Pointers and Objects

CSE 373 - Data Structures
April 3, 2002

Readings and References

- Reading

- Other References
  - *Pointers and Memory*, by Parlante
  - Chapters 5 and 6, *The C Programming Language*, Kernighan and Ritchie

What is a pointer?

- A pointer is a reference to another item
  - the other item is sometimes called the "pointee"
  - the pointee is often a variable
  - the pointee can also be a function (a procedure)
- The contents of a pointer tell you where to look in order to find the object of interest
- The declaration of the pointer says what it is supposed to point to

Some declarations

```
int num;                /* an integer */
int *numP;              /* pointer to integer */

double sum;             /* a double value */
double *sumP;           /* pointer to double */

struct Symbol mySym;    /* a Symbol */
struct Symbol *symP     /* pointer to Symbol */
```

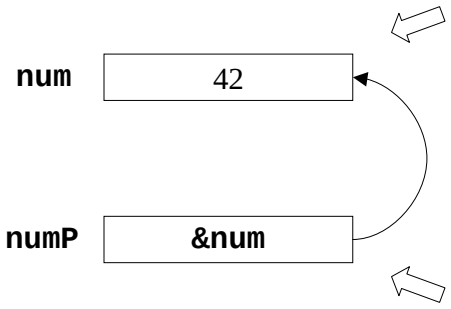
Reference and dereference

- Use "&" operator to take the address of a variable and store it in a pointer variable
  - `numP = &num;`
  - `sumP = &sum;`
  - `symP = &mySym;`
- Then use "*" operator to dereference a pointer
  - `*numP = 42` *is the same as* `num = 42`
  - `*sumP = 17.0` *is the same as* `sum = 17.0`
  - `(*symP).val = 2` *is the same as* `mySym.val = 2`

Pointers and Pointees

```
numP = &num;
sumP = &sum;
symP = &mySym;

*numP = 42;
*sumP = 17.0;
(*symP).val = 2;

printf("%i %3.1f %lu\n",num,sum,mySym.val);
printf("%i %3.1f %lu\n",*numP,*sumP,(*symP).val);
-----------------
42 17.0 2
42 17.0 2
```


Example from Pointers & Memory



A simple **`int`** variable. The current value is the integer 42. This variable also plays the role of pointee for the pointer **`numP`** below.

A pointer variable. The current value is a reference to the pointee **`num`** above.

What form does data take?

- Integers
  - -1, 0, 255, 65535, ...
- Floating point
  - 1.5, 3.14159, 1E75, …
- character strings
  - "abc", "def"
- and that was about it in the old days

But real data is more complex

- Airplane definition
  - › engine count, crew count, passenger capacity, range, operating cost per seat mile, …
- Student record
  - › name, student id, major, school address, home address, credits to date, current enrollment, …
- Major fields of study
  - › responsible department, curriculum, students, ...

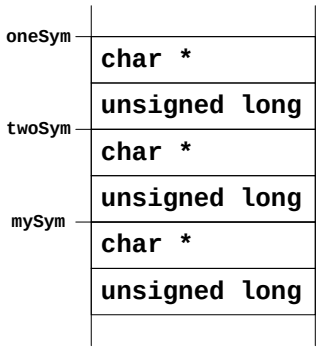
Why have **`struct`**s?

- Because the logical objects that you use in your programs are more complex than just a single int or double value
- A structured block lets you manipulate related data as one element

```
struct Symbol {
  char *name;
  unsigned long val;
};

struct Symbol oneSym;
struct Symbol twoSym;
struct Symbol mySym;

oneSym.name = "one";
oneSym.val = 1;

twoSym.name = "two";
twoSym.val = oneSym.val+1;
```


What is a **`struct`** variable?

- A single variable declared as a **`struct`** refers to a particular block of memory
- the individual fields are at fixed offsets from the start of the block



What can you do with a **`struct`**?

- The legal operations on a structure are
  - › accessing its members
  - › copying it or assigning to it as a unit
  - › taking its address with &

```
struct Symbol oneSym;
struct Symbol twoSym;
struct Symbol mySym;

oneSym.name = "one";
oneSym.val = 1;

twoSym.name = "two";
twoSym.val = oneSym.val+1;

mySym = twoSym;
```

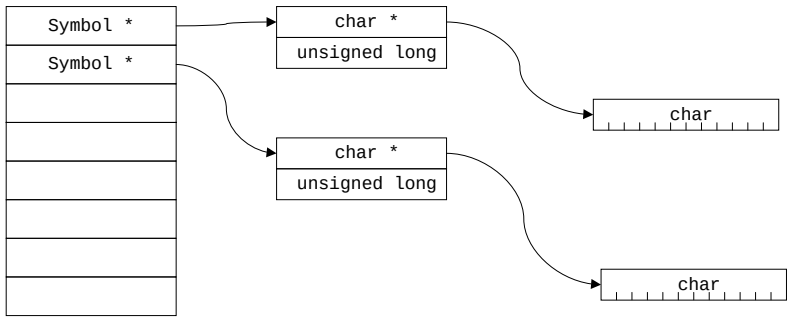
How can **`struct`**s be costly?

- Copying a struct is a nice automatic feature
  - › but it can lead to a lot of copying
- Our Symbol structs only require a few bytes
  - › but imagine the size of some of the other examples - airplanes, student records, department descriptions
- Copying complete **`struct`**s can get very costly very quickly

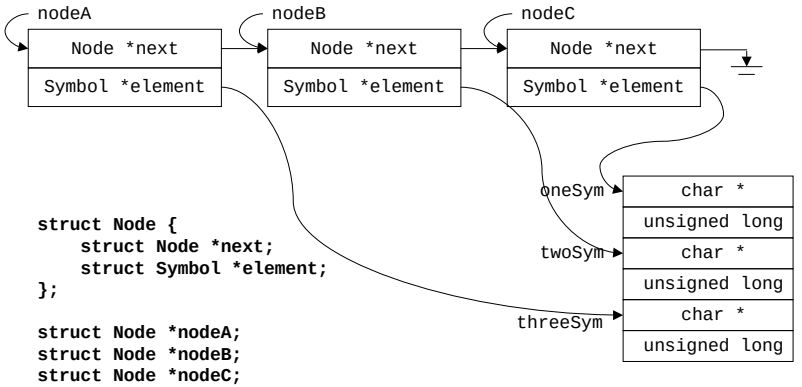
Pointers to the rescue

- Take the address of a struct variable and store it in a pointer variable
- Then you can manipulate the pointers, leaving the original data where it is and just moving pointer values around
- An array of pointer values is one way to define a list of objects (struct variables)

A short array of Symbol pointers



Pointers allow you to link objects




```
struct Node {
    struct Node *next;
    struct Symbol *element;
};

struct Node *nodeA;
struct Node *nodeB;
struct Node *nodeC;
```

```
nodeA = malloc(sizeof(struct Node));
FatalErrorMemory(nodeA);

nodeB = malloc(sizeof(struct Node));
FatalErrorMemory(nodeB);

nodeC = malloc(sizeof(struct Node));
FatalErrorMemory(nodeC);

(*nodeA).next = nodeB;
(*nodeA).element = &oneSym;

nodeB->next = nodeC;
nodeB->element = &twoSym;

nodeC->next = NULL;
nodeC->element = &threeSym;

nodeP = nodeA;
while (nodeP != NULL) {
    printf("%p (%s) points to %p\n",
        nodeP,nodeP->element->name,nodeP->next);
    nodeP = nodeP->next;
}
```

Three Linked Nodes

```
0x80498f0 (one) points to 0x8049900
0x8049900 (two) points to 0x8049910
0x8049910 (three) points to (nil)
```

Structure of Homework

- Each programming project will require you to implement a small set of functions to implement the particular data type
- You will be given main.c and a header file describing the functions to implement
  - › based very closely on the functions in the book
- You are also supplied with some utility headers

Sum is an example homework

- If Sum had been given as a homework you would have gotten
  - › mainSum.c - the driver
  - › sum.h - the function prototype
  - › element.h - utility header for ElementType
  - › reporter.h - simple error reporting macros
- You would have written sum.c
  - › implement the sum function for Symbol objects

sum for **`int`**s (from first lecture)

- Find the sum of the first **`num`** integers stored in an array **`v`**.

```
int sum( int v[ ], int num){
    int temp_sum, i;
    temp_sum = 0;
    for ( i = 0; i < num; i++ )
        temp_sum += v[i] ;
    return temp_sum;
}
```

sum for ElementType objects

```
#include "element.h"
#include "sum.h"
int sum(ElementType v[], int num, ElementGetValue gv) {
  int temp_sum;
  int i;
  temp_sum = 0;
  for (i=0; i<num; i++) {
    temp_sum += (int)(*gv)(v[i]);
  }
  return temp_sum;
}
```

- Note that sum uses the function pointed to by parameter gv to get the value from each item pointed to by the pointer entries in v[]

element.h

```
typedef void *ElementType;
typedef void ElementPrintLabel(ElementType e);
typedef unsigned long ElementGetValue(ElementType e);
typedef int Comparator(ElementType a, ElementType b);
```

- ElementType is a pointer to a data object
- ElementPrintLabel, ElementGetValue, and Comparator are all functions
- void * is cast to the proper type in each function depending on the implementation

reporter.h

```
#define FatalErrorMemory(var) \
if ((var)==NULL) {printf("Fatal Error - Memory …

#define FatalErrorBound(v,b) \
if ((v)>(b)) {printf("Fatal Error - Bound …

#define FatalErrorObjectNotFound(v,b) \
if ((v)==NULL) {printf("Fatal Error - \"%s\" Not Found …
```

- Macros that provide message and exit for memory allocation errors, bounds checks, and functions that return NULL if object not found