COMP 151 – Lectures 1, 2, 3
Final Exam – May 23, 2001
Exam Duration – 3 hours

Name: ______________________ E-mail: ______________________
Student No: ______________________ Lecture/Lab Section: ______________________

Answer all of the following questions. Do your work on the exam paper.
If necessary, you may write on the back of the sheets.
The last three blank sheets of this exam may be torn off and used as scratch paper.

1. **(5 points)** Consider the following class and function definitions:

```
#include <iostream>
class B
{ public:
     virtual void display()
       {cout << " Displaying B" <<endl;}
};

class D: public B
{  public:
     virtual void display()
       {cout << " Displaying D "<<endl;}
};

void f( B & b)
     { b.display();}

void g( B b)
     { b.display();}
```

Now, assume the following object declarations:

```
D d;
B b=d;
B * p = &d;
```

Assume the above definitions and declarations.
For each of the five function calls listed below, write down the output generated. *1 pt each*

(a) `b.display();` *Displaying B*

(b) `p->display();` *Displaying D*

(c) `f(d);` *Displaying D*

(d) `g(d);` *Displaying B*

(e) `f(b);` *Displaying B*

2. **(10 points)** In this question there are two classes **B** and D. Their class definitions are contained in file **D.h**. There is also a file **D.cpp** containing the implementation of `D`'s member functions that you will have to write.

```
//*****************  file D.h ********************************
class B
{ public:
    B(int val)  {x = val;}
    int get_x() {return x;}
    virtual void return_x() = 0;
  private:
  int x;
};

class D: public B
{ public:
  D (int val);              // sets the B::x member of D to val
  virtual void return_x();  // prints the B::x part of D
};
//***************** End of D.h *******************************
```

As an example: when the following driver is compiled together with **D.cpp** and then run, the appended output is printed.

```
#include "D.h"
void main()
{D d(5);
 d.return_x();
}
OUTPUT
------
B::x member = 5
```

To answer this question you need to write the file **D.cpp** that contains the implementations of **D**'s member functions.

```
//******************  file D.cpp ************************
#include <iostream>
#include "D.h"

// Write the implementation of D's constructor below this line

(7 pts)
D::D(int val): B(val){};


//Note: This is the only way of writing the constructor.
//It is not possible to write something such as
//D::D(int val): {B::x = val};



// Write the implementation of D's return_x() below this line

(3 pts)
void D::return_x() { cout << "B::x member = " << get_x() << endl;}


//Note:  Since x is private in B , get_x() must be used
//to access its value.
```

3. **(15 points)** This question concerns the following piece of code:

```
#include <iostream>

class B
{public:
   B()  {x = new int; *x = 5;  cout << "CB" << endl;}
  ~B() {delete x;              cout << "DB" << endl;}

  void display(){cout << "*x = " << *x << endl;}
private:
  int *x;
};

class D : public B
{public:
   D()  {y = new int; *y = 3;  cout << "CD" << endl;}
   ~D() {delete y;             cout << "DD" << endl;}

   void display(){cout << "*y = " << *y << endl;}
 private:
  int* y;
};


 void main()
 {D d;
  d.display();
  cout << "*******1*******" << endl;
  B * p = new D;
  p->display();
  cout << "*******2*******" << endl;
  delete p;
  cout << "*******3*******" << endl;
 }
```

(a) Write the complete output of the program: *6 pts*

```
OUTPUT
------
CB
CD
*y = 3
*******1*******
CB
CD
*x = 5
*******2*******
DB
*******3*******
DD
DB
```

(b) The program has a memory leak. Identify it and explain why it occurs. *4 pts*

The memory leak occured in line 25 when `delete p` *was called. In line 22 when* `B * p = new D;` *was called* slicing *occurred. Since* `p` *is a* B pointer *that was directed towards* D object `d`*, when* `d`*, is accessed via* `p` *only its B members can be accessed. In particular, when* `delete p` *is called, the B destructor is called and not the D one so only the B part of d is destroyed. This means that the dynamically allocated int pointed to by* `d.y` *is not destroyed. Since it can no longer be accessed, this is a memory leakage.*

(c) Now show how to fix the memory leak. *5 pts*

The only changes you are allowed to make are to modify the constructors and/or destructors in B and D, i.e., lines 4, 5, 12 and 13. You should specify which constructors and/or destructors you want to replace and then write the new versions below.

The solution is to add the word virtual *to line 5, i.e., replace the current B destructor in line 5 by*

```
virtual ~B() {delete x;             cout << "DB" << endl;}
```

4. **(6 points)** The following three small programs differ only in the use of the word *virtual*. You should write each program's output in the space provided: *1 pt per output line*

(a)
```
#include <iostream>
class B
{ public:
  virtual void display() {cout << " Inside B "<< endl;}
};

class D : public B
{ public:
  virtual void display() {cout << " Inside D "<< endl;}
};

void main()
{ D d;   B  & b = d;
  d.display();
  b.display();
}
OUTPUT
-------
 Inside D
 Inside D
```

(b)
```
#include <iostream>
class B
{ public:
  void display() {cout << " Inside B "<< endl;}
};

class D : public B
{ public:
  virtual void display() {cout << " Inside D "<< endl;}
};

void main()
{ D d;  B  & b = d;
  d.display();
  b.display();
}
OUTPUT
-------
 Inside D
 Inside B
```

(c)
```
#include <iostream>
class B
{ public:
  virtual void display() {cout << " Inside B "<< endl;}
};

class D : public B
{ public:
  void display() {cout << " Inside D "<< endl;}
};

void main()
{ D d;   B  & b = d;
  d.display();
  b.display();
}
OUTPUT
-------
Inside D
Inside D
```

5. **(5 points)** There are five parts to this question. *1 pt each*

For all of the parts you should assume the existence of the following class definition:

```
class B {
  public: int x;
  protected: int y;
  private: int z;
};
```

In each of the parts below you are given the definition of either one or two classes. For each part decide whether the given definition is **legal** or **illegal**, i.e., would the compiler accept it or not; *underline* the correct answer. The definitions in the different parts are independent of each other.

(a) <u>**legal**</u> / **illegal**

```
class D : private B
      { public: int get_x() {return x;};};
```

(b) <u>**legal**</u> / **illegal**

```
class D : private B
      { public: int get_y() {return y;};};
```

(c) **legal** / <u>**illegal**</u>

```
class D : private B
      { public: int get_z() {return z;};};
```

(d) <u>**legal**</u> / **illegal**

```
class D : protected   B {};
class E : public D
      { public: int get_x() {return x;}; };
```

(e) **legal** / <u>**illegal**</u>

```
class D :  private   B {};
class E :  public D
      { public: int get_y() {return y;} };
```

6. **(7 points)** Assume the following definitions of the classes **shape**, **circle** and **square**.

```
class shape {
public:
 virtual double area() const =0;
 virtual void printShapeName() const =0;
 virtual void print() const=0;
};

class circle:public shape {
public:
 circle(int t) {radius=t;};
 double area() const {return 3.14*radius*radius;};
 virtual void printShapeName() const {cout<<"circle"<<endl;};
 virtual void print() const=0;
private:
 int radius;
};

class square:public shape{
public:
 square(int x) {side=x;};
 double area() const {return side*side;};
 virtual void printShapeName() const {cout<<"square"<<endl;};
 virtual void print() const {cout<<"square side: "<<side<<endl;};
private:
 int side;
};
```

For each of the following declarations, underline whether it is legal or illegal. *1 pt each*

Legal	Illegal	`shape x(5);`
Legal	Illegal	`circle y(5);`
Legal	Illegal	`square z(5);`
Legal	Illegal	`shape* w=new square(5);`
Legal	Illegal	`circle* ww=new square(5);`
Legal	Illegal	`square* www=new square(5);`
Legal	Illegal	`circle* zz=new circle(5);`

7. **(15 points)** Examine the following program and its output.

```
#include <list>
#include <vector>
#include <iostream>
#include <string>

template<class Iterator>
void Display(Iterator start, Iterator end)
{ . . . To be filled in . . . }

template <typename Iterator, typename  T>
void Truncate(Iterator start, Iterator end, T val)
{ . . . To be filled in . . . }

void main()
{ list<int>  L;
  vector<string> S;

  for (int i =1; i < 13; i++)
    L.push_back(i*i % 13);

 S.push_back("A"); S.push_back("H"); S.push_back("B");
 S.push_back("D"); S.push_back("T"); S.push_back("C");

 cout << "L:     "; Display(L.begin(), L.end()); cout << endl;
 cout << "S:     "; Display(S.begin(), S.end()); cout << endl;

 Truncate(L.begin(), L.end(), 5);
 Truncate(S.begin(), S.end(), "F");
 cout << "L:     "; Display(L.begin(), L.end()); cout << endl;
 cout << "S:     "; Display(S.begin(), S.end()); cout << endl;
}
OUTPUT
------
L:     1 4 9 3 12 10 10 12 3 9 4 1
S:     A H B D T C
L:     1 4 5 3 5 5 5 5 3 5 4 1
S:     A F B D F C
```

The program contains two function templates that work on container classes whose code is missing and that you need to fill in.

(a) *6 pts.* **Display** sequentially prints out all items beginning from *start* up to and including the item immediately preceding *end*. The items are printed on the same line and separated by one blank space. Fill in the code for **Display** below. We give you the first two lines.

```
template<class Iterator>
void Display(Iterator start, Iterator end)
{ for( Iterator p = start; p != end; p++)
      cout << *p << " ";
}
```

(b) *9 pts.* **Truncate** runs through all items beginning from *start* up to and including the item immediately preceding *end*. For each item it checks if it is greater than *val* or not. If the item is greater than *val* it replaces the item by *val*. If the item is not greater than *val* it does nothing. Fill in the code for **Truncate** below. We give you the first two lines.

```
template <typename Iterator, typename  T>
void Truncate(Iterator start, Iterator end, T val)
{ for (Iterator p = start; p != end; p++)
 if (*p > val)
   (*p) = val;
};
```

Note: This problem asked you to change *the items in the container. It did not ask you to* output *them; outputting was therefore considered an error.*

8. **(17 points)** Examine the following program and its output:

```
#include <iostream>
#include <string>
#include <list>

template<typename T>
class ListArray
{ public:
    void push_back (const T & val)  {L.push_back(val);}
    T pop_back() {return L.pop_back();}
    int size()  {return L.size();}
    T & operator[ ] (int i);
  private:
    list<T> L;
};

template <typename T>
T & ListArray<T>::operator[ ] (int i)
{ .. To be filled in ...}

template <typename T>
void Display (ListArray<T> & LA)
{ .. To be filled in ...}

 void main()
{ ListArray<int> LA;
 LA.push_back(2); LA.push_back(3); LA.push_back(5);
 cout << "LA: " ; Display(LA); cout << endl;
 LA[1] =7;
 cout << "LA: " ; Display(LA); cout << endl;

 ListArray<string> LS;
 LS.push_back("Now"); LS.push_back("Is"); LS.push_back("The");
 cout << "LS: ";  Display(LS); cout << endl;
 LS[0] = "Yesterday"; LS[1] = "Was";
 cout << "LS: "; Display(LS); cout << endl;
}
OUTPUT
------
LA: 2 3 5
LA: 2 7 5
LS: Now Is The
LS: Yesterday Was The
```

Note that the implementations of the **ListArray** member functions **Display** and **operator**[ ] are missing. The goal of this question is to fill them in correctly.

(a) **operator**[ ] overloads the subscript operator so that it works for **ListArray**.

Note: In standard C++ as taught in lecture, the subscript operator is *not* defined for the **list** class. i.e., you *may not* write `L[i]` where `L` is the private member of the calling object. You must therefore write the code for the operator `ListArray<T>::operator[ ] (int i)`, "from scratch". Finally, your overloaded subscript operator *does not* have to error- check if the index is out of range.
Fill in the code below. We give you the first two lines. *12 pts*

```
template <typename T>
T & ListArray<T>::operator[ ] (int i)
         {list<T>::iterator p = L.begin();
          for (int j=0; j < i; j++)
              p++;
          return *p;
 }
```

Note: There were some answers that tried to replace the **for** *loop by* `p = p +i;` *or* `p +=i;` *, i.e., trying to extend the rules of pointer aritmetic to iterators. This does not work in C++.*

(b) **Display** prints out all items in *LA* in sequential order. The items are printed on the same line and separated by one blank space. Fill in the code for **Display** below. We give you the first two lines. *5 pts*

```
template <typename T>
void Display (ListArray<T> & LA)
     { for(int i = 0; i < LA.size(); i++)
          cout << LA[i] << " ";
     }
```

Note: One common error in solving this problem was in trying to use iterators to run through `LA.L`*. Unfortunately,* `L` *is a* private *member of* `LA` *so* `Display` *is not able to access* `L`*.*

Another error was in trying to use iterators to run through `LA` *by starting at* `LA.begin()` *and printing through to* `LA.end()`*. The difficulty with this approach is that* `LA` *is not a STL container class so* `LA.begin()` *and* `LA.end()` *do not exist.*

9. **(20 points)** In this problem you are given a class **PtrArray** defined in file *PtrArray.h*. The file is missing the implementations of the copy constructor and overloaded assignment operator. These are what you will need to provide.

```
\\***************** PtrArray.h************************
#include <iostream>
typedef int* Intptr;
class PtrArray {
 public:
   PtrArray (int i)                      \\ Constructor
    {  size=i;
       A=new Intptr[i];
       for(int j=0; j<i; ++j)
         A[j]=NULL;
    };
   ~PtrArray() { *** Code omitted *** };  \\Destructor

   void set(int i,int val)
    {  if(A[i]==NULL)
          A[i]=new int;
       *A[i]=val;
    };

   int get(int i) const
    { if (A[i]==NULL) return 0;
         else return *A[i];
    };

 void print()
   {for(int i=0; i<size; ++i)
       if (A[i]==NULL)
         cout <<"N  ";
       else  cout<<*A[i]<<"  ";
    cout<<endl;
   };

  PtrArray(const PtrArray& a)            \\ Copy Constructor
   { To be filled in in part (A)   }
  PtrArray& operator=(const PtrArray& a) \\Overloaded Assignment Operator
    { To be filled in in part (B)   }

private:
 Intptr * A;  int size;
};
```

PtrArray contains an array of pointers. The constructor of the class dynamically creates the array and sets all the pointers to NULL (which is another name for 0). The function **set(int i, int val)** inserts *val* into the memory location pointed to by `A[i]`. If `A[i] == NULL`, then it first dynamically allocates a memory location for `A[i]` to point to.

`get(int i)` returns the integer pointed to by A[i]. If `A[i] == NULL`, it returns 0.

`print( )`, displays, in sequential order, the integers pointed to by the `A[i]`. For NULL `A[i]` it prints the letter "N".

As an example, here is a sample driver program and its output:

```
#include <iostream>
#include "PtrArray.h"
void main() {
  PtrArray a(5);
  cout << "a: "; a.print();
  a.set(2,9); a.set(4,6);
  cout << "a: "; a.print();

  PtrArray b=a;
  cout << "b: "; b.print();

  PtrArray c(4);
  c.set(1,7);
  cout << "c: "; c.print();

  a=a;
  cout << "a: ";  a.print();

  c=a;
  cout << "c: "; c.print();
}
OUTPUT
------
a: N  N  N  N  N
a: N  N  9  N  6
b: N  N  9  N  6
c: N  7  N  N
a: N  N  9  N  6
c: N  N  9  N  6
```

(a) Write the copy constructor. Your copy constructor code should be consistent with the output of the sample driver program. Note that the copy constructor must make a deep copy and **is not allowed to cause a memory leak.** You may assume that the definition of the copy constructor is being included *inside* the class definition of **PtrArray**. We provide the header line for the copy constructor. *8 pts*

```
PtrArray(const PtrArray& a)   \\ Copy Constructor
{  size=a.size;
   A=new Intptr[size];
   for(int j=0; j<size; j++)
     if ((a.A)[j] == NULL)
          A[j] = NULL;
     else
       {A[j]=new int;
       *A[j]=a.get(j);}
  };
```

Note: There were two major mistakes found in many attempted solutions to this problem. The first mistake was that some people forgot to allocate new memory by doing `A[j]=new int;` *for appropriate values of* `i`. *If this memory was not allocated then doing* `*A[j]=a.get(j);}` *could cause a crash.*

The second mistake was in replacing the test `if ((a.A)[j] == NULL)` *by* `if (a.get(i) == 0)`. *These two tests* are not *equivalent. It is possible that* `(a.get(i) == 0)` *but* `((a.A)[j] != NULL)`. *This can occur, for example, if* `a.set(i,0)` *had been run sometime in the past.*

(b) Write the code for the overloaded assignment operator, `operator=()`, for **PtrArray**. Your assignment operator code should be consistent with the output of the sample driver program. Note that the assignment operator must make a deep copy and is not allowed to cause a memory leak. It must also allow self assignment of the type "`a=a;`". You may assume that the definition of the assignment operator is being included *inside* the class definition of **PtrArray**. We provide the header line for the assignment operator. *12 pts*

```
PtrArray& operator=(const PtrArray& a) \\Overloaded Assignment Operator
{  if(this!=&a)
   {for(int i=0; i<size; ++i)
        delete A[i];
    delete [] A;
    size=a.size;
    A=new Intptr[size];
    for(int j=0; j<size; ++j)
     if ((a.A)[j] == NULL)
          A[j] = NULL;
     else
       {A[j]=new int;
       *A[j]=a.get(j);}
  }
  return *this;
 };
```

Note: As well as the mistakes discussed in part (a) another error found here was that some solutions forgot to include

`for(int i=0; i<size; ++i) delete A[i];`

Without this line, or something similar, there would be a memory leak.