Part I: 15 Multiple choice questions (2 points each)

Answer all of the following questions. READ EACH QUESTION CAREFULLY. Fill the correct bubble on your mark-sense sheet. Each correct question is worth 2 points. Choose the one BEST answer for each question. Assume that all given C++ code is syntactically correct unless a possibility to the contrary is suggested in the question.

In code fragments, YOU SHOULD ASSUME THAT ANY NECESSARY HEADER FILES HAVE BEEN INCLUDED. For example, if a program does input or output, you should assume that header files for the appropriate stream library have been #included, and that "using namespace std;" has been provided, even if it is not shown in a question.

Remember not to devote too much time to any single question, and good luck!

1. Consider the program:

```
int main() {
        int k = 0;
        int j = 0;

        while (k == j) {
                int k;
                for (k = 0; k < 3; k++)
                        j = j+k;
        }
        k++;

        cout << "J is " << j << endl;
        cout << "K is " << k << endl;

        return 0;
}
```

What is printed?

A. J is 3
K is 4.

B. ***J is 3***
K is 1

C. J is 0
K is 1.

D. J is 0
K is 4.

E. None of the above.

2.
```
class WeirdClass {
  public:
    virtual void setName( string name );
    virtual void setID( int id ) = 0;

  protected:
    int id;

  private:
    string name;
};
```

A correct implementation for setName would be:

A. name = name;

B. name = this->name;

C. ***this->name = name;***

D. this.name = name;

E. name = this.name;

3. **Consider the following function definitions:**

```
void baffle(string s, int i);
void baffle(string s1, string s2);
void baffle(int i);
void baffle(string s, double x);
```

Which of the following function calls will be rejected by the C++ compiler, if any?

A. baffle(5);

B. baffle("this test is too long", 17);

C. baffle("this test is too", " long");

D. ***baffle("??");***

E. All of these function calls are legal – none will be rejected by the compiler

4. Given the following declarations:

```
class B {
public:
  void f1();
  virtual void f2();
};

class D: public B {
public:
  void f1();
  virtual void f2();
};
```

And the following code:

```
B *b = new D;

b->f1();
b->f2();
```

Which functions are called:

A. B::f1(), B::f2()

B. ***B::f1(), D::f2()***

C. D::f1(), D::f2()

D. D::f1(), B::f2()

5.
```
class FourInts
{
public:
   FourInts();
   FourInts(const FourInts& other);
   ~FourInts();
private:
   int *data;
};

FourInts::FourInts() {
    data = new int[ 4 ];
}

FourInts::FourInts( const FourInts& other ) {
   data = other.data;
}

FourInts::~FourInts() {
   delete[] data;
}

int main() {
   FourInts a;
   FourInts b = a;
   return 0;
}
```

Q: The main program will crash when it is executed because of a memory management error. Which function in class FourInts can be changed to *fix* the bug?

A FourInts()

B. ***FourInts(const FourInts& other)***

C. ~FourInts()

D. You have to change more than one function to fix the bug.

6. Assume that you are running a program in which asserts are enabled (i.e., MSVC debug mode, not MSVC release mode). What happens when the following line of code is executed?

```
assert(false);
```

A. Nothing -- assert is always ignored.

B. ***The program will be terminated with a suitable error message.***

C. An error message will be displayed, then execution will continue.

D. The program won't compile because the argument to assert must be an integer.

E. None of the above.

7. An example of overloading in C++ is:

A. ***Having multiple constructors for a class.***
B. Attempting to read in a value which is larger than the maximum value that a variable can hold.
C. Placing more than one class definition in a single .h file.
D. A function that executes too many recursive function calls and runs out of memory.
E. The multiple places that the keyword "const" can be used.

8. Which is NOT an essential feature of a proper recursive function?

A. One or more base cases.
B. One or more recursive cases.
C. The function calling itself at some point.
D. Testing for a base case before calling a recursive cases.
E. ***Not permitted to do any I/O***

9. What is true about destructors?

i. There can be more than one destructor for a class, as long as each has a unique set of parameters.
ii. The name of a destructor has to start with the character ~
iii. The only statement that can appear in a destructor is "delete".

A. i only
B. ***ii only***
C. iii only
D. ii & iii
E. all of the above

10. Which of the following statements are true about const variables?

i. Their values never change.
ii. They must be initialized when defined.
iii. They can only be used as parameters to const member functions.
iv. They are always global.

A. ***i & ii***
B. i, ii & iii
C. ii, iii & iv
D. i & iv
E. All of the above.

11. What is true about const member functions of classes? (e.g. a member function whose declaration includes "const" after the parameter list, as in: string getTitle() const;)

i. They always return the same unique value each time they are called.
ii. They cannot modify the values of any members of the class.
iii. They cannot modify any of their parameters.

A. i only
B. ***ii only***
C. iii only
D. ii & iii
E. All of the above.

12. The following three functions are supposed to compute n! (1*2*3*...*n for positive n, 1 otherwise). Which ones work properly if called with an arbitrary integer expression as an argument?

i.
```
int factorial(int n) {
    if (n <= 0)
        return 1;
    int product = 1;
    for (int i = 1; i <= n; i++)
        product = product * i;
    return product;
}
```

ii.
```
int factorial(const int &n) {
    if (n <= 0)
        return 1;
    int product = 1;
    for (int i = 1; i <= n; i++)
        product = product * i;
    return product;
}
```

iii.
```
int factorial(int n) {
    if (n <= 0)
        return 1;
    else if (n == 1)
        return 1;
    else
        return n * factorial(n - 1);
}
```

A. i only
B. ii only
C. iii only
D. ***two of the above but not all***
E. ***all three work properly***

[This question had a mistake. Version ii should not have worked, because an arbitrary expression normally can't appear when the parameter type is a reference (int&) – the argument would normally have to be a variable. However, since the parameter is declared const, it does work, for technical reasons that are really beyond the scope of CSE143. So both answers were allowed.]

The last three multiple choice questions and the first short answer question all ask about the smurf class hierarchy. These classes are slightly modified versions of the ones discussed in section. The code is given on the last two pages of the exam; feel free to tear that sheet off for reference.

We STRONGLY SUGGEST that you first turn the page and answer the first short answer question before you do the rest of the questions on this page. The answer to that question will help you here.

For the following questions, consider the following main function:

```
int main() {
        TricksterSmurf Jokey(true);      // line A
        Smurf Baby(true);
        Smurf* ptr1 = &Baby;
        ptr1 = new SkilledSmurf;         // line B
        SkilledSmurf* ptr2 = &Jokey;
        ptr2->Sing();                    //line C1
        ptr2->DoSkill();                 //line C2
        ptr2->Shop();                    //line C3
        return 0;
}
```

13. Which constructors are called at line A?

i) TricksterSmurf()
ii) TricksterSmurf(bool)
iii) SkilledSmurf()
iv) Smurf()
v) Smurf(bool)

A. i only
B. ii only
C. i & iii & iv
D. ***ii & iii & iv***
E. ii & iii & v

14. Take a look at line B. Is there anything wrong with it?

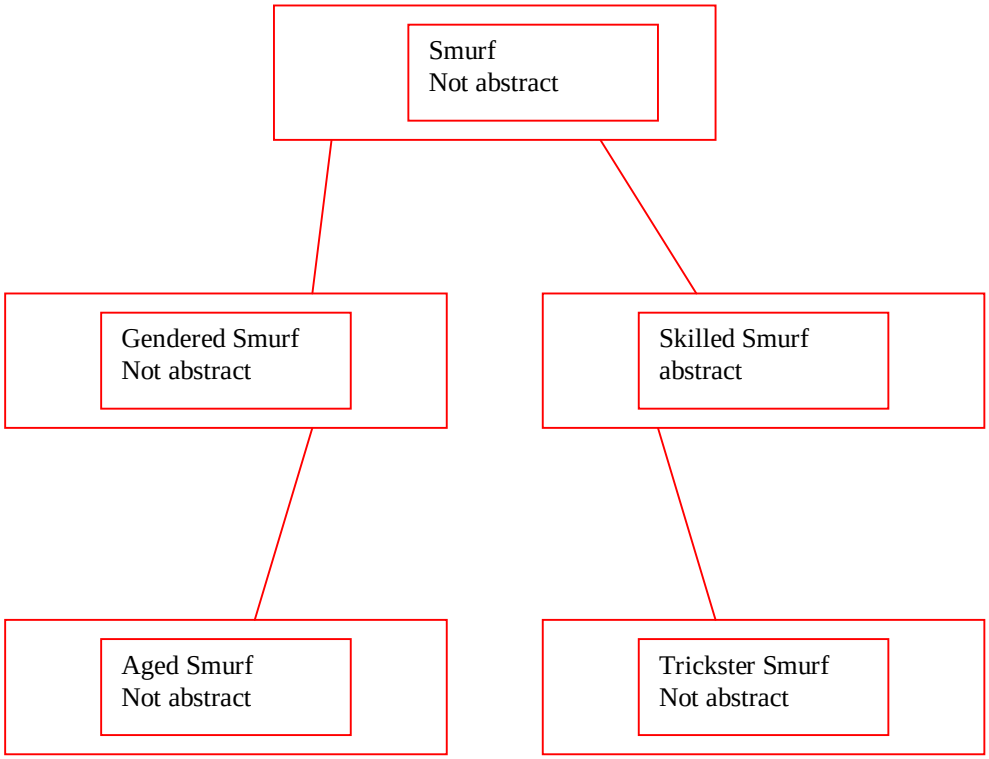
A. ptr1 pointed somewhere already, so this will cause a memory leak.
B. SkilledSmurf and Smurf are different classes, and ptr1 can only point to Smurfs.
C. ***SkilledSmurf is an abstract class, so you can't instantiate it.***
D. All of the above.
E. None of the above. This line is perfectly fine.

15. Which lines are illegal?

A. C1
B. C2
C. ***C3***
D. C1 & C3
E. None. They're all legal.

Part II: 2 Short Answers (4 points each)

16. Draw a picture of the smurf class hierarchy given on the last sheet in this exam. Draw a box for each class with lines connecting it to all of its derived classes, and put each base class *above* all of its derived class(es). In each box, *write whether or not the class is abstract.*



17. Briefly describe the difference between a virtual function and a pure virtual function. Be sure to explain the difference between a class that contains pure virtual functions and one that does not.

An ordinary virtual function is a member function of a class and an implementation of it must be provided. A pure virtual function ("=0" notation) is pure specification. An implementation is not provided.

A class that contains (or inherits) any pure virtual functions is an abstract class. Instances of such classes cannot be created directly. An abstract class is normally intended to be used as a base class for derived classes that directly or indirectly override the pure virtual functions and provide implementations for them.

[Aside for the language lawyers: Although it is technically possible in C++ to provide an implementation of a pure virtual function, such functions cannot be called directly (via dynamic dispatch), and this is a very unusual usage. Even if an implementation is provided, the class is still abstract, and instances of it cannot be created directly.]

Part III: Programming Question (12 points)

18. For this question, add a copy constructor and assignment operator (operator =) to the Inventory class from HW3. These functions should perform a deep copy of an Inventory. The next page is blank for your use.

Declarations for the copy constructor and assignment operator have been added to the Inventory class below. Feel free to add additional private functions if they are helpful in your solution. Be sure to include their declaration(s) *with an appropriate heading comment* in the class and their implementation(s) on the next page.

HINT: You may be able to simplify your solution by defining a private copy function that is used by both the assignment operator and the copy constructor.

A change from HW3: You should assume the the BookInfo class has been extended to include an assignment operator in addition to the copy constructor you wrote for HW3. This means you can use assignment to copy BookInfo objects without having to write the code for BookInfo::operator= yourself.

```
// This class is used to hold a list of books.  There is no maximum number of books.
class Inventory {
 public:

  // construct an empty inventory that can hold 4 books
  Inventory( );

  // free any memory held by this inventory
  ~Inventory ();

  //construct an inventory which is a deep copy of other
  Inventory(const Inventory &other);

 //reset this Inventory to be a deep copy of other
 Inventory &operator = (const Inventory &other);

private:


// make this Inventory a deep copy of other (including allocating a suitable array)
// pre: any previously allocated books array has been deleted
void copy(const Inventory &other);
```

```
  int size;              // size of the array – books[0..size – 1] is defined
  int inUse;             // number of elements actually in use
  BookInfo *books;       // pointer to dynamically allocated array of BookInfo objects.
                         // information about current inventory is stored in books[0..inUse - 1].
  };                     // [You may assume that the constructor is able to successfully allocate
                         //  this array and initialize books to point to it.]
```

Write your implementations of the Inventory copy constructor and operator= on this page.

```
// construct anIinventory which is a deep copy of other
 Inventory::Inventory(const Inventory &other) {
        copy(other);
}

// assign a deep copy of other to this Inventory
 Inventory & Inventory::operator = (const Inventory &other) {

        // return immediately if this is self assignment
        if (this == &other)
                return *this;

        // free old storage then copy other and return a reference to this Inventory
        delete [ ] books;
        copy(other);
        return *this;
}


// make this Inventory a deep copy of other (including allocating a suitable array)
// pre: any previously allocated books array has been deleted
void Inventory::copy(const Inventory &other) {

        books = new BookInfo[other.size];
        for (int i = 0; i < other.inUse; i++)
                books[i] = other.books[i];
        size = other.size;
        inUse = other.inUse;
}
```

*[Note: The grading of operator= ignored small errors in the type of the return value and the check for self-assignment. As long as there was a reasonable attempt to check for self-assignment and return a reference to the current Inventory object, we didn't deduct points for missing &'s or *'s. We also did not require heading comments on function implementations. They should be provided in real code, but that was not expected during this exam.]*

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//  Smurf classes for last multiple choice and first short-answer questions
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
enum Gender {MALE, FEMALE};
enum Color {WHITE, RED};

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
class Smurf {
 public:
      Smurf();
      Smurf(bool isShort);
      virtual void Sing();
      string GetHeight();
 private:
      string height;
protected:
      string songWords;
};

Smurf::Smurf() {
      songWords = "La la, la-la la la, la la-la la la";
      height    = "three apples tall";
}

Smurf::Smurf(bool isShort) {
      songWords = "La la, la-la la la, la la-la la la";
      if (isShort)
          height = “two apples tall";
      else
          height = “three apples tall”;
}

void Smurf::Sing() {
      cout << songWords << endl;
}

string Smurf::GetHeight() {
      return height;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
class GenderedSmurf : public Smurf {
 public:
      GenderedSmurf(Gender gender);
      bool IsMaleSmurf();
protected:
      Gender gender;
};

GenderedSmurf::GenderedSmurf(Gender gender) {
      this->gender = gender;
}

bool GenderedSmurf::IsMaleSmurf() {
      return (gender == MALE);
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

```
class AgedSmurf : public GenderedSmurf {
 public:
      AgedSmurf(Color pants, Gender gender);
      bool IsPapaSmurf();
 private:
      Color pants;
};

AgedSmurf::AgedSmurf(Color pants, Gender gender) : GenderedSmurf(gender) {
      this->pants = pants;
}

bool AgedSmurf::IsPapaSmurf() {
      if (pants == RED && gender == MALE)
                  return true;
      return false;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
class SkilledSmurf : public Smurf {
 public:
      SkilledSmurf();
      virtual void DoSkill() = 0;
};

SkilledSmurf::SkilledSmurf() { }

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
class TricksterSmurf : public SkilledSmurf {
 public:
      TricksterSmurf();
      TricksterSmurf(bool hasGifts);
      void DoSkill();
      virtual void Shop();
 private:
      bool hasGifts;
};

TricksterSmurf:: TricksterSmurf() {
      hasGifts = true;
}

TricksterSmurf:: TricksterSmurf(bool hasGifts) {
      this->hasGifts = hasGifts;
}

void TricksterSmurf::DoSkill() {
      if (hasGifts ) {
                  cout << "Happy unbirthday.  Here's a present for you." << endl;
                  cout << "Ouch!!  That wasn't supposed to blow up in MY face!!" << endl;
      }
      else
         cout << "I'm all out of presents!  Better go shopping." << endl;
}

void TricksterSmurf::Shop() {
      hasGifts = true;
}
```