Question 1. True or False

Circle **T** if the statement is true, otherwise circle **F** if the statement is false.

1. C macro functions always produce syntactically correct C/C++ code. **T** **(F)**

2. Once a macro is used, it cannot be used during the same macro expansion. **(T)** **F**

3. To prevent infinite recursion, it is not possible for concatenation to form a token that can be used as a macro. **T** **(F)**

4. The main benefit of optional compilation over inheritance is performance (speed). **T** **(F)**

5. Clever use of C preprocessor macro allows for static introspection. **T** **(F)**

6. You can protect your C macros from side effects by adding parentheses around arguments that may have side effects, e.g. `#define foo(x) (x) * 3` **T** **(F)**

Question 2. Multiple Choices

Which of the following are higher order functions?

i. `sum`

ii. `zip`

iii. `sizeof`

iv. `constexpr` functions

(v.) `filter`

Question 3. Short Question

a) Expand the C preprocessor macro function, MYSTERY.

```
#define S(s) #s                          #define v hello
#define C(t, a...) t ## a                #define m world
#define X(t, a...) C(_ ## t, a)
#define I(f, a) f a
#define P(p) (p, printf)

#define MYSTERY(m) X(v, I(C, P(s)))(v, S(v) S(m), m)

MYSTERY(m);    // expand this
```

```
MYSTERY(m)
MYSTERY(world)   // no direct # or ## in MYSTERY so must
                 // expand argument first

X(v, I(C, P(s)))(v, S(v) S(world), world) // cannot expand v because
                                          // of ##, can expand I

X(v, I(C, (s, printf)))(v, S(v) S(world), world)   // expanded P
X(v, C (s, printf)) (v, S(v) S(world), world)      // expanded I
X(v, sprint) (v, S(v) S(world), world)             // expanded C
C(_v, sprintf)(v, S(v) S(world), world)            // expanded X
_vsprintf(v, S(v) S(world), world)                 // expanded C
_vsprintf(hello, S(v) S(world), world)             // replaced v
_vsprintf(hello, "v" S(world), world)              // expanded S

// Cannot replace v inside S(v) because of #
_vsprintf(hello, "v" "world", world)
```

b) `PP_NARG` can count the number of arguments that is passed into the macro function. However, it doesn't work correctly if no arguments are passed in. Fix the macro so that it also supports no arguments.

```
#define PP_NARG(...) PP_NARG_(__VA_ARGS__, PP_RSEQ_N())
#define PP_NARG_(...) PP_ARG_N(__VA_ARGS__)
#define PP_ARG_N( _1, _2, _3, _4, _5, _6, _7, _8, _9, N, ...) N
#define PP_RSEQ_N() 9, 8, 7, 6, 5, 4, 3, 2, 1, 0
```

```
#define PP_NARG(...) PP_NARG_(~, ##__VA_ARGS__, PP_RSEQ_N())
#define PP_NARG_(...) PP_ARG_N(__VA_ARGS__)
#define PP_ARG_N( _1, _2, _3, _4, _5, _6, _7, _8, _9, _10, N, ...) N
#define PP_RSEQ_N() 9, 8, 7, 6, 5, 4, 3, 2, 1, 0
```

c) Rewrite the following code snippet to use optional compilation for deciding whether to perform eviction or not, and whether to print debugging messages or not. Your answer must compile when the `-Wunused-variable` and `-Wunused-parameter` compiler options are turned on.

```
bool debug_enabled = true;
bool evict_enabled = true;

void add_sample(std::queue<int> & q, int v, size_t max=20)
{
    if (evict_enabled && q.size() >= max) {
        int e = q.front();
        q.pop();
        if (debug_enabled) {
            std::cout << e << " was evicted\n";
        }
    }

    q.push(v);
    if (debug_enabled) {
        std::cout << v << " was added\n";
    }
}
```

```
void add_sample(std::queue<int> & q, int v, size_t max=20)
{
#ifdef EVICT_ENABLED
    if (q.size() >= max) {
#ifdef DEBUG_ENABLED
        int e = q.front();
        std::cout << e << " was evicted\n";
#endif /* DEBUG_ENABLED */
        q.pop();
    }
#else
    (void)max; // avoid unused parameter error
#endif /* EVICT_ENABLED */

    q.push(v);
#ifdef DEBUG_ENABLED
    std::cout << v << " was added\n";
#endif /* DEBUG_ENABLED */
}
```

Question 4. Programming Questions

a) Write a *safe* macro function, `FIND_MIN`, which takes an array of arbitrary type and a size, and finds the minimum value within the array. You may assume the array element type supports the less than operator. The `FIND_MIN` is used like this:

```
int a[] = { 3, 5, 93, -3, 7, 4, -12 };
constexpr int nelems = sizeof(a)/sizeof(int);
int * p = nullptr;

FIND_MIN(p, a, nelems);
if (p != nullptr)
    std::cout << "minimum is " << *p << std::endl;
else
    std::cout << "array is empty" << std::endl;
```

In other word, `p` is a pointer that will point to the element within the array once the minimum is found.

```
#define FIND_MIN(ptr, arr, size) do { \
    if ((size) > 0) { \
        (ptr) = &(arr)[0]; \
        for (int i = 1; i < (size); i++) { \
            if ((arr)[i] < *(ptr)) \
                (ptr) = &(arr)[i]; \
        } \
    } \
} while(0)
```

b) Given the following macro constant definition:

```
#define ATMOSPHERE \
    R(EXOSPHERE, 700., 10000.) \
    R(THERMOSPHERE, 80., 700.) \
    R(MESOSPHERE, 50., 80.) \
    R(STRATOSPHERE, 12., 50.) \
    R(TROPOSPHERE, 0., 12.)
```

Use the X macro technique to generate an enum named `Atmosphere` that uses the first argument of each X macro element as the name of each enum constant, plus two additional constants: `BELOW_GROUND` and `OUTER_SPACE`. Then, generate a function named `height_to_atmosphere` that take a height and returns the appropriate enum constant. For example, for a height of 100, the function should return `THERMOSPHERE` because it is within the range of 80km and 700km. Note that the function should return `BELOW_GROUND` if height is negative, and `OUTER_SPACE` if height is greater than 10,000.

```
enum Atmosphere {
    BELOW_GROUND,
#define R(e, ...) e,
    ATMOSPHERE
#undef R
    OUTER_SPACE,
};

Atmosphere height_to_atmosphere(double height) {
    if (height < 0) {
        return BELOW_GROUND;
    } else
#define R(e, lower, upper) \
    if (height >= lower && height < upper) { \
        return e; \
    } else

    ATMOSPHERE
#undef R
    {}

    return OUTER_SPACE;
}
```

c) Use a C macro function *instead of* C++ template to create a stack class that is equivalent to this C++ template class:

```
template<typename T>
class Stack {
public:
    bool push(const T & v); // returns true if success
    bool pop();             // returns true if success
    T * top();              // returns top element
    void print() const;     // prin the stack
};
```

Your macro function will be used like this:

```
DEFINE_STACK(IntStack, int);

// in main.cpp
IntStack is;
is.push(7);
cout << "top: " << *is.top() << endl;
```

```
// SOLUTION
#define DEFINE_STACK(CLS, TYPE) class CLS { \
   std::vector<TYPE> array; \
public: \
   CLS() {} \
   bool push(const TYPE & v) { \
     unsigned idx = array.size(); array.push_back(v); return true; \
   } \
   bool pop() { \
     if (array.size() == 0) return false; \
     array.pop_back(); return true; \
   } \
   TYPE * top() { \
     if (array.size() == 0) return nullptr; return &array.back(); \
   } \
   void print() const { \
     for (auto it = array.begin(); it != array.end(); it++) \
          cout << (*it) << " "; \
     cout << "\n"; \
   } \
}
```