CS 106B
Lecture 16: Dynamic Memory Allocation
Monday, October 31, 2016

Programming Abstractions
Fall 2016
Stanford University
Computer Science Department

Lecturer: Chris Gregg

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

• Logistics
  • Chris Gregg's office hours for Tuesday moved to Wednesday 5-6pm
• Midterm information:
  • Thursday November 3rd 7:00-9:00pm
  • Last name starts with A-R: Cemex Auditorium (GSB)
  • Last name starts with S-Z: Braun Auditorium (Mudd Chemistry Building)
• Midterm Review Session Video online
• Two practice midterms on the website.

• More on Pointers
  • Mystery function
  • Pointers and Structs
  • The -> operator
• Dynamic Memory Allocation
  • The new and delete keywords
  • The "heap"
Recap of Pointer Syntax from the last lecture

• Pointer Syntax #1:
  • To declare a pointer, use the * symbol. Example:
    
    ```
    string *scaryPetPtr = 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 scaryPet = "werewolf"; // has some address (example: 0x12a5)  
    scaryPetPtr = &pet; // puts the address of scaryPet  
    // into the petPtr variable.  
    // The value of petPtr is now 0x12a5
    ```

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

a memory address!
Mystery Function: What prints out?

```cpp
void scaryMystery(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;
    scaryMystery(c, a, &b);
    cout << a << " " << b << " " << c << " " << endl;
    return 0;
}
```
```cpp
void scaryMystery(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;
    scaryMystery(c, a, &b);
    cout << a << " " << b << " " << c << " " << endl;
    return 0;
}
```

**Answer:**

```
a 4 0x12
b 8 0xab
c -3 0xf3
```
void scaryMystery(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;
    scaryMystery(c, a, &b);
    cout << a << " " << b << " " << c << " " << endl;
    return 0;
}
Mystery Function: What prints out?

```c
void scaryMystery(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;
    scaryMystery(c, a, &b);
    cout << a << " " << b << " " << c << " " << endl;
    return 0;
}
```

Answer:

4   8  -3
Mystery Function: What prints out?

```c
void scaryMystery(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;
    scaryMystery(c, a, &b);
    cout << a << " " << b << " " << c << " " << endl;
    return 0;
}
```

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

```cpp
void scaryMystery(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;
    scaryMystery(c, a, &b);
    cout << a << " " << b << " " << c << " " << endl;
    return 0;
}
```

Answer:

```
a
4 0x12
b
7 0xab
c
-3 0xf3
```
void scaryMystery(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;  
    scaryMystery(c, a, &b);  
cout << a << " " << b << " " << c << " " << endl;  
    return 0;
}
Mystery Function: What prints out?

```c
void scaryMystery(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;
    scaryMystery(c, a, &b);
    cout << a << " " << b << " " << c << " " << endl;
    return 0;
}
```

Answer:

```
4   8  -3
```

```c
11  7  -3
```

```
0x12 0xab 0xf3
```
Mystery Function: What prints out?

```c
void scaryMystery(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;
    scaryMystery(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>11</td>
<td>7</td>
<td>-3</td>
</tr>
</tbody>
</table>

4 8 -3
Mystery Function: What prints out?

```c++
void scaryMystery(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;
    scaryMystery(c, a, &b);
    cout << a << " " << b << " " << c << " " << endl;
    return 0;
}
```

Answer:

```
4   8  -3
-2  11   7
```

```
void scaryMystery(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;
    scaryMystery(c, a, &b);
    cout << a << " " << b << " " << c << " " << endl;
    return 0;
}
```
Mystery Function: What prints out?

```c++
void scaryMystery(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;
    scaryMystery(c, a, &b);
    cout << a << " " << b << " " << c << " " << endl;
    return 0;
}
```

Answer:

```
4 8 -3
-2 11 7
```

{blackboard}

```
void scaryMystery(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;
    scaryMystery(c, a, &b);
    cout << a << " " << b << " " << c << " " << endl;
    return 0;
}
```
void scaryMystery(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;
    scaryMystery(c, a, &b);
    cout << a << " " << b << " " << c << " " << endl;
    return 0;
}
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:

```
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, \texttt{\rightarrow}:

```cpp
Date d;
d.month = 7;
Date* dPtr = &d;
cout << dPtr->month << endl;
```

We will use the arrow syntax almost exclusively when using structs.
Pointers and 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;  // 7
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. 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.
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:

```
type *variable = new type; // allocate one element
```

or

```
type *variable = new type[n]; // allocate n elements
```
Dynamic Memory Allocation: new

• Examples:

    ```
    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.
Arrays

- 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!

```cpp
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
                          // space you were given
```

- When using arrays, you have to work with the limitations. We're taking off the training wheels!
Dynamic Memory Allocation: under the hood

- 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 in a row to give you, then returns a pointer to that location (red is used, blue is free):
Dynamic Memory Allocation: under the hood

• 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
```

• 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).

```c
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):

```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.
Dynamic Memory Allocation: under the hood

- 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. **Sadly, the compiler can't tell. You're on your own!**
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)   Maybe. The OS can say "I don't think so!" but it isn't guaranteed.
3. Nothing, as no one else is using that area
Dynamic Memory Allocation: under the hood

- 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

Maybe. The OS might be okay with it, for now...it isn't guaranteed.
Dynamic Memory Allocation: under the hood

- 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

![Diagram showing memory allocation](image)
"All the News That's Fit to Print"

The New York Times

Vol. XXXVIII No. 47,679
New York, Friday, November 4, 1983

PENTAGON REPORTS IMPROPER CHARGES FOR CONSULTANTS

CONTRACTORS CRITICIZED

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

BY JOHN H. CUNNINGHAM Jr.
Special to The New York Times

WASHINGTON, Nov. 3 — A Pentagon investigation has found that the nation’s largest military contractors repeatedly charge the Defense Department for hundreds of millions of dollars paid to consultants, often without justification.

The report of the investigation said that neither the military’s own policies nor those of the contractors are adequate to assure that the Government does not improperly pay for privately arranged consulting work. Former Defense Department officials said the Pentagon was proposing to change the rules.

While it is not improper for military contractors to hire consultants to prepare learning programs for the Pentagon, the work must directly benefit the military if it is to be paid for by the Defense Department. Often, Pentagon investigators discovered, this use is not made.

Broadly Look at Consultants

The Justice Department’s continuing criminal investigation has focused on contractors and their role in the designing and selling of weapons, and the Defense Department has been criticized for using consultants too heavily. That article was from Page A18, column 2.

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.

Nationally, the percentage of eligible Americans who are registered is estimated to be 35.5 percent, down 12 points from the 1984 levels.

The group’s study concluded that in many of the 38 states where these figures are available the decline was even greater.

Gov Michael S. Dukakis, holding the picture taken by a 10-year-old tax a town meeting in Franklin, N.H., a dairy farm in the Northeast where he is being called a liberal, Mr. Bush said yesterday that “this election is not about ideology.” Page A18.
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.
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??
**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??
- 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.

• It is! **delete does not delete any variables! Instead, it follows the pointer and returns the memory to the OS!**

• First: notice that we delete `multiple`, and then use it again!

• Is that allowed??

• 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 origArray = {1, 5, 7}?

```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
}
```

```c++
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):

```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? 5?

```cpp
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.

![Image of Stanford Word message box]

• 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 next lecture, 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
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=B7lVHq-cgeU
• Morris Worm: https://en.wikipedia.org/wiki/Morris_worm
• Buffer Overflow vulnerabilities: https://en.wikipedia.org/wiki/Buffer_overflow
Extra Slides (will cover next time)
Dynamic Memory Allocation: your responsibilities

- 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;
}
```