

`n = n + 10;` Shorthand expression like `n++`. Note you wrote a reverse function yourself.

Example, with int of -253, generate '3', '5', '2', '-', '\0'. Then reverse.

Note function `itoa` is not in UNIX Library, function `atoi` on the other hand IS in the UNIX Library. To achieve `itoa`, would use `sprintfL`

```
sprintf(s, "%d", n)    /* char s[ ], int n                * /
```

Daily Course Notes for CS240

Computer Architecture 1

Patrick E. O'Neil

Class 8.

Give QUIZ 1. Exam 1 on Class 14 through end of Chap 4: Open Book/Notes.
Hw 2 due next class. Hw 3 is or soon will be on-line.
Finishing Chapter 2 of K&R. Read K&R through Chapter 4.2.
We will go through Chapter 3 very quickly. Not a lot is new.

Any more questions from prior lecture?
Lots of little tag-ends now. Try not to tune out!

How to convert Uppercase chars to lowercase. VERY USEFUL.

```
int lower(int c) /* this is function tolower(int c) if
                  #include <ctype.h> (see K&R, App. B2) */
{
    if (c >= 'A' && c <= 'Z')
        return c + 'a' - 'A'; /* c - 'A' is letter number... */
    /* ...where 'A' = 0, so c - 'A' + 'a' is corresponding
       lower letter */
    else
        return c;
}
```

Idea of conversion, casts K&R Sect 2.7.

```
char c1, c2 = 15;
int j = 2379;
float g = 12.1;
```

```
printf ("%d\n", c2 + j); /* what is type, what printed out? */
                        ( int, 15 + 2379 = 2394 )
c1 = j;                /* value of c1? */ ( 2379 % 256 = 75 )
printf ("%d\n", c1 + c2); /* value printed out? */ ( 90 )
printf ("%d\n", c2 + (char) j); /* value? */ ( 90 )
printf ("%9.1f\n", j + g); /* type, value? */
                        ( float, 2379. + 12.1 = 2391.1 )
printf ("%d\n", j + (int) g); /* type, value? */
                        ( int, 2379 + 12 = 2391 )
```

Declarations and initialization. Section 2.4.

```
int x, a[20];           /* external vars, initialized to zero */
int w = 37, v[3] = {1, 2, 3}; /* happens once */

main ( )
{
    func ( )             /* entered 1000 times per second */
    {
        int y, b[20];    /* automatic -- contains junk on entry */
        int s = 37, t[3] = {1, 2, 3}; /* happens on each entry */
    }
}
```

The C language never does a possibly significant amount of work without this being specified by the programmer — unlike Java. The `main()` function is treated just as `func()` is (could call `main` recursively).

Increment and decrement operators. Section 2.8.

`++n` and `--n` vs. `n++` and `n--`.

Both `++n` and `n++` have effect of `n = n + 1`; Except that evaluating `++n` as an expression, `n` is incremented BEFORE `n` is evaluated, and in `n++`, value is returned THEN `n` is incremented.

Example.

```
int arr[4] = {1, 2, 3, 4}, n = 2;

printf("%d\n", arr[n++]); /* values printed? */
printf("%d\n", arr[--n]);
printf("%d\n", arr[++n]);
```

Assignment operators. Section 2.10.

Just as `i++` means `i = i + 1`; we have `i += 3` means `i = i + 3`;
We can also use other operators.

```
int i = 3;

i += 3;      /* value now? */ ( 6 )
i <=<= 2;    /* value now? */ ( 24 )
i |= 0x02;   /* value now? */ (24 = 0x18. 0x18 | 0x02 = 0x1a)
```

Homework Exercise 2-9. (Cover Lightly)

Say why `x &= (x-1)` sets the rightmost 1-bit to 0: see below!. Use to write faster version of `bitcount`. Example:

`char x = 0xa4` has bits 10100100

```
x      10100100
x-1=   10100011
&      10100000  New value for x
```

Do it again.

```
x=      10100000
x-1=    10011111
&       10000000
```

Again.

```
x=      10000000
x-1=    01111111
&       00000000  Bits all counted.
```

Conditional Expressions. Section 2.11. Example of `z = max(a, b)`

```
if (a > b) \
  z = a;   | can be written: z = (a > b)? a: b; (see pg 53)
else      / (Same as Java!)
  z = b;  / (NESTS: can also write: z = (a > b)? a: ((x < y)? b: 0)
```

Can always do this as long as alternatives only set one variable value.

Note that precedence, pg. 53, specifies the binding order, not the temporal order. Consider following two statements:

```
n = 5;
m = n-- + 7;
```

In second statement, grouping is: `m = ((n--) + 7);`

The value of `n--` is evaluated first, but we still take value 5 for `n`. The value of `n` is not decremented to 4 until the WHOLE EXPRESSION has been evaluated (in this case, the whole assignment statement). Temporal order of this kind is not specified by binding of operators in expression.

In fact (experiment), hard to predict sometimes. With the following code fragment:

```
n = 3;  
n = (n++)*(n++);
```

Do we get $3*3+1+1 = 11$? It might be different on different machines, so **don't do this!**

With "gcc" we get result 4! With gcc -O (optimize) we get 11!! (Optimized compilation keeps result in one register and increments twice at end.)

printf ("%d %d\n", ++n, n); ??? Unclear what is printed out. Which expression is evaluated first?

Chapter 3 next time. Very quick.

Class 9.

Read Chapter 4 for next time. Exam 1 on class 14 through end of Chap 4;
Make-Up policy. Go over QUIZ 1 answers. Turn in hw 2. Should have started
hw 3 by now.

Here is an example of If-Else, section 3.2. You can study this, I will not
cover anything in class with a bar on the left. ****Just like JAVA****

Example. If a number *n* is divisible by both 2 and 3, set the flag bit 0x08
in the char flag variable *v*. Otherwise, if it is not divisible by 5, set the *v*
bit 0x10.

```
n == 6 -- v = 0000 1000
n == 9 -- v = 0001 0000
n == 10 - v = 0000 0000
n == 30 - v = 0000 1000
```

Easiest way is:

```
int n;
unsigned char v;

if (n % 2 == 0 && n % 3 == 0)
    v |= 0x08;
else if (n % 5 != 0)
    v |= 0x10;
```

Is it appropriate to use else here? (Yes.) NEED else since second condition
and first overlap. If left out else, what would be different? (*n* == 6)

Now consider:

```
if (n % 2 == 0)
    if (n % 3 == 0)
        v |= 0x08;
else /* What is true here? (n divisible by 2, not by 3) */
    (else goes with second if – indentation of else is wrong)
```

Now this.

```
if (n % 2 == 0) {
    if (n % 3 == 0)
        v |= 0x08;
}
else /* What is true here? */ (n not divisible by 2)
```

Consider:

```
if (C1)
    ST1
else if (C2)
    ST2
.
.
else if (Ck)
    STk
else STk+1
```

Note. Could be that C1 and C2 and C3 intersect as conditions. (Draw Venn diagram) But only ONE of the ST1 . . STk+1 ever get executed.

Else-If. Sections 3.3. Consider the two constructions:

```
int func1( int n)      int func2( int n)
{                      {
    if ( n % 2 == 0)      if ( n % 2 == 0)
        n /= 2;          n /= 2;
    if ( n % 6 == 0)      else if ( n % 6 == 0)
        n /= 3;          n /= 3;
    return n;             return n;
}                      }
```

What's the difference between the two? What is func1(12)? Answer: 2
What is func2(12)? Answer: 6 Work them through.

Switch. Section 3.4. Consider cascading else-if sequence:

```
if (i == 1)          /* NOTE:  == */
    statement-1
else if (i == 2)
    statement-2
.
.
else if (i == 49)
    statement-49
else
    statement-50;    /* The default of "catch-all" */
```

This is a standard way of choosing between a set of alternatives. But also have switch statement in special limited situations. ****LIKE JAVA****

```

switch (i) {
    case 1:  statement-1
        break;
    case 2:  statement-2
        break;
    .
    .
    case 49: statement-49
        break;
    default: statement-50;
}

```

The way the if-else cascade works is to test if $i == 1$, then if that fails if $i == 2$, When we test $i == 27$, we have performed 26 prior tests.

With the switch statement, we achieve the same effect, but probably faster: usually compiled into assembly language as a jump table, an array of "go to" instructions subscripted by the value of i , so that if $i = 27$ we "look up" the goto at address 27 (only) and go execute that goto.

Else-if can be more general, not just simple tests whether a variable is equal to a constant.

```

int axtoi (char s[ ])
{
    int i, n, flag;

    n = 0;  flag = 1;
    for ( i = 0; flag; i++)
        switch (s[i]) {
            case '0':case '1':case '2':case '3':case '4': /* fall... */
            case '5':case '6':case '7':case '8':case '9': /* through */
                n = 16*n + (s[i] - '0');
                break; /* need this so don't fall through */
            case 'a':case 'b':case 'c':case 'd':case 'e':
            case 'f':
                n = 16*n + (s[i] - 'a' + 10);
                break; /* could also have cases for A...F now */
            default:
                flag = 0; /* alternatively, goto ret; ... */
                break; /* then don't need flag */
        }
    ret:  return n;
}

```

Note the need for a break statement. The default action is to cascade down to the next case, and we must explicitly say when to stop cascading and leave the switch. (In ***JAVA*** have "labeled breaks" to goto ret)

Break statement works for: for loop, while loop, do loop and switch. Brings you to end of loop or switch statement ONE LEVEL ONLY. Use goto to break out TWO OR MORE levels. IT IS OK TO USE GOTO as long as always do it in a FORWARD direction. (Seen this above, then don't need flag.)

Break, continue, goto and labels. Section 3.7 & 3.8.

Continue Statement. Section 3.7. (Same as ***JAVA***)

The continue statement is like a break, except it causes next pass of loop to start (break causes whole loop to end). Routine to bring Upper Case letters in s[] to lower case.

```
void lower (char s[ ])
{   int i;

    for ( i = 0; s[i] != 0; i++) {
        if (s[i] < 'A' || s[i] > 'Z')
            continue;
        s[i] = s[i] - 'A' + 'a';
    }
}
```

There are easier ways to do this, as in K&R example: if (is Upper Char) something, rather than if not continue. Or could use if (TRUE) something, else all other cases.

When would continue be a real use? When we are in the middle of a long set of tests, lot of continues possible (not easy to give such an example).

For loop. Section 3.5. (Same as *** JAVA***)

Note any part of for loop can be left out. for(init; loop-test; increment) If init or increment expression is left out, just not evaluated (program must be initializing, incrementing by other means). If leave out loop-test, assumed permanently true and loops forever. Must break or goto to end.

Comma operator: "," most often used in for (NOT in JAVA). Pair of expressions separated by "," are evaluated left-to-right and type/value of expression is type value of result. See P53, precedence.

```

#include <string.h>      /* need this header file          */
/* reverse:  reverse string of characters s[ ] in place    */
void reverse ( char s[ ])
{
    char c, i, j; /* Single expressions with commas      */
    for ( i = 0, j = strlen(s) - 1; i < j; i++, j--) {
        c = s[i]; s[i] = s[j]; s[j] = c;
        /* Note can write many statements per line */
    }
}

```

char t[] = "Hello, world";
reverse (Show how cursors move with this string. If instead of $i < j$, have $j \geq 0$, what does this do? (brings chars back into place).

Next example is just something you should know (if-else form not such a big deal). Homework exercise on this.

```

/* binsearch:  find x in v[0] <= v[1] <= . . . <= v[n-1]    */
int binsearch ( int x, int v[ ], int n) /* returns subscript */
{
    int low, high, mid;  (Would be int[] v in ***JAVA***)

    low = 0;
    high = n - 1;
    while ( low <= high) {
        mid = (low + high)/2;
        if (x < v[mid])
            high = mid - 1;
        else if (x > v[mid])
            low = mid + 1;
        else /* found match */
            return mid;
    }
    return -1; /* no match */
}

```

Idea is same as 20 questions. Think of a number from 1 to 1000 (or 1 to 1024). (Have someone do this and ask what number is. Now show how to guess it.) Is it > 512? Next: > 512 + or - 256? Only ever ask one question.

Next go through binsearch with $n = 11$. $v[] = \{ 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23 \}$. Try with $x = 11$. Write it all out step by step.

low = 0, high = 10, mid = 5. $11 < v[5] = 13$, so high = 4.
low = 0, high = 4, mid = 2, $11 \leq 7$, $11 > 7$, so low = 3, So on.

What if had chosen $x = 10$? All the same except now fall out of loop, return -1.

Note often ask second question, but never asked second question in 20 questions. If we were charged money for the time spent to answer each question, we'd take a different approach.

E.g., we were doing this in the code above: if ($x < \text{mid}$) do 1 else if ($x > \text{mid}$) do 2, else return; /* on return we know $x == \text{mid}$ */

Clearly not efficient -- we can do better. This is on the homework.

Class 10.

Exam 1 Class 14 through end of Chapter 4. (Hand out practice exam.) Bit harder Make-Up Exam. Both Exams returned. Then WITHDRAWAL deadline. BOOKS BEING RETURNED FROM BOOKSTORES SOON. LAST CHANCE. Starting Chapter 4 for assignment due soon.

Sections 4.1 - 4.5. A program is a set of definitions of variables and functions. The functions can occur in any order, and the source program can be split into multiple source files.

We will be writing a program to input a stream of characters from stdin, and parse the stream as an input to a reverse polish calculator:

123 21 + 567 432 - * (Means: (123. + 21.) * (567. - 432.))

and then perform the calculations on a stack, leaving the final answer as a single float number on the stack. (Book assumes double; our assignment changes to int. We won't need to deal with negative numbers: 2 - 6 won't happen, 6 - 2 will be OK.)

We use the following files containing functions. (leave following non-indented lines up on board).

main.c: main() -- does the calculation. Uses getop, push, pop, atof. Pg 76.

```
type = getop(s) - 123, type == NUMBER (i.e., 0): push(atof(s))
type = getop(s) - 21, type == NUMBER (i.e., 0): push(atof(s))
type = getop(s) - + , type == '+': push( pop( ) + pop( ))
```

getop.c: int getop(char s[]) — trivial parsing (returns NUMBER (value 0) or char value, such as '+'), uses getch, ungetch.

stack.c: void push(double f) and double pop(void). Use nothing

getch.c: int getch(void) and void ungetch(int). Use nothing.

To compile the program, use form:

```
gcc main.c getop.c stack.c getch.c -o calcit
```

TWO PHASES. Phase 1, Compilation of multiple files. Each compilation knows only what it finds in each source file (with header files included).

After Phase 1, compilation, you used to see a set of what are called "relocatable object" files, main.o, getop.o, stack.o, getch.o (standard suffix .o), result of compilation; now these get cleaned up.

A relocatable object file is one where the compilation has been done (they are binary files in machine language), but they still have not been loaded together into an "executable" file, calcit (an executable file can simply be entered and flow of control will progress).

Note, we will see shortly how we can still generate the .o files.

Phase 2, Load/Link. The relocatable object files have hooks which allow them to be "loaded/linked" together at arbitrary positions in memory, and so that a function call from one file will be found, located in another file. (Demonstrate this.)

If we had to correct a bug in ONE of these files, getop.c, would not have to recompile everything. Just Recompile the single file.

```
gcc -c getop.c /* compile, generate getop.o file only, no load phase */
```

If you didn't use -c, would try to Load as well and fail because no main(). Now the load/link step can be done explicitly, by writing:

```
gcc main.o getop.o stack.o getch.o -o calcit /* load all together */
```

A shorthand for the single file compilation and load/link is:

```
gcc main.o getop.c stack.o getch.o -o calcit /* same as above 2 */
```

There is something called a "makefile" in the hw4 assignment directory. If you type the command "make", the makefile will execute a set of rules so that any source file, such as getop.c, which has been modified since it was last compiled, will be recompiled, and calcit will then be reloaded.

Makefiles are particularly useful for large projects, with more than one implementor. You are responsible for understanding makefile syntax. See comments of file as well and make sure you can follow what's happening!

A makefile has a set of "rules" (**Glass & Ables** Ch. 11) of the form:

```
<target>: <dependency list of files used in creating target>
          <command to use to achieve target>
```

The <command> MUST be preceded by a TAB! It won't work otherwise. The <target> is normally a file, but not always.

When you type "make", the rules of the makefile are considered. If a <target> *file* is older than one of its dependency list of files, or *nonexistent*, the rule is triggered and the command is issued automatically.

This continues until all files have ages that do not trigger rules. For example, in the makefile of the hw4 assignment, we have four rules like:

```
getop.o: getop.c calc.h /* recompile if change header file */
          $(CC) -g -c getop.c /* $(CC) stands for gcc compile */
```

CC is defined early in the makefile with the line:

```
CC = gcc
```

Then using \$(CC) causes substitution. There's also a rule to perform the load:

```
calcit: main.o getop.o stack.o getch.o
          $(CC) main.o getop.o stack.o getch.o -o calcit
```

The user can type the command "make", or make with a named target. "make getop.o", for example, to cause the rules leading up to the target getop.o to be executed, if needed.

When the user types "make" without a target name, the FIRST rule listed will be the default target constructed. Thus the "calcit" rule should be the first rule listed.

Draw a tree to show how makefile views dependencies of time changes. calcit above main.o & getop.o & etc., main.o above main.c & calc.h.

At the end of the makefile, there is a rule with a target name "clean".

```
clean:
          rm *.o
```

Since "clean" is not a file, there is no dependency list (ages of files don't matter); this rule is only called when you give the command "make clean". It deletes any files you want that will reduce the clutter in the directory.

Communication between files.

Recall that a program is a set of definitions of variables and functions, possibly in several files. However, the normal scope of a function or external variable definition lasts from the point of declaration to the end of that file.

In order to have an external variable defined in one file known to logic in another (foreign) file, it is necessary to use a special declaration in the foreign file, an "extern" declaration.

In file 1 (where m is defined as a variable external to any function):

```
int m; (like an "eye" of a "hook and eye")
```

In file 2, an extern declaration:

```
extern int m; (like a "hook" of a "hook and eye")
```

This extern declaration tells the compiler and link/loader to look for the name m of int type in a different file. Names like this are hooks in compiled object files. If the extern declaration were not present, other file declares would be invisible and compilation would announce that any use of m, without a source-file local declaration, was illegal.

NOTE THAT AN EXTERNAL (GLOBAL) VARIABLE IS simply a variable whose definition lies outside any function -- does NOT have to have "extern" first (in fact such a declaration would not define the variable).

Note that extern definition can be inside a function. However cannot refer to a variable defined inside a different function (scope is function only).

The same sort of thing occurs with functions. In one file, the function is defined, in another file the "functional prototype" is given to make it clear what the return type and argument types are (arguments of different types in a function call will be converted as if casted). (Also need functional prototype to describe functions coming later in same file.)

There is a difference though, in that the functional connection will be made between files even if no prototype is used.

The compiler will make an implicit assumption when it first encounters a function use — the compiler will assume a return type of int, and there will be no argument conversion.

Thus, for example, the getop.c file must have functional prototypes for getch() and ungetch(), and main.c needs prototypes for push(), pop(), and getop(). Here is the file getch.c as it should be coded.

```
#include <ctype.h> /* OK to use C library functions, like
isdigit() IN THIS ASSIGNMENT ONLY!! */

int getch(void);    /* functional prototypes */
void ungetch(int);

/* getop: get next operator or numeric operand */
int getop(char s[ ])
{ . . . }          (LEAVE ON BOARD, MAY FILL IN LATER)
```

Similarly, main.c needs getop, push and pop. especially important since double pop(void); is not the default return type.

Now there is nothing wrong with having more functional prototypes than you need. Header files are a way of saving a lot of typing. A header file will contain a needed set of functional prototypes and extern declared variables, as well as #define statements, macro definitions, and typedefs to be defined at a later point. When we say:

```
#include <stdio.h>
```

we are copying a set of declarations into the current file at this position, from another file named stdio.h. The angle braces surrounding the file name mean that the compiler will search for the file in a default directory (e.g. /usr/include, but not for gcc). Go look at stdio.h in that directory. We can also create our own file, in the current directory, calc.h, and write:

```
#include "calc.h"
```

Quotes surrounding mean search for the header file in current directory (with source files). In our case, calc.h file contains the functional prototypes for push, pop, getop, getch, ungetch, and at the beginning:


```
#define NUMBER 0
```

When you want to use math functions (SIN() or SQRT()), write `#include <math.h>`, but also the compiler must be invoked with a special flag:

```
gcc -lm . . . (lm means library, math)
```

Let's consider the `getop`, `getch`, `ungetch` functions. In `getop()`, we will try to input a char string for a floating point number. But if the string ends with a '+', we want to put that back in `stdin`, somehow, for a successive invocation of `getch`. We do this with `ungetch()`. Here is the file `getch.c`.

```
#define BUFSIZE 100
```

```
static char buf[BUFSIZE]; /* buffer for ungetch          */
static int bufp = 0;      /* next free position in buf    */
```

(static is difficult idea. When external, means variable can't be referenced from another file even if declared extern; when internal to a function, static means variable lives forever, not on stack.)

```
int getch(void)          /* get a pushed back?) char from stdin */
{
    return (bufp > 0) ? buf[--bufp] : getchar( );
}
void ungetch(int c)       /* push char back so can getch later   */
{
    if (bufp >= BUFSIZE)
        printf("ungetch: too many characters in buffer.\n");
    else
        buf[bufp++] = c;
}
```

Go over logic. If still time, do `getop`, push & pop, then `main`.

Class 11.

EXAM 1 on Class 14. OPEN BOOK, NOTES, CLASS PERIOD.

FINISH UP Chapters 3 & 4, Start reading Chapt 5. Hw4 soon on-line.

Go over Hw 2 solns. Hw 3 due next class -- **HARD DEADLINE!!**

You are responsible for: 4.6 Static Variables.

An **automatic** variable is one normally declared inside a function: it appears on the stack when the function is called and disappears when the function returns (therefore it doesn't hold a value between function invocations); the scope of the variable is private to the function (and certainly private to the file containing the function).

Idea of **external** variable: if you declare a variable outside of any function, it does not appear on the stack, and holds values across all function invocations (we can think of it as static, but that's confusing). It is accessible by any function in the file but is private to file UNLESS you declare the same variable name and type "extern" in other file.

If declare a variable **static** outside of any function, it becomes private to the file (cannot be referenced by "extern" from another file). KIND OF CONFUSING NAME! E.g., see pg. 82, static char buf[BUFSIZE], explained on pg. 83.

But **Internal static** means something QUITE DIFFERENT, and occurs if define a variable INSIDE a function:

```
unsigned int rand(void)
{
    static unsigned int seed;
    . . .
}
```

Such a variable is PRIVATE to the function (cannot be referenced outside it) but does NOT appear on the stack and keeps its value across function invocations.

4.11 C Preprocessor. Explain idea -- preprocessing before compilation.

4.11.1 #include copies other file (called header files, xxx.h) into source file before compiling -- has been covered.

Note there is a rule that a header file should not contain definitions of variables or functional code that takes up memory space at runtime.

Should only include declarations (of structs), #define statements, etc.

4.11.2 Macro Substitution (NOT in JAVA). Simplest form:

```
#define name replacement-text (e.g., #define NUMBER 0)
```

Single line, but can continue long definition using backslash character (\)

Macros are substitutions which have ARGUMENTS. Adds tremendous power!

```
#define square(A) ((A)*(A)) -- always enclose whole expression in ( )s
```

Now if write in program: `n = square(x) + 1;`

will be replaced by:

```
n = ((x)*(x)) + 1;
```

Show how precompiler looks for Macro string and left parenthesis, assigns argument replacement, then does substitution. `n = square(p+1) + 1;`

Be careful! Macros do not understand what they're doing, don't understand about expressions. Only doing precise character substitution. If had written:

```
#define square(A) A*A
```

would translate: `n = square(p+1);` as:

`n = p+1*p+1;` Not what you expected. If `p = 6`, result is 13. Shows we should have instead defined `square(A)` by: `#define square(A) (A)*(A)`

Recall that in `stdio.h`, `getchar()` is defined as a macro to improve performance. See `/usr/include` for various header files! Now another example:

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

Note \ simply says to compiler following line is really same logical line.

What does the following macro do? The way it is to be used is to swap the values of two variables, say they are u, v, and w of type char:

```
exchg(char, u, v);
```

This will be turned into the following text string (one statement per line).

```
{ char d; /* *NOTE* this dummy variable decl. is block local */
  d = u;
  u = v;
  v = d; }
```

Recall idea that function calls are CALL BY VALUE (can't exchange variable values in a function unless pointers are passed). This is NOT true for Macros, because the statements within a Macro expansion act like in-line code!!!!

Quick point. What does the following expand to (had on an exam)?

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

move(char, u, v, w); --this will become (next pg, indented)

```
{char d; /* this dummy variable declaration is block local */
  d = u;
  u = v; /* if start with u == 1, v == 2, w == 3,... */
  v = w; /* end with u == 2, v == 3, w == 1 */
  w = d; }
```

Substitutions do NOT take place within quotation marks. If we want to print out the variable makeup of an expression and then the evaluated value (e.g., $x/y = 17.2$) it doesn't work to do it like this:

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

(%g is same as %f: see p 246.) In writing the above, we might hope that when we encounter the following in the program logic:

```
dprint(x/y);
```

We will see: $x/y = 17.2$ Instead, it will expand as:

```
printf ("expr = %g\n", x/y);
```

Want to see line: $x/y = 17.2$

See instead: $\text{expr} = 17.2$, because there is no substitution in quotes.

However, there is a way to get around this, using a special convention with the # character. When we define:

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

The special form #expr means: (1) do substitution, (2) put quotes around result. Now if we write:

```
dprint(x/y);
```

this expands as:

```
printf("x/y" " = %g\n", x/y);    ** It Works! **
```

4.11.3 K&R Conditional Inclusion

There are a few reasons why we might want to control when precompiler directives such as #define or #include are executed. This can be done at precompilation time with conditionals meaningful at that time.

```
#if          (works with conditions such as !defined(SYMCONST)
#ifdef SYMCONST (like saying #if defined(SYMCONST)
#ifndef SYMCONST (like saying #if !defined(SYMCONST)
#elif        (like else if)
#else        (like else)
#endif      (end scope covered by originating #if)
```

These conditionals work for any C statements in their scope, and can be used to drop code (and the memory space it takes up) in some conditions. For example, we might define a DEBUG condition.

```
#define DEBUG
```

Then we can print out certain information in that case

```
#ifdef DEBUG
printf ( . . . )
#endif
```

A software release might have different header files to for different UNIX OS's. Then before main() we could show which OS is being used. Start with:

```
#define SYSV 100
#define BSD  101
#define MSDOS 102
```

And depending on which system is being used (say SYSV) we write:

```
#define SYSTEM SYSV (or we could write: #define SYSTEM 100)
```

Then we can define other symbolic constants in conditional way:

```
#if SYSTEM == SYSV
    #define HDR "sysv.h"
#elif SYSTEM == BSD
    #define HDR "bsd.h"
#elif SYSTEM == MSDOS
    #define HDR "msdos.h"
#else
    #define HDR "default.h"
#endif

#include HDR
```

Another reason for using such conditionals is that we often include header files in other headers, recursively. It might happen as a result that we include two headers that both recursively include another file.

It can lead to a compiler warning, maybe even an error, if the same functional prototype appears twice in a compilation.

Therefore we do NOT want to include the declarations contained in a header twice. The following precompiler conditionals achieve this, when placed inside a header named xxx.h.

```
#ifndef XXX_HDR
#define XXX_HDR /* we don't need to provide a value here */
                (Next time XXX_HDR will be defined so #ifndef will fail)
. . . . . (contents of header file go here)
#endif      /* XXX_HDR */
```

Register Variables. Section 4.7. For speed!

```
void f(register int n)
{
    register int i;

    for( . . .; . . .; i++, n--)
```

Gives a lot of extra speed for frequently modified variables. There are a limited number of registers in a machine (where arithmetic is usually performed -- have to load register from memory location of variable, perform increment/decrement, store value back).

```
L    r1, VAR
INC  r1
ST   r1, VAR
```

Declaring a variable as a "register" variable is an advisory to the compiler to make the normal location of the variable in the register (so don't have to pick it up and put it back in memory).

Advisory only, because number of registers is quite limited, and only in creating the code is it clear what is best. Still, usually works.

A static variable cannot be register type (use register for all time). Can only have automatic variables and formal function parameters (treated like local variables) as register variables.

A pointer is to a position in memory. If n is a register variable, can't use expression &n!

Recursion. Section 4.10

Any C function you use may call itself recursively, either directly or after intermediate function calls. Recall that each time a call is made, a frame is placed on the stack, containing passed arguments, automatic variables, and a position to return to.

```
int f( int u, int v)
{
    int w, x;
    .   .   .
    f( w+ 6, x -7)
    .   .   .
}
```

and call to f from main with values f (100, 100).

Show what stack will look like.

Only reason for recursion is that it may make the code easier to understand.

Class 12. (Prior lecture marks end of material for Exam 1, but makeup will include ptrs.)

Exam 1 Class 14, two classes, Graded Class 14, Make-Up after Class 15.
Go over solutions to HW3. HW4 on-line.

Now Chapter 5, pointers and arrays. Will be simple questions on Make-up.

Consider the declarations:

```
int x = 1, y = 2, z[10];
int *ip; /* ip is a pointer to int */
```

Always read *xp as: "the thing pointed to by xp"

When we declare a pointer, we do it indirectly by saying: "The thing pointed to by ip is an int." We can never declare a pointer without saying what type of object it is pointing to. A pointer could even point to another pointer (which points to an int).

Now draw a picture of memory, with locations for x, y, z[0] . . . z[9], and ip.

		Memory Address (Hex)
x	00.....001	1024
y	00.....010	1028
z[0]	102C

z[9]	1054
ip	..0001'0000'0010'0100	1058 (Junk init, later Hex 1024)

Now if we execute the following statements:

```
ip = &x; /* ip now points to x -- draw arrow in picture */
```

This is another new operator, &, and the value of &x is "the pointer to x"

```
y = *ip; /* y set to x (the thing pointed to by ip) picture */
        /* just as if wrote y = x; */
*ip = 0; /* x is set to 0 */
ip = &z[0]; /* ip now points to z[0] */
*ip = 3; /* set z[0] to 3 */
*ip = *ip + 10; /* set z[0] to 13 */
```

Note that & and * are both unary operators Look at table on pg 53.

```
y = *ip + 1; /* adding 1 to the thing pointed to by ip */
*ip += 1; /* incrementing the thing pointed to by ip */
```


Now look at:

```
++*ip;      /* increment the thing pointed to by ip      */
             (same as ++(*ip); binds first right to left)
*ip++      (how bind? *(ip++). increment ip, return value)
(*ip)++;    /* increments thing pointed to by ip          */
             (needs ( )s, otherwise same as *(ip++); the thing pointed to
             by the incremented ip.)
```

It IS possible to increment a pointer. All machine addresses are byte addresses, and a pointer is a number corresponding to the byte address of the thing pointed to. When you increment a pointer, the pointer-address is incremented by THE NUMBER OF BYTES IN THE THING IT POINTS TO.

For example a char pointer cp declared as

```
char *cp;
```

when incremented, as in cp++, the pointer-address in machine terms will be incremented by 1, but an int pointer-address will be incremented by 4, and a double pointer-address will increase by 8. The int pointer however is not THOUGHT OF as being incremented by 4, that's hidden from the C programmer -- it's simply said to be incremented by an integer value, 1.

To reiterate what we covered earlier, recall that the following function

```
**** DOESN'T WORK ****      **** POINTER VERSION INSTEAD ****
void swap (int a, int b)    void swap (int *pa, int *pb)
{
    int dummy;
    dummy = a;
    a = b;
    b = dummy;
}
{
    \
    int dummy;
    dummy = *pa;
    *pa = *pb;
    *pb = dummy;
}
```

(Left-hand example doesn't work in Java either, and no simple ptr fix.)

With the second version, to call swap to exchange the values of two variables x and y, write: swap(&x, &y);

Section 5.3. Pointers and arrays. Declaration/Initialization

```
int a[10], *pa = &a[0]; /* initialize pa to point to a[0] */
```

Note that although we read *pa as "the thing pointed to by pa", when we are performing initialization in the declaration, * is part of the type. Then really initializing pa, setting to the thing after the = sign, not *pa.

Now a point which is relatively difficult to grasp is that an array name (without subscripting) and a pointer are treated in THE SAME WAY by C. We could write:

```
pa = &a[0]; or we can write the equivalent: pa = a;
```

It is as if the array name "a" was a specially initialized pointer which points to the array element subscripted by 0.

But a is an unchanging (constant) pointer: $a = a+1$ is not possible; like writing $7 = 7+1$. (Also, **defining** (not just **declaring**) an array sets aside space for contents! No contents declared when declare a pointer!)

Now we see that the way incrementation works for pointers is perfect for picking off successive elements of an array it points to:

```
*pa means same as a[0]
*(pa + 1) means the same as a[1].
*(pa + m) means the same as a[m].
```

Consider the example:

```
int a[ ] = {0, 2, 4, 6, 8, 10, 12, 14, 16, 18};
int *pa;

pa = &a[3];
What is the value of *(pa + 3) ?
What is the value of *pa + 3 ?
What happens when evaluate the expression *pa++? What is value?
What happens when we evaluate ++*pa? What is the value?
```

Be careful: $*pa++$ evaluates from right to left, meaning that we can write the parentheses as $*(pa++)$, but the $++$ still happens later in time !!!

Now perhaps even more surprising, we see that an array name can be incremented in the same way as a pointer (in an expression -- it cannot be changed permanently).

```
*(a + m) is the same as *(pa + m), where pa = a.
&a[m] is the same thing as a + m
```

On the other hand, $*(pa + m)$ can be written as $pa[m]$.

We are finally in a position to understand how C programmers deal with strings in functions. Here is a variant of the way we used to do it.

```
int strlen(char s[ ])    /* return count of chars before \0 */
{
    int n;

    for (n = 0; *(s+n); n++)
        ;    (use s+n as ptr, dereference, and test value non-zero)
    return n;
}
```

We can call this with arguments such as: `strlen(arrayname)` or `strlen(ptr)` where the array and pointer are of char type, or `strlen("hello, world")`. This can be made more efficient.

```
int strlen(char *s)      /* return count of chars before \0      */
{                        /* strlen is a C library function      */
    char *cp;

    for ( cp = s; *s ; s++)    (no sum)
        ;
    return s - cp;
}
```

We don't keep track of the subscript `n`, only increment the pointer `s`; `s` is just copy, so no real change to calling variable occurs; show on stack!

The expression `s - cp` has an int value, corresponding to the number of elements we have advanced in the array `s`. Slightly more efficient: ALL C programmers use pointers in cases like this.

Section 5.5. Here are another few programs using char pointers.

```
void strcpy ( char *s, char *t)    /* copy t to s */
{
    while ( *s++ = *t++ )          /* stops when *t = '\0'      */
        ;                          /* look at page 105 examples */
}
```

Go over why this works. Note there is an assignment statement inside the `while ()`, not a comparison, and the copy stops AFTER the assignment to `*s` of the final value 0.

Here's another one.

```
/* strcmp:  return < 0 if s < t (alpha order), > 0 if s > t,
           0 if s == t */
int strcmp ( char * s, char * t)    (note can put space after *)
{
    for ( ; *s == *t;  s++, t++)
        if ( *s == '\0')
            return 0;
    return *s - *t;
}
```

Why this works. If ever see `*s == '\0'`, since `*s == *s`, we know we have compared strings and found no mismatch. Otherwise, on the first mismatch (`*s != *t`), we will return `*s - *t`, `< 0` if `*s < *t` and `> 0` if `*s > *t`.

Another nice example (simplest form yet: see pg 39, K&R).

```
/* strlen:  return number of chars in s[ ] before '\0' */
int strlen (char *s)
{
    char *p = s;

    while ( *p++)
        ;
    return p - s - 1;
}
```

Class 13. Exam 1.

Class 14. Make-Up Next Class.

Hw 4 is on-line,

Review. Introduced pointers.

```
int x = 3, a[ ] = {1,3,5,7,9,11,13,15,17,19},
    *pa = &a[4], *pb = &a[1];
```

What is the value of: $*(a + 2)$?	Same as $a[2]$
What is the value of: $pb - pa$?	-3
What is the value of: $pb[1]$?	Same as $a[2]$
What is the effect of: $*pa += 5$?	$a[4] += 5$
What is the effect of: $*(pa += 2)$?	$pa = \&a[6]$, value is $a[6]$
What is the effect of: $*(a += 2)$?	Illegal, can't increment a
What is the value of: $pa[3]$?	Same as $a[9]$

Recall pointer and array are just the same, except that array name takes up no memory space and therefore can't take on a new value.

Valid pointer arithmetic is:

- (1) Setting one pointer to the value of another pointer of the same type.
 $pa = pb$ or $pa = \&a[3]$
- (2) Adding pointer and an integer: $pa + 3$, $pa - 5$
- (3) Subtracting or comparing two pointers to members of the same array (same type): $pa - pb$, $\text{if}(pa \leq pb)$. Note: $pa - pb$ is an integer.
- (4) Assigning or comparing a pointer (of any type) to zero, also called NULL (in stdio). $pa = 0$; $\text{if}(pa \neq \text{NULL})$; **BUT NOT $pa > 0$**

Note that a NULL pointer doesn't point to anything (indicates a failure of routine meant to return a pointer).

All other arithmetic is invalid.

If we add new declarations to above, which assignments below are valid?:

```
char s[ ] = "Hello, world!", *cp = &s[4];

cp = cp - 3; YES
pa = cp; NO Might be: pa = (int *) cp; (alignment problem?)
pa = pa + pb; NONONO
s[4] = (cp < pa)? 'a': 'b'; NO
s[4] = (cp > 0)? 'a': 'b';
cp = NULL;
```

Section 5.6. Pointer Arrays; Pointers to Pointers. Recall that if we define

```
char a[10];
```

We are setting aside space in memory for the elements of the array a, but a can be treated as a pointer. We can write *a or *(a + 5).

Now think about the declaration:

```
char *a[10];
```

What does the array a contain, class? (Pointers to char: strings!) Though it's hard to think about, we could now write:

****a**, or ***(*(a + 5) + 2)** (The third char in the string ptd to by a[5]).

Now what is the use of keeping an array of pointers to char strings? book gives the example of reading in a sequence of lines, placing them "somewhere" (in malloc() blocks, perhaps) and building an array of pointers to them.

```
char *lineptr[MAXLINES];
```

```
•---> defghi
•---> klmnop
•---> abc
```

Then sort everything, changing the POINTERS rather than moving the strings around (they might be quite long). More efficient as a sort.

```
•---\---> defghi
•---/\---> klmnop
•---/\---> abc
```

(Relevant to hw4 tail problem)

How to write out lines in ptr order.

```
void writelines(char * lineptr[ ], int nlines)
{
    while (nlines-- > 0)
        printf("%s\n", *lineptr++);
}
```

___ could be char **lineptr!!!
___ could be lineptr[i++], if had var i

How to write out the hex ascii value of the third char in each line. Change printf to:

```
    look up  
    /  
    printf("%x\n", *((*lineptr++) + 2); (%x does not expect ptr)
```

Section 5.10, Command-line arguments. The main() function is provided arguments by the runtime C environment. (Must declare them to use them.)

```
main (int argc, char *argv[ ])
```

The value of argc is the number of char strings in the array argv[] (array of ptrs to command line tokens separated by white space).

argv[0] always points to the name of the runtime name typed in by the user to invoke main, and if there are no other arguments then argc = 1.

If there are other arguments, for example, if the program was compiled as echo, and the user typed

```
echo hello, world
```

then argc will be 3; argv[0] points to "echo", that is: e c h o \0, argv[1] points to "hello," (comma in token), and argv[2] points to "world". argc counts the number of strings pointed to, starting with subscript [0]

The program can print back the arguments typed in by the user after echo (this is the meaning of calling the program "echo").

```
int main (int argc, char *argv[ ]) (Picture: argc = 3, argv[ ])  
{  
    while (--argc > 0)  
        printf("%s%s", *++argv, (argc > 1) ? " " : "");  
    printf("\n");  
    return 0;  
}
```

Note that *++argv starts by pointing to first argument, incrementing argv (copied pointer, not array name, and therefore incrementable) to AFTER argv[0] which contains a pointer to program name. Stop when --argc == 0.

Nice example given in Section 5.10. The program compiled with the run file name find looks for a pattern in standard input and prints out lines found. Want to add options -x and -n to the invocation: find *pattern*.

```
find [-x] [-n] pattern
```

could also be `find -xn pattern`. `-x` means all lines *except* the lines with this pattern (e.g., students who do NOT have "U.S." in line), and `-n` means include line number in print out (location of reference in 20,000 lines of help text).

Program will parse first, set of option flags: except and number. Loop to parse options and set up flags is:

```
DRAW argv[ ] What is meant? loop through argv entries while first
                / char ptd to is '-' (maybe find -x -n pattern)
while (--argc > 0 && (*++argv)[0] == '-')
    while (c = *++argv[0]) <--- What meant? incr ptr to string
        switch (c) {
            of arg, maybe more than one not '\0'
            case 'x' : except = 1; break;
            case 'n' : number = 1; break; (maybe find -xn pattern)
            default : printf("find: illegal option %c\n", c); break
        }
```

Very important how `(*++argv)[0]` and `*++argv[0]` differ.

Class 15. START USING LIBRARY FUNCTIONS!!

Skip 5.7, 5.8 for now. Will not cover Sect. 5-12, complex declarations.

Sect 5.4. Idea of alloc and free. The call `p = alloc(n)` returns a pointer `p` to a block of usable space, `n` characters in length, to be used by the program, which gives the block of space back when it is finished by calling `afree(p)`.

Example in Section 5.4 is "rudimentary", meaning that it must be called last allocated, first returned. Stack-like. Draw diagram, left-to-right.

```
#define ALLOCSIZE 10000 /* size of available space */

static char allocbuf[ALLOCSIZE]; /* storage for alloc */
static char *allocp = allocbuf; /* next free position */

char *alloc(int n) /* return pointer to n character block */
{
    if (allocbuf+ALLOCSIZE-allocp >= n) { /* if request fits */
        allocp += n; /* point to after this block */
        return allocp - n; /* return p pointing to this block */
    } else /* not enough room */
        return 0;
}

void afree (char * p) /* return block pointed to by p */
{
    if (p >= allocbuf && p < allocbuf+ALLOCSIZE) /* ptr OK? */
        allocp = p;
    else
        printf("Error in afree call, return ptr %x\n", p);
}
```

Explain. `if()` in `alloc` might better be:

```
right of new allocation fits to left of right end of allocbuf
/                               /
if ( allocp + n <= allocbuf + ALLOCSIZE)
```

`if()` in `afree` is too trusting in book. Might better check a bit more:

```
if (p >= allocbuf && p < allocp)
```

If not, there is an error. I added the `else ERROR` in `afree()`

There are better functions `malloc()` and `free()`, library functions (so we will have to use other names if we write our own).

Former hw 5 asks you to write better alloc and freef. Explain a bit of this later, allow as special extra-credit homework.

Section 5.11. Pointers to Functions.

The idea here is that we sometimes want to make a procedure insensitive to variations in data by passing pointers to functions. The functions pointed to will handle these data variations differently. The example given is to sort things, either numerically or lexicographically.

Note that the char string "123" is less than the char string "2", but `atof("2")` is less than `atof("123")`. `strcmp(char *s1, char *s2)` will do the first kind of compare. (Returns <0, 0, >0.) Create

```
int numcmp(char *s1, char *s2)
```

to do the second kind, convert s1 and s2 to double, and return appropriate -1, 0 or 1.

Now declare `qsort` (not same as `qsort` of C library, pg. 253:

```
/* qsort: sort v[left] . . . v[right] into increasing order */
void qsort(void *v[ ], int left, int right, int (*comp) (void *, void *))
{ . . . }
```

See page 120. I will go into detail on this function in a moment.

Note that the declaration: `void *xx` means `xx` points to ANY type (used in arguments). So `void *v[]` declaration adds to flexibility to sort other type objects in future. Don't want to make `qsort()` routine aware of all types. Any pointer can be cast to `(void *)` and back without information loss.

Now argument in `qsort`: `int (*comp) (void *, void *)` is a function pointer declaration, of a function with two arguments returning an int.

The declaration: `int *comp (void *, void *)` would be a function returning a pointer to an int (and should not appear as an argument of `qsort`).

Note implicit declaration: `*comp` is a function. Function would be decl:

```
int xx(void *, void *); <-generic compare.
```

The var comp is a fn pointer. It gets invoked in the body with the call:

if ((*comp) (v[i], v[left]) < 0) . . . (*comp) is the function pointed at.

Now look at the main() routine in pg. 119, **where qsort is called**. (Copy pgm on board.) Call looks like:

```

        An array of pointers to things (chars?)
        /
qsort((void **) lineptr, 0, nlines-1,
      (int (*)(void *, void *)) (numeric ? numcmp : strcmp));
      \
      this is a cast to function ptr of either numcmp or strcmp

```

Cover qsort logic: Idea. (1) find some value in array, move all smaller values in the array to earlier positions, all larger values to later positions. (2) Now recursively repeat 1 with the smaller values ; do the same with the larger values. End up with all values sorted.

Copy qsort routine on board (pg. 120); example: v[0] to v[5] = (9, 5, 7, 1, 3}

Works well as long as each value found has about the same number of array elements larger and smaller. Efficiency of 20 Questions.

* **NOTE**: We will **not** cover Section 5.12, Complicated Declarations.

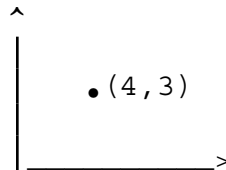
Chapter 6. A **struct** is a collection of variables, possibly of different types, grouped under a single name for common reference as a unit.

Example of a point in coordinate space, point p in x-y space follows:

```

tag (optional)
/
struct point {
    int x; <-- members
    int y; <--
} pt;
\----- struct variable
or struct {
    int x, y;
} pt, q;

```



pt q

pt.x	pt.y	q.x	q.y
------	------	-----	-----

Layout in memory: |-----|-----|-----|-----|

Given that we have the tag, point, can declare other variables (structs) by

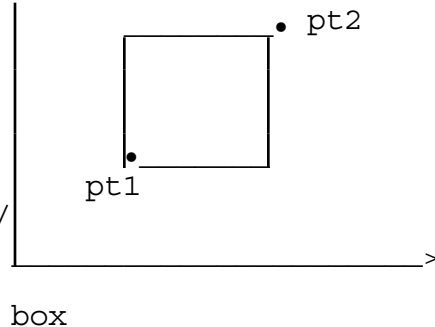
```
struct point pt1, maxpt = {320, 200};
** struct point is like a new "type" **
```

Reference to members:

```
pt1.x = 320; pt1.y = 200; /-- prints out as (320, 200)
printf("(%d, %d)\n", pt1.x, pt1.y);
```

Struct inside a struct (nesting).

```
struct rect {
    struct point pt1; /* lower left */
    struct point pt2; /* upper right */
};
struct rect box;
```



```
Memory layout of box:
-----
                box.pt1                box.pt2
-----
box.pt1.x  box.pt1.y  box.pt2.x  box.pt2.y
|-----|-----|-----|-----|
```

Idea is that pt1 is lower left hand corner and pt2 is upper right hand corner. I.e., $\text{box.pt1.x} < \text{box.pt2.x}$ && $\text{box.pt1.y} < \text{box.pt2.y}$

```
int rectarea (struct rect box) /* Find area of a rectangle */
{
    return (box.pt2.x - box.pt1.x) * (box.pt2.y - box.pt1.y);
}
```

Class 16. Quiz 2 Class 18. START USING LIBRARY FUNCTIONS!!

Hw 5 up, structs ptrs. Review structs. struct point; struct rect;
Layout in memory. Assignment: 5.1-5.5, 6.1-6.5, 6.9. Read through 6.5 for
next time or you might have problems following class.

What can we do with a struct? (1) Reference member: rect.pt1.x (2)
Assignment of entire struct: pt1 = pt2; (3) Create a pointer to: p = &pt1;

(Need declaration: struct point *p;)

Not legal to compare whole struct: pt1 == pt2 (INVALID)

```
/* ptinrect: return 1 if point p in rect r, 0 else */ (LEAVE UP)
int ptinrect ( struct point p, struct rect r)
{
    return p.x >= r.pt1.x && p.x <= r.pt2.x
        && p.y >= r.pt1.y && p.y <= r.pt2.y;
}
```

Suggest to class they try to create ptbordrect, which returns 1 if point p
on border of rect r.

Pointers to structs. As we mentioned earlier, can declare a pointer to a
struct.

```
struct point *pp;
```

```
    /-- (look at precedence)
```

Then can refer to (*pp).x and (*pp).y, but it is more common to use the
notation: pp -> x (same as (*pp).x) and pp -> y (same as (*pp).y) instead.

```
    \--- (careful: pp points to struct, x names member)
```

```
struct rect r, *rp = &r;
```

```
(rp -> pt1).x = 22; (same as saying r.pt1.x = 22;) (or
                    (*rp).pt1.x = 22;)
```

-> is hyphen followed by GT symbol

Highest precedence on p 53: . , -> , () (function calls), [] (subscripts)
(know everything about precedence now!)

```
struct string {
    int len;
    char *cp;
} *p;
```


Class 17. Typedef. Quiz next time on Chs. 5 & 6 through Section 6.6.

Now to prepare for Memory Allocation assignment, hw4, we skip ahead.

Section 6.9, bit fields.

Idea, might want to "pack" information, several bits at a time, into a larger data item, a (four-byte longword?) in memory, to save space.

The book gives the example of naming flags in a word. Uses struct form:

```
struct {
    unsigned int flg1 : 1; /* ": 1" means "1 bit in length" */
    unsigned int flg2 : 1; <-- These are called "fields"
    unsigned int flg3 : 1;
} flag; /* variable */
```

Minimum size data item is implementation dependent, as is order that fields sit in memory. But here's an idea of layout in memory.

```
flag          (flg1, flg2, flg3)
               \\\
|-----|-----|-----|-----|
```

Can set values (in this example: `flag.flg1 = 1;` if `(flag.flg1 = 0)`, etc. But we can have LONGER fields, like little integers. (Change `flg1 : 1` to `: 2`. Now can use integers 0, 1, 2, 3. `flag.flg1 = 2;` if `flag.flg1 > 1`), etc.

Bit fields are used when want significance of number other than 8 bits (char), 16 bits (short on some machines) or 32 bits (long on most machines). Used in hw6 assignment.

```
struct blockr {
    unsigned tag : 8;
    unsigned size : 24;
};
```

Now introduce alloc and freef assignment, hw5.

```
struct blockl {
    unsigned tag : 8;
    unsigned size : 24;
    struct blockl *nextp;
    struct blockl *prevp;
};

Later
static struct blockl
*freep /* head of chain */
*cursorp /* into chain */
```

Now consider the idea of a Binary tree given in Section 6.5. Nodes contain pointer to left children and right children.

Words encountered in stdin are inserted into nodes as "keyvalues" to look up an inforec (here just int count of number of times word encountered).

```
struct tnode {          /* the tree node      PUT ON BOARD          */
    char *word;          /* points to the word encountered      */
    int count;           /* could be more general inforec      */
    struct tnode *left;  /* left child: words < one at this node */
    struct tnode *right; /* right child: word > one at this node */
};
```

As encounter new words (C identifiers) in stdin, we add nodes to tree so far built, set count to 1; if word already exists, increment node count by 1.

Picture of binary tree, page 139, as encounter words: Now is the time for all good men to come to the aid of the (instead of their) party.

Here is code to do this on pg 140. *****NOTE***** that from now on in the course you are encouraged to use library functions whenever possible and learn everything about the library functions in Appendix B of K&R.

```
#include <stdio.h>
#include <ctype.h> (see functions in App. B2, pp. 248-249, K&R)
#include <string.h> (see functions in App. B3, pp. 249-250, K&R)
#define MAXWORD 100

struct tnode *addtree(struct tnode *, char *);
void treeprint(struct tnode *);
int getword(char *, int); (***ASIDE TO LOOK AT THIS, pg. 136***)

int getword(char *word, int lim) {
    int c, getch(void); void ungetch(int); char *w = word;
    while (isspace(c = getch())) /* skip spaces; c first non-space */
        ;
    if (c != EOF) *w++=c;        /* c might be EOF here (empty word) */
    if (!isalpha(c)) {          /* c (identifier) starts with alpha */
        *w = '\0'; return c;    /* if not, have single char or EOF */
    }
    for( ; --lim > 0; w++)
        if (!isalnum(*w = getch())) { /* identifier alpha or digits */
            ungetch(); break;        /* e.g., + might be needed later */
        }
    *w = '\0'; return word[0];
}
```



```

int main( )
{
    struct tnode *root;    /* ptr to root for binary tree          */
    char word[MAXWORD];    /* word to get from stdin    */

    root = NULL;           /* initially, no tnodes in tree */
    while (getword(word, MAXWORD) != EOF) /* getword from stdin */
        if (isalpha(word[0])) /* from ctype.h library */
            root = addtree(root, word); /* expect not NULL anymore */
    treeprint(root);        /* print it out in order */
    return 0;
}
    OK that root ptr points to NULL; it is set recursively

```

Now here is addtree. Need a few functions on pg 142 & 143 (See below).

```

struct tnode *talloc(void); /* allocate. return ptr to tnode */
char *strdup(char *); /* allocate space, copy word there */

struct tnode *addtree(struct tnode *p, char *w)
{
    /* p may be NULL */
    /* finds place in tree for word; increments count if found,
    /*...else extends tree -- recursive
    int cond;

    /* for empty tree, root is NULL
    if (p == NULL) {
        /* nothing at this node yet
        p = talloc(); /* allocate a new node
        p->word = strdup(w); /* allocate space, copy word there
        p->count = 1; /* count is 1
        p->left = p->right = NULL; /* this works; see precedence
    } else if ((cond = strcmp(w, p->word)) == 0) /* string.h
        p->count++; /* repeated word, increment count
    else if (cond < 0) /* note cond remembers strcmp above
        p->left = addtree(p->left, w); /* less: go left subtree
    else /* more, go to right subtree
        p->right = addtree(p->right, w);
    return p;
}

```

Note, of course, that addtree is recursive. If it has to add a node, might need to add root or any left or right child anywhere down the tree.

We pass in as first argument struct tnode *p, and also pass back the same type: always the node we are at the current recursive nesting level.

This means that when main() calls addtree, it resets root value EVERY TIME, even if the root has been created long ago. Allows deletion.

But any time a new node is created, at root or p->left or p->right from node above, will be set for the FIRST time by return from addtree.

Why do we need to pass back struct tnode *p? We have struct tnode *p as an argument. Can't we just set it there?

That is, couldn't we return void from addtree and write (in main):

```
addtree(root, word); instead of root = addtree(root, word); ?
```

The answer is no, because addtree would be unable to modify root -- we would have to pass a pointer to root before it could be modified.

Even though root is already a pointer, struct tnode *p, it is p that must be modified. We would have to declare addtree with a pointer to p as arg 1:

```
void addtree(struct tnode **p, char *w);
```

and call it from main (and recursively) by the invocation:

```
addtree(&root, word);
```

before addtree could set the value for root directly instead of by return. Same is true of lower levels when add left or right child.

Now consider how to write talloc to allocate a tnode and return a pointer.

```
struct tnode *talloc(void)
{
    return (struct tnode *) malloc (sizeof(struct tnode));
}
```

Now the strdup function to create space for a character string, copy a word into it (up to null terminator) and return the char pointer.

```
char *strdup(char *s)
{
    char *p;
    p = (char *) malloc(strlen(s)+1); /* +1 for null, '\0' */
    if p != NULL /* would mean malloc failed */
        strcpy(p, s); /* see library function in string.h */
    return p;
}
```

Note need to return p, and how in addtree(), when we found a passed tnode pointer p was NULL, we first created the node, then filled in the word:

```
p = talloc(); /* allocate a new node */
p->word = strdup(w); /* allocate space at p->word, copy w there */
```

Another function that allocates space, does something, and returns the ptr.

You are responsible for Table Lookup in Section 6.6, although it is not going to be covered in class (used in hw4). There are good Exam questions about this.

Class 18. Note: QUIZ Next class; we will start covering Glass & Ables UNIX. You need to bring G&A to class for Quizzes, etc.

Section 6.8. Unions.

Union is like a struct in every respect, including form of pointer to union. contactp -> noorg.address. Members of a union are like in struct, except they OVERLAY each other.

Example: employee is either represented by Organized labor or not. If Non-Organized, need to keep home address and telephone for contact in event of extra shift work offerings; if Organized, contact should be made through the Shop Steward. Have structs: (**PUT ON BOARD**)

```
struct isorg {
    char steward[20]; /* shop steward name */
    char phone[12]; /* phone number, with area code */
};
struct noorg {
    char address[30]; /* employee home address */
    char phone[12]; /* phone number of employee */
};

struct employee {
    char age; /* between 18 and 65 */
    char dept[20]; /* department name */
    char orgflg; /* 1 if organized, 0 if not */
    union contact { /* one of two possibilities */
        struct isorg iso;
        struct noorg nono;
    } contct;
} emp;
```

Have code like:

```
if (emp.orgflag)
    printf("Contact through shop steward, %s, phone: %s\n",
        emp.contct.iso.steward, emp.contct.iso.phone);
else
    printf("Contact directly at: %s, phone: %s\n",
        emp.contct.nono.address, emp.contct.nono.phone);
```

Space taken up in union is max space of members: in example above, it is 30 + 12 (or to be safe: sizeof(struct contact)).

Section 6.7. Typedef.

The idea here is to allow a programmer to define new types. The form of use is:

```
typedef oldtype newtypename; (e.g. typedef int Boolean)
```

For example:

```
struct tnode {
    char *word;
    int count;
    struct tnode *left;
    struct tnode *right;
}
```

Used in example in 6.5, listing all words in text and count of number of times encountered. Use binary tree structure to do this.

```
typedef struct tnode Treenode;
typedef struct tnode *Treeptr;
```

Treenode and Treeptr are new types. By our convention, typedef types begin with a capital letter (not general convention). Note these names are in the position of variables to be declared, except declaration preceded by a "typedef". Can now write.

```
Treeptr talloc(void)
{
    return (Treeptr)malloc(sizeof(Treenode));
}
```

What good are they? Couldn't we always write "struct tnode *" in place of Treeptr and "struct tnode" in place of Treenode? Yes, but not always a struct that's involved. For example, what is the appropriate size of an int to represent the difference between two pointers? Depends on the machine, doesn't it? ANSI C has type ptrdiff_t:

```
typedef int ptrdiff_t; /* max result of subtracting two pointers */
    \ This is a typedef; note not capital letter
```

On another machine, ptrdiff might be long int!

Subtle bugs can arise replacing a simple type with a typedef. E.g., int can be register variable, but struct cannot, so can cause error in compile.

Another example. Recall how we had to rewrite stack.c in the program calc because there was a new type of object (unsigned int) to place on the stack. We could instead declare an object to be placed on a stack thus.

```
typedef float Object;
```

Now define a Stack type as having the declaration:

```
typedef struct {
    int sp;      /* Stack position, initially zero when empty */
    Object arr[MAX]; /* array of objects representing Stack */
} Stack;
```

Now, after a Stack has been created, we can push a new Object with:

```
int push(Object x, Stack *stkp) /* push new object on stack */
{
    if (stkp -> sp < MAX)      /* there is room on the stack */
        stkp -> arr[stkp -> sp++] = x;
    else
        printf("error:  stack full.\n");
}
```

We can write a set of primitives in these terms and change the type of Object which lives on the stack merely by changing the typedef definition of Object.

Note that we can also use typedef to define a name for an array shape.

```
typedef char Name[NAMELEN]; /* now Name is array type */
Name firstname, lastname; /* firstname & lname are arrays */
```

Section 5.7, 5.8. Multi-Dimensional Arrays.

Rectangular Multi-dimensional arrays can be declared; not very much used. Example in text of converting a month & day (July 21, 1990) into a day of the year (193 or something). Have to worry about Leap Year. Consider:

```
static char daytab[2][13] = {
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};
```

Book suggests two separate rows of day counts rather than perform a calculation of days in February. Now `daytab[0][3]` is 31 (same as `daytab[1][3]`) but `daytab[1][2]` is 29.

Note that the rightmost subscript varies fastest as one looks at how data lies in memory. `daytab[0][0]`, then `daytab[0][1]`, then `[0][2]`, . . . This is standard in most languages. But there is a difference.

NOTE THAT `daytab[1, 3]` IS WRONG!

The array declared as `char daytab[2][13]` can be thought of as:

```
char (daytab[2])[13]; /* for [ ], see pg. 53; Left-To-Right associativity*/
```

Each one dimensional array element (`daytab[0]`, `daytab[1]`) is like the NAME of an array (`char <name>[13]`). As if we had declared:

```
char xx[13]; char yy[13]
```

where `xx` is `daytab[0]` and `yy` is `daytab[1]`. Look at initialization above; first init `daytab[0]`, then `daytab[1]`. Now recall duality of pointer and array.

```
char (daytab[2])[13]; Think of it as:  
char (*daytab)[13]; /* not quite the same; no 2 elts in memory * /
```

Here `daytab` is a pointer to an array of `char[13]`. Thus `*daytab` is an array and `daytab[0]` is an array of 13 elements . Think of

```
char (*daytab)[13]; /* doesn't set aside 13 element array in memory*/
```

as meaning that `daytab` is a pointer to `char[13]`. Examples follow.

```
(*daytab)[3] is the same as daytab[0][3]  
    Note need ( ) above because of precedence  
(*daytab + 1)[2] is the same as daytab[1][2]  
*(*(daytab + 1) + 2) is also the same as daytab[1][2]
```

NEXT IS IMPORTANT. In passing a two dimensional array to a function, need not specify bound on first subscript (doesn't help -- there's no checking), but must specify second.

This is how system knows where daytab[0] [j] ends and daytab[1] [0] begins. The expression *((char *)daytab + 13*i+j) is same as daytab[i] [j], but cast daytab so doesn't jump 13 bytes at a time.

```
void func(char daytab[2] [13]) { . . . } is tolerated
void func(char daytab[ ] [13]) { . . . } is OK
void func(char (*daytab) [13]) { . . . } is also OK
```

IN GENERAL, array handling is poor in C; FORTRAN much better. Efficiency and object-oriented implications. Reason for C problem? NXN array can't be handled by function: Need to specify second subscript in call.

As it is pointed out in Section 5.9, the declarations

```
int a[10] [20]
int *b[10]
```

differ in that, although a[3] [4] and b[3] [4] are both legal, the declaration of a actually sets aside 200 elements in memory, whereas the declaration of b only sets aside 10 locations for pointers.

Then b[3] is a pointer to an int, and b[3] [4] is the same as (b[3]) [4] , the fourth integer on from the integer pointed to by b[3].