“Arrays and Pointers”

Using Bloodshed Dev-C++

Heejin Park

Hanyang University

Introduction

- Arrays
- Multidimensional Arrays
- Pointers and Arrays
- Functions, Arrays, and Pointers
- Pointer Operations
- Protecting Array Contents
- Pointers and Multidimensional Arrays
- Variable-Length Arrays (VLAs)
- Compound Literals

Arrays

Arrays

- Recall that an ***array*** is composed of a series of elements of one data type.
- You use ***declarations*** to tell the compiler when you want an array.

```
int main(void)
{
    float candy[365];        /* array of 365 floats */
    char code[12];           /* array of 12 chars   */
    int states[50];          /* array of 50 ints    */
    ...
}
```

Arrays

Initialization

- Arrays are often used to store data needed for a program.
- You know you can initialize single-valued variables in a declaration with expressions such as

```
int fix = 1;

float flax = PI * 2;
```

Arrays

Initialization

- C extends initialization to arrays with a new syntax, as shown next:

```
int main(void)
{
    int powers[8] = {1,2,4,6,8,16,32,64}; /* ANSI only */
    ...
}
```

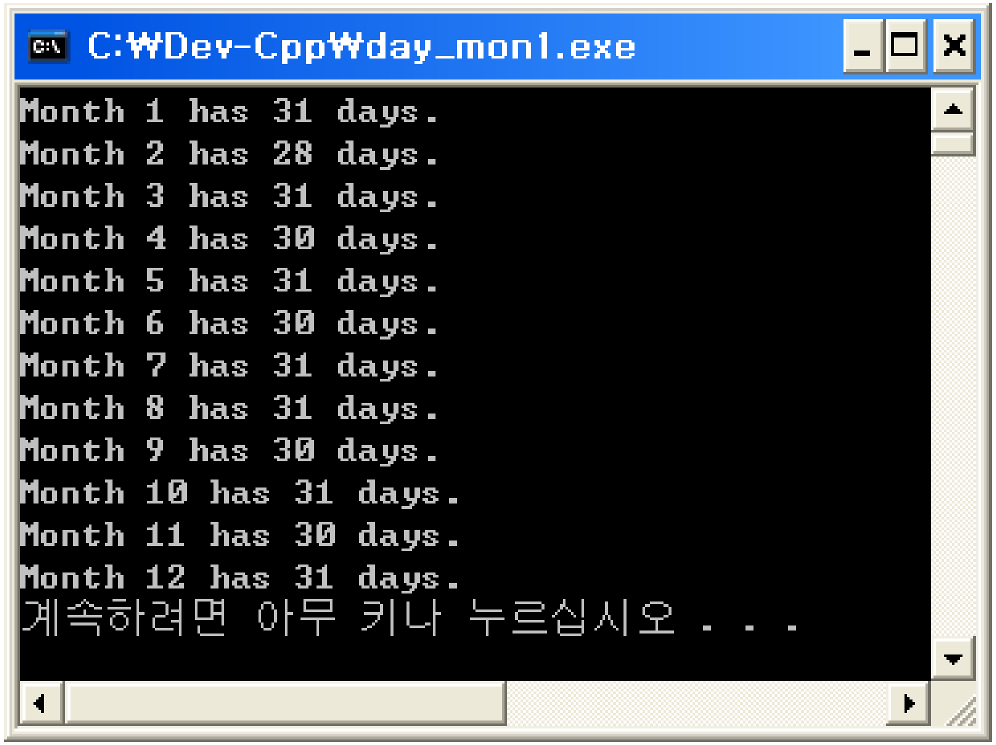
Arrays

The day_mon1.c Program

```
#include <stdio.h>
#define MONTHS 12

int main(void)
{
    int days[MONTHS] = {31,28,31,30,31,30,31,31,30,31,30,31};
    int index;

    for (index = 0; index < MONTHS; index++)
        printf("Month %d has %2d days.\n", index +1,
               days[index]);

    return 0;
}
```

Arrays

The day_mon1.c Program



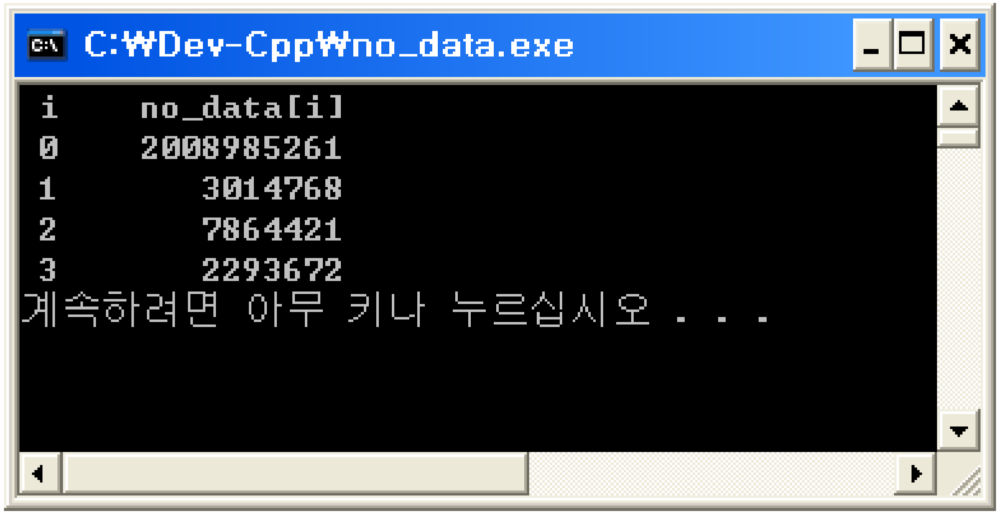
Arrays

The no_data.c Program

```
#include <stdio.h>
#define SIZE 4

int main(void)
{
    int no_data[SIZE];   /* uninitialized array */
    int i;

    printf("%2s%14s\n",
           "i", "no_data[i]");
    for (i = 0; i < SIZE; i++)
        printf("%2d%14d\n", i, no_data[i]);

    return 0;
}
```

Arrays

The no_data.c Program



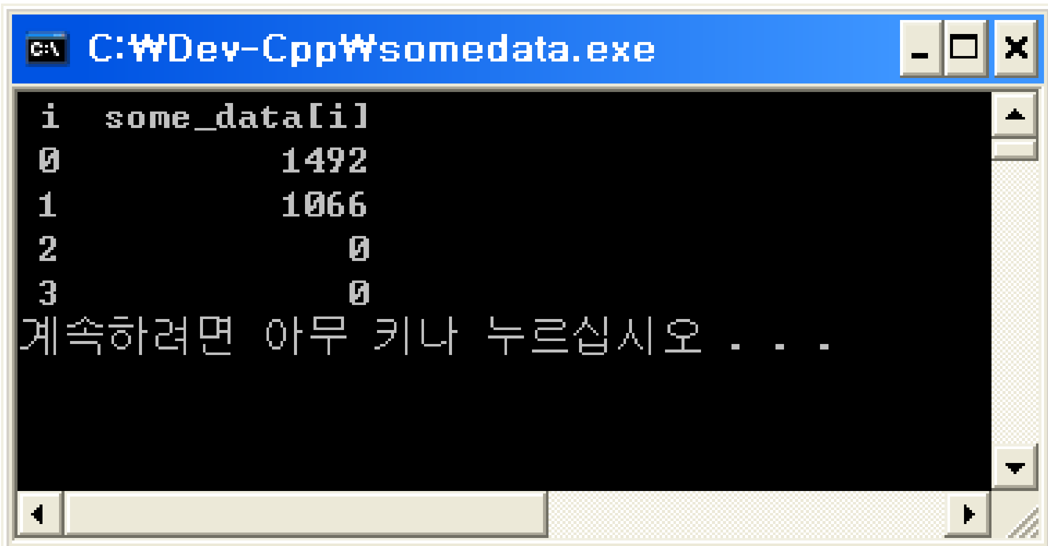
Arrays

The somedata.c Program

```
#include <stdio.h>
#define SIZE 4

int main(void)
{
    int some_data[SIZE] = {1492, 1066};
    int i;

    printf("%2s%14s\n",
           "i", "some_data[i]");
    for (i = 0; i < SIZE; i++)
        printf("%2d%14d\n", i, some_data[i]);

    return 0;
}
```

Arrays

The somedata.c Program



Arrays

The day_mon2.c Program

```
#include <stdio.h>

int main(void)
{
    const int days[] = {31,28,31,30,31,30,31,31,30,31};
    int index;

    for (index = 0; index < sizeof days / sizeof days[0]; index++)
        printf("Month %2d has %d days.\n", index +1,
               days[index]);

    return 0;
}
```

Arrays

The day_mon2.c Program

```
C:\Dev-Cpp\day_mon2.exe
Month  1 has 31 days.
Month  2 has 28 days.
Month  3 has 31 days.
Month  4 has 30 days.
Month  5 has 31 days.
Month  6 has 30 days.
Month  7 has 31 days.
Month  8 has 31 days.
Month  9 has 30 days.
Month 10 has 31 days.
계속하려면 아무 키나 누르십시오 . . .
```

Arrays

Designated Initializers (C99)

- C99 has added a new capability: **designated initializers**.
- With traditional C initialization syntax, you also have to initialize every element preceding the last one:

```
int arr[6] = {0,0,0,0,0,212};  // traditional syntax
```

- With C99, you can use an index in brackets in the initialization list to specify a particular element:

```
int arr[6] = {[5] = 212}; // initialize arr[5] to 212
```

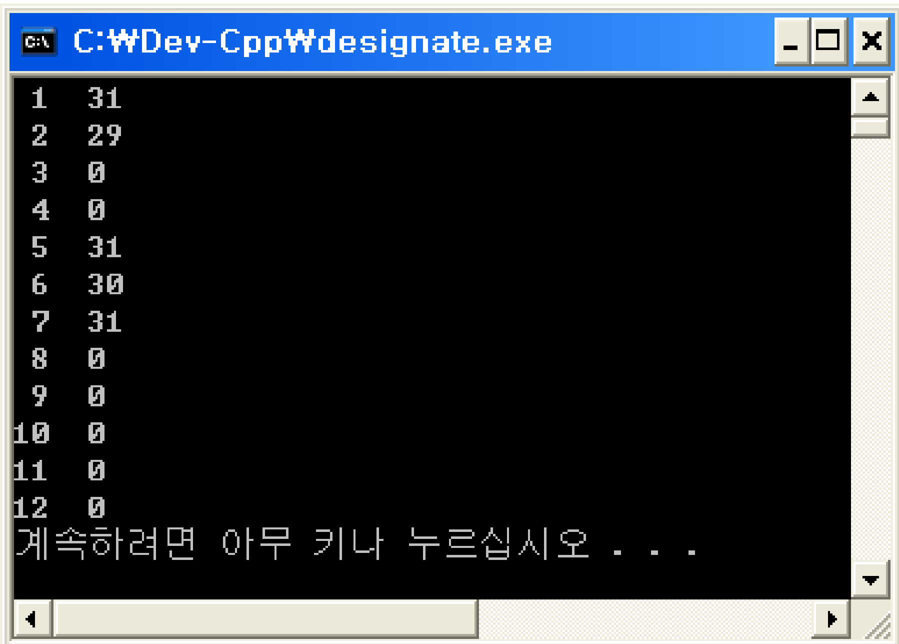
Arrays

The designate.c Program

```
#include <stdio.h>
#define MONTHS 12

int main(void)
{
    int days[MONTHS] = {31,28, [4] = 31,30,31, [1] = 29};
    int i;

    for (i = 0; i < MONTHS; i++)
        printf("%2d  %d\n", i + 1, days[i]);

    return 0;
}
```

Arrays

The designate.c Program



Arrays

Assigning Array Values

- After an array has been declared, you can assign values to array members by using an array index, or subscript.
- For example, the following fragment assigns even numbers to an array:

```
#include <stdio.h>
#define SIZE 50

int main(void)
{
    int counter, evens[SIZE];

    for (counter = 0; counter < SIZE; counter++)
        evens[counter] = 2 * counter;
  ...
}
```

Arrays

Assigning Array Values

- C doesn't let you assign one array to another as a unit.
- The following code fragment shows some forms of assignment that are not allowed:

```
#define SIZE 5
int main(void)
{
    int oxen[SIZE] = {5,3,2,8};       /* ok here      */
    int yaks[SIZE];

    yaks = oxen;                      /* not allowed  */
    yaks[SIZE] = oxen[SIZE];          /* invalid      */
    yaks[SIZE] = {5,3,2,8};           /* doesn't work */
```

Arrays

Array Bounds

- You have to make sure you use array indices that are within bounds; that is, you have to make sure they have values valid for the array.
- For instance, suppose you make the following declaration:

```
int doofi[20];
```

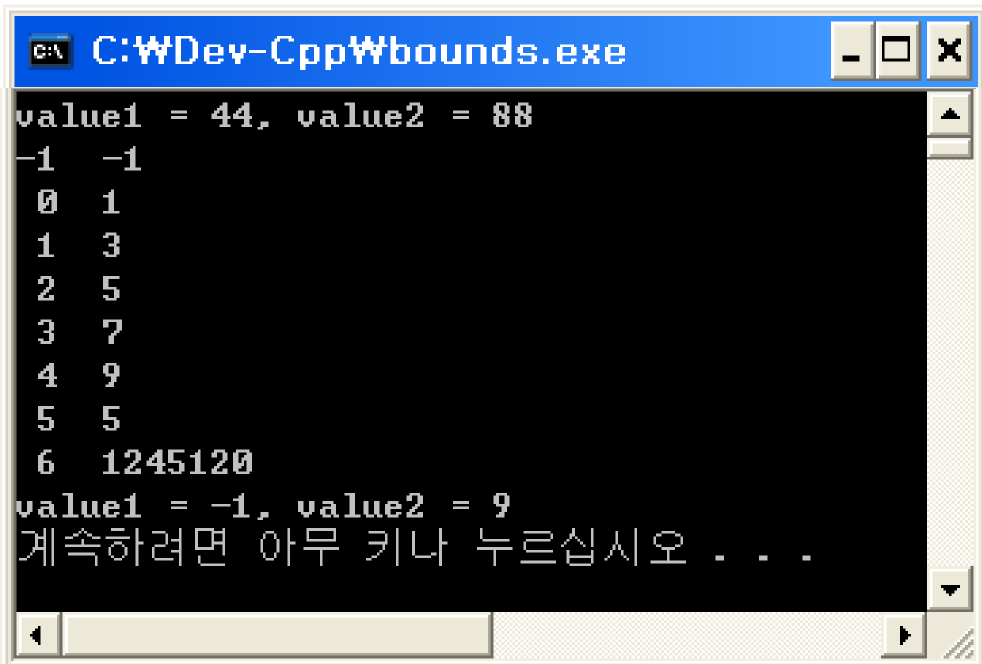
Arrays

The bounds.c Program

```
#include <stdio.h>
#define SIZE 4

int main(void)
{
    int value1 = 44;
    int arr[SIZE];
    int value2 = 88;
    int i;

    printf("value1 = %d, value2 = %d\n", value1, value2);
    for (i = -1; i <= SIZE; i++)
        arr[i] = 2 * i + 1;

    for (i = -1; i < 7; i++)
        printf("%2d  %d\n", i , arr[i]);
    printf("value1 = %d, value2 = %d\n", value1, value2);

    return 0;
}
```

Arrays

The bounds.c Program



Arrays

The bounds.c Program

- One simple thing to remember is that array numbering begins with 0.
- One simple habit to develop is to use a symbolic constant in the array declaration and in other places the array size is used:

```
#define SIZE 4
int main(void)
{
    int arr[SIZE];
    for (i = 0; i < SIZE; i++)
    ...
```

Arrays

Specifying an Array Size

- So far, the examples have used integer constants when declaring arrays:

```
#define SIZE 4
int main(void)
{
    int arr[SIZE];      // symbolic integer constant
    double lots[144];   // literal integer constant
    ...
```

Arrays

Specifying an Array Size

- A `sizeof` expression is considered an integer constant, but a `const` value isn't.
- Also, the value of the expression must be greater than 0:

```
int n = 5;
int m = 8;
float a1[5];                    // yes
float a2[5*2 + 1];              // yes
float a3[sizeof(int) + 1];      // yes
float a4[-4];                   // no, size must be > 0
float a5[0];                    // no, size must be > 0
float a6[2.5];                  // no, size must be an integer
float a7[(int)2.5];             // yes, typecast float to int constant
float a8[n];                    // not allowed before C99
float a9[m];                    // not allowed before C99
```

Multidimensional Arrays

Multidimensional Arrays

- The better approach is to use an array of arrays.
- The master array would have five elements, one for each year. Each of those elements, in turn, would be a 12-element array, one for each month.
- Here is how to declare such an array:

```
float rain[5][12];  // array of 5 arrays of 12 floats
```

Multidimensional Arrays

Multidimensional Arrays

- One way to view this declaration is to first look at the inner portion:

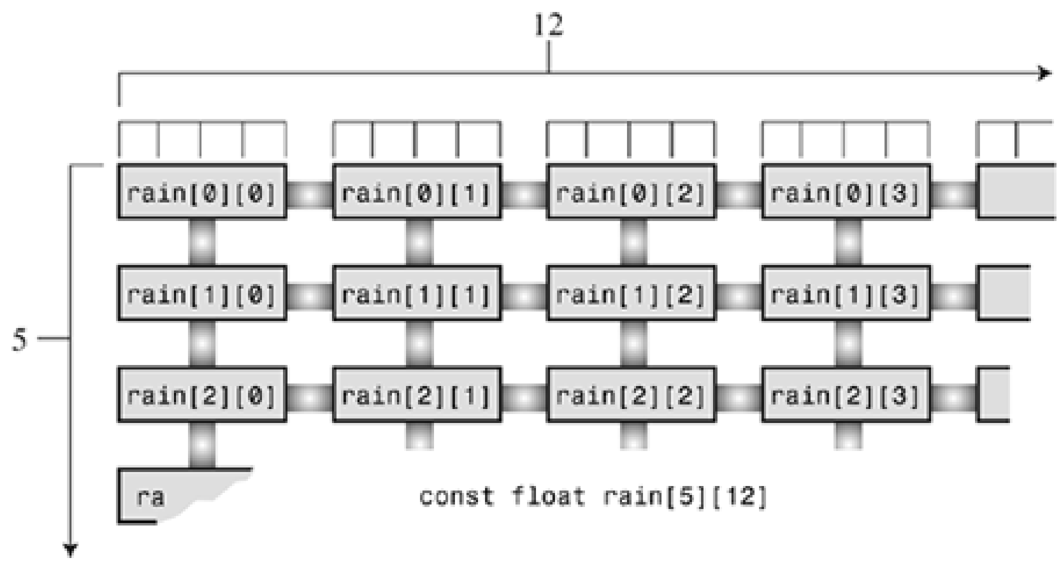
```
float rain[5][12];   // rain is an array of 5 somethings
```

- It tells us that rain is an array with five elements. But what is each of those elements? Now look at the remaining part of the declaration:

```
float rain[5] [12];   // an array of 12 floats
```

Multidimensional Arrays

- Two-dimensional array



Multidimensional Arrays

The rain.c Program(1/2)

```
#include <stdio.h>
#define MONTHS 12    // number of months in a year
#define YEARS   5    // number of years of data

int main(void)
{
 // initializing rainfall data for 2000 - 2004
    const float rain[YEARS][MONTHS] =
    {
        {4.3,4.3,4.3,3.0,2.0,1.2,0.2,0.2,0.4,2.4,3.5,6.6},
        {8.5,8.2,1.2,1.6,2.4,0.0,5.2,0.9,0.3,0.9,1.4,7.3},
        {9.1,8.5,6.7,4.3,2.1,0.8,0.2,0.2,1.1,2.3,6.1,8.4},
        {7.2,9.9,8.4,3.3,1.2,0.8,0.4,0.0,0.6,1.7,4.3,6.2},
        {7.6,5.6,3.8,2.8,3.8,0.2,0.0,0.0,0.0,1.3,2.6,5.2}
    };

    int year, month;
    float subtot, total;

    printf(" YEAR    RAINFALL  (inches)\n");
```

Multidimensional Arrays

The rain.c Program(2/2)

```
    for (year = 0, total = 0; year < YEARS; year++)
    {
        for (month = 0, subtot = 0; month < MONTHS; month++)
            subtot += rain[year][month];
        printf("%5d %15.1f\n", 2000 + year, subtot);
        total += subtot;
     }
    printf("\nThe yearly average is %.1f inches.\n\n",
            total/YEARS);
    printf("MONTHLY AVERAGES:\n\n");
    printf(" Jan  Feb  Mar  Apr  May  Jun  Jul  Aug  Sep  Oct ");
    printf(" Nov  Dec\n");

    for (month = 0; month < MONTHS; month++)
    {              // for each month, sum rainfall over years
        for (year = 0, subtot =0; year < YEARS; year++)
            subtot += rain[year][month];
        printf("%4.1f ", subtot/YEARS);
    }
    printf("\n");
    return 0;
}
```

Multidimensional Arrays

The rain.c Program

```
C:₩Dev-Cpp₩rain.exe

 YEAR     RAINFALL  (inches)
 2000             32.4
 2001             37.9
 2002             49.8
 2003             44.0
 2004             32.9

The yearly average is 39.4 inches.

MONTHLY AVERAGES:

 Jan  Feb  Mar  Apr  May  Jun  Jul  Aug  Sep  Oct  Nov  Dec
 7.3  7.3  4.9  3.0  2.3  0.6  1.2  0.3  0.5  1.7  3.6  6.7
계속하려면 아무 키나 누르십시오 . . .
```

Multidimensional Arrays

Initializing a Two-Dimensional Array

- Initializing a two-dimensional array builds on the technique for initializing a one-dimensional array.
- First, recall that initializing a one-dimensional array looks like this:

```
sometype ar1[5] = {val1, val2, val3, val4, val5};
```

Multidimensional Arrays

Initializing a Two-Dimensional Array

- for `rain, val1` would be a value appropriate for initializing a one-dimensional array of `float`, such as the following:

```
{4.3,4.3,4.3,3.0,2.0,1.2,0.2,0.2,0.4,2.4,3.5,6.6}
```

Multidimensional Arrays

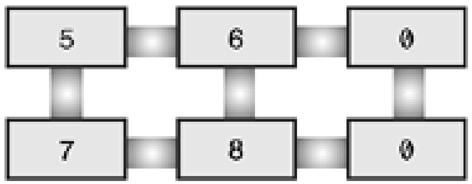
Initializing a Two-Dimensional Array

- That is, if `sometype` is array-of-12-`double`, `val1` is a list of 12 `double` values.
- Therefore, we need a comma-separated list of five of these things to initialize a two-dimensional array, such as `rain`:

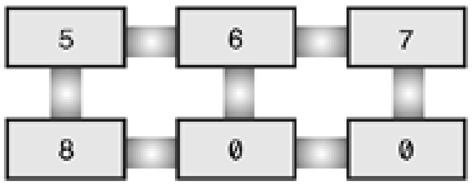
```
const float rain[YEARS][MONTHS] =
{
    {4.3,4.3,4.3,3.0,2.0,1.2,0.2,0.2,0.4,2.4,3.5,6.6},
    {8.5,8.2,1.2,1.6,2.4,0.0,5.2,0.9,0.3,0.9,1.4,7.3},
    {9.1,8.5,6.7,4.3,2.1,0.8,0.2,0.2,1.1,2.3,6.1,8.4},
    {7.2,9.9,8.4,3.3,1.2,0.8,0.4,0.0,0.6,1.7,4.3,6.2},
    {7.6,5.6,3.8,2.8,3.8,0.2,0.0,0.0,0.0,1.3,2.6,5.2}
};
```

Multidimensional Arrays

- Two methods of initializing an array




```
New:
int sq[2][3] = {
                            {5,6},
                            {7,8}
                         };
```




```
int sq[2][3]={5,6,7, 8};
```

Multidimensional Arrays

More Dimensions

- Everything we have said about two-dimensional arrays can be generalized to three-dimensional arrays and further.
- You can declare a three-dimensional array this way:

```
int box[10][20][30];
```

Pointers and Arrays

Pointers and Arrays

- array notation is simply a disguised use of pointers.
- An example of this disguised use is that an array name is also the address of the first element of the array.
- That is, if `flizny` is an array, the following is true:

```
flizny == &flizny[0];
// name of array is the address of the first element
```

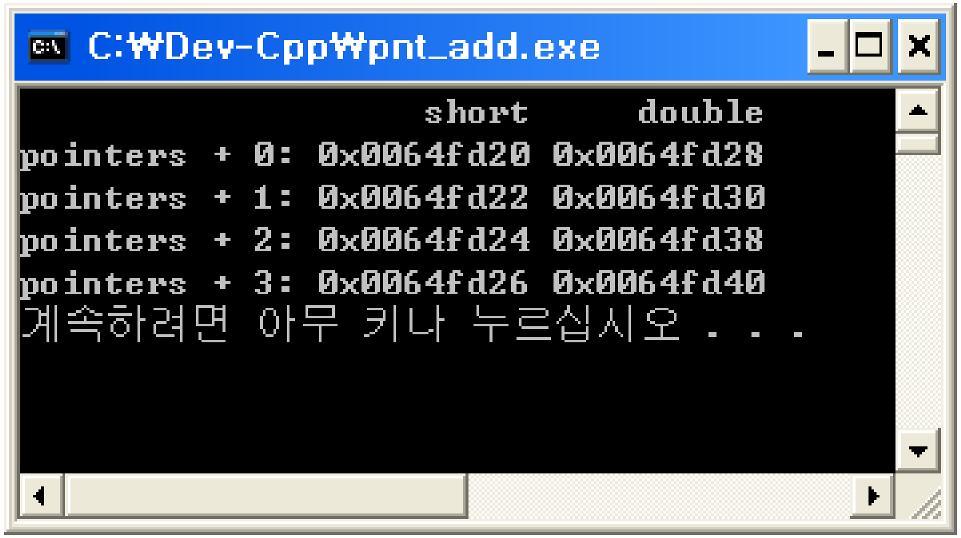
Pointers and Arrays

The pnt_add.c Program

```
#include <stdio.h>
#define SIZE 4

int main(void)
{
    short dates [SIZE];
    short * pti;
    short index;
    double bills[SIZE];
    double * ptf;

    pti = dates;    // assign address of array to pointer
    ptf = bills;
    printf("%23s %10s\n", "short", "double");
    for (index = 0; index < SIZE; index ++)
        printf("pointers + %d: %10p %10p\n",
                index, pti + index, ptf + index);

    return 0;
}
```

Pointers and Arrays

The pnt_add.c Program



Pointers and Arrays

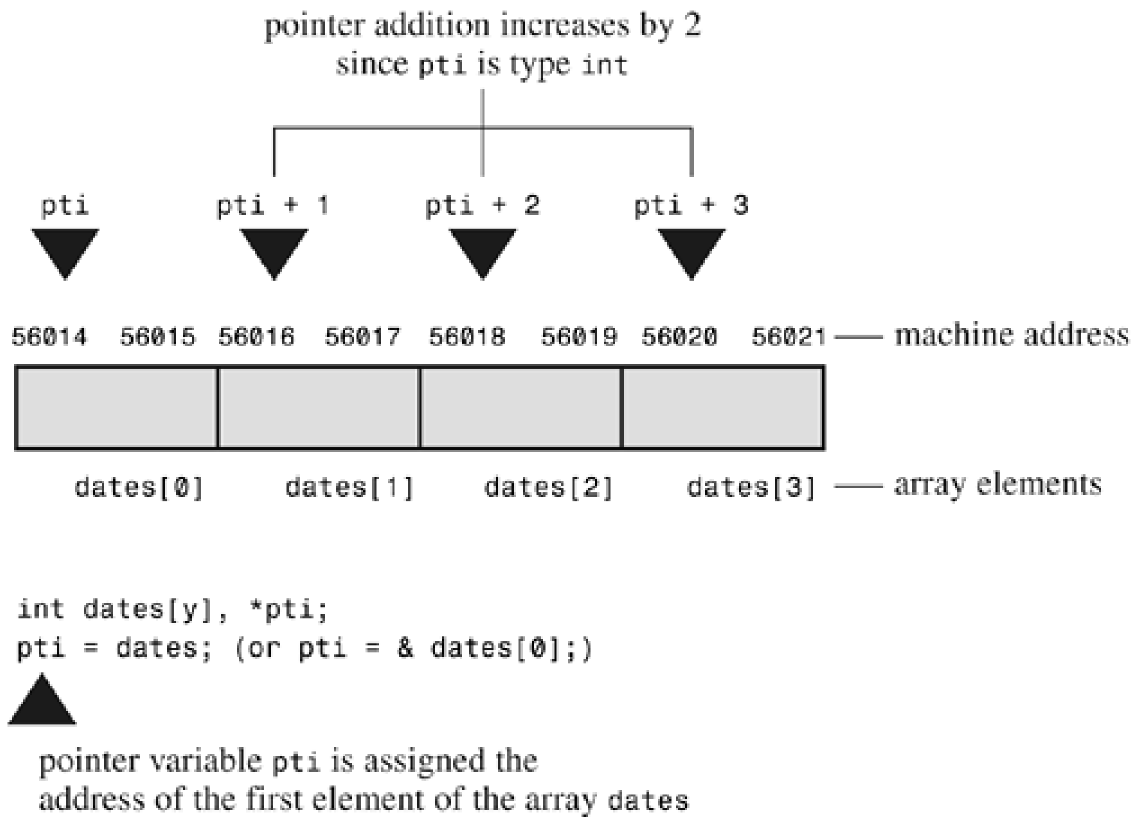
The pnt_add.c Program

- The second line prints the beginning addresses of the two arrays, and the next line gives the result of adding 1 to the address, and so on.
- Keep in mind that the addresses are in hexadecimal, so 30 is 1 more than 2f and 8 more than 28. What?

```
0x0064fd20 + 1 is 0x0064fd22?

0x0064fd30 + 1 is 0x0064fd38?
```

Pointers and Arrays

An array and pointer addition



Pointers and Arrays

An array and pointer addition

- Now we can define more clearly what is meant by pointer-to-`int`, pointer-to-`float`, or pointer-to–any other data object:
- The value of a pointer is the address of the object to which it points.
- Applying the * operator to a pointer yields the value stored in the pointed-to object.
- Adding 1 to the pointer increases its value by the size, in bytes, of the pointed-to type.

Pointers and Arrays

An array and pointer addition

- As a result of C's cleverness, you have the following equalities:

```
dates +2 == &date[2]            /* same address */

*(dates + 2) == dates[2]        /* same value   */
```

Pointers and Arrays

An array and pointer addition

- The indirection operator (`*`) binds more tightly than +, so the latter means `(*dates)+2`:

```
*(dates +2)        /* value of the 3rd element of dates       */

*dates +2          /* 2 added to the value of the 1st element  */
```

Pointers and Arrays

The day_mon3.c Program

```
#include <stdio.h>
#define MONTHS 12

int main(void)
{
    int days[MONTHS] = {31,28,31,30,31,30,31,31,30,31,30,31};
    int index;

    for (index = 0; index < MONTHS; index++)
        printf("Month %2d has %d days.\n", index +1,
               *(days + index));   // same as days[index]

    return 0;
}
```

Pointers and Arrays

The day_mon3.c Program

```
C:\Dev-Cpp\day_mon3.exe
Month  1 has 31 days.
Month  2 has 28 days.
Month  3 has 31 days.
Month  4 has 30 days.
Month  5 has 31 days.
Month  6 has 30 days.
Month  7 has 31 days.
Month  8 has 31 days.
Month  9 has 30 days.
Month 10 has 31 days.
Month 11 has 30 days.
Month 12 has 31 days.
계속하려면 아무 키나 누르십시오 . . .
```

Functions, Arrays, and Pointers

Functions, Arrays, and Pointers

- Suppose you want to write a function that operates on an array.
- Ex)
  - suppose you want a function that returns the sum of the elements of an array.
  - Suppose marbles is the name of an array of `int`. What would the function call look like? A reasonable guess would be this:

```
total = sum(marbles); // possible function call
```

Functions, Arrays, and Pointers

Functions, Arrays, and Pointers

- What would the prototype be?
- Remember, the name of an array is the address of its first element, so the actual argument `marbles`, being the address of an `int`, should be assigned to a formal parameter that is a pointer-to-`int`:

```
int sum(int * ar);   // corresponding prototype
```

Functions, Arrays, and Pointers

Functions, Arrays, and Pointers

- We're left with a couple choices of how to proceed with the function definition.
- The first choice is to code a fixed array size into the function:

```
int sum(int *ar)        // corresponding definition
{
    int i;
    int total = 0;

    for( i = 0; i < 10; i++)    // assume 10 elements
        total += ar[i];     // ar[i] the same as *(ar + i)
return total;
}
```

Functions, Arrays, and Pointers

Functions, Arrays, and Pointers

- A more flexible approach is to pass the array size as a second argument:

```
int sum(int * ar, int n)  // more general approach
{
    int i;
    int total = 0;

    for( i = 0; i < n; i++)    // use n elements
        total += ar[i];        // ar[i] the same as *(ar + i)
return total;
}
```

Functions, Arrays, and Pointers

Functions, Arrays, and Pointers

- There's one more thing to tell about function parameters.
- In the context of a function prototype or function definition header, and only in that context, you can substitute `int ar[]` for `int * ar`:

```
int sum (int ar[], int n);
```

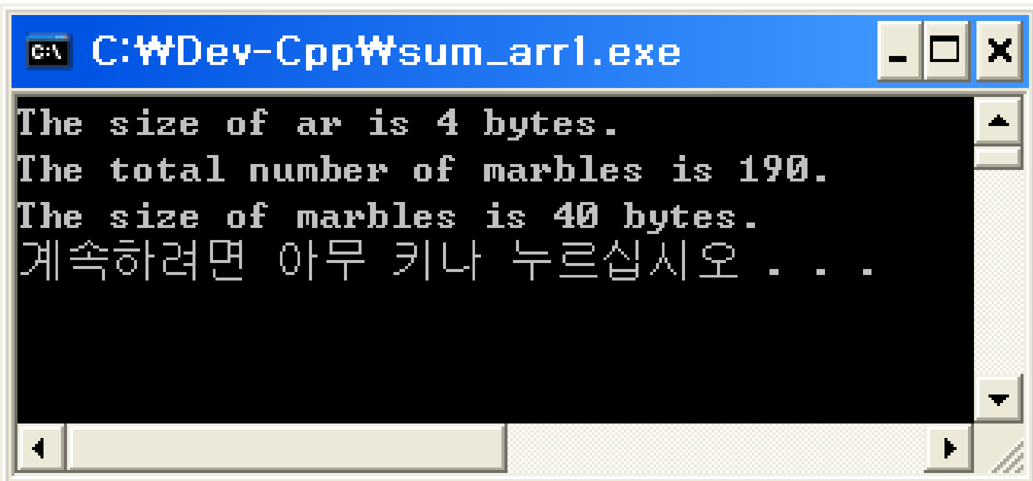
Functions, Arrays, and Pointers

The sum_arr1.c Program

```
#include <stdio.h>
#define SIZE 10

int sum(int ar[], int n);
int main(void)
{
    int marbles[SIZE] = {20,10,5,39,4,16,19,26,31,20};
    long answer;

    answer = sum(marbles, SIZE);
    printf("The total number of marbles is %ld.\n", answer);
    printf("The size of marbles is %zd bytes.\n",
          sizeof marbles);
    return 0;
}

int sum(int ar[], int n)     // how big an array?
{
    int i;
    int total = 0;

    for( i = 0; i < n; i++)
        total += ar[i];
    printf("The size of ar is %zd bytes.\n", sizeof ar);

    return total;
}
```

Functions, Arrays, and Pointers

The sum_arr1.c Program



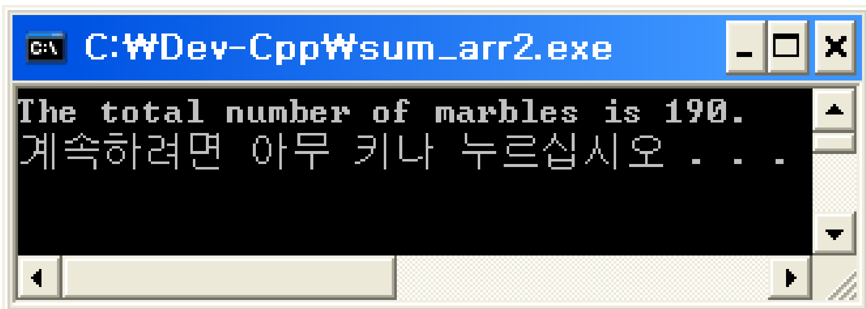
Functions, Arrays, and Pointers

The sum_arr2.c Program

```
#include <stdio.h>
#define SIZE 10

int sump(int * start, int * end);
int main(void)
{
    int marbles[SIZE] = {20,10,5,39,4,16,19,26,31,20};
    long answer;
    answer = sump(marbles, marbles + SIZE);
    printf("The total number of marbles is %ld.\n", answer);

    return 0;
}
/* use pointer arithmetic   */

int sump(int * start, int * end)
{
    int total = 0;

    while (start < end)
    {
        total += *start; /* add value to total              */
        start++;         /* advance pointer to next element */
    }
    return total;
}
```

Functions, Arrays, and Pointers

The sum_arr2.c Program



Functions, Arrays, and Pointers

The sum_arr2.c Program

- Note that the `sump()` function uses a different method from `sum()` to end the summation loop. The `sum()` function uses the number of elements as a second argument, and the loop uses that value as part of the loop test:

```
for( i = 0; i < n; i++)
```

Functions, Arrays, and Pointers

The sum_arr2.c Program

- The `sump()` function, however, uses a second pointer to end the loop:

```
while (start < end)
```

- Note that using this "past-the-end" pointer makes the function call neat:

```
answer = sump(marbles, marbles + SIZE);
```

Functions, Arrays, and Pointers

The sum_arr2.c Program

- Because indexing starts at `0`, `marbles + SIZE` points to the next element after the end.
- If `end` pointed to the last element instead of to one past the end, you would have to use the following code instead:

```
answer = sump(marbles, marbles + SIZE - 1);
```

Functions, Arrays, and Pointers

The sum_arr2.c Program

- You can also condense the body of the loop to one line:

```
total += *start++;
```

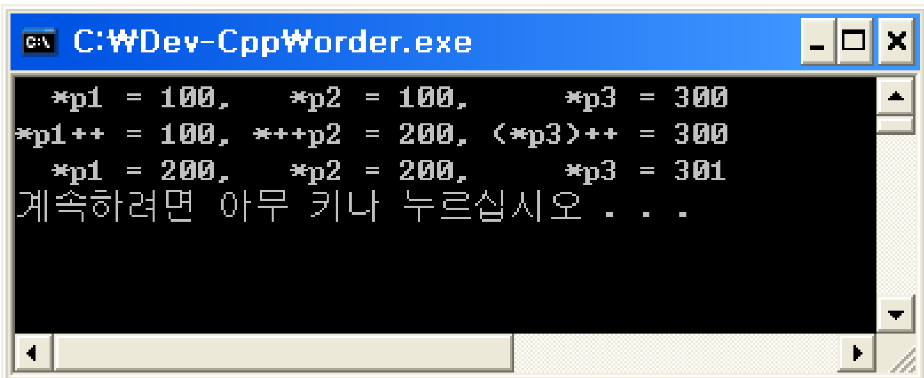
Functions, Arrays, and Pointers

The order.c Program

```
#include <stdio.h>

int data[2] = {100, 200};
int moredata[2] = {300, 400};

int main(void)
{
    int * p1, * p2, * p3;

    p1 = p2 = data;
    p3 = moredata;
    printf("  *p1 = %d,   *p2 = %d,     *p3 = %d\n",
              *p1     ,   *p2     ,     *p3);
    printf("*p1++ = %d, *++p2 = %d, (*p3)++ = %d\n",
            *p1++     , *++p2     , (*p3)++);
    printf("  *p1 = %d,   *p2 = %d,     *p3 = %d\n",
              *p1     ,   *p2     ,     *p3);

    return 0;
}
```

Functions, Arrays, and Pointers

The order.c Program



Functions, Arrays, and Pointers

Comment: Pointers and Arrays

- As you have seen, functions that process arrays actually use pointers as arguments,
- but you do have a choice between array notation and pointer notation for writing array-processing functions.

- As far as C goes, the two expressions `ar[i]` and `*(ar+i)` are equivalent in meaning.
- However, using an expression such as `ar++` only works if `ar` is a pointer variable.

Pointer Operations

Pointer Operations

- Just what can you do with pointers?
- C offers several basic operations you can perform on pointers, and the next program demonstrates eight of these possibilities.
- To show the results of each operation, the program prints the value of the pointer, the value stored in the pointed-to address, and the address of the pointer itself.

Pointer Operations

The ptr_ops.c Program(1/2)

```
#include <stdio.h>

int main(void)
{
    int urn[5] = {100,200,300,400,500};
    int * ptr1, * ptr2, *ptr3;

    ptr1 = urn;          // assign an address to a pointer
    ptr2 = &urn[2];      // ditto

    printf("pointer value, dereferenced pointer, pointer address:\n");
    printf("ptr1 = %p, *ptr1 =%d, &ptr1 = %p\n",
           ptr1, *ptr1, &ptr1);

    // pointer addition
    ptr3 = ptr1 + 4;
    printf("\nadding an int to a pointer:\n");
    printf("ptr1 + 4 = %p, *(ptr4 + 3) = %d\n",
            ptr1 + 4, *(ptr1 + 3));
    ptr1++;             // increment a pointer

    printf("\nvalues after ptr1++:\n");
    printf("ptr1 = %p, *ptr1 =%d, &ptr1 = %p\n",
           ptr1, *ptr1, &ptr1);
    ptr2--;             // decrement a pointer
```

Pointer Operations

The ptr_ops.c Program(2/2)

```
    printf("\nvalues after --ptr2:\n");
    printf("ptr2 = %p, *ptr2 = %d, &ptr2 = %p\n",
           ptr2, *ptr2, &ptr2);

    --ptr1;                  // restore to original value
    ++ptr2;                  // restore to original value

    printf("\nPointers reset to original values:\n");
    printf("ptr1 = %p, ptr2 = %p\n", ptr1, ptr2);
                              // subtract one pointer from another
    printf("\nsubtracting one pointer from another:\n");
    printf("ptr2 = %p, ptr1 = %p, ptr2 - ptr1 = %d\n",
            ptr2, ptr1, ptr2 - ptr1);
                             // subtract an integer from a pointer
    printf("\nsubtracting an int from a pointer:\n");
    printf("ptr3 = %p, ptr3 - 2 = %p\n",
            ptr3,  ptr3 - 2);

    return 0;
}
```

Pointer Operations

The ptr_ops.c Program

C:₩Dev-Cpp₩ptr_ops.exe

```
pointer value, dereferenced pointer, pointer address:
ptr1 = 0022FF50, *ptr1 =100, &ptr1 = 0022FF4C

adding an int to a pointer:
ptr1 + 4 = 0022FF60, *(ptr4 + 3) = 400

values after ptr1++:
ptr1 = 0022FF54, *ptr1 =200, &ptr1 = 0022FF4C

values after --ptr2:
ptr2 = 0022FF54, *ptr2 = 200, &ptr2 = 0022FF48

Pointers reset to original values:
ptr1 = 0022FF50, ptr2 = 0022FF58

subtracting one pointer from another:
ptr2 = 0022FF58, ptr1 = 0022FF50, ptr2 - ptr1 = 2

subtracting an int from a pointer:
ptr3 = 0022FF60, ptr3 - 2 = 0022FF58
계속하려면 아무 키나 누르십시오 . . .
```

Pointer Operations

The ptr_ops.c Program

- The following list describes the basic operations that can be performed with or on pointer variables:
- **Assignment**
  - You can assign an address to a pointer.
  - Typically, you do this by using an array name or by using the address operator (&).
- **Value finding (dereferencing)**
  - The * operator gives the value stored in the pointed-to location.

Pointer Operations

The ptr_ops.c Program

- The following list describes the basic operations that can be performed with or on pointer variables:
- **Taking a pointer address**
  - Like all variables, pointer variables have an address and a value.
  - The `&` operator tells you where the pointer itself is stored.
- **Adding an integer to a pointer**
  - You can use the + operator to add an integer to a pointer or a pointer to an integer.

Pointer Operations

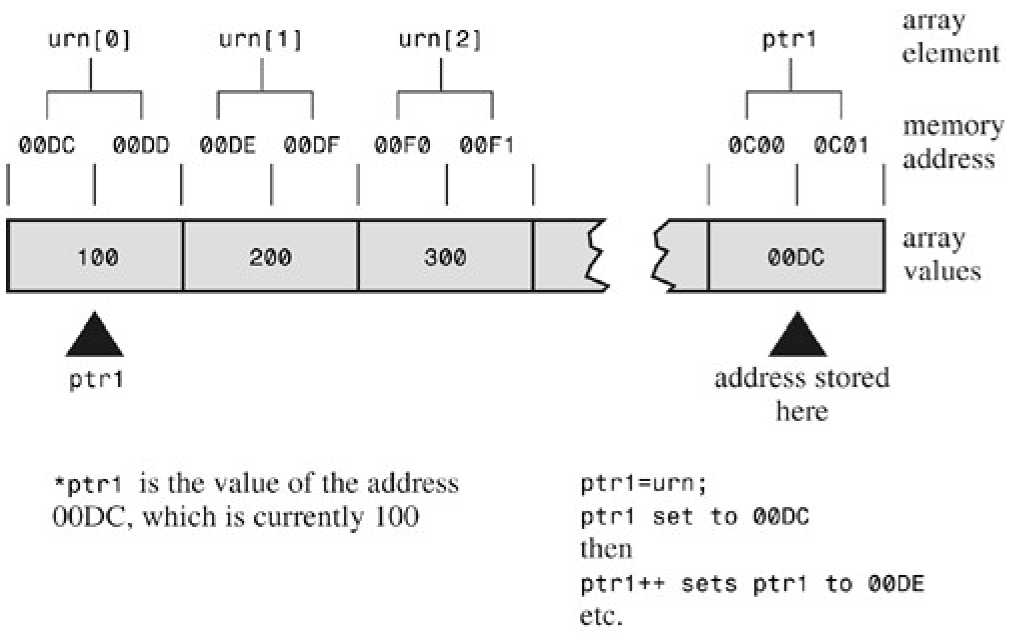
The ptr_ops.c Program

- The following list describes the basic operations that can be performed with or on pointer variables:
- **Incrementing a pointer**
  - Incrementing a pointer to an array element makes it move to the next element of the array.

Pointer Operations

The ptr_ops.c Program

- The following list describes the basic operations that can be performed with or on pointer variables:
- **Incrementing a type int pointer**



Pointer Operations

The ptr_ops.c Program

- The following list describes the basic operations that can be performed with or on pointer variables:
- **Subtracting an integer from a pointer**
  - You can use the − operator to subtract an integer from a pointer; the pointer has to be the first operand or a pointer to an integer.
- **Decrementing a pointer**
  - Of course, you can also decrement a pointer.

Pointer Operations

The ptr_ops.c Program

- The following list describes the basic operations that can be performed with or on pointer variables:
- **Differencing**
  - You can find the difference between two pointers.
  - Normally, you do this for two pointers to elements that are in the same array to find out how far apart the elements are.
- **Comparisons**
  - You can use the relational operators to compare the values of two pointers, provided the pointers are of the same type.

Pointer Operations

The ptr_ops.c Program

- There are some cautions to remember when incrementing or decrementing a pointer.
- Given

```
int urn[3];
int * ptr1, * ptr2;
```

- the following are some valid and invalid statements:

Valid	Invalid
`ptr1++;`	`urn++;`
`ptr2 = ptr1 + 2;`	`ptr2 = ptr2 + ptr1;`
`ptr2 = urn + 1;`	`ptr2 = urn * ptr1;`

Protecting Array Contents

Protecting Array Contents

- The usual rule is to pass quantities by value unless the program needs to alter the value, in which case you pass a pointer.
- Arrays don't give you that choice; you must pass a pointer.
  - The reason is efficiency.

Protecting Array Contents

Protecting Array Contents

- For example, here's a function that adds the same value to each member of an array:

```
void add_to(double ar[], int n, double val)
{
    int i;
    for( i = 0; i < n; i++)
        ar[i] += val;
}
```

- Therefore, the function call

```
add_to(prices, 100, 2.50);
```

Protecting Array Contents

Protecting Array Contents

- Here, for example, the expression `ar[i]++` results in each element having 1 added to its value:

```
int sum(int ar[], int n)  // faulty code
{
    int i;
    int total = 0;

    for( i = 0; i < n; i++)
        total += ar[i]++;   // error increments each element
    return total;
}
```

Protecting Array Contents

Using const with Formal Parameters

- If a function's intent is that it not change the contents of the array,
- use the keyword `const` when declaring the formal parameter in the prototype and in the function definition.
- Ex) the prototype and definition for `sum()` should look like this:

```
int sum(const int ar[], int n);   /* prototype  */

int sum(const int ar[], int n)    /* definition */
{
    int i;
    int total = 0;

    for( i = 0; i < n; i++)
        total += ar[i];
    return total;
}
```

Protecting Array Contents

The arf.c Program(1/2)

```
#include <stdio.h>
#define SIZE 5

void show_array(const double ar[], int n);
void mult_array(double ar[], int n, double mult);
int main(void)
{
    double dip[SIZE] = {20.0, 17.66, 8.2, 15.3, 22.22};

    printf("The original dip array:\n");
    show_array(dip, SIZE);
    mult_array(dip, SIZE, 2.5);
    printf("The dip array after calling mult_array():\n");
    show_array(dip, SIZE);

    return 0;
}
```

Protecting Array Contents

The arf.c Program(2/2)

```
/* displays array contents */
void show_array(const double ar[], int n)
{
    int i;

    for (i = 0; i < n; i++)
        printf("%8.3f ", ar[i]);
    putchar('\n');
}

/* multiplies each array member by the same multiplier */
void mult_array(double ar[], int n, double mult)
{
    int i;

    for (i = 0; i < n; i++)
        ar[i] *= mult;
}
```

Protecting Array Contents

The arf.c Program

```
C:₩Dev-Cpp₩arf.exe
The original dip array:
  20.000   17.660    8.200   15.300   22.220
The dip array after calling mult_array():
  50.000   44.150   20.500   38.250   55.550
계속하려면 아무 키나 누르십시오 . . .
```

Protecting Array Contents

More About const

- Earlier, you saw that you can use `const` to create symbolic constants:

```
const double PI = 3.14159;
```

- Listing 10.4 showed how to use the const keyword to protect an array:

```
#define MONTHS 12
...
const int days[MONTHS] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

Protecting Array Contents

More About const

- If the program code subsequently tries to alter the array, you'll get a compile-time error message:

```
days[9] = 44;      /* compile error */
```

- Pointers to constants can't be used to change values.
- Consider the following code:

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
const double * pd = rates;
// pd points to beginning of the array
```

Protecting Array Contents

More About const

- The second line of code declares that the type `double` value to which pd points is a `const`.
- That means you can't use `pd` to change pointed-to values:

```
*pd = 29.89;         // not allowed
pd[2] = 222.22;      // not allowed
rates[0] = 99.99;  // allowed because rates is not const
```

Protecting Array Contents

More About const

- Whether you use pointer notation or array notation, you are not allowed to use `pd` to change the value of pointed-to data.
- Note, however, that because `rates` was not declared as a constant, you can still use `rates` to change values.
- Also, note that you can make `pd` point somewhere else:

```
pd++;   /* make pd point to rates[1] -- allowed */
```

Protecting Array Contents

More About const

- A pointer-to-constant is normally used as a function parameter to indicate that the function won't use the pointer to change data.
- For example, the `show_array()` function from Listing 10.14 could have been prototyped as

```
void show_array(const double *ar, int n);
```

Protecting Array Contents

More About const

- There are some rules you should know about pointer assignments and `const`.
- First, it's valid to assign the address of either constant data or nonconstant data to a pointer-to-constant:

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
const double locked[4] = {0.08, 0.075, 0.0725, 0.07};
const double * pc = rates;     // valid
pc = locked;                   // valid
pc = &rates[3];                // valid
```

Protecting Array Contents

More About const

- However, only the addresses of nonconstant data can be assigned to regular pointers:

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
const double locked[4] = {0.08, 0.075, 0.0725, 0.07};
double * pnc = rates;                // valid
pnc = locked;                        // not valid
pnc = &rates[3];                     // valid
```

Protecting Array Contents

More About const

- A practical consequence of these rules is that a function such as `show_array()` can accept the names of regular arrays and of constant arrays as actual arguments,
- because either can be assigned to a pointer-to-constant:

```
show_array(rates, 5);     // valid
show_array(locked, 4);    // valid
```

Protecting Array Contents

More About const

- A function such as `mult_array()`, however, can't accept the name of a constant array as an argument:

```
mult_array(rates, 5, 1.2);     // valid
mult_array(locked, 4, 1.2);    // not allowed
```

Protecting Array Contents

More About const

- There are more possible uses of `const`.
- For example, you can declare and initialize a pointer so that it can't be made to point elsewhere.

- The trick is the placement of the keyword `const`:

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
double * const pc = rates;    // pc points to beginning of the array
pc = &rates[2];               // not allowed
*pc = 92.99;                  // ok -- changes rates[0]
```

Protecting Array Contents

More About const

- Finally, you can use const twice to create a pointer that can neither change where it's pointing nor change the value to which it points:

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
const double * const pc = rates;
pc = &rates[2];                      // not allowed
*pc = 92.99;                         // not allowed
```

Pointers and Multidimensional Arrays

Pointers and Multidimensional Arrays

- How do pointers relate to multidimensional arrays?
- And why would you want to know?
- Functions that work with multidimensional arrays do so with pointers,
  - so you need some further pointer background before working with such functions.

Pointers and Multidimensional Arrays

Pointers and Multidimensional Arrays

- Suppose you have this declaration:

```
int zippo[4][2];   /* an array of arrays of ints */
```

- Then zippo, being the name of an array,
  - is the address of the first element of the array.

Pointers and Multidimensional Arrays

Pointers and Multidimensional Arrays

- Let's analyze that further in terms of pointer properties:
- Because `zippo` is the address of the array's first element, `zippo` has the same value as `&zippo[0]`.
- Adding 1 to a pointer or address yields a value larger by the size of the referred-to object.
- Dereferencing a pointer or an address yields the value represented by the referred-to object.

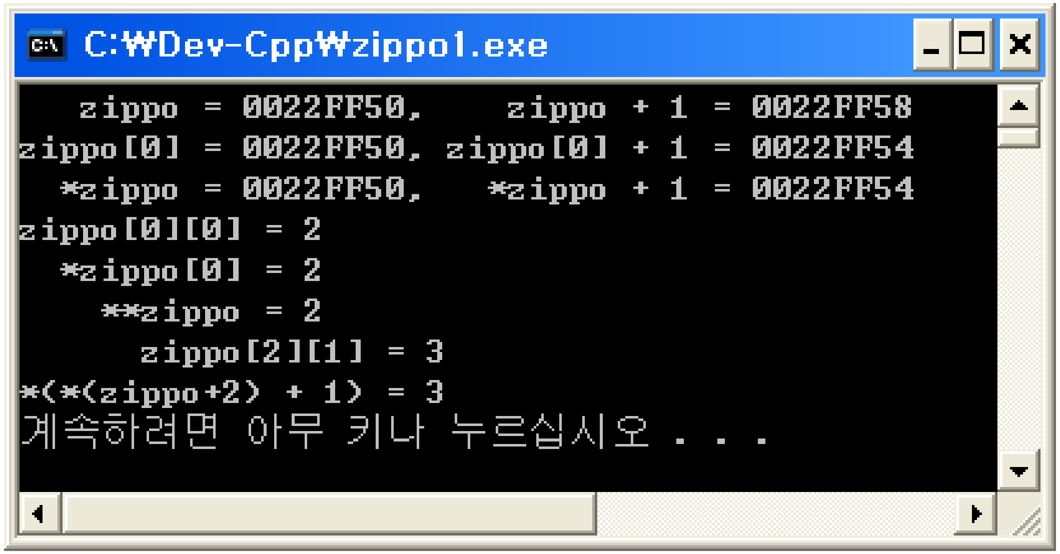
Pointers and Multidimensional Arrays

The zippo1.c Program

```
#include <stdio.h>

int main(void)
{
    int zippo[4][2] = { {2,4}, {6,8}, {1,3}, {5, 7} };

    printf("   zippo = %p,    zippo + 1 = %p\n",
               zippo,         zippo + 1);
    printf("zippo[0] = %p, zippo[0] + 1 = %p\n",
            zippo[0],      zippo[0] + 1);
    printf("  *zippo = %p,   *zippo + 1 = %p\n",
              *zippo,        *zippo + 1);
    printf("zippo[0][0] = %d\n", zippo[0][0]);
    printf("  *zippo[0] = %d\n", *zippo[0]);
    printf("    **zippo = %d\n", **zippo);
    printf("      zippo[2][1] = %d\n", zippo[2][1]);
    printf("*(*(zippo+2) + 1) = %d\n", *(*(zippo+2) + 1));

    return 0;
}
```

Pointers and Multidimensional Arrays

The zippo1.c Program



Pointers and Multidimensional Arrays

The zippo1.c Program

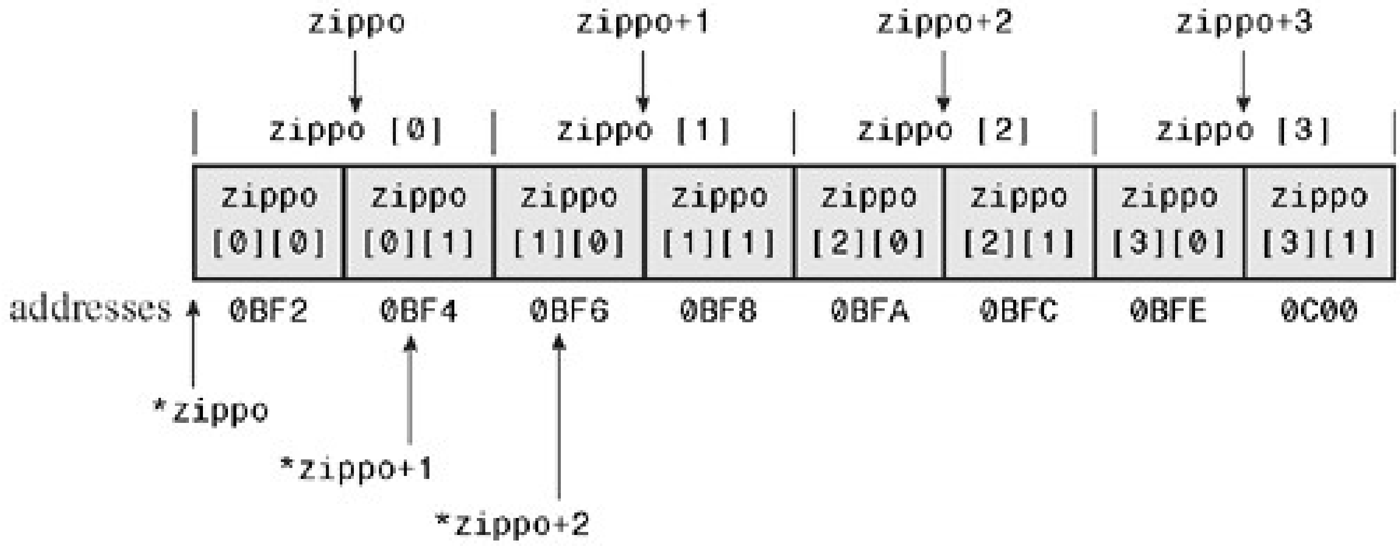
- You probably should make the effort at least once in your life to break this down.
- Let's build up the expression in steps:

`zippo`	the address of the first two-`int` element
`zippo+2`	the address of the third two-`int` element
`*(zippo+2)`	the third element, a two-`int` array, hence the address of its first element, an `int`
`*(zippo+2)+1`	the address of the second element of the two-`int` array, also an `int`
`*(*(zippo+2)+1)`	the value of the second `int` in the third row (`zippo[2][1]`)

Pointers and Multidimensional Arrays

The zippo1.c Program

- **An array of arrays**



Pointers and Multidimensional Arrays

Pointers to Multidimensional Arrays

- `pz` must point to an array of two `int`s, not to a single `int`.

```
int (* pz)[2];  // pz points to an array of 2 ints
```

- This statement says that `pz` is a pointer to an array of two ints.
- Why the parentheses? Well, `[ ]` has a higher precedence than `*`.
  - Therefore, with a declaration such as

```
int * pax[2];
```

Pointers and Multidimensional Arrays

The zippo2.c Program

```
#include <stdio.h>

int main(void)
{
    int zippo[4][2] = { {2,4}, {6,8}, {1,3}, {5, 7} };
    int (*pz)[2];
    pz = zippo;

    printf("   pz = %p,    pz + 1 = %p\n",
              pz,          pz + 1);
    printf("pz[0] = %p, pz[0] + 1 = %p\n",
             pz[0],     pz[0] + 1);
    printf("  *pz = %p,   *pz + 1 = %p\n",
            *pz,         *pz + 1);
    printf("pz[0][0] = %d\n", pz[0][0]);
    printf("  *pz[0] = %d\n", *pz[0]);
    printf("    **pz = %d\n", **pz);
    printf("      pz[2][1] = %d\n", pz[2][1]);
    printf("*(*(pz+2) + 1) = %d\n", *(*(pz+2) + 1));

    return 0;
}
```

Pointers and Multidimensional Arrays

The zippo2.c Program

C:₩Dev-Cpp₩zippo2.exe

```
   pz = 0022FF50,      pz + 1 = 0022FF58
pz[0] = 0022FF50, pz[0] + 1 = 0022FF54
  *pz = 0022FF50,     *pz + 1 = 0022FF54
pz[0][0] = 2
  *pz[0] = 2
    **pz = 2
      pz[2][1] = 3
*(*(pz+2) + 1) = 3
계속하려면 아무 키나 누르십시오 . . .
```

Pointers and Multidimensional Arrays

The zippo2.c Program

- you can represent individual elements by using array notation or pointer notation with either an array name or a pointer:

```
zippo[m][n] == *(*(zippo + m) + n)

pz[m][n] == *(*(pz + m) + n)
```

Pointers and Multidimensional Arrays

Pointer Compatibility

- The rules for assigning one pointer to another are tighter than the rules for numeric types.
- Ex) you can assign an int value to a double variable without using a type conversion, but you can't do the same for pointers to these two types:

```
int n = 5;
double x;
int * p1 = &n;
double * pd    = &x;

x = n;                  // implicit type conversion
pd = p1;                // compile-time error
```

Pointers and Multidimensional Arrays

Pointer Compatibility

- These restrictions extend to more complex types.
- Suppose we have the following declarations:

```
int * pt;
int (*pa)[3];
int ar1[2][3];
int ar2[3][2];
int **p2;    // a pointer to a pointer
```

Pointers and Multidimensional Arrays

Pointer Compatibility

- Then we have the following:

```
pt = &ar1[0][0];          // both pointer-to-int
pt = ar1[0];              // both pointer-to-int
pt = ar1;                 // not valid
pa = ar1;                 // both pointer-to-int[3]
pa = ar2;                 // not valid
p2 = &pt;                 // both pointer-to-int *
*p2 = ar2[0];             // both pointer-to-int
p2 = ar2;                 // not valid
```

Pointers and Multidimensional Arrays

Pointer Compatibility

- In general, multiple indirection is tricky. For instance, consider the next snippet of code:

```
int * p1;
const int * p2;
const int ** pp2;

p1 = p2;    // not valid -- assigning const to non-const
p2 = p1;    // valid     -- assigning non-const to const
pp2 = &p1; // not valid -- assigning non-const to const
```

Pointers and Multidimensional Arrays

Pointer Compatibility

- As you saw earlier, assigning a `const` pointer to a non-const pointer is invalid, because you could use the new pointer to alter `const` data.
- But assigning a non-`const` pointer to a `const` pointer is okay, provided that you're dealing with just one level of indirection:

```
p2 = p1; // valid -- assigning non-const to const
```

Pointers and Multidimensional Arrays

Pointer Compatibility

- But such assignments no longer are safe when you go to two levels of indirection.
- If it were allowed, you could do something like this:

```
const int **pp2;
int *p1;
const int n = 13;

pp2 = &p1; // not allowed, but suppose it were
*pp2 = &n; // valid, both const, but sets p1 to point at n
*p1 = 10;  // valid, but changes const n
```

Pointers and Multidimensional Arrays

Functions and Multidimensional Arrays

- Let's write a function to deal with two-dimensional arrays.
- One possibility is to use a `for` loop to apply a one-dimensional array function to each row of the two-dimensional array.
- That is, you could do something like the following:

```
int junk[3][4] = { {2,4,5,8}, {3,5,6,9}, {12,10,8,6} };
int i, j;
int total = 0;

for (i = 0; i < 3 ; i++)
   total += sum(junk[i], 4);
   // junk[i]--one-dimensional array
```

Pointers and Multidimensional Arrays

Functions and Multidimensional Arrays

- You can declare a function parameter of this type like this:

```
void somefunction( int (* pt)[4] );
```

- Alternatively, if (and only if) `pt` is a formal parameter to a function, you can declare it as follows:

```
void somefunction( int pt[][4] );
```

Pointers and Multidimensional Arrays

The array2d.c Program(1/3)

```
#include <stdio.h>
#define ROWS 3
#define COLS 4

void sum_rows(int ar[][COLS], int rows);
void sum_cols(int [][COLS], int );     // ok to omit names
int sum2d(int (*ar)[COLS], int rows); // another syntax
int main(void)
{
     int junk[ROWS][COLS] = {
             {2,4,6,8},
             {3,5,7,9},
             {12,10,8,6}
     };

     sum_rows(junk, ROWS);
     sum_cols(junk, ROWS);
     printf("Sum of all elements = %d\n", sum2d(junk, ROWS));

     return 0;
}
```

Pointers and Multidimensional Arrays

The array2d.c Program(2/3)

```
void sum_rows(int ar[][COLS], int rows)
{
    int r;
    int c;
    int tot;

    for (r = 0; r < rows; r++)
    {
        tot = 0;
        for (c = 0; c < COLS; c++)
            tot += ar[r][c];
        printf("row %d: sum = %d\n", r, tot);
    }
}
void sum_cols(int ar[][COLS], int rows)
{
    int r;
    int c;
    int tot;

    for (c = 0; c < COLS; c++)
    {
        tot = 0;
        for (r = 0; r < rows; r++)
            tot += ar[r][c];
        printf("col %d: sum = %d\n", c, tot);
    }
}
```

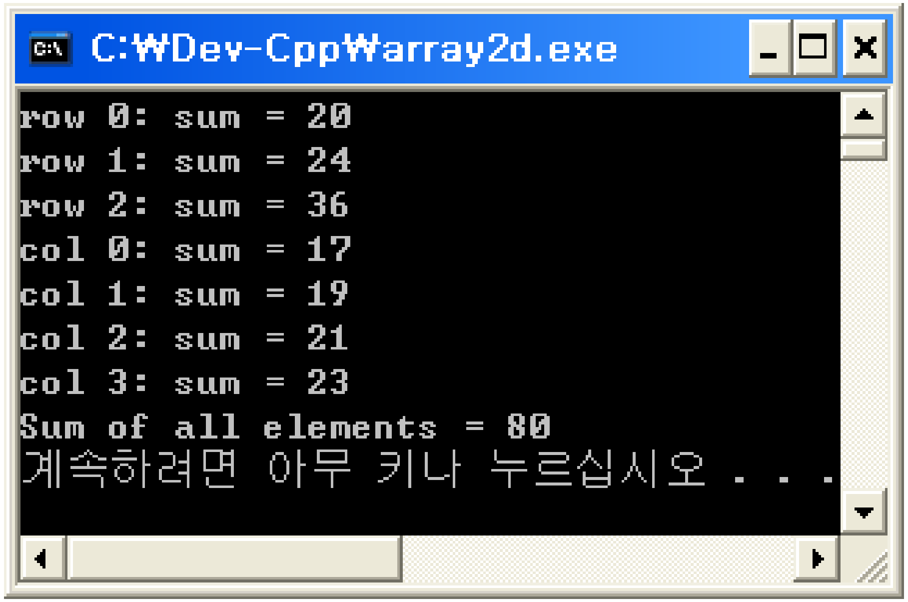
Pointers and Multidimensional Arrays

The array2d.c Program(3/3)

```
int sum2d(int ar[][COLS], int rows)
{
    int r;
    int c;
    int tot = 0;

    for (r = 0; r < rows; r++)
        for (c = 0; c < COLS; c++)
            tot += ar[r][c];

    return tot;
}
```

Pointers and Multidimensional Arrays

The array2d.c Program



Pointers and Multidimensional Arrays

The array2d.c Program

- Be aware that the following declaration will not work properly:

```
int sum2(int ar[][], int rows); // faulty declaration
```

- The declaration

```
int sum2(int ar[][4], int rows); // valid declaration
```

- says that `ar` points to an array of four `int`s,
  - so `ar+1` means "add 16 bytes to the address."

Pointers and Multidimensional Arrays

The array2d.c Program

- You can also include a size in the other bracket pair, as shown here, but the compiler ignores it:

```
int sum2(int ar[3][4], int rows);
// valid declaration, 3 ignored
```

- This is convenient for those who use `typedefs`:

```
typedef int arr4[4];              // arr4 array of 4 int
typedef arr4 arr3x4[3];           // arr3x4 array of 3 arr4

int sum2(arr3x4 ar, int rows);    // same as next declaration
int sum2(int ar[3][4], int rows); // same as next declaration
int sum2(int ar[][4], int rows);  // standard form
```

Pointers and Multidimensional Arrays

The array2d.c Program

- In general, to declare a pointer corresponding to an *N*-dimensional array, you must supply values for all but the leftmost set of brackets:

```
int sum4d(int ar[][12][20][30], int rows);
```

- That's because the first set of brackets indicates a pointer,
- whereas the rest of the brackets describe the type of data object being pointed to, as the following equivalent prototype illustrates:

```
int sum4d(int (*ar)[12][20][30], int rows);
```

Variable-Length Arrays (VLAs)

Variable-Length Arrays (VLAs)

- You can describe the number of rows with a function parameter, but the number of columns is built in to the function.
- For example, look at this definition:

```
#define COLS 4

int sum2d(int ar[][COLS], int rows)
{
    int r;
    int c;
    int tot = 0;

    for (r = 0; r < rows; r++)
        for (c = 0; c < COLS; c++)
            tot += ar[r][c];
    return tot;
}
```

Variable-Length Arrays (VLAs)

Variable-Length Arrays (VLAs)

- Next, suppose the following arrays have been declared:

```
int array1[5][4];
int array2[100][4];
int array3[2][4];
```

- You can use the sum2d() function with any of these arrays:

```
tot = sum2d(array1, 5);    // sum a 5 x 4 array
tot = sum2d(array2, 100); // sum a 100 x 4 array
tot = sum2d(array3, 2);    // sum a 2 x 4 array
```

Variable-Length Arrays (VLAs)

Variable-Length Arrays (VLAs)

- C is being positioned to take over from FORTRAN,
- so the ability to convert FORTRAN libraries with a minimum of fuss is useful.
- This need was the primary impulse for C99 introducing variable-length arrays, which allow you to use variables when dimensioning an array.
  - For example, you can do this:

```
int quarters = 4;
int regions = 5;
double sales[regions][quarters]; // a VLA
```

Variable-Length Arrays (VLAs)

Variable-Length Arrays (VLAs)

- Let's look at a simple example that shows how to write a function that will sum the contents of any two-dimensional array of `int`s.
- First, here's how to declare a function with a two-dimensional VLA argument:

```
int sum2d(int rows, int cols, int ar[rows][cols]);   // ar a VLA
```

Variable-Length Arrays (VLAs)

Variable-Length Arrays (VLAs)

- Note that the first two parameters are used as dimensions for declaring the array parameter `ar`.
- Because the `ar` declaration uses `rows` and `cols`, they have to be declared before `ar` in the parameter list.
  - Therefore, the following prototype is in error:

```
int sum2d(int ar[rows][cols], int rows, int cols);
// invalid order
```

Variable-Length Arrays (VLAs)

Variable-Length Arrays (VLAs)

- The C99 standard says you can omit names from the prototype.
- but in that case, you need to replace the omitted dimensions with asterisks:

```
int sum2d(int, int, int ar[*][*]);
```

- Second, here's how to define the function:

```
int sum2d(int rows, int cols, int ar[rows][cols])
{
    int r;
    int c;
    int tot = 0;

    for (r = 0; r < rows; r++)
        for (c = 0; c < cols; c++)
            tot += ar[r][c];
    return tot;
}
```

Variable-Length Arrays (VLAs)

The vararr2d.c Program(1/2)

```
#include <stdio.h>
#define ROWS 3
#define COLS 4

int sum2d(int rows, int cols, int ar[rows][cols]);
int main(void)
{
     int i, j;
     int rs = 3;
     int cs = 10;
     int junk[ROWS][COLS] = {
            {2,4,6,8},
            {3,5,7,9},
            {12,10,8,6}
     };
     int morejunk[ROWS-1][COLS+2] = {
            {20,30,40,50,60,70},
            {5,6,7,8,9,10}
     };
     int varr[rs][cs];  // VLA

     for (i = 0; i < rs; i++)
         for (j = 0; j < cs; j++)
             varr[i][j] = i * j + j;
```

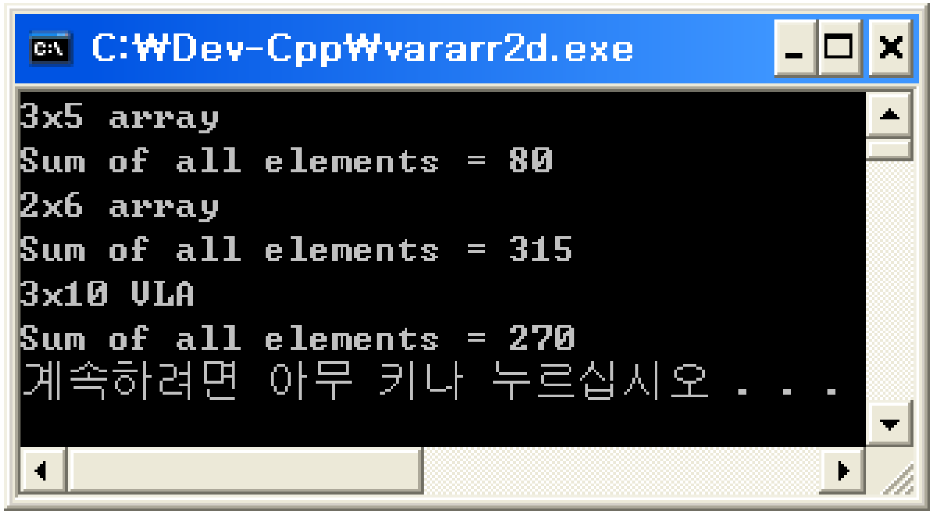
Variable-Length Arrays (VLAs)

The vararr2d.c Program(2/2)

```
     for (i = 0; i < rs; i++)
         for (j = 0; j < cs; j++)
             varr[i][j] = i * j + j;

     printf("3x5 array\n");
     printf("Sum of all elements = %d\n",
             sum2d(ROWS, COLS, junk));
     printf("2x6 array\n");
     printf("Sum of all elements = %d\n",
             sum2d(ROWS-1, COLS+2, morejunk));
     printf("3x10 VLA\n");
     printf("Sum of all elements = %d\n",
             sum2d(rs, cs, varr));
     return 0;
}
// function with a VLA parameter
int sum2d(int rows, int cols, int ar[rows][cols])
{
    int r;
    int c;
    int tot = 0;

    for (r = 0; r < rows; r++)
        for (c = 0; c < cols; c++)
            tot += ar[r][c];
    return tot;
}
```

Variable-Length Arrays (VLAs)

The vararr2d.c Program



Variable-Length Arrays (VLAs)

The vararr2d.c Program

- The following snippet points out when a pointer is declared and when an actual array is declared:

```
    int thing[10][6];
    twoset(10,6,thing);
    ...
}
void twoset (int n, int m, int ar[n][m])   // ar a pointer to
                                           // an array of m ints
{
    int temp[n][m];       // temp an n x m array of int
    temp[0][0] = 2;       // set an element of temp to 2
    ar[0][0] = 2;         // set thing[0][0] to 2
}
```

Compound Literals

Compound Literals

- For arrays, a compound literal looks like an array initialization list preceded by a type name that is enclosed in parentheses.
- For example, here's an ordinary array declaration:

```
int diva[2] = {10, 20};
```

- And here's a compound literal that creates a nameless array containing the same two `int` values:

```
(int [2]){10, 20}      // a compound literal
```

Compound Literals

Compound Literals

- Just as you can leave out the array size if you initialize a named array,
- you can omit it from a compound literal, and the compiler will count how many elements are present:

```
(int [])(50, 20, 90} // a compound literal with 3 elements
```

- One way is to use a pointer to keep track of the location.
  - That is, you can do something like this:

```
int * pt1;
pt1 = (int [2]) {10, 20};
```

Compound Literals

Compound Literals

- Another thing you could do with a compound literal is pass it as an actual argument to a function with a matching formal parameter:

```
int sum(int ar[], int n);
...
int total3;
total3 = sum((int []){4,4,4,5,5,5}, 6);
```

- You can extend the technique to two-dimensional arrays, and beyond.
- Here, for example, is how to create a two-dimensional array of `ints` and store the address:

```
int (*pt2)[4];
pt2 = (int [2][4]) { {1,2,3,-9}, {4,5,6,-8} };
```

Compound Literals

The flc.c Program(1/2)

```
#include <stdio.h>
#define COLS 4

int sum2d(int ar[][COLS], int rows);
int sum(int ar[], int n);
int main(void)
{
      int total1, total2, total3;
      int * pt1;
      int (*pt2)[COLS];

      pt1 = (int [2]) {10, 20};
      pt2 = (int [2][COLS]) { {1,2,3,-9}, {4,5,6,-8} };

      total1 = sum(pt1, 2);
      total2 = sum2d(pt2, 2);
      total3 = sum((int []){4,4,4,5,5,5}, 6);
      printf("total1 = %d\n", total1);
      printf("total2 = %d\n", total2);
      printf("total3 = %d\n", total3);
      return 0;
}
```

Compound Literals

The flc.c Program(2/2)

```
int sum(int ar[], int n)
{
    int i;
    int total = 0;

    for( i = 0; i < n; i++)
        total += ar[i];
    return total;
}

int sum2d(int ar[][COLS], int rows)
{
    int r;
    int c;
    int tot = 0;

    for (r = 0; r < rows; r++)
        for (c = 0; c < COLS; c++)
            tot += ar[r][c];
    return tot;
}
```

Compound Literals

The flc.c Program