Storage allocation options: extern, auto, static, etc.

23 January 2009
CMPT166
Dr. Sean Ho
Trinity Western University

TRINITY WESTERN UNIVERSITY

Review of last time

- <string>
- <fstream>, ifstream, getline(), ofstream
- <vector>

- Addendum: pre-increment / post-increment
  - i++ returns the value of I first, then increments
  - ++i increments first, then returns the new value of I
    - i=5; cout << ++i; // prints 6

Storage flags for variables

- When declaring a variable in C++, we have several optional flags/modifiers we can apply:
- extern
- auto
- static
- register
- const vs. volatile

Global variables

- Global entities (variables, functions) are available to all parts of the program
  - Even code linked in from other files
- Anything declared in the top level of the file (outside any function or class) is global
- A global object can be accessed by another file, but that file still needs a declaration so the compiler knows that the global object exists
  - Use extern to indicate the object is allocated elsewhere

Memory allocation and linkage

- Every global variable in the program must have its memory allocated exactly once
- If multiple files within the program want to access the global variable, it should be allocated in only one of those files
    - The other files declare it extern to tell the compiler that it will be allocated elsewhere
- The linker assembles all the object files and connects all the references to the global variable

extern: sample usage

- File global1.cpp:
  - int globalApples;
  - void multApples(); // declaration only
  - void main() {
    - **globalApples = 5;**
    - **multApples(); }**
- File apples.cpp:
  - extern int globalApples; // in global1.cpp
  - void multApples() {
    - **globalApples *= 2; }**

auto: local variables

- Anything declared inside a function or class is local to that function or class
  - Scope rules: where the entity is accessible
- The keyword auto also indicates a variable needs to be local in scope
  - Don't usually need to use auto; it's default
    - **void getInput() {**
      - **auto string myInput;**
      - **cin >> myInput;**
    - **}**

register: fast access

- Registers are hardware memory very close to the CPU, very fast access but very limited space
- The register keyword asks the compiler to make access to this variable as fast as possible
  - register int criticalApples;
  - Cannot use pointers with registers
  - Cannot be global or static
- Generally, the compiler does a good job of placing your variables in memory, so register is not needed

static: persistent data

- Usually, local variables in a function are deallocated when the function finishes
- static: the variable stays around; keeps its old value from the last time the function was run
  - Initialization is done only the first time
    - void incCounter() {
      - static int i = 0;
      - cout << “i = “ << ++i << endl; }
  - Each call to incCounter() adds one to i
  - i is still only accessible from inside incCounter() (not global!)

static: file scope

- static has a second meaning, when applied to a function or to a global variable:
  - File scope: this name is unavailable outside this file
- If we changed the preceding example to declare globalApples static:
  - File global1.cpp:
    - **static int globalApples;**
  - File apples.cpp:
    - **extern int globalApples; // fails!**

const and volatile

- In C++ the compiler can enforce constants:
  - const int numApples = 10;
    - Must initialize in the declaration
    - Tags the variable as unchangeable
- The volatile keyword hints to the compiler that this variable may change quite often
  - volatile int myMood;
    - Compiler does optimizations accordingly