Lecture 17: Declarations

Sarah Nadi
nadi@ualberta.ca
Department of Computing Science
University of Alberta

CMPUT 201 - Practical Programming Methodology

[With material/slides from Guohui Lin, Davood Rafei, and Michael Buro. Some content taken from K.N. King's slides based on course text book]

Agenda

- Declarations
- Storage classes
- type specifier
- type qualifier
- Deciphering complex declarations

Readings

- Textbook Chapter 18.1-18.5

Declarations

- We have already seen most types of declarations, but today we will go into more details

Why Are Declarations Important?

- Declarations provide the compiler with the information it needs to understand the identifiers it sees

General Form of a Declaration

`declaration-specifiers declarators;`

- *Declaration specifiers* describe properties of the variables or functions being declared. Three categories of specifiers:
  - *storage classes:* `auto, static, extern, register` (at most one per declaration)
  - *type qualifiers:* `const, volatile,` and `restrict` (only in C99).
  - *type specifiers*: keywords such as `void`, `char`, `short, int, long, float, signed, unsigned`, and specifications of structures, unions, and enumerations. Type names created by `typedef` are type specifiers as well.
- *Declarators* provide the names of the identifiers. Multiple declarators are separated by commas.

Example Declarations

```
static float x, y, *p;
```

Example Declarations

```
static float x, y, *p;
```

storage class

Example Declarations

type specifier

```
static float x, y, *p;
```

storage class

Example Declarations

type specifier

```
static float x, y, *p;
```

declarators

storage class

Example Declarations

```
const char month[] = "January";
```

Example Declarations

```
const char month[] = "January";
```

type qualifier

Example Declarations

```
const char month[] = "January";
```

type qualifier

type specifier

Example Declarations

declarator

```
const char month[] = "January";
```

type qualifier

type specifier

Example Declarations

declarator

```
const char month[] = "January";
```

type qualifier

type specifier

initializer

Example Declarations

```
extern const unsigned long int a[10];
```

Example Declarations

```
extern const unsigned long int a[10];
```

storage class

Example Declarations

type qualifier

```
extern const unsigned long int a[10];
```

storage class

Example Declarations

type qualifier

```
extern const unsigned long int a[10];
```

storage class

type specifier

Example Declarations

type qualifier

declarator

```
extern const unsigned long int a[10];
```

storage class

type specifier

Properties of Variables

- ***Storage duration:*** determines when memory is set aside for the variable and when memory is released.
  - A variable with *automatic storage duration* is allocated when the surrounding block is executed & deallocated when the block terminates, causing the variable to loose its value.
  - A variable with *static storage duration* stays at the same storage location as long as the program is running, allowing it to retain its value indefinitely

Storage Duration Example

```
int i;

void f ( ) {
 int j;
}
```

Storage Duration Example

by default, global variables have static storage duration

```
int i;

void f ( ) {
 int j;
}
```

Storage Duration Example

by default, global variables have static storage duration

```
int i;

void f ( ) {
 int j;
}
```

default for local variables is automatic storage duration

Properties of Variables
Cont'd

- ***Scope:*** the scope of a variable is the portion of program text in which the variable can be referenced.
    - A variable can have either *block scope* (variable can be referenced from its declaration point to the end of the enclosing block) or *file scope* (variable is visible from its point of declaration to the end of the enclosing file).

Scope Example

```
int i;

void f ( ) {
 int j;
}
```

Scope Example

file scope —
can be accessed from this point downwards in the file

```
int i;

void f ( ) {
 int j;
}
```

Scope Example

file scope —
can be accessed from this point downwards in the file

```
int i;

void f ( ) {
 int j;
}
```

block scope — can be accessed only within the current block (in this case function f)

Properties of Variables *Cont'd*

NEW

- ***Linkage:*** the linkage of a variable determine the extent to which it can be shared by different parts of the program.
  - A variable with *external linkage* may be shared by several (perhaps all) files in a program.
  - A variable with *internal linkage* is restricted to a single file (if a variable with the same name appears in a different file, it is treated as a different variable), but may be shared by the functions in that file.
  - A variable with *no linkage* belongs to a single function and can't be shared at all.

Default Storage Duration, Scope, and Linkage

- Variables declared *inside* a block (including a function body) have **automatic** storage duration, **block** scope, and **no** linkage.
- Variables declared *outside* any block, at the outermost level of a program, have **static** storage duration, **file** scope, and **external** linkage.

Default Storage Duration, Scope, and Linkage

- Variables declared *inside* a block (including a function body) have **automatic** storage duration, **block** scope, and **no** linkage.
- Variables declared *outside* any block, at the outermost level of a program, have **static** storage duration, **file** scope, and **external** linkage.

```
int i;

void f(){
  int j;
}
```

static storage duration,
file scope,
external linkage

automatic storage duration,
block scope,
no linkage

Default Storage Duration, Scope, and Linkage

- Variables declared *inside* a block (including a function body) have **automatic** storage duration, **block** scope, and **no** linkage.
- Variables declared *outside* any block, at the outermost level of a program, have **static** storage duration, **file** scope, and **external** linkage.

```
int i;
```

static storage duration,
file scope,
external linkage

```
void f(){
  int j;
}
```

automatic storage duration,
block scope,
no linkage

You can alter a variable's default storage duration and linkage by specifying an explicit storage class: **`auto`, `static`, `extern`, or `register`**

The `auto` Storage Class

- legal only for variables that belong to a block (i.e., not those declared at the file level)
- An `auto` variable (as the name suggests) has automatic storage duration
- Almost never specified, since it is the default for variables declared in a block — exists for historic reasons

The `static` Storage Class

- Can be used with all variables, regardless of where they are declared. However, the keyword does different things depending on where it is used:
  - when used **outside a block**: the keyword `static` changes the *variable's linkage* from external linkage to **internal** linkage
  - when used **inside** a block: the keyword `static` changes the *storage duration* of the variable from automatic to **static.** Remember that this means the variable is only initialized once in the beginning and any changes to its value are retained in subsequent executions of this block.

Examples of using `static`

```c
static int i;

void f(){
  static int j;
  //has access to i
}
```

file1.c

static storage duration,
file scope,
***internal* linkage**

static storage duration,
block scope,
no linkage

```c
void g(){
  //cannot access to i
}
```

file2.c

since other files cannot access i, the static keyword can help implement a technique known as information hiding (e.g., what the private keyword does in C++ or Java)

The `extern` Storage Class

- The extern storage class enables several source files to share the same variables.

- Remember that to share functions among files, we put the declarations of the function in one header file that everyone can include, and then have one definition of the function in one source file. Sharing variables among files can be done in the same way

- So far, we have not really differentiated between the declaration and definition of a variable. The definition of a variable is the point at which the compiler sets space aside for it

The `extern` Storage Class

Cont'd

```
int i = 10;

void f(){
  int j;
}
```

demo:
extern/
extern2/
extern-static/

The `extern` Storage Class

Cont'd

```
int i = 10;

void f(){
  int j;
}
```

declares i and defines it (i.e., allocates space for it)

Declares j and defines it (but does not initialize it)

demo:
extern/
extern2/
extern-static/

The `extern` Storage Class

Cont'd

```
int i = 10;

void f(){
  int j;
}
```

declares i and defines it (i.e., allocates space for it)

Declares j and defines it (but does not initialize it)

```
extern int i;

void g(){
}
```

demo:
extern/
extern2/
extern-static/

The `extern` Storage Class

Cont'd

```
int i = 10;

void f(){
  int j;
}
```

declares i and defines it (i.e., allocates space for it)

Declares j and defines it (but does not initialize it)

```
extern int i;

void g(){
}
```

declares i but does NOT define it
extern informs the compiler that i is defined elsewhere in the program (most likely a different file) so there's no need to allocate space for it. Global variables by default have external linkage.

demo:
extern/
extern2/
extern-static/

The `extern` Storage Class

Cont'd

```
int i = 10;

void f(){
  int j;
}
```

declares i and defines it (i.e., allocates space for it)

Declares j and defines it (but does not initialize it)

```
extern int i;

void g(){
}
```

declares i but does NOT define it
extern informs the compiler that i is defined elsewhere in the program (most likely a different file) so there's no need to allocate space for it. Global variables by default have external linkage.

extern declarations always have static storage duration. You can also declare an extern variable inside a function in order to link it to a definition outside the function

demo:
extern/
extern2/
extern-static/

The `register` Storage Class

- A register is a storage area located in a computer's CPU. Data stored in a register can be accessed and updated faster than data stored in ordinary memory
- By using the `register` keyword in a variable's declaration, you ask the compiler to store the variable in a register instead of keeping it in main memory like other variables
- The `register` keyword can only be used for variables declared in a block. The `register` keyword is a request so the compiler is free to store a register variable in memory if it choose to do so
- Since registers don't have memory addresses, it is illegal to use the & operator on register variables

Storage Class of a Function

- Functions have two storage class options: `static` and `extern`
- By default, functions have external linkage (can be called from other files)
- If the function is declared as `static`, then it has internal linkage, and can only be called in that file

Declarations Summary

```
int a;
extern int b;
static int c;

void f(int d, register
int e)
{
    auto int g;
    int h;
    static int i;
    extern int j;
    register int k;
}
```

Var.	Storage Duration	Scope	Linkage
a	Static	File	External
b	Static	File	*
c	Static	File	Internal
d	Automatic	Block	None
e	Automatic	Block	None
g	Automatic	Block	None
h	Automatic	Block	None
i	Static	Block	None
j	Static	Block	*
k	Automatic	Block	None

* In most cases, `b` and `j` will be defined in another file and will have external linkage.

If they are defined in the current file, they will have internal linkage

Copyright © 2008 W. W. Norton & Company.
All rights reserved.

Type Qualifiers

- `const`: creates a “read-only” variable
- `restrict`: only used with pointers. If a pointer p is declared with the restrict keyword, a promise is made that only p or derivations of p (e.g., p+1) will be used to access the object to which it points. This can help the compiler perform certain optimizations in the code.
- `volatile`: related to low-level programming with bit manipulation (will not cover in this course)

[nice example of restrict here: https://en.wikipedia.org/wiki/Restrict]

Deciphering Complex Declarations

- Always read declarators from inside out: locate the identifier that's being declared, and start deciphering from there
- When there's a choice, always favor [ ] and ( ) over *. If * precedes the identifier and [ ] follows it, the identifier represents an array, not a pointer. Likewise, if * precedes the identifier and ( ) follows it, the identifier represents a function, not a pointer. Of course, you can override any normal priority by using parentheses).

Examples of Complex Declarations

```
int *mystery[10];
```

Examples of Complex Declarations

```
int *mystery[10];
```

- The identifier is `mystery`

Examples of Complex Declarations

```
int *mystery[10];
```

- The identifier is `mystery`
- Since `*` precedes `mystery` and `[ ]` follows it, we give preference to `[ ]`, so `mystery` is *an array of pointers.*

Examples of Complex Declarations

```
int *mystery[10];
```

- The identifier is `mystery`
- Since `*` precedes `mystery` and `[ ]` follows it, we give preference to `[ ]`, so `mystery` is *an array of pointers.*
- What type of pointer? An integer

Examples of Complex Declarations

```
int *mystery[10];
```

- The identifier is `mystery`
- Since `*` precedes `mystery` and `[ ]` follows it, we give preference to `[ ]`, so `mystery` is *an array of pointers.*
- What type of pointer? An integer
- `mystery` is an *array of integer pointers*

Examples of Complex Declarations *Cont'd*

```
void (*mystery)(int);
```

Examples of Complex Declarations *Cont'd*

```
void (*mystery)(int);
```

- The identifier is `mystery`

Examples of Complex Declarations *Cont'd*

```
void (*mystery)(int);
```

- The identifier is `mystery`
- `mystery` is enclosed in parenthesis, which gives `*` precedence so we now know that `mystery` is a pointer

Examples of Complex Declarations *Cont'd*

```
void (*mystery)(int);
```

- The identifier is `mystery`
- `mystery` is enclosed in parenthesis, which gives `*` precedence so we now know that `mystery` is a pointer
- But `mystery` is followed by `(int)`, so the only possible explanation is that `mystery` is a pointer to a function that takes an `int` argument

Examples of Complex Declarations *Cont'd*

```
void (*mystery)(int);
```

- The identifier is `mystery`
- `mystery` is enclosed in parenthesis, which gives `*` precedence so we now know that `mystery` is a pointer
- But `mystery` is followed by `(int)`, so the only possible explanation is that `mystery` is a pointer to a function that takes an `int` argument
- Then this means that `void` is the return type of this function

Examples of Complex Declarations *Cont'd*

```
int (*mystery)[N];
```

Examples of Complex Declarations *Cont'd*

```
int (*mystery)[N];
```

- The identifier is `mystery`

Examples of Complex Declarations *Cont'd*

```
int (*mystery)[N];
```

- The identifier is `mystery`
- `mystery` is enclosed in parenthesis, which gives `*` precedence so we now know that `mystery` is a pointer

Examples of Complex Declarations *Cont'd*

```
int (*mystery)[N];
```

- The identifier is `mystery`
- `mystery` is enclosed in parenthesis, which gives `*` precedence so we now know that `mystery` is a pointer
- We next have to “process” the [ ] since they take precedence. So this means that mystery points to an array of N elements

Examples of Complex Declarations *Cont'd*

`int (*mystery)[N];`

- The identifier is `mystery`
- `mystery` is enclosed in parenthesis, which gives `*` precedence so we now know that `mystery` is a pointer
- We next have to "process" the [ ] since they take precedence. So this means that mystery points to an array of N elements
- Next we see the `int` so this means that the type of the array is `int`

Examples of Complex Declarations *Cont'd*

```
int (*mystery)[N];
```

- The identifier is `mystery`
- `mystery` is enclosed in parenthesis, which gives `*` precedence so we now know that `mystery` is a pointer
- We next have to "process" the [ ] since they take precedence. So this means that mystery points to an array of N elements
- Next we see the `int` so this means that the type of the array is `int`
- Thus, `mystery` is a *pointer to an array of N integers*