Common Misunderstandings from Exam I Material

Kyle Dewey
Stack and Heap Allocation with Pointers
char c = 'c';
char* p1 = malloc(sizeof(char));
char** p2 = &p1;

• Where is \texttt{c} allocated?

• Where is \texttt{p1} itself allocated?

• Where is what \texttt{p1} points to allocated?

• Where is \texttt{p2} itself allocated?

• Where is what \texttt{p2} points to allocated?

• Nearly everyone got this one wrong
Key to Solving

- Draw out a memory diagram
- The following slides go through this process
char c = ...;

- c is a local variable, and local variables are allocated on the stack
- The value is initially undefined
char c = 'c';

- Any time = is used, it assigns to that place directly, so since c is on the stack, 'c' gets put into that place on the stack
char c = 'c';
char* p1 = ...;

- `p1` is a local variable, and local variables are allocated on the stack
- The value is initially undefined
```c
char c = 'c';
... = malloc(sizeof(char));
```

- *malloc* allocates something on the heap
- The value is initially undefined
char c = 'c';
char* p1 = malloc(sizeof(char));

- = puts something in place directly

- This means `p1` holds a pointer to the space allocated on the heap
char c = ‘c’;
char* p1 = malloc(sizeof(char));
char** p2 = ...;

- $p2$ is a local variable, and local variables are allocated on the stack
- The value is initially undefined
char c = 'c';
char* p1 = malloc(sizeof(char));
char** p2 = &p1;

- puts something in place directly
- The & operator creates a pointer to p1
• Where is c allocated?
• Where is p1 itself allocated?
• Where is what p1 points to allocated?
• Where is p2 itself allocated?
• Where is what p2 points to allocated?

```
Heap

Stack

sizeof(char) ???

'c'

p1 addr

p2 addr
```
main's return value
main's Return Value

```c
int main() {
    return 0;
}
```

- What is returned is a code to the operating system
- It is **not** part of the output
- By convention, 0 means "everything ok", and non-zero is an error code of some sort
When Destructors are Called
Destructor Call

• The destructor for an object is called automatically right before the object is deallocated

• Which two ways can memory be deallocated? (Hint: which two ways can we allocate memory?)
Destructor Call

- The destructor for an object is called **automatically** right before the object is deallocated.

- Which two ways can memory be deallocated?
  - Stack: function return
  - Heap: delete
void test() {
    Des d(1);
    ...; // some other code
}

int main() {
    Des* p = new Des(0);
    test();
    delete p;
    return 0;
}
void test() {
    Des d(1); // allocates d on stack
    ...; // some other code
    // d is deallocated off of stack
    // right before test returns
}

int main() {
    // allocates on heap below
    Des* p = new Des(0);
    test();
    delete p; // deallocated off heap
    return 0;
}
void test() {
    Des d(1); // allocates d on stack
    ...; // some other code
    // d is deallocated off of stack
    // right before test returns
    // destructor called
}

int main() {
    // allocates on heap below
    Des* p = new Des(0);
    test();
    delete p; // deallocated off heap
    // destructor called
    return 0;
}
bool and Boolean Expressions
Booleans

- C++ has a special `bool` type, which permits values of `true` and `false`
- Something is either less than something else or isn’t: `bool` is perfect here

```cpp
bool firstLessThanSecond(int x, int y);```

Sunday, July 27, 14
Boolean Expressions

• Tests (e.g., \( x < y \)) already return \( \text{bool} \)

• There is no need to add another conditional to it

```cpp
bool firstLessThanSecond(int x, int y) {
    // \text{if} isn't needed here
    if (x < y) {
        return true;
    } else {
        return false;
    }
}
```
Boolean Expressions

• Tests (e.g., \(x < y\)) already return boolean

• There is no need to add another conditional to it

```cpp
bool firstLessThanSecond(int x, int y) {
  // if isn’t needed here
  return x < y;
}
```
public / private
public/private

- A particular class has access to all its own private members

- This includes
  - All methods
  - Constructors
  - Destructors

- Methods that take in other instances of the same class
class Square {
    public:
    // constructor
    // other methods

    bool lessThan(const Square& o) const {
        return size < o.size;
    }

    private:
    int size;
};
class Square {
    public:
        // constructor
        // other methods

        bool lessThan(const Square& o) const {
            return size < o.size;
        }

    private:
        int size;
};

Access ok: size is an instance variable of Square, and lessThan is a method on Square.
Access ok: `size` is an instance variable of `Square`, `lessThan` is a method on `Square`, and `o` is an instance of `Square`.
insertAtSecond/
removeFromSecond
insertAtSecond
removeFromSecond

• Memory diagrams are very helpful here

• Loops aren’t needed (can just grab the first, second, and third elements directly)

• No need to implement your own length method

• **Length 0:** `head == NULL`

• **Length 1:** `head != NULL && head->getNext() == NULL`
Command-line Arguments
int main(int argc, char** argv) {
    ...
    return 0;
}

• **argc** holds the number of arguments, including how the command was invoked

• **argv** holds the actual arguments
int main(int argc, char** argv) {
    ...
    return 0;
}

Command: ./a.out

argc: 1
argv: { "./a.out" }
int main(int argc, char** argv) {
    ...
    return 0;
}

Command: ./a.out foo

argc: 2
argv: { "./a.out", "foo" }
```c
int main(int argc, char** argv) {
    ...
    return 0;
}
```

**Command:** ./a.out foo bar

**argc:** 3

**argv:** { ./a.out, foo, bar }