Item3: Use const whenever possible

Compiler makes sure the constraints are not violated

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

For what’s pointed to is constant

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

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

Consider:

std::vector<int> vec;

const std::vector<int>::iterator iter = // iter acts like a T\* const

vec.begin();

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

++iter; // error! iter is const

std::vector<int>::const\_iterator cIter = // cIter acts like a const T\*

vec.begin();

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

++cIter; // fine, changes cIter

Generally, having a function return a constant value is inappropriate, but sometimes can reduce error. Ex.

class Rational { ... };

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

Rational a, b, c;

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

Unless you need to modify a parameter or local object, be sure to declare const parameters. Const member function🡺make the interface of a class easier to understand, possible to work with const objects

Note: const member function 🡺 not allow them to modify the object

class T{

int value;

public:

int getValue() const{return value;}

};

Const functions can be called in any type of object. Non-const functions can be only called by non-const objects.

Example:

class TextBlock {

public:

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

{ return text[position]; }

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

{ return text[position]; }

private:

std::string text;

};

TextBlock tb("Hello");

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

const TextBlock ctb("World");

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

void print(const TextBlock& ctb) // in this function, ctb is const

{

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

}

tb[0] = ’x’; // fine — writing a non-const TextBlock

ctb[0] = ’x’; // error! — writing a const TextBlock

🡺return type of the non-const[] is called 🡺{return text[0]=’x’;} call by reference

If [] return simple char, statement won’t compile🡺 illegal to modify return value of a function that returns a built-in type(int/ float/double/char/void/bool)

Bitwise constness: a member function is const **<==>** it doesn’t modify the object’s data member(excluding static). Counterintuitive example of bitwise constness:

class CTextBlock {

public:

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

{ return pText[position]; } // declaration of operator[]

private:

char \*pText;

};

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”

🡺 logical constness: const member function might modify some of the bits in the object. Example:

class CTextBlock {

public:

std::size\_t length() const;

private:

char \*pText;

mutable std::size\_t textLength; // last calculated length of textblock

mutable bool lengthIsValid; // whether length is currently valid

};

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

{

if (!lengthIsValid) {

textLength = std::strlen(pText); // error! can’t assign to textLength

lengthIsValid = true; // and lengthIsValid in a const

} // member function

return textLength;

}

🡺 not bitwise const(textLength and lengthIsValid may be modified), compilers disagree 🡺 solution: mutable, mutable frees non-static data member from the constraints of bitwise constness

In the case, the const version of operator[] does exactly what non-const version does🡺 cast, having non-const operator[] call the const version is a safe way to avoid code duplication. Example:

class TextBlock {

public:

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

…..

return text[position];

}

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

return const\_cast<char&>( // cast away const on op[]’s return type;

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

);

// add const to \*this’s type, and call const version of op[]

}

};

1. static cast: non-const to const🡺 add const to \*this(our call to operator[] will call the const version)

2.const cast: const to non-const🡺 remove const from const operator[]’s return value

Having a const version to call a non-const one is wrong: the object could be changed;

Non-const function can do whatever it wants

Conclusion1: Declare sth. const help compilers detect usage errors

Conclusion2: compiler enforce bitwise constness, but you should program using logical constness

Conclusion3: code duplication between const and non-const can be avoided by having the non-const version call the const version