Instructions:

1. The exam will be graded out of 73.
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 the following functions:

```
bool bit( unsigned int n, unsigned int k );
bool set( unsigned int &n, unsigned int k );
bool flip( unsigned int &n, unsigned int k );
bool reset( unsigned int &n, unsigned int k );
```

where:

- The first returns `true` if the k^{th} bit of `n` is `1` and `false` otherwise.
- The second sets the k^{th} bit of `n` to `1` and returns `true` if the original value of that bit was `1` and `false` otherwise.
- The third flips the k^{th} bit of `n` (so `0` to `1` and `1` to `0`) and returns `true` if the original value of that bit was `1` and `false` otherwise.
- The last sets the k^{th} bit of `n` to `0` and returns `true` if the original value of that bit was `1` and `false` otherwise.

For example, if the first argument was a local variable `n` that was assigned `6` and the second was `2`, then all four functions would return `true` and the second leaves `n` unchanged while the third and fourth change the value of `n` to `2`. You may, of course, use helper functions.

2. (6 points) In this question, you can use `cout` and `endl` without prefixing them with `std::` namespace. Implement the function `void waterloo( unsigned int n )` that draws the following:

   When $n = 0$, the output is a new line only.

   When $n = 1$, the output is:

```
\/\/
```

   When $n = 2$, the output is:

```
\  /\  /
 \/  \/
```

   When $n = 3$, the output is:

```
\    /\    /
 \  /  \  /
  \/    \/
```

   When $n = 4$, the output is:

```
\      /\      /
 \    /  \    /
  \  /    \  /
   \/      \/
```

   You may, of course, use helper functions.

   1. 2 marks for progress towards a solution.
   2. 1 mark for getting the correct spacings around the and /.
   3. 1 mark for stuff printed on n lines (even if it is wrong).
   4. 1 mark for a progression of spaces from one line to the next.
   5. 1 mark for escaping the backslash
   6. 1 bonus marks for trying to use helper functions and 2 bonus marks for actually having useful helper functions even if they do not work perfectly (but not more than 6/6)

3. (5 points) Write a function that finds the maximum common entry (a value that appears in both arrays) found in two arrays. If the two arrays have no common entries, return `-std::numeric_limits<double>::infinity()`.

```
void max( double a1[], std::size_t cap1,
          double a2[], std::size_t cap2 );
```

For example, if the two arrays are `{5, 3, 11, 9, 5, 7}` and `{8, 3, 12, 3, 7, 8, 1, 4}`, the maximum common entry is `7`.

4. (4 points) Write a function that rounds an integer (`int`) to the closest integer that has just three significant digits (all others are zero). This means that for any number greater than or equal to 1000 will have only zeros after the third significant digit. For example, 532432 would be rounded to 532000, 32552 to 32600, and −5725 to −5730. and any number less than one thousand in absolute value is left unchanged. The rounding works as follows:

   - If the digit to the right of the third significant digit is 5 or greater, then round the third digit up and set all digits to the right of the third significant digit to zero. For example, 57350, 57362, 57379, 57384 and 57399 are all rounded up to 57400.
   - Otherwise, just set all digits to the right of the third significant digit to zero. For example, 57349, 57338, 57321, 57316 and 57307 would all be rounded down to 57300.

   For more examples, 54350 rounds to 54400, and −8224999 rounds to −8220000 (the minus sign does not impact the rounding). Also, 599500 rounds to 600000. At the extreme, 999500 gets rounded to 1000000, but that's fine, because all the digits after the third are still zero.

   Hint: This question can be done using `*10`, `/10` and `%10`.

```
int round3( int n ) {
```

5. (4 points) List all problems with this code? Recall $2^{10} \approx 1000$.

```
int my_function( unsigned int n ) {
  int *a_data;

  if ( n < 1000 ) {
    n = 1000;
    a_data = new int[n];
  }

  if ( (a_data != nullptr) && (n < 100000) ) {
    n = 100000;
    a_data = new int[n];
  }

  for ( unsigned short k{ 0 }; k <= n; ++k ) {
    a_data[k] = k;
  }

  int sum{ 0 };

  for ( unsigned short k{ 0 }; k <= n; ++k ) {
    sum += a_data[(3*k) % n];
  }

  return sum;
}
```

6. (5 points) Inside `main()`, you have an `int` array `arr` of capacity `cap`, and there is a Boolean-valued function `bool f( int value )` that was written by another author. Write code that would re-order the entries in the array and where `n` is assigned a value so that:

   1. All entries from index `0` to index `n - 1` return `true` when passed to `f(...)`, and
   2. All entries from index `n` to index `cap - 1` return `false` when passed to that same function.

   For example, if the array entries were

   ```
   8, 14, 15, 5, 19, 9, 2, 0, 3, 10, 5, 3, 6
   ```

   and the function `f` returned `true` whenever the argument is `10` or greater, then `n` must be assigned `4`, but there are many possible orderings of the array; for example, both of these would be valid:

   ```
   14, 10, 15, 19, 8, 9, 2, 0, 3, 5, 5, 3, 6
   19, 10, 15, 14, 9, 2, 5, 8, 5, 3, 3, 0, 6
   ```

   All that is required is that those entries for which the function `f` return `true` occupy the first four entries, and all other entries appear after that.

   ```
   // You will assume that someone else has authored
   // the definition of this function elsewhere
   bool f( int value );

   int main() {
      std::size_t cap{ /* some value... */ };
      int arr[cap];

      // Assume code here fills up the array

      std::size_t n{ 0 }; // You must find 'n'
   ```

   1. 1 mark for returning the correct n, with 0.75 marks if the value of n is off by one, and 0.5 if a mistake that is easily fixed miscalculates n.
   2. 1 mark for testing
   3. 2 marks for a reasonable strategy to move all entries that are true to the start, and all false entries to the other end.
   4. 1 more mark for a correct solution with no issues like memory leaks.

   NO MARKS OFF if they use a dynamically sized local array.

7. (3 points) Write the output of the program below.

```
#include <iostream>

void f( int *p_int   );
void g( int *&pr_int );

void f( int *p_int ) {
  // Assume the operating system returns address 0x000ffff for the new.
  p_int = new int{42};
}

void g( int *&pr_int ) {
  // Assume the operating system returns address 0xffff0000 for the new.
  pr_int = new int{42};
}

int main() {

  int *p_f{nullptr};
  int *p_g{nullptr};

  f(p_f);
  g(p_g);

  std::cout << p_f << " : " << p_g << std::endl;
  return 0;
}
```

8. (4 points) Write a function `scan` that takes an array `double arr[]` and its capacity `cap` as arguments, and replaces the `k`$^{\text{th}}$ entry with the sum of the original entries in the array from index `0` up to `k - 1`. This will be done for each index `k` from `0` to `cap - 1`. The sum of all the entries in the original array is the value that will be returned.

   For example, the array with entries `{3.0, 2.1, 5.0, 4.0, 4.1, 7.0}` would have those entries replaced with `{0.0, 3.0, 5.1, 10.1, 14.1, 18.2}` and `25.2` would be returned. Notice that the first entry is always zero.

9. (6 points) Write a modified selection sort algorithm that sorts while simultaneously eliminates duplicate entries. Suppose you find that `n` entries are duplicates of other entries. In this case, the sorted entries will start at index `n` and go up to index `cap - 1`, and the value `n` will be returned. The function declaration will be

```
std::size_t sort_unique( int arr[], std::size_t cap );
```

For example, **{4, 5, 5, 9, 9, -2, 2, -2, -2, 5, 3, 6}** becomes **{?, ?, ?, ?, ?, -2, 2, 3, 4, 5, 6, 9}** and the value `5` is returned. What is in entries with indices `0` through `4` is not specified, so what appears in those entries will not impact your grade in this question.

Note: You must modify the selection sort algorithm.

10. (6 points) Write a class that implements an $n \times n$ matrix using a single array. Create an $n \times n$ matrix and initialize all the entries to `0`. You will implement this matrix as an array with $\texttt{n}^2$ entries where the rows start at indices `0`, `n`, `2n`, etc.

    You will author:

    1. The constructor that takes a non-negative integer `n` that allocates sufficient memory. If the user enters an argument of 0, then create a 1×1 matrix.
    2. A destructor that deallocates any dynamically-allocated memory.
    3. An `operator()` that takes two unsigned integers as arguments and returns the $(i, j)^{\text{th}}$ entry of the matrix, and throwing a `std::domain_error` exception if either argument is greater than or equal to `n`. You just have to return the correct entry of the array corresponding to (i, j); it will automatically be returned by reference.
    4. A `zero_diag` function that returns `true` if either diagonal (the two diagonals that form an "X") sums to zero.

```
class Matrix {
  public:
    Matrix( unsigned int n );
    double &operator()( unsigned int i, unsigned int j );
    bool zero_diag() const;
   ~Matrix();
  private:
    // Declare any member variables here



};

// Constructor, destructor and member function definitions here
```

11. (3 points) Write a recursive function that implements the following mathematical expression:

$$F(n) = \begin{cases} 0 & n = 0 \\ 3 & n = 1 \\ F(n-1) + 2F(n-2) & n \geq 2 \end{cases}$$

```
// `FR' for `r'ecursive
unsigned int FR( unsigned int n ) {









}
```

Next, calculate the first five, six or seven values of this function, determine a mathematical formula for the answers, and implement it using a non-recursive function. You may wish to recall that 2^k may be calculated using `1 << k`.

```
// `FC' for `c'alculated
unsigned int FC( unsigned int n ) {









}
```

12. (3 points) Write a function that returns the number of divisors of a given non-negative integer argument. For example, the number of divisors of any prime number p is $\sigma(p) = 2$: 1 and p itself. (σ is the Greek letter "sigma".) Also, $\sigma(1) = 1$ and $\sigma(0) = 0$. The number of divisors of 6 is $\sigma(6) = 4$, for $1, 2, 3, 6$ are divisors of 6, and the number of divisors of 24 is $\sigma(24) = 8$, for $1, 2, 3, 4, 6, 8, 12, 24$ divide 24.

```
unsigned int sigma( unsigned int n ) {









}
```

Next, you will assume that your implementation of the above function is correct. Recall that $\sum_{k=1}^{n} F(k)$ equals $F(1) + F(2) + \cdots + F(n)$, so the function $p(n)$ is defined recursively as:

$$p(n) = \frac{1}{n} \sum_{k=1}^{n} p(n-k)\sigma(k).$$

The sum is guaranteed to have a factor of n that can be divided out exactly. Implement the function $p(n)$ using recursion.

```
unsigned int p( unsigned int n ) {
```

13. (5 points) The following takes an array of a given capacity and raises each entry to the power n.

```
void power( double array[], std::size_t capacity, unsigned int n ) {
  if ( n == 0 ) {
    for ( std::size_t k{ 0 }; k < capacity; ++k ) {
      array[k] = 1.0;
    }
  } else if ( n == 1 ) {
    // Write the state of the stack before
    // this consequent block returns.
    return;
  } else if ( n%2 == 0 ) {
    for ( std::size_t k{ 0 }; k < capacity; ++k ) {
      array[k] *= array[k];
    }

    power( array, capacity, n/2 );
  } else {
    double *a_tmp{ new double[capacity] };

    for ( std::size_t k{ 0 }; k < capacity; ++k ) {
      a_tmp[k] = array[k];
      array[k] *= array[k];
    }

    power( array, capacity, n/2 );

    for ( std::size_t k{ 0 }; k < capacity; ++k ) {
      array[k] *= a_tmp[k];
    }

    delete[] a_tmp;
  }
}
```

Write the state of the stack when the parameter `n` equals `1`. Assume that the first time you call `new`, the address `0xab00` is returned, the second time, `0xac00` is returned, the third time, `0xad00` is returned, and so on. You only have to show the call stack, so local variables and parameters, and any value they have. Be sure to indicate where on the stack each function call starts.

```
int main() {
  std::size_t const N{ 2 };
  double data[N]{ 0.5, 1.0 };
  power( data, N, 7 );
  return 0;
}
```

14. (8 points) Implement the constructor, destructor, and the two member functions described below of a linked list class that stores integers but where entries are always inserted so that the linked list is sorted with the smallest value first. If a value `n` is being inserted into a linked list that already contains that value, then instead of inserting a new node, the `count_` of the node already storing `n` will be incremented by one. When `pop_front()` is called, if the count of the first node is `1`, the node is removed as with a normal linked list, otherwise the `count_` is simply decremented by one. When `pop_front()` is called on an empty list, throw a `std::runtime_error` exception. Note that under no circumstances will `count_` ever be zero.

```
class Node;
class SL;

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

    int value_;
    Node *p_next_;
    unsigned int count_;
};

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

// Short for "S"orted "L"ist
class SL {
  public:
    SL();
   ~SL();
    void insert( int n );
    int pop_front();
  private:
    Node *p_head_;
};
```

15. (4 points) Write the member function `clear_middle()` for the `Linked_list` that deletes all nodes between the first node and the last node, and returns the number of nodes that were deleted. If the linked list has two or fewer nodes, do nothing and return `0`. You must implement `clear_middle()` by accessing or modifying the member variables of the linked list and node classes.

```
class Linked_list;
class Node;

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

    int value_;
    Node *p_next_;
};

class Linked_list {
public:
    // Constructor, destructor and other member functions...

    std::size_t clear_middle();

private:
    Node *p_head_;
};
```

16. (3 points) Write the output of the program below.

```
#include <iostream>

// Class declarations
class A;
class B;
class C;
class D;

// Function declarations
int main();

// Class definitions
class A {
    public:
        virtual void what() const;
        virtual void me() const;
};
class B : public A {
    public:
        virtual void what() const override;
};
class C : public B {
    public:
        virtual void what() const override;
        virtual void me() const override;
};
class D : public A {
    public:
        virtual void what() const override;
};

// Member function definitions
void A::what() const { std::cout << "A::what" << std::endl; }
void A::me()   const { std::cout << "A::me"   << std::endl; }
void B::what() const { std::cout << "B::what" << std::endl; }
void C::what() const { std::cout << "C::what" << std::endl; }
void C::me()   const { std::cout << "C::me"   << std::endl; }
void D::what() const { std::cout << "D::what" << std::endl; }

// Function definitions
int main() {
    A a{};
    B b{};
    C c{};
    D d{};
    A &ar{ c };
    B &br{ c };
    a.what();
    ar.what();
    b.what();
    br.what();
    b.me();
    ar.me();

    return 0;
}
```