char* x = new char;
yields a memory address
1. allocates memory for a char
2. declares a pointer to a char
3. sets pointer to memory address

int* x = new int[n];
yields a memory address
1. allocates memory for int array
2. declares a pointer to an integer
3. sets pointer to memory address
### stack vs. heap

While typical declarations allocate memory on the **stack**, using `new` allocates memory in the **heap**

<table>
<thead>
<tr>
<th></th>
<th>stack</th>
<th>heap</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>scope</strong></td>
<td>local</td>
<td>global</td>
</tr>
<tr>
<td>allocate one thing</td>
<td>int x;</td>
<td>int* px = <strong>new</strong> int;</td>
</tr>
<tr>
<td>allocate array</td>
<td>int x[5]; int* pax = &amp;x[0];</td>
<td>int* pax = <strong>new</strong> int[5];</td>
</tr>
<tr>
<td>cleanup one thing</td>
<td>do nothing :D</td>
<td><strong>delete</strong> px;</td>
</tr>
<tr>
<td>cleanup array</td>
<td>do nothing :D</td>
<td><strong>delete</strong> [] pax;</td>
</tr>
</tbody>
</table>

#### code snippet

```cpp
char s[12];
strcpy(s, "hello mississippi");
while (s != 0) {
    if (*s == *(s+1)) {
        *s++;
    }
}
cout << s;
```

```cpp
char* s = **new** char[12];
strcpy(s, "hello mississippi");
while (s != 0) {
    if (*s == *(s+1)) {
        *s++;
    }
}
cout << s;
delete s;
```

Both approaches print "helmo mististipqi"
allocating memory in functions

```cpp
void fxn2(char* s, int n) {
    char s2[] = "hey";
    if (n >= 4) {
        strcpy(s, s2);
    }
}

char* fxn1(int n) {
    char* s = &str[n];
    fxn2(s, n);
    return s;
}

int main() {
    char* word = fxn1(7);
    cout << word << endl;
    delete [] word;
}
```

please note that n should also appear on the stack in this example, but it will be omitted due to lack of foresight on my part. sorry -- Brian
allocating memory in functions

void fxn2(char* s, int n) {
    char s2[] = "hey";
    if (n >= 4) {
        strcpy(s, s2);
    }
}

char* fxn1(int n) {
    char* s = new char[n];
    fxn2(s, n);
    return s;
}

int main() {
    char* word = fxn1(7);
    cout << word << endl;
    delete [] word;
}
allocating memory in functions

**Dynamic Allocation**

```c
void fxn2(char* s, int n) {
    char s2[] = "hey";
    if (n >= 4) {
        strncpy(s, s2);
    }
}

char* fxn1(int n) {
    char* s = new char[n];
    fxn2(s, n);
    return s;
}

int main() {
    char* word = fxn1(7);
    cout << word << endl;
    delete [] word;
}
```

**Local Allocation**

```c
void fxn2(char* s, int n) {
    char s2[] = "hey";
    if (n >= 4) {
        strncpy(s, s2);
    }
}

char* fxn1(int n) {
    char* s = new char[n];
    fxn2(s, n);
    return s;
}

int main() {
    char* word = fxn1(7);
    cout << word << endl;
    delete [] word;
}
```
allocating memory in functions

dynamic allocation

```c
void fxn2(char* s, int n) {
    char s2[] = "hey";
    if (n >= 4) {
        strcpy(s, s2);
    }
}
char* fxn1(int n) {
    char* s = new char[n];
    fxn2(s, n);
    return s;
}
int main() {
    char* word = fxn1(7);
    cout << word << endl;
    delete [] word;
}
```

local allocation

```c
void fxn2(char* s, int n) {
    char s2[] = "hey";
    if (n >= 4) {
        strcpy(s, s2);
    }
}
char* fxn1(int n) {
    char* s = new char[n];
    char str[n];
    char* s = &str[0];
    fxn2(s, n);
    return s;
}
int main() {
    char* word = fxn1(7);
    cout << word << endl;
    delete [] word;
}
```
allocating memory in functions

dynamic allocation

```c
void fxn2(char* s, int n) {
    char s2[] = "hey";
    if (n >= 4) {
        strcpy(s, s2);
    }
}
char* fxn1(int n) {
    char* s = new char[n];
    fxn2(s, n);
    return s;
}
int main() {
    char* word = fxn1(7);
    cout << word << endl;
    delete [] word;
}
```

local allocation

```c
void fxn2(char* s, int n) {
    char s2[] = "hey";
    if (n >= 4) {
        strcpy(s, s2);
    }
}
char* fxn1(int n) {
    char str[n];
    char* s = &str[0];
    fxn2(s, n);
    return s;
}
int main() {
    char* word = fxn1(7);
    cout << word << endl;
}
```
memory management practice (new/delete)

**Stack** memory is automatic, **heap** memory is not managed by the operating system.

**Memory leak** occurs when memory allocated is not de-allocated before the program completes.

---

**Code**

```
int *s1, *s2;
int s3[4];
s1 = new int[4];
s2 = new int[6];
delete [] s1;
```

**Compiles/Runs?**

```
yep
```

```
int *s1, *s2;
int s3[4];
s1 = new int[4];
s2 = &s3[0];
delete [] s1;
delete [] s2;
```

**Leak?**

```
nope
```

```
int *s1, *s2;
int s3[4];
s1 = new int[4];
s2 = &s3[0];
delete [] s1;
```

**Compiles/Runs?**

```
yep
```

```
int *s1, *s2;
int s3[4];
s1 = new int[4];
s2 = new int[6];
s1 = s2;
delete [] s1;
```

**Leak?**

```
yep
```

```
int *s1, *s2;
int s3[4];
s1 = new int[4];
s2 = new int[6];
delete [] s1;
s1 = s2;
delete [] s2;
```

**Leak?**

```
yep, 24 bytes
```

```
int *s1, *s2;
int s3[4];
s1 = new int[4];
s2 = new int[6];
delete [] s1;
```

```
yep, 16 bytes
```

**Tip:** “for every new, there should be an associated delete”
A bit more about delete

Q: Can multiple objects be deleted on the same line?  nope

each call to delete must be on its own line, so given pointers a, b, c (e.g. “int *a, *b, *c;”):

**BREAKS**

delete a, b, c;

**RUNS**

delete a;
delete b;
delete c;

Q: How does delete know how much memory to give back to the operating system?

“When you allocate memory on the heap, your allocator will keep track of how much memory you have allocated. This is usually stored in a "head" segment just before the memory that you get allocated. That way when it's time to free the memory, the de-allocator knows exactly how much memory to free.”


Q: Let's say we’re given “int* A = new int[10];” and “int* B = new int;”. What happens if you do “delete [ ] B;”? What if you do “delete A;”

how much is it leaking? (problem code)

```cpp
int sum(int* x, int n) {
    int* part_sum = new int[n];
    part_sum[0] = x[0];
    for (int k = 1; k < n; k++) {
        part_sum[k] = part_sum[k-1] + x[k];
    }
    return part_sum[n - 1];
}
```

- 4 * n bytes
- fix: add “delete [ ] part_sum;”

```cpp
int* pow(int* x, int p, int n) {
    int* dst = new int[n];
    int* src = x;
    for (int j = 1; j < p; j++) {
        for (int k = 0; k < n; k++) {
            dst[k] = src[k] * x[k];
        }
        src = dst;
        dst = new int[n];
    }
    return src;
}
```

- 4 * p * n bytes
- fix: add “delete [ ] src;” before “src = dst;”

```cpp
bool lift(int dst, int now) {
    int* start = new int;
    *start = now;
    if (*start == dst) {
        return true;
    } else {
        while (*start < dst) {
            *start++;
        }
        delete start;
        return true;
    }
}
```

- 4 bytes
- fix: add “delete start;” in the first if cond
how much is it leaking? (fixed code)

```c
int sum(int* x, int n) {
    int* part_sum = new int[n];
    part_sum[0] = x[0];
    for (int k = 1; k < n; k++) {
        part_sum[k] = part_sum[k-1] + x[k];
    }
    int total = part_sum[n - 1];
    delete [] part_sum;
    return total;
}
```

- 4 * n bytes
  - fix: add “delete [] part_sum;”

```c
int pow(int* x, int p, int n) {
    int* dst = new int[n];
    int* src = new int[n];
    for (int k = 0; k < n; k++) {
        src[k] = x[k];
    }
    for (int j = 1; j < p; j++) {
        for (int k = 0; k < n; k++) {
            dst[k] = src[k] * x[k];
        }
        delete [] src;
        src = dst;
        dst = new int[n];
    }
    delete [] dst;
    return src;
}
```

- 4 * p * n bytes
  - fix: add “delete [] src;” before “src = dst;”
  - p must be larger than 2

```c
int* lift(int dst, int now) {
    int* start = new int;
    *start = now;
    if (*start == dst) {
        delete start;
        return true;
    } else {
        while (*start < dst) {
            *start++;
        }
        delete start;
        return true;
    }
}
```

- 4 bytes
  - fix: add “delete start;” in the first if cond
structures

**struct** is a group of data grouped under a single name. The data does not have to be of the same type.

```c
struct student {
    int sid;
    char* name;
    int year;
    string field;
    int age;
    double wage;
    double gpa;
};

struct prof {
    bool tenure;
    string name;
    string field;
    double wage;
};
```

```c
int main() {
    student brian;
    prof david;

    // set some things
    brian.sid = 123456789;
    david.name = "David";

    // see some things
    cout << brian.name << endl;
    cout << david.field << endl;
    
    if (david.tenure) {
        david.wage++;
    }
}
```

*bad call, segfaults since name isn’t set*
**classes**

**class** is pretty much the same as a struct, but slightly more private

```cpp
class student {
public:
    int sid;
    char* name;
    int year;
    string field;
    int age;
    double wage;
    double gpa;
};

class prof {
public:
    bool tenure;
    string name;
    string field;
    double wage;
};

int main() {
    student brian;
    prof david;

    // set some things
    brian.sid = 123456789;
    david.name = "David";

    // see some things
    cout << brian.name << endl;
    cout << david.field << endl;
    if (david.tenure) {
        david.wage++;
    }
}
```

In a struct, all functions and variables default to public access. In a class, they default to private.
**member functions** are functions specific to a class (e.g. get and set functions)

```cpp
class prof {
    private:
        bool tenure;
        string name;
        string field;
        double wage;

    public:
        void talk() {
            cout << "Did you read ";
            cout << "the spec?";
        }

        // accessor fxns
        string get_name() {
            return name;
        }

        void set_name(string s) {
            name = s;
        }

        // accessor fxns
        ...
};

int main() {
    prof david;

    // set some things
    david.set_name("David");
    david.set_tenure(true);

    // see some things
    cout << "Prof. " << david.get_name();
    cout << " says, "" << david.talk();
    cout << " Field: " << david.get_field();

    if (david.get_tenure()) {
        david.set_wage(david.get_wage() + 1);
    }
}
```
int main() {
    prof david;

    // set some things
    david.name("David");
    david.tenure(true);

    // see some things
    cout << "Prof. " << david.name();
    cout << " says, """ << david.talk();
    cout << "/"" << endl;

    if (david.tenure()) {
        david.wage(david.wage() + 1);
    }

    cout << "Field: " << david.field();
}

- private variables preceded by "_"
- get/set functions overloaded with same name
Constructors are used to create instances of an object.

- If not defined, a default constructor is created.

```cpp
class prof {
private:
    bool _tenure;
    string _name;
    string _field;
    double _wage;

public:
    prof() { //default
        _tenure = false;
        _name = "Prof";
        _field = "CS";
        _wage = 200000;
    }

    prof(bool t, string n, string f, double w) {
        _tenure = t;
        _name = n;
        _field = f;
        _wage = w;
    }
    ...
};

int main() {
    prof bill;
    prof david(true, "David", "CS", 250000);

    // set some things
    bill.name("Bill");
    bill.tenure(true);

    // see some things
    cout << "Prof. " << bill.name();
    cout << "Prof. " << david.name();

    if (david.tenure()) {
        david.wage(david.wage() + 1);
    }
}
```
**destructors**

The **destructor** is called when an object is deleted and the memory is returned to the operating system.

```cpp
class prof {
private:
    bool _tenure; char* _name;
    string _field; double _wage;

public:
    ~prof() { //destruct
        delete [] _name;
    }

    prof(bool t, string n,
         string f, double w) { //construct
        _tenure = t;
        *_name = new char[f.size()];
        strcpy(_name, f.c_str());
        _field = f;
        _wage = w;
    }

    // set some things
    bill.name(“Bill”);
    bill.tenure(true);

    // see some things
    cout << “Prof. “ << bill.name();
    cout << “Prof. “ << david.name();
    if (david.tenure()) {
        david.wage(david.wage() + 1);
    }

    delete david;
    delete bill;
};
```

- Destructor only needs to remove un-managed memory on the heap.

---

- The destructor is called when an object is deleted and the memory is returned to the operating system.
- The destructor only needs to remove un-managed memory on the heap.
- For this to work, the default constructor (no arguments) must initialize name to a char array.
class dis {
private:
    student** _students; \[array of student pointers\]
    char* _name;
    int _nstudents;

public:
    dis(string name, int n);
    ~dis();

    mean_gpa();
};

dis::dis(string name, int n) {
    _name = new char[name.size()];
    strcpy(_name, name.c_str());
    _nstudents = n;
    // create students
    _students = new student*[n];
    for (int i = 0; i < n; i++) {
        _students[i] = new student;
    }
}

dis::~dis() {
    delete [] _name;
    for (int i = 0; i < _nstudents; i++) {
        delete _students[i];
    }
    delete [] _students;
}

double dis::mean_gpa() {
    double sum_gpa = 0.0;
    for (int i = 0; i < _nstudents; i++) {
        // sum_gpa += (*_students[i]).gpa();
        sum_gpa += _students[i]->gpa();
    }
    return sum_gpa / _nstudents;
}

int main() {
    dis* 1G = new dis("1G", 10);
    dis* 1A = new dis("1A", 8);
    if (1G.mean_gpa() < 1A.mean_gpa()) {
        cout << "Brian gets fired" << endl;
    } else {
        cout << "Liuli gets fired" << endl;
    }
    delete 1G;
    delete 1A;
}
project 7 tips

- check out the warmup online. It's really good practice and short and easy to do. Most of the code is already written for you so you don't have to write much to have something up and running, which can be a daunting task when starting out with classes and objects and such things

- download the example and play it (yay games)

- do the 5 experiments