C Primer

CS 351: Systems Programming
Michael Saelee <lee@iit.edu>
I have stopped reading Stephen King novels. 
Now I just read C code instead.

- Richard O’Keefe
Agenda

1. Overview
2. Basic syntax & structure
3. Compilation
4. Visibility & Lifetime
Agenda

5. Pointers & Arrays
6. Dynamic memory allocation
7. Composite data types
8. Function pointers
Not a Language Course!

- Resources:
  - K&R (*The C Programming Language*)
  - comp.lang.C FAQ ([c-faq.com](http://c-faq.com))
NAME
   strlen - find length of string

LIBRARY
   Standard C Library (libc, -lc)

SYNOPSIS
   #include <string.h>

   size_t
   strlen(const char *s);

DESCRIPTION
   The strlen() function computes the length of the string s.

RETURN VALUES
   The strlen() function returns the number of characters that precede the
   terminating NUL character.

SEE ALSO
   string(3)
§Overview
C is ...

- imperative
- statically typed
- weakly type checked
- procedural
- low level
A language that doesn't have everything is actually easier to program in than some that do.

- Dennis Ritchie
§Basic syntax & structure
Primitive Types

- **char**: one byte integer (e.g., for ASCII)
- **int**: integer, *at least* 16 bits
- **float**: single precision floating point
- **double**: double precision floating point
Integer type prefixes

- signed (default), unsigned
  - same storage size, but sign bit on/off
- short, long
  - sizeof (short int) ≥ 16 bits
  - sizeof (long int) ≥ 32 bits
  - sizeof (long long int) ≥ 64 bits
Recall C’s weak type-checking...

/* types are implicitly converted */
char c = 0x41424344;
int i = 1.5;
unsigned int u = -1;
float f = 10;
double d = 2.5F; // note 'F' suffix for float literals

printf("c = '%c', i = %d, u = %u, f = %f, d = %f\n", c, i, u, f, d);

/* typecasts can be used to force conversions */
int r1 = f / d,
r2 = f / (int) d;

printf("r1 = %d, r2 = %d\n", r1, r2);

c = 'D', i = 1, u = 4294967295, f = 10.000000, d = 2.500000
r1 = 4, r2 = 5
Basic Operators

- Arithmetic: +, -, *, /, %, ++, --, &,&, |, ~
- Relational: <, >, <=, >=, ==, !=
- Logical: &&, ||, !
- Assignment: =, +=, *=, ...
- Conditional: bool ? true_exp : false_exp
True/False

- 0 = False
- Everything else = True
  - But *canonical* True = 1
Boolean Expressions

! (0) → 1
0 || 2 → 1
3 && 0 && 6 → 0
!(1234) → 0
!!(-1020) → 1
Control Structures

- if-else
- switch-case
- while, for, do-while
- continue, break
Variables

- Must declare before use
- Declaration implicitly allocates storage for underlying data
- Note: not true in Java!
Functions

- C’s *top-level* modules

- Procedural language vs. OO: no classes!
Declaration vs. Definition

- Declaration (aka prototype): arg & ret type
- Definition: function body
- A function can be declared many times but only defined once
Declarations reside in *header* (.h) files, Definitions reside in *source* (.c) files

(Suggestions, not really requirements)
hashtable.h

```c
unsigned long hash(char *str);
hashhtable_t *make_hashtable(unsigned long size);
void ht_put(hashtable_t *ht, char *key, void *val);
void *ht_get(hashtable_t *ht, char *key);
void ht_del(hashtable_t *ht, char *key);
void ht_iter(hashtable_t *ht, int (*f)(char *, void *));
void ht_rehash(hashtable_t *ht, unsigned long newsize);
int ht_max_chain_length(hashtable_t *ht);
void free_hashtable(hashtable_t *ht);
```

hashtable.c

```c
#include "hashtable.h"

unsigned long hash(char *str) {
    unsigned long hash = 5381;
    int c;
    while ((c = *str++))
        hash = ((hash << 5) + hash) + c;
    return hash;
}

hashtable_t *make_hashtable(unsigned long size) {
    hashtable_t *ht = malloc(sizeof(hashtable_t));
    ht->size = size;
    ht->buckets = calloc(sizeof(bucket_t *), size);
    return ht;
}

...
### hashtable.h

```c
unsigned long hash(char *str);
hashtable_t *make_hash_table(unsigned long size);
void ht_put(hashtable_t *ht, char *key, void *val);
void *ht_get(hashtable_t *ht, char *key);
void ht_del(hashtable_t *ht, char *key);
void ht_iter(hashtable_t *ht, int (*f)(char *, void *));
void ht_rehash(hashtable_t *ht, unsigned long newsize);
int ht_max_chain_length(hashtable_t *ht);
void free_hash_table(hashtable_t *ht);
```

### main.c

```c
#include "hashtable.h"

int main(int argc, char *argv[]) {
    hashtable_t *ht;
    ht = make_hash_table(atoi(argv[1]));
    ...
    free_hash_table(ht);
    return 0;
}
```
§Compilation
main.c

#include <stdio.h>

int main () {
    printf("Hello world!\n");
    return 0;
}

$ gcc main.c -o prog
$ ./prog
Hello world!
gcc -c greet.c     -o greet.o
$ gcc -c main.c      -o main.o
$ gcc greet.o main.o -o prog
$ ./prog
Hello, Michael
“Preprocessing”

- preprocessor directives exist for:
  - text substitution
  - macros
  - conditional compilation
```c
#define msg "Hello world!\n"

int main () {
    printf(msg);
    return 0;
}
```

```
$ gcc -E hello.c

int main () {
    printf("Hello world!\n");
    return 0;
}
```
`#define PLUS1(x) (x+1)`

```
int main () {
    int y;
    y = y * PLUS1(y);
    return 0;
}
```

```
$ gcc -E plus1.c

int main () {
    int y;
    y = y * (y+1);
    return 0;
}
```
```c
#define SAYHI

int main () {
    #ifdef SAYHI
        printf("Hi!");
    #else
        printf("Bye!");
    #endif
    return 0;
}
```

```
$ gcc -E hello.c

int main () {
    printf("Hi!");
    return 0;
}
```
“Linking”

- Resolving calls/references and definitions
  - e.g., putting absolute/relative addresses in the (assembly) call instruction
- Note: dynamic linking is also possible (link in shared library at run-time)
```c
#include <stdio.h>
#include "greet.h"

void greet(char *name) {
    printf("Hello, %s\n", name);
}

int main() {
    greet("Michael");
    return 0;
}
```

```
gcc -c greet.c -o greet.o
$ gcc -c main.c -o main.o
$ gcc greet.o main.o -o prog
$ ./prog
Hello, Michael
```
$ objdump -d greet.o
0000000000000000 <greet>:
  0:   55                      push   %rbp
  1:   48 89 e5                mov    %rsp,%rbp
  4:   48 83 ec 10             sub    $0x10,%rsp
  8:   48 89 7d f8             mov    %rdi,-0x8(%rbp)
 c:   48 8b 45 f8             mov    -0x8(%rbp),%rax
10:   48 89 c6                mov    %rax,%rsi
13:   bf 00 00 00 00          mov    $0x0,%edi
18:   b8 00 00 00 00          mov    $0x0,%eax
 id:   e8 00 00 00 00          callq  22 <greet+0x22>
22:   90                      nop
23:   c9                      leaveq
24:   c3                      retq

$ objdump -d main.o
0000000000000000 <main>:
  0:   55                      push   %rbp
  1:   48 89 e5                mov    %rsp,%rbp
  4:   bf 00 00 00 00          mov    $0x0,%edi
  9:   e8 00 00 00 00          callq  e <main+0xe>
e:   b8 00 00 00 00           mov    $0x0,%eax
13:   5d                      pop    %rbp
14:   c3                      retq
$ objdump -d prog

00000000004003f0 <printf@plt-0x10>:
  4003f0: ff 35 12 0c 20 00 pushq 0x200c12(%rip)  # 601008 <GLOBAL_OFFSET_TABLE_+0x8>
  4003f6: ff 25 14 0c 20 00 jmpq *0x200c14(%rip) # 601010 <GLOBAL_OFFSET_TABLE_+0x10>
  4003fc: 0f 1f 40 00 nopl 0x0(%rax)

0000000000400400 <printf@plt>:
  400400: ff 25 12 0c 20 00 jmpq *0x200c12(%rip) # 601018 <GLOBAL_OFFSET_TABLE_+0x18>
  400406: 68 00 00 00 00 pushq $0x0
  40040b: e9 e0 ff ff ff jmpq 4003f0 <_init+0x28>

0000000000400526 <main>:
  400526: 55 push %rbp
  400527: 48 89 e5 mov %rsp,%rbp
  40052a: bf e4 05 40 00 mov $0x4005e4,%edi
  40052f: e8 07 00 00 00 callq 40053b <greet>
  400534: b8 00 00 00 00 mov $0x0,%eax
  400539: 5d pop %rbp
  40053a: c3 retq

000000000040053b <greet>:
  40053b: 55 push %rbp
  40053c: 48 89 e5 mov %rsp,%rbp
  40053f: 48 83 ec 10 sub $0x10,%rsp
  400543: 48 89 7d f8 mov %rdi,-0x8(%rbp)
  400547: 48 8b 45 f8 mov -0x8(%rbp),%rax
  40054b: 48 89 c6 mov %rax,%rsi
  40054e: bf ec 05 40 00 mov $0x4005ec,%edi
  400553: b8 00 00 00 00 mov $0x0,%eax
  400558: e8 a3 fe ff ff callq 400400 <printf@plt>
  40055d: 90 nop
  40055e: c9 leaveq
  40055f: c3 retq
“Linking”

- But!
  - Don’t always want to allow linking a call to a definition
    - e.g., to hide implementation
  - Want to support selective public APIs
“Linking”

- But!

- Also, how to separate declaration & definition of a variable? (and why?)
§ Visibility & Lifetime
**Visibility**: *where* can a symbol (var/fn) be seen from, and how do we refer to it?

**Lifetime**: *how long* does allocated storage space (e.g., for a var) remain useable?
```c
int glob_i = 0;

int main() {
    int i = 10;
    glob_i = 10;
    foo();
    printf("%d, %d\n", i, glob_i);
    return 0;
}

void foo() {
    i++;
    glob_i++;
}
```

```
$ gcc -Wall -o demo viz_life.c
viz_life.c: In function ‘main’:
viz_life.c:6: warning: implicit declaration of function ‘foo’
viz_life.c:7: warning: implicit declaration of function ‘printf’
viz_life.c:7: warning: incompatible implicit declaration of built-in function ‘printf’
viz_life.c: At top level:
viz_life.c:11: warning: conflicting types for ‘foo’
viz_life.c:6: warning: previous implicit declaration of ‘foo’ was here
viz_life.c: In function ‘foo’:
viz_life.c:12: error: ‘i’ undeclared (first use in this function)
viz_life.c:12: error: (Each undeclared identifier is reported only once
viz_life.c:12: error: for each function it appears in.)
```
```c
#include <stdio.h>

void foo();

int glob_i = 0;

int main() {
    int i = 10;
    glob_i = 10;
    foo();
    printf("%d, %d\n", i, glob_i);
    return 0;
}

void foo() {
    int i;
    i++; 
    glob_i++;
}
```

```
$ gcc -Wall -o demo viz_life.c
$ ./demo
10, 11
```
$ gcc -Wall -o demo sum.c main.c
sum.c: In function `sumWithI':
sum.c:2: error: `I' undeclared (first use in this function)
main.c: In function `main':
main.c:6: warning: implicit declaration of function `sumWithI'

```c
int sumWithI(int x, int y) {
    return x + y + I;
}
```

```c
#include <stdio.h>

int I = 10;

int main() {
    printf("%d\n", sumWithI(1, 2));
    return 0;
}
```
```c
#include <stdio.h>

int sumWithI(int x, int y) {
    int I;
    return x + y + I;
}

int main() {
    printf("%d\n", sumWithI(1, 2));
    return 0;
}
```

```
$ gcc -Wall -o demo sum.c main.c
$ ./demo
-1073743741
```
problem: variable *declaration & definition* are implicitly tied together

note: definition = *storage allocation + possible initialization*
extern keyword allows for declaration *sans definition*
```c
#include <stdio.h>

int sumWithI(int x, int y) {
    extern int I;
    return x + y + I;
}

int I = 10;

int main() {
    printf("%d\n", sumWithI(1, 2));
    return 0;
}
```

```
$ gcc -Wall -o demo sum.c main.c
$ ./demo
demo
13
```
... and now global variables are visible from everywhere.

Good/Bad?
static keyword lets us limit the visibility of things
int sumWithI(int x, int y) {
    extern int I;
    return x + y + I;
}

#include <stdio.h>

int sumWithI(int, int);

static int I = 10;

int main() {
    printf("%d\n", sumWithI(1, 2));
    return 0;
}

$ gcc -Wall -o demo sum.c main.c
Undefined symbols:
  "_I", referenced from:
    _sumWithI in ccmvi0RF.o
ld: symbol(s) not found
collect2: ld returned 1 exit status
```c
#include <stdio.h>

int sumWithI(int x, int y); 
int I = 10;

int main() {
    printf("%d\n", sumWithI(1, 2));
    return 0;
}
```

```
static int sumWithI(int x, int y) {
    extern int I;
    return x + y + I;
}
```

$ gcc -Wall -o demo sum.c main.c
Undefined symbols:
"_sumWithI", referenced from:
    _main in cc9LhUBP.o
ld: symbol(s) not found
collect2: ld returned 1 exit status
**static** also forces the *lifetime* of variables to be equivalent to **global**

(i.e., stored in static memory vs. stack)
```c
#include <stdio.h>

int sumWithI(int x, int y) {
    static int I = 10; // init once
    return x + y + I++;
}

int main() {
    printf("%d\n", sumWithI(1, 2));
    printf("%d\n", sumWithI(1, 2));
    printf("%d\n", sumWithI(1, 2));
    return 0;
}
```

$ gcc -Wall -o demo sum.c main.c
$ ./demo
13
14
15
recap:

- by default, variable *declaration* also results in *definition* (storage allocation)

- `extern` is used to declare a variable but use a separate definition
recap:

- by default, functions & global vars are visible within all linked files

- **static** lets us limit the visibility of symbols to the defining file
recap:

- by default, variables declared inside functions have *local lifetimes* (stack-bound)

- `static` lets us change their storage class to static (aka “global”)
§ Pointers
(don’t panic!)
a pointer is a variable declared to store a memory address
Q: by examining a variable’s contents, can we tell if the variable is a pointer?

e.g., \texttt{0x0040B100}
No!

- a pointer is designated by its \textit{static (declared) type}, not its contents
A pointer declaration also tells us the type of data to which it should point.
declaration syntax: `type *var_name`
```c
int *ip
char *cp;
struct student *sp;
```
```c
int
char
struct student
```
Important pointer-related operators:

& : address-of

* : dereference (*not the same as the * used for declarations!!!*)
```c
int i = 5;  /* i is an int containing 5 */
int *p;     /* p is a pointer to an int */

p = &i;     /* store the address of i in p */

int j;     /* j is an uninitialized int */
j = *p;     /* store the value p points to into j*/
```
```c
int main() {
    int i, j;
    int *p, *q;
    i = 10;
    p = j;
    return 0;
}
```

$ gcc pointers.c
pointers.c: In function 'main':
pointers.c:5: warning: assignment makes pointer from integer without a cast
```c
int main() {
    int i, j, *p, *q;
    i = 10;
    p = &j;
    q = *p;
    return 0;
}
```

```
$ gcc pointers.c
pointers.c: In function ‘main’: 
pointers.c:6: warning: assignment makes pointer from integer without a cast
```
int main() {
    int i, j, *p, *q;
    i = 10;
    p = &j;
    q = &p;
    return 0;
}

$ gcc pointers.c
pointers.c: In function ‘main’:
pointers.c:6: warning: assignment from incompatible pointer type
```c
int main() {
    int i, j, *p, *q;
    i = 10;
    p = &j;
    q = p;
    *q = *i;
    return 0;
}
```

```
$ gcc pointers.c
pointers.c: In function ‘main’:
pointers.c:7: error: invalid type argument of ‘unary *’
```
```c
int main() {
    int i, j, *p, *q;

    i = 10;
    p = &j;
    q = p;
    *q = i;
    j = q;

    return 0;
}
```

$ gcc pointers.c
pointers.c: In function ‘main’: pointers.c:8: warning: assignment makes integer from pointer without a cast
```c
int main() {
    int i, j, *p, *q;

    i = 10;
    p = &j;
    q = p;
    *q = i;
    *p = *q * 2;
    printf("i=%d, j=%d, *p=%d, *q=%d\n", i, j, *p, *q);
    return 0;
}
```

```
$ gcc pointers.c
$ .a.out
i=10, j=20, *p=20, *q=20
```
```c
int i, j, *p, *q;
i = 10;
p = &j;
q = p;

*p = *q * 2;

*i = *j; 
```
why have pointers?
```c
int main() {
    int a = 5, b = 10;
    swap(a, b);
    /* want a == 10, b == 5 */
    ...
}

void swap(int x, int y) {
    int tmp = x;
    x = y;
    y = tmp;
}
```
```c
int main() {
    int a = 5, b = 10;
    swap(&a, &b);
    /* want a == 10, b == 5 */
    ...
}

void swap(int *p, int *q) {
    int tmp = *p;
    *p = *q;
    *q = tmp;
}
```
pointers enable *action at a distance*
```c
void bar(int *p) {
    *p = ...; /* change some remote var! */
}

void bat(int *p) {
    bar(p);
}

void baz(int *p) {
    bat(p);
}

int main() {
    int i;
    baz(&i);
    return 0;
}
```
action at a distance is an *anti-pattern* i.e., an oft used but typically crappy programming solution
back to swap

```c
void swap(int *p, int *q) {
    int tmp = *p;
    *p = *q;
    *q = tmp;
}

int main() {
    int a = 5, b = 10;
    swap(&a, &b);
    /* want a == 10, b == 5 */
    ...
}
```
... for swapping pointers?

```c
int main() {
    int a, b, *p, *q;
    p = &a;
    q = &b;

    swap(p, q);
    /* want p to point to b, q to a */
    ...
}

void swap(int *p, int *q) {
    int tmp = *p;
    *p = *q;
    *q = tmp;
}
```
```c
int main() {
    int a, b, *p = &a, *q = &b;

    swap(&p, &q); /* want p to point to b, q to a */
}

void swap(int *p, int *q) {
    int tmp = *p;
    *p = *q;
    *q = tmp;
}
```

$ gcc pointers.c
pointers.c: In function ‘main’:
pointers.c:10: warning: passing argument 1 of ‘swap’ from incompatible pointer type
pointers.c:10: warning: passing argument 2 of ‘swap’ from incompatible pointer type
int main() {
    int a, b, *p = &a, *q = &b;

    swapp(&p, &q);
    /* want p to point to b, q to a */
}

void swapp(int **p, int **q) {
    int *tmp = *p;
    *p = *q;
    *q = tmp;
}

(int **) declares a
pointer to a pointer to an int
Uninitialized pointers

- are like all other uninitialized variables
  - i.e., contain garbage
- dereferencing garbage ...
  - if lucky → crash
  - if unlucky → ???
“Null” pointers

- never returned by & operator
- safe to use as sentinel value
- written as $\theta$ in pointer context
  - for convenience, #define’d as NULL
“Null” pointers

```c
int main() {
    int i = 0;
    int *p = NULL;

    ...

    if (p) {
        /* (likely) safe to deref p */
    }
}
```
§ Arrays
contiguous, indexed region of memory
Declaration: `type arr_name[size]`

- remember, declaration also allocates storage!
int  i_arr[10];    /* array of 10 ints */
char c_arr[80];    /* array of 80 chars */
char td_arr[24][80]; /* 2-D array, 24 rows x 80 cols */
int  *ip_arr[10];  /* array of 10 pointers to ints */

/* dimension can be inferred if initialized when declaring */
short grades[] = { 75, 90, 85, 100 };

/* can only omit first dim, as partial initialization is ok */
int sparse[][10] = {{ 5, 3, 2 },
                    { 8, 10 },
                    { 2 }};

/* if partially initialized, remaining components are 0 */
int zeros[1000] = { 0 };

/* can also use designated initializers for specific indices*/
int nifty[100] = { [0] = 0,
                   [99] = 1000,
                   [49] = 250 };
In C, arrays contain *no metadata* i.e., *no implicit size, no bounds checking*
```c
int main() {
    int i, arr[10];

    for (i=0; i<100; i++) {
        arr[i] = 0;
    }
    printf("Done\n");

    return 0;
}
```

$ gcc arr.c
$ ./a.out

*(runs forever ... no output)*
int main() {
    int arr[10], i;

    for (i=0; i<100; i++) {
        arr[i] = 0;
    }

    printf("Done\n");

    return 0;
}

$ gcc arr.c
$ ./a.out
Done
[1] 10287 segmentation fault ./a.out
$
direct access to memory can be dangerous!
pointers ♥ arrays

- an array name is bound to the address of its first element
  - i.e., array name is a const pointer

- conversely, a pointer can be used as though it were an array name
```c
int *pa;
int arr[5];

pa = &arr[0]; /* <=> */ pa = arr;
arr[i]; /* <=> */ pa[i];
*arr; /* <=> */ *pa;

int i;

pa = &i; /* ok */
arr = &i; /* not possible! */
```
§Pointer Arithmetic
follows naturally from allowing array subscript notation on pointers
int arr[100];

int *pa = arr;

pa[10] = 0;  /* set tenth element */
/* so it follows ... */

*(pa + 10) = 0;  /* set tenth element */
/* surprising! "adding" to a pointer 
accounts for element size -- does not 
blindly increment address */
takeaway:

- pointer arithmetic makes use of pointee data types to compute byte offsets
when “passed” as arguments, arrays degenerate into pointers i.e., no aggregate size information!
/ * alt syntax for param `p` is valid, but misleading ... `p` is a pointer in `foo`, not an array! */
void foo(char p[]) {
    printf("In foo, sizeof p = %lu\n", sizeof p);
}

int main() {
    char str[80];
    printf("In main, sizeof str = %lu\n", sizeof str);
    foo(str);
    return 0;
}
**strings** are just \( \theta \) terminated char arrays
char str[] = "hello!";
char *p = "hi";
char tarr[][5] = {"max", "of", "four"};
char *sarr[] = {"variable", "length", "strings"};
/* printing a string (painfully) */

int i;
char *str = "hello world!";
for (i = 0; str[i] != 0; i++) {
    printf("%c", str[i]);
}

/* or just */

printf("%s", str);
/* Beware: */

```c
int main() {
    char *str = "hello world!";
    str[12] = 10;
    printf("%s", str);
    return 0;
}
```

$ ./a.out
[1] 10522 bus error ./a.out
/* the fleshed out "main" with command-line args */

int main(int argc, char *argv[]) {
    int i;
    for (i=0; i<argc; i++) {
        printf("%s", argv[i]);
        printf("%s", ((i < argc-1)? ", " : "\n") );
    }
    return 0;
}

$ ./a.out testing one two three
./a.out, testing, one, two, three
§ Dynamic Memory Allocation
**dynamic** vs. **static** (lifetime = forever)  
vs. **local** (lifetime = LIFO)
C requires *explicit* memory management
- must request & free memory manually
- if forget to free $\rightarrow$ memory **leak**
basic C “malloc” API (in stdlib.h):

- malloc
- realloc
- free
malloc lib is *type agnostic*
i.e., it doesn’t care what data types we store in requested memory
need a “generic” / type-less pointer:

(void *)
assigning from/to \((\text{void} \ *)\) to/from any other pointer \textit{will never produce warnings}

… Hurrah! (but \textit{dangerous})
void *malloc(size_t size);
void *realloc(void *ptr, size_t size);
void free(void *ptr);

all sizes are in bytes;
all ptrs are from previous malloc requests
/* clone into dynamically alloc'd memory */
char *strdup(const char *str) {
    char *nstr = malloc(strlen(str) + 1);
    strcpy(nstr, str);
    return nstr; /* someone else must free this! */
}

/* one way to do a "generic" swap */
void swap(void *p, void *q, int size) {
    void *tmp = malloc(size);
    memcpy(tmp, p, size);
    memcpy(p, q, size);
    memcpy(q, tmp, size);
    free(tmp);
}
int i, j, k=1;
int *jagged_arr[5]; /* array of 5 pointers to int */
for (i=0; i<5; i++) {
    jagged_arr[i] = malloc(sizeof(int) * k);
    for (j=0; j<k; j++) {
        jagged_arr[i][j] = k;
    }
    k += 1;
}

/* use jagged_arr ... */
for (i=0; i<5; i++) {
    free(jagged_arr[i]);
}
```c
int i, j, k=1;
int *jagged_arr[5]; /* array of 5 pointers to int */
for (i=0; i<5; i++) {
    jagged_arr[i] = malloc(sizeof(int) * k);
    for (j=0; j<k; j++) {
        jagged_arr[i][j] = k;
    }
    k += 1;
}
```

(gdb) run
Starting program: /Users/lee/demo/a.out
Breakpoint 1, main () at demo.c:18
(gdb) p jagged_arr
$1 = {0x1001000e0, 0x100103ad0, 0x100103ae0, 0x100103af0, 0x100103b00}
(gdb) p jagged_arr[0][0]
$2 = 1
(gdb) p *jagged_arr[0]
$3 = 1
(gdb) p *(int (*) [5])jagged_arr[4]
$4 = {5, 5, 5, 5, 5}
what if first dimension (num of “rows”) of jagged array isn’t known?
int **make_jagged_arr(int nrows, const int *dims) {
    int i, j;
    int **jarr = malloc(sizeof(int *) * nrows);
    for (i=0; i<nrows; i++) {
        jarr[i] = malloc(sizeof(int) * dims[i]);
    }
    return jarr;
}

void free_jagged_arr(int **jarr, int nrows) {
    int i;
    for (i=0; i<nrows; i++)
        free(jarr[i]);
    free(jarr);
}

int main() {
    int **jarr = make_jagged_arr(5, (int [5]){3, 4, 2, 1, 8});

    /* use jarr ... */

    free_jagged_arr(jarr, 5);
}
golden rule of memory management:

for every malloc, you must have a corresponding free!
very handy tool for detecting/debugging memory leaks: **valgrind**
int **make_jagged_arr(int nrows, const int *dims) {
    ... 
}

void free_jagged_arr(int **jarr, int nrows) {
    int i;
    for (i=0; i<nrows; i++)
        free(jarr[i]);
    /* free(jarr); */
}

int main() {
    int **jarr = make_jagged_arr(5, (int [5]){3, 4, 2, 1, 8});
    free_jagged_arr(jarr, 5);
}

$ valgrind ./a.out
==23535== HEAP SUMMARY:
==23535==     in use at exit: 40 bytes in 1 blocks
==23535==     total heap usage: 6 allocs, 5 frees, 112 bytes allocated
==23535==
==23535== LEAK SUMMARY:
==23535==     definitely lost: 40 bytes in 1 blocks
==23535==     indirectly lost: 0 bytes in 0 blocks
==23535==      possibly lost: 0 bytes in 0 blocks
==23535==       still reachable: 0 bytes in 0 blocks
==23535==         suppressed: 0 bytes in 0 blocks
int **make_jagged_arr(int nrows, const int *dims) {
    ... }

void free_jagged_arr(int **jarr, int nrows) {
    int i;
    /* for (i=0; i<nrows; i++)
        free(jarr[i]); */
    free(jarr);
}

int main() {
    int **jarr = make_jagged_arr(5, (int [5]){3, 4, 2, 1, 8});
    free_jagged_arr(jarr, 5);
}

$ valgrind ./a.out
==24106==  HEAP SUMMARY:
==24106==    in use at exit: 72 bytes in 5 blocks
==24106==  total heap usage: 6 allocs, 1 frees, 112 bytes allocated
==24106==
==24106==  LEAK SUMMARY:
==24106==    definitely lost: 72 bytes in 5 blocks
==24106==    indirectly lost: 0 bytes in 0 blocks
==24106==    possibly lost: 0 bytes in 0 blocks
==24106==    still reachable: 0 bytes in 0 blocks
==24106==    suppressed: 0 bytes in 0 blocks
int **make_jagged_arr(int nrows, const int *dims) {
    ...
}

void free_jagged_arr(int **jarr, int nrows) {
    int i;
    free(jarr);
    for (i=0; i<nrows; i++)
        free(jarr[i]);
}

int main() {
    int **jarr = make_jagged_arr(5, (int [5]){3, 4, 2, 1, 8});
    free_jagged_arr(jarr, 5);
}

$ valgrind ./a.out
==25084== 5 errors in context 1 of 1:
==25084== Invalid read of size 8
==25084==    at 0x4005AA: free_jagged_arr (demo.c:19)
==25084==    by 0x400613: main (demo.c:26)
==25084==  Address 0x4c29040 is 0 bytes inside a block of size 40 free'd
==25084==    at 0x4A0595D: free (vg_replace_malloc.c:366)
==25084==    by 0x400593: free_jagged_arr (demo.c:17)
==25084==    by 0x400613: main (demo.c:26)

==25084== HEAP SUMMARY:
==25084==     in use at exit: 0 bytes in 0 blocks
==25084==   total heap usage: 6 allocs, 6 frees, 112 bytes allocated
==25084== All heap blocks were freed -- no leaks are possible
§Composite Data Types
≈ objects in OOP
C structs create user defined types, based on primitives (and/or other UDTs)
/* type definition */
struct point {
    int x;
    int y;
}; /* the end ';' is required */

/* point declaration (& alloc!) */
struct point pt;

/* pointer to a point */
struct point *pp;

/* combined definition & decls */
struct point {
    int x;
    int y;
} pt, *pp;
component access: dot ('.') operator

```c
struct point {
    int x;
    int y;
} pt, *pp;

int main() {
    pt.x = 10;
    pt.y = -5;

    struct point pt2 = { .x = 8, .y = 13 }; /* decl & init */

    pp = &pt;

    (*pp).x = 351; /* comp. access via pointer */

    ...
}
```
`.(*pp).x = 351;`  

`.x` has higher precedence than `*`.

```
$ gcc point.c
... error: request for member 'x' in something not a structure or union
```
But (*pp).x is painful

So we have the ‘->’ operator
- component access via pointer

```
struct point {
    int x;
    int y;
} pt, *pp;

int main() {
    pp = &pt;
    pp->x = 10;
    pp->y = -5;

    ...
}
```
/* Dynamically allocating structs: */

struct point *parr1 = malloc(N * sizeof(struct point));
for (i=0; i<N; i++) {
    parr1[i].x = parr1[i].y = 0;
}

/* or, equivalently, with calloc (which zero-inits) */
struct point *parr2 = calloc(N, sizeof(struct point));

/* do stuff with parr1, parr2 ... */
free(parr1);
free(parr2);
In C all args are *pass-by-value*!

```c
void foo(struct point pt) {
    pt.x = pt.y = 10;
}

int main() {
    struct point mypt = { .x = 5, .y = 15 };
    foo(mypt);
    printf("(%d, %d)\n", mypt.x, mypt.y);
    return 0;
}
```

(5, 15)
/* self-referential struct */
struct ll_node {
    void *val;
    struct ll_node next;
};

$ gcc ll.c
ll.c:4: error: field ‘next’ has incomplete type

problem: compiler can’t compute size of next — depends on size of ll_node, which depends on size of next, etc.
/* self-referential struct */
struct ll_node {
    void *val;
    struct ll_node *next; /* need a pointer! */
};

struct ll_node *make_node(void *val, struct ll_node *next) {
    struct ll_node *n = malloc(sizeof(struct ll_node));
    n->val = val;
    n->next = next;
    return n;
}

void free_llist(struct ll_node *head) {
    struct ll_node *p=head, *q;
    while (p) {
        q = p->next;
        free(p);
        p = q;
    }
}
int main() {
    struct ll_node *head = make_node("list!", NULL);
    head = make_node("linked", head);
    head = make_node("a", head);
    head = make_node("I'm", head);

    struct ll_node *p;
    for (p=head; p; p=p->next) {
        printf("%s ", (char *)p->val);
    }

    free_llist(head);
    return 0;
}

I'm a linked list!
§ Function pointers
motivation: functions as values, and higher order functions
**square** x = x*x

> **square** 5
25

> **map square** [1,2,3]
[1,4,9]

> **map \(\mu x \rightarrow x^2\)** [1,2,3]
[1,4,9]

> **map \(^2\)** [1,2,3]
[1,4,9]
compose \( f \circ g = \lambda x \rightarrow f (g \ x) \)

\( \text{even} = \text{compose} \ (==0) \ (\mod\ 2) \)

\[
\begin{align*}
> & \quad \text{even 4} \\
& \quad \text{True}
\end{align*}
\]

\[
\begin{align*}
> & \quad \text{map even [1..5]} \\
& \quad [\text{False}, \text{True}, \text{False}, \text{True}, \text{False}]
\end{align*}
\]
can’t quite do this in C …
but we can kinda fake it with pointers!
```c
int square(int x) {
    return x * x;
}

int cube(int x) {
    return x * x * x;
}

int main() {
    int (*f)(int) = square;
    printf("%d\n", (*f)(10));

    f = cube;
    printf("%d\n", (*f)(10));
    return 0;
}
```

```
100
1000
```
int (*f)(int) ... @#&%*!!!
“Spiral Rule”

http://c-faq.com/decl/spiral.anderson.html

1. Starting with the unknown element, move in a spiral/clockwise direction; when encountering the following elements replace them with the corresponding English statements:

   - [] → array of...

   - (t1, t2) → function with args of type t1, t2 returning ...

   - * → pointer(s) to ...

2. Keep doing this in a spiral/clockwise direction until all tokens have been covered.

3. Always resolve anything in parentheses first!
\texttt{int f}
int foo[100]
int *foo[100][20]
int (*foo)[100]
int foo(int, int)
int (*foo)(int * )
char *(*foo[100])(char *)
cdecl.org
```c
int square(int x) {
    return x * x;
}

void map(int (*f)(int), int *arr, int n) {
    int i;
    for (i=0; i<n; i++) {
        arr[i] = (*f)(arr[i]);
    }
}

int main() {
    int i, arr[] = {1, 2, 3, 4, 5};
    map(square, arr, 5);
    for (i=0; i<5; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

int even_p(int n) {
    return n % 2 == 0;
}

int sum_if(int (*pred)(int), int *arr, int n) {
    int i, sum = 0;
    for (i=0; i<n; i++) {
        if ((*pred)(arr[i]))
            sum += arr[i];
    }
    return sum;
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    printf("%d\n", sum_if(even_p, arr, 5));
    return 0;
}
```c
#define NUM_SCREENS 3
define NUM_KEYS 12

int kfn_9(int);
int kfn_8(int);
int kfn_7(int);
...
int kfn_menu(int);
int kfn_sel(int);
int kfn_up(int);
int kfn_down(int);
...

int process_key(int screen, int key, int duration) {
    static int (*kfn_tab[NUM_SCREENS][NUM_KEYS])(int) = {
        { kfn_9, kfn_8, kfn_7, kfn_6, ... },
        { kfn_menu, kfn_sel, kfn_dial, ... },
        { kfn_up, kfn_down, kfn_left, ... }
    };
    return (*kfn_tab[screen][key])(duration);
}
```
§Addendum: typedef
declarations can get a little … wordy

-unsigned long int size;
-void (*fn)(int);
-struct llnode *lst;
typedef lets us create an alias for an existing type
syntax:

```
typedef oldtype newtype;
```

- looks like a regular variable declaration to the right of `typedef` keyword
`/* declare `int_t` as an alias for `int` */
typedef int int_t;

main() {
    int i;
    int_t j;
    i = j = 10;
    printf("%d, %d, %lu, %lu",
            i, j, sizeof(int), sizeof(int_t));
}
/* declare `intp_t` as an alias for `int *` */
typedef int *intp_t;

main() {
    int i;
    intp_t p;
    p = &i;
}
/* define both preceding aliases */

typedef int int_t, *intp_t;

main() {
    int_t i;
    intp_t p;
    p = &i;
}
/* common integer aliases (see stdint.h) */

/* used to store "sizes" and "offsets" */
typedef unsigned long int size_t;
typedef long int off_t;

/* for small numbers; 8 bits only */
typedef signed char int8_t;
typedef unsigned char uint8_t;

/* for large numbers; 64 bits */
typedef long int int64_t;
typedef unsigned long int uint64_t;
/* fn pointer typedef */
typedef int (*handler_t)(int);

int kfn_menu(int duration) { /* ... */ }

main() {
    handler_t fp = kfn_menu;
    int ret = (*fp)(0);
    ...
}
/* linked-list type aliases */
typedef struct ll_node node, *node_p, *list;

struct ll_node {
    void *val;
    node_p next;
};

main() {
    node n = { .val = NULL, .next = NULL };
    list l = &n;
}