Sample Test 1

Name: **CS 3358** **Grade:**

1. What will happen if a function is executed and the *precondition* for the function is not met?
   - ☐ An error message will be printed.
   - ☐ The program will loop indefinitely.
   - ☐ The system will crash.
   - ☒ Any of the above results could happen.
2. Answer true or false for this statement: For all possible inputs, a *linear* algorithm to solve a problem will perform faster than a *quadratic* algorithm to solve the same problem.
   - ☐ TRUE.
   - ☒ FALSE.
3. Answer true or false for this statement: An algorithm with *worst case* time behavior of 3n takes at least 30 operations for every input of size n=10.
   - ☐ TRUE.
   - ☒ FALSE
4. Here is the definition of **`SomeClass`** class:

```
class SomeClass
{
    friend f1(SomeClass z);

    public:
        SomeClass( );
        void f2() const;
        void f3(int i);
    private:
        int size;
};
```

   Suppose that **`x`** and **`y`** are both **`SomeClass`** objects. Write the word "**Yes**" or "**No**" in each location of the following table to indicate whether the indicated statement is *legal* or *illegal* in that location:

Statement	In **`main`**	In **`const`** member function **`f2`**	In **`friend`** function **`f1`**
`x = y;`	**Yes**	**Yes**	**Yes**
`x.size = y.size;`	**No**	**Yes**	**Yes**
`x.size = 3;`	**No**	**Yes**	**Yes**
`x.f3(42);`	**Yes**	**Yes**	**Yes**

5. Suppose that you define a new class called **`SomeClass`**. For two **`SomeClass`** objects **`x`** and **`y`**, you would like the expression **`x + y`** to be a new **`SomeClass`** object. Show a prototype of the function that you must write to allow expressions such as **`x + y`** to be valid.

   Either overload operator + as a *non-member* function:
   `SomeClass operator+(const SomeClass& s1, const SomeClass& s2);`
   or overload operator + as a *member* function:
   `SomeClass operator+(const SomeClass& s2) const;`

6. Here is the start of a class declaration:

```
class SomeClass
{
   public:
      void x(SomeClass f);
      void y(const SomeClass f);
      void z(SomeClass f) const;
   ...
```

   Which of the three member functions can alter the **`private`** data members of the **`SomeClass`** object that activates the function?

   - ☐ Only **`x`** can alter the **`private`** data members of the object that activates the function.
   - ☐ Only **`y`** can alter the **`private`** data members of the object that activates the function.
   - ☐ Only **`z`** can alter the **`private`** data members of the object that activates the function.
   - ☒ ***Two*** of the functions can alter the **`private`** data members of the object that activates the function.
   - ☐ ***All*** of the functions can alter the **`private`** data members of the object that activates the function.

7. When should you use a **`const`** *reference* parameter?

   - ☐ Whenever the data type might be many bytes.
   - ☐ Whenever the data type might be many bytes, the function changes the parameter within its body, and you *do not* want these changes to alter the actual argument.
   - ☐ Whenever the data type might be many bytes, the function changes the parameter within its body, and you *do* want these changes to alter the actual argument.
   - ☒ Whenever the data type might be many bytes and the function does not change the parameter within its body.

8. Which kind of functions can access **`private`** data members of a class?

   - ☐ **`friend`** functions of the class
   - ☐ **`private`** member functions of the class
   - ☐ **`public`** member functions of the class
   - ☒ *All* of the above
   - ☐ *None* of the above

9. When developing a class that contains a data member that is a pointer pointing to dynamically allocated memory, what member functions should be provided for the class?

   - ☐ Overloaded assignment operator.
   - ☐ Copy constructor.
   - ☐ Destructor.
   - ☒ All of the above.

10. Listed below are some desired effects. Indicate, by circling the appropriate character(s), whether each can be accomplished with *overloaded functions* (**O**) and/or a *function with default arguments* (**D**) and/or a *function template* (**T**) or *none of them* (**N**).
    - **[(O) (D) T N] `CalcMass(density, volume)`** returns the mass of an object having a density of **`density`** and a volume of **`volume`**, whereas **`CalcMass(density)`** returns the mass of an object having a density of **`density`** and a *unit volume* (1.0 cubic feet, say). All quantities are of type **`double`**.
    - **[(O) D T N] `Repeat(10, "Rats!")`** displays the string argument 10 times, whereas **`Repeat("Dang it!")`** displays the string argument 3 times.
    - **[(O) D (T) N] `CalcAvg(3, 6)`** returns the **`int`** average of two **`int`** arguments, whereas **`CalcAvg(3.0, 6.0)`** returns the **`double`** average of two **`double`** arguments.
    - **[O D T (N)] `SmartAssign("Excellent")`** returns the character **`'E'`** or a pointer to the string **`"Excellent"`** depending on whether the return value is assigned to a **`char`** variable or to a *pointer* to **`char`** variable.

11. The function **`Mystery`** is defined as shown below:

```
int Mystery(int x, int n)
{
   if (n < 1)
   {
      cerr << "Bad value for n" << endl;
      exit(EXIT_FAILURE);
   }
   else if (n == 1)
      return x;
   else
      return(x + Mystery(x, n – 1));
}
```

- What value is returned by the function call **`Mystery(3, 1)`**?

  3

- What value is returned by the function call **`Mystery(3, 2)`**?

  6

- What value is returned by the function call **`Mystery(3, 3)`**?

  9

- Generalizing from the preceding three results, describe what **`Mystery(x, n)`** will return, assuming **`x`** and **`n`** are valid (and are not too large as to cause any overflow problem).

  product of **`x`** and **`n`** (**`x`** added up **`n`** times)

12. It is intended that the **AllocMem** function shown below dynamically creates an integer array of some user-specified size and returns the address of the dynamic array to the calling function through the pointer argument. Will the function work as intended? Correct it if your answer is no.

```
int AlloMem(int* arrayPtr)
{
   int numOfItems;

   cout << "Number of items to process: ";
   cin >> numOfItems;
   arrayPtr = new int [numOfItems];
   if (arrayPtr == 0)
      cerr << "AlloMem() Error: \"new\"-failure" << endl;
   return numOfItems;
}
```

No. The pointer argument must be passed by *reference* as shown corrected below:

```
void AllocMem(int*& arrayPtr)
```

13. Re-write the **AllocMem** function (the corrected version if correction is needed) in Question 12 as a *function template* so that it can be used to create a dynamic array of *any* type.

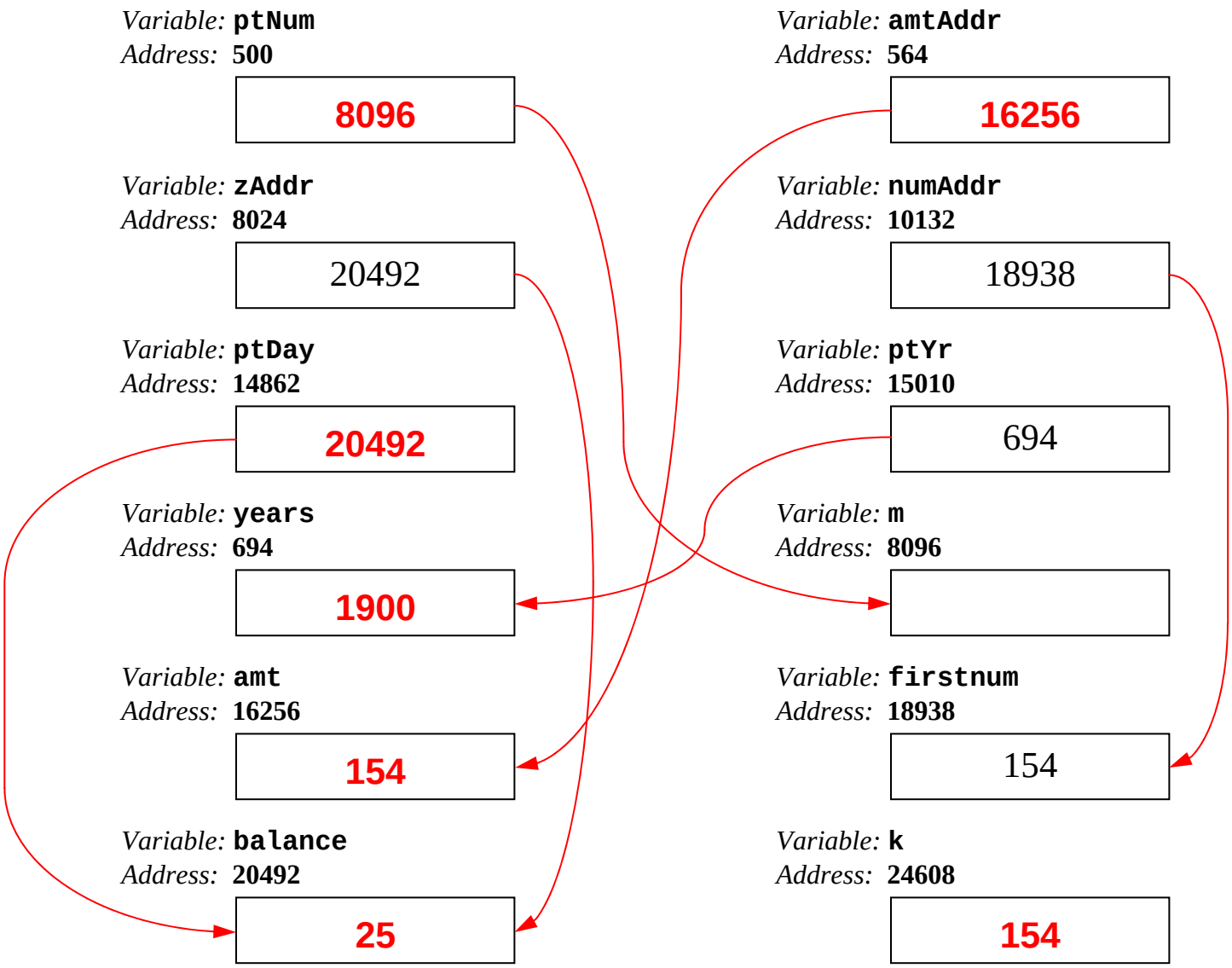
```
template <class T>
void AllocMem(T*& arrayPtr)
{
   int numOfItems;

   cout << "Number of items to process: ";
   cin >> numOfItems;
   arrayPtr = new T [numOfItems];
   if (arrayPtr == 0)
      cerr << "AllocMem() Error: \"new\"-failure" << endl;
}
```

14. C++ allows many of its built-in operators to be overloaded for a user-defined class. The overloading of a majority (but *not* all) of these overloadable operators can be implemented in *three* different ways with respect to function type. First identify the three ways, then identify the *restrictions* as they apply to the overloading of the *assignment operator* (**=**), *the stream extraction operator* (**>>**) and the *stream insertion operator* (**<<**).

```
The three ways:
(1) As member functions
(2) As non-member, non-friend functions
(3) As non-member, friend functions
The restrictions:
(1) = must be overloaded with member functions
(2) >> and << cannot be overloaded using member functions (must use
    either non-member, non-friend or non-member, friend functions).
```

15. For the variables and addresses illustrated below, fill in the appropriate data as determined by the following statements. Also, use *arrows* to indicate the relationships among the variables (*i.e.*, which variables point to which other variables).

- `ptNum = &m;`
- `amtAddr = &amt;`
- `*zAddr = 25;`
- `k = *numAddr;`
- `ptDay = zAddr;`
- `*ptYr = 1900;`
- `*amtAddr = *numAddr;`



16. For the following list of function prototypes, check only the boxes of those that have the *same* signature.

- ☒ `int calc(int a, int b);`
- ☒ `int calc(int c, int d);`
- ☐ `int calc(int a, int& b);`
- ☐ `int calc(int a, const int& b);`
- ☐ `int calc(int a, int* b);`
- ☐ `int calc(int a, int*& b);`
- ☒ `void calc(int a, int b);`
- ☐ `int calc(int a, double b);`
- ☐ `int calc(double b, int a);`
- ☐ `int calc(int a, int b, int c);`

17. Briefly but clearly describe the *memory-related* problem manifested in the following code segment?

```
int *ptr1 = new int;
if (ptr1 == 0)
{
   cerr << "Error allocating memory" << endl;
   exit(EXIT_FAILURE);
}
int *ptr2 = 0;
*ptr1 = 100;
ptr2 = ptr1;
cout << "*ptr2 = " << *ptr2 << endl;
delete ptr2;
cout << "*ptr1 = " << *ptr1 << endl;
```

The statement **`ptr2 = ptr1;`** makes **`ptr2`** and **`ptr1`** both point to the dynamic memory allocated through the first statement. The statement **`delete ptr2;`** releases this dynamic memory. The expression **`*ptr1`** in the last statement attempts to dereference the memory after it has been released and thus constitutes a memory access violation.

18. Briefly but clearly describe the *memory-related* problem manifested in the following code segment?

```
do
{
   int *intPtr = new int [10];
   if (intPtr == 0)
   {
      cerr << "Error allocating memory" << endl;
      exit(EXIT_FAILURE);
   }

   // ... dynamic array is used here to solve a problem

   cout << "Do another (y or n)? ";
   cin >> answer;
} while (answer != 'n' && answer != 'N');
```

The dynamic memory acquired through each execution of the statement **`int *intPtr = new int [10];`** is not being released. The code segment thus suffers from the notorious "memory leak" problem.

19. In the Sample Solution for Homework 2, there is an oddly-looking member function reproduced below in prototype:

```
const char * const GetName() const;
```

Explain the significance of each use of the **`const`** keyword in the statement.

1st **`const`** → pointer returned cannot be used to change the character it is pointing at.
2nd **`const`** → pointer returned cannot have its value changed (be made to hold a different address).
3rd **`const`** → **`GetName`** is an accessor (not a mutator).

20. Here is a function declaration:

```
void goo(int *x, int *y)
{
   int z = *x;
   *x = *y;
   *y = z;
   x = y;
}
```

Draw a picture of memory after these statements:

```
int i = 1, k = 2;
int *p1 = &i, *p2 = &k;
   goo(p1, p2);
```