# Structured Types

Structured data type: collection of components whose organization is characterized by method used to access individual components.

Examples of structured types in C++

array _____

struct _____

union _____

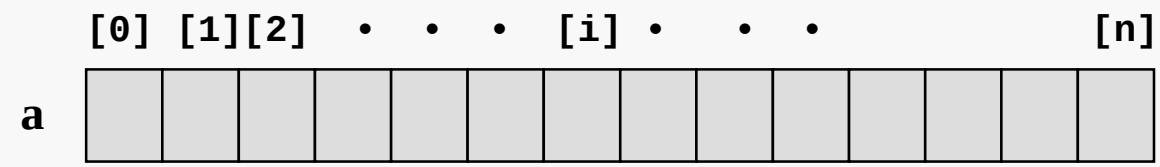
class _____

# Definitions

Array : sequential list of variables of the same type.

Mathematics: $a_0$ $a_1$ $a_2$ • • • $a_i$ • • • $a_n$

C++: a[0] a[1] a[2] • • • a[i] • • • a[n]




- Array elements have the same properties as simple variables of the same type.
- Array elements are operated upon the same as corresponding simple typed variables.

# Array Declaration

The declaration of an array variable must specify the name of the array, the type of element the array is to contain, and the dimension (or size) of the array.

The dimension must be a constant or a constant expression.

```
const int BufferSize = 256;
const int MaxSize    =  12;
char  Buffer[BufferSize];     // constant integer dimension
int   DiceFreq[MaxSize - 1];  // constant integer expression,
                              // used as dimension
int   RollValue = 11;         // variable integer
float Payroll[RollValue];     // NOT valid - RollValue is not
                              // a constant
```

# Indices

An array is indexed using the integer data type and indices always start at 0.

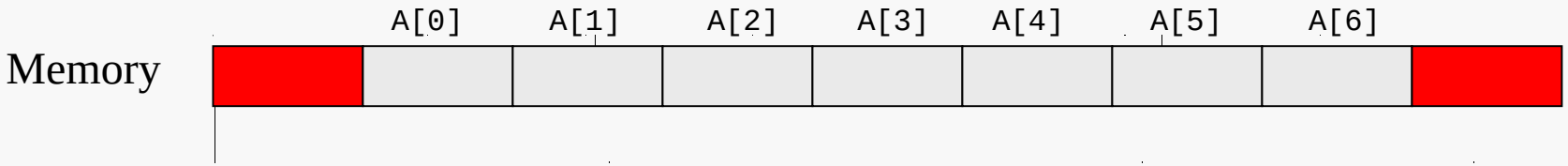
```
const int BufferSize = 256;
char  Buffer[BufferSize];

Buffer[0] = 'x';      // assigns value to first array element

cout << Buffer[132];  // prints value of 133rd array element
```

Logically valid indices for an array range from 0 to N-1, where N is the dimension of the array.

```
char A[7];
```

stands for

A[0], A[1], A[2], A[3], A[4], A[5], and A[6]  **Logically**, there is no A[7] !



# Accessing

An array is a structured data type which stores multiple values of the same data type.

Each value is referenced via an index into the array.

Array declaration:

```
const int MaxStudents = 30;
int Quiz[MaxStudents];
int Exam[MaxStudents];
int i, Hold, NextQuiz;
```

Array references:

```
Exam[1]  =  95;
cout << setw(5) << Quiz[5];
NextQuiz =  Quiz[i + 1];
```

# Aggregate Operations

An operation on a data structure as a whole, as opposed to an operation on an individual component of the data structure. Consider the declarations:

```
const int Size = 50;
int x[Size];
int y[Size];
```

and assume the arrays `x[]` and `y[]` have been initialized, which of the following aggregate operations are allowed?

| | | |
|---|---|---|
| Assignment: | `x = y;` | _____ |
| Comparison: | `x = = y;` | _____ |
| I/O: | `cout << x;` | _____ |
| Arithmetic: | `x + y` | _____ |
| Return: | `return x;` | _____ |
| Parameter: | `someFunc(x);` | _____ |

# Implementing Aggregate Operations

The operations discussed on the previous slide that are not supported automatically may still be implemented by user-defined code. For example, an equality test could be written as:

```
bool AreEqual = true;

for (int index = 0; ( index < Size && AreEqual ); index++) {
    if (x[index] != y[index])
       AreEqual = false;
}
```

The standard way to process an array is to write a for loop and use the loop counter as an index into the array; as the loop iterates, the index "walks" down the array, allowing you to process the array elements in sequence.

Note the use of the Boolean variable in the for loop header. This causes the for loop to exit as soon as two unequal elements have been found, improving efficiency at run-time.

# Array Usage

Like other C++ variables, arrays do not contain useful initial values when declared.

```
const int Max = 100;
int Height[Max];
for (int i = 0; i < Max; i++)
    Height [ i ]  =  0;
```

A common **logical** error in processing arrays is exceeding the valid index range:

```
int X[Max];
```

`X['a']` will not compile.

`X[100]` will compile (according to the Standard).

Recall Boolean Short Circuiting:

```
int i = 0 ;
while( (i < Max) && (X[i] != desiredElement) ) {
  i++;
}
```

# More on Array Indices

What happens when a statement uses an array index that is out of bounds?

First, there is no automatic checking of array index values at run-time (some languages do provide for this). Consider the code fragment below:

```
const int NumScores = 100;
int MaxScore = 25;
int Score[NumScores];
for (int i = 0; i <= NumScores; i++) // initialize too many
    Score [ i ]  =  0;
```

On the last pass through the for loop body, the int value 0 will be stored at the (logical) location `Score[100]`. But what is that, since the only logically valid locations in `Score[]` would be `Score[0]` through `Score[99]`?

`Score[100]` is interpreted as referring to the next `int`-sized chunk of memory following the last cell of the array, so the value 0 will be stored there . . .

. . . whether that makes sense or not.

# More on Array Indices

Consider the possibilities. The memory location Score[100] may:

- store a variable declared in your program
- store an instruction that is part of your program (unlikely on modern machines)
- not be allocated for the use of your program

In the first case, the error shown on the previous slide would cause the value of that variable to be altered. Since there is no statement that directly assigns a value to that variable, this effect seems very mysterious when debugging.

In the second case, if the altered instruction is ever executed it will have been replaced by a nonsense instruction code. This will (if you are lucky) result in the system killing your program for attempting to execute an illegal instruction.

In the third case, the result depends on the operating system you are using. Some operating systems, such as Windows 95/98 do not carefully monitor memory accesses and so your program may corrupt a value that actually belongs to another program (or even the operating system itself). Other operating systems, such as Windows NT or UNIX, will detect that a memory access violation has occurred and suspend or kill your program.

# Array Initialization at Declaration

An array variable may be initialized at the point of declaration:

```
int height[3] = {5, 7, 8};
```

If there are fewer initializers than elements in the array, the remaining elements are automatically initialized to 0. Thus:

```
int height[3] = {0};
```

is equivalent to:

```
int height[3] = {0, 0, 0};
```

and:

```
int n[10] = {-1};
```

explicitly initializes the first array element to -1 and implicitly initializes the other nine elements of the array to 0.

Providing too many initializers causes a compile-time error:

```
int height[3] = {5, 7, 8, 9};
```

# Array Initialization via a Loop

Of course, an array variable also may be initialized via a simple loop:

```
const int MaxScores = 75;
double Score[MaxScores];

for (int idx = 0; idx < MaxScores; idx++) {
   Score[idx] = 0.0;
}
```

The loop may easily be modified to assign different values to different array elements or to obtain the initial values from a file or a function call.

Remember: one of the Deadly Sins of Programming is failure to properly initialize variables.

In some problems array bounds errors are often much easier to detect if you initialize your arrays to dummy values that can be distinguished from logically correct values (using the value -1.0 instead of 0.0 above, for instance).

# Array Elements as Parameters

A single array element can be passed as an actual parameter to a function.

```
void Swap(float& a1, float& a2)
{
// Local data
   float temp;
   temp = a1;
   a1 = a2;
   a2 = temp;
   return;
}
```

**Simple float reference parameters**

To exchange `Status[3]` and `Status[5]` inside array Status we invoke swap as:

```
Swap(Status[3], Status[5]);
```

Note well: when passing a single array element, the actual parameter is the array name, with an index.

In other words, `Status` refers to the entire array while `Status[k]` refers to the single element at index `k`.

# Entire Arrays as Parameters

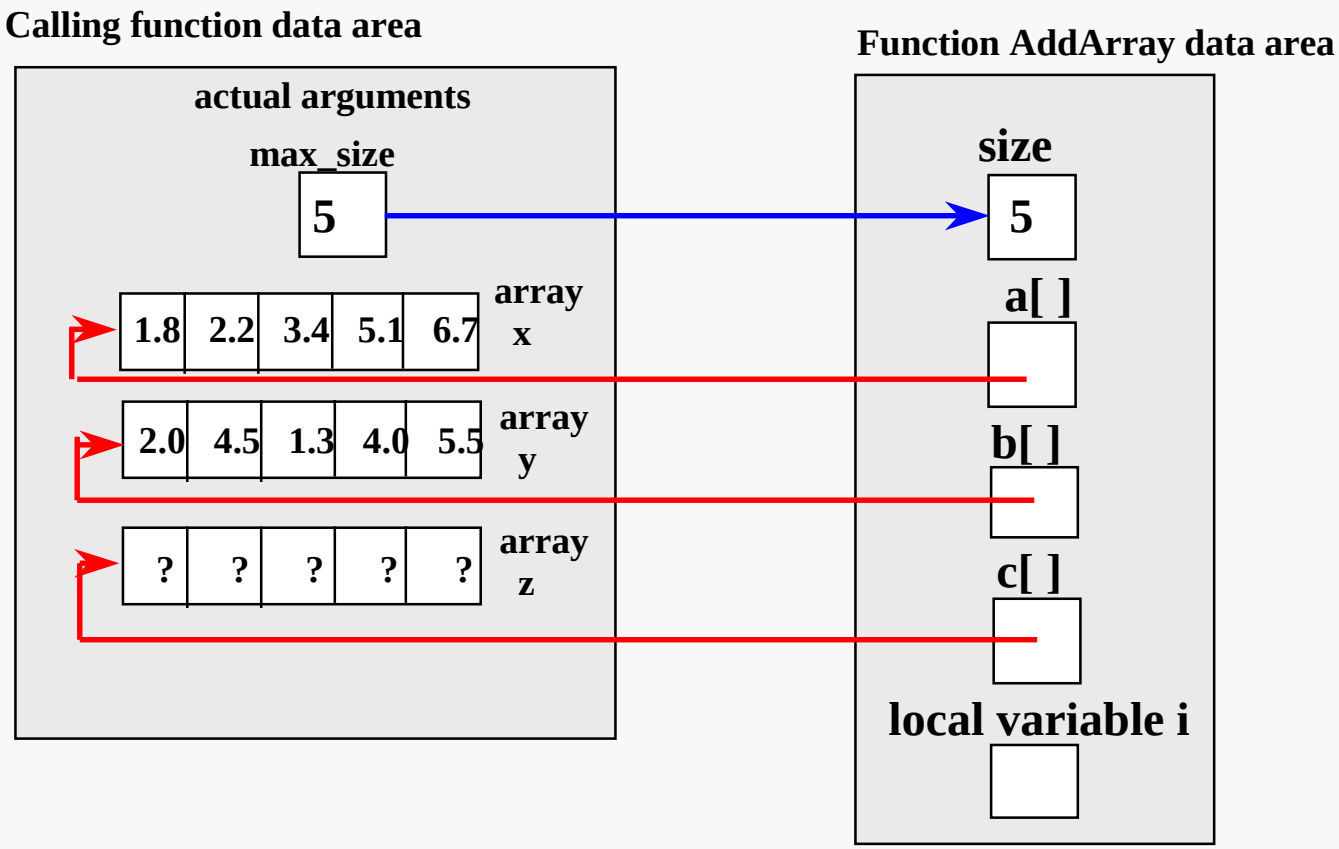
- Entire arrays can also be passed as parameters.
- By default arrays are automatically passed using pass-by-reference.
- Preceding a formal array parameter by the keyword `const` results in the array being passed by constant reference, the effect being that the actual array parameter cannot be modified by the called function.

```
// Array elements with subscripts ranging from 0 to size - 1
//      are summed element by element
// Pre:  a[i] and b[i] (0 <= i <= size -1) are defined
// Post: c[i] = a[i] + b[i] (0 <= i <= size - 1)
void AddArray(int size, const float a[], const float b[], float c[])
{
   for (int i = 0; i < size; i++)
        c[i] = a[i] + b[i];            // add and store result
   return;
}
```

Invoke AddArray as follows: `AddArray(MaxSize, X, Y, Z);`

Note well: when passing an entire array, the actual parameter is the array name, without an index.

# Arrays as Parameters



# Cautionary Note

The equality test code presented earlier can be incorporated into a function, as shown below. Note that there is no way for the function to determine whether the array indices are within logically correct bounds.

If the function is passed a value for `NumCells` that is larger than the dimension of either of the actual array parameters, then memory locations that exist outside the array boundaries will be compared. This type of logical error is extremely difficult to debug.

```
bool AreEqual(const int A[], const int B[], const int NumCells) {
  for (int index = 0; index < NumCells; index++) {
      if (A[index] != B[index])
         return false;
  }
  return true;
}
```

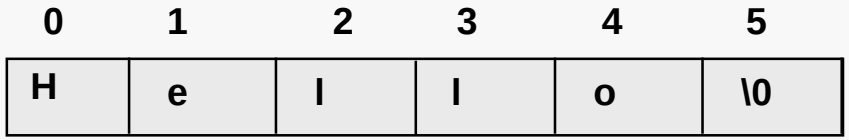
If global constants for the maximum array size has been defined then the assert() function can be used to check that the array accesses will be safe. For example the following could be placed at the beginning of the function:

```
assert ( (Numcells < MAX_ARRAY_A_SIZE) &&
         (Numcells < MAX_ARRAY_B_SIZE) ) ;
```

# C-style Character Strings

A string like "Hello" can be thought of as just an array of chars:

```
char String1[6] = “Hello”;
```




Same effect as: `char String1[6] = {‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’};`

The array contains the five characters *plus* a special string termination character called the NULL character, inserted automatically when the specified array initialization is done. Thus, the dimension of the char array must be one greater than the length of the (longest) string to be stored in the array.

It is legal to omit the array dimension, as shown below. In that case, the dimension is automatically set to the length of the initializing string literal plus 1:

```
char String2[] = “What’s your name?”;
```

# Character Strings: input

It is possible to use the extraction operator to read a character string into a char array:

```
char string2[20];

cin >> string2;   // Reads a string from keyboard into string2;
                  // there is no designation as to the size of the array;
                  // programmer's responsibility to insure that array is
                  // big enough - cin does not check
```

However, this is not the preferred technique; extraction operator string input:

- skips leading whitespace, then reads consecutive characters until the next whitespace character is encountered, which may not be appropriate.
- does append a NULL, which is good.
- does not consider the dimension of the target array, so it may store characters in memory past the end of the array, which is very bad.

See the succeeding slides on the family of get( ) functions for a better approach.

# Character Strings: output

```
const int MAXSTR = 20 ;
char string2[MAXSTR] = “Good morning!”;

cout << string2;   // Outputs string2 to the screen;
                   // there is no designation as to the size of the array;
                   // cout displays the characters until the NULL
                   // character is reached
```

```
const int MAXSTR = 20 ;
char string2[MAXSTR] = “Good morning!”;
int i = 0;
bool endofstring = false ;

endofstring = ( string2[i] == ‘\0’ );
while ( (i < MAXSTR) && (!endofstring) ) {
    cout << setw(1) << string2[i];  // print char
    i++ ;                           // count
    endofstring = ( string2[i] == ‘\0’ ); // check
}
```

Almost equivalent to the above code is the code to the left. Care must be taken to ensure that the code does not produce problems, such as garbage in the output stream or a binary output file, if the string is not terminated by the NULL character.

# strcat( ) function

```
void strcat(char dest[], char source[]); // see footnote†
```

concatenates the character string `source` onto the end of the character string `dest`, taking proper care of the NULLs

```
#include <iostream.h>
#include <string.h>            // for standard C-string functions

void main() {
   const int MAXLINE = 80 ;
   char a[MAXLINE] = "The quick brown fox";
   char b[MAXLINE] = " jumps over the lazy dog";
   strcat(a, b);
   cout << "a contains:  " << a << endl;
}
```

produces the output:

```
a contains: The quick brown fox jumps over the lazy dog
```

( † the return type of these functions aren't really void, but we're not ready to discuss that yet.)

# strlen( ) function

```
int strlen(char source[]);
```

returns the length of the character string `source`, excluding the terminating NULL character

```
#include <iostream.h>
#include <string.h>

void main() {
        const int MAXLINE = 80 ;
        char a[MAXLINE] = "The quick brown fox";
        int length;
        length = strlen(a);
        cout << "The following string has " << length <<
           " characters:" << endl;
        cout << a << endl;
}
```

produces the output:

```
The following string has 19 characters:
The quick brown fox
```

# strcpy( ) function

```
void strcpy(char dest[], char source[]);
```

copies the character string `source` into the character string `dest`

```
#include <iostream.h>
#include <string.h>

void main() {
        const int MAXLINE = 80 ;
        char a[MAXLINE] = "The quick brown fox";
        char b[MAXLINE] = " jumps over the lazy dog";
        strcpy(a, b);
        cout << a << endl;
}
```

produces the output:

```
 jumps over the lazy dog
```

# strcmp( ) function

```
int strcmp(char first[], char second[]);
```

compares the character strings `first` and `second` lexicographically, returning:

| | |
|---|---|
| < 0 | if first precedes second |
| 0 | if first and second are the same |
| > 0 | if first follows second |

The comparison is based on the ASCII code sequence, so upper-case letters precede lower-case letters.

# strcmp( ) function



```
#include <iostream.h>
#include <string.h>

void main() {
   char a[] = “Frederick”;
   char b[] = “Freddy”;
   int  k;
   k = strcmp(a, b);
   if (k < 0)
       cout << a << “ before ” << b << endl;
   else if (k == 0)
       cout << a << “ equals ” << b << endl;
   else // k > 0
       cout << b << “ before ” << a << endl;
}
```

produces the output: `Freddy before Frederick`

# Character Strings: input (get)

To input characters that contain whitespace either the get( ) or getline( ) functions must be used. The get function can be passed a character array and an integer parameter. Reading continues until the `int` parameter number of characters **- 1** has been read or the new line character is encountered. The get( ) function appends a NULL character to the end of the string.

```
ifstream inFile;
const int FILELENMAX = 51;
char fileName[FILELENMAX]; // 50 char name limit

cout << "Enter the input file name: ";
cin.get(fileName, FILELENMAX);
cin.ignore(100, '\n');

inFile.open(fileName, ios::nocreate);
if (!inFile) {
   cout << "** Can't open input file **" << endl;
  return EXIT_FAILURE;
}
```

Note: if a new line character is encountered get( ), does NOT remove the new line character from the input stream. The ignore( ) function can be used to explicitly remove the new line character.

# Character Strings: input (get)

There is yet another variation of the get( ) function which takes three parameters. The third parameter is a char, specifying the stop character (which defaults to a newline when you use the two-parameter get( ) function.

```
ifstream inFile;
const FILELENMAX = 51 ;
Const DEPTLENMAX = 10 ;
char Name[FILELENMAX];            // 50 char name limit
char Department[DEPTLENMAX];      //  9 char dept limit

cin.get(fileName, FILELENMAX);
cin.ignore(100, '\n');

inFile.open("employee.dat", ios::nocreate);
while (inFile) {
   inFile.get(Name, FILELENMAX, '\t');            // read to tab
   inFile.ignore(100, '\t');
   inFile.get(Department, DEPTLENMAX, ' ');       // read to space
   cout << Name << '\t' << Department << endl;
}
```

# Character Strings: input (getline)

The getline function is similar to the get function for reading a line of input. However the getline function automatically removes the next new line character from the input stream. It does not store the new line character in the string

```
ifstream inFile;
const FILELENMAX = 51 ;
char fileName[FILELENMAX]; // 50 char file name limit

cout << "Enter the input file name: ";
cin.getline(fileName, FILELENMAX);

inFile.open(fileName, ios::nocreate);
if (!inFile) {
   cout << "** Can't open input file **" << endl;
   return EXIT_FAILURE;
}
```

# Parallel Arrays

Arrays can contain values of only one type.
Parallel arrays can be used to handle related data of different types.
Two arrays can be considered parallel if their indexes and entries are logically related. For example, to store an inventory list containing a price and an availability code, two arrays can be declared as follows:

```
const int NUMPARTS = 500;

float Price[NUMPARTS];

char  Available[NUMPARTS];
```

| Index (Part #) | Price | Available |
|---|---|---|
| 0 | 1.27 | I |
| 1 | 4.50 | O |
| 2 | 0.37 | X |

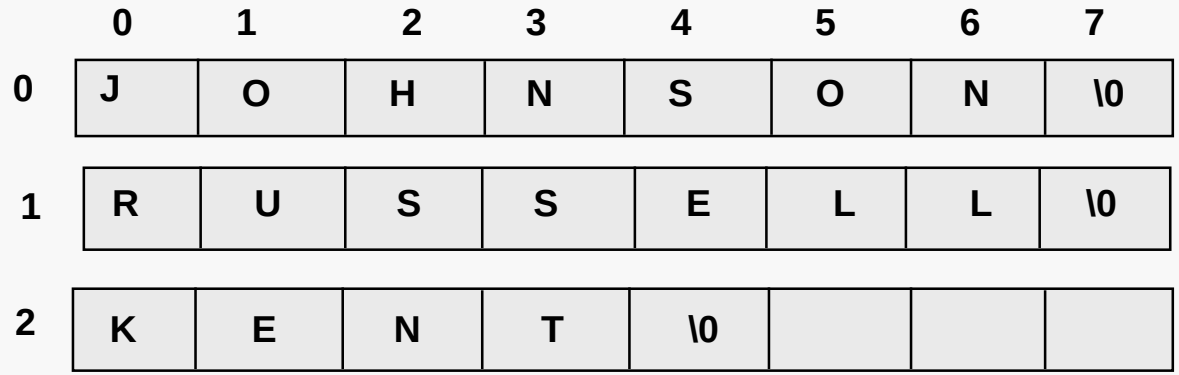
Assuming the arrays have been initialized, the statement below prints an inventory list. Note that Price[Part] and Available[Part] refer to the same logical item:

```
for (int Part = 0; Part < NumParts; Part++)
   cout << setw( 5) << Part
        << setw(10) << setprecision(2) << Price[Part]
        << setw( 5) << Available[Part] << endl;
```

# Array of Arrays

The component type of a C++ array may be of any type, even other arrays.

```
char Names [3][8] = {"JOHNSON",   // Names is a two dimensional
                     "RUSSELL",   //       table of characters
                     "KENT"};
```




`Names[2]` 3rd name in the list (Remember arrays are indexed from 0)
`Names[1][4]` 5th character of 2nd name

This declaration may be thought of in two ways. `Names` is a two-dimensional array of `chars`; `Names` is also a one-dimensional array of C-style character strings.

# Array of Arrays (cont)

```
int array1[2][3]  = { {1, 2, 3} , {4, 5, 6} };
int array2[2][3]  = { 1, 2, 3, 4, 5 };
int array3[2][3]  = { {1, 2} , {4 } };
```

If we printed these arrays by rows, we would find the following initializations had taken place:

Rows of array1:

```
1 2 3
4 5 6
```

Rows of array2:

```
1 2 3
4 5 0
```

Rows of array3:

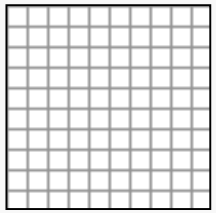
```
1 2 0
4 0 0
```

```
for (int row = 0; row < 2; row++) {
    for (int col = 0; col < 3; col++) {
        cout << setw(3)
             << array1[row][col];
    }
    cout << endl;
}
```

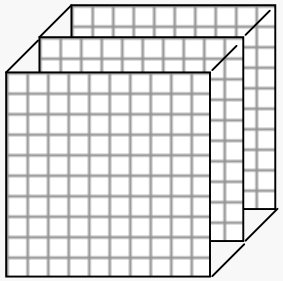
# Multi-Dimensional Arrays

An array can be declared with multiple dimensions.

2 Dimensional

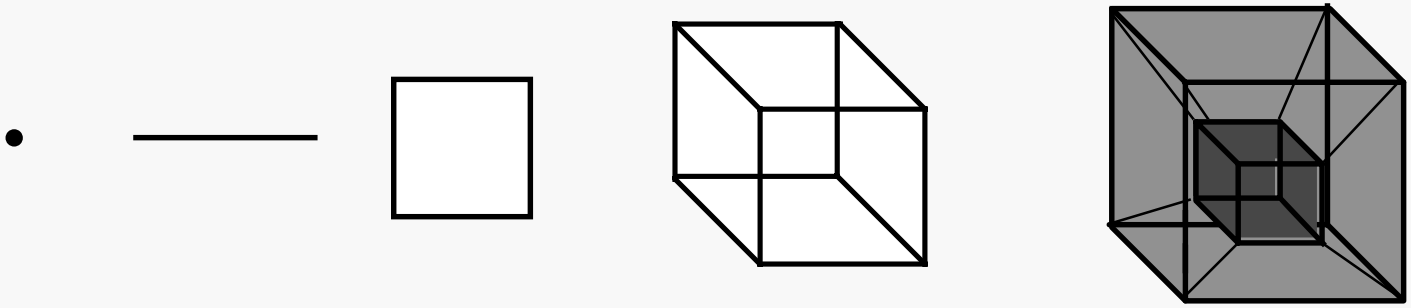


3 Dimensional



```
double Coord[100][100][100];
```

Multiple dimensions get difficult to visualize graphically.



# Multiple Dimensional Arrays as Parameters

When passing a two-dimensional array as a parameter, the base address is passed, as is the case with one-dimensional arrays.

But now the number of columns in the array parameter must be specified. This is because arrays are stored in row-major order, and the number of columns must be known in order to calculate the location at which each row begins in memory:

address of element (r, c) = base address of array
+ r*(number of elements in a row)*(size of an element)
+ c*(size of an element)

```
void someFunc(const int TwoD[][NUMCOLS], const int rows) {
    for (int i = 0; i < rows; i++) {
       for (int j = 0; j < NUMCOLS; j++)
          TwoD[i][j] = -1;
    }
}
```