

lvalue vs rvalue

object that persists beyond a single expression temporary value

has an address has no address

variable that has a name/const increment, decrement

class members ternary operator

strong literal func call like
std::move(x)

& to reference && to ref

Statics

```
//Only one copy per class, single
resouce can be shared between
instances, cant be initialized
inside class def'n
// static const: can be
initialized with an initializer
// static constexpr: MUST be
initialized with an initializer
class X {
static int m=5 //err
static in n; //ok
const static int p{5};
constexpr static int arr[]={1,3};
}
int X::n=5; //ok
```

Move

takes ownership of member variables from another obj

faster and avoid mem alloc (unlike copy const'r)

shallow copy

move assignment does similar

special_ptr

```
class name {
public:
name(const char* s):data{s}{};
void display_name() { cout <<
data;}
~name() {}
private:
string data;
};
void modify_name(name* m) {}
int main() {
unique_ptr<name> ptr1(new
name("D"));
//Can use -> (and *) on unique_ptr
ptr1->display_name();
// To get raw ptr use get() method
modify_name(ptr1.get());
// Use std::move to transfer
unique_ptr<name>
ptr2(std::move(ptr1));
// assign a new pointer to ptr1
ptr1.reset(new name("H"));
// assign a new pointer to ptr2
// D now auto deleted
ptr2.reset(new name("S"));
//Use make_shared<T> func to create
shared_ptr
auto ptr = make_shared<name>
("K");
//ptr and anotherPtr point to K
shared_ptr<name> sptr2=ptr;
cout<<ptr.use_count()<<"\n";
// ptr switch to D, K
// not deleted sptr2 still holding
ptr.reset(new name("D"));
// S deleted at the end of
// this func, ptr2 out of scope
}
```

6 std member functions

default constructor C();

copy c'tor C(const C&);

copy-assign C& operator=(const C&);

destructor ~C();

move c'tor C(C&&);

move assign C& operator=(C&&)

Insertion Operator

```
friend ostream& operator<<(ostream&
os, const class& c);
```

Add Functor

```
struct add_x {
add_x(int x) : x(x) {}
int operator()(int y) const {
return x + y; }
private:
int x;
};
// Now you can use it like this:
add_x add42(42); // create an
instance
int i = add42(8); // and "call" it
assert(i == 50);
```

Templates

```
template <typename T> // Function
T get_max(T a, T b) {
return (a > b ? a : b);
}
double max = get_max<double>(m, n);
// set default type by setting
K=string or V=25
template <typename K, typename V>
class Entry{ //Class
K key;
V value;
public:
```



By eskimobrand

cheatography.com/eskimobrand/

Not published yet.

Last updated 10th December, 2018.

Page 1 of 2.

Sponsored by CrosswordCheats.com

Learn to solve cryptic crosswords!

<http://crosswordcheats.com>

Templates (cont)

```
Entry(K key, V value) : key{key},
value{value}{}
};
// Generic Copy Template
template<typename InIter, typename
OutIter>
OutIter copy(InIter init, InIter
end, OutIter res) {
    while (init != end) {
        res++ = *init++;
    }
    return res;
}
```

diff between ref and pointer

pointer can be null

pointer can be reassigned

can get address of pointer

pointers can iterate over array

Dynamic cast

casts a ptr of one type to a ptr of another type within an inheritance hierarchy

allows with ptrs and ref to polymorphic types (must contain virtual func)

returns nullptr on failure

```
dynamic_cast<target_type>
(variable)
```

const cast has same syntax and is used to cast away const qualifier

static cast has same syntax, works on nonpolymorphic types, only works if 1 or both types can be implicitly converted

slicing happens when casting non ref or ptrs, it is when a derived class loses functionality

Common Functions

```
//swap
void swap(class& lhs, class& rhs) {
    std::swap(lhs.mem, rhs.mem);
}
//assignment operator
class& class::operator=(class
other) {
    swap(*this, other);
    return *this;
}
```

unique_ptr

Template, wraps a 'raw' pointer

Ensures pointer is deleted on destruction

Auto deletes the obj it is storing when: Destroyed(OOS), Value changes by assignment, Value changes by call to reset func

cannot be shared or copied

use for class data members and local variables in functions

Lambda

```
//Capture clause used to pass
variables from surrounding scope
into lambda
//[ ] no capturing, [=] outside
captured by val cannot be modified,
[&] outside captured by reference,
[var] only var captured val, [&var]
only var captured by ref
//ascending sort lambda
auto asc = [] (const int& a, const
int& b) {return a < b;};
std::sort(vector.begin(),
vector.end(), asc);
//ascending sort functor
struct ascSort {
    bool operator() (const int& a,
const int& b)
        return a < b;
};
```

Singleton

```
class singleton{
public:
    static singleton& get_instance() {
        //Guaranteed to be destroyed.
        static singleton instance;
        return
instance; //Instantiated on first
use.
    }
private:
    int test_value;
    singleton() {}
public:
    singleton(singleton const&) =
delete;
    void operator=(singleton const&)
= delete;
    int get_value() { return
test_value++; }
};
int main() {
    // singleton s; //wont compile
    cout << singleton::get_instance().get
_value(); //ok; }
```

Java enum vs C++ enum

like a class	treated as an int
can have methods	can be assigned values (even same values)