C++ Cheat Sheet

by Benjy via cheatography.com/56918/cs/16149/

enum

```
enum Color {
    COLOR_RED, // assigned 0
    COLOR_BLUE // assigned 1
};
Color color=COLOR_RED;
// input/output
int input{};
std::cin >> input;
Color color=static_cast<Color>-
(input)
```

Prefer enum class if possible (C++11)

enum class (C++11)

```
enum class Color {
    RED,
    BLUE
};
Color color=Color::RED;
std::cout << static_cast<int>(c-
olor);
```

typedef

```
typedef double distance_t;
using distance_t = double; //
C++11
```

struct

```
struct Employee {
    short id;
    int age;
};
Employee joe = {1, 42}; //
initializer list
joe.age = 43; // member
selection operator
```

std::array - C++11

```
#include <array>
std::array<int, 3> tab {1, 2,
3};
tab[0] // or tab.at(0) with
bounds checking
tab.size() // length not
sizeof()
```

std::vector

```
#include <vector>
std::vector<int> tab {1, 2, 3,
4};
tab[0] // or tab.at(0) with
bounds checking
tab.size()
tab.resize(3); // expensive
// capacity & stack-like
behavior
std::vector<int> stack;
stack.reserve(5); // set
capacity
stack.push_back(3);
int x = stack.back(); // top of
stack
stack.pop_back();
```

capacity = allocated memory
length = active values
Setting capacity avoid resizing the array

std::tuple - C++11

```
#include <tuple>
// using namespace std;
tuple<int, double> t = make_t-
uple(3, 6.7);
int a = get<0>(t);
// std::tuple<int, double>
foo();
int a;
double b;
std::tie(a,b) = foo()
auto [a, b] = foo(); // C++17
```

Pointers

```
int value=5;
int *const p=&value; //const
pointer
const int value=5;
const int *p; // non-const
pointer to const int
// const pointer to const value
const int *const p=&value;
// int foo(int x);
int (*ptr)(int); // function
pointer
int (*const ptr)(int) = foo; //
const pointer
#include <functional>
std::function<int(int)> ptr =
foo; // C++11
ptr(5); // implicit dereference
or (*ptr)(5)
```

References

```
int value=5;
int &ref=value; // non-const ref
const int value=5;
const int &ref=value; // treats
value as const
```

References are similar to const pointers (cannot change address)

new & delete (dynamic allocation)

```
// scalar version
int *p=new int(3);
delete p;
// array version
int *p = new int[3] {1, 2, 3};
// C++11 syntax
delete[] p;
```

Classes

```
class Foo {
private:
    // non-static member initialization
    int m_value = 10; // C++11
    // static member variable //
    static int m_id;
public:
    // constructor and destructor
    Foo(int value):m_value(value);
    ~Foo();
   // copy constructor
    Foo(const Foo& foo)
                :m_value(foo.m_value);
    // static member function
    static int get_id(); // see definition
    // const member function
    void print() const;
    // friend function
    friend void reset(Foo &foo);
};
Foo(6); // anonymous object
// static
int Foo::m_id=0;
int Foo::get_id() { return m_id; } // no static
int id = Foo::get_id();
// const member function
void Foo::print() const { whatever }
const Foo foo;
foo.print(); // no constness violation
// friend function
```

Classes (cont)

```
friend void reset(Foo &foo) { foo.m_value=0; }
```

1) no implicit default constructor if at least one constructor provided
2) implicit copy constructor and overloaded operator= use memberwise initialization (shallow copy)

std::initializer_list (C++11)

```
#include <initalizer_list>
int m_size;
int *m_array;
// prevent shallow copy (dynamic mem alloc)
Foo(const Foo& foo) = delete;
Foo& operator=(const Foo& foo) = delete;
Foo(const std::initializer_list<int>& list)
    :m_size(static_cast<int>(list.size))
{
    // deep copy with for-each loop
    int count=0;
    for (auto& value : list)
        m_array[count++] = value;
}
Foo& operator=(const std::initializer_list<int> &list)
{
    // same but delete[] m_array
    // if same size (no need for new or delete)
    // deep copy
}
Foo foo { 1, 2, 3, 4 }; // call constructor
Foo foo = { 1, 2, 3, 4 }; // call operator=
```

For-each loops - C++11

```
for (int num : tab)
for (auto num : tab) // avoid type conversion
for (auto &num : tab) // avoid copy
for (const auto &num : tab) // read-only
```

For-each loops don't work with array decayed to pointer (fixed or dynamic)

std::rand()

```
#include <cstdlib>
#include <ctime>
std::stand(static_cast<unsigned int>(std::time(nullptr))); // seed with time
std::rand(); // [0, RAND_MAX]
// [min, max]
return min + (std::rand()%(max--min+1))
//better
static constexpr double fraction {1.0/(RAND_MAX+1.0)};
return min + static_cast<int>((max-min+1.0)*(std::rand()*fraction))
```

% is biased towards low numbers

Input validation with std::cin

```
// extraneous input
std::cin.ignore(32767, '\n'); // clears buffer
// extraction failure
if (std::cin.fail()) { // type or overflow
    std::cin.clear(); // back to normal mode
    std::cin.ignore(32767, '\n');
}
// switch-like validation
```

Input validation with std::cin (cont)

```
int x {};
do {
    std::cin >> x;
} while (x<0);
```

std::cerr and assert

```
#include <cstdlib> // for exit
std::cerr << "Error message";
exit(1);
#include <cassert>
assert(x==0); // combines cerr
and exit
assert(x==0 && "Error message");
// check at compile-time
static_assert(x==0, "Error
message"); // C++11
```

Operator overloading

```
int m_value;
int *m_array=nullptr;
// ret by value (ref to out of
scope var)
friend Foo operator+(const Foo
&foo, const Foo& bar);

// subscript []
// ret by ref to assign (l-
value)
int& operator[](const int idx)
{ return m_array[idx]; }

// overload int typecast
operator int() const { return
m_value; }
// std::cout
std::ostream& operator<<(s-
td::ostream& out, const Foo
&foo)
{
    out << m_value;
    return out;
}
// std::cin
std::istream& operator>>(s-
td::istream& in, Foo &foo)
{
    in >> m_value;
    return in;
}
// increment operator as member
function
Foo& operator++(); // prefix,
++x
Foo operator++(int) // postfix,
x++
{
    Foo tmp(foo);
    // call implicit copy
constructor
    // or assignment with tmp =
foo
    // but beware of shallow
copy
    // increment this
    return tmp; // return by
value
}
```

1) unary op or if modif to left operand
-> member function
no modif, binary as friend or regular one
2) [], =, (), -> as member