

1. (15 points) Pointers & Dynamic Arrays: Write a function that takes as parameters the pointers of two sorted integer arrays and the two array sizes, and returns the pointer of a new integer array that is the sorted merging of the two arrays. For example, if the two parameter arrays are [3, 7, 10, 16, 22] and [1, 2, 5, 10, 12, 17] the resulting array is [1, 2, 3, 5, 7, 10, 10, 12, 16, 17, 22].

**Solution:**

```
int *merge(int A[], int B[], int LenA, int LenB) {
    int *ret = new int[LenA + LenB];

    int aptr = 0;
    int bptr = 0;
    int i = 0;

    while (aptr < LenA && bptr < LenB)
        if (A[aptr] < B[bptr])
            ret[i++] = A[aptr++];
        else
            ret[i++] = B[bptr++];

    while (aptr < LenA)
        ret[i++] = A[aptr++];

    while (bptr < LenB)
        ret[i++] = B[bptr++];

    return ret;
}
```

2. (20 points) Basic Classes: Write the specification and implementation (no inline code) for a Rectangle class that stores a height and width (decimal values possible), constructor that takes the height and width as well as a default constructor that sets height and width to 1 each. The non-default constructor should error check the input so that any negative input is changed to 0. Accessors and mutators, mutators should do the same error checking. A member function for the area and a member function for the perimeter.

**Solution:**

```
1 #ifndef RECTANGLE_H_
2 #define RECTANGLE_H_
3
4 #include <iostream>
5
6 using namespace std;
7
8 class Rectangle {
9     private:
10         double height;
11         double width;
12
13     public:
14         Rectangle(double h = 1, double w = 1);
15         ~Rectangle();
16
17         void setHeight(double);
18         double getHeight();
19         void setWidth(double);
20         double getWidth();
21         double Area();
22         double Perimeter();
23 };
24
25 #endif
26
27 #include <iostream>
28
29 #include "Rectangle.h"
30
31 using namespace std;
32
33 Rectangle::Rectangle(double h, double w) {
34     if (h < 0)
35         h = 0;
36     height = h;
37
38     if (w < 0)
39         w = 0;
40     width = w;
41 }
42
43 Rectangle::~Rectangle() {}
44
45 void Rectangle::setHeight(double h) {
46     if (h < 0)
47         h = 0;
48     height = h;
49 }
50
51 double Rectangle::getHeight() { return height; }
52
53 void Rectangle::setWidth(double w) {
54     if (w < 0)
55         w = 0;
56     width = w;
57 }
58
59 double Rectangle::getWidth() { return width; }
60
61 double Rectangle::Area() { return height * width; }
62
63 double Rectangle::Perimeter() { return 2 * (height + width); }
```

3. (20 points) Inheritance & Polymorphism: In this exercise we construct class structures, automobile, car, and truck. automobile is to be the base class and the other two will be derived classes off of this. For the mutators listed below you do not need to do any error checking on the values.

- automobile is to store the name of the auto that can be set by its constructor but defaults to “auto”. It also stores the mileage of the vehicle. It also has a destructor, member functions to access and change the name and mileage, and a virtual function toString that will print out the name and mileage.
- car inherits off of automobile, it further stores the number of doors the car has. It has a constructor allowing input of a name and a default that uses “car” as the name. It also has a destructor and accessor and mutator for the doors. Its toString function will, in addition to printing out the name and mileage, will print out the number of doors.
- truck inherits off of automobile, it further stores the length of the truck bed. It has a constructor allowing input of a name and a default that uses “truck” as the name. It also has a destructor and accessor and mutator for the bed length. Its toString function will, in addition to printing out the name and mileage, will print out the length of the bed.

With these classes and functions in place the following program will produce the output below.

```
#include <iostream>
#include <vector>

#include "auto.h"
#include "car.h"
#include "truck.h"

using namespace std;

int main() {
    vector<automobile *> autos;
    car *carptr = new car;
    carptr->setDoors(4);
    carptr->setMiles(80000);
    autos.push_back(carptr);

    truck *truckptr = new truck("Sport Pickup");
    truckptr->setBedLength(6);
    truckptr->setMiles(35000);
    autos.push_back(truckptr);

    truckptr = new truck;
    truckptr->setBedLength(8);
    truckptr->setMiles(125300);
    autos.push_back(truckptr);

    carptr = new car("Corvette");
    carptr->setDoors(2);
    carptr->setMiles(15200);
    autos.push_back(carptr);

    for (auto v : autos)
        cout << v->toString() << endl;

    return 0;
}
```

Output:

```
car Miles: 80000 Doors: 4
Sport Pickup Miles: 35000 Bed Length: 6
truck Miles: 125300 Bed Length: 8
Corvette Miles: 15200 Doors: 2
```

**Solution:**

```

1  #ifndef AUTO_H_
2  #define AUTO_H_
3
4  #include <iostream>
5
6  using namespace std;
7
8  class automobile {
9      protected:
10         string name;
11         int miles;
12
13     public:
14         automobile(string n = "auto");
15         virtual ~automobile();
16         string getName();
17         void setName(string n);
18         int getMiles();
19         void setMiles(int m);
20         virtual string toString();
21 };
22
23 #endif

```

---

```

1  #include <iostream>
2
3  #include "auto.h"
4
5  automobile::automobile(string n) { name = n; }
6  automobile::~~automobile() {}
7  string automobile::getName() { return name; }
8  void automobile::setName(string n) { name = n; }
9  int automobile::getMiles() { return miles; }
10 void automobile::setMiles(int m) { miles = m; }
11 string automobile::toString() { return name + " Miles: " + to_string(miles); }

```

---

```

1  #ifndef CAR_H_
2  #define CAR_H_
3
4  #include <iostream>
5
6  #include "auto.h"
7
8  using namespace std;
9
10 class car : public automobile {
11     protected:
12         int doors;
13
14     public:
15         car(string n = "car");
16         ~car();
17         int getDoors();
18         void setDoors(int m);
19         string toString();
20 };
21
22 #endif

```

---

```

1  #include <iostream>
2
3  #include "car.h"
4
5  using namespace std;
6
7  car::car(string n) : automobile(n) {}
8  car::~~car() {}
9  int car::getDoors() { return doors; }
10 void car::setDoors(int m) { doors = m; }

```

```
11 string car::toString() {
12     return automobile::toString() + "   Doors: " + to_string(doors);
13 }
```

---

```
1 #ifndef TRUCK_H_
2 #define TRUCK_H_
3
4 #include <iostream>
5
6 #include "auto.h"
7
8 using namespace std;
9
10 class truck : public automobile {
11     protected:
12         int bedlength;
13
14     public:
15         truck(string n = "truck");
16         ~truck();
17         int getBedLength();
18         void setBedLength(int m);
19         string toString();
20 };
21
22 #endif
23
24 #include <iostream>
25
26 #include "truck.h"
27
28 truck::truck(string n) : automobile(n) {}
29 truck::~~truck() {}
30 int truck::getBedLength() { return bedlength; }
31 void truck::setBedLength(int m) { bedlength = m; }
32 string truck::toString() {
33     return automobile::toString() + "   Bed Length: " + to_string(bedlength);
34 }
```

4. (20 points) Operator Overloading: Say we have a class structure called `Point` that represents a point  $(x, y, z)$  in three dimensions. The data,  $x$ ,  $y$ , and  $z$  are stored in an array of three doubles, specifically, `double P[3];` is its declaration. Write the following overloads for the `+`, `-` and `*` operators. Give both the specification and implementation of each with no inline code. You may assume that the `Point` class has a non-default constructor that takes three double parameters, specifically, `Point p(x, y, z);` is a valid creation of the point `p`.

- The addition of two points is another point and is done by adding the respective  $x$ ,  $y$ , and  $z$  values. So in mathematical notation,  $(x_1, y_1, z_1) + (x_2, y_2, z_2) = (x_1 + x_2, y_1 + y_2, z_1 + z_2)$ .
- The subtraction of two points is another point and is done by subtracting the respective  $x$ ,  $y$ , and  $z$  values. So in mathematical notation,  $(x_1, y_1, z_1) - (x_2, y_2, z_2) = (x_1 - x_2, y_1 - y_2, z_1 - z_2)$ .
- Multiplying a number and a point (called scalar multiplication) is just multiplying each entry by that number. So in mathematical notation,  $a \cdot (x_1, y_1, z_1) = (a \cdot x_1, a \cdot y_1, a \cdot z_1)$ . This can also be done on either side of the point, that is,  $(x_1, y_1, z_1) \cdot a = (a \cdot x_1, a \cdot y_1, a \cdot z_1)$ .

**Solution:**

```

1 Point operator+(const Point&);
2 Point operator-(const Point&);
3 Point operator*(double);
4 friend Point operator*(double, const Point&);

1 Point Point::operator+(const Point &pt) {
2     Point p(P[0] + pt.P[0], P[1] + pt.P[1], P[2] + pt.P[2]);
3     return p;
4 }
5
6 Point Point::operator-(const Point &pt) {
7     Point p(P[0] - pt.P[0], P[1] - pt.P[1], P[2] - pt.P[2]);
8     return p;
9 }
10
11 Point Point::operator*(double a) {
12     Point p(a * P[0], a * P[1], a * P[2]);
13     return p;
14 }
15
16 Point operator*(double a, const Point &rhs) {
17     Point p = rhs;
18     return p * a;
19 }

```

## 5. (20 points) Linked Lists:

- (a) Write an `appendNode` function for the templated linked list. The function will take in a single parameter of the templated type, create the node of `ListNode` type and attach it to the end of the list. The start of the list is a pointer called `head`.

**Solution:**

```

1  template<class T> void LinkedList<T>::appendNode(T newValue) {
2      ListNode<T> *newNode;
3      ListNode<T> *nodePtr;
4
5      newNode = new ListNode<T>(newValue);
6
7      if (!head)
8          head = newNode;
9      else {
10         nodePtr = head;
11
12         while (nodePtr->next)
13             nodePtr = nodePtr->next;
14
15         nodePtr->next = newNode;
16     }
17 }
```

- (b) Write a copy constructor for the templated linked list class. Assume the class name is `LinkedList`.

**Solution:**

```

1  template <class T> LinkedList<T>::LinkedList(const LinkedList &obj) {
2      head = nullptr;
3      ListNode<T> *ptr = obj.head;
4
5      while (ptr){
6          appendNode(ptr->value);
7          ptr = ptr->next;
8      }
9  }
```

6. (15 points) Stacks, Queues, and STL: Write a function called `balanced` that takes in a string (which represents a mathematical expression) and uses an STL stack to determine if the parentheses are balanced. The function will return `true` if the parentheses are balanced and `false` otherwise. For example, an input of  $(x+4/(x-1)) * (y+7) / (z * (x + 4*y))$  would return `true` and an input of  $x+4/(x-1))$  would return `false`.

**Solution:**

```

1  bool balanced(string exp) {
2      stack<char> perst;
3
4      for (unsigned long i = 0; i < exp.length(); i++) {
5          char ch = exp[i];
6          if (ch == '(')
7              perst.push(ch);
8          else if (ch == ')') {
9              if (perst.empty())
10                 return false;
11                 char top = perst.top();
12                 if (top != '(')
13                     return false;
14                 perst.pop();
15             }
16         }
17
18         if (perst.empty())
19             return true;
20         else
21             return false;
22     }
```

7. (10 points) Recursion: The  $D$  sequence is defined to be  $D(1) = 1$ ,  $D(2) = 1$ , and

$$D(n) = D(D(n-1)) + D(n-1-D(n-2))$$

Write a recursive function  $D$  that takes in a single long parameter  $n$  and returns a long that is the value of  $D(n)$ .

**Solution:**

```
1 long D(long n) {
2     if (n <= 0)
3         return 0;
4
5     if (n == 1 || n == 2)
6         return 1;
7
8     return D(D(n - 1)) + D(n - 1 - D(n - 2));
9 }
```

8. (15 points) Binary Search Trees:

- (a) Write the recursive insert function for the templated binary search tree class. This function is to take a pointer to a tree node and a pointer to a new node and insert the new node in the correct position in the BST. That is, if we were to run the following code, the insert function will put this new node in the correct position in the tree.

```
1 TreeNode *newNode = new TreeNode;
2 newNode->value = item;
3 newNode->left = newNode->right = nullptr;
4 insert(root, newNode);
```

**Solution:**

```
1 template<class T>
2 void BinaryTree<T>::insert(TreeNode *&nodePtr, TreeNode *&newNode) {
3     if (nodePtr == nullptr)
4         nodePtr = newNode;           // Insert the node.
5     else if (newNode->value < nodePtr->value)
6         insert(nodePtr->left, newNode); // Search the left branch
7     else
8         insert(nodePtr->right, newNode); // Search the right branch
9 }
```

- (b) Write an in-order traversal function for the templated binary search tree class that displays the node value to the console screen.

**Solution:**

```
1 template<class T>
2 void BinaryTree<T>::displayInOrder(TreeNode *nodePtr) const {
3     if (nodePtr) {
4         displayInOrder(nodePtr->left);
5         cout << nodePtr->value << endl;
6         displayInOrder(nodePtr->right);
7     }
8 }
```

9. (15 points) AVL Trees:

- (a) What criteria is the balancing of the AVL tree determined by?
- (b) Draw the graphs of the four subtree scenarios that cause a node insertion into an AVL tree to become unbalanced. For each, state the rotations (in order) that need to be done to balance the tree. Draw graphs of the final subtrees and all intermediate subtrees during the rotation process. Make sure that the nodes of the subtrees are labeled so that the reader can see where each node goes at each step.

10. (15 points) Function Pointers: Write a function for the templated binary tree class, called `apply` that will take in as its only parameter a pointer to a function. The parameter function will be void, have a single parameter sent by reference, and update its value. Have this `apply` function call a recursive version of `apply` that will visit every node in the tree and apply the function to the value of the node. Of course, write the recursive `apply` function as well.

**Solution:**

```

1 void apply(void (*fct) (T &));
2 void apply(TreeNode *, void (*fct) (T &));
3
4 template <class T> void BinaryTree<T>::apply(void (*fct) (T &)) {
5     apply(root, fct);
6 }
7
8 template <class T>
9 void BinaryTree<T>::apply(TreeNode *nodePtr, void (*fct) (T &)) {
10     if (!nodePtr)
11         return;
12
13     apply(nodePtr->left, fct);
14     fct(nodePtr->value);
15     apply(nodePtr->right, fct);
16 }

```

11. (15 points) Complexity:

- (a) State the precise mathematical definitions of Big- $O$ , Big- $\Omega$ , and Big- $\Theta$ . Also give the common meaning of each, specifically, what bound does it indicate?

**Solution:**

- A function  $g(n)$  is  $O(f(n))$  if there exist a constants  $c > 0$  and  $n_0$  such that, for every  $n > n_0$ ,  $|g(n)| \leq cf(n)$ . This is an Upper Bound on the complexity.
- A function  $g(n)$  is  $\Omega(f(n))$  if there exist a constants  $c > 0$  and  $n_0$  such that, for every  $n > n_0$ ,  $|g(n)| \geq cf(n)$ . This is a Lower Bound on the complexity.
- A function  $g(n)$  is  $\Theta(f(n))$  if there exist a constants  $c_1 > 0$ ,  $c_2 > 0$ , and  $n_0$  such that, for every  $n > n_0$ ,  $c_1f(n) \leq |g(n)| \leq c_2f(n)$ . This is a Tight Bound on the complexity.

- (b) Fill out the time complexity table below.

**Solution:**

Algorithm	Best	Average	Worst
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$
Quick Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$
Merge Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$
Tree Sort with BST	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$
Linear Search on Array	$\Omega(1)$	$\Theta(n)$	$O(n)$
Binary Search on Sorted Array	$\Omega(1)$	$\Theta(\log(n))$	$O(\log(n))$



## 12. (10 points) Maps:

- (a) In the following function, write a declaration for the variable `it` without using `auto` or any initializing that when placed above the loop would allow the removal of the `auto` keyword.

```

1 template <class T, class U> void PrintMap(map<T, U> M) {
2     for (auto it = M.begin(); it != M.end(); it++) {
3         cout << it->first << " " << it->second << endl;
4     }
5 }

```

Specifically, what would replace `// Declare it.` in the code below.

```

1 template <class T, class U> void PrintMap(map<T, U> M) {
2     // Declare it.
3     for (it = M.begin(); it != M.end(); it++) {
4         cout << it->first << " " << it->second << endl;
5     }
6 }

```

**Solution:**

```

1 typename map<T, U>::iterator it;

```

- (b) Write a main that creates a map using both the key and value as strings. The key will be the person's last name and the value will be the person's first name. Load into the map the following names in the following order: Don Spickler, John Doe, Jack Allen, Jane Doe, Andrew Jackson. Then have the main call the above `PrintMap` function. Show the output that would be printed to the console.

**Solution:**

```

1 int main() {
2     map<string, string> M;
3     M["Spickler"] = "Don";
4     M["Doe"] = "John";
5     M["Allen"] = "Jack";
6     M["Doe"] = "Jane";
7     M["Jackson"] = "Andrew";
8
9     PrintMap(M);
10
11     return 0;
12 }

```

```

Allen Jack
Doe Jane
Jackson Andrew
Spickler Don

```

13. (10 points) Heaps: Given the specification for the Heap class below,

```

1  template <class T> class Heap {
2      private:
3          vector<T> data;
4          int parent(int i) { return (i - 1) / 2; }
5          int left(int i) { return 2 * i + 1; }
6          int right(int i) { return 2 * i + 2; }
7          void Heapify(int i);
8      public:
9          Heap(){};
10         void insert(T);
11         T dequeue();
12 };

```

Write insert, dequeue, and Heapify without using functions from the algorithm library.

**Solution:**

```

1  template <class T> void Heap<T>::insert(T item) {
2      data.push_back(item);
3
4      int i = data.size() - 1;
5      while (i != 0 && data[parent(i)] < data[i]) {
6          swap(data[parent(i)], data[i]);
7          i = parent(i);
8      }
9  }
10
11 template <class T> void Heap<T>::Heapify(int i) {
12     int largest = i;
13     int l = left(i);
14     int r = right(i);
15
16     if (l < data.size() && data[l] > data[largest]) {
17         largest = l;
18     }
19
20     if (r < data.size() && data[r] > data[largest]) {
21         largest = r;
22     }
23
24     if (largest != i) {
25         swap(data[i], data[largest]);
26         Heapify(largest);
27     }
28 }
29
30 template <class T> T Heap<T>::dequeue() {
31     T retitem = data[0];
32     data[0] = data[data.size() - 1];
33     data.pop_back();
34     Heapify(0);
35     return retitem;
36 }

```