Question 1: [20 points]

(a) Consider the following C++ program:

```
#include <iostream>
using namespace std;

int main()
{
for(int i = 0; i < 2; i++)
 for(int i = 0; i < 2; i++)
  for(int i = 0; i < 1; i++)
   cout << "outer i= " << i <<  " middle i= " << i << " inner i= " << i << endl;
return 0;
}
```

The output produced by this program is (choose one of the following as your answer):

(i)
```
outer i= 0 middle i= 0 inner i= 0
   outer i= 0 middle i= 1 inner i= 0
   outer i= 1 middle i= 0 inner i= 0
   outer i= 1 middle i= 1 inner i= 0
```

(ii)
```
outer i= 0 middle i= 0 inner i= 0
   outer i= 0 middle i= 0 inner i= 0
   outer i= 0 middle i= 0 inner i= 0
   outer i= 0 middle i= 0 inner i= 0
```

(iii) No output will be produced as the program will not compile due to the fact that there are multiple declarations of variable `i` within the scope of function `main()`.

(iv) A run-time error will occur during the first iteration of the outermost for loop.

(v) `Segmentation fault`

(b) Consider the following C++ program:

```
#include <iostream>
using namespace std;

struct keith
{
 private:
   int telecaster;
   keith() { telecaster = 1952; }
 public:
   void print_telecaster() {  cout << telecaster << endl; }
};
```

```
int main()
{
keith richards;
richards.print_telecaster();
return 0;
}
```

Choose from one of the following possible answers:

(i) The program will not compile.

(ii) The program will produce as output : `1952`

(iii) A run-time error will occur immediately after the line: `richards.print_telecaster();`

(iv) A stack overflow will occur due to infinite recursion.

(v) The program will compile, but no output is produced since the method `telecaster()` has no return value.

(c) Explain in words (one single word will suffice) what the following C++ statements will create (**Hint**: whatever it is, there will be lots of it ... 4 000 000 bytes of it) :

```
int* ptr;
ptr = new int [1000000];
cout << ptr << endl;
ptr = new int (5);
cout << *ptr << endl;
delete ptr;
```

(d) Fill in the two blanks which appear in the comment in the following portion of code:

```
int* ptr;
ptr = new int (5);
cout << *ptr << endl;
delete ptr;
// ptr is now said to be a:    _____________  _____________
// Hint: first blank = 8 letters, second blank = 7 letters
*ptr = 6 ; // this is a dangerous thing to do on account of
           //  the two blanks...
```

(e) TRUE or **FALSE**: The name of an array is synonymous with the starting address of the first element of the array.

Question 2: [20 points]

In order to be able to generate a bar chart which plots the number of students whose grade falls into certain ranges of grades (see Figure 1 for a sample bar chart), an instructor has to first count the number of students whose grades fall into each category or "bin". The instructor for this course uses the "tally" method of first counting by hand (using a sheet of paper upon which each "bin" has been indicated) the number of students whose grade falls into each of the different bins (<= 20%, 21-25%, 26-30%, etc). As shown in Figure 2, this is a cumbersome and error-prone method and somewhat primitive. The objective of this question is to help the instructor by writing a C++ program which will automate the tabulation of this grade data.

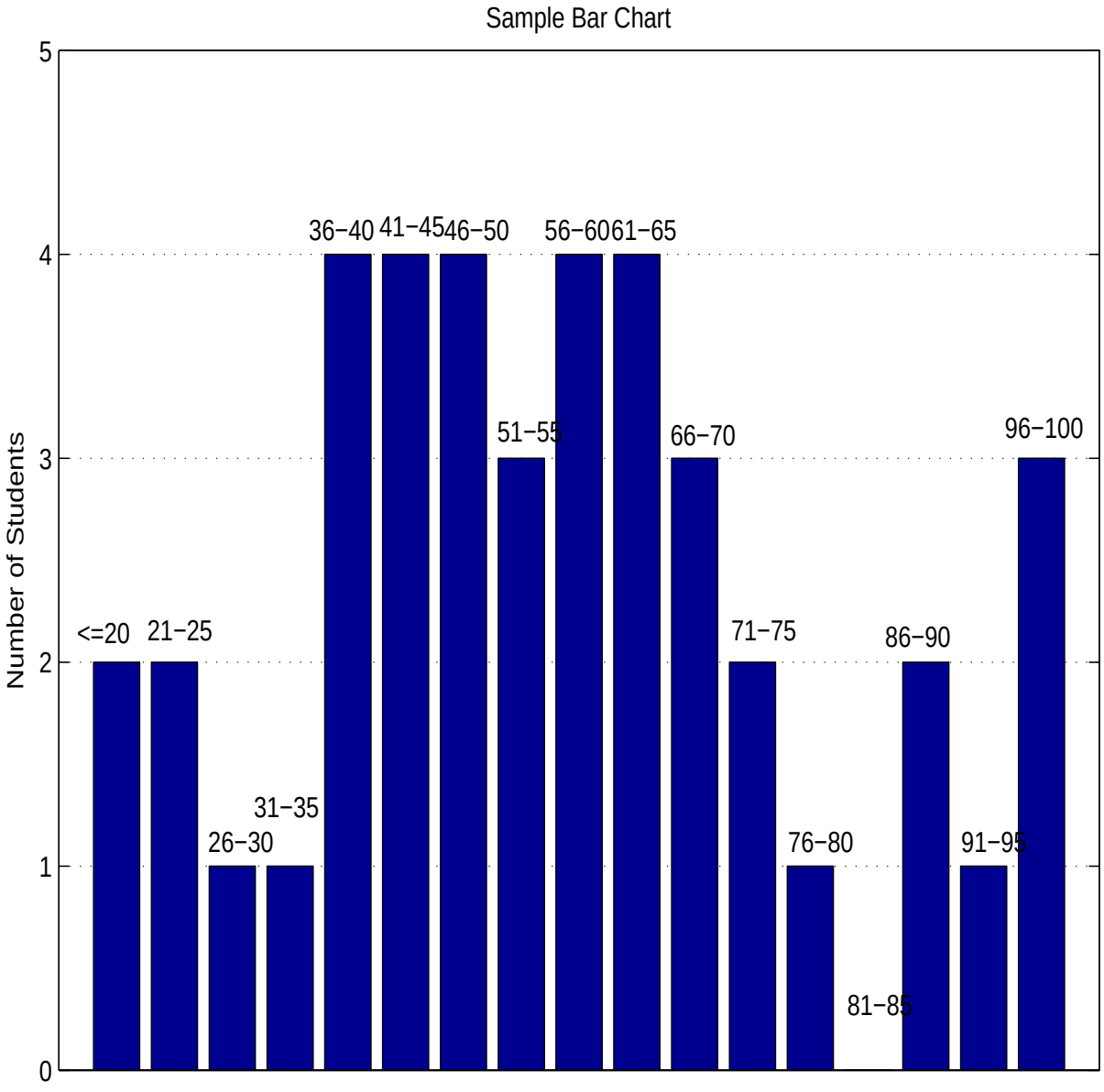

Figure 1: A sample bar chart.

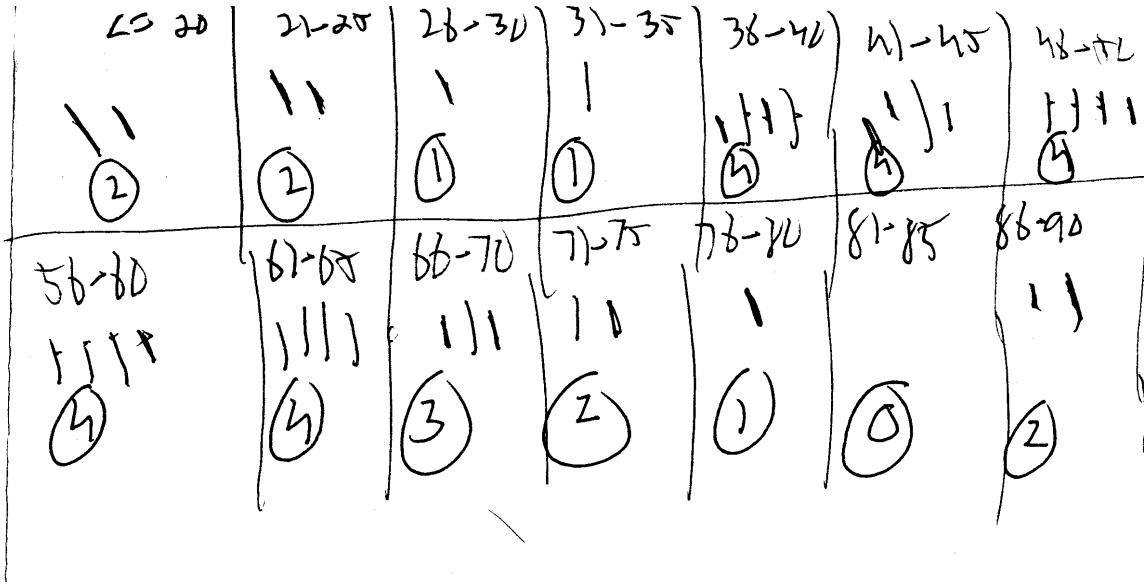

Figure 2: A messy (but actual) "tally" sheet showing number of students in each "bin".

This question deals with using an array of `structs` to keep track of the number of students whose grade falls into each grade category. In the interest of simplicity, we will use the following 6 categories of grades:

```
0  <= grade <  20
20 <= grade <= 39
40 <= grade <=  59
60 <= grade <= 79
80 <= grade <= 99
      grade  = 100
```

The instructor also wants to keep track of the percentages of the number of students within each grade category. The following `const`, `struct` and array definitions can be used to keep track of this information:

```
const int size = 6 ; // 6 different "bins" a grade can fall into

struct element
{
  int how_many; // used to keep track of number of students
                // whose grade falls into each "category"
  float percentage;
};

element bins[size]; // an array of structs, one array element
                    // for each possible "category"
```

We will use the variable `bins[0].how_many` to keep track of the number of students whose grade is in the first "bin" of 0-19%. Similary, `bins[1].how_many` will be used to store the number of students whose grade falls within the second "bin" of 20-39%, etc. The `bins[0].percentage` variable will be used to compute the percentage of students in the class whose grade falls into this category. Similarly, the `bins[i].percentage` variable will be used to compute the percentage of students in the class whose grade falls into that bin.

A picture is said to be worth a thousand words, so Figure 3 is a pictorial representation of what the previous words described. In Figure 3, we assume that 3 students had grades which fell into the 0-19% category, 5 students had grades in the 20-39% category, 6 students had grades in the 40-59% vategory, 10 students had grades in the 60-79% category, 7 students had grades in the 80-99% range, and 2 students had a grade of 100%.

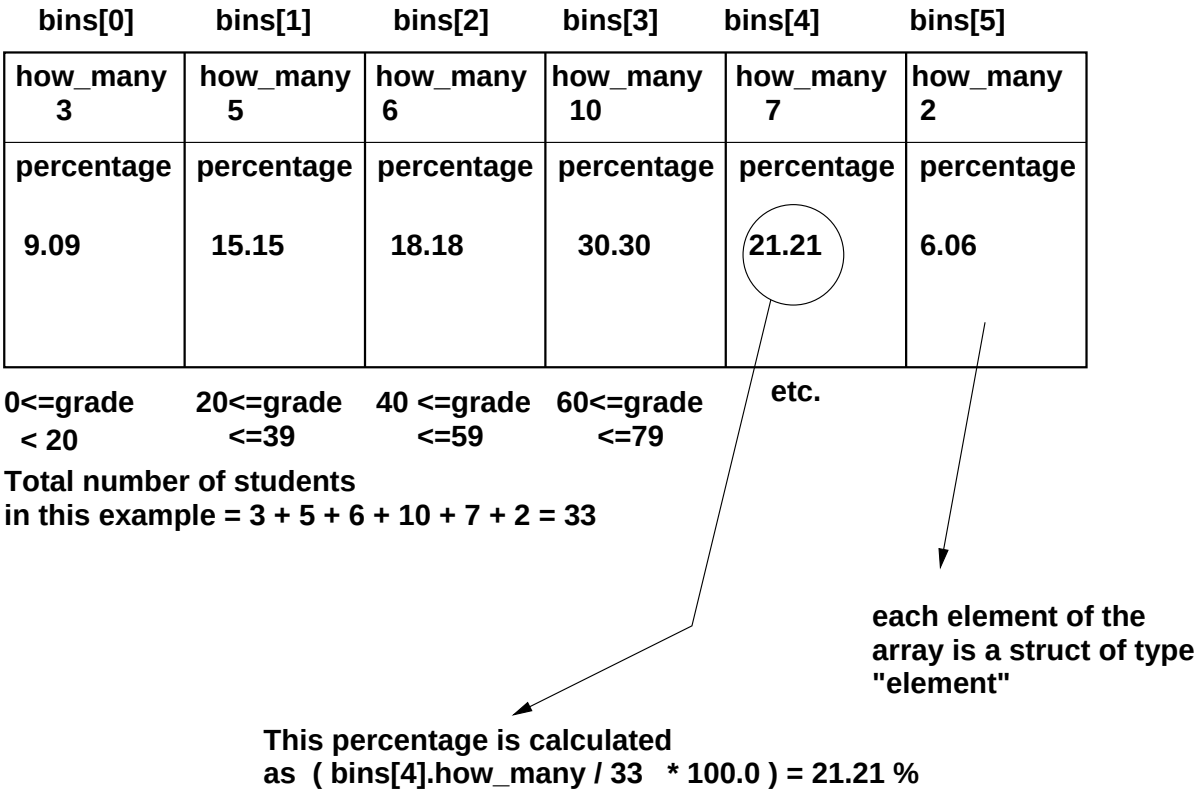

Figure 3: A pictorial representation of the "bins" array.

Write a C++ program which will **repeatedly** ask the user to enter an integer value representing a student's grade. The program will then "add" this grade (by updating the appropriate `how_many` field) to the appropriate element of the array called `bins` (depending upon which category the grade belongs to). A negative grade value will be used as a **flag** value to designate that there are no more grades to be entered. The program is to keep track of the total number of grades which have been entered. After all the grades have been entered, the program is to compute the values of the percentage field of each element of the `bins` array and print these percentages. To ensure correctness, the program will compute the total number of grades entered by summing up the val-

ues of how_many stored in the bins array and compare this sum to the total number of grades entered. In addition, the program will sum up the values of the percentage field stored in the array and print it out to make certain that it is equal to 100%.

You **must** use the following functions in your program:

```
void initialize_bins(element a[])
{
// this function sets the how_many field to 0 for every element of the passed array

}

void put_grade_into_a_bin( element a[], int grade)
{
// this function determines which "bin" the grade  is to be placed into
// it updates the field how_many of the appropriate bin as necessary
// HINT: use several if statements
}

void compute_percentages( element a[], int class_size )
{
// this function receives the class size and the  array
// and computes the percentage field for every array element
}

/ / This function is given to you:
float sum_percentages(element a[] )
{
 float sum = 0.0;
 for(int i = 0 ; i < size ; i++)
 {
   sum = sum + a[i].percentage;
 }
return sum;
}

void print_percentages(element a[])
{
// this function prints out the percentage field of
// every element in the passed array
}

int find_total(element a[])
{
// this function computes and returns the sum of all the how_many fields of the passed array
}
```

You are to give the complete definitions (give the complete C++ code) of the above functions and then write a `main()` function which makes use of these functions. As a starting part, here is a portion of the main program:

```
#include <iostream>
using namespace std;

struct element
{
  int how_many;
  float percentage;
};

const int size = 6 ; // 6 different "bins" a grade can fall into

// function prototypes
// you don't have to write these in your solution
// they are here simply for sake of "completeness"

void initialize_bins(element a[]);
void put_grade_into_a_bin( element a[], int grade);
void compute_percentages( element a[], int class_size );
float sum_percentages(element a[] );
void print_percentages(element a[]);
int find_total(element a[]);

int main()
{
element bins[size] ; // declare the "bins" array
int num_of_student = 0; // a counter to keep track of number of grades
int grade; // used to store the grade

// first, intialize the how_many field of all the array elements to 0
// do this by invoking one of the functions...

// ask the user to enter a student grade
cout << "Please enter the student grade " ;


// now enter into a loop ..

while (  // some condition    )
{

// you do the rest!!!
}


// a few more things to do,here is a hint:
compute_percentages(bins, num_of_student);

return 0;
}
```

Question 3: [20 points]

Consider the following recursive function and a main program which makes a call to the function:

```
#include <iostream>
using namespace std;

bool recursive_and(bool a[], int size, int left, int right)
// uses recursion to find the logical AND of all the values
// of an array of boolean values.
{
 cout << “size = “ << size << “ left = “ << left << “ right =  “ << right <<
 endl;
 if ( size == 2 )
  return ( a[left] && a[right] ) ;
 else
  return  ( recursive_and(a, size/2, left, left + size/2 - 1) &&
            recursive_and(a, size/2, left + size/2, right)  );

}

int main()
{
bool test[8] = { true, true, true, true,
                 true, true, true, true};
bool answer;
answer = recursive_and(test, 8, 0, 7) ;
cout << “The answer is: “ << answer << endl;
return 0;
}
```

(a) Draw the function call graph assuming that function is invoked as in the main() program.

(b) Give the output produced by the program.

(c) Rewrite the function as an iterative version (i.e. one which does not use recursion).
Use the following prototype:

```
bool iter_and(bool a[], int size)
```

Question 4: [20 points]

Answer **TRUE** or **FALSE** to each of the following questions:

(a) In C++, directly comparing variables of either `float` or `double` for equality can lead to unexpected results.

(b) Arrays are always passed by value to a function.

(c) Variables local to a function, as well as any pass-by-value arguments received by a function, are stored in a special portion of main memory called the stack.

(d) In general, a recursive implementation of a function will run slower than an iterative implementation of the same function.

(e) In a `struct`, everything is `public` by default, unless made explicitly `private`.

ANSWER ONLY ONE OF THE FOLLOWING TWO QUESTIONS:

Question 5: [20 points]

Consider the following program which makes use of the file input/output capability of C++:

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
ifstream infile1("file1.dat");
ifstream infile2("file2.dat");
ofstream outfile("file3.dat", ios::binary | ios::out);
int mick, keith, bigger ;

infile1 >> mick;
infile2 >> keith;

while (( mick != -99) && ( keith != -99))
{
  if  ( mick >= keith)
  {
    bigger = mick;
  }
  else
 {
   bigger = keith;
 }
 cout << "Bigger = " << bigger << endl;
 outfile.write( (char*) &bigger, sizeof(int) );
 infile1 >> mick;
 infile2 >> keith;
}
infile1.close();
infile2.close();
outfile.close();
return 0;
}
```

(a) Give the **output** (both on the screen and any disk file that may be created) produced by the program assuming the Linux 'more' command applied to two input files results in:

```
ted@brownsugar Final_Winter_2011 8:14pm >more file1.dat
1
2
3
4
5
-99
ted@brownsugar Final_Winter_2011 8:16pm >more file2.dat
5
4
3
2
1
-99
```

(b) What will the file size (in number of bytes) be for `file3.dat`?

(c) Explain in words why the following occurs when we try to view the contents of `file3.dat` with the Linux more command as in:

ted@brownsugar Final_Winter_2011 8:16pm >more file3.dat

ted@brownsugar Final_Winter_2011 8:16pm >

Question 6: [20 points]

Implement (by giving the complete code for the class methods, the constructor, and the destructor) a class which will be used to describe a "box" which has a certain length, width, and height. The class defintion is as follows:

```
class box
{
int* l; // pointers to dynamically allocated ints for length , width,
        // and height
int* w;
int* h;
public:
box();  // default constructor
~box(); // destructor
float volume();
void  set_values(int l1, int  w1, int h1);
void  show_values();
box operator+(box arg);
};
```

The class contains three private data members which are pointers to integers. The pointers shall be given values by the constructor function.

The constructor function is to dynamically allocate space from the heap to hold three integers. The three pointers l, w, and h shall be set to the addresses of the dynamically allocated memory. The dynamically allocated memory shall be intialized to the value 1 (giving the box a volume of 1).

The destructor function is to delete the memory which was allocated by the constructor.

The method `volume()` is to return as a float the volume of the box.

The method `set_values(int l1, int w1, int h1)` is used to give new values to the length, width, and height of the box (these values are stored somewhere in the heap and are pointed to by the three pointers l,w, and h).

The method `show_values()` displays the values of the length, width, and height of the box.

The class overloads the "+" operator to add two "boxes" together by adding the lenghts, widths, and heights of the two boxes to yield a "result" box.

Typical use of this class in a `main()` program would be similar to:

```
int main()
{

box mick, keith;
box ron;

mick.show_values(); // will output 1 1 1
keith.show_values(); // will also output 1 1 1
ron = mick + keith ;
ron.show_values(); //  will output 2 2 2

mick.set_values(5,6,7);
mick.show_values(); // will output 5 6 7

return 0;
}
```

END OF EXAM.

"There ain't no cure for the C++ blues" - E. Cochran.