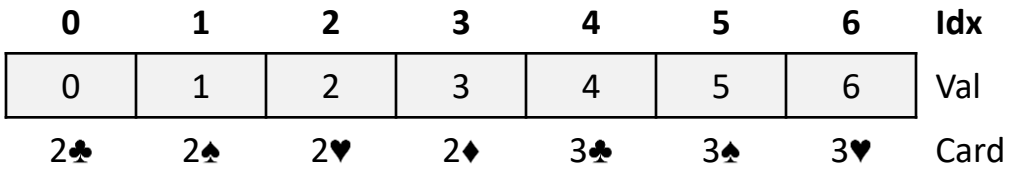
CS 103 Unit 3b – Classes

Object-Oriented Approach

- Model the application/software as a set of objects that interact with each other
- Objects fuse data (i.e. variables) and functions (a.k.a methods) that operate on that data into one item (i.e. object)
- Objects replace global-level functions as the primary method of encapsulation and abstraction
  - Encapsulation: Hiding implementation and controlling access
    - Group data and code that operates on that data together into one unit
    - Only expose a well-defined interface to control misuse of the code by other programmers
  - Abstraction
    - Hiding of data and implementation details
    - How we decompose the problem and think about our design at a higher level rather than considering everything at the lower level

Object-Oriented Programming

- Objects contain:
  - Data members
    - Data needed to model the object and track its state/operation (just like structs)
  - Methods/Functions
    - Code that operates on the object, modifies it, etc.
- Example: Deck of cards
  - Data members:
    - Array of 52 entries (one for each card) indicating their ordering (for our purposes we'll just use integers `0-51` to represent the 52 cards)
    - Top index

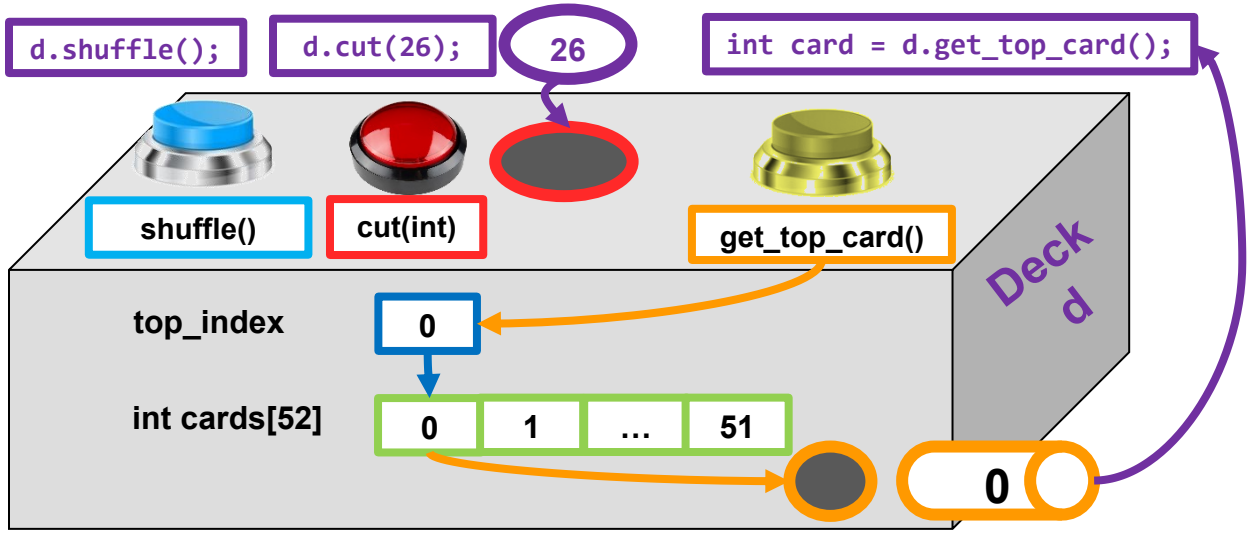


  - Methods/Functions
    - `shuffle(), cut(int where), get_top_card()`

Visualizing What Objects Do

- An object has
  - Internal state (data members)
  - And functions that allow you to update the state, perform operations on the object, and retrieve data or results from the object

Calls from main() after declarding:
`Deck d;`



C++ Classes

- Classes are the programming construct used to define objects, their data members, and methods/functions
- Similar idea to structs
- Steps:
  1. Define the class' data members and function/method prototypes

```
#include <iostream>
using namespace std;
class Deck {
 public:


   void shuffle();
   void cut(int where);
   int get_top_card();
 private:
   int cards[52];
   int top_index;
};
```

1

cardgame.cpp

C++ Classes

- Classes are the programming construct used to define objects, their data members, and methods/functions
- Similar idea to structs
- Steps:
  1. Define the class' data members and function/method prototypes
  2. Write the function/method implementations
  3.
- Terminology:
  - Class = Definition/Blueprint of an object

```
#include <iostream>
using namespace std;
class Deck {
 public:


   void shuffle();
   void cut(int where);
   int get_top_card();
 private:
   int cards[52];
   int top_index;
};
// member function implementation
void Deck::shuffle() {
  for(int i=51; i > 0; i--){
    int r = rand() % i;
    swap(cards[i], cards[r]);
  }
}
int Deck::get_top_card()
{ return cards[top_index++]; }
```

1

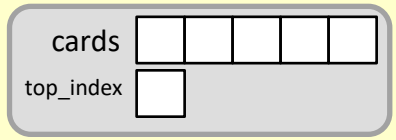
2

cardgame.cpp

C++ Classes

- Classes are the programming construct used to define objects, their data members, and methods/functions
- Similar idea to structs
- Steps:
  1. Define the class' data members and function/method prototypes
  2. Write the function/method implementations
  3. Instantiate/Declare object variables and use them by calling their methods
- Terminology:
  - Class = Definition/Blueprint of an object
  - Object = Instance of the class, actual allocation of memory, variable, etc.

```
#include <iostream>
using namespace std;
class Deck {
 public:                                          1


   void shuffle();
   void cut(int where);
   int get_top_card();
 private:
   int cards[52];
   int top_index;
};
// member function implementation
void Deck::shuffle() {                            2
  for(int i=51; i > 0; i--){
    int r = rand() % i;
    swap(cards[i], cards[r]);
  }
}
int Deck::get_top_card()
{ return cards[top_index++]; }

// Main application
int main(int argc, char *argv[]) {
  Deck d;
  int hand[5];
  d.shuffle();
  d.cut();
  for(int i=0; i < 5; i++){                       3
    hand[i] = d.get_top_card();
  }...
}
```



cardgame.cpp

Common Class Structure – Split Files

- Common to separate class into **separate source code files** so it can easily be reused in different applications
- 1) In a **header (.h) file**, define the class':
  - **1.) data members** and
  - **2.) function/method prototypes (usually in a separate header file)**
  - Must define the class using the syntax:
    - `class name { ... };`
- 2) In a **corresponding .cpp file**, write the member functions/methods
- 3) In an **application (usually separate .cpp file)**, instantiate/Declare object variables and use them by calling their methods
- How do you think you compile the application at the command line?
  - Must list all the .cpp files on the g++ command

deck.h

```
class Deck {
 public:

   void shuffle();
   void cut(int where);
   int get_top_card();
 private:
   int cards[52];
   int top_index;
};
```

1

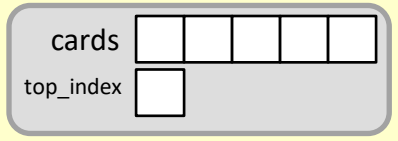
deck.cpp

```
#include<iostream>
#include "deck.h"

// Code for each prototyped method
```

2

cardgame.cpp

```
#include<iostream>
#include "deck.h"

int main(int argc, char *argv[]) {
 Deck d;
  int hand[5];
  d.shuffle();
  d.cut();
  for(int i=0; i < 5; i++){
    hand[i] = d.get_top_card();
  }...
}
```



3

Access Specifiers

- Each function or data member can be classified as public, private, or protected
  - These classifications support encapsulation by ensuring that no other programmer writes code that uses or modifies your object in an unintended way.
  - Makes private data/method members to be INACCESSIBLE to non-member functions of the class, forcing non-members to ONLY utilize the PUBLIC interface
  - **Private**: Can call or access only by methods/functions that are part of that class
  - **Public**: Can call or access by any other code
  - **Protected**: More on this later
- Everything private by default so you must use `public:` to make things visible
- Make the interface public and the guts/inner-workings private

deck.h

```
class Deck {
 public:
   Deck();   // Constructor
   ~Deck();  // Destructor (see next slide)
   void shuffle();
   void cut(int where);
   int get_top_card();
 private:
   int cards[52];
   int top_index;
};
```

deck.cpp

```
#include<iostream>
#include "deck.h"


// Code for each prototyped method
```

cardgame.cpp

```
#include<iostream>
#include "deck.h"

int main(int argc, char *argv[]) {
  Deck d;
  int hand[5];

  d.shuffle();
  d.cut();

  d.cards[0] = ACE; //won't compile
  d.top_index = 5;  //won't compile
}
```

Public / Private and Structs vs. Classes

- In C++ the **ONLY difference** between `structs` and `classes` is that `structs` default to **public** access, `classes` default to private access
- Thus, other code (non-member functions of the class) ***cannot*** access private class members directly

student.h

```
class Student {  // what's the difference
struct Student { // between these two

   Student();   // Constructor 1
   Student(string name, int id, double gpa);
                // Constructor 2
   ~Student();  // Destructor
   ...
   string name_;
   int id_;
   double gpa_;
};
```

grades.cpp

```
#include<iostream>
#include "student.h"
int main()
{
  Student s1;  string myname;
  cin >> myname;
  s1.name_ = myname;
    // compile error if 'class' but not
    // if 'struct'
  ...
}
```

Constructors / Destructors

- **Question**: What is the initial value of the d.cards and d.top_index??

- When an object comes alive, we need to have **appropriate initial values in our data members** so that future calls to member functions will work appropriately…
- …and when an object dies (goes out of scope or is deleted) we may need to perform some cleanup operations
- To help us with these tasks C++ provides **Constructor (ctor)** and **Destructor (dtor)** functions

deck.h

```
class Deck {
 public:

   void shuffle();
   void cut();
   int get_top_card();
 private:
   int cards[52];
   int top_index;
};
```

1

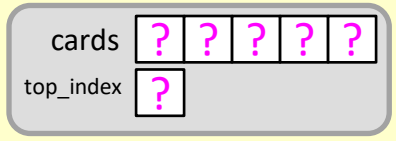
deck.cpp

```
#include<iostream>
#include "deck.h"

// Code for each prototyped method
```

2

cardgame.cpp

```
#include<iostream>
#include "deck.h"

int main(int argc, char *argv[]) {
  Deck d;
  int hand[5];
  d.shuffle();
  d.cut();
  for(int i=0; i < 5; i++){
    hand[i] = d.get_top_card();
  }...
}
```



3

Constructors / Destructors

- **Constructor** (**ctor** for short) is a function of the same name as the class itself (e.g. `Deck()`)
  - It is called automatically when the object is created (either when declared or when allocated via **new**)
  - Use to initialize your object's data members to some desired or known initial state
  - **Returns nothing**
- **Destructor** (**dtor** for short) is a function of the same name as class itself with a ~ in front (e.g. `~Deck()`)
  - Called automatically when object goes out of scope (i.e. when it is deallocated by 'delete' or when scope completes)
  - Use to free/delete any memory allocated by the object or close any open file streams, etc.
  - **Returns nothing**
  - **[Note: Currently we do not have occasion to use destructors; we will see reasons later on in the course]**

deck.h

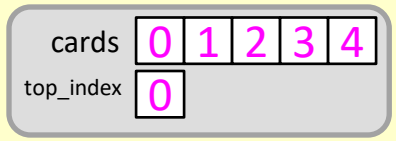
```
class Deck {
 public:
   Deck();   // Constructor
   ~Deck();  // Destructor
   ...
};
```

deck.cpp

```
#include<iostream>
#include "deck.h"

Deck::Deck() {
  top_index = 0;
  for(int i=0; i < 52; i++){
    cards[i] = i;
  }
}

Deck::~Deck()
{ /* not needed for this class */ }
```

cardgame.cpp

```
#include "deck.h"

int main() {
  Deck d;  // Deck() is called
  ...

  return 0;
  // ~Deck() is called since
}
```



Multiple Constructors

- Can have multiple constructors with different argument lists to provide options to client software for how they'd like to initialize the object
  - Constructor with NO ARGUMENTS is known as the **DEFAULT** constructor

```
#include<iostream>
#include "student.h"

int main()
{
  Student s1;  // calls default ctor
  string myname;
  cin >> myname;
  s1.set_name(myname);
  s1.set_id(214952);
  s1.set_gpa(3.67);

  Student s2(myname, 32421, 4.0);
              // calls "initializing" ctor

}
```

student.h

```
class Student {
 public:
   Student();   // Default ctor
   Student(std::string name, int id, double gpa);
                // "Initializing" ctor
   ~Student();  // Destructor
   std::string get_name();
   int get_id();
   double get_gpa();

   void set_name(std::string name);
   void set_id(int id);
   void set_gpa(double gpa);
 private:
   std::string name_;
   int id_;
   double gpa_;
};
```

Note: Often name data members with special decorator (id_ or gpa_) to make it obvious to other programmers that this variable is a data member

student.cpp

```
Student::Student()
{
  name_ = "Jane Doe"; id_ = 0; gpa_ = 2.0;
}

Student::Student(string name, int id, double gpa)
{
  name_ = name; id_ = id; gpa_ = gpa;
}
```

Accessor / Mutator Methods

- Define public "get" (accessor) and "set" (mutator) functions to let other code access desired private data members
- "Good practice": Use 'const' after argument list for functions that DON'T modify data members
  - Ensures data members are not altered by this function

student.h

```
class Student {
 public:
   Student();   // Constructor 1
   Student(string name, int id, double gpa);
                // Constructor 2
   ~Student();  // Destructor
   std::string get_name() const;
   int get_id() const;
   double get_gpa() const;

   void set_name(string s);
   void set_gpa(double g);
 private:
   std::string name_;
   int id_;
   double gpa_;
};
```

student.cpp

```
#include "student.h"
using namespace std;

std::string Student::get_name() const
{   return name_; }
int Student::get_id() const
{   return id_; }
void Student::set_name(string s)
{  name_ = s;  }

void Student::set_gpa(double g)
{  gpa_ = g; }
```

```
#include<iostream>
#include "student.h"
using namespace std;
int main()
{
  Student s1;
  string myname;
  cin >> myname;
  s1.set_name (myname);
  string name_copy;
  name_copy = s1.get_name();
  ...
}
```

Writing Member Functions

- What's wrong with the code on the left vs. code on the right

```
void f1()
{
  top_index = 0;
}
```

```
Deck::Deck()
{
  top_index = 0;
}
```

- Compiler needs to know that a function is a member of a class
- Include the name of the class followed by '`::`' just before name of function
- This allows the compiler to check access to private/public variables
  - Without the scope operator [i.e. `int get_top_card()` rather than `int Deck::get_top_card()`] the compiler would think that the function is some outside function (not a member of Deck) and thus generate an error when it tried to access the data members (i.e. cards array and top_index).

deck.h

```
class Deck {
 public:
   Deck();   // Constructor
   ~Deck();  // Destructor
   void shuffle();
   ...
};
```

deck.cpp

```
#include<iostream>
#include "deck.h"
Deck::Deck() {
  top_index = 0;
  for(int i=0; i < 52; i++){
    cards[i] = i;
  }
}
Deck::~Deck()
{
}

void Deck::shuffle()
{
  cut(); //calls cut() for this object
  ...
}
int Deck::get_top_card()
{
  top_index++;
  return cards[top_index-1];
}
```

Calling Member Functions (1)

- When **outside the class scope** (i.e. in `main()` or some outside function)
  - Must precede the member function call with the object name of the specific object that it should operate on and the dot operator (e.g. `d1.shuffle()`)
  - `d1.shuffle()` indicates the code of `shuffle()` should be operating implicitly on d1's data member vs. d2 or any other Deck object

d1

cards[52]	0	1	2	3	4	5	6	7
top_index	0							

d2

cards[52]	0	1	2	3	4	5	6	7
top_index	0							

```
#include<iostream>
#include "deck.h"

int main() {
  Deck d1, d2;
  int hand[5];

  d1.shuffle();
  // not Deck.shuffle() nor
  // shuffle(d1), etc.

  for(int i=0; i < 5; i++){
    hand[i] = d1.get_top_card();
  }
}
```

d1

cards[52]	41	27	8	39	25	4	11	17
top_index	1							

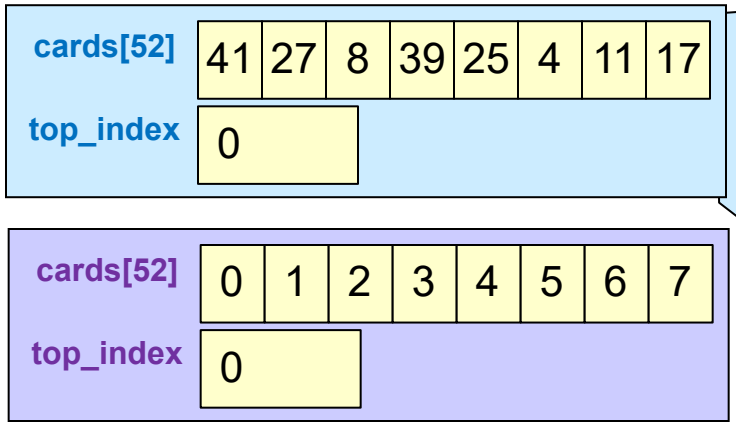
Calling Member Functions

- When **inside the class scope** (i.e. in main or some outside function), no preceding object is necessary
- Within a member function we can just call other member functions directly.

d1's data will be modified (shuffled and cut)

d1 is implicitly passed to shuffle()

Since shuffle was implicitly working on d1's data, d1 is again implicitly passed to cut()



poker.cpp

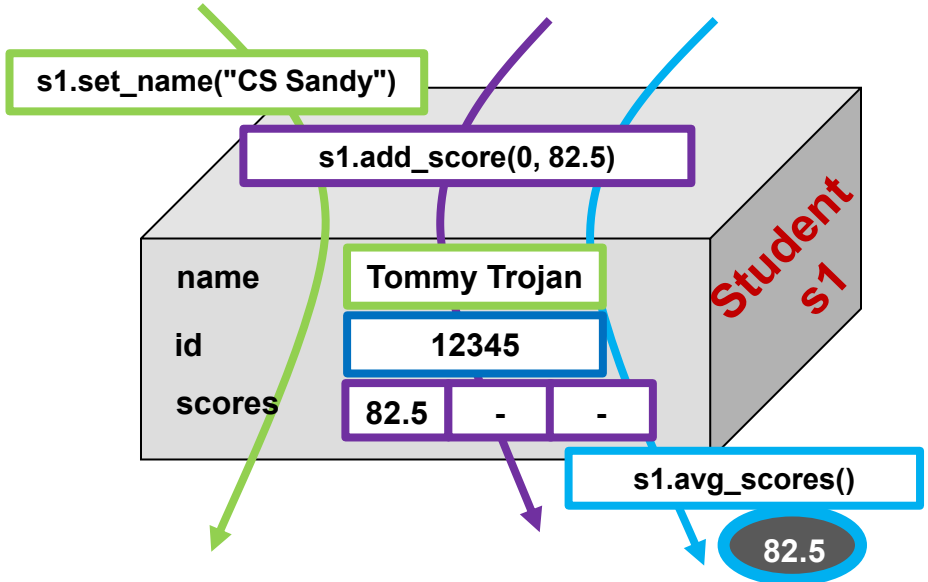
```
#include<iostream>
#include "deck.h"

int main(int argc, char *argv[]) {
  Deck d1, d2;
  int hand[5];

  d1.shuffle();
  ...
}
```

deck.cpp

```
#include<cstdlib>
#include "deck.h"

void Deck::shuffle()
{
  cut(); // calls cut()
         // for this object
  for(i=0; i < 52; i++){
    int r = rand() % (52-i);
    int temp = cards[r];
    cards[r] = cards[i];
    cards[i] = temp;
  }
}
void Deck::cut()
{
  // swap 1st half of deck w/ 2nd
}
```

Big Picture

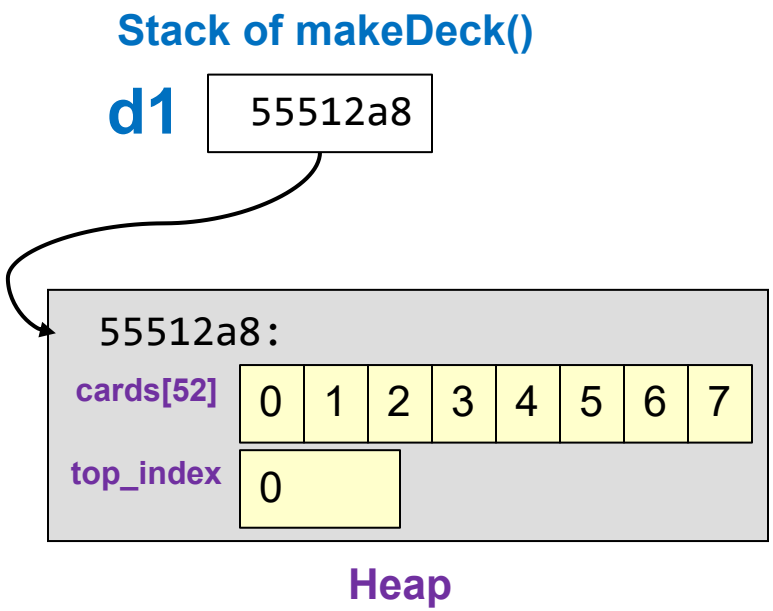
- Data members represent the "**state**" of the object
  - State = values that need to be remembered or retained across various function calls (for the lifetime of the object) to be able to perform appropriate tasks.
  - **Data members** should be the information needed by / across multiple member functions (e.g. set then get, add_score then avg_scores(), etc.)
- Member functions:
  - Modify or perform computation on that state
  - **Input arguments** of member functions should be values needed only DURING or FOR that particular function call timeline.




```
class Student {
 public:
   Student();   // Default ctor
   Student(string name, int id); // Init. ctor
   void set_name(std::string n);
   void add_score(int id, double score);
   double avg_scores() const;
 private:
   std::string name_;
   int id_;
   double scores_[10];
};

int main()
{
  Student s1("Tommy Trojan", 12345);
  s1.set_name("CS Sandy");
  s1.add_score(0, 82.5);
  cout << s1.avg_scores() << endl;
  return 0;
}
```

Class Pointers

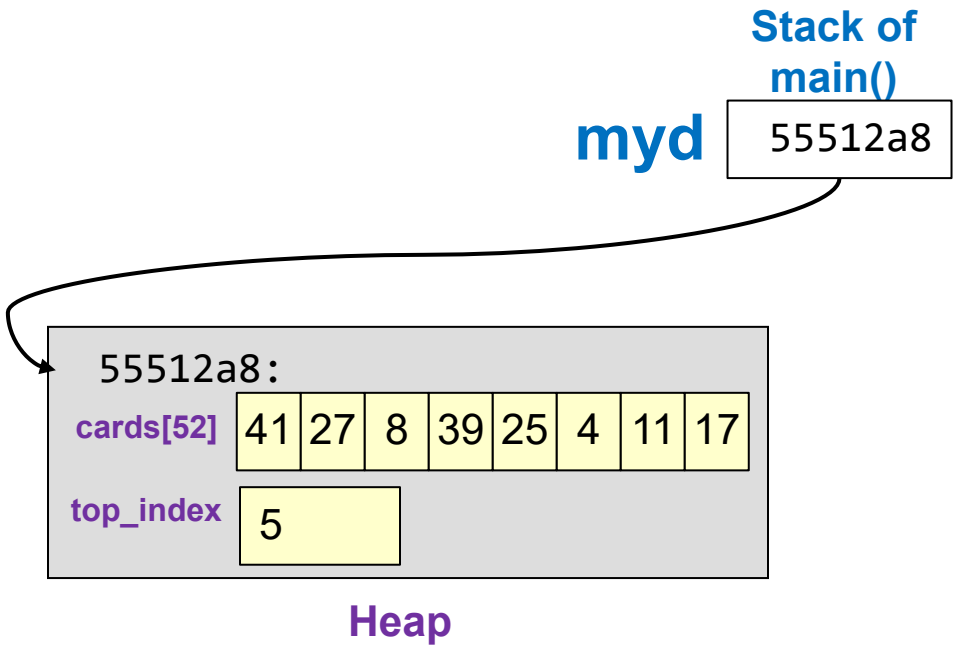
- Can declare pointers to these new class types
- Use -> operator to access member functions or data




```
#include<iostream>
#include "deck.h"

Deck* makeDeck(){
  Deck* d1 = new Deck;
  d1->shuffle();
  d1->cut();
  return d1;
}
int main()
{
 int hand[5];
 Deck* myd = makeDeck();
 for(int i=0; i < 5; i++){
    hand[i] = myd->get_top_card();
 }
 // More code

 delete myd;
 return 0;
}
```

Class Pointers

- Can declare pointers to these new class types
- Use -> operator to access member functions or data




```
#include<iostream>
#include "deck.h"

Deck* makeDeck(){
  Deck* d1 = new Deck; // Ctor called
  d1->shuffle();
  d1->cut();
  return d1;
}
int main()
{
 int hand[5];
 Deck* myd = makeDeck();
 myd->shuffle();
 for(int i=0; i < 5; i++){
    hand[i] = myd->get_top_card();
 }
 // More code

 delete myd;  // Dtor called
 return 0;
}
```

Exercises

- In-class Exercises