LECTURE 15: SCOPE REVIEW, PREPROCESSOR, RECURSION (K&R Ch. 4)

REVIEW

scope	where a variable can be seen
lifetime	the period over which the variable exists

Automatic Variable
Declared inside a function (between { and }).
Its scope is the function where it was declared (it is only accessible within the braces).
It cannot be referenced from outside the braces.
It appears on the stack when the function is called and disappears when the function returns.
Its initial value is undefined (i.e., garbage) unless explicitly initialized in the source code.
Its value does not persist between function calls.

Static Variable
Declared inside the braces of a function (with the keyword static).
Its scope is the function where it was declared (it is only accessible within the braces).
It cannot be referenced from outside the braces.
It does NOT appear on the stack (stored in data segment).
It is guaranteed to be initialized to zero.
It retains its value across function invocations.

External Variable	
Declared outside the braces of any function in the file.	
Its scope is any function in the file (i.e., it is a global variable).	
It is private to the file in which it is declared unless you declare the same variable name with type "extern" in another file.	Example: file1.h contains: extern int global_var; file2.c contains: int global_var = 54; file3.c contains: printf("%d", global_var)
It does NOT appear on the stack (stored in data segment).	
It is guaranteed to be initialized to zero.	
It retains its value across function invocations.	
Avoid global variables if possible!	

External Static Variable
Declared outside the braces of any function (with the keyword static).
Its scope is any function in the file (i.e., it is a global variable, but only for this file).
It is private to the file and cannot be referenced from another file (not even by using extern).
It does NOT appear on the stack (stored in data segment).
It is guaranteed to be initialized to zero.
It retains its value across function invocations.
Safer than a global variable.

REGISTER VARIABLES

Declaring a variable as a "register" variable is an advisory to the compiler to keep the normal location of the variable in a register, if possible.	
Gives extra speed for frequently modified variables.	
There are a limited number of registers in machine (where arithmetic is usually performed):	load register from memory location of variable, perform increment/decrement, store value back.
Undefined (i.e. garbage) value unless explicitly initialized in the source code	

```
void f(int n) {
  int i;
  for(...; ...; i++, n--)

/* during each pass */
LD    r1, &i
INC   r1
ST    r1, &i
LD    r1, &n
DEC   r1
ST    r1, &n
```

```
void f(register int n) {
  register int i;
  for(...; ...; i++, n--)

/* during each pass */
INC   r1   (i held in r1)
DEC   r2   (n held in r2)
```

register declarations are advisory only: the number of registers is quite limited. Still, it usually works.
static variables cannot be register type (this would lock that register to this one variable for the life of the program).
automatic variables and function parameters (which are treated like local variables) can be register variables.
a pointer points to a position in memory; therefore, we cannot use the address of operator with a register variable (registers are not part of main memory and do not have addresses).

C PREPROCESSOR

Recall that preprocessor directives start with #.

The preprocessor handles:

1. Inclusion of other files -- usually .h files:
   `#include "filename.h"`
2. Simple macro substitution (find/replace):
   `#define name substitute text`
3. Macro substitution with arguments (enclose in ()s):
   `#define square(A) ((A)*(A))`
   `n = square(x) + 1;` becomes `n = ((x)*(x)) + 1;`
4. Conditional inclusion

MACRO SUBSTITUTION

Macros do not understand C expressions.
They are only doing precise character substitution.

WRONG!!!:

```
#define square(A) A * A
n = square(p + 1);  becomes  n = p + 1 * p + 1;
```
You wanted : $p^2 + 2p + 1$
You got : $2p + 1$

Macros must be defined on a single line, but you can continue a long definition to the next line using \ :

```
#define exchg(t, x, y) {t d; d = x; x = y; \
y = d;}
```

Using above macro, exchg(char, u, v) produces the following (show one statement per line for clarity):

```
{
char d;  /* Note: d is defined locally within block */
d = u;
u = v;
v = d;
}
```

Recall: function calls are call by value (calling program's argument values cannot be modified inside the function unless pointers are passed).
This is not true for macros.
Why? Because statements within a macro expansion act like in-line code.

A frequently used macro may take more memory than a function, but does not require call/return and stack frames! (Macro will usually execute faster).

Substitutions are not done within quotation marks.
Can use the # character to get around this:

WRONG:

```
#define dprint(expr) printf("expr  = %f\n", expr)
[...]  /* say x / y == 17.2 */
dprint(x/y);
You wanted : printf("x/y = %f\n", x/y);
You got    : printf("expr = %f\n", x/y);
```

CORRECT:

```
#define dprint(expr) printf(#expr  " = %g\n", expr)
```

The special form #expr means:
  -do substitution for the macro "expr"
  -put quotes around result

now dprint(x/y); expands as:

```
printf("x/y" " = %f\n", x/y); /* 2 strings concatenated */
```

CONDITIONAL INCLUSION

Allows us to control when precompiler directives such as #define or #include are executed.
Done before compilation using conditionals that are meaningful at that time.
Conditionals work for any C statements in their scope, and can be used to drop unneeded code (and save memory space) under some conditions.
The special operator defined is used in #if expressions to test whether a certain name is defined as a macro.

#if	with conditions, such as !defined(SYM)
#ifdef SYM	like saying #if defined(SYM)
#ifndef SYM	like saying #if !defined(SYM)
#elif	like else if
#else	like else
#endif	end scope of originating #if

Applications:

A software release might need different .h header files included for different OSs.

```
Before main() we might define:
  #define SYSV 100
  #define BSD 101
For a specific system (say SYSV) we write:
  #define SYSTEM SYSV
Now we can define .h file symbolic constants conditionally:
#if SYSTEM = = SYSV
  #define HDR "sysv.h"
  #elif SYSTEM = = BSD
  #define HDR "bsd.h"
  #else
  #define HDR "default.h"
  #endif
  #include HDR
```

Some .h files include other headers recursively. As a result, it might happen that we include two header files that both recursively include the same .h file.

This can lead to compiler warnings/errors if a function prototype appears twice during a compilation.

We can use conditionals inside a header to prevent this (in the example, our header is called xxx.h):

```
#ifndef XXX_HDR
#define XXX_HDR  // note we don't need to give a value
                 // here next time XXX_HDR will be defined
                 // so #ifndef will fail
// ... contents of .h file go here ...
#endif      /* XXX_HDR */
// this is known as an include guard
```

RECURSION

Any C function may call itself recursively, either directly or after intermediate function calls.

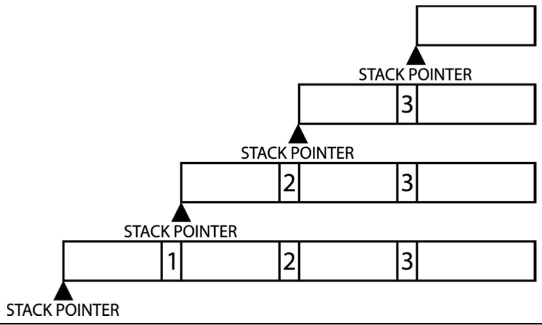
Each time a call is made:	a new frame is placed on the stack
	it contains passed arguments, a position to return to, and automatic variables (if any)

It is up to the programmer to make sure the recursion terminates -- otherwise you get stack overflow!

Example: stack during recursion.

```
int factorial (int n){
  return (n  > 1) ?  n * factorial (n - 1) : 1;
}

int result = factorial(3);
```



recursion performance
Time relative to while/for loops -Calling/creating stack frame takes a lot of time -Returning/removing stack frame costs time, too
Memory relative to while/for loops -Stack frames eat up memory → need large space! -In non-virtual memory system → stack overflow?
Recursion is rarely used in hard real-time and/or embedded systems for these reasons