Preprocessor Directives

Arrays

Collect 100 temperature samples

```
{int t1, t2, t3, …,t99,t100
                    }
```

```
{
int temperature[100];

}
```

Array Declaration

int a[100]; /*square brackets*/

a[0], a[1], …,a[99] /*starts from 0 */

n=10;m=11;

a[n]

a[n+m]

a[100] illegal /* error message: array exceed bound */

Array Initialization

int b[9]={1,5,9,4};

b[0]=1,b[1]=5, b[2]=9, b[3]=4, b[4]=0,…,b[8]=0

(remaining spaces are filled with zeroes)

int c[] = {1,3,5,7,9} /*automatically recognizes c[5];*/

printf("%d\n",c[2]); /*prints 5 */

c is the pointer constant points to the address of array c

Value of c is set to the address of the first cell of the array; this value cannot be changed

c = 6 /*Error*/

Pointer Constant & Pointer Variable

Pointer constant c

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

Pointer variable ptr

```
int* ptr; /*ptr is a pointer to integer cells; ptr can hold the address of an integer cell*/
ptr =c; /*ptr stores the address of the first cell of array c*/
ptr = &c[0]; /*same as above*/
ptr = &c[2]; /*ptr stores the address of the third cell of array c*/
```

The sizeof operator

int d[30];

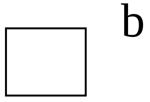
the value of sizeof (d) is 30*4 in system with 4 bytes for integers

➢Array Indexes and Cell Offsets

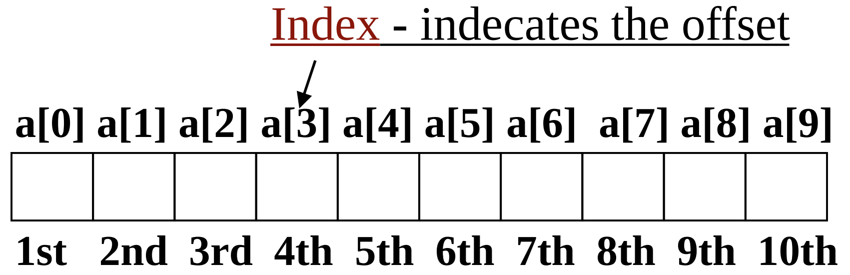
.

Variable definition — Cell

int b;



int a[10]; /* has 10 elements */



♦ In dealing with arrays, we use square brackets in two quite different ways.

1) When we define an array, the number of cells specified in square brackets.

```
int a[ 10 ];      /* defines a to be an array consisting of 10 elements. */
```

2) When we access a specific array element, we use the array's name together with an index enclosed in square brackets. Assuming the preceding definition for a, all of the expressions

```
a[ 0 ]      /* designates the first cell and has index 0*/
a[ 1 ]      /* designates the second cell and has index 1*/
a[ 2 ]      /* designates the third cell and has index 2*/
…
a[ 9 ]      /* designates the tenth cell and has index 9*/
```

Notice!

```
a[10]      /* Illegal reference. C may not warn
           you when your program contains a
           meaningless index */
```

♦ The number of cells in an array is given as a bracketed expression only once, in definition; thereafter, the bracketed expression is an index.

```
float quarks[16];              /* has 16 elements with type float */
int index1 = 1;
int index2 = 2;
quarks[0]                      /* has index 0 and the first element in array*/
quarks[6]                      /* has index 6 and the seventh element in array */
quarks[index1]                 /* has index 1 and the second element in array */
quarks[index1 + index2]        /* has index 3  and the fourth element in array */
quarks[index1 - index2]        /* has meaningless index -1. Warning depend on system*/
quarks[index1 * index2]        /* has index 2  and the third element in array */
quarks[index1 * index2 + 1]    /* has index 3 and the 4th element in array*/
quarks[3 / 2 + 1]              /* has index 2 and the third element in array*/
```

```
quarks[index1 / index2 ]       /* has index 0 and the first element in array */
quarks[index1 / index2 + 1]    /* has index 1 and the second element in array */
quarks[index2 / index1]        /* has index 2 and the third element in array */
```

♦ In defining an array, the number of cells is normally given as an integer constant, but an expression is permitted if the value of the expression is known at compile time.

```
#define MAX     10
   main() {
         int index1, index2;
         float temps[ MAX + 1];        /* Permitted because compiler can
                                          determine the value of MAX + 1. */
         int quarks[index1 + index2];  /* Illegal because index1 + index2 is not
                                          known until the run time. */
   }
```

```
int id[ 4 ] = { 26, 6, 600, 99 }
```

id				
	26	6	600	99
	[0]	**[1]**	**[2]**	**[3]**

/* Initializing the array **id** in the definition.*/

int numbs[9] = { 1, 2, 3, 5, 6, 8 };

numbs									
	1	2	3	5	6	8	0	0	0
	[0]	**[1]**	**[2]**	**[3]**	**[4]**	**[5]**	**[6]**	**[7]**	**[8]**

/*fewer initial values than there are cells, each of the reminder cells is initialized to 0 */

int numbs[6] = { 1, 2, 3, 5, 6, 8, 9 };

/* Error to supply more initial values than there are cells.*/

♦ If we define and initialize arrays, we can omit the integers that specify the number of cell. The compiler allocates exactly as many as cells are needed to store the initial data.

int age[ ] = { 6, 1, 8, 9, 0, 5 };

age						
	6	1	8	9	0	5
	[0]	**[1]**	**[2]**	**[3]**	**[4]**	**[5]**

Notice: We can omit the number of cells in a definition only if the array is initialized.

- **Arrays and Pointers**

The array's name provides the address of its first cell, and the index provides the offset from this first cell.

```
#include <stdio.h>
#include <stdlib.h>
main ( ) {
        int lucky_nums[ ] = { 1, 2, 6, 8, 9 };
        /* Print 5th lucky number */
        printf ( "%d\n", lucky_nums[4] );          /* Print 9 */
        return EXIT_SUCCESS;
}
```

base address → 200

1	2	6	8	9
[0]	**[1]**	**[2]**	**[3]**	**[4]**

lucky_nums

/* The system locates the 5th cell by using lucky_nums as a pointer to a base address and the index 4 as an offset from this base address. */

An array's name as a pointer to the first cell in the arrays.

lucky_nums as a **pointer** to **base address** ⟷ &lucky_nums[0] ⟷ 200

Notice:

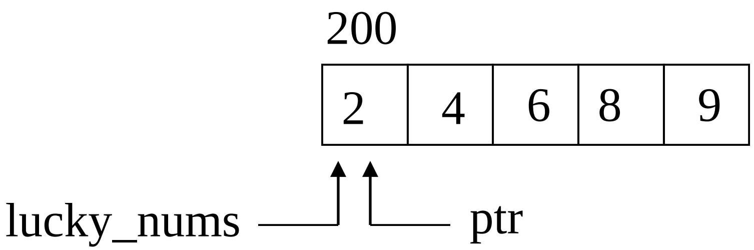
- An array’s name is a **pointer constant**; its value is set to the address of the first cell of the array when the array is defined and cannot be changed thereafter.

```
lucky_nums = 345;   /* ERROR - because luchy_nums is a pointer constant */
```

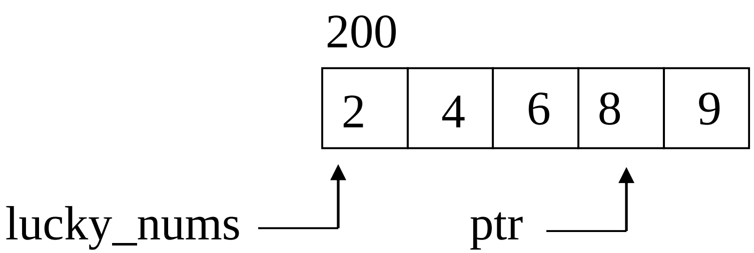
- C support **pointer variable** in addition to pointer constant. Like a pointer constant, a pointer variable holds the address of some cell; but unlike a pointer constant, a pointer variable can have its value set through an assignment operation.

```
#include <stdio.h>
#include <stdlib.h>
main( ) {
        int lucky_nums[ ] = { 2, 4, 6, 8, 9};
        int * ptr;                      /* ptr’s definition, define ptr pointer to int */
        ptr = lucky_nums;               /* ptr --> 1st cell */
        ptr = &lucky_nums[ 0 ];         /* ditto */
        ptr = &lucky_nums[ 3 ];         /* ptr --> 4th cell */
        return EXIT_SUCCESS;
}
```

Note: the pointer variable ptr, unlike the pointer constant lucky_nums, can occur as the left-hand side of an assignment statement.



The pointer ptr holds the address of the first cell in the array lucky_nums.



The pointer ptr holds the address of the 4th cell in the array lucky_nums.

- **Implementation Issues: What the C Programmer Can Ignore**

↗The C programmer can access any element in an array without knowing how big each element is.

int a[5];

int { 16 bits, on one computer system
32 bits, on another system

On either system, we can access the third element as a[2] or the fourth element as a[3] or …. regardless of whether and how the elements differ in number of bits.

- **The sizeof Operator and Arrays**

The **sizeof** operator may be used with arrays.

int disease_codes[30];

the value of

sizeof (disease_codes)

is

30 * 4 = 120

This means that the array disease_codes occupies 120 bytes.

➢ Real World Application: The Fourier Transform

♦ *Problem*

Computer a specified number of terms of the discrete Fourier transform (h1, h2) of the function g. If g(t) is defined for t = 0, … r - 1, the functions h1 and h2 are defined by the formulas

$$\mathbf{h1(s) = \sum_{t=0}^{r-1} g(t)\cos\left(\frac{2\pi st}{r}\right)}$$

$$\mathbf{h2(s) = \sum_{t=0}^{r-1} g(t)\sin\left(\frac{2\pi st}{r}\right)}$$

for s = 0, …, r - 1. The function h1 is the cosine component, and h2 is the sine component of the Fourier transform. The values h1(s) and h2(s) for s = 0, …, r - 1 are called the *Fourier coefficients* for g.

Print the amplitude spectrum

$$\mathbf{\sqrt{h1[s]^2 + h2[s]^2}}$$

for the Fourier coefficients that were computed, except for s = 0.

Sample Input/Output

Input

$\left.\text{32}\right\}$ r

$$\left.\begin{array}{l}\text{0.000000 0.577774 1.089790 1.479450 1.707107 1.755349}\\\text{1.630986 1.363469 1.000000 0.598102 0.216773 -0.092410}\\\text{-0.292893 -0.368309 -0.324423 -0.187593 0.000000 0.187593}\\\text{0.324423 0.368309 0.292893 0.092410 -0.216773 -0.598102}\\\text{-1.000000 -1.363469 -1.630986 -1.755349 -1.707107 -1.479450}\\\text{-1.089790 -0.577773}\end{array}\right\}\text{ r values of g}$$

$\left.\text{10}\right\}$ the number of amplitude spectrum values to print.

Output

Spectrum:

16.000002

15.999996

0.000002

0.000001

0.000001

0.000001

0.000002

0.000001
0.000000
0.000002

♦ ***Solution***

We use three float arrays: g[300], h[300], and h2[300] to store the values of g, h1, and h2. The program can accommodate a function g of up to 300 values, and we can compute up to 300 terms of the Fourier series. After reading the values of g, we use nested for loop to compute the Fourier coefficients and the another for loop to print the amplitude spectrum.

Note: We cannot first read the number of values of g and then define our arrays to be of that size, because C requires the size of the array to be specified as a constant.

♦ ***C Implementation***

/* This program computes the first max_coeff + 1 terms of the discrete Fourier transform (h1, h2) of the function g, which is represented as the array

g[0], g[1], …, g[resolution - 1]

h1 is the cosine component and h2 is the sine component of the Fourier transform; h1 and h2 are represented as the arrays

```
        h1[ 0 ], h1[ 1 ], …, h1[ max_coeff ]
        h2[ 0 ], h2[ 1 ], …, h2[ max_coeff ]
The program prints the amplitude spectrum
        sqrt( h1[i]^2 + h2[i]^2)
for I = 1, 2, …, max_coeff.
The input must be of the form
resolution
g[0] g[1]  … g[ resolution - 1]
max_coeff
*/

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#define MAX_RESOLUTION      300

main( ) {
```

```
int resolution;
float scale;
float g[ MAX_RESOLUTION ], h1[MAX_RESOLUTION ],
        h2[MAX_RESOLUTION ];
float pi = 3.14159265;
int I, s, t, max_coeff;
scanf ( “%d”, &resolution );
scale = 2 * pi / resolution;
for( I = 0; I < resolution; I++ )
        scanf ( “%f”, &g[ I ] );
scanf ( “%d”, &max_coeff );
/*compute Fourier coefficients */
for ( s = 0; s <= max_coeff; s++ ) {
        h1[ s ] = h2[ s ] = 0.0;
        for( t = 0; t < resolution; t++ ) {
                h1[ s ] += g[ t ] * cos(scale * s * t );
                h2[ s ] += g[ t ] * sin(scale * s * t );
        }
}
```

```
    printf ( “\n\nSpectrum:\n” );
    for( I = 1; I <= max_coeff; I++ )
        printf( “%f\n”, sqrt( h[I] * h1[I] + h2[I] * h2[I] ) ); return
    EXIT_SUCCESS;
}
```