Arrays

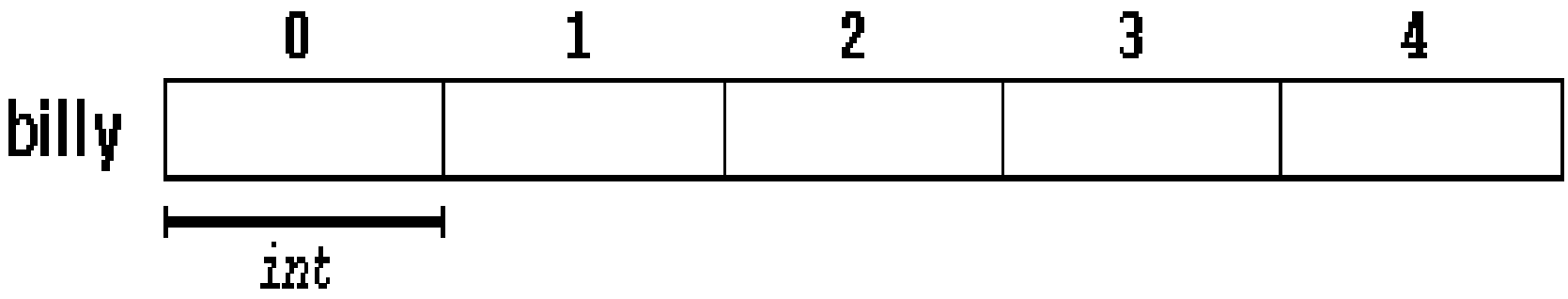
CSM 387 Data Structures 1

Definition

- An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier.
- That means that, for example, we can store 5 values of type *int* in an array without having to declare 5 different variables, each one with a different identifier.

Example

- For example, an array to contain 5 integer values of type *int* called billy could be represented like this:



- *int* billy [5];

Initializing arrays

- Regular Array of local scope will NOT be initialized to any default value
  - Content undefined
- Global and static arrays are automatically initialized with their default values
  - Fundamental types, eg. Zeros

Initialization

- *int* billy [5] = { 16, 2, 77, 40, 12071 };
- *int* billy [] = { 16, 2, 77, 40, 12071 };

- This declaration would have created an array like this:

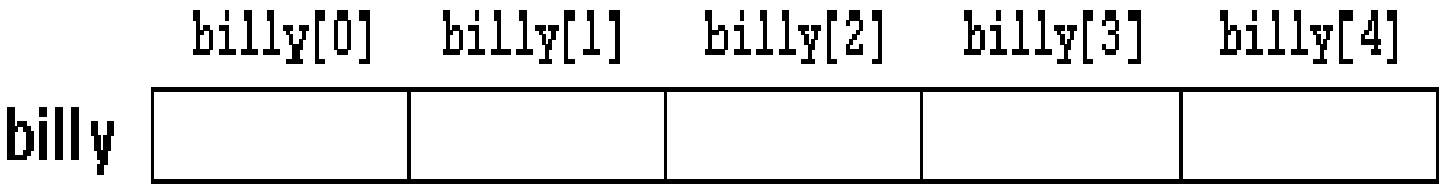
	0	1	2	3	4
billy	16	2	77	40	12071

Partial initialization

- An array can also be partially initialized.
- *int* billy [5] = { 16, 2, , , 12071 };
- The above array declaration assigns initial values to billy[0], billy[1] and billy[4].
- But data elements billy[2] and billy[3] do not have initial value.

Accessing values

- In C++, access of array elements are through the identifier of an array and an index.
- Its possible to read and modify array values
- *name* [index]



- Thus, *billy[2] =75; //* modification
- *A = billy[2]; //* variable A is 75 - read

Properties of an array

- For an array of *n* elements
  - The indices of its elements range from 0 to *n*-1
  - An array element can be modified with an assignment statement
  - Array elements can be used in arithmetic expressions
- In C++, the identifier of a one-dimensional array represents the address of the first array element.

Warning!!

- In C++ it is syntactically correct to exceed the valid range of indices for an array.
- This can create problems, since accessing out-of-range elements do not cause compilation errors but can cause runtime errors.
- This is justified with the use of pointers in C++.

The use of []

```
int billy[5];          // declaration of a new array
billy[2] = 75;         // access to an element of the array.
```

Array operations

```
billy[0] = a;
billy[a] = 75;
b = billy [a+2];
billy[billy[a]] = billy[2] + 5;
```

What is the output?

```
• // arrays example
• #include <iostream>
• using namespace std;
• int billy [] = {16, 2, 77, 40, 12071};
• int n, result=0;
•
• int main ()
• {
•   for ( n=0 ; n<5 ; n++ )
•   {
•     result += billy[n];
•   }
•   cout << result;
•   return 0;
• }
```

Output

- 12206
- 16+2+77+40+12071

Exercise

- Suppose the address register of a CPU is 32-bit long and the the size of an integer is four bytes. If the starting address of billy[0] is 0x22FF50, what is the starting address of billy[4]?

Exercise

- Suppose the address register of a CPU is 32-bit long and the size of an integer is four bytes. If the starting address of billy[0] is 0x22FF50, what is the starting address of billy[4]?

Array arithmetic

- *billy+2* is an arithmetic operation of address which means the address of billy adds two times of the size of a given data type, that is 4 for the integer type.
- Thus, billy+2 is equivalent 0x22FF50+4*2 which is exactly the address of billy[2], 0X22FF58,

Indexing 1D array

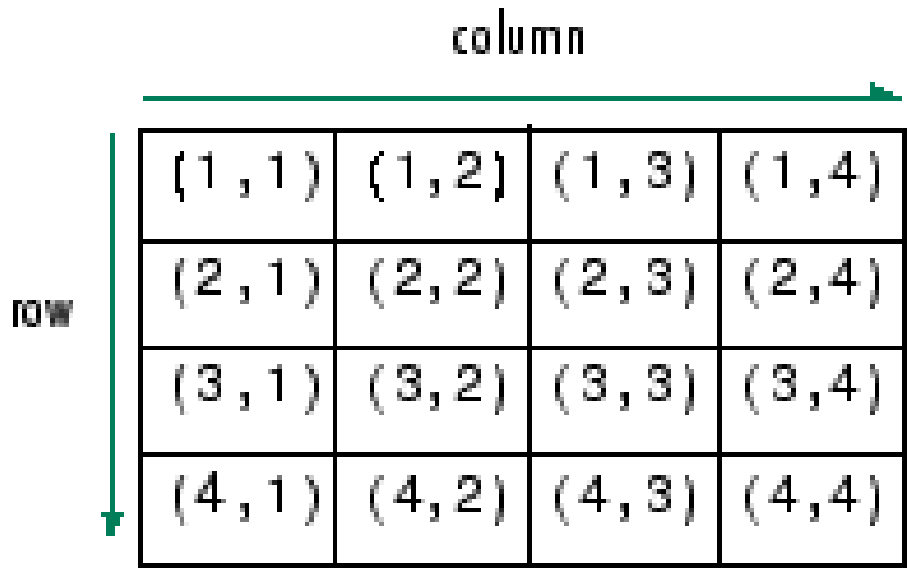
- Let the starting address of array *a[n]* be *loc* and the type of the elements of *a[n]* be ***T***.
- The address of *a[0]* = *loc* or &*a[0]* = *loc*
- Then the starting address of element *a[i]* is

- *a+i = &a[i] = loc + sizeof(T) * i.*

- *Since there are i elements before a[i] and each element occupying sizeof(t) bytes*

Multidimensional arrays

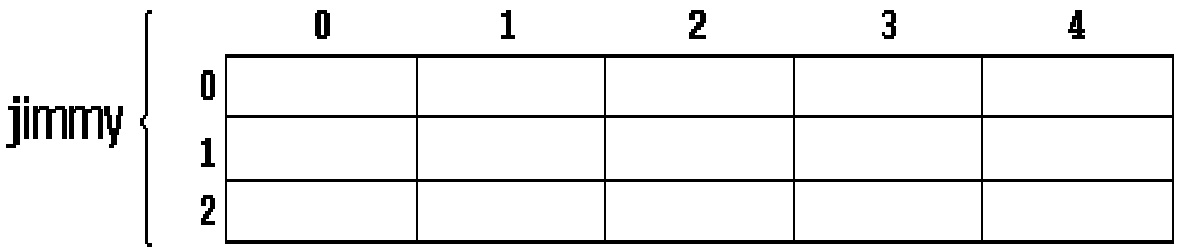
- Multidimensional arrays can be described as "arrays of arrays". For example, a 2-dimensional array can be imagined as a 2-dimensional table made of elements, all of them of a same uniform data type.



Declaration

- jimmy represents a 2-dimensional array of 3 per 5 elements of type int. The way to declare this array in C++ would be:

- *int jimmy [3][5];*



Definition

- A two dimensional array is actually a one-dimensional array of *m* elements such that each element is a one-dimensional array of size *n* of the specified type.

- int jimmy[m][n];

Example

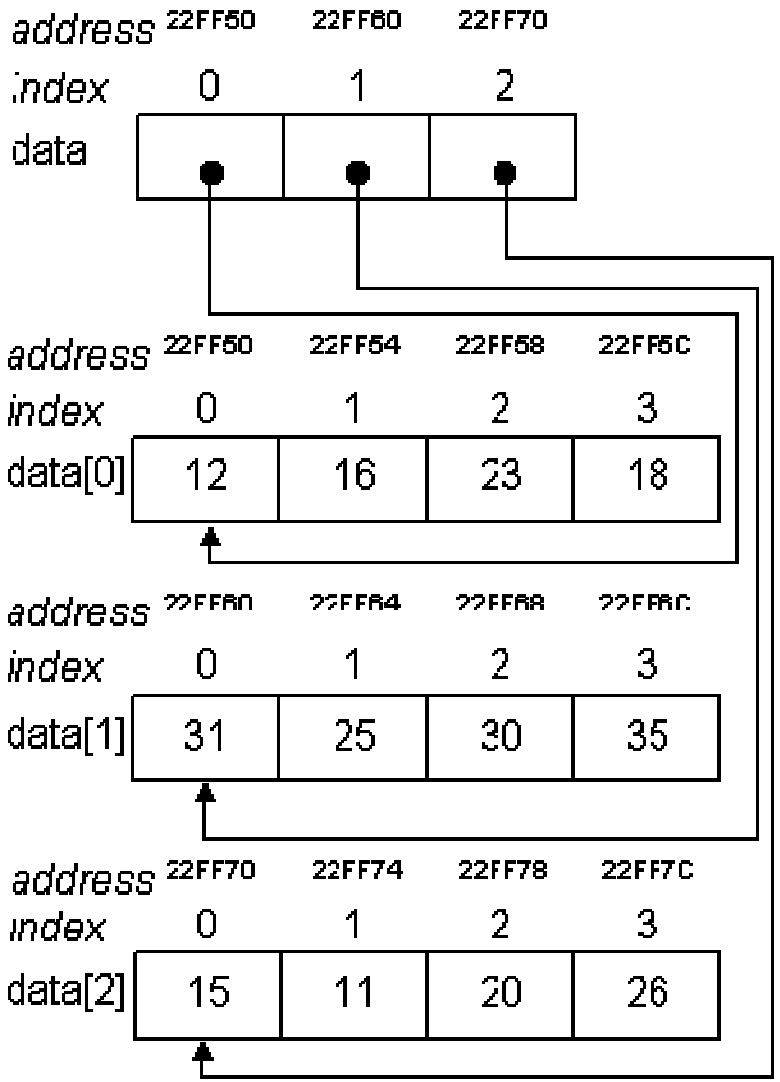
- int jimmy[3][5];

- Array *jimmy* is a one-dimensional array of three elements *jimmy[0], jimmy[1],* and *jimmy[2]* such that each element is a one-dimensional array of five elements.

Initialization

- With a one-list of numbers enclosed by curl brackets
- **int** jimmy[3][5] = {12, 16, 23,6, 18, 31, 25, 30, 35,16, 15, 11, 20, 26,36};
- With a two-level list enclosed by curly brackets such that the first level elements are lists of numbers enclosed by curly brackets.
- **int** jimmy[3][5] = {{12, 16, 23, 18, 6},
  {31, 25, 30, 35, 16},
  {15, 11, 20, 26, 36}};

Diagram of array data[3][4]



Generalization

- In general, let the starting address of two-dimensional array *a[m][n]* be *loc* and the type of the elements of *a[m][n]* be ***T***. Let us view array *a[m][n]* as the following *m x n* matrix:

$$
\left.\begin{bmatrix}
a_{0,0} & a_{0,1} & \cdots & \cdots & \cdots & a_{0,n-1} \\
a_{1,0} & a_{1,1} & \cdots & \cdots & \cdots & a_{1,n-1} \\
\vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\
a_{i,0} & a_{i,1} & \cdots & a_{i,j} & \cdots & \cdots \\
\vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\
\underbrace{a_{m-1,0} \quad a_{m-1,1} \quad \cdots}_{j \text{ elements in row } i} & & & \cdots & \cdots & a_{m-1,n-1}
\end{bmatrix}\right\} \begin{array}{l} i \text{ rows and } n \text{ elements} \\ \text{in each row} \end{array}
$$

$(i \times n + j)$-th element

2D array indexing

- There are *i* rows before element $a_{i,j}$ such that each row has *n* elements, and there are *j* elements in row *i* before element $a_{i,j}$. Totally, there are *i* x *n*+*j* elements, that each element is of **sizeof**(T) bytes, in front of $a_{i,j}$.
- The address of &a[0][0]=loc and the starting address of array element a[i][j] is given as:

*(a+i)+j = a[i]+j = &a[i][j] = loc + **sizeof**(T) * (i * n + j).

Three-dimensional array

matrix 0 matrix 1

$$\underbrace{\begin{bmatrix} a_{0,0} & a_{0,0} & \cdots & a_{0,p-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,p-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,p-1} \end{bmatrix} \begin{bmatrix} a_{0,0} & a_{0,0} & \cdots & a_{0,p-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,p-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,p-1} \end{bmatrix}}_{j \text{ matrices and } n \times p \text{ elements in each matrix}} \cdots$$

matrix j

$$\begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & \cdots & \cdots & a_{0,p-1} \\ a_{1,0} & a_{1,1} & \cdots & \cdots & \cdots & a_{1,p-1} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{j,0} & a_{j,1} & \cdots & a_{j,k} & \cdots & \cdots \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & \cdots & \cdots & a_{n-1,p-1} \end{bmatrix} \cdots$$

j rows and p elements in each row

k elements in row j

$(j \times n \times p + j \times p + k)$-th element

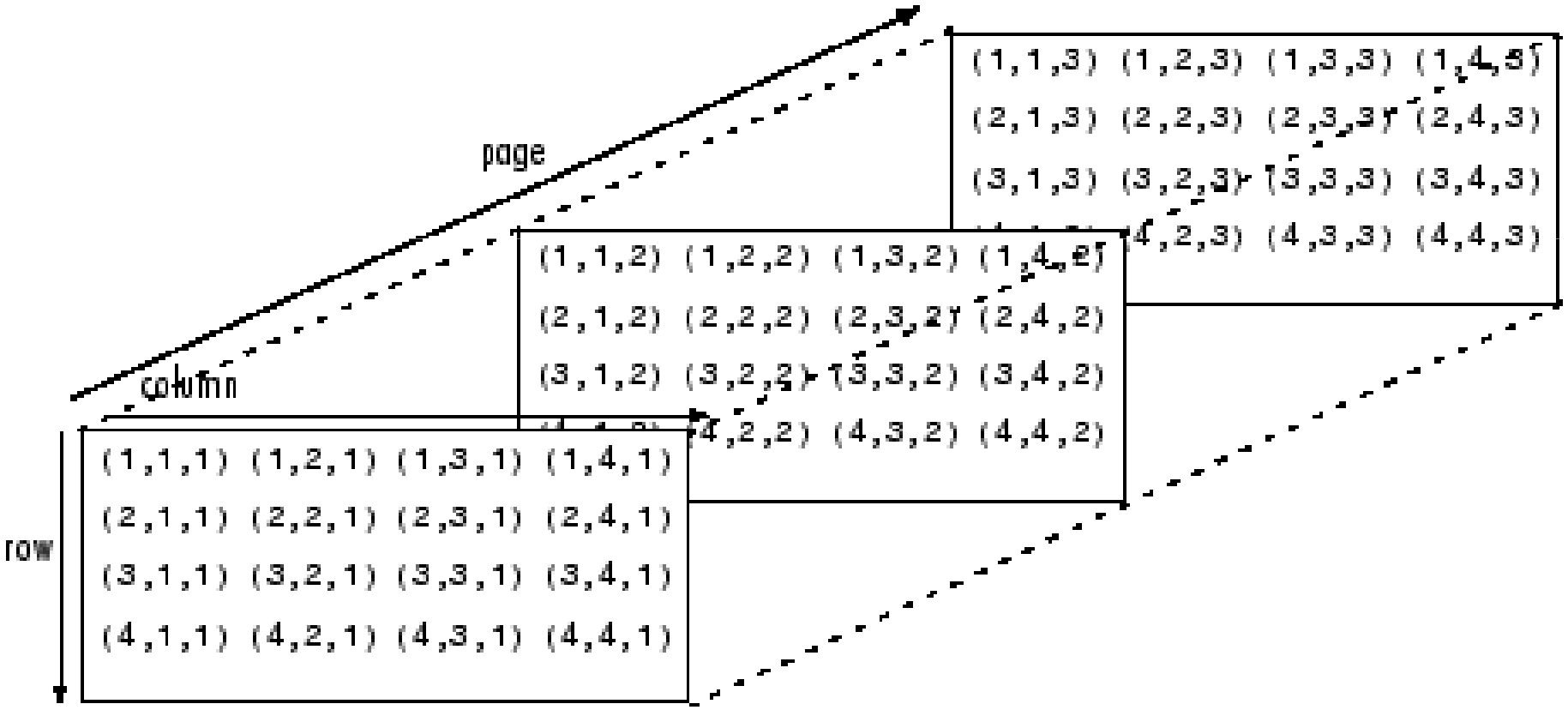
matrix n-2 matrix n-1

$$\cdots \begin{bmatrix} a_{0,0} & a_{0,0} & \cdots & a_{0,p-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,p-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,p-1} \end{bmatrix} \begin{bmatrix} a_{0,0} & a_{0,0} & \cdots & a_{0,p-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,p-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,p-1} \end{bmatrix}$$

Array a [m] [n] [p]

is matrices of size *n x p*

3D array (cont)



3D array indexing

- There are *i* matrices, of size *nxp*, before the one containing matrix *i*.
- In matrix *i*, there are *j* rows before element $a_{j,k}$ such that each row has *p* elements, and there are *k* elements in row *j* before element $a_{i,j}$.
- There are *i*x*n*x*p*+*j*x*p*+*k* elements, that each element is of **sizeof**(T) bytes, in front of $a_{j,k}$ of matrix *i*.

3D array indexing (cont)

- The address of array a is &a[0][0][0]=loc and the starting address of array element a[i][j][k] is given as:

&a[i][j][k] = loc + sizeof(T) * (i * n * p + j * p + k)

Array indexing

- 1D
  - *a+i = &a[i] = loc + **sizeof**(T) * i.*
- 2D
  - *(a+i)+j = a[i]+j = &a[i][j] = loc + **sizeof**(T)*(i *n + j).
- 3D
  - &a[i][j][k] = loc + **sizeof**(T) * (i * n * p + j * p + k)