Dynamic Memory

Review: automatic variables

- ***Automatic variable***: memory is ***allocated*** (reserved) and ***deallocated*** (freed up) automatically.
- Always stored on the stack.
- "Normal" way to make a variable
- Up until last Friday, all variables in our programs were automatic.

NSA Spying

- Suppose we work for the NSA and we are creating a program to manage our spying database.
- We want to write a function that loads the spy database into our program.

```
database load_database() {
  database db;
  // load all the records of everyone
  // on earth into db
  return db;
}

int main() {
  database db = load_database();
  // launch drones at whomever we want…
}
```

```
database* load_database() {
  database db;
  // load all the records of everyone
  // on earth into db
  return &db;
}

int main() {
  database * db = load_database();
  // launch drones at whomever we want…
}
```

Dynamic memory allocation

- We need a way to declare a variable so that it will not be deallocated when it ***goes out of scope.***
- Dynamic memory allocation to the rescue!

Dynamic memory allocation

- `type * ptr = new type;`
  - allocate memory on the heap for one new variable of type type and return a pointer to it.
- `delete ptr;`
  - deallocate the memory pointed to by ptr
  - good idea to then set `ptr` to `nullptr`
- You must deallocate all your memory when you are done with it!

```
database* load_database() {
  database * db = new database;
  // load all the records of everyone
  // on earth into db
  return db;
}

int main() {
  database * db = load_database();
  // launch drones at whomever we want…
  delete db;
}
```

Dynamic memory gotchas

- The *pointer* to the dynamic memory is still an automatic variable, so it must be passed and returned from functions like normal.
- You can copy that pointer as much as you want, but you must `delete` it exactly once (no matter how many copies there are floating around).

Dynamic memory gotchas

- After memory is `deleted`, it may be allocated for something else, so any existing pointers to that memory should be considered invalid.
- Deleting the same memory twice is bad.
- You can delete memory anytime you want.

Try this

- Allocate two new ints on the heap (dynamically).
- Set them equal to 10 and 20 and print them.
- Switch the pointers so each pointer now points to the opposite int.
- Print them again.
- Deallocate the integers.

Allocating lots of vars at once

- `type * ptr = new type[num];`
  - allocate memory on the heap for **`num`** new variables of type `type` and return a pointer to it.
- `delete[] ptr;`
  - deallocate the memory pointed to by ptr
  - only use delete[] with new[]
  - only use delete with new

Variables that grow and/or shrink

- Using new type[num] still doesn't make the dynamic memory grow or shrink.
- So how do vectors work?
  - A vector starts off my allocating (using new) a "default" amount of space for items in the vector.
  - If we add too many things to a vector, it will allocate more space, copy everything in the vector into the new space, then delete[] the old space.

Try this

- Allocate (on the heap) an array of 5 doubles.
- Assign some numbers to the array.
- [Pretend that we want to add more numbers.]
- Allocate (on the heap) a second array of 10 doubles.
- Copy the doubles from the old array into the new one.
- delete[] the old array.
- Print the new array.
- delete[] the new array.