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.

IDEA: Since it is static, it was never allocated on local stack. Does it work for non-static members too?

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.

Month m = Month::feb;

Month m1 = 10; // error, can't assign using integer

int n = m; // error, can't assign Month type to int

To get underlying integer value, use int(m).

Forcing an assign is possible using

Month m1(100);

Enums can be used with >, < and other comparison operators also.

Enum can’t be printed directly to a stream. Either override << operator, or convert to int before inserting on ostream.

Operator overloading

1. Only built-in operators can be overloaded and their cardinality can't be changed.

2. At least one of the operands

If a class does not initialize a member variable explicitly, it is intialized according to its default constructor. This is not same as being unintialized.

An uninitialized int has garbage value, however, a int initialized via default constructor {} has a value of 0. Similarly double is 0.0

When any object is passed to a function as a constant reference, only const methods of that object can be called on it within the function to which its reference is passed.

When initializing variables in initializer list of constructor, use same order as they appear in class definition. This is because initialization actually happens in the order in which members are declared in class definition. This is just to remind that if user is trying to initialize on variable from another (NOT recommended), it may not work as expected.

**Argument Dependant Lookup (ADL)**

This is hairy. Suppose you define your class *Date* in a namespace called *chrono.* You want to override << operator to provide a printable representation of Date objects. You write the function to override it within your namespace. Ideally, if you call this function (basically *operator<<*) from outside the chrono namespace, the function name has to be qualified with chrono::. But this is NOT always required. What happens is that since we are calling this overloaded operator from a different namespace (global or std, whatever), the compiler looks for the overload in current namespace and global namespace. If it can't find the function, it looks for the function in the namespace in which arguments to the function are defined. If it finds the function there, it uses that function.

So basically, this works not only for overloaded functions, but for any functions.

When a file stream goes out of scope, its associated file is closed. It is not ncessary to call *stram.close()*, as it is called implicitly when the scope ends.

Similarly, defining a stream with file name opens the file for reading or writing. It is not necessary to call *stream.open()*.

To reuse a stream with a different file, it is necessary to close it first.

**I/O error handling**

*good()* - Operation succeded. *if(cin.good())* is equivalent to *if(cin)*

*eof()* - No more input.

*bad()* - when something really bad happens, like device failed or read error. Generally non-recoverable. Bail out.

*fail()* - Something unexpected happened. Like expecting an int but found a char. This is usually recoverable situation. Use *cin.clear()* to reset the stream state to good.

Always check for *bad()* before *fail()*, as fail() returns true even when badbit is set.

When stream fails, i.e., failbit is set, the character which caused stream to fail is not taken off from the stream. You can clear the stream state and try to read it again to a char.

*ist.clear()* - clears the stream state flags, which means the stream is good now.

*ist.clear(ios\_base::failbit)* – passing in an rgument here means that all state is cleared, and the the passed in argument is set. That mean whatever the state was before, now only failbit is set on stream.

*ist.exceptions()* - this returns the current exception mask of the stream.

*ist.exceptions(iostate except)* – this sets the exception mask of the stream. That means the exception is thrown when any of the state specified in the mask is set.

*ist.exceptions (ist.exceptions() | ios\_base::badbit);*

After this statement, if the stream is set to *bad* state anytime, ist will throw *ios\_base::failure* exception defined in standard library.

An *ostream* can be tested for all the same er rors as istream.

Whenever a file is opened, check the associated stream for goodness before starting to work on it.

A class member variables can be intialized whenever they are declared.

class X {

int i = 10;

};

This is valid code. But this is added in C++11, and there are non-obvious rules at play here.

Below is similar code that gives error

class Day {

vector<double> hours1{vector<double>(24, not\_a\_reading)}; // not error

vector<double> hours2(24,-1); // error

};

Thus, until rules are clear, prefer not to intialize at declaration.

What looks like is that class member is allowed to be initialized via new constructor syntax, which uses { }, or using explicit assignment operator ( = ), but not via old constructor syntax.

Okay, that was stupid. *vector<double> hours2(24,-1)* looks like a function call made wrong. That's why it doesnt compile.

However,

vector<double> hours2{24, -1}

compiles fine. But, this creates a vector with 2 doubles, 24 and -1, which isn't what we want. So, {} can be used for copy constructor and initialization, or for invoking constructor. In this case, it could mean either initialization or invoking constructor, but it is treated as initialization list.

After much more thought, I think this is a tricky area and not very well thought out behavior.

*vector<double> hours2(24,-1)*  is still allowed inside a function, so it's not at all clear why it can't be allowed for class member initialization. Since there are no parameter types specified, it should be clear that we are trying to create an object, and not declare a function.

One more interesting fact. Default constructor cannot be called explicitly. i.e.,

*string s();* is always a function declaration, it should never be confused with the default constructor invocation.

**Stream Manipulation**

cout << 15 << hex << “ “ << 15;

This prints *15 f*

hex is a modifier which changes the stream. It is also called *stream manipulator*. The change is sticky. It persists until it is changed back using *dec* manipulator.

*showbase* manipulator asks the stream to show bases of integer values. Bases are indicated using leading 0x or 0.

cout << 1234 << "\t" << hex << 1234 << "\t" << oct << 1234 << endl;

cout << showbase << dec;

cout << 1234 << "\t" << hex << 1234 << "\t" << oct << 1234 << endl;

Output:

1234 4d2 2322

1234 0x4d2 02322

*noshowbase* resets it back.

Similarly, if *hex* manipulator is used while reading in ints, stream can read in hexadecimals directly (without leading 0x or 0). Any integers read are interpreted as hexadecimal ints. So, *fe34* is a valid input with this setting.

Now, cin treats all inputs as decimal base by default. *dec* manipulator is always set by default. If we set *hex* manip on cin, it starts treating input as hexadecimal by default. Let's say we want to treat every input as it is. i.e., if 0x is prefixed, it should be hexadecimal, if 0 is prefixed, it should be octal, and with no prefix, input should be treated as decimal.

This requires us to clear all defaults.

cin.unsetf(ios::dec);

cin.unsetf(ios::hex);

cin.unsetf(ios::oct);

This will do the job. Since any set mask is removed, we can write code like

cin >> num;

cout << num << endl;

And we know that if we enter 0x12, it will be treated as 18 (decimal value) and so on.

Floating point output

By default, there is no stream manipulator set for floating point output. This setting too has a peculiar behavior. The number is printed using 6 digits, and either fixed or scientific notation is chosen to represent the number in best possible way. Assume default is set for below example

cout << "\t" << 1.1 << endl // 1.1

<< "\t" << 123.456 << endl // 123.456

<< "\t" << 1234.5678 << endl // 1234.57

<< "\t" << 12345678.912 << endl; // 1.23457e+07

In first 2 cases, numbers can be expressed in 6 or less digits, so it is printed exactly as provided.

In next case, number has greater than 6 digits, so number is rounded off to 6 digit precision for display.

In last case, there are more than 6 digits to left of decimal point, hence it is not possible to display the number using 6 digits without scientific notation. If cout had written 12345600, it would still have 8 digits of precision (0s to the left to decimal are significant), and that would be wrong. Hence, it is displayed in scientific format.

Fixed format

In fixed format, numbers are displayed with a fixed number of digits after decimal point. By default, this number is 6. Therefore, all floating point numbers have exactly 6 digits after decimal point

cout << "\t" << 1.0 << endl // 1.000000

<< "\t" << 1.23 << endl // 1.230000

<< "\t" << 123.456 << endl // 123.456000

<< "\t" << 1234.5678 << endl // 1234.567800

<< "\t" << 12345678.912 << endl; // 12345678.912000

Number of digits after decimal point may be changed. More on this later.

Scientific format

cout << "\t" << 1.0 << endl // 1.000000e+00

<< "\t" << 1.23 << endl // 1.230000e+00

<< "\t" << 123.456 << endl // 1.234560e+02

<< "\t" << 1234.5678 << endl // 1.234568e+03

<< "\t" << 12345678.912 << endl; // 1.234567e+07

Fixed number of digits after decimal point. This can be changed.

Default precision is 6 for all 3 formats above. It can be changed by using *setprecision()* method.

With C++11, we also have a *defaultfloat* manipulator. It has the same effect as *cout.unsetf(ios\_base::floatfield)*, which removes both fixed and scientific flags from *fmtflags* (format flags).

Fields

Output field width can be set using *setw()* modifier. It is not sticky. It has to be set just before output.

It pads the output with space to the left. It works for integers, floats, chars and even strings.

In case output value has greater length than width, it is not truncated.

Reading and writing binary files

ifstream ifs("input.pdf", ios\_base::binary);

if(!ifs) {

cerr << "Could not read input file as binary" << endl;

return 0;

}

vector<int> v; // read input bytes

int x;

while(ifs.read(as\_bytes(x), sizeof(x))) {

v.push\_back(x);

}

ofstream ofs("output.pdf", ios\_base::binary);

if(!ofs) {

cerr << "Could not create output binary file" << endl;

return 0;

}

size\_t i = 0;

while(i < v.size() &&

ofs.write(as\_bytes(v[i++]), sizeof(x)))

;

Bytes are read into a byte buffer using ifs.read() method.

as\_bytes has been implemented in std\_lib\_failities.h as follows

template<class T> char\* as\_bytes(T& i) // needed for binary I/O

{

void\* addr = &i; // get the address of the first byte

// of memory used to store the object

return static\_cast<char\*>(addr); // treat that memory as bytes

}

This just returns the pointer to memory as bytes.