My C++

Cheat Sheat

# Clear the input buffer

cin.clear();

# Ternary operator

m = (m==3) ? 1 : 2; // “wrap around”

# Operator overloading

std::ostream& operator<<(std::ostream& os, const myclass& obj)

{

os << obj.somevalue;

return os;

}

ostream& operator<<(ostream& os, const Da te& d)

Month **operator++(**Month& m) // prefix increment operator

{

m = (m==Dec) ? Jan : Month(int(m)+1); // “wrap around”

return m;

}

bool operator==(const Name\_pairs& np1, const Name\_pairs& np2);

ifstream& operator>>(ifstream& is, Reading& r)

An overloaded operator must have at least one user-defined type as operand:

int opera tor+(int,int); // **error***: y***o***u* **ca***n’***t overload b***u***ilt***-***i***n +*

Vector opera tor+(const Vector&, const Vector &); // *OK*

Vector opera tor+=(const Vector&, int);

# Initialization

int a ; // no initialize r

double d = 7; // initialize r using the = syntax

vector<int> vi(10); // initialize r using the ( ) syntax

vector<int> vi2 {1,2,3,4}; // initialize r using the { } syntax

# Type converion

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

# Bracket initization

signed char c[1] { -128 };

# Type casting

int x1 = narrow\_cast<int>(2.9);

if(x!=int(x)) error("Not an int.");

# Checking vector indexes for error

in case you want it to throw an exception, use std::vector::at1 instead of operator[]

try {

agrid.at(-1) = 5;

}

catch (const std::out\_of\_range& e) {

cout << "Out of Range error.";

}

# Iterate through all array elements

for(int i : vi)

cout << "i = " << i << endl;

# (member) initializer list.

Da te::Da te(int yy, int mm, int dd) // constructor

:y{yy}, m{mm}, d{dd} // note : membe r initialize rs

{

}

# member function bodies in the class declaration

The obvious rule of thumb is: Don’t put member function bodies in the class declaration unless you

know that you need the performance boost from inlining tiny functions. Large functions, say five or

more lines of code, don’t benefit from inlining and make a class declaration harder to read. We rarely

inline a function that consists of more than one or two expressions.

cla ss Da te {

public:

void a dd\_day(int n)

{

// . . .

}

# Clearing incorrect cin input

else if (!cin.eof()) {

cin.clear();

cout << "invalid character ('" << char(cin.get()) << "') ignored - try again\n";

}

# Class Constructor

class Date {

public:

Date(); // de fault constructor

Date(yea r, Month, da y);

Date(int y); // January 1 of year y

priva te:

int y {2001};

Month m {Month::ja n};

int d {1};

};

struct Date {

int y, m, d;

Date(int y, int m, int d); // check for valid date and initialize

};

Date last {2000,12,31};

Date last = {2000,12,31};

Date::Date(int yy, int mm, int dd)

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

{

}

class Yea r { // *y***e ar i***n* **[mi***n:***max) ra***ng***e**

static const int min = 1800;

static const int ma x = 2200;

public:

class Invalid { };

**Year(int x) : y{x} { if (x<min || ma x<=x) throw Invalid{}; }**

Date::Date()

**:y{2001}, m{Month::jan}, d{1}**

{

}

# ENUM

enum class Month {

jan = 1, feb = 2, mar = 3, apr = 4, may = 5, jun = 6,

jul = 7, aug = 8, sep = 9, oct = 10, nov = 11, dec = 12

};

Month m2 = feb; // **e rror***:* **fe b is** *n***ot i***n* **sco***p***e**

m = 7; // **e rror***:* **ca***n’***t assi***gn* **a***n* **i***n***t to a Mo***n***th**

int n = m; // **e rror***:* **ca***n’***t assi***gn* **a Mo***n***th to a***n* **i***n***t**

Month mm = Month(7); // **co***n***ve rt i***n***t to Mo***n***th (***un***che cke d)**

# Plain ENUM

Less strict

enum Month {

jan = 1, feb = 2, mar = 3, apr = 4, may = 5, jun = 6,

jul = 7, aug = 8, sep = 9, oct = 10, nov = 11, dec = 12

};

Month m = feb; // *OK:* **feb i***n* **sco***p***e**

Month m2 = Month::feb; // **also** *OK*

m = 7; // **e rror***:* **ca***n’***t assi***gn* **a***n* **i***n***t to a Mo***n***th**

int n = m; // *OK:* **we ca***n* **assi***gn* **a Mo***n***th to a***n* **i***n***t**

Month mm = Month(7); // **co***n***ve rt i***n***t to Mo***n***th (***un***che cke d)**

# String split

#include <sstream>

std::stringstream isbn\_stream(isbn);

std::vector<std::string> seglist;

std::string segment;

while (std::getline(isbn\_stream, segment, '-'))

seglist.push\_back(segment);

# Opening a file

If you want to read from a file or write to a file you have to open a stream specifically for that file. An ifstream is an istream for reading from a file, an ofstream is an ostream for writing to a file, and an fstream is an iostream that can be used for both reading and writing. Before a file stream can be used it must be attached to a file.

struct Rea ding {

int hour;

double temperature;

};

intma in()

{

cout << "Plea se enter input file name: ";

string iname;

cin >> iname;

ifstream ist {iname};

if (!ist) error("can't open input file ",iname);

string oname;

cout << "Please enter name of output file: ";

cin >> oname;

ofstream ost {oname};

if (!ost) error("ca n't open output file ",oname);

vector<Rea ding> temps;

int hour;

double temperature;

while (ist >> hour >> temperature) {

if (hour < 0 || 23 <hour) error("hour out of range");

temps.push\_ba ck(Rea ding{hour,temperature});

}

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

ost << '(' << temps[i].hour << ','

<< temps[i].tempera ture << ")\n";

}

# Set stream to throw exception if bad

// make ist throw if it goes bad

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

The notation may seem odd, but the effect is simply that from that statement onward, ist will throw

the standard library exception ios\_base::failure if it goes bad().

# For each

for (auto &element : container)

std::for\_each(m\_attack.begin(), m\_attack.end(),

[](Attack \* attack)

for (auto attack = m\_attack.begin(); attack != m\_attack.end(); ++attack)

# Rand int

#include <time.h>

srand (time(NULL));

/\* generate secret number between 1 and 10: \*/

iSecret = rand() % 10 + 1;

float RandomNumber(float Min, float Max)

{

return ((float(rand()) / float(RAND\_MAX)) \* (Max - Min)) + Min;

}

# Custom sort

sort(data.begin(), data.end(),

[](const Reading& r1, const Reading& r2)

{

return r1.temperature > r2.temperature;

}

);

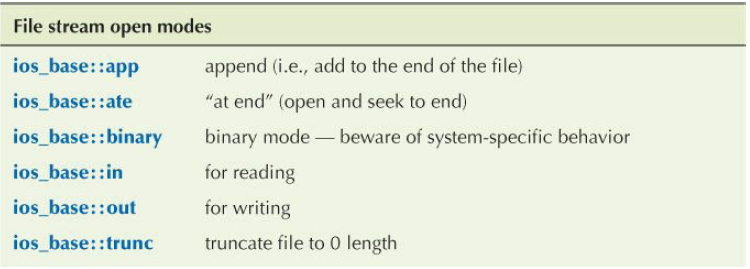
# Number precision

Cout << setprecision(5) - floats – sticks (for entire cout)

#include <iomanip>

Cout << setw(4) - integers and floats – does not stick

# Stream open modes

ofstream ofs {name, ios\_ba se::a pp}; 

fstream fs {"myfile", ios\_ba se::in|ios\_ba se::out}; // **both i***n* **a***n***d o***u***t**

# string stream

string s;

ostringstream os {s};

os << setw(8) << la bel << ": "

<< fixed << setprecision(5) << temp.temp << temp.unit;

someobject.display(Point(100,100), os.str().c\_str()); // TO STRING

# byte read and write

ofstream ofs {oname,ios\_ba se::bina ry};

if (!ofs) error("ca n't open output file ",oname);

// **read from bi***n***ar***y* **file** *:*

for(int x; ifs.read(as\_bytes(x),sizeof(int)); )

// **write to bi***n***ar***y* **file** *:*

for(int x : v)

ofs.write(as\_bytes(x),sizeof(int));

# Disabling class copy

Shape(const Shape&) =delete; // *p***reve***n***t co***py***i***ng*

Shape& opera tor=(const Shape&) = delete;

# Class derivation

struct Circle : Sha pe { /\* *. . .* \*/ };

class Circle : public Shape { public: /\* *. . .* \*/ };

# virtual functions

struct Sha pe {

// *. . .*

virtual void draw\_lines() const;

virtual void move();

// *. . .*

};

void Shape::move() { /\* *. . .* \*/ } // *OK*

virtua l void Sha pe::dra w\_lines() const { /\* *. . .* \*/ } // **e rror**

struct B {

virtua l void f() const { cout << "B::f "; }

void g() const { cout << "B::g "; } // *n***ot virt***u***al**

};

struct D : B {

void f() const { cout << "D::f "; } // **ove rride s** *B::***f**

void g() { cout << "D::g "; }

};

struct DD : D {

void f() { cout << "DD::f "; } // **doe s***n’***t ove rride** *D::***f (***n***ot co***n***st) DIFFERENT TYPE**

void g() const { cout << "DD::g "; }

};

# Class override

struct B {

virtua l void f() const { cout << "B::f "; }

void g() const { cout << "B::g "; } // *n***ot virt***u***al**

};

struct D : B {

void f() const override { cout << "D::f "; } // **ove rride s** *B::***f**

void g() override { cout << "D::g "; } // **e rror***: n***o virt***u***al** *B::g* **to ove rride**

};

struct DD : D {

void f() override { cout << "DD::f "; } // **e rror***:* **doe s***n’***t ove rride**

// *D::***f (***n***ot co***n***st)**

void g() const override { cout << "DD::g "; } // **e rror***: n***o virt***u***al** *D::g*  **to ove rride**

# Abstract class

*define constructor as protected*

*OR*

class B { // **abstract base class**

public:

virtual void f() =0; // *pu***re virt***u***al f***un***ctio***n =0 indicates that it must be overriden*

virtual void g() =0;

};

B b; // **e rror***: B* **is abstract**

cla ss D1 : public B {

public:

void f() override;

void g() override;

};

D1 d1; // *OK*