Instructions:

1. There are 63 marks.
2. Turn off all electronic media and store them under your desk.
3. If you are asked to author code, it must be implemented using C++. Unless we specifically ask for syntactically correct C++ code, we may ignore one or two small syntactical errors such as a missing semi colon.
4. You may use `cout` and `endl` instead of `std::cout` and `std::endl`.
5. You may ask only one question during the examination: "May I go to the washroom?"
6. Asking any other question will result in a deduction of 5 marks from the exam grade.
7. If you think a question is ambiguous, write down your assumptions and continue.
8. Do not leave during first hour or after there are only 15 minutes left.
9. Do not stand up until all exams have been picked up.
10. If a question only asks for an answer, you do not have to show your work to get full marks; however, if your answer is wrong and no rough work is presented to show your steps, no part marks will be awarded.

1. (4 points) Write a function `unsigned int paired_bits( unsigned short n );` that returns the number of times that two consecutive bits in `n` are equal. For example, given the argument `0b1100011101000010`, there are three consecutive pairs of `1`s (the three consecutive `1`s contain two pairs of consecutive `1`s) and five consecutive pairs of `0`s (three pairs appearing in the four consecutive `0`s), so the result would be eight. An `unsigned short` occupies 16 bits, so there are 15 possible pairs. Recall that `3` in binary is `0b11`.

```
unsigned int paired_bits( unsigned short n ) {
```

2. (2 points) This is a continuous function that is 0 if both arguments are negative, 1 if either argument is greater than or equal to 1, and continuous between these two planes.

   Write this function f that returns 0 if $x \leq 0$ and $y \leq 0$, 1 if $x \geq 1$ or $y \geq 1$, x if $y \leq 0$, y if $x \leq 0$, and $x + y - xy$ otherwise. The conditions must be tested in the order given.

```
double f( double x, double y ) {
```

3. (3 points) Write a function that returns the position of the leading 1 (that is, the first bit equal to 1 that is closest to the most significant bit), where bit 0 is the least-significant bit. If the number is zero, return 32. For example, 1 would return 0, 2 and 3 would return 1, and 4 through 7 would return 2.

```
unsigned int leading_one( unsigned int n ) {
```

4. (4 points) Given two arrays `a1[]` and `a2[]` with capacities `cap1` and `cap2`, respectively, copy all entries that do not appear in both arrays into a third array `out[]`, and return the number of entries that were copied to that third array. This is called the *symmetric difference* between the two arrays. You should assume that the output array is sufficiently large, so with a capacity of at least `cap1 + cap2`.

```
std::size_t symm_diff(
       int const a1[], std::size_t const cap1,
       int const a2[], std::size_t const cap2,
       int out[]
) {
```

5. (6 points) Describe each of the following and give a code snippet (code, which copied into `main()`, would compile and execute) that demonstrates each of these. Your description should explain why these are considered problems.

   1. A wild pointer.
   2. A memory leak.
   3. A dangling pointer.

6. (5 points) The array idx[] contains cap indices (indexes) that are guaranteed to be in order with the first being greater-than or equal to 0 and the last less-than or equal to the capacity of a second array ary[]. Write a function that returns the maximum sum of the entries in ary[] from idx[0] to idx[1] - 1, idx[1] to idx[2] - 1, all the way up to the sum from idx[cap - 2] to idx[cap - 1] - 1. The easiest implementation will require nested loops, and you will have to track the overall maximum sum as well as the sum of the current range.

   For example, given the two arrays

```
double      ary[8]{1.2, 3.5, 2.5, 5.3, 2.6, 2.1, 8.2, 5.3};
std::size_t idx[3]{2, 4, 7};
```

   where the capacity cap == 3, this function would return the maximum of $2.5 + 5.3$ (indices 2 and 3) and $2.6 + 2.1 + 8.2$ (indices 4, 5 and 6).

```
double max_sum( double ary[], std::size_t idx[], std::size_t cap ) {
```

7. (6 points) Write a selection sort, but unlike the algorithm shown in class, yours will find the smallest entry and move it into the first available position. For example, if the array is `{-4, 7, 5, -9, -2, 9, 2, -1, -2, -5, 3, 6}` after one iteration, the array will contain `{-9, 7, 5, -4, -2, 9, 2, -1, -2, -5, 3, 6}`, and after a second, the array will contain `{-9, -5, 5, -4, -2, 9, 2, -1, -2, 7, 3, 6}`.

   You may not use any standard library functions, but you are welcome to implement helper functions which you can then call. You may solve this either by putting all your code in the `selection_sort(...)` function body, or you can also implement the helper functions and call them from within the `selection_sort(...)` function body. Two suggested helper functions are given, but you can design your own, instead.

```
void selection_sort( double array[], std::size_t cap ) {
```

```
// You only have to implement this if you use it.
void swap( double &a, double &b ) {
```

```
// You only have to implement this if you use it.
//  - Find the minimum starting from array[begin]
//    and going up until array[end - 1]
std::size_t find_min( double array[], std::size_t begin, std::size_t end ) {
```

8. (5 points) Write a function `char *max_concat( char *s1, char *s2, std::size_t n )` that concatenates up to n characters from two null-terminated C-style strings (`s1` and `s2`) into a new dynamically allocated string. The function should return a pointer to the new concatenated string, which will be null-terminated. The function must not copy more than n characters in total.

   For example, if we were concatenating the strings `"Hello "` and `"Waterloo"`, if `n == 20`, the result would be the string `"Hello Waterloo"`, if `n == 10`, the result would be `"Hello Wate"`, and if `n == 4`, the result would be `"Hell"`. If `n == 0`, the result would be the empty string: an array containing a single null character.

   If you want to use a helper function `std::size_t strlen( char *s )` (or any other help function), you are welcome to implement it.

```
char *max_concat( char *s1, char *s2, std::size_t n ) {
```

9. (3 points) Draw the call stack the first time the program gets to the comment that says "Draw the call stack here..." Your call stack need only identify which sections are for which function call, what are the parameters and local variables on the call stack, together with their values.

   Note that the function `A(...)` is recursive, and you will have more than one call to `A(...)` on the call stack.

```
unsigned int A( unsigned int m, unsigned int n, unsigned int p ) {
  if ( p == 0 ) {
    // Draw the call stack here...
    return m + n;
  } else if ( n == 0 ) {
    if ( p == 1 ) {
      return 0;
    } else if ( p == 2 ) {
      return 1;
    } else {
      return m;
    }
  } else {
    return A( m, A( m, n - 1, p ), p - 1 );
  }
}

int main() {
  int n{ 42 };
  n = A( 1, 1, 1 );
  std::cout << n << std::endl;
  return 0;
}
```

10. (4 points) Write a function that takes an array `endpts` of floating-point numbers as input. You are guaranteed that the capacity `cap` of the array is an even number. You are also guaranteed that the floating-point numbers are increasing in the array; for example,

```
{-10.3, -5.2, -3.0, 2.5, 5.7, 6.1, 18.8, 23.0}
```

This array of eight floating-point numbers represents four intervals

$$[-10.3, -5.2], [-3.0, 2.5], [5.7, 6.1], [18.8, 23.0].$$

That is, all numbers between -10.3 and -5.2 inclusive, all numbers between -3.0 and 2.5 inclusive, and so on. An array of $2n$ floating-point numbers would represent n closed intervals.

You are asked to write a function to determine if the argument assigned to `x` is in one of these intervals. If it is found, you will indicate the interval by a value between `0` and `cap/2 - 1`. If it is not in one of these intervals, you will return `cap/2`.

For example, if the parameter `x` is assigned `-9.3` with the above array, the returned value would be `0`; if it was `6.1`, `2` would be returned; and if it was `-15.2`, `-4.0` or `6.103`, in all of these, `4` would be returned.

```
unsigned int is_in( double x, double endpts[], int cap );

unsigned int is_in( double x, double endpts[], int cap ){
    // The array has an even capacity
    assert( cap%2 == 0 );
```

11. (8 points) Implement the constructor, destructor, and the three member functions described below of a linked list class that stores integers. Negative integers are kept in one linked list, positive integers in another, and attempts to insert a 0 are ignored. Calls to pop a negative or positive number removes the first node of the corresponding list, and the value stored in that node is returned. If there is nothing to pop, `0` is returned.

```
class Node;
class List;

class Node {
  public:
    Node( int value, p_next = nullptr );

    int value_;
    Node *p_next_;
};

Node::Node( int value, p_next ):
value_{ value },
p_next_{ p_next },
count_{ 1 } {
  // Empty constructor...
}

class List {
  public:
    List();
   ~List();

    bool push( int n );
    int pop_pos();
    int pop_neg();
  private:
    Node *p_neg_h_;
    Node *p_pos_h_;
};
```

Additional space for the previous question...

12. (4 points) Write the member operator `operator<( LL const &other ) const` for the `LL` class that determines if this linked list is "less than" the argument list. We start with two pointers, `p1` initialized with `p_head_` and `p2` initialized with `other.p_head_`.

    1. If both `p1` and `p2` are `nullptr`, then return `false`, as the lists have all entries equal.
    2. Otherwise, if `p1` equals `nullptr`, return `true`.
    3. Otherwise, if `p2` equals `nullptr`, return `false`.
    4. Otherwise, if the value stored at `p1` is less than the value stored at `p2`, return `true`.
    5. Otherwise, if the value stored at `p1` is greater than the value stored at `p2`, return `false`.
    6. Otherwise, increment both pointers (that is, move both pointers to the next node in each of the linked lists).

```
class Node {
  public:
    int value_;
    Node *p_next_;
};

class LL {
  public:
    bool operator<( LL const &other ) const;
  private:
    Node *p_head_;
};

bool LL::operator<( LL const &other ) const {
```

13. (7 points) Recall that Project 5 implemented sets using linked lists, where any one set could not have two nodes that store the same value. The `merge( Set &other )` member function moves all nodes in the `other` set that have values that do not appear in this set into this set. There are no calls to `new` and `delete`. If a node in the other set contains a value in this set, that node stays in the other set.

    Implement the member function `merge(...)` from Project 5 as follows:

    1. As long as `other.p_head_` is not `nullptr` and stores the address of a node not containing a value in this set, move that node to the start of this set and point `other.p_head_` to the next node in the other set. This will need to be a while loop.
    2. At this point, set a local variable `Node *ptr` to `other.p_head_`. This is either `nullptr` or it is storing the address of a node storing a value known to be in this set. Then proceed as follows:
    3. As long as `ptr` is not `nullptr` do the following:
        (a) As long as `ptr->next_` is not `nullptr` and stores the address of a node not containing a value in this set, move that node to the start of this set and point `ptr->next_` to the next node in the other set. This, too, will need to be a while loop.
        (b) At this point, `ptr->next_` is either `nullptr` or it is storing the address of a node storing a value known to be in this set. Assign `ptr` the value of `ptr->next_` and repeat Step 3.
    4. Return the number of nodes merged into this set.

    You may implement a `bool Set::find( int n ) const` member function that returns `true` if `n` is in this set, and `false` otherwise, and use it in your solution.

```
class Set {
    public:
        // Move any items from 'other', whose values
        // do not appear in 'this', to 'this'.
        // Leave any items that already appear
        // in both sets, in both sets.
        std::size_t merge( Set &other );
    private:
        Node *p_head_;
};

class Node {
    public:
        // You don't need any public member functions

    private:
        int   value_;
        Node *next_;

    // Allow any member function in the class
    // 'Set' to access or modify the member
    // variables of any instance of this class.
    friend class Set;
};

// Implmement your solution on the next page
```

```
std::size_t Set::merge( Set &other ) {
```

14. (2 points) Circle the statements that are true about inheritance and polymorphism given one base class and a class derived from that base class.

    1. The derived class can add new member variables and new member functions not in the base class.
    2. The derived class can remove unnecessary member variables found in the base class.
    3. For polymorphism of member functions to work, the `virtual` keyword must be used in the base class.
    4. When you call an overridden member function in the derived class, the function in the base class is immediately called first, and only then is the function defined in the derived class called.

You may use the space below for solutions to previous questions, but be sure to indicate your solution continues on Page 17.

You may use this space for solutions to previous questions, but be sure to indicate your solution continues on Page 18.