Class 10 Pointer-2

Example

- Write a function to calculate the size of a string. The prototype of the function is given as follows.

```
//str – the pointer of a string
//returned value – the size of the string
int calStrLen(char *str);
```

C		L	a	n	g	u	a	g	e	\0

The idea to resolve the problem:

Since the null character (‘\0’) indicates the tail of a string, we can calculate the size of a string by counting all non-null characters of a string from the beginning.

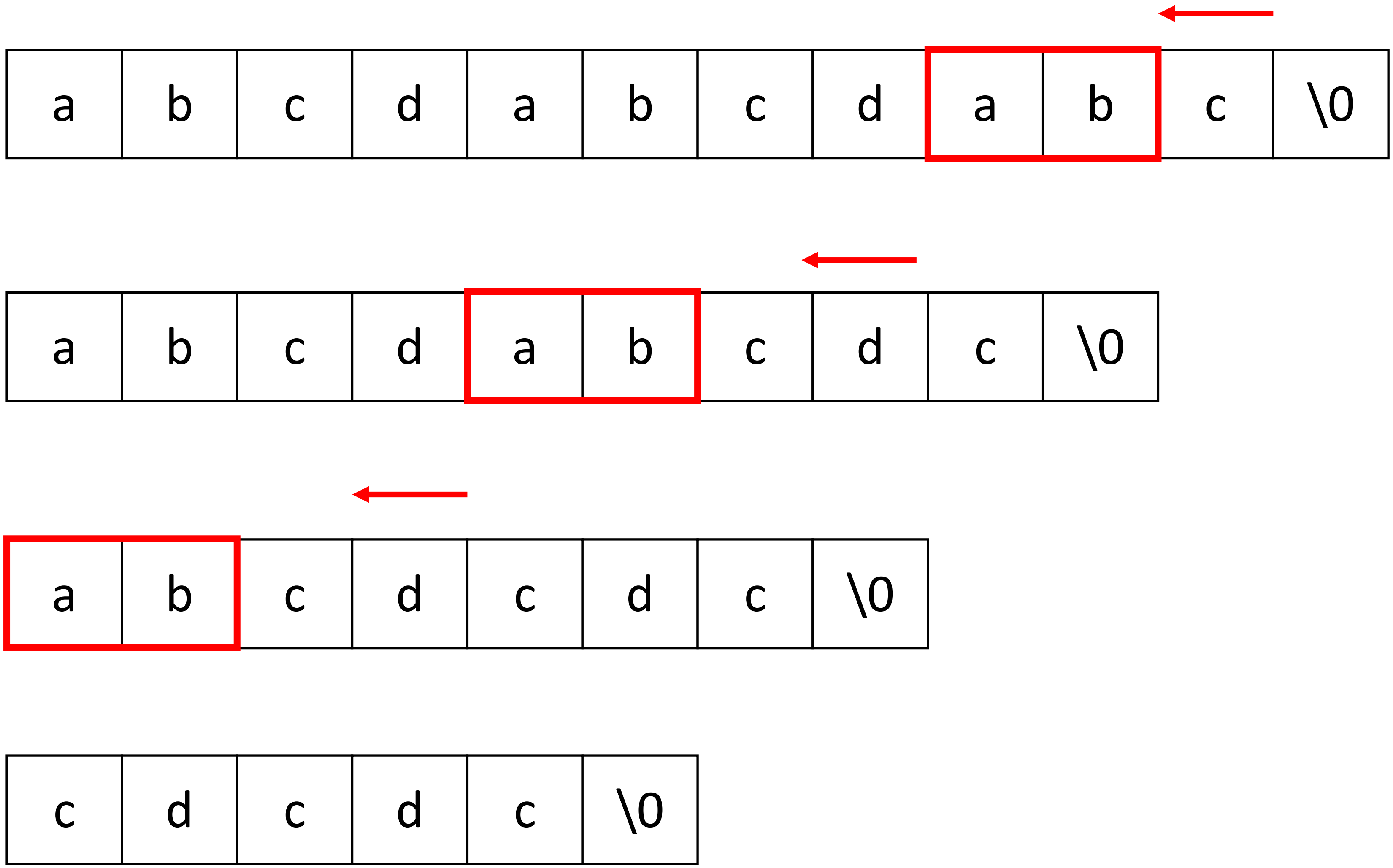
```c
#include <stdio.h>
#include <string.h>

int calStrLen(char *str)
{
    int i = 0;
    char *p = str;
    if( str!= 0 )
    {
       for ( ; ; i++, p++)
       {
           if ( *p == '\0' ) break;
       }
    }
    return i;
}

int main()
{
    char p[256] = "Hello World!";
    printf("string size = %d\n", calStrLen(p));
    printf("string size = %d\n", strlen(p));
    return 0;
}
```

Example

- Write a function to analyze a string, and delete all sub-strings that are included in it. The prototype of the function is given as follows.

```
//str1 – a string, the results are also saved here
//str2 – the sub-string that is included in str1
//returned value – how many sub-strings that are deleted
int delStr(char *str1, char *str2);
```

```
str1 :  “abcdabcdabc”
str2 :  “ab”
```



a	b	c	d	a	b	c	d	a	b	c	\0

a	b	c	d	a	b	c	d	c	\0

a	b	c	d	c	d	c	\0

c	d	c	d	c	\0

```
#include <stdio.h>
#include <string.h>

int delStr(char *str1, char *str2);

int main()
{
    char pstr1[256];
    char pstr2[256];
    int found;

    printf(“Input str1 : \n”);
    scanf(“%s”, pstr1);
    printf(“Input str2 : \n”);
    scanf(“%s”, pstr2);

    printf(”str1 : %s\n", pstr1);
    printf(“str2 : %s\n”, pstr2);
    found = delStr(pstr1, pstr2);
    printf("found = %d\n", found);
    printf(”res  : %s\n", pstr1);
    return 0;
}
......
```

```
int delStr(char *str1, char *str2)
{
    int size1 = strlen(str1);
    int size2 = strlen(str2);
    int num = 0, i, j;
    for( j = size1-size2 ; j >=0 ; j-- )
    {
        for ( i = 0 ; i < size2 ; i++ )
           if (str1[j+i] != str2[i]) break;
        if ( i == size2 )
        {
           i = j;
           while( (str1[i] = str1[i+size2]) != '\0' )
               i++;
           num++;
        }

    }
    return num;
}
```

2D Array & Pointer

- 2D array has a sequential memory

```
int iArr[3][4]={ {1,2,3,4},{5,6,7,8},{9,10,11,12}};
```

iArr

0012FF50 1	0012FF54 2	0012FF58 3	0012FF5C 4
0012FF60 5	0012FF64 6	0012FF68 7	0012FF6C 8
0012FF70 9	0012FF74 10	0012FF78 11	0012FF7C 12

2D Array & Pointer

- 2D array can also be seen as a 1D array whose elements form another 1D array

```
int iArr[3][4]={ {1,2,3,4},{5,6,7,8},{9,10,11,12}};
```

iArr

iArr[0]	0012FF50 1	0012FF54 2	0012FF58 3	0012FF5C 4
iArr[1]	0012FF60 5	0012FF64 6	0012FF68 7	0012FF6C 8
iArr[2]	0012FF70 9	0012FF74 10	0012FF78 11	0012FF7C 12

iArr is the name of a 1D array, and its element is iArr[i].
iArr[i] is the name of another 1D array

2D Array & Pointer

iArr

```
int iArr[3][4]={ {1,2,3,4},{5,6,7,8},{9,10,11,12}};
```

iArr[0]	0012FF50 1	0012FF54 2	0012FF58 3	0012FF5C 4
iArr[1]	0012FF60 5	0012FF64 6	0012FF68 7	0012FF6C 8
iArr[2]	0012FF70 9	0012FF74 10	0012FF78 11	0012FF7C 12

- Row Address
  - The address of a 1D array
  - Jump to the next row

```
iArr, iArr+i, &iArr[i]
```

- Column Address
  - The address of data
  - Jump to the next column

```
*iArr, *(iArr+i), iArr[i]
```

Example

```
#include <stdio.h>

int main()
{
    int iArr[3][4]={ {1,2,3,4},{5,6,7,8},{9,10,11,12}};

    printf(“%d\n”, iArr);     // address of the 1st row
    printf(“%d\n”, *iArr);    // address of the 1st row and 1st column
    printf(“%d\n”, **iArr);   // value of the 1st row and 1st column

    printf(“%d\n”, iArr[1]);         // address of the 2nd row and 1st column
    printf(“%d\n”, iArr+1);          // address of the 2nd row
    printf(“%d\n”, *iArr+1);         // address of the 1st row and 2nd column
    printf(“%d\n”, iArr[1]+1);       // address of the 2nd row and 2nd column
    printf(“%d\n”, *(iArr+1));       // address of the 2nd row and 1st column
    printf(“%d\n”, *(iArr[1]+1));// value of the 2nd row and 2nd column

    return 0;
}
```

Pointer to a 1D Array

```
type (* pointer_name)[size];
```

- It is a pointer that points to a 1D array whose length is indicated by *size*
- The parentheses (小括号) can not be omitted
- Data type of each element in the 1D array is given by *type*

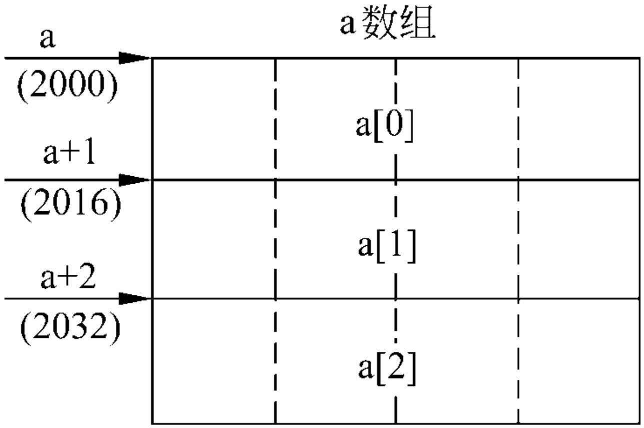
Pointer to a 1D Array

```
int iArr[3][4]={{1,2,3,4},{5,6,7,8},{9,10,11,12}};

int (*p)[4];// the size must be 4

p = iArr + 1;
printf(“%d\n”, p);      // Address of the 2nd row
printf(“%d\n”, *p);     // Address of the 2nd row and 1st column
printf(“%d\n”, **p);    // 5
printf(“%d\n”, *p[0]);  // 5

printf("%d\n", *p[1]);       // 9
printf("%d\n", *(*p+1));     // 6
printf("%d\n", *(p[0]+1));   // 6
printf("%d\n", *(p[1]+1));   // 10
```

Pointer to a 1D Array

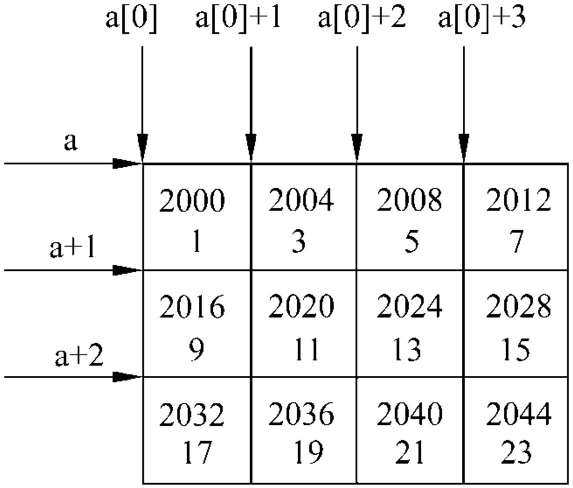
- Pointer that Points to a Row (1D Array)

```
int a[3][4];
int (*pRow)[4] = a;
pRow++;  // pRow = a + 1
```



- Pointer that Points to a Column (data)

```
int a[3][4];
int *pCol = a[1] + 2;
pCol++;  // pCol = a[1] + 3
```



Example

- Calculate the average scores of each course for a group of students
  - The score should be stored as a 2D array
  - The function prototype is given as follows

```
float average(float (*p)[5]);
```

	Jack	Tom	Lucy	James	Poly
Math	80	61	59	85	76
C	75	65	63	87	77
English	92	71	70	90	85

```
float average(float (*p)[5])
{
    int i;
    float sum;
    for ( i = 0 ; i < 5 ; i++ )
        sum += *(*p+i);
    return sum/5;
}

int main()
{
    float score[3][5];
    float courseAve[3];
    int i;

    ......

    for ( i = 0 ; i < 3 ; i++ )
        courseAve[i] = average(score + i);

    ......
}
```

Array of Pointer (指针数组)

```
type *pointer_name[size];
```

- An array each of whose element is a pointer

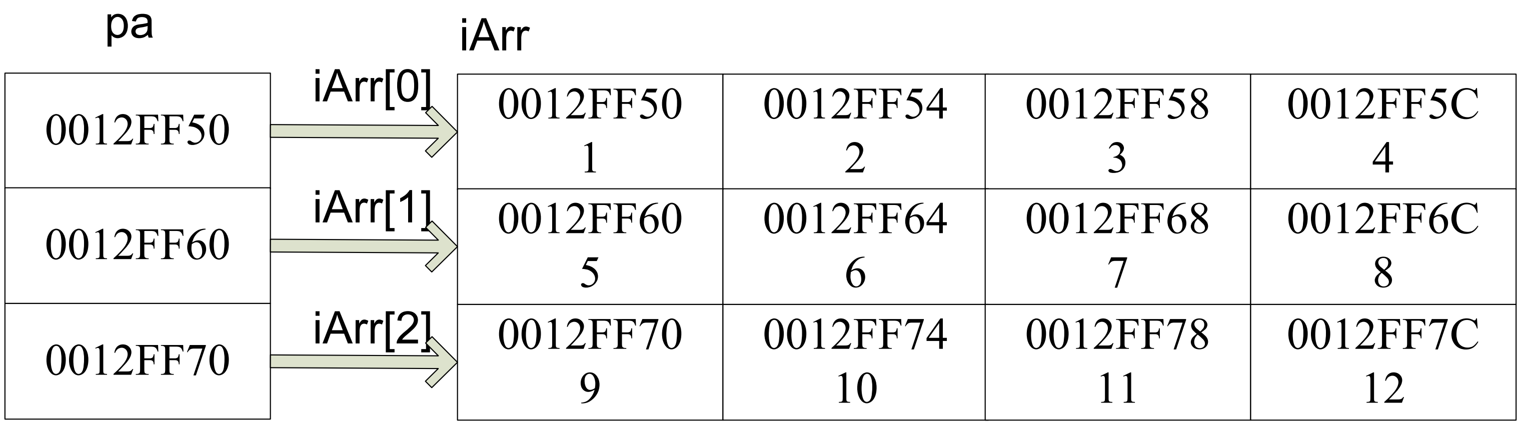
- Array of pointer is used to point to a 2D array. The element of the array is assigned to the address of a row of the 2D array.

- Different from the pointer to 1D array

```
type (* pointer_name)[size];
```

Array of Pointer (指针数组)

```
int iArr[3][4]={{1,2,3,4},{5,6,7,8},{9,10,11,12}};
int *pa[3];
pa[0] = iArr[0];
pa[1] = iArr[1];
pa[2] = iArr[2];

printf(“%d\n”, pa[0][1]);   // 2
printf(“%d\n”, *(pa[0]+1)); // 2
```



pa

0012FF50
0012FF60
0012FF70

iArr

0012FF50 1	0012FF54 2	0012FF58 3	0012FF5C 4
0012FF60 5	0012FF64 6	0012FF68 7	0012FF6C 8
0012FF70 9	0012FF74 10	0012FF78 11	0012FF7C 12

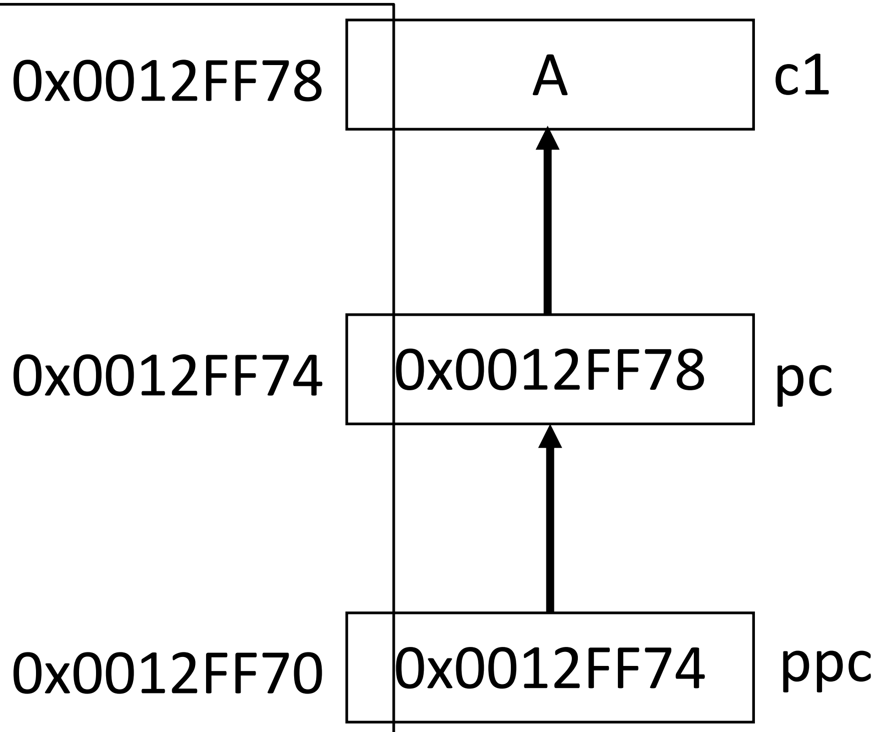
Pointer to a Pointer (指向指针的指针)

```
type **pointer_name;
```

- A pointer that points to another pointer
- Its value is the address of a pointer

Pointer to a Pointer (指向指针的指针)

```
char c1 = ‘A’;
char *pc;
char **ppc;
pc = &c1;
ppc = &pc;

printf(“%c\n”, c1);
printf(“%c\n”, *pc);
printf(“%c\n”, **ppc);

printf(“%d\n”, *ppc); //0x0012FF78
printf(“%d\n”, ppc);  //0x0012FF74
```



0x0012FF78 A c1

0x0012FF74 0x0012FF78 pc

0x0012FF70 0x0012FF74 ppc

Memory in C Program

Code Area	Binary Codes of Programs
String Constant Area	String Constants (available during the execution of the program)
Static Stored Area	Constant, Global Variable, Static Variables (available during the execution of the program)
Stack Area	Local Variables, Formal Parameters, Returned Values (memory is allocated when the function is called, and released when the function is finished)
Heap Area	Memory allocated by using functions, i.e. malloc() The memory should be released by using free()

Functions to Allocate Memory on Heap

- malloc()

```
void * malloc(unsigned int size);
```

  - Allocate the memory of *size* bytes, and the initial address of the allocated memory is returned
  - It returns a (void *) pointer that can be assigned a pointer of any type by using type-castering.
  - A NULL pointer (0) is returned if memory allocation is failed
  - including the header <stdlib.h> before calling this function

```
#include <stdlib.h>
int *p1;
float *p2;

//allocate 4 bytes memory on heap
p1 = (int *)malloc(4);
//allocate 10*sizeof(float)(4) bytes on heap
p2 = (float *)malloc(10*sizeof(float));
```

How about this?

```
int *p;
p = (int *)malloc(2);
```

Error!

Allocated memory size is smaller than the size of int, so sizeof() is usually used with malloc()

Functions to Allocate Memory on Heap

```
void * calloc(unsigned int num, unsigned int size);
```

- calloc()
  - allocate the memory of num*size bytes on heap, and the initial address of the allocated memory is returned
    - num – the number of data
    - size – the size of the data
  - It returns a (void *) pointer that can be assigned a pointer of any type by using type-castering.
  - A NULL pointer (0) is returned if memory allocation is failed
  - including the header <stdlib.h> before calling this function

```
#include <stdlib.h>
int *p1 = (int *)calloc(10, sizeof(int));
int *p2 = (int *)malloc(10*sizeof(10));
```

Functions to Allocate Memory on Heap

```
void * realloc(void *ptr, unsigned int size);
```

- realloc()
  - reallocate the memory of *size* bytes for the pointer *ptr* that has been allocated. If the allocation is successful, the function returns the initial address of the reallocated memory; if not, it returns a NULL pointer.
  - including the header <stdlib.h> before calling this function

```
#include <stdlib.h>
int *p1;

p1 = (int *)malloc(4);
...
p1 = (int *)realloc(p1, 10*sizeof(int));
```

Functions to Release Memory on Heap

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

- free()
  - Release the memory that has been allocated
  - ptr – the initial address of the allocated memory
  - It is the programmers responsibility to release the memory on heap
  - malloc() and free() are used in a pair
  - memory leak occurs if allocated memory is not released

```
#include <stdlib.h>

int *p1;
p1 = (int *)malloc(10*sizeof(int));
...
free(p1);
```

When Should We Use Memory Allocation?

- Memory allocation is required when we can not determine the memory size in advance.

- Example
  - Write a program to calculate the average scores of *N* courses. The number *N* is inputted by uses when the program is executed.
    - It is required to allocate the memory to save the scores of the *N* courses.

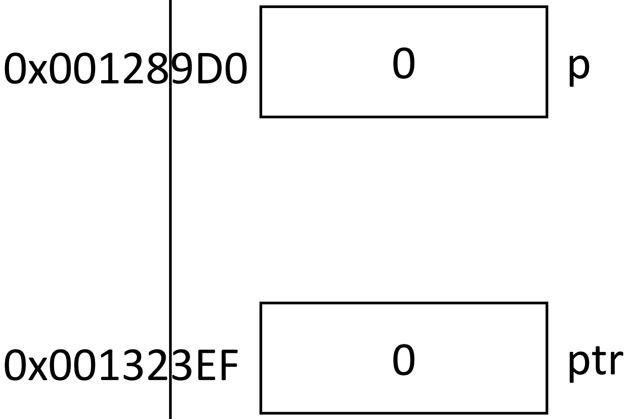
```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int n, i;
    float *pScores = 0, average = 0.0f;

    printf(“Input n: ”);
    scanf(“%d”, &n);
    if ( (pScore = (float *)malloc(n*sizeof(float)) == 0 )
        exit(0);

    for ( i = 0 ; i < n ; i++ )
    {
       scanf(“%f”, pScores+i);
       average += *(pScores+i);
    }

    average /= n;
    printf(“Average = %f\n”, average);
    if (pScore)  free(pScore);
    return 0;
}
```

Notes of Dynamic Memory Allocation

- The total size of the memory that can be allocated by a program depends on the size of system memory
    - Actually it is smaller than system memory
    - DO NOT allocate the memory that is larger than the system memory
- It is the programmers’ responsibility to allocate and release the memory on heap
- DO NOT allocate lots of small memory on heap
    - It is not efficient to allocate lots of small memory
    - Allocate the memory that is large enough at a time

Memory Allocation inside a Function

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

void func(int *ptr, int n)
{
    ptr = (int *)malloc(n*sizeof(int));
}

int main()
{
    int *p = 0;
    int n = 10;
    func(p, 10);

    if ( p )
      printf(“Memory Allocation Successful\n”);
    else
       printf(“Memory Allocation Failed\n”);

    return 0;
}
```



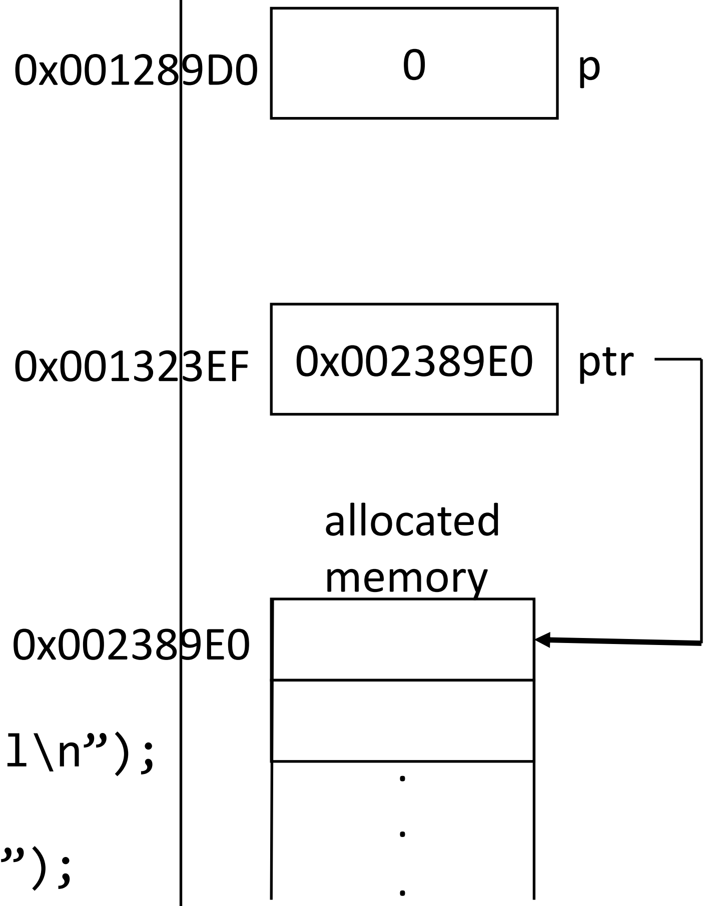
0x001289D0 | 0 | p

0x001323EF | 0 | ptr

This is the output

Memory Allocation inside a Function

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

void func(int *ptr, int n)
{
    ptr = (int *)malloc(n*sizeof(int));
}

int main()
{
    int *p = 0;
    int n = 10;
    func(p, 10);

    if ( p )
      printf(“Memory Allocation Successful\n”);
    else
       printf(“Memory Allocation Failed\n”);

    return 0;
}
```



0x001289D0 | 0 | p

0x001323EF | 0x002389E0 | ptr

allocated memory

0x002389E0

Notes of Dynamic Memory Allocation

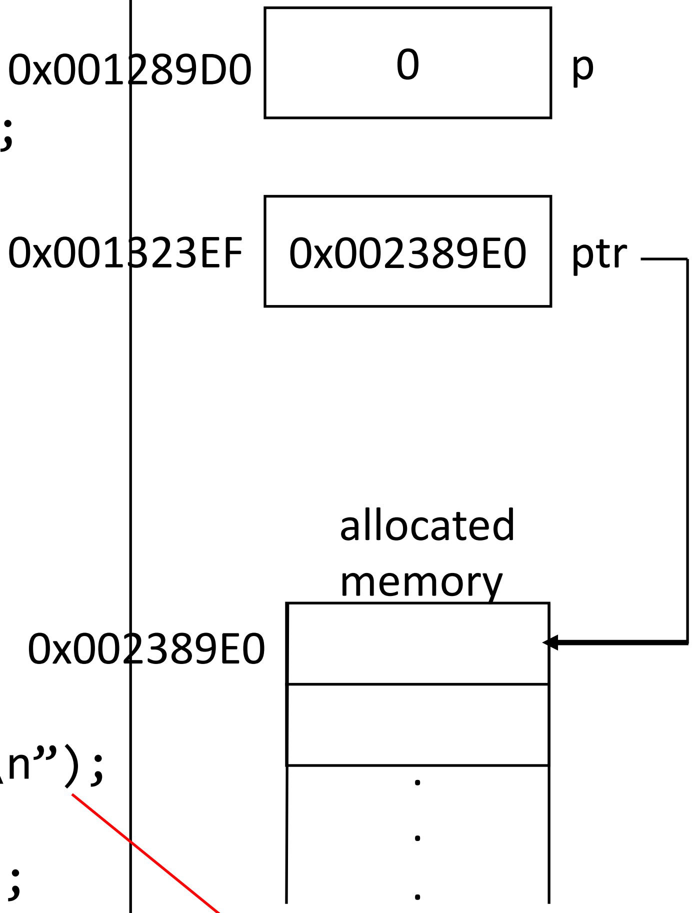
- If a memory is allocated inside a function, the pointer can not be passed to the called function by using the pointer that is the formal parameter of the function
    - The formal parameter is vanished after the execution of the function
    - The allocated memory still exists, but its address (pointer) is not available

Notes of Dynamic Memory Allocation

- Two ways to pass the pointer allocated inside a function
  - By a formal parameter that is the pointer to a pointer
  - By the returned value

Passing Allocated Pointer by Returned Value

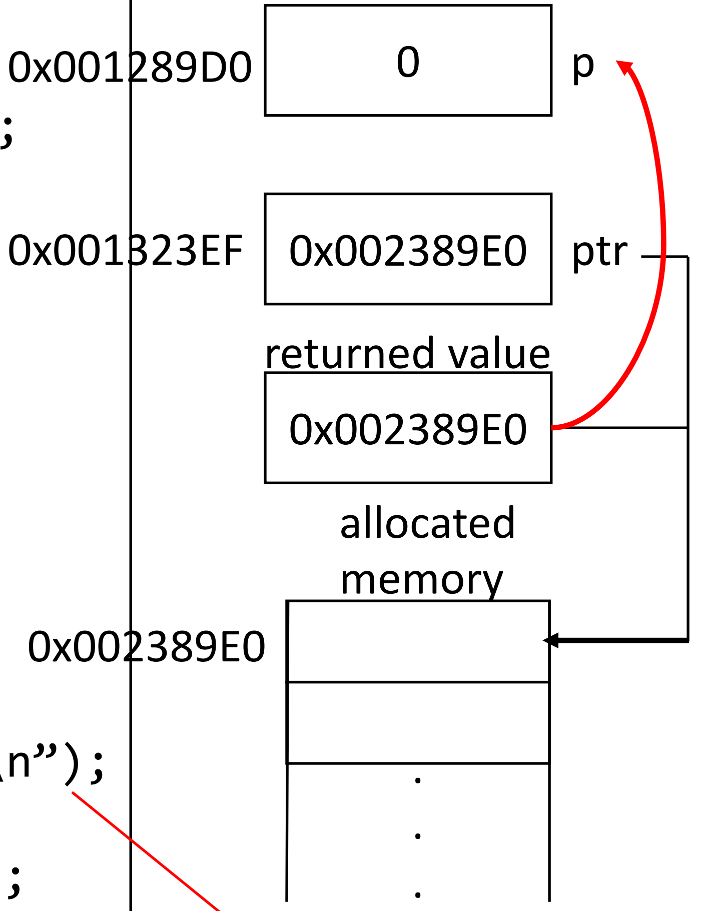
```c
#include <stdio.h>
#include <stdlib.h>

int* func(int n)
{
    int *ptr = (int *)malloc(n*sizeof(int));
    return ptr;
}

int main()
{
    int *p = 0;
    int n = 10;
    p = func(10);

    if ( p )
      printf(“Memory Allocation Successful\n”);
    else
       printf(“Memory Allocation Failed\n”);

    return 0;
}
```



This is the output

Passing Allocated Pointer by Returned Value

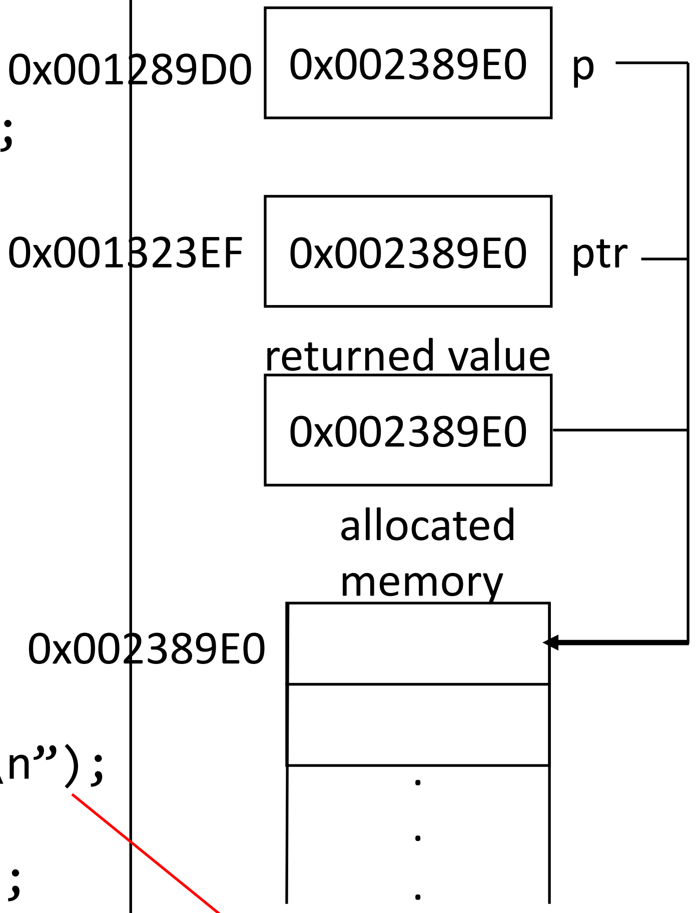
```
#include <stdio.h>
#include <stdlib.h>

int* func(int n)
{
    int *ptr = (int *)malloc(n*sizeof(int));
    return ptr;
}

int main()
{
    int *p = 0;
    int n = 10;
    p = func(10);

    if ( p )
      printf(“Memory Allocation Successful\n”);
    else
       printf(“Memory Allocation Failed\n”);

    return 0;
}
```



This is the output

Passing Allocated Pointer by Returned Value

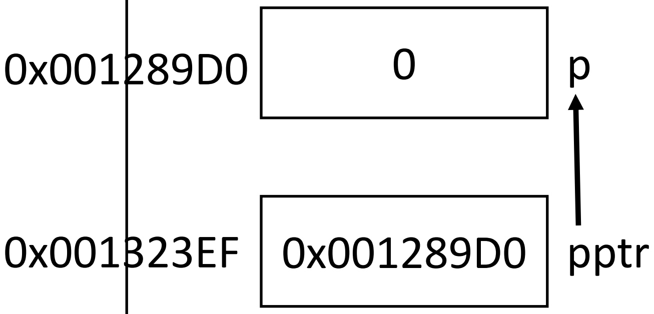
```
#include <stdio.h>
#include <stdlib.h>

int* func(int n)
{
    int *ptr = (int *)malloc(n*sizeof(int));
    return ptr;
}

int main()
{
    int *p = 0;
    int n = 10;
    p = func(10);

    if ( p )
      printf(“Memory Allocation Successful\n”);
    else
       printf(“Memory Allocation Failed\n”);

    return 0;
}
```



This is the output

Passing Allocated Pointer by the Pointer to a Pointer

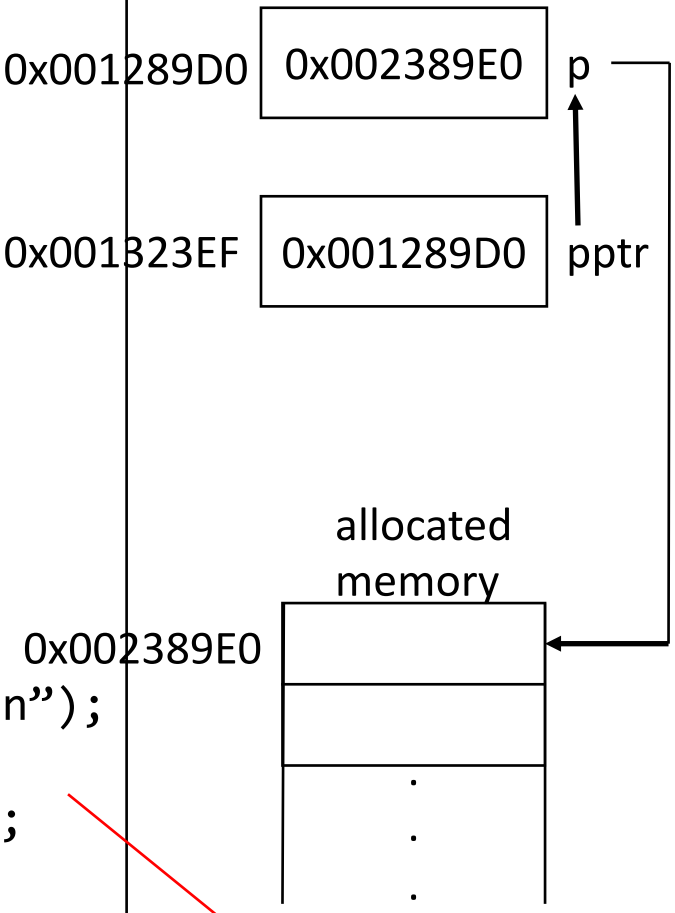
```
#include <stdio.h>
#include <stdlib.h>

void func(int **pptr, int n)
{
    *pptr = (int *)malloc(n*sizeof(int));
}

int main()
{
    int *p = 0;
    int n = 10;
    func(&p, 10);

    if ( p )
      printf(“Memory Allocation Successful\n”);
    else
       printf(“Memory Allocation Failed\n”);

    return 0;
}
```



This is the output

Passing Allocated Pointer by the Pointer to a Pointer

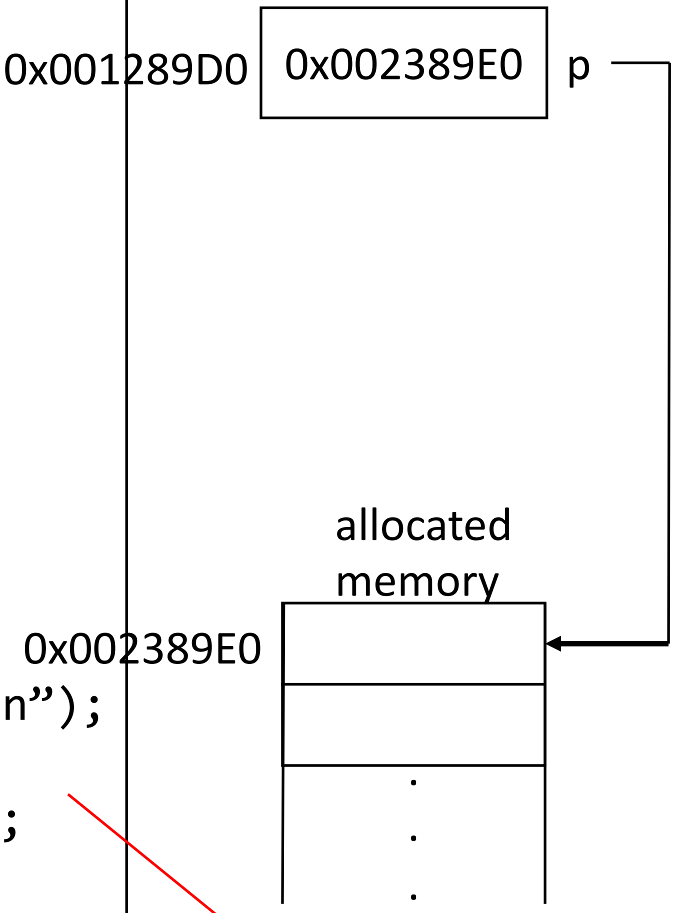
```
#include <stdio.h>
#include <stdlib.h>

void func(int **pptr, int n)
{
  ➡ *pptr = (int *)malloc(n*sizeof(int));
}

int main()
{
    int *p = 0;
    int n = 10;
    func(&p, 10);

    if ( p )
      printf(“Memory Allocation Successful\n”);
    else
       printf(“Memory Allocation Failed\n”);

    return 0;
}
```



This is the output

Passing Allocated Pointer by the Pointer to a Pointer

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

void func(int **pptr, int n)
{
    *pptr = (int *)malloc(n*sizeof(int));
}

int main()
{
    int *p = 0;
    int n = 10;
    func(&p, 10);

    if ( p )
      printf(“Memory Allocation Successful\n”);
    else
       printf(“Memory Allocation Failed\n”);

    return 0;
}
```



This is the output

Exercise

Indicate where the following memory are allocated

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

int i1 = 0;

int main()
{
    int i2;
    char cArr[3] = “abc”;
    char *pc1, *pc2;
    char *pc3 = “123456”;
    static int i3 = 0;
    pc1 = (char *)malloc(10);
    pc2 = (char *)malloc(20);
    free(pc1);
    free(pc2);
}
```

Pointer to a Function (指向函数的指针)

```
type (* pointer_name)(parameter lists);
```

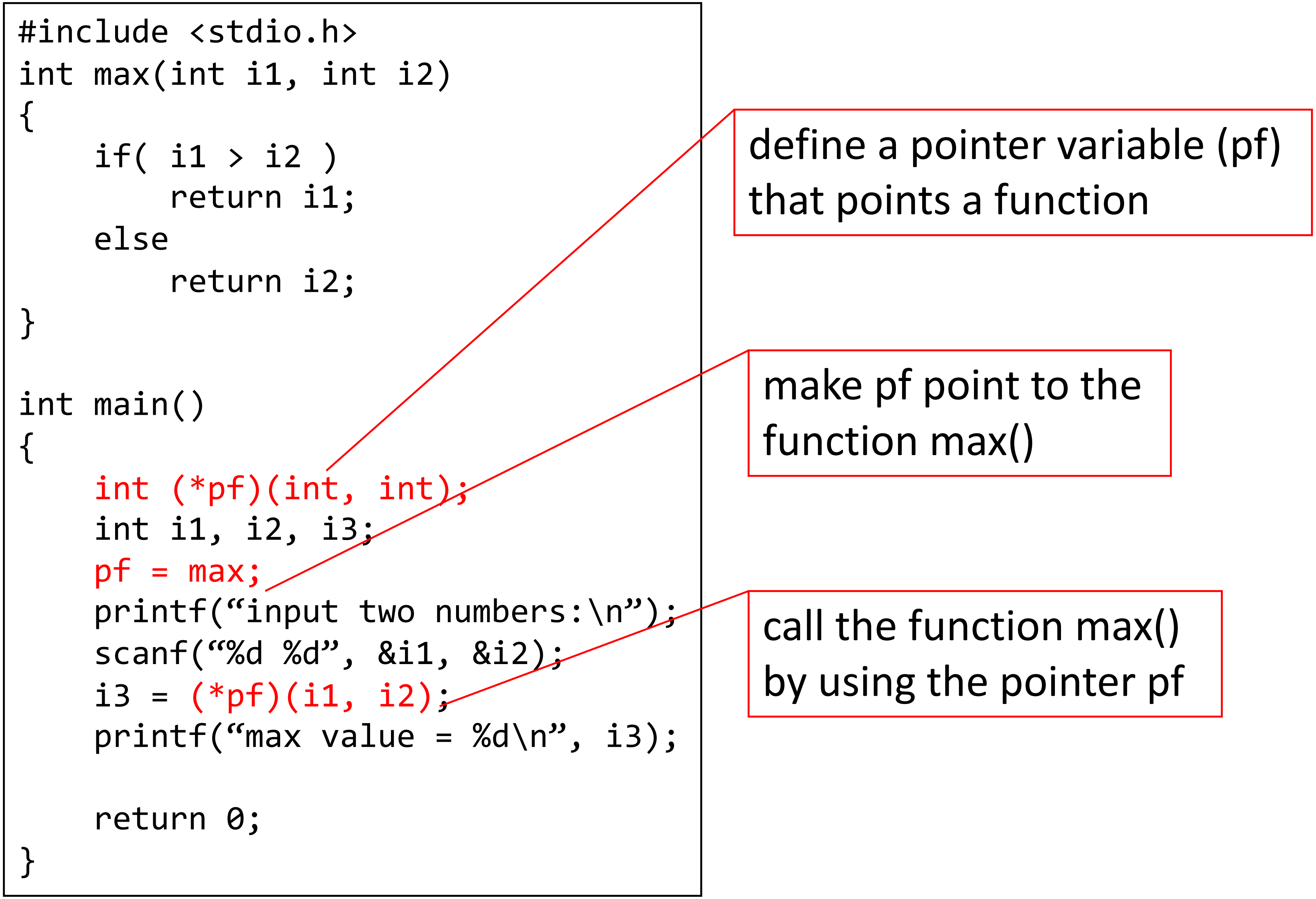
- Each function requires a space on memory, we can make a pointer to take the initial address of this memory.
  - A pointer that points to the memory of a function

```
int (*p1)(int, int);
float (*p2)(int, float);
```

- It is different from a function that returns a pointer

```
type * function_name (parameter list)
{
      Function Body;
}
```

Example



```
#include <stdio.h>
int max(int i1, int i2)
{
    if( i1 > i2 )
        return i1;
    else
        return i2;
}

int main()
{
    int (*pf)(int, int);
    int i1, i2, i3;
    pf = max;
    printf(“input two numbers:\n”);
    scanf(“%d %d”, &i1, &i2);
    i3 = (*pf)(i1, i2);
    printf(“max value = %d\n”, i3);

    return 0;
}
```

define a pointer variable (pf) that points a function

make pf point to the function max()

call the function max() by using the pointer pf

Parameters of Main Function

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

- Main function can have the parameter list

- Although it is not possible to make a function to call the main function, the main function can be called by OS or a user from the console (控制台)

- Parameter list of the Main Function
  - argc – the number of parameters
  - argv*[] – the array each element of which is a char pointer that points to different parameters

Parameters of Main Function

source code of prog.exe

```
#include <stdio.h>

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

    for ( i = 0 ; i < argc ; i++ )
        printf(“%s\n”, argv[i]);

    return 0;
}
```

Run prog.exe in the console

```
c:> prog.exe  hello  world
prog.exe
hello
world

c:> prog.exe 2 1 here
prog.exe
2
1
here
```

argv[0] – the filename of the executable program
argv[1] – the 1st parameter
argv[2] – the 2nd parameter
……

- Calculate the summation of two integers. The two integers should be passed into the program by the parameters of the main function

source code of prog.exe

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i, j, sum;
    if ( argc != 3 )
    {
        printf(“parameters error!”);
        return 1;
    }
    i = atoi(argv[1]);
    j = atoi(argv[2]);
    sum = i + j;
    printf(“sum = %d\n”, sum);
    return 0;
}
```

Run prog.exe in the console

```
c:> prog.exe  2  3
sum = 5

c:> prog.exe 2 3 here
parameters error!

c:>
```

atoi() – change a string to an integer number

atof()– change a string to a floating number