chenz11@rpi.edu	Zhengneng Chen	lab section: 6
room: DARRIN 318	zone: NAVY row: 7 seat: 3L	6-7:50pm

CSCI-1200 Data Structures — Spring 2018
Exam 1 — Monday, February 4th, 6-7:50pm

	/ 3	
1	/ 35	
2	/ 26	
3	/ 16	
4	/ 20	
Total	/ 100	

Name one of your undergraduate lab mentors:

Samuel

Name your lab graduate TA:

Chelsea

- **This exam has 14 pages: 10 pages of questions, two blank pages, and the 2 pages of notes you uploaded to Submitty. Let us know if you are missing a page. DO NOT SEPARATE YOUR EXAM.**
- This test is closed-book and closed-notes *except for 2 sheets of notes on 8.5x11 inch paper (front & back) that may be handwritten or printed.* Put away all other papers and books. Computers, cell phones, calculators, PDAs, MP3 players, etc. are not permitted and **must be turned off and placed under your desk.**
- Many of the problems require you to write a C++ function. If the function declaration is not given, part of your grade will depend on specifying the parameters correctly. In solving any problem, you may write additional functions to better structure your solution.
- Write your answer in the box provided below each question. Be sure to write neatly. If we can't read your solution, we won't be able to give you full credit for your work. Please state clearly any assumptions that you made in interpreting a question.
- You can assume appropriate STL `#include` statements and `using namespace std`. You are allowed to use `std::` if you prefer.

1 Parcel Delivery [/ 35]

In the following problem you will finish the implementation of a program that is designed to keep track of several delivery drivers. Each driver is represented by a `Driver` object which has an ID, a name, a maximum capacity in kg that their vehicle can carry, and the packages they are currently carrying. Each package is represented by a `Parcel` object. For this problem, you can assume that all weights and capacities are integers, and that there will not be duplicate driver IDs.

First, here's `main.cpp` and the output that it produces:

```
#include "Driver.h"
#include "Parcel.h"
// print_drivers implemented, but not included in handout
// add_parcel written in 1.4

int main(){
   std::vector<Driver> drivers;
   drivers.push_back(Driver(124,"Chris",50));
   drivers.push_back(Driver(8,"Sam",150));
   drivers.push_back(Driver(35,"Taylor",200));

   print_drivers(drivers);
   add_parcel(drivers,Parcel("A7X",25),0);
   add_parcel(drivers,Parcel("A7X",25),8);
   add_parcel(drivers,Parcel("S41",126),8);
   add_parcel(drivers,Parcel("AK3",10),35);
   add_parcel(drivers,Parcel("P1",1),124);
   add_parcel(drivers,Parcel("P2",1),124);
   add_parcel(drivers,Parcel("P3",1),124);
   print_drivers(drivers);
   std::sort(drivers.begin(), drivers.end(), bySmallestWeight);
   print_drivers(drivers);
   return 0;
}
```

The output:

```
Driver Chris (#124) is carrying 0 of 50 kgs:
Driver Sam (#8) is carrying 0 of 150 kgs:
Driver Taylor (#35) is carrying 0 of 200 kgs:

Could not find driver #0
Added parcel A7X to driver #8
Failed to add parcel S41 to driver #8
Added parcel AK3 to driver #35
Added parcel P1 to driver #124
Added parcel P2 to driver #124
Added parcel P3 to driver #124
Driver Chris (#124) is carrying 3 of 50 kgs: #P1 (1) kg #P2 (1) kg #P3 (1) kg
Driver Sam (#8) is carrying 25 of 150 kgs: #A7X (25) kg
Driver Taylor (#35) is carrying 10 of 200 kgs: #AK3 (10) kg

Driver Chris (#124) is carrying 3 of 50 kgs: #P1 (1) kg #P2 (1) kg #P3 (1) kg
Driver Taylor (#35) is carrying 10 of 200 kgs: #AK3 (10) kg
Driver Sam (#8) is carrying 25 of 150 kgs: #A7X (25) kg
```

1.1 `Parcel` Class Declaration (`Parcel.h`) [/6]

Start by writing the class declaration in the `Parcel.h` file. The Parcel class should support the constructor used in `main.cpp`, and should have two accessors, *getWeight* and *getID*. For this problem, please do not use constructor initializer lists. You do not need to use include guards. Remember that one line functions can be written in the .h file.

```
#include <string>

class Parcel {
public:
    Parcel(std::string id="", int weight=0);
    const int& getWeight() const { return weight_; }
    const std::string getID() const { return id_; }

private:
    std::string id_;
    int weight_;
};
```

sample solution: 9 line(s) of code

1.2 `Parcel` Class Implementation (`Parcel.cpp`) [/5]

Now write the class implementation for the Parcel class in `Parcel.cpp`.

```
#include "Parcel.h"

Parcel::Parcel(std::string id="", int weight=0) {
    id_ = id;
    weight_ = weight;
}
```

sample solution: 6 line(s) of code

1.3 Completing the Driver Class Implementation (Driver.cpp) [/10]

Assume that the implementation of the Driver class is complete except for *get_current_weight* and *bySmallestWeight*. The .h file looks like this:

```
#include "Parcel.h"
class Driver{
public:
   Driver(int id, const std::string& name, int capacity);
   int getCapacity() const;
   const std::string& getName() const;
   int getID() const;
   const std::vector<Parcel>& getParcels() const;
   //getCurrentWeight definition would go here. Returns total weight the Driver is carrying.
   bool addParcel(const Parcel& p); //Returns true if parcel was added, false if it was too big.
private:
   int m_id;
   std::string m_name;
   int m_capacity;
   std::vector<Parcel> m_parcels;
};

//bySmallestWeight definition would go here. Used in main.cpp
```

Finish the .cpp file by implementing both of the missing functions:

```
const int& Driver::getCurrentWeight() const {
    int weight_count = 0;
    for (int i=0; i<getParcels().size(); ++i){
        weight_count += getParcels()[i].getWeight();
    }
    return weight_count;
}

bool bySmallestWeight(const Driver& d1, const Driver& d2){
    return d1.getCurrentWeight() < d2.getCurrentWeight();
}
```

sample solution: 11 line(s) of code

14/16

1.4 *add_parcel* Implementation (main.cpp) [6]

Finally, write the *add_parcel* function that goes in main.cpp.

```
void add_parcel (std::vector<Driver>& drivers, const Parcel& P, const int& d_id){
    for (int i=0; i < drivers.size(); ++i) {
        if (drivers[i].getID() == d_id) {
            if (!p.addParcel(p)) {
                std::cout << "Failed to add Parcel" << P.getID();
                std::cout << " to driver #" << d_id << std::endl;
            } else {
                std::cout << "Added parcel " << P.getID();
                std::cout << " to driver #" << d_id << std::endl;
            }
            break;
        }
        if (i == drivers.size() - 1) {
            std::cout << "Could not find driver #" << d_id << std::endl;
        }
    }
}
```

sample solution: 16 line(s) of code

2 Memory Diagramming [/ 26]

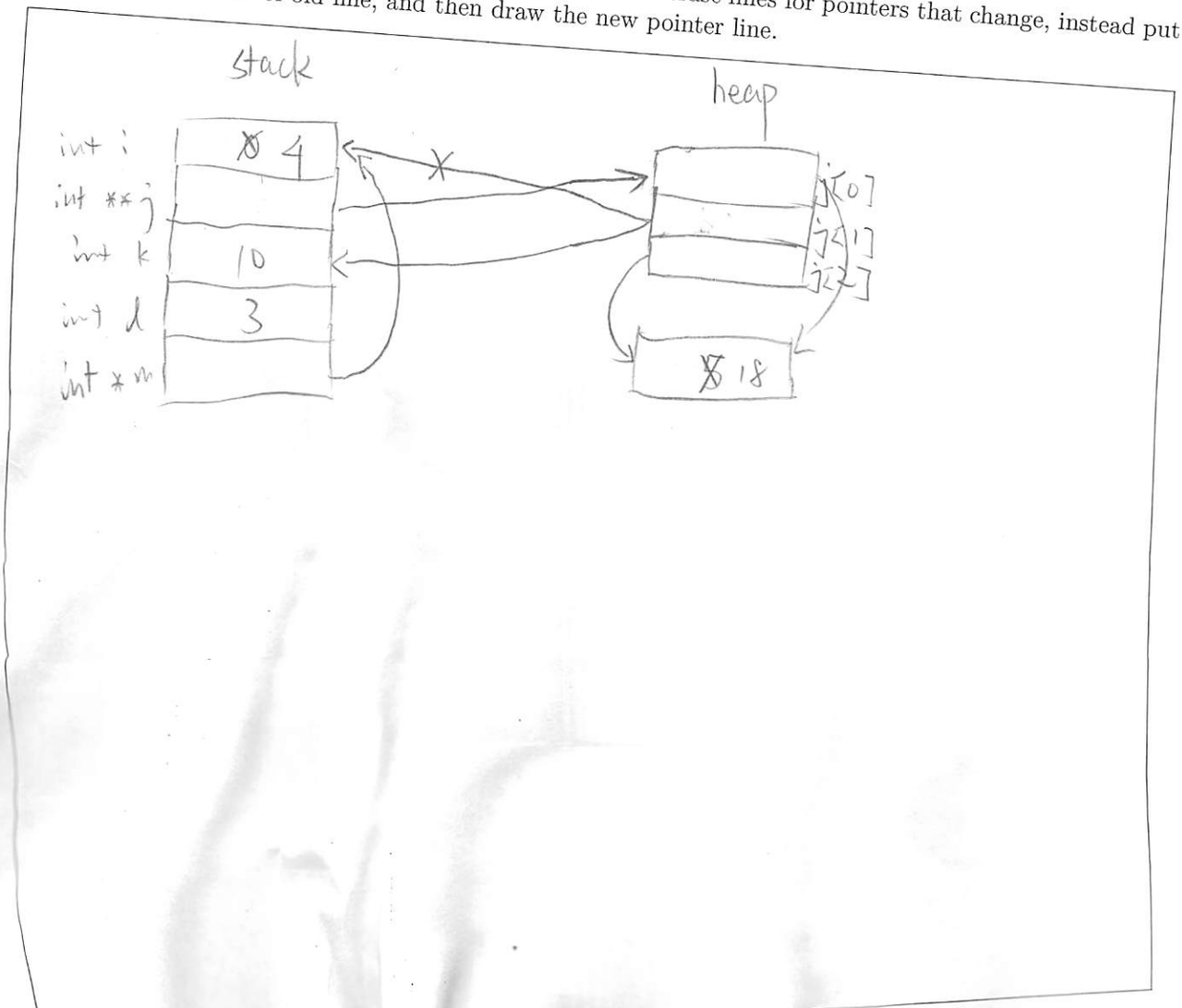
Zhengneng Chen chenz11@rpi.edu

Consider the following code:

```
int i,**j,k,l,*m;
i = 0;
j = new int*[3];
j[0] = new int;
j[1] = &i;
m = *(j+1);
j[1] = &k;
k=10;
*(j[0]) = 5;
j[2] = j[0];
*(j[0]) = 18;
*m = 4;
l = 3;
```

2.1 Memory Diagram [/18]

First, draw a memory diagram for the above code. Do not erase lines for pointers that change, instead put an x over the middle of old line, and then draw the new pointer line.



2.2 Code Output [/8]

Next, write the output running this code will give:

```
std::cout << "i: " << i << std::endl;
std::cout << "j[0]: " << *j[0] << std::endl;
std::cout << "j[1]: " << *j[1] << std::endl;
std::cout << "j[2]: " << *j[2] << std::endl;
std::cout << "k: " << k << std::endl;
std::cout << "l: " << l << std::endl;
std::cout << "m: " << *m << std::endl;
```

i: 4

j[0]: 18

j[1]: 10

j[2]: 18

k: 10

l: 3

m: 4

3 Short Answer Round [/ 16]

For each of the following statements, write if it is true or false, and then write 1-2 *complete* sentences explaining why.

3.1 Return Type [/4]

True or False If we are returning a string, we should always return using `const std::string&`.

False. The return should not always use "const" since we might like to modify return value sometimes.

3.2 const Speed [/4]

True or False Using `const` types changes how fast the program runs.

False. "const" is just protection to arguments of a function. In fact, "&" reference matters to the speed of code execution.

3.3 Reference Efficiency [/4]

True or False Passing by reference can be more efficient than passing by value.

True. Passing by value means the copy of whole argument's memory. This is a waste of time if we're trying to copy a large part of memory.

3.4 const Members [/4]

True or False Every member function should have a `const` at the end of it. (e.g. `int get_var() const`).

False. "const" is not necessary if we'd like to modify member variable which is protected by "private".

4 Phrase Counting [. / 20]

In this problem you will write a function to count how many times a string appears inside a collection of other strings. Provided below is a code fragment which examines four strings: "banana", "bandana", "cabana", and "banabanbana". For this example, the output of the function is how many times each word had the letters "bana" consecutively in it. In this case, "bandana" has 0 instances of "bana" since the letter d gets in the way.

```
std::vector<std::string> words;
words.push_back("banana");
words.push_back("bandana");
words.push_back("cabana");
words.push_back("banabanabana");

std::vector<int> counts = count_phrase(words,"bana");
for(unsigned int i=0; i<counts.size(); i++){
  std::cout << words[i] << " contains \"bana\" " << counts[i]
            << " time(s)." << std::endl;
}
```

size = 6 size = 4

The expected output in this case:

```
banana contains "bana" 1 time(s).
bandana contains "bana" 0 time(s).
cabana contains "bana" 1 time(s).
banabanabana contains "bana" 3 time(s).
```

i < 6-4+1 = i<3

0, 2

For this problem, the only STL string function you can use is *size()*. Do not use any C-style string functions (e.g. *strcmp()*).

Your answer should go in the box on the next page.

Write *count_phrase*:

```
std::vector<int> count_phrase(const std::vector<std::string>& words,
                              const std::string& letters) {
    std::vector<int> counts;
    for (int i = 0; i < words.size(); ++i) {
        int count = 0;
        for (int j=0; j< words[i].size()- letters.size()+1 ; ++j ){
            for (int k=0; k < letters.size() ; ++k) {
                if ( words[i][j+k] != letters[k]) {
                    break;
                }
                if. ( k == letters.size() -1) {
                    count += 1;
                }
            }
        }
        counts.push_back( count);
    }
    return counts;
}
```

sample solution: 21 line(s) of code

(blank page)

(blank page)

Data structure cheat sheet Exam 1

Zhengneng Chen chenz11@rpi.edu

Lecture 1 - Intro to C++, STL, & Strings

In general, the "Rules of Thumb" for using value and reference parameters:
– When a function needs to provide just one simple result, make that result the return value of the function and pass other parameters by value.
– When a function needs to provide more than one result, these results should be returned using multiple reference parameters.

The individual characters of a string can be accessed using the subscript operator [] (similar to arrays)

Lecture 2 - STL Strings & Vectors

A char array can be initialized as: $charh[] = 'H','e','l','l','o','!','\0'$; or as: $charh[] = "Hello!"$; In either case, array h has 7 characters, the last one being the null character.

The C language provides many functions for manipulating these "C-style strings". We don't study them much anymore because the "C++ style" STL string library is much more logical and easier to use. One place we do use them is in file names and command-line arguments.
You can obtain the C-style string from a standard string using the member function c_str, as in s1.c_str().

In particular, what is the difference between the use of a[0] on the left hand side of the assignment statement and b[0] on the right hand side? Syntactically, they look the same. But,
– The expression b[0] gets the char value, 'T', from string location 0 in b. This is an r-value.
– The expression a[0] gets a reference to the memory location associated with string location 0 in a. This is an l-value.
– The assignment operator stores the value in the referenced memory location.
The difference between an r-value and an l-value will be especially significant when we get to writing our own operators later in the semester.

A vector acts like a dynamically-sized, one-dimensional array. Capabilities:
– Holds objects of any type
– Starts empty unless otherwise specified
– Any number of objects may be added to the end — there is no limit on size. – It can be treated like an ordinary array using the subscripting operator.
– A vector knows how many elements it stores! (unlike C arrays)
– There is NO automatic checking of subscript bounds.

If you are passing a vector as a parameter to a function and you want to make a (permanent) change to the vector, then you should pass it by reference.
What if you don't want to make changes to the vector or don't want these changes to be permanent?
– The answer we've learned so far is to pass by value.
– The problem is that the entire vector is copied when this happens! Depending on the size of the vector, this can be a considerable waste of memory.
The solution is to pass by constant reference: pass it by reference, but make it a constant so that it can not be changed.
As a general rule, you should not pass a container object, such as a vector or a string, by value because of the cost of copying.

Lecture 3 - Classes I

What is a type?
– It is a structuring of memory plus a set of operations that can be applied to that structured memory.
– Every C++ object has a type (e.g., integers, doubles, strings, and vectors, etc., including custom types). – The type tells us what the data means and what operations can be performed on the data.
The basic ideas behind classes are data abstraction and encapsulation:
– In many cases when we are using a class, we don't know (don't need to know) exactly how that memory is structured. Instead, what we really think about what operations are available.
– Data abstraction hides details that don't matter from the end user and identifies details that do matter.
– The user sees only the interface to the object: the collection of externally-visible data and operations.
– Encapsulation is the packing of data and functions into a single component.
Information hiding
– Users have access to interface, but not implementation.
– No data item should be available any more than absolutely necessary.

Within class scope, the member functions and member variables are accessible without the name of the object.
Member functions are like ordinary functions except:
• They can access and modify the object's member variables.
• They can call the other member functions without using an object name.
• Their syntax is slightly different because they are defined within class scope.

Member functions that do not change the member variables should be declared const
• For example: bool Date::isEqual(const Date &date2) const;
• This must appear consistently in both the member function declaration in the class declaration (in the .h file) and in the member function definition (in the .cpp file).
• const objects (usually passed into a function as parameters) can ONLY use const member functions. Remember, you should only pass objects by value under special circumstances. In general, pass all objects by reference so they aren't copied, and by const reference if you don't want/need them to change.

Good software design requires a lot of practice, but here are some ideas to start from:
• Begin by outlining what the class objects should be able to do. This gives a start on the member functions.
• Outline what data each object keeps track of, and what member variables are needed for this storage.
• Write a draft class declaration in a .h file.
• Write code that uses the member functions (e.g., the main function). Revise the class .h file as necessary.
• Write the class .cpp file that implements the member functions.

• Operators should only be defined if their meaning is intuitively clear.
• Sorting Date objects makes sense because arguably chronological is the only natural, universally agreed-upon way to do this.
• Similarly, sorting STL string objects makes sense because alphabetical is the accepted order. So yes, the STL
string class has overloaded operator<, and that's why sorting them works like magic.
• In contrast, if we defined a Person class (storing their name, address, social security number, favorite color, etc.), we probably wouldn't agree on how to sort a vector of people. Should it be by name? Or by age? Or by height? Or by income?
Instead, it would be better to have comparison helper functions that can be used as needed. E.g., alpha_by_name, youngest_first, tallest_first, etc.

1 Lecture 4 - Classes II: Sort, Non-member Operators

• Note the helpful convention used in this example: all member variable names end with the "_" character.
• The special pre-processor directives #ifndef __student_h_, #define __student_h_, and #endif ensure that this files is included at most once per .cpp file.
For larger programs with multiple class files and interdependencies, these lines are essential for successful compilation. We suggest you get in the habit of adding these include guards to all your header files.

• The accessor functions for the names are defined within the class declaration in the header file. In this course, you are allowed to do this for one-line functions only! For complex classes, including long definitions within the header file has dependency and performance implications.

2 Lecture 5 - Pointers, Arrays, & Pointer Arithmetic

• x is an ordinary float, but p is a pointer that can hold the memory address of a float variable. The difference is explained in the picture above.
• Every variable is attached to a location in memory. This is where the value of that variable is stored. Hence, we draw a picture with the variable name next to a box that represents the memory location.

• Each memory location also has an address, which is itself just an index into the giant array that is the computer memory.
• The value stored in a pointer variable is an address in memory. The statement p = &x; takes the address of x's memory location and stores it (the address) in the memory location associated with p.
• Since the value of this address is much less important than the fact that the address is x's memory location, we depict the address with an arrow.
• The statement: *p = 72; causes the computer to get the memory location stored at p, then go to that memory location, and store 72 there. This writes the 72 in x's location.
Note: *p is an l-value in the above expression.

• In the example below, p, s and t are all pointer variables (pointers, for short), but q is NOT. You need the * before each variable name.
• There is no initialization of pointer variables in this two-line sequence, so the statement below is dangerous, and may cause your program to crash! (It won't crash if the uninitialized value happens to be a legal address.)

• The unary (single argument/operand) operator * in the expression *p is the "dereferencing operator". It means "follow the pointer" *p can be either an l-value or an r-value, depending on which side of the = it appears on.
• The unary operator & in the expression &x means "take the memory address of."
• Pointers can be assigned. This just copies memory addresses as though they were values (which they are).
• Assignments of integers or floats to pointers and assignments mixing pointers of different types are illegal.
• Comparisons between pointers of the form if (p == q) or if (p != q) are legal and very useful! Less than and greater than comparisons are also allowed. These are useful only when the pointers are to locations within an array.

• Like the int type, pointers are not default initialized. We should assume it's a garbage value, leftover from the previous user of that memory.
• Pointers that don't (yet) point anywhere useful are often explicitly assigned to NULL.
– NULL is equal to the integer 0, which is a legal pointer value (you can store NULL in a pointer variable).
– But NULL is not a valid memory location you are allowed to read or write. If you try to dereference or follow a NULL pointer, your program will immediately crash. You may see a segmentation fault, a bus error, or something about a null pointer dereference.
– NOTE: In C++11, we are encouraged to switch to use nullptr instead of NULL or 0, to avoid some subtle situations where NULL is incorrectly seen as an int type instead of a pointer. For this course we will assume NULL and nullptr are equivalent.
– We indicate a NULL or nullptr value in diagrams with a slash through the memory location box.
• Comparing a pointer to NULL is very useful. It can be used to indicate whether or not a pointer variable is pointing at a useable memory location.
• But don't make the mistake of assuming pointers are automatically initialized to NULL.

• The array code above that uses [] subscripting, can be equivalently rewritten to use pointers.
• The assignment: p = a; takes the address of the start of the array and assigns it to p.
• This illustrates the important fact that the name of an array is in fact a pointer to the start of a block of memory. We will come back to this several times! We could also write this line as: p = &a[0]; which means "find the location of a[0] and take its address".
• By incrementing, ++p, we make p point to the next location in the array.
– When we increment a pointer we don't just add one byte to the address, we add the number of bytes (sizeof) used to store one object of the specific type of that pointer. Similarly, basic addition/subtraction of pointer variables is done in multiples of the sizeof the type of the pointer.
– Since the type of p is double, and the size of double is 8 bytes, we are actually adding 8 bytes to the address when we execute ++p.
• The test p<a+n checks to see if the value of the pointer (the address) is less than n array locations beyond the start of the array. We could equivalently have used the test p != a+n
• Note that there may or may not be unused memory between your array and the other local variables. Similarly, the order that your local variables appear on the stack is not guaranteed (the compiler may rearrange things a bit in an attempt to optimize performance or memory usage). A buffer overflow (attempting to access an illegal array index) may or may not cause an immediate failure, depending on the layout of other critical program memory.

Zhengheng Chen chenz211@rpi.edu

• Arrays may be sorted using std::sort, just like vectors. Pointers are used in place of iterators. For example, if a is an array of doubles and there are n values in the array, then here's how to sort the values in the array into increasing order: std::sort(a, a+n);

3 Lecture 6 -

Automatic memory: memory allocation inside a function when you create a variable. This allocates space for local variables in functions (on the stack) and deallocates it when variables go out of scope.
• Dynamic memory: explicitly allocated (on the heap) as needed.

• The expression new int asks the system for a new chunk of memory that is large enough to hold an integer and returns the address of that memory. Therefore, the statement int * p = new int; allocates memory from the heap and stores its address in the pointer variable p.
• The statement delete p; takes the integer memory pointed by p and returns it to the system for re-use.
• This memory is allocated from and returned to a special area of memory called the heap. By contrast, local variables and function parameters are placed on the stack as discussed last lecture.
• In between the new and delete statements, the memory is treated just like memory for an ordinary variable, except the only way to access it is through pointers. Hence, the manipulation of pointer variables and values is similar to the examples covered in Lecture 5 except that there is no explicitly named variable for that memory other than the pointer variable.
• Dynamic allocation of primitives like ints and doubles is not very interesting or significant. What's more important is dynamic allocation of arrays and objects.

• The expression new double[n] asks the system to dynamically allocate enough consecutive memory to hold n double's (usually 8n bytes).
– What's crucially important is that n is a variable. Therefore, its value and, as a result, the size of the array are not known until the program is executed and the the memory must be allocated dynamically.
– The address of the start of the allocated memory is assigned to the pointer variable a.
• After this, a is treated as though it is an array. For example: a[i] = sqrt(i); In fact, the expression a[i] is exactly equivalent to the pointer arithmetic and dereferencing expression *(a+i) which we have seen several times before.
• After we are done using the array, the line: delete [] a; releases the memory allocated for the entire array and calls the destructor (we'll learn about these soon!) for each slot of the array. Deleting a dynamically allocated array without the [] is an error (but it may not cause a crash or other noticeable problem, depending on the type stored in the array and the specific compiler implementation).
– Since the program is ending, releasing the memory is not a major concern. However, to demonstrate that you understand memory allocation & deallocation, you should always delete dynamically allocated memory in this course, even if the program is terminating.
– In more substantial programs it is ABSOLUTELY CRUCIAL. If we forget to release memory repeatedly the program can be said to have a memory leak. Long-running programs with memory leaks will eventually run out of memory and crash.