Review: Algorithms & Programming (Part 2)

Arrays, Structs, Functions, Header Files, Pointers

Assoc.Prof.Dr. Fatih ABUT

ARRAYS

- An array is a collection of elements of the same type that are referenced by a common name.
- Compared to the basic data type (`int`, `float` & `char`) it is an aggregate or derived data type.
- All the elements of an array occupy a set of contiguous memory locations.
- Why need to use array type?
- Consider the following issue:

"We have a list of 1000 students' marks of an integer type. If using the basic data type (int), we will declare something like the following…"

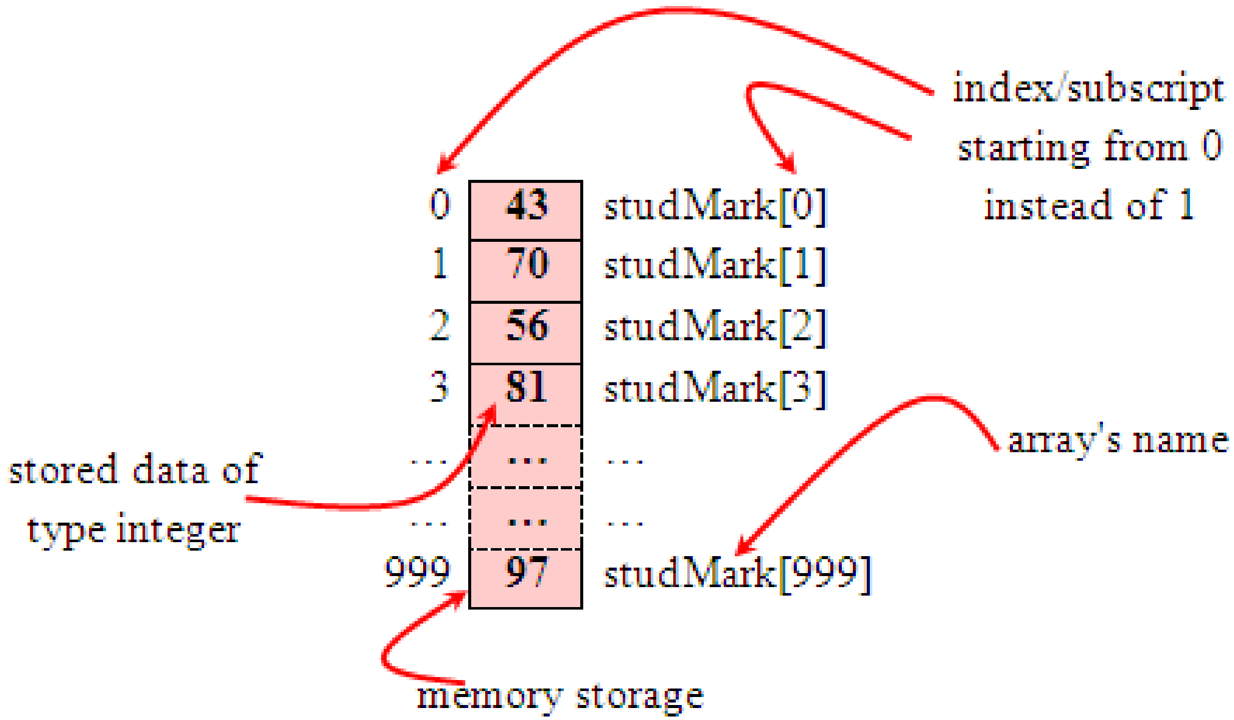
```
int  studMark0, studMark1, studMark2, ..., studMark999;
```

ARRAYS

- By using an array, we just declare like this,

```
int  studMark[1000];
```

- This will reserve 1000 contiguous memory locations for storing the students’ marks.
- Graphically, this can be depicted as in the following figure.



ARRAYS

One Dimensional Array: Declaration

- Dimension refers to the array's size, which is how big the array is.
- A single or one dimensional array declaration has the following form,

```
array_element_data_type array_name[array_size];
```

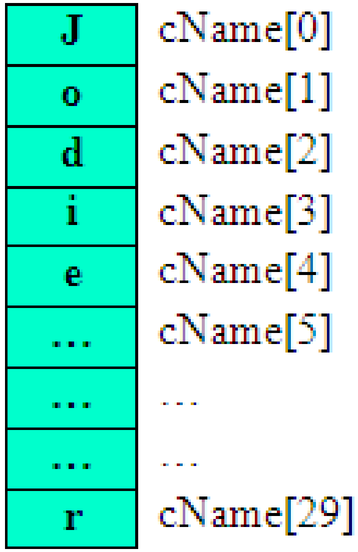
- Here, *array_element_data_type* define the base type of the array, which is the type of each element in the array.
- *array_name* is any valid C / C++ identifier name that obeys the same rule for the identifier naming.
- *array_size* defines how many elements the array will hold.

ARRAYS

- For example, to declare an array of 30 characters, that construct a people name, we could declare,

```
char   cName[30];
```

- Which can be depicted as follows,



- In this statement, the array character can store up to 30 characters with the first character occupying location `cName[0]` and the last character occupying `cName[29]`.
- Note that the <u>index runs from `0` to `29`</u>. In C, an index always <u>starts from `0`</u> and ends with <u>array's (size-1)</u>.
- So, take note the difference between the <u>array size and subscript/index</u> terms.

ARRAYS

- Examples of the one-dimensional array declarations,

```
int      xNum[20], yNum[50];
float    fPrice[10], fYield;
char     chLetter[70];
```

- The first example declares two arrays named `xNum` and `yNum` of type `int`. Array `xNum` can store up to 20 integer numbers while `yNum` can store up to 50 numbers.
- The second line declares the array `fPrice` of type `float`. It can store up to 10 floating-point values.
- `fYield` is basic variable which shows array type can be declared together with basic type provided the type is similar.
- The third line declares the array `chLetter` of type char. It can store a string up to 69 characters.
- Why 69 instead of 70? Remember, a string has a <u>null terminating character (`\0`) at the end</u>, so we must reserve for it.

ARRAYS

- An array may be initialized at the time of declaration.

- Initialization of an array may take the following form,

```
type   array_name[size] = {a_list_of_value};
```

- For example:

```
int    idNum[7] = {1, 2, 3, 4, 5, 6, 7};
float  fFloatNum[5] = {5.6, 5.7, 5.8, 5.9, 6.1};
char   chVowel[6] = {'a', 'e', 'i', 'o', 'u', '\0'};
```

- The first line declares an integer array `idNum` and it immediately assigns the values 1, 2, 3, ..., 7 to `idNum[0]`, `idNum[1]`, `idNum[2]`,. . ., `idNum[6]` respectively.

- The second line assigns the values 5.6 to `fFloatNum[0]`, 5.7 to `fFloatNum[1]`, and so on.

- Similarly the third line assigns the characters '`a`' to `chVowel[0]`, '`e`' to `chVowel[1]`, and so on. Note again, for characters we must use the single apostrophe/quote (') to enclose them. Also, the last character in `chVowel` is `NULL` character ('`\0`').

Example #1: Finding the smallest element in an array

```c
#include <stdio.h>

int main() {
    int array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
    int i, smallest;

    smallest = array[0];

    for(i = 1; i < 10; i++) {
        if( smallest > array[i] )
            smallest = array[i];
    }

    printf("Smallest element of array is %d", smallest);

    return 0;
}
```

Example #2: Finding the second largest element in an array

```c
#include <stdio.h>

int main() {
    int array[10] = {101, 11, 3, 4, 50, 69, 7, 8, 9, 0};
    int i, largest, second;

    if(array[0] > array[1]) {
        largest = array[0];
        second  = array[1];
    } else {
        largest = array[1];
        second  = array[0];
    }

    for(i = 2; i < 10; i++) {
        if( largest < array[i] )    {
            second = largest;
            largest = array[i];
        } else if( second < array[i] ) {
            second =  array[i];
        }
    }

    printf("Largest - %d \nSecond - %d \n", largest, second);

    return 0;
}
```

ARRAYS

Two Dimensional/2D Arrays

- A two dimensional array has <u>two subscripts/indexes</u>.
- The <u>first subscript</u> refers to the <u>row</u>, and the <u>second</u>, to the <u>column</u>.
- Its declaration has the following form,

```
data_type    array_name[1st dimension size][2nd dimension size];
```

- For examples,

```
int      xInteger[3][4];
float    matrixNum[20][25];
```

- The first line declares `xInteger` as an integer array with 3 <u>rows</u> and 4 <u>columns</u>.
- Second line declares a `matrixNum` as a floating-point array with 20 <u>rows</u> and 25 <u>columns</u>.

ARRAYS

- For an array `Name[6][10]`, the array size is 6 x 10 = 60 and equal to the number of the colored square. In general, for

```
array_name[x][y];
```

- The array size is = First index **x** second index = xy.
- This also true for other array dimension, for example three-dimensional array,

`array_name[x][y][z];` => First index **x** second index **y** third index = xyz

- For example,

`ThreeDimArray[2][4][7]` = 2 x 4 x 7 = 56.

- And if you want to illustrate the 3D array, it could be a cube with wide, long and height dimensions.

```c
#include <stdio.h>

int main(){

int sum=0;

//Ahmet, Mehmet, Ayse
//AP1, AP2, OOP, SWENG
int scores[3][4] = {
    {43, 31, 12, 32},   /*  initializers for row indexed by 0 */
    {66, 5, 6, 7},   /*  initializers for row indexed by 1 */
    {8, 4, 10, 11}   /*  initializers for row indexed by 2 */
};


for(int i=0; i<4;i++){
     sum += scores[1][i];
}

printf("avg. score of std Mehmet: %f\n\n", sum/4.0);


sum = 0;

for(int i=0; i<3;i++){
     sum += scores[i][2]; //2=OOP
}

printf("average score of OOP: %f\n\n", sum/3.0);


int highest = scores[0][0];

for(int i=0; i<3;i++){

     for(int j=0; j<4;j++){

          if(highest < scores[i][j]){
               highest = scores[i][j];
          }

     }
}

printf("Highest score: %d\n", highest);
```

	AP1	AP2	OOP	SWENG
Ahmet	43	31	12	32
Mehmet	66	5	6	7
Ayşe	8	4	10	11

Example 2-dimensional Array

Structs

Definition, Nested Structs, Struct Arrays

Structures

- There is no class in C, but we may still want non-homogenous structures
  - So, we use the struct construct
    - struct for structure
- A struct is a data structure that comprises multiple types, each known as a member
    - each member has its own unique name and a defined type
- Example:
    - A student may have members: name (char[ ]), age (int), GPA (float or double), sex (char), major (char[ ]), etc

The struct Definition

- struct is a keyword for defining a structured declaration
- Format:

```
struct name {
    type1 name1;
    type2 name2;
    …
};
```

name1 and name2 are *members* of name

- name represents this structure's tag and is optional
  - we can either provide name
  - or after the } we can list variables that will be defined to be this structure
- We can also use typedef to declare name to be this structure and use name as if it were a built-in type
  - typedef will be covered later in these notes

Examples

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

struct point p1, p2;
```

p1 and p2 are both points, containing an x and a y value

```
struct {
    int x;
    int y;
}  p1, p2;
```

p1 and p2 both have the defined structure, containing an x and a y, but do not have a tag

```
struct  point {
    int x;
    int y;
}  p1, p2;
```

same as the other two versions, but united into one set of code, p1 and p2 have the tag point

For the first and last sets of code, point is a defined tag and can be used later (to define more points, or to declare a type of parameter, etc) but in the middle code, there is no tag, so there is no way to reference more examples of this structure

Accessing structs

- A struct is much like an array
  - The structure stores multiple data
    - You can access the individual data, or you can reference the entire structure
  - To access a particular member, you use the . operator
    - as in student.firstName or p1.x and p1.y
  - To access the struct itself, reference it by the variable name
    - Legal operations on the struct are assignment, taking its address with &, copying it, and passing it as a parameter
      - p1 = {5, 10}; // same as p1.x = 5; p1.y = 10;
      - p1 = p2; // same as p1.x = p2.x; p1.y = p2.y;

typedef

- typedef is used to define new types
  - The format is
    - typedef description name;
  - Where description is a current type or a structural description such as an array declaration or struct declaration
  - Examples:

```
typedef  int Length;   // Length is now equivalent to the type int

typedef char[10] String;     // String is the name of a type for a character array of size 10

typedef struct student {      // declares a node structure that contains
        int mark [2];
        char name [10];
        float average;
 }aStudent;

aStudent s1;      // this allows us to refer to aStudent instead of struct node
```

Functions and Header Files in C

Introduction to Functions

- A complex problem is often easier to solve by dividing it into several smaller parts, each of which can be solved by itself.
- This is called ***structured*** programming.
- These parts are sometimes made into ***functions*** in C++.
- **`main()`** then uses these functions to solve the original problem.

Advantages of Functions

- Functions separate the concept (what is done) from the implementation (how it is done).
- Functions make programs easier to understand.
- Functions can be called several times in the same program, allowing the code to be reused.

C Functions

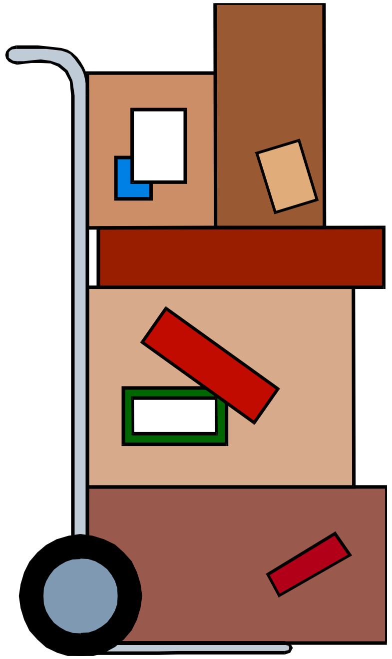
- C allows the use of both internal (user-defined) and external functions.

- External functions (e.g., **`abs`**, **`ceil`**, **`floor`**, **`rand`**, sqrt, etc.) are usually grouped into specialized headers (e.g., **`stdlib.h`**, **`math.h`**, etc.)

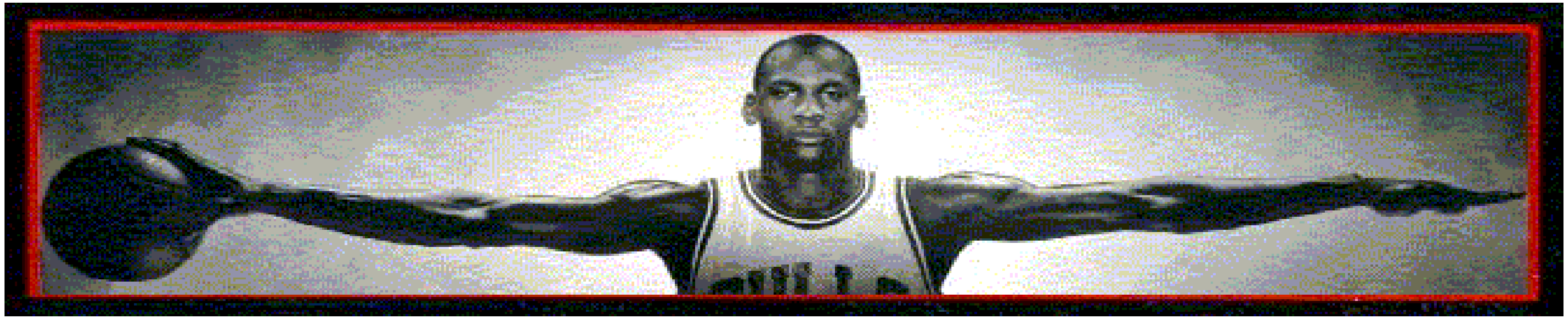
User-Defined Functions

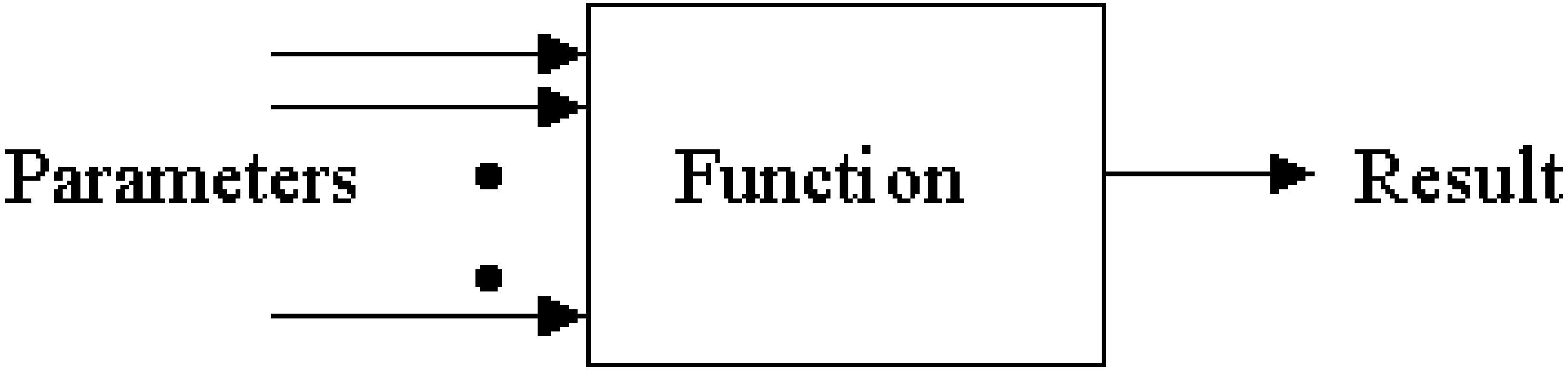
- C programs usually have the following form:

```
// include statements
// function prototypes
// main() function
// function definitions
```



Function Input and Output





Function Definition

A function definition has the following syntax:

<type> <function name>(<parameter list>){

<local declarations>

<sequence of statements>

}

For example: Definition of a function that computes the absolute value of an integer:

```
int absolute(int x){
    if (x >= 0)  return x;
    else     return -x;
}
```

Function Call

- A **function call** has the following syntax:

```
<function name>(<argument list>)
```

Example: int distance = absolute(-5);

- The result of a function call is a value of type <type>

Arguments/Parameters

- one-to-one correspondence between the arguments in a function call and the parameters in the function definition.

```
int argument1;
double argument2;
// function call (in another function, such as main)
result = thefunctionname(argument1, argument2);

// function definition
int thefunctionname(int parameter1, double parameter2){
// Now the function can use the two parameters
// parameter1 = argument 1, parameter2 = argument2
```

Absolute Value

```c
#include <stdio.h>

int absolute (int);// function prototype for absolute()

int main(){
    int num, answer;
     printf("Enter an integer (0 to stop): ");
    scanf(%d, &num);

    while (num!=0){
        answer = absolute(num);
        printf("The absolute value of %d is: %d", num, answer);
        scanf(%d, &num);
    }
   return 0;
}

// Define a function to take absolute value of an integer
int absolute(int x){
    if (x >= 0) return x;
    else return -x;
}
```

Function Prototype

- The function prototype declares the input and output parameters of the function.

- The function prototype has the following syntax:

```
<type> <function name>(<type list>);
```

- Example: A function that returns the absolute value of an integer is: `int absolute(int);`

Function Definition

- The function definition can be placed anywhere in the program after the function prototypes.

- If a function definition is placed in front of `main()`, there is no need to include its function prototype.

Absolute Value (alternative)

- Note that it is possible to omit the function prototype if the function is placed before it is called.

```
#include "stdio.h"

// Define a function to take absolute value of an integer
int absolute(int x){
    if (x >= 0) return x;
    else return -x;
}

int main(){
    int num, answer;
      printf("Enter an integer (0 to stop): ");
    scanf(%d, &num);

     while (num!=0){
        answer = absolute(num);
        printf("The absolute value of %d is: %d", num, answer);
        scanf(%d, &num);
     }
     return 0;
}
```

Function of three parameters

```
#include <stdio.h>

double total_second( int hour, double minutes, double second)
{
    return hour*3600 + minutes * 60 + second;
}

int main(){
    double tot_sec = total_second(1,1.5, 2);
    printf(" %f",tot_sec);
    return 0;
}
```

Arrays as Funtion Parameters

```
// Program to calculate the average of ages by passing an array to a function

#include <stdio.h>
float average(int age[]);

int main()
{
    float avg;
    int age[] = {23, 55, 22, 3, 40, 18};
    avg = average(age); // Only name of an array is passed as an argument
    printf("Average age = %.2f", avg);
    return 0;
}

float average(int age[])
{
    int i;
    float avg, sum = 0.0;
    for (i = 0; i < 6; ++i) {
        sum += age[i];
    }
    avg = (sum / 6);
    return avg;
}
```

Return Array via Functions

```
/* main function to call above defined function */
int main () {

   /* a pointer to an int */
   int *p;
   int i;

   p = getRandom();

   for ( i = 0; i < 10; i++ ) {
      printf( "*(p + %d) : %d\n", i, *(p + i));
   }

   return 0;
}
```

```
#include <stdio.h>

/* function to generate and return random
numbers */
int * getRandom( ) {

   static int r[10];
   int i;

   /* set the seed */
   srand( (unsigned)time( NULL ) );

   for ( i = 0; i < 10; ++i) {
      r[i] = rand();
      printf( "r[%d] = %d\n", i, r[i]);
   }

   return r;
}
```

Return Multiple Values via Functions (1)

```
#include <stdio.h>
#include <math.h>    // for sin() and cos()

void getSinCos(double degrees, double &sinOut, double &cosOut)
{
    // sin() and cos() take radians, not degrees, so we need to convert
    const double pi = 3.14;
    double radians = degrees * pi / 180.0;

    sinOut = sin(radians);
    cosOut = cos(radians);
}

int main()
{
    double sin = 0.0;
    double cos = 0.0;

    getSinCos(30.0, sin, cos);

    printf("The sin is %lf\n", sin);
    printf("The cos is %lf", cos);

    return 0;
}
```

Return Multiple Values via Functions (2)

```
#include <stdio.h>

void getAreaCirc(double radius, double &areaOut, double &circumOut)
{
    const double pi = 3.14;

    areaOut = pi * radius * radius;
    circumOut = 2 * pi * radius;
}

int main()
{
    double areaOut = 0.0;
    double circumOut = 0.0;

    getAreaCirc(3.0, areaOut, circumOut);

    printf("The area of circle is %f\n", areaOut);
    printf("The circum of circle is %f\n", circumOut);

    return 0;
}
```

Structs as Funtion Parameters

- We may pass structs as parameters and functions can return structs
  - Passing as a parameter:
    - void foo(struct point x, struct point y) {…}
      - notice that the parameter type is not just the tag, but preceded by the reserved word struct
  - Returning a struct:

```
struct point createPoint(int a, int b)
{
    struct point temp;
    temp.x = a;
    temp.y = b;
    return temp;
}
```

Inputting a struct in a Function

- We will need to do multiple inputs for our struct
  - Rather than placing all of the inputs in main, let's write a separate function to input all the values into our struct
    - The code to the right does this
  - But how do we pass back the struct?
    - Remember C uses pass by copy
      - the struct is *copied* into the function so that p in the function is different from y in main
      - after inputting the values into p, nothing is returned and so y remains {0, 0}<<<<

```
#include <stdio.h>

struct point  {
        int x;
        int y;  };

void getStruct(struct point);
void output(struct point);

void main( )    {
        struct point y = {0, 0};
        getStruct(y);
        output(y);  }

void getStruct(struct point p)   {
        scanf("%d", &p.x);
        scanf("%d", &p.y);
        printf("%d, %d", p.x, p.y);  }

void output(struct point p)   {
        printf("%d, %d", p.x, p.y);  }
```

One Solution for Input

- In our previous solution, we passed the struct into the function and manipulated it in the function, but it wasn't returned
  - Why not? Because what was passed into the function was a copy, not a reference (or so-called pointer)
    - So structs differ from arrays as structs are not pointed to
- In our input function, we can instead create a temporary struct and return the struct rather than having a void function

```
void main( )
{
        struct point y = {0, 0};
        y = getStruct( );
        output(y);
}
```

```
struct point getStruct( )
{
        struct point temp;
        scanf("%d", &temp.x);
        scanf("%d", &temp.y);
        return temp;
}
```

Nested structs

- In order to provide modularity, it is common to use already-defined structs as members of additional structs
- Recall our point struct, now we want to create a rectangle struct
  - the rectangle is defined by its upper left and lower right points

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

struct rectangle {
    struct point pt1;
    struct point pt2;
}
```

If we have

```
struct rectangle r;
```

Then we can reference

```
r.pt1.x, r.pt1.y,
r.pt2.x and r.pt2.y
```

Arrays of structs

- To declare an array of structs (once you have defined the struct):
  - struct rectangle rects[10];
  - rects now is a group of 10 structures (that consist each of two points)
  - You can initialize the array as normal where each struct is initialized as a { } list as in {5, 3} for a point or {{5, 3}, {8, 2}} for a rectangle

Example

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

struct rectangle {
     struct point p1;
     struct point p2;
};

void printRect(struct rectangle r)
{
        printf("<%d, %d> to <%d, %d>\n", r.p1.x, r.p1.y, r.p2.x, r.p2.y);
}

void main( )
{
    int i;
    struct rectangle rects[ ] = {{{1, 2}, {3, 4}}, {{5, 6}, {7, 8}}}; // 2 rectangles
    for(i=0;i<2;i++) printRect(rects[i]);
}
```

Function Overloading

```c
#include <stdio.h>
#include <stdlib.h>

int add(int a, int b) {
    printf("method 1\n");
    return a + b;
}


int add(int a, char b) {
    printf("method 2\n");
    return a + b;
}


int add(char a, int b) {
    printf("method 3\n");
    return a + b;
}


float add(int a, int b, float c) {
    printf("method 4\n");
    return a + b + c;
}


float add(int a, float b) {
    printf("method 5\n");
    return a + b;
}


double add(int a, double b) {
    printf("method 6\n");
    return a + b;
}

```

```c
int main() {

    int i1 = 1, i2 = 2;
    char c = 'A';
    float f=3.2;
    double d = 4.3;


    printf("%d\n", add(i1, i2)); // 3

    printf("%d\n", add(i1, c));

    printf("%d\n", add('c', i2));

    printf("%f\n", add(i1, i2, f));

    printf("%f\n", add(i1, f));

    printf("%f\n", add(i1, d));

    return 0;
}
```

Recursive Functions

```c
/*
Example: Program to calculate power using recursion
*/

#include <stdio.h>

int power(int n1, int n2);

int main()
{
    int base, powerRaised, result;

    printf("Enter base number: ");
    scanf("%d",&base);

    printf("Enter power number(positive integer): ");
    scanf("%d",&powerRaised);

    result = power(base, powerRaised);

    printf("%d^%d = %d", base, powerRaised, result);
    return 0;
}

int power(int base, int powerRaised)
{
    if (powerRaised != 0)
        return (base*power(base, powerRaised-1));
    else
        return 1;
}
```

```c
/*
Example: Factorial of a Number Using Recursion
*/

#include <stdio.h>
long int multiplyNumbers(int n);

int main()
{
    int n;
    printf("Enter a positive integer: ");
    scanf("%d", &n);
    printf("Factorial of %d = %ld", n, multiplyNumbers(n));
    return 0;
}

long int multiplyNumbers(int n)
{
    if (n >= 1)
        return n*multiplyNumbers(n-1);
    else
        return 1;
}
```

```c
/*
Example: Sum of Natural Numbers Using Recursion
*/

#include <stdio.h>
int sum(int n);

int main()
{
    int number, result;

    printf("Enter a positive integer: ");
    scanf("%d", &number);

    result = sum(number);

    printf("sum = %d", result);
    return 0;
}

int sum(int num)
{
    if (num!=0)
        return num + sum(num-1);
    else
        return num;
```

Recursive Funcs
vs.
Iterations?

Header files

- In applications with multiple *C* programs, function prototypes are typically provided in *header files*
  - I.e., the ‘`.h`’ files that programmers include in their code
- Grouped by related functions and features
  - To make it easier for developers to understand
  - To make it easier for team development
  - To make a package that can be used by someone else

Typical *C* Programming Style

- A lot of small *C* programs, rather than a few large ones
    - Each `.c` file contains closely related functions
    - Usually a small number of functions
- Header files to tie them together

Example: Calculate the permulation and combination by using header files!

main.c
main.o
Makefile.win
perm-comb.c
perm-comb.dev
perm-comb.exe
perm-comb.h
perm-comb.layout
perm-comb.o
permutation.exe

main.c — perm_comb

```c
#include <stdio.h>
#include "perm-comb.h"

int main(int argc, char *argv[]) {

    int n, r;

    printf("Enter n: ");
    scanf("%d", &n);

    printf("Enter r: ");
    scanf("%d", &r);

    printf("Permutation = %ld\n", permutation(n, r));
    printf("Combination = %ld", combination(n, r));

    return 0;
}
```

perm-comb.h — perm_comb

```c
long int permutation(int n, int r);
long int combination(int n, int r);
long int factorial(int num);
```

```c
#include "perm-comb.h"

long permutation(int n, int r)
{
    return factorial(n) / factorial(n-r);
}

long combination(int n, int r)
{
    return permutation(n, r) / factorial(r);
}

long factorial(int num)
{
    long long fact = 1;

    while(num > 0)
    {
        fact *= num;
        num--;
    }

    return fact;
}
```

`#include`

- `#include <foo.h>`
  - Search the system's directories *in order* for a file of the name `foo.h`
  - Directories can be added with '`-I`' switch to `gcc` command
    - E.g., `gcc -I myProject/include foo.c`
    - Precedes system directories in search order
- `#include "foo.h"`
  - Search the directory where the source program is found first, *before* `-I` and system directories

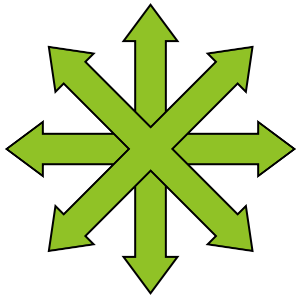
Pointers in C

Computer Memory

- Computers store data in memory slots
- Each slot has an *unique address*
- Variables store their values like this:

Addr	Content	Addr	Content	Addr	Content	Addr	Content
1000	i: 37	1001	j: 46	1002	k: 58	1003	m: 74
1004	a[0]: ‘a’	1005	a[1]: ‘b’	1006	a[2]: ‘c’	1007	a[3]: ‘\0’
1008	ptr: 1001	1009	…	1010		1011	

Addressing Concept



- Pointer stores the **address** of another entity
- It **refers** to a memory location

```
int i = 5;
int *ptr;                        /* declare a pointer variable */
ptr = &i;                        /* store address-of i to ptr */
printf(“*ptr = %d\n”, *ptr);     /* refer to referee of ptr */
```

What actually *ptr* is?

- **ptr** is a variable storing **an address**
- ptr is **NOT** storing the actual value of i

```
int i = 5;
int *ptr;
ptr = &i;
printf("i = %d\n", i);
printf("*ptr = %d\n", *ptr);
printf("ptr = %p\n", ptr);
```

Twin Operators

- &: Address-of operator
  - Get the *address* of an entity
    - e.g. `ptr = &j;`

Addr	Content	Addr	Content	Addr	Content	Addr	Content
1000	i: 40	1001	j: 33	1002	k: 58	1003	m: 74
1004	ptr: 1001	1005		1006		1007	

Twin Operators

- *: De-reference operator
  - Refer to the *content* of the referee
    - e.g. `*ptr = 99;`

Addr	Content	Addr	Content	Addr	Content	Addr	Content
1000	i: 40	1001	j: 99	1002	k: 58	1003	m: 74
1004	ptr: 1001	1005		1006		1007	

An Illustration

```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	5
j	int	integer variable	10

An Illustration

```
int i = 5, j = 10;
int *ptr;    /* declare a pointer-to-integer variable */
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	5
j	int	integer variable	10
ptr	int *	integer pointer variable	

An Illustration

```
int i = 5, j = 10;
int *ptr;
int **pptr;  /* declare a pointer-to-pointer-to-integer variable */
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	5
j	int	integer variable	10
ptr	int *	integer pointer variable	
pptr	int **	integer pointer pointer variable	
		Double Indirection	

An Illustration

```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;    /* store address-of i to ptr */
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	5
j	int	integer variable	10
ptr	int *	integer pointer variable	address of i
pptr	int **	integer pointer pointer variable	
*ptr	int	de-reference of ptr	5

An Illustration

```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr; /* store address-of ptr to pptr */
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	5
j	int	integer variable	10
ptr	int *	integer pointer variable	address of i
pptr	int **	integer pointer pointer variable	address of ptr
*pptr	int *	de-reference of pptr	value of ptr (address of i)

An Illustration

```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	3
j	int	integer variable	10
ptr	int *	integer pointer variable	address of i
pptr	int **	integer pointer pointer variable	address of ptr
*ptr	int	de-reference of ptr	3

An Illustration

```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	7
j	int	integer variable	10
ptr	int *	integer pointer variable	address of i
pptr	int **	integer pointer pointer variable	address of ptr
**pptr	int	de-reference of de-reference of pptr	7

An Illustration

```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	7
j	int	integer variable	10
ptr	int *	integer pointer variable	address of j
pptr	int **	integer pointer pointer variable	address of ptr
*ptr	int	de-reference of ptr	10

An Illustration

```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	7
j	int	integer variable	9
ptr	int *	integer pointer variable	address of j
pptr	int **	integer pointer pointer variable	address of ptr
**pptr	int	de-reference of de-reference of pptr	9

An Illustration

```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	7
j	int	integer variable	9
ptr	int *	integer pointer variable	address of i
pptr	int **	integer pointer pointer variable	address of ptr
*pptr	int *	de-reference of pptr	value of ptr (address of i)

An Illustration

```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Data Table			
Name	Type	Description	Value
i	int	integer variable	-2
j	int	integer variable	9
ptr	int *	integer pointer variable	address of i
pptr	int **	integer pointer pointer variable	address of ptr
*ptr	int	de-reference of ptr	-2

Pointer Arithmetic

- What's `ptr + 1`?
- ➔ The next memory location!
- What's `ptr - 1`?
- ➔ The previous memory location!
- What's `ptr * 2` and `ptr / 2`?
- ➔ Invalid operations!!!

Pass by Value vs. Pass by Reference

- Modify behaviour in argument passing

```
void f(int j)
{
  j = 5;
}
void g()
{
  int i = 3;
  f(i);
}
```

i = 3

```
void f(int *ptr)
{
  *ptr = 5;
}
void g()
{
  int i = 3;
  f(&i);
}
```

i = 5

Pointers Example #1

Question 1.

Consider the following C language program.

```
#include <stdio.h>
int main() {
	int a = 7, b = 3;
	int *ptr1;
	int **ptr2;

	ptr1 = &b;
	printf("Output 1: %d\n", *ptr1);
	printf("Output 2: %d\n", ++b);

	ptr2 = &ptr1;
	printf("Output 3: %d\n", *ptr2);
	printf("Output 4: %d\n", **ptr2);

	*ptr1 = **ptr2 + a–;
	printf("Output 5: %d", b);

	return 0;
}
```

Output 1:

Output 2:

Output 3:

Output 4:

Output 5:

Pointers Example #2

Question 2.

Consider the following C language program.

```
int main() {
      int a = 9, b = 2;
      int *ptr1;
      int **ptr2;

      ptr1 = &a;
      printf("Output 1: %d\n", *ptr1);

      a += 2;

      printf("Output 2: %d\n", --a);

      ptr2 = &ptr1;
      printf("Output 3: %d\n", *ptr2);
      printf("Output 4: %d\n", **ptr2);

      *ptr1 = **ptr2 + b++;
      printf("Output 5: %d", a);

      return 0;
}
```

Output 1:

Output 2:

Output 3:

Output 4:

Output 5:

Pointers Example #3

Question 3.

Consider the following C language program.

```
#include <stdio.h>
int main() {
        int a = 5, b = 8;
        int *ptr1;
        int **ptr2;
        int ***ptr3;

        ptr1 = &b;
        ptr2 = &ptr1;
        printf("Output 1: %d\n", *ptr2);
        printf("Output 2: %d\n", **ptr2);

        *ptr1 = **ptr2 + ++a;
        printf("Output 3: %d\n", b);

         ptr3 = &ptr2;

        ***ptr3 = **ptr2 + *ptr1;

        printf("Output 4: %d\n", ***ptr3);

        return 0;
}
```

Output 1:

Output 2:

Output 3:

Output 4: