C++ Programming Principles and Practice

In C++, any class object can be thrown as exception. This is valid code

class Bad\_area {};

int area(int length, int width)

{

if (length < 0 || width < 0)

throw Bad\_area();

return length \* width;

}

If we try to access an element outside Vector's bounds, *out\_of\_range* error is thrown.

try {

vector<int> vi;

for(size\_t i=0;i<5;++i)

vi.push\_back(i+1);

for(size\_t i=0;i<=vi.size();++i)

cout << vi[i] << endl;

}

catch(out\_of\_range &error) {

cerr << "Oops, out of range error!" << endl;

return 1;

}

If we detect an error condition at runtime and want to throw an exception, *runtime\_error* type of exception can be thrown.

void error(string s) {

throw runtime\_exception(s);

}

It takes a string argumentm, which can be extracted in catch using *e.what()*.

Note that *out\_of\_range* is NOT a *runtime\_error*.

All subclasses of exceptions, irrespective of type can be caught using

catch (exception &e) { /\* do something\*/ }

To catch anything that is thrown, use

catch (...) { /\* catch all\*/ }

During assignment of an object to another, constructor is not called. Behind the scenes, some compiler defined assignment function may be called, but I am not sure right now.

Token t('8',100);

Token t2 = t;

Constructor is called only in first statement.

In C++, a constructur cannot be called from another like a method. The following is invalid.

Token (char ch) {

Token(ch,0);

}

This compiles, but it creates 2 Token objects instead, which is not at all desired behavior.

With C++11, however, following is valid

Token (char ch): Token (ch, 0) { }

ASIDE:

Using make utility to compile programs on command line is very simple. Use *make calc1* to compile a file named as calc1.cpp with default options.

If you want to compile all files to use C++11 standard, create Makefile in the top level directory and include at the top

CPPFLAGS=-std=c++11 -Wall

-Wall is recommended as it shows all warnings. -Werror treats all warnings as errors.

In C++11, there's a new universal sytax to initialize variables using {}

int i {20};

This initializer gives a warning/error on narrowing conversions

int i {2.3}; // generates a compiler warning on g++

Similarly, *int(3.5)* is allowed for casting 3.5 to int, but *int{3.5}* will generate warning/error as this is a narrowing down conversion.

In C++11, *constexpr* was introduced. It is used to initialize constants whose value is known at compile time. Needless to say, once initilized, it can't be changed.

*const* on the other hand, can be initialized using dynamic values, but can't be changed once initialized.

Following code is valid

constexpr int max = 10;

constexpr x = max + 2;

For initializing x, any value

But following is not valid

int i;

cin >> i;

const int max = i;

constexpr int x = max+2; // error, value of max not known at compile time

cout << x << endl;

*constexpr* values can be used even as switch case labels.

A vector can now be initialized directly using an initializer list, like an array (in C++11)

*vector<int> v = {1,2,3,4};*

vector can be defined with an initial size. All values are given default values according to the type.

*vector<int> v(6); // all 6 elements are 0.*

An int can be initialized with default value as follows

*int i {}; // i = 0*

C++11 has a for-each type of loop to loop through elements of vector.

vector<int> v{1,2,3,4};

for(int i:v)

cout << i << endl;

This is also called *range-for-loop* for some unknown reason.

If any variables are declared inside switch statement, the switch case needs to be enclosed in braces.

It is possible to put a character already read from stream back on to stream. I do not know if all streams support it, but *cin* does

cin.putback(ch)

To read all characters, including spaces and newline, use *cin.get()* function. Using *cin* with *>>* operator skips all whitespace charaters.

It is possible to lie using putback function, i.e., we can putback a character that wasn't originally read from stream. There is another function called *unget()* that doesn't take any paramter and puts back last read character from stream back to stream. Like putback, it can place back unlimited number of characters on stream.

Consider the function below

int incr (int &x) {

return ++x;

}

This function cannot be called using a const argument, or an integer literal.

int i = incr(10); // error, as 10 is a literal

Always have a good reason for writing a function that takes a non-const reference argument, as its usage will be limited to passing non-const references. Use pass by non-const reference only when you are sure that the function will modify the argument.

void f(T x); //(1)

f(y); //(2)

T x = y; //(3)

Call (2) is legal only when initialization in (3) is also legal. Both x's get the same value in each case.

**constexpr functions**

1. It has to return something, can't return void.

2. The only statement it can have is a return statement in function body.

3. A constexpr function can only call other constexpr functions from within its function body.

4. The result of a constexpr function can be assigned to a constexpr variable.

5. A constexpr function can take in a non-constexpr argument, but then the returned value is also NOT constexpr. It is an error to assign such a return value to a constexpr variable.

6. For a constexpr function to return a constexpr value, all the arguments must also be constexpr.

7. constexpr functions can be evaluated at compile time. This means that the result of their evaluation may be available at compile time to the program. This is ensured when the result is assigned to a constexpr variable.

Defining a constexpr function:

constexpr double square(double x) { return x\*x; }

Calling it:

void func(double d) {

double x = square(d); // OK, as x is not constexpr

constexpr double y = square(d); // error, as d is not constexpr

constexpr double p = 10;

constexpr double z = square(p); // OK, as p is constexpr

}

If the compiler determines that the function is not “simple enough”, it will not allow you to declare it as a constexpr function.

Another reason why globals are bad.

File 1:

int y = 1;

File 2:

extern int y;

int z = y+2;

There is no defined order in which globals are initialized. Z can be 2 or 3 depending on which file is initialized first. Avoid global and be very suspicious of any global that is initilized with non-constexpr.

A good way to define a global constant from a non-trivial initializer:

const Date& default\_date() {

static const Date date(1970,1,1);

return date;

}

This creates the date object only once and returns a const reference.

It seems that it is okay to return a const reference and that might mean it is not allocated on local stack of the function. Maybe this will clear up later why it is OK to return const reference to a local variable.

With C++11, {} syntax is preferable for initialization.

We can use = optionally before {..}

Date next {2014, 8, 3};

Date next = {2104, 8, 3};

These are equivalent. If it involves any explicit constructors etc or some other vodoo, i'll see that later.

Best way to initialize member variables in a class is to use member intializer list

Date::Date(int y, int m, int d): year{y}, month{m}, date{d}

Other way could have been

Date::Date(int y, int m, int d) {

year = y;

month = m;

date = d;

}

But this is same as default initializing the numbers and then assigning values to them. And there is a risk of using them before initialization.

Enums should be used using *enum class* syntax, as it provides scoping for enum constants.