1. History & Introduction

- Dennis Ritchie C99.
- 注释
  //
  /* 分行 */

static type system.

identifier (function name)

```
int my_add (int a, int b){
    return a+b;
}
my_add { 1, 2}
```

top level expression

arguments

- infix notation 1+3.
- 没有 expotiation operator (ep. x^n)
- rounds towards 0 (向下取整)
- trace_X (exp) 路径追踪

int. long. bool. char. double string. ptr symbol

- entry point : main (return 0)

- Bool F : 0 T : 1
- Character char %c
- float float %f
- double (比 float 更精确)

a. 97

("%c", a) ≠ ("%d", a)

- Control flow
  1. compound ({...})
  2. function call
  3. conditionals (if)
  4. iteration (loop)

* return statement ends the function

- "=" & "=="

"=" 赋值 initialization

"!=" / "==" 判断是否相等 (equality operator)

x=y ≠ y=x

```
int n = 5;              // initialization syntax
n = 6;                  // assignment operator
```

Example 1.5.5: dangling else

```
// this program demonstrates the dangers of a "dangling else"

#include "cs136.h"

// win_or_lose(y) ... what DOES this function do ?!?
void win_or_lose(int y) {
  printf("win_or_lose(%d):\n", y);
  if (y > 0)
    if (y != 7){
      printf("you lose\n");
  else
    printf("you win!\n");} // when does this print?
}

int main(void) {
  win_or_lose(-1); // no output
  win_or_lose(1);
  win_or_lose(7);
}
```

Code Output

```
win_or_lose(-1):
win_or_lose(1):
you lose
win_or_lose(7):
you win!
main.c:11:3: warning: add explicit braces to avoid dangling else [-Wdangling-else]
  else
  ^
1 warning generated.
```

不输出. 因为默认{}

2. Imperative C

- functional programming paradigm
  - ① "mathematical" (= "pure") — only return values; answer depend on argument values
  - ② only constants
- imperative programming paradigm ("impure")
  - ① function 改变 return values
  - ② 使用 variables & constant
  - ③ {...} compound statement
  - ④ side effects

- 3 types of statement
  - compound ~ ({...}) 按顺序执行
  - expression ~ 目的: generate side effects (值被丢掉)
  - control flow ~ (return. if. else) 控制执行顺序.

not all expression statement generate side effect

- side effects

If num passed to function. It output a message and returns num.

若 function 除了输出 value 时, 还产生 output. 则 function 有 side effect.

* assert & trace_int 改变 side effects
* function may only have side-effects
* 所有有 side-effect in function 为 impure function

① O (produce output)

printf(" ").

format specifier. → 整数 接行 %d \n %%

```
printf("2 plus 2 is: %d\n", 2 + 2);
```

```
printf("five\n")
```

return value 为 5. (字符数)
output 为 "five"

* %04d 一共四个数字. 数字用0补齐

ep. 0042.

② I (read input)

int n = read_int();

③ Mutation (change the value of variables)

- mutate global variable

改变 variable int 值 ep. 不断定义 n = ...

(Constant is immutable)

A global mutable variable is "impure" even if it has no side effects
ep n+=1

- mutate a variable through pointer parameter

✳✳ mutating a local variable/parameter don't have side effects
Since it does not affect state outside of the function

- immutable → constant. (const)
- mutable → variable
  - global scope 为全部
  - local "block scope": 从 initialize 到 函数结束

↑ (identifier 前不带 const 都是 variable)

Testing Terminology

Now that we've learned more about testing, we can summarize testing terminology:

Assertion testing: using assertions to test our code — test return value
I/O-driven testing: uses input and expected output — test output
Testing harness: using input to call functions

White box testing: you can see the code being tested
Black box testing: you can not see the code being tested
Unit testing: testing one piece at a time

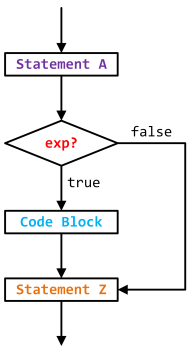
Sometimes the changes fix the original bug but introduce other bugs, breaking code that previously worked. In this case the code is said to have "regressed" (gotten worse).
Regression testing reruns all tests to check if everything still works (after a change to any part of the code) to ensure that changes made don't introduce new bugs.

3. Memory and Control Flow

- Control flow. program 如何执行.

types
- compound statement (block)
- function call
- conditional (if / else)
- iteration (loops) → 需要 mutation (side-effect) 使循环停止

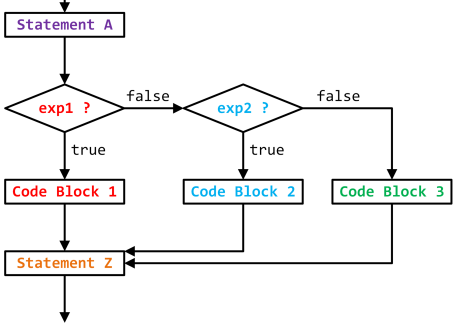
if




```
Statement A;
if (exp) {
  Code Block;
}
Statement Z;
```

exp 相互无关

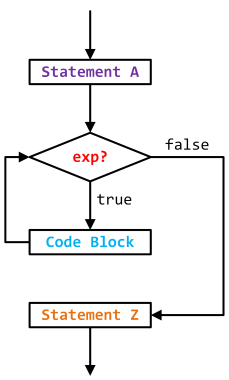
else if




```
Statement A;
if (exp1) {
  Code Block 1;
} else if (exp2) {
  Code Block 2;
} else {
  Code Block 3;
}
Statement Z;
```

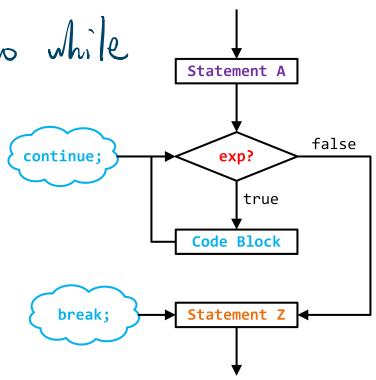
exp 相互联

while



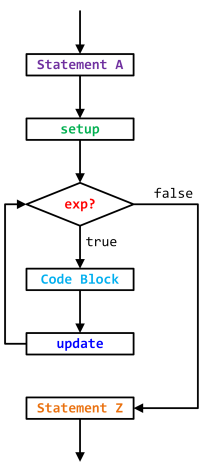

```
Statement A;
while (exp) {
  Code Block;
}
Statement Z;
```

do while




```
Statement A;
while (exp) {
  Code Block;
}
Statement Z;
```

for




```
Statement A;
for (setup; exp; update) {
  Code Block;
}
Statement Z;
```

simple recursion

```c
int recursive_sum(int k) {
  if (k <= 0) {
    return 0;
  }
  return k + recursive_sum(k - 1);
}
```

while loop

```c
int iterative_sum(int k) {
  int s = 0;
  while (k > 0) {
    s += k;
    --k;
  }
  return s;
}
```

accumulative recursion

```c
int accsum(int k, int acc) {
  if (k <= 0) {
    return acc;
  }
  return accsum(k - 1, k + acc);
}
int recursive_sum(int k){
  return accsum(k, 0);
}
```

while loop

```c
int iterative_sum(int k) {
  int acc = 0;
  while (k > 0) {
    acc += k;
    --k;
  }
  return acc;
}
```

do while

while

```c
int main(void) {
  int i = 0;

  printf("Do..while loop:\n");
  // do...while
  i = 5;                        // setup
  do {
    printf("%d\n", i);          // statement
    --i;                        // update
  } while (i >= 0);             // expression

  printf("while loop:\n");
  // while
  i = 5;                        // setup
  while (i >= 0) {              // expression
    printf("%d\n", i);          // statement
    --i;                        // update
  }
}
```

在最开始就不满足 while 后条件 in 情况下，
do…while will do 1 more iteration than while

while loop

for loop

```c
int main(void) {
  int i = 0;

  printf("While loop:\n");
  i = 5;                          // setup
  while (i >= 0) {                // expression
    printf("%d\n", i);
    --i;                          // update
  }

  printf("For loop:\n");
  for (i = 5; i >= 0; --i) {
    printf("%d\n", i);
  }
}
```

返回

5
4
3
2
1
0

set up statement.

```
while (expression) {
  body;
  update (缩写)
}
```

```
for ( setup ; expression ; update ) {
  body;
}
```

recursion 在 function 中不断调用自己

iteration 用 loops.

efficiency 相同

- Memory.

one bit of shortage has 2 states : 0 1.

byte : smallest accessible unit of memory.

address: position of memory.

$2^{bytes} = bits$

identifier	type	# bytes	starting address
n	int	4	0x5000

variable definition 中作用 : ① find space to store variable

② track address

③ 将值存进 address

size operator : (sizeof) 是 operator.

int. : 4 bytes (32 bits) 2^{32} possible values can represents

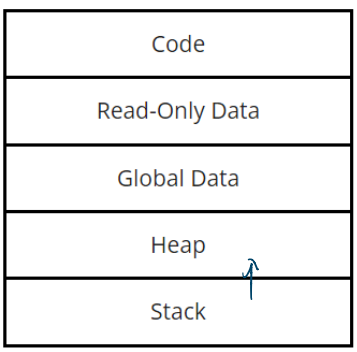
If you have a machine that uses 32-bit addresses, what is the maximum amount of RAM that can be addressed?

4GB (4,294,967,296 bytes, i.e. 2^{32})

$2^{32} \rightarrow 2^{16} \rightarrow 2^{8} \rightarrow 2^{4}$

overflow 溢出 : 可生成任何值

- Sections of memory.



← the call stack is stored

- Structure struct

```c

struct posn {       // name of the structure
  int x;            // type and field names
  int y;
};                  // don't forget this ;

// posn_equal(a, b) returns true the structs a and b are equal and
//   false otherwise
bool posn_equal (struct posn a, struct posn b) {
  return (a.x == b.x) && (a.y == b.y);
}

int main(void) {
  struct posn p = {1, 2};

  p.x = 5;                  // VALID MUTATION
  p.y = 6;                  // VALID MUTATION

  // alternatively:
  struct posn new_p ={5, 6};
  p = new_p;                // VALID MUTATION

  trace_int(p.x);
  trace_int(p.y);

  if (posn_equal (p, new_p)) {
    printf("The structures p and new_p are equal!\n");
  }

  printf("The value of p is (%d, %d)\n", p.x, p.y);
}
```

== & !=

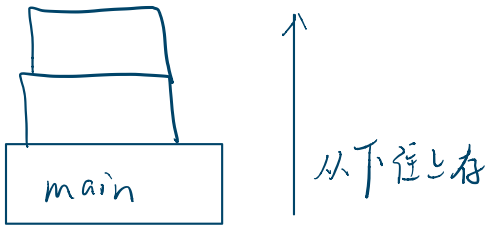
不适用于 structure

Code Output

```
The structures p and new_p are equal!
The value of p is (5, 6)
>>> [main.c|main|26] >> p.x => 5
>>> [main.c|main|27] >> p.y => 6
```

class exercise 3-1

- Stacks & recursion



stack frame { argument value / local variable / return address

before function can be called.

all arguments must be values

return address : 调用此函数前程序的位置.

main 的 return address : OS (operating system)

```c
int pow4(int j) {
    printf("inside pow4\n");
    int k = j * j;
    // Snapshot
    return k * k;
}

int main(void) {
    printf("inside main\n");
    int i = 1;
    printf("%d\n", pow4(i + i));
}
```

```
=====================
pow4:
  j: 2
  k: 4
  return address: main:11
=====================
main:
  i: 1
  return address: OS
=====================
```

调用pow4 func：行数

global variable 在执行前保留

local variable 在调用函数时保留

- recursion

```c
int sum_first(int n) {
  if(n == 0) {   // Snapshot here
    return 0;
  } else {
    return n + sum_first(n - 1);
  }
}

int main(void) {
  int a = sum_first(2);
  //...
}
```

```
=========================
sum_first:
    n: 0
    return address: sum_first:5
=========================
sum_first:
    n: 1
    return address: sum_first:5
=========================
sum_first:
    n: 2
    return address: main:10
=========================
main:
    a: ???
    return address: OS
=========================
```

第10行

未知

4. Pointer

- pointer : variable that stores a memory address %p

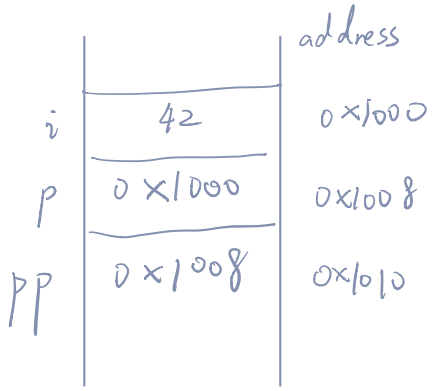
* : 定义 pointer.

& : 赋值 address operator produce location of an identifier

```
int *p = &i
```

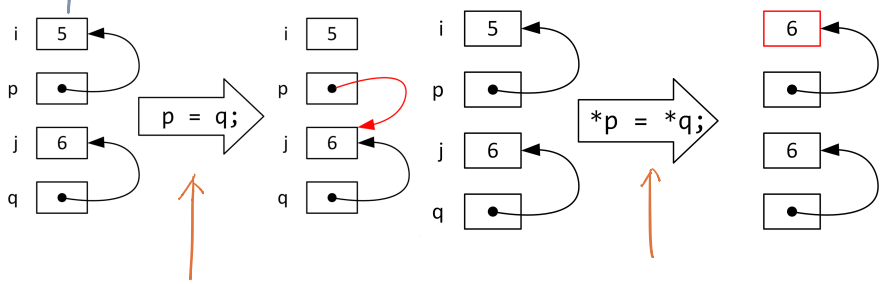
pointer ∈ variable → store value (address)

size of pointer. 64 bits (8 bytes)

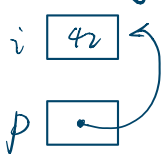



```
printf ("%p", &p)
printf ("%d", *p)
int **pp = &p.
```

pointer to pointer

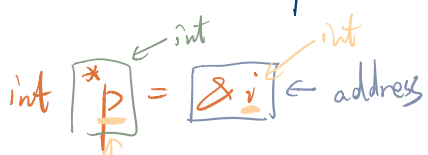


- memory diagram



int i = 42

int *p = &i;



p: pointer to an integer i = i in address

*p: p 里所存的值 = i in 值

p 与 &i return 相同

p 指向 q 指向 in 址

p in 值改变 (p=q)

p 指向 q 指向 in 值

p 指向 in 值改变.

(value = q 指向 in)

indirection operator

- 在 Structure 中出现 pointer.

(.) 优先级高于 (*)

ptr → field ≡ (*ptr).field

indirection selection operator

```c
struct posn {
  int x;
  int y;
};

int main(void) {
  struct posn my_posn = {0, 0};
  struct posn *ptr = &my_posn;

  *ptr.x = 3;                  // awkward
  ptr->y = 4;     "->相当于"*."  // much better

  trace_int(ptr->x);
  trace_int(ptr->y);
}
```

- Pointer assignment 指针赋值

aliasing : 多个指针访问同一地址

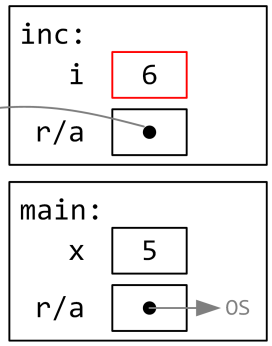
```
i = 1;
int *p1 = &i;
int *p2 = p1;
int **p3 = &p1;
```

*p3 p2 p1 都访问 i 的值

- Mutation

```
void inc(int i) {
  ++i;
}

int main(void) {
  int x = 5;
  inc(x);
}
```



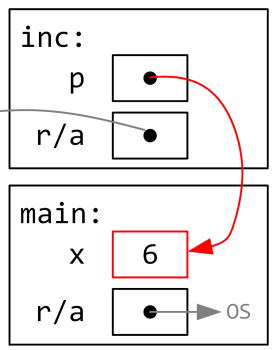
使用 pointer 不会破坏原始值

x 永远 = 5.

```
void inc(int *p) {
  *p += 1;
}

int main(void) {
  int x = 5;
  inc(&x);
}
```

x=5
x=6
x=7
⋮



incl(&x) 相当于 地址所指 i 的值

side effect → mutate a variable through a pointer parameter

Example 4.2.3: mutation side effects

```
1  // This program illustrates a "swap" function in C
2  // that uses pointers
3
4  #include "cs136.h"
5
6  // swap(px, py) "swaps" the contents of *px and *py
7  // effects: modifies *px and *py
8  // requires: px and py are valid pointers
9  void swap(int *px, int *py) {
10   assert(px);
11   assert(py);
12   int temp = *px;
13   *px = *py;
14   *py = temp;
15 }
16
17
18 int main(void) {
19   int a = 3;
20   int b = 4;
21   trace_int(a);
22   trace_int(b);
23   swap(&a, &b);          // Note the &
24   trace_int(a);
25   trace_int(b);
26 }
```

检测 *px 与 *py 中存的值不是 NULL

↑

给 swap value

Code Output

```
>>> [main.c|main|21] >> a => 3
>>> [main.c|main|22] >> b => 4
>>> [main.c|main|24] >> a => 4
>>> [main.c|main|25] >> b => 3
```

function 中不能 return pointer. (如果一开始没有这个 rule)

```c
// This program demonstrates returning a pointer (syntax) and
//   returning a pointer to a stack frame

#include "cs136.h"

// bad_idea(n) returns a pointer to the bad_idea frame (BAD!)
int *bad_idea(int n) {
  return &n;            // NEVER do this
}

// ptr_to_max(a, b) returns either a or b: whichever points to
//   the larger value
// requires: a, b are valid pointers
int *ptr_to_max(int* a, int* b) {
  assert(a);
  assert(b);
  if (*a >= *b) {
    return a;
  }
  return b;
}

int main(void) {
  int x = 3;
  int y = 4;

  int *p = ptr_to_max(&x, &y);        // note the &
  assert(p == &y);
  printf("%p\n", bad_idea(13));
}
```

Code Output

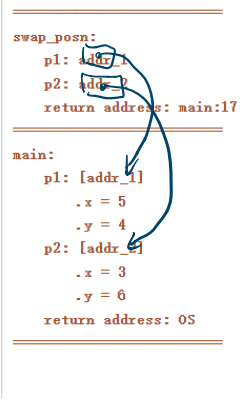
```
0x7f05ca100060
main.c:8:11: warning: address of stack memory associated with parameter 'n' returned [-Wreturn-stack-address]
  return &n;            // NEVER do this
         ^
1 warning generated.
```

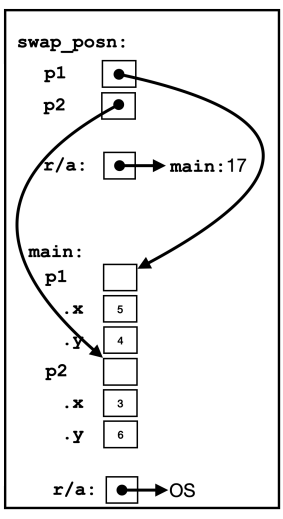
stack frames with pointers

```c
void swap_int(int *i, int *j) {
  int temp = *i;
  *i = *j;
  *j = temp;
}

void swap_posn(struct posn *p1,
               struct posn *p2) {
  swap_int(&(p1->x), &(p2->x));
  // Snapshot
  swap_int(&(p1->y), &(p2->y));
}

int main(void) {
  struct posn p1 = {3, 4};
  struct posn p2 = {5, 6};
  swap_posn(&p1, &p2);
}
```





STACK

- Input & Pointer

scanf("%d", &i)

↳ return value : int.

成功读到值或 EOF (end of file)

side effect of scanf:
① a value is read from input
② i is mutable
③ ... is mutated

4.3.1

```c
// read_sum() reads ints from input (until failure)
//   and returns their sum
// effects: reads input
int read_sum(void) {
  int sum = 0;
  int n = 0;
  while (scanf("%d", &n) == 1) {
    sum += n;
  }
  return sum;
}

int main(void) {
  printf("%d\n", read_sum());
}
```

用于判断是否还有值

4.3.2

```c
int readint(void) {
  int i = 0;
  int result = scanf("%d", &i);
  if (result == 1) {
    return i;
  }
  return READINT_FAIL;
}
```

读取值 in type

将读取的值存入 &i 中.

```c
int count_char(void) {
  int count = 0;
  char c = '\0';
  while (scanf(" %c", &c) == 1) {
    count++;
  }
  return count;
}
```

"%c" 不忽略空格

" %c" 忽略空格

4.3.3

- Structure & pointer

原因：① avoid copying the structure
② mutate the contents of the structure

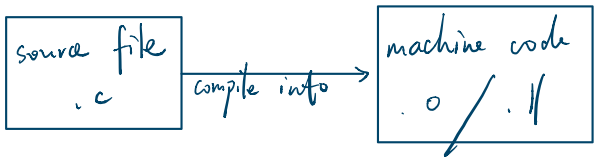
`int *p;`	`p` can point at any mutable integer, you can modify the int (via `*p`)
`const int *p;`	`p` can point at any integer, you can NOT modify the integer (via `*p`)
`int * const p = &i;`	`p` always points at the integer `i`, `i` must be mutable and can be modified (via `*p`)
`const int * const p = &i;`	`p` always points at the integer `i`, you can not modify `i` (via `*p`)

- function pointers

```
5  // io_apply(f) reads in each int [n] from input
6  //   and prints out f(n)
7  // requires: f is a valid pointer
8  // effects: produces output
9  //          reads input
10 void io_apply(int (*f)(int)) {
11   assert(f);
12   int n = 0;
13   while (scanf("%d", &n) == 1) {
14     printf("%d\n", f(n));
15   }
16 }
17
18 // sqr(i) calculates i^2
19 int sqr(int i) {
20   return i * i;
21 }
22
23 int main(void) {
24   io_apply(sqr);
25 }
```

return-type (*function-name)(parameter-type...))

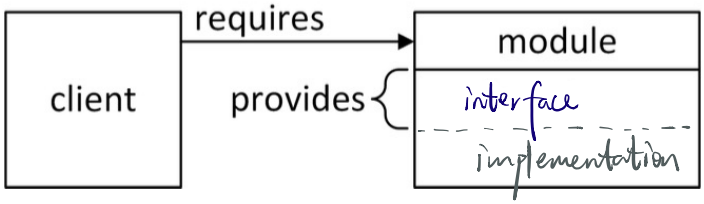
5. Modularization



- def. modularization

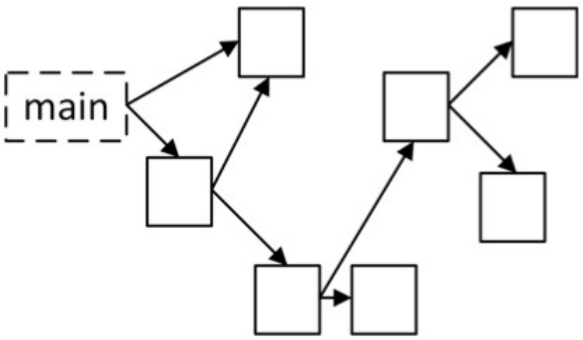
divide programs into well defined modules

(main.c)
→ "client" requires the functions that a (fun.c) module provides.
↳ ≈ libraries



functions in same module should share same purpose

→ module dependency graph. * no cycles. * have "roots" (main)



properties
- reusability 可以时用于…
- abstraction 简略 · 简洁
- readability 文档齐全
- maintainability 可维护

Concept Check 5.1.4

2.0/2.0 points (graded)

Suppose we have developed a module containing functions that model sunlight reflecting off of moving water (like a river or stream). For each term choose the number for the statement below that best illustrates an example of this term.

1. A client can use the module without understanding the physics used in the calculations
2. The module is well documented readability
3. The module is used in both video games and animated movies
4. Some functions in the module can be updated to execute more efficiently without changing the interface

Re-usability

3 ✔

Abstraction

1 ✔

outside

Readability

2 ✔

Maintainability

4 ✔

Show answer

- definition includes declaration

int foo(int x); → declaration

int foo(int x){
...
} → definition

```c
// Declaration (odd.h):
bool is_odd(int number);

// Definition (odd.c):
bool is_odd(int number) {
  return (number % 2 != 0);
}
```

```c
// this program demonstrates VARIABLE declarations

#include <stdio.h>
#include "cs136-trace.h"

extern int g;               // variable DECLARATION

int main(void) {
  printf("%d\n", g);        // this is now ok
  trace_int(g);
}

int g = 7;                  // variable DEFINITION
```

main.c module.c - Add File

```c
// main.c
// This program demonstrates PROGRAM scope
//   with a variable declaration

#include <stdio.h>
#include "cs136-trace.h"

// NEW declaration
extern int g;

// g IN scope

int main(void) {
  printf("%d\n", g);        // ok
}
```

variable declaration

可跨文件

在一个 directory 中

- type of scope
  - local 仅 function 中
  - global
    - program scope: "accessible to every file" if they are declared → 所有 C 中 in function
    - module scope: "static" 仅在此文件中被 call

- module interface (placed in ...h) header files

def. list of function that module provides

- 与 implementation (...c) 分开
- provided to clients

hidden from client

- information hiding.
- hide implementation details

(provide machine code ...ll instead ...c)

- hidding data. { security / flexibility

包含
1. overall description
2. function declaration
3. documentation (ep. purpose)

..., C won't compile if u don't include ...h

C isn't good at hiding data

☆ 若 b.c 中 include a.h.
则 b.h 中要写 #include "a.h"

- #include

a preprocessor directive temporarily "modifies" file before it is run
在run之前对前预处理

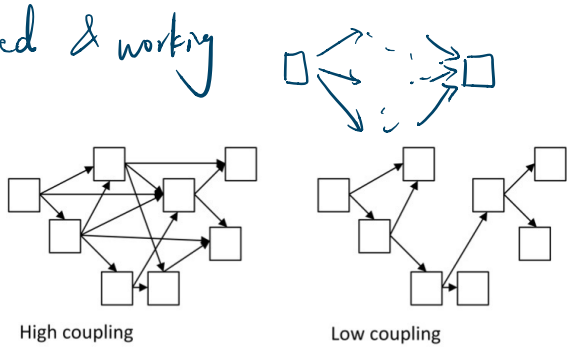
< > specify module which is one of the standard modules (C provide) ep. libraries
" " "regular modules" (自己编写的)

<assert.h>	provides the function assert
<limits.h>	provides the constants INT_MAX and INT_MIN
<stdbool.h>	provides the bool data type and the constants true and false
<stdlib.h>	provides the constant NULL and abs()

- cohesion & coupling

high cohesion : all interface functions are related & working towards a common goal

low coupling : little interaction between modules



- opaque structure : module only provides incomplete declaration in interface

incomplete declaration : only pointers to structure can be defined

* supply a complete definition of structure in .c file

- transparent structure : client can access and use without pointer.

put complete definition of structure in .h file

- ADT (abstract data type) 数据类型

def. mathematical model for storing and accessing data through operations.
implemented as data storage modules.
implemented with data structure

Collection ADT: an ADT designed to store an arbitrary number of items.

- 数据结构 data structure
  - dictionary ADT (pairs of keys & values)
    - lookup
    - insert
    - remove
  - stack ADT
    - push
    - pop
    - top
    - is_empty
  - queue ADT
    - add_back
    - remove_front
    - front
    - is_empty
  - sequence ADT
    - item_at
    - insert_at
    - remove_at
    - length.
  - set ADT
    - add
    - remove
    - length
    - member

⊗ class exercise

6. Arrays

6.2.1 ?

- Compound data storage : structure & array.
- def. can create array of any built-in data type in C
  a data structure that contain fixed number of element all same type

- array initialization
  uninitialized global array are zero-filled.
  uninitialized local array are filled with arbitrary values from stack

- length a[n] = {...}
  size bytes num = # element × # size
  n.

```c
// This program demonstrates how the value of an array
// is the same as the address of the first element

#include "cs136-trace.h"

int main(void) {
  int a[6] = {4, 8, 15, 16, 23, 42};

  trace_ptr(a);
  trace_ptr(&a);
  trace_ptr(&a[0]);
  trace_ptr(&a[1]);

  trace_int(a[0]);
  trace_int(*a);
}
```

array doesn't have values

Code Output

```
>>> [main.c|main|9] >> a => 0x7f691b200020
>>> [main.c|main|10] >> &a => 0x7f691b200020
>>> [main.c|main|11] >> &a[0] => 0x7f691b200020
>>> [main.c|main|12] >> &a[1] => 0x7f691b200024
>>> [main.c|main|14] >> a[0] => 4
>>> [main.c|main|15] >> *a => 4
```

- array & pointers

- pointer arithmetic : $p \pm i = p \pm i \times \text{size of } (*p)$

- 6.2.1
- 6.2.2
- 6.3.3

若两个 p point to same type，则可以相减

$$p_1 - p_2 = \frac{p_1 - p_2}{\text{size of } (*p)}$$

a[0] ≡ *a

```
 9 int sum_array(const int *a, const int len) {
10   assert(a);
11   assert(len > 0);
12   int sum = 0;
13   for (const int *p = a; p < a + len; ++p) {
14     sum += *p;
15   }
16   return sum;
17 }
18
19 // traditional array notation
20
21 int sum_array_old(const int a[], const int len) {
22   assert(a);
23   assert(len > 0);
24   int sum = 0;
25   for (int i = 0; i < len; ++i) {
26     sum += a[i];
27   }
28   return sum;
29 }
```

改 a，因为 a 不会改变

- quick sort 排序

```c

// quick_sort_range(a, first, last) sorts the elements of
//   a in the range a[first]..a[last] (inclusive)
//   in ascending order
// requires: a is a valid array in the range first..last [not as
// effects: modifies a
void quick_sort_range(int a[], int first, int last) {
  assert(a);
  if (last <= first) return;  // length is <= 1

  int pivot = a[first];       // first element is the pivot
  int pos = last;             // where to put next larger

  for (int i = last; i > first; --i) {
    if (a[i] > pivot) {
      swap(&a[pos], &a[i]);
      --pos;
    }
  }
  swap(&a[first], &a[pos]);   // put pivot in correct place
  quick_sort_range(a, first, pos - 1);
  quick_sort_range(a, pos + 1, last);
}

// quick_sort(a, len) sorts the elements of a in ascending order
// requires: len > 0
// effects: modifies a
void quick_sort(int a[], int len) {
  assert(a);
  assert(len > 0);
  quick_sort_range(a, 0, len - 1);
}

int main(void) {
  int a[7] = {8, 6, 7, 5, 3, 0, 9};
  print_array(a, 7);
  quick_sort(a, 7);
  print_array(a, 7);
}
```

Code Output

```
8, 6, 7, 5, 3, 0, 9.
0, 3, 5, 6, 7, 8, 9.
```

- binary search 找数

ample 6.3.5: Binary Search

main.c - Add File

```c
// This program demonstrates a binary search
// on a sorted array

#include <assert.h>
#include "cs136-trace.h"

// find_sorted(item, a, len) finds the index of item in a,
//   or returns -1 if it does not exist
// requires: a is sorted in ascending order [not asserted]
//           len > 0
int find_sorted(int item, const int a[], int len) {
  assert(a);
  assert(len > 0);
  int low = 0;
  int high = len - 1;
  while (low <= high) {
    int mid = low + (high - low) / 2;
    if (a[mid] == item) {
      return mid;
    } else if (a[mid] < item) {
      low = mid + 1;
    } else {
      high = mid - 1;
    }
  }
  return -1;
}

int main(void) {
  int a[10] = {1, 2, 3, 5, 7, 11, 13, 17, 19, 23};

  assert(find_sorted(1, a, 10) == 0);
  assert(find_sorted(13, a, 10) == 6);
  assert(find_sorted(23, a, 10) == 9);
  assert(find_sorted(8, a, 10) == -1);
}
```

必须用于 sorted list

use "divide and conquer"

- Oversized Array (len 过大)

```
int arr[10] = {115, 116, 136, 245, 246};
```

Select the correct dropdown option.

```
printf("%d", arr[6]);
```

0

```
printf("%d", arr[0]);
```

115

```
printf("%d", arr[-1]);
```

An error/warning or unexpected behaviour will occur due to invalid array index.

```
printf("%d", arr[136]);
```

An error/warning or unexpected behaviour will occur due to invalid array index.

7. Efficiency

- algorithm. step-by-step description of how to solve a problem.

- efficiency.

  time ~ : how long it takes to solve a problem.

  space ~ : how much memory requires to solve a problem.

  power ~

- running time $T(n)$

  measure the number of elementary operators required.

  n : size / length of data

  worst case running time

- Big O

  "order" of running time : "dominant" term. without constant coefficient.

  2 algorithm are equivalent if they have same order.

growth rate (小)

↓

$O(1)$

$O(\log n)$

$O(n)$

$O(n \log n)$

$O(n^2)$

$O(n^3)$

$O(2^n)$

↓ (大)

加法 : result = 较大数

$O(\log n) + O(n) = O(n)$

乘法 : result = 两者相乘

$O(\log n) \times O(n) = O(n \log n)$

一、计算

① Simple

```
// max(a, b) returns the max of the elements
// time: O(1)+O(1)+ ... +O(1) = O(1)
int max(int a, int b) {
  if (a > b) {
    return a;
  } else {
  return b;
  }
}
```

② Iterative analysis (summations)

1. 内层 → 外层
2. worst case 迭代次数. 数据大小
3. $T(n)$

$$\sum_{i=1}^{n/10} O(1) = \sum_{i=1}^{n} O(1)$$

ep.

```
for (i = 1; i <= n; ++i) {
  printf("*");
}
```

$$T(n) = \sum_{i=1}^{n} O(1) = O(1) \times n = O(n)$$

$$\sum_{i=1}^{\log n} O(1) = O(\log n)$$

$$\sum_{i=1}^{n} O(1) = O(n)$$

$$\sum_{i=1}^{n} O(n) = O(n^2)$$

$$\sum_{i=1}^{n} O(i) = O(n^2)$$

$$\sum_{i=1}^{n} O(i^2) = O(n^3)$$

ep.

```
for (i = 0; i < n; ++i) {
  for (j = 0; j < i; ++j) {
    printf("*");
  }
  printf("\n");
}
```

← nested loops

inner loop: $\sum_{j=0}^{j=i-1} O(1) = O(i)$

outer loop: $\sum_{i=0}^{i=n-1} (O(1) + O(i)) = O(n) + O(n^2) = O(n^2)$

外层 内层.

③ Recursion

1. order of function. (不包括recursion)
2. 下一个递归 in data size
3. recurrence relation
4. close-form sol.

```
int sum_first(int n) {
  if (n == 0) {
    return 0;
  } else {
    return n + sum_first(n - 1);
  }
}
```

1. non-recursive operators: $O(1)$, +, -, ==.
2. $n-1$
3. $T(n) = O(1) + T(n-1)$
4. $T(n) = O(n)$

The recurrence relations we encounter in this course are:

$T(n) = O(1) + T(n - k_1)$	$= O(n)$
$T(n) = O(n) + T(n - k_1)$	$= O(n^2)$
$T(n) = O(n^2) + T(n - k_1)$	$= O(n^3)$
$T(n) = O(1) + T(\frac{n}{k_2})$	$= O(\log n)$
$T(n) = O(1) + k_2 * T(\frac{n}{k_2})$	$= O(n)$
$T(n) = O(n) + k_2 * T(\frac{n}{k_2})$	$= O(n \log n)$
$T(n) = O(1) + T(n - k_1) + T(n - k_1')$	$= O(2^n)$

where $k_1, k_1' \geq 1$ and $k_2 > 1$

- Sorting (same efficiency)

Algorithm	best case	worst case
selection sort	$O(n^2)$	$O(n^2)$
insertion sort	$O(n)$	$O(n^2)$
quick sort	$O(n \log n)$	$O(n^2)$

$> \sum_{i=1}^{n} \sum_{j=1}^{i} O(1) = O(n^2)$

- Binary Search

$$T(n) = \sum_{i=1}^{\log_2 n} O(1) = O(\log n)$$

n 为 loop in 次数

```
void f4(int n) {
  for (int i = 0; i < n; i += 2) {
    for (int j = n; j > 0; j /= 2) {
      printf("*");
    }
    printf("\n");
  }
}
```

inner loop: $\sum_{i=1}^{\log n} O(1) = O(\log n)$

outer loop: $\sum_{i=1}^{n} O(\log n) + O(1)$

$= O(n \log n) + O(n)$

$= O(n \log n)$

8. String

- C string : array of char. terminated by null character.

terminator '\0' = 0

- initialization initial in 位置输 '10'

- null terminated

while (s[i]) 检测 array 是否结束

- string len int strlen(s)

```c
// my_strlen(s) behaves the same as the built-in strlen function
//   the length does not include the null character
// time: O(n)
int my_strlen(const char s[]) {
  assert(s);
  int len = 0;
  while (s[len]) {
    ++len;
  }
  return len;
}

// my_strlen_ptr(s) behaves the same as the built-in strlen
//   the length does not include the null character
// time: O(n)
int my_strlen_ptr(const char *s) {
  assert(s);
  const char *p = s;
  while (*p) {
    ++p;
  }
  return (p - s);
}
```

- lexicographical order int strcmp(s1, s2)

```c
// my_strcmp(s1, s2) behaves the same as strcmp and returns:
//   s1 preceeds s2      => negative int
//   s1 identical to s2  => 0
//   s1 follows s2       => positive int
// time: O(n), n is min of the lengths of s1, s2
int my_strcmp(const char s1[], const char s2[]) {
  assert(s1);
  assert(s2);
  int i = 0;
  while (s1[i] == s2[i] && s1[i]) {
    ++i;
  }
  return s1[i] - s2[i];
}

int main(void) {
  char a[] = "the same?";
  char b[] = "the same?";
  char c[] = "the samejQuery22408322300989689644_1654783296792";
  char *s = a;

  assert(a != b);
  assert(a == s);

  assert(my_strcmp(a, b) == 0);
  assert(my_strcmp(a, c) < 0); // -43
  assert(my_strcmp(c, a) > 0); //  43
}
```

("ab", "a") → 1

("a", "b") → -1

("", "ab") → -1

- strcpy　　char *strcpy(char *dest, const char *src)

将 dest 与 src 连接在一起

```c
// my_strcpy(dest, src) behaves the same as the built-in strcpy function
// time: O(n), n is the length of src
char *my_strcpy(char *dest, const char *src) {
  assert(dest);
  assert(src);
  char *d = dest;
  while (*src) {
    *d = *src;
    ++d;
    ++src;
  }
  *d = '\0';
  return dest;
}

int main(void) {
  char a[] = "cat";
  char b[4] = "dog";
  my_strcpy(b, a);
  trace_string(a);
  trace_string(b);
  assert(!strcmp(a, b)); // !a==b
}
```

buffer overflow (overrun)　　∴确保 dest array 足够大

↳ arr[2] = "hi"　数组大小为3. 'h' 'i' '\0'

致命

- strcat　　char *strcat(char* dest, const char* src)

将 src 加入到 dest 后面 (合并两个 char)

```c
// my_strcat(dest, src) behaves the same as the built-in strcat function
// time: O(n + m) n,m are lengths of src,dest
char *my_strcat(char *dest, const char *src) {
  assert(dest);
  assert(src);
  strcpy(dest + strlen(dest), src);
  return dest;
}

int main(void) {
  char a[] = "cat";
  char b[7] = "dog";
  my_strcat(b, a);
  trace_string(a);
  trace_string(b);
  assert(!strcmp(b, "dogcat"));
}
```

array comparison always evaluate to false

if 确定等于0都可以运行

- I/O

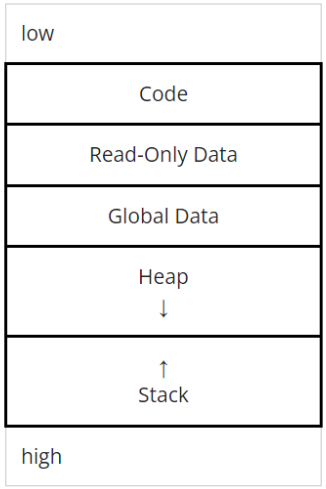
用 scanf to read string will lead to buffer overrun problems.

- String literal

def. "(string)" in an expression. (及 initialization)

read-only data section of memory

9. Dynamic Memory



- Heap.

def. final section in C memory model.

advantage of heap-allocated memory: 分配内存
1) dynamic : the allocation size can be determined at run time
2) resizable : a heap allocation can be resized
3) duration : heap allocation persist until they are "freed"
4) safety : if memory runs out, it can be detected and handled properly.

- dynamic memory

side effect: frees a. 动态内存
modifies "state" of the heap

most common use of dynamic memory: allocating space for arrays and structures
advantage : function can obtain memory that persists after obtained
side effects : modifies the "state" of heap.

- malloc (memory allocation)

requires s bytes of memory from heap.

malloc(s) ⟶ return pointer / NULL

- realloc

resize array

步骤 : 1. create new array.
2. copy items
3. free old

realloc (p. newsize). O(n)
↑
≥ 原array in size

导致 dangling pointer 悬空指针
free a;

导致 memory leak
重复定义 a.

- merge sort (n log n) ← worst case

- uninitialized.

```c
// This program demonstrates how memory returned
// from malloc is UNINITIALIZED

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

struct posn {
  int x;
  int y;
};

int main(void) {
  int *my_array = malloc(10 * sizeof(int));
  my_array[3]= 2;
  printf("the mystery value of my_array[0]: %d\n", my_array[3]);
  printf("the mystery value of my_array[0]: %d\n", my_array[0]);
  free(my_array);

  struct posn *my_posn = malloc(sizeof(struct posn));
  printf("the mystery values of my_posn: (%d, %d)\n", my_posn->x, my_posn->y);
  free(my_posn);
}
```

分配 40 bytes memory 存到 *my_array

Code Output

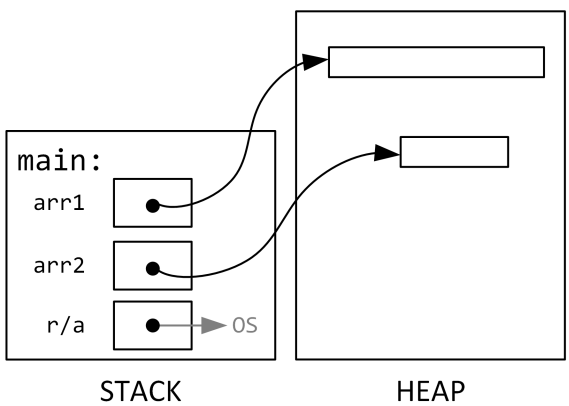
```
the mystery value of my_array[0]: 2
the mystery value of my_array[0]: -1094795586
the mystery values of my_posn: (-1094795586, -1094795586)
```

initialized

uninitialized

- Visualizing heap

```c
int main(void) {
int *arr1 = malloc(10 * sizeof(int));
int *arr2 = malloc(5 * sizeof(int));
//...
}
```



out of memory malloc (sizeof(int) * INT_MAX)

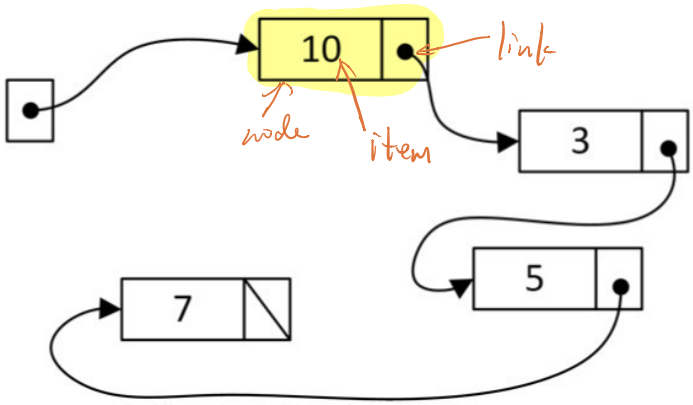
free(p) return pointer p back to heap

使p变为 dangling pointer (悬指针)

10. ADT

- linked list

  represented as a structure that contains a link (pointer) to the front node



recursive data structure

```c
struct llnode {
  int item;
  struct llnode *next;
};

struct llist {
  struct llnode *front;
};
```

- Queue ADT

  add_back

  remove_front

  front

  is_empty

- Tree ADT

```c
struct bstnode {
    int item;
    struct bstnode *left;
    struct bstnode *right;
    int count;                  // *****NEW
};
```

- dictionary ADT.

  lookup
  insert
  remove

- ADT design.

combination of { primitives, structures, array, linked lists, trees, graph }

traditional collection ADT { stack, queue, sequence, dictionary }

- dynamic array — sequenced data. "specific position"
- sorted dynamic array — data are frequently searched. × add remove
- linked list — sequenced. element 在头/尾加减
- self-balancing binary search tree — unsequenced. search / add remove

- generic ADT implementation.