ECS 36A, May 16, 2024
Announcements

• Grades for the midterm are posted
• Thursday and Friday discussion sections will go through the midterm
• Homework 3 is out now
Structures

• Data structure used to group elements of a different type together
• Example: student registration number database
  • See element below

```c
struct student {
    char *name; /* student name */
    int regnumber; /* registration number */
};
```

Referring to a Structure

Here’s how you declare a variable of the structure:

```c
struct student xyzzy, *pxyzzy;
```

It’s clumsy to write that, so you can define an alias for the type:

```c
typedef struct student STUDENT;
```

The latter essentially produces a new type, STUDENT, that can be used wherever struct student can:

```c
STUDENT xyzzy, *pxyzzy;
```
Another Declarations

```c
struct student {
    char *name;    /* student name */
    int regnumber; /* registration number */
} xyzzy, *pxyzzy;
```

- Declares type `struct student` with 2 fields, `xyzzy` (an instance of `struct student`) and `pxyzzy` (a pointer to an instance of `struct student`)
And Now, With a Typedef

typedef struct student {
    char *name;    /* student name */
    int regnumber; /* registration number */
} STUDENT;

STUDENT xyzzy, *pxyzzy;

This defines a new type, STUDENT, which is the same as the type struct student. Here xyzzy is a variable of type STUDENT and pxyzzy is a pointer to an instance of STUDENT.
But Be Careful

- `typedef` defines an alias for a type
- `#define` does textual substitution

```c
typedef int *PINT;
PINT a, b, c;
```

- Now `a`, `b`, and `c` are all pointers to integers

```c
#define PINT int *
PINT a, b, c;  /* becomes int * a, b, c; */
```

- Now `a` is a pointer to an integer, and `b` and `c` are integers
Linked List

• A list composed of instantiations of structures
  • One element is whatever is to be sorted (int, for us)
  • Another element is a pointer to the next element; NULL if none
Structure for This List

```c
struct node {
    int num;
    struct node *next;
};
struct node *list;
```

- The `struct node` contains:
  - `num`: holds the integer that you read in.
  - `next`: holds the pointer to the next element in the linked list; it’s NULL at the end.
- `list`: points to the first element of the list.
Changing How Memory Is Allocated

• Now you can allocate memory one element (“node”) at a time
• Insertion at beginning is like this (see ”linked.c”, ll. 72–76):
  ```c
  new->next = first;
  list = new;
  ```
• Insertion in the middle between prev and succ is (see “linked.c”, ll. 78–97):
  ```c
  new->next = succ;
  prev->next = new;
  ```
• Insertion at the end, after last (see “linked.c”, same lines as above) :
  ```c
  last->next = new;
  ```
Insertion

headList

1

5

12

23

49
Insertion: At the Beginning of the List

First, change the pointer in the new node to point to the head of the list (where headList points; just copy the pointer)
Insertion: At the Beginning of the List

Next, change the pointer to the head of the list to point to the new node
Code for This

• new is a pointer to the new node, headList points to the head of the list
• First, make new point to the old head of the list
  \[ \text{new} \rightarrow \text{next} = \text{headList}; \]
• Next, make the pointer to the head of the list point to new
  \[ \text{headList} = \text{new}; \]
Insertion: In the Middle of the List

First, scan down the list until you reach the node before which the new node goes.

new node goes after this one
Insertion: In the Middle of the List

Change the pointer in the new node to point to the first node \textit{after} where the new node is to go

new node goes after this one
Insertion: In the Middle of the List

Next, have the pointer in the node *before* where the new node is to go point to the new node

New node goes after this one
Code for This

• new is a pointer to the new node, headList points to the head of the list, and p is a pointer to node
• First, find the node that new goes after
  for(p = headList;
      p != NULL && p->next < new->next;
      p = p->next)
  /* do nothing */;
• Next, change the pointer in new to point to the node after where this one goes
  new->next = p->next;
• Finally, make the node p points to point to new
  p->next = new;
Insertion: At the End of the List

First, scan down the list until you reach the end node.

new node goes after this one

headList
Insertion: At the End of the List

Next, change the pointer in the end node to point to the new node

new node goes after this one
Code for This

• new is a pointer to the new node, headList points to the head of the list, and p is a pointer to node
• First, find the node at the end
  for(p = headList;
      p != NULL && p->next != NULL;
      p = p->next)
  /* do nothing */;
• Next, change the pointer in what p points to to point to new
  p->next = new;
• This may be an excess, but make sure new’s pointer field is NULL
  new->next = NULL;
Multiple Arrays

• Need to store several data of different types about something
• Example: sort planets by their diameters
• Use 2 arrays
  • char *names[9]
  • int diameters[9]
• When sorting, need to keep both arrays aligned
  • So when swapping 2 elements of array diameter, the corresponding elements of array names must also be swapped
• Alternate approach: use structures!
Same with Structures

• Instead of 2 arrays, combine into one structure for each element, and use an array of structures

```c
struct celestial {
    char *name; /* pointer to name of planet */
    int diameter; /* diameter of planet in km */
} planets[9];
```

• This allocates space for 9 planets

• When you swap elements, you only need to swap one, not two, as in the parallel arrays case
**gdb**

- A dynamic debugger
- To run it, compile your program with the \texttt{-g} option
  - This adds in debugging information \texttt{gdb} uses
  - You can use \texttt{gdb} without it but it simplifies the use greatly
- Then load it into \texttt{gdb} by:
  \[
  \texttt{gdb \ executable}
  \]
- Note you use the executable file and \textit{not} the source code file
  - You can also load the executable once \texttt{gdb} starts
Inside the *gdb* Shell

• Once started, you get a prompt “(gdb)”
• If you forgot to name the executable in the command line:
  \[(gdb)\] *file executable*
• One other handy feature
  \[(gdb)\] *help*
• You will get a list of commands you can ask for help on
• Then type
  \[(gdb)\] *help command*
Executing the program

• Type:

    (gdb) run arg_1 . . . arg_n

• This runs the program with command line arguments \(arg_1\) through \(arg_n\)
  • If there are no command line arguments, just type ``run’’

• If there are no problems, the program runs to completion

• If the program stop with a message like this, there’s a problem

Program received signal SIGSEGV, Segmentation fault.
0x00000555555551b5 in nfact (n=<error reading variable: Cannot access memory at address 0x7fffffff7fefe>) at nfact2.c:12
Stopping the Program Before It Ends

• A *breakpoint* causes the execution to stop at that point

• Here’s an example:

  (gdb) break 15
  Breakpoint 1 at 0x5555555551b8: file nfact2.c, line 15.

• This causes execution to stop when it reaches line 15
  • If you have multiple source files, name the file before the number:
  
  (gdb) break nfact2.c:15

• It shows some useful information

  Breakpoint 1, nfact (n=15) at nfact2.c:15
  15 x = nfact(n+1);
Conditional Breakpoints

• Causes a breakpoint to stop execution when a condition is met

• Here’s an example:

  (gdb) break 15 if n >= 20

  Breakpoint 1 at 0x5555555551b8: file nfact2.c, line 15.

• This causes execution to stop when it reaches line 15 and n is 20 or more
  • If you have multiple source files, name the file before the number:

    (gdb) break nfact2.c:15 15 if n >= 20
What Can You Do When Stopped?

• You can continue the execution from the breakpoint:
  \[(gdb) \text{ continue}\]

• You can execute one statement at a time to step through the program
  • If it encounters a function, it goes into that function and executes one statement at a time
    \[(gdb) \text{ step}\]
  • n (next) is like s but treats the function as part of the statement and does not go into it
    \[(gdb) \text{ next}\]
Printing Values

• You can print the value of an expression

  (gdb) print expression

• If you prefer hexadecimal

  (gdb) print/x expression
Watchpoints

• Like breakpoints, but keyed to variables

\[(gdb) \text{ watch } x\]

• Whenever x changes values, the program stops and \textit{gdb} prints old and new values of x
Other Useful Commands

• backtrace
• where
  • These show the stack, that is, the functions that have been called and not yet returned
• delete 2
  • Delete breakpoint 2 (or watchpoint 2)
• info breakpoints
  • List the breakpoints (and watchpoints)
• info frame
  • Show the current frame