CSC 357 Lecture Notes Week 2

C Program Structure
Arrays and Structs
Dynamic Memory Management

Updates to Program 1 Testing

- "`-v`" option to run.csh
- scoring works
- simplification to `sgrep` option parsing
- simplification to a couple patterns
- please recopy `testing` dir

I. Relevant reading.

A. K&R chapters 5 and 6, section 8.7.

B. Selected parts of Stevens and selected man pages, as cited in writeups.

II. Initial example -- simple linked list in C.

A. See attached listings for

- `linked-list.h`
- `linked-list.c`
- `list-node.{h,c}`
- `linked-list-test.c`
- `std-macros.h`

Linked List Example, cont'd

- `LinkedList.java`, `ListNode.java`, `LinkedListTest.java`
- `Makefile`

III. C program structure.

A. Collections of `.c` and `.h` files.

B. Preprocessor directives `#include` and `#define`.

IV. `#define`.

A. Used for constants, as in

```
#define MAXLINE 1000
```

B. By convention, all uppercase.

`#define`, cont'd

C. Also be used for parameterized *macros*, as in `std-macros.h`.

D. The general form of a macro is:

`#define` *name optional-parameters body*

`#define`, cont'd

E. E.g.,

```
#define new(t)
    (t*) malloc(sizeof(t))
```

`#define`, cont'd

F. Macros invoked strictly by *in-place textual substitution.*

1. E.g.,

```
ListNode* node = new(ListNode);
```

expands to

```
ListNode* node =
    (ListNode*) malloc(
        sizeof(ListNode));
```

#define, cont'd

2. Expansion done by C preprocessor.

3. Inspect preprocessor output using `gcc -E`.

V. Memory allocation (K&R Section 5.4).

A. The '`&`' operator is of limited practical utility for building dynamically linked data structures.

B. As illustrated in Part 1 of this week's lecture notes, programmers need to allocate new blocks of memory for such data structures.

C. Section 5.4 of K&R talks about the implementation of a simplistic `alloc` function.

D. In practice, C programmers use the library-supplied `malloc`, as well as derivatives `calloc` and `realloc`.

E. The signature of `malloc` is the following:

```
void* malloc(size_t size);
```

1. The type `size_t` is an `int` or `long`; the `size` parameter is the number of bytes to be allocated.

2. `void*` is the type of a generic pointer; in practice, the `void*` return value from `malloc` is always cast to a more specific type of pointer.

F. Here are typical examples of malloc:

```
/* Allocate memory for a 100-char string. */
char* some_string = (char*) malloc(100);

/* Allocate memory for an integer array ... */
int* a = (int*) malloc(array_size);

/* Allocate memory for a structured data value. */
typedef struct {int x; char y; char z[20];} SomeStruct;
SomeStruct* s = (SomeStruct*) malloc(sizeof(SomeStruct)
```

G. The last of these examples is so frequently used, that a macro like `new` can be very handy.

1. The definition of `new` is:

```
#define new(t) (t*) malloc(sizeof(t))
```

2. It is used, for example, like this:

```
SomeStruct* s = new(SomeStruct);
```

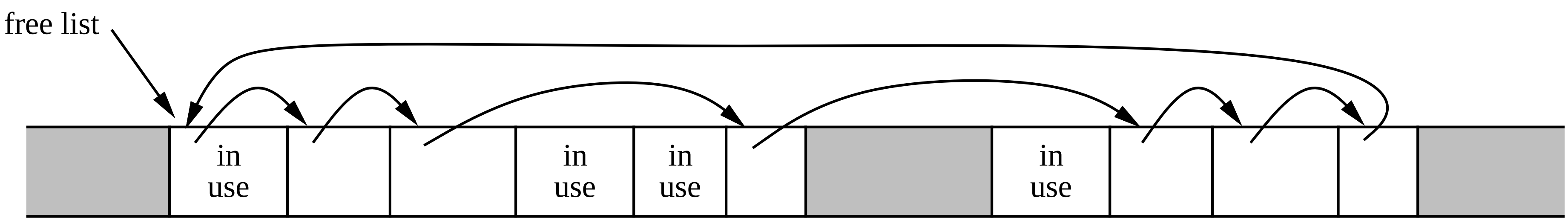
H. You should read the man page for `malloc` and related library functions (`man malloc`).

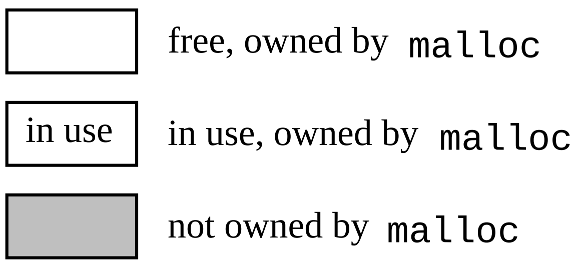
VI. How malloc works (K&R Section 8.7).

A. Malloc is reasonably straightforward C program.

B. Figure from Page 185 of K&R:

How malloc works, cont’d





How malloc works, cont'd

C. When user requests `malloc` searches freelist.

1. Can use "first fit" strategy.

2. Alternatively, can use "best fit" strategy.

How malloc works, cont'd

D. If no free block big enough, `malloc` asks OS using `sbrk`.

E. When the user `frees`, `malloc` searches and coalesces.

How malloc works, cont'd

F. Standard implementation of `malloc` does little error checking.

1. `malloc`'s memory pool can get corrupted.

2. There are packages that do more checking.

3. E.g., "`smartalloc`".

VII. More on pointers and arrays (K&R Sections 5.6 - 5.10, 5.12).

A. Read and understand these sections.

B. You can skip Section 5.10 for now.

VIII. Structures (K&R chapter 6).

A. We've seen structs in lecture, lab examples.

B. A set of variables collected under common name; vars are *fields* of the struct.

C. Compared to Java, struct is equivalent to a class with all public data fields and no methods.

IX. Basics of structures (K&R Section 6.1).

A. Syntax of a structure declaration

```
struct struct-tag {
        fields
}
```

where *struct-tag* is a name, and *fields* are variable declarations; the tag is optional.

Basics, cont'd

B. Structure fields are also referred to as *members*; the two terms are synonymous.

C. Here's a simple example:

```
struct point {
    int x;
    int y;
}
```

Basics, cont'd

D. A `struct` declaration defines a type, and so can be used directly to declare struct-type variables.

1. I.e.,

```
struct { ... } x, y, z;
```

is syntactically analogous to

```
int x, y, z;
```

Basics, cont'd

2. If a `struct` declaration contains a tag, then it can be used in subsequent decls, as in

```
struct point pt;
```

(but cleaner-looking naming is with `typedef`)

Basics, cont'd

E. Structs can be initialized in a declaration, as in

```
struct point maxpt = {320, 200};
```

Basics, cont'd

F. Struct fields are accessed with '`.`' operator, as in

```
pt.x = 10;
pt.y = 20;
printf("%d,%d", pt.x, pt.y);
```

Basics, cont'd

G. Nested `struct` defs, as in

```
struct rect {
    struct point pt1;
    struct point pt2;
};
```

Basics, cont'd

H. If we declare

```
struct rect screen;
```

then

```
screen.pt1.x
```

refers to the x coordinate of the `pt1` field.

X. Structures and functions (K&R Section 6.2).

A. Legal operations on structs are assignment, address-of, and member access.

B. For large structs, passing a struct pointer as a parameter is more efficient.

C. Pointers to structs are also necessary when creating dynamically-linked data structures.

Structs and functions, cont'd

D. There are two notations for accessing the fields of a pointed-to struct, such as

```
struct point *pp;
```

1. Expression `(*pp).x` accesses the `x` field.
2. Alternative equivalent notation is `pp->x`.

XI. Arrays of structures (K&R Section 6.3).

A. Arrays of structs are an important working data structure in C.

B. For example, a very simple word-count table:

Arrays of structs, cont'd

```
#define MAXWORDS 100

struct {
    char* word;
    int count;
} wordtab[MAXWORDS];
```

Arrays of structs, cont'd

C. Assuming the fields of the ith table element have been properly initialized:

```
wordtab[i].word[j] = getchar();
wordtab[i].count++;
```

XII. Pointers to structures (K&R Section 6.4).

A. When an array of structs is sparse, an array of pointers to structs can be more efficient.

B. Consider the following declarations:

Pointers to structs, cont'd

```
struct wordcnt {
    char* word;
    int count;
};

struct wordcnt wordtab[MAXWORDS];
struct wordcnt* wordtabp[MAXWORDS];
```

Pointers to structs, cont'd

C. Before any elements of `wordtabp` have been set, `wordtabp` is half as big as `wordtab`.

D. When contents of a table may be partially unfilled, using struct pointers is advantageous.

XIII. Self-referential structures (K&R Sec 6.5).

A. C allows a struct field to be declared as a pointer to the struct itself.

B. E.g.,

Self-referential structs, cont'd

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

Self-referential structs, cont'd

C. This is a *recursive* data type def.

XIV. Table lookup (K&R Section 6.6).

A. This section of K&R defines a simple hash table.

B. Have a look.

XV. Typedefs (K&R Section 6.7).

A. Typedef provides a convenient way to give a mnemonic name to a data type definition.

B. The typedef can be as simple as

```
typedef int Length;
```

used in declarations like

```
Length len, maxlen;
Length getLength(...);
```

Typedefs, cont'd

C. Typedefs also add readability to struct defs

```
typedef struct {
    char* word;
    int count;
} WordCount;

WordCount wordtab[MAXWORDS];
WordCount* wordtabp[MAXWORDS];
```

Typedefs, cont'd

D. When non-recursive struct is typedef'd, the struct tag need not be present.

E. But for recursive types, tag must be present for self-referencing

Typedefs, cont'd

```
typedef struct tnode {
    char* word;
    int cound;
    struct tnode* left;
    struct tnode* right;
} TreeNode;

TreeNode* tree;
```

Typedefs, cont'd

F. The following equivalent-looking definition does NOT work:

```
typedef struct tnode {
    char* word;
    int cound;
    TreeNode* left;        /* INVALID *
    TreeNode* right;       /* INVALID *
} TreeNode;
```

XVI. Unions (K&R Section 6.8).

A. A union var may hold values of different types.

B. Suppose we want a variable that can hold one of an int, double, string, or boolean.

Unions, cont'd

```
typedef union {
    int int_val;
    double double_val;
    char* string_val;
    unsigned char bool_val;
} GenericValue;
```

Unions, cont'd

C. Syntactically, unions are declared and accessed in precisely the same way as structs.

1. Union fields are accessed with '.'.

2. Pointer-to-union fields are accessed with '->'.

Unions, cont'd

D. The semantic difference is that a struct value contains *all* its data fields, whereas a union value contains *one of* its data fields.

Unions, cont'd

E. As explained on pages 147-148 of K&R, *"It is the programmer's responsibility to keep track of which type is currently stored in a union; ..."*

F. For this reason, union types are often *tagged* to keep track of the current value.

Unions, cont'd

1. Union tags are frequently implemented with enums.

2. E.g.,

Unions, cont'd

```
typedef enum {
    INT, DOUBLE, STRING, BOOL
} ValueTag;

typedef struct {
    ValueTag tag;
    GenericValue val;
} TaggedGenericValue;
```

Unions, cont'd

3. Some example usage

Unions, cont’d

```
void PrintTaggedGenericValue(TaggedGenericValue v)
  switch (v.tag) {
    case INT:
      printf("%d0, v.val.int_val);
      break;
    case DOUBLE:
      printf("%f0, v.val.double_val);
      break;
    case STRING:
      printf("%s0, v.val.string_val);
      break;
    case BOOL:
      printf("%s0, v.val.bool_val ? "true" : "false
  }
}
```

```
main() {
  TaggedGenericValue tval;

  tval.val.int_val = 10;
  tval.tag = INT;
  PrintTaggedGenericValue(tval);

  tval.val.bool_val = 0;
  tval.tag = BOOL;
  PrintTaggedGenericValue(tval);

}
```

Unions, cont'd

4. The idea is that the union type appears in the context of a struct that has information indicating which union value is current.

XVII. Bit-fields (K&R Section 6.9).

A. Bit-fields provide access to individual binary bits in a word of memory.

B. Such access can save space, e.g., bool as bit.

C. Also provide direct access to hardware devices.

D. We'll talk more about bit-fields later.

XVIII. A culminating example.

A. Attached code listings illustrate key concepts.

B. The commenting style is `doxygen`-compliant.

C. NOTE: Unanswered questions `person-record-test.c`.