Storage Classes of Variables and Preprocessing

Cscd240 Yun Tian (Ph.D.)

Concept

- Each variable has a storage class, besides its type, address and value.
- The storage classes are:
  - auto, static, extern, and register

Auto Variable

- Any variable declared within a block is considered as automatic local variable
  - Stored on the stack
  - Memory space (storage) for the variable is allocated automatically when the block is entered, and deallocated on exit.
    - When a block is reentered, all former values are gone.
  - Keyword ‘auto’ can be left out.
  - Blocks can occur anywhere in a program

Auto Variable

- Any variable declared within a block is considered as automatic local variable
  - Blocks can occur anywhere in a program
    - Function bodies,
    - Loop
    - Swith statement
    - Anywhere in code
  - Demo on local machine
    - scopeTest1.c

extern

- Any variable declared **outside** all blocks is considered as global variable,
- It has storage class of extern.
- Scope is global,
  - Meaning visible to all your source code for a project.
- Duration is the entire program execution.
  - Meaning it is alive during the whole program execution

extern

- The keyword ‘extern’ is used to tell the compiler that the variable is defined in another file and has storage class extern.
  - The linker will come in later to find where these global variables are defined.
- Demo of extern on local machine

register

- The keyword ‘register’ is used to tell the compiler that a general purpose register be used for the variable.
- Avoid memory fetches and makes arithmetic much faster.
- But it depends on the availability of a register at runtime.
- No (&) operation for this type of variable.

static

- A static variable inside a function keeps its former value between invocations.
  - Scope: local to function
  - Duration: whole program execution.
- A static global variable or a static function is ‘seen’ only in the file in which it is defined.
  - Limit its scope to that source file where it is defined.
  - Duration: entire program execution for variables
- Demo on local machine

C preprocessing

- C preprocessor is just a text substitution tool and
- It instructs compiler to do required pre-processing before actual compilation.
- All preprocessor commands begin with a ‘#’
- Some important directives on the next page.

C preprocessing

- #define
  - Substitutes a preprocessor macro
- #include
  - Inserts a particular header from another file
- #undef
  - Undefines a preprocessor macro
- #ifdef
  - Returns true if this macro is defined
- #ifndef
  - Returns true if this macro is not defined
- #if
  - Tests if a compile time condition is true
- #else
  - The alternative for #if
- #elif
- #else an #if in one statement
- #endif
  - Ends preprocessor conditional
- #pragma
  - Issues special commands to the compiler, using a standardized method
  - https://gcc.gnu.org/onlinedocs/gcc/Pragmas.html

Using C preprocessing

- Conditional compilation
  - #ifdef, #ifndef, #if, #elif, #endif
- 1, Used for code portability
- E.g.

```
#define UNIX
……
#ifdef UNIX
        strcpy(path, “/home/tony”);
#else
        strcpy(path, “\home\tony”;
#endif
```

Using C preprocessing

- Conditional compilation
  - #ifdef, #ifndef, #if, #elif, #endif
- 2, avoid multiple inclusions -- #ifndef (Used in lab 11)
- E.g. in file myinclude.h

```
#ifndef MYINCLUDE_H
#define MYINCLUDE_H
…….the rest of the header file is placed here
#endif
```

- Scenario where it is useful.

  Assume you have f1.h where you #include myinclude.h,

  Assume you have f3.h where you #include myinclude.h,

  Assume you have f2.h where you #include f1.h and f3.h,

  In this case, the preprocessor make sure the content in myinclude.h will be only included once in f2.h.

Using C preprocessing

- Conditional compilation
  - #ifdef, #ifndef, #if, #elif, #endif
- 3, debugging -- #if
- E.g.

```
#define DEBUG  1  //on top of your source file
…….
#if DEBUG
   printf("debug info: x = %d\n", x);
#endif
```

Scenario where it is useful.

When you first debugging and testing your program, you like the debug information to be displayed on the stdout. That is, you have to do '#define DEBUG 1' on the top of your source file.

After testing and debugging, you like to suppress these debug information. You do this by commenting out the first line.

Using C preprocessing

- Conditional compilation
  - #ifdef, #ifndef, #if, #elif, #endif
- 3, debugging -- #if
- E.g. in tester.c of Demo on local machine

```
…….
#if DEBUG
    printf(“debug info: x = %d\n”, x);
#endif
```

You can define DEBUG when compiling your code:

```
gcc –D DEBUG=1  tester.c –o tester
```