Precept 14
Week 8 Wed/Thu

Princeton University
COS 217: Introduction to Programming Systems
C Variable Declarations and Definitions

Variable **declaration** is a statement that informs the compiler of the name, type, scope, linkage, and duration of the variable. A variable **definition** is a declaration that causes the compiler to allocate memory.

Scope (compile-time concept)

File: The variable is accessible within the file in which it is declared, from the point of declaration to the end of the file.

Block: The variable is accessible within the block in which it is declared, from the point of declaration to the end of the block.

Linkage (link-time concept)

External: The variable is accessible from multiple files.

Internal: The variable is accessible from only the file in which it is declared.

Duration (run-time concept)

Temporary: The variable exists only during the execution of the function or block in which it is declared. Physically, the variable's value is stored in the runtime Stack.

Process: The variable exists throughout the entire process. Physically, the variable's value is stored in the Data Section (if the programmer specifies an initial value) or the BSS Section (if the programmer does not specify an initial value). The variable's value is initialized at program startup. If in the BSS section, its initial value is 0.

C Code	Decl/Def	Scope	Linkage	Duration	Location
`int a = 5;`	definition	file	external	process	
`int b;`	definition*	file	external	process	
~~`extern int c = 5;`~~	definition	file	external	process	
`extern int d;`	declaration	file	external	process	???
`static int e = 5;`	definition	file	internal	process	
`static int f;`	definition	file	internal	process	
`void fun(int g) {`	definition	block	internal	temporary	
`    int h = 5;`	definition	block	internal	temporary	
`    int i;`	definition	block	internal	temporary	
~~`    extern int j = 5;`~~	ILLEGAL				
~~`    extern int k;`~~	declaration	block	???	process	???
`    static int l = 5;`	definition	block	internal	process	Data
`    static int m;`	definition	block	internal	process	BSS
`    ...`					
`}`					

* Special rule: If a definition of `b` appears in another .c file, then this becomes a declaration.

2

Examples of Global Variable Declarations and Definitions

Suppose a program consists of file1.c and file2.c (only). Consider these combinations of global variable declarations and definitions:

	file1.c	file2.c	Result
	Reasonable combinations:		
1	`static int i = 5;`	`static int i = 5;`	`static def / static def => OK`
2	`static int i = 5;`	`static int i;`	`static def / static def => OK`
3	`static int i;`	`static int i;`	`static def / static def => OK`
4	`int i = 5;`	`extern int i;`	`def / decl => OK`
5	`int i;`	`extern int i;`	`def / decl => OK`
	Less reasonable combinations:		
6	`int i = 5;`	`int i;`	`def / decl => OK (by special rule)`
7	`int i;`	`int i;`	`def / decl => OK (by special rule)`
8	`int i = 5;`	`static int i = 5;`	`def / static def => OK`
9	`int i = 5;`	`static int i;`	`def / static def => OK`
10	`int i;`	`static int i = 5;`	`def / static def => OK`
11	`int i;`	`static int i;`	`def / static def => OK`
	Erroneous combinations:		
12	`int i = 5;`	`int i = 5;`	`def / def => error`
13	`extern int i;`	`extern int i;`	`decl / decl => error`
14	`extern int i;`	`static int i = 5;`	`decl / static def => error`
15	`extern int i;`	`static int i;`	`decl / static def => error`

3

Princeton University
COS 217: Introduction to Programming Systems
C Function Declarations and Definitions

Function **declaration**

A statement that informs the compiler of the function's signature (i.e., its name, return type, number of parameters, types of parameters) and linkage. Has no "body." Need not specify parameter names.

Function **definition**

A declaration that causes the compiler to generate executable code. Has a "body." Specifies parameter names.

C Code	Decl/Def	Linkage	Comment
int **f1**(int);	declaration	External	Common
extern int **f2**(int);	declaration	External	
static int **f3**(int);	declaration	Internal; f3 must be defined in same compilation unit	
int **f4**(int i) { ... }	definition	External	Common
extern int **f5**(int i) { ... }	definition	External	Rare
static int **f6**(int i) { ... }	definition	Internal	Common

Note: It is somewhat rare to use the `extern` keyword with a function declaration or definition, just because `extern` is the default.

4

Princeton University
COS 217: Introduction to Programming Systems
Complex C Declarations

The document shown below is retrieved from: https://cseweb.ucsd.edu/~gbournou/CSE131/rt_lt.rule.html. It describes the "right-left rule," a technique for understanding and composing complex C declarations.

The "right-left" rule is a completely regular rule for deciphering C declarations. It can also be useful in creating them.

First, symbols. Read

*	as "pointer to"	- always on the left side
[]	as "array of"	- always on the right side
()	as "function returning"	- always on the right side

as you encounter them in the declaration.

STEP 1

Find the identifier. This is your starting point. Then say to yourself, "identifier is." You've started your declaration.

STEP 2

Look at the symbols on the right of the identifier. If, say, you find "()" there, then you know that this is the declaration for a function. So you would then have "identifier is function returning". Or if you found a "[]" there, you would say "identifier is array of". Continue right until you run out of symbols *OR* hit a *right* parenthesis ")". (If you hit a left parenthesis, that's the beginning of a () symbol, even if there is stuff in between the parentheses. More on that below.)

STEP 3

Look at the symbols to the left of the identifier. If it is not one of our symbols above (say, something like "int"), just say it. Otherwise, translate it into English using that table above. Keep going left until you run out of symbols *OR* hit a *left* parenthesis "(".

Now repeat steps 2 and 3 until you've formed your declaration. Here are some examples:

```
     int *p[];

1) Find identifier.          int *p[];
                                  ^
   "p is"

2) Move right until out of symbols or left parenthesis hit.
                             int *p[];
                                   ^^
   "p is array of"

3) Can't move right anymore (out of symbols), so move left and find:
                             int *p[];
                                 ^
   "p is array of pointer to"

4) Keep going left and find:
                             int *p[];
                             ^^^
   "p is array of pointer to int".
```

(or "p is an array where each element is of type pointer to int")

Another example:

```
int *(*func())();
```

```
1) Find the identifier.      int *(*func())();
                                    ^^^^
   "func is"

2) Move right.               int *(*func())();
                                        ^^
   "func is function returning"

3) Can't move right anymore because of the right parenthesis, so move left.
                             int *(*func())();
                                   ^
   "func is function returning pointer to"

4) Can't move left anymore because of the left parenthesis, so keep going
   right.                    int *(*func())();
                                           ^^
   "func is function returning pointer to function returning"

5) Can't move right anymore because we're out of symbols, so go left.
                             int *(*func())();
                                 ^
   "func is function returning pointer to function returning pointer to"

6) And finally, keep going left, because there's nothing left on the right.
                             int *(*func())();
                             ^^^
   "func is function returning pointer to function returning pointer to int".
```

As you can see, this rule can be quite useful. You can also use it to sanity check yourself while you are creating declarations, and to give you a hint about where to put the next symbol and whether parentheses are required.

Some declarations look much more complicated than they are due to array sizes and argument lists in prototype form. If you see "[3]", that's read as "array (size 3) of...". If you see "(char *,int)" that's read as "function expecting (char *,int) and returning...". Here's a fun one:

```
int (*(*fun_one)(char *,double))[9][20];
```

I won't go through each of the steps to decipher this one.

Ok. It's:

"fun_one is pointer to function expecting (char *,double) and returning pointer to array (size 9) of array (size 20) of int."

As you can see, it's not as complicated if you get rid of the array sizes and argument lists:

```
int (*(*fun_one)())[][];
```

You can decipher it that way, and then put in the array sizes and argument lists later.

Some final words:

It is quite possible to make illegal declarations using this rule, so some knowledge of what's legal in C is necessary. For instance, if the above had been:

```
int *((*fun_one)())[][];
```

```
it would have been "fun_one is pointer to function returning array of array of
                                          ^^^^^^^^^^^^^^^^^^^^^^^^
pointer to int".  Since a function cannot return an array, but only a
pointer to an array, that declaration is illegal.
```

Illegal combinations include:

```
        []() - cannot have an array of functions
        ()() - cannot have a function that returns a function
        ()[] - cannot have a function that returns an array
```

In all the above cases, you would need a set of parens to bind a * symbol on the left between these () and [] right-side symbols in order for the declaration to be legal.

Here are some legal and illegal examples:

int i;	an int
int *p;	an int pointer (ptr to an int)
int a[];	an array of ints
int f();	a function returning an int
int **pp;	a pointer to an int pointer (ptr to a ptr to an int)
int (*pa)[];	a pointer to an array of ints
int (*pf)();	a pointer to a function returning an int
int *ap[];	an array of int pointers (array of ptrs to ints)
int aa[][];	an array of arrays of ints
int af[]();	an array of functions returning an int (ILLEGAL)
int *fp();	a function returning an int pointer
int fa()[];	a function returning an array of ints (ILLEGAL)
int ff()();	a function returning a function returning an int (ILLEGAL)
int ***ppp;	a pointer to a pointer to an int pointer
int (**ppa)[];	a pointer to a pointer to an array of ints
int (**ppf)();	a pointer to a pointer to a function returning an int
int *(*pap)[];	a pointer to an array of int pointers
int (*paa)[][];	a pointer to an array of arrays of ints
int (*paf)[]();	a pointer to a an array of functions returning an int (ILLEGAL)
int *(*pfp)();	a pointer to a function returning an int pointer
int (*pfa)()[];	a pointer to a function returning an array of ints (ILLEGAL)
int (*pff)()();	a pointer to a function returning a function returning an int (ILLEGAL)
int **app[];	an array of pointers to int pointers
int (*apa[])[];	an array of pointers to arrays of ints
int (*apf[])();	an array of pointers to functions returning an int
int *aap[][];	an array of arrays of int pointers
int aaa[][][];	an array of arrays of arrays of ints
int aaf[][]();	an array of arrays of functions returning an int (ILLEGAL)
int *afp[]();	an array of functions returning int pointers (ILLEGAL)
int afa[]()[];	an array of functions returning an array of ints (ILLEGAL)
int aff[]()();	an array of functions returning functions returning an int (ILLEGAL)
int **fpp();	a function returning a pointer to an int pointer
int (*fpa())[];	a function returning a pointer to an array of ints
int (*fpf())();	a function returning a pointer to a function returning an int
int *fap()[];	a function returning an array of int pointers (ILLEGAL)
int faa()[][];	a function returning an array of arrays of ints (ILLEGAL)
int faf()[]();	a function returning an array of functions returning an int (ILLEGAL)
int *ffp()();	a function returning a function returning an int pointer (ILLEGAL)

Review Activity Instructions:

Part 1: Determine the memory section (BSS, DATA or STACK) in the Location column of the table for the boldface variables **a, b, e, f, g, h** and **i** in each line on Page 1 of the handouts.

Part 2: Interpret each of the following using the right-left rule:

```
int x[5];
int x(int);
int *x();
int *x[];
int (*x)();
```

Part 3: Interpret each of the following using the `cdecl` command:

```
int x[5];
int x(int);
int *x();
int *x[];
int (*x)();
```

Review Activity Answers:

Part 1: Determine the memory section (BSS, DATA or STACK) in the Location column of the table for the boldface variables **a, b, e, f, g, h** and **i** in each line on Page 1 of the handouts.

int **a** = 5;	**DATA** (a is an explicitly initialized global variable)
int **b**;	**BSS** (b is an uninitialized global variable)
static int **e** = 5;	**DATA** (e is an explicitly initialized global variable)
static int **f**;	**BSS** (f is an uninitialized global variable)
void fun(int **g**){	**STACK** (g is a function argument)
int **h**;	**STACK** (h is a local variable)
int **i**;	**STACK** (i is a local variable)
}	

Part 2: Interpret each of the following using the right-left rule:

int x[5];	**x is an array of 5 int**
int x(int);	**x is function accepting an int and returning an int**
int *x();	**x is a function returning a pointer to an int**
int *x[];	**x is an array of pointer to int**
int (*x)();	**x is a pointer to a function returning int**

Part 3: Interpret each of the following using the cdecl command:

int x[5];	**declare x as array 5 of int**
int x(int);	**declare x as function (int) returning int**
int *x();	**declare x as a function returning pointer to int**
int *x[];	**declare x as array of pointer to int**
int (*x)();	**declare x as a pointer to a function returning int**