Procedure Programming
Characters

Instructor: Jeeho Ryoo

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

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

```c
#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;
}
```

Input buffer

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
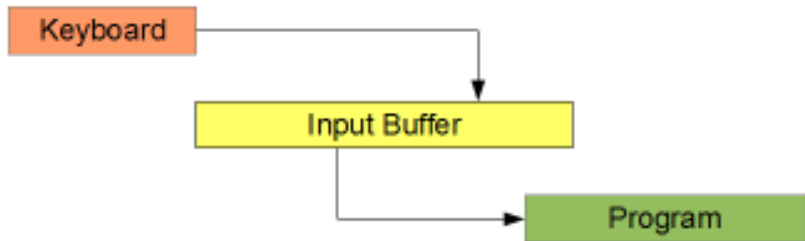
fputs()

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'
```

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)
```

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!*

Definition — *The Heap*

A region of memory provided by most operating systems for allocating storage *not* in *Last in, First out* discipline
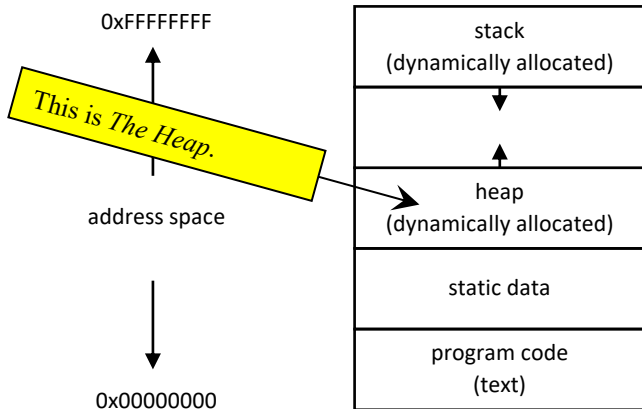
       i.e., not a stack

Must be explicitly allocated *and* released
May be accessed *only* with pointers

       Remember, an array is equivalent to a pointer

Many hazards to the *C* programmer

Static Data Allocation

See `<stdlib.h>`
`void *malloc(size_t size);`
`void free(void *ptr);`
`void *calloc(size_t nmemb, size_t size);`
`void *realloc(void *ptr, size_t size);`

`malloc()` — allocates `size` bytes of memory from the heap and returns a pointer to it.
        NULL pointer if allocation fails for any reason

`free()` — returns the chunk of memory pointed to by `ptr`
        Must have been allocated by `malloc` or `calloc`

See `<stdlib.h>`

```
void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```

`malloc()` — allocates `size` bytes of memory from the heap and returns a pointer to it.
- NULL pointer if allocation fails for any reason

`free()` — returns the chunk of memory pointed to by `ptr`
- *Must* have been allocated by `malloc` or `calloc`

`free()` knows *size* of chunk allocated by `malloc()` or `calloc()`

`calloc()` is just a variant of `malloc()`

`malloc()` is analogous to `new` in *C++* and Java

- `new` in *C++* actually calls `malloc()`

`free()` is analogous to `delete` in *C++*

- `delete` in *C++* actually calls `free()`
- Java does not have `delete` — uses *garbage collection* to recover memory no longer in use

Typical usage of `malloc()` and `free()`

```c
char *getTextFromSomewhere(…);

int main(){
    char * txt;
    …;
    txt = getTextFromSomewhere(…);
    …;
    printf("The text is %s.", txt);
    free(txt);
}
```

Typical usage of **malloc()** and **free()**

```
char * getTextFromSomewhere(
    char *t;
    ...
    t = malloc(stringLength);
    ...
    return t;
}

int main(){
    char * txt;
    …;
    txt = getTextFromSomewhere(…);
    …;
    printf("The text is %s.", txt);
    free(txt);
}
```

Typical usage of `malloc()` and `free()`

```c
char * getTextFromSomewhere(…){
    char *t;
    ...
    t = malloc(stringLength);
    ...
    return t;
}

int main(){
    char * txt;
    …;
    txt = getTextFromSomewhere(…);
    …;
    printf("The text is %s.", txt);
    free(txt);
}
```

Pointer to text is assigned to *txt* in calling function

```c
char * getTextFromSomewhere(…){
    char *t;
    ...
    t = malloc(stringLength);
    ...
    return t;
}

int main(){
    char * txt;
    …;
    txt = getTextFr…
    …;
    printf("The tex…          …xt);
    free(txt);
}
```

main() must remember to free the storage pointed to by **txt**

Definition – *Memory Leak*

The steady loss of available memory due to forgetting to `free()` everything that was `malloc`'ed.

- Bug-a-boo of most large *C*

If you "forget" the value of a pointer to a piece of `malloc`'ed memory, there is no way to find it again!

- Killing the program frees *all* memory!

String Manipulation in *C*

Almost all *C* programs that manipulate text do so with **malloc**'ed and **free**'d memory

No limit on size of string in *C*

Need to be aware of sizes of character arrays!

Need to remember to free storage when it is no longer needed
- *Before* forgetting pointer to that storage!

`printf(const char *format, ...)`
- Format string may contain **`%s`** – inserts a string argument
  (i.e., **`char *`**) up to trailing **`'\0'`**

`scanf(const char *format, ...)`
- Format string may contain **`%s`** – scans a string into argument
  (i.e., **`char *`**) up to next "white space"
- Adds **`'\0'`**

Related functions
- **`fprintf()`**, **`fscanf()`** – to/from a file
- **`sprintf()`**, **`sscanf()`** – to/from a string

Example Hazard

```
char word[20];
…;
scanf("%s", word);
```

scanf will continue to scan characters from input until a space, tab, new-line, or **EOF** is detected

- An unbounded amount of input
- May overflow allocated character array
- Probable corruption of data!
- **scanf** adds trailing **'\0'**

Solution:

```
scanf("%19s", word);
```