C++ offers the

notion of a symbolic constant, that is, a named object to which you can’t give a new value after it has been initialized

constexpr double pi = 3.14159;

constexpr int max = 17; *// a literal is a constant expression*

int val = 19;

max+2 *// a constant expression (a const int plus a literal)*

val+2 *// not a constant expression: it uses a variable*

***A constexpr symbolic constant must be given a value that is known at compile time***

constexpr int max = 100;

void use(int n)

{

constexpr int c1 = max+7; // OK: c1 is 107

constexpr int c2 = n+7;// error: we don’t know the value of c2

}

**To handle cases where the value of a “variable” that is initialized with a value that is not known at compile time but never changes after initialization, C++ offers a second form of constant (a const):**

constexpr int max = 100;

void use(int n)

{

constexpr int c1 = max+7; **// OK: c1 is 107**

const int c2 = n+7; **// OK, but don’t try to change the value of c2**

c2 = 7; **// error: c2 is a const**

}

Note that a<b<c means (a<b)<c and that a<b evaluates to a Boolean value: true or false. So, a<b<c will be equivalent to

either true<c or false<c.

Use **++a** for increment

Please note that this is a logical argument about readability and correctness, not an argument about efficiency. Contrary

to popular belief, modern compilers tend to generate exactly the same code from a=a+1 as for **++a** when a is one of the built-

in types. Similarly, we prefer a\*=scale over a=a\*scale.

**Swtich statement**

**Here are some technical details about switch-statements:**

**1. The value on which we switch must be of an integer, char, or enumeration (§9.5) type. In particular, you cannot switch**

**on a string.**

**2. The values in the case labels must be constant expressions (§4.3.1). In particular, you cannot use a variable in a case**

**label.**

**3. You cannot use the same value for two case labels.**

**4. You can use several case labels for a single case.**

**5. Don’t forget to end each case with a break. Unfortunately, the compiler probably won’t warn you if you forget.**

To select based on a string you have to use an if-statement or a map (Chapter 21).

A switch-statement generates optimized code for comparing against a set of constants. For larger sets of constants, this

typically yields more efficient code than a collection of if-statements. However, this means that the case label values must be

constants and distinct.

For loop vs while loop

However, using a for-statement yields more

easily understood and more maintainable code whenever a loop can be defined as a for-statement with a simple initializer,

condition, and increment operation. Use a while-statement only when that’s not the case.

Range-for-loop

vector<int> v = {5, 7, 9, 4, 6, 8};

for (int x : v) // for each x in v

cout << x << '\n';

5.6 Exceptions

The fundamental idea is to separate detection of an error (which should be done in a called function) from the handling of an error (which should be done in the calling function) while ensuring that a detected error cannot be ignored;

Note that we used cerr rather than cout for our error output: cerr is exactly like cout except that it is meant for error

output. By default both cerr and cout write to the screen, but cerr isn’t optimized so it is more resilient to errors, and on some

operating systems it can be diverted to a different target, such as a file. Using cerr also has the simple effect of documenting

that what we write relates to errors. Consequently, we use cerr for error messages.

int main()

try {

// our program

**return 0;**

// 0 indicates success

}

catch (exception& e) {

cerr << "error: " << e.what() << '\n';

keep\_window\_open();

**return 1;**

// 1 indicates failure

}

catch (...) {

cerr << "Oops: unknown exception!\n";

keep\_window\_open();

**return 2;**

// 2 indicates failure

}

Note again that the return value from main() is passed to “the system” that invoked the program. Some systems (such as

Unix) often use that value, whereas others (such as Windows) typically ignore it. A zero indicates successful completion and a

nonzero return value from main() indicates some sort of failure.

Chapter 8

You can not overload function declarations that differ only by return type.

We have a preference for the { } initializer syntax. It is the most general and it most explicitly says “initializer.” We tend to

use it except for very simple initializations, where we sometimes use = out of old habits, and ( ) for specifying the number of

elements of a vector

int my\_find(vector<string> vs, string s, **int**)

{

for (int i = 0; i<vs.size(); ++i)

if (vs[i]==s) return i;

return –1;

}

// 3rd argument unused

You can find the complete grammar for function definitions in the ISO C++ standard

void g(int a, int& r, const int& cr)

{

++a;

// change the local a

++r;

// change the object referred to by r

int x = cr;

// read the object referred to by cr

}

int main()

{

int x = 0;

int y = 0;

int z = 0;

g(x,y,z);

g(1,2,3);

g(1,y,3);

// x==0; y==1; z==0

// error: reference argument r needs a variable to refer to

// OK: since cr is const we can pass a literal

}

So, if you want to change the value of an object passed by reference, you have to pass an object. Technically, the integer literal

2 is just a value (an rvalue), rather than an object holding a value. What you need for g()’s argument r is an lvalue, that is,

something that could appear on the left-hand side of an assignment.

**Note that a const reference doesn’t need an lvalue. It can perform conversions exactly as initialization or pass-by-value.**

**Basically, what happens in that last call, g(1,y,3), is that the compiler sets aside an int for g()’s argument cr to refer to:**

**Click here to view code image**

**g(1,y,3);**

**// means: int\_\_compiler\_generated = 3; g(1,y,\_\_compiler\_generated)**

If you really mean to truncate a double value to an int, say so explicitly:

Click here to view code image

void ggg(double x)

{

int x1 = x;

int x2 = int(x);

int x3 = static\_cast<int>(x);

// truncate x

// very explicit conversion (§17.8)

ff(x1);

ff(x2);

ff(x3);

ff(x);

ff(int(x));

**ff(static\_cast<int>(x));**

// truncate x

// very explicit conversion (§17.8)

}

That way, the next programmer to look at this code can see that you thought about the problem.

Click here to view code image

constexpr double xscale = 10;

constexpr double yscale = 0.8;

// scaling factors

constexpr Point scale(Point p) { return {xscale\*p.x,yscale\*p.y}; };

Assume that Point is a simple struct with members x and y representing 2D coordinates. Now, when we give scale() a

Point argument, it returns a Point with coordinates scaled according to the factors xscale and yscale. For example:

Click here to view code imagevoid user(Point p1)

{

Point p2 {10,10};

Point p3 = scale(p1);

Point p4 = scale(p2);

// OK: p3 == {100,8}; run-time evaluation is fine

// p4 == {100,8}

constexpr Point p5 = scale(p1); // error: scale (p1) is not a constant

// expression

constexpr Point p6 = scale(p2); // p6 == {100,8}

// . . .

}

A constexpr function behaves just like an ordinary function until you use it where a constant is needed. Then, it is calculated

at compile time provided its arguments are constant expressions (e.g., p2) and gives an error if they are not (e.g., p1). To

enable that, a constexpr function must be so simple that the compiler (every standard-conforming compiler) can evaluate it. In

C++11, that means that a constexpr function must have a body consisting of a single return-statement (like scale()); in

C++14, we can also write simple loops. A constexpr function may not have side effects; that is, it may not change the value

of variables outside its own body, except those it is assigned to or uses to initialize.

class Date {

public:

Date(int yy, int mm, int dd)

:y{yy}, m{mm}, d{dd}

{

// . . .

}

void add\_day(int n)

{

// . . .

}

int month() { return m; }

// . . .

private:

int y, m, d;

};

// year, month, day

The first thing we notice is that the class declaration became larger and “messier.” In this example, the code for the constructor

and add\_day() could be a dozen or more lines each. This makes the class declaration several times larger and makes it harder

to find the interface among the implementation details. Consequently, we don’t define large functions within a class

declaration.

However, look at the definition of month(). That’s straightforward and shorter than the version that places

Date::month() out of the class declaration. For such short, simple functions, we might consider writing the definition right in

the class declaration.

**Note that month() can refer to m even though m is defined after (below) month(). A member can refer to a function or**

**data member of its class independently of where in the class that other member is declared. The rule that a name must be**

**declared before it is used is relaxed within the limited scope of a class.**

Writing the definition of a member function within the class definition has three effects:

• The function will be inline; that is, the compiler will try to generate code for the function at each point of call rather than

using function-call instructions to use common code. This can be a significant performance advantage for functions, such

as month(), that hardly do anything but are used a lot.

• All uses of the class will have to be recompiled whenever we make a change to the body of an inlined function. If the

function body is out of the class declaration, recompilation of users is needed only when the class declaration is itself

changed. Not recompiling when the body is changed can be a huge advantage in large programs.

• The class definition gets larger. Consequently, it can be harder to find the members among the member function

definitions.

enum class Month {

jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec

};

The “body” of an enumeration is simply a list of its enumerators. The class in enum class means that the enumerators are in

the scope of the enumeration. That is, to refer to jan, we have to say Month::jan.

In addition to the enum classes, also known as scoped enumerations, there are “plain” enumerations that differ from scoped

enumerations by implicitly “exporting” their enumerators to the scope of the enumeration and allowing implicit conversions to int

CHAPTER 8

The order of evaluation of sub-expressions is governed by rules designed to please an optimizer rather than to make life simple

for the programmer. That’s unfortunate, but you should avoid complicated expressions anyway, and there is a simple rule that

can keep you out of trouble: if you change the value of a variable in an expression, don’t read or write it twice in that same

expression. For example:

Click here to view code image

v[i] = ++i;

v[++i] = i;

int x = ++i + ++i;

cout << ++i << ' ' << i << '\n';

f(++i,++i);

// don’t: undefined order of evaluation

// don’t: undefined order of evaluation

// don’t: undefined order of evaluation

// don’t: undefined order of evaluation

// don’t: undefined order of evaluation

Note in particular that = (assignment) is considered just another operator in an expression, so there is no guarantee that the

left-hand side of an assignment is evaluated before the right-hand side. That’s why v[++i] = i is undefined.