Managing dynamic memory

- Constructor (usually) allocates it

```
StringVar(const char a[]);
...
StringVar::StringVar(const char a[]) :
                    max_length(strlen(a)) {
    value = new char[max_length + 1];
    strcpy(value, a);
}
```

- But what happens when the object is destroyed?

```
StringVar s1("hot"); // on stack, will go out of scope soon
```

- Solution is to define a destructor (a.k.a. dtor)

Destructors - dtors

- A dtor is invoked whenever an object goes out of scope, or by `delete` for objects on free store
  - Compiler supplies a default one if you don't
  - Default won't free dynamic memory or other resources
- Defined like a ctor, but with a ~ in front, and it may not take any arguments

```
~StringVar();
...
StringVar::~StringVar() { delete [] value; }
```

- *Can* invoke directly on an object (unlike ctors)

```
stringPtr->~StringVar(); // rarely done though
```

Manager functions (inc. Big 3)

- 4 functions every class must properly manage:
  - Default ctor, copy ctor, dtor, and assignment operator
    - Compiler supplies defaults of all 4, but often should redefine
  - Latter three also known as "The Big Three" – *if you need to redefine one of them, then you need to redefine all three of them*
- Copy ctor – `StringVar(const StringVar&);`
  - Compiler-supplied version makes a "shallow copy"
  - Invoked when initializing with object as argument: `StringVar s(otherString);`
    - Or by "C-style" syntax: `StringVar s = otherString;`
  - Also invoked to pass (or return) an object by value to (or from) a function

See 11-11.cpp and 11-12.cpp (also in ~mikec/cs32/Savitch/Chapter11/)

Implementing StringVar copy ctor

- Question: why not just keep the default copy ctor for StringVar objects?
- Ans: Need a complete, independent copy of the argument – even if the argument is *this
  - Therefore must create new dynamic array, and copy all characters to the new array

```
StringVar::StringVar(const StringVar& other) :
  max_length(other.length()) {
    value = new char[max_length + 1];
    strcpy(value, other.value);
}
```

Why redefine the = operator?

- Given these declarations:

```
StringVar s1("cat"), s2("rabbit");
```

- The following statement is legal:

```
s1 = s2;
```

- But without redefining operator=, we would have s1.value and s2.value both pointing to the same memory location (a "shallow copy")
  - Furthermore, s1's old value is now a memory leak
- So: `StringVar& StringVar::operator=`
  `(const StringVar& right);`

Defining operator= [version 1]

- The definition of = for StringVar could be as follows:

```
StringVar& StringVar::operator=
                  (const StringVar& right){

    int new_length = strlen(right.value);
    if (( new_length) > max_length)
        new_length = max_length;

    for(int i = 0; i < new_length; i++)
        value[i] = right.value[i];
    value[new_length] = '\0';
}
```

- Notice anything wrong with this version?

Defining operator= [version 2]

```
StringVar& StringVar::operator=
                    (const StringVar& right){
    delete[] value;
    int new_length = strlen(right.value);
    max_length = new_length;
    value = new char[max_length + 1];

    for(int i = 0; i < new_length; i++)
        value[i] = right.value[i];
    value[new_length] = '\0';
}
```

- That solves problem of incompletely copied strings, but …
- What if somebody uses it as follows? `s1 = s1;`

Defining operator= [finally?]

- Idea is to delete value only if more space needed:

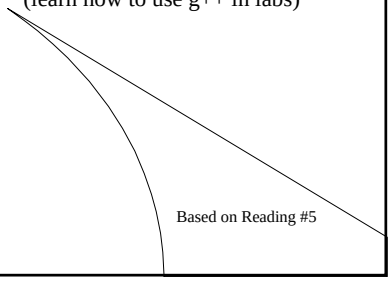
```
StringVar& StringVar::operator=
                    (const StringVar& right){
    int new_length = strlen(right.value);
    if (new_length > max_length) {
        delete[] value;
        max_length = new_length;
        value = new char[max_length + 1];
    }
    for(int i = 0; i < new_length; i++)
        value[i] = right.value[i];
    value[new_length] = '\0';
}
```

Demos: advanced class design

- ~mikec/cs32/demos/IntArray/ files
  - Mostly about dealing with objects pointing to dynamic memory
- ~mikec/cs32/demos/String/ files
  - Full-featured string-like class, with many overloaded operators and other functions that are not part of the textbook's StringVar class

About building a program so Linux (the OS) can run it

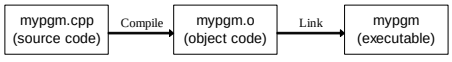
Starting to learn what gcc/g++ does (learn how to use g++ in labs)



Based on Reading #5

Program building

- Have: source code – human readable instructions
- Need: machine language program – binary instructions and associated data regions, ready to be executed
- g++/gcc does two basic steps: compile, then link
  - To compile means translate to object code
  - To link means to combine with other object code (including library code) into an executable program



Link combines object codes

- From multiple source files and/or libraries
  - e.g., always libc.a

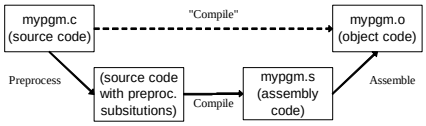
- Use -c option with gcc/g++ to stop after creating .o file

```
-bash-4.2$ gcc -c mypgm.c ; ls mypgm*
mypgm.c  mypgm.o
```

  - Is necessary to compile a file without a main function
- Later link it to libraries – alone or with other object files:

```
-bash-4.2$ gcc -o mypgm mypgm.o ; ls mypgm*
mypgm  mypgm.c  mypgm.o
```

Compiling: 3 steps with C/C++



- First the preprocessor runs
  - Creates temporary source code with text substitutions as directed
  - Use `gcc -E` (or just `cpp`) to run it alone – output goes to `stdout`
- Then the source is actually compiled to assembly code
  - Use `gcc -S` to stop at this step and save code in `.s` file
- Last, assembler produces the object code (machine language)

Automate builds with make
(a short intro to Lab06)

- make is a Unix/gnu tool that executes actions as necessary to satisfy dependencies
- First create a "Makefile" (learn tips in Lab06 and Hw6)

```
pgm: pgm.o                  # dependency
        gcc  pgm.o –o pgm # action (tab required)
pgm.o: pgm.c
        gcc -c pgm.c
```

- Why bother learning, and using the make tool?
  - Some projects have many, many modules; even many programmers. Automated, so guarantees complete and up-to-date builds, without needless steps.
  - Just type "make" – the program does the rest