LECTURE 9: SCOPE (K&R §§ 4.3 – 4.9)

	local automatic	local static	external	external static
description	a variable declared within a block	similar to local automatic, but is preserved in memory	declared ahead of/outside all {}	static variables declared ahead of/outside all {}
available	within the block where it is created	within the block where it is created	to all functions within the program load	to functions within this file only
duration	comes into existence between the {} and disappears once return is performed	preserved in memory	preserved like local static	preserved like local static
default value	garbage (i.e., whatever was in that memory position before)	0	0	0

DEFINITIONS

block	A section of code that is grouped together.
	In C, blocks are delimited by curly braces.
	{ [block statements] }
variable	A name used to refer to some location in memory that holds a value we want to work with.
scope	the area of a program where a variable can be referenced.

INTERNAL VARIABLES (LOCAL AUTOMATIC)

Arguments and variables defined inside functions are internal.

They are local to the block where they are defined.

Internal variables come into existence when the function is entered and disappear when it is left.

These variables are said to be automatic.

scope	The block where the variable was declared.
initialized to	Undefined (i.e., garbage) value unless explicitly initialized in the source code.
initialization happens	Each time the function or block is entered.

The parameters of a function are in effect local variables.

INTERNAL VARIABLES (LOCAL STATIC)

Local alternative to automatic.

Static variables declared inside a function are preserved in memory.

scope	The block where the variable was declared.
initialized to	0 Also can be explicitly initialized otherwise, in which case the initializer must be a constant expression
initialization happens	If initialized, it is done once, before the program starts execution.
Note:	Function may behave differently when it is called with different values preserved in local static variables. Makes it harder to test a function because you need to test with all possible values of local static variables.

EXTERNAL VARIABLES (GLOBAL)

Variables or functions defined outside of functions are external.

External variables are globally accessible.

Values are preserved like static local variables.

Can be used to pass data between functions.

scope	From the point at which it is declared to the end of the file being compiled.
initialized to	0 Also can be explicitly initialized otherwise, in which case the initializer must be a constant expression
initialization happens	If initialized, it is done once, before the program starts execution.
Note:	We avoid the use of external variables for a couple of reasons: They can be accessed from anywhere: If their value gets corrupted, hard to trace how. They make functions dependent on their external environment.

Software architecture/design standards often prohibit use of external variables.

EXTERNAL VARIABLES (EXTERNAL STATIC)

The static declaration, applied to an external variable or function, limits the scope of that object to the source file being compiled.

Otherwise, same characteristics as normal external variables.

extern KEYWORD

extern declaration	Necessary if an external variable is to be referred to before it is defined, or if it is defined in another source file.

definition vs declaration outside of any function:

int sp;
double val[MAXVAL];
	extern int sp;
	extern double val[];

Define external variables sp and val:	Declare that sp is an int and val is an array of doubles:
cause storage to be set aside serve as declaration for rest of the source file	No storage is reserved.

register VARIABLES

A register declaration advises the compiler that this variable will be heavily used.

We want it placed in a machine register, but the compiler is free to ignore this suggestion if it needs registers.

Can only be applied to automatic variables.

LECTURE 9: INTRODUCTION TO POINTERS: POINTERS AS ARGUMENTS

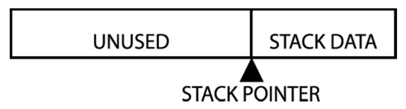
CALL BY VALUE vs CALL BY REFERENCE

Recall that simple variables passed as function parameters in C are actually only copies on the stack.

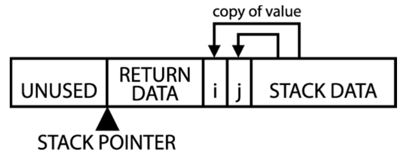
```
foo(i, j);

void foo(int i, int j)
{
}
```

before call / after return



after call / before return



Note:	Stack pointer is a register.

This is call by value.

You can't change arguments in original location within the function; you can only change the stack copy.

To make changes to original location, you must pass pointers to original variables.

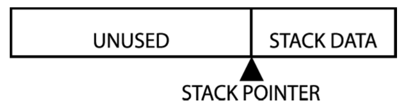
POINTERS AS ARGUMENTS

Pointer variables passed as function parameters in C are still stack copies, but they can be used to access original location indirectly.

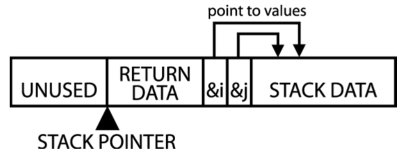
```
foo(&i, &j);

void foo(int *i, int *j)
{
}
```

before call / after return



after call / before return



EXAMPLE: SWAPPING VALUES

You want to write a function that will swap the values of two int variables, a and b.

```
void exchgint (int a, int b) {
  int dummy;
  dummy = a;
  a = b;
  b = dummy;
}
```

This DOES NOT WORK!

Why?
Because you are only changing the values of the copies of a and b.

You need pointers to swap the values of the variables at the original location:

```
void exchgint (int *pa, int *pb) {
  int dummy;
  dummy = *pa;
  *pa = *pb;
  *pb = dummy;
}
```

Now you are passing the memory location of the variables a and b and can change them.

We'll go over pointers in more detail, but start thinking about this distinction.

ARRAY NAMES

An array name is automatically passed as a pointer.

With arrays you do not need to create a pointer yourself with the "address of" operator (&).	`int array1[10], array2[10];`
	`foo(array1, array2);`

unless you are passing a pointer to a specific array element (recall our printf statement in visitype).	`foo(&array1[4*i],&array2[4*j]);`