Arrays, Pointers, and Strings

Fall 2014

Jinkyu Jeong
(jinkyu@skku.edu)
ARRAYS
Array

- An array is a set of subscripted variables of the same type
  
  ```
  int a[10];
  char a[20];
  ```

- Why we need arrays?
  - Making code more compact and readable

```c
int a0, a1, a2, ..., a99;
scanf("%d%d...%d", &a0, &a1, ... &a99);
printf("%d%d...%d\n", a0, a1, ... a99);
```

```c
int a[100], i;
for(i=0; i<100; ++i)
    scanf("%d", &a[i]);
for(i=0; i<100; ++i)
    printf("%d", a[i]);
```
1-Dimensional Arrays

- **Declaration**
  
  ```c
  int num[5]; /* size should be constant */
  ```

- **Representation in memory**
  - 5 int variables are consecutively allocated in memory

- **Initializing an array**
  
  ```c
  int num[5] = {1, 2, 3, 4, 5};
  int num[] = {1, 2, 3, 4};
  int num[5] = {0};
  ```
  
  - External or static arrays are initialized to zero by default
  - Local arrays are not initialized
1-Dimensional Arrays

- **Subscripting an array**
  
  ```c
  int x, a[N];
  x = a[expr];
  ```
  
  - `expr` - integral expression whose result is between $0 \sim N-1$

- **Examples**
  
  ```c
  int a[5], x=2;
  a[0] = 1; a[1] = 2;
  a[++x] = 4;
  a[x+1] = 5;
  a[5] = 10 /* index out of range */
  ```
String

- **String is an array of type char**
  - String is a null-terminated
    \[ \text{NULL} == \theta == '\0' \]

- **char array initialization**
  ```
  char a[5] = {'a', 'b', 'c', 'd', 'e'};
  char b[] = {'a', 'b', 'c'};
  char c[] = {'a', 'b', 'c', '\0'};
  char d[] = "abc";
  ```
Arrays: Example

- Write backward

```c
#include <stdio.h>
void wrt_it(void)
{
    int c;
    if (((c=getchar()) != '\n'))
        wrt_it();
    putchar(c);
}
int main(void)
{
    printf("Input a line: ");
    wrt_it();
    printf("\n\n");
    return 0;
}
```

```c
#include <stdio.h>
void wrt_it(void)
{
    int c[100], n=0;
    while ((c[n]=getchar()) != '\n')
        ++n;
    while ( n > 0 )
        putchar(c[--n]);
    return 0;
}
int main(void)
{
    printf("Input a line: ");
    wrt_it();
    printf("\n\n");
    return 0;
}
```
POINTERS
### Pointer: First Glance

- **Declaration**
  ```c
  int *a; /* a pointer variable storing an address of integer storage */
  ```
  
- A special variable storing **an address** (not a value)

- **Points to a variable** whose address is the value of the pointer variable
Variables and Address

- Every variable has its own address in memory
  - Memory is byte-addressable

```c
int main(void)
{
    double d = 3.14;
    int i = 10;
    char c = 'a';
    ...
}
```
Variables and Address

- Address operator
- Expresses an address of a variable

```c
int main(void)
{
    double d = 3.14;
    int i = 10;
    char c = 'a';
    int a = (int)&i;
    ...
}
```
Pointer: Initialization

- Initializing a pointer variable

```c
int *p1, i, *p2;
p1 = 0;
p1 = NULL; /* equivalent to p = 0; */
p2 = &i;
p2 = (int *)1776; /* an absolute address */
char c;
p2 = &c /* compiler complains */
```

Address is 4 bytes in 32-bit machine and 8 bytes in 64-bit machine
Pointer: Dereference

* Dereferencing (indirection) operator
  - Specifies a variable (memory) pointed (referenced) by a pointer
  - * in pointer declaration is different from dereferencing operator *

**Usage**

```c
int i=10, j, *p;
p = &i;
j = *p; /* j = 10 */
*p = 20;
```

![Diagram showing pointer declaration and dereferencing examples](image-url)
### Pointer: Usage

#### Declarations and initializations

```c
int i = 3, j = 5, *p = &i, &q = &j, *r;
double x;
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Equivalent expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>i == * &amp; i</td>
<td>i == (* (&amp; i))</td>
<td>1</td>
</tr>
<tr>
<td>p == &amp; * p</td>
<td>p == (&amp; (* p))</td>
<td>1</td>
</tr>
<tr>
<td>p == &amp; i</td>
<td>p == (&amp; i)</td>
<td>1</td>
</tr>
<tr>
<td>* * &amp; p</td>
<td>* (* (&amp; p))</td>
<td>3</td>
</tr>
<tr>
<td>r = &amp; x</td>
<td>r = (&amp; x)</td>
<td>/* illegal */</td>
</tr>
<tr>
<td>7 * * p / * q + 7</td>
<td>(((7 * (* p))) / (* q)) + 7</td>
<td>11</td>
</tr>
<tr>
<td>* (r = &amp; j) *= * p</td>
<td>(* (r = (&amp; j))) * = (* p)</td>
<td>15</td>
</tr>
</tbody>
</table>
Call-by-Value

- Call-by-value: passing values of variables to a function
  - Only copies the value of an argument

```c
void func(int i) {
    ...
}
void main(void) {
    int i = 10;
    func(i);
    ...
}
```

Program

```
func()                                           main()
    i: 10                                 i: 10
```

Diagram
Call-by-Reference (1)

- Call-by-reference: passing the address of variables to a function
  - Parameters are declared as pointer variables

```c
void func(int *i)
{
    ...
    *i = 20;
}
void main(void)
{
    int i = 10;
    func(&i);
    ...
}
```

Program

```
function
i: 4000
main()
4000 i: 10
```
Call-by-Reference (2)

- Call-by-reference: passing the address of variables to a function
  - Parameters are declared as pointer variables
  - Callee can change the value of the caller's variable

```c
void func(int *i) {
    ...
    *i = 20;
}
void main(void) {
    int i = 10;
    func(&i);
    ...
}
```

Program

<table>
<thead>
<tr>
<th>func()</th>
</tr>
</thead>
<tbody>
<tr>
<td>i: 4000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>main()</th>
</tr>
</thead>
<tbody>
<tr>
<td>4000 i: 20</td>
</tr>
</tbody>
</table>
Call-by-Reference: Example

- Swap two variables

```c
#include <stdio.h>

void swap(int *, int *
);

int main(void)
{
    int i=3, j=5;
    swap(&i, &j);
    printf("%d %d
", i, j);
    return 0;
}

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

5 3
Call-by-Reference: Example

- **scanf**

```c
int i;
char c;
char str[10];
scanf("%d%c%s", &i, &c, str);
```

- Parameters are called by reference
- `scanf` function stores the read value into each variable
- String (an array of char)?
Arrays and Pointers (1)

- Pointer +/- integer points to the next/previous storage
  - Address displacement is calculated by compiler
  - * (dereference operator) dereferences the pointed storage

```c
int *p = (int*)5000;

printf("%d\n", (int)(p));  // 5000
printf("%d\n", (int)(p+1)); // 5004
printf("%d\n", (int)(p+2)); // 5008
printf("%d\n", (int)(p+3)); // 5012
printf("%d\n", (int)(p+4)); // 5016
```
Arrays and Pointers (2)

- In C, arrays and pointers are interrelated
  - Array name is a (constant) pointer

```c
int a[5];

printf("%d\n", (int)(a));
printf("%d\n", (int)(a+1));
printf("%d\n", (int)(a+2));
printf("%d\n", (int)(a+3));
printf("%d\n", (int)(a+4));
```

Address  4000  4004  4008  4012  4016

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
<td>a+1</td>
<td>a+2</td>
<td>a+3</td>
<td>a+4</td>
</tr>
<tr>
<td>*(a)</td>
<td>*(a+1)</td>
<td>*(a+2)</td>
<td>*(a+3)</td>
<td>*(a+4)</td>
<td></td>
</tr>
</tbody>
</table>
Arrays and Pointers (3)

- Pointer +/- integer points to the next/previous storage
  - A pointer can be used like an array

```c
int *p = (int*)5000;
```

<table>
<thead>
<tr>
<th>Address</th>
<th>5000</th>
<th>5004</th>
<th>5008</th>
<th>5012</th>
<th>5016</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>*(p)</td>
<td>*(p+1)</td>
<td>*(p+2)</td>
<td>*(p+3)</td>
<td>*(p+4)</td>
</tr>
<tr>
<td>p</td>
<td>p</td>
<td>p+1</td>
<td>p+2</td>
<td>p+3</td>
<td>p+4</td>
</tr>
</tbody>
</table>

```c
printf("%d\n", (int)(p));
printf("%d\n", (int)(p+1));
printf("%d\n", (int)(p+2));
printf("%d\n", (int)(p+3));
printf("%d\n", (int)(p+4));
```
Arrays and Pointers (4)

- In C, arrays and pointers are interrelated
  - Array name is a (constant) pointer

```c
int a[5], i, *p;

a       is equivalent to  &a[0]
a[i]    is equivalent to  *(a + i)
a + i    is equivalent to  &a[i]
p[i]    is equivalent to  *(p + i)
p + i    is equivalent to  &p[i]
a = p; ++a; a += 2; &a /* illegal */
```
Pointer Arithmetic (1)

- Pointer arithmetic is one of the powerful features of C
- Makes us easily manipulate addresses

```c
double a[2], *p, *q;
p = a;
q = p + 1;
printf("%d\n", q - p);
printf("%d\n", (int)q - (int)p);
```

<table>
<thead>
<tr>
<th>4000</th>
<th>a[0]: ??</th>
</tr>
</thead>
<tbody>
<tr>
<td>4008</td>
<td>a[1]: ??</td>
</tr>
<tr>
<td>4016</td>
<td>p: 4000</td>
</tr>
<tr>
<td>4020</td>
<td>q: 4008</td>
</tr>
</tbody>
</table>

q = p + 1, so q - p = 1
4008 - 4000 = 8
**Pointer Arithmetic (2)**

- **Arithmetic operations**
  - Addition, subtraction only
  - Specifies logical distance

```c
int *p;
p = (int*) 4004;
p = p - 1;
```

<table>
<thead>
<tr>
<th>...</th>
<th>4000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4004</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>p: 4004</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
### Pointer Arithmetic (3)

- **Arithmetic operations**
  - Addition, subtraction only
  - Specifies logical distance

  ```
  int *p;
p = (int*) 4000;
p = p - 1;
  ```

- Address value itself increases (or decreases) **by the size of variable** pointed by the pointer variable
### Comparison

- Equality operators
- Relational operators
- Regard addresses of pointer variables as the values of them

```c
int a[2], *p, *q;
p = a;
q = &a[1];
p < q; /* true */
p == a; /* true */
```
Arrays as Function Arguments

- When an array is passed as an argument to a function, the base address value is passed.
  - The array elements are not copied

- Function headers for an array
  - `double sum(double a[], int n)`
  - `double sum(double *a, int n)`

- Function prototypes
  - `double sum(double [], int);`
  - `double sum(double *, int);`
Arrays as Function Arguments: Example

double sum(double a[], int n)
{
    int i;
    double sum = 0.0;
    for (i=0; i<n; ++i)
        sum += a[i];
    return sum;
}

int main(void)
{
    double a[10];
    int i;
    for (i=0; i<10; ++i)
        scanf("%f", &a[i]);
    sum(a, 10);
    return 0;
}
Why Use Pointers

- It is hard to manage all values managed in a program in a code

- Use pointer to link them

- Manage one variable to point the head of them

- Organize sophisticated data structure (ex, linked list, tree, ...)
• **CPU (central processing unit)**

  ![Diagram of CPU and ALU](image)

  - **Registers (r1, r2, r3)**
    - Load/store values from/to memory
    - Supply values to ALU
  - **ALU (arithmetic logic unit)**
    - Perform arithmetic and logical computations

- **CPU**
  - Registers (r1, r2, r3)
    - Load/store values from/to memory
    - Supply values to ALU
  - **ALU** (arithmetic logic unit)
    - Perform arithmetic and logical computations

- **Input**
  - r1
  - r2

- **Output**
  - r3

- **Memory**
  - 4000
  - 4004
  - 4008
  - 4012
  - 4016
  - ...
**Pointer & Computer Architecture**

```
int a=10, b=20, c;
int *p = &a, *q = &b;
c = a + b;
c = *p + *q;
```

**C language**

```
load #4000, r1
load #4004, r2
add r1, r2, r3
store r3, #4008
```

**Machine language**

```
... 4000 a: 10
     4004 b: 20
     4008 c:
     4012 p: 4000
     4016 q: 4004
... Memory
```
**Pointer & Computer Architecture**

**C language**

```c
int a=10, b=20, c;
int *p = &a, *q = &b;
c = a + b;
c = *p + *q;
```

**Machine language**

- `load value(#4012), r1`
- `load value(#4016), r2`
- `add r1, r2, r3`
- `store r3, #4008`

**Input**
- `r1`
- `r2`

**Output**
- `r3`

**Memory**
- `4000: a: 10`
- `4004: b: 20`
- `4008: c:`
- `4012: p: 4000`
- `4016: q: 4004`
- `...`

**ALU**
- **CPU**

---

GEBD029: Basis and Practice in Programming | Fall 2014 | Jinkyu Jeong (jinkyu@skku.edu)
DYNAMIC MEMORY ALLOCATION
Dynamic Memory Allocation

- Memory of a program

Program

- Stack
  Stores function-local variables

- Heap
  Dynamically allocated memory

- Data, BSS segment
  Stores global/static variables

- Program Code
Dynamic Memory Allocation

- How recursion works

Program

<table>
<thead>
<tr>
<th>Stack</th>
<th>Stores function-local variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>main()</td>
<td></td>
</tr>
<tr>
<td>wrt_it()</td>
<td></td>
</tr>
<tr>
<td>wrt_it()</td>
<td></td>
</tr>
<tr>
<td>wrt_it()</td>
<td></td>
</tr>
<tr>
<td>wrt_it()</td>
<td></td>
</tr>
</tbody>
</table>

Heap

Dynamically allocated memory

Data, BSS segment

Stores global/static variables

Program Code
Dynamic Memory Allocation

Memory of a program

Program Code

Data, BSS segment

Heap

Stack

Stores function-local variables

- C library provides functions to dynamically allocate and free memory in the heap space

Dynamically allocated memory

Stores global/static variables
Why Dynamic Memory Allocation?

- Read N integers and print sum of them

```c
#include <stdio.h>

int main(void)
{
    int n, i, array[100];
    printf("how many numbers? ");
    scanf("%d", &n);
    for (i=0; i<n; ++i)
        scanf("%d", &array[i]);
    printf("sum: %d\n", sum(array, n));
    return 0;
}
```

how many numbers? 1000
Why Dynamic Memory Allocation?

- Dynamic memory allocation
  - Dynamically allocate a specified amount of memory

- `void *malloc(int m)`
  - Allocate `m`-bytes of memory

- `void *calloc(int m, int n)`
  - Allocate `m * n` bytes of memory and initialize it to zero
  - If failed, both functions return NULL

- `calloc(n, m)` is equivalent to
  
  ```c
  p = malloc(n*m);
  memset(p, 0, m*n);
  ```
Why Dynamic Memory Allocation?

- Read N integers and print sum of them

```c
#include <stdio.h>

int main(void)
{
    int n, i, *array;
    printf("how many numbers? ");
    scanf("%d", &n);
    array = (int*)malloc(sizeof(int)*n);
    for (i=0; i<n; ++i)
        scanf("%d", &array[i]);
    printf("sum: %d\n", sum(array, n));
    free(array);
    return 0;
}
```

Casting is not essential because the return type of malloc()/calloc() is void * which is of type generic pointer.
Memory Release

- You’d better free the allocated space
  - `free(p);`
  - `p` must be the pointer to the space allocated by `calloc()` or `malloc()`

- If you forget to free,
  - It will be freed when the process exits for some systems like Linux, Windows
  - For some other systems, nothing is guaranteed

- Reset pointer after releasing memory
  ```c
  int* p = malloc(4);
  ...
  free(p);
  p = NULL;
  ```
STRINGS
String

- String is an array of type char
  - String is a null-terminated
    
    \[
    \text{NULL} == 0 == '\0'
    \]

- char array initialization

  ```
  char a[5] = {'a', 'b', 'c', 'd', 'e'};
  char b[] = {'a', 'b', 'c'};
  char c[] = {'a', 'b', 'c', '\0'}; /* string */
  char d[] = "abc"; /* string */
  ```

  `These are not string but arrays of char`

```
  c: a b c \n \0
d: a b c \n \0
```

char *p = "abc";

```
p: a b c  \
```

# Strings

- **End-of-string sentinel: \'\0\'**

```c
#include <stdio.h>

void my_printf(const char* fmt, char* p);

int main(void)
{
    char s[] = "abc\n";
    printf("%s", s);
    return 0;
}

void my_printf(const char* fmt, char* p)
{
    while (*p != '\0')
        putchar(*p++);
}
```
### String: Example

#### Word counting

```c
#include <ctype.h>

int word_cnt(const char *s)
{
    int cnt = 0;
    while (*s != '\0') {
        while (isspace(*s))
            ++s;
        if (*s != '\0') {
            ++cnt;
            while (!isspace(*s) && *s != '\0')
                ++s;
        }
    }
    return cnt;
}
```

```
a b c    d e f \0
```
String-Handling Functions

- ANSI C Lib contains many useful functions
  - `char *strcat(char *s1, const char *s2);`
    - Concatenate two strings and store result in *s1
    - What if there is no space after s1?
  - `int strcmp(const char *s1, const char *s2);`
    - Returns negative, zero, positive depending on the lexicographical order
  - `char *strcpy(char *s1, const char *s2);`
    - Copy s2 to s1
    - What if s2 is longer than s1?
  - `size_t strlen(const char *s);`
    - Counts the number of characters in s
    - size_t is usually unsigned int
Caveats

- **Think of strings delimited by '\0'**.
  - `strlen()` excludes the terminating '\0'
  - The size of a string (a char array) **must** include the storage needed for the '\0'
  - The programmer should make sure that the string bounds are not overrun

```c
char *src = "hello world", *dst;
int len = strlen(src);
dst = malloc(len + 1);
strcpy(dst, src);
```

- **Do not compare char pointers**

```c
char s1[] = "abc", s2[] = "abc";
if ( s1 == s2 ) /* false */
  printf("the same!!\n");
```
String-Handling Functions:
Example

```c
unsigned strlen(const char* s)
{
    register int n;
    for (n=0; *s != '\0'; ++s)
        ++n;
    return n;
}

char *strcat(char *s1, const char *s2)
{
    register char *p = s1;
    while (*p)
        ++p;
    while (*p++ = *s2++)
        return s1;
}
```
String-Handling Functions: Example

char* strcpy(char *s1, const char *s2)
{

}
# String-Handling Functions: Example

<table>
<thead>
<tr>
<th>Declaration and initializations</th>
</tr>
</thead>
<tbody>
<tr>
<td>char s1[] = &quot;beautiful big sky country&quot;, s2[] = &quot;how now brown cow&quot;;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>strlen(s1)</td>
<td>25</td>
</tr>
<tr>
<td>strlen(s2+8)</td>
<td>9</td>
</tr>
<tr>
<td>strcmp(s1, s2)</td>
<td>negative integer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Statements</th>
<th>What gets printed</th>
</tr>
</thead>
<tbody>
<tr>
<td>printf(&quot;%s&quot;, s1 + 10);</td>
<td>big sky country</td>
</tr>
<tr>
<td>strcpy(s1 + 10, s2 + 8);</td>
<td></td>
</tr>
<tr>
<td>strcat(s1, &quot;s!&quot;);</td>
<td></td>
</tr>
<tr>
<td>printf(&quot;%s&quot;, s1);</td>
<td>beautiful brown cows!</td>
</tr>
</tbody>
</table>
Reading/Writing Strings

- `printf("%s", s);`
- `scanf("%s", s);`
  - `s` should be large enough to store the input and `\0`
  - Read a string delimited by white space
- `puts(const char* s);`
  - Print string with `\n`
- `gets(const char* s);`
  - Read a string by line
  - `s` should be large enough to store the input and `\0`

```c
char line[1024];
gets(line);
puts(line);
puts(line);
```
MULTI-DIMENSIONAL ARRAYS
Multi-dimensional Arrays

- Declaration
  
  ```
  data_type variable_name[size][size];
  ```

- Examples
  
  ```
  int a[100]; /* one-dimensional array */
  int b[10][10]; /* two-dimensional array */
  int c[10][10][10]; /* three-dimensional array */
  ```

- Logical memory layout

  ```
  int a[2][3]  row 0
               row 1
  ```

<table>
<thead>
<tr>
<th>row 0</th>
<th>row 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>col 0</td>
<td>col 1</td>
</tr>
<tr>
<td>a[0][0]</td>
<td>a[0][1]</td>
</tr>
<tr>
<td>a[1][0]</td>
<td>a[1][1]</td>
</tr>
</tbody>
</table>
Multi-dimensional Arrays

- **Physical memory layout**

  int a[2][3]  

  ![Diagram of int a[2][3] with rows 0 and 1]

  - Looked like a one-dimensional array

- **Actually!!**

  - Two-dimensional array is an array of arrays
    - int a[2][3] → two int[3]'s
  - Three-dimensional arrays
    - int a[2][3][4] → two int[3][4]'s → six int[4]'s
Multi-dimensional Arrays

- **Initializing a multi-dimensional array**
  
  ```
  int a[2][3] = {{0, 1, 2}, {3, 4, 5}};
  int a[2][3] = {0, 1, 2, 3, 4, 5};
  int a[][3] = {{0, 1, 2}, {3, 4, 5}};
  int a[2][3] = {0};
  ```

- **Subscripting a multi-dimensional array**
  
  ```
  int a[2][3], x=1, y=2;
  a[x][y] = 10;
  a[++]x][y+1] = 5;
  ```
Multi-dimensional Arrays

- Base address of a two-dimensional array is not \texttt{a} but \texttt{&a[0][0]}

- Storage mapping function
  
  \begin{verbatim}
  int a[2][3];
  a[i][i]; /* equals *(&a[0][0] + 3*i + j) */
  &a[1][2] == &a[0][0] + 3*1 + 2
  \end{verbatim}

- Equivalent expressions to \texttt{a[i][j]}
  
  \begin{itemize}
    \item *(a[i] + j)
    \item *((a + i))[j]
    \item *(((a + i)) + j)
    \item *(&a[0][0] + 3 * i + j)
  \end{itemize}
Parameter Passing

- **Two-dimensional array**
  - int sum(int a[][5])
  - int sum(int a[3][5]) /* compiler ignores 3 */
  - int sum(int (*a)[5])

→ Specifying an array of a[5]'s

```c
int sum(int a[][5], int n)
{
    int i, j, sum = 0;
    for (i=0; i < n; ++i )
        for (j=0; j < 5; ++j )
            sum += a[i][j];
    return sum;
}
```
Multi-Dimensional Arrays: Example

- **Matrix multiplication**

  - $A \times B = C$ /* each is 3 by 3 matrix */

```c
#include <stdio.h>
void read_matrix(int x[][3])
{
    int i;
    for (i=0; i<3; ++i)
        scanf("%d%d%d", &x[i][0], &x[i][1], &x[i][2]);
}
void matrix_multi(int a[][3], int b[][3], int c[][3])
{
    int i, j, k;
    for(i=0; i<3; ++i)
        for(j=0; j<3; ++j)
            for(k=0; k<3; ++k)
                c[i][j] += a[i][k] * b[k][j];
}

int main(void)
{
    int a[3][3], b[3][3], c[3][3]={0};
    int i, j, k;
    printf("input matrix a(3x3)\n");
    read_matrix(a);
    printf("input matrix b(3x3)\n");
    read_matrix(b);
    matrix_multi(a, b, c);
    for(i=0; i<3; ++i) {
        for(j=0; j<3; ++j)
            printf("%d ", c[i][j]);
        printf("\n");
    }
    return 0;
}
```
Arrays of Pointers

- `char *w[N];`
  - An array of pointers
  - Each pointer is to char type

**Ragged array**

```c
char *p[2]={"abc","1234567890"}; /*ragged array*/
```

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>\0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>\0</td>
<td></td>
</tr>
</tbody>
</table>
```

```c
char a[2][11]={"abc", "1234567890"};
```

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>\0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>\0</td>
<td></td>
</tr>
</tbody>
</table>
```

Read the sort_words example in the textbook
Arguments to main( )

- argc and argv are used for main()
  - argc is the number of arguments
  - argv is an array of pointers
    - argv[0] is the name of the main program
    - Then naturally, argc >= 1

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
    int i;
    printf("argc = %d\n", argc);
    for (i=0; i<argc; ++i)
        printf("argv[%d] = %s\n", i, argv[i]);
    return 0;
}
```

$ ./a.out hello world
argc = 3
argv[0] = ./a.out
argv[1] = hello
argv[2] = world
Double Pointer

- `int **a;`
  - Pointer variable to a pointer variable

- **Usage**
  - Call-by-reference to update a pointer variable

```c
void swap_string(char **p, char **q)
{
    char* tmp;
    tmp = *p;
    *p = q;
    *q = tmp;
}
```

Program:

```
main()
{
    swap_string(&p, &q);
}
```
Double Pointer

- int **a;
  - Pointer variable to a pointer variable

- Usage
  - Call-by-reference to update a pointer variable

```c
void swap_string(char **p, char **q)
{
    char* tmp;
    tmp = *p;
    *p = q;
    *q = tmp;
}
```

Program

```
main()
```

```
swap_string()
```

```
tmp p q
```

```
x y
```

```
a b c \theta
```

```
x y z \theta
```
Double Pointer

- int **a;
  - Pointer variable to a pointer variable

- Usage
  - Call-by-reference to update a pointer variable

```c
void swap_string(char **p, char **q)
{
    char* tmp;
    tmp = *p;
    *p = q;
    *q = tmp;
}
```

Program

```c
swap_string(

main()
```
Functions as Arguments

- A function name can be passed as an argument

- Think a function name as a pointer (like an array)

- `type (*func)(parameter_list)`
  - `func` is a pointer to a function
  - `*func` is a function
  - `(*func)(x)` is a call to the function
  - `func(x)` is a call to the function

- If you are still confused, just follow the example
```c
#include <math.h>
#include <stdio.h>

double f(double);

double sum_square(double (*)(double), int, int);

int main(void)
{
    printf("%.7f\n%.7f\n", " First computation: ", sum_square(f, 1, 10000), "Second computation: ", sum_square(sin, 2, 13));
    return 0;
}

double sum_square(double (*)(double), int m, int n)
{
    int k;
    double sum = 0.0;
    for (k=m; k<=n; ++k )
        sum += (*f)(k) * (*f)(k);
    return sum;
}

double f(double x)
{
    return 1.0/x;
}
```
Functions as Arguments

- double g(double) returns double
- double *g(double) returns a pointer

Equivalent function prototype definitions

```c
double sum_square(double f(double x), int m, int n);
double sum_square(double f(double ), int m, int n);
double sum_square(double f(double ), int , int );
double sum_square(double (*f)(double x), int , int );
double sum_square(double (* )(double x), int , int );
```
const volatile

- **const int N = 3;**
  - It cannot be changed after initialization
  - It cannot be used for array definition like `int k[N];`

- **extern const volatile int real_time_clock;**
  - **extern** means that look for it elsewhere, either in this file or in some other file
  - **volatile** means the variable can be modified by other part of a computer,
  - **const** specifies that you cannot change the value, JUST READ it