Harry Langford
hjel2@cam.ac.uk

1 2008 Paper 3 Question 3

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2008p3q3.pdf

A hardware engineer stores a FIFO queue of bits in an int on a platform with 32-bit ints and 8-bit chars using the following C++ class:

```
class BitQueue{
        int valid_bits;         // the number of valid bits held in the queue
        int queue;        // least significant bit is most recent bit added
public:
        BitQueue(): valid_bits(0), queue(0) {}
        void push(int val, int bsize);
        int pop(int bsize);
        int size();
};
```

(a) Write an implementation of `BitQueue::size`, which should return the number of bits currently held in queue.

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

(b) Write an implementation of `BitQueue::push`, which places the bsize least significant bits from val onto queue and updates valid.bits. An exception should be thrown in cases where data would otherwise be lost.

```
/* Using the standard exception library is not necessary but it's
 * good practice. We can throw any object. However, if the object
 * derives exception then general purpose code can handle the exception
 * better -- and this scales better for large projects rather than
 * throwing various unrelated objects from every different function */
#include <exception>
#include <string>
using namespace std;

struct BitQueueFullException : exception{
    int overflow;
    explicit BitQueueFullException(int i) : overflow(i){}
    const char* what() const throw(){
            return "BitQueue overflowed by " + to_string(i) + "bits";
    }
};

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

(c) Write an implementation of `BitQueue::pop`, which takes `bsize` bits from queue, provides them as the bsize least significant bits in the return value, and updates valid.bits. An exception should be thrown when any requested data is unavailable.

```
#include <exception>
#include <string>
using namespace std;

struct BitQueueEmptyException : exception{
        const char* what() const throw(){
                return "BitQueue was popped from when empty";
        }
};

int BitQueue::pop(int bsize){
    if (bsize > valid_bits){
        throw BitQueueEmptyException();
    }
    if (!bsize){
        return 0;
    }
    valid_bits -= bsize;
    if (bsize == 32){
            return queue&-1;
    }
    else{
            return (queue&((((1<<bsize-1)-1<<1)+1)<<valid_bits))>>valid_bits;
    }
}
```

(d) The hardware engineer has built a communication device together with a C++ library function send to transmit data with the following declaration.

```
void send(char);
```

Use the BitQueue class to write a C++ definition for:

```
void sendmsg(const char* msg);
```

Each of the characters in msg should be encoded, in index order, using the following binary codes: `'a'=0`, `'b'=10`, `'c'=1100` and `'d'=1101`. All other characters should be ignored. Successive binary codes should be bit-packed together and the code 111 should be used to denote the end of the message. Chunks of 8-bits should be sent using the send function and any remaining bits at the end of a message should be padded with zeros. For example, executing `sendmsg("abcd")` should call the send function twice, with the binary values 01011001 followed by 10111100.

```
struct UnsupportedCharException : exception{};
```

```
void sendmsg(const char* msg){
    BitQueue bq;
    while (msg[0]){
        switch (msg[0]){
            case 'a':
                bq.push(0, 1);
                break;
            case 'b':
                bq.push(2, 2);
                break;
            case 'c':
                bq.push(12, 4);
                break;
            case 'd':
                bq.push(13, 4);
                break;
            default:
                throw UnsupportedCharException();
        }
        if (bq.size() >= 8){
            send(bq.pop(8));
        }
        msg++;
    }
    bq.push(7, 3);
    if (bq.size() > 8){
        send(bq.pop(8));
    }
    bq.push(0, 8);
    send(bq.pop(8));
}
```

2 2009 Paper 3 Question 1

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2009p3q1.pdf

Explain all of the following C or C++ features. You may use a short fragment of code to complement your explanation.

(a) The declaration of a C++ class illustrating constructor, variable and method.

Declaring a C++ class declares its existence. This means it can be instantiated (if it is not abstract), "friended" and subclassed. A constructor is called when we create a new instance of the class. This allows us to instantiate the class into a valid state. Methods operate on the state of the class. Method declarations can be made inside the class while the method is defined elsewhere as showed below.

```
class C{
        int x;
        const int y;
public:
        C(int xp, int yp) : x(xp), y(yp){};
        void inc();
};

void C::inc(){
        x++;
}
```

(b) The use of a virtual destructor

Destructors are called when an object goes out of scope. When an object goes out of scope, the objects destructor and all the destructors of all superclasses will be called (in order). This means that attributes are deallocated properly. However, if the static type of an object is a superclass, then the destructor which is called is the superclasses destructor. This may cause a memory leak or other errors (for example if the subclass allocated memory on heap). However, if the destructor is declared as virtual then the destructor that is called will be the runtime type (the subclass). This will avoid memory leakage as intended. However, in order to determine at runtime which destructor to call, we will need a vtable. This will create additional time and space overhead.

```
#include <vector>
using namespace std;

struct A{
    vector<int> *vec1 = new vector<int>(10000, 0);
    A()= default;
    virtual ~A() {
        cout<<"A"<<endl;
        delete vec1;
    }
};

struct B : A{
    B()= default;
    vector<int> *vec2 = new vector<int>(10000, 0);
    ~B() override{
        cout<<"B"<<endl;
        delete vec2;
    }
};

void f(){
        A *a = new B(); // static type of A (A) != runtime type (B)
        delete a; // without a virtual function this will leak vec2
}
```

(c) The difference between `malloc()` and `free()`; and new and delete

`malloc(i)` is a function in the `stdlib.h` header which allocates `i` untyped bytes onto the heap and returns a pointer to the first one. We can then interpret these bytes as whatever type we wish.

`new` can be placed before an object instantiation and will instantiate the object on the heap and return a typed pointer to it. This pointer will be typed and the objects constructor will be run.

The main differences between `new` and `malloc()` are that `malloc` returns an untyped pointer while `new` returns a typed pointer; and `malloc` does not initialise memory while `new` will run the objects constructor.

`free(p)` takes a pointer `p` to bytes that were allocated on the heap and deallocates them. It does not do this recursively and there is no way to overload `free` to perform different deallocation for different datatypes. If the datatype we wish to deallocate itself contains pointers to other data on the heap we wish to deallocate, then we must remember to free that ourselves. We must therefore know the runtime type of any object we wish to deallocate and remember to deallocate it and all attributes properly.

delete takes a pointer to an object p, runs the objects destructor (and the destructor of all superclasses) and deallocates the memory where p is held. The destructors can be changed to deallocate attributes or recurse through a data structure.

The main difference between delete and free() is that delete runs the destructor and deallocates while free() only deallocates. delete is therefore higher level than free().

```
struct list{
        struct list *next;
}

void free_list(struct list *lst){
        while (*lst){
                struct list *tmp = lst;
                lst = lst->next;
                free(tmp);
        }
}

int main(void){
        // allocates 8 bytes on the heap and returns an untyped pointer.
        void *p = malloc(10000);
        // we can then interpret these bytes however we want
        int *i = p;
        struct list *l = p;
        l->next = malloc(8);
        /* there is no way to define a destructor in C with free
         * so free(l) will cause a memory leak
         * we have to define our own function to free memory and remember
         * the actual type and call the correct function to deallocate the
         * data */
        free_list(l);
}

#include <vector>
using namespace std;

struct A{
        vector<int> *vec = new vector<int>(10000, 0);
        A()= default;
        ~A(){
                delete vec;
        }
}
int main(){
        A *a = new A(); // the return type is typed and initialised
        delete a; // the destructor is run so vec is also deallocated
        return 0;
}
```

(d) Overloading an operator

Operator overloading allows us to program in more concise and intuitive ways, without calling many different verbose functions for basic things. However, it is very easy to create inconsistencies. For example in the below example == is defined but != is not defined – which could cause usability issues.

```
#include <iostream>
using namespace std;

class Vec2{
        const int x;
        const int y;
public:
        Vec2(int xp, int yp) : x(x2), y(y2){}
        bool operator==(const Vec2d &other) const{
                return x == other.x && y == other.y;
        }
}

int main(){
        Vec2 v1(1, 1);
        Vec2 v2(1, 2);
        if (v1 == v2){ // this is true
                cout<<"equal!"<<endl;
        }
        if (!(v1 != v2)){ // this fails compilation
                cout<<"equal!"<<endl;
        }
        return 0;
}
```

(e) Pointer arithmetic

Pointer arithmetic allows us to increment and decrement pointers. This is particularly useful when iterating through an array or a string in C . However, pointer arithmetic is very low level and notoriously prone to bugs. It is less relevant in C++ as we often use higher-level types such as *vector¡¿* or *string*. Classic misuses of pointer arithmetic are iterating through memory in the heap – usually consecutively allocated memory is allocated adjacent in the heap, however this is not guaranteed and so this is not allowed.

```
bool contains(const char *str, const char c){
        while (str[0]){ // while not "\0"
                if (str[0] == c){
                        return true;
                }
        }
        return false;
}
```

However, in C++ it would be more common to write code using higher level objects which do not need pointer arithmetic.

```
bool contains(string str, const char c){
        for (int i = 0; i < str.length(); i++){
                if (str[i] == c){
                        return true;
                }
        }
        return false;
}
```

(f) Catching and throwing exceptions including the passing of a user-defined structure

Exceptions are designed for control flow in exceptional circumstances. Some methods can fail and may do so recoverably. In these cases, an exception is thrown which will unwind the stack until the exception is caught. Unwinding the stack involves running the destructors of all objects which go out of scope. If a destructor throws an exception, then the program will terminate.

```
#include <exception>
#include <string>
using namespace std;

struct IndexOutOfBoundsException{
        IndexOutOfBoundsException(int iarg, narg) : i(iarg), n(narg){}
        const char *what() const throw(){
                return "Index " + to_string(i) + "is out of bounds for" +
                        "BoundsCheckedArray of length " + to_string(n);
        }
}

class BoundsCheckedArray{
        const int n;
public:
        BoundsCheckedArray(int i) : n(i) {}
        int &operator[](int i){
                if (i < 0 || i >= n){
                        throw IndexOutOfBoundsException(i, n);
                }
        }
};
```

(g) The meaning of the keywords `static` and `const`

A `static` variable is initialised when the program starts – there is one shared copy of a static variable. `static` variables in functions are preserved between function calls. `static` variables in classes or structs are shared by all instances of that class or struct.

`static` variables and functions are only accessible from inside the file in which they were declared.

```
int f(){
        static int i = 0;
        return i++;
}

int main(){
        f(); // returns 0
        f(); // returns 1
        f(); // returns 2
        ...
        return 0;
}

class C(){
        static int x;
public:
        C(int i) : x(i);
        int getX(){return x;}
};
```

```
int main(){
        C c1(1);
        c.getX();  // 1
        C c2(10);
        c.getX();  // 10
        c2.getX(); // 10
        return 0;
}
```

3 2012 Paper 3 Question 3

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2012p3q3.pdf

In this question, where appropriate, you may use a short fragment of code to complement your explanation.

(a) (i) What is the difference between a local and global variable in C? (Consider variable scope, storage and initialisation)

Global variables are declared and defined in the global scope. Therefore they can be accessed by any function in the program (and in other programs if declared extern). While Local variables are defined inside functions (in the local scope) and are therefore only visible inside functions.

Local variables are stored on the stack while global variables are stored in the data segment.

Global variables are initialised when the program starts while local variables are initialised only when the function declares them. Local variables go out of scope when the block they were declared in ends – at this point they are deallocated. Global variables are only deallocated when the program ends. Static local variables are an exception to these rules – they are initialised at startup and are stored on the data segment. Static local variables can be viewed as global variables which are only visible from one function.

```
int x = 0;
void f(void){
        int y = 0;
        x; // valid!
        y; // valid!
}
x; // invalid!
y; //valid!
```

(ii) What are the properties of a static member variable in a C++ class?

Static member variables are shared between all instances of a class. They are stored on the data segment and are initialised when the program first starts.

```
class A{
        static int x = 0;
        A()=default;
public:
        int getX(){return x++;}
}

int main(){
        A a1;
        A a2;
        A a3;
        a1.getX(); // 0
        a2.getX(); // 1
        a1.getX(); // 2
        a3.getX(); // 3
        return 0;
}
```

(b) (i) Briefly explain pointer arithmetic in C. Give an example code snippet involving pointers in which it would be *inappropriate* to use pointer arithmetic, and explain why.

In C, we can add offsets to pointers and access the memory saved there. This is infamous for causing critical bugs, security exploits and segmentation faults. Without great care, pointer arithmetic is very likely to lead to issues.

A common assumption that causes bugs is that consecutively malloced memory will be adjacent. The following code attempts to implement a dynamically sized string:

```
void addChar(char *start, int *len, char c){
        malloc(1);
        *len++;
        *(start + len) = c; // equivalent to start[len] = c;
}
```

However, the C specification makes no guarantees that memory will be allocated contiguously on the heap. Therefore this code will eventually fail (and hopefully cause a segmentation fault) – even if it works for a short while before it does.

(ii) Explain how in some respects pointers are equivalent to arrays, and give one respect in which they differ.

Both arrays and pointers can be used to store contiguous blocks of memory of user-specified size. Both arrays and pointers are passed as a reference to the first element in this contiguous block meaning that only one pointer-size is passed as argument. Additionally, neither array nor pointers are bounds-checked – therefore it is equally easy to perform an illegal access on an array as on a pointer.

Arrays are strongly typed while pointers have no type constraints. You cannot have an array of type `void []` or typecast arrays to arrays of other types; while you can have pointers of type `void *` and can typecast pointers.

(c) Explain why a function might be declared virtual in a C++ superclass.

Declaring a function virtual in a C++ superclass means the implementation of the function called is decided at runtime based on the dynamic type of the object rather than the objects static type. This allows java-style polymorphism. However, it also increases the runtime overhead associated with function calls as we have to look up which function to call in the vtable rather than jump to an address known at compile time.

(d) (i) How does the use of `void *` pointer in C allow a form of polymorphism? Give an example function declaration using the `void *` pointer.

`void *` is an untyped pointer. This allows us to operate on different datatypes without restriction. However, it also means we don't know the size of the datatype we are operating on or the size of any elements in the datatype or what operations we can do on the datatype. Therefore for all nontrivial operations, we have to pass the size of the datatype, the size of elements in the datatype and any functions we wish to apply. This style of programming enables polymorphism but is very awkward to program with and does not typecheck. For example in the example below, there is no guarantee that the comparison function compares the same type as the array we are sorting.

```
void sort(void *start, void *end, int esize, int cmp(void *, void *));
```

(ii) What is the main problem with the use of `void *`, and how does C++ improve on this? Give the improved function declaration in C++ for your example function in part (d)(i).

If we use `void *` we do not know different occurrences of `void *` have the same type. For example we could call `quicksort` with arguments of type `long long` but a comparison function which compared `int`s. This would give invalid results but would compile correctly and would not fail at runtime – it will just interpret each long long as being two integers and give an unintended result.

C++ addresses this by using templates and polymorphism. Polymorphism allows us to cast any object to any of its superclasses safely and therefore allows us to pass any subclass as argument to any function expecting a superclass.

Templates allow us to define implementations of functions which take specific types (or specific values). These can ensure that the types passed are the same or have a particular relation. For example the declaration below ensures that all arguments have the same type. C++ templates can be used to implement compile-time metaprogramming – a Turing Powerful language.

```
template <class T> void quicksort(T *start, T *end, int cmp(T, T));
```

(e) (i) Why might it be useful to define a copy constructor for a C++ class? Give an example of a copy constructor for a simple class.

Copy constructors allow us to initialise a new object with the same values as an old object – whatever that may be. This allows us to copy an existing object in the correctly and initialise an object to a state without running the same set of operations as we did on the original object. If we want to share some attributes then we can and this can be more space efficient.

For example consider the following struct `Vec` which uses reference counting. When we create a new `Vec`, we do not want to copy the reference count. Copy constructors allow us to copy other fields without copying the reference count.

```
struct Vec{
        int refs;
        int x, y;
        Vec(int i, int j): x(i), y(j), refs(1){}
        Vec(const Vec& other): x(other.x), y(other.y), refs(1){}
};
```

(ii) Why might it be useful to explicitly define the assignment operator (=) for a C++ class? Give an example definition of the assignment operator for a simple class.

Copy constructors can only be used when declaring a new object – copy constructors always allocate new memory. If we have an existing object that we wish to copy another object into, then we must explicitly define the assignment operator.

Consider the reference counting example. If we were to assign to this class then we would not want to copy reference counts. Explicitly defining = allows us to do this.

```
Vec::operator=(const Vec& other){
        x = other.x;
        y = other.y;
}
```