dynamically allocated memory

char* x = new char;

1. allocates memory for a char
2. declares a pointer to a char
3. sets pointer to memory address

int* x = new int[n];

1. allocates memory for int array
2. declares a pointer to an integer
3. sets pointer to memory address
while typical declarations allocate memory on the **stack**, using **new** allocates memory in the **heap**

<table>
<thead>
<tr>
<th></th>
<th><strong>stack</strong></th>
<th><strong>heap</strong></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>use</td>
<td>pax[3] = 12; or *(pax+3) = 12</td>
<td>pax[3] = 12; or *(pax+3) = 12</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

**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;
}
```

---

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

**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;
}
```
### 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];
    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;
}
```
allocating memory in functions

**dynamic allocation**

```cpp
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**

```cpp
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;
}
```

---

- stack
- heap

- prints "hey"
- prints random garbage

- `char* word`
- `char* word`
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 Examples

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

### Compiles/Runs?

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

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

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

### Leaks?

- `int *s1, *s2;
int s3[4];
s1 = new int[4];
s2 = new int[6];
delete [] s1;
delete [] s2;` yep, 24 bytes

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

- `int *s1, *s2;
int s3[4];
s1 = new int[4];
s2 = new int[6];
s1 = s2;
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; ”):

<table>
<thead>
<tr>
<th>BREAKS</th>
<th>RUNS</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete a, b, c;</td>
<td>delete a;  delete b;  delete c;</td>
</tr>
</tbody>
</table>

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)

```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];
    }
    return part_sum[n - 1];
}
```

4 * n bytes

**fix:** add “delete [] part_sum;”

```c
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;”

```c
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
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;
}

bool 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;
    }
}

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

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

p must be larger than 2

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

- public: anything with access to the student object can access the public variables
- private: only student objects can access the private variables

restrictive access is good programming practice to ensure the integrity of your objects (i.e. prevent random objects from screwing up your object’s state or data (e.g. changing field to “the crow flies at midnight”))
constructors

constructors are used to ensure objects are valid instances of a class (i.e. that all private member variables are initialized to sensible starting values)

```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;
    }

    // set some things
    void name(string name) { _name = name; }
    void tenure(bool tenure) { _tenure = tenure; }
    void wage(double wage) { _wage = wage; }
    bool tenure() const { return _tenure; }
    double wage() const { return _wage; }
    string name() const { return _name; }
    string field() const { return _field; }
};

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);
    }
}
```

- if no constructors are defined, a default constructor is automatically created for you
- for some systems, if you define another constructor, you have to define a default as well
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();
    // for this to work, the default constructor (no arguments) must initialize name to a char array
    if (david.tenure()) {
        david.wage(david.wage() + 1);
    }

    delete david;
    delete bill;
};
```

- destructors only need to remove **un-managed** memory on the heap.
  (notice that we did not have to delete _field, because it’s a string object declared in-place with the object)
## Arrays of Objects (and Object Pointers)

**How Many Different Ways Can I Get an Array of 10 Objects of Class Cat?** *(Assume a Cat Object Takes 50 Bytes)*

<table>
<thead>
<tr>
<th>Declare</th>
<th>cat kittens[10];</th>
<th>cat* kittens[10];</th>
<th>cat* kittens = new cats[10];</th>
<th>cat** kittens = new cats*[10];</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explain</td>
<td>creates <strong>array of 10 cats</strong> on the stack</td>
<td>creates <strong>array of 10 cat pointers</strong> on the stack</td>
<td>allocates a array on the heap to store 10 cat objects</td>
<td>allocates an array on the heap to store 10 cat pointers</td>
</tr>
<tr>
<td>Stack</td>
<td>500 bytes for 10 cats (contiguous in memory)</td>
<td>40 bytes for 10 cat pointers +500? (if cats on the stack)</td>
<td>4 bytes for kittens pointer</td>
<td>4 bytes for kittens pointer +500? (if cats on the stack)</td>
</tr>
<tr>
<td>Heap</td>
<td>None</td>
<td>+500? (if cats on the heap)</td>
<td>500 bytes for the cat objects (contiguous in memory) +500? (if cats on the heap)</td>
<td></td>
</tr>
<tr>
<td>Allocate</td>
<td>Already done (used default constructor)</td>
<td>for (int k=0; k&lt;10; k++) kittens[k] = new cat(X); <em>(X is input arguments)</em></td>
<td>Already done (used default constructor)</td>
<td>for (int k=0; k&lt;10; k++) kittens[k] = new cat(X); <em>(X is input arguments)</em></td>
</tr>
<tr>
<td>Cats Dynamically</td>
<td></td>
<td>for (int k=0; k&lt;10; k++) kittens[k] = new cat(X); <em>(X is input arguments)</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Feed Cats</td>
<td><em>(given “void feed(cat c)”)</em> if (kitten[10].hungry()) feed(kitten[10]);</td>
<td>if (kitten[10]-&gt;hungry()) feed(*(kitten[10]));</td>
<td>if (kitten[10]-&gt;hungry()) feed(*(kitten[10]));</td>
<td>if (kitten[10]-&gt;hungry()) feed(*(kitten[10]));</td>
</tr>
<tr>
<td>Feed Cats2</td>
<td><em>(given “void feed(cat</em> c)”)* if (kitten[10].hungry()) feed(&amp;kitten[10]);</td>
<td>if (kitten[10]-&gt;hungry()) feed(kitten[10]);</td>
<td>if (kitten[10]-&gt;hungry()) feed(kitten[10]);</td>
<td>if (kitten[10]-&gt;hungry()) feed(kitten[10]);</td>
</tr>
<tr>
<td>Remove Cats</td>
<td>N/A (memory is all on the stack, so we don’t have to do anything)</td>
<td>for (int k=0; k&lt;10; k++) delete kittens[k];</td>
<td>delete [] kittens;</td>
<td>for (int k=0; k&lt;10; k++) delete kittens[k]; // #1 delete [] kittens; // #2</td>
</tr>
</tbody>
</table>

---

**Comment:** Don’t mixup this order.