**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; // uppercase names are usually for macros, hence the name

// change

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 you 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

You 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

Constness 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 cont 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 amember function to be const?

There are two prevailing notions:

1. *Bitwise constness* (also known as *physical constness*) and
2. *logical 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.

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.**