The C Programming Language

- C is a high-level language — structured
- C is a low-level language — machine access
- C is a small language, extendable with libraries
- C is permissive: assumes you know what you're doing
- Good: efficient, powerful, portable, flexible
- Bad: easy to make errors, obfuscation, little support for modularization

Intro to C

```
#include <stdio.h>

int main() {
  int i;
  extern int gcd(int x, int y);
  for (i = 0; i < 20; i++) {
     printf("gcd of 12 and %d is %d\n",
            i, gcd(12,i));
  }
  return (0);
}
```

The rest of the file

```
int gcd(int x, int y) {
  int t;
  while (y) {
      t = x;
      x = y;
      y = t % y;
  }
  return (x);
}
```

About C

- Similar to Java - Java took best of C
- `#include` - use declarations of functions
- `main()` returns int, the exit status
- Functions must be
  - declared - tells compiler how to use function
  - defined - creates the item
- Declarations must appear before code

Basic Control Structures

- Functions - can omit `extern` declaration
- `for` loop - like Java
  - body is one statement
  - braces `{ }` enclose blocks
  - blocks introduce scope level
  - can't mix declarations and non-declarations
    - `for (int i ...` - illegal in ANSI C

More about C

- Uninitialized variables have no default value!
- No run-time checking!
- No polymorphism (`printf` format strings)
- No objects

Compile: `gcc -Wall -g -o gcd gcd.c`

C data types

- basic types and literals (King: Ch 7)

```
int i = 38;          long el = 38L;
int hex = 0x2a;      int oct = 033;
printf("i = %d, el = %ld, hex = %d, oct = %d\n",
       i, el, hex, oct);
```

```
i = 38, el = 38, hex = 42, oct = 27
```

```
double d1 = 0.3;   double d2 = 3.0;
double d3 = 6.02e23;
printf("d1 = %f, d2 = %f, d3 = %e\n", d1, d2, d3)
```

```
d1 = 0.300000, d2 = 3.000000, d3 = 6.020000e+23
```

C literals and types

Literal	Value	Type
`38`	38	`int`
`38L`	38	`long int`
`0x2a` (hex)	42	`int`
`033` (octal)	27	`int`
`38.0`	38.0	`double`
`38.0f`	38.0	`float`

C data types

- Most things in C are ints:
  - Boolean values are ints
    - 0 means false, nonzero means true
  - characters are ints (ASCII code)
    - `'a'`==97, `'\n'`==10, `'\033'`==033==27
  - enumerations are really ints
- signed vs. unsigned types
- char, int, long, … are just different sizes of integers.

Data Type Conversion

- The expression on the right side is converted to the type of the variable on the left.

```
char c;
int i = c;     /* c is converted to int */
double d = i; /* i is converted to double */
```

- This is no problem as long as the variable’s type is at least as “wide” as the expression.

```
char c = 500; /* compiler warning */
int k = d1;
printf("c = %c, k = %d\n", c, k);
```

```
c =  , k = 0
```

Data Type Capacity

- What happens when the following code is executed?

```
char c = 127;
int d;

printf("c = %d\n", c);
c++;

d = 512 / c;
printf("c = %d, d = %d\n", c, d);
```

Mixed Mode Arithmetic

```
double m = 5/6; /* int / int = int */
printf("Result of 5/6 is %f\n", m);
```

```
Result of 5/6 is 0.000000
```

```
double n = (double)5/6; /* double / int = double */
printf("Result of (double)5/6 is %f\n", n);
```

```
Result of (double)5/6 is 0.833333
```

```
double o = 5.0/6; /* double / int = double */
printf("Result of 5.0/6 is %f\n", o);
```

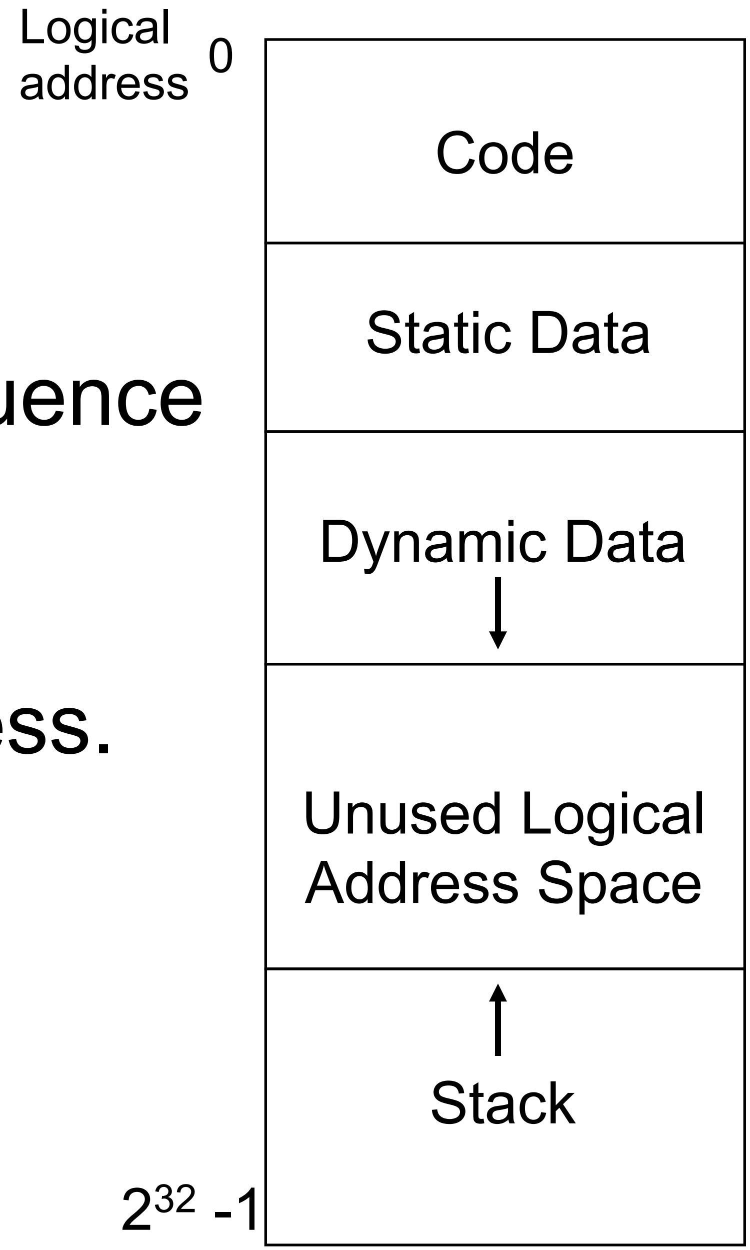
```
Result of 5.0/6 is 0.833333
```

```
int p = 5.0/6; /* double / int = double but then
                  converted to int */
printf("Result of 5.0/6 is %d\n", p);
```

```
Result of 5.0/6 is 0
```

Memory model

- Memory is just a sequence of bytes
- A memory location is identified by an address.



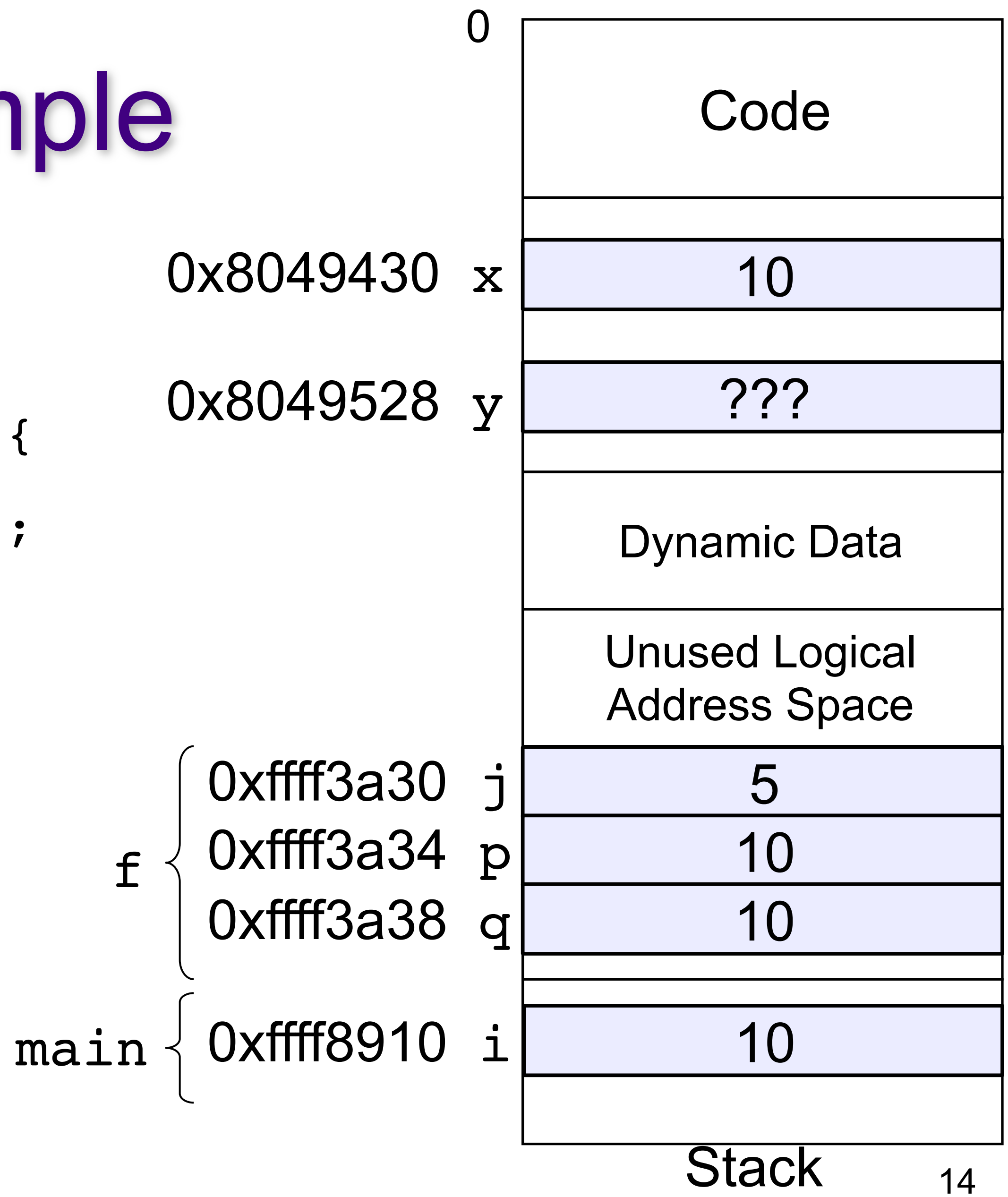
Example

```
int x = 10;
int y;

int f(int p, int q) {
    int j = 5;
    return p * q + j;
}

int main() {

    int i = x;
    y = f(i, i);
    return 0;
}
```

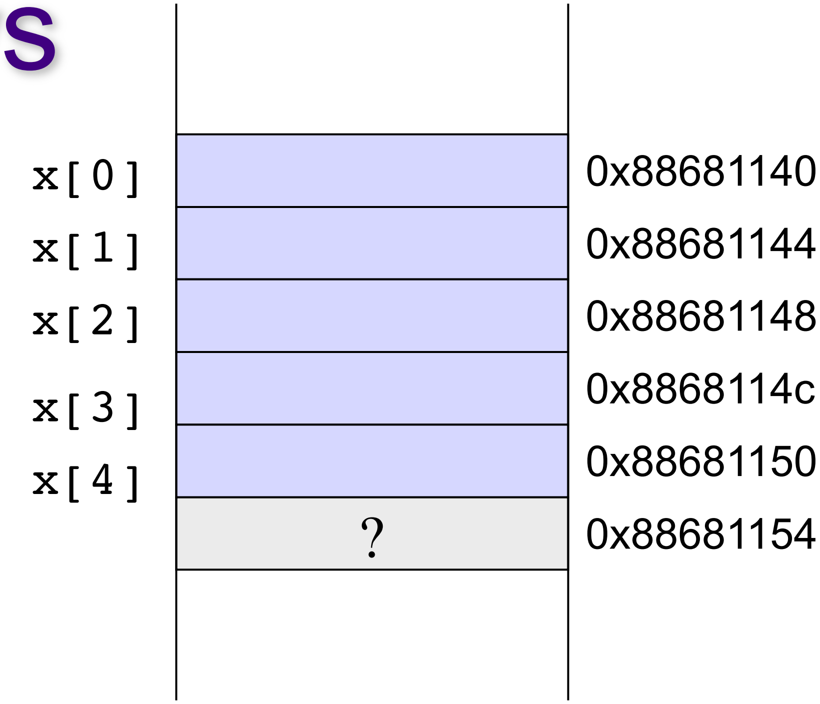


Arrays

- Arrays in C are a contiguous chunk of memory that contain a list of items of the same type.
- If an array of ints contains 10 ints, then the array is 40 bytes. There is nothing extra.
- In particular, the size of the array is not stored with the array. There is *no* runtime checking.

Arrays

```
int x[5];
for (i = 0; i <= 5; i++) {
    x[i] = i*i;
}
```



- No runtime checking of array bounds
- Behaviour of exceeding array bounds is “undefined”
    - → program might appear to work
    - → program might crash
    - → program might do something apparently random

Initializing arrays

The `sizeof` operator is evaluated by the compiler

Declaration/Definition

```
int a[10]; /*declare 'a' as an
                 array of 10 ints*/
sizeof(a) == 10 * sizeof(int) == 40;
```

Static initialization:

```
char letters[4] = {'a', 'q', 'e', 'r'};
```

Initialization loop:

```
for(i = 0; i < N; i++) {
   a[i] = 0;
}
```

Arrays

- Warning: It is the programmer's responsibility to keep track of the size of an array!
- Often define a maximum size.
- Pre-processor directives are used for constants:
  - E. g., `#define MAXSIZE 30`

Pointers

- A pointer is a higher-level version of an address.
- A pointer has type information.

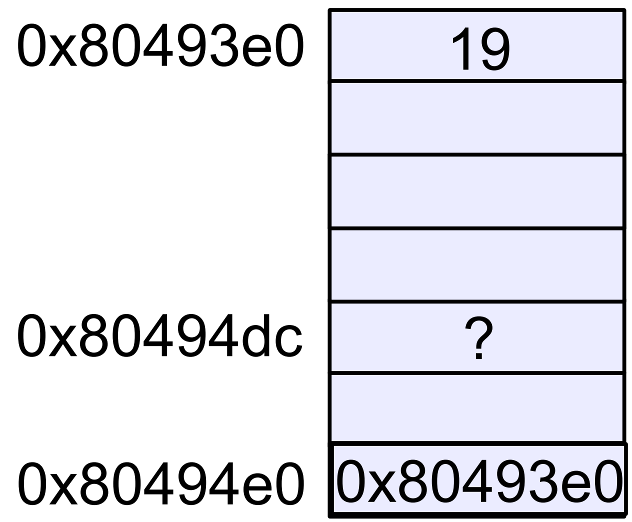
```
int i;
int *p;  /* declare p to point to type int */
*p = i;  /* dereference p – set what p points to*/
p = &i   /* Give p the value of the address of i*/
char *c = p; /* Warning: initialization from
                incompatible pointer type */
```

Important!

- `int *p;`
- Memory is allocated to store the **pointer**
- No memory is allocated to store what the pointer points to!
- Also, `p` is **not** initialized to a valid address or null.
- I.e., `*p = 10;` is wrong unless memory has been allocated and p set to point to it.

A picture

```
int i = 19;
int *p;
int *q;
*p = i; /*error*/
q = &i
```



Symbol Table

i	0x80493e0
p	0x80494dc
q	0x80494e0

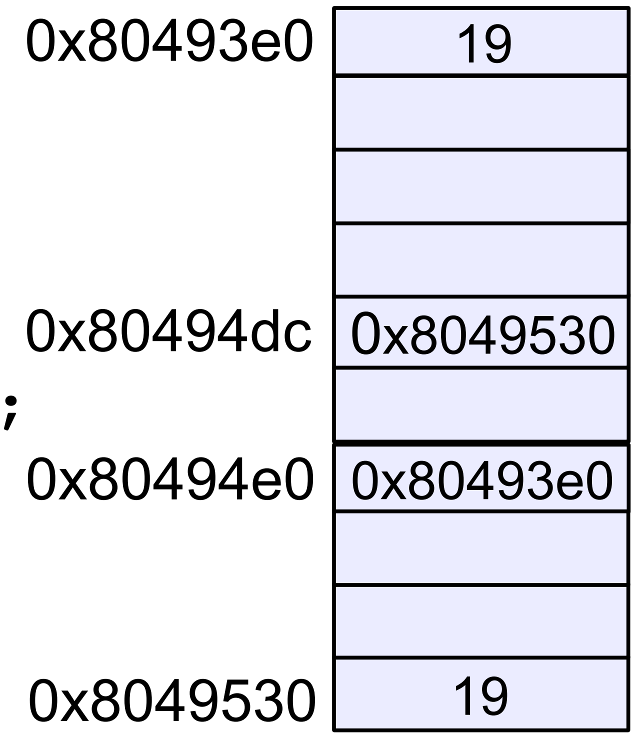
A picture

```
int i = 19;
int *p;
int *q;

q = &i
p = (int *)malloc(sizeof(int));
*p = i;
```

Symbol Table

i	0x80493e0
p	0x80494dc
q	0x80494e0

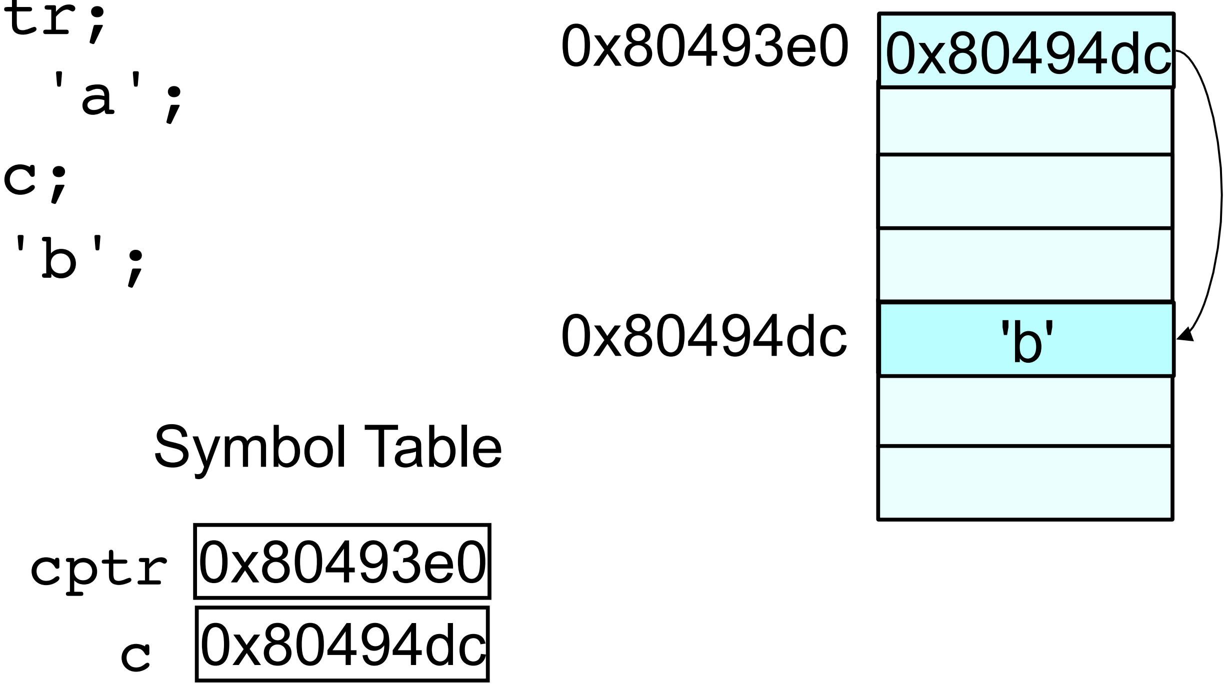


Pointers and Arrays

- Recall the pointer syntax:
- `char *cptr;`
  - declares a pointer to a char
  - allocates space to store a pointer (to a char)
- `char c = 'a';`
- `cptr = &c;`
  - `cptr` gets the value of the address of `c`
  - the value stored at the memory location referred to by `cptr` is the address of the memory location referred to by `c`;
- `*cptr = 'b';` – dereference `cptr`
  - the address stored at `cptr` identifies the memory location where `'b'` will be stored.

Pointers and Arrays

```
char *cptr;
char c = 'a';
cptr = &c;
*cptr = 'b';
```



Arrays vs. Pointers

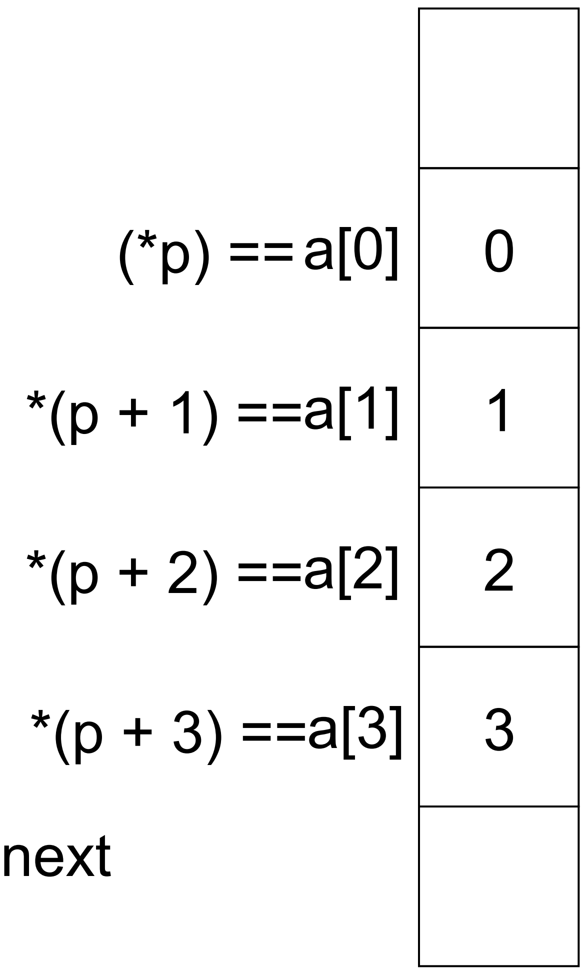
- An array name in expression context is used as into a pointer to the zero'th element.
- E.g.

```
int a[3] = {1, 3, 5};
int *p = a;  p = &a[0];
p[0] = 10;
printf("%d %d\n", a[0], *p);
```

Example

```
int a[4] = {0, 1, 2, 3};
int *p
p = a;
int i = 0;

for(i = 0; i < 4; i++) {
  printf("%d\n", *(p + i));
}
```



Why does adding 1 to `p` move it to the next spot for an int, when an int is 4 bytes?

Pointer Arithmetic

- Pointer arithmetic respects the type of the pointer.
- E.g.,

```
int i[2] = {1, 2};
int *ip;
ip = i;
*(ip + 1) += 2;
```

(really adds 4 to `ip`)

```
char c[2] = {'a','z'};
char *cp;
cp = c;
*(cp + 1) = 'b';
```

(really adds 1 to `cp`)

- C knows the size of what is being pointed at from the *type* of the pointer.

Pointer Arithmetic

- The array access operator [ ] is really only a shorthand for pointer arithmetic + dereference
- These are equivalent in C:

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

- C translates the first form into the second.
  - pointers and arrays are nearly the same in C!

Passing Arrays as Parameters

```
int main()
{
   int i[3] = {10, 9, 8};
   printf("sum is %d\n", sum(i)); /*??*/
   return 0;
}
int sum( What goes here? ) {
}
```

- What is being passed to the function is the name of the array which decays to a pointer to the first element – a pointer of type int.

Passing Arrays as Parameters

```
int sum( int *a ) {
    int i, s = 0;
    for(i = 0; i < ??; i++)
        s += a[i];  /* this is legal */
    return s;
}
```

`sizeof(a)==4` since **a** is just a pointer here

- How do you know how big the array is?
- Remember that arrays are not objects, so knowing where the zero'th element of an array is does not tell you how big it is.
- Pass in the size of the array as another parameter.

Array Parameters

`int sum(int *a, int size)`

- Also legal is:

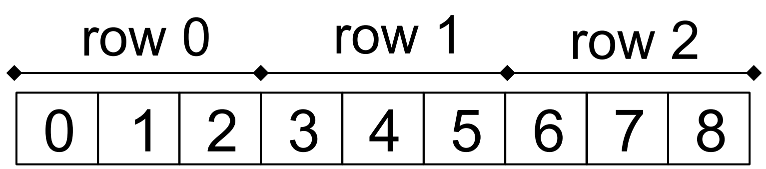
`int sum(int a[], int size)`

- Many advise against using this form.
  - You really are passing a pointer-to-int not an array.
  - You still don't know how big the array is.
  - Outside of a formal parameter declaration `int a[];` is illegal

⇨ `int a;` and `int a[10];` are completely different things

Multi-dimensional arrays

- Remember that memory is a sequence of bytes.




```
int a[3][3] = {   {0, 1, 2},
                  {3, 4, 5},
                  {6, 7, 8}};
```

- Arrays in C are stored in row-major order
- row-major access formula

```
x[i][j] == *(x + i * n + j)
```

where `n` is the row size of `x`

But use array notation!

Structs

- A collection of related data items

```
struct  record {
   char name[MAXNAME];
   int count;
};
```

/* The semicolon is important! It terminates the declaration. */

```
struct record rec1; /*allocates space for the record */
strncpy(rec1.name, ".exe", MAXNAME);
struct record *rec2;
rec2 = malloc(sizeof(struct record));
strncpy(rec2->name, ".gif", MAXNAME);
```

structs as arguments

```
/* Remember: pass-by-value */
void print_record(struct record r) {
   printf("Name = %s\n", r.name);
   printf("Count=%d\n", r.count);
}
print_record(rec1);
print_record(*rec2);
```

Passing pointer or struct?

```
/* Incorrect */
void incr_record(struct record r) {
   r.count++;
}

/* Correct */
void incr_record(struct record *r) {
   r->count++;
}
```

Summary

- The name of an array can also be used as a pointer to the zero'th element of the array.
- This is useful when passing arrays as parameters.
- Use array notation rather than pointer arithmetic whenever you have an array.