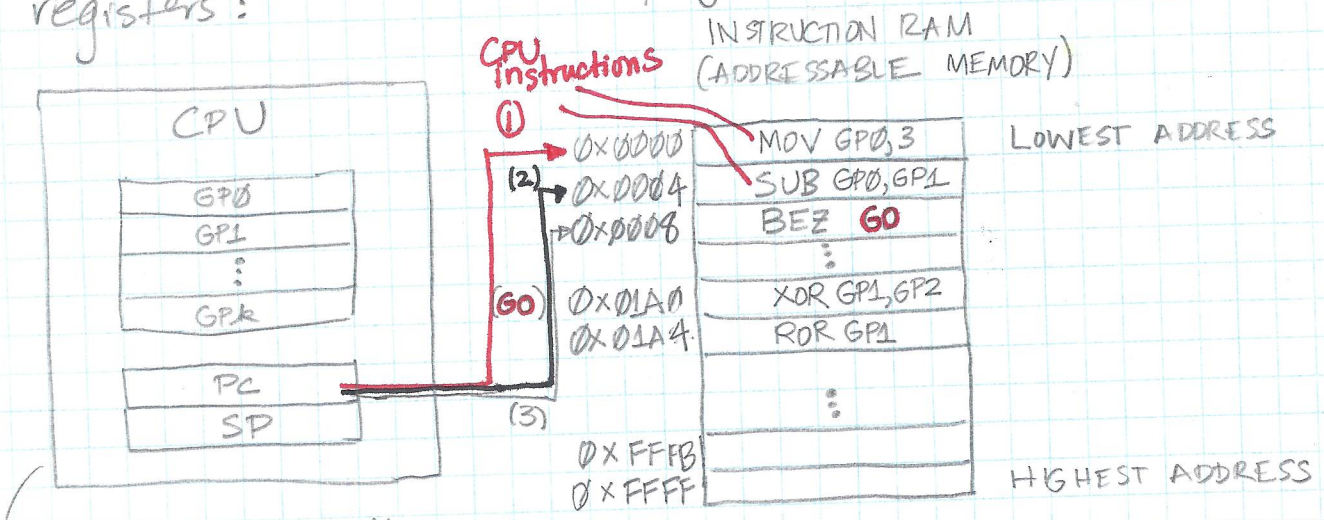
INTRODUCTION

WHAT IS A COMPUTER PROGRAM?

- in a sense, a computer program is a "script" performed by a microprocessor; modern digital computers, based on finite state machines, were conceived in order to be general purpose devices, capable of acting on a set of instructions given to them by human users.

- an understanding of computer programs comes from an understanding of how microprocessors work. a microprocessor contains various registers, some for general-purpose use, and others that have a specific function, such as the stack pointer (SP) and program counter (PC) registers:



→ microprocessor with registers shown; recall that a CPU has an instruction set from which programs can be built

- The CPU performs the first instruction at the address pointed to by the PC register at (1)
- When the instruction at 0x0000 completes, the PC points to the next instruction at (2)
- Again once the second instruction is completed the PC moves to the next instruction (3)
- Note that BRANCHING can occur, as with the "branch if equal to zero" instruction at address 0x0008.

- if the result of "SUB GP0, GP1" was zero, the PC is loaded with the address 0x01A0 (corresponding to the "GO" label in the program)

- from this example we can think of a computer program as a series of instructions with a (default) sequential flow of logic that may also involve branching (i.e., movement of the PC that is not sequential).

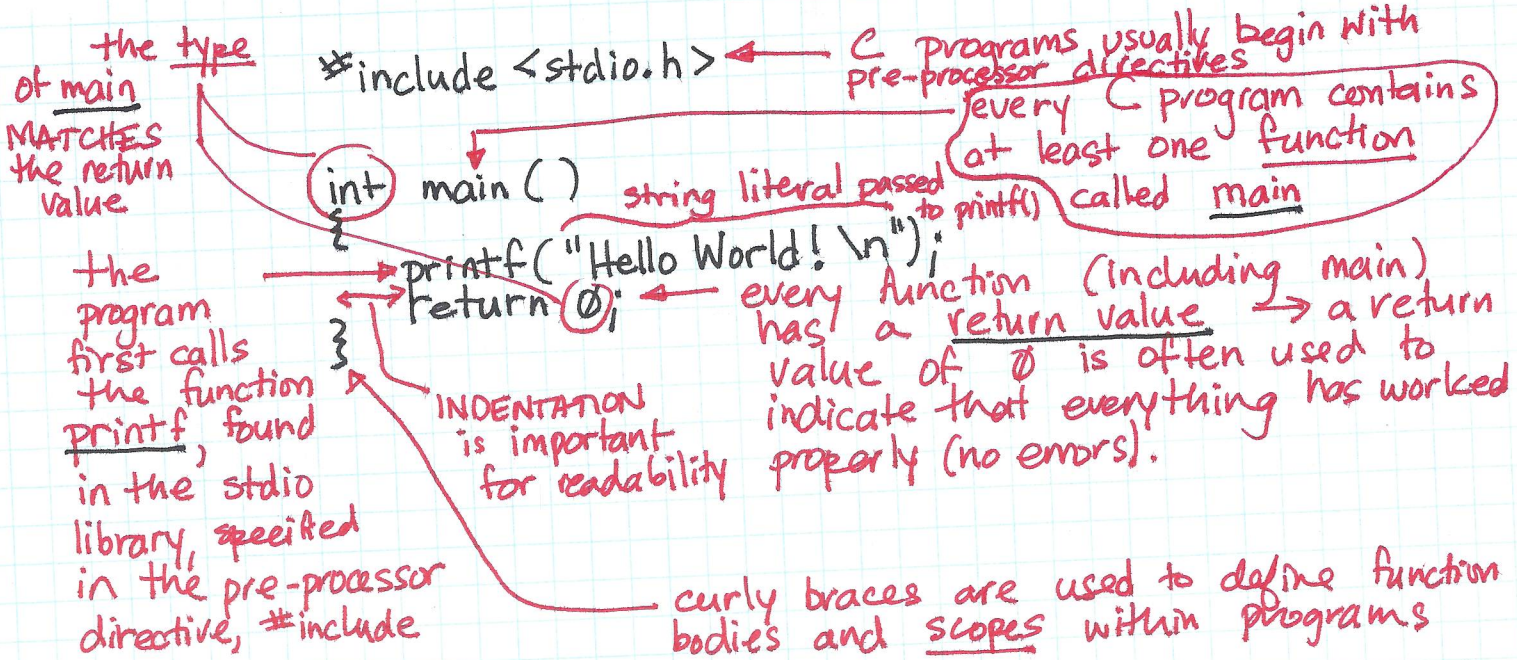
- the instruction set allows us to program a computer using assembly language, however for large programs which need portability assembly language is challenging to work with, requiring that the program be re-written if it must run on a different microprocessor architecture.

- It is because of the tedious nature of assembly language that developers have created high-level languages, which allow us to program computers without having to deal with the specifics of the underlying CPU architecture.

- C was developed in the late 1960s and early 1970s as a language for developing operating systems which could run on multiple computer platforms; the UNIX kernel, for example, was written in C and subsequently became popular because of the widespread adoption made possible by the portability of the UNIX software. Today, billions of devices run variants of UNIX, in the form of Android OS, Linux or Mac OS.

AN INTRODUCTION TO C

- Consider the following "Hello World!" C program:



- enter this program, build it and execute it in your development environment!

SHORTEST POSSIBLE C PROGRAM: (with no compile-time errors or warnings) `void main(){}`

variables

- programs need variables to store, process and evaluate data
- in C, variables have definite types; the basic types in C include:

`int`	(integer type)
`char`	(character type) ← for single characters
`float`	(floating-point type) ← for numbers with fractional components
`void`	(a generic type, e.g. when no value is needed)

- note: it is possible to store string literals (like "Hello World!\n" above) in variables, in a pointer type, `char*`.

- here is a slightly expanded Hello-World program: (please see the latest version on the course Git repository")

```
#include <stdio.h>

/* global variables */

char    message[50] = "
char    user_name[30]
```

global variables: can be used any where in the program

```
int main()
{
    /* local variables */

    int mood;
    char pets;
```

local variables: defined only within the scope in which they are defined — in C, scopes are usually explicitly defined with { ... }

```
    /* request user's name */
    printf("Please enter your name: ");
    scanf("%s", user_name);
```

another function in stdio, which retrieves input from the "standard input", in this case, the computer keyboard

format specifier (string literal)

```
    /* display message */
    message = "Hello! It is nice to meet you, ";
    printf("%s", message);
    printf("%s!\n", user_name);
```

string literal

~~string literals are terminated with a null character~~

```
    /* ask questions */
    printf("\nI hope you do not mind answering some questions! \n");

    printf("On a scale of 1 to 10, how would you
            rate your mood today? (please enter a
            number): ");

    scanf("%d", &mood);

    printf("\n Do you have any pets? (please enter
            Y or N)? ");

    scanf("%c", &pets);

    printf("\n\n Thank you for answering my questions, %s,
            goodbye!", user_name);

    /* exit successfully */
    return 0;
}
```

- enter, build and execute this program

- this program is more interesting than the Hello-World program, but it is a little disappointing. The program collects data, but does little with it
- let's expand the program so that its response changes, depending on the user's responses.
- for example, if the user gives a low mood value, the response could be encouraging; if the user's mood is good, the program could acknowledge that.
- in C, we can use a special control structure to make decisions. It is called an if statement. Consider the following code fragment:

```
char mood_response[30];
mood_response = "that's Wonderful," ; /* default response */
if (mood < 6)
{
    mood_response = "I am sorry, I hope you feel better soon, ";
}
printf(mood_response);
printf(" %s !\n", user_name);
```

- insert this code fragment into your program, and test the results.

- the if statement also has an "else" option to expand its capabilities:

```
if (...)
{
    :
}
else
{
    :
}
```

thus we could rewrite the above fragment as

```
char *mood_response
if (mood >= 7)
{
    mood_response = "That's Wonderful \0";
}
else
{
    mood_response = "I am sorry, I hope
        you feel better soon \0";
}
printf ...
```

- try out this version of the program. Do you notice any differences?

- the if-else statement can be laddered to allow even greater decision-making ability:

```
if (...)
{
    :
}
else if (...)
{
    :
}
else if (...)
{
    :
}
else
{
    :
}
```

in C, true literally equals 1 and false is equal to 0

Please note! TRUE OR FALSE (1) (0)

Relational

>	greater than
>=	g.t. or equal
<	less than
<=	l.t. or equal
==	equal
!=	not equal

Logical

&&	AND
||	OR
!	NOT

- For example:

```
char mood_response[30];
if (mood < 3)
{
    mood_response = "Oh that's dreadful. Sorry, ";
}
else if((mood == 3) || (mood < 7))
{
    mood_response = "I am sorry, feel better soon, ";
}
else /* mood must be > 7 */
{
    mood_response = "That's Wonderful, ";
}
```

logical OR

technically not needed!

- try out this code block, or a variation!

- HOMEWORK: produce an expanded version of the 'hello_!' program that takes into account whether the user has pets, as follows

- if the user has a mood ≤ 3, the program responds with:

  "Maybe cuddling with your pet will help!" if the user has a pet, and

  "Studies show that pets can improve our overall sense of well-being. Why not get one now?!" if the user does not have a pet.

- if the user has a mood ≥ 4, but < 7, the program responds with:

  "I am sure spending time with your pet will cheer you up!" if the user has a pet

  and

  "All you need is to spend time with a new friend. A visit to the animal shelter is the ticket!" if the user does not have a pet.

- if the user has a mood ≥ 7 the response should be:

  "Great news! Celebrate with your pet!" if the user has a pet and

  "That's great! Imagine if could just share your happiness with a new friend. Please visit your local animal shelter!" if the user does not have a pet.

C EXPRESSIONS

- Expressions in C are formed from operators and variables or constants. We begin with a discussion of variables.
- there are five basic (atomic) variable types in C.

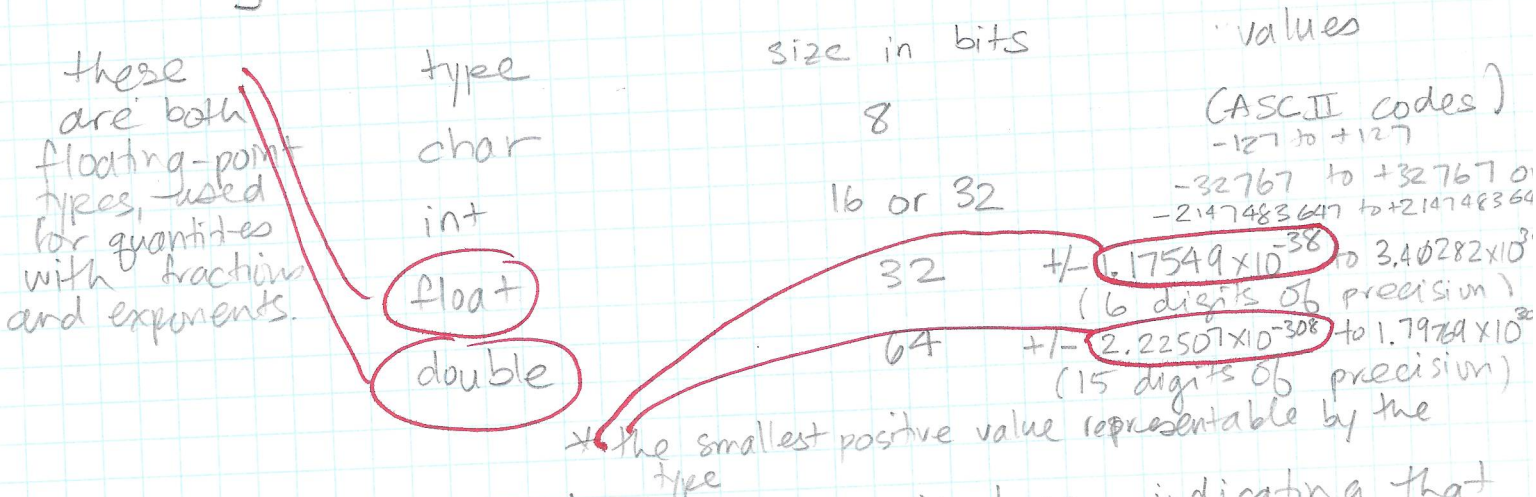
```
char
int
float
double
void   <-- also known as "valueless"
```

- there are nine data types in C, but they are based on these types

- recall that types are used to define variables as in:

```
char   ch;
int    num;
float  grade;
double distance;
void   blank;
```

and that a certain number of bytes is taken up by a variable of a given type. The number of bytes depends on compiler/platform you are using. Typically, in a 32-bit environment we have:



- the type void is a generic type indicating that no value is returned or that special objects — called pointers — are created; it has no fixed associated size. To be discussed later...

modifying basic types

- the basic types (except for void) can be modified with special keywords to fit various situations:

signed
unsigned
long
short

→ these can be applied to int, but double only accommodates long

Note that "signed int" is redundant because an integer is already signed

type	size in bits	range
char	8	-127 to 127
unsigned char	8	0 to 256
signed char	8	-127 to 127
int	16 or 32	-32767 to 32767
unsigned int	16 or 32	0 to 65535
short int	16	-32767 to 32767
long int	32	-2 147 483 647 to 2 147 483 647
double	64	
long double	80	

naming variables / functions / labels / etc.

- various user-defined objects are named using identifiers, several characters in length.

Note that:

- the first character must be a letter or underscore
- subsequent letters must be letters, digits, or underscores (other punctuation marks are not allowed)
- the identifier cannot be a C keyword (that is, a reserved word corresponding to a C-language component).

Correct	Incorrect
Sum32	32sum
phidget_7	ph!dget?
_me_myself_I	_me...myself...I

Arrays

- As we have seen, it is possible to store string literals in arrays of characters. Recall our last program contained the definition

```
char user_name[30];
```

meaning that a user name of up to 30 characters may be contained in the variable. This variable is an example of an array. For example the name "Azealia" would be stored as

byte 0	user_name[0]	A
byte 1	user_name[1]	z
byte 2	user_name[2]	e
⋮	user_name[3]	a
	user_name[4]	l
	user_name[5]	i
	user_name[6]	a
	user_name[7]	\0

← note that the first element in an array has index 0

recall that a char variable occupies 8 bits (corresponding to ASCII code)

← most string literals are terminated automatically with a null "\0" character

- Note that we can create arrays of any of the atomic data types or their modified versions, e.g.,

```
short int smallish[11]
long double bignums[23]
unsigned char uchars[512]
```

- if the array name appears without [·], a pointer to the array is implied. This pointer points to the first element of the array.

- a pointer is a variable that contains a memory address, and we will have much more to say about pointers in the future

Operators

C has a rich set of operators. An operator can be used to modify the values of variables. There are four classes of operator:

- arithmetic
- relational
- logical
- bitwise

An expression in C is composed of one or more operands (variables, constants) that are combined by operators. For example:

```
int x, y, z;
x = 10;
y = 20;
z = x * y + 23;
```

← assignment operator
← assignment operator
operands →
← operators: assignment, *, +

this code fragment consists of three expressions. The operands 10, 20 and 23 are integer constants. The operands x, y and z are variables (integer type). The first expression:

```
x = 10;
```

is an assignment. The variable x is set equal to 10.

Operators can be binary (having two operands) or unary (having a single operand)

Arithmetic		
Operator	Function	Use
+	unary sign ch.	+expr
-	unary sign ch.	-expr
*	multiplication	expr * expr
/	division	expr / expr
%	remainder	expr % expr
+	addition	expr + expr
-	subtraction	expr - expr

```
x = -17;
y = +3;
z = -x % 3 * z
```

- The following code is valid in C:

```
int a, b, c;
a = b = c = 10;   /* expression with multiple assignments
```

- the assignment operator is "right to left" associative, meaning that the expression can be re-written as

```
a = (b = (c = 10));
```

① the variable c is first set to 10; the assignment operator then returns its left operand c

② next, b is set equal to c, and b is returned

③ finally, a is set equal to b, a is returned but there is nothing to receive this returned value.

More operators:

Arithmetic		
Operator	function	Use
++	unary increment	expr++, ++expr
--	unary decrement	expr--, --expr

prefix example:

```
x = 10;
y = ++x;  /* y is set to 11 */
```

postfix example:

```
x = 10
y = x++;  /* y is set to 10 */
```

- in either case, x is set to 11.

Relational	
Operator	Meaning
>	greater than
>=	greater than or equal
<	less than
<=	less than or equal
==	equal
!=	not equal

NOTE: In C, true is interpreted as any value other than zero, and false is equal to zero. Expressions that use relational or logical operators return 0 for false and 1 for true.

even more operators

Arithmetic

Operator	Function	Use	Equivalence
+=	binary sum, assignment	expr1 += expr2	expr1 = expr1 + expr2
-=	binary difference, assignment	expr1 -= expr2	expr1 = expr1 - expr2

Logical

Operator	Meaning
&&	AND
||	OR
!	NOT

- what does the expression
  ((1 > 0) && (3 > 2)) || (6 != 5)
  return?
- how about:

  !1 <= 0 && 0 || 0

NOTE THE PRECEDENCE OF RELATIONAL AND LOGICAL OPERATORS:

HIGHEST (evaluated first)
↓
LOWEST (evaluated last)

```
!
> >= < <=
== !=
&&
||
```

evaluate the following

```
  1010 1111
^ 0101 1101
-----------
```

```
8 << 1
1 << 3
16 >> 1
-5 (in binary)
```

BITWISE - USEFUL IN EMBEDDED DRIVERS!

(often used with operands of unsigned int type)

Operator	Meaning
&	and
|	or
^	xor
~	one's complement (not)
>>	shift right
<<	shift left

C STATEMENTS

- A statement in C is a part of the program that is executable — it corresponds to a set of assembly language instructions

Statements can fall into the following categories:

- selection
- iteration
- jump
- label
- expression
- block

For example, selection can be performed by a conditional statement, such as if-else, which we have seen.

An expression (as discussed above) can be a statement in standalone form when followed by a semi-colon

The "return 0;" in main() is an example of a jump statement

- One category we have not yet discussed is iteration, which can be of three types:

  - while
  - for
  - do-while

- What do you suppose the following code does?

```
char d;
int x;
x = 7;
d = '*';
while (x != 0)
{
    printf("%c", d);
    --x;
}
```

character constants are enclosed by single quotes

For understanding while loops, it is sometimes helpful to identify an invariant description for it; in this case, the invariant is "x is the number of characters to be printed"

Q: how many times does this loop execute?

- in debugging code — that is, finding the cause of functional errors in our programs — it is useful to be able to tabulate the progress of algorithms

iteration (pass before })	value of x (before --x;)	displayed result
1	7	*
2	6	* *
3	5	* * *
4	4	* * * *
5	3	* * * * *
6	2	* * * * * *
7	1	* * * * * * *
8 → exit occurs because x=0		

- another variation is the do-while loop: can you chart the progress of the following code?

```
char d;
int x;
x = 7;
d = '*';
do
{
    printf("%c", d);
    --x;
} while (x != 0)
```

How many asterisks are printed now?

- Here is the for-loop version:

```
char d;
d = '*';
for (int x = 7; x != 0; --x)
{
    printf("%c", d);
}
```

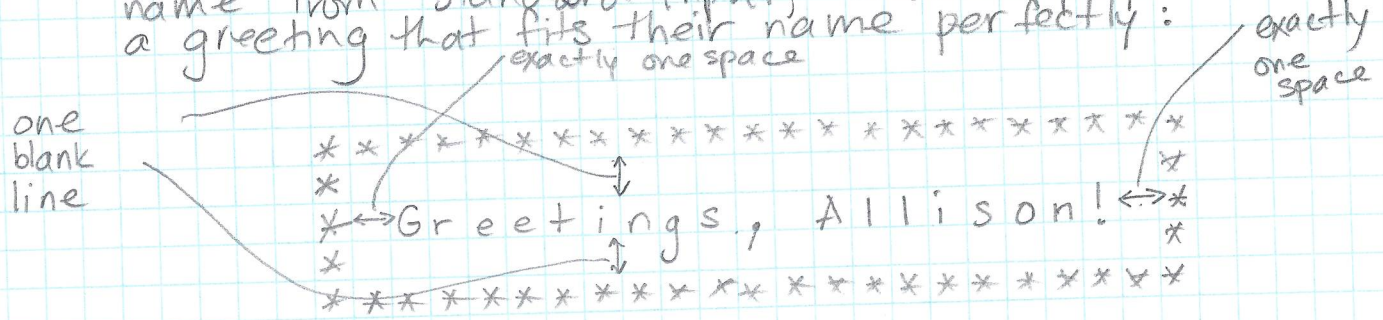
→ after each iteration x is altered based on this expression and also this condition

HOMEWORK

① Look up the strlen() function found in the string.h library. What is its purpose?

Look up the related functions, strcpy() and strcat(). How are they used?

② Using the tools you have learned so far, write a program that takes a user's name from standard input, and displays a greeting that fits their name perfectly:



③ Write a C-program that calculates one's gain on a principal amount of $1000, given an annual compounded interest rate of 6%, over 30 years. Use an approach based on iteration.

④ Can you modify the above program to accept the principal amount, interest rate and investment period, from the user?

C STATEMENTS CONT'D

- recall that so far we have covered
  - Selection Statements:
    - if, if/else, if/else if/else
  - Iteration Statements:
    - for, while, do/while
  - Jump Statements:
    - return

- switch
  - another selection statement with multiple branches is the switch statement.
  - it has the general form:

```
switch (expression)
{
   case constant1:
      :
      break;
   case constant2:
      :
      break;
      :
   default:
      :
}
```

- (expression) must evaluate to an int or char type
- expression is compared against these constant values
- when a match occurs, the corresponding statements of the matching case are executed, followed by a break statement, which exits the switch statement
- if no match is found, the default case is selected and since it comes last, no break is needed.

EXAMPLE:

```
printf("press a key: ");
ch = getchar();
switch (ch)
{
    case 'a':
      printf("a is for apple");
      break;
    case 'b':
      printf("b is for bee");
      break;
    case 'c':
      printf("c is for melon");
      printf("get it?");
      break;
    default:
      printf("that is all, folks!");
}
```

Nesting

- note that statements we have covered so far can be nested, that is, statements of one kind can be placed within statements of the same kind.
- for example, a nested for loop would be:

```
int sum;
for (int i=0; i!=10; ++i)
{
    sum = 0;
    for (int j=0; j!=20; ++j)
    {
        sum += j*i + 3;
    }
    ...
}
```

- a nested if statement could look like:

```
if (a==3 && b==7)
{
    if (c==22 || d>12.3)
    {
        if (e<((7*f+3)%2)
        {
            :
        }
    }
    else
    {
    }
}
else if (a==4 && b==8)
{
    :
}
else
{
    :
}
```

Infinite Loops

- can be useful in certain situations, particularly in embedded systems design
- an infinite loop can be realized using any iteration statement:
  - an infinite for loop:

```
for (;;)
```

example:

```
for (;;)
    printf ("This statement runs forever!\n");
```

  - an infinite while loop:

```
while (1)
{
    :
}
```

example:

```
while (1)
    printf ("This statement runs forever!\n");
```

- we can use a break statement to exit an (otherwise) infinite loop:

```
ch = '\0';
for (;;)
{
    ch = getchar(); /* get a character from the keyboard */
    if (ch == 'y' || ch == 'n')
        break; /* exit the loop */
}

switch (ch)
{
    case 'y':
        printf("you typed YES!");
        break;
    case 'n':
        printf("you answered NO!");
        break;
}
```

for loops with no bodies

- often used in embedded programming to create delays, for example

```
#define   DELAY   1048576
          :

printf("delay loop starting!\n");
for (int t=0; t != DELAY; ++t);
printf("delay loop ended!\n");
```

note that declaring a variable within a loop is not permitted in C89, but allowed in C99 (and C++).

- in-class lab / homework:
  - compose a C program that demonstrates the examples so far as a series, but don't get stuck in an infinite loop!

exit()

- found in stdlib.h, you can use the function exit() to break out of your program
- this is often used to indicate an error, and you can use the argument of exit() to indicate the error code.

for example:

```
int exit_code = 0;
if ( ! graphics_card() )
{
    exit_code = 1;
    exit (exit_code);
}
else if ( ! joy-stick_found() )
{
    exit_code = 2;
    exit (exit_code);
}
```

- the stdlib library also contains the macros EXIT_SUCCESS and EXIT_FAILURE which can be used as arguments (return codes) for exit(), as well

FUNCTIONS

- the general form of a C function is:

```
return-type function-name (parameter list)
{
    :
    return expression;
}
```

expression must be of type return-type;
if return-type is void, then there is no expression

- examples of functions include:

```
int decrement (int arg)
{
    arg--;
    return arg;
}
```

arg is a function local variable; if a global variable happens to have the same name, the local variable is assumed inside the function

```
int is_found (char *s, char c)
{
    /* return 1 if c found in array s, 0 otherwise */
    while (*s)
        if (*s == c)
            return 1; /* function returns early */
        else
            s++;
    return 0;
}
```

an array without the brackets, is a pointer.

e.g.
```
int s[5];
    :
int *p;
p=s;
```
now, p points to the array.

```
void delay (int amount)
{
    for (int t=0; t != amount; ++t);
    return;
}
```

- a function must be declared, before main() as follows:

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

/* function prototypes */
int decrement (int);
void delay (int);

int main ()
{
        :
    return 0
}

/* function bodies */
int decrement (int arg)
{
    return --arg;
}

int delay (int amount)
{
    int counter = amount;
    while (counter != 0)
    {
        -- counter;
    }
}
```

usually just the parameter types are needed

- HOMEWORK:

to demonstrate, students must write a program that calls the functions, obviously

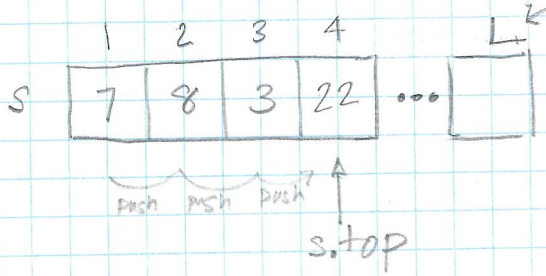
- write a function to return the average of the values in an integer array
- write a function to find median value of an array of sorted numbers (numbers are stored in ascending order)

Note:
- for an even number of elements, the median is the average of the two elements closest to the array middle.
- for an odd number of elements, the median is simply the middle element.

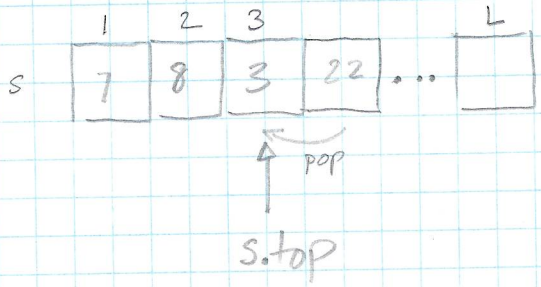
BASIC DATA STRUCTURES

- material here from Introduction to Algorithms, Third Edition, Cormen et al.

stack: LIFO

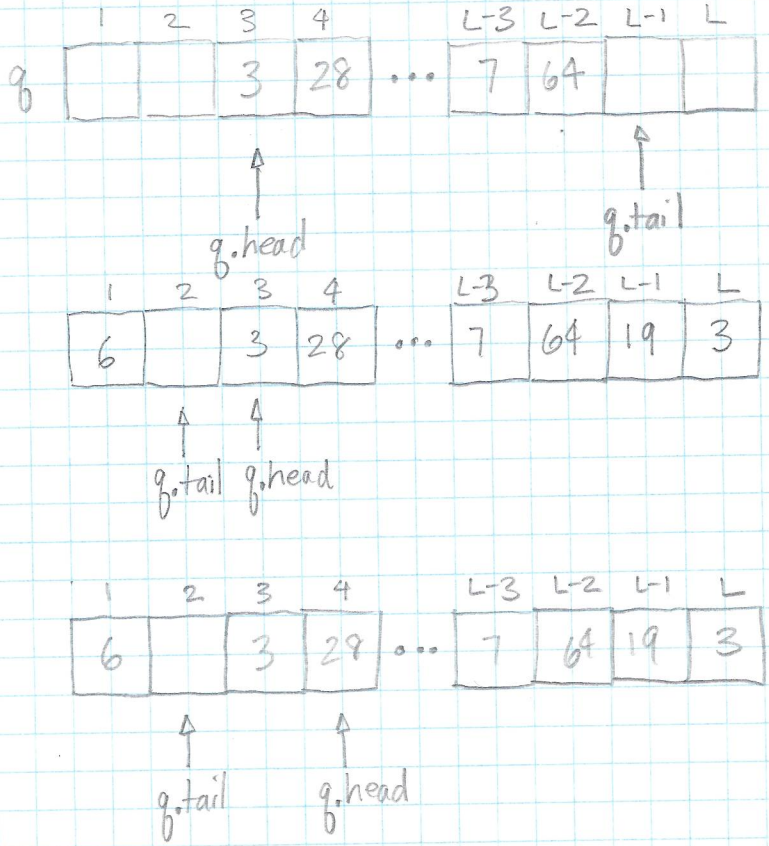


push(s,22);



pop(s);
(returns 22)

queue: FIFO



enqueue(q,19);
enqueue(q,3);
enqueue(q,6);

dequeue(q);
(returns 3)

- stacks & queues are frequently used in embedded system software.

- these are basic data structures that can be realized using a number of techniques, like arrays or pointers.

Algorithms : thinking through w/ pseudo code

Stacks

i) stack_empty (s):

```
if s.top == 0
    return TRUE
else return FALSE
```

ii) push (s, x):

```
s.top = s.top + 1
s[s.top] = x
```

iii) pop (s):

```
if stack_empty (s)
    error "underflow"
else
    s.top = s.top - 1
    return s[s.top + 1]
```

Queues

i) enqueue (q, x):

```
q[q.tail] = x
if q.tail == L
    q.tail = 1
else
    q.tail = q.tail + 1
```

ii) dequeue (q):

```
x = q[q.head]
if q.head == L
    q.head = 1
else
    q.head = q.head + 1
return x
```

Implementation

- let's realize a stack along with empty-querying, pushing and popping in C

- in this example, our stack will store integers (int type)

```
#include <stdlib.h>

#define   L       1024   /* the number of integers
#define   TRUE    1U        we will store */
#define   FALSE   0U

int       S[L];    /* stack declaration as
                      a global variable */

size_t    S_top = 0;  /* the stack pointer s.top */

typedef unsigned short int bool_t;


bool_t stack_empty (void)
{
      if (S_top == 0)
      {
          return TRUE;
      }
      else
      {
          return FALSE;
      }
}

void   push (int x)
{
      ++S_top;
      S[S_top - 1] = x;
      return;
}
```

```
int pop(void)
{
    if stack_empty()
    {
        printf("underflow error\n");
        exit(EXIT_FAILURE);
    }
    else
    {
        return s[s_top--];
    }
}
int main()
{
    /* in-class lab: write a program to illustrate
       use of the stack */

    int loadarr[10] = {52, -29, 36, 1154, 72,
                       0  68, 44, 33, 59};

    /* load stack */
    size_t i;
    for (i=0; i!=10, ++i)
    {
        push(loadarr[i]);
    }

    /* pop stack */
    int x;
    while (stack_empty() == FALSE)
    {
        x = pop();
        printf("%d\n", x);
    }
    return 0;
}
```

- keeping the stack "together" using a struct

```
struct  s_struct
{
    int   data[L];
    size_t top;
};
typedef struct s_struct  stack_t;

stack_t  s;   /* declaration */
/* now, access via */
s.data[i]   } or, if using
s.top++             stack_t *sp; :

                (sp->data)[i];
                (sp->top)++;
```

using struct pointers is necessary in C if we wish to make changes to passed parameters in function calls

- please study the basic data structures starter code in the Git repository

- Homework: write a program, queue.c, which implements a queue, along with the queue functions enqueue() and dequeue(), in the style of stack.c.