Procedure Programming
## Characters

Instructor: Jeeho Ryoo

Announcements

- Lab 2 submissions
  - Push to github
  - Check in with me in lab session
  - No crazy corner cases FOR NOW
  - In case you wonder why I…
- Assignment 1 is coming
  - Max group of 3
  - Within the same set
  - No whining due to smaller group size
  - I will make a channel in Discord and this channel will be your golden bible
- Quiz 1 in lecture next week
  - Live coding
  - We will practice the platform in lab next week
  - Coverage beginning including today's lecture

Announcements

- Midterm schedule
  - Finally set
  - Oct 26
  - 9-11
  - NE01 239
  - What I usually do around midterm
- Email protocol exceptions for this week only
  - I want some feedback on Leetcode in lab

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | ... | 2^n-1 |
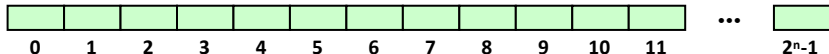
All modern processors have memories organized as sequence of *numbered bytes*
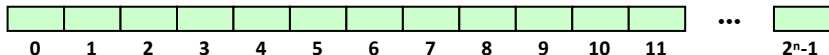
  Many (but not all) are linear sequences

Definitions:–

  Byte: an 8-bit memory cell capable of storing a value in range 0 … 255

  Address: number by which a memory cell is identified

												...	
0	1	2	3	4	5	6	7	8	9	10	11		2^n-1

Larger data types are sequences of bytes – e.g.,
- `short int` – 2 bytes
- `int` – 2 or 4 bytes
- `long` – 4 or 8 bytes
- `float` – 4 bytes
- `double` – 8 bytes

(Almost) always aligned to multiple of size in bytes

Address is "first" byte of sequence (i.e., byte zero)
- May be low-order or high-order byte
- *Big endian* or *Little endian*

Definition – *Pointer*

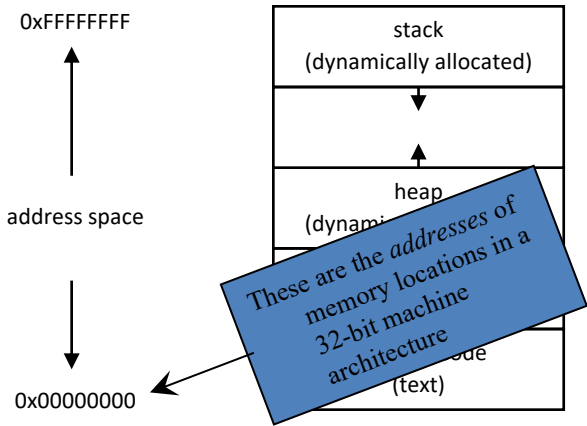A *value* indicating the *number* of (the first byte of) a data object

> Also called an *Address* or a *Location*

Used in machine language to identify which data to access

> E.g., *stack pointer* is address of most recent entry of *The Stack*

Usually 2, 4, or 8 bytes, depending upon machine architecture

Memory Addressing

0xFFFFFFFF

address space

0x00000000

stack
(dynamically allocated)

heap
(dynami...

...de
(text)

These are the *addresses* of memory locations in a 32-bit machine architecture

Pointers in *C*

Used *everywhere*
- For building useful, interesting, data structures
- For returning data from functions
- For managing arrays

`'&'` unary operator generates a *pointer* to `x`
- E.g., `scanf("%d", &x);`
- E.g., `p = &c;`
- Operand of `'&'` must be an *l-value* — *i.e.,* a legal object on left of assignment operator (`'='`)

Unary `'*'` operator *dereferences* a pointer
- i.e., gets value pointed to
- E.g. `*p` refers to value of `c` (above)
- E.g., `*p = x + y; *p = *q;`

Pointers in *C*

Used *everywhere*

    For building useful, interesting, data structures

    For returning data from functions

    For managing arrays

`'&'` unary operator generates a *pointer* to `x`

    E.g., `scanf("%d", &x);`

    E.g., `p = &c;`

    Operand of `'&'` must be an *l-value* — *i.e.,* a legal object on left of assignment operator (`'='`)

Unary `'*'` operator *dereferences* a pointer

    i.e., gets value pointed to

    E.g. `*p` refers to value of `c` (above)

    E.g., `*p = x + y; *p = *q;`

Not the same as binary `'&'` operator (bitwise AND)

Declaring Pointers in *C*

`int *p;` — a pointer to an `int`

`double *q;` — a pointer to a `double`

`char **r;` — a pointer to a pointer to a `char`

type `*s;` — a pointer to an object of type *type*

Pointer declarations:–read from *right* to *left*

```
const int *p;
```
- `p` is a pointer to an integer constant
- i.e., pointer can change, thing it points to cannot

```
int * const q;
```
- `q` is a constant pointer to an integer variable
- i.e., pointer cannot change, thing it points to can!

```
const int * const r;
```
- `r` is a constant pointer to an integer constant

Pointer Arithmetic

```
int *p, *q;
q = p + 1;
```
Construct a pointer to the next *integer* after ***p** and assign it to **q**

```
double *p, *r;
int n;
r = p + n;
```
Construct a pointer to a *double* that is **n** *doubles* beyond ***p**, and assign it to **r**

n may be negative

```
long int *p, *q;
p++; q--;
```
Increment **p** to point to the next **long int**; decrement **q** to point to the previous **long int**

```
float *p, *q;
int n;
n = p − q;
```
n is the number of floats between ***p** and ***q**; i.e., what would be added to **q** to get **p**

Pointer Arithmetic (continued)

```
long int *p, *q;
p++; q--;
```
Increment **p** to point to the next **long int**; decrement **q** to
point to the previous **long int**

C never checks that the resulting pointer is valid

```
float *p, *q;
int n;
n = p − q;
```
n is the number of floats between ***p** and ***q**; i.e., what would
be added to **q** to get **p**

Why introduce pointers in the middle of a lesson ?

Arrays and pointers are *closely related* in *C*

In fact, they are essentially the same thing!

Esp. when used as parameters of functions

```
int A[10];
int *p;
```

- *Type* of `A` is `int *`
- `p = A;` and `A = p;` are legal assignments
- `*p` refers to `A[0]`
  `*(p + n)` refers to `A[n]`
- `p = &A[5];` is the same as `p = A + 5;`

`double A[10];` *vs.* **`double *A;`**

Only difference:–

> **`double A[10]`** sets aside *ten* units of memory, each large enough to hold a **`double`**, and **`A`** is initialized to point to the zeroth unit.
>
> **`double *A`** sets aside *one* pointer-sized unit of memory, not initialized
>
> - You are expected to come up with the memory elsewhere!
>
> Note:– all pointer variables are the same size in any given machine architecture
>
> - Regardless of what types they point to

C does *not* assign arrays to each other
e.g,

```
double A[10];
double B[10];

A = B;
```

- assigns the pointer value **B** to the pointer value **A**
- Original contents of array **A** are untouched (and possibly unreachable!)

Arrays as Function Parameters

```
void init(float A[], int arraySize);
void init(float *A, int arraySize);
```

- Are identical function prototypes!
- Pointer is passed by value
- i.e. caller copies the *value* of a pointer to `float` into the parameter `A`
- Called function can reference *through* that pointer to reach thing pointed to

Streams

All input and output is done with streams

Input steam
    A sequence of bytes flowing into a program
Output stream
    A sequence of bytes flowing out of a program

Standard Input/Output Streams

stdin stands for Standard Input
      Keyboard, Scanner are standard input devices
Standard input is data going <u>into</u> a program

stdout stands for Standard output
      Screen, printer are standard output device
Standard output is data going <u>out</u> from a computer

printf()

Output function
printf() prints the value given to the console
Built into the input output header file stdio.h

```
#include <stdio.h>
int main()
{
    printf("Good Morning");
    return 0;
}
```

scanf()

Input function
Also built into the header file stdio.h
scanf() takes input from the user
Uses same format identifier %d as printf()

> *int x;*
> *scanf("%d", &x);*

Note the **&** symbol, scanf requires a memory address
(called address-of operator)

%d will scan or print an integer aka int (*example: 5*)
%f will scan or print a floating point aka float (*example: 5.1*)
%c will scan or print a character aka char (*example: m*)
%s will scan or print a char array aka string (*example: george*)

scanf() and printf() Example

```c
#include <stdio.h>

int main()
{
    int x;
    printf("Enter a value...");
    scanf("%d", &x);
    printf( "\nYou entered: %d", x);
    return 0;
}
```

printf() and scanf() Strings Example

```
char  s[20];
printf("How are you doing?...");
scanf("%s", s);
printf( "\nYou entered: %s", s);
return 0;
```

A string in scanf doesn't require the **&** because it's an array (discussed earlier)
Note even with strings whitespace is a delimeter by default

getchar()

Input function
Also built into the header file stdio.h
It reads <u>one</u> character
Example

> *char c;*
> *c = getchar();*

Input function

Reads up to and <u>including</u> a newline character (enter)

Takes in three arguments

    A **string**

    Number of character to copy (**int**)

    File type (in our case it's the standard input, **stdin**)

        fgets(str, 20, stdin);

fgets() Example

Write a program that prints a sentence to the console using fgets()
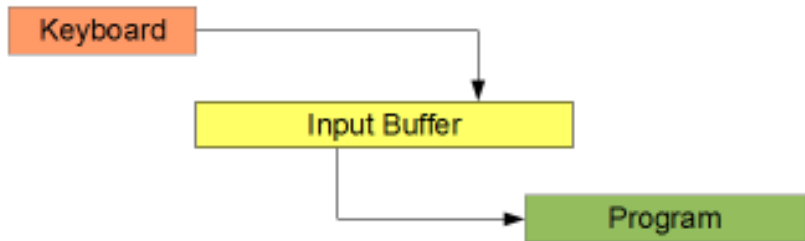
```c
#include <stdio.h>
int main()
{
    char str[100];
    fgets(str, 100, stdin);
    printf( "\nYou entered: %s", str);
    return 0;
}
```

Holds the input data temporarily between the device and program
Once enter (newline) is pressed it signals the program to start executing data from the buffer
The program only retrieves the data that it needs , and leaves rest in buffer

fputs()

Output function
Simple string output
Takes two arguments
    A **string**
    File type (in our case it's the standard output, ***stdout***)

    fputs(str, stdout); is equivalent to *printf("%s", str)*

fputs() Example

Write a program using fputs()

```c
#include <stdio.h>
int main()
{
    char str[100];
    fputs("Input something\n", stdout);
    fgets(str, 100, stdin);
    fputs(str, stdout);
    return 0;
}
```

Characters in *C*

`char` is a one-byte data type capable of holding a character

       Treated as an arithmetic integer type

       (Usually) **`unsigned`**

May be used in arithmetic expressions

       add, subtract, multiply, divide, etc.

Character constants

       `'a'`, **`'b'`**, **`'c'`**, …**`'z'`**, **`'0'`**, **`'1'`**, … **`'9'`**, **`'+'`**, **`'-'`**, **`'='`**, **`'!'`**, **`'~'`**, etc, **`'\n'`**, **`'\t'`**, **`'\0'`**, etc.

       A-Z, a-z, 0-9 are *in order*, so that arithmetic can be done

Definition:– A *string* is a character array ending in `'\0'` — i.e.,

```
char s[256];
char t[] = "This is an initialized
string!";
char *u = "This is another string!";
```

String constants are in double quotes
> May contain any characters
> Including `\"` and `\'`

String constants *may not* span lines
> However, they may be concatenated — e.g.,
> `"Hello, " "World!\n"` is the same as `"Hello, World!\n"`

Let

```
char *u = "This
is another
string!";
```

Then

```
u[0] == 'T'
u[1] == 'h'
u[2] == 'i'
…
u[21] == 'g'
u[22] == '!'
u[23] == '\0'
```

Support for Strings in *C*

Most string manipulation is done through functions in `<string.h>`

String functions *depend* upon final `'\0'`
So you don't have to count the characters!

Examples:–

`int strlen(char *s)` – returns length of string
Excluding final `'\0'`

`char* strcpy(char *s, char *ct)` – Copies string `ct` to string `s`, return `s`
`s` must be big enough to hold contents of `ct`
`ct` may be smaller than `s`

Examples (continued):–
`int strcmp(char *s, char *t)`
- lexically compares **`s`** and **`t`**, returns **`<0`** if **`s < t`**, **`>0`** if **`s > t`**, zero if **`s`** and **`t`** are identical

`char* strcat(char *s, char *ct)`
- Concatenates string **`ct`** to onto end of string **`s`**, returns **`s`**
- **`s`** must be big enough to hold contents of both strings!

Other string functions
`strchr(), strrchr(), strspn(), strcspn()`
`strpbrk(), strstr(), strtok(), …`

String Conversion Functions in *C*

See `<stdlib.h>`
```
double atof(const char *s)
int atoi(const char *s)
long atol(const char *s)

double strtod(const char *s, char **endp)
long strtol(const char *s, char **endp, int base)
unsigned long strtoul(const char *s, char **endp, int base)
```

Dilemma

Question:–
>   If strings are arrays of characters, …
>   and if arrays cannot be returned from functions, …
>   how can we manipulate variable length strings and
>   pass them around our programs?

Answer:–
>   Use storage allocated in *The Heap!*