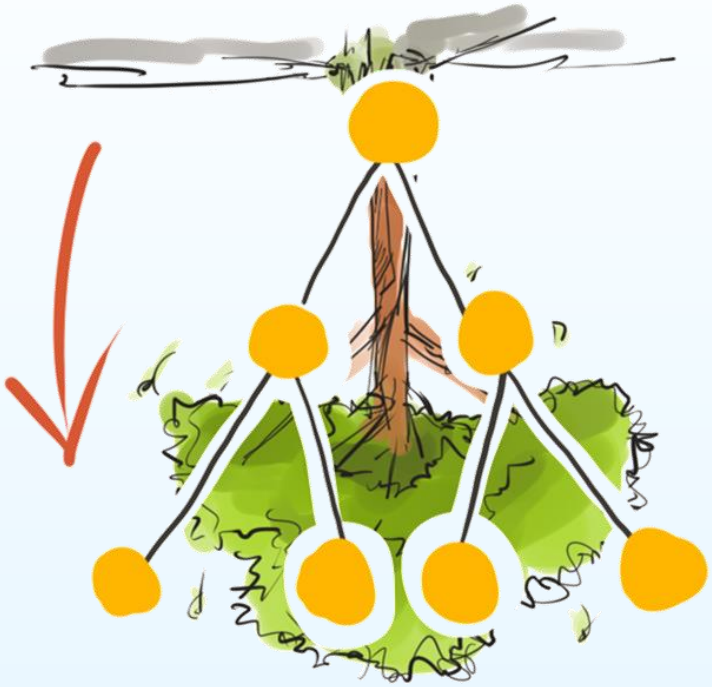


Lecture 3: C Programming (Part III)

01204212 Abstract Data Types and Problem Solving

Department of Computer Engineering
Faculty of Engineering, Kasetsart University
Bangkok, Thailand.

Department of Computer Engineering Kasetsart University

MIKE LABORATORY

Outline

- Array
  - One and Multi-Dimensional Arrays
  - Passing Array to Function
- Pointer
  - Address-of and Indirect Operators
  - Dynamic Memory Allocation
- String
  - Standard Library Functions

What is an Array?

- An array is a group of consecutive memory location

```
char c[10];
```

- Each element has the same name and data type

- The array index always starts with 0

Symbol		Address	Value
high	…		
		0x7fffb5	
		0x7fffb4	
	9	0x7fffb3	
	8	0x7fffb2	
	7	0x7fffb1	
	6	0x7fffb0	
	5	0x7fffaf	
	4	0x7fffae	
	3	0x7fffad	
	2	0x7fffac	
	1	0x7fffab	
c	0	0x7fffaa	
		0x7fffa9	
		0x7fffa8	
low	…		

One-dimensional Array

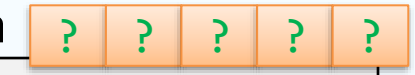
Syntax:

```
datatype variable[size];
datatype variable[size] = {val1, val2, ...};
```

only the first initialization

- Uninitialized array

```
int a[5];
```



- Initialized array

```
int a[5] = {1,2,3,4,5};
```

- Partially initialized array

```
int a[5] = {1,2};
```

- Array with omitting size

```
int a[] = {1,2,3};
```

Array Usage

- Use the array subscript operator `[]`
- The index must be
  - first at `0` and last at `size-1`
  - non-negative integer

```
int a[5] = {1,2,3,4,5};
printf("%d\n", a[1]);
```

- The array is mutable

```
a[4] = 10;
```

Example: Digit Counting

```
1:  #include <stdio.h>
2:
3:  int main(void) {
4:    char c;
5:    int  i, iCount[10];
6:
7:    for (i=0; i<10; i++)
8:      iCount[i] = 0;
9:    printf("Enter number: ");
10:   while ((c = getchar()) != EOF)
11:     if ((c >= '0') && (c <= '9'))
12:       iCount[c-'0']++;
13:   printf("Digit\tCount\n");
14:   printf("-----\t-----\n");
15:   for (i=0; i<10; i++)
16:     printf("%5d\t%5d\n", i, iCount[i]);
17:   return 0;
18: }
```

```
ASCII
'0'  0x30  '0'-'0' = 0
'1'  0x31  '1'-'0' = 1
'2'  0x32  '2'-'0' = 2
'3'  0x33  '3'-'0' = 3
'4'  0x34  '4'-'0' = 4
'5'  0x35  '5'-'0' = 5
'6'  0x36  '6'-'0' = 6
'7'  0x37  '7'-'0' = 7
'8'  0x38  '8'-'0' = 8
'9'  0x39  '9'-'0' = 9
```

Crtl-d

```
Enter number: 12123
Digit   Count
-----   -----
    0       0
    1       2
    2       2
    3       1
    4       0
    5       0
    6       0
    7       0
    8       0
    9       0
```

Symbolic Constants

- You cannot do like these:

```
int         size = 5;
double      scores[size] = {0,0,0,0,0};
```

```
const int   size = 5;
double      scores[size] = {0,0,0,0,0};
```

- But this is OK

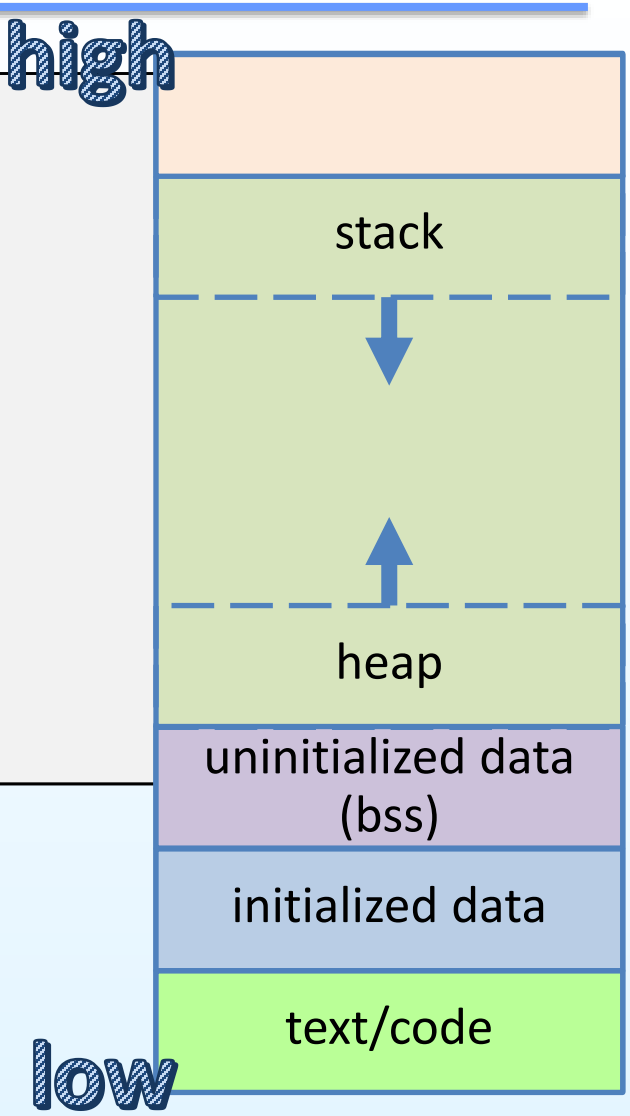
```
int         size = 5;
double      scores[size];
```

- Or, use a symbolic constant instead

```
#define     SIZE 5
double      scores[SIZE] = {0,0,0,0,0};
```

Where these arrays are stored?

```
1:  #include <stdio.h>
2:  #define SIZE 10
3:
4:  int a[SIZE] = {1};
5:  int b[SIZE];
6:
7:  int main(void) {
8:    int c[SIZE] = {1};
9:    int d[SIZE];
10:
11:   return 0;
12: }
```



Caution! Caution! Caution!

- C does NOT check the array bounds
  - Even though the index points to an element within the array
  - Beyond the end or before the start of arrays

- It is the programmer's responsibility to avoid indexing outside the array
  - Prone to data corruption
  - May cause a segmentation fault
  - Could expose system to a security hole

The `sizeof` Operator

- The sizeof is a compile-time unary operator and used to compute the size in byte of its operand
- The result of sizeof is an unsigned integer type, denoted by `size_t`

Syntax:

```
sizeof (datatype);
sizeof object;
```

Example: The `sizeof` Operator

```
Size of a     : 4 bytes
Size of a+b   : 4 bytes
Size of char  : 1 bytes
Size of int   : 4 bytes
Size of float : 4 bytes
Size of double: 8 bytes
Size of size_t: 8 bytes
```

```
1:  #include <stdio.h>
2:
3:  int main(void) {
4:    int a = 1;
5:    int b = 2;
6:
7:    printf("Size of a     : %ld bytes\n", sizeof a);
8:    printf("Size of a+b   : %ld bytes\n", sizeof (a+b));
9:    printf("Size of char  : %ld bytes\n", sizeof (char));
10:   printf("Size of int   : %ld bytes\n", sizeof (int));
11:   printf("Size of float : %ld bytes\n", sizeof (float));
12:   printf("Size of double: %ld bytes\n", sizeof (double));
13:   printf("Size of size_t: %ld bytes\n", sizeof (size_t));
14:   return 0;
15: }
```

Getting Size of Array

- Get the size in bytes of memory allocated for an array

```
int a[] = {0,0,0,0,0};
printf("Size of a : %ld bytes\n", sizeof a);
```

```
Size of a : 20 bytes
```

- Compute the number of elements in an array

```
int a[] = {0,0,0,0,0};
int nElements = sizeof a / sizeof a[0];
printf("# elements in a : %d\n", nElements);
```

```
# elements in a : 5
```

Multi-dimensional Array

Syntax:

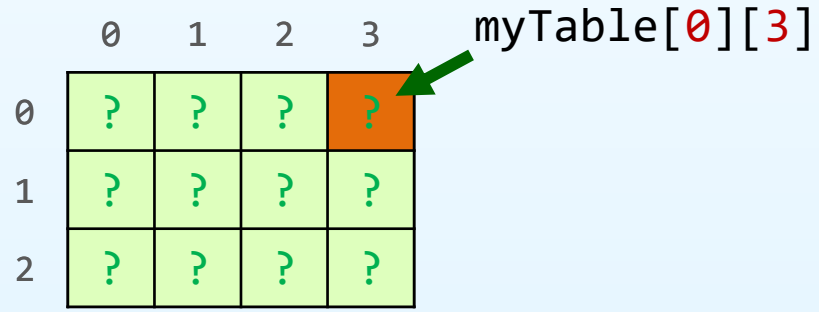
```
datatype variable[size1][size2]...[sizeN];
```

```
char   table[10][20];
int    cube[10][10][10];
double nDim[5][10][5][100];
```

Note: Two-dimensional array can be represented by a table

```
int myTable[3][4];
```

row column



Multi-dimensional Array

Syntax:

```
datatype variable[size1][size2]...[sizeN]
            = {{val11, val12, ...},
               {val21, val22, ...},
                ...,
               {valN1, valN2, ...}};
```

only the first initialization

However, any unspecified elements are set to zero

```
int myTable[3][4] = {{1,6},{2,2,4,3}};
```

myTable	0	1	2	3
0	1	6	0	0
1	2	2	4	3
2	0	0	0	0

Multi-dimensional Array

Syntax:

```
datatype variable[size1][size2]...[sizeN]
            = {{val11, val12, ...},
               {val21, val22, ...},
                ...,
               {valN1, valN2, ...}};
```

only the first initialization

Moreover, the first dimension can be unsized

```
int myTable[][4] = {{1,6},{2,2,4,3}};
```

myTable	0	1	2	3
0	1	6	0	0
1	2	2	4	3

Example: Power Table

```
1:  #include <stdio.h>
2:  #define  ROW 4
3:  #define  COL 5
4:
5:  int main(void) {
6:    int table[ROW][COL] = {{1, 2, 3, 4, 5}};
7:    int i, j;
8:
9:    for (i=1; i<ROW; i++)
10:     for (j=0; j<COL; j++)
11:       table[i][j] = table[i-1][j] * table[0][j];
12:   for (i=0; i<ROW; i++) {
13:     for (j=0; j<COL; j++)
14:       printf ("%5d", table[i][j]);
15:     printf ("\n");
16:   }
17:   return 0;
18: }
```

```
    1    2    3    4    5
    1    4    9   16   25
    1    8   27   64  125
    1   16   81  256  625
```

Array of Characters

- In C, string is represented by an array of characters
- Character arrays can be initialized using a string literal

```
char myStr[] = "Hello";
```

or a brace initialization

```
char myStr[] = {'H','e','l','l','o','\0'};
```

Note: the string is terminated with a null character (`'\0'`)

But, you CANNOT re-assign with either the string literal or brace initialization

```
char myStr[10];
myStr = "Hello";
myStr = {'H','e','l','l','o','\0'};
```

Accessing Array of Characters

```
1: #include <stdio.h>
2:
3: int main(void) {
4:   char myStr[] = "Hello";
5:
6:   printf ("%c\n", myStr[3]);
7:   return 0;
8: }
```

```
l
```

high

Symbol		Address	Value
	…		
		0x7fffb2	
		0x7fffb1	
	5	0x7fffb0	00000000 (0 = '\0')
	4	0x7fffaf	01101111 (111 = 'o')
	3	0x7fffae	01101100 (108 = 'l')
	2	0x7fffad	01101100 (108 = 'l')
	1	0x7fffac	01100101 (101 = 'e')
myStr	0	0x7fffab	01001000 (72 = 'H')
		0x7fffaa	
		0x7fffa9	
		0x7fffa8	
	…		

low

Printing out the Array Name

```
1: #include <stdio.h>
2:
3: int main(void) {
4:   char myStr[] = "Hello";
5:
6:   printf ("%p\n", myStr);
7:   printf ("%s\n", myStr);
8:   return 0;
9: }
```

```
0x7fffab
Hello
```

high

Symbol		Address	Value
		…	
		0x7fffb2	
		0x7fffb1	
	5	0x7fffb0	00000000 (0 = '\0')
	4	0x7fffaf	01101111 (111 = 'o')
	3	0x7fffae	01101100 (108 = 'l')
	2	0x7fffad	01101100 (108 = 'l')
	1	0x7fffac	01100101 (101 = 'e')
myStr	0	0x7fffab	01001000 (72 = 'H')
		0x7fffaa	
		0x7fffa9	
		0x7fffa8	
		…	

low

Getting String and Storing in Array

```
1:  #include <stdio.h>
2:  #define  SIZE 10
3:
4:  int main(void) {
5:    char myStr[SIZE];
6:
7:    printf("Enter a string: ");
8:    scanf("%s", myStr);
9:    return 0;
10: }
```

```
Enter a string: hello world
```

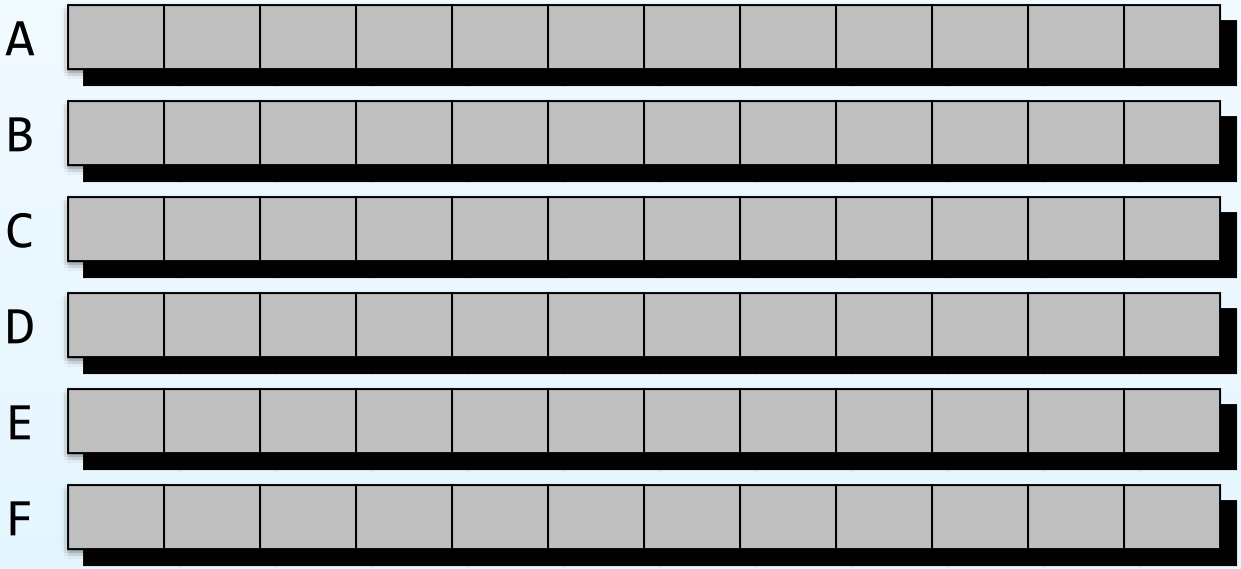
Note: Characters are read until a whitespace is encountered

What is the value of myStr[6]?

Symbol		Address	Value
high	…		
	9	0x7fffb2	????????
	8	0x7fffb1	????????
	7	0x7fffb0	????????
	6	0x7fffaf	????????
	5	0x7fffae	00000000 (0 = '\0')
	4	0x7fffad	01101111 (111 = 'o')
	3	0x7fffac	01101100 (108 = 'l')
	2	0x7fffab	01101100 (108 = 'l')
	1	0x7fffaa	01100101 (101 = 'e')
myStr	0	0x7fffa9	01001000 (72 = 'H')
		0x7fffa8	
low	…		

Examples: Array of Characters

```
char A[10];
char B[10] = {'H', 'e', 'l', 'l', 'o'};
char C[10] = {'H', 'e', 'l', 'l', 'o', '\0'};
char D[]   = {'H', 'e', 'l', 'l', 'o', '\0'};
char E[10] = "Hello";
char F[]   = "Hello";
```



Re-assigning Arrays

```
char a[10];
int  b[10];
int  c[10] = {1,2,3,4,5,6,7,8,9,10};
```

- You cannot do like these:

```
a = "Hello";
```

```
a = {'H','e','l','l','o','\0'};
```

```
b = {1,2,3,4,5,6,7,8,9,10};
```

```
b = c;
```

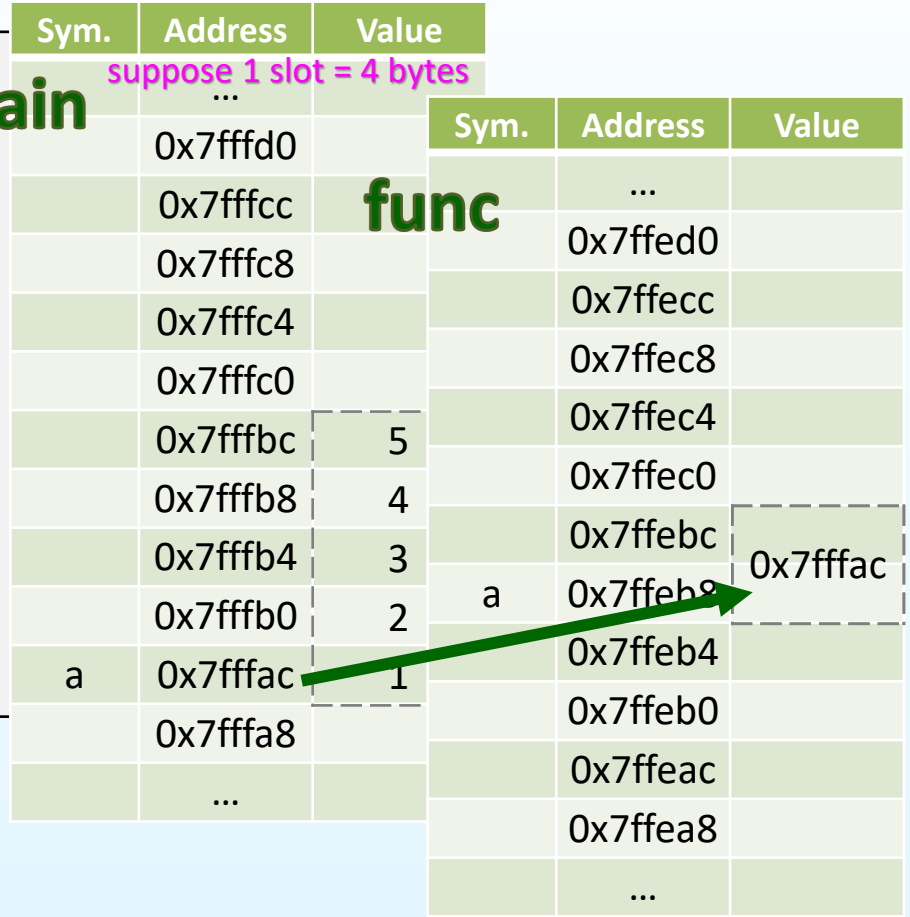
You must copy each element of the arrays instead

Passing Array to Function

- Caller copies the value (address of the array) to the parameter of the function

```c
#include <stdio.h>

void func(int a[]) {
  printf("func:%p %ld\n", a, sizeof a);
}

int main(void) {
  int a[5] = {1,2,3,4,5};

  printf("main:%p %ld\n", a, sizeof a);
  func(a);
  return 0;
}
```

```
main:0x7fffac 20
func:0x7fffac 8
```

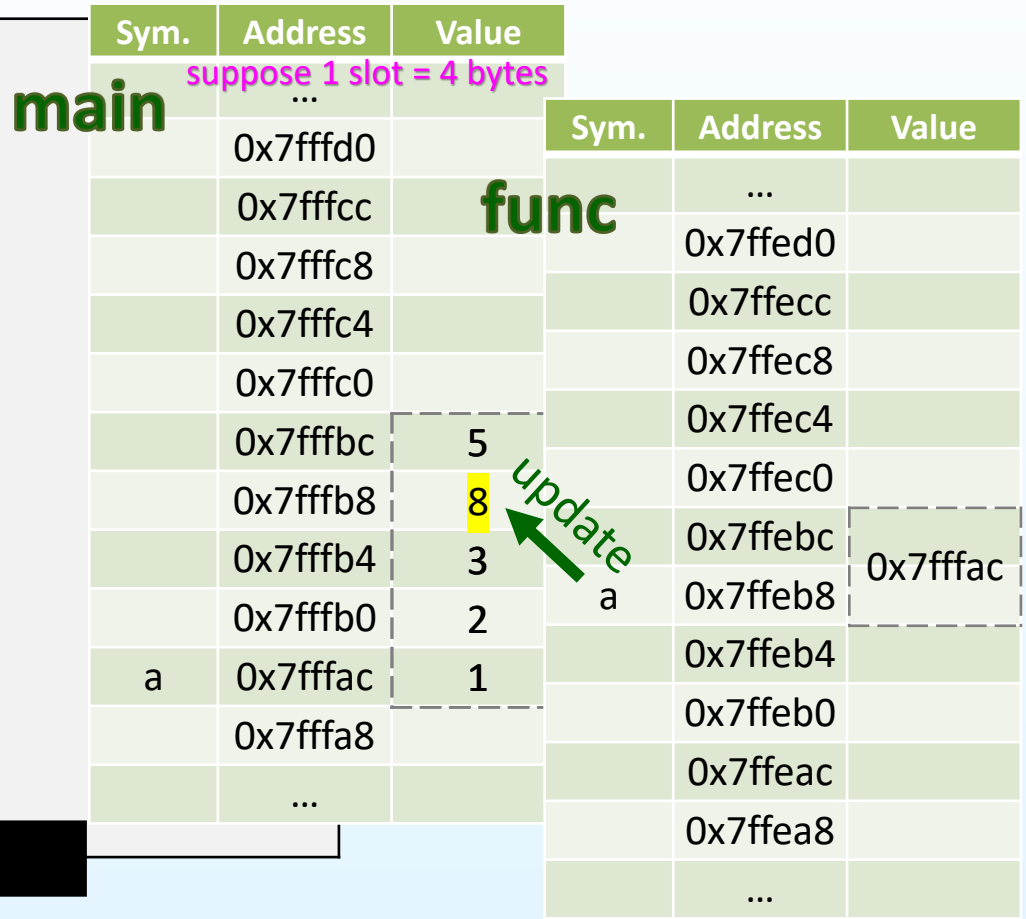


Passing Array to Function

- Function can reference through the parameter to reach (also, update) the array of the caller

```c
#include <stdio.h>

void func(int a[]) {
  a[3] *= 2;
}

int main(void) {
  int a[5] = {1,2,3,4,5};
  int i;

  func(a);
  for (i=0; i<5; i++)
    printf("%d ", a[i]);
  printf("\n");
  return 0;
}
```

```
1 2 3 8 5
```

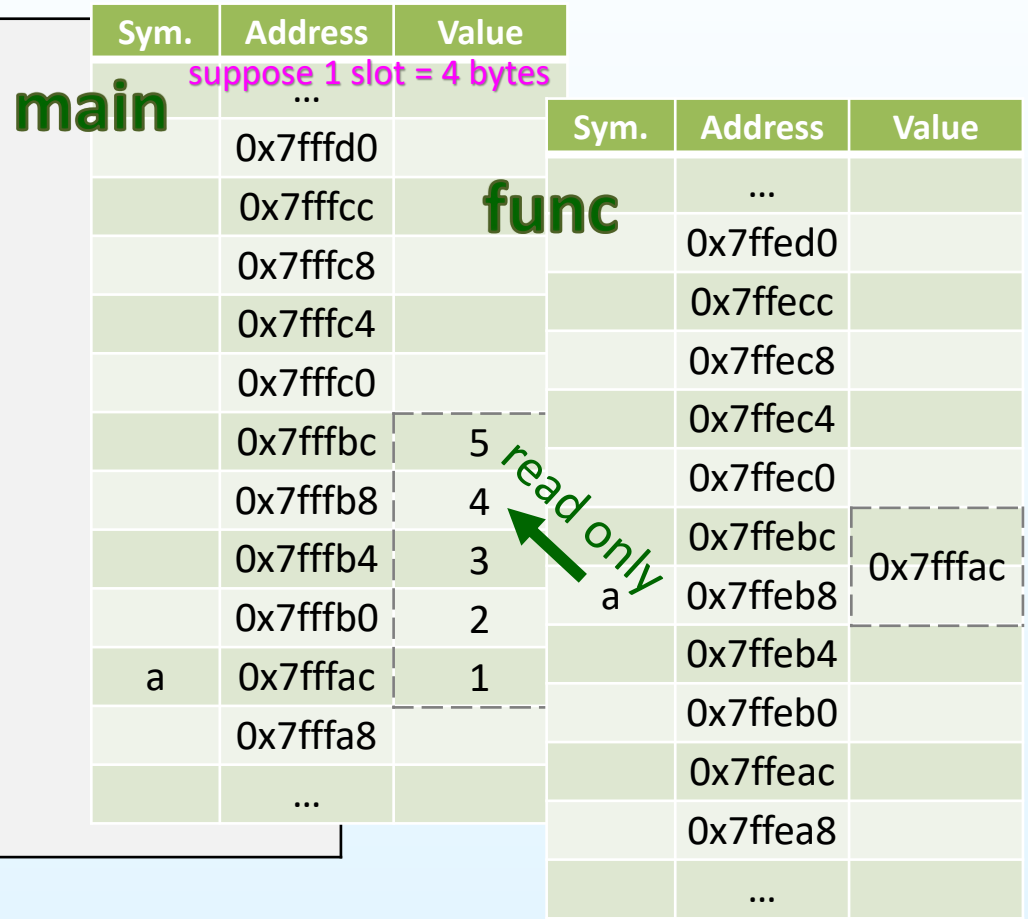


Passing Array to Function

- Should specify `const` if you don't want the function changing the value of any elements

```
#include <stdio.h>

void func(const int a[]) {
  a[3] *= 2;
}

int main(void) {
  int a[5] = {1,2,3,4,5};
  int i;

  func(a);
  for (i=0; i<5; i++)
    printf("%d ", a[i]);
  printf("\n");
  return 0;
}
```

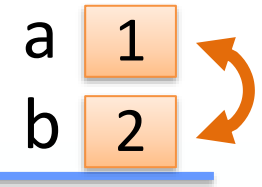
Cause a compile error



Outline

- Array
  - One and Multi-dimensional Arrays
  - Passing Array to Function
- Pointer
  - Address-of and Indirect Operators
  - Dynamic Memory Allocation
- String
  - Standard Library Functions

Problem: Swap Two Numbers



Program #1:

```c
#include <stdio.h>

int main(void) {
  int a = 1, b = 2, tmp;

  tmp = a;
  a = b;
  b = tmp;
  printf("%d %d\n", a, b);
  return 0;
}
```

```
2 1
```

Program #2:

```c
#include <stdio.h>

void swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}

int main(void) {
  int a = 1, b = 2;

  swap(a, b);
  printf("%d %d\n", a, b);
  return 0;
}
```

```
1 2
```

What is a Pointer?

- Pointer is a variable that contains the address of another variable (i.e., concept of indirection)

- We use a pointer for
  - Passing data to functions by reference
  - Handling arrays more efficiently
  - Performing dynamic memory allocation
  - Constructing complex data structures

The Address of Computer Memory

```c
int  num    = 932;
char letter = 'A';
char text[] = "abc";

printf("data : %d\n", num);
printf("addr : %p\n", &num);
printf("data : %c\n", letter);
printf("addr : %p\n", &letter);
printf("data : %s\n", text);
printf("addr : %p\n", &text);
printf("addr : %p\n", &(text[0]));
printf("addr : %p\n", text);
```

```
data : 932
addr : 0x7fffaf
data : A
addr : 0x7fffae
data : abc
addr : 0x7fffaa
addr : 0x7fffaa
addr : 0x7fffaa
```

Symbol	Address	Value
high …		
	0x7fffb2	932
	0x7fffb1	
	0x7fffb0	
num	0x7fffaf	
letter	0x7fffae	'A'
	0x7fffad	'\0'
	0x7fffac	'c'
	0x7fffab	'b'
text	0x7fffaa	'a'
	0x7fffa9	
	0x7fffa8	
low …		

The Pointer

Syntax:

```
datatype *variable = NULL;
```

```
int num;
int *numPtr = NULL;

numPtr = &num;
*numPtr = 520;
printf("data : %d\n", num);
printf("addr : %p\n", numPtr);
printf("data : %d\n", *numPtr);
printf("addr : %p\n", &numPtr);
```

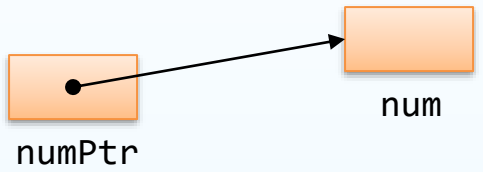
```
data : 520
addr : 0x7fffaf
data : 520
addr : 0x7fffa7
```

Symbol	Address	Value
high	…	
	0x7fffb2	520
	0x7fffb1	
	0x7fffb0	
num	0x7fffaf	
	0x7fffae	0x7fffaf
	0x7fffad	
	0x7fffac	
	0x7fffab	
	0x7fffaa	
	0x7fffa9	
	0x7fffa8	
numPtr	0x7fffa7	
low		

Pointer Operators

- Address-of operator (&)
  - Return address of operand

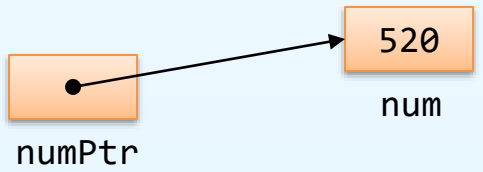
```
int num;
int *numPtr = NULL;

numPtr = &num;
```



numPtr points to num

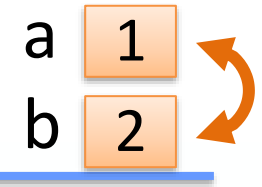
- Indirect operator (*)
  - Return alias of what its operand points to

```
int num;
int *numPtr = NULL;

numPtr = &num;
*numPtr = 520;
```



indirect assign to num

Problem: Swap Two Numbers



Program #1:

```c
#include <stdio.h>

int main(void) {
  int a = 1, b = 2, tmp;

  tmp = a;
  a = b;
  b = tmp;
  printf("%d %d\n", a, b);
  return 0;
}
```

```
2 1
```

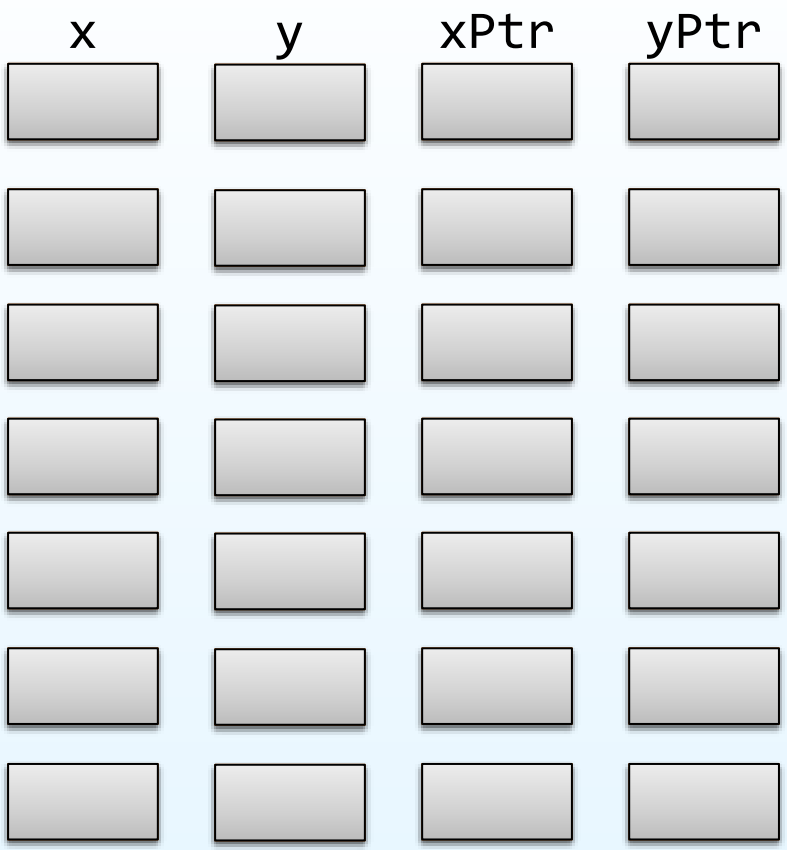
Program #2: Calling by reference

```c
#include <stdio.h>

void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int main(void) {
  int a = 1, b = 2;

  swap(&a, &b);
  printf("%d %d\n", a, b);
  return 0;
}
```

```
2 1
```

Example: Pointers

```
int x, y;
int *xPtr, *yPtr;

x = 0; y = 0;

xPtr = &x; *xPtr = 7;

yPtr = &y; *yPtr = 9;

*xPtr = *yPtr;

xPtr = yPtr; *xPtr = 5;

(*xPtr)++;
```



Pointer and Variable Type

- Notice that any pointer can only points to address of the same type variables

```
char  cLetter;
int   iSum;
float fRate;

char  *pLetter;
int   *pSum;
float *pRate;

pLetter = &cLetter;
pSum    = &iSum;
pRate   = &fRate;
```

Size of Pointer

- Also, can use the `sizeof` operator compute the size in byte of any pointers

```
char  *pLetter;
int   *pSum;
float *pRate;

printf("sizeof(char *)  = %ld\n", sizeof pLetter);
printf("sizeof(int *)   = %ld\n", sizeof pSum);
printf("sizeof(float *) = %ld\n", sizeof pRate);
```

```
sizeof(char *)  = 8
sizeof(int *)   = 8
sizeof(float *) = 8
```

Pointer and Array

```
int  num[4];
char letter[5];
int  *pNum = NULL;
char *pLetter = NULL;

pNum = &(num[0]); *pNum = 253;
pNum = &(num[1]); *pNum = 355;
pLetter = &(letter[0]);
*pLetter = 'A';
pLetter = &(letter[2]);
*pLetter = letter[0]+3;
pLetter++;
*pLetter = 'b';
pLetter -= 2;
*pLetter = 'd';
pNum += 2;
*pNum = 10;
pNum++;
*pNum = 200;
```

Pointer Arithmetic

Segmentation Fault !!!

C compiler never checks that the resulting pointer is valid!!!

Sym.	Address	Value
	…	
	0x7fffb2	10
	0x7fffb1	
	0x7fffb0	
3	0x7fffaf	
	0x7fffae	???
	0x7fffad	
	0x7fffac	
2	0x7fffab	
	0x7fffaa	355
	0x7fffa9	
	0x7fffa8	
1	0x7fffa7	
	0x7fffa6	253
	0x7fffa5	
	0x7fffa4	
num 0	0x7fffa3	
4	0x7fffa2	???
3	0x7fffa1	'b'
2	0x7fffa0	'D'
1	0x7fff9f	'd'
letter 0	0x7fff9e	'A'

Sym.	Address	Value
	0x7fff9d	0x7fffb3
	0x7fff9c	
	0x7fff9b	
	0x7fff9a	
	0x7fff99	
	0x7fff98	
	0x7fff97	
pNum	0x7fff96	
	0x7fff95	0x7fff9f
	0x7fff94	
	0x7fff93	
	0x7fff92	
	0x7fff91	
	0x7fff90	
	0x7fff8f	
pLetter	0x7fff8e	
	0x7fff8d	
	0x7fff8c	
	0x7fff8b	
	0x7fff8a	
	0x7fff89	
	…	

Pointer Arithmetic: Subtraction

```
int  num[4];
int  *p1 = NULL;
int  *p2 = NULL;

p1 = &(num[0]);
p2 = &(num[3]);
printf("Diff = %ld\n", p2-p1);
```

```
Diff = 3
```

```
int  num[4];
char letter[5];
int  *p1 = NULL;
char *p2 = NULL;

p1 = &(num[0]);
p2 = &(letter[0]);
printf("Diff = %ld\n", p2-p1);
```

Cause a compile error

Sym.	Address	Value
	…	
	0x7fffb2	???
	0x7fffb1	
	0x7fffb0	
3	0x7fffaf	
	0x7fffae	???
	0x7fffad	
	0x7fffac	
2	0x7fffab	
	0x7fffaa	???
	0x7fffa9	
	0x7fffa8	
1	0x7fffa7	
	0x7fffa6	???
	0x7fffa5	
	0x7fffa4	
num 0	0x7fffa3	

Sym.	Address	Value
	0x7fffa2	0x7fffa3
	0x7fffa1	
	0x7fffa0	
	0x7fff9f	
	0x7fff9e	
	0x7fff9d	
	0x7fff9c	
p1	0x7fff9b	
	0x7fff9a	0x7fffaf
	0x7fff99	
	0x7fff98	
	0x7fff97	
	0x7fff96	
	0x7fff95	
	0x7fff94	
p2	0x7fff93	
	…	

Recap: Passing Array to Function

- Function can reference through the parameter to reach (also, update) the array of the caller

```c
#include <stdio.h>

void func(int a[]) {
  a[3] *= 2;
}

int main(void) {
  int a[5] = {1,2,3,4,5};
  int i;

  func(a);
  for (i=0; i<5; i++)
    printf("%d ", a[i]);
  printf("\n");
  return 0;
}
```

```
1 2 3 8 5
```

In C, arrays and pointers are closely related; they are the same thing in some situations

e.g.,
```c
int a[5] = {0};
int *p = a;
p[2] = 5;
*(p+3) = 8;
```

a

0	0	5	8	0

For the function parameters, `int a[]` and `int *a` are identical, so that we can define

```c
void func(int *a) {
  *(a+3) *= 2;
}
```

or

```c
void func(int *a) {
  a[3] *= 2;
}
```

Arrays and Pointers

- Only difference:
  - For the statement `int a[5];` it sets aside five units of memory
  - While the statement `int *a;` it sets aside one pointer-sized unit of memory, so that you are expected to refer to the memory elsewhere

- You cannot do this:

```
int a[5];
int b[5];

a = b;
```

- But you can do like this:

```
int *a;
int *b;

a = b;
```

a points to the same address where b is pointing to

Recall: The `scanf()` Function

- The reason why we use the operator & is that the value read from a user is directly stored into the memory addressing by that parameter

```
char letter;
int  number;

scanf("%c %d", &letter, &number);
```

- However, for the array of characters (i.e., string), the parameter name refers to the address by itself

```
char text[10];

scanf("%s", text);
```

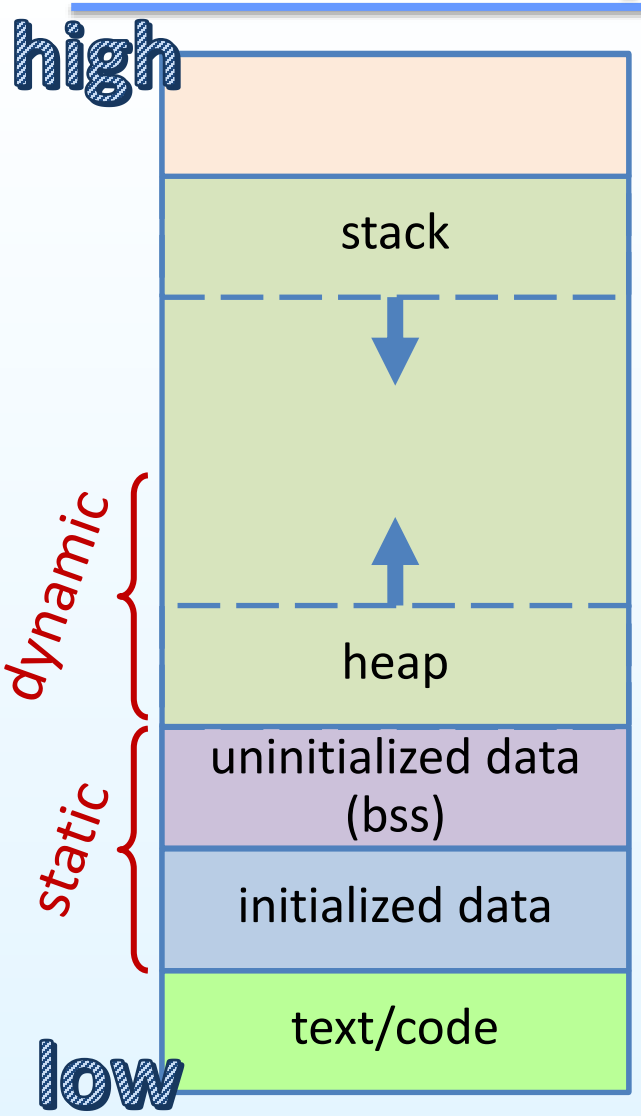
Memory Allocation

- Memory allocation is blocks of information in a memory system

- To allocate memory, if the memory management system finds sufficient free memory, it will allocate only as much memory as needed, keeping the rest available to satisfy future request

- There are two types: static and dynamic memory allocation

Static and Dynamic Memory Allocation



- In static memory allocation, requested memory will be allocated at compile time
  - e.g., global variables and variables declared as static
- In dynamic memory allocation, memory is allocated during runtime or program execution
  - The memory is allocated from the heap
  - A way to access this memory is through pointers

Why Allocating Memory Dynamically?

- When we do not know how much amount of memory would be needed for the program beforehand
- When we want data structures without any upper limit of memory space
- When we want to manage memory space more efficiently such as deallocating the unused memory to reduce waste and further use

Memory Allocation Functions

- C provides 4 library functions under `<stdlib.h>` header file to facilitate dynamic memory allocation

Function	Description
`malloc()`	Allocate requested size of bytes and return a pointer first byte of allocated space
`calloc()`	Allocate space for an array elements, initialize to zero and then return a pointer to memory
`free()`	Deallocate the previously allocated space
`realloc()`	Change the size of previously allocated space

The `malloc()` Function

- Dynamically allocate a single large block of memory with the specified size
- Return a pointer of type `void` which can be cast into a pointer of any form
- Initialize each block with default garbage value

Syntax:

```
void *malloc(size_t size);
```

The `free()` Function

- Dynamically deallocate the previous allocated memory
- Help to reduce wastage of memory by freeing it

Syntax:

```
void free(void *ptr);
```

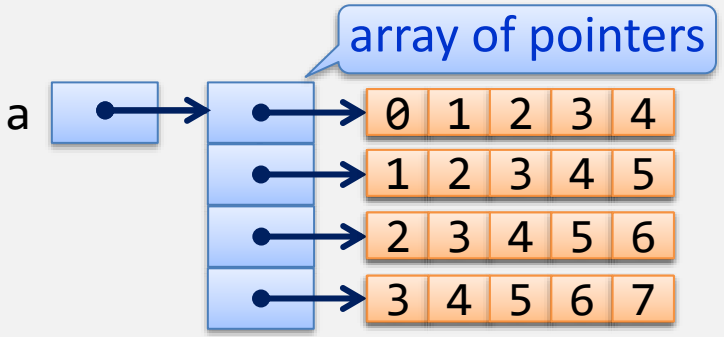
Example: Dynamic Memory Allocation

```
1:  #include <stdio.h>
2:  #include <stdlib.h>
3:  #define  SIZE 5
4:
5:  int main(void) {
6:    int *a = NULL;
7:    int i;
8:
9:    a = (int *)malloc(sizeof(int) * SIZE);
10:   for (i=0; i<SIZE; i++)
11:     a[i] = i+1;
12:
13:   for (i=0; i<SIZE; i++)
14:     printf("a[%d] = %d\n", i, a[i]);
15:
16:   free(a);
17:   return 0;
18: }
```

```
a[0] = 1
a[1] = 2
a[2] = 3
a[3] = 4
a[4] = 5
```

Example: 2D Array Allocation

```
1:  #include <stdio.h>
2:  #include <stdlib.h>
3:  #define  ROW 4
4:  #define  COL 5
5:
6:  int main(void) {
7:    int **a = NULL;
8:    int i, j;
9:
10:   a = (int **)malloc(sizeof(int *) * ROW);
11:   for (i=0; i<ROW; i++) {
12:     a[i] = (int *)malloc(sizeof(int) * COL);
13:     for (j=0; j<COL; j++)
14:       a[i][j] = i+j;
15:   }
16:
17:   for (i=0; i<ROW; i++)
18:     free(a[i]);
19:   free(a);
20:   return 0;
21: }
```



Outline

- Array
  - One and Multi-dimensional Arrays
  - Passing Array to Function
- Pointer
  - Address-of and Indirect Operators
  - Dynamic Memory Allocation
- String
  - Standard Library Functions

Strings in C Program

- An array of characters

Program #1:

```
1:  #include <stdio.h>
2:
3:  int main(void) {
4:    char text[] = "hello";
5:
6:    printf("%c\n", text[0]);
7:    printf("%s\n", text);
8:    printf("%ld\n", sizeof text);
9:    printf("%ld\n", sizeof *text);
10:   text[0] = 'H';
11:   // text = "world";
12:   printf("%s\n", text);
13:   return 0;
14: }
```

Cause a compile error

```
h
hello
6
1
Hello
```

- A pointer to characters

Program #2:

```
1:  #include <stdio.h>
2:
3:  int main(void) {
4:    char *text = "hello";
5:
6:    printf("%c\n", text[0]);
7:    printf("%s\n", text);
8:    printf("%ld\n", sizeof text);
9:    printf("%ld\n", sizeof *text);
10:   // text[0] = 'H';
11:   text = "world";
12:   printf("%s\n", text);
13:   return 0;
14: }
```

Cause a segmentation fault

```
h
hello
8
1
world
```

Standard Library Functions for String

Some libraries are mentioned here

- The `<stdlib.h>` header file
  - String conversion
- The `<stdio.h>` header file
  - Standard input/output for string
- The `<string.h>` header file
  - String manipulation
  - String comparison
  - String searching
  - String length computation

The <stdlib.h> Header File

- Convert string of digits to integer or floating-point values

Function Prototype	Description
`double atof(const char *nptr);`	Convert the string `nptr` to `double`
`int atoi(const char *nptr);`	Convert the string `nptr` to `int`
`long atol(const char *nptr);`	Convert the string `nptr` to `long`
`long long atoll(const char *nptr);`	Convert the string `nptr` to `long long`

```
char num_str1[] = "111";
char num_str2[] = "222";

printf("%d\n", atoi(num_str1) + atoi(num_str2));
```

```
333
```

The <stdio.h> Header File

- Manipulate character and string data

Function Prototype	Description
`int getchar(void);`	Input the next character from the standard input
`char *gets(char *s);`	Input characters from the standard input into array `s`
`int putchar(int c);`	Print the character stored in `c`
`int puts(const char *s);`	Print the string `s`
`int sprintf(char *str, const char *format, ...);`	Equivalent to `printf()`, but storing output into array `str`
`int sscanf(const char *str, const char *format, ...);`	Equivalent to `scanf()`, but reading input from the string `str`

Example: I/O Functions for Strings

```
char   text[10];
char   grade;
double point;

sprintf(text, "Get %c %.0lf", 'A', 4.0);
sscanf(text, "Get %c %lf", &grade, &point);
printf("%s\n", text);
printf("%c\n", grade);
printf("%lf\n", point);
sprintf(text, "Hello world!!!");
```

Segmentation Fault !!!

```
Get A 4
A
4.000000
```

The <string.h> Header File

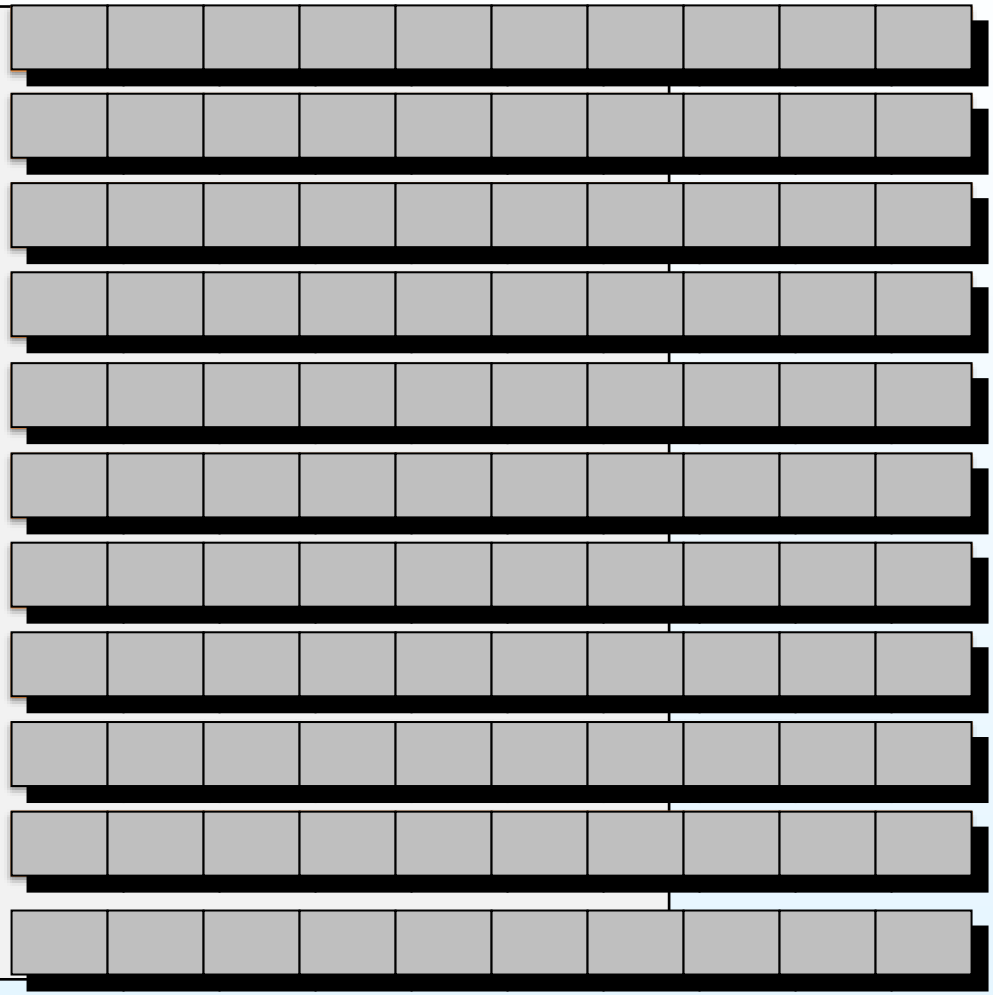
- Manipulate string data, search strings, tokenize strings

Function Prototype	Description
`char *strcpy(char *dest, const char *src);`	Copy the string `src` into array `dest`
`char *strncpy(char *dest, const char *src, size_t n);`	Copy at most `n` characters of the string `src` into array `dest`
`char *strcat(char *dest, const char *src);`	Append the string `src` to array `dest`; the first character of `src` overwrites the null character of `dest`
`char *strncat(char *dest, const char *src, size_t n);`	Append at most `n` characters of the string `src` to array `dest`; the first character of `src` overwrites the null character of `dest`

Other functions: `strstr()`, `strtok()`, `strdup()`, etc.

Example: String Manipulation

```
char text[10];
strcpy(text, "My world");
strncpy(text, "ABCDE", 3);
strncpy(text, "ABCDE", 5);
strncpy(text, "ABCDE", 6);
strncpy(text+3, "ABCDE", 3);
strncpy(text+8, "ABCDE", 1);
strcpy(text, "ABCDE");
strcat(text, "123");
strncat(text, "copy", 1);
strncat(text, "string", 3);
```



The <string.h> Header File

- Compare numeric ASCII codes of characters in string

Function Prototype	Description
`int strcmp(const char *s1, const char *s2);`	Compare the string `s1` to `s2`; return a negative number if `s1<s2`, zero if `s1==s2`, or a positive number if `s1>s2`
`int strncmp(const char *s1, const char *s2, size_t n);`	Compare up to `n` characters of string `s1` to `s2`; return a negative number if `s1<s2`, zero if `s1==s2`, or a positive number if `s1>s2`

Example: String Comparison

Program #1:

```
 1: #include <stdio.h>
 2:
 3: int main(void) {
 4:   char text1[] = "hello";
 5:   char text2[] = "hello";
 6:
 7:   if (text1 == text2)
 8:     printf("Equal\n");
 9:   else
10:     printf("Not equal\n");
11:   return 0;
12: }
```

```
Not equal
```

Program #2:

```
 1: #include <stdio.h>
 2: #include <string.h>
 3:
 4: int main(void) {
 5:   char text1[] = "hello";
 6:   char text2[] = "hello";
 7:
 8:   if (strcmp(text1,text2) == 0)
 9:     printf("Equal\n");
10:   else
11:     printf("Not equal\n");
12:   return 0;
13: }
```

```
Equal
```

The <string.h> Header File

- Find the length of string

Function Prototype	Description
`size_t strlen(const char *s);`	Return the number of characters (before null character) in the string `s`

```
char   text[10] = "hello";

printf("%ld\n", sizeof text);
printf("%ld\n", strlen(text));
```

```
10
5
```

Any Question?