Ellie Makin (erm67)

C/C++ supervision work 3

C++ Questions

1.

```
class LinkList
{
public:
    LinkList(int* array=nullptr, int size=0) {
        if (size <= 0) {
            head = -1;
            tail = nullptr;
        }
        else {
            head = *array;
            tail = new LinkList(array+1, size-1);
        }
    }

    LinkList(const LinkList& other) {
        head = other.head;
        if (other.tail != nullptr)
            tail = new LinkList(*other.tail);
        else
            tail = nullptr;
    }

    LinkList& operator=(const LinkList& other) {
        LinkList* temp = new LinkList(other);
        tail = temp->tail;
        temp->tail = nullptr;
        delete temp;
        return *this;
    }

    ~LinkList() {
        if (tail != nullptr)
            delete tail;
    }

    int pop() {
        int result = head;
        if (tail != nullptr)
        {
            head = tail->head;
            LinkList* temp = tail;
            tail = temp->tail;
            temp->tail = nullptr;
```

```
            delete temp;
        }
        return result;
    }

private:
    int head;
    LinkList* tail;
};
```

2. It is initialised the first time that execution reaches the definition in `f`.

3.

```
class Matrix
{
public:
    /*
        | a b |
        | c d |
    */
    Matrix(float a, float b, float c, float d) :
        a(a), b(b), c(c), d(d)
    { }

    Matrix operator+(const Matrix& m)
    {
        return Matrix(
            a + m.a,
            b + m.b,
            c + m.c,
            d + m.d
        );
    }

    Matrix operator-(const Matrix& m)
    {
        return Matrix(
            a - m.a,
            b - m.b,
            c - m.c,
            d - m.d
        );
    }

    Matrix operator*(const Matrix& m)
    {
        return Matrix(
            a*m.a + b*m.c,
            a*m.b + b*m.d,
            c*m.a + d*m.c,
            c*m.b + d*m.d
        );
    }

    Matrix operator/(const Matrix& m)
    {
```

```
        float det = 1/(m.a*m.d - m.b*m.c);
        Matrix inv(
            det*m.d,
            det*-m.b,
            det*-m.c,
            det*m.a
        );
        return *this * inv;
    }

    float a, b, c, d;
};
```

4.

```
class Vector
{
public:
    Vector(float x0, float x1) :
        x0(x0), x1(x1)
    { }

private:
    float x0;
    float x1;
};

class Matrix
{
public:
    /*
    Matrix stuff
    */

    friend Vector operator*(const Vector& v)
    {
        return Vector(
            a*v.x0 + b*v.x1,
            c*v.x0 + d*v.x1
        );
    }

    /*
    Matrix stuff
    */
};
```

5.

```
class A
{
public:
    ~A(){}
};
Stack(const Stack& s) {
```

```
        head = s.head;
        if (s.tail != nullptr)
            tail = new Stack(*s.tail);
        else
            tail = nullptr;
    }

    Stack& operator=(const Stack& s) {
        Stack* temp = new Stack(s);
        tail = temp->tail;
        temp->tail = nullptr;
        delete temp;
        return *this;
    }
class B :
    public A
{
public:
    B() {
        i = new int(1);
    }

    ~B() {
        delete i;
    }

    int* i;
};

int main()
{
    A* test = new B();
    delete test;
}
```

6. Abstract classes are never instantiated directly - only their derived classes. However, they are commonly used as the target type of casts, so often it will be that an object of a non-abstract type is stored in a variable of an abstract type. In this case, the destructor of the abstract class is likely to be the wrong one, but if it was not declared as `virtual`, then it will be called by default. Declaring it as `virtual` solves this problem.

7. Since the function exits without calling `fclose` if it encounters malformed input, eventually it will reach the limit for the maximum number of concurrently open files. This is the fixed C code:

```
int process_file(char *name) {
    FILE *p = fopen(name, "r");
    if (p == nullptr) return ERR_NOTFOUND;
    while (...) {
        ...
        if (...) {
            fclose(p);
            return ERR_MALFORMED;
        }
        process_one_option();
        ...
```

```
    }
    fclose(p);
    return SUCCESS;
}
```

And then the C++ version:

```
class FileReader
{
public:
    FileReader(char* name) {
        pFile = fopen(name, "r");
    }

    ~FileReader() {
        fclose(pFile);
    }

    FILE* pFile;
};

int process_file(char *name) {
    FileReader f = FileReader(name);
    FILE* p = f.pFile;
    if (p == nullptr) return ERR_NOTFOUND;
    while (...) {
        ...
        if (...) return ERR_MALFORMED;
        process_one_option();
        ...
    }
    return SUCCESS;
}
```

8. (and 9)

```
#include <stdio.h>

template<typename T>
class Stack {
public:
    Stack() : head(0) {}

    Stack(const Stack& s) {
        Item* ps = s.head;
        Item** ppt = &head;
        while (ps) {
            *ppt = new Item(ps->val);
            ps = ps->next;
            ppt = &(*ppt)->next;
        }
    }

    ~Stack() {
        Item* p = head;
```

```
        while (p != nullptr) {
            Item* temp = p;
            p = p->next;
            temp->next = nullptr;
            delete temp;
        }
    }

    Stack& operator=(const Stack& s) {
        Stack* temp = new Stack(s);
        head = temp->head;
        temp->head = nullptr;
        delete temp;
        return *this;
    }

    void push(T v) {
        Item* new_head = new Item(v);
        new_head->next = head;
        head = new_head;
    }

    T pop() {
        if (head == nullptr)
            return T();
        T result = head->val;
        Item* temp = head;
        head = head->next;
        temp->next = nullptr;
        delete temp;
        return result;
    }

private:
    struct Item {
        T val;
        Item* next;
        Item(T v) : val(v), next(0) {}
    };

    Item* head;
};

int main() {
    Stack<char> s;
    s.push('a'), s.push('b'), s.push('c');
    printf("%d\n", s.pop());
    Stack<char> t = Stack<char>(s);
    printf("%d\n", s.pop());
    printf("%d\n", t.pop());
    printf("%d\n", t.pop());
    printf("%d\n", t.pop());
}
```

10.

```cpp
#include <stdio.h>

template <int n, int d>
struct divide_check {
    enum { result = (n % d == 0) || divide_check<n, d-1>::result };
};

template <int n>
struct divide_check<n, 1> {
    enum { result = 0 };
};

template <int n>
struct prime {
    enum { result = !divide_check<n, n-1>::result };
};

int main() {
    printf("%d\n", prime<7>::result);
}
```

11. If we compile with `g++ -S -m32 -masm=intel`, then we can see in the compiled assembly that the result of the computation is already there:

```
    .file   "test.cpp"
    .intel_syntax noprefix
    .text
    .section    .rodata
.LC0:
    .string "%d\n"
    .text
    .globl  main
    .type   main, @function
main:
    // main intro
    add    eax, OFFSET FLAT:_GLOBAL_OFFSET_TABLE_
    sub    esp, 8
    push   1                    // <-- literal value here
    lea    edx, .LC0@GOTOFF[eax]
    push   edx
    mov    ebx, eax
    call   printf@PLT
    // more stuff
```

12.

```cpp
#include <stdio.h>

class Employee
{
public:
    Employee(int h, int s) :
        hours(h), salary(s)
    { }
```

```
    virtual int wage() {
        return hours * salary;
    }

protected:
    int hours, salary;
};

class Manager :
    public Employee
{
public:
    Manager(int h, int s, int b) :
        Employee(h, s), bonus(b)
    { }

    int wage() override {
        return Employee::wage() + bonus;
    }

private:
    int bonus;
};

int main() {
    Manager m(40,10,20);
    Employee* e = &m;
    printf("%d\n", e->wage());
    return 0;
}
```

The C++ solution is better than the C version because:

1. The constructors are part of the classes, so we don't have to use a macro to generate new instances.
2. The `wage` function is contained within the classes, so we don't have to pass the struct again as a parameter.
3. Since `Manager` is a subclass of `Employee`, we no longer have to cast the pointer, which may lead to unsafe behaviour in some cases.
4. We avoid (some) code repetition since we are able to inherit functions and variables from the parent class.

2008 Paper 3 Question 3

a.

```
int BitQueue::size() {
    return valid_bits;
}
```

b.

```
void BitQueue::push(int val, int bsize) {
    if (valid_bits + bsize > 32)
        throw;
    int new_bits = val & ((1 << bsize) - 1);
    queue = (queue << bsize) | new_bits;
    valid_bits += bsize;
}
```

c.

```
int BitQueue::pop(int bsize) {
    if (bsize > valid_bits)
        throw;
    int result = queue >> (valid_bits - bsize);
    valid_bits -= bsize;
    return result;
}
```

d.

```
void sendmsg(const char* msg) {
    BitQueue q;
    int code, len;
    while (true) {
        switch (*msg) {
            case 'a':
                code = 0b0; len = 1;
                break;
            case 'b':
                code = 0b10; len = 2;
                break;
            case 'c':
                code = 0b1100; len = 4;
                break;
            case 'd':
                code = 0b1101; len = 4;
                break;
            case '\0':
                code = 0b111; len = 3;
        }
        q.push(code, len);
        if (q.size() >= 8)
            send(q.pop(8));
        if (*msg == '\0')
            break;
        msg++;
    }
    if (q.size() > 0) {
        q.push(0, 8 - q.size());
        send(q.pop(8));
    }
}
```