CS 106B
Lecture 15: Dynamic Memory Allocation

Friday, May 4, 2018

Programming Abstractions
Spring 2018
Stanford University
Computer Science Department

Lecturer: Chris Gregg

reading:
Programming Abstractions in C++, Chapter 11
Today's Topics

• Logistics
  • Question from a student: "Is there any other interesting youtube channel you can recommend besides Numberphile?" Glad you asked! (see YouTube Video page at the end of the slides)

• More on Pointers
  • Mystery function
  • Pointers and Structs
  • The -> operator

• Dynamic Memory Allocation
  • The new and delete keywords
  • The "heap"
• Pointer Syntax #1:
  • To declare a pointer, use the * symbol. Example:
    ```
    string *laserPtr = NULL; // creates a pointer to a string and
    // sets it to point to a non-usable address.
    ```

• Pointer Syntax #2:
  • To put a variable's address into a pointer, use the & symbol. Example:
    ```
    string laser = "green"; // has some address (example: 0x12a5)
    laserPtr = &laser; // puts the address of laser
    // into the laserPtr variable.
    // The value of laserPtr is now 0x12a5
    ```

• Pointer Syntax #3:
  • To get value of the variable a pointer points to, use the "*". Example:
    ```
    string laserCopy = *laserPtr; // laserCopy is now "green"
    ```
What is a pointer??

a memory address!
void mystery(int a, int& b, int* c) {
    a++;
    (*c)--;  
    b += *c;
    cout << a << " " << b << " " << *c << " " << endl;
}

int main() {
    int a = 4;
    int b = 8;
    int c = -3;
    cout << a << " " << b << " " << c << " " << endl;
    mystery(c, a, &b);
    cout << a << " " << b << " " << c << " " << endl;
    return 0;
}
**Mystery Function: What prints out?**

```cpp
void mystery(int a, int& b, int* c) {
    a++;
    (*c)--;
    b += *c;
    cout << a << " " << b << " " << *c << " " << endl;
}

int main() {
    int a = 4;
    int b = 8;
    int c = -3;
    cout << a << " " << b << " " << c << " " << endl;
    mystery(c, a, &b);
    cout << a << " " << b << " " << c << " " << endl;
    return 0;
}
```

**Answer:**

```
4   8  -3
0x12 0xab 0xf3
4   8  -3
0x5e 0x7c 0xab
```

- A: 4
- B: 8
- C: -3
Mystery Function: What prints out?

```cpp
void mystery(int a, int& b, int* c) {
    a++;
    (*c)--;  
    b += *c;
    cout << a << " " << b << " " << *c << " " << endl;
}

int main() {
    int a = 4;
    int b = 8;
    int c = -3;
    cout << a << " " << b << " " << c << " " << endl;
    mystery(c, a, &b);
    cout << a << " " << b << " " << c << " " << endl;
    return 0;
}
```

Answer:

```
4 8 -3
```

```
4 8 0x7c
```
Mystery Function: What prints out?

```c
void mystery(int a, int& b, int* c) {
    a++;
    (*c)--;  
    b += *c;
    cout << a << "  " << b << "  " << *c << "  " << endl;
}

int main() {
    int a = 4;
    int b = 8;
    int c = -3;
    cout << a << "  " << b << "  " << c << "  " << endl;
    mystery(c, a, &b);
    cout << a << "  " << b << "  " << c << "  " << endl;
    return 0;
}
```

Answer:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8</td>
<td>-3</td>
</tr>
</tbody>
</table>

4 8 -3
Mystery Function: What prints out?

```c
void mystery(int a, int& b, int* c) {
    a++;
    (*c)--;  // *c--;
    b += *c;
    cout << a << " " << b << " " << *c << " " << endl;
}
int main() {
    int a = 4;
    int b = 8;
    int c = -3;
    cout << a << " " << b << " " << c << " " << endl;
    mystery(c, a, &b);
    cout << a << " " << b << " " << c << " " << endl;
    return 0;
}
```

Answer:

```
4 8 -3
```
Mystery Function: What prints out?

```cpp
void mystery(int a, int& b, int* c) {
    a++;
    (*c)--;   // b += *c;
    cout << a << " " << b << " " << *c << " " << endl;
}

int main() {
    int a = 4;
    int b = 8;
    int c = -3;
    cout << a << " " << b << " " << c << " " << endl;
    mystery(c, a, &b);
    cout << a << " " << b << " " << c << " " << endl;
    return 0;
}
```

Answer:

4 8 -3
void mystery(int a, int& b, int* c) {
    a++;
    (*c)--;
    b += *c;
    cout << a << " " << b << " " << *c << " " << endl;
}

int main() {
    int a = 4;
    int b = 8;
    int c = -3;
    cout << a << " " << b << " " << c << " " << endl;
    mystery(c, a, &b);
    cout << a << " " << b << " " << c << " " << endl;
    return 0;
}
Mystery Function: What prints out?

```cpp
void mystery(int a, int& b, int* c) {
    a++;
    (*c)--;
    b += *c;
    cout << a << " " << b << " " << *c << " " << endl;
}

int main() {
    int a = 4;
    int b = 8;
    int c = -3;
    cout << a << " " << b << " " << c << " " << endl;
    mystery(c, a, &b);
    cout << a << " " << b << " " << c << " " << endl;
    return 0;
}
```

Answer:

4 8 -3
Mystery Function: What prints out?

```cpp
void mystery(int a, int& b, int* c) {
    a++;
    (*c)--;
    b += *c;
    cout << a << " " << b << " " << *c << " " << endl;
}

int main() {
    int a = 4;
    int b = 8;
    int c = -3;
    cout << a << " " << b << " " << c << " " << endl;
    mystery(c, a, &b);
    cout << a << " " << b << " " << c << " " << endl;
    return 0;
}
```

Answer:

4 8 -3
void mystery(int a, int& b, int* c) {
    a++;
    (*c)--;
    b += *c;
    cout << a << " " << b << " " << *c << " " << endl;
}

int main() {
    int a = 4;
    int b = 8;
    int c = -3;
    cout << a << " " << b << " " << c << " " << endl;
    mystery(c, a, &b);
    cout << a << " " << b << " " << c << " " << endl;
    return 0;
}
```cpp
void mystery(int a, int& b, int* c) {
    a++;
    (*c)--;
    b += *c;
    cout << a << " " << b << " " << *c << " " << endl;
}

int main() {
    int a = 4;
    int b = 8;
    int c = -3;
    cout << a << " " << b << " " << c << " " << endl;
    mystery(c, a, &b);
    cout << a << " " << b << " " << c << " " << endl;
    return 0;
}
```
Mystery Function: What prints out?

```cpp
void mystery(int a, int& b, int* c) {
    a++;
    (*c)--;
    b += *c;
    cout << a << " " << b << " " << *c << " " << endl;
}

int main() {
    int a = 4;
    int b = 8;
    int c = -3;
    cout << a << " " << b << " " << c << " " << endl;
    mystery(c, a, &b);
    cout << a << " " << b << " " << c << " " << endl;
    return 0;
}
```

Answer:

```
4   8  -3
-2  11   7
11   7  -3
```
Pointers and Structs

- Pointers can point to a struct or class instance as well as to a regular variable.
- One way to do this would be to dereference and then use dot notation:

```cpp
Date d;
d.month = 7;
Date* dPtr = &d;
cout << (*dPtr).month << endl;
```
Pointers and Structs

- Pointers can point to a struct or class instance as well as to a regular variable.
- One way to do this would be to dereference and then use dot notation:

```cpp
date
  month  10
  day    31
daysInMonth() toString() ...

Date d;
d.month = 7;
Date* dPtr = &d;
cout << (*dPtr).month << endl;
```

- But, this notation is cumbersome, and the parenthesis are necessary because the "dot" has a higher precedence than the *.
• So, we have a different, and more intuitive syntax, called the "arrow" syntax, \( \rightarrow \):

\[
\begin{align*}
\text{Date } & \ d; \\
& \ d.\text{month} = 7; \\
\text{Date* } & \ dPtr = &d; \\
\text{cout } & \ll dPtr-&\text{month} \ll \text{ endl;} \\
\end{align*}
\]

• We will use the arrow syntax almost exclusively when using structs.
The arrow syntax can be used to set a value or call a function in a struct via a pointer, as well:

```cpp
Date d;
d.month = 7;
Date* dPtr = &d;
cout << dPtr->month << endl;
p->day = 1;
cout << p->day << endl; // 1
cout << p->toString() << endl; // 10/1
```
Dynamic Memory Allocation

- So far in this class, all variables we have seen have been local variables that we have defined inside functions. Sometimes, we have had to pass in object references to functions that modify those objects. For instance, take a look at the following code:

```c++
void squares(Vector<int> &vec, int numSquares) {
    for (int i=0; i < numSquares; i++) {
        vec.add(i * i);
    }
}
```

- This function requires the calling function to create a vector to use inside the function. This isn't necessarily bad, but could we do it a different way? In other words, could we create the Vector inside the function and just pass it back?
Dynamic Memory Allocation

- Could we create the Vector inside the function and just pass it back?

```cpp
Vector<int> squares(int numSquares) {
    Vector<int> vec;
    for (int i=0; i < numSquares; i++) {
        vec.add(i * i);
    }
    return vec;
}
```

- Does this work?
- It actually does, but there is an issue — you have to make a copy of the Vector, which is inefficient (though...these days if the creator of the Vector class is clever, we won't have to make a copy). Remember, we would rather not pass around large objects.
Dynamic Memory Allocation

- Okay...maybe we can do this?

```cpp
Vector<int> &squares(int numSquares) {
    Vector<int> vec;
    for (int i=0; i < numSquares; i++) {
        vec.add(i * i);
    }
    return vec;
}
```

- Does this work?
  - No :( This is actually really bad. Why? The scope of `vec` is only the function, and you are not allowed to pass back a reference to a variable that goes out of scope.
Dynamic Memory Allocation

• Well, how about with pointers? Can we do this?

```cpp
Vector<int> *squares(int numSquares) {
    Vector<int> vec;
    for (int i=0; i < numSquares; i++) {
        vec.add(i * i);
    }
    return &vec;
}
```

• Does this work?
• No :( This is also really bad. Why? Same as before: the scope of `vec` is only the function, and you are not allowed to pass back a pointer to a variable that goes out of scope. When the function ends, the variable is destroyed, and your program will almost certainly crash.
Dynamic Memory Allocation

• What do we want here? What's the big deal?

• What we really want is really two things:
  1. a way to reserve a section of memory so that it remains available to us throughout our entire program, or until we want to destroy it (give it back to the operating system)
  2. a way to reserve any amount of memory we want at the time we need it.

• You might think that global variables are what we want, but that would be incorrect.
• Global variables can be accessed by any function in our program, and that isn't what we want. Also, global variables have a fixed size at compile time, and that isn't what we want, either.
Dynamic Memory Allocation: new

- C++ allows you to request memory from the operating system using the keyword `new`. This memory comes from the "heap" whereas variables you simply declare come from the "stack." Both of those terms will become important in CS 107, but for now, you need to know this:

  - Variables on the stack have a scope based on the function they are declared in.
  - Memory from the heap is allocated to your program from the time you request the memory until the time you tell the operating system you no longer need it, or until your program ends.

- To request memory from the heap, we use the following syntax:
  
  ```cpp
  type *variable = new type; // allocate one element
  ```
  or
  
  ```cpp
  type *variable = new type[n]; // allocate n elements
  ```
Dynamic Memory Allocation: new

• Examples:

```cpp
int *anInteger = new int; // create one integer on the heap
int *tenInts = new int[10]; // create 10 integers on the heap
```

• The second example (`tenInts`) is very powerful — the memory you are given is an array guaranteed by the operating system to be contiguous. So, that's how we allocate an array of items dynamically!

• Notice that `new` returns a pointer to the type you request — this is important! This is why we need to learn about pointers — in order to dynamically allocate memory, you have to use a pointer.
We have been using Vectors in class so far, and we've said "Oh, a Vector is just built on top of an array." So, let's talk about arrays for a bit. They are "lower level" than Vectors, and they are more limited.

```java
int firstArray[10]; // create a static array on the stack;
// size of 10 is known at compile time

int *secondArray = new int[10]; // create 10 integers on the
// heap. Dynamically allocated.

// fill the arrays with values
for (int i=0; i < 10; i++) {
    firstArray[i] = i*2; // evens
    secondArray[i] = i*2 + 1; // odds
}
```

Arrays are not objects, and they don't have functions, so there isn't any function like `firstArray.length()`. You have to keep track of the length!
Arrays

- You have to keep track of the length!

```c
const int arrayLen = 10;

int firstArray[arrayLen]; // create a static array on the stack;
                           // size of 10 is known at compile time

int *secondArray = new int[arrayLen]; // create 10 integers on the
                                      // heap. Dynamically allocated.

// fill the arrays with values
for (int i=0; i < arrayLen; i++) {
    firstArray[i] = i*2; // evens
    secondArray[i] = i*2 + 1; // odds
}
```

- Notice, by the way, that we access our arrays by using bracket notation:
  - `firstArray[i]` gives us the value at index `i` in the array.
Arrays

- Unlike a vector, you can't just add another element past the end -- you are limited to the amount you asked for.

```cpp
const int arrayLen = 10;

int *myArray = new int[arrayLen]; // create 10 integers on the heap. Dynamically allocated.

// fill the array with values
for (int i=0; i < arrayLen; i++) {
    secondArray[i] = i*2 + 1; // odds
}

// add another?
secondArray.add(42); // nope!! Arrays don't have functions
secondArray[10] = 42; // nope!! Off the end of the memory
```

- When using arrays, you have to work with the limitations. We're taking off the training wheels!
The following statement requests an array of ten integers from the operating system (OS).

```c
int *tenInts = new int[10]; // create 10 integers on the heap
```

The OS looks for enough unallocated memory on the heap in a row to give you, then returns a pointer to that location (red is used, blue is free):
The following statement requests an array of ten integers from the operating system (OS).

```cpp
int *tenInts = new int[10]; // create 10 integers on the heap
```

For the above statement, the OS might pick row 3, column 4 for your request.
• The following statement requests an array of ten integers from the operating system (OS).

```java
int *tenInts = new int[10]; // create 10 integers on the heap
```

• For the above statement, the OS might pick row 3, column 4 for your request.
The following statement requests an array of ten integers from the operating system (OS).

\[
\text{int } \ast\text{tenInts} = \text{new int}[10]; // \text{create 10 integers on the heap}
\]

For the above statement, the OS might pick row 3, column 4 for your request.
• What would happen if you *do* try to write a value into a location you don't own?
• Possibilities:
  1. Compiler won't let you.
  2. Crash (seg fault)
  3. Nothing, as no one else is using that area
• What would happen if you do try to write a value into a location you don't own?
• Possibilities:
  1. Compiler won't let you.
  2. Crash (seg fault)
  3. Nothing, as no one else is using that area

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

tenInts

<table>
<thead>
<tr>
<th>43</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7c</td>
</tr>
</tbody>
</table>

Not your memory!
What would happen if you *do* try to write a value into a location you don't own?

- Compiler won't let you.
- Crash (seg fault)  
  - Maybe. The OS can say "I don't think so!" but it isn't guaranteed.
- Nothing, as no one else is using that area
What would happen if you *do* try to write a value into a location you don't own?

- Compiler won't let you.
- Crash (seg fault)
- Nothing, as no one else is using that area

Maybe. The OS might be okay with it, for now...it isn't guaranteed.
What would happen if you *do* try to write a value into a location you don't own?

Possibilities:
1. Compiler won't let you.
2. Crash (seg fault)
3. Nothing, as no one else is using that area
Buffer Overflows

The New York Times

VOL.CXXXVIII... No. 47,679
NEW YORK, FRIDAY, NOVEMBER 4, 1983

Late Edition


35 CENTS

“Virus” in Military Computers Disrupts Systems Nationwide

By John Markoff

An error in a program that runs on a computer at the National Aeronautics and Space Administration’s (NASA) Goddard Space Flight Center in Greenbelt, Md., has caused widespread disruptions to NASA’s computer networks.

The virus, which was discovered last week, has infected hundreds of computers at the center and has caused problems for NASA’s mission to Mars.

NASA is working with the National Security Agency (NSA) to contain the virus and prevent it from spreading to other government agencies.

PENTAGON REPORTS IMPROPER CHARGES FOR CONTRACTORS

By John W. Cushman Jr.

Inquiry Shows Routine Billing of Government by Industry on Fees, Some Dubious

Departments of Defense and Accounting have been accused of improperly charging the government for fees.

The report, which is expected to be released next week, has caused a stir in Congress.

The report’s findings suggest that the Defense Department has been overcharging the government for fees.

The report has caused a stir in the defense industry, which has been accused of overcharging the government for fees.

Registration Off Since 1984 Vote

There has been a pronounced decline in the percentage of eligible Americans who are registered to vote, a research group reports.

National, the percentage of eligible Americans who are registered to vote is 45.6 percent, down 11 points from the 1984 level.

The group’s study concluded that in many of the states where low-filer standards are available, the deck was among

Continued on page 48, column 2

The New York Times
Buffer Overflows

• In 1988, a computer "worm" written by Cornell graduate student Robert Morris, Jr. proliferated through government and university computers, bringing down the nascent Internet.

• The worm took advantage of a "buffer overflow" in a program, by writing code into a location that was outside the area that the program was given.

• The worm tricked the program into running its code, and was able to work its way through the network to other computers.

• The worm had a bug that made it eat up all of the computer's memory, thereby crashing the systems, one by one.
Buffer Overflows

- Robert Morris, Jr. became the first person in the U.S. convicted under the Computer Fraud and Abuse Act, and was fined, performed community service and served a three-year probation.
- He claimed that he was trying to demonstrate computer security faults, but the court did not believe him.
- He did bounce back: now he is a professor of computer science at MIT, and he co-founded the start-up incubator, Y-Combinator.
Dynamic Memory Allocation: delete

- The memory you request is yours until the end of the program, if you need it that long.
- You can pass around the pointer you get back as much as you'd like, and you have access to that memory through that pointer in any function you pass the pointer to.
- But, what if you are done using that memory? Let's say you create an array of 10 ints, use them for some task, and then are done with the memory?
- In this case, you delete the memory, giving it back to the Operating System:

```cpp
int *tenInts = new int[10]; // create 10 integers on the heap
for (int i=0; i < 10; i++) {
    tenInts[i] = randomInteger(1,1000);
}
someFunction(tenInts);
// done using tenInts
delete [] tenInts; // the [] is necessary for an array
```
Dynamic Memory Allocation: delete

- **delete** is sometimes confusing. Take a look at the following function:

```c
void arrayFun(int *origArray, int length) {
    // allocate space for a new array
    int *multiple = new int[length];

    for (int i=0; i < length; i++) {
        multiple[i] = origArray[i] * 2; // double each value
    }

    printArray(multiple, length); // prints each value doubled

    delete [] multiple; // give back the memory

    multiple = new int[length * 2]; // now twice as many
    for (int i=0; i < length; i++) {
        multiple[i*2] = origArray[i] * 2; // double each value
        multiple[i*2+1] = origArray[i] * 3; // triple the value
    }

    printArray(multiple, length * 2);

    delete [] multiple; // clean up
}
```
**Dynamic Memory Allocation: delete**

- **delete** is sometimes confusing. Take a look at the following function:

```c
void arrayFun(int *origArray, int length) {
    // allocate space for a new array
    int *multiple = new int[length];

    for (int i=0; i < length; i++) {
        multiple[i] = origArray[i] * 2; // double each value
    }

    printArray(multiple, length); // prints each value doubled

    delete [] multiple; // give back the memory

    multiple = new int[length * 2]; // now twice as many
    for (int i=0; i < length; i++) {
        multiple[i*2] = origArray[i] * 2; // double each value
        multiple[i*2+1] = origArray[i] * 3; // triple the value
    }

    printArray(multiple, length * 2);

    delete [] multiple; // clean up
}
```

- First: notice that we delete `multiple`, and then use it again!
- Is that allowed??
• **delete** is sometimes confusing. Take a look at the following function:

```c
void arrayFun(int *origArray, int length) {
    // allocate space for a new array
    int *multiple = new int[length];

    for (int i=0; i < length; i++) {
        multiple[i] = origArray[i] * 2; // double each value
    }

    printArray(multiple, length); // prints each value doubled
    delete [] multiple; // give back the memory

    multiple = new int[length * 2]; // now twice as many
    for (int i=0; i < length; i++) {
        multiple[i*2] = origArray[i] * 2; // double each value
        multiple[i*2+1] = origArray[i] * 3; // triple the value
    }

    printArray(multiple, length * 2);
    delete [] multiple; // clean up
}
```

- First: notice that we delete `multiple`, and then use it again!
- Is that allowed??
  - It is! **delete does not delete any variables! Instead, it follows the pointer and returns the memory to the OS!**
  
- However, you are not allowed to use the memory after you have **deleted** it.
  
- This does not preclude you from re-using the pointer itself.
Dynamic Memory Allocation: delete

- What does this print out, by the way for an \texttt{origArray} = \{1, 5, 7\}?

```cpp
void arrayFun(int *origArray, int length) {
    // allocate space for a new array
    int *multiple = new int[length];

    for (int i=0; i < length; i++) {
        multiple[i] = origArray[i] * 2; // double each value
    }

    printArray(multiple, length); // prints each value doubled

    delete [] multiple; // give back the memory

    multiple = new int[length * 2]; // now twice as many
    for (int i=0; i < length; i++) {
        multiple[i*2] = origArray[i] * 2; // double each value
        multiple[i*2+1] = origArray[i] * 3; // triple the value
    }

    printArray(multiple, length * 2);

    delete [] multiple; // clean up
}
```

```cpp
void printArray(int *array, int length) {
    cout << "[";
    for (int i=0; i < length; i++) {
        cout << array[i];
        if (i < length-1) {
            cout <<", ";
        }
    }
    cout << "]" << endl;
}
```

Output:

\[2, 10, 14\]
\[2, 3, 10, 15, 14, 21\]
Dynamic Memory Allocation: under the hood

• The memory you request is yours until the end of the program, if you need it that long.
• You can pass around the pointer you get back as much as you'd like, and you have access to that memory through that pointer in any function you pass the pointer to.

• Without knowing it, you have been using dynamic memory all along, through the use of the standard and Stanford library classes. The string, Vector, Map, Set, Stack, Queue, etc., all use dynamic memory to give you the data structures we have used for all our programs.
Thought experiment: the scary world without dynamic memory

• What if (horror!) we took away the Stanford library and asked you to write a Microsoft Word clone. Maybe you would start with something like this (although you'd probably make a Page class, instead):

```cpp
struct Page {
    string text;
    double leftM, rightM, topM, bottomM; // margins
    string header, footer;
    int textColor;
};
```

• How many pages should we allow the user of Stanford Word?
Thought experiment: the scary world without dynamic memory

• What if (horror!) we took away the Stanford library and asked you to write a Microsoft Word clone. Maybe you would start with something like this (although you'd probably make a Page class, instead):

```c++
struct Page {
    string text;
    double leftM, rightM, topM, bottomM; // margins
    string header, footer;
    int textColor;
};
```

• How many pages should we allow the user of Stanford Word? 5?

```c++
int main() {
    Page pages[5]; // array of 5 pages
}
```
Thought experiment: the scary world without dynamic memory

- People probably wouldn't buy your program if you limited them to five pages.

- Okay, let's make it bigger. How big? 6 pages? 100 pages? 1,000,000 pages?
- This is a no-win battle.
  - Too small, and your user might be unhappy.
  - Too big? Waste of memory! Your program would hog memory if you did the following:

```c
int main() {
    Page pages[1000000]; // array of a million pages
}
```
Next Time: Building a Vector class with arrays

• In the two lectures, we will discuss how the Vector is built, using dynamic memory.
• We will need to keep track of all the details ourselves:
  • How much space we have allocated for the Vector
  • How many items are in the Vector
  • How to add / remove / insert into the Vector
  • How to expand the Vector

Thanks for using Stanford Word! You can have unlimited pages!
Recap

- Dynamic Memory Allocation:
  - `new`: used to request heap memory that lasts for the rest of your program, or until you don't need it anymore.
  - `delete`: used to return memory to the operating system.
  - If you use `new` to request memory, you should `delete` it somewhere in your program.
  - You are **not allowed** to use memory that has been `deleted`.
  - Deleting memory does not somehow "delete" the pointer variable -- it goes to the location in memory pointed to, and tells the operating system that we are done with it.
References and Advanced Reading

• References:
  • new and delete: https://www.tutorialspoint.com/cplusplus/cpp_dynamic_memory.htm
  • Video on dynamic memory allocation: https://www.youtube.com/watch?v=OrDjGp_y1H4

• Advanced Reading:
  • Fun video on pointers: https://www.youtube.com/watch?v=B7IVHq-cgeU
  • Morris Worm: https://en.wikipedia.org/wiki/Morris_worm
  • Buffer Overflow vulnerabilities: https://en.wikipedia.org/wiki/Buffer_overflow
Chris's Favorite YouTube Channels

• Numberphile: https://www.youtube.com/user/numberphile
  • Cool videos about math, mathematical games and puzzles, number theory, etc.

• Computerphile: https://www.youtube.com/channel/UC9-y-6csu5WGm29I7JiwpnA
  • Sister site (produced by the same folks) as Numberphile, focusing on computers

• standupmaths: https://www.youtube.com/channel/UCSju5G2aFaWMqn-0YBtq5A
  • Numberphile spin-off from one of its contributors

• EEVBlog: https://www.youtube.com/channel/UCr-cm90DwFJC0W3f9jIBs5jA
  • If you are interested in electronics, this is the place to go. Great tutorials, commentary, etc., by a down-to-earth Australian. "Don't turn it on, take it apart!"

• bigclivedotcom: https://www.youtube.com/channel/UCtM5z2gkrGRuWd0JQMx76qA
  • Scottish version of EEVBlog, with detailed under-the-hood electronics

• AvE: https://www.youtube.com/channel/UChWv6Pn_zP0rl6lgGt3MyfA
  • Irreverent Canadian who demonstrates the mechanical engineering end of the spectrum

• Kurzgesagt — In a Nutshell: https://www.youtube.com/user/Kurzgesagt
  • Short videos on cool stuff

• VSauce: https://www.youtube.com/user/Vsauce
  • Science, math, philosophy, etc.
Operator Overloading: the What and the Why

• More Tiny Feedback: "Not sure I understood operator overriding [sic] very well."

• C++ allows us to overload operators. What does that mean?
  • It means that if we create a class, we can utilize common operators (e.g., +, *, <<, etc.) to do what we want.

• Why do we do it?
  • We want to make things easier and more straightforward for the user of our class.
Let's look back at our Fraction class. We implemented a mult() function:

```cpp
void Fraction::mult(Fraction other)
{
    // multiplies a Fraction
    // with this Fraction
    num *= other.num;
    denom *= other.denom;

    // reduce the fraction
    reduce();
}
```

But, this isn't the normal way we multiply numbers. We can fix that!
• Our .h file needs a friend function:

```cpp
class Fraction {
public:
    ... 

friend Fraction operator*(const Fraction &first, const Fraction &second);
    ...

    ... 

• The "const" just means we aren't allowed to change either number when multiplying.
Operator Overloading: Example

• The function in our .cpp file:

```cpp
Fraction operator*(const Fraction &first, const Fraction &second) {
    int newNum = first.num * second.num;
    int newDenom = first.denom * second.denom;

    return Fraction(newNum, newDenom); // will be reduced automatically
}
```

• We simply return a new Fraction (which copies its elements. Luckily, they aren't pointers!)

• We call the function like this:

```cpp
Fraction two_thirds = Fraction(2,3);
Fraction three_fourths = Fraction(3,4);
Fraction result = two_thirds * three_fourths; // result = 1/2
```
With great power comes great responsibility

You have a responsibility when using dynamic memory allocation to `delete` anything you have requested via `new`.

This is the contract you make with the operating system: if you're done with the memory, you should return it. The OS will take it back when your program ends, but this wastes memory, and this is called a "memory leak."

```cpp
const int INIT_CAPACITY = 10000000;

class Demo {
public:
    Demo(); // constructor
    string at(int i);
private:
    string *bigArray;
};

demo::Demo() {
    bigArray = new string[INIT_CAPACITY];
    for (int i=0; i<INIT_CAPACITY; i++) {
        bigArray[i] = "Lalalalalalalalala!";
    }
}

string Demo::at(int i) {
    return bigArray[i];
}

int main() {
    for (int i=0; i<10000; i++) {
        Demo demo;
        cout << i << ": " << demo.at(1234) << endl;
    }
    return 0;
}
```

This program crashed my entire computer when I ran it. Why?
Dynamic Memory Allocation: your responsibilities

• This program crashed my entire computer when I ran it. Why?
• We're allocating a *ton* of memory, and not deleting it!
• We can fix it by adding a "destructor" -- when the class instance goes out of scope, the destructor is called, cleaning up the memory for us.

```cpp
const int INIT_CAPACITY = 10000000;

class Demo {
public:
    Demo(); // constructor
    ~Demo(); // destructor
    string at(int i);
private:
    string *bigArray;
};

Demo::Demo() {
    bigArray = new string[INIT_CAPACITY];
    for (int i=0; i<INIT_CAPACITY; i++) {
        bigArray[i] = "Lalalalalalalalalala!";
    }
}

Demo::~Demo() {
    delete[] big_array;
}

string Demo::at(int i) {
    return bigArray[i];
}

int main()
{
    for (int i=0; i<10000; i++){
        Demo demo;
        cout << i << " : " << demo.at(1234) << endl;
    }
    return 0;
}
```