Chapter 11: Pointers

Outline

Pointer variables
More About Pointer Operators
More About Pointer Variable Declarations
The Size of Pointer Variables
Pointer Arithmetic
Calling Functions by Reference Using Pointers
Relationship Between Pointers and Arrays
Using Pointers to Pass One-Dimensional Arrays to a Function
Dynamic Allocation of Memory
Functions Return Pointers
Pointer to Pointers
Arrays of Pointers
Pointers

Outline

Pointer to Functions
Pointer to arrays
Dynamic Memory Allocation of Arrays
Using Pointers to Pass Multi-Dimensional Arrays to a Function
Using Pointer for Hardware Interface
A pointer is defined as a variable which contains the address of another variable or dynamically allocated memory.

With an asterisk ‘*’ in front of the variable names, the variables of pointer type can be declared similar to variables of other data types.

For example,

```c
int i, *p;
```
The Size of Pointer Variables

The variable of pointer type is used to hold an address of the memory. The size of a variable of pointer type is implementation-dependent. It is typically 4 bytes for 32-bit machines and 8 bytes for 64-bit machines. It is independent of the variable type such as char or double that it points to.
Example:

/* File: ptrsize.c */
#include <stdio.h>

int main() {
    char *pc;
    int *pi;
    double *pd;

    printf("sizeof(char *) = %u\n", sizeof(char *));
    printf("sizeof(pc) = %u\n", sizeof(pc));
    printf("sizeof(pi) = %u\n", sizeof(pi));
    printf("sizeof(pd) = %u\n", sizeof(pd));
    return 0;
}

Output when the program is executed in a 32-bit machine:

sizeof(char *) = 4
sizeof(pc) = 4
sizeof(pi) = 4
sizeof(pd) = 4
More About Pointer Operators

• `&` (address operator)
  – To obtain the address of a variable, add the address operator `&` in front of the variable name.
    ```c
    int i = 10;
    int *iPtr;
    iPtr = &i;  // iPtr gets address of i
    iPtr <=> &i
    ```

• `*` (indirection/dereferencing operator)
  – To declare a variable of pointer, add the asterisk `*` in front of the identifier.
  – To obtain the value of a variable, add the indirection operator `*` in front of a pointer’s variable name.
    ```c
    *iPtr <=> i
    *iptr gives the value of i (iptr points to i)
    – * can be used for assignment
    *iptr = 5; // changes i to 5
    ```
/* File: poperators.c */
#include <stdio.h>

int main() {
    int i, j, *p; /* declare i and j as int, p as pointer to int */
    i = 10;
    j = 0;
    p = &i;    /* p is assigned with the address of i.
               p points to i, *p <=> i */
    printf("The value of i is %d\n", i);
    printf("The value of j is %d\n", j);
    printf("The address of i is %p\n", &i);
    printf("The value of p is %p\n", p);
    printf("The value of *p is %d\n", *p);

    *p = 20;    /* The memory pointed by p is assigned with 20 */
    j = *p;     /* j is assigned with the value pointed by p */
    printf("After indirection operations\n");
    printf("The value of i is %d\n", i);
    printf("The value of j is %d\n", j);
    printf("The address of i is %p\n", &i);
    printf("The value of p is %p\n", p);
    printf("The value of *p is %d\n", *p);
    return 0;
}
A sample memory layout for two integers and one pointer

For the statement

\[ p = &i, \]

if the address of i is 0x00E81E20, the value of p is 0x00E81E20 as shown below.

For before and after indirection operations

\[ *p = 20; \]
\[ j = *p; \]
Output:

The value of i is 10
The value of j is 0
The address of i is 00E81E20
The value of p is 00E81E20
The value of *p is 10
After indirection operations
The value of i is 20
The value of j is 20
The address of i is 00E81E20
The value of p is 00E81E20
The value of *p is 20
More About Pointer Variable Declarations

– Using following format to declare a pointer variable.

\[ \text{type} \ *\text{pointer\_name}; \]

For example, following statement declares a pointer to an \textit{int}

\[ \text{int} \ *\text{iPtr}; \]

– Can declare pointers to any data type such as int, float, double, and complex, etc.

– Without initialization, the value of declared pointers is undefined in Ch.

It is \texttt{0} or \texttt{NULL} in Ch and C++.

• \texttt{0} or \texttt{NULL} pointer points to nothing (\texttt{NULL} preferred)

– Pointer to \texttt{void}, (\texttt{void} *)

• Generic pointer, represents any type. \texttt{void} pointers cannot be dereferenced

\[ \text{void} \ *\text{p}; \quad /* \ *\text{p} \text{ is invalid } */ \]
The asterisk in declaration is not an indirection operator even for declaration with initialization. For example,

```c
> int i
> int *p = &i // initialization
> i = 10
 10
> *p
 10
```
## Pointer Arithmetic

Following arithmetic operations can be performed on pointers:

- **Add an integer to a pointer** (\( p+i \) or \( p+=i \))

  If \( p \) is a pointer to some element of an array, then \( p+i \) (or \( p+=i \)) increments it to point to \( i \)th element beyond where it currently points to. For example, if \( p \) points to the 4th element of an integral array \( a \) with 6 elements which has address of 0x008C9550,

  ```
  int *p, a[6];
  p = &a[3];
  
  p+2 (or \( p+=2 \)) points to
  0x008C955C+2*sizeof(int)=0x008C9564
  ```

- **Subtract an integer from a pointer** (\( p-i \) or \( p-=i \))

  \( p-2 \) (or \( p-=2 \)) points to
  
  0x008C955C-2*sizeof(int)=0x008C9554
Pointer Arithmetic (Cont.)

- Increment/decrement pointer (++ or --)

If p is a pointer to some element of an array, then ++p (p+1) increments p to point to the next element. --p (p-1) decrements to the previous element. For example, if p points to the 4th element of an integral array a which has address of 0x008C9550,

```
p = &a[3];
```

++p points to 0x008C955C+sizeof(int)=0x008C9560.

--p points to 0x008C955C-sizeof(int)=0x008C9558.
Pointer Arithmetic (Cont.)

```c
> int a[6] = {100, 200, 300, 400, 500, 600}, *p
> p = &a[0]  // p = 0x008C9550, p points to a[0]
> *p
100
> p += 3       // p = 0x008C955C, p points to a[3]
> *p
400
> *(p+1)
500
> *(p-1)
300
> *p+1
401
> *(p+2) = 12   // set a[5] = 12
> *(p-2) = 13   // set a[1] = 13
```

The contents of array `a` after assignment statements.

```
0x008C9550  a[0] = 100
p−2  → 0x008C9554  a[1] = 13
0x008C9558  a[2] = 300
p  → 0x008C955C  a[3] = 11
0x008C9560  a[4] = 500
p+2  → 0x008C9564  a[5] = 12
```
Pointer Arithmetic (Cont.)

- Subtract pointers from each other (p2-p1)
- p1 and p2 shall be the same data type. p2-p1 gives the number of elements from p1 to p2.
- `ptrdiff_t` is a signed integer type defined in the header file `stddef.h`

```c
#include <stddef.h>
int a[6], *p1, *p2;

ptrdiff_t n;
p1 = &a[1]
p2 = &a[4]
n = p2 - p1; // n is 3
```
• The increment operator ‘++’ and decrement operator ‘--’ allows for performing two actions at once.
• The expression *p++ accesses what p points to, while simultaneously incrementing p so that it points to the next element.
• The pre-increment form *++p increments p, then accesses what it points to.

Example:

```c
char dest[100], src[100] = “original string”;
char *dp = &dest[0], *sp = &src[0];

/* copy src to dest inside function strcpy() */
while(*sp != ‘\0’) {
    *dp++ = *sp++;
}
*dp = ‘\0’;
```

• *dp++ = *sp++ ; is equivalent to *dp=*sp; dp++; sp++;
• When pointer arithmetic is performed, it must be within the valid range. For example, if an array has 10 elements, elements a[10] and a[-1] cannot be accessed because by default, the valid subscript for a 10-element array ranges from 0 to 9.
Pointer Arithmetic (Cont.)

The code fragments below are the same.

```c
char dest[100], src[100] = "original string";
char *dp = &dest[0], *sp = &src[0];

/* copy src to dest inside function strcpy() */
while(*sp != '\0') {
    *dp++ = *sp++;
}
*dp = '\0';
```

```c
char dest[100], src[100] = "original string";
char *dp = &dest[0], *sp = &src[0];
/* copy src to dest inside function strcpy() */
while(*dp++ = *sp++) ;
```
Calling Functions by Reference
Using Pointers

- Pointers can be used to pass the addresses of variables to functions and access variables through their addresses indirectly.
- Through this method, it is possible to write functions to alter arguments passed into them.
Program 1:

Write a program to swap the values of a and b.

Output:

```c
/* File: noswap.c */
#include <stdio.h>

void swap(int x, int y) {
    int temp;
    temp = x;
    x = y;
    y = temp;
}

int main() {
    int a = 5, b = 6;

    swap(a, b);
    printf("a = %d, b = %d\n", a, b);  // a = 5, b = 6
    return 0;
}
```

Note: Program 1 would not be able to swap the values of a and b, but only the values of x and y.
Before swap operations

\[
\begin{array}{c|c|c}
\text{a} & 5 & \text{OXE0000000} \\
\hline
\text{b} & 6 & \text{OXE0000004} \\
\hline
\end{array}
\quad
\begin{array}{c|c|c}
\text{c} & 5 & \text{OXFFFF0000} \\
\hline
\text{y} & 6 & \text{OXFFFF0004} \\
\hline
\text{temp} & \text{unknown} & \text{OXFFFF0008} \\
\hline
\end{array}
\]

After swap operations

\[
\begin{array}{c|c|c}
\text{a} & 5 & \text{OXE0000000} \\
\hline
\text{b} & 6 & \text{OXE0000004} \\
\hline
\end{array}
\quad
\begin{array}{c|c|c}
\text{c} & 6 & \text{OXFFFF0000} \\
\hline
\text{y} & 5 & \text{OXFFFF0004} \\
\hline
\text{temp} & 5 & \text{OXFFFF0008} \\
\hline
\end{array}
\]
Program 2:

Output:

```
/* File: swap.c */
#include <stdio.h>

void swap(int *pa, int *pb) {
    int temp;
    temp = *pa;
    *pa = *pb;
    *pb = temp;
}

int main() {
    int a = 5, b = 6;
    swap(&a, &b);
    printf("a = %d, b = %d\n", a, b); // a = 6, b = 5
    return 0;
}
```

Note: Program 2 would be able to swap the values of a and b, and yield a=6 and b=5.
Before swap operations

<table>
<thead>
<tr>
<th>a</th>
<th>5</th>
<th>0x80000000</th>
<th>pa</th>
<th>0x00000000</th>
<th>0xFFFF0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>6</td>
<td>0x80000004</td>
<td>pb</td>
<td>0x80000004</td>
<td>0xFFFF0004</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>temp</td>
<td>unknown</td>
<td>0xFFFF0008</td>
</tr>
</tbody>
</table>

After swap operations

<table>
<thead>
<tr>
<th>a</th>
<th>6</th>
<th>0x80000000</th>
<th>pa</th>
<th>0x00000000</th>
<th>0xFFFF0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>5</td>
<td>0x80000004</td>
<td>pb</td>
<td>0x80000004</td>
<td>0xFFFF0004</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>temp</td>
<td>5</td>
<td>0xFFFF0008</td>
</tr>
</tbody>
</table>

*pa <==> a
*pb <==> b
Example:

Function `scanf()` uses pointers to pass results through its arguments.

```c
/* File: scanfc.c */
#include <stdio.h>

int main() {
    int num;

    printf("Please enter an integer number:\n");
    scanf("%d", &num);  // pass a pointer to scanf()
    printf("Your input number is %d\n", num);
    return 0;
}
```
Example:
Write a function disc() to calculate both the area and circumference of a metal disc. The radius of the disc is passed as an argument. Its area and circumference shall also be passed back to the calling function through its other two arguments.

```c
/* File: disc.c
   Use a function to calculate the area and circumference of a disc. Pass the results by reference using pointers.
*/
#include <stdio.h>
#include <math.h>   /* for M_PI */
/* function disc() calculates area and circumference of a disc with a given radius. */
void disc(double radius, double *area, double *circum) {
    *area = M_PI * radius *radius;
    *circum = 2*M_PI*radius;
}
```
```c
int main() {
    double radius;        /* radius of the disc */
    double area, circum;  /* area and circumference of the disc */

    /* printf("Enter the radius for the disc (m): ");
     scanf("%lf", &radius); */
    radius = 3.5;         /* radius is 3.5 meters */

    disc(radius, &area, &circum);
    printf("area = %.16f (m^2)\n", area);
    printf("circumference = %.16f (m)\n", circum);
    return 0;
}
```

**Output**

```
> disc.c
area = 38.484510 (m^2)
circumference = 21.991149 (m)
```
Using const Qualifier with Pointers

- The type qualifier const can be used to avoid the modification of the object pointed by a pointer.

```c
/* File: strcopy.c */
#include <stdio.h>
/* a sample implementation of strcpy() */
char *strcopy(char *dp, const char *sp) {
    char *retp = dp;

    while(*sp != '\0')
        *dp++ = *sp++;
    *dp = '\0';
    return retp;
}
int main() {
    const char *src = "original string";
    char dest[100];

    strcopy(dest, src);
    printf("dest = %s\n", dest);
    return 0;
}
```

*dp++ = *sp++; is equivalent to
*dp = *sp;
dp++; 
sp++;

**Output:**

dest = original string
Relationship Between Pointers and Arrays

- Arrays and pointers are closely related
  - Array name acts as a constant pointer. Its value is the address of the first element of the array. So \( p = \texttt{a}; \) is equivalent to \( p = \&\texttt{a}[0]; \).
  - Pointers can do array subscripting operations.

- Assume an integral array \texttt{a} has 6 elements. The 3rd element of array \texttt{a} can be accessed by
  \[
  \text{int a[6] = \{1,2,3,4,5,6\};}
  \]
  \[
  (1) \texttt{a[2]}
  \]
  \[
  (2) \texttt{*(a+2)}
  \]
  \[
  \text{int } \ast \texttt{p};
  \]
  \[
  \texttt{p = a}; \quad \text{or} \quad \texttt{p = \&a[0];}
  \]
  \[
  (3) \texttt{*(p+2)}
  \]
  \[
  (4) \texttt{p[2]}
  \]
Example:

The sample code below illustrates four methods to access elements of an array:

```
> int i
> int a[10], *p
> a[3] = 3 // method 1
> for(i=0; i<10; i++) printf("%d", a[i])
0 0 0 3 0 0 0 0 0 0
> *(a+4) = 4 // method 2
> for(i=0; i<10; i++) printf("%d", a[i])
0 0 0 3 4 0 0 0 0 0
> p = a
> *(p+5) = 5 // method 3
> p[6] = 6 // method 4
> for(i=0; i<10; i++) printf("%d", a[i])
0 0 0 3 4 5 6 0 0 0
```
• Relation of multi-dimensional array and pointer

**Example:**

```c
int a2[3][4] = {1, 2, 3, 4
                  5, 6, 7, 8,
                  9, 10, 11, 12};
int *p;
p = &a2[1][0];
p[0] = 15;       // a2[1][0] = 15
*(p + 2) = 17;   // a2[1][2] = 17
```
Demos

> int a[5] = {1, 2, 3, 4, 5}, *p
> p = a + 2

What is the output from each expression below?

p[2], p[-2], *p, p[0], *a, *(a+2), a[5],
p[10], p[-10], a[10]
Using Pointers to Pass One-Dimensional Arrays to a Function

• When an array is passed to a function, what is actually passed is the address of the first element of the array. In the called function, this argument is a local variable of pointer type. That is why pointers can be used to pass one-dimensional arrays to functions.
C for Engineers and Scientists: An Interpretive Approach

/* File: pass1Dcarray.c */
#include <stdio.h>
#include <stdlib.h>
double func(int n, double *a);

int main() {
    double a[5] = {1, 2, 3, 4, 5};
    double b[6] = {1, 2, 3, 4, 5, 6};
    double c[2][3] = {1, 2, 3, 4, 5, 6};
    double sum;

    sum = func(5, a);
    printf("sum = %f\n", sum);

    sum = func(6, b);
    printf("sum = %f\n", sum);

    sum = func(2*3, &c[0][0]);
    printf("sum = %f\n", sum);

    return 0;
}

double func(int n, double *a) {
    double sum = 0;
    int i;

    for(i=0; i<n; i++) {
        sum += *(a+i);
    }

    return sum;
}

/* File: pass1Dcarrayvla.c */
#include <stdio.h>
#include <stdlib.h>
double func(int n, double aa[n]);

int main() {
    double a[5] = {1, 2, 3, 4, 5};
    double b[6] = {1, 2, 3, 4, 5, 6};
    double c[2][3] = {1, 2, 3, 4, 5, 6};
    double sum;

    sum = func(5, a);
    printf("sum = %f\n", sum);

    sum = func(6, b);
    printf("sum = %f\n", sum);

    sum = func(2*3, &c[0][0]);
    printf("sum = %f\n", sum);

    return 0;
}

double func(int n, double aa[n]) {
    double sum = 0;
    int i;

    for(i=0; i<n; i++) {
        sum += aa[i];
    }

    return sum;
}
• The function definition
  
  double func(int n, double *a)

  can be replaced by

  double func(int n, double a[])

• The statement

  sum += *(a+i);

  can be replaced by

  sum += a[i];
Dynamic Allocation of Memory

• A problem with using a fixed-size array is that either the array size is too small to handle special cases, or it is too big and the resource is wasted.
• Without using variable length arrays in C99, this problem can be solved by dynamically allocating memory using the standard functions `malloc()`, `calloc()`, and `realloc()`. The function prototype for `malloc()`

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

• Each allocation of memory will yield a pointer to an object disjoint from any other object.
• The pointer that is returned points to the start (lowest byte address) of the allocated space.

Example:

```c
char *str;
...
int num = 100;
str = (char *)malloc(num*sizeof(char));
```
• The return type of function `malloc()` is a pointer to void. It is typically cast to the pointer type of the lvalue.
The `sizeof` operator can be used to compute the size, in bytes, of a variable or type. It is useful to allocate memory for variables whose sizes are unknown. For example,

```c
int num = 100;

double *dp = (double *)malloc(num*sizeof(double));
```

Whenever the function `malloc()` is unable to allocate the requested memory, it returns a `NULL` pointer.

It is important to check the returned value of `malloc()` when it is called. For example, one can perform a check similar to the code below:

```c
int *p = (int *)malloc(num*sizeof(int));
if(p == NULL)
{
    printf("out of memory\n");
    return -1;
}
```

NOTE: If function `malloc()` returns `NULL`, the code should return to its caller, or exit from the program entirely after printing the error message.
• It is usually a good idea to deallocate memory once it is no longer being used.

• Dynamically allocated memory is deallocated with the function `free()`. For example, if \( p \) contains a pointer previously returned by function `malloc()`, the statement `free(p)` can be used to release the memory dynamically allocated.
• Sample Problem

The system in Figure 1 (a) consists of a single body with mass \( m \) moving on a horizontal surface. An external force \( p \) acts on the body. The coefficient of kinetic friction between body and horizontal surface is \( \mu \). The freebody diagram for the system is shown in Figure 1 (b).

![System Diagram](image1)

(a) System Diagram

![Free Body Diagram](image2)

(b) Free Body Diagram

Figure 1: The system diagram and FBD of a sample problem
The nomenclature related to the modeling of the system is listed below.

\[ \begin{align*}
  m & \quad \text{mass of the body} \\
  x & \quad \text{position of the body} \\
  v & \quad \text{velocity of the body} \\
  a & \quad \text{acceleration of the body} \\
  g & \quad \text{gravitational acceleration} \\
  \mu & \quad \text{friction coefficient} \\
  f & \quad \text{friction force} \\
  N & \quad \text{normal force}
\end{align*} \]

**Equation of motion:**

The equation of the motion of the system can be derived based on the Newton's second law.

\[ \begin{align*}
  N &= mg \quad (1) \\
  f &= \mu N \quad (2) \\
  p-f &= ma \quad (3)
\end{align*} \]

From equation (1), (2) and (3), the formula for calculating the acceleration of the rigid body can be derived as follows.

\[ a = \frac{(p - \mu mg)}{m} \quad (4) \]
• Problem Statement

For the system shown in Figure 1(a), given $m = 5 \text{ kg}$, $g = 9.81 \text{ m/s}^2$, $\mu = 0.2$.

The external force $p$ is expressed as a function of time $t$,

$$p(t) = 4\sin(t - 3) + 20 \quad \text{when } t \geq 0$$

Write a program to calculate and print the accelerations $a$ during $t = 0$ to $t = 10$ with variable number of points (Using standard function `malloc()` to allocate memory for $a$ and $t$ dynamically).
C for Engineers and Scientists: An Interpretive Approach

/* File: acceldynmemory.c */
#include <stdio.h>
#include <chplot.h>
#include <stdlib.h>
#define M_G 9.81

int accel(int n) {
    double *a, *t, t0, tf;
    double mu, m, p;
    int i;

    mu = 0.2;
    m = 5;
    t0 = 0;
    tf = 10;
    a = (double *) malloc(sizeof(double)*n);
    if(a == NULL) {
        printf("Error: not enough memory\n");
        return -1;
    }
    t = (double *) malloc(sizeof(double)*n);
    if(t == NULL) {
        printf("Error: not enough memory\n");
        return -1;
    }
    printf("  time   accel (m/s^2)\n");
    printf("----------------------\n");
    for(i=0; i<n; i++) {
        t[i] = t0+i*(tf-t0)/(n-1);
        p = 4*sin(t[i]-3)+20;
        a[i] = (p-mu*m*M_G)/m;
        printf("a[%d] = %f\n", i, a[i]);
    }
    free(a);
    free(t);
    return 0;
}

int main() {
    int retval;

    retval = accel(100);  // with 100 points
    if (retval == -1)
        printf("Error: function call failed.\n");
    retval = accel(11); // with 11 points
    if (retval == -1)
        printf("Error: function call failed.\n");

    return 0;
}

Note:
If function accel() fails, it returns -1.
Function `calloc()`

```c
void *calloc(size_t num, size_t elsize);
```

The `calloc()` function allocates an amount of memory equal to `num*elsize`. When using `calloc()` function to allocate memory for an array, the number of elements of the array is specified by the first argument. The second argument specifies the number of bytes to be allocated for each element. The space allocated is initialized to zeros.

```c
int num = 100;
int *p = calloc(num, sizeof(int));
```
Function `realloc()`

```c
void *realloc(void *ptr, size_t size);
```

The `realloc()` function changes the size of the previously allocated memory pointed to by `ptr` to that specified by `size`. The value of `size` may be greater or less than the original.

- If `ptr` is Null, then the call is equivalent to `malloc(size)`. If `size` is equal to zero, then the call is equivalent to `free(ptr)`.
- The `realloc()` function returns a pointer to the newly allocated memory, which may be different from `ptr`, or Null if the request fails.
- If the function call of `realloc()` fails, the original block is left unchanged.
- If pointer `ptr` was allocated memory for an integer array of 10 elements, the following statement will reallocate memory for the pointer to reduce the size of the array to 5 elements. Memory for the other five elements are returned to the system’s memory for reuse.

```c
ptr = realloc(ptr, 5*sizeof(int));
```
Functions Return Pointers

- When a function passes or returns a pointer, the pointer shall point to a global variable or dynamically allocated memory.

- A variable of automatic duration has a local scope. The variable of automatic duration and its associated memory will not be available when the program control is out of the scope. Using a pointer to point to the address of the variable of automatic duration is a programming bug.

- Function can also return a pointer related to an address passed from its argument.

- If the function is called successful, it returns a valid pointer. Otherwise, it returns NULL.

The example below illustrates three cases. The function fun1() and fun2() are correct. They return pointers pointed to a global variable and dynamically allocated memory which should be freed later. The function fun3() is wrong because it returns a pointer pointed to a local variable.

Function fun2() returns a valid pointer on success. Otherwise, it returns NULL.
Example:

```c
/* File: returnptr_err.c */
#include <stdio.h>

int g = 10;
int *fun1(void) {
    int *p;
    /* ... */
    p = &g;
    return p;  // return a pointer to a global variable
}

int *fun2(int n) {
    int *p;
    p = (int *)malloc(sizeof(int)*n);
    if(p == NULL) {
        printf("Error: no memory\n");
    }
    /* ... */
    return p;  // return a pointer to a dynamically allocated memory
}

int *fun3(void) {
    int i, *p;
    p = &i;
    return p;  // Error: return a pointer to the address of
                // variable of automatical duration.
}

int main() {
    int *p;
    p = fun1();   // ok
    *p = 10;      // <==> g = 10;
    p = fun2(100);   // ok
    *p = 10;      // ok
    free(p);
    p = fun3();   // bad
    *p = 10;      // Error: program may crash
    return 0;
}
```
Pointers to Pointers

- A pointer to a pointer is defined as a variable which contains the address of another pointer.
- Consider the following code:

```c
int i; // i is an integer
i = 10;
int *p = &i; // p is a pointer pointes to i
int **pp = &p; // pp pointes to p
```

Here, `**p` refers to memory address of pointer `p` which refers to the memory address of variable `i`. It can be illustrated by the following figure:
Example:

```c
> int i, *p
> p = &i       // assign address of i to p
> 00E81E20
> *p = 10
> 10
> printf("i = %d\n", i);
> i = 10
> int **pp
> pp = &p       // assign address of p to pp
> 00E82040
> printf("**pp = %d\n", **pp)
> **pp = 10
```

\[**pp == *(*pp) == *p == i\]
If the address of \( i \) is 0x00E81E20, the value of \( p \) is 0x00E81E20 as shown below. Assume \( pp \) has address of 0x00E83060.
An array of strings can be handled by a pointer to pointer below.

```c
> char *s1 = "ab";
> char *s2 = "py";
> char **pp;
> pp = (char **)malloc(3*sizeof(char *));
4006c8d0
> pp[0] = s1;
ab
> pp[1] = s2;
py
00000000
> pp[1][1]
y
```

![Diagram](image)
Use a pointer to pointer for dynamic allocation of memory.

- The program on the following slide is an example of how this can be done.
- The first call to function `func()` assigns ‘a’ to point to ‘g’, which is equal to 10.
- The next call to `func()` will have ‘a’ point to allocated memory for an `int` array of size 3, which is specified by the second input argument.
/* File: pointer2.c */
#include <stdio.h>
#include <stdlib.h>

int func(size_t n, int **aa);
int g=10;

int main() {
    int *a, status;
    func(0, &a);   // a pointer to g
    printf("*a = %d, a[0] = %d\n", *a, a[0]);
    status =func(1, &a);// a pointer to allocated memory
    if(!status) {
        printf("*a = %d, a[0] = %d\n", *a, a[0]);
        free(a);
    }
    return 0;
}

int func(size_t n, int **aa) {
    if(n==0) {
        *aa = &g;
    }
    else {
        *aa = (int *)malloc(n*sizeof(int));
        if(*aa == NULL) {
            printf("Error: no memory\n");
            return -1;
        }
        **aa = 20;
    }
    return 0;
}
Array of Pointers

- Since pointers are also variables, arrays of pointers are supported. Below is an example of how arrays of pointers can be declared and used:

```c
char *s1 = "abc";
char *s2 = "123";
char *p[3]; // declares an array of 3 pointers to char;
p[0] = s1;
p[1] = s2;
p[2] = NULL;
```
• Arrays of pointers are very useful in some cases. Consider the following 2-dimensional array of char

```c
char day1[7][10] = {“Sunday”, “Monday”, “Tuesday”, “Wednesday”, “Thursday”, “Friday”, “Saturday”};
```

![Diagram of 2-dimensional array]

```plaintext
day1

S Sunday\0
M Monday\0
T Tuesday\0
W Wednesday\0
T Thursday\0
F Friday\0
S Saturday\0
```
Array of pointers to char

```c
```

The expression `day2[1][3]` has the value of 'd' of "Monday".

The advantage of using `day2`, the array of pointers, is that each pointer can point to arrays with different length rather than the fixed length of 10 bytes.
Example:

/* File: daystr.c */
#include <stdio.h>
#define DAY 7

const char *daystr(int n) {
    static const char day[7] = {
        "Sunday", "Monday", "Tuesday",
        "Wednesday", "Thursday", "Friday", "Saturday"};
    return (n < 0 || n > 6) ? NULL : day[n];
}

int main() {
    int i;
    const char *p;
    for(i = 0; i < DAY; i++) {
        p = daystr(i);
        printf("%s
", p);
    }
    return 0;
}
> daystr.c
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Use NULL to indicate the end of array of strings.

**Example**: A list of names terminated by NULL.

```c
char *name[] = {"John",
                "Doe",
                "Peter",
                NULL};

int i;
for(i=0; name[i]!=NULL; i++) {
    printf("name[%d] = %s\n", i, name[i]);
}
```

**Output:**

name[0] = John
name[1] = Doe
name[2] = Peter
Problem Statement

Calculate the mean value for accessing a Web page for everyday of the week in a given five weeks. Display names of days using an array of pointer to char.
/* File: dayhit.c */
#define DAY 7
#define WEEK 5

const char *day[] = {"Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"};

int main() {
    int a[][DAY] = {
        367, 654, 545, 556, 565, 526, 437,
        389, 689, 554, 526, 625, 537, 468,
        429, 644, 586, 626, 652, 546, 493,
        449, 689, 597, 679, 696, 568, 522,
        489, 729, 627, 729, 737, 598, 552};

    int i, j, daytotal;
    double meanval;

    printf("%-10s%-10s
", "Day", "Mean hit");
    printf("-----------------------\n");
    for(j = 0; j < DAY; j++) {
        daytotal = 0;
        for(i = 0; i < WEEK; i++) {
            daytotal += a[i][j];
        }
        meanval = (double)daytotal/WEEK;
        printf("%-10s%8.2f\n", day[j], meanval);
    }
    return 0;
}
Execution and Output

> day.c

<table>
<thead>
<tr>
<th>Day</th>
<th>Mean hit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sunday</td>
<td>424.60</td>
</tr>
<tr>
<td>Monday</td>
<td>681.00</td>
</tr>
<tr>
<td>Tuesday</td>
<td>581.80</td>
</tr>
<tr>
<td>Wednesday</td>
<td>623.20</td>
</tr>
<tr>
<td>Thursday</td>
<td>655.00</td>
</tr>
<tr>
<td>Friday</td>
<td>555.00</td>
</tr>
<tr>
<td>Saturday</td>
<td>494.40</td>
</tr>
</tbody>
</table>
Pointer to Functions

- A function pointer is a variable containing the address of the function.
- A function’s address is the entry point of the function. So, a function’s pointer can be used to call a function.
- A function pointer can be assigned, placed in arrays, passed to functions, returned by functions, etc.
- Below are some declarations of function pointers.

```c
void (*pf1) (void);
int  (*pf2) ();
int  (*pf3) (double f);
```

- Similar to an array name, a function name also represents for the address of a function. However, the address operator ‘&’ is ignored. For example, the statement

```c
pf = &fun;
```

is treated as

```c
pf = fun;
```
**Example: Call function by using function’s pointer**

```c
/* File: funptr.c */
#include <stdio.h>

int fun(double f) {
    printf("f = %f\n", f);
    return 0;
}

int main() {
    int (*pf)(double f);

    fun(10.0);
    pf = fun; // no & before fun
    pf(20.0); // call function fun by calling pf

    return 0;
}
```

**Output:**

```
f = 10.000000
f = 20.000000
```
• **Typedef**

The keyword *typedef* provides a mechanism for creating synonyms (or aliases) for previously defined data types. For example, the declaration

```c
typedef int i32_t;
```

makes the name `i32_t` a synonym for `int`. The type `i32` is exactly the same as type `int` in declarations, cast, and other situations. Following declarations are equivalent after *typedef*.

```c
i32_t i;  <=>  int i;
```
• More Examples

1) In Windows 32-bit machines,

```c
typedef unsigned int size_t;
size_t i = 10;  
```  

The data type `size_t` is typedefed in header files `stdio.h` and `stdlib.h`.

2) `typedef int *IPTR;`
```c
IPTR p1, p2;  
```  

3) `typedef int (*FUNCPTR)(double);`
```c
FUNCPTR pf;  
```
**Example**: Call function by using function’s pointer and typedef.

```c
#include <stdio.h>

typedef int (*FUNCPTR)(double);

int fun(double f) {
    printf("f = %f\n", f);
    return 0;
}

int main() {
    FUNCPTR pf;

    fun(10.0);
    pf = fun; // no & before fun
    pf(20.0); // call function fun by calling pf
    return 0;
}
```

**Output:**

```
f = 10.000000
f = 20.000000
```
Using a pointer to function as a function argument.

```c
/* File: funptrarg.c */
#include<stdio.h>

typedef int (*FUNCPTER)(double);

int f1(double f) {
    printf("f = %f\n", f);
    return 0;
}

/* or int f2( int (*pf)(int), double f) {
int f2(FUNCPTER pf, double f) {
    pf(f);
    return 0;
}

int main() {
    f2(f1, 5.0);
    return 0;
}
```

Output:

```
f = 5.000000
```
• **Sample Problem**

The system in Figure 1 (a) consists of a single body with mass \( m \) moving on a horizontal surface. An external force \( p \) acts on the body. The coefficient of kinetic friction between body and horizontal surface is \( \mu \). The freebody diagram for the system is shown in Figure 1 (b).

![System Diagram and Free Body Diagram](image-url)

Figure 1: The system diagram and FBD of a sample problem
The nomenclature related to the modeling of the system is listed below.

- **m** -- mass of the body
- **x** -- position of the body
- **v** -- velocity of the body
- **a** -- acceleration of the body
- **g** -- gravitational acceleration
- **µ** -- friction coefficient
- **f** -- friction force
- **N** -- normal force

**Equation of motion:**

The equation of the motion of the system can be derived based on the Newton's second law.

$$N = mg \quad (1)$$

$$f = \mu N \quad (2)$$

$$p-f = ma \quad (3)$$

From equation (1), (2) and (3), the formula for calculating the acceleration of the rigid body can be derived as follows.

$$a = (p - \mu mg)/m \quad (4)$$
• **Problem Statement**

For the system shown in Figure 1(a), given $m = 5$ kg, $g = 9.81$ m/s$^2$, $\mu = 0.2$. The external force $p$ is a function of time $t$ expressed in the following two formulas.

1) $p(t) = 4\cos(t-3)$ when $t \geq 0$
2) $p(t) = 4(t^2-3)$ when $t \geq 0$

Write a program to calculate and plot the accelerations $a$ during $t = 0$ to $t = 10$ with variable number of points. The user can select a formula for the external force.
Program1:

Plot accelerations using pointer to function.

```c
/* File: accelfunptr.c */
#include <stdio.h>
#include <chplot.h>
define M_G 9.81

double force1(double t) {
    double p;
    p = 4*cos(t-3);
    return p;
}

double force2(double t) {
    double p;
    p = 4*(t*t-3);
    return p;
}

int accel(int n, int num) {
    double a[n], t[n], t0, tf;
    double mu, m, p;
    double (*force)(double t);
    int i;
    mu = 0.2;
    m = 5;
    t0 = 0.0;
    tf = 10.0;
    if(num==1)
        force = force1;
    else
        force = force2;
    for(i=0; i<n; i++) {
        t[i] = t0+i*(tf-t0)/(n-1);
        p = force(t[i]);
        a[i] = (p-mu*m*M_G)/m;
    }
    plotxy(t, a, "Acceleration Plot", "time (s)", "accel (m/s^2)");
    return 0;
}

int main() {
    accel(200, 1); /* with 200 points and force= 4*cos(t-3); */
    accel(200, 2); /* with 200 points and force= 4*(t*t-3); */
    return 0;
}
```
Output of the program with 200 points:

Acceleration Plot
Program2:

Using pointer to function as argument of a function.

```c
/* File: accelpassfun.c */
#include <stdio.h>
#include <chplot.h>
define M_G   9.81

double force1(double t) {
    double p;
    p = 4*cos(t-3);
    return p;
}

double force2(double t) {
    double p;
    p = 4*(t*t-3);
    return p;
}

int accel(int n, double (*force)(double t)) {
    double a[n], t[n], t0, tf;
    double mu, m, p;
    int i;

    mu = 0.2;
    m = 5;
    t0 = 0.0;
    tf = 10.0;
    for(i=0; i<n; i++) {
        t[i] = t0 + i*(tf-t0)/(n-1);
        p = force(t[i]);
        a[i] = (p-mu*m*M_G)/m;
    }

    plotxy(t, a, "Acceleration Plot", "time" "(s)", "accel (m/s^2)");
    return 0;
}

int main() {
    accel(200, force1); /* with 200 points and force= 4*cos(t-3); */
    accel(200, force2); /* with 200 points and force= 4*(t*t-3); */
    return 0;
}
```
† Slides for optional topics in C
Example 2

```c
> double d[10], *dptr
> dptr = &d[0]
00E815F0
> dptr = dptr + 1
00E815F8
> float complex z1[5], *zp1
> zp1 = &z1[0]
00E87E60
> zp1 = zp1 + 1
00E87E68
> double complex z2[5], *zp2
> zp2 = &z2[0]
00E87B00
> zp2 = zp2 + 1
00E87B10
> 
```
The following example demonstrates how an array of pointers can be used to eliminate complicated storage management and overheads of moving lines.

**Example:**

```c
char *p[3] = {"ABC", "HIJKL", "EF"};
char *tmp;
...
tmp = p[1];
p[1] = p[2];
```

(Before swapping texts.)

(After swapping texts.)
• Function Returns a Pointer to a Function

```
int (*fun(int i))(double);
```

Function `fun()` returns a pointer to a function with an argument of double type and return type of int. Using typedef, the above declaration can be written as

```
typedef int (*FUNCPTTR)(double);
FUNCPTTR fun(int i);
```
Example:

```c
/* File: returnfunptr.c */
#include <stdio.h>

typedef int (*FUNCPTR)(double);
int fun(double f) {
    printf("f = %f\n", f);
    return 0;
}

FUNCPTR fun2(int i)
/* or int (*fun2(int i))(double) */
{
    if(i)
        return fun;
    else
        return NULL;
}

int main() {
    FUNCPTR pf;
    pf = fun2(1); // return fun in fun2
    pf(20.0); // call function fun by calling pf
    return 0;
}
```

Output:

```
f = 20.000000
```
Pointer to Arrays

- Pointers and one-dimensional arrays

```c
int *p, a[3], b[4];
p = a    // p[i] == a[i]  
    /* ... */
p = b    // p[i] == b[i] 
```

The pointer `p` points to the address `&a[0]` after the statement `p = a`. The variable names `p` and `a` can be used interchangeably. The pointer `p` can point to a different array, such as `b`, by assigning an address of other arrays.
Pointers and multi-dimensional arrays

A pointer to arrays of fixed-length is declared as

\[ \text{type} \ (\ast \text{name}) \ [\text{expr}] ; \]

where \text{type} contains the declaration specifiers that specify a type, \text{name} is a declarator that contains an identifier, and \text{expr} is an integral constant expression.

\begin{verbatim}
int (*p2)[3], int a1[2][3], a2[3][3];
p2 = a1;  // p2[i][j] == a1[i][j]
...
p2 = a2;  // p2[i][j] = a2[i][j]
\end{verbatim}

- declare \text{p2} as a pointer to arrays of three elements of \textbf{int} type. First, \text{p2} points to array \text{a1} of 2*3 elements. The element \text{p2}[i][j] share the same memory space with element \text{a1}[i][j]. Later, pointer \text{p2} points to array \text{a2} with 3*3 elements. Then \text{p2} and \text{a2} can be used interchangeably.

- The pointer \text{p2} is a pointer to an array of 3 elements. It can only point to two-dimensional arrays with 3 columns, regardless of the first dimension.
```c
/* File: pointer2array.c */
#include <stdio.h>

int main() {
    int a[2][3] = {1,2,3,4,5,6};
    int b[2][3] = {10,20,30,40,50,60};
    int (*p)[3];

    p = a;   // p <==> a
    printf("p[1][1] = %d\n", p[1][1]);
    p[1][2] = 100;
    printf("a[1][2] = %d\n", a[1][2]);

    p = b;   // p <==> b
    printf("p[1][1] = %d\n", p[1][1]);
    printf("b[1][2] = %d\n", b[1][2]);

    return 0;
}
```

Output:

```
p[1][1] = 5
a[1][2] = 100
p[1][1] = 50
b[1][2] = 200
```
Dynamic Memory Allocation of Arrays

- Dynamic memory allocation of one-dimensional arrays

```c
double *a1;
/* ... source code to obtain vectLen at runtime */
a1 = (double *)malloc(vectLen*sizeof(double));
if(a1 == NULL){
    fprintf(stderr, "ERROR: %s(): no enough memory\n", __func__);
    exit(1);
}
/* ... source code to handle vector a1 */
free(a1);
/* ... source code no longer use a1 */
```
Example:
Use a pointer to pointer to allocate memory dynamically for a one-dimensional array.

Output:

```c
int main() {
    int *a=NULL, i;

    func(3, &a);    // a pointer to allocated memory
    for(i=0; i<3; i++) {
        a[i] = i;
        printf("a[%d] = %d\n", i, a[i]);
    }
    if(a != NULL)
        free(a);
    return 0;
}

void func(size_t n, int **a) {
*0 = (int *)malloc(n*sizeof(int));
if(*0 == NULL) {
    printf("Error: no memory\n");
}
}
```
• Dynamic memory allocation of two-dimensional arrays. (Method 1: The memory for the two-dimensional arrays is in a single contiguous block.)
int i;
    double **a2;
    /* ... source code to obtain m and n at runtime */
a2 = (double **)malloc(m * sizeof(double*));
if(a2 == NULL) {
    fprintf(stderr, "ERROR: %s(): no enough memory\n", __func__);
    exit(1);
}
a2[0] = (double *)malloc(m * n * sizeof(double));
if(a2[0] == NULL) {
    fprintf(stderr, "ERROR: %s(): no enough memory\n", __func__);
    exit(1);
}
for(i = 1; i < m; i++) {
    a2[i] = a2[0] + i * n;
}
    /* ... source code to handle vector a2 */
free(a2[0]);
free(a2);
    /* ... source code no longer use a2 */
```c
> int i, j
> double **a2
> a2 = (double **)malloc(3 * sizeof(double*)); // with size (3 X 4)
> 4006cdc0
> for(i=0; i<3; i++) printf("%p ", a2[i])
> 00000000 00000000 00000000
> a2[0] = (double *)malloc(3 * 4 * sizeof(double));
> a2[1] = a2[0] + 4                         // a2[i] = a2[0] + i * n
> a2[2] = a2[0] + 2*4                       // a2[i] = a2[0] + i * n
> for(i=0; i<3; i++) printf("%p ", a2[i])   // 1-dimension of pointer
> 4007bee8 4007bf08 4007bf28
> for(i=0; i<3; i++) for(j=0; j<4; j++) printf("%1.1f ", a2[i][j])
> 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
> a2[1][1]=3                                  // method 1
> *(*a2+1)+2)=4                               // <=> a2[1][2]=4, method 2
> for(i=0; i<3; i++) for(j=0; j<4; j++) printf("%1.1f ", a2[i][j])
> 0.0 0.0 0.0 0.0 0.0 3.0 4.0 0.0 0.0 0.0 0.0 0.0
```
• Dynamic memory allocation of two-dimensional arrays. (Memory for each row of the two-dimensional arrays is allocated separately.)
int i;
    double **a2;
    /* ... source code to obtain m and n at runtime */
a2 = (double **)malloc(m * sizeof(double*));
    if(a2 == NULL) {
        fprintf(stderr, "ERROR: %s(): no enough memory\n", __func__);
        exit(1);
    }
    for(i = 0; i < m; i++) {
        a2[i] = (double *)malloc(n * sizeof(double));
        if(a2[i] == NULL) {
            fprintf(stderr, "ERROR: %s(): no enough memory\n", __func__);
            exit(1);
        }
    }
    /* ... source code to handle vector a2 */
    for(i = 0; i < m; i++)
        free(a2[i]);
    free(a2);
    /* ... source code no longer use a2 */
Using Pointers to Pass Multi-Dimensional Arrays to a Function

- In C, a multi-dimensional array can be viewed as a one-dimensional array when each element of a multi-dimensional array is stored consecutively in the memory. The next example shows how to use a pointer to pass a 2*3 array \( a \) to a function.
Output:

Array a =
1 2 3
4 5 6
a[1][2] = 6

/* File: pass2Das1D.c */
#include<stdio.h>

void printArray(int m, int n, int *a) {
    int i, j;
    for(i = 0; i < m*n; i++) {
        if(i == n)
            printf("n");
        printf("%d ", *(a+i));
    }
    printf("n");
/* access element a[i][j] for mxn array a using *(a+i*n+j) */
i = 1;
j = 2;
printf("a[%d][%d] = %d \n", i, j, *(a+i*n+j));
}

int main() {
    int a[2][3] = {1, 2, 3,
                   4, 5, 6};
    printf("Array a = \n");
    printArray(2, 3, &a[0][0]);
    return 0;
}
• Use a double pointer (an array of pointers) to pass two-dimensional arrays to a function.

  – The following example shows how this can be done.
  – As can be seen from the example program, double pointers, `pa1` and `pa2`, are assigned to point to two 2D arrays, `a1` and `a2`, respectively. Of course, memory has to be allocated for each double pointer.
  – The double pointers are then used to pass the 2D arrays into function `func()` to sum all the elements the array.
**Example:**

```c
/* File: pass2Dcarray.c */
#include <stdio.h>
#include <stdlib.h>
#define M1 2
#define N1 3
#define M2 3
#define N2 4
double func(int m, int n, double **a);

int main() {
    double a1[M1][N1]={1, 2, 3, 4, 5, 6};
    double a2[M2][N2]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
    double **pa1, **pa2, sum;
    int i;
    pa1 = (double **)malloc(M1*sizeof(double*));
    // allocate array of pointers
    for(i=0; i<M1; i++) {
        pa1[i] = a1[i];   // pa1[i] points to i-th row of a1
    }
    sum = func(M1, N1, pa1);
    printf("sum = %f\n", sum);
    free(pa1);            // free allocated array

    pa2 = (double **)malloc(M2*sizeof(double*));
    for(i=0; i<M2; i++) {
        pa2[i] = a2[i];   // pa2[i] points to i-th row of a2
    }
    sum = func(M2, N2, pa2);
    printf("sum = %f\n", sum);
    free(pa2);            // free allocated array
    return 0;
}
```

**Output:**

```plaintext
sum = 21.000000
sum = 78.000000
```
• \( \text{pa1}[i][j] \) and \( \text{a1}[i][j] \) share the same memory.
• \( \text{pa1} \) is a pointer to a pointer to double.
• After allocating memory, \( \text{pa1} \) can be considered as an array of pointers to double. Pointer \( \text{pa1}[0] \) points to &\( \text{a1}[0][0] \). Pointer \( \text{pa1}[1] \) points to &\( \text{a1}[1][0] \).
• \( \text{pa2}[i][j] \) and \( \text{a2}[i][j] \) share the same memory with data type double.
Example: program pass2Dcarrayvla.c is equivalent to pass2Dcarray.c

```c
/* File: pass2Dcarrayvla.c */
#include <stdio.h>
#include <stdlib.h>

#define M1 2
#define N1 3
#define M2 3
#define N2 4

double func(int m, int n, double a[m][n]);
/* or double func(int m, int n, double a[*][*]); */
/* or double func(int, int, double [*][*]); */

int main() {
    double a1[M1][N1]={1.0, 2.0, 3.0, 4.0, 5.0, 6.0};
    double a2[M2][N2]={1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0};

    double sum;

    sum = func(M1, N1, a1);
    printf("sum = %f\n", sum);

    sum = func(M2, N2, a2);
    printf("sum = %f\n", sum);
    return 0;
}

double func(int m, int n, double a[m][n]) {
    double sum = 0;
    int i, j;

    for(i=0; i<m; i++) {
        for(j=0; j<n; j++) {
            sum += a[i][j];
        }
    }

    return sum;
}
```
Using Pointer for Hardware Interface

One important feature of C is its capability for hardware interface by accessing a specific memory location in a computer. This is achieved by pointing a pointer to a specific memory location or register by using cast operation.

For example, the following statements will assign the integer value at the memory location 0xFFF68FFE to variable i and set the byte at the memory address 0xFFFFF000 to 0b01101001.

```c
int i, *iptr;
char *cptr;
iptr = (int*)0xFFF68FFE;// point to the memory location at 0xFFF68FFE
i = *iptr;              // i equals to the value at 0xFFF68FFE;
cptr = (char*)0xFFFFF000;// point to the memory location at 0xFFFFF000
*cptr = 0b01101001;      // 0b01101001 is assigned to 0xFFFFF000
```
Additional topics in C
Implement a menu using array of pointers to functions

/* File: menu.c */
#include <stdio.h>
#include <stdlib.h>

int opt0(void) {
    printf("to handle option 0\n");
    return 0;
}

int opt1(void) {
    printf("to handle option 1\n");
    return 0;
}

int opt2(void) {
    printf("to exit\n");
    exit(0);
}

int getChoice(void) {
    int i;
    printf("input the choice (0,1,2): ");
    scanf("%d", &i);
    if (i > 2 || i < 0)
        i = 2;
    return i;
}

typedef int (*PF)(void);
int main() {
    PF options[] = { /* or int
        (*options[])() = { */
        opt0,
        opt1,
        opt2,
    };

    do {
        options[getChoice()]();
    } while(1);
    return 0;
}
• Passing a one-dimensional array with zero-offset to a function where the passed arrays are treated as nonzero-offset arrays
The subscript range of arrays d1 and d2 in the main function are from 0 to 9. These two arrays are passed to the function funct(). The formal arrays dd1 and dd2 are treated as arrays with subscript range from 1 to 10.

```c
/* File: oneDarrayrange1.c */
#include <stdio.h>
int main() {
    int i, oldlow = 0, oldup = 9, newlow = 1, newup = 10;
    double d1[10], d2[10];
    void funct(double dd1[], double *dd2, int low, int up);

    for(i=oldlow; i<=oldup; i++)
        d1[i] = 2; d2[i] = 2;
    funct(d1-newlow,d2-newlow,newlow,newup);
    for(i=oldlow; i<=oldup; i++)
        printf("d1[i] = %f\n",d1[i]);
}

void funct(double dd1[], double *dd2, int low, int up) {
    int i;
    for(i=low; i<=up; i++)
        dd1[i] += 2; dd2[i] += 2;
}
```
• Passing a one-dimensional array with nonzero-offset to a function where the passed arrays are treated as nonzero-offset arrays
/ * File: oneDarrayrange2.c */
#include <stdio.h>
#include <stdlib.h>

int main() {
    int i, low=1, up=10;
    double *d1, *d2;
double *mallocVector(int low, int up);
void funct(double dd1[], double *dd2, int low, int up);
void freeVector(double *d, int low);

    /* allocate memory for vectors d1[1:10], d2[1:10] */
d1 = mallocVector(low, up);
d2 = mallocVector(low, up);
for (i=low; i<=up; i++) {
    d1[i] = 2; d2[i] = 2;
}
funct(d1, d2, low, up);
for (i=low; i<=up; i++)
    printf("d1[i] = %f\n", d1[i]);
/* free memory for vectors d1[1:10], d2[1:10] */
freeVector(d1, low);
freeVector(d2, low);
}

void funct(double dd1[], double *dd2, int low, int up) {
    int i;
    for (i=low; i<=up; i++) {
        dd1[i] += 2; dd2[i] += 2;
    }
}

double *mallocVector(int low, int up) {
    double *v;
    v=(double *)malloc((unsigned) (up-low+1)*sizeof(double));
    return v-low;
}

void freeVector(double *v, int low) {
    free((char*) (v+low));
}
Slides for optional topics in Ch
/* File: oneDarrayrange2.ch */
#include <stdio.h>
#include <stdlib.h>

int main() {
    int i, low=1, up=10;
    double d1[low:up], d2[low:up];
    void funct(double dd1[1:], double dd2[1:], int low, int up);

    for(i=low; i<=up; i++){
        d1[i] = 2; d2[i] = 2;
    }
    funct(d1,d2,low,up);
    for(i=low; i<=up; i++)
        printf("d1[i] = %f\n",d1[i]);
}

void funct(double dd1[1:], double dd2[1:], int low, int up) {
    int i;
    for(i=1; i<=up; i++){
        dd1[i] += 2; dd2[i] += 2;
    }
}