Introduction to C++11

CS 355

C++

Bjarn Stroustrup

"C++ is a general purpose programming language with a bias towards systems programming that…"

- is a better C,
- supports **data abstraction**,
- supports **object oriented programming**,
- supports **generic programming**.

C++ is a better C

- C provides a direct and efficient model of the underlying hardware.
  - C is the de facto *systems programming* language used to write Unix.
  - C maps to machine code in a fairly obvious way and needs very little run-time support.
  - C, with its simple imperative model and weak typing, gives the programmer (almost) direct access to the machine (and plenty of rope to hang himself with).
- C++ is a direct descendent of C (but not a superset) that provides stronger type checking and a wider range of programming styles.
- C++ statements and expressions need *no run-time support* except for the operators `new, delete, typeid, dynamic_cast,` and `try/throw`.

Procedural Programming

- **Imperative Programming**
  - A program is modeled as a sequence of instructions (*i.e.,* commands = imperatives).
  - Matches CPU’s *fetch-decode-execute* cycle.
- **Procedural Language**
  - Focus on processing and designing data structures.
  - Statements are grouped into reusable, well defined procedures with simple flow control mechanisms that process structured data.

Data Abstraction

- **Abstract Data Type (ADT)** : A type that is defined by a set of operations (i.e., an interface).

- **Encapsulation**: Internal implementation details are hidden.

    - *Private* vs *Public privileges* to data and code.

- *Abstract* and *Concrete types* with well defined constructors (allocation and initialization) and destructors (de-initialization and de-allocation).

Object Oriented Programming (OOP)

- ADT support
- **Inheritance** via *class hierarchies*
  - C++ supports multiple-inheritance!
- **Polymorphism**
  - C++ provides *virtual functions* whose run-time behavior can vary based on specifics of type.
- In the spirit of *Simula,* programs can manipulate objects of a variety of types through well-defined interfaces.

Generic Programming

- Can define a *single algorithm* that operates on any type that supports the implied operations.

```
sort(arrayOfInts);
sort(vectorOfMonkeys);
sort(flockOfJuJuBirds);
```

- Define *container types* whose payload can be any type (even a primitive type) that supports the necessary operations.
- **Templates** provide *parametric polymorphism*.
- The *C++ Standard Library* is a collection of templates.

Brief C++ History

- 1979, C with Classes, Bjarne Stroustrap, Bell (later AT&T) Labs
  - Added Simula style classes to C
  - Front end generated C code.
- 1984 renamed C++
- 1985 first commercial release
  - Used preprocessor macros for templates 😝
- 1991 templates and exception handling added
- 1998 ISO C++ standard
- 2002 C++0x standard started
- 2011 ISO C++11 standard complete
- 20?? C++14

C++11 Features
added to C++98

- `auto` : static type inference
- `constexpr` : statically evaluated expressions
- `nullptr`
- delegating constructors, `initializer_list` constructors
- universal and uniform initialization (using curly braces { }'s)
- lambda expressions
- `std::move` semantics (avoid copy overhead)
- `for(e:range)` statement
- variadic templates
- `final` and `override`
- multithreading support, etc, etc, etc…

Playing Card Type in C99

```
enum {ACE=1, JACK=11, QUEEN, KING};
enum {SPADES, CLUBS, DIAMONDS, HEARTS};

struct Card {
  short rank; // SPADES .. HEARTS
  short suit; // ACE .. KING
};

typedef struct Card Card;
```

struct tag

typename

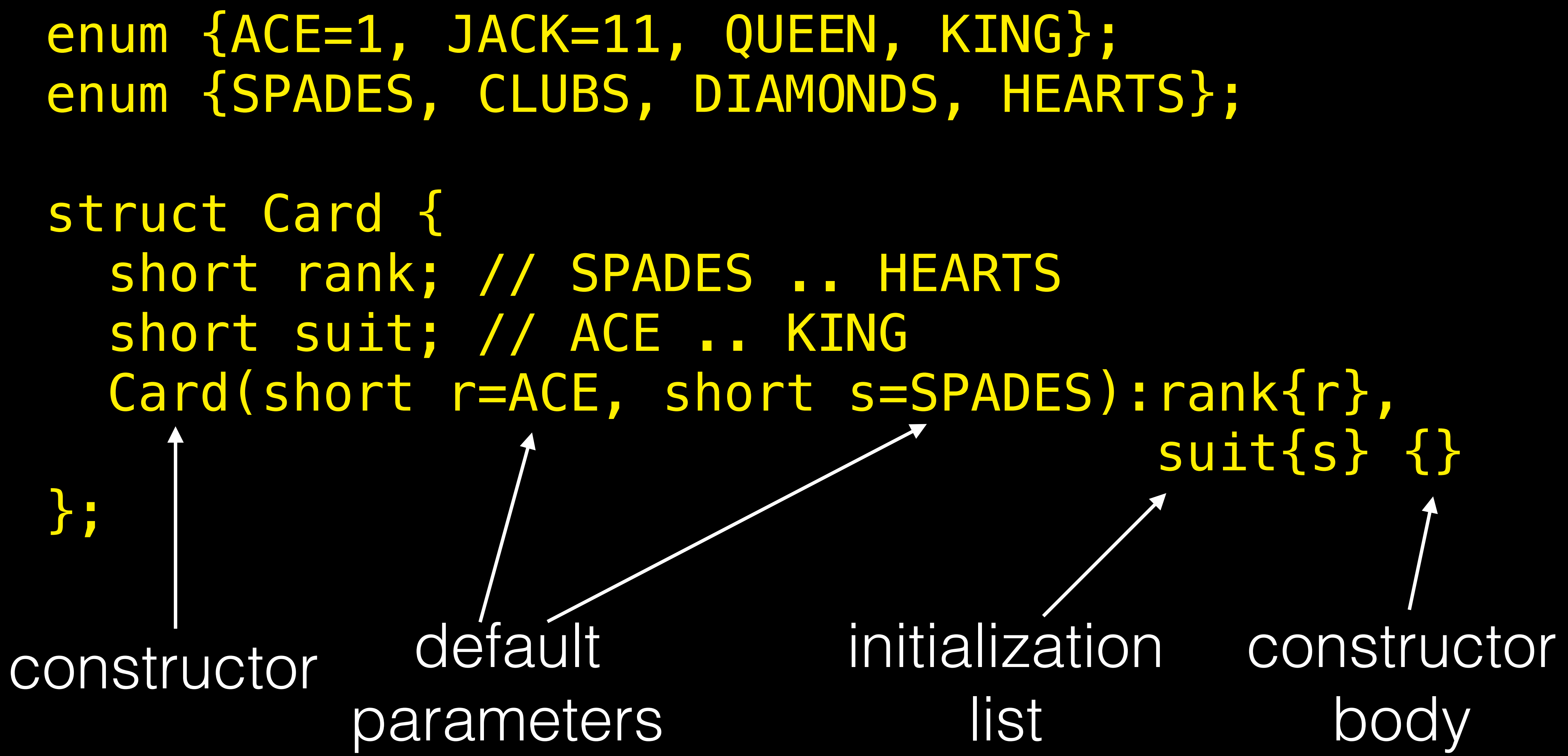
Card Objects in C99

```c
Card deathCard = {ACE, SPADES};

void foo() {
  Card dangerCard = {QUEEN, HEARTS};
  Card *wildCard = (Card *) malloc(sizeof(Card));
  *wildCard = (Card) {JACK, HEARTS};
  // ...
  free(wildCard);
}
```

- `deathCard` : global / statically allocated.
- `dangerCard` : local / auto (stack) allocated
- `*wildCard` : dynamically allocated.

Playing Card Type in C++11



```
enum {ACE=1, JACK=11, QUEEN, KING};
enum {SPADES, CLUBS, DIAMONDS, HEARTS};

struct Card {
  short rank; // SPADES .. HEARTS
  short suit; // ACE .. KING
  Card(short r=ACE, short s=SPADES):rank{r},
                                    suit{s} {}
};
```

constructor

default parameters

initialization list

constructor body

Card Objects in C++11

```
Card deathCard = {ACE, SPADES};

void foo() {
  Card dangerCard = {QUEEN, HEARTS};
  Card *wildCard = new Card(JACK, HEARTS);
  // ...
  delete wildCard;
}
```

- `deathCard` : global / statically allocated.
- `dangerCard` : local / auto (stack) allocated
- `*wildCard` : dynamically allocated.

Poker Deck Interface

Operation	Description
`Deck();`	Constructor: create deck with 52 poker cards.
`Card deal();`	Remove and return card from top of deck.
`int count() const;`	Returns number of cards remaining in Deck (non-mutating).
`void restock();`	Reload deck with all 52 cards.
`void shuffle(int n);`	Shuffle remaining cards *n* times.

Deck Class

```
class Deck { // Poker Deck (no jokers)
private:
  int numCards;   // 0 .. 52
  Card cards[52]; // deal from top
public:
  Deck() : numCards{52} {restock();}
  // ...
};
```

- New classes defined with `struct`, `class`, and (rarely) `union`.
- **Encapsulation**: instance var's are *private* (default for `class`).
  - Non-privileged code can only access *public* members.
- **Initialization**: default *constructor* guarantees that newly created objects will be in well-defined state.

Count and Deal

```
class Deck {
private:
  int numCards;
  Card cards[52];
public:
  // ...
  int count() const {return numCards;}
  Card deal() {return cards[--numCards];}
};
```

non-mutator

- `count()` acts as “getter” method for `numCards` (non-mutator).
- `deal()` returns card from top of deck (mutator).
  - Precondition: `numCards > 0` (unchecked).
- Methods will (probably) be *inlined*, (no function call overhead).

Restock and Shuffle Deck

```
class Deck {
   // ...
public:
   // ...
  void restock();   // defined elsewhere
  void reshuffle(); // not inlined
};
```

Note that if this is defined in a header file, then the compiler may not have the source code for the definition so it can not be inlined.

Restock Deck

```
void Deck::restock() {
  numCards = 52;
  int n = 0;
  for (int s = SPADES; s <= HEARTS; s++)
    for (int r = ACE; r <= KING; r++)
      cards[n++] = Card(r,s);
}
```

- Class `Deck` defines namespace `Deck.`
- `Deck::restock` (`::` is namespace resolution operator).
    - Defined outside class definition.
    - Compiled separately.
    - Not inlined.

Shuffling Deck

```
#include <cstdlib> // C++, <stdlib.h> in C

void Deck::shuffle(int num) {
  for (int n = 0; n < num; n++)
    for (int i = 0; i < numCards; i++) {
      const int j = arc4random() % numCards;
      const Card tmp = cards[i];
      cards[i] = cards[j];
      cards[j] = tmp;
    }
}
```

`const` : not modified after initialization.

A more versatile `Deck` class

Poker Deck	52 cards rank: A, 2, 3, ..., 10, J, Q, K suits: ♤♧♡♢
Pinochle Deck	48 cards (2 of each kind) rank: A, 9, 10, J, Q, K suits: ♤♧♡♢
Double Pinochle Deck	Two Pinochle Decks

Versatile Deck

```
class Deck {  // Poker, Pinochle, or Double Pinochle
private:
  int capacity; // 52, 48, 96
  int numCards; // 0 .. capacity
  Card *cards;  // dynamically alloc’ed array of cards

  Deck(int n) : capacity{n}, numCards{n},
                cards{new Card[n]} {restock();}
public:
  static Deck PokerDeck() {return Deck(52);}
  static Deck PinochleDeck() {return Deck(48);}
  static Deck DoublePinochleDeck() {return Deck(48*2);}
  // ... count(), deal(), restock(), shuffle()
  ~Deck() {delete[] cards;}
};
```

- *Private constructor* : dynamically allocates **`cards`** array.
- *“Static constructors”* : *class methods* for creating **`Deck`** objects.
- **Destructor** for deallocating **`cards`** array (used **`delete[]`** operator).

Object Lifetime

```
Deck deckA = Deck::PokerDeck();

int main() {
  // ...
}
```

- `deckA` (static / global scope)
  - Lifetime = program lifetime
  - Initialized before `main()` invoked.
  - Destructor called after `main()` finished.
    - Destructor invoked even on abnormal termination (*e.g.*, `exit()`)

Object Lifetime, cont…

```
void foo() {
  Deck deckB = Deck::DoublePinochleDeck();
  for (int i = 0; i < 10; i++) {
    Deck deckC = Deck::PinochleDeck();
    // ...
    // deckC leaving scope (destructor called).
  }
  // deckB leaving scope (destructor called).
}
```

- `deckB/DeckC` (stack / local)
  - Constructed each time declaration is elaborated.
  - Destroyed after leaves scope (even abnormally *e.g.*, `throw`).
  - `deckC` initialized / destroyed 10x on each invocation of `foo().`

Object Lifetime, cont...

```
void foo() {
  // ...
  for (int i = 0; i < 10; i++) {
    static Deck deckD = Deck::PinochleDeck();
    // ...
  }
}
```

- `deckD` (static = single copy/ local scope)
  - Initialized the first time its declaration is elaborated.
  - Destroyed after `main()` terminates.

Object Lifetime, cont...

```
Deck* deckE;

void foo() {
  deckE = new Deck(Deck::PokerDeck());
  // ...
}

main() {
  foo();
  delete deckE;
}
```

- `deckD` (dynamically allocated off heap)
  - Lifetime controller by programmer.
  - Initialized when `new` operator invoked.
  - Destroyed when `delete` operator invoked.

References

```
Deck deckB = Deck::DoublePinochleDeck();
Deck& d = deckB;
d.shuffle(5);
Card c = d.deal();
```

- `Deck&` = reference type
- `d` is a reference (alias) of `deckB`.
    - *i.e.*, `d` and `deckB` both refer to the same object.
- References are bound when initialized, and can not be assigned to.

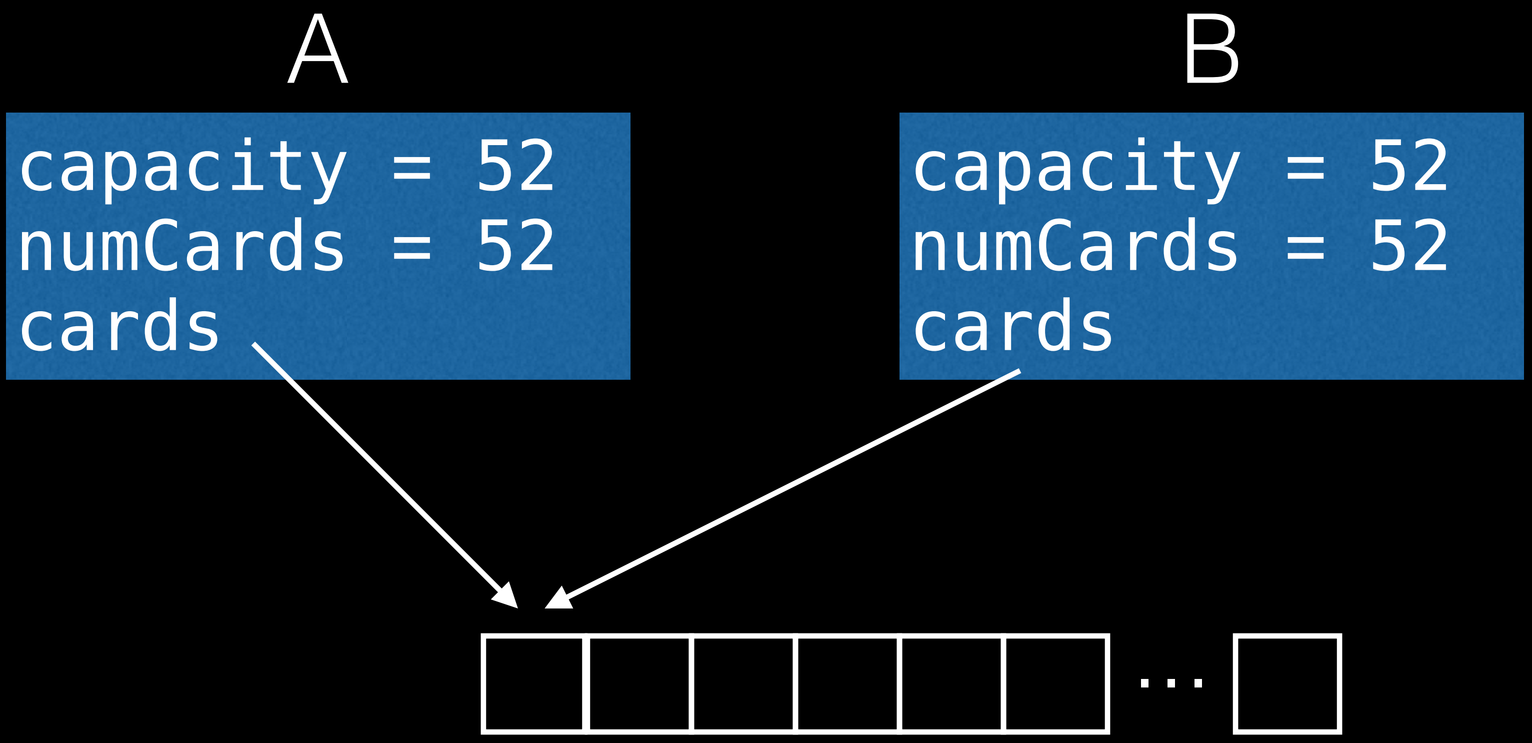
Reference Arguments

```
void foo(Deck  deck); // pass by value
void bar(Deck* deck); // pass by ptr
void doo(Deck& deck); // pass by ref
void dad(const Deck& deck); // immutable ref
```

- `foo(d)` : pass by value, copy of d created.
- `bar(&d)` : pass ptr to d, `bar()` can modify d.
- `doo(d)` : pass by reference, `doo()` can modify d.
- `dad(d)` : pass by reference, `dad()` can not modify d.
  - `dad(Deck::PokerDeck())` : allows literal (compiler generated temporary) to be passed by reference.

Default Copy is Shallow

```
Deck A = Deck::PokerDeck();
Deck B = A; // copy
```



A

```
capacity = 52
numCards = 52
cards
```

B

```
capacity = 52
numCards = 52
cards
```

Both `A.cards` and `B.cards` point to the same array!
Any mutation of A and B will cause inconsistency!
Destructor will `delete[] cards` twice!

Defining Deep Copy

```
class Deck {
  // ...
public
  Deck(const Deck& other); // Copy Constructor
  Deck& operator=(const Deck& other); // Copy Assign
  // ...
};
```

- To specify a *deep copy* we must...
  - define a **copy constructor**, and
  - overload the **assignment operator**.

Copy Constructor

```
Deck::Deck(const Deck& other)
    : capacity{other.capacity},
      numCards{other.numCards},
      cards{new Card[other.capacity]}
  {
      for (int i = 0; i < capacity; i++)
        cards[i] = other.cards[i];
  }
```

- Object to copy is passed by constant reference.
  - Pass by value would trigger infinite recursion! Why?
  - Allows `Deck` literals to by copied.
- A new array is allocated and we *clone* the contents of the original.

Copy Assignment

```
Deck& Deck::operator=(const Deck& other) {
❶   if (this != &other) {
❷     capacity = other.capacity;
      numCards = other.numCards;
❸     delete[] cards;
❹     cards = new Card[capacity];
      for (int i = 0; i < capacity; i++)
❺       cards[i] = other.cards[i];
    }
❻   return *this;
  }
```

1. Protect against self assignment.

2. Copy payload where normal assignment is sufficient.

3. Delete old array.

4. Allocate new array

5. Copy array contents

6. Return reference to overwritten object.

Making Copies

```
Deck A = Deck::PokerDeck();
Deck B = A; // copy construct
Deck C{A};  // copy construct (curly syntax)
Deck D = Deck::PinochleDeck();
D = A;      // copy assignment
```

`B, C, D` all end up with deep copies of `A`.

Swapping Copies

```
Deck A = Deck::PokerDeck();
Deck B = Deck::PinochleDeck();
//...
Deck tmp = A;  // copy construct
A = B;         // copy assign
B = tmp;       // copy assign
```

Note that three copies had to be made
to swap the contents of two objects.

We really just want to **move** the contents
of A into B, and vice versa (no copies necessary).

Defining Move

```
class Deck {
  // ...
public
  Deck(Deck&& other); // Move Constructor
  Deck& operator=(const Deck&& other); // Move Assign
  // ...
};
```

- && means *r-value reference* in C++
    - r-value = object that can appear on the *right-hand* side of the assignment operator.
    - l-value = object that can appear on the *left-hand* side of the assignment operator.
- To specify a *move* we must...
    - define a **move constructor**, and
    - overload the **assignment operator** for a move**.**

Move Constructor

```
Deck::Deck(Deck&& other) : capacity{0},
                           numCards{0},
                           cards{nullptr}
{
  capacity = other.capacity;
  numCards = other.numCards;
  cards = other.cards;
  other.capacity = 0;
  other.numCards = 0;
  other.cards = nullptr;  // protect delete[]
}
```

- We move payload from `other`.
- `cards` ptr is simply copied (no allocation).
- `other.cards` is set to nullptr so deleting `other` has no effect.

Move Assignment

```cpp
Deck& Deck::operator=(Deck&& other) {
  if (this != &other) {
      capacity = other.capacity;
      numCards = other.numCards;
      delete[] cards;
      cards = other.cards;
      other.capacity = 0;
      other.numCards = 0;
      other.cards = nullptr; // protect delete[]
  }
  return *this;
}
```

- We avoid “move to self” and return reference to target object.
- We delete old `cards` and simply copy `other.cards`.
- We “zero out” `other` to protect `delete`.

Swapping Objects by Moving

```
Deck A = Deck::PokerDeck();
Deck B = Deck::PinochleDeck();
//...
Deck tmp = std::move(A); // move constructor
A = std::move(B)         // move assign
B = std::move(tmp);      // move assign
```

Note that no copies had to be made
to swap the contents of two objects.

Nothing allocated nor deleted during swap
(`delete[](nullptr)` does nothing)!