LECTURE 21: TYPEDEF, UNION, BIT FIELDS, MULTIDIMENSIONAL ARRAYS (K&R §§ 5.7, 6.7–6.9)

typedef (K&R, § 6.7)

typedef creates a new name for an existing type.

`typedef int Length;`	Length is now a synonym for int.
`typedef int Boolean;`	Boolean is now a synonym for int.
`typedef char *String;`	String is now a synonym for char*.

These declared types can be used the same way primitive types are, e.g.,

```
  Length len, maxlen;
  String p, lineptr[MAXLINES], alloc(int);
  int strcmp(String, String);
  p = (String) malloc(100);
```

typedef just creates a new name for an existing data type; it does not add any new semantics.

We could have used typedef with our binary tree.

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

With the above typedefs in place, we could have coded talloc as follows:

```
  treeptr talloc(void) {
    return (treeptr) malloc(size of(treenode));
  }
```

typedef allows for clearer code as long as the names used are descriptive.

```
  treeptr root;               |  struct tnode *root;
```

The above declarations are equivalent, but the one on the left is much easier to quickly read and understand.

It also can be used to create machine-independent variable types:

```
  typedef int size_t;     /* size of types */
  typedef int ptrdiff_t;  /* difference of pointers */
```

UNIONS (K&R, § 6.8)

A union is a variable that may hold objects of different types (at different times or for different instances of use).

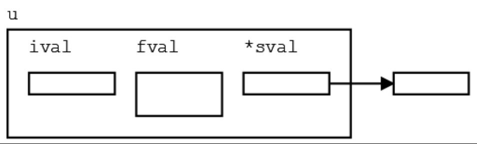
A union is a single variable that can legitimately hold any one of several types.

If we have a table of variables, and the variables could be either of type int, float, or character pointer, but we want each table entry to occupy the same amount of space, we could use the following union:

```
  union u_tag {
    int ival;
    float fval;
    char *sval;
  } u;
```

A union is allocated enough space to hold the largest type in its list of possible types.

A union is like a struct, but all members have a zero offset from the base address of the union (which intuitively tells us it can only hold one of them at a time).



The operations allowed on unions are the same as those allowed on structs:

access a member of a union:	`union u_tag x;`
	`x.ival = … ;`

assign to union of the same type:	`union u_tag y;`
	`y = x;`

create a pointer to / take the address of:	`union u_tag x;`
	`union u_tag * px = &x;`

access a member of a union via a pointer:	`px->ival = ...;`

Your program must keep track of which type of value has been stored in a union variable and process it as the correct member type.

You cannot do type conversions by trying to access one of the other types a union can hold:

```
  x.ival = 12;      /* put an int in the union */
  float z = x.fval; /* will not work! */
```

BIT FIELDS (K&R § 6.9)

Bit fields are used to get a field size other than the primitive type field sizes.

Bit fields allow us to pack data one or more bits at a time into a larger data structure in memory to save space, e.g., 32 single-bit fields into an int.

We can use bit fields instead of using masks, shifts, and ors to manipulate groups of bits.

We use the form of a struct with some added notation:

```
  struct {
    unsigned int flg1 : 1; /* called a "bit field" */
    unsigned int flg2 : 1; /* ": 1" →"1 bit in length " */
    unsigned int flg3 : 2; /* ": 2" →"2 bits in length" */
  } flag;    /* variable */
```

(fields must be ints, good idea to declare as unsigned to guarantee positive values)
(field lengths must be integral values)

Access bit fields by member name same as any struct.
The values the members can hold are limited by the number of bits we specified above:

```
  flag.flg1 = 0;  /* or = 1;  */
  flag.flg3 = 0;  /* or = 1; = 2; =3;  */
```

Almost everything about bit fields is implementation dependent, including:

-whether a field may overlap a word boundary

-whether the bits are assigned from left to right or right to left

MULTI-DIMENSIONAL ARRAYS (K&R § 5.7)

```
We can declare a rectangular multi-dimensional array with
the following syntax:
  char array [2][3];  /* array [row] [column] */
```

This is a one-dimensional array where each element is another array.

```
The above array has six elements:
array[0][0], array[0][1], array[0][2]
array[1][0], array[1][1], array[1][2]
```

```
Multidimensional array for number of days in a month:
  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}
  };
```

```
Use the second row for day counts in a leap year:
days in feb on a non-leap year: daytab[0][2]
days in feb on a     leap year: daytab[1][2]
```

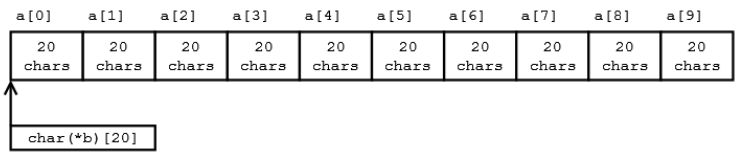
The array declared as `char daytab[2][13]`	can be thought of as `char (daytab[2])[13]`
daytab is a pointer to an array of 2 elements, each element being an array of 13 elements.	
`(daytab [0]) [13]`	`(*daytab) [13]`
`(daytab [1]) [13]`	`(*(daytab+1)) [13]`

Remember the difference between declaring pointers and arrays:

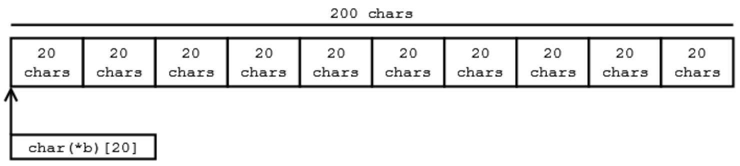
`char a[10][20];`	200 char-sized memory locations set aside
`char (*b)[20]`	1 pointer to an array of elements, each of which is 20-characters in length

```
For the second declaration, code must set the pointer equal
to an already-defined array or use malloc to allocate new
memory for an array:
  char (*b)[20] = a;
or
  char (*b)[20] = (char (*)[]) malloc(200);
```

char(*b)[20] = a;



char(*b)[20] = (char{*)[]) malloc(200);



```
     example of a pointer to a multi-dimensional array
int grid[320][240];
int (*grid_ptr)[240];

int doSomethingWithGrid(int (*array)[240]);

int main() {
  grid_ptr = grid; /* set grid_ptr to point at grid */
  doSomethingWithGrid(grid_ptr);
}
```