

# 198:211 Computer Architecture

Week 4  
Fall 2010

- Topics:
  - Memory Management
  - File I/O

1

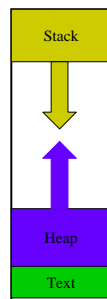
## Dynamic Allocation

- When a variable is declared, memory is allocated for that variable
  - `int i, j;`
- What if we don't a priori how many instances of a variable are needed?
- E.g., a **variable number of integers** – as many as the user wants to enter.
  - We can't allocate an array of integers, because we don't know the maximum number of that might be required.
  - Even if we do know the maximum number, it might be wasteful to allocate that much memory because most of the time only a few integers may be needed.
- **Solution:**  
Allocate storage for data dynamically, as needed.

2

## Memory partition

- Stack
  - Local variables on stack; grows and shrinks based on calls
    - dynamic
- Heap
  - Global variables, static constants
  - Dynamic (malloc)



3

## Enter Dynamic Memory

Memory allocated from the **stack** has a limited life-time.

- What do we do if we want memory that exists outside the "life-time" of functions

Two ways to "get memory"

- Declare a variable - gets placed on the **stack**
- Ask the run-time system for a "chunk" of memory

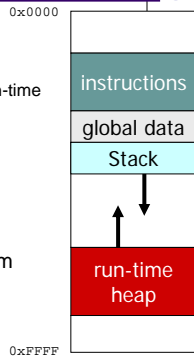
Requests for dynamic chunks of memory performed using a call to the underlying runtime system (a system call).

- Command: **malloc** and **free**

4

## Dynamic Memory or heap

- Dynamic request for memory are honored from heap.
- Memory Management taken care of by the run-time system. Complicated...
- While functions are being called and returning, activations records are being added and removed from the heap.
  - This does **NOT** effect the heap.
  - Once allocated it will stay there until freed
- Question to ask: why do they grow from different directions?



## malloc

- The Standard C Library provides a function for allocating memory at run-time: **malloc**.
- `void *malloc(int numBytes);`
- It returns a **generic pointer** (`void*`) to a contiguous region of memory of the requested size (in bytes).
- The bytes are allocated from a region in memory called the **heap**.
  - The run-time system keeps track of chunks of memory from the heap that have been allocated.

## Example 1

```
#include <stdio.h>
#include <malloc.h>

main(){
    int *base;
    int i,j;
    int cnt=0;
    int sum=0;
    printf("how many integers you have to store \n");
    scanf("%d",&cnt);
    /* how many integers? */
    base = (int *)malloc(cnt * sizeof(int));

    if(!base)
        printf("unable to allocate size \n");
    else {
        for(j=0;j<cnt;j++){
            *(base+j)=5;
        }
    }
}
```

## Using malloc

- To use malloc, we need to know how many bytes to allocate. The **sizeof** operator is used to calculate the size of a particular type.
- `planes = malloc(n * sizeof(Flight));`
- We also need to change the type of the return value to the proper kind of pointer – this is called “**casting**.”
- `planes = (Flight *) malloc(n* sizeof(Flight));`

## Example 2

```
int airbornePlanes;
Flight *planes;

printf("How many planes are in the air?");
scanf("%d", &airbornePlanes);

planes =
    (Flight*) malloc(sizeof(Flight) *
        airbornePlanes);
if (planes == NULL) {
    fprintf(stderr, "Error in allocating the
data array.\n");
    ...
}
planes[0].altitude = ...
```

If allocation fails,  
malloc returns NULL.

Note: Can use array notation  
or pointer notation.

9

## free

- Once the data is no longer needed, it should be released back into the heap for later use.
- This is done using the **free** function, passing it the same address that was returned by malloc.
  - `void free(void*);`
  - `free(base); /* in example 1 */`
- If allocated data is not freed, the program might run out of heap memory and will be unable to continue.

10

## Command Line Arguments

- You can use the command line to provide input to your programs.
- ```
int main( int argc, char * argv [] ){
...
}
```
- `argc` - the number of arguments (including program name)
- `argv` - point to array of char arrays (strings)

11

## Command Line (example)

- ```
int main(int argc, char * argv []){
    int i;
    printf("%d arguments\n", argc);
    for(i=0; i<argc; i++)
        printf("\t%d: %s\n", i, argv[i]);
    return 0;
}
```

12

## File I/O

- For our purposes, a **file** is a sequence of ASCII characters stored on some device.
  - Allows us to process large amounts of data without having to type it in each time or read it all on the screen as it scrolls by.
- **Each file is associated with a stream.**
  - May be input stream or output stream (or both!).
- The type of a stream is a "**file pointer**", declared as:

```
FILE *infile;
```
- The `FILE` type is defined in `<stdio.h>`.

13

## example

- `#include<stdio.h>`

```
void main(){
    FILE *fp;
    fp = fopen("PA1.ANS", "w");
    fprintf(fp, "%s %d\n", "OBAMA", 44);
    fclose(fp);
}
```

14

## fopen

- The `fopen` (**pronounced "eff-open"**) function associates a physical file with a stream.
- ```
FILE *fopen(char* name, char* mode);
```
- **First argument: name**
  - The name of the physical file, or how to locate it on the storage device. This may be dependent on the underlying operating system.
- **Second argument: mode**
  - How the file will be used:
    - "r" -- read from the file
    - "w" -- write, starting at the beginning of the file
    - "a" -- write, starting at the end of the file (append)

15

## fprintf and fscanf

- Once a file is opened, it can be read or written using `fscanf()` and `fprintf()`, respectively.
- These are just like `scanf()` and `printf()`, except an additional argument specifies a file pointer.
- `FILE *fpout; fpin;`
- `fpout = fopen("PA1.out", "w");`
- `fprintf(fpout, "The answer is %d\n", x);`
- `fpin = fopen("PA1.in", "r");`
- `fscanf(fpin, "%s %d/%d/%d %lf\n", name, &bMonth, &bDay, &bYear, &gpa);`

16

## A String is an Array of Characters



- Allocate space for a string just like any other array:
- `char outputString[16];`
- Space for string must contain room for terminating zero (i.e. `\0`)
- Special syntax for initializing a string:
- `char outputString[16] = "Result = ";`
- ...which is the same as:
- `outputString[0] = 'R';`  
`outputString[1] = 'e';`  
`outputString[2] = 's';`  
...

17

## I/O with Strings



- `printf` and `scanf` use `"%s"` format character for string
- `printf` -- print characters up to terminating zero
- `printf("%s", outputString);`
- `scanf` -- read characters until whitespace, store result in string, and terminate with zero
- `scanf("%s", inputString);`

```
char outputString[16] = "Result =";
printf("Length: %d\n", strlen(outputString));
printf("Length: %d\n", sizeof(outputString));
```

18

## Useful functions for Strings



- Useful string related functions in `string.h`
- `char * strcpy(ct, cs)` Copy string `cs` to `ct`
  - `char * ct, cs`
- `int strcmp(str1, str2)` Compare string `str1` to `str2`
  - `char * str1, str2`
  - A zero value indicates that both strings are equal. A value greater than zero indicates that the first character that does not match has a greater value in `str1` than in `str2`; And a value less than zero indicates the opposite.
- `size_t strlen(ls)` Returns length of `ls`
  - `char * ls`

19

## Standard C Library



- I/O commands are not included as part of the C language.
- Instead, they are part of the **Standard C Library**.
  - A collection of functions and macros that must be implemented by any ANSI standard implementation.
  - Automatically linked with every executable.
  - Implementation depends on processor, operating system, etc., but interface is standard.
- Since they are not part of the language, compiler must be told about function interfaces. Standard **header files** are provided, which contain declarations of functions, variables, etc.

20

## Basic I/O Functions

- The standard I/O functions are declared in the `<stdio.h>` header file.

| • Function             | Description                                 |
|------------------------|---------------------------------------------|
| • <code>putchar</code> | Displays an ASCII character to the screen.  |
| • <code>getchar</code> | Reads an ASCII character from the keyboard. |
| • <code>printf</code>  | Displays a formatted string.                |
| • <code>scanf</code>   | Reads a formatted string.                   |
| • <code>fopen</code>   | Open/create a file for I/O.                 |
| • <code>fprintf</code> | Writes a formatted string to a file.        |
| • <code>fscanf</code>  | Reads a formatted string from a file.       |

21

## Text Streams

- All character-based I/O in C is performed on **text streams**.
- A stream is a **sequence of ASCII characters**, such as:
  - the sequence of ASCII characters printed to the monitor by a single program
  - the sequence of ASCII characters entered by the user during a single program
  - the sequence of ASCII characters in a single file
- **Characters are processed in the order in which they were added to the stream.**
  - E.g., a program sees input characters in the same order as the user typed them.
- Standard input stream (keyboard) is called **stdin**.
- Standard output stream (monitor) is called **stdout**.
- Standard error output stream (monitor) is called **stderr**.

22

## Character I/O

- `putchar(c)` Adds one ASCII character (c) to stdout.
- `getchar()` Reads one ASCII character from stdin.

• These functions deal with "raw" ASCII characters; no type conversion is performed.

```
char c = 'h';  
...  
putchar(c);  
putchar('h');  
putchar(104);
```

Each of these calls prints 'h' to the screen.

23

## Buffered I/O

- In many systems, characters are **buffered** in memory during an I/O operation.
  - Conceptually, each I/O stream has its own buffer.
- **Keyboard input stream**
  - Characters are added to the buffer only when the newline character (i.e., the "Enter" key) is pressed.
  - This allows user to correct input before confirming with Enter.
- **Output stream**
  - Characters are not flushed to the output device until the newline character is added.

24

## Input Buffering

- ```
printf("Input character 1:\n");
inChar1 = getchar();

printf("Input character 2:\n");
inChar2 = getchar();
```
- After seeing the first prompt and typing a single character, nothing happens.
- Expect to see the second prompt, but character not added to stdin until Enter is pressed.
- When Enter is pressed, newline is added to stream and is consumed by second `getchar()`, so `inChar2` is set to `'\n'`.

25

## Output Buffering

- ```
putchar('a');
/* generate some delay */
for (i=0; i<DELAY; i++) sum += i;

putchar('b');
putchar('\n');
```
- User doesn't see any character output until after the delay.
- 'a' is added to the stream before the delay, but the stream is not flushed (displayed) until `'\n'` is added.

26

## Formatted I/O

- `printf` and `scanf` allow conversion between ASCII representations and internal data types.
- `Format string` contains text to be read/written, and `formatting characters` that describe how data is to be read/written.
- `%d` signed decimal integer
  - `%f` signed decimal floating-point number
  - `%x` unsigned hexadecimal number
  - `%c` ASCII character
  - `%s` ASCII string

27

## Special Character Literals

- Certain characters cannot be easily represented by a single keystroke, because they
  - correspond to whitespace (newline, tab, backspace, ...)
  - are used as delimiters for other literals (quote, double quote, ...)
- These are represented by the following sequences:
  - `\n` newline
    - `\t` tab
    - `\b` backspace
    - `\\` backslash
    - `\'` single quote
    - `\"` double quote
    - `\0nnn` ASCII code *nnn* (in octal)
    - `\xnnn` ASCII code *nnn* (in hex)

28

## printf

- Prints its first argument (format string) to `stdout` with all formatting characters replaced by the ASCII representation of the corresponding data argument.

```
int a = 100;
int b = 65;
char c = 'z';
char banner[10] = "Hola!";
double pi = 3.14159;

printf("The variable 'a' decimal: %d\n", a);
printf("The variable 'a' hex: %x\n", a);
printf("'a' plus 'b' as character: %c\n", a+b);
printf("A char %c.\t A string %s\n A float %f\n",
       c, banner, pi);
```

29

## scanf

- Reads ASCII characters from `stdin`, matching characters to its first argument (format string), converting character sequences according to any formatting characters, and storing the converted values to the addresses specified by its data pointer arguments.

```
char name[100];
int bMonth, bDay, bYear;
double gpa;

scanf("%s %d/%d/%d %lf",
      name, &bMonth, &bDay, &bYear, &gpa);
```

30

## scanf Conversion

- For each data conversion, `scanf` will skip whitespace characters and then read ASCII characters until it encounters the first character that should NOT be included in the converted value.

- `%d` Reads until first non-digit.
- `%x` Reads until first non-digit (in hex).
- `%s` Reads until first whitespace character.

- Literals in format string must match literals in the input stream.
- Data arguments must be pointers, because `scanf` stores the converted value to that memory address.

31

## scanf Return Value

- The `scanf` function returns an **integer**, which indicates the **number of successful conversions** performed.

- This lets the program check whether the input stream was in the proper format.

- **Example:**

```
int rv = scanf("%s %d/%d/%d %lf",
              &name, &bMonth, &bDay, &bYear,
              &gpa);
```

| Input Stream | Return Value |
|--------------|--------------|
|--------------|--------------|

|                    |   |
|--------------------|---|
| Mudd 02/16/69 3.02 | 5 |
|--------------------|---|

|                    |   |
|--------------------|---|
| Muss 02 16 69 3.02 | 2 |
|--------------------|---|

↑  
Doesn't match literal '/', so scanf quits after second conversion.

32

## Bad scanf Arguments



- Two problems with scanf data arguments

- **1. Not a pointer**

- ```
int n = 0;
scanf("%d", n);
```

- Will use the value of the argument as an address.

- **2. Missing data argument**

- ```
scanf("%d");
```

- Will get address from stack, where it expects to find first data argument.

- If you're lucky, program will crash because of trying to modify a restricted memory location (e.g., location 0). Otherwise, your program will just modify an arbitrary memory location, which can cause very unpredictable behavior.

33