CSE 333 19wi Final Exam March 20, 2019

Name _____ ID # _____

There are 7 questions worth a total of 120 points. Please budget your time so you get to all of the questions. Keep your answers brief and to the point.

The exam is closed book, closed notes, closed electronics, closed telepathy, open mind. If you don't remember the exact syntax for something, make the best attempt you can. We will make allowances when grading. Don't be alarmed if there seems to be more space than is needed for your answers – we tried to include more than enough blank space.

There are two pages of reference information following the blank page at the end. You may remove those pages. They will not be scanned or graded.

Relax, you are here to learn.

Please wait to turn the page until everyone is told to begin

Score _____ / 120

1. _____ / 16
2. _____ / 18
3. _____ / 24
4. _____ / 20
5. _____ / 20
6. _____ / 20
7. _____ / 2

Note: Please write your answers only on the specified pages. Reference pages and pages with only questions and explanations will not be scanned for grading, and you should remove them from the exam.

There is an extra blank page after the last question at the end of the exam if you need additional space for one or more answers. That page will be graded if it contains answers.

Question 1. (16 points) C++ STL. We have a C++ map that contains information about available flights between airports. Each key in the map is a string with a 3-letter airport code. The corresponding value is a vector of strings containing the codes of all other airports reachable directly from this airport. We will assume that if airport A appears in the destinations list for airport B then B will also appear in the destinations list for A. For example, the following map includes information that there is a flight from SEA to SFO and vice versa, but not from SEA to MSP or vice versa.

```
map<string, vector<string>> flights =
  { {"SEA", {"SFO", "JFK"}},
    {"JFK", {"SEA", "MSP", "SFO"}},
    {"MSP", {"JFK"}},
    {"SFO", {"SEA", "JFK"}}
  };
```

Complete the definition of function `path_exists` below. The arguments to `path_exists` are a flights table like the one above, and a vector of strings with a sequence of airport codes. The function should return true if it is possible to get from the first airport in the list to the last by taking direct flights between each pair of adjacent airports in the list. For example (abusing notation a little to write a vector value using curly braces), `path_exists(flights, {"SEA", "SFO", "JFK", "MSP"})` should return `true` since there are flights from SEA to SFO, from SFO to JFK, and from JFK to MSP, while `path_exists(flights, {"SEA", "MSP"})` should return false. You should assume there are at least two airport codes in the route list.

```
bool path_exists(map<string, vector<string>> flights,
                 vector<string> route) {




}
```

(more space on the next page if needed)

Question 1. (cont.) Additional space for your answer if needed. Do not remove this page.

Question 2. (18 points) A somewhat(?) straightforward exercise involving classes. Consider the following program, which, as is customary, does compile and execute without errors. (Headers and `using namespace std;` omitted)

```
// point on 2-d plane
class Point {
 public:
  Point(): x_(0.0), y_(0.0) { cout << "Point() ctr" << endl; }
  Point(double x, double y): x_(x), y_(y)
                             { cout << "Point(x,y) ctr" << endl; }
  Point(const Point &other): x_(other.x_), y_(other.y_)
                             { cout << "Point copy ctr" << endl; }
  ~Point()
     { cout << "~Point(" << x_ << "," << y_ << ") dtr" << endl; }
  Point &operator=(const Point &other) {
    cout << "Point op=" << endl;
    if (this != &other) {
      x_ = other.x_; y_ = other.y_;
    }
    return *this;
  }
 private:
  double x_, y_;
};

// rectangle defined by upper-left and lower-right corners
class Rectangle {
 public:
  Rectangle()        { cout << "Rectangle() ctr" << endl; }
  Rectangle(const Point &ul, const Point &lr): ul_(ul), lr_(lr)
                     { cout << "Rectangle(ul,lr) ctr" << endl; }
  Rectangle(const Rectangle &other)
                                 : ul_(other.ul_), lr_(other.lr_)
                     { cout << "Rectangle copy ctr" << endl; }
  ~Rectangle() { cout << "~Rectangle dtr" << endl; }
  Rectangle &operator=(const Rectangle &other) {
    cout << "Rectangle op=" << endl;
    if (this != &other) {
      ul_ = other.ul_; lr_ = other.lr_;
    }
    return *this;
  }
 private:
  Point ul_, lr_;  // upper-left and lower-right corners
};
```

Remove this page from the exam, then continue with the problem on the next pages.
Do not write anything on this page. It will not be scanned for grading.

Question 2. (cont.) Now, here is a main program that uses these classes. At the bottom of the page, write the output produced when this program is executed.

```
int main() {
  Point p1;
  Point p2(1,4);
  cout << "---(1)---" << endl;
  Point p3 = p2;
  cout << "---(2)---" << endl;
  Rectangle r1;
  cout << "---(3)---" << endl;
  Point p4(2,3);
  Rectangle r2(p2,p4);
  cout << "---(4)---" << endl;
  r1 = r2;
  cout << "---(5)---" << endl;
  return 0;
}
```

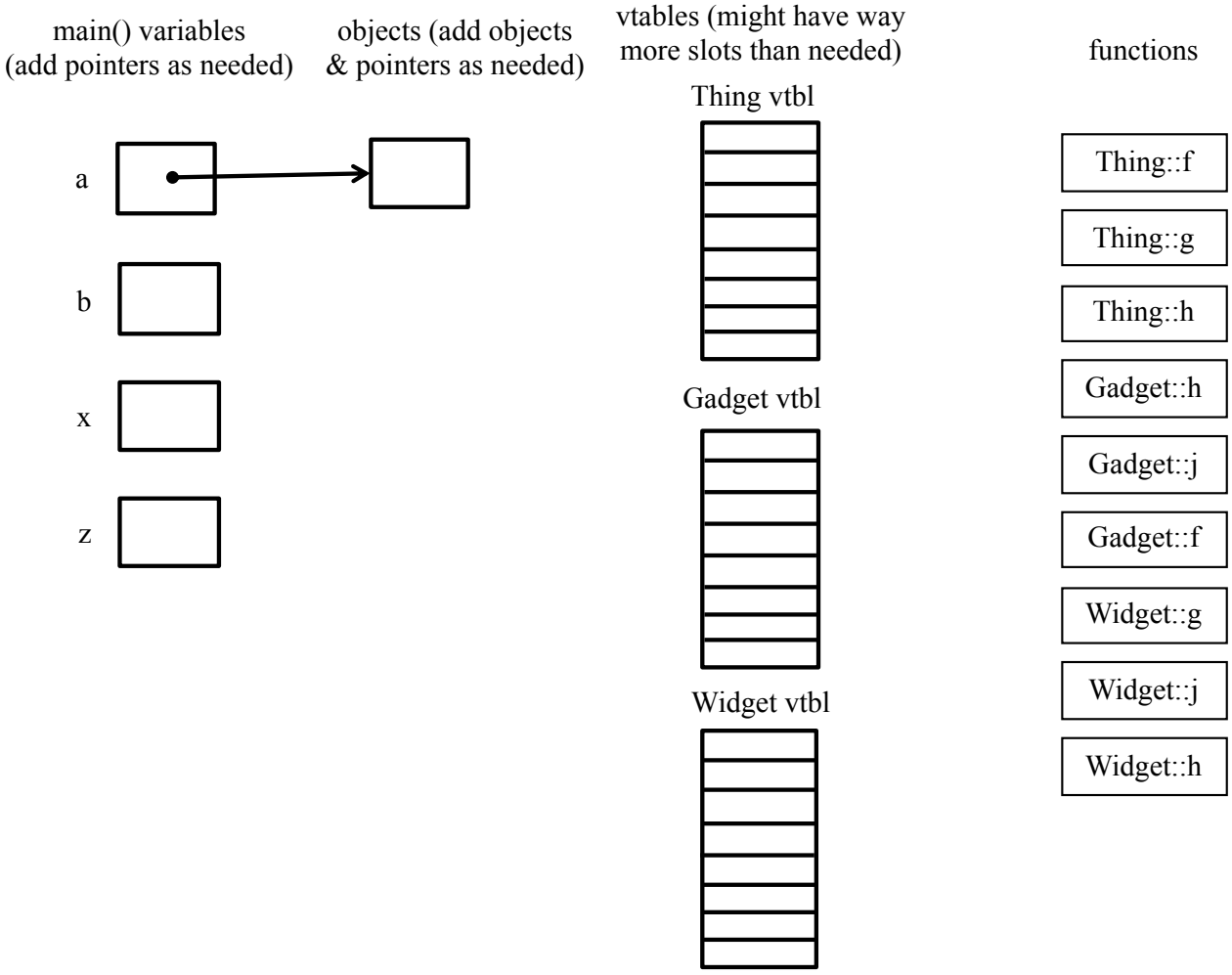
Output:

Question 3. (24 points) Here we have another of those bizzare programs that seem to keep appearing on these exams with subclasses and virtual functions. As usual it does compile and execute with no errors.

```
class Thing {
public:
  virtual void f() {      cout << "Thing::f" << endl; }
          void g() {      cout << "Thing::g" << endl; }
          void h() { g(); cout << "Thing::h" << endl; }
};

class Gadget: public Thing {
public:
  virtual void h() {      cout << "Gadget::h" << endl; }
          void j() { g(); cout << "Gadget::j" << endl; }
          void f() { h(); cout << "Gadget::f" << endl; }
};

class Widget: public Gadget {
public:
          void g() {      cout << "Widget::g" << endl; }
  virtual void j() { f(); cout << "Widget::j" << endl; }
          void h() {      cout << "Widget::h" << endl; }
};

int main() {
  Thing *a = new Gadget();
  a->f();
  a->g();
  a->h();
  cout << "---(1)---" << endl;
  Thing *b = new Widget();
  b->f();
  b->g();
  b->h();
  cout << "---(2)---" << endl;
  Gadget *x = new Widget();
  x->f();
  x->g();
  x->h();
  x->j();
  cout << "---(3)---" << endl;
  Widget *z = new Widget();
  z->f();
  z->g();
  z->h();
  z->j();
  return 0;
}
```

Remove this page from the exam, then continue with the problem on the next pages.
Do not write anything on this page. It will not be scanned for grading.

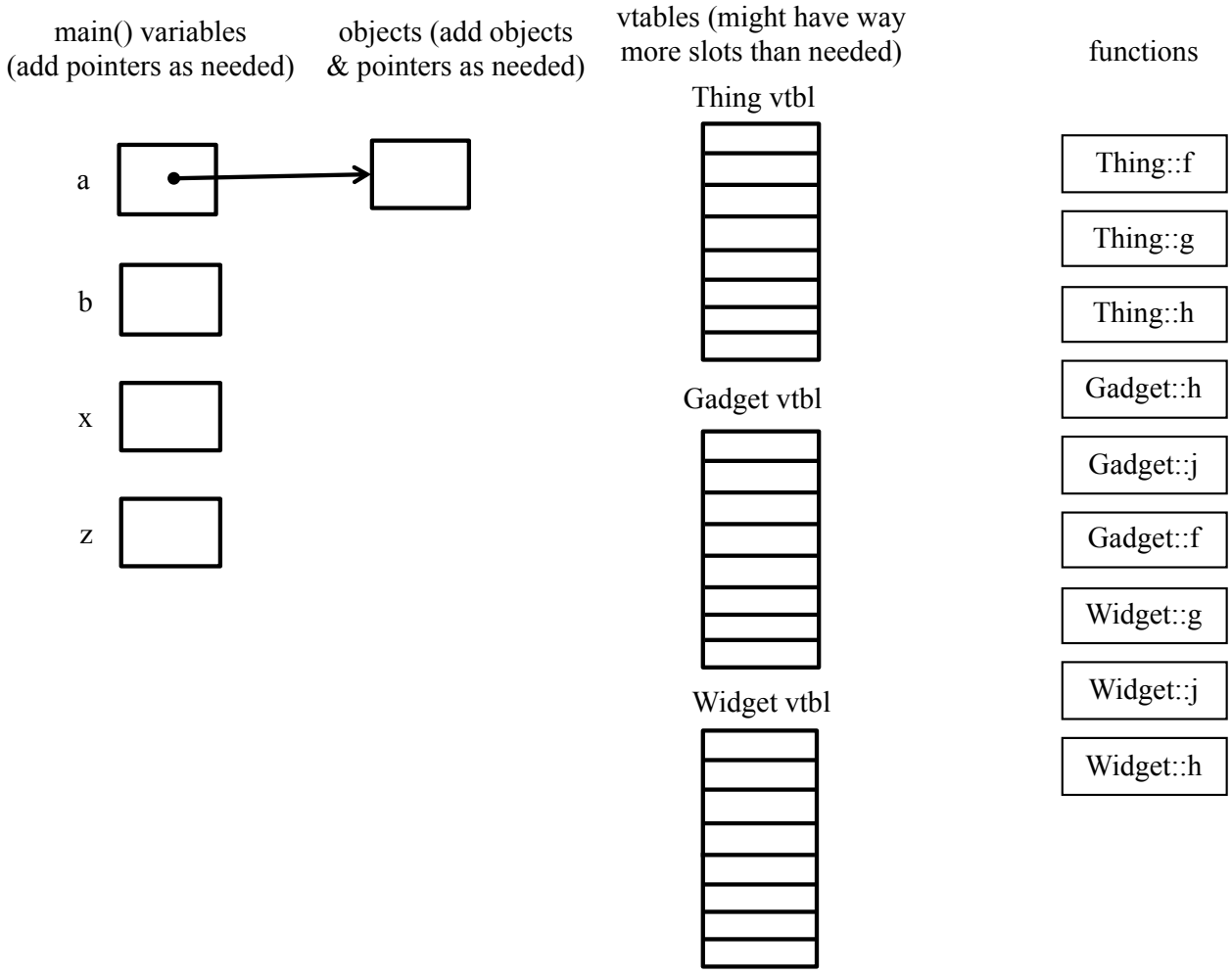
Question 3. (cont.) (a) (6 points) Complete the diagram below to show all of the variables, objects, virtual method tables (vtables) and functions in this program. Parts of the diagram are supplied for you.



(b) (6 points) What does this program print when it executes? (use multiple columns if needed)

(continued on the next page)

Question 3. (cont.) (c) (6 points) Now suppose we take the original program and place the word "virtual" in front of every function definition in every class. Complete the diagram below to show all of the variables, objects, virtual method tables (vtables) and functions in this altered program.



(d) (6 points) What output does this altered program produce when it is executed? Again, use multiple columns if you need more room.

Question 4. (20 points, 4 each) Smart pointers. Below we have several small programs, each of which calls a function `foo`, and each of which uses smart pointers. Some of them are buggy, and some of them work correctly. Your job is to determine, for each program, the following:

(1) Does it compile?
(2) Is its behavior correct (i.e. no memory leaks, run-time errors, or undefined behavior)? Choose n/a if it did not compile.
(3) If you answered "no" to either of the above, explain concisely what the problem is and how to fix it.

You should assume that none of the `foo` functions will attempt to free, delete, or otherwise modify their argument. You can assume that the actual value returned by `main` is not relevant.

(a)
```
int foo(int *n);  // defined elsewhere

int main() {
  std::unique_ptr<int> best_course(new int(333));
  int ans = foo(best_course);
  return ans;
}
```

Does it compile? (circle) yes no not sure

Does it execute correctly (circle) yes no n/a (didn't compile)

What is the problem (if any) and how do we fix it? (leave blank if not applicable)

(b)
```
int foo(std::shared_ptr<int> p);

int main() {
  std::unique_ptr<int> best_course(new int(333));
  int ans = foo(std::shared_ptr<int>(best_course.release()));
  return ans;
}
```

Does it compile? (circle) yes no not sure

Does it execute correctly (circle) yes no n/a (didn't compile)

What is the problem (if any) and how do we fix it? (leave blank if not applicable)

(continued next page)

Question 4. (cont.)

(c)
```
int foo(int, std::shared_ptr<int> p);

int main() {
  std::shared_ptr<int> best_course(new int[10]);
  int ans = foo(10, best_course);
  return ans;
}
```

Does it compile? (circle) yes no not sure

Does it execute correctly (circle) yes no n/a (didn't compile)

What is the problem (if any) and how do we fix it? (leave blank if not applicable)

(d)
```
int foo(std::unique_ptr<int> n);

int main() {
  std::unique_ptr<int> best_course(new int(333));
  int ans = foo(best_course);
  return ans;
}
```

Does it compile? (circle) yes no not sure

Does it execute correctly (circle) yes no n/a (didn't compile)

What is the problem (if any) and how do we fix it? (leave blank if not applicable)

(e)
```
int foo(const int *p);

int main() {
  std::unique_ptr<int> best_course(new int(333));
  int ans = foo(best_course.release());
  return ans;
}
```

Does it compile? (circle) yes no not sure

Does it execute correctly (circle) yes no n/a (didn't compile)

What is the problem (if any) and how do we fix it? (leave blank if not applicable)

Question 5. (20 points) Networking – an insecure password server. We've been working on the prototype of a network authentication server. We're not ready to do any of that fancy encryption or security stuff yet – first we need to be able to read plain text usernames and passwords over the network and validate them.

The main part of the server code that accepts a connection from the client and closes the socket when done has already been implemented. The missing part is the logic to handle the client request. For this problem, implement the function `handle_client(cfd)` (on the next page), which reads a request from the client file descriptor, looks up the user name in a C++ map, and checks to see if the password in the network request matches the one found in the username / password data structure.

Specifically, your code needs to perform the following steps:

1. Read a request from the client socket. The request will have a maximum length of `REQ_MAX_LEN` bytes, although it might be less. You can allocate a buffer of this size for the input and not worry about excessive amounts of input. You *do* need to handle `EINTR` and `EAGAIN` results from `read` as usual.
2. The client request will have the form "*username*`\r\n`*password*`\r\n\r\n`" with no leading blanks. You need to parse the request and extract the *username* and *password* strings from it.
3. Check the password map to determine if the username and password match properly (i.e. the password is valid for that username). This is simple string matching.
4. Send a response back to the user. It should be either "`OK\r\n\r\n`" if the username and password match, or "`DENIED\r\n\r\n`" if the username is known but the password does not match the username, or, if the userame is not found, the response should be "`BADUSER\r\n\r\n`". Since you are using `write` to send the data to the client, you also need to worry about `EINTR` and `EAGAIN` results and retry until all of the data is written.

The username/password data is stored in a global variable named `passwd` that has the following declaration:

```
extern std::map<std::string, std::string> passwd;
```

The map keys are usernames and the associated values are the passwords. You should use this variable in your code without needing to declare it further.

Remove this page from the exam, then write your answer on the next page. **Do not write anything on this page.** It will not be scanned for grading.

Reminder: there are several possibly useful reference pages at the end of the exam.

(continued next page)

Question 5. (cont.) Give an implementation of the `handle_client` function below. The parameter `cfd` is the client socket file descriptor returned from the server's accept function, and this socket should be used to communicate with the client. Assume all necessary header files are already `#include`d.

```
static const int REQ_MAX_LEN = 255; // max input request length
extern std::map<std::string, std::string> passwd;  // passwords

void handle_client(int cfd) {
    // Write your solution below




}
```

(more space on the next page if needed)

Question 5. (cont.) Additional space for your answer if needed. Do not remove this page.

Question 6. (20 points) Threads. Consider the following simple C++ program, which creates three threads. Each thread increments two global variables, then prints their values. After all three threads finish, the final values of the global variables are printed.

```
#include <stdio.h>
#include <pthread.h>

int x = 0;
int y = 0;

void * worker(void * ignore) {
  x = x + 1;
  y = y + x;
  printf("x = %d, y = %d\n", x, y);
  return NULL;
}
int main() {
  pthread_t t1, t2, t3;
  int ignore;
  ignore = pthread_create(&t1, NULL, &worker, NULL);
  ignore = pthread_create(&t2, NULL, &worker, NULL);
  ignore = pthread_create(&t3, NULL, &worker, NULL);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  pthread_join(t3, NULL);
  printf("final x = %d, y = %d\n", x, y);
  return 0;
}
```

(a) (10 points) What output is produced by this program when it executes? If it is possible to get different output each time the program executes, write down two possible outputs. If the program always produces the same output, write that output here and indicate it is the only possible one. (Assume that `printf` executes atomically so that printing of output lines will not produce garbled output.)

(b) (10 points) Circle *all* of the possible values that variables `x` and `y` could have at the end of any possible executions of this program, as printed in the final line of output:

x:	0	1	2	3	4	5	6	7	8	9	≥10
y:	0	1	2	3	4	5	6	7	8	9	≥10

Question 7. (2 free points – all answers get the free points) Draw a picture of something you plan to do over spring break!

Congratulations on lots of great work this quarter!!
Have a great spring break and say hello when you get back!
The CSE 333 staff

Extra space for answers, if needed. Please be sure to label which question(s) are answered here, and be sure to put a note on the question page so the grader will know to look here.

Reference information. Here is a collection of information that might, or might not, be useful while taking the test. You can remove this page from the exam if you wish.

C++ strings: If `s` is a string, `s.length()` and `s.size()` return the number of characters in it. `s.find(`*search_string, start_pos* = 0`)` returns the location of the first occurrence of *search_string* starting no earlier than *start_pos* or returns `string::npos` if not found. Subscripts (`s[i]`) can be used to access individual characters.

C++ STL:

- If `lst` is a STL `vector`, then `lst.begin()` and `lst.end()` return iterator values of type `vector<...>::iterator`. STL `lists` and `sets` are similar.
- A STL `map` is a collection of `pair` objects. If p is a `pair`, then `p.first` and `p.second` denote its two components. If the `pair` is stored in a `map`, then `p.first` is the key and `p.second` is the associated value.
- If `m` is a `map`, `m.begin()` and `m.end()` return iterator values. For a `map`, these iterators refer to the `Pair` objects in the `map`.
- If `it` is an iterator, then `*it` can be used to reference the item it currently points to, and `++it` will advance `it` to the next item, if any.
- Some useful operations on STL containers (lists, maps, sets, etc.):
  - `c.clear()` – remove all elements from c
  - `c.size()` – return number of elements in c
  - `c.empty()` – true if number of elements in c is 0, otherwise false
- Additional operations on `vectors`:
  - `c.push_back(x)` – copy x to end of c
- Some additional operations on `maps`:
  - `m.insert(x)` – add copy of `x` to `m` (a key-value pair for a `map`)
  - `m.count(x)` – number of elements with key `x` in `m` (0 or 1)
  - `m.find`(*item*) – iterator pointing to element with key that matches *item* if found, or `m.end()` if not found.
  - `m[k]` can be used to access the value associated with key `k`. If `m[k]` is read and has never been accessed before, then a <key,value> `Pair` is added to the map with `k` as the key and with a value created by the default constructor for the value type (0 or `nullptr` for primitive types).
- Some additional operations on `sets`
  - `s.insert(x)` – add `x` to `s` if not already present
  - `s.count(x)` – number of copies of `x` in `s` (0 or 1)
- You may use the C++11 `auto` keyword, C++11-style `for`-loops for iterating through containers, and any other features of standard C++11, but you are not required to do so.

More reference information. You can also remove this page if you wish.

Some POSIX I/O and TCP/IP functions:

- int **accept**(int sockfd, struct socckaddr *addr, socklen_t *addrlen);
- int **bind**(int sockfd, const struct sockaddr *addr, socklen_t addrlen)
- int **close**(int fd)
- int **connect**(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
- int **freeaddrinfo**(struct addrinfo *res)
- int **getaddrinfo**(const char *hostname, const char *service,
  const struct addrinfo *hints, struct addrinfo **res)
  - Use NULL or listening port number for second argument
- int **listen**(int sockfd, int backlog)
  - Use SOMAXCONN for backlog
- off_t **lseek**(int fd, off_t offset, int whence)
  - whence is one of SEEK_SET, SEEK_CUR, SEEK_END
- ssize_t **read**(int fd, void *buf, size_t count)
  - if result is -1, errno could contain EINTR, EAGAIN, or other codes
- int **socket**(int domain, int type, int protocol)
  - Use SOCK_STREAM for type (TCP), 0 for protocol, get domain from address info struct (address info struct didn't fit on this page – we'll include it later if needed)
- ssize_t **write**(int fd, const void *buf, size_t count)

Some pthread functions:

- pthread_create(thread, attr, start_routine, arg)
- pthread_exit(status)
- pthread_join(thread, value_ptr)
- pthread_cancel (thread)
- pthread_mutex_init(pthread_mutex_t * mutex, attr) // attr=NULL usually
- pthread_mutex_lock(pthread_mutex_t * mutex)
- pthread_mutex_unlock(pthread_mutex_t * mutex)
- pthread_mutex_destroy(pthread_mutex_t * mutex)

Basic C memory management functions:

- void * **malloc**(size_t size)
- void **free**(void *ptr)
- void * **calloc**(size_t number, size_t size)
- void * **realloc**(void *ptr, size_t size)