Structures

- A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. The variables contained in a structure are knows as its members.
- Use the `struct` keyword followed by an optional tag to declare a structure. For example, in the following declaration, `point` is the tag and `x, y` are members of the structure.

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

struct point pt;
```

- A structure can be initialized by following its definition with a list of initializers, each a constant expression, for the members:

```
struct point pt = { 100, 200 };
```

The Dot Operator

- A member of a particular structure is referred to in an expression by a construction of the form

  structure-name.member

  The structure member operator . (dot) connects the structure name and the member name. For example:

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

- Structures can be nested. For example:

  ```
  struct rect {
      struct point bottomLeft;
      struct point topRight;
  };

  struct rect screen;
  ```

  Then `screen.topRight.x` refers to the `x` coordinate of the `topRight` member of screen.

Operations on Structures

- The only legal operations on a structure are
  - copying it or assigning to it as a unit,
  - taking its address with &,
  - and accessing its members using the dot . operator.
- Copy and assignment include passing arguments to functions and returning values from functions as well.
- Structures may not be compared.
- A structure may be initialized by a list of constant member values.

Pointers to Structures

- Structure pointers are just like pointers to ordinary variables. For example:

  ```
  struct point *pp;
  ```

  says that pp is a pointer to a structure of type `struct point`. If pp points to a point structure, `*pp` is the structure, and `(*pp).x` and `(*pp).y` are the members. To use pp, we might write, for example,

  ```
  struct point origin, *pp;
  pp = &origin;
  printf("origin is (%d,%d)\n", (*pp).x, (*pp).y);
  ```

- Pointers to structures are so frequently used that an alternative notation is provided as a shorthand. If p is a pointer to a structure, then

  p->member-of-structure

  refers to the particular member. So we could write instead

  ```
  printf("origin is (%d,%d)\n", pp->x, pp->y);
  ```

Precedence of . and -> operators

- Both . and -> associate from left to right, so if we have

```
struct rect r, *rp = &r;
//then these four expressions are equivalent:
r.pt1.x
rp->pt1.x
(r.pt1).x
(rp->pt1).x
```

- The structure operators . and ->, together with () for function calls and [] for subscripts, are at the top of the precedence hierarchy and thus bind very tightly.
- For example, given the declaration:

```
struct {
    int len;
    char *str;
} *p;
```

  What happens with: `++p->len`, `(++p)->len`, `(p++)->len`.
  What happens with: `*p->str`, `*p->str++`, `(*p->str)++`, `*p++->str`.
- See example `lab/C-examples/structures/structs-ex0.c`

More Structure Examples

- The following declaration creates a new type `struct record`.

```
struct record {
    char *name;
    long int studentId;
    char *major;
};
```

- Now we can use the new type to declare variables, arrays, pointers etc.
  For example:

```
struct record r;  /* static declaration */
struct record students1[1000]; /* a static or fixed  1-dim. array*/
struct record students2[100][1000]; /* a static 2-dim. array*/
struct record *student; /* a pointer to a struct record */

n = 1000;
student = (struct record *) malloc(sizeof(struct record)*n);

for (i=0; i<n; i++) {
    student[i].studentId = i; /* give a fake id */
    student[i].major = NULL; /* no major yet */
    student[i].name = (char *) malloc(sizeof(char)*MAX_NAME_LENGTH);
}
```

Self-referential Structures

- It is valid for a structure to contain a pointer to itself! For example, here is the declaration of a structure to represent a node in a singly linked list:

```
struct node {
    int keyValue;
    struct node *next;
};
```

- Here is some code that uses the above node structure.

```
struct node *head = (struct node *) malloc(sizeof(struct node));

(*head).keyValue = 10;
/* the following is equivalent to the above */
head->keyValue = 10;

/* undefined as malloc doesn't initialize the allocated memory */
head->next = ????
```

Typedef

- C provides a facility called typedef for creating new data type names. It does not create a new data type, just a new name for an existing type.

```
typedef int Length;

Length maxlen = 100,
Length *lengths[];

typedef char *String;
String s = (String) malloc(sizeof(char) * maxlen);
```

- Use typedef for simplifying declarations and make the program more readable. For example, we can use it to simplify the structure declarations for the node in a singly linked list example:

```
typedef struct node Node;
typedef struct node *NodePtr;
```

- Typedefs parametrize a program against portability problems. If typedefs are used for data types that may be machine-dependent, only the typedefs need change when the program is moved.

Singly Linked List (1)

```
#ifndef __SINGLYLINKEDLIST_H
#define __SINGLYLINKEDLIST_H
/* C-examples/singlyLinkedList/SinglyLinkedList.h */
#include <stdio.h>
#include <stdlib.h>

typedef struct node Node;
typedef struct node *NodePtr;
typedef int ItemType;

struct node {
    ItemType  item;
    NodePtr next;
};

/* prototypes */
NodePtr addAtFront(NodePtr L, NodePtr node);
NodePtr reverseList(NodePtr L);
void printList(NodePtr L);

#endif /* __SINGLYLINKEDLIST_H */
```

Singly Linked List (2)

```
/* C-examples/singlyLinkedList/SinglyLinkedList.c */
/* SinglyLinkedList.c: Functions to manipulate a linked list.  */
#include <stdio.h>
#include <stdlib.h>
#include "SinglyLinkedList.h"

NodePtr addAtFront(NodePtr L, NodePtr node)
{
    if (node == NULL) return L;
    if (L == NULL) {
        L = node;
        node->next = NULL;
    } else {
        node->next = L;
        L = node;
    }
    return L;
}

/* ... */
```

Singly Linked List (3)

```
/* C-examples/singlyLinkedList/SinglyLinkedList.c  (contd) */
NodePtr reverseList(NodePtr L)
{
    NodePtr list = NULL;
    while (L) {
        NodePtr tmp = L;
        L = L->next;
        tmp->next = list;
        list = tmp;
    }
    return list;
}

void printList(NodePtr L)
{
    while (L) {
        printf(" %d -->",L->item);
        L = L->next;
    }
    printf(" NULL \n");
}
```

Using the Singly Linked List class

```
/* C-examples/singlyLinkedList/SimpleTest.c */
#include <stdio.h>
#include <stdlib.h>
#include "SinglyLinkedList.h"

int main(int argc, char **argv)
{
    int i;
    int n;
    NodePtr list, node;
    if (argc != 2) {
            fprintf(stderr, "Usage: %s <list size> \n",argv[0]);
            exit(1);
    }
    n = atoi(argv[1]);

    list = NULL;
    for (i=0; i<n; i++) {
        node = (NodePtr) malloc(sizeof(Node));
        node->item = i;
        if (node == NULL) {
            printf("Error allocating node for linked list\n");
            exit(1);
        }
        list = addAtFront(list, node);
    }
    printList(list);
    list=reverseList(list);
    printList(list);
    return 0;
}
```

Freeing the list

- **Inclass Exercise**. Add a new function to free a singly linked list.

  ```
  void freeList(NodePtr L) {
  }
  ```

- What happens if the list isn't null-terminated properly? Suppose the last node has a bad pointer? Or if some nodes get bypassed due to bugs in the list code?
- We need help! Valgrind is a memory-checker tool that catches memory errors (reading/writing to invalid memory locations due to bad pointers) and memory leaks (memory that is allocated but not freed that the program cannot reference).

Doubly-Linked List Example (1)

- To create a more flexible list than the previous example, we will describe the setup for a doubly linked list with a header represented by the following structure. The list structure keeps track of the head and tail nodes of the list as well as its size.

```
typedef struct list List;
typedef struct list *ListPtr;
struct list {
    int size;
    NodePtr head;
    NodePtr tail;
};
```

- Each node contains pointers to previous and next nodes as well as to the data stored within the node.

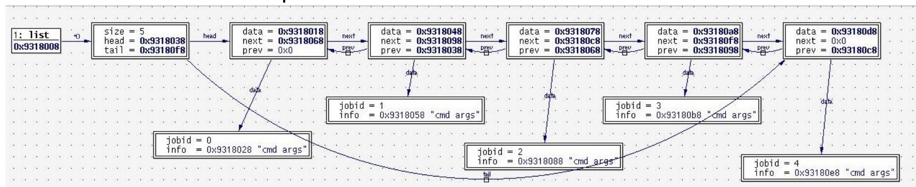
```
typedef struct node Node;
typedef struct node *NodePtr;
struct node {
    JobPtr data;
    NodePtr next;
    NodePtr prev;
};
```

Doubly-Linked List Example (2)

- Instead of storing a primitive type, this list will store a pointer to a struct as the data in the node. For our example, the data stored is a *job*. Each *job* has an *id* and some associated *info*.

```
typedef struct job  Job;
typedef struct job *JobPtr;
struct job {
    int jobid;
    char *info;
};
```

- Here is an example:



Doubly-Linked List Example (3)

- We will provide the following interface for the List file:

```
/* from List.h */
ListPtr createList(); /* constructor */
void freeList(ListPtr L); /* destructor */

int getSize(ListPtr L);
Boolean isEmpty(ListPtr L);

void addAtFront(ListPtr list, NodePtr node);
void addAtRear(ListPtr list, NodePtr node);

NodePtr removeFront(ListPtr list);
NodePtr removeRear(ListPtr list);
NodePtr removeNode(ListPtr list, NodePtr node);

NodePtr search(ListPtr list, int key);
void reverseList(ListPtr  L);
void printList(ListPtr L);
```

Doubly-Linked List Example (4)

- We will provide the following interfaces for the Node and Job file:

```
/* from Node.h */
NodePtr createNode (JobPtr data);
void freeNode(NodePtr node);

/* from Job.h */
JobPtr createJob (int, char *);
void freeJob(JobPtr job);
char *toString(JobPtr);
```

The Doubly-Linked List Example (5)

- A partially complete doubly liked list code is included in the examples folder: `C-examples/doublyLinkedLists/`
- The program consists of following main files (with other supporting files):

`List.h`	declaration of the List interface
`List.c`	implementation of the List interface
`Node.h`	declaration of the Node interface
`Node.c`	implementation of the Node interface
`Job.h`	declaration of the Job interface
`Job.c`	implementation of the Job interface
`SimpleTest.c`	a simple program that uses doubly linked lists
`UnitTestList.c`	a unit test skeleton program (incomplete)
`RandomTestList.c`	a random test program

Using the Doubly-Linked List

- **In Class Exercise 1**: Create a doubly linked list named L1 and create and add two nodes to it with jobs names "job1" with id 16000 and "job2" with id 3200.
- **In Class Exercise 2**: Write a declaration for an array of doubly-linked lists of size `n` and allocate space for it and fill it with pointers to `n` empty lists.
- **In Class Pondering** How would you declare a list of lists? A queue of lists? A stack of lists? A list of queues?

Testing

- Use unit-tests to increase confidence in your code.
- Check for boundary conditions!
- Use assertions to check that the program satisfies certain conditions at particular points in its execution. Assertions can be of three types:
  - Preconditions: at start of a function
  - Postconditions: at the end of a function
  - Invariants: over a block of code, for example, a loop
- In C, we can use the `assert` macro from the `assert.h` header file as shown below:

```
#include <assert.h>
assert (size <= LIMIT)
```

- If the assert fails, the program is terminated with an error message similar to shown below:

```
a.out: test.c:9: main: Assertion `size < 100' failed.
Aborted (core dumped)
```

Unit Tests for Doubly Linked List

- We will write our own version of assert that just breaks out of the current function, which represents a test case for our unit test program.

```
#define myassert(expr) if(!(expr)){ fprintf(stderr,
   "\t[assertion failed] %s: %s\n", __PRETTY_FUNCTION__,
                              __STRING(expr)); return FALSE; }
```

- Let's review the file:

  `C-examples/doublyLinkedLists/UnitTestList.c`

  for a skeleton of a unit test program for the doublylinked list. You will complete this in your project!

Comments on Doubly Linked List Code

- How would you use the code for storing data item of another type?
- How can we store different data types without modifying the code? (generic programming)
- How would use use the doubly linked list code in multiple projects?
- How can we use compiled code in multiple projects without recompiling and including into each project? (libraries and plugins)
- Coming soon to a classroom near you!

Union (1)

- A union is a variable that may hold (at different times) objects of different types and sizes, with the compiler keeping track of size and alignment requirements.
- Consider the following union declaration:

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

- Then if we have a variable named `utype` to keep track of the type of data stored, we could write the following code:

```
if (utype == INT)
    printf("%d\n", u.ival);
if (utype == FLOAT)
    printf("%f\n", u.fval);
if (utype == STRING)
    printf("%s\n", u.sval);
else
    printf("bad type %d in utype\n", utype);
```

Union (2)

- The intended purpose of union is to save memory by using the same memory region for storing different objects at different times.
- In effect, a union is a structure in which all members have offset zero from the base, the structure is big enough to hold the “widest” member, and the alignment is appropriate for all of the types in the union.
- The same operations are permitted on unions as on structures: assignment to or copying as a unit, taking the address, and accessing a member.
- A union may only be initialized with a value of the type of its first member.

Another Union Example (1)

- We will use a structure with a union inside it to store one of three types of data values.

```
enum utype{
  INT,
  DOUBLE,
  STRING
};
struct data {
    union udata {
        int ival;
        double dval;
        char *sval;
    } data;
    enum utype storedType;
};
```

- Without the union, the structure would need to store three separate variables, one for each type.

Another Union Example (2)

- Now we can use a switch statement to access the right type of values.

```
void printData(struct data value)
{
    switch (value.storedType) {
        case INT: printf("ival = %d \n", value.data.ival); break;
        case DOUBLE: printf("fval = %f \n", value.data.dval); break;
        case STRING : printf("sval = %s \n", value.data.sval); break;
    }
}
```

- The following shows a sample usage:

```
    myCloset.data.ival = 10;
    myCloset.storedType = INT;
    printData(myCloset);

    myCloset.data.dval = 3.14159;
    myCloset.storedType = DOUBLE;
    printData(myCloset);

    myCloset.data.sval = "Cool!";
    myCloset.storedType = STRING;
    printData(myCloset);
```

Recommended Assignments

- Read Sections 6.1 through 6.9. Skim though Sections 6.5 and 6.6 as they are more difficult.
- Techniques introduced:
  - Using *doxygen* tool for javadoc style comments.
  - Using `assert` macros for effective unit-testing.
  - Using `valgrind` tool to check for memory errors and leaks.

Recommended Exercises

1. Write the header file for declaring a queue that uses dynamically allocated nodes? What typical operations would you provide. Write their prototypes.
2. Write the header file for declaring a stack that uses dynamically allocated nodes? What typical operations would you provide. Write their prototypes.
3. Compare an array-based implementation for a queue/stack with the dynamically allocated version.
4. Write the declaration for a list of queues.

References

- http://haifux.org/lectures/239/
- http://valgrind.org