ECS 36A, May 14, 2024

Announcements

- We'll post the grades for midterms some time tomorrow
- Thursday and Friday discussion sections will go through the midterm
- Homework 3 will be out later today

Another Recursive Program: sort.c

- This sorts integers by finding the smallest number and putting it at the beginning
- Basic idea:

```
if number of elements in list is 1 or 0:
   list is sorted – just return
find the smallest number in the list
swap it and the first number
sort the rest of the list
```

Problem

- sort.c reads from an array of known length
- User must enter numbers into the program
- The compiler can compute the length (or the user can enter it)

So how do we get around this?

Dynamic Memory Allocation

- Static memory allocation occurs when you declare a variable

```
int num;
```

- Compiler creates space for this variable
- There is also a pool of memory (the “heap”) that is available but initially unused
- Dynamic memory occurs when you obtain memory space from the heap
  - Allocate: obtain the space from the pool
  - Allocation: the space you get
  - Deallocate, free: releaseg memory that has been allocated; it goes back to the heap

A Useful Operator

- To get the number of bytes in a data type, use sizeof
- Example: on a 32-but machine:
  - sizeof(char) is 1
  - sizeof(int) is 4
  - sizeof(float) is 4
  - sizeof(double) is 8
- Works for variables, too
  - if *a* is an int, sizeof(a) is 4

Allocation Functions: *malloc*()

- Basic function

```
void *malloc(size_t space)
```

- Allocate *space* bytes of memory, returning its address; returns NULL if not available
  - Type size_t is same as unsigned int
- Declared void * so that it can be coerced into any type of pointer

```
char *p;
if ((p = (char *) malloc(100)) == NULL)
    error handling
```

Allocation Functions: *realloc*()

- Enlargening space already allocated (say pmem points to it):

```
void *realloc(void *pmem, size_t nbytes)
```

- This allocates *nbytes* of space, and the contents of **pmem* are copied into the beginning of the new space
  - The new space may simply extend what *pmem* points to
  - Or, it may be completely new space, in which case what *pmem* points to is deallocated
  - If insufficient memory available, returns NULL and leaves the space *pmem* points to untouched, neither moved nor deallocated

Allocation Functions: *calloc*()

- Variant

```
void *calloc(size_t nelt, size_t space)
```

- Like malloc, but:
  - Gives you space in terms of elements and size of element, rather than a number of bytes
  - Memory is zeroed out; malloc() does not do so, and whatever is in that memory before call to malloc() is there once allocated

Allocation Functions: *realloc*()

- Common way to use this:

```
if ((pmem = realloc(pmem, 1000)) == NULL) . . .
```

  - On success, *pmem* now points to a chunk of memory of size 1000 bytes
  - On failure, *pmem* is now NULL — and you lose the address of the memory pmem used to point to

- Here's the right way:

```
tempptr = realloc(pmem, 1000);
if (tempptr == NULL) error handling;
else pmem = tempptr;
```

Deallocation Function: *free*()

- To release memory allocated by one of the allocation functions, use:

```
void free(void *pmem)
```

- If *pmem* is NULL, this does nothing
- Do *not* free memory that has already been freed!
  - Called a *double free error* and can often be a vulnerability
  - In all cases, the result is undefined

But Be Careful!

```
char a[100]
```

- You can get the size of an array like this:

```
sizeof(a)
```

- This works *because* `a` *is a pointer constant*

However . . .

```
char *a;
if ((a = malloc(sizeof(char) * 100)) == NULL)
    perror("bad malloc");
```

- Tempting to get the size of the allocated space like this:

```
sizeof(a)
```

  - Here, `a` is a pointer *variable*, so sizeof returns the number of bytes in that pointer, *not* the size of the array!
- To get the number of bytes in an array, use

```
sizeof(a[0]) * 100
```

  where 100 is the number of elements in the array
  - The `a[0]` is one element; works as all elements are of the same type

Another Recursive Program: usort1.c

- Problem with earlier sort.c: numbers are embedded in program
- Better: have users enter the numbers
- Basic idea:

```
ask user how many numbers they want sorted
allocate the space
read in that many integers – if EOF entered, quit at once
```

Another Recursive Program: usort15.c

- Problem with usort1.c: users have to say how many numbers they want sorted
- Better: let users enter the numbers to be sorted and have the computer count
- Basic idea:

```
allocate initial space
read in that many integers – if EOF entered, sort what you have
    check that there is room to add the entered number
    if not, reallocate space to increase room
```

Example: Dynamically Allocated Input Buffer

- Problem: *fgets* requires a maximum length to input
  - So it will fit into the input buffer without overflow
  - May read only part of a line
- Solution: write a function that will allocate space for any length line

Requirements

- Function must be able to input line of any length without knowing what that length may be
- Interface needs to be as similar to that of *fgets* as possible

Solution #1: For Interface

```
char *dyngets(char *buf, int n, FILE *fp)
```

- `char *buf`
  - If non-NULL, pointer to input buffer; *dyngets* acts exactly like *fgets*
  - If NULL, one line is stored in allocated space
- `int n`
  - size of array `buf`
  - *ignored* if `buf` is NULL
- `FILE *fp`
  - File pointer to source of input

Solution #2: Allocation

- Create a buffer that is preserved across calls
  - Use a static variable to point to this and the size of the buffer
- Static variable in function keeps variable and its value around after function returns

General Structure

- If buf is not NULL, call *fgets* and return its value
- Otherwise:
  1. Read a character; if end of file, go to step 6
  2. If there is room in the internal buffer, put character in and go to step 1
  3. If there is not room in the internal buffer, allocate (or reallocate) an internal buffer of length INCREMENT + length of current internal buffer
  4. Add the new INCREMENT to the length of the internal buffer
  5. Go to step 2
  6. Return pointer to internal buffer

Program Structure

- Main routine is *dyngets*
- It calls a function to insert the character
  - Allocation is done here

Main Routine

- Check to see if buf is non-NULL; if so, call *fgets* and return its return value
- Read characters, calling the insertion function for each
  - If EOF is read as the first character of the line, return NULL
  - Otherwise, tack on a newline if it is present
  - Terminate the internal buffer with ‘\0’
- Return pointer to internal space

Character Insertion Routine

- First, see if internal buffer is completely full
  - If so, increment the allocated space number
  - If nothing allocated yet, use malloc() to allocate the desired space
  - Otherwise, use realloc() to reallocate the space
- Append the character to the input line

Compiling With a Program

List multiple files for gcc

- For *dyngets*:

```
gcc –ansi –pedantic –Wall –g –o mcat mcat.c dyngets.c
```

- What is happening: for each file
  - Run the C preprocessor on the file to handle thee macros
  - Compile the file to produce an assembly language “.s” file
  - Assemble the resulting “.s” file to produce an object “.o” file
- Then for all files:
  - The linking loader merges all the “.o” files and some system libraries into an executable

Another Recursive Program: usort1.c

- Problem with earlier sort.c: numbers are embedded in program
- Better: have users enter the numbers
- Basic idea:

```
ask user how many numbers they want sorted
allocate the space
read in that many integers – if EOF entered, quit at once
```