LECTURE 22: REVIEW

C STANDARDS

A standard or specification is a document that defines a programming language in some way (e.g., listing the syntax and semantics of the language, describing the behavior of a compiler for the language, etc.).

K&R C	Everything before standardization.	(1972–1989)
C89 / ANSI C	ANSI's 1989 standard. (ANSI also ratified C99, but as generally used, ANSI C refers to C89 (which is the same as C90)	(1989–1990)
C90	ISO's 1990 standard.	(1990–1999)
C99	ISO's 1999 revision of the standard. Most C compilers still follow this version.	(1999–2011)
C11	ISO's 2011 revision of the standard	The current definition of the C language.

ANSI C is the best-supported version of the standard, and as such is the best choice for writing portable code.

DECLARATION vs. DEFINITION

In C, all variables must be declared before they are used.	ANSI C: always at the top of the block where they are used.
	C99: allows declaration throughout the code (GCC supports this)

Declaration

Specify the interpretation given to each identifier (i.e., for a variable, its type, for a definition, its protoype).

Function prototypes and variables declared extern are declarations, as no storage is reserved.

Definition

A declaration that reserves storage.

int x; is a definition even though it does not contain an initialization value because storage is set aside for x.

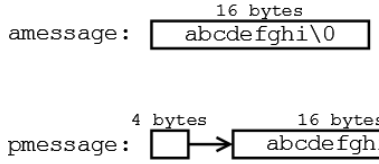
CHAR POINTERS/ARRAYS

char amessage[] = "abcdefghi";

space set for the array in memory (depending on where it is defined).

char *pmessage = "abcdefghi";

space set aside for a pointer, which holds the location of a string literal (where this is stored is implementation-defined).



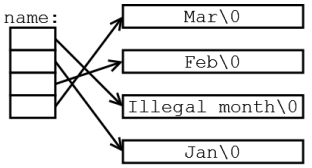
2-DIMENSIONAL ARRAYS

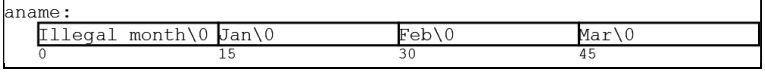
char *name[] ={"Illegal month", "Jan", "Feb", "Mar"};

an array of 4 string pointers to string literals.

char aname[][15]={"Illegal month", "Jan", "Feb", "Mar"};

an array of 4 arrays of 15 chars each.





MANIPULATING STRUCTS

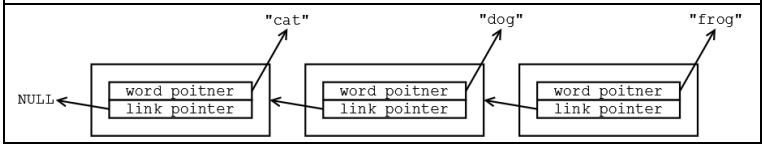
```
#include <stdio.h>
/* define an "item" structure */
struct item {
  char *word;          /* string member */
  struct item *link;  /* pointer to another item */
};
struct item *addlist(struct item *, struct item *);
void printlist(struct item *);
int main ()  /* structs built by main program */
{            /* all variables are automatic (on stack) */
  struct item cat  = {"cat", NULL},
              dog  = {"dog", NULL},
              frog = {"frog", NULL};
  struct item *list = NULL;
  list = addlist(list, &cat);
  list = addlist(list, &dog);
  list = addlist(list, &frog);
  printlist(list);
  return 0;
}  /* all memory used is freed upon return */
struct item *addlist(struct item *root,
                     struct item *animal) {
  animal->link = root;
  return animal;
}
void printlist(struct item *root) {
  for ( ; root != NULL; root = root->link)
    printf("%s\n", root->word);
}
```

Dynamic Memory Version

```
int main ()  /* structs built in addlist function */
{            /* variables point to dynamic memory */
  struct item *list = NULL;
  list = addlist(list, "cat");  /* uses malloc */
  list = addlist(list, "dog");  /* uses malloc */
  list = addlist(list, "frog"); /* uses malloc */
  printlist(list);
  list = dellist(list);  /* must free all memory used */
  return 0;
}

struct item *addlist(struct item *root, char *wd){
  /* allocate memory for a struct item and point
     the pointer "new" at it */
  struct item *new = (struct item *)
                      malloc(sizeof(struct item));
  /* allocate memory for the input word */
  new->word = (char *) malloc(strlen(wd)+1);
  /* copy the word to the allocated memory */
  strcpy(new->word, wd);
  /* point the pointer member at the root argument */
  new->link = root;
  /* return the pointer to the new struct item,
     this is assigned to list in main */
  return new;
}

/* Delete the list from the end of the list */
struct item *dellist(struct item *root) {
  struct item *temp;
  while (root != NULL) {
    free((void *) root->word);
    temp = root->link;
    free((void *) root);
    root = temp;
  }
  return root;
}
```



FUNCTION POINTERS

A C function is a pointer to a place in memory where the code for that function is located.

Example

If we have multiple functions of the form:

```
  int fname(double, char)
```

e.g.,

```
  int func(double avg, char grade);
```

We can define a function pointer that points to this type of function:

```
  int (*fp)(double, char);
```

And then assign the pointer to one of the functions:

```
  fp = &func;
```

We can now invoke the function by dereferencing the pointer:

```
  double m = 0.95;
  char g = 'A';
  int x = (*fp)(m, g);
```