operator [], here problem with redundant code which is appear in both methods.

In this case, the const version of operator[] does exactly what the non-const version does, it just has a const-qualified return type.

class TextBlock {

public:

...

const char& operator[](std::size\_t position) const

{

// do bounds checking, log access data and verify data integrity

return text[position];

}

char& operator[](std::size\_t position)

{

// do bounds checking, log access data and verify data integrity

return text[position];

}

private:

std::string text;

};

Casting away the const on the return value is safe (not possible in vice-versa), in this case, because whoever called the non-const operator [] must have had a non const object in the first place. Otherwise they couldn’t have called a non-const function, so having the non-const operator [] call the const version is a safe way to avoid code duplication.

class TextBlock {

public:

...

const char& operator[](std::size\_t position) const // same as before

{

// do bounds checking, log access data, verify data integrity

if(text.length() < position){

cout<<"Invalid index ::"<<position<<endl;

return text[text.length()-1];

}

else{

cout<<"Index Validation successful\n";

}

return text[position];

}

char& operator[](std::size\_t position) // now just calls const op[]

{

// 1. add const to \*this’s type;

// 2. call const version of op[]

// 3. cast away const on op[]’s return type;

return const\_cast<char&>( static\_cast<const TextBlock&>(\*this) [position] );

}

};

**Things to Remember**

✦ Declaring something const helps compilers detect usage errors. Const can be applied to objects at any scope, to function parameters and return types, and to member functions.

✦ Compilers enforce bitwise constness, but you should program using logical constness.

✦ When const and non-const member functions have essentially identical implementations, code duplication can be avoided by having then non-const version call the const version.

**Item 4: Make sure that objects are initialized before they’re used**

class PhoneNumber { ... };

class ABEntry { // ABEntry = “Address Book Entry”

public:

ABEntry(const std::string& name, const std::string& address, const std::list<PhoneNumber>& phones);

private:

std::string theName;

std::string theAddress;

std::list<PhoneNumber> thePhones;

int numTimesConsulted;

};

ABEntry::ABEntry(const std::string& name, const std::string& address,

const std::list<PhoneNumber>& phones)

{

theName = name; // these are all assignments,

theAddress = address; // not initializations

thePhones = phones;

numTimesConsulted = 0;

}

The rules of C++ stipulate that data members of an object are initialized *before* the body of a constructor is entered. Inside the ABEntry constructor, theName, theAddress, and thePhones aren’t being initialized, they’re being *assigned*. Initialization took place earlier — when their default constructors were automatically called prior to entering the body of the ABEntry constructor.

ABEntry::ABEntry(const std::string& name, const std::string& address, const std::list<PhoneNumber>& phones)

: theName(name),

theAddress(address), // these are now all initializations

thePhones(phones),

numTimesConsulted(0)

{} // the ctor body is now empty

The assignment-based version first called default constructors to initialize theName, theAddress, and thePhones, then promptly assigned new values on top of the default-constructed ones. All the work performed in those default constructions was therefore wasted. The member initialization list approach avoids that problem, because the arguments in the initialization list are used as constructor arguments for the various data members. In this case, theName is copy-constructed from name, theAddress is copy-constructed from address, and thePhones is copy-constructed from phones.

if ABEntry had a constructor taking no parameters, it could be implemented like this:

ABEntry::ABEntry()

: theName(), // call theName’s default ctor;

theAddress(), // do the same for theAddress;

thePhones(), // and for thePhones;

numTimesConsulted(0) {} // but explicitly initialize numTimesConsulted to zero

Sometimes the initialization list *must* be used, even for built-in types. For example, data members that are const or are references must be initialized; they can’t be assigned.

Base classes are initialized before derived classes , and within a class, data members are initialized in the order in which they are declared. In ABEntry, for example, theName will always be initialized first, theAddress second, thePhones third, and numTimesConsulted last. This is true even if they are listed in a different order on the member initialization list.

*Initialization of static object & data members.*

Static objects inside functions are known as *local static objects* (because they’re local to a function), and the other kinds of static objects are known as *non-local static objects*(i.e., an object that’s global, at namespace scope, or static in a class or at file scope)

Static objects are destroyed when the program exits, i.e., their destructors are called when main finishes executing.

Suppose there is at least two separately compiled source files, each of which contains at least one nonlocal static object. And the actual problem is this:

if initialization of a non-local static object in one translation unit uses a non-local static object in a different translation unit, the object it uses could be uninitialized, because *the relative order of initialization of nonlocal static objects defined in different translation units is undefined*.

**Before proceeding need to understand the extern keyword role.**

The extern storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern', the variable cannot be initialized however, it points the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function, which will also be used in other files, then extern will be used in another file to provide the reference of defined variable or function. Just for understanding, extern is used to declare a global variable or function in another file.

The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

**First File: main.c**

#include <stdio.h>

int count;

extern void write\_extern(); //Here only declaration, definition is somewhere else

main() {

count = 5; write\_extern();

}

**Second File: support.c**

extern int count; //Here only declaration, definition is somewhere else

void write\_extern(void) {

printf("count is %d\n", count);

}

Here, extern is being used to declare count in the second file, where as it has its definition in the first file, main.c. Now, compile these two files as follows −

$gcc main.c support.c

It will produce the executable program a.out. When this program is executed, it produces the following result −

count is 5

We want to create a special object at global or namespace scope representing the single file system

class FileSystem

{ // from your library’s header file

public:

...

std::size\_t numDisks() const; // one of many member functions

...

};

extern FileSystem tfs; // declare object for clients to use (tfs =“the file

// system” ) definition is in some .cpp file in

// our library

A FileSystem object is decidedly non-trivial, so use of the tfs object before it has been constructed would be disastrous.

Now suppose some client creates a class for directories in a file system. Naturally, their class uses the tfs object:

class Directory { // created by library client

public:

Directory ( params );

...

};

Directory::Directory( params )

{...

std::size\_t disks = tfs.numDisks(); // use the tfs object

...

}

Directory tempDir( params ); // directory for temporary files

Now the importance of initialization order becomes apparent:

tfs is initialized before tempDir, because tempDir’s constructor will attempt to use tfs. But tfs and tempDir were created by different people at different times in different source files — they’re non-local static objects defined in different translation units. How can you be sure that tfs will be initialized before tempDir?

Determining the “proper” order in which to initialize non-local static objects is hard. Very hard. Unsolvable hard.

**Solution**:

Fortunately, a small design pattern (Singleton) change eliminates the problem entirely. All that must be done is to move each non-local static object into its own function, where it’s declared static. These functions return references to the objects they contain. Clients then call the functions instead of referring to the objects. *In other words, non-local static objects are replaced with local static objects.*

C++’s guarantee that local static objects are initialized when the object’s definition is first encountered during a call to that function.

So if you replace direct accesses to non-local static objects with calls to functions that return references to local static objects, you’re guaranteed that the references you get back will refer to initialized objects.

As a bonus, if you never call a function emulating a non-local static object, you never incur the cost of constructing and destructing the object, something that can’t be said for true non-local static objects.

class FileSystem { ... }; // as before

FileSystem& tfs() // this replaces the tfs object; it could be

{ // static in the FileSystem class

static FileSystem fs; // define and initialize a local static object

return fs; // return a reference to it

}

class Directory { ... }; // as before

Directory::Directory( params ) // as before, except references to tfs are

{ // now to tfs()

...

std::size\_t disks = tfs().numDisks();

...

}

Directory& tempDir() // this replaces the tempDir object; it

{ // could be static in the Directory class

static Directory td( params ); // define/initialize local static object

return td; // return reference to it

}

now client refer to tfs() and tempDir() instead of tfs and tempDir. These functions are returning references to objects instead of using the objects themselves.

**Things to Remember**

✦ Manually initialize objects of built-in type, because C++ only sometimes initializes

them itself.

✦ In a constructor, prefer use of the member initialization list to assignment inside the

body of the constructor. List data members in the initialization list in the same

order they’re declared in the class.

✦ Avoid initialization order problems across translation units by replacing

non-local static objects with local static objects.