

# C++ basics cheat sheet

Michele Iarossi\*

October 19, 2021 - Version 1.03 - GNU GPL v3.0

In the following code snippets, the standard I/O library and namespace are always used:

```
#include <iostream>
using namespace std;
```

## Type safety

Universal and uniform initialization prevents narrowing conversions from happening:

```
// safe conversions
double x {54.21};
int a {2342};
```

```
// unsafe conversions
int y {x};
char b {a};
```

## constexpr

constexpr must be known at compile time:

```
constexpr int max = 200;
constexpr int c = max + 2;
```

⇒ const variables don't change at runtime:

```
// cannot be declared as
// constexpr because
// the value of n
// is not known at compile time
const int m = n + 1;
```

## Type casting

```
// int 15 to double 15.0
double num;
num = static_cast<double>(15);
```

## Functions

Function declaration with default trailing arguments

```
// if year is omitted,
// then year = 2000
void setBirthday(int day,
                 int month, int year=2000);
```

## Random numbers

```
#include <cstdlib>
#include <ctime>
```

```
// seed the generator
srand( time(0) );
// integer random number between
// 0 and RAND_MAX
int n = rand();
```

## Arrays

```
// range based for statement
int arr[] = {2, 4, 6, 8};
```

```
for (int& x : arr)
    x++;
// outputs 3579
for (auto x : arr)
    cout << x;
cout << endl;
```

## Pointers

⇒ Simple object:

```
// simple pointer to double
double *d = new double(5.123);
```

```
// delete the storage
// on the freestore
delete d;
```

⇒ Dynamic array:

```
// dynamic array of 10 doubles
double *dd = new double[10];
```

```
// delete the storage
// on the freestore
delete [] dd;
```

⇒ Dynamic matrix:

```
// dynamic matrix of 5 x 5 doubles
// memory allocation
double **m = new double*[5];
for (int i=0; i<5; i++)
    m[i] = new double[5];
```

```
// memory initialization
for (int i=0; i<5; i++)
    for (int j=0; j<5; j++)
        m[i][j] = i*j;
```

```
// memory deallocation
for (int i=0; i<5; i++)
    delete[] m[i];
delete[] m;
```

## C-Strings

```
#include <cstring>
#include <cstdlib>
```

```
// C-string for max 10 characters
// long string + null char '\0'
const int SIZE = 10 + 1;
char msg[SIZE] = "Hello!";
```

```
// correct looping over C-strings
int i = 0;
while ( msg[i] != '\0' && i < SIZE)
{
    // process msg[i]
}
```

```
// safe string copy,
// at most 10 characters are copied
strncpy(msg, srcStr, 10);
```

```
// safe string compare,
// at most 10 characters
// are compared
strncmp(msg, srcStr, 10);
```

```
// safe string concatenation,
// at most 10 characters
// are concatenated
strncat(msg, srcStr, 10);
```

```
// from C-string to int,
// long, float
int n = atoi("567");
long n = atol("1234567");
double n = atof("12.345");
```

## Standard I/O

```
#include <iomanip>
```

```
// set flag
cout.setf(ios::fixed);
// unset flag
cout.unsetf(ios::fixed);
```

```
// set ios::fixed or
// ios::scientific notation
cout.setf(ios::fixed);
cout << fixed;
```

```
// set precision
cout.precision(4);
cout << setprecision(4);
```

```
// set character text width
cout.width(10);
cout << setw(10);
```

```
// set ios::left or
// ios::right alignment
cout.setf(ios::left);
cout << left;
```

```
// always show decimal
// point and zeros
cout.setf(ios::showpoint);
cout << showpoint;
```

```
// always show plus sign
cout.setf(ios::showpos);
cout << showpos;
```

## Character I/O

```
// read any character from cin
// (doesn't skip spaces,
// newlines, etc.)
char nextChar;
cin.get(nextChar);
```

```
// write a character to cout
cout.put(nextChar)
```

```
// read a whole line of 80 chars
char line[80+1];
cin.getline(line, 81);
```

```
// put back a char to cin
// nextChar will be the next
// char read by cin.get()
cin.putback(nextChar);
```

---

\*michele@mathsophy.com

## Files

```
#include <fstream>

// input file
ifstream inStream;
// output file
ofstream outStream;

// open
inStream.open("infile.dat");
outStream.open("outfile.dat");

// check for failure
if ( inStream.fail() ||
    outStream.fail() )
{
    // file opening failed
}

// read/write data
inStream >> data1 >> data2;
outStream << data1 << data2;

// checking for end of file
while ( ! inStream.eof() )
{
    inStream >> next;
}

// close file
inStream.close();
outStream.close();
```

## Strings

```
#include <string>

// initialization
string s1 = "Hello";
string s2("World");

// concatenation
string s3 = s1 + ",_" + s2;

// read a line
string line;
getline(cin, line);

// access to the ith character
// (no illegal index checking)
s1[i];

// access to the ith character
// (with illegal index checking)
s1.at(i);

// append
s1.append(s2);

// size and length
s1.size();
s1.length();

// substring from position 5
// and length 4 characters
s4.substr(5,4);

// find (returns string::npos
// if not found)
s3.find("World");

// find starting from position 5
s3.find("l",5);

// C-string
s3.c_str();

// from string to int,
// long, float
int n = stoi("456");
long n = stol("1234567");
double n = stod("12.345");

// from numeric type to string
string s = to_string(123.456);
```

## Vectors

```
#include <vector>

// vector with base type int
vector<int> v = {2, 4, 6, 8};

// vector with 10 elements
// all initialised to 0
vector<int> v(10);

// range-for-loop
for (auto x : v)
    cout << x << endl;

// access to the ith element
cout << v[i];

// add an element
v.push_back(10);

// size
cout << v.size();

// capacity: number of
// elements currently allocated
cout << v.capacity();

// reserve more capacity
// e.g. at least 64 ints
v.reserve(64);
```

## Classes

If you give no constructor, the compiler will generate a default constructor that does nothing. If you give at least one constructor, then the C++ compiler will generate no other constructors.

```
class Car
{
public:
    // constructor
    Car(double);
    // mutators
    void setEngineSize(const
        double&);
    // accessors
    double getEngineSize() const;
    // friend function
    friend bool equal(const Car&,
        const Car&);

private:
    double engineLiter;
};

// constructor
// with initialization list
Car::Car(double engineSize) :
    engineLiter(engineSize)
{
}

// parameter passed by
// reference for efficiency
void Car::setEngineSize(const
    double &size)
{
    engineLiter = size;
}

// constant member function
// doesn't change the object
double Car::getEngineSize() const
{
    return engineLiter;
}

// friend function with
// direct access to
// private members
bool equal(const Car &car1,
    const Car &car2)
{
    return car1.engineLiter ==
        car2.engineLiter;
}
```

## Operator overloading

The behaviour is different if overloaded as class members or friend functions.

⇒ As class members:

```
class Euro
{
    // constructor for euro
    Euro(int);
    // constructor for euro and
    // cents
    Euro(int,int);
    // works for Euro(5) + 2,
    // equivalent to
    // Euro(5).operator+( Euro(2) )
    // doesn't work for 2 + Euro(5)
    // 2 is not a calling object
    // of type Euro !
    Euro operator+(const Euro&);
    friend Euro
        operator+(const Euro&,
            const Euro&);

private:
    int euro;
    int cents;
};
```

⇒ As friend members:

```
class Euro
{
    // constructor for euro
    Euro(int);
    // constructor for euro and
    // cents
    Euro(int,int);
    // works for every combination
    // int arguments are converted
    // by the constructor to Euro
    // objects
    friend Euro
        operator+(const Euro&,
            const Euro&);
    // insertion and extraction
    // operators
    friend ostream&
        operator<<(ostream&,
            const Euro&);
    friend istream&
        operator>>(istream&, Euro&);

private:
    int euro;
    int cents;
};
```

## Copy constructor / Assignment operator

If not defined, C++ automatically adds the default copy constructor and the default assignment operator. They might not be correct if dynamic variables are used, because class members are simply copied.

```
class IntList
{
    // constructor with
    // size of the list
    IntList(int);
    // copy constructor
    IntList(IntList&);
    // assignment operator
    IntList& operator=(const IntList&);

private:
    int *p;
    int size;
}

// call the copy constructor
// secondList is initialised
// from firstList
IntList secondList(firstList);

// call the assignment operator
thirdList = firstList;
```

## Inheritance

Constructors, destructor, private member functions, copy constructor and assignment operator are not in-

herited! Derived classes get the default ones if they are not explicitly provided but are present in the base class.

```
// a simple book class
class Book
{
public:
    Book(string t,int p);
    void print(ostream& os);
protected:
    int pages;
    string title;
};
```

⇒ Redefinition of function members:

```
// a simple textbook class
class Textbook : public Book
{
public:
    Textbook(string t,int p,
              string s);
    // redefinition of print()
    // from the base class
    void print(ostream& os);
protected:
    string subject;
};
```

⇒ **protected** members can be accessed by derived function members:

```
// has access to protected
// members of the base class
void Textbook::print(ostream& os)
{
    os << "The_title_of_this_"
        << "textbook_is_" <<
        title << " _and_the"
        << " _textbook_is_" <<
        pages << " _pages_long."
        << endl;
    os << "The_subject_is_"
        << subject
        << " " << endl;
}
```

⇒ With redefinition, no polymorphism!

```
Book *abook = &aMathTextbook;
// call Book::print()
// not Textbook::print()!
abook->print(cout);
```

## Polymorphism

**virtual** allows for late binding, i.e. polymorphism. Function members are overridden in the derived class. Note: Destructors should also be declared **virtual**. When derived objects are referenced by base class pointers, the destructor of the derived class is called if it is declared virtual.

```
// a simple book class
class Book
{
public:
    Book(string t,int p);
    virtual ~Book();
    void print(ostream& os);
protected:
    int *pages;
    string *title;
};
```

```
Book::Book(string t, int p)
{
    pages = new int(p);
    title = new string(t);
}
```

```
Book::~~Book()
{
    delete pages;
    delete title;
}
```

```
// a simple textbook class
class Textbook : public Book
{
public:
    Textbook(string t,int p,
```

```
        string s);
    virtual ~Textbook();
    // overriding of print()
    // from the base class
    virtual void print(ostream& os);
protected:
    string *subject;
};
```

```
Textbook::Textbook(string t,
                    int p, string s) :
    Book(t,p)
{
    subject = new string(s);
}
```

```
Textbook::~~Textbook()
{
    delete subject;
}
```

```
Book *abook = &aMathTextbook;
// call Textbook::print()!
abook->print(cout);
```

## Exceptions

The value thrown by **throw** can be of any type.

```
// exception class
class MyException
{
public:
    MyException(string s);
    virtual ~MyException();
    friend ostream&
        operator<<(ostream&,
                   const MyException& e);
protected:
    string msg;
};

try
{
    throw MyException("error");
}
catch (MyException e)
{
    cout << e;
}
// everything else
catch (...)
{
    exit(1);
}
```

⇒ Functions throwing exceptions should list the exceptions thrown in the exception specification list. These exceptions are not caught by the function itself!

```
// exceptions of type DivideByZero or
// OtherException are
// to be caught outside the function.
// All other exceptions end the program
// if not caught inside the function.
void myFunction( ) throw (DivideByZero,
                          OtherException);
```

```
// empty exception list;
// all exceptions end the
// program if thrown but
// not caught inside the function.
void myFunction( ) throw ( );
```

```
// all exceptions of all
// types treated normally.
void myFunction( );
```

## Templates

⇒ Function templates:

C++ does not need the template declaration. The template function definition is included directly.

```
// generic swap function
template<class T>
void swap(T& a, T& b)
{
```

```
    T temp = a;

    a = b;
    b = temp;
}
```

```
int a, b;
char c,d;
```

```
// swaps two ints
swap(a,b);
```

```
// swaps two chars
swap(c,d);
```

⇒ Class templates:

Methods are defined as template functions

```
template<class T>
class AList
{
    // constructor with
    // size of the list
    AList(int size);
    // destructor
    ~AList();
    // copy constructor
    AList(AList<T>& b);
    // assignment operator
    AList<T>& operator=(const
                        AList<T>& b);
private:
    T *p;
    int size;
}
```

```
// constructor definition
template<class T>
AList<T>::AList(int size)
{
    p = new T[size];
}
```

```
// variable declaration
AList<double> list;
```

## Iterators

An iterator is a generalization of a pointer. Different containers have different iterators.

```
#include <vector>
```

```
vector<int> v = {1,2,3,4,5};
// mutable iterator
vector<int>::iterator e;
```

```
// bidirectional access
e = v.begin();
++e;
// print v[1]
cout << *e << endl;
--e;
// print v[0]
cout << *e << endl;
```

```
// random access
e = v.begin();
// print v[3]
cout << e[3] << endl;
```

```
// change an element
e[3] = 9;
```

```
// constant iterator (only read)
vector<int>::constant_iterator c;
```

```
// print out the vector content
// (read only)
for (c = v.begin(); c != v.end(); c++)
    cout << *c << endl;
```

```
// not allowed
// c[2] = 2;
```

```
// reverse iterator
vector<int>::reverse_iterator r;
```

```
// print out the vector content
// in reverse order
for (r = v.rbegin(); r != v.rend(); r++)
```

```
cout << *r << endl;
```

## Containers

⇒ Sequential containers: **list**

```
#include <list>
```

```
list<double> data = {1.32,-2.45,5.65};
```

```
// adds elements
data.push_back(9.23);
data.push_front(-3.94);

// bidirectional iterator
// no random access
list<double>::iterator e;
```

```
// erase
e = data.begin();
++e;
data.erase(e);
```

```
// print out the content
for (e = data.begin();
     e != data.end(); e++)
    cout << *e << endl;
```

⇒ Adapter containers: **stack**

```
#include <stack>
```

```
stack<double> numbers;
```

```
// push on the stack
numbers.push(5.65);
numbers.push(-3.95);
numbers.push(6.95);
```

```
// size
cout << numbers.size()
```

```
// read top data element
double d = numbers.top();
```

```
// pop top element
numbers.pop();
```

⇒ Associative containers: **set**

```
#include <set>
```

```
set<char> letters;
```

```
// inserting elements
letters.insert('a');
letters.insert('d');
// no duplicates!
letters.insert('d');
letters.insert('g');
```

```
// erase
letters.erase('a');
```

```
// const iterator
set<char>::const_iterator c;
for (c = letters.begin();
     c != letters.end(); c++)
    cout << *c << endl;
```

⇒ Associative containers: **map**

```
#include <string>
#include <map>
#include <utility>
```

```
// initialization
map<string,int> dict =
    { {"one",1}, {"two",2} };
pair<string,int> three("three",3);
```

```
// insertion
dict.insert(three);
dict["four"] = 4;
dict["five"] = 5;
```

```
// iterator
map<string,int>::iterator two;
```

```
// find
two = dict.find("two");
```

```
// erase
```

```
dict.erase(two);
```

```
// ranged loop
for (auto n : dict)
    cout << "(" << n.first
        << "," << n.second
        << ")" << endl;
```

## Algorithms

```
#include <vector>
#include <algorithm>
```

```
vector<int> v = {6,2,7,13,4,3,1};
vector<int>::iterator p;
bool found;
```

```
// find
p = find(v.begin(),v.end(),13);
```

```
// merge sort
sort(v.begin(),v.end());
```

```
// binary search
found = binary_search(v.begin(),
                     v.end(),3);
```

```
// reverse
reverse(v.begin(),v.end());
```

## References

- [1] Walter Savitch. *Problem Solving with C++*, 10th edition. Pearson Education, 2018
- [2] Bjarne Stroustrup. *Programming: Principles and Practice Using C++*, 2nd edition. Addison Wesley, 2015
- [3] Josh Lospinoso. *C++ Crash Course: A Fast-Paced Introduction*, 1st edition. No Starch Press, 2019