E-GENTING PROGRAMMING COMPETITION, 2004

WORKSHOP HANDOUT, WEEK 5, VERSION 1

1 BACKGROUND

1.1 *Table Driven Programming*

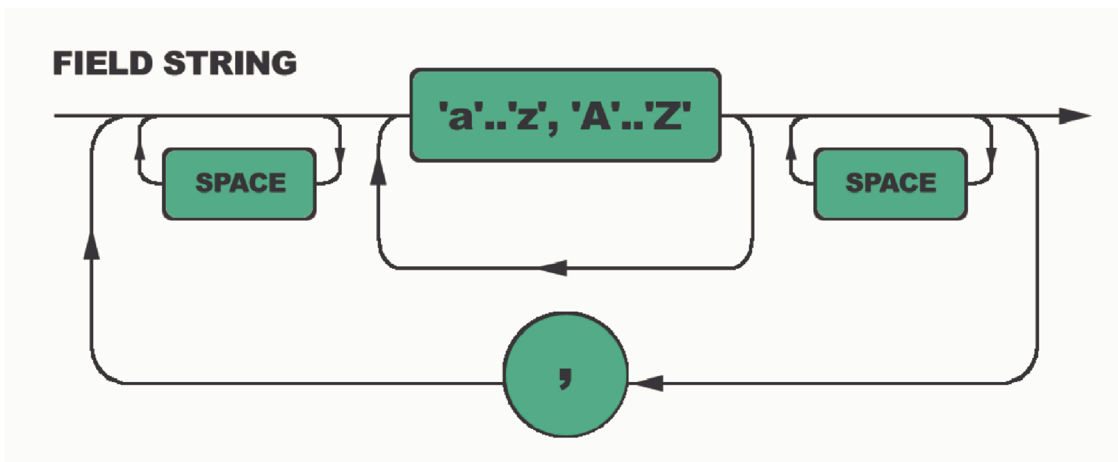
```
static final int    STEP_TABLE[] = {1, 4, 2, 8};
                              // Step sequence table
static final int    NEXT_PHASE[][] =
     {{1, 2, 3, 0}, {3, 0, 1, 2}};
                              // Next phase map

// To move one step to the right:
phase = NEXT_PHASE[0][phase];
wiperControl.setSCR(STEP_TABLE[phase]);

// To move one step to the left:
phase = NEXT_PHASE[1][phase];
wiperControl.setSCR(STEP_TABLE[phase]);
```

1.2 *Syntax Diagrams and Simple Parsers*

Consider this syntax diagram for a string of field identifiers separated by commas and spaces.



A parser is easily derived from the syntax diagram:

```
    sr = new StringReader (fieldString);
    ch = sr.read();
    for (;;) {
        while (ch == ' ') ch = sr.read();
        if ( ! fieldChar(ch))
            reject ("invalid field identifier");
        sw = new StringWriter ();
        while (fieldChar(ch)) {
            sw.write (ch);
            ch = sr.read();
        }
        while (ch == ' ') ch = sr.read();
        // process the identifier in sw.toString()
        if (ch != ',') break;
        ch = sr.read();
    }
    if (ch != -1) reject ("unexpected character");
```

1.3 *Tailoring Library Classes*

```
     struct Cust_t {
          unsigned long  custId;
          string         custSurname;
          string         custGivenNames;
     };
```

```
class CustIsLess_c {
public:
     bool
     operator () (
          const Cust_t    &c1,
          const Cust_t    &c2)
     const
     {
          if (c1.custSurname < c2.custSurname) return 1;
          if (c1.custSurname > c2.custSurname) return 0;
          return c1.custGivenNames < c2.custGivenNames;
     }
};
```

```
// Declare a vector of Cust_t structures

vector <Cust_t> custVec(100);

// Instantiate the comparator and execute the sort

CustIsLess_c custIsLess;
sort (custVec.begin(), custVec.end(), custIsLess);

// Alternatively (but essentially the same thing)

sort (custVec.begin(), custVec.end(), CustIsLess_c());
```

1.4 Add Operation

```
const int PRECISION = 128;

bool
Add (
     bool       acc[PRECISION],       // Accumulator
     const bool op[PRECISION])        // Operand
{
     int        i;                    // Index
     bool       c;                    // Carry bit
     c = 0;
     for (i = 0; i < PRECISION; i++) {
          acc[i] ^= op[i] ^ c;
          c = (!acc[i] & (op[i]|c)) | (op[i]&c);
     }
     return c;
}
```

2 EXERCISES

2.1 Number Converter

Using a syntax diagram and lookup tables, write a program that converts a number between zero and ninety-nine in English text, to its equivalent value expressed in digits. For example, ‘seventeen’ should convert to 17 and ‘eighty-five’ should convert to 85. The program should avoid storing repeated copies of words. For example, the word ‘eighty’ should be stored only once.

2.2 Telescope Controller

The controller of a telescope must be programmed to point the telescope at a new heading in the minimum time. Given a current telescope heading in degrees east of North and a new heading in the same units, write a program to determine whether the telescope should turn clockwise or anticlockwise (when viewed from the top) in order to turn the minimum distance to point towards the new heading. The program must display the direction and number of degrees through which the telescope must turn.

2.3 Packet Sequencing

A communications protocol labels data packets with packet numbers which start at 0 increment through to 31 and then return to zero again. Packets can be lost, duplicated and received out-of-order. Each receiver maintains a current packet number 'c', being the number of the last acknowledged packet. The receiver only acknowledges a packet when all the packets in the stream up to and including the acknowledged packet have been received. Packets with packet numbers from c + 1 to c + 15 (mod 32) are assumed to be packets after the current packet. Packets with packet numbers c – 16 to c – 1 (mod 32) are assumed to come before the current packet. Write a program to determine the position of a received packet relative to the current packet given any current packet number and any received packet number. Packets before the current packet should have negative relative positions and packets after the current packet should have positive relative positions. For example, if the current packet number is 18 and the received packet number is 16, the relative position of the received packet is –2.

2.4 Alternative Sorting Techniques

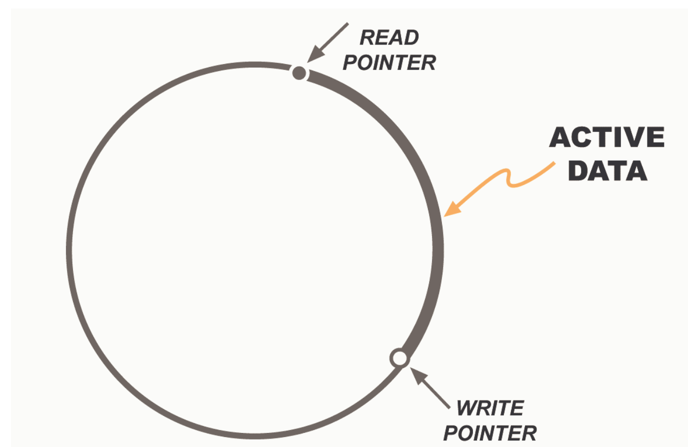
Write a program that uses two methods to sort a vector of Cust_t structures containing pseudo-randomly generated strings and checks that the result of the two sorts is the same. The first sort must use the function-call operator described in the lecture and the second sort must use multiple calls to the stable_sort template.

2.5 Subtract Function

Program a Subtract function similar to the Add function described in the lecture.

2.6 Ring Buffer

A ring buffer is data structure, sometimes referred to as a circular queue, which can store a finite amount of data on a first-in, first-out basis.



A ring buffer consists of a storage buffer, usually drawn with its end connected to its beginning in the form of a ring, and two pointers: a read pointer and a write pointer.

When data is appended to the ring buffer, the data is copied to the buffer starting at the position of the write pointer and then the write pointer is moved to the end of the copied data.

When data is extracted from the ring buffer, the data in the ring buffer, starting from the location addressed by the read pointer, is copied into application memory and then the read pointer is moved forward to effectively delete the extracted data from the buffer.

Your task is to write a ring buffer class in accordance with the specification below:

```
class RingBuffer_c {
public:
     RingBuffer_c (size_t capacity);
                    // Construct the ring buffer
     ~RingBuffer_c ();
                    // Destruct the ring buffer
     size_t RingWrite (const void *buf, size_t len);
                    // Append data to the buffer
     size_t RingRead (void *buf, size_t len);
                    // Extract data from the buffer
     size_t RingFree ();
                    // Return the amount of free storage
};
```

The constructor must allocate the buffer memory and initialise the buffer pointers. Its declaration is:

```
class RingBuffer_c {
public:
     RingBuffer_c (size_t capacity);
     //...
};
```

`capacity`
: is the useable capacity of the ring buffer in octets.

The destructor must release the buffer memory. Its declaration is:

```
class RingBuffer_c {
public:
     ~RingBuffer_c ();
     //...
};
```

The `RingWrite` method must append data to the ring buffer. If the length of the data to be appended is greater then the remaining capacity of the ring buffer, `RingWrite` must append as much data as it can and then return the length actually appended. `RingWrite`'s declaration is:

```
class RingBuffer_c {
public:
     size_t RingWrite (const void *buf, size_t len);
     //...
};
```

`buf`
: is a pointer to a buffer containing the data to be appended to the ring buffer.

`len`
: is the number of octets that the application would like to have appended to the ring buffer.

`return value`
: is the number of octets actually appended to the ring buffer. If the value of the `len` parameter is greater than the available storage, the return value may be less than `len`.

The `RingRead` method must extract data from the ring buffer. `RingRead` must extract data from the ring buffer in the same octet-by-octet order in which it was appended to the ring buffer. `RingRead`'s declaration is:

```
class RingBuffer_c {
public:
     size_t RingRead (void *buf, size_t len);
     //...
};
```

`buf`
: is a pointer to a buffer that is to receive the data extracted from the ring buffer.

`len`
: is the number of octets that the application would like to have extracted from the ring buffer.

`return value`
: is the number of octets actually extracted from the ring buffer. If the ring buffer contains less data than the value of the `len` parameter, the return value may be less than `len`.

The `RingFree` method must return the number of unused octets in the ring buffer. `RingFree`'s declaration is:

```
class RingBuffer_c {
public:
     size_t RingFree ();
     //...
};
```

`return value`
: is the number of unused octets in the ring buffer. The `RingWrite` method must be able to append this number of bytes to the ring buffer.

2.7 Pocket Calculator

Use a syntax diagram to design and program a pocket calculator program that processes the add, multiply and precedence (i.e. bracket) operators on unsigned integers.