1. Smart Pointers  30 points A copy of the Smart Pointers handout is attached for reference. A code snippet is given below using the Smart Pointers concept. In the table below, fill in the value the specified variable at the specified line number. For at least one entry, you will have to make some assumptions and an educated guess. A few of the rows of the table are filled in for you as a starting point. Note: The last entry at line 21 means after all local variable destructors have been called, but before subroutine main actually exits.

```c
// SPointer class and implementation defined here.
void PassByValue(SPointer sp)
{
    sp.Set(0, 'K'); // Change the zeroth char of the string
}
void PassByReference(SPointer& sp)
{
    sp.Set(0, 'K'); // Change the zeroth char of the string
}
int main()
{
    SPointer sp1("This is a test");
    SPointer sp2(sp1); // Copy constructor
    SPointer sp3("ShortString");
    sp3 = sp2;
    PassByValue(sp2);
    PassByReference(sp1);
}
```

Program q1.cc

<table>
<thead>
<tr>
<th>After Line Number</th>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>sp1.st</td>
<td>0xB0001000</td>
</tr>
<tr>
<td>15</td>
<td>*(sp1.refCount)</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>sp2.st</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>*(sp2.refCount)</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>sp3.st</td>
<td>0xB0001020</td>
</tr>
<tr>
<td>17</td>
<td>*(sp3.refCount)</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>sp3.st</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>*(sp3.refCount)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>sp.st</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>*(sp.refCount)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>sp.st</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>*(sp.refCount)</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>sp.st</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>*(sp.refCount)</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>sp.st</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>*(sp.refCount)</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>sp1.st</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>*(sp1.refCount)</td>
<td></td>
</tr>
</tbody>
</table>
class SPointer
{
public:
    SPointer(char*);
    SPointer(const SPointer&); // Need a copy constructor
    ~SPointer(); // Need a destructor
    SPointer& operator=(const SPointer& rhs); // Need an assignment operator

    char Get(int i); // Get the "i'th" character from the array
    void Set(int i, char c); // Set the "i'th" character to the char 'c'

private:
    char* st; // This is the "shared" pointer
    int* refCount; // This keeps up with how many references to st
    int lth;
};

// Constructor
SPointer::SPointer(char* s)
: lth(strlen(s) + 1) // lth is set to the length of the string "s"
{
    st = new char[lth]; // Allocate the dynamic memory from heap
    strcpy(st, s); // Copy the string s to the heap
    refCount = new int(1); // Create the reference count variable, set to 1
}

// Copy constructor
SPointer::SPointer(const SPointer& c)
: st(c.st), refCount(c.refCount), lth(c.lth)
{
    (*refCount)++; // Increment the reference count
}

// Destructor
SPointer::~SPointer()
{
    (*refCount)--; // Decrement the reference count
    if (*refCount == 0)
    { // This is the last reference, delete
        delete [] st;
        delete refCount;
        st = 0;
    }
}

// Assignment operator
SPointer& SPointer::operator=(const SPointer& rhs)
{
    if (st != rhs.st)
    { // not self assignment
        (*refCount)--;
        if (*refCount == 0)
        { // Last reference to my string, delete
            delete [] st;
            delete refCount;
        }
    }
}

Program smartpointers.cc
st = rhs.st; // Use same pointer as right-hand-side
refCount = rhs.refCount; // Use same reference count as rhs
lth = rhs.lth; // Use same length as rhs
(*refCount)++; // Count this reference
}
return *this;
}

// Get and Set functions
char SPointer::Get(int i)
{
  if (i < lth) return st[i];
  return '0'; // Out of range, just return 0
}

void SPointer::Set(int i, char c)// Set the "i’th" character to the char ‘c’
{ // Set the new value. However, we must make a copy of the data
  // This is called "Copy on Write" semantics
  if (i >= lth) return; // Out of range
  if (*refCount > 1)
  { // If this is not the only reference, we need to realloc and copy
    (*refCount)--; // Decrement reference count
    char* newSt = new char[lth]; // Get a new memory array
    refCount = new int(1); // Get a new reference count = 1
    memcpy(newSt, st, lth); // Copy the data
    st = newSt; // And set the new pointer
  }
  st[i] = c; // Change specified character
}
2. Templated Subroutines 20 Points

The code snippet on the next page implements a `Sort` subroutine, that sorts a collection of objects specified by a pair of parameters specifying the first and (last + 1) elements to be sorted. Note that this subroutine is *generic* in the sense that we implemented it for any arbitrary type `T`.

At lines 49 through 52 we attempt to instantiate the `Sort` routine with four different parameter types. Which of the four instantiations of `Sort` will compile and which will not? State reasons why you think the call will compile properly or not.
// Define a template subroutine to compute the sort a collection of elements
// specified by two iterators

template <class T>
void Sort(T b, T e)
{
    // This sort is inefficient, and used for illustrative purposes only
    while(b != e)
    {
        T i = b;
        while(i != e)
        {
            if (*i < *b)
                // Need to swap. This iter_swap is defined in "algorithm"
                iter_swap(i, b); // Swap the two values
            ++i;
        }
        ++b;
    }
}

class A {
public:
A() : a(0) {};
A(int a0) : a(a0) {};
    bool operator>(const A& rhs) { return a > rhs.a; } // Greater-than operator
public:
int a;
};

class B {
public:
    B() : b(0) {};
B(int b0) : b(b0) {};
    bool operator<(const B& rhs) { return b < rhs.b; } // Less-than operator
public:
int b;
};

int main()
{
    char s[] = "This is a test";
    A a1[10] = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
    B b1[10] = {0, 1, 7, 6, 5, 4, 6, 7, 8, 9};

    // Now call the Sort routine with differing parameters
    // Will all of these compile?
    Sort(s, s + 14);
    Sort(&a1[0], &a1[10]);
    Sort(&b1[0], &b1[10]);
    Sort( b1[0],  b1[10]);
}

Program q2.cc
3. **Object Cloning 25 Points**

What is printed by the program below? Explain your answer. Use the back of this page as needed. The printing is done by the `Hello` calls at lines 31 (which is in `Sub1` called from lines 40, 41, 42, and 43), plus the `Hello` call at line 45 (a total of 5 `hello` messages).

```cpp
class Base {
public:
  virtual void Hello() { cout << "Hello from Base" << endl; }
  virtual Base* Clone() { return new Base(*this); }
};

// A derives from Base
class A : public Base {
public:
  virtual void Hello() { cout << "Hello from A" << endl; }
};

// B derives from Base
class B : public Base {
public:
  virtual void Hello() { cout << "Hello from B" << endl; }
  virtual Base* Clone() { return new B(*this); }
};

// C derives from A
class C : public A {
public:
  virtual void Hello() { cout << "Hello from C" << endl; }
  virtual Base* Clone() { return new C(*this); }
};

void Sub1(Base& p)
{
  Base* newBase = p.Clone();
  newBase->Hello();
}

int main()
{
  Base base;
  A a;
  B b;
  C c;
  Sub1(base);
  Sub1(a);
  Sub1(b);
  Sub1(c);
  Base d(*a.Clone()); // Copy constructor
  d.Hello();
}

Program q3.cc
```
4. Discrete Fourier Transform  

Excerpt from the FFT Lab Handout

Given an array of length $N$ containing complex discrete–time samples of some signal $h$, the Discrete Fourier Transform (DFT) $H$ is also an array of complex values of length $N$, defined as:

$$H[n] = \sum_{k=0}^{N-1} W^{nk} h[k]$$

where $W = e^{-j\frac{2\pi}{N}} = \cos(2\pi/N) - j\sin(2\pi/N)$

where $j = \sqrt{-1}$

For all equations here, we use the following notational conventions. $h$ is the discrete–time sampled signal array. $H$ is the Fourier transform array of $h$. $N$ is the length of the sample array, and is always assumed to be an even power of 2. $n$ is an index into the $h$ and $H$ arrays, and is always in the range $0 \ldots (N - 1)$. $k$ is also an index into $h$ and $H$, and is the summation variable when needed. $j$ is the square root of negative one.

From equation 1, it appears that to compute the DFT for a sample of length $N$, it must take $N^2$ operations, since to compute each element of $H$ requires a summation overall all samples in $h$. However, Danielson and Lanczos demonstrated a method that reduces the number of operations from $N^2$ to $N\log_2(N)$. The insight of Danielson and Lanczos was that the FFT of an array of length $N$ is in fact the sum of two smaller FFT’s of length $N/2$, where the first half–length FFT contains only the even numbered samples, and the second contains only the odd numbered samples.

End of Excerpt

Suppose that we implemented Discrete Fourier Transform (DFT) using a naive approach, simply using the basic definition for the DFT given in equation 1. We executed this implementation on a sample set consisting of 65,536 elements, and observed an execution time of 30 minutes. Since this execution time seems too long, we decided to implement a more efficient version of the algorithm, using the Danielson–Lanczos binary decomposition method and the Cooley–Tukey bit–reversal trick that we used for our lab 4.

Estimate the running time for the new, efficient algorithm, when running on the same sample set of 65,536 elements. Assume that for both implementations we pre–computed the $W$ array (as we did in our implementation). Also, you can ignore any one–time overhead such as the time needed to read in the sample set from disk, to print out results, and to pre–compute the $W$ values. Explain your answer, state any assumptions you made, and show your work. Clearly, there is no exact “right answer”. Your answer must be reasonable, your assumptions must be valid, and your answer must be consistent with your assumptions.