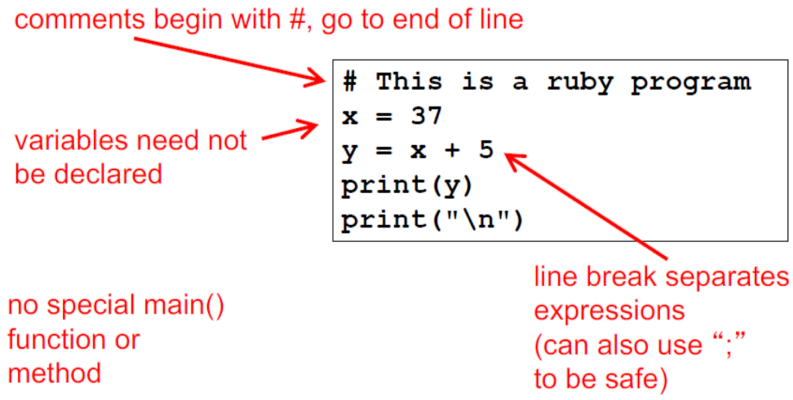
# Ruby

## Language features

- Ruby is: dynamic scripting language, imperative, object-oriented scripting language
- Implicit declarations
  - (Java and C have explicit declarations)
  - ***Explicit declarations*** identify allowed names aka variable must be declared before used) Ex: int x;  x = y+1; return x + y;
  - ***Implicit declarations*** (variable do not need to be declared)  Ex: x = y+1 return x +y;
- Dynamic typing
  - (Java and C have static typing)
- Everything is an object (no distinction between objects and primitive data, even "null" is an object, null is nil in Ruby)
- No outside access to private object state (must use getters and setters)
- No method overloading
- Class-based and Mixin inheritance




**comments**: # this is a comment
**print** = print on one line, needs \n to go to next line
**puts** = prints and goes directly to another line
***super()*** = acts just like call to original method
***aliasing : multiple variables referencing the same object***
Ex:

```
person1 = "Tony"
person2 = person1
```

## Type checking

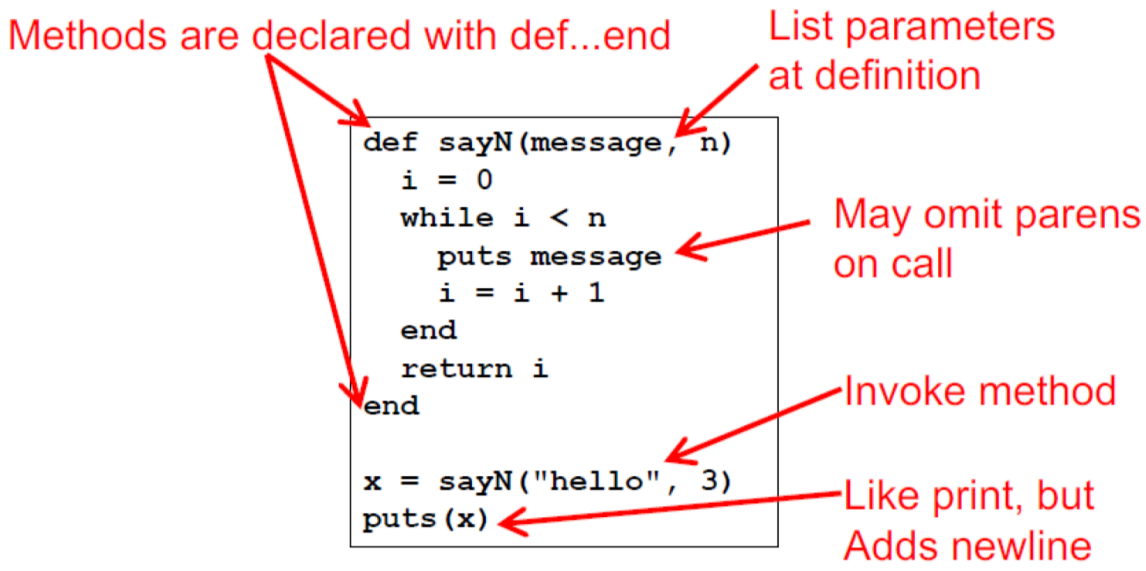
***Static type checking/Static typing***

- Before program is run (types of all expr are determined; disallowed ops cause compile-time error)
- Static types are often explicit (aka manifest) (specified at variable declaration –Java, C; may also be inferred- compiler determines type based on usage)

***Dynamic Type Checking***

- During program execution (can determine type from run-time value; type is checked before use; disallowed operations cause run-time exception)
- Dynamic type are not manifest (aka implicit) ( variable are just introduced/used without types) (Ruby)

## Methods




Should begin with lowercase letter & be defined before they are called
***Constants***: variable names that begin with uppercase letter (only assigned once)

***Formal parameters***: Var parameters used in method Ex: def sayN(message, n)
***Actual arguments***: Values passed in to method at a call Ex: x = sayN("hello", 3)
***Top-level methods*** are global

***Method return values***:

- value of return = value of last executed statement in method
- they can return multiple results (as a list)

***These are the same***:

```
def add_three(x)
  return x+3
end
```

```
def add_three(x)
  x+3
end
```

Instances var (with @) can be accessed only by instance methods. Outside class, they ***required accessors***:

A typical getter

```
def x
  @x
end
```

A typical setter

```
def x= (value)
  @x = value
end
```

***Shortcut in ruby***:

```
class ClassWithXandY
  attr_accessor "x", "y"
end
```

Says to generate the x= and x and y= and y methods

**No method overloading**

- Only one initialize method
- Issue an exception/warning if class defines more than 1 initialize methof (last initialize method defined is the valid on)
- No over loading of methods
- can code up own overloading by using a variable number of args, and checking at run-time the number/types of args

Ex:

```
class Animal
   def eat(food)
      "I ate #{food}"
   end
def eat(food,amount)
      "I ate #{amount} pounds of #{food}"
   end
End
animal = Animal.new
```

puts animal.eat("meat") => ERROR because overloading, only the last method is valid.
puts animal.eat("meat", 23) => I ate 23 pouds of meat

## Naming

- *Class names, module names*, and *constants* should start with uppercase letter.
- *Class variables* start with @@
- *Local variables, method parameters*, and *method names* should all start with lowercase letter or with _
- *Global variables* are prefixed with $
- *Instance variables* begin with @

```
local_variable
CONSTANT_NAME / ConstantName / Constant_Name
:symbol_name
@instance_variable
@@class_variable
$global_variable
ClassName
method_name
ModuleName
```

## Global variables

Class variables beginning with @@ (static in Java)

Class variables beginning with $
Global variables//

## Assignment statements

Chaining assignment statements

```
a = b = 1 + 2 + 3
puts a #=> 6
puts b #=> 6
a = (b = 1 + 2) + 3
puts a #=> 6
puts b #=> 3
```

Parallel assignment statements

```
a = 1
b = 2
a, b = b, a
puts a #=> 2
puts b #=> 1
x = 0
a, b, c = x, (x += 1), (x += 1)
puts a #=> 0
puts b #=> 1
puts c #=> 2
puts x #=> 2
```

## Everything is an object

- *All* values are (references to) objects
- Objects communicate via *method calls*
- Each object has its own (private) *state*.
- Every object is an instance of a *class*
  - an obj class determines its beh
  - class contains method & field def

Examples

- (-4).abs ← No-argument instance method of Fixnum
  - integers are instances of class Fixnum
- 3 + 4
  - infix notation for "invoke the + method of 3 on argument 4"
- "programming".length
  - strings are instances of String
- String.new
  - classes are objects with a new method
- 4.13.class
  - use the class method to get the class for an object
  - floating point numbers are instances of Float

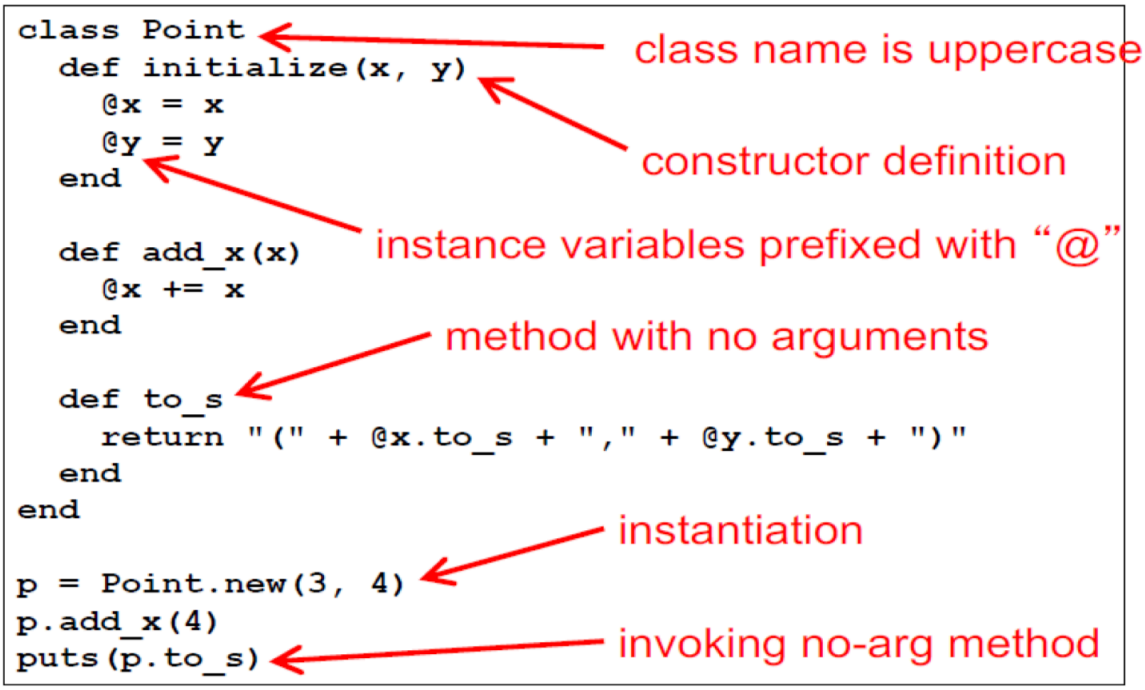
**Nil Object**

- All unitialized fields set to nil
- nil is obj of class NilClass
    - it's a singleton obj (there's only 1 instance of it) (can't have new method)
    - nil has methods like to_s

**Classes**

- Class names begin with uppercase letter
- *class* keyword defines a class
- *new* method creates an obj Ex s = String.new creates new String and s refer to it
- Every class inherits from Object => Classes are also obj
    - So you can manipulate them however you like
    - You can get names of all methods of a class Ex Object.methods
- @x and @y are instance (object) variables

| Object | Class (aka *type*) |
|---|---|
| 10 | Fixnum |
| -3.30 | Float |
| "CMSC 330" | String |
| String.new | String |
| ['a', 'b', 'c'] | Array |
| Fixnum | Class |



**Strings**

- Substitution in double-quoted strings with #{}
- Content of #{} maybe be an arbitrary expression
- ***printf*** ("Hello, %s\n", name);
- ***sprint***(%d: %s", count, Time.now);
- ***to_s*** return a String representation of an object (can be invoke implicitly)
- ***inspect*** converts any obj to a string
- ***String methods:*** .length; s1 == s2 (string contents); s.chomp; s.chomp!

**Control Statements**

***Control statement*** = one that affects which instruction is executed next

***while loops***

- executes its body zero or more times as long as its condition is true

```
i = 0
while i < n
  i = i + 1
end
```

***conditionals***

```
if grade >= 90 then
  puts "You got an A"
elsif grade >= 80 then
  puts "You got a B"
elsif grade >= 70 then
  puts "You got a C"
else
  puts "You're not doing so well"
end
```

***All Ruby conditional and looping statements must be terminated with end keyword.***

In conditionals:
***guard***: expression that determines which branch is taken.

```
if grade >= 90 then
...
```

Guard

The true branch is taken if the guard evaluates to anything except: false; nil

| ***Unless cond then … else … end*** | ***Until cond body end*** |
|---|---|
| Same as "If not cond then … else … end" | • Same as "while not cond body end"<br>• executes the body as long as boolean-expression is false |

```
unless grade < 90 then
  puts "You got an A"
else unless grade < 80 then
  puts "You got a B"
end
```

```
until i >= n
  puts message
  i = i + 1
end
```

You can write if and unless after a modifiers.

Ex puts “You succeed!” if grade >= 90

### *Case* statement

- When comparison returns true, search stops and body associated with comparison is executed.
- statement then returns the value of the last expression executed.
- If no comparison matches and an else clause is present, its body will be executed; otherwise, the statement returns nil.

```
number = 11
case number
  when 1, 3, 5, 7, 9
    puts "Odd."
  when 0, 2, 4, 6, 8, 10
    puts "Even."
  else
    puts "Number is out of range."
end
```

### Mixins

- Another form of code re-use
- include A “inlines” A’s methods at that point
- referred-to variables/methods captured from context
- in effect: it adds those methods to curr class

```
class OneDPoint
  attr_accessor "x"
  include Comparable
  def <=>(other)# used by Comparable
    if @x < other.x then return -1
    elsif @x > other.x then return 1
    else return 0
    end
  end
end
```

```
p = OneDPoint.new
p.x = 1
q = OneDPoint.new
q.x = 2
x < y # true
puts [y,x].sort
# prints x, then y
```

**Arrays**

| | |
|---|---|
| a = [ "number", 1, 2, 3.14 ] | # Array with four elements. |
| puts a[0] | # Access and display the first element.<br>#=> number |
| a[3] = nil | # Set the last element to nil. |
| puts a | # Access and display entire array.<br>#=> number 1 2 nil |
| myarray = [ 1, 2, 3, 4, 5, 6 ]<br>puts myarray[1...3] | # [i...j] from index i to j<br># excluding j<br># ***Exclusive range*** => 2 3. |
| puts myarray[1..3] | # [i..j] from index i to j<br># including j<br>***# Inclusive*** range. => 2 3 4 |
| puts myarray[1,3] | # Range between 1st up to 3rd<br># ***consecutive, inclusive***.<br>#=> 2 3 4. |

Ruby ***allows a negative index***, forcing the array to count from the end.

```
irb(main):001:0> myarr = [1, 2, 3, 4]
=> [1, 2, 3, 4]
irb(main):002:0> myarr[-2]
=> 3
irb(main):003:0> myarr[-1]
=> 4
irb(main):004:0> myarr[-4]
=> 1
irb(main):005:0> myarr[-5]
=> nil
irb(main):006:0> myarr[-6]
=> nil
```

- Arrays are growable (increase in size automatically as you access elements)and can also shrink. (contents shift left when you delete element)
- [] is the empty array == to Array.new

iterate through an array with while:

```
a = [1,2,3,4,5]
i = 0
while i < a.length
  puts a[i]
  i = i + 1
end
```

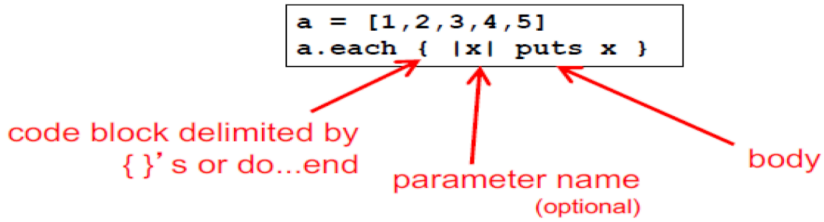
**.pop() & .push()**

```
irb(main):005:0> print myarr
[1, 3, 4]=> nil
irb(main):006:0> myarr.push(4)
=> [1, 3, 4, 4]
irb(main):007:0> myarr.pop(1)
=> [4]
irb(main):008:0> print myarr
```

## Code Blocks

***code block***: piece of code that is invoked by another piece of code. They are useful for encapsulating repetitive computations

***Array iteration with code blocks*** : array class has ***each*** method




*Sum up elements of an array*:

```
a = [1,2,3,4,5]
sum = 0
a.each { |x| sum = sum + x }
printf("sum is %d\n", sum)
```

Print out each segment of string as divided up by commas (can use any delimiter):

```
s = "Student,Sally,099112233,A"
s.split(',').each { |x| puts x }
```

More examples of codeblocks:

```
3.times { puts "hello"; puts "goodbye" }
5.upto(10) { |x| puts(x + 1) }
[1,2,3,4,5].find { |y| y % 2 == 0 }
[5,4,3].collect { |x| -x }
```

| **n.times** | **n.upto(m)** | **a.find** | **a.collect** | **a.collect!** |
|---|---|---|---|---|
| Runs code block n times | Runs code block for int n..m | Return first element x of arr such that the block return true for x | Applies block to each element of array and return new array | Modifies the original |
| **Output** | | | | |
| TIMES:<br>hello<br>bye<br>hello<br>bye<br>hello<br>bye | UPTO:<br>6<br>7<br>8<br>9<br>10<br>11 | FIND:<br>false<br>true<br>false<br>true<br>false | COLLECT<br>-5<br>-4<br>-3 | Same result but the initialize arr[5,4,3] no longer exist it is now [-5,-4,-3] |

```
File.open("test.txt", "r") do |f|
  f.readlines.each { |line| puts line }
end
```

```
10.downto(1) { |count| puts count }
```

```
0.step(10,2) { |count| puts count }
```

| File.open(".txt", "r") | f.readlines | f.each_line | n.downto(m) | n.step(m,x) |
|---|---|---|---|---|
| Takes codeblock with file argument. Will automatically closed after block executed | Reads all lines from a file ande returns an arr of lines read | Apply code block to each newline-separated substring | Runs code block for int n..m<br>DOWNTO:<br>10<br>9<br>8<br>7<br>6<br>5<br>4<br>3<br>2<br>1 | Runs code starting n to m by adding x<br>STEP:<br>0<br>2<br>4<br>6<br>8<br>10 |

**Yield**

Any method can be called with a code block. (inside method, block is called with *yield*)

```
def countx(x)
  for i in (1..x)
    puts i
    yield
  end
end

countx(4) { puts "foo" }
```

After the code block completes

```
1
foo
2
foo
3
foo
4
foo
```

**High-order programming**: methods take other methods as args like each method

## Ranges

1.. 3 : obj of class Range. inclusive
1... 3 : obj of class Range. Exclusive
can be used for letters

***method to_a***: will convert ranges to array

## Regular expressions

***regular expression:*** way of specifying a pattern of characters to be matched in a string.

| Pattern | Description |
|---|---|
| `/Lisp\|Lava/` | Matches a string containing *Lisp*, or *Lava*. |
| `/L(isp\|ava)/` | As above. |
| `/ab+c/` | Matches a string containing an *a*, followed by one or more *b*s, followed by a *c*. |
| `/ab*c/` | matches a string containing an *a*, followed by zero or more *b*s, followed by a *c*. |
| `.` | Matches any character. |
| `/[Colloqui[um\|a]/` | Matches *Colloquium*, or *Colloquia*. |

## Introspection

***Introspection***: ability of a computational system to consult (but not modify) its own structure.

In Ruby, we can obtain the following type of knowledge about a program:

- What objects it contains.
- The contents and behaviors of objects.
- The current class hierarchy.

Example:
We can execute reflective queries to obtain knowledge about the system.

```
require "CoordinateV2.rb"
require "XYZCoordinate.rb"
p1 = Coordinate.new(0, 0)
p2 = XYZCoordinate.new(0,0,0)
def p2.whatIam
  return "The origin on the 3D system."
end
```

If we inspect the system for objects of type Coordinate

```
ObjectSpace.each_object(Coordinate) { |p|
  puts p.inspect
}
```

Results:

```
#<XYZCoordinate:0x28455d8 @y=0, @z=0, @x=0>
#<Coordinate:0x2846028 @y=0, @x=0>
```

We can check whether or not a particular object may respond to a message

```
puts p1.respond_to?("setX")        #=> false
puts p2.respond_to?("whatIam")     #=> true
```

We can also determine the class and unique id of objects, and test their relationship to classes:

```
puts p1.id                          #=> 21113660
puts p1.class                       #=> Coordinate
puts p2.class                       #=> XYZCoordinate
puts p2.instance_variables          #=> @y @z @x
puts p2.kind_of? Coordinate         #=> true
puts p2.kind_of? XYZCoordinate      #=> true
puts p1.kind_of? XYZCoordinate      #=> false
puts p2.instance_of? Coordinate     #=> false
puts p2.instance_of? XYZCoordinate #=> true
```

# C

C is Procedural programming

Five official "versions" of C

- K&R C
- C90
- NA1
- C99
- C11

## C standard library

| | |
|---|---|
| math.h | Defines common mathematical functions. |
| stdio.h | Defines core input and output functions. |
| stdlib.h | Defines numeric conversion functions, pseudo-random number generation functions, memory allocation, process control functions. |
| string.h | Defines string manipulation functions. |

## Data types

***Data type (or simply a type):*** description of the possible values that a construct can store or compute to, together with a collection of operations that manipulate that type.

**Class of data types**

| | |
|---|---|
| ***Boolean type*** | contains the values true and false. |
| ***Numeral type*** | includes integers that represent whole numbers and floating points that represent real numbers. |
| ***Character type*** | member of a given set (ASCII) and, finally, strings are sequences of alphanumeric characters |

*Distinction between simple types and composite* (or aggregate) types based on whether or not the values of a type can contain subparts.

$$short\ int \leq int \leq long\ int\ float \leq double \leq long\ double$$

***Constant***: defines a data type whose value cannot be modified.
Ex: float **const** pi = 3.14 or with **#define** TRUE 1

***Composite type*** : one that is composed by primitive types or other composite types.
Composite type == ***data structure***: a way to organize and store data so that it can be accessed and manipulated efficiently.
Common composite types = array and records

## Strings

***String*** (***multibyte string***): contiguous sequence of characters (of type char) terminated by and including the first null character.
***length of a string***: number of bytes preceding null character and value of a string is the sequence of the values of the contained characters, in order.
***null wide character***: wide character with code value zero.
***Wide string***: contiguous sequence of wide characters (of type wchar_t) terminated by and including the first null wide character.
***Length of a wide string***: number of wide characters preceding the null wide character and the value of a wide string is the sequence of code values of the contained wide characters, in order.

Example

```
/*
 * A program that illustrates strings in C.  Designed for C99, but should run
 * fine in C90 with a lot of warnings.
 */
#include <stdio.h>
#include <string.h>
#include <wchar.h>

/* Simple strings from the basic character set */
char s1[] = {'d', 'o', 'g', (char)0};
char s2[] = {'d', 'o', 'g', '\0'};
char* s3 = "dog";

void inspectString(char* s) {
   int i, n;
   printf("[%s] length=%d codepoints=[ ", s, strlen(s));
   for (i = 0, n = strlen(s)+1; i < n; i++) {
      printf("%02x ", (unsigned char)s[i]);
   }
   printf("]\n");
}
int main() {
   inspectString(s1);
   inspectString(s2);
   inspectString(s3);
return 0;
}
```

## Type conversion

In C, **implicit type conversion** (or coercion) is the automatic type conversion done by the compiler.

| | |
|---|---|
| %c | char single character |
| %d or %i | int signed integer |
| %e or %E | float or double exponential format |
| %f | float or double signed decimal |
| %p | pointer address stored in pointer |
| %s | array of char sequence of characters |

## Functions

***function***: (named) block that normally receives some input, performs some task and normally returns a result.

The ***general form*** of a function definition in C is

return-type function-name ( parameter-list ) { body }

where return-type is the type of the value that the function returns, function-name is the name of the function, and parameter-list is the list of parameters that the function takes, defined as

( type1 parameter1, type2 parameter2, ... )

```
#include<stdio.h>
long factorial(int);
int main() {
  ...
}
long factorial(int n) {
  ...
}
```

## Global and local variables

***Global variable:*** defined at the top of the program file and can be accessed by all functions.
***Local variable:*** accessed only within the function, which it is declared, called the scope of the variable.

Global variables have default initializations, whereas local variables do not.

***Shadowing***: (not good programming practice) in the case where the same name is used for a global and local variable then local variable takes preference within its scope.

```
#include<stdio.h>
int a = 3;
int func() {
  int a = 5;
  return a;
}
int main() {
  printf("From main: %d\n", a);
  printf("From func: %d\n", func());
  printf("From main: %d\n", a);
}
```

The output is as follows:

```
From main: 3
From func: 5
From main: 3
```

### Variable and function modifiers

Two modifiers are used to explicitly indicate the visibility of a variable or function: **extern modifier** indicates that a variable or function is defined outside the current file, whereas the **static modifier** indicates that the variable or function is visible only from within the file it is defined in.

The **default (i.e. no modifier)** indicates that the variable or function is defined in the current file and it is visible in other files.

| | |
|---|---|
| ***extern*** | Variable/function is defined outside of current file. |
| (***blank***) | Variable/function is defined in current file and visible outside. |
| ***static*** | Variable/function is visible only in current file. |

### Sorting

function bubbleSort()

```
void bubbleSort(int numbers[], int
array_size) {
   int i, j;
   for (i = (array_size - 1); i > 0; i--) {
     for (j = 1; j <= i; j++) {
       if (numbers[j-1] > numbers[j])
         swap(&numbers[j-1],
&numbers[j]); }}}
```

function swap()

```
static void swap(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp; }
```

## Arrays

C arrays are extraordinarily primitive. They do **not** know how big they are, so you can read and write beyond the array bounds.

```
int f() {
    int x;
    int a[4];
    int y;
    int z = 23;
    a[5] = 100;
    return z;
}
```

Arrays can be either ***static*** (i.e,. fixed-length) or ***dynamic*** (i.e. expandable). An array of size n is indexed by integers from 0 up to and including n − 1.

There are **no array-index-out-of-bounds-exceptions** in C because

- C does not "remember" how big you created your array
- C does not have exceptions

***Example of program with arrays:***

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

// To get primes up to and including 1000, the sieve has to have a slot at
// index 1000.  But indices must start at 0, so there have to be 1001 slots
// in the array.

#define SIZE 1001

// Fills the first n slots of array s with the given value.
void fillArray(bool s[], bool value, int n) {
  for (int i = 0; i < n; i++) {
    s[i] = value;
  }
}

// This function writes false in each slot of the array corresponding to a
// nonprime number.  First, we know 0 and 1 are not prime. Then for each
// value starting with 2, if the value is still thought to be prime, we
// write false in each slot corresponding to its multiples.
void checkOffComposites(bool s[], int n) {
  s[0] = false;
  s[1] = false;
  for (int i = 2; i * i < n; i++) {
    if (s[i]) {
      for (int j = i + i; j < n; j += i) {
        s[j] = false;
```

```
        }
      }
   }
}

// main() just calls the worker functions.
int main() {
   bool sieve[SIZE];
   fillArray(sieve, true, SIZE);
   checkOffComposites(sieve, SIZE);
   displayTrueIndices(sieve, SIZE);
   return 0;
}
```

## Pointers

***Pointer***: type that references ("points to") another value by storing that other value's address.

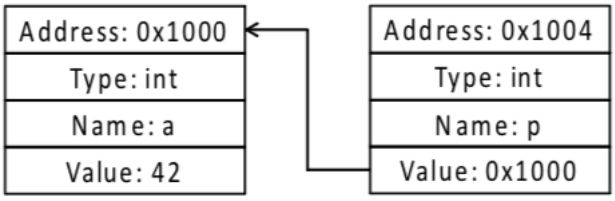
***pointer variable*** (also called an ***address variable***) is declared by putting * in front of its name. Ex int *ptr;

| | | | |
|---|---|---|---|
| * | ***deference operator*** | Given a pointer, obtain the value of the object referenced (pointed at) | (*p).x or p->x |
| **&** | ***address of operator*** | Given an object, use & to point to it. The & operator returns the address of the object pointed to. | `p.x` |

Example of pointers:

```
#include <stdio.h>
 int main() {
      int a=42;
      int *p;
      p= &a;
      printf("p: %d\n", *p);
      return 0;
}
```

**Output: 42**



## Pointers and arrays

Elements of an array are assigned consecutive addresses. We can use a pointer to an array in order to iterate through the array's elements.

```
int arr[5];
int *ptr;
ptr = &arr[0]; //we assign the first element of the array as the value of the pointer
*(ptr + 1)       //same arr[1]
```

Example explore pointer arithmetic to assign an array to a pointer, and then use pointer to display values of first three elements of array. We're displaying content of array by ***dereferencing the pointer***.

```
#include <stdio.h>
int main() {
      int arr[5] = {1, 3, 5, 7, 11};
      int *ptr;
      ptr = &arr[0];
      printf("arr[0]: %d, arr[1]: %d, arr[2]: %d\n",
           *ptr, *(ptr + 1), *(ptr + 2));
      return 0;
}
```

The **output** is: arr[0]: 1, arr[1]: 3, arr[2]: 5

## Function pointers

***Function pointers***: A pointer can also refer to a function, since functions have addresses.

Ex long (*ptr)(int);

this declares a function pointer;

It points to fct that takes integer argument and returning long integer.

ptr = &factorial;

this makes ptr point to function factorial(..).

function can be invoked by dereferencing the pointer while passing arguments as any regular function call, only in this case we refer to this as an ***indirect call***.

## Records

***Record/structure***: collection of elements, fields (or members), which can possibly of different types.

Syntax of *declaring a structure* in C is

```
struct <name> {
        field declarations
};
```

Example of a record

```
#include<stdio.h>
   typedef struct {
float x;
     float y;
   } coordinate;
   int main() {
     coordinate p1 = {0, 0};
     coordinate p2 = {.x = 1, .y = 3};
     coordinate p3;
     coordinate p4;
     p3.x = 2;
     p3.y = 7;
     p4 = p3;
     printf("p1 = (%.0f, %.0f)\n", p1.x, p1.y);
     printf("p2 = (%.0f, %.0f)\n", p2.x, p2.y);
     printf("p3 = (%.0f, %.0f)\n", p3.x, p3.y);
     printf("p4 = (%.0f, %.0f)\n", p4.x, p4.y);
     return 0;
}
```

**Output**

```
p1 = (0, 0)
p2 = (1, 3)
p3 = (2, 7)
p4 = (2, 7)
```

We use the dot (.) operator to access fields of individual records: line[0].x accesses the x field of the first element (record) of line.

```
int main() {
      coordinate line[2] = {
            {0, 0},
            {11, 19}
      };
      printf("Line points: (%.0f, %.0f), and (%.0f, %.0f).\n",
                  line[0].x, line[0].y, line[1].x, line[1].y );
}
```

```
Line points:

(0, 0),

(11, 19)
```

**Memory Management**

Consider this example

```
#include<stdio.h>
#include <stdlib.h>
int main() {
    int *array = malloc(3 * sizeof(int));
    if (array == NULL) {
        printf("ERROR: Out of memory.\n");
        return 1;
    }
    *array = 1;
    *(array + 1) = 3;
    *(array + 2) = 5;
    printf("%d\n", *array);
    printf("%d\n", *(array + 1));
    printf("%d\n", *(array + 2));
    free(array);
    return 0;
}
```

**Output** 1 3 5

| | |
|---|---|
| int *array = malloc(3 * sizeof(int)); | We *request the allocation of enough memory for an array of three elements* of type int. This is a request & the allocation of memory is *not guaranteed to succeed*.<br>○ If successful, function malloc *returns a pointer to a block of memory*.<br>○ If not successfull, malloc will *return the special value NUL*L to indicate that for some reason the memory has not been allocated. |
| if (array == NULL) {...} | To indicate success we now have to verify that our array pointer is not NUL. |
| *array = 1;<br>*(array + 1) = 3;<br>*(array + 2) = 5; | Then proceed to assign values to the elements of the array and subsequently display them. |
| free(array); | Once we no longer need the array, we have to release the allocated memory back to the system. |

| ***Memory management functions*** | |
|---|---|
| malloc | Allocates the specified number of bytes. |
| realloc | Increases or decreases the size of the specified block of memory. |
| calloc | Allocates the specified number of bytes and initializes them to zero |
| free | Releases the specified block of memory back to the system. |

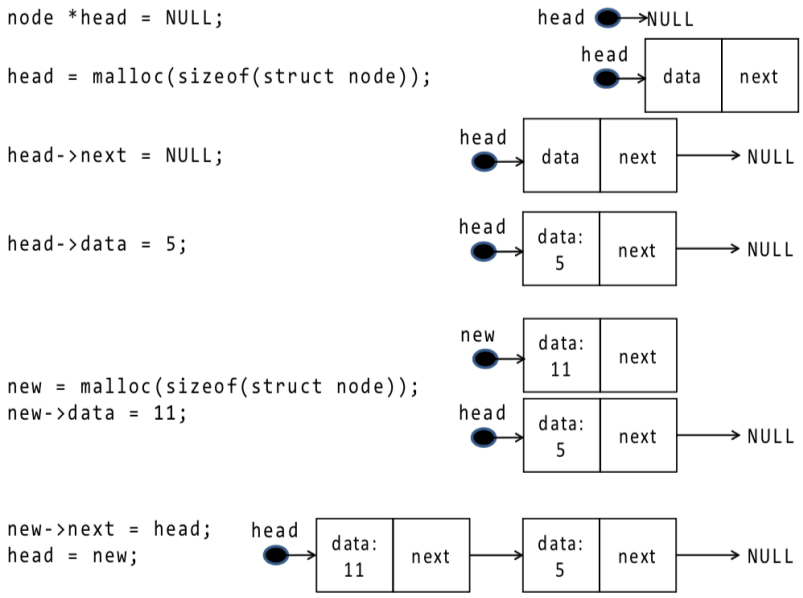
**Linked list data structure**

It can be used to implement several common abstract data types, including stacks, and queues.

Example

```
#include<stdio.h>
#include <stdlib.h>
/*A node is represented as a record */
struct node {
      int data;
      struct node *next;
};

int main() {
/* Initially the list is empty, thus the head of the list points to NULL */
    struct node *head = NULL;
    struct node *new;
/* request memory for the head of the list*/
    head = malloc(sizeof(struct node));
    if (head == NULL) {...}
/*have the head's next field point to null and assign some value to the data field*/
    head->data = 5;
    head->next = NULL;
    new = malloc(sizeof(struct node));
    if (new == NULL) {...}
    new->data = 11;
/*need to make sure that next field of new item points to node currently pointed to by
head and new item becomes new head*/
    new->next = head;
    head = new;
    printf("%d ", head->data);
    printf("%d ", (head->next)->data);
    return 0;
}
```




**Files**

How to open and close a file

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

int main(int argc, char** argv) {
      if (argc != 3) {
            puts("Exactly two commandline arguments needed");
            return 1;
      }
      else {
            FILE* in = fopen(argv[1], "r");
            if (!in) {
               printf("File %s does not exist\n", argv[1]);
               return 2;
      }
      FILE* out = fopen(argv[2], "w");
      while (1) {
               int c = fgetc(in);
               if (c == EOF) break;
               fputc(toupper(c), out);
      }
      fclose(in);
      fclose(out);
      return 0;
   }
}
```