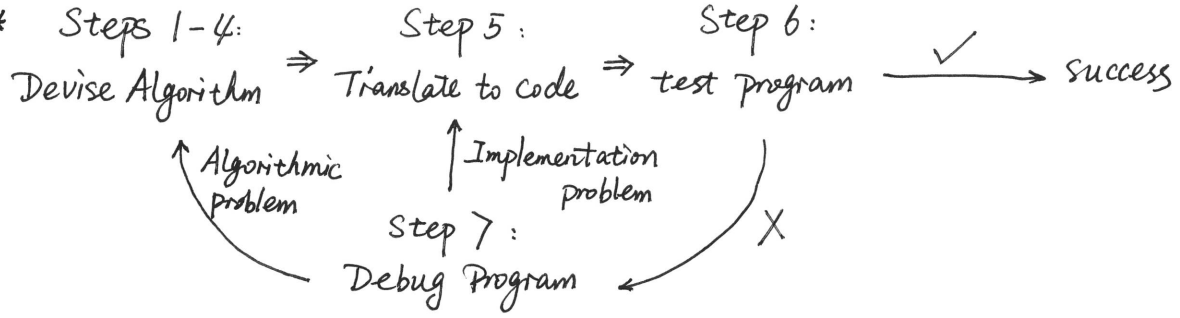
All of Programming.

notes by Ellie Zheng.

Chapter 1 : Introduction

08/30/2016

■ Seven-Step :



* The first 4 steps :

Step 1 : Work an instance yourself.

possible problems { problem is ill-specified
lack domain knowledge.

Step 2 : Write down exactly what you just did.

to be precise and arithmetic. How you represent your problems with numbers.

Step 3 : Generalize your steps.

① replace the particular values with mathematical expressions of the parameters.

② find repetition — find steps which are almost repetitive and see the pattern.

Step 4 : Test your algorithm.

with good & complete test case, especially corner cases.

check for { mis-generalizing in Step 3.
cases that are not considered in designing.

Chapter 2 Reading Code

■ Terms

- Identifier : a word that can be used to name sth.
  can contain letters, "_", and numbers (not start with)
- lvalue / rvalue : an object that has a name / not an object
  names / expression, seperated by assignment operator.
- expression : whether or not persist beyond a single express
  a combination of values and operations which evaluates to a value.
- Fuaction : gives a name to a parameterized computation.
- Frame (stack frame) : holds variables belonging to one function.
- Block : codes between matching open & close curly braces.
- Escape sequences : \ characters, gives the remaining characters special meaning.
- selection expression / label : immediately after switch / case.
- Syntatic sugar : shorthand for a variety of common operations.
- Syntax / Semantics : grammatical rules / meaning of programs.

■ Variables

- *declaration + assignment

  can be seperated : `int a, b;` // type name;

  or combined : `int a = expression;` // Initialization.

- Uninitialized variable :

  the value of it will be whatever value happened to be in the memory location previously.

- Operators.
  - mathematical operators : +, -, *, /, % (↓ mod)

    rules { precedence
    | associativity
  - logical operators
    * relational operators : ==, !=, <=, <, >, >=
    * boolean operators : !, &&, ||
      - ! is unary operator, i.e., one operand.
      - &&, || evaluate the first operand, if the entire result is determined, then the second operand is not evaluated at all.
    * False = 0, True = any non-zero values.
- Scope : the region of code which it is visible
  - { global
    | local
  - The scope of a local variable is within a block, i.e., between { } where it is declared.

Chapter 3 Types

type = size + interpretation of a series of bits

■ Numbers - hardware representation.

- Integers :

75	Decimal (unsigned) : base 10
$1+2^1+2^3+2^6=75$	↕
01001011	Binary : base 2.
0100 1011 $2^2=4$ $1+2^1+2^3=11=B$	↕
0x4B	Hexadecimal (Hex) : base 16

```
2 | 75 --- 1
2 | 37 --- 1
2 | 18 --- 0
2 | 9  --- 1
2 | 4  --- 0
2 | 2  --- 0
2 | 1  --- 1
    0
```

```
void DtoB ( int d ) {
        if ( d/2 )
            DtoB ( d/2 );
        Cout << d % 2 ;
}
```

- Points : fractional portion.

Decimal (0.625) → Binary : = 0.101

$0.625 \times 2 = 1.250$ --- 1
$0.250 \times 2 = 0.500$ --- 0
$0.500 \times 2 = 1.000$ --- 1

Binary (0.101) → Decimal : = 0.625

binary point

$1\times2^{-1} + 0\times2^{-2} + 1\times2^{-3}$
$= 0.5 + 0.125 = 0.625.$

- signed integer : 2's complement,

N-bit int, the most significant bit is sign bit, represent $-2^{N-1} \sim 2^{N-1}-1$

```
0111 = 7
0001 = 1
0000 = 0
1111 = -1
1110 = -2
1000 = -8
```

$0001_2 \xrightarrow{1st} 1110_2 \xrightarrow{2nd} 1111_2$
(1_{10}) (-1_{10})

```
 0001 | 1
+1111 |+-1
------------
1 0000 | = 0.
```

■ Basic Data Types

type	size (typical)	Hardware representation / interpretation
char	1 byte (8 bits)	'A' = 65 (ASCII character)
int	4 byte (32 bits)	* unsigned : 11001000 = 200 * signed : 2's complement (补码) 1 1001000 sign bit ↓ 取非 (反码) 1 0110111 ↓ 加1 (原码) 1 0111000 = -56 |signed| = 2^n - |unsigned|
float	4 byte (32 bits)	1 bit | 8 bits | 23 bits
double	8 byte (64 bits)	1 bit | 11 bits | 52 bits sign exponent mantissa (floating point notation) $(-1)^s \times m \times 2^e$

- short / long.

■ Types of expressions

1. Constant.
   (i) Default : int or double.
   (ii) Modify types : apply a letter suffix
   U - unsigned
   L - long
   f - float
2. ~~Ex~~ Variables : whatever types at declaration.
3. Expressions with binary operators : e1 op e2
   (I) type(e1) == type(e2) == type(e1 op e2)

(II) Type (e1) != type (e2)

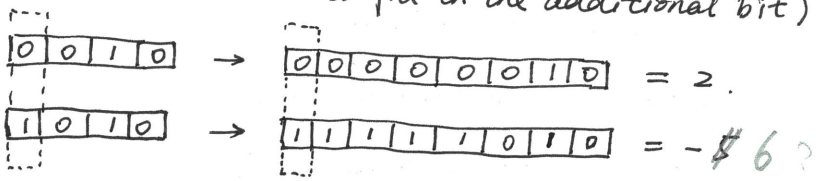
(i) Type conversion (type promotion)

↳ which compiler attempts to add before issueing an error msg.

① Bit representations must be changed:

1) smaller signed int → longer signed int :

sign extended (the sign bit copied an appropriate number of times to fill in the additional bit)

e.g., 

2) smaller unsigned int → longer unsigned int .

zero extended (the additional bits are filled with zeros)

e.g., 

3) longer int → shorter int :

bit pattern truncated (the most significant bits are thrown away)

4) old type → new type :

fully calculate the representation of the value of the new type.

② Changing how the type is interpreted.

e.g.1 comparing a signed int and an unsigned int

```
unsigned int bigNum = 100;
int littleNum = -100;
if ( bigNum > littleNum)
        printf ("Not the case\n");
else
        printf ("True\n");
```

In such cases, warnings will be produced by the compiler.

e.g.2, char + int → int. as there are more bits in int than in char.

Overflow (underflow): an operation results in a number that is too large to be represented by the result type of the operation.

(ii) Casting

↳ which a programmer explicitly requests.

place the desired type in parenthesis before the expression.

e.g., `int a = 4, b = 5;`

`a/b` → int, 0.

`a/(double)b` → double, 0.8.

`a - b/2` → int, 2

`a - b/2.0` → double, 1.5

`a - b/2f` → double, 1.5.

■ Custom Data Types.

(I) struct

- bundle multiple variables into a single entity, with a logical purpose.
- convention: `_tag_t`.
- Declare:

```
struct _~~rec~~tag_t {
    int a;
    ⋮
};   Don't forget ;
typedef struct _tag_t tag_t;
```

(custom type.)

or

```
typedef struct _tag_t {
    int a;
    ⋮
} tag_t;
```

- use:

```
tag_t myVariable;
myVariable.a = 1;
```

or

```
struct _tag_t myVariable;
```

(II) typedef.

- syntax: typedef type type-name.
- convention: tag_t

(i) simplify the use of structs.

(ii) name a type according to its meaning and use.

e.g.,
```
typedef unsigned int rgb_t;
int main(void) {
    rgb_t red, green, blue;
    ⋮
}
```

(iii) Limit the definition of a particular type to a single place in the code base. (= 定义宏，便于批量修改)

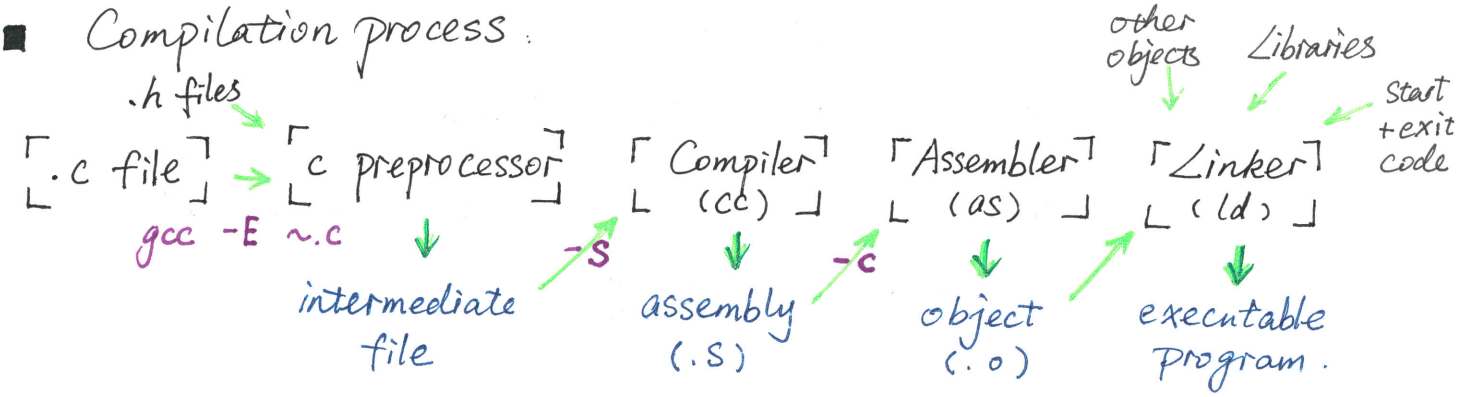
(III) enum: enumerated type

- named constant.
- useful for a type of data with a set of values that labeled by their conceptual name
  (the particular numerical values do not matter.
  | or they occur naturally in a sequential progression.
- syntax:

```
enum _tag_t {
    ITEM1 = 0,   // = 0 by default, or can be set to a
    ITEM2 = 5,   //   value explicitly.
    ITEMn
};
void printItem ( enum_tag_t enum_name ) {
    switch ( enum_name ) {
        case ITEM1:
            ⋮
        case ITEM2:
            ⋮
    }
}
```

Chapter 5 Compiling and Running.

■ Compilation process:



□ 1. Preprocessor

↳ (take .c file + headers included
| expand macros.

- # include directives : - not executed
  - tell the preprocessor to literally include the contents of the named file.
  - Convention : use < > for standard c headers,
    use " " for custermized headers.

- What's in header files :

① function prototype : - tells the compiler that the function exists somewhere in the program.
- tells return type + argument types and numbers for checking.

e.g., int function (double a, char* b);

② macro : - advantage : ① convenient for bulk changes.
② more readable
③ portability (e.g., INT_MAX in limits.h)

(i) Define a constant :

e.g., # define ~~Exit~~ EXIT_SUCCESS 0.

in stdlib.h.

(ii) take arguments, and expanded textually by the compiler.

The text resulting from the expansion needs to be valid.

e.g., # define SQUARE (x) ((x)*(x))

then SQUARE (y-3) would be expanded into:

((y-3)*(y-3))

③ type declarations.

It can be in standard C headers, such as:

- FILE type in stdio.h.
- int32_t type in stdint.h that guaranteed to be a 32-bit signed int on any platform.

or can be in custermized headers, with syntax the same as typedef.

☐ 2. Compiler

↳ reads the pre-processed source code, and translates it into assemby.

- Assembly
  - ① The lowest level type of human readable code.
  - ② Each statement = one machine instuction.
- may ask the compiler to optimize the code to make the resulting assembly run faster, gcc -O_ option.

- compiling process :

  STEP 0 : Lexical Analysis

  error if identifiers are illegal, or illegal characters appear.

  STEP 1 : Syntatic Analysis – Parsing

  - understand the program according to the rules of C.
  - error if has incorrect syntax (gramma)
    e.g., missing semicolon, undeclared, ~~wt~~ invalid expression (1+2+;), etc.

  STEP 2 : Somatic Analysis – type-checking.

  - determines the type of every expression, ~~accordi~~ and checks if they are compatible with the way being used according to language specifications.
  - checks that functions have right # number and type of arguments passed to them.

☐ 3. Assembler

↳ assemble the assembly into an object file.

- object file
  - contains machine-executable instructions (numerical encoding)
  - may reference functions it doesn't define
    e.g., those in C library, or written in other files.
  - each source file can be individually compiled to an object file, to be linked together.

4. Linker

↳ { resolve the references of object files, e.g., ~~types~~, functions, defined symbols.

combine one or more object files, various libraries, and some startup code, and produces the actual executable binary.

Linker errors (ld):

① Symbol error means:

- did not define the symbol;
- did not include the object file that define the symbol when linking.
- the symbol was specified as only visible inside the .o. (static)

② Don't include any files twice

③ Don't #include <....c>, and .h only contains prototype (not the function's definition).

④ Must specifically request the linked to link with external libraries other than C library, with the "-l" option.

Tips for dealing with cc errors:

1. If later errors are confusing, fix the first error, then try to recompile before attempting to fix them.
2. If parts of an error msg are complete unfamiliar
   - if the rest of the error msg makes sense, try to understand and fix them
   - if not, Google.

3. Use editor's features to find mismatched braces and ( )s.
4. Be confident in your fix for an error. If not understand what's wrong and how to fix it, find out and be sure, rather than randomly changing things.

Compiling options:

-Wall -Werror -Wsign-compare -Wwrite-strings -Wtype-limits.

Chapter 6 Testing and Debugging

■ Testing

↳ finding bugs in code.

key: incremental testing (bottom-up): one func. $\xrightarrow{\text{tested}}$ next func...

1. Black Box Testing

↳ consider only the expected behaviors of the function to devise test cases, which are likely to be error prone

① test the basic functionality of the code.

② devise corner cases (input that require specific behavior)

2. White Box Testing

↳ involve examining the code to devise test cases. – test coverage

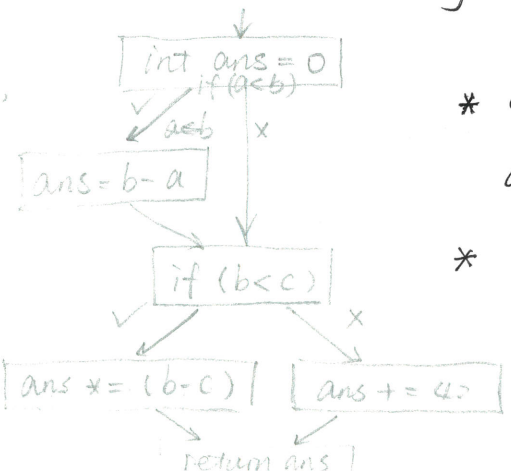
① Statement coverage

- every statement in the function is executed.

② desision coverage

- all possible outcomes of decisions (e.g., if/else, switch/case choices) are exercised.
- i.e., cover every edge in a control flow graph (CFG)

e.g.,



* a call graph can draw how the execution arrow moves between functions.
* a CFG is for one function at a time.

③ path coverage.

– span all valid paths through CFGs.

3. Regression Testing.

↳ make sure the modification have not broken existing functionality.

to run the regression test suite which have worked in the past to ensure they still work on the current code.

① "nightly regressions" (automated)

② before checking to Git.

4. Code Review.

↳ code walk-through to a reviewer.

■ Techniques in testing.

1. Generate Tests.

① how to generate.

- according to some algorithm. e.g., iterate.
- random testing

② what to test.

- answer – implement both the complex / optimized algorithm and the simpler / slower algorithm, and test their results on many test cases.
- properties – check certain properties the system should be

obeyed, and check for certain types of errors, which require a test harness - an additional program to run and test the main parts of the code.

2. Asserts

↳ checking invariants maintained in the middle of execution.
= a property that is always true at a given point.

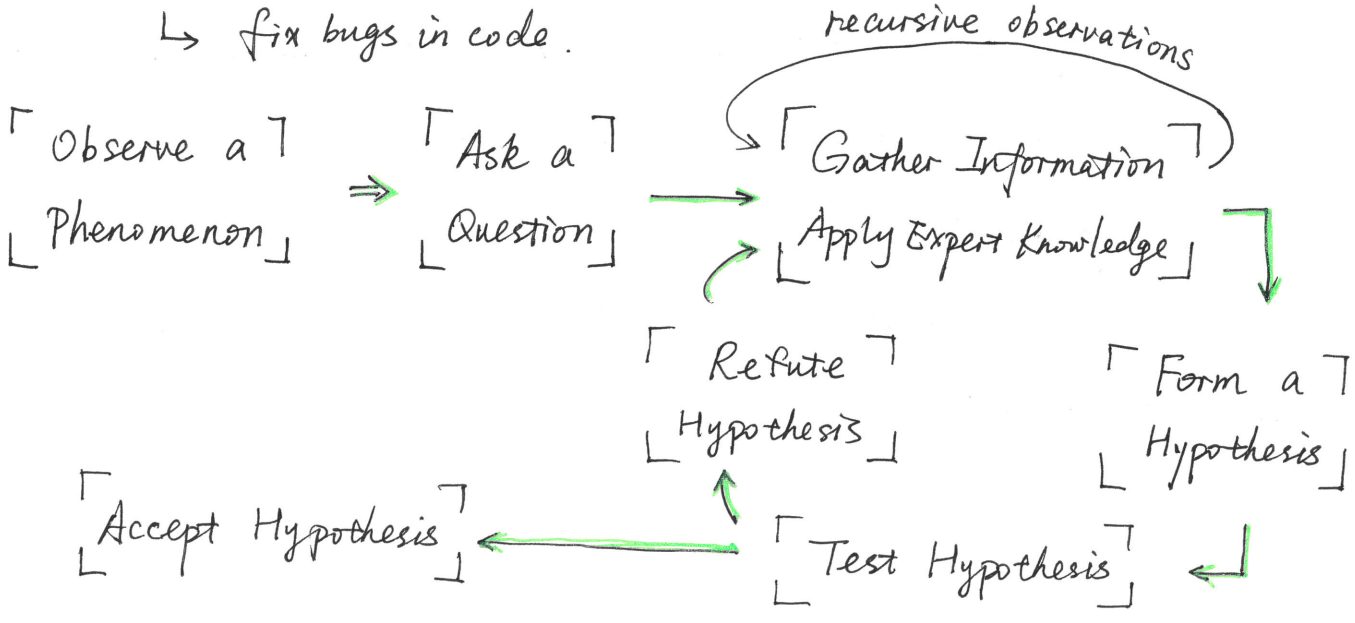
① Add an assert statement where you know an ~~so~~ invariant should be true: `assert(expr);`

② If true, then nothing happens;
If false, then prints a msg stating an assertion failed, and aborts the program. (fail fast)

③ Keep asserts active, ~~which~~ the slowdown is negligible.
To turn it off, pass the `-DNDEBUG` option to the compiler.

■ Debugging

↳ fix bugs in code.



- How to gather information.

  1. Insert print statement.

  2. Use a debugger, e.g., gdb

     ① Set breakpoints / watchpoints

     ↓ on a particular line ↓ when a particular "box" changes.

     ② Once stoped, examine the state of program by printing the values of expressions.

     ③ Continue executing either ~~#~~ until the debugger stop naturally, or one statement at a time.

- Testing hypothesis.

  * The formed hypothesis should be testable and actionable.

  1. Constructing test cases.

     e.g., it will ... with input that ...

  2. Inspecting the internal state of the program.

     e.g., if the hypothesis suggests sth. about the behavior before it reaches the point where we observe the symptom.

  3. Adding asserts.

     to check if we are violating an invariant of our algorithm.

  4. Code inspection.

- Before rejecting a hypothesis, consider there may be multiple errors in the code.

  ① Defer the investigation of the 1st error, and try to debug the second.

  ② Confirm the suspicion that the difference in behavior is in fact a symptom of a different problem.

Chapter 7 Recursion

■ Recursive function

1. Feature of recursive function:
   - ① have a base case (a condition in which it can give an answer without calling itself)
   - ② the recursive case always make progress towards the base case.
   - → It works by assuming that the recursive case works correctly, and using that fact to make the current case work correctly. (induction)
2. Theory : how to prove the recursion will always terminate ?
   - ① The parameters (arguments to be recursed) ~~is~~ have well-ordering.
   - ② every time we are recursing with values that are "less than" what was passed in.
3. Well-ordered set.
   - a total ordering where every subset of the set has a least element
   - ~~do~~ can be any types (number, data structures, etc).
   - may use ordinal numbers to number the elements.
     - → we can make a measure function which maps the parameter values to the naturals / ordinals.
4. Bad algorithms make recursion very slow by recomputing the same thing millions of times.

Solution: ① Rethink the algorithm.

② Memoization: keeping a table of values that already computed.

■ Tail Recursion.

- head recursion: perform computation after making a recursive call
- tail recursion: the only recursive call is the last thing before returning.
- ∈ tail call: a function call where the caller returns IMMEDIATELY AFTER the called function returns, without further computation.
- Advantage: ① the current frame can be reused.

  ② can be used (~~may~~ and must be) as an alternative for iteration in functional programming languages.

  (where a value cannot be modified once created)
- ☆ tail recursion ⟺ iteration:

  ① Normal initialization in func()

  ② Create a func_helper(), which takes ALL variables (related) as arguments (such that no value needs to be stored after calling)

  ③ The base case in func_helper() is 「! condition」 in the while loop.

  ④ The recursive case is the other code executed in the loop. call func_helper with updated arguments.

For example :

* Iteration :

```
func (...) {
    initialization ;
    while ( condition ) {
        Some statements ;
        x = updated_x ;
        ans = updated_ans;
    }
    return ans ;
}
```

* Tail recursion

```
func_helper ( x, ans ) {
    if ( ! condition )
        return ans ;
    Some statements ;
    return func_helper (
        updated_x, updated_ans);
}
func (...) {
    initialization ;
    return func_helper ( x, ans);
}
```

■ Mutual Recursion

↳ two or more functions which call each other.

(inside each function doesn't require recursion)

- How to make sure they will terminate ?

  → The mutaally recursive pair makes progress towards a base case.

- Declaration :

  → write the prototype for the second function before the first.

- Example :

  recursive descent parsing, where many functions (each parses a specific part of the input) mutually recurse to accomplish parsing jobs.

Chapter 8 Pointers

■ Memory

1. Addressing:

- byte-addressable memory: each different address refers to one byte of data, named by the hardware.
- each variable has a base address, which is one-byte.

  e.g., a 32-bit int occupies 4 bytes, i.e., 4 spaces of address.
- The size of pointers = the size of addresses (i.e., how many bits are used to name an address)

  e.g., a 32-bit machine has points with 4 bytes in size.

Example:

a 32-bit system:

	Address (← 4 byte →)	data (← 1 byte →)
x	107	0
	106	0
	105	0
	104	5
c1	103	83
c2	102	0
p	101	0
	100	0
=&x	99	0
	98	104

```
int x = 5;
char c1 = "S";
char c2 = "\0";
int *p = &x;
```

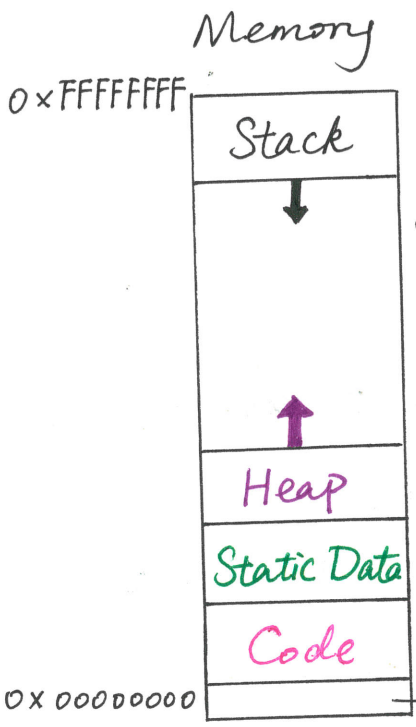
Note: sizeof(*x) = 4 bytes, ~~so it~~ = 1 byte per line × 4.

sizeof(p) = 4 bytes = 4 bytes for an address

The size of pointers has nothing to do with the type it points to.

2. How programs are stored in memory.

calling convention: the arguments to a function resides in the stack frame of the function that calls it.



local variables

- Divided into stack frames (each below the one that called it)
- Before creating frames for a called function, a return address will be stored.

dynamically allocated data

global variables, which are accessible until the entire program terminates

program instructions. typically 32 bit per instruction

3. Memory check tools, e.g., valgrind / -fsanitize=address.

■ Pointers Basics

1. Concept: a pointer is a variable whose value is an address of the variable it points to. It's a type constructor.
   (an arrow points to a box)

2. Declaration: ① the name of the pointer
   ② the type of variable it will be pointing to.

```
(point_to_which_type *) name_of_pointer; // here * is part of the type name
```

3. Assignment: using the "address-of" operator &

```
name_of_pointer = & lvalue ;
```

- & gives an arrow pointing at its operand (gets the address), & unitary
- (& lvalue) is not a lvalue, i.e., the location of a varible can be accessed, but cannot be changed.

- A pointer can only point to addressible data, i.e., lvalues.

4. Dereference: using the "desitination" operator *

```
* name_of_pointer = rvalue;
lvalue = * name_of_opointer;
```

- * dereferences a pointer, to read/write the value of the box it points to.
- (*pointer) can be both lvalue or rvalue.

■ Dangling pointer: a pointer to something whose memory has been deallocated.

■ Void pointer (void *)

1. void * is a special type for NULL (defined in stdio.h/stdlib.h), a pointer with numerical value of 0. where nothing will be placed.
2. A void pointer indicates a pointer to ANY type, and is compatible with any other pointer type.
3. It ~~nei~~ cannot be dereferenced, nor be done arithmetic on.
   - A Segmentation fault occurs when trying to dereference NULL. (or trying to access any "blank" region of memory)
4. Application:
   ① return NULL in the "there's no answer" case.
   ② return NULL if a function fails to create something as it's supposed to do
   ③ use NULL to indicate the end of a sequence.

■ Pointers to Sophisticated Types

1. Pointers to Pointers

- * and & are inverse operations.

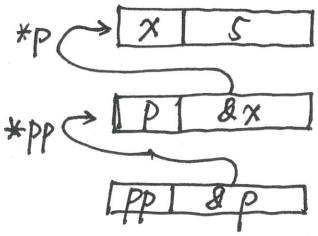
(int *) p ⇄ pp (int **)

+ &: take the address

+ *: access the value

variable	type	conversion
p = &x → x	int	*p = x
p → p	int*	
pp = &p → pp	int**	*pp = p

**pp = x



- *&e = &*e = e

- Possible errors related to assigning:

① Incompitable type. e.g., *pp = x;

② Left side is not a lvalue. e.g., &p = pp; // pp = &p is correct.

2. Pointers to Structs.

- The order of operations: * and .

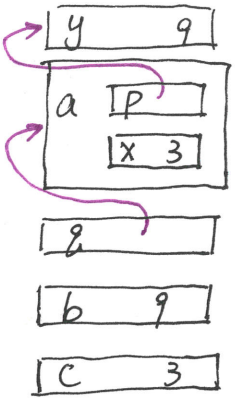
*a.b = *(a.b) // . has priority.

- The -> operator: ① dereferance a pointer to a struct ② select a field.

a -> b = (*a).b

which means: a is a (struct *)

b is a field of that struct.



e.g., int b = *a.p; //=*(a.p)=*q->p

int c = q->x; //=(*q).x = a.x

3. Const

↳ a qualifier (which modifies a type) that regulates the variable of a const type not allowed to change.

(1) Pointers to a pre-defined const type variable:

```
const int x = 3;
const int *p = &x;
```

where x cannot be modified neither by: `x = 4` (x) — error

nor by: `*p = 4` (x) — error

Otherwise we will get compiler error like this:

「assignment of read-only location '*p'」

Besides, such pointers MUST be defined as `const int *`,

If we write: `const int x = 3;`

`int *p = &x;` (x) - warning.

We will get compiler warning like this:

「initialization discards 'const' qualifier from pointer target type [enabled by default]」

because not specifing 'const' means p can be useed to modify or read a value.

(2) Pointers to a const type variable, but the variable is not declared as const:

```
int x = 3;
const int *p = &3;
```

where x can be modified by : ~~int~~ x = 4 ; (✓)

but cannot be modified by : *p = 4 ; (X) – error

because we are saying "x is variable that can be modified", and then "p is a pointer which we can only use to read the value it points at, not modify it" – this does not impose new, nor violate existing, restrictions on x.

(3) The pointer itself is const:

```
int * const p = &x ;
```

where *p (and x) can be changed : *p = 4 ; (✓)

However, p itself cannot be changed : p = &y ; (X) – error

In sum, only the data directly declared as const cannot be changed.

For example :

declaration	Can we change... **p	*p	p
int ** p	✓	✓	✓
const int **p	X	✓	✓
int * const *p	✓	X	✓
int ** const p	✓	✓	X
const int * const *p	X	X	✓
const int ** const p	X	✓	X
int * const * const p	✓	X	X
const int * const * const p	X	X	X

Chapter 9 Arrays

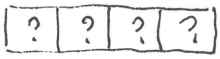
■ Basics

- An array is an indexable sequence of data of the same type.
- In C, arrays are stored in consecutive addresses in memory.
- The name of the array is a pointer to the first element of the array. NOT a lvalue.

■ Declaration.

1. No initialization :

```
int myArray[size];
```

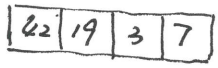


where size can be a constant, or a non-constant expression (after C99).

2. With initialization data :

① # (elements) = size :

```
int myArray[4] = {42, 19, 3, 7};
```



② # (elements) < size :

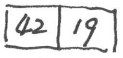
```
int myArray[4] = {42, 19};
```

③ # (elements) > size :

Receive ar warning.

④ Omit the array size :

```
int myArray[] = {42, 19};
```



3. Same Syntax to initialize Struct.

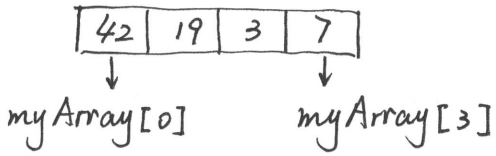
```
typedef struct _myStruct {
        int x;
        int y;
} mystruct;

myStruct p = {3, 4};    or    myStruct p = {.x=3, .y=4};
```

An array of structs are composable :

```
myStruct p[] = { {.x=3, .y=4}, {.x=5, .y=6} };
```

■ Accessing an Array : two equivalent ways.

① Indexing (zero-based)



indexing out-of-bound is an error that the compiler cannot detect.

② Pointer arithmetic :

```
*(myArray + i) == myArray[i]
```

i.e., `myArray + i == &myArray[i]`

■ Passing Arrays as Parameters

① Must explicitly pass the size of the array along with the array itself.

e.g., `int myFunc(int *myArray, size_t size) {...}`

or `int myFunc(int myArray[], size_t size) {...}`

② Make a struct which puts the array and its size together, then pass around.

- `size_t` : a type name for " unsigned integers that describe the size of things. It's recommended to use it for describing / iterating the index of an array.
- Be careful not to return an array created locally in the stack frame. i.e., a dangling pointer, to avoid unexpected behavior.

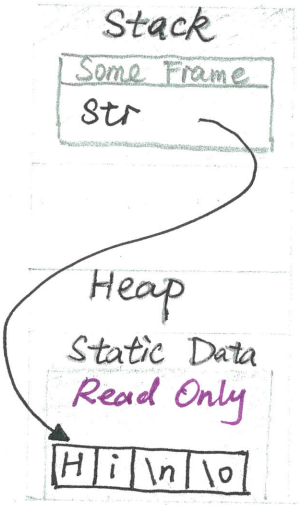
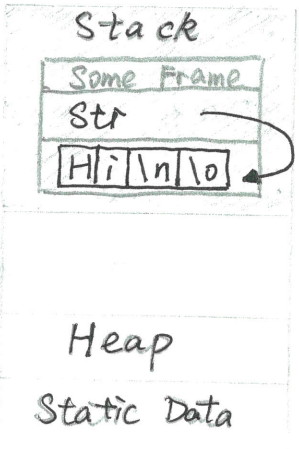
warning : function returns address of local variable [-Wreturn-local-addr]

Use heap (malloc) instead.

Chapter 10 Use of Pointers

■ String

I. String Literals v.s. Array of Chars

	String Literals	Array of chars. (string)
Definition	* a sequence of chars * terminated by '\0' (null terminator) * written down using " "	* a sequence of chars. * (if is a string, ended with '\0')
Declaration	`const char * str = "Hi\n";`	`char str[] = "Hi\n";`
Layout in Memory		
Can Modify?	`str = another_str;` ✓ `*str = 'a';` ✗ `str[i] = 'a';` ✗ (Segmentation fault)	`str = another_str;` ✓ `*str = 'a';` ✓ `str[i] = 'a';` ✓
printf ("%s") i.e., accessed as 'string'.	print out the literal as assigned.	print out the chars until encounter '\0' i.e., may access pass its bound if not null terminated.
Other Notes	• *str cannot be modified, thus cannot be the destination of strcpy, etc.	• Better to request extra space at initialization to hold more coming chars.

II. Literals

1. Definition : a sequence of chars written down as in quotation marks.

   The compiler will add '\0' for literals only.

2. Pointer type : const char *

   - If declared as : char * str = "Hi",

     then receive a warning with (-Wwrite-strings) flag.

     otherwise it's compiled successfully without notice by default.

   - If try to modify str:

     then the program will crash with a segmentation fault.

3. Storage of literals

   - ① The loader places the literal data into read-only portion of Static data section.

     (loader: the portion of the OS which reads the .exe from the disk and initializes system its memory appropriately)

     ② The loader marks that portions as non-writable in the page table.

     (page table: the structure that OS maintains to describe the program's memory to the hardware.

     ③ Attempting to modify read-only data will cause the hardware to trap into the OS, and then OS will kill it.

     (trap: transferring execution control from your program to the OS kernel)

   - Why the compiler puts literal into read-only portion ?

     ∵ The literals may be reused, and thus should be changed.

     e.g., the compiler is free to put the two identical literals in one location.

     i.e., str1 = "..."; str2 = "...";

     then 

     str1 = str2.

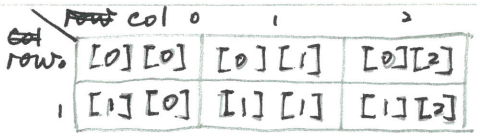
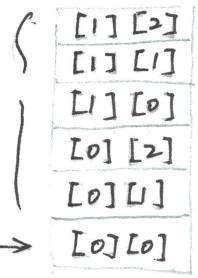
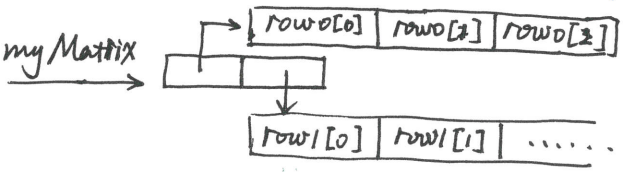
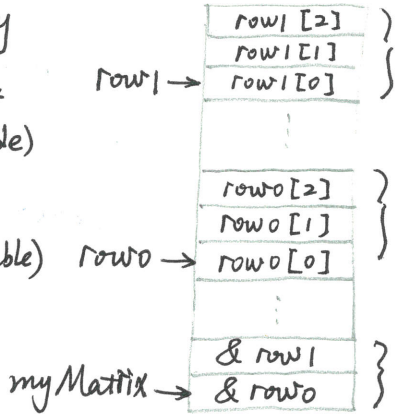
III. Principles when dealing with strings.

1. Strings are just arrays of chars. String variable is simply a pointer. Nothing special.
2. Use the knowledge of types / pointers / arrays to understand everything about strings.
3. The only difference is that there are many built-in functions for String operation based on the fact that strings end with '\0'.
4. Therefore, make sure your array of chars is correctedly ended with '\0' before using any string-related functions.

■ Multidimensional Arrays.

	Multidimensional Array	Array of Pointers (to Arrays)

Declaration	`double myMatrix[2][3];`
	~~col~~ row, ~~row~~ col
		`double row0[3];`
		`double row1[3]; // can be any size.`
		`double * myMatrix[2] = {row0, row1};`

Layout in Memory	 consecutive  Total memory = col * row * sizeof(double)	 Total memory = col * row * sizeof(double) + row * sizeof(* double) 
lvalue?	myMatrix is NOT an lvalue myMatrix[n] is NOT an lvalue	myMatrix is NOT an lvalue. myMatrix[n] is an lvalue

They're not the same types, cannot convert implicitly

1. A few more words about array of pointers of arrays:

① evaluating myMatrix[i] = read a value from memory,
instead of calculating an offset (like multidimension arrays do)
⇒ performance implications.

② Rows can have different size.

③ myMatrix[i] is an lvalue (a pointer), so we can change where the pointer points, or even have two rows pointing at the same array.

④ use "char [ ][n]" in function arguments.

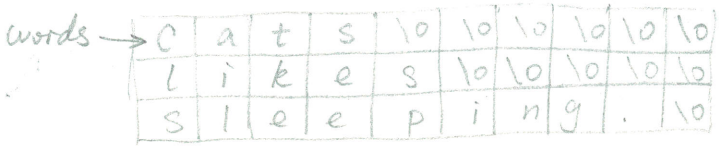
2. Array of Strings

① Cannot omit the size of the second dimension.
≈ an array must know the size of its elements. (type)

```
char strs [ ][3] = { "ab", "cd", "ef", "gh" };
```

② If declared as a multidimensional array of chars, the uninitialized slots will be filled by '\0':

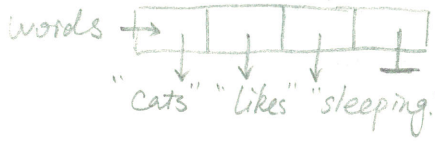
```
char words [3][10] = {"Cats", "Likes", "sleeping."}
```



Make sure there's enough space for '\0'.

③ Declared as an array of pointers to strings allows us to have items of different lengths:

```
const char * words [ ] = { "Cats", "Likes", "sleeping.", NULL }
```



It's common to end an array of strings with a NULL, such that loops can iterate the array without knowing the size in priori.

∴ When writing functions about strings, deal with the special case of NULL.

■ Function Pointers

1. Motivation :

Make a function pointer a parameter to a function we are writting, So that we can use the same framework or pattern to realize different functions.

2. Declaration :

- As an argument : `return_type (* func_name) (argument_type, ...)`

e.g.,
```
void doToAll ( int * data, int n, int (*f) (int) ) {
    for ( int i=0 ; i<n ; i++ ) {
        data [i] = f ( data[i] ) ;
    }
}
```

Technigally, the name of any function is a pointer to that function.

- use typedef to generalize function-pointers that have the same return type and arguments' types :

`typedef return_type ( * new_type_name ) ( argument_type, ... )`

e.g.,
```
typedef int ( * int_func_t ) ( int ) ;
void doToAll ( int * data, int n, int_func_t f ) {
    for ( int i=0 ; i<n ; i++ )
        data [i] = f ( data[i] ) ;
}
```

3. Use `void *` as argument's type in function pointers.
(a pointer to an unspefified type of data)

- motivation : allow a function to treat any type of data using the same algorithm.

- There's a danger any time you use void *

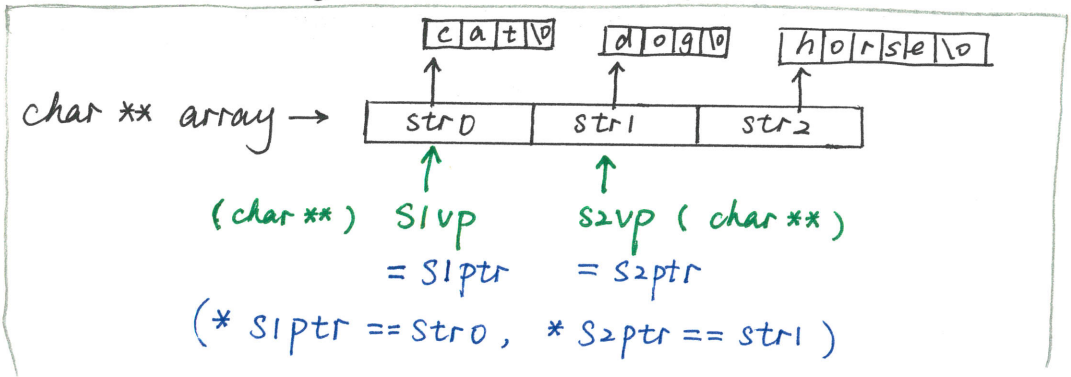
① Convert the parameters to the correct type.
especially deal with multiple-depth pointers.

② If (void * var) is an array, take the size of its elements as an argument as well. ~~Oth~~
Otherwise one can never iterate through the elements / do pointer arithmetic to an unspecified array type.

③ Usually make an wrapper function to provide a simpler interface.

- Example: qsort:

```
// C library has a sorting function. "base" is the array "size" is the size of each element in the array.
void qsort ( void * base, size_t nmemb, size_t size,
             int (* compare) ( const void *, const void * ) );

// Use qsort to sort an array of strings
int compareStrings ( const void * s1vp, const void * s2vp ) {
    const char * const * s1ptr = s1vp; // Void * → const char * const *
    const char * const * s2ptr = s2vp; // can neither change the char nor the *s1vp (str0)
    return strcmp ( * s1ptr, * s2ptr );
}
// wrapper function
void sortStringArray ( const char ** array, size_t nelements ) {
    qsort ( array, nelements, sizeof ( const char * ), compareStrings );
}
```



■ Security Hazards

↳ security vulnerabilities : an opportunity for a malicious user to abuse the software and compromise the functionality of the system in some way.

I. Buffer Overflow.

- bug : provide a possibility to write a string into an array which is too small for it.
- opportunity : overwrite the return address.
- method : ① craft the input string so that the new return address points to her code, ~~where~~ then
  ② (craft the input string so that) the program does whatever her instructions tell it to do

II. Unsanitized Input.

- fact : some programs use strings in a way that certain characters are special.

  e.g., ① printf – %
  ② command shell – ` (back-ticks)
  ③ SQL – ' (single quotes that end the literal)

- bug :

  ① `char * input = readAString();`
  `printf(input);` // correct : `printf("%s", input);`

  ② not carefully perform inputs sanitizing when reading a string from the users and include it in some context where characters have special meaning.

- method :

① format string attacks :

The attacker crafts an input which contains %-conversion, and let printf take the data where these arguments should be, format them as directed, and print them.

Even worse, use %n which writes the number of characters printed so far into a memory location specified by an (int *) passed as the appropiate argument, to modify the memory.

② Command shell :

write commands to execute within ` ` that gives them access to the system to gain / modify information.

③ SQL :

Type a ' and a ; to end the corrent command, followed by an an arbitrary command of his choosing, in :

SELECT * from Users WHERE name = ' strFromUser'

Make use of format string :

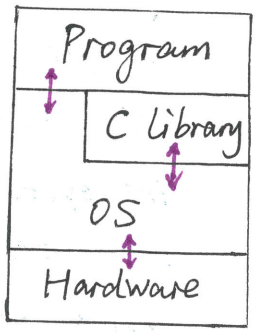
```
const char * fmt = "%d\n";
if ( printInHex)
        fmt = "%x\n";
printf ( fmt, Somenumber);
```

Chapter 11 Interacting With the User and System

■ The Operating System.

1. OS : low-level software responsible for managing all of the resources on the system for all programs.

2. System Call : transfers control from the program into the OS.
   - through which the program asks the OS to access hardware on its behalf.
   - (make system calls directly
     | make a ~~func~~ library call : e.g., printf used "write"



3. Global variable:
   - variables whose scope is the ~~a~~ entire program
   - declared outside of any function
   - ~~scope is~~ location is not part of any frame.

4. Errors from System Calls
   - `errno`: a golobal variable set by a failed system call.
     the value can be compared with the various constants defined in <errno.h>
   - `perror("")` : prints a descriptive error message based on current errno value + a user ~~inp~~ defined string.

Note: Not to call anything that might change errno before calling perror.
e.g., printf may change errno.

■ Passing Arguments.

```
int main ( int argc , char ** argv ) { }
```

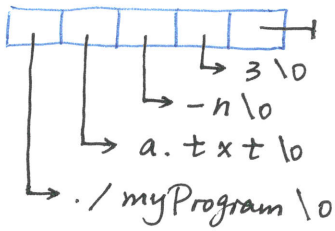
argc: how many arguments were passed in. each seperated by ␣

argv: array of strings

```
$ ./myProgram a.txt -n 3
```

argc = 4

argv →



Access Patterns to the Command Line Arguments:

1. Iterating over the elements of argv.
2. Complex option processing: used for taking flags & options.

e.g., gcc -o test. text.c
(-o: flag; test.: option.)

```
#include<unistd.h>
int getopt ( int argc, char * const argv[], const char * optstring );
```

- getopt: the option character (e.g., o)
- optstring: valid option characters. if followed by ":", then the argument is required; if "::", then optional.

```
int opterr ; // if set to 0, then no error msg is printed.
int optout ; // store the of an unknown option character if met.
int optind ; // the next argv[index] to be processed. Initially 1.
char * optarg ; // point to the value of the option argument.
```

e.g.,
```
while (( c = getout (argc, argv, "abc:")) != -1){
    // call getout succesively to read in all flags.
        switch (c) {
            case 'a':
                ...
            case 'b':
                ...
            case 'c':
                cvalue = optarg;
                break;
            case '?': // if not in the optstring, or missing option argument.
                if ( optout == 'c' )
                    fprintf( stderr, "option -%c requires an argument \n", optout);
                else if ( isprint (optout))
                    fprintf (stderr, "unknown option "-%c" \n", optopt );
                else
                    fprintf (stderr, "unknow option character '\x%x' \n.
                                                                optout );
                return 1;
            default: abort();
        }
}
```

- The environment pointer : char ** envp
  ↳ to an array of strings containing the value of variables : name=value.

  Functions : getenv, setenv, putenv, unsetenv.

■ Process Creation

I. The shell makes a call to create a new process (fork)
   ↓ make an identical copy of current process

II. The shell makes another system call (execve) to replace its running program with the requested program. (binary, argv, envp)

III. OS executes the execve system call.
   1. Destroy the currently running program.
   2. Load the specified executable binary into memory
   3. Set the ~~executable~~ execution arrow to a starting location.

IV. The start-up code calls main()
   ↳ in object.
   1. calls various functions which initialize the C library.
   2. Counts the elements of argv to compute argc.
   3. calls main

V. main() is executed and returns to the function that calls it (start-up)

VI. The start-up code performs any cleanup required by the C library, and calls exit. → quits the program, and passes returning status.

VII. The shell makes a system call which waits for its child process(es) to exit, and collects their return values.

■ Files : Open . Read . Write . Close

I. Openning a File

including stdin / stdout / stderr

↳ Creates a stream (File * , a sequence of data , which can be read or written) which has a current position which may advance upon an operation.

1. Function : fopen

```
File * fopen ( const char * file_name , const char * mode );
```

"w"

mode :

Mode	read and/or write	Not exist?	Truncate?	Position
r	r	fails	no	beginning
r+	rw	fails	no	beginning
w	w	created	yes	beginning
w+	rw	created	yes	beginning
a	w	created	no	end
a+	rw	created	no	end

⇓

even if the program moves the current position elsewhere

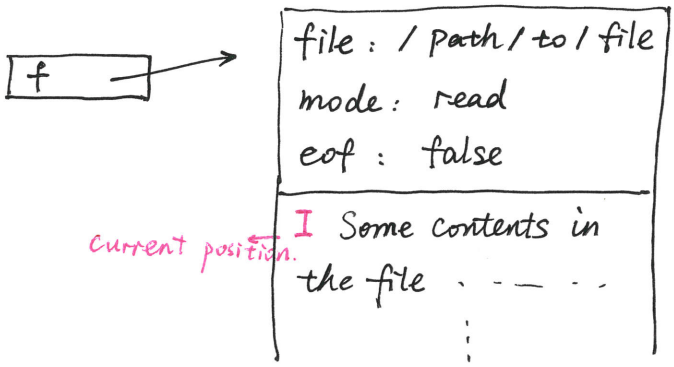
2. FILE Struct .

- contains a file descriptor instead of the file name

  ↓

  a numerical identifier returned to the program by the OS when making open system call.

- { the state of the file.
  data + position .



II. Reading a File

① fgetc : read one character at a time.

```
int fgetc ( FILE * stream);
```

- EOF (stdio.h) : ~~a~~ char that indicates the end of file, usually -1.
- fgetc( ) advances the current position in the stream.

Don't cast to char when judging EOF.

example :

```
int c ;
while ( (c= fgetc(f)) != EOF ){
        printf ("%c", c);
}
```

② fgets : read one line at a time.

include \n, the last digit is always \0.

```
char * fgets ( char * str, int size, FILE * stream)
```

↑ store to str[size]

- return str if it succeeds. In this case, str is \0- terminated.
- return NULL if :
  - (i) it encounts EOF before reading any data.
  - (ii) it encounters some error

use feof and/or ferror to distinguish the two cases.

example :

```
long total = 0 ;
char line [LINE_SIZE] ;
while ( fgets ( line, LINE_SIZE, f ) != NULL ) {
        if ( strchr ( line, '\n' ) == NULL ) {
                printf ("Line is too long! \n");
                return EXIT_FAILURE ;
        }
        total += atoi ( line ) ;
}
```

③ fread : read non-textual data, e.g., images, video, sound, or other file where data is in a binary format.

size_t fread (void * ptr, size_t size, size_t nitems, FILE * stream)

ptr: pointer to the data to write. size: size of each item. nitems: number of items

As with fgets(f...), employ feof / ferror to obtain more information if fread returns fewer items than you requested.

④ fgets (getline) + sscanf / fscanf : read in formatted input.

III. Writing Files

1. All functions write at the current position in the file, and advance it accordingly
2. Functions
   - ① fprintf : formatted print.
     e.g., fprintf (f, "%d\n", c);
     use it to print to the standard output : fprintf (stdout, ...)
     or ~~to~~ print error message : fprintf (stderr, ...)
   - ② fputc (c) : print a char
   - ③ fputs (*string) : print string + \n.
   - ④ fwrite : ~~prin~~ write non-textual data.

     size_t fwrite (const void * ptr, size_t size, size_t nitems, FILE * stream)

Note : errors may be detected ~~error~~ later (C function buffers the data)

IV. Closing Files

1. Function : fclose

   int fclose (FILE * stream);

   return 0 (success), return EOF & set errno if fails.

2. Closing the stream sends any buffered write data to the OS, then asks the OS to close the associated file descriptor.
   May not access the file once closed.

3. Fail to close :

   ① Reason : the most serious one ~~is~~ arises in circumstances where the data cannot actually to be written to the underlying hardware device.

   ② How to handle the error : CANNOT try again (i.e., use the stream).

   - In an interactive program : inform the user & take corrective action before proceeding.
     e.g., use fopen~~ing~~ / fwrit~~ing~~ / fclose~~ing~~ after the disk is freed up.

   ③ ALWAYS check the return status.

e.g.,
```
int main ( int argc, char ** argv ) {
    if ( argc != 2 ) {
        fprintf ( stderr, "  " );
        return EXIT_FAILURE;
    }
    FILE * f = fopen ( argv[1], "r" );
    if ( f == NULL ) {
        perror ( " " );
        return EXIT_FAILURE;
    }
        :
    if ( fclose ( f ) != 0 ) {
        perror ( " " );
        return EXIT_FAILURE;
    }
```

Other Interactions

- Make accesses appear to be similar to accessing files. "everything is a file"

1. Transfer data across the network.

   ① Make the socket system call to obtain a file descriptor for a socket
   (socket: logical abstraction over which data is transfered)

   ② Send data across the socket by using "write" system call,
   or read from the network by using "read"

2. Pipe

   cat text.txt | grep "name" | tail -1

   ↔ read / write. ↔

3. Device Special files.

   ↳ appears as file (e.g., can be seen by "ls"), but have a special file type indicating the OS should perform functionality, not read / write.
   usually in dev/ directory.

- Typically handled by system calls, but may not involve file descriptors.
  e.g., get time of day, fork, execve, etc.

- When the OS needs to ~~perform~~ inform the program of something asynchronously.
  — at any time during its operation. ⇒ signals.

Chapter 12 Dynamic Allocation

■ Dynamic Memory Allocation

- request a specific amount of memory to be allocated on the **heap**.
- not freed when function returns
- When an allocation request cannot be satisfied from the existing free block of memory, the allocation library request (by making system call) that the upper boundary of the heap be increased. i.e., causing the heap to grow.


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

■ void * malloc (size_t size)

- argument : how many bytes to allocate.

  ⇓

  use sizeof ( ) operator. can take either a type or an expression

  ↳ evaluated at compile time, not run time.

  example:

```
int * myArray = malloc ( 6* sizeof (*myArray) );
```

- return : a pointer to the allocated memory
  - if size = 0, then return either a NULL or a unique pointer value that can be successfully pass to free()
  - If malloc fails, then NULL is returned.
  - Check the return value before trying to use the pointer.
  - Making the caller handle the error is a good strategy.

- Mallocing complex structures:

  ① If you're writing a function returning a pointer of struct, then malloc the structure first (no matter the struct contains arrays or not).

③ If the struct contains arrays, then the arrays need to be malloc'd explicitely as well.

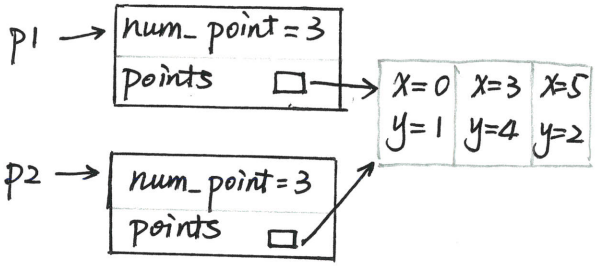
Example:

```
typedef struct {
        int x;
        int y;
} point_t;

typedef struct {
        int num_points;
        point_t * points;
} polygon_t;

// return a pointer to a struct.
polygon_t *  makeRetangle ( point_t c1, point_t c2 ) {
        // malloc the whole struct first
        polygon_t * answer = malloc ( sizeof (* answer));
        answer -> num_points = 4;
        // malloc the pointer in the struct as well.
        answer -> points = malloc ( answer -> num_points *
                                    sizeof ( * answer -> points ));
        /* Assignment */
        return answer;
}
```

- create a copy of struct allocated in heap

Shallow Copy

```
polygon_t *p2 = malloc();
*p2 = *p1;
```



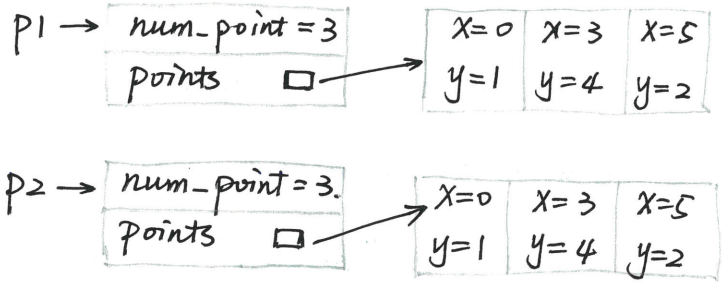
free():

If free(points) & free(p1),

p2->points will be a dangling pointer

Deep Copy

```
polygon_t *p2 = malloc();
p2->num_points = p1->num_points;
p2->points = malloc();
for (int i=0; i<p2->num_points; i++){
    p2->points[i] = p1->points[i];
}
```



If p1 is changed or freed, p2 is not affected.

■ void free(void *ptr);

- argument: the starting address of the memory returned by malloc.
  - ① If ptr == NULL: no operation is performed.
  - ② If ptr ≠ the start of a block in the heap: error, crash.
    - ∵ malloc allocates more memory than asked for, and keep a bit for itself at the start.
      When calling free(), the free() calculates the address of the metadata for the pointer.
  - ③ If ptr * is not on the heap: error, crash.
  - ④ If double freeing (free the same block more than once): crash, segfault.

- Dereference pointers that are freed (including free()) is an error.
- Each malloc should correspond to a free.

  ∴ if a memory block contains other pointers to other blocks on the heap, you should free the "inner" blocks first, then the "outer" blocks.

- Memory Leak.

  ↳ When you lose all reference to a block of memory, and the memory is still allocated.

  ① Not calling free() when the function is returned.

  ② Change the pointer to that block. e.g., reassign, move.

  Pay attention to "hidden" mallocs, e.g., getline().

■ void * realloc (void * ptr, size_t size)

↳ Resizes a malloced region of memory.

- argument:

① if (ptr == NULL), realloc = malloc

② if ptr = original memory allocated via malloc, then return a pointer to the new larger location on the heap.

- The new location in memory does not need to be near the origial location.
- A possible memory leak by writting:

```
p = realloc(p, 14 * sizeof(*p));
```

If realloc fails, p = NULL but does not free the original block.

■ ssize_t getline (char ** linep , size_t * sz , FILE * stream);

↳ reads a single line from stream (arbitrary length) , and store it in heap.

- ** linep :

① If (* linep == NULL),

getline( ) will first allocate (malloc) an implementation-specific-amont of memory , to store the characters.

If it's not long enough , getline( ) realloc repeatedly as needed.

② If (* linep → a malloced buffer) ,

getline( ) will start with * linep (size = * sz bytes) ;

If not enough , realloc ( ) as needed.

Note that the size of malloc / realloc is not necessarily equal to the number of chars stored.

- * sz :

Initially passed in as the size of malloced buffer.

Updated to current ~~size~~ bytes of ** linep allocated.

- returned ssize_t :

① The number of chars written (counting \n , not counting \0).
(bytes)

② If reaches EOF , or an error, then return -1.

e.g., File f =
Hello
Ellie

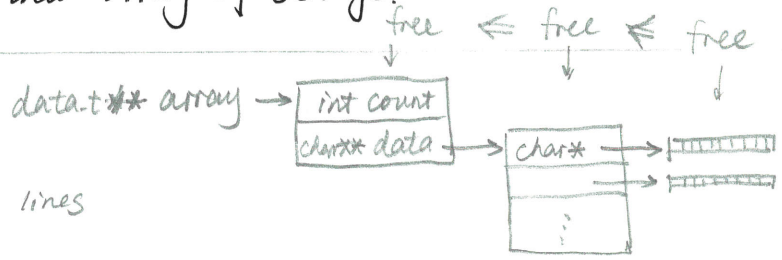
len=getline ( ) ⇒ * linep → | H | e | l | l | o | \n | \0 | ..... |

* sz ≥ 7

len = 6

One can write a loop where no buffer is provided the first time, and the same buffer is reused on subsequnt iterations.

Example: Store file contents into array of strings.




```
typedef struct {
        char ** data ;
        int count ;  // number of lines
} data_t ;

data_t * storeData ( FILE * f ) {
        data_t * array = malloc ( sizeof ( * array ) ) ;
        /* check failure. Default otherwise */
        array -> count = 0 ;
        array -> data = NULL ; // avoid using uninitialized data.
        char * line = NULL ;
        size_t sz = 0 ;
        while ( getline ( & line , &sz, f ) >= 0 ) {
                array -> data = realloc ( array -> data , (array -> count + 1) *
                                        sizeof ( * array -> data) ) ;
                array -> data [ array -> count ] = line ;
            // line = NULL ;
                array -> count ++ ;
        }
        free ( line ) ; // Don't forget.
        return array ;
}
void freeArray ( data_t * array) {
        for ( int i=0 ; i< array -> count ; i++ ) {
                free ( array -> data [i] ) ;
        }
        free ( array -> data ) ;
        free ( array ) ;
}
```

Example2 : Find the longest line in a file.

```
int main ( ~~or~~ int argc , char ** argv ) {
        if ( argc != 2 ) {
                fprintf ( stderr, " " );
                return EXIT_FAILURE;
        }
        FILE * f = fopen ( argv[1], "r" );
        if ( f== NULL ) {
                perror ( " " );
                return EXIT_FAILURE;
        }
        char * longest = NULL;
        size_t longest_length = 0;
        char * ~~curren~~line = NULL;
        size_t sz = 0;
        while ( getline ( & line, & sz , f ) >= 0 ) {
                size_t current_length = strlen ( line );
                if ( current_Length > longest_length ) {
                        free ( longest );
                        longest = strdup ( line );
                        longest_length = current_length;
                }
        }
        if ( fclose (f) != 0 ) {
                perror ( " " );
                return EXIT_FAILURE;
        }
        if ( longest != NULL ) {
                printf ( "%s", longest );
                free ( longest );
        }
        free ( line );
        return EXIT_SUCCESS;
}.
```

Chapter 13 Programming in the Large

Analogy : program ⟷ function
= =
document paragraph

① Higher-level design of the program.

② Clearly defined interfaces between multiple appropriately sized modules.

■ Abstraction

↳ the seperation of interface from implementation.
what it does how it does

1. Abstracting code out into function as one logical unit.
2. Size / Complexity of a function : <= 7 logical tasks.
3. Avoid duplication of code : hard to maintain / fix.
4. Designing software with hierarchical abstractingon:

① bottom-up : start with the smallest building blocks first, and build successively larger components from them.

pros : good for incremental testing
cons : have to build the right blocks

② top-down : start with high-level algorithm design, and design supporting functions, until reaching small enough functions that you do not need to abstract out any more pieces.

pros : you know exactly what pieces you need at every step.
cons : testing :

(i) building & testing in a bottom-up fashion.

(ii) writing "test stubs" : simple implementations of the smaller pieces which do not actually work in general, but exhibit the right behaviors.

■ Readability.

1. Function Size

① fit comfortably into one screen.

② 7 logical units.

2. Naming

The length of a variable's name should be proportional to the size of its scope, and the complexity of its use.

① naming conventions

e.g., ▲ add _t on the type name.

▲ constants in all capitals.

▲ prefix indicating types.

② glue together multiple words.

e.g., num_letters_skipped, numLettersSkipped (inner caps / camel case)

3. Formatting.

① indent to reflect the block struction.

② place a newline at the end of a statement.

③ placement of curly braces : "Java" style. vs., ITBS, Allman Style, GNUS.

4. Commenting / Documentation

① document large scale design

↳ describe how the modules fit together, the interface between modules.

② describle each component.

↳ interface, parameters, return values, side effect, etc

③ do not comment the obvious.

④ explain the unexpected.

■ Working in Teams

1. Use of a good revision control system. (e.g., Git, Subversion, Mercurial)
2. intergration
3. pair programming : driving ↔ navigating.

■ Even larger programs

- Write small piece of code → test them → next piece ...
  NOT the big bang approach.
- Discipline