Pointers Part 2: Dynamic Memory Allocation
Dynamic Memory Allocation

• I want an array for student scores but I don’t know how many students we have until the user tells me
• What size should I use to declare my array?
  – int scores[??]
• Doing the following is not supported by all C++ compilers:
  ```
  int num;
  cin >> num;
  int scores[num]; // compilers require the array size to be statically known
  ```
• All compilers support allocating memory *dynamically* (i.e. at run-time)
Dynamic Memory & the Heap

• Code usually sits at low addresses
• Global variables somewhere after code
• System stack (memory for each function instance that is alive)
  – Local variables
  – Return link (where to return)
  – etc.
• Heap: Area of memory that can be allocated and de-allocated during program execution (i.e. dynamically at run-time) based on the needs of the program
• Heap grows downward, stack grows upward…
  – In rare cases of large memory usage, they could collide and cause your program to fail or generate an exception/error
C++ new & delete operators

• **new** allocates memory from heap
  – followed with the type of the variable you want or an array type declaration
     • double *dptr = new double;
     • int *myarray = new int[100];
  – can obviously use a variable to indicate array size
  – returns a pointer of the appropriate type
     • if you ask for a new int, you get an int * in return
     • if you ask for an new array (new int[10]), you get an int ** in return

• **delete** returns memory to heap
  – followed by the pointer to the data you want to de-allocate
     • delete dptr;
  – **use delete [] for arrays**
     • delete [] myarray;
int main()
{
    int *scores;
    int num;
    cout << "How many students?" << endl;
    cin >> num;
    scores = new int[num];
    // can now access scores[0] .. scores[num-1];
    return 0;
}

int main()
{
    int *scores;
    int num;
    cout << "How many students?" << endl;
    cin >> num;
    scores = new int[num];
    // can now access scores[0] .. scores[num-1];
    delete [] scores
    return 0;
}
Class Constructors / Destructors

- Constructor is a function of the same name as the class itself
  - It is called automatically when the object is created (either when declared or when allocated via ‘new’)
  - Use to initialize your object to desired state
  - Returns nothing

- Destructor is a function of the same name as class itself with a ‘~’ in front
  - Called automatically when object goes out of scope (i.e. when it is deallocated by ‘delete’ or when scope completes)
  - Use to delete any memory allocated by the object
  - Returns nothing

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

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

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

Deck::~Deck() {
}

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

int main() {
    Deck *dptr;
    dptr = new Deck();  // Deck() is called
    dptr->shuffle();
    delete dptr;  // ~Deck() is called
}
Class Pointers

- Can declare pointers to classes
- Use ‘->’ operator to access member functions or data

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

int main() {
    Deck *d1;
    int hand[5];
    d1 = new Deck();
    d1->shuffle();
    for(int i=0; i < 5; i++){
        hand[i] = d1->get_top_card();
    }
    delete d1;
}
```