ECE326

PROGRAMMING LANGUAGES

Lecture 12 : Move Semantics

Kuei (Jack) Sun

ECE

University of Toronto

Fall 2019

Assignment 2

- Calculate the optimal strategy for Easy Blackjack
- Approach: Divide and Conquer
  - For every encounter (a particular hand vs. the dealer's)
    - Calculate the *expected value* (EV) of available actions
      - i.e. surrender, hit, stand, double, split
    - Select the action with the highest EV
- Expected value
  - Probability weighted average of all possible value
    - Assume finitely many outcomes
    - Value: for this assignment, monetary value (not point value of hands)

Example

- Standing on hard 18 against dealer's hard 16

Dealer	Outcome	Probability	EV
17	+$1	1/13	1/13
18	0	1/13	0
19, 20, 21	-$1	3/13	-3/13
bust	+1	8/13	8/13

- $EV = P_1V_1 + P_2V_2 + P_3V_3 + P_4V_4$
  $EV = (1/13)(1) + (1/13)(0) + (3/13)(-1) + (8/13)(1)$
  $EV = 6/13 = 0.462$

Stand EV Table

- Once calculated, fill the corresponding entry in table
  - standing on hard 18 against dealer’s hard 16

dealer‘s hand

	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	AA	A2	A3	A4	A5	A6
4	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------
5	------	------	------	------	------	------	------	------	------	------	------	------	------	------					------	------	------	------	------
6	------	------	------	------	------	------	------	------	------	------	------	------	------	------								------	------
7	------	------	------	------	------	------	------	------	------	------	------	------	------	------					------	------	------	------	------
8	------	------	------	------	------	------	------	------	------	------	------	------	------	------					------	------	------	------	------
9	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------
10	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------
11	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------
12	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------
13	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------
14	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------
15	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------
16	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------
17	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------
18	----	------	------	------	------	------	------	------	------	------	------	------	0.462	------	------	------	------	------	------	------	------	------	------
19	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------
20	------			------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------
21	------			------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------
AA	------			------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------
A2	------			------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------
A3	------			------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------
A4	------			------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------
A5	------		------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------
A6	------		------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------
A7	------		------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------
A8	------		------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------
A9	------		------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------

player‘s hand

Dynamic Programming

- Used extensively throughout the assignment
- Many overlapping subproblems
- Example
  - Reuse the dealer's probability table for different starting hand
    - E.g. this time player has hard 19 against the dealer's hard 16

Dealer	Outcome	Probability	EV
17,18	+$1	2/13	2/13
19	0	1/13	0
20, 21	-$1	2/13	-2/13
bust	+1	8/13	8/13

Hints and Advice

- Make sure you know the rules of Easy Blackjack well
- Calculating expected value
  - Make sure probabilities of all outcomes add up to 1
    - Use assert(isclose(psum)) to crash immediately if check fails
- Use Python's debugger
  - `python3 –m pdb main.py`
  - Debug at point of failure
    - e.g. unhandled error, assertion, etc.
- Partially done milestones are still worth marks!

Move Semantics

and rvalue reference

Passing Arguments

- C++ has two ways of passing arguments to functions
- Pass by value
  - The value of an argument is copied into the formal parameter
  - Example:

```
Complex add(Complex c, int a) {
    c.r += a;
    return c;
}
…
Complex c(5, 2);
Complex cp = add(c, 3);
```

    - Both `c` and `a` are copied before function is invoked

Pass by Value

- Arguments are *copied* to parameters
- Arguments can be variable, literal, or expression
- If variable is passed, it is guaranteed to be unaffected by the function call
- If an object is passed, its copy constructor will be called
  - To make a copy of the object for the function call
  - This can be expensive

Pass by Reference

- Reference to variable is passed to parameter
- Argument can only be a variable
- Function can modify value of the argument
  - Tip: pass by const reference can guarantee to caller that function will not modify the argument

    ```
    void swap(int & a, int & b) {
        int temp = a;
        a = b;
        b = temp;
    }
    ```

  - Passing a pointer by value is NOT pass by reference!

lvalue vs rvalue

- Notice pass by reference only accepts variable
  - Not literal or expression
  - Because it only accepts *lvalue*
- lvalue (i.e. left value)
  - Value that is stored in memory – has an address
  - Appears on *left* side of an assignment (by value)
- rvalue (i.e. right value)
  - Value that is temporary, not necessarily in memory
  - Appears on *right* side of an assignment (by value)

Example

```
int i, j, *p;

i = 6;      // OK – i is lvalue, 6 is rvalue
j = i;      // OK – j is lvalue, i can be converted to rvalue

9 = j;      // FAIL – left operand must be lvalue
j*2 = 3;    // FAIL – j*2 is an rvalue (temporarily calculated)

p = &6;             // FAIL – rvalue has no address
p = &i;             // OK – lvalue has an address
*p = 5;             // OK – dereferenced pointer is an lvalue
*((int *)123) = 3;  // OK – dereferencing static address

((i < 4) ? i : j) = 6;  // OK – operator returns lvalue
```

Example

```
int i, j;
int & r = 5;        // FAIL – cannot bind non-const lvalue
                    //        reference to rvalue
int & r = j;        // OK – lvalue reference binds to lvalue
const int & r = 5;  // OK – const lvalue reference to lvalue

void foo(int & a) { … }
foo(5);              // FAIL – argument requires lvalue
foo(j);              // OK

int foo() { return 5; }
foo() = 3;               // FAIL – foo() returns rvalue

int & global() { return i; }
global() = 4;            // OK – global() returns lvalue
```

Rvalue Reference

- A *reference* to rvalue!
- Extends the lifetime of rvalue until reference expires
  - E.g. goes out of scope
  - Allows reference to temporary objects and modify them
    - Helps reduce making redundant copies

```
string a = “hello ”;
string b = “world”;
string c = a + b;    // may invoke copy constructor
string && r = a + b; // rvalue reference to temporary object
                     // no extra copy is made
```

- Regular references are called *lvalue* references

Example

```
struct Foo {
    int * p;
    Foo(int a) : p(new int(a)) {}
    Foo(const Foo & f) : p(new int (*f.p)) {
        cout << "copy " << *p << endl;
    }
    ~Foo() { delete p; }
    Foo operator+(const Foo & f) { return Foo(*p + *f.p); }
};

Foo a(3);
Foo b(5);
Foo c = a + b;
Foo && r = a + a;
```

```
>> ./foo

>> # no output (copy constructor
   #             not called)
```

Copy Elision

- Optimization technique for avoiding copying of objects
- Return value optimization
  - Instead of copying temporary object to final location, build object at final location directly
  - Must be object of exactly same type
- Can change behaviour of program if copy constructor has side effects (e.g. increment a global variable)
- Enabled by default
  - Can be turned off: `-fno-elide-constructors`

Pass by Rvalue Reference

- Copy elision only applies to return values
- When pass by value, copy still must be made
  - For large objects, you should *usually* pass by const reference
  - Unless you need to modify a local copy
    - Search algorithm where you make a hypothetical move, go deeper in recursion, and undo the move (may be useful in assignment 2!)
- Standard C++ library
  - Uses this to maintain genericity while improving performance
- Forwarding
  - Allows arguments to be forwarded without additional copies

Move Constructor

```
struct Foo {
    int * p;
    Foo(int a) : p(new int(a)) {}
    Foo(const Foo & f) : p(new int (*f.p)) {
        cout << "copy " << *p << endl;
    }
    ~Foo() { delete p; }
    Foo operator+(const Foo & f) { return Foo(*p + *f.p); }

    /* move constructor */
    Foo(Foo && f) : p(f.p) {
        /* we “moved” f.p to this->p, so must set f.p to
         * to nullptr otherwise ~Foo() will delete it! */
        f.p = nullptr;
    }
};
```

Move Assignment

```
struct Foo {
    int * p;
    Foo(int a) : p(new int(a)) {}
    ~Foo() { delete p; }
    Foo operator+(const Foo & f) { return Foo(*p + *f.p); }
    …

    /* move assignment */
    Foo & operator=(Foo && f) {
        if (this == &f) return *this;
        delete p;       // delete current resource
        p = f.p;        // move resource from other
        f.p = nullptr;  // makes sure this->p is not deleted
        return *this;   // when temporary object f dies
    }
};
```

std::move

- Forces move semantics on a variable
  - Convention: moved variable should become “empty”
    - Reality: some developers may leave moved object in unsafe state

```
#include <utility>

string a = “hello world”;
string b = a;              // copy constructor is called
string c = std::move(a);  // move constructor is called

// prints nothing (a is an empty string), could be worse…
cout << a << endl;
```

  - Use to forward arguments or as part of move constructor

std::move

```
bool Rule::can_surrender(Hand hand) { … }

struct BJ {
    std::vector<Hand> hands;
    Rule * rule;

    /* forwarding example (DO NOT USE) */
    bool can_surrender(Hand hand) {
        return rule->can_surrender(std::move(hand));
    }

    /* move constructor */
    BJ(BJ && other) :
        hands(std::move(other.hands)), rule(other.rule) {
        other.rule = nullptr;
    }
};
```

Calls copy constructor to make a copy of argument

Calls move constructor to move argument for function call

std::move

- Use when variable will be “consumed”
  - i.e. will not be used afterward
- Example
  - Set up an array of strings

```
std::vector<string> words;
while (true) {
    std::string line;
    getline(std::cin, line);
    if (line.length() == 0)
        break;
    /* move line to the new element, instead of copy */
    words.push_back(std::move(line));
}
```

noexcept

- Move semantics should not acquire new resources
  - Just reassigning content to new location
- Performance improvement if noexcept specified

```
struct BJ {
    std::vector<Hand> hands;
    Rule * rule;

    /* move constructor */
    BJ(BJ && other) noexcept :
        hands(std::move(other.hands)), rule(other.rule) {
        other.rule = nullptr;
    }
};
```

Limitation

- POD types
  - Plain old data types – same as a C struct
  - Does not use object-oriented features
    - NO copy constructor, static member fields, virtual table, etc.
  - Does not benefit as much from move semantics
    - Unless it holds pointers to larger structures
  - Should always pass large PODs by lvalue-reference (const or non-const) unless local copy is needed
- Primitive types
  - Always pass by value if read-only