MARKING KEY

The University of British Columbia
Computer Science 252
Final Exam
8:30 AM, Thursday, December 16, 2004

MARKING KEY

Instructors: K. Booth & N. Hutchinson **Time: 150 minutes** **Total marks: 150**

Name_____Student No_____
(PRINT) (Last) (First)

Signature_____Lecture Section _____

☛ **This examination has 20 pages. Check that you have a complete paper.**

This is a closed book exam. Notes, books or other materials are not allowed, but you may use **one** piece of 8-1/2 x 11 paper (both sides) with your own notes either hand-written, photocopied, or both.

Answer all the questions on this paper. Give short but precise answers. Always use point form where it is appropriate. The marks for each question are given in { braces }. Use this to manage your time.

Good luck.

READ AND OBSERVE THE FOLLOWING RULES:

1. Each candidate should be prepared to produce, upon request, his or her Library/AMS card.

2. No candidate shall be permitted to enter the examination room after the expiration of twenty minutes, or to leave during the first twenty minutes of the examination.

3. Candidates are not permitted to ask questions of the invigilators, except in cases of supposed errors or ambiguities in examination questions.

4. **CAUTION** — Candidates guilty of any of the following, or similar dishonest practices shall be immediately dismissed from the examination and shall be liable to disciplinary action.

   a. Making use of any books, papers or memoranda, calculators or computers, audio or visual cassette players, or other memory aid devices, other than those authorized by the examiners.

   b. Speaking or communicating with other candidates.

   c. Purposely exposing written papers to the view of other candidates. The plea of accident or forgetfulness shall not be received.

Page	Marks	Max
3	9.22	10
4	7.48	10
5	7.53	10
6	7.33	10
9	10.45	13
10	15.06	18
11	8.36	9
12	10.95	13
13	3.30	8
14	7.89	12
15	12.56	16
17	5.57	10
18	7.48	11
Total	**113.56**	**150**

1. Multiple choice questions { 40 marks -- 10 marks per page }

On the next four pages there are a series of short fill-in-the-blanks statements. All of your answers are to be selected from the list below.

1)	abstract	24)	dynamic	47)	polymorphism
2)	accessor	25)	elliptical	48)	precedence
3)	alpha	26)	Elvis	49)	prime
4)	assignment operator	27)	executable	50)	private
5)	base	28)	exit	51)	probe
6)	bucket	29)	fragmentation	52)	protected
7)	bus error	30)	function	53)	public
8)	cast	31)	hash function	54)	pure
9)	circular	32)	header	55)	recursive
10)	class	33)	height	56)	reference
11)	collision	34)	key	57)	root
12)	constructor	35)	leaf	58)	segmentation fault
13)	conversion constructor	36)	lumberjack	59)	singly
14)	copy constructor	37)	memory leak	60)	source
15)	core	38)	midnight	61)	static
16)	current directory	39)	mutator	62)	template
17)	dangling pointer	40)	noon	63)	title page
18)	default constructor	41)	null	64)	triply
19)	depth	42)	object	65)	value
20)	derived	43)	overload	66)	virtual
21)	destructor	44)	override	67)	vituperative
22)	doubly	45)	parrot	68)	VPTR
23)	dummy	46)	polygamy	69)	VTABLE

Each statement will have one (or more) ▀▀▀▀▀▀▀▀▀ within it, which is where the missing term or phrase would appear (multiple times in some cases). Choose the best answer from among those above and write its number in the space provided in the first column. Do not write the term or phrase. It may be a good idea to read over the list of terms and phrases before you start answering. Some of the terms listed may not appear in any of the statements, some may appear in more than one statement, and some many appear in exactly one statement.

Continue on to the next page…

You may remove this page from the exam booklet.

Read the instructions on the previous page. Enter **the number** for your answer in the **first** column. **Do not write words**.

(a)	**27**	The final result of successful compiling, linking and loading is the ■■■■■■■■■■ file for a program.
(b)	**15**	On Unix a file with the name ■■■■■■■■■■ is created when there is a bug in your program that causes the program to exit abnormally. It is a good idea to remove these files on a regular basis because they take up a lot of disk space.
(c)	**32**	Normally the declarations for a C++ class are in the ■■■■■■■■■■ file for the class.
(d)	**60**	Normally the definitions for a C++ class are in the ■■■■■■■■■■ file for the class.
(e)	**62**	By convention, when templatization of classes is performed, the definitions for a C++ class are often placed in the ■■■■■■■■■■ file for the class.
(f)	**42**	The result of just the compilation step is an intermediate representation for the data and functions in a single file. This is contained in the ■■■■■■■■■■ file produced by the compilation.
(g)	**48**	The ■■■■■■■■■■ of an operator in C++ determines when it will be evaluated within an expression if there are no parentheses that specify the order of evaluation. When operators are overloaded, the ■■■■■■■■■■ cannot be changed because it is part of the syntax of the C++ language.
(h)	**2**	■■■■■■■■■■ methods always have the keyword **`const`** at the end of their declaration because they may read the values of member variables but they never change those values.
(i)	**39**	■■■■■■■■■■ methods never have the keyword **`const`** at the end of their declaration because they may change the values of member variables.
(j)	**14**	■■■■■■■■■■ methods always have exactly one parameter whose type is the class itself and they always have the same name as the class to which they belong.

(continued on the next page)

(continued from previous page)

(k)	**14**	When objects are assigned a value during initialization or when the return value of a function is assigned to a temporary in the calling function, the [redacted] method is used to make the assignment.
(l)	**43**	A method in a subclass that has a different signature but the same name as a method in the superclass is said to [redacted] the method in the superclass.
(m)	**44**	A method in a subclass class that has the same signature and the same name as a method in a superclass is said to [redacted] the method in the superclass class.
(n)	**21**	The [redacted] method for a superclass is often declared to be virtual because subclasses may add member variables that point to dynamically allocated memory that the superclass will not know about.
(o)	**52**	A [redacted] member variable of a superclass is always accessible (readable and writeable) in any subclass but is never accessible to clients of the superclass.
(p)	**61**	A call to a non-virtual member function of a superclass will use [redacted] binding if the method is invoked through a pointer to an object and the object is an instance of the superclass.
(q)	**61**	A call to a non-virtual member function of a superclass will use [redacted] binding if the method is invoked through a pointer to an object and the object is an instance of a subclass.
(r)	**24**	A call to a virtual member function of a superclass will use [redacted] binding if the method is invoked through a pointer to an object and the object is an instance of the superclass.
(s)	**24**	A call to a virtual member function of a superclass will use [redacted] binding if the method is invoked through a pointer to an object and the object is an instance of a subclass.
(t)	**61**	A call to a virtual member function of a superclass will use [redacted] binding if the method is invoked on an object that is an instance of a subclass.

(continued on the next page)

(continued from previous page)

(u)	**68**	Every object in a class has its own [redacted] if and only if the class has at least one virtual method.
(v)	**69**	All objects in a class share the same [redacted] if and only if the class has at least one virtual method.
(w)	**41**	The [redacted] tree has height zero.
(x)	**19**	In binary trees the [redacted] of a node is the length of the path from the root to that node. This is also referred to as the level of the node.
(y)	**35**	In binary trees every [redacted] node has no children.
(z)	**57**	In binary trees there is always at most one [redacted] node.
(aa)	**41**	Only the [redacted] tree has no nodes at all.
(bb)	**54**	`virtual void method( void ) = 0;` declares an abstract or [redacted] virtual function.
(cc)	**1**	Any class that has one or more abstract functions is considered to be [redacted] and no object in the class can be instantiated.
(dd)	**20**	It is possible for objects in a class with one or more abstract functions to be contained within objects of a [redacted] class, even though no objects can be instantiated on their own.

(continued from previous page)

(ee)	42 69	The [redacted] for a subclass always contains the entire [redacted] for the superclass and perhaps additional entries.
(ff)	68	If two objects have a [redacted] with the same value, they are guaranteed to be instances of the same class.
(gg)	34	In hashing the [redacted] for an entry usually is a unique identifier for the entry.
(hh)	31	In hashing the [redacted] transforms the unique identifier for an entry into an index into an array where the entry might be stored.
(ii)	6	Closed address hashing (also known as chained hashing) uses the index into an array to determine the chain or [redacted] into which the entry will be stored.
(jj)	11	In hashing if two entries are transformed into the same index a [redacted] exists, which must be resolved in some manner by checking a sequence of indices until it is determined whether an entry is in the table.
(kk)	51	In hashing each index in the sequence of indices that are checked to find an entry is called a [redacted].
(ll)	3	In hashing the parameter [redacted] is a measure of how full a table is.
(mm)	49	In hashing it is often convenient if the size of the array is [redacted] because this makes it easier to have a sequence of indices that cover the entire table.
(nn)	45	The [redacted]'s passed on!This [redacted] is no more! He has ceased to be! 'E's expired and gone to meet 'is maker! 'E's a stiff! Bereft of life, 'e rests in peace! If you hadn't nailed 'im to the perch 'e'd be pushing up the daisies! 'Is metabolic processes are now 'istory! 'E's off the twig! 'E's kicked the bucket, 'e's shuffled off 'is mortal coil, run down the curtain and joined the bleedin' choir invisibile!! THIS IS AN EX-[redacted]!!"

Below is a declaration for the templatized **Deque** class from a header file**, Deque.h,** suitable for Phase 1 of the project. It uses a doubly-linked list with a dummy node within each object. On the next page is the definition for the **add()** method from a template file**, Deque.template**. No pre- or post-conditions or class invariants have been specified, but you should assume that they are similar to what was required for the project. The declaration and the source code will be useful when you answer the next three questions on this exam.

header file

```
template <class Item_type>
class Deque
{
public:
   Deque( );
   Deque( const Deque<Item_type>& other );
   ~Deque( );
   void add( const Item_type& item, bool front_end = false )
        throw ( runtime_error );
   Item_type remove( bool front_end = true )
         throw ( logic_error );
   bool empty() const;
   int size() const;
   Deque& operator=( const Deque<Item_type>& other );
   Deque operator+( const Deque<Item_type>& other ) const;
   bool operator==( const Deque<Item_type>& other ) const;
   void  print( bool verbose=false ) const;
private:
   struct Node
   {
      Item_type data;
      Node* next;
      Node* prev;
   };
   int count;    // Number of values currently in the deque
   Node head;    // Dummy node
   static const char* stuff[5];
   virtual const char** labels() const { return stuff; };
   void copy ( const Deque<Item_type>& other );
   void clear ( );
};
```

Continue on to the next page…

You may remove this page from the exam booklet.

Below is a definition for the **`add()`** method suitable for Phase 1 of the project. You may assume that the code works correctly. It uses a doubly-linked list with a dummy node within each object. It throws the appropriate exception if it cannot allocate enough memory to add an item. If an exception is thrown, the **Deque** is left unchanged. This means that subsequent method invocations may rely on the class invariants being true even though the **`add()`** method failed.

template file

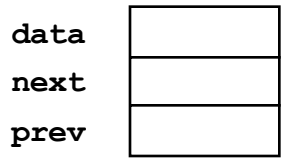
```
template <class Item_type>
void Deque<Item_type>::add(const Item_type& item, bool front_end )
     throw ( runtime_error )
{
   static runtime_error Msg( "Cannot add to Deque" );

   Node* newNode;
   try
   {
       newNode = new Node;
   }
   catch ( bad_alloc )
   {
      throw( Msg );
   }
   if ( front_end )
   {
       newNode->next = head.next;
       newNode->prev = &head;
       head.next->prev = newNode;
       head.next = newNode;
   }
   else
   {
       newNode->next = &head;
       newNode->prev = head.prev;
       head.prev->next = newNode;
       head.prev = newNode;
   }
   newNode->data = item;
   count ++;
}
```

Continue on to the next page…

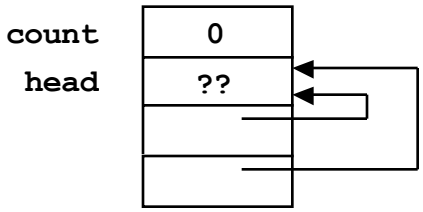
You may remove this page from the exam booklet.

2. Circular doubly-linked lists with dummy nodes { 13 marks }

A **`Node`** within the **`Deque`** class looks like this in memory, with its first member of type **`Item_type`** and its second and third members of type pointer to a **`Node`**.



A newly constructed **`Deque`** looks like the following in memory, with its first member an **`int`** and its second member a **`Node`**. The **`data`** field is undefined ("**`??`**" means the contents of memory are undefined) in the dummy **`Node`** because it never has a value. The **`next`** pointer points to the first **`Node`** in the list and the **`prev`** pointer points to the last **`Node`** in the list. Initially, the two pointers in **`head`** point back to **`head`** itself, because there are no other **`Nodes`** (yet) in the list.

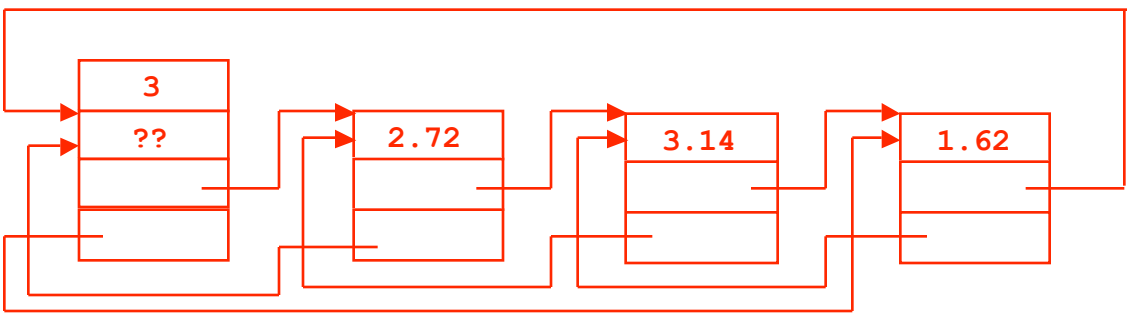


Draw a diagram of the **`Deque A`** after the following four statements have been executed. Show all pointers explicitly. Show all values that are defined; use "**`??`**" to indicate all undefined values.

It may be helpful to look at the source code for the **`add()`** method before making your drawing.

```
Deque<float> A;
A.add( 3.14 );
A.add( 2.72, true );
A.add( 1.62, false );
```

ANSWER: 1 mark for each value or pointer, ?? is mandatory



3. Manipulating circular doubly-linked lists { 18 marks }

Write a complete definition for the templatized version of the **`remove()`** method where **`Item_type`** specifies the type of the items stored in a **`Deque`**. Be sure to include appropriate pre- and post-conditions and throw the proper exceptions.

ANSWER:

```
template < class Item_type >
Item_type Deque<Item_type> :: remove ( bool front_end )
          throw ( logic_error )

// Mutator
// PRE:  (none) OR the Deque must not be empty.
// POST: An item is removed from the Deque and returned, unless the
// Deque is empty, in which case an exception is thrown.

{
  if (count == 0) {
     string msg = (string) labels()[0] + " is Empty";
     throw( logic_error( msg ) );
  }
  Node* oldNode;
  if ( front_end )
     oldNode = head.next;
  else
     oldNode = head.prev;
  Item_type item = oldNode->data;
  oldNode->prev->next = oldNode->next;
  oldNode->next->prev = oldNode->prev;
  delete oldNode;
  count--;
  return item;
}
```

4. The Big Three { 22 marks }

There are three methods that in many classes have a standard implementation using two helper functions, **`copy()`** and **`clear()`**. The helper functions contain all of the code that is specific to the implementation, which means the three methods can use generic code that does not depend in any way on the details of the implementation. In this course we always use the standard generic implementations for these three methods. Most classes require a default constructor and the "The Big Three" methods, each of which has a name that describes its role in the class, similar to how the name of the default constructor describes its role.

(a) { 9 marks } Below are the bodies of the generic implementations for the "Big Three". For each block of code, write the name of the "Big Three" method that corresponds to the code.

```
{
   if ( this != &other )
   {
      clear();
      copy( other );
   };
   return *this;
}
```

ANSWER: overloaded assignment operator — 3 marks

```
{
   clear();
}
```

ANSWER: destructor — 3 marks

```
{
   copy( other );
}
```

ANSWER: copy constructor — 3 marks

(b) { 13 marks } Write the complete implementation for the **`clear()`** helper function as it should appear in the **`Deque.template`** file. Include all necessary template and function header information, and complete pre- and post-conditions. The code you write must be compatible with the way that **`clear()`** is used by the generic methods on the previous page, and it must use the data structures defined in the class declaration. It may be useful to refer to the diagram provided in Question 2.

ANSWER:

```
template <class Item_type >
void Deque<Item_type> :: clear()

// Helper function
// PRE:  none
// POST: deletes all dynamic memory associated with the Deque

{
  Node* p = head.next; // p points to first actual Node (if any)

  while ( p!=&head )  // continue until we get back to the dummy Node
    {
      Node* q = p->next; // remember the next Node
      delete p;          // delete the next actual Node
      p = q;             // get ready for the next iteration
    }
}
```

5. Exception handling { 8 marks }

The following program will produce exactly one line of output when it is compiled under g++ and then run on the undergrad servers with an integer value provided as input. The output is "**Abort**" if the program fails to catch an exception that is thrown some place in the program.

```
#include <iostream>
#include <exception>
#include <stdexcept>

using namespace std;

int main(){
   exception* p = new runtime_error("Dynamic exception");
   int n;
   cin >> n;
   try {
      try {
         switch (n) {
            case 1:  throw (runtime_error("n=1"));
            case 2:  throw (*p);
            case 3:  throw (logic_error("n=3"));
            default: throw (3.14159);
         }
      }
      catch (logic_error&  e1) {cout << e1.what() << endl;}
      catch (exception ) {throw;}
      catch (runtime_error&  e2) {cout << e2.what() << endl;}
      catch (...) {cout << "What?" << endl;}
   }
   catch (exception* e3) {cout << "Unexpected: " << e3->what() << endl;}
   catch (runtime_error& e4) {cout << "Finally: " << e4.what() << endl;}
}
```

In the four lines below, write the one line of output produced by executing the above program with each of the four indicated values for **n** as input.

ANSWER:

Input is 1: **Finally: n=1** 2 marks

Input is 2: **Abort** 2 marks

Input is 3: **n=3** 2 marks

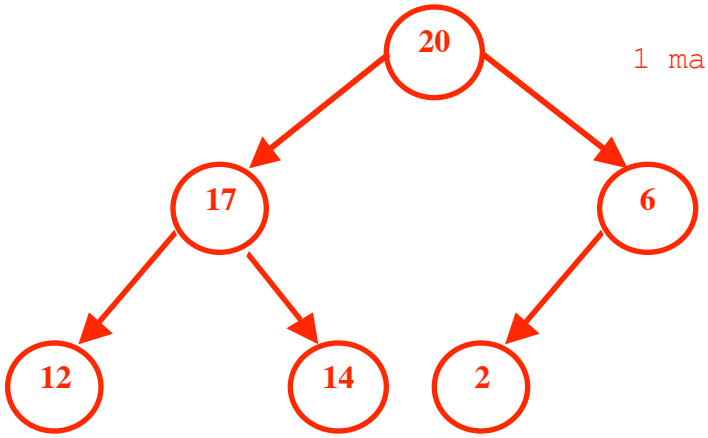
Input is 4: **What?** 2 marks

6. Heaps { 12 marks }

(a) { 6 marks } Draw the tree that corresponds to the heap that is stored in the following array.

20	17	6	12	14	2

ANSWER:



1 mark for each node

(b) { 1 marks } Is this a min-heap or a max-heap?

ANSWER:

1 mark

It is a max-heap.

(c) { 5 marks} At each step the Heapsort algorithm places the current first element in the array in its final position by swapping with the last element in the heap and then adjusts the remaining elements to re-establish the heap property. Show the contents of the array after each of the five steps required for Heapsort to sort the above array.

1 mark for each row (1/2 for 1-2 wrong, 0 otherwise

ANSWER:

17	14	6	12	2	20

14	12	6	2	17	20

12	2	6	14	17	20

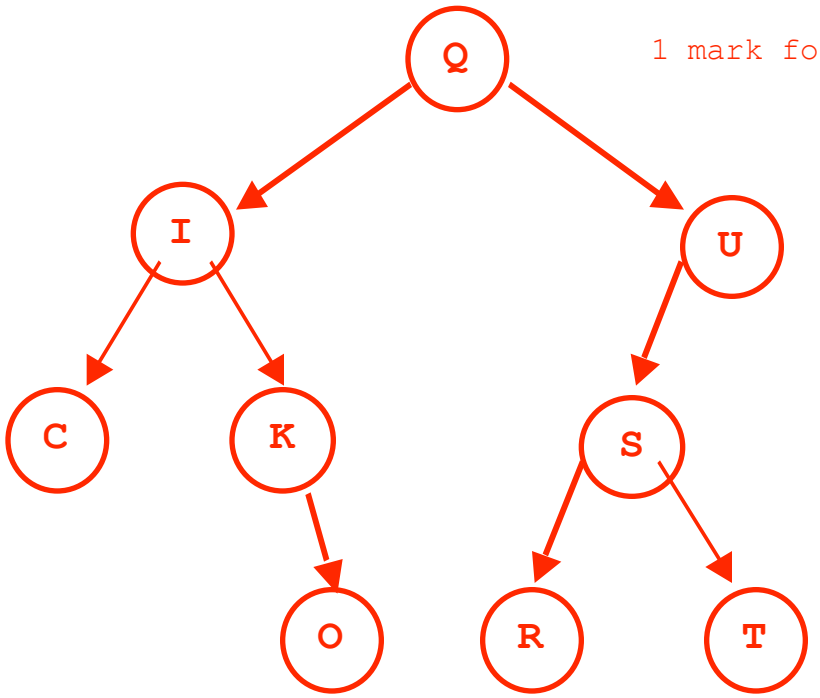
6	2	12	14	17	20

2	6	12	14	17	20

7. Binary search trees { 16 marks }

(a) Insert the nine letters **Q-U-I-C-K-S-O-R-T** into a binary search tree starting with the **Q** and ending with the **T** (the hyphens are not part of the input, just the nine alphabetic characters). Draw the resulting tree.

ANSWER:



1 mark for each node

For the last four parts the answers should be consistent with the tree that the student drew, if the tree is at all plausible.

(b) The root of the tree is **Q** . 1 mark

(c) The preorder traversal of the tree is the string **Q I C K O U S R T** . 2 marks

(d) The postorder traversal of the tree is the string **C O K I R T S U Q** . 2 marks

(e) The inorder traversal of the tree is the string **C I K O Q R S T U** . 2 marks

8. Memory organization { 10 marks }

The question on the next page refers to the following program. You will be asked about the scope and the extent of each identifier in the program.

```
double bippy = 0.0;

class A
{
public:
   A() { x = y++; }
   ~A() { y--; }
   int x;
   static int y;
};

int A::y = 0;

static double foo( int bar )
{
   return bar*bar;
}

int main()
{
   A object;
   {
     A* p = new A;
     bippy = foo( p->y + p->x );
     delete p;
   }
}
```

Continue on to the next page…

You may remove this page from the exam booklet.

Fill in the following table with information about the scope and the extent (storage class or segment) for each of the identifiers in the table.

(a) In the second column indicate the scope (visibility) for each identifier.

(b) In the third column indicate the segment of memory in which each identifier is stored. If the storage class depends on the storage class of an instantiated object, write "depends on the object", otherwise write the name of one of the four storage segments or "no storage" if the identifier has no storage associated with it.

1/2 mark for each entry, any answer is OK for those with *

Identifier	Scope	Extent / storage class or segment
`bippy`	global scope	static segment
`A`	global scope	no storage
`x`	class scope	depends on the object
`y`	class scope	static segment
`foo`	file scope	code segment
`bar`	local scope	stack segment
`main`	global scope	code segment
`object`	local scope	stack segment
`p`	local scope	stack segment
the memory pointed to by `p`	(no scope) *	heap segment

9. Hashing { 11 marks }

(a) { 9 marks } Insert the following three-digit numbers into the two hash tables using open hashing. For the left table use linear hashing, but for the second use quadratic hashing. In both cases the hash function is the number modulo 7 (i.e., the remainder after dividing the number by 7)

737, 936, 506, 814, 220

The first number (**737**) has been inserted into the left (Linear Hashing) table for you. Insert **737** into the right (Quadratic Hashing) table and then insert each of the other numbers into both tables in the order given.

1 mark for each entry

Linear Hashing
737
506
814
936
220

Quadratic Hashing
737
506
220
936
814

(b) { 2 marks } What is the average number of probes required to look up the five numbers in the two hash tables? Calculate the average for each table and enter the two (different) numbers in the spaces below.

ANSWER:

Linear Hashing: 2.2 probes 1 mark

Quadratic Hashing: 1.8 probes 1 mark

Continue your answers here – make sure to identify the question(s) whose answer(s) are here!

MARKERS: Be sure to check the last two pages for answers to earlier questions!

Continue your answers here – make sure to identify the question(s) whose answer(s) are here!

MARKERS: Be sure to check the last two pages for answers to earlier questions!