

Structured Data Type - Array

- Data types examined so far are atomic:
 - int, long
 - float, double
 - char
- Called “simple” data types because vars hold a single value
- If limited to “simple” data types, many programming applications are tedious
- Solution: use structured data types - types that represent more than one piece of data

Outline

Structured Types

A. Arrays

1. Syntax of declaration
2. Layout in memory
3. Referencing array element
 - a. Subscript use (abuse)
 - b. Array elements as variables
4. Array Initialization
5. Processing arrays
 - a. using loop
 - b. types of processing

Outline (cont)

Structured Types

A. Arrays (cont)

6. Passing array/part of array
 - a. element of array
 - b. entire array
7. Multidimensional arrays
 - a. declaration
 - b. referencing
 - c. processing
 - d. initialization

Techniques

Outline (cont)

Techniques

A. Sorting

1. What is sorted?
2. Selection sort
3. Insertion sort

B. Searching

1. (Un)successful searches
2. Sequential search
 - a. Unsorted array
 - b. Sorted array
3. Binary search (sorted array)

Sample Problem

Problem: track sales totals for 10 people

Daily data:

Employee #	Sale
3	9.99
7	16.29
9	7.99
3	2.59
.	
.	
.	
7	49.00

Representing Sales Data

```
FILE* sdata;
int numE;
float amtS;
float S0, S1, S2, S3, S4, S5, S6, S7, S8,
      S9 = 0.0; /* One var for each employee */

if ((sdata = fopen("DailySales", "r")) ==
    NULL) {
    printf("Unable to open DailySales\n");
    exit(-1);
}
```

Updating Sales Data

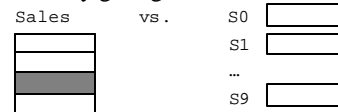
```

while (fscanf (sdata, "%d%f", &numE, &amtS)
== 2) {
    switch (numE) {
        case 0: S0 += amtS; break;
        case 1: S1 += amtS; break;
        case 2: S2 += amtS; break;
        case 3: S3 += amtS; break;
        case 4: S4 += amtS; break;
        case 5: S5 += amtS; break;
        case 6: S6 += amtS; break;
        case 7: S7 += amtS; break;
        case 8: S8 += amtS; break;
        case 9: S9 += amtS; break;
    }
    /* What if 100 employees? */
}

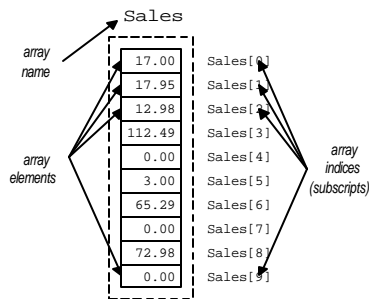
```

Data Structure

- A *Data Structure* is a *grouping* of data items in memory under one name
- When data items same type, can use *Array*
- Using an array, we can set aside a block of memory giving the block a name:



Parts of an Array



Declaring a 1D Array

Syntax: *Type Name*[*Integer Literal*]

Type can be any type we have used so far
 Name is a variable name used for the whole array
 Integer literal in between the square brackets ([]) gives the size of the array (number of sub-parts)
 Size must be a constant value (no variable size)
 Parts of the array are numbered starting from 0
 1-Dimensional (1D) because it has one index

Example:

```

float Sales[10];
/* float array with 10 parts numbered 0 to 9 */

```

Array Indices

- The array indices are similar to the subscripts used in matrix notation:
 Sales[0] is C notation for Sales₀
 Sales[1] is C notation for Sales₁
 Sales[2] is C notation for Sales₂
- The index is used to refer to a part of array
- Note, C does not check your index (leading to index-out-of-range errors)

Accessing Array Elements

- Requires
 - array name,
 - subscript labeling individual element
- Syntax: *name*[*subscript*]
- Example
 Sales[5] refers to the sales totals for employee 5
 Sales[5] can be treated like any float variable:
 Sales[5] = 123.45;
 printf("Sales for employee 5: \$%7.2f\n", Sales[5]);

Invalid Array Usage

Example:

```
float Sales[10];
Invalid array assignments:
Sales = 17.50; /* missing subscript */
Sales[-1] = 17.50; /* subscript out of range */
Sales[10] = 17.50; /* subscript out of range */
Sales[7.0] = 17.50; /* subscript wrong type */
Sales['a'] = 17.50; /* subscript wrong type */
Sales[7] = 'A'; /* data is wrong type */
Conversion will still occur:
Sales[7] = 17; /* 17 converted to float */
```

Array Initialization

- Arrays may be initialized, but we need to give a *list* of values
- Syntax:
Type Name[Size] =
{ value0, value1, value2, ..., valueSize-1 };
value0 initializes *Name[0]*, *value1* initializes *Name[1]*, etc.
values must be of appropriate type (though automatic casting will occur)

Array Initialization

Example:

```
int NumDays[12] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }; /* Jan is 0, Feb is 1, etc. */
int NumDays[13] = { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }; /* Jan is 1, Feb is 2, */
```

Note:

if too few values provided, remaining array members not initialized
if too many values provided, a syntax error or warning may occur (extra values ignored)

Sales Data as an Array

Recall data:

Employee #	Sale
3	9.99
7	16.29
9	7.99
3	2.59
.	
.	
7	49.00

Idea: declare Sales as array, update array items

Updating Sales Data

```
while (fscanf(sdata, "%d%f", &numE, &amtS) == 2) {
    switch (numE) {
        case 0: Sales[0] += amtS; break;
        case 1: Sales[1] += amtS; break;
        case 2: Sales[2] += amtS; break;
        /* cases 3-8 */
        case 9: Sales[9] += amtS; break;
    }
}
```

Q: What's the big deal??

A: Can replace entire switch statement with:
Sales[numE] += amtS;

Updating with Arrays

```
while (fscanf(sdata, "%d%f", &numE, &amtS) == 2) {
    Sales[numE] += amtS;
}
```

When referencing array element can use any expression producing integer as subscript
[] is an operator, evaluates subscript expression then the appropriate location from the array is found
Note, when we have an integer expression, we may want to check subscript before using:

```
if ((numE >= 0) && (numE <= 9))
    Sales[numE] += amtS;
else
    /* Problem employee # */
```

Using Array Elements

Array elements can be used like any variable

read into:

```
printf("Sales for employee 3: ");
scanf("%f",&(Sales[3]));
```

printed:

```
printf("Sales for employee 3 $%7.2f\n",Sales[3]);
```

used in other expressions:

```
Total = Sales[0] + Sales[1] + Sales[2] + ...;
```

Arrays and Loops

- Problem: initialize Sales with zeros

```
Sales[0] = 0.0;
```

```
Sales[1] = 0.0;
```

```
...
```

```
Sales[9] = 0.0;
```

- Should be done with a loop:

```
for (I = 0; I < 10; I++)
```

```
    Sales[I] = 0.0;
```

Processing All Elements of Array

- Process all elements of array A using for:

```
/* Setup steps */
```

```
for (I = 0; I < ArraySize; I++)
```

```
    process A[I]
```

```
/* Clean up steps */
```

- Notes

I initialized to 0

Terminate when I reaches *ArraySize*

Initialize with Data from File

```
if ((istream = fopen("TotalSales", "r"))
    == NULL) {
    printf("Unable to open
    TotalSales\n");
    exit(-1);
}
• File TotalSales
1276.17 (Emp 0's Sales)
917.50 (Emp 1's Sales)
...
2745.91 (Emp 9's Sales)
```

```
for (I = 0; I < 10; I++)
    fscanf(istream, "%f",
           &(Sales[I]));
```

```
fclose(istream);
```

Array Programming Style

- Define constant for highest array subscript:

```
#define MAXEMPS 10
```

- Use constant in array declaration:

```
float Sales[MAXEMPS];
```

- Use constant in loops:

```
for (I = 0; I < MAXEMPS; I++)
```

```
    fscanf(istream, "%f", &(Sales[I]));
```

- If MAXEMPS changes, only need to change one location

Summing Elements in an Array

```
Sales          I    Total
117.00 Sales[0]          0.00
129.95 Sales[1]    0    117.00 (Emp 0 sales)
276.22 Sales[2]    1    246.95 (0 + 1 sales)
...
197.81 Sales[9]    9    1943.89 (All emps)
```

```
total = 0.0;
for (I = 0; I < MAXEMPS; I++)
    total += Sales[I];
```

Maximum Element of an Array

```

Sales      Sales[0]      I      maxS
117.00     Sales[0]             117.00
129.95     Sales[1]             1      129.95 (Max of 0,1)
276.22     Sales[2]             2      276.22 (Max of 0,1,2)
...
197.81     Sales[9]             ...
197.81     Sales[9]             9      276.22 (Max of all)

maxS = Sales[0];
for (I = 1; I < MAXEMPS; I++)
    if (Sales[I] > maxS)
        maxS = Sales[I];
/* Note I starts at 1 */

```

Printing Elements of an Array

```

Sales      Sales[0]      Output:
117.00     Sales[0]             Emp Num      Sales
129.95     Sales[1]             -----      -----
276.22     Sales[2]             0            117.00
...
197.81     Sales[9]             1            129.95
printf("Emp Num      Sales\n");
printf("-----      -----\n");
for (I = 0; I < MAXEMPS; I++)
    printf("%4d%13.2f\n", I,
           Sales[I]);

```

Passing an Element of an Array

- Each element of array may be passed as parameter as if it were variable of base type of array (type array is made of)
- When passing array element as reference parameter put & in front of array element reference ([] has higher precedence)
 - does not hurt to put parentheses around array reference though
 - example &(Sales[3])

Passing Array Element Examples

```

Passing by value:          Passing by reference:
void printEmp(int eNum, float eSales) {
    printf("%4d%13.2f\n", eNum, eSales);
}
in main:
printf("Emp Num      Sales\n");
printf("-----      -----\n");
for (I = 0; I < MAXEMPS; I++)
    printEmp(I, Sales[I]);

void updateSales(float *eSales, float newS) {
    *eSales = *eSales + newS;
}
in main:
while (fscanf(sdata, "%d%f", &numE, &amtS) == 2)
    for (I = 0; I < MAXEMPS; I++)
        updateSales(&(Sales[numE]), amtS);

```

Passing Whole Array

- When we pass an entire array we always pass the address of the array
 - syntax of parameter in function header:


```
BaseType NameforArray[]
```

 - note, no value between [] - compiler figures out size from argument in function call
 - in function call we simply give the name of the array to be passed
 - since address of array is passed, changes to elements in the array affect the array passed as argument (no copy of array is made)

Passing Whole Array Example

```

float calcSalesTotal(float S[]) {
    int I;
    float total = 0.0;
    for (I = 0; I < MAXEMPS; I++)
        total += S[I];
    return total;
}
in main:
printf("Total sales is %7.2f\n",
       calcSalesTotal(Sales));

```

Size of an Array

- Sometimes we know we need an array, but not how big the array is going to be
- One solution:
 - allocate a very large array
 - integer to indicate num elements of array in use
 - loops use num elements when processing
- Example:

```
float Sales[MAXEMPS];
int NumEmps = 0;
processing:
for (I = 0; I < NumEmps; I++)
    process Sales[I];
```

Sorting Arrays

- One common task is to sort data
- Examples:
 - Phone books (by name)
 - Checking account statement (by check #)
 - Dictionaries (by word)
 - NBA scoring leaders (by points)
 - NBA team defense (by points)

Sorting Order

Team defense		Individual Scoring	
90.0	Bulls	31.0	Jordan
90.3	Heat	28.7	Malone
91.0	Knicks	27.9	O'Neill
...		...	
98.9	76ers	0.3	Bricklayer

Sorted in *ascending* order Sorted in *descending* order

Sorting in Arrays

Data sorted usually same type, sorted in array

A	A
6	1
4	4
8	6
10	8
1	10

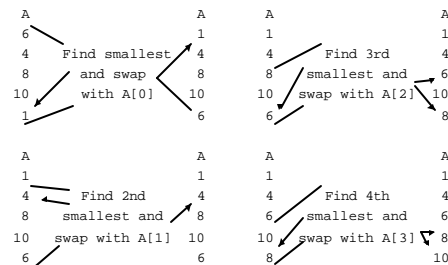
before sort after sort

Selection Sorting

One approach (N is # elements in array):

1. Find smallest value in A and swap with A[0]
2. Find smallest value in A[1] .. A[N-1] and swap with A[1]
3. Find smallest value in A[2] .. A[N-1] and swap with A[2]
4. Continue through A[N-2]

Selection Sort Example



Selection Sort Notes

- If array has N elements, process of finding smallest repeated N-1 times (outer loop)
- Each iteration requires search for smallest value (inner loop)
- After inner loop, two array members must be swapped
- Can search for largest member to get descending-order sort

Selection Sort Algorithm

```

For J is 0 to N-2      A      K      Smallest
Find smallest value in A[J], 6      6
A[J+1] .. A[N-1]     4      1      4
Store subscript of smallest
in Index              8      2
Swap A[J] and A[Index] 10     3
Find smallest in A[J..N-1] 1      4      1

Suppose J is 0
Smallest = A[0];
for (K = 1; K < N; K++)
    if (A[K] < Smallest)
        Smallest = A[K];
    
```

But we need location of smallest, not its value to swap

Selection Sort Algorithm

```

Find location of smallest rather than value:
SmallAt = 0;
for (K = 1; K < N; K++)
    if (A[K] < A[SmallAt])
        SmallAt = K;

Swapping two elements:
Temp = A[J];
A[J] = A[SmallAt];
A[SmallAt] = Temp;
    
```

J is 0, find smallest:

A	K	SmallAt
6		0
4	1	1
8	2	
10	3	
1	4	4

Swap A[SmallAt], A[0]

Code for Selection Sort

```

for (J = 0; J < N-1; J++) { /* 1 */
    SmallAt = J;           /* 2 */
    for (K = J+1; K < N; K++) /* 3 */
        if (A[K] < A[SmallAt]) /* 4 */
            SmallAt = K;      /* 5 */
    Temp = A[J];           /* 6 */
    A[J] = A[SmallAt];    /* 7 */
    A[SmallAt] = Temp;    /* 8 */
}
    
```

Selection Sort Example

S#	J	K	Sml	Effect	S#	J	K	Sml	Effect
1	0			Start outer	4				False, skip 5
2	0			Init SmallAt	3	4			Repeat inner
3	1			Start inner	4				False, skip 5
4				True, do 5	6-8				Swap A[1], A[1]
5	1			Update SmallAt	1	2			Repeat outer
3	2			Repeat inner	2	2			Init SmallAt
4				False, skip 5	3	3			Start inner
3	3			Repeat inner	4				False, skip 5
4				False, skip 5	3	4			Repeat inner
3	4			Repeat inner	4				True, do 5
4				True, do 5	5	4			Update SmallAt
5	4			Update SmallAt	6-8				Swap A[2], A[4]
6-8				Swap A[0], A[4]	1	3			Repeat outer
1	1			Repeat outer	2	3			Init SmallAt
2				Init SmallAt	3	4			Start inner
3	2			Start inner	4				True, do 5
4				False, skip 5	5	4			Update SmallAt
3	3			Repeat inner	6-8				Swap A[3], A[4]

Modularizing Sort

- Want to make sort more readable
- Also, make sort a separate function
- First make Swap a separate function
- Have to pass array elements as reference params

```

void Swap(int *n1, int *n2) {
    int temp;
    temp = *n1;
    *n1 = *n2;
    *n2 = temp;
}

for (J = 0; J < N-1; J++) {
    SmallAt = J;
    for (K = J+1; K < N; K++)
        if (A[K] < A[SmallAt])
            SmallAt = K;
    Swap(&(A[J]), &(A[SmallAt]));
}
    
```

Modularizing Sort - Find Smallest

- Can make process of finding smallest a separate function
 - Sort becomes:
- ```

int findSmallest(
int Aname[], int J, int N)
{
 int MinI = J;
 int K;
 for (K = J+1; K < N; K++)
 if (Aname[K] <
 Aname[MinI])
 MinI = K;
 return MinI;
}

void sort(int A[], int N) {
 int J;
 int SmallAt;
 for (J = 0; J < N-1; J++)
 {
 SmallAt =
 findSmallest(A,J,N);
 Swap(&(A[J]),
 &(A[SmallAt]));
 }
}

```
- Note whole array passed

## Modularizing Sort Itself

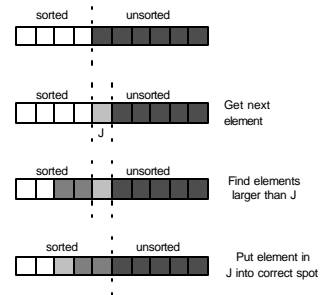
- Can then make the sort routine itself a function
  - Localize variables such as J, SmallAt in function
  - Pass N only if different from size of array
- ```

void sort(int A[], int N) {
    int J;
    int SmallAt;
    for (J = 0; J < N-1; J++)
    {
        SmallAt =
        findSmallest(A,J,N);
        Swap(&(A[J]),
            &(A[SmallAt]));
    }
}
    
```

Another Sort: Insertion

- Idea: sort like a hand of cards
- Algorithm:
 - Assume A[0] .. A[0] is sorted segment
 - Insert A[1] into sorted segment A[0 .. 0]
 - Insert A[2] into sorted segment A[0 .. 1]
 - Insert A[3] into sorted segment A[0 .. 2]
 - continue until A[N-1] inserted

Inserting into Sorted Segment



Insertion Sort Code

```

void insertIntoSegment(int Aname[], int J) {
    int K;
    int temp = Aname[J];
    for (K = J; (K > 0) && (Aname[K-1] > temp); K--)
        Aname[K] = Aname[K-1];
    Aname[K] = temp;
}

void sort(int A[], int N) {
    int J;
    for (J = 1; J < N; J++)
        insertIntoSegment(A,J);
}
    
```

Searching Arrays

Prob: Search array A (size N) for value Num

Example:

search array for a particular student score

A	N
6	5
4	
8	Num
10	8
1	

Sequential search: start at beginning of array, examine each element in turn until desired value found

Sequential Search

- Uses:

- event-controlled loop
- must not search past end of array

- Code:

```
index = 0;
while ((index < N) && (A[index] != Num)) /* 1 */
    index++; /* 2 */ /* 3 */
```

- When loop finishes either:

- index is location of value or
- index is N

- Test

```
if (index < N) /* 4 */
    printf("Found at %d\n", index); /* 5 */
else
    printf("Not found\n"); /* 6 */
```

Sequential Search Example

A	N	S#	Num	N	index	effect
6	5		8	5		
4		1			0	init index
8	Num	2				true, do 3
10	8	3			1	inc index
1		2				true, do 3
		3			2	inc index
		2				false, exit
		4				true, do 5
		5				print

Sequential Search Example

A	N	S#	Num	N	index	effect
6	5		5	5		
4		1			0	init index
8	Num	2				true, do 3
10	5	3			1	inc index
1		2				true, do 3
		3			2	inc index
		2				true, do 3
		3			3	inc index
		2				true, do 3
		3			4	inc index
		2				true, do 3
		3			5	inc index
		2				false, exit
		4				false, do 6
		6				print

Sequential Search (Sorted)

```
index = 0;
while ((index < N) && (A[index] < Num)) /* 1 */
    index++; /* 2 */ /* 3 */
if ((index < N) && (A[index] == Num)) /* 4 */
    printf("Found at %d\n", index); /* 5 */
else
    printf("Not found\n"); /* 6 */
```

Can stop either when value found or a value larger than the value being searched for is found

While loop may stop before index reaches N even when not found (need to check)

Sorted Sequential Search Example

A	N	S#	Num	N	index	effect
1	5		5	5		
4		1			0	init index
6	Num	2				true, do 3
8	5	3			1	inc index
10		2				true, do 3
		3			2	inc index
		2				false, exit
		4				false, do 6
		6				print

Can do better when sorted

Binary Search

Example: when looking up name in phone book, don't search start to end

Another approach:

- Open book in middle
- Determine if name in left or right half
- Open that half to its middle
- Determine if name in left or right of that half
- Continue process until name is found (or not)

Code for Binary Search

```

first = 0;
last = N - 1;
mid = (first + last) / 2;
while ((A[mid] != num) && (first <= last)) {
    if (A[mid] > num)
        last = mid - 1;
    else
        first = mid + 1;
    mid = (first + last) / 2;
}

```

Binary Search Example

```

0: 4 0 first Num: 101
1: 7
2: 19
3: 25
4: 36
5: 37
6: 50 6 mid
7: 100 7 first 7 first
8: 101 8 mid -- found
9: 205 9 last
10: 220 10 mid
11: 271
12: 306
13: 321 13 last 13 last

```

Binary Search Example

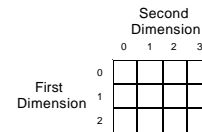
```

0: 4 0 first Num: 53
1: 7
2: 19
3: 25
4: 36
5: 37
6: 50 6 mid
7: 100 7 first 7 first 7 first, 6 last
8: 101 8 mid last, 7 first
9: 205 9 last mid
10: 220 10 mid
11: 271
12: 306
13: 321 13 last 13 last

```

2-Dimensional Array Declaration

- Syntax: *BaseType Name[IntLit1][IntLit2];*
- Examples:
 - int Scores[100][10]; /* 100 x 10 set of scores */
 - char Maze[5][5]; /* 5 x 5 matrix of chars for Maze */
 - float FloatM[3][4]; /* 3 x 4 matrix of floats */



2D Array Element Reference

- Syntax: *Name[intExpr1][intExpr2]*
- Expressions are used for the two dimensions in that order
- Values used as subscripts must be legal for each dimension
- Each location referenced can be treated as variable of that type
- Example: Maze[3][2] is a character

2D Array Initialization

- 2D arrays can also be initialized
- Syntax:


```

BaseType Name[Dim1][Dim2] = { val0, val1,
                               val2, val3, val4, ... };

```

 values are used to initialize first row of matrix, second row, etc.


```

BaseType Name[Dim1][Dim2] = {
    { val0-0, val0-1, val0-2, ... } /* first row */
    { val1-0, val1-1, val1-2, ... } /* second row */
    ... };

```

Processing 2D Arrays

- Use nested loops to process 2D array

```
Type Name[Dim1][Dim2];
for (J = 0; J < Dim1; J++)
    for (K = 0; K < Dim2; K++)
        process Name[J][K];
```

- Example: print 5x5 Maze

```
for (J = 0; J < 5; J++) {
    for (K = 0; K < 5; K++)
        printf("%c", Maze[J][K]);
    printf("\n");
}
```

2D Example

```
int Scores[100][10]; /* 10 scores - 100 students */
int J;
int K;

for (J = 0; J < 100; J++) {
    printf("Enter 10 scores for student %d: ", J);
    for (K = 0; K < 10; K++)
        scanf("%d", &(Scores[J][K]));
}
```

Passing 2D Array Parameters

- A single value can be passed as either a value or as a reference parameter
- 2D array may be passed by reference using name, syntax of parameter:

```
Type ParamName[][Dim2]
Dim2 must be the same literal constant used in
declaring array
```

- Each row of 2D array may be passed as a 1D array

Passing Element of Array

```
int Scores[100][10]; /* 10 scores - 100 students */
int J;
int K;

void readScore(int *score) {
    scanf("%d", score);
}

for (J = 0; J < 100; J++) {
    printf("Enter 10 scores for student %d: ", J);
    for (K = 0; K < 10; K++)
        readScore(&(Scores[J][K]));
}
```

Passing Row of Array

```
int Scores[100][10]; /* 10 scores - 100 students */
int J;

void readScore(int *score) {
    scanf("%d", score);
}

void readStudent(int Sscores[], int Snum) {
    int K;

    printf("Enter 10 scores for student %d: ", Snum);
    for (K = 0; K < 10; K++)
        readScore(&(Sscores[K]));
}

for (J = 0; J < 100; J++)
    readStudent(Scores[J], J);
```

Passing Entire Array

```
int Scores[100][10]; /* 10 scores - 100 students */
void readScore(int *score) {
    scanf("%d", score);
}

void readStudent(int Sscores[], int Snum) {
    int K;
    printf("Enter 10 scores for student %d: ", Snum);
    for (K = 0; K < 10; K++)
        readScore(&(Sscores[K]));
}

void readStudents(int Ascores[][10]) {
    int J;
    for (J = 0; J < 100; J++)
        readStudent(Ascores[J], J);
}

readStudents(Scores);
```

2D Array Example

```
int Scores[100][10]; /* 10 scores - 100 students */
int totalStudent(int Sscores[]) {
    int J;
    int total = 0;
    for (J = 0; J < 10; J++)
        total += Sscores[J];
    return total;
}
float averageStudents(int Ascores[][10]) {
    int J;
    int total = 0;
    for (J = 0; J < 100; J++)
        total += totalStudent(Ascores[J]);
    return (float) total / 100;
}
```

Multi-Dimensional Array Declaration

- Syntax:
BaseType Name[IntLit1][IntLit2][IntLit3]...;
- One constant for each dimension
- Can have as many dimensions as desired
- Examples:
int Ppoints[100][256][256]; /* 3D array */
float TimeVolData[100][256][256][256];
/* 4D array */

Multi-Dim Array Reference

- To refer to an element of multi-dimensional array use one expression for each dimension
syntax:
Name[Expr1][Expr2][Expr3] / 3D */*
Name[Expr1][Expr2][Expr3][Expr4] / 4D */*
etc.
- First expression - first dimension, 2nd expression - 2nd dimension, etc.
- C does not check dimensions