C Primer

CS 351: Systems Programming

Michael Saelee <lee@iit.edu>

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY

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`)
  - UNIX man pages (`kernel.org/doc/man-pages/`)

>man strlen

```
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

C	Java
Procedural	Object-oriented
Source-level portability	Compiled-code portability
Manual memory management	Garbage collected
Pointers reference addresses	Opaque memory references
Manual error code checking	Exception handling
Manual namespace partitioning	Namespaces with packages
Lower level libraries	High level class libraries

Language Philosophies

C: "Make it efficient and simple, and let the programmer do whatever she wants"

Java: "Make it portable, provide a huge class library, and try to protect the programmer from doing stupid things."

A language that doesn't have everything *is actually* easier to program in *than some that do.*

- Dennis Ritchie

C standard is established by the
American **N**ational **S**tandards **I**nstitute;

current spec: ANSI C11

What's in C11?

- Language syntax & semantics
- Runtime features & behavior
  - Type info, memory model, etc.
- (Very limited) standard library API

Why is this interesting?

- Because of what C11 *leaves out*!
  - e.g., we're used to vast standard libraries & platform independence
    - … not with C!
  - a lot of decisions left to the compiler

§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…

```c
/* 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

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`
- "Infinitely abusable" `goto`

Variables

- Must declare before use
- Declaration implicitly **allocates** storage for underlying data
  - Note: not true in Java!
- Scope: global, local, static

Functions

- C's *top-level* modules
- Procedural language vs. OO: no classes!

```java
public class Demo {
  public static void main (String[] args) {
     System.out.printf("Hello world!");
  }
}
```

VS.

```c
int main (int argc, char *argv[]) {
    printf("Hello world!");
    return 0;
}
```

Declaration vs. *Definition*

- (Distinction doesn't exist in Java)
- *Declaration* (aka *prototype*): arg & ret type
- *Definition*: function body
- At compile-time, function call only requires declaration

Important: many declarations are ok, but only a *single* definition!

Declarations reside in *header* (.h) files,
Definitions reside in *source* (.c) files

(Suggestions, not really requirements)

memlib.h

```
void mem_init(void);
void mem_deinit(void);
void *mem_sbrk(int incr);
void mem_reset_brk(void);
void *mem_heap_lo(void);
void *mem_heap_hi(void);
size_t mem_heapsize(void);
size_t mem_pagesize(void);
```

"API"

memlib.c

```
void mem_init(void)
{
    /* allocate the storage we will use to model the available VM */
    if ((mem_start_brk = (char *)malloc(MAX_HEAP)) == NULL) {
        fprintf(stderr, "mem_init_vm: malloc error\n");
        exit(1);
    }

    mem_max_addr = mem_start_brk + MAX_HEAP;  /* max legal heap address */
    mem_brk = mem_start_brk;                  /* heap is empty initially */
}

/*
 * mem_deinit - free the storage used by the memory system model
 */
void mem_deinit(void)
{
    free(mem_start_brk);
}

/*
 * mem_reset_brk - reset the simulated brk pointer to make an empty heap
 */
void mem_reset_brk()
{
    mem_brk = mem_start_brk;
}
...
```

memlib.h

```
void mem_init(void);
void mem_deinit(void);
void *mem_sbrk(int incr);
void mem_reset_brk(void);
void *mem_heap_lo(void);
void *mem_heap_hi(void);
size_t mem_heapsize(void);
size_t mem_pagesize(void);
```

"API"

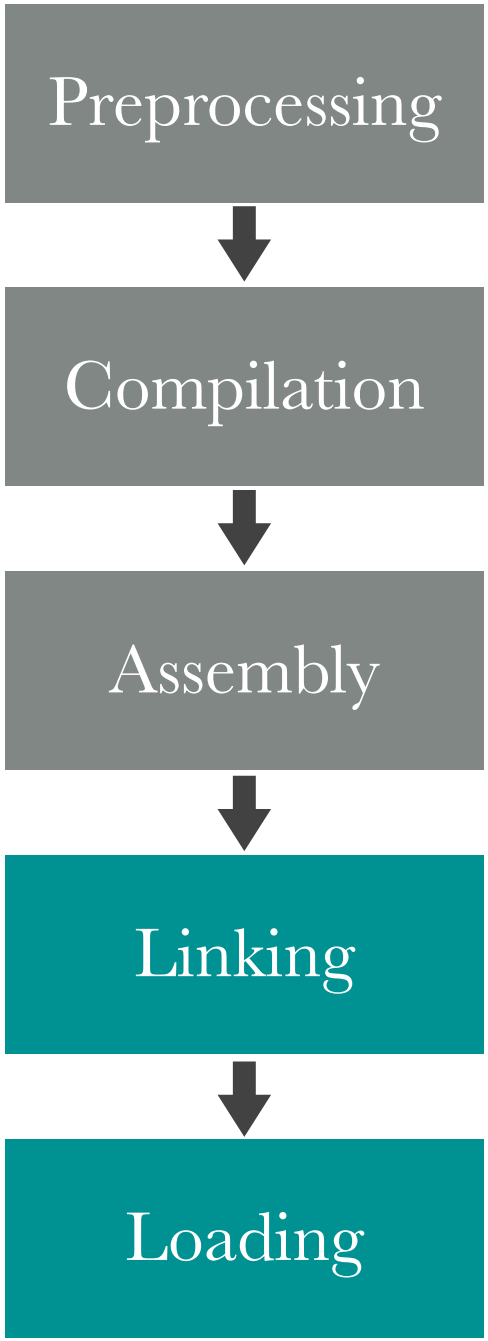
main.c

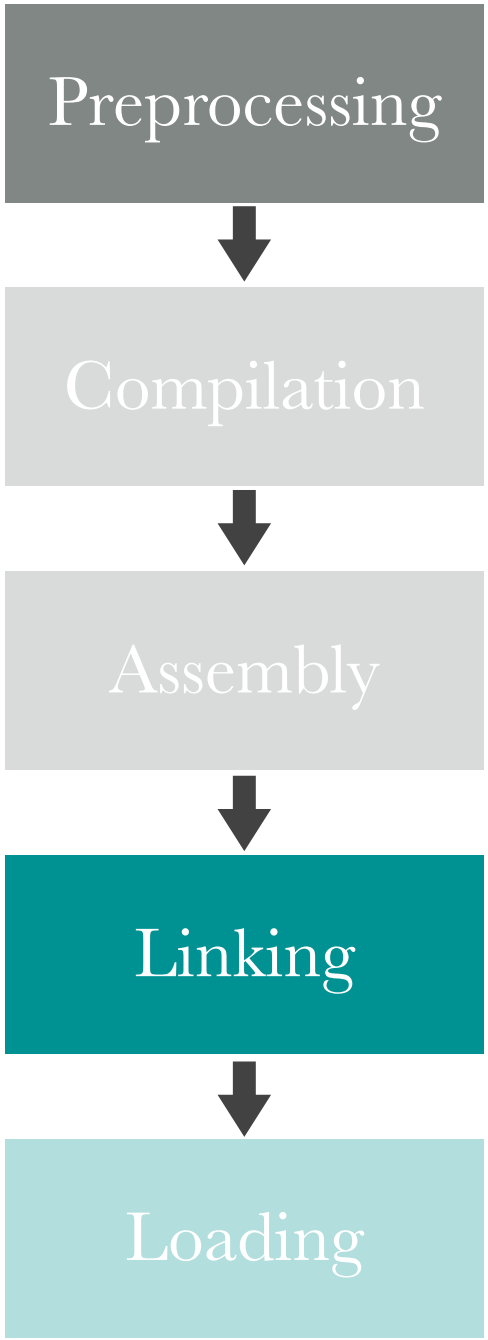
```
#include "memlib.h"

int main(int argc, char **argv)
{
    /* Initialize the simulated memory system in memlib.c */
    mem_init();

    /* Evaluate student's mm malloc package using the K-best scheme */
    for (i=0; i < num_tracefiles; i++) {
        ...
    }
    ...
}
```

§Compilation

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

```
$ gcc -o demo hello.c
$ ./demo
Hello world!
$
```


Preprocessing
Compilation
Assembly
Linking
Loading


Preprocessing
Compilation
Assembly
Linking
Loading

"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;
}
```

```c
#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*)

"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?

```
 1  int glob_i = 0;
 2
 3  int main() {
 4      int i  = 10;
 5      glob_i = 10;
 6      foo();
 7      printf("%d, %d\n", i, glob_i);
 8      return 0;
 9  }
10
11  void foo() {
12      i++;
13      glob_i++;
14  }
```

```
$ 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
```

sum.c

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

main.c

```c
#include <stdio.h>

int I = 10;

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

```
$ 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'
```

sum.c

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

main.c

```c
#include <stdio.h>

int sumWithI(int, int);

int I = 10;

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*

sum.c

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

main.c

```c
#include <stdio.h>

int sumWithI(int, int);

int I = 10;

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

```
$ gcc -Wall -o demo sum.c main.c
$ ./demo
13
```

… and now global variables are visible from *everywhere*.

Good/Bad?

`static` keyword lets us
limit the *visibility* of things

sum.c

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

main.c

```c
#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
```

sum.c

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

main.c

```c
#include <stdio.h>

int sumWithI(int, int);

int I = 10;

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

```
$ 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)

sum.c

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

main.c

```c
#include <stdio.h>

int sumWithI(int, int);

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*

what's a *memory address*?

- an address that can refer to a datum in memory
- width determined by machine *word size*
  - e.g., 32-bit machine → 32-bit address

given address size w, range = 0 to 2^w-1

e.g., for word size = 32, the following are valid memory addresses:

```
-0

-100

-0xABCD1234

-0xFFFFFFFF
```

i.e., an address is *just a number*

Q: by examining a variable's contents, can we tell if the variable is a pointer?

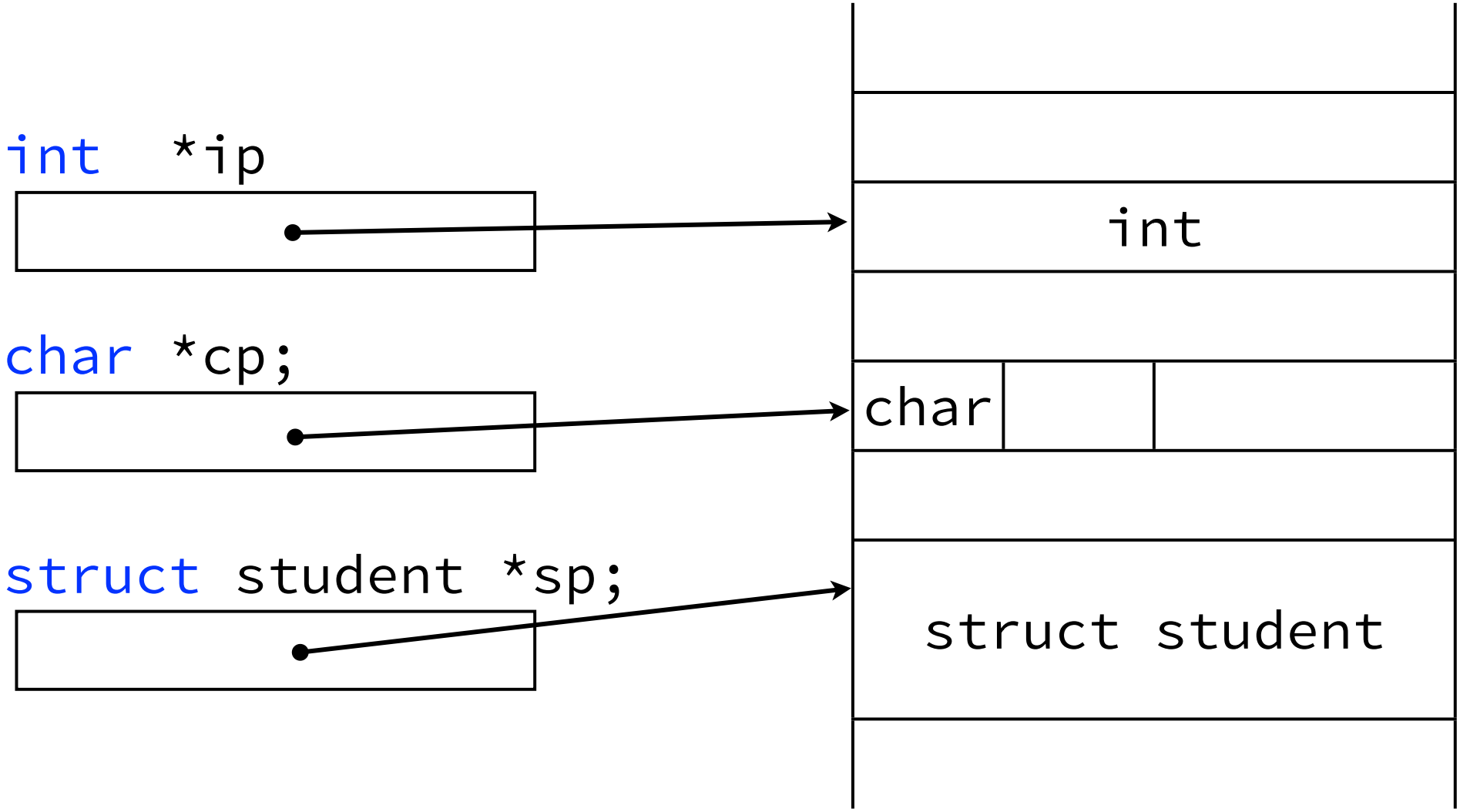
e.g., `0x0040B100`

No!

- a pointer is designated by its *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`


int *ip
char *cp;
struct student *sp;
int
char
struct student

Important pointer-related operators:

& : address-of

* : dereference (*not the same as the * used for declarations!!!*)

Class	Operator	Associativity
-	`() [] . ->` `expr++ expr--`	left-to-right
Unary	`& * + - ! ~` `++expr --expr`	right-to-left
Binary	arithmetic (incl. `*`), relational, logical	left-to-right
Ternary	`?:` (conditional)	right-to-left
Assignment	`= += -= ...`	left-to-right

Operator precedence

```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*/
```

```
1   int main() {
2       int i, j;
3       int *p, *q;
4       i = 10;
5       p = j;
6
7       return 0;
8   }
```

```
$ gcc pointers.c
pointers.c: In function ‘main’:
pointers.c:5: warning: assignment makes pointer from integer without a cast
```

```
1   int main() {
2       int i, j, *p, *q;
3
4       i = 10;
5       p = &j;
6       q = *p;
7
8       return 0;
9   }
```

```
$ gcc pointers.c
pointers.c: In function ‘main’:
pointers.c:6: warning: assignment makes pointer from integer without a cast
```

```
1   int main() {
2       int i, j, *p, *q;
3
4       i = 10;
5       p = &j;
6       q = &p;
7
8       return 0;
9   }
```

```
$ gcc pointers.c
pointers.c: In function ‘main’:
pointers.c:6: warning: assignment from incompatible pointer type
```

```
 1  int main() {
 2      int i, j, *p, *q;
 3
 4      i = 10;
 5      p = &j;
 6      q = p;
 7      *q = *i;
 8
 9      return 0;
10  }
```

```
$ gcc pointers.c
pointers.c: In function ‘main’:
pointers.c:7: error: invalid type argument of ‘unary *’
```

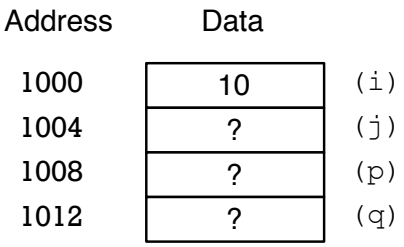
```
1   int main() {
2       int i, j, *p, *q;
3
4       i = 10;
5       p = &j;
6       q = p;
7       *q = i;
8       j = q;
9
10      return 0;
11  }
```

```
$ gcc pointers.c
pointers.c: In function ‘main’:
pointers.c:8: warning: assignment makes integer from pointer without a cast
```

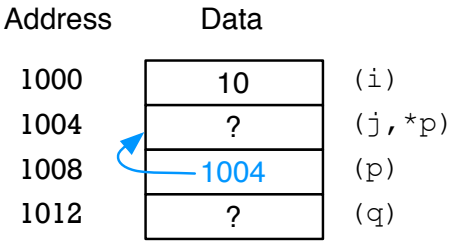
```c
1   int main() {
2       int i, j, *p, *q;
3
4       i = 10;
5       p = &j;
6       q = p;
7       *q = i;
8       *p = (*q) * 2;
9       printf("i=%d, j=%d, *p=%d, *q=%d\n", i, j, *p, *q);
10      return 0;
11  }
```

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

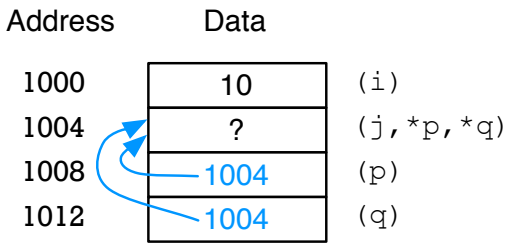
```
int i, j, *p, *q;
i = 10;
```



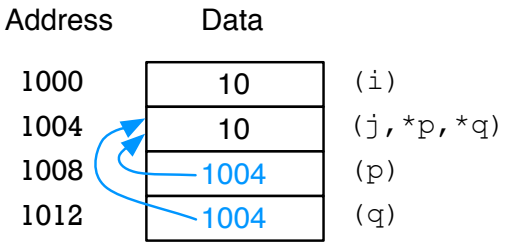

```
p = &j;
```



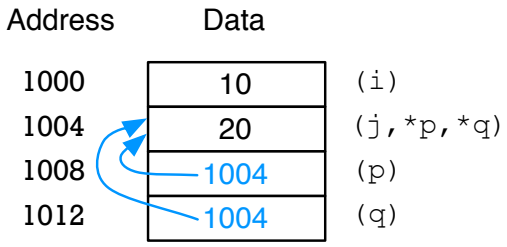

```
q = p;
```




```
*q = i;
```




```
*p = (*q) * 2;
```



why have pointers?

```
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;
}
```

in Java ...

```
public class Container {
    int x, y;

    public static void swap (Container c) {
        int tmp = c.x;
        c.x = c.y;
        c.y = tmp;
    }

    public static void main (String[] args) {
        Container c = new Container();
        c.x = 5;
        c.y = 10;
        swap(c);
        System.out.printf("c.x=%d; c.y=%d\n", c.x, c.y);
    }
}
```

```
$ javac Container.java
$ java Container
c.x=10; c.y=5
```

Java is inconsistent in how arguments are passed to methods

- primitives are passed *by-value* (copy)
- objects are not copied, instead, we pass references to objects

in C, args are *always* passed by-value

i.e., we always pass *copies* of args

```python
# Python example: returning multiple values

from math import sqrt

def quad_roots(a, b, c):
    discr = b*b - 4*a*c
    if discr < 0:
        return (None, None) # no real roots
    r1 = (-b + sqrt(discr)) / 2*a
    r2 = (-b - sqrt(discr)) / 2*a
    return (r1, r2)

x0, x1 = quad_roots(1, 3, 2)
print("root 1 =", x0, "; root 2 = ", x1)
```

```
$ python qroots.py
root 1 = -1.0 , root 2 =  -2.0
```

```c
/* now in C */

#include <math.h>

double quad_roots(double a, double b, double c) {
    double discr = b*b - 4*a*c;
    if (discr < 0)
        return /* ?!? */;
    double r1 = (-b + sqrt(discr)) / 2*a;
    double r2 = (-b - sqrt(discr)) / 2*a;
    return /* ?!? */;
}
```

```c
/* now in C */

#include <math.h>

int quad_roots(double a, double b, double c,
               double *pr1, double *pr2) {
    double discr = b*b - 4*a*c;
    if (discr < 0)
        return 0; /* no real roots */
    *pr1 = (-b + sqrt(discr)) / 2*a;
    *pr2 = (-b - sqrt(discr)) / 2*a;
    return 1;
}

int main() {
    double x0, x1;
    if (quad_roots(1, 3, 2, &x0, &x1))
        printf("root 1 = %0.1f; root 2 = %0.1f\n", x0, x1);
    return 0;
}
```

```
$ gcc -o qroots qroots.c
$ ./qroots
root 1 = -1.0; root 2 =  -2.0
```

```c
/* now in C */

#include <math.h>

int quad_roots(double a, double *b_r1, double *c_r2) {
    double b = *b_r1, c = *c_r2;
    double discr = b*b - 4*a*c;
    if (discr < 0)
        return 0; /* no real roots */
    *b_r1 = (-b + sqrt(discr)) / 2*a;
    *c_r2 = (-b - sqrt(discr)) / 2*a;
    return 1;
}

int main() {
    double x0=3, x1=2; /* x0 & x1 are value-return args */
    if (quad_roots(1, &x0, &x1))
        printf("root 1 = %0.1f; root 2 = %0.1f\n", x0, x1);
    return 0;
}
```

```
$ gcc -o qroots qroots.c
$ ./qroots
root 1 = -1.0; root 2 =  -2.0
```

(now forget you ever saw that last slide)

pointers enable *action at a distance*

```
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

note: moral is not “don’t use pointers”;
it’s “don’t use pointers unthinkingly”

back to swap

```
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?

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

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

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

problem: can't change value of p or q inside swap … (same problem as before)

solution: pass *pointers to* p and q

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

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

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

```
$ 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
```

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

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

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

`(int **)` declares a *pointer to a pointer* to an `int`

```c
/* swaps values in `int` vars */
void swap(int *p, int *q) {
    int tmp = *p;
    *p = *q;
    *q = tmp;
}

/* swaps values in `int *` vars */
void swapp(int **p, int **q) {
    int *tmp = *p;
    *p = *q;
    *q = tmp;
}
```

can we write a *generic* swap function in C?

kind of … but not that easily! (see later)

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 `0` as in *pointer context*
  - for convenience, `#define`’d as `NULL`

"Null" pointers

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

    ...

    if (p) {
        /* (likely) safe to deref p */
    }
}
```

“Null” pointers

```c
void foo(char *first, ...);

int main() {
    /* explicit typecast to establish pointer context */
    foo("var", "args", "terminated", "with", (char *)0);
    return 0;
}
```

The other "null"

- ASCII "null zero" — used to terminate strings (as last char)
  - Written `'\0'`
  - Numerical value = `0`
- Don't get confused!

§Arrays

contiguous, indexed region of memory

Declaration: `type arr_name[size]`

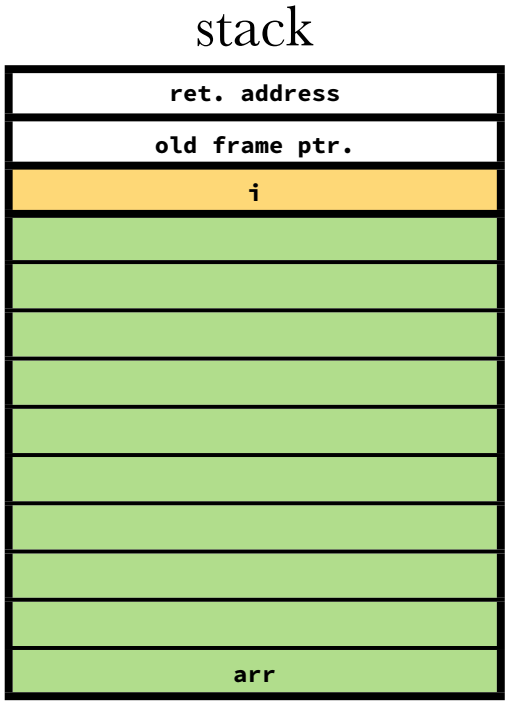
- remember, declaration also allocates storage!

```
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*

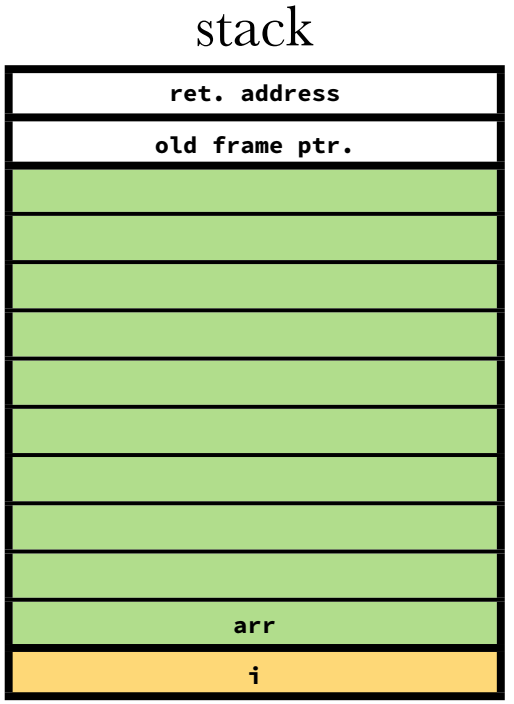
i.e., ***no*** *implicit size*, ***no*** *bounds checking*

```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)
```

```c
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

```
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! */
```

```c
int main() {
    int i, j, k, *p, mat[5][5];

    k=0;
    for (i=0; i<5; i++) {
        for (j=0; j<5; j++) {
            mat[i][j] = k++;
        }
    }

    p = (int *)mat; /* `p` is a 1-D view of a 2-D array */

    for (i=0; i<25; i++) {
        printf("%d ", p[i]);
    }
}
```

```
$ ./a.out
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
```

```c
int main() {
    int i, j;
    int mat[25];
    for (i=0; i<25; i++) {
        mat[i] = i;
    }
    for (i=0; i<5; i++) {
        for (j=0; j<5; j++) {
            /* treat `mat` as a multi-dim array of 5x5 ints */
            printf("%2d ", ((int (*)[5])mat)[i][j]);
        }
        printf("\n");
    }
}
```

```
$ $ ./a.out
 0  1  2  3  4
 5  6  7  8  9
10 11 12 13 14
15 16 17 18 19
20 21 22 23 24
```

§Pointer Arithmetic

follows naturally from allowing array subscript notation on pointers

```
int *pa;
int arr[100];

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 */
```

```
char *pa;
int arr[100];
arr[10] = 0xDEADBEEF;

pa = (char *)arr; /* explicit typecast */

pa[10] = 0;       /* set tenth char */

*(pa + 10) = 0;   /* set tenth char */

printf("%X\n", arr[10]);
```

```
$ ./a.out
DEADBEEF
```

```
char *pa;
int arr[100], offset;
arr[10] = 0xDEADBEEF;

pa = (char *)arr;

offset = 10 * sizeof (int);

*(pa + offset) = 0;

printf("%X\n", arr[10]);
```

```
$ ./a.out
DEADBE00
```

```
char *pa;
int arr[100], offset;
arr[10] = 0xDEADBEEF;

pa = (char *)arr;

offset = 10 * sizeof (int);

*(int *)(pa + offset) = 0;

printf("%X\n", arr[10]);
```

```
$ ./a.out
0
```

`sizeof`: an operator to get the size *in bytes*

- can be applied to a datum or type

```c
int i, *p, arr[10];
char c, *str = "hello";
double d;

printf("sizeof i          = %lu\n", sizeof i);
printf("sizeof p          = %lu\n", sizeof p);
printf("sizeof arr        = %lu\n", sizeof arr);
printf("sizeof c          = %lu\n", sizeof c);
printf("sizeof str        = %lu\n", sizeof str);
printf("sizeof d          = %lu\n", sizeof d);
```

```
sizeof i          = 4
sizeof p          = 8
sizeof arr        = 40
sizeof c          = 1
sizeof str        = 8
sizeof d          = 8
```

```c
printf("sizeof (int)      = %lu\n", sizeof (int));
printf("sizeof (int *)    = %lu\n", sizeof (int *));
printf("sizeof (int [10]) = %lu\n", sizeof (int [10]));
printf("sizeof (char)     = %lu\n", sizeof (char));
printf("sizeof (char *)   = %lu\n", sizeof (char *));
printf("sizeof (double)   = %lu\n", sizeof (double));
```

```
sizeof (int)      = 4
sizeof (int *)    = 8
sizeof (int [10]) = 40
sizeof (char)     = 1
sizeof (char *)   = 8
sizeof (double)   = 8
```

```c
/* Beware! */

void foo(char *p) {
    printf("In foo,  sizeof  p  = %lu\n", sizeof 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;
}
```

```
In main, sizeof str = 80
In foo,  sizeof  p  = 8
In foo,  sizeof *p  = 1
```

when “passed” as arguments,
arrays degenerate into pointers

i.e., no aggregate size information!

```c
/* 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;
}
```

```
In main, sizeof str = 80
In foo,  sizeof p   = 8
```

more pointer arithmetic:

- +/− by *x*: move forward/back *x elements*
- </≤/≥/>: which element is first?
- difference (*p1*−*p2*): # elements apart

```
double arr[] = { 12.5, 1.0, 0.0, 5.2, 3.5 },
       *p = &arr[0],
       *q = &arr[2];

printf("%d\n",   p < q);   /* => 1    */
printf("%ld\n",  q - p);   /* => 2    */
q += 2;
printf("%.1f\n", *q);      /* => 3.5  */
```

strings are just `0` terminated char arrays

```
char str[]      = "hello!";
char *p         = "hi";
char tarr[][5] = {"max", "of", "four"};
char *sarr[]    = {"variable", "length", "strings"};
```

```c
/* 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);
```

```c
/* Beware: */

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

```
$ ./a.out
[1]    10522 bus error  ./a.out
```

```c
/* 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
```

e.g., `strcpy`: copy chars from source to dest array (including terminating `0`)

```
void strcpy(char dst[], char src[]) {
    int i=0;
    while ((dst[i] = src[i]) != '\0')
        i++;
}
```

```
void strcpy(char *dst, char *src) {
    while ((*dst = *src) != '\0') {
        dst++;
        src++;
    }
}
```

```
void strcpy(char *dst, char *src) {
    while ((*dst++ = *src++) != '\0') ;
}
```

```
void strcpy(char *dst, char *src) {
    while (*dst++ = *src++) ;
}
```

```c
/* actually ... */
char *strcpy(char *restrict dst, const char *restrict src) {
    char *ret = dst;
    while (*dst++ = *src++) ;
    return ret;
}
```

miscellaneous modifiers:

- *const*; e.g., `const int *p`
  - data pointed to won't be modified
  - not the same as `int *const p`
- *restrict*; e.g., `int *restrict p`
  - no other pointer refers to this data

<string.h>

strcpy, strcat
strcmp, strlen, strchr
memcpy, memmove, ...

e.g., `clone`: allocate and return a *copy* of a given string

```
char *clone(const char *str) {
    char buf[strlen(str) + 1]; /* not pretty */
    strcpy(buf, str);
    return buf;  /* NO! */
}
```

```
char *clone(const char *str) {
    static char buf[BUF_LEN]; /* global buf */
    static char *p = buf;     /* track position */
    int nbytes = strlen(str) + 1;
    char *rp = p;

    if (p + nbytes > &buf[BUF_LEN])
        return 0;             /* fail if buf full */
    strcpy(p, str);
    p += nbytes;

    return rp;
}
```

§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 → memory **leak**

vs., e.g., Java, which has *implicit* memory management via *garbage collection*

- allocate (via `new`) & forget!

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 `(void *)` to/from
any other pointer *will never produce warnings*

… Hurrah! (but *dangerous*)

```
void *malloc(size_t size);

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

void  free(void *ptr);
```

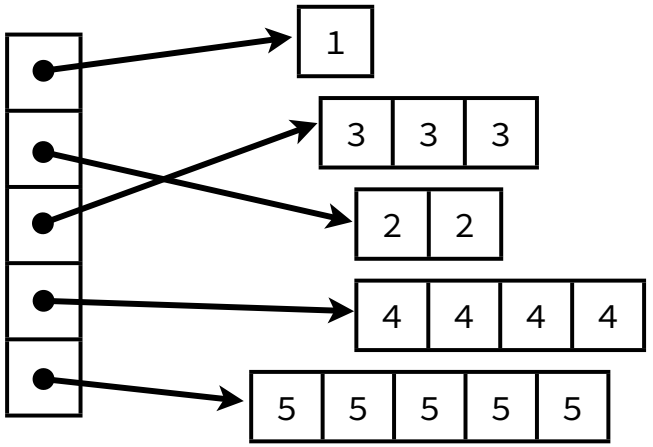
all `sizes` are in bytes;

all `ptrs` are from previous malloc requests

```c
/* clone into dynamically alloc'd memory */
char *clone(const char *str) {
    char *nstr = malloc(strlen(str) + 1);
    strcpy(nstr, str);
    return nstr; /* someone else must free this! */
}
```

```c
/* 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]);
}
```



```
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?

```c
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]);
        for (j=0; j<dims[i]; j++)
            jarr[i][j] = 0xDEADBEEF; /* for testing */
    }
    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);
}
```

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

    /* use jarr ... */

    free_jagged_arr(jarr, 5);
}
```

```
(gdb) break main
Breakpoint 1 at 0x100000e44: file demo.c, line 26.
(gdb) run
Starting program: /Users/lee/demo/a.out
Breakpoint 1, main () at syntax.c:26
26          int **jarr = make_jagged_arr(5, (int [5]){3, 4, 2, 1, 8});
(gdb) next
27          free_jagged_arr(jarr, 5);
(gdb) p (int *[5])*jarr
$1 = {0x1001000e0, 0x100103b00, 0x100103b10, 0x100103b20, 0x100103b30}
(gdb) p /x *(int (*) [2])jarr[3]
$2 = {0xdeadbeef, 0x0}
(gdb) p /x *(int (*) [4])jarr[1]
$9 = {0xdeadbeef, 0xdeadbeef, 0xdeadbeef, 0xdeadbeef}
```

```
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);
}
```

golden rule of memory management:

for *every malloc*, you must have a *corresponding free*!

very handy tool for detecting/debugging memory leaks: **valgrind**

```c
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
```

```c
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==
==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)

```c
/* 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;
```

```c
/* combined definition & decls */
struct point {
    int x;
    int y;
} pt, *pp;
```

component access: dot ('.') operator

```
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;   ?=   *pp.x = 351;
```

recall: ‘.’ 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;

    ...
}
```

```c
/* 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 */
struct point *parr2 = calloc(N, sizeof(struct point));


/* do stuff with parr1, parr2 ... */


free(parr1);
free(parr2);
```

`sizeof` works with `struct`s, too, but with sometimes surprising results:

```
struct point {
    int  x;
    int  y;
};

struct foo {
    char name[10];
    int  id;
    char flag;
};
```

```
struct point p;
struct foo f;

printf("point size       = %lu\n", sizeof(struct point));
printf("point comps size = %lu\n", sizeof(p.x)
                                 + sizeof(p.y));
printf("foo size         = %lu\n", sizeof(struct foo));
printf("foo comps size   = %lu\n", sizeof(f.name)
                                 + sizeof(f.id)
                                 + sizeof(f.flag));
```

```
point size       = 8
point comps size = 8
foo size         = 20
foo comps size   = 15
```

and recall, in C *all* args are *pass-by-value*!

```
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)
```

```c
/* self-referential struct */
struct ll_node {
    void *val;
    struct llnode 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.

```c
/* self-referential struct */
struct ll_node {
    void *val;
    struct llnode *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;
    }
}
```

```c
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 (\x -> x*x) [1,2,3]
[1,4,9]

> map (^2) [1,2,3]
[1,4,9]
```

```
compose f g = \x -> f (g x)

even = compose (==0) (`mod` 2)
```

```
> even 4
True

> map even [1..5]
[False,True,False,True,False]
```

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!

```
int f
```

```
int foo[100]
```

```
int *foo[100][20]
```

```
int (*foo)[100]
```

```
int foo(int, int)
```

```
int (*foo)(int *)
```

```
char *(*foo[100])(char *)
```

```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;
}
```

```
1 4 9 16 25
```

```c
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
jmp_buf jbuf;

void handler(int cause) {
    printf("SEGV arrived.\n");
    longjmp(jbuf, 0);
}

int main() {
    int *p;
    signal(SIGSEGV, handler);
    while(1) {
        setjmp(jbuf);
        printf("I probably shouldn't do this, but ...\n");
        p = (int *)random();
        printf("*p = %x\n", *p);
    }
}
```

```
I probably shouldn't do this, but ...
SEGV arrived.
I probably shouldn't do this, but ...
SEGV arrived.
I probably shouldn't do this, but ...
SEGV arrived.
```

```
#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

```c
/* 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));
}
```

```
10, 10, 4, 4
```

```c
/* 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;
```

```c
/* 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 llnode node, *nodep, *list;

struct llnode {
    void *val;
    struct llnode *next;
};

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

```c
/* typedef from anonymous enum and structure */
typedef enum { DIAMOND, CLUB, HEART, SPADE } suit_t;

typedef struct {
    int value;
    suit_t suit;
} card_t, *hand_t;

main() {
    card_t carr[] = {{ .value = 10, .suit = DIAMOND },
                     { .value = 5,  .suit = SPADE   },
                     { .value = 12, .suit = HEART   },
                     { .value = 8,  .suit = DIAMOND },
                     { .value = 10, .suit = SPADE   }};
    hand_t hand = carr;
}
```

</C_Primer>