The University Of Michigan

EECS402 Lecture 05

Andrew M. Morgan

Savitch Ch. 5
Arrays
Multi-Dimensional Arrays

Consider This Program

- Write a program to input 3 ints and output each value and their sum, formatted like a math problem

```
int i0;
int i1;
int i2;
int sum;

cout << "Enter int #1: ";
cin >> i0;
cout << "Enter int #2: ";
cin >> i1;
cout << "Enter int #3: ";
cin >> i2;

sum = i0 + i1 + i2;

cout << i0 << " + " << i1 << " + " << i2 << " = " << sum << endl;
```

```
Enter int #1: 54
Enter int #2: 102
Enter int #3: 7
54 + 102 + 7 = 163
```

Update To Program #1

EECS 402

- Your boss was so impressed, you are asked to update the program to work with 5 ints instead of 3

```
int i0;
int i1;
int i2;
int i3;
int i4;
int sum;

cout << "Enter int #1: ";
cin >> i0;
cout << "Enter int #2: ";
cin >> i1;
cout << "Enter int #3: ";
cin >> i2;
cout << "Enter int #4: ";
cin >> i3;
cout << "Enter int #5: ";
cin >> i4;

sum = i0 + i1 + i2 + i3 + i4;

cout << i0 << " + " << i1 << " + " << i2 << " + " <<
        i3 << " + " << i4 << " = " << sum << endl;
```

```
Enter int #1: 50
Enter int #2: 30
Enter int #3: 108
Enter int #4: 1215
Enter int #5: 74
50 + 30 + 108 + 1215 + 74 = 1477
```


Further Updates To Sum Program

EECS 402

- The previous programs worked fine and solved the problems that was presented
- Changing from 3 to 5 ints was easy-ish
  - lots of copy/paste operations
  - inevitably forget to update something along the way, resulting in need for debugging after the fact
- Now your boss asks for a program that works on 100 ints
  - Do you copy/paste 95 more inputs and outputs, update the variable names, and hope you did everything correctly?
- What if you are then requested to write one for 87 ints, and then 1000 ints, and then 743 ints, etc?

Intro To Arrays

- Array: A list of variables, *all of the same data type* that can be accessed via a common name
- The length of an array (the number of elements in the list) can be of any ***fixed*** length
  - That is, length indicated via a named constant or a literal value only!
- Syntax for declaring an array:
  - dataType arrayName[ARRAY_LENGTH];
    - dataType: Any available data type (int, float, user-defined types, etc)
    - arrayName: The name of the array (i.e. the common name used to access any variable in the list)
    - ARRAY_LENGTH: The number of elements that can be accessed via this array
- Example:
  - int quizGrades[10];
    - Declares an array of 10 integer elements, with the name "quizGrades"

More Info On Arrays

- Individual elements of the array are accessed by "indexing"
  - To index into an array, use the square brackets
  - In C/C++ array indices start at 0, and end at (length – 1)
  - Example: quizGrades[4] accesses the fifth element of the array
    - [0] would be the first, [1] the second, [2] the third, [3] the fourth, [4] the fifth
  - "quizGrades" is an array of ints, but "quizGrades[4]" is an int, and can be used anywhere an int variable can be used
- If an int variable requires 4 bytes of memory, then the declaration:
  - int quizGrades[10];
  - sets aside 40 bytes (10 ints at 4 bytes each) of memory
  - Elements can be accessed using the following:
    - quizGrades[0], quizGrades[1], quizGrades[2], quizGrades[3], quizGrades[4], quizGrades[5], quizGrades[6], quizGrades[7], quizGrades[8], quizGrades[9]

A Quick Side Topic: Built-in Type Sizes

- Different kinds of data requires different amounts of memory to store it
  - We can store a character like 'T' or '+' in one byte
  - A large number such as 58,461,832 cannot possibly be represented in one byte though
  - If the amount of memory used for a piece of data depended on its *value*, the runtime environment would have a LOT more work to do
- All built-in datatypes are a fixed size
  - Most commonly:
    - char: 1 byte (integer values -128 to +127)
    - int: 4 bytes (allows a range of ~4.2 billion (i.e. -2.1 billion to 2.1 billion))
    - float: 4 bytes
    - double: 8 bytes
    - bool: 1 byte

Arrays Stored In Memory

- Array elements are ***always*** stored in contiguous memory locations
  - This means the value arrayVar[4] is stored immediately following arrayVar[3] in memory
- Say you have the following array declarations:

```
char cAry[4];
int  iAry[4];
```

How much memory is set aside for cAry?

Memory Address	Memory Contents	Memory Address	Memory Contents
1000		1012	
1001		1013	
1002		1014	
1003		1015	
1004		1016	
1005		1017	
1006		1018	
1007		1019	
1008		1020	
1009		1021	
1010		1022	
1011		1023	

Arrays Stored In Memory

EECS 402

- Array elements are ***always*** stored in contiguous memory locations
  - This means the value arrayVar[4] is stored immediately following arrayVar[3] in memory
- Say you have the following array declarations:

```
char cAry[4];
int  iAry[4];
```

How much memory is set aside for cAry?
4 chars * 1 byte each = 4 bytes

Let's say cAry[0] gets placed at memory address 1000. Where is cAry[1] located?

Memory Address	Memory Contents	Memory Address	Memory Contents
1000	cAry[0]	1012	
1001		1013	
1002		1014	
1003		1015	
1004		1016	
1005		1017	
1006		1018	
1007		1019	
1008		1020	
1009		1021	
1010		1022	
1011		1023	

EECS 402

9

Arrays Stored In Memory

EECS 402

- Array elements are ***always*** stored in contiguous memory locations
  - This means the value arrayVar[4] is stored immediately following arrayVar[3] in memory
- Say you have the following array declarations:

```
char cAry[4];
int  iAry[4];
```

How much memory is set aside for cAry?
4 chars * 1 byte each = 4 bytes

Let's say cAry[0] gets placed at memory address 1000. Where is cAry[1] located?
1 char's worth of memory after cAry[0]'s starting address
1000 + 1 byte = 1001

Memory Address	Memory Contents	Memory Address	Memory Contents
1000	cAry[0]	1012	
1001	cAry[1]	1013	
1002	cAry[2]	1014	
1003	cAry[3]	1015	
1004		1016	
1005		1017	
1006		1018	
1007		1019	
1008		1020	
1009		1021	
1010		1022	
1011		1023	

EECS 402

10

Arrays Stored In Memory

EECS 402

- Array elements are ***always*** stored in contiguous memory locations
  - This means the value arrayVar[4] is stored immediately following arrayVar[3] in memory
- Say you have the following array declarations:

```
char cAry[4];
int  iAry[4];
```

Where will iAry[0] be stored?

Memory Address	Memory Contents	Memory Address	Memory Contents
1000	cAry[0]	1012	
1001	cAry[1]	1013	
1002	cAry[2]	1014	
1003	cAry[3]	1015	
1004		1016	
1005		1017	
1006		1018	
1007		1019	
1008		1020	
1009		1021	
1010		1022	
1011		1023	

EECS 402

11

Arrays Stored In Memory

EECS 402

- Array elements are ***always*** stored in contiguous memory locations
  - This means the value arrayVar[4] is stored immediately following arrayVar[3] in memory
- Say you have the following array declarations:

```
char cAry[4];
int  iAry[4];
```

Where will iAry[0] be stored?
    Trick question – no way to know!
    Probably at 1004, but….

Memory Address	Memory Contents	Memory Address	Memory Contents
1000	cAry[0]	1012	
1001	cAry[1]	1013	
1002	cAry[2]	1014	
1003	cAry[3]	1015	
1004	iAry[0]	1016	
1005		1017	
1006		1018	
1007		1019	
1008		1020	
1009		1021	
1010		1022	
1011		1023	

EECS 402

12

Arrays Stored In Memory

EECS 402

- Array elements are ***always*** stored in contiguous memory locations
  - This means the value arrayVar[4] is stored immediately following arrayVar[3] in memory
- Say you have the following array declarations:

```
char cAry[4];
int  iAry[4];
```

Where will iAry[0] be stored?
   Trick question – no way to know!
   Probably at 1004, but….

Where will iAry[1] be stored?

Memory Address	Memory Contents	Memory Address	Memory Contents
1000	cAry[0]	1012	
1001	cAry[1]	1013	
1002	cAry[2]	1014	
1003	cAry[3]	1015	
1004	iAry[0]	1016	
1005		1017	
1006		1018	
1007		1019	
1008		1020	
1009		1021	
1010		1022	
1011		1023	

EECS 402

13

Arrays Stored In Memory

EECS 402

- Array elements are ***always*** stored in contiguous memory locations
  - This means the value arrayVar[4] is stored immediately following arrayVar[3] in memory
- Say you have the following array declarations:

```
char cAry[4];
int  iAry[4];
```

Where will iAry[0] be stored?
   Trick question – no way to know!
   Probably at 1004, but….

Where will iAry[1] be stored?
   1 int's worth of memory after iAry[0]'s starting address
   1004 + 4 bytes = 1008

And iAry[2] will be stored at 2 ints worth of memory after iAry[0]'s starting address
   1004 + 4 bytes * 2 = 1012

Memory Address	Memory Contents	Memory Address	Memory Contents
1000	cAry[0]	1012	iAry[2]
1001	cAry[1]	1013	
1002	cAry[2]	1014	
1003	cAry[3]	1015	
1004	iAry[0]	1016	iAry[3]
1005		1017	
1006		1018	
1007		1019	
1008	iAry[1]	1020	
1009		1021	
1010		1022	
1011		1023	

EECS 402

14

Arrays Stored In Memory

EECS 402

- Array elements are ***always*** stored in contiguous memory locations
  - This is what makes arrays so powerful!
  - Any individual element can be accessed very quickly
    - Knowledge of the element size and the memory address of the first element is all that is needed to determine the location of any element
    - ElementAddress = ArrayStartAddress + (Index * sizeOfArrayElement)

Assume that chars require 1 bytes of memory and ints require 4 bytes of memory.
The following declarations could result in the following layout of memory

```
char cAry[4];
int  iAry[4];
```

When you access cAry[2], address is computed:

```
1000 + 2 * 1 = 1002
```

When you access iAry[3], address is computed:

```
1004 + 3 * 4 = 1016
```

StartAddress Index ElemSize ElemAddress

1000	cAry[0]	1012	iAry[2]
1001	cAry[1]	1013	
1002	cAry[2]	1014	
1003	cAry[3]	1015	
1004	iAry[0]	1016	iAry[3]
1005		1017	
1006		1018	
1007		1019	
1008	iAry[1]	1020	
1009		1021	
1010		1022	
1011		1023	

EECS 402

15

Using An Array For The Sum Program

EECS 402

- The sum program can be rewritten using a single array

```
int i;
int valsToSum[3];
int sum = 0;
for (i = 0; i < 3; i++)
{
  cout << "Enter int #" << i + 1 << ": ";
  cin >> valsToSum[i];
}

cout << valsToSum[0];
for (i = 1; i < 3; i++)
{
  cout << " + " << valsToSum[i];
}
cout << " = ";
for (i = 0; i < 3; i++)
{
  sum += valsToSum[i];
}
cout << sum << endl;
```

```
Enter int #1: 45
Enter int #2: 109
Enter int #3: 13
45 + 109 + 13 = 167
```


16

Extending To Sum Five Ints

- No copy/paste is required this time, just a few minor changes

```
int i;
int valsToSum[5];
int sum = 0;
for (i = 0; i < 5; i++)
{
  cout << "Enter int #" << i + 1 << ": ";
  cin >> valsToSum[i];
}

cout << valsToSum[0];
for (i = 1; i < 5; i++)
{
  cout << " + " << valsToSum[i];
}
cout << " = ";
for (i = 0; i < 5; i++)
{
  sum += valsToSum[i];
}
cout << sum << endl;
```

```
Enter int #1: 4
Enter int #2: 14
Enter int #3: 20
Enter int #4: 7
Enter int #5: 1
4 + 14 + 20 + 7 + 1 = 46
```


Even Better Version Of Sum Program

- Using a named constant for the array size allows for even easier updates

```
const int ARRAY_LENGTH = 3;
int i;
int valsToSum[ARRAY_LENGTH];
int sum = 0;
for (i = 0; i < ARRAY_LENGTH; i++)
{
  cout << "Enter int #" << i + 1 << ": ";
  cin >> valsToSum[i];
}

cout << valsToSum[0];
for (i = 1; i < ARRAY_LENGTH; i++)
{
  cout << " + " << valsToSum[i];
}
cout << " = ";
for (i = 0; i < ARRAY_LENGTH; i++)
{
  sum += valsToSum[i];
}
cout << sum << endl;
```

```
Enter int #1: 86
Enter int #2: 42
Enter int #3: 13
86 + 42 + 13 = 141
```

Extending To Five, One More Time

```
const int ARRAY_LENGTH = 5;
int i;
int valsToSum[ARRAY_LENGTH];
int sum = 0;
for (i = 0; i < ARRAY_LENGTH; i++)
{
  cout << "Enter int #" << i + 1 << ": ";
  cin >> valsToSum[i];
}

cout << valsToSum[0];
for (i = 1; i < ARRAY_LENGTH; i++)
{
  cout << " + " << valsToSum[i];
}
cout << " = ";
for (i = 0; i < ARRAY_LENGTH; i++)
{
  sum += valsToSum[i];
}
cout << sum << endl;
```

One simple change needed to support *any* number of elements

```
Enter int #1: 32
Enter int #2: 14
Enter int #3: 75
Enter int #4: 10
Enter int #5: 6
32 + 14 + 75 + 10 + 6 = 137
```


Array Bounds

- C/C++ does not do any bounds checking for you
  - Assumption is that programmer knows what he/she is doing
  - Formula for computing element address is used, even if index value is out-of-bounds for the array
  - The following example does not give any compile-time warnings or errors, nor does it give any run-time errors!

```
int main()
{
  int i;
  int ary[4];
  int var=0;
  for (i = 1; i <= 4; i++)
  {
    cout << "Enter int #" << i << ": ";
    cin >> ary[i];
  }
  cout << "Var: " << var << endl;
  return 0;
}
```

(Possible Results)

```
Enter int #1: 6
Enter int #2: 4
Enter int #3: 3
Enter int #4: 2
Var: 2
```

Explanation Of Previous Program

- Note: var was initialized to 0, was never changed in the program, but gets printed as 2 in the example output

```
int main()
{
  int i;
  int ary[4];
  int var=0;
  for (i = 1; i <= 4; i++)
  {
    cout << "Enter int #" << i << ": ";
    cin >> ary[i];
  }
  cout << "Var: " << var << endl;
  return 0;
}
```

1000	i	1012	ary[2]
1001		1013	
1002		1014	
1003		1015	
1004	ary[0]	1016	ary[3]
1005		1017	
1006		1018	
1007		1019	
1008	ary[1]	1020	var
1009		1021	
1010		1022	
1011		1023	

When i == 4, ary[i] address is computed as:

`1004 + 4 * 4 = 1020`

StartAddress, Index, ElemSize, ElemAddress

More On Array Bounds

- Why doesn't C/C++ do range checking for you?
  - Efficiency
    - Arrays are used a lot in programming
    - If every time an array was indexed, the computer had to do array bounds checking, things would be very slow
- In the previous example, programmer was only "off-by-one"
  - This is a very common bug in programs, and is not always as obvious as the previous example
  - In this case, the variable "var" was stored in that location and was modified
- What happens if the index is off far enough such that the memory address computed does not belong to the program?
  - Segmentation Fault

Segmentation Faults

- Segmentation faults (a.k.a. "seg faults") occur when your program tries to access a memory location that is does not have access to

```
int main()
{
  int ary[4];
  ary[0] = 10;
  cout << "Set ary[0]" << endl;
  ary[3] = 20;
  cout << "Set ary[3]" << endl;
  ary[9] = 30;
  cout << "Set ary[9]" << endl;
  ary[185] = 40;
  cout << "Set ary[185]" << endl;
  ary[600] = 50;
  cout << "Set ary[600]" << endl;
  ary[900] = 60;
  cout << "Set ary[900]" << endl;
  return 0;
}
```

(Possible Results)

```
Set ary[0]
Set ary[3]
Set ary[9]
Set ary[185]
Segmentation fault
```


Segmentation Faults, Cot'd

- A seg fault is considered to be a "crash" of the program
  - Program crashes are unacceptable and need to be prevented
- Just because the program didn't seg fault, does not mean there were no bounds problems
  - In the previous program, array was indexed using values 9 and 185, which are both out-of-bounds, without seg faulting

Memory address calculations

ary[9]: `1000 +   9 * 4 = 1036`

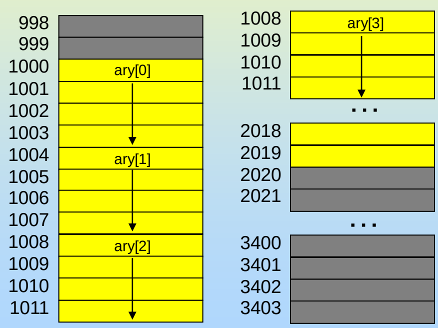
ary[185]: `1000 + 185 * 4 = 1740`

ary[600]: `1000 + 600 * 4 = 3400`

Memory location belonging to your program

Memory location belonging to a different program

Seg fault occurs when trying to access memory that does not belong to your program



Initializing Array Values

- Array values can be initialized at the time they are declared
  - Assigned to comma-separated list of values enclosed in curly braces
  - If array length is unspecified, it is assumed to be exact size to fit initial values

```
int oddary[5] = {1, 3, 5, 7, 9}; //These two are
int oddary2[] = {1, 3, 5, 7, 9}; //equivalent..
```

  - If length is specified, but not enough initial values are provided, extra values are initialized to zero

```
int zeroAry[100] = {0};   //100 zeros!
int careful[100] = {100}; //100 followed by 99 zeros!
```

  - Use a loop to assign all elements to a specific value

```
int aryOf100s[100];       //uninitialized
for (i = 0; i < 100; i++)
{
  aryOf100s[i] = 100;     //elements assigned here to 100
}
```


Array Elements As Parameters

- Given the following array declaration:
  - int ary[5];
  - Indexing into the array results in an int
  - This int can be used anywhere an int can be used

```
void printInt(int val)
{
  cout << "Int is: " << val << endl;
}

int main()
{
  int iary[5] = {3, 5, 7, 9, 11};

  printInt(iary[3]);
  printInt(iary[4]);
  return 0;
}
```

Int is: 9
Int is: 11

Whole Arrays As Parameters

EECS 402

- Entire array can be passed into a function
- Example: Write a function that returns the sum of all values in an array
  - Specifying array length in parameter is optional, and usually not included

```
int sumAry(
     int num,  //# of elems in ary
     int ary[] //array of vals to sum
     )
{
  int sum = 0;
  int i;

  for (i = 0; i < num; i++)
  {
    sum += ary[i];
  }
  return sum;
}
```

```
int main()
{
  int iary[5]={3, 5, 7, 9, 11};
  int x;

  x = sumAry(5, iary);
  cout << "Sum: " << x << endl;

  return 0;
}
```

Sum: 35

Arrays Passed By Reference

EECS 402

- Arrays are passed by reference *by default*
  - No special syntax (i.e. no '&') is required to pass arrays by reference
- Why?
  - Pass-by-value implies a copy is made
  - If arrays were passed-by-value, every element of the entire array would have to be copied
    - For large arrays especially, this would be extremely slow
    - Also uses a lot of memory to duplicate the array
- Changing contents of an array inside a function changes the array as stored in the calling function as well!

Arrays Passed By Reference, Example

EECS 402

```
void sumArys(int num, int a[],
             int b[], int c[])
{
  int i;

  for (i = 0; i < num; i++)
  {
    c[i] = a[i] + b[i];
  }
}
```

```
int main()
{
  int i;
  int iary1[5] = {3, 5, 7, 9, 11};
  int iary2[5] = {2, 4, 6, 8, 10};
  int iary3[5]; //Uninitialized

  sumArys(5, iary1, iary2, iary3);

  for (i = 0; i < 5; i++)
  {
    cout << "iary3[" << i << "]: "
         << iary3[i] << endl;
  }

  return 0;
}
```

```
iary3[0]: 5
iary3[1]: 9
iary3[2]: 13
iary3[3]: 17
iary3[4]: 21
```

Changing "c" array in sumArys changes "iary3" in main, since arrays are passed by reference by default

Arrays As Parameters, Cot'd

EECS 402

- If you want to prevent array contents from changing in a function, use keyword "const"
  - Results in array being passed by "constant reference"
  - Array is still passed by reference – no inefficient copy is made
  - Keyword const prevents contents from being modified in the function
- Why bother?
  - To protect yourself from making mistakes
  - What would output of previous program be if sumArys was as follows:

```
void sumArys(int num, int a[],
             int b[], int c[])
{
  int i;
  for (i = 0; i < num; i++)
  {
    a[i] = b[i] + c[i];
  }
}
```

(Possible Results)

```
iary3[0]: 12
iary3[1]: -13377364
iary3[2]: -13392368
iary3[3]: 0
iary3[4]: 0
```

Updated sumArys

- Keyword const used for array parameters that won't change

Arrays "a" and "b" can not be changed within sumArys, only "c" can

```
void sumArys(
      int num,
      const int a[],
      const int b[],
      int c[]
      )
{
  int i;

  for (i = 0; i < num; i++)
  {
    c[i] = a[i] + b[i];
  }
}
```

This version provides a compile-time error, preventing problems resulting from this mistake

```
void sumArys(
      int num,
      const int a[],
      const int b[],
      int c[]
      )
{
  int i;

  for (i = 0; i < num; i++)
  {
    a[i] = b[i] + c[i];
  }
}
```


Using Variables As Array Sizes

- Array sizes must be specified as:
  - Named Constants: NUM_QUIZZES, TOTAL_STUDENTS, etc
  - Literal Values: 10, 62, etc
- Array sizes can ***not*** be variable!
- The following program *should not* compile, and will NOT be allowed in this course!

```
//This is an invalid program!!
int main()
{
  int num;

  cout << "Enter length of array: ";
  cin >> num;
  int iary[num]; //num is not constant!!!

  return 0;
}
```

Note: Adding the "-pedantic" flag to the g++ command line will ensure this is noticed
→ some g++ "extensions" that are often default ON will allow this unacceptable code to compile)
→ use "-pedantic" to make sure your code is standard compliant

```
prompt&% g++ -Wall -std=c++98 -pedantic -Werror arysize.cpp
arysize.cpp: In function 'int main()':
arysize.cpp:11:15: error: ISO C++ forbids variable length array 'iary' [-Werror=vla]
   int iary[num]; //num is not constant!!!
               ^
cc1plus: all warnings being treated as errors
```

One Problem - What Is The Output?

```
int main(void)
{
  const int SIZE = 5;
  int i;
  int iary[SIZE] = {2,4,6,8,10};

  while (i < SIZE)
  {
    cout << iary[i] << endl;
    i++;
  }

  return 0;
}
```


Another Problem – What Is The Output?

```
int main(void)
{
  const int SIZE = 5;
  int i = 0;
  int iary[SIZE] = {2,4,6,8,10};

  while (i < SIZE)
  {
    cout << iary[i] << endl;
    iary[i]++;
  }

  return 0;
}
```

Two-Dimensional Arrays

- Arrays are not limited to one dimension
- A 2-D array is often called a matrix

Dimension 2 (columns)

Dimension 1 (rows)

	0	1	2	3	4	5
0	1	2	3	4	5	6
1	2	4	6	8	10	12
2	3	6	9	12	15	18
3	4	8	12	16	20	24
4	5	10	15	20	25	30
5	6	12	18	24	30	36

Declaring a 2-D Array

- Syntax for a 2-D array is similar to a 1-D
  - dataType arrayName[numRows][numCols];
- While there are 2 dimensions, each element must still be of the same data type
- To declare matrix shown on previous slide (6 rows, 6 columns)
  - int matrix[6][6];
- If ints are stored in 4 bytes, then the above declaration sets aside 6 * 6 *4 = 144 bytes of memory
- A 2-D array is really just a 1-D array, where each individual element is itself a 1-D array

Initializing a 2-D Array

- Initialization of 1-D array was a comma separated list of values enclosed in curly braces
- 2-D array initialization is an initialization of a 1-D array of 1-D arrays

1-D Array

```
int matrix[6][6] = {{ 1, 2, 3, 4, 5, 6},
                    { 2, 4, 6, 8,10,12},
                    { 3, 6, 9,12,15,18},
                    { 4, 8,12,16,20,24},
                    { 5,10,15,20,25,30},
                    { 6,12,18,24,30,36}};
```


Assigning 2-D Array Elements

- Individual elements can be assigned using two sets of brackets
  - The following code creates a matrix with the same values as shown earlier, but uses a mathematical formula instead of initialization

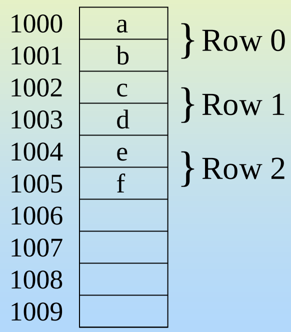
```
int matrix[6][6];

for (i = 0; i < 6; i++)
{
  for (j = 0; j < 6; j++)
  {
    matrix[i][j] = (i + 1) * (j + 1);
  }
}
```

2-D Arrays In Memory

EECS 402

- As with ***all*** arrays, 2-D arrays are stored contiguously in memory
- Computer memory is inherently 1-dimensional though
  - Row 2's elements are stored immediately following the last element from row 1

```
char cary[3][2] = {{'a','b'},{'c','d'},{'e','f'}};
```



2-D Array Memory Address Determination

EECS 402

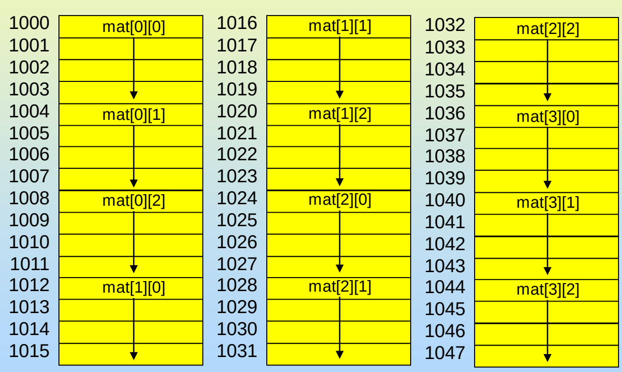
- As with 1-D arrays, any element's address can be computed very quickly
  - Knowledge of array starting address, size of each element, ***and the number of columns in each row*** is required

```
int mat[4][3];
```

mat[1][2]
1000 + (3 * 1) * 4 + 2 * 4 = 1020

mat[3][1]
1000 + (3 * 3) * 4 + 1 * 4 = 1040

StartAddr #Columns RowIndex ElementSize ColumnIndex ElementSize ElementAddr



Single 2-D Array Elements

EECS 402

- Use single element anywhere variable of that type can be used

```
void printInt(int i)
{
  cout << "Int is: "  << i << endl;
}

int main()
{
  int matrix[3][2] = {{2,4},{6,8},{10,12}};

  printInt(matrix[1][0]);
  printInt(matrix[2][1]);

  return 0;
}
```

```
Int is: 6
Int is: 12
```


Using Rows Of 2-D Arrays

EECS 402

- Since a 2-D array is a 1-D array of 1-Darrays, each row can be used a 1-D array

```
int sumAry(
     int num,   //# of elems in ary
     const int ary[] //array of vals
                     //to sum
     )
{
  int sum = 0;
  int i;

  for (i = 0; i < num; i++)
  {
    sum += ary[i];
  }

  return sum;
}
```

```
int main()
{
  int matr[3][5]={{3, 5, 7, 9, 11},
                  {2, 4, 6, 8, 10},
                  {1, 2, 3, 4,  5}};
  int x;

  x = sumAry(5, matr[0]);
  cout << "Row1 Sum: " << x << endl;
  x = sumAry(5, matr[1]);
  cout << "Row2 Sum: " << x << endl;

  return 0;
}
```

```
Row1 Sum: 35
Row2 Sum: 30
```

Whole 2-D Arrays As Parameters

- When passing a 2-D array as a parameter, function must know the number of columns
  - Recall this is required in order to compute address of an element
  - First dimension can be left unspecified, second dimension ***can not!***

```
void printAry(int rows,
              const char ary[][2])
{
  int i;
  int j;
  for (i = 0; i < rows; i++)
  {
    for (j = 0; j < 2; j++)
    {
      cout << ary[i][j] << " ";
    }
    cout << endl;
  }
}
```

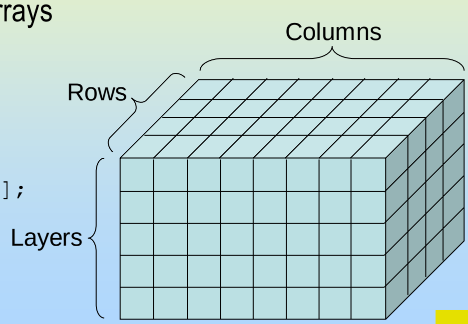
```
int main()
{
  char matrix[3][2] =
                   {{'a','b'},
                    {'c','d'},
                    {'e','f'}};
  printAry(3, matrix);
  return 0;
}
```

```
a b
c d
e f
```


Three-Dimensional Arrays

- C++ arrays can have as many dimensions as necessary
- 1-D Array: List of values
- 2-D Array: Matrix of values, consisting of rows and columns
- 3-D Array: Cube of values, consisting of rows, columns, and layers
  - In other words, a 3-D array is a 1-D array of matrices
  - In other words, a 3-D array is a 1-D array, where each element is a 1-D array whose elements are 1-D arrays
  - Got that?

```
const int LAYERS = 5;
const int ROWS = 4;
const int COLS = 8;
int threeDAry[LAYERS][ROWS][COLS];
```



A Quick 3D Array Example

```
void printAry(
     int num,
     const char ary[][4])
{
  int i;
  int j;
  for (i = 0; i < num; i++)
  {
    for (j = 0; j < 4; j++)
    {
      cout << ary[i][j] << " ";
    }
    cout << endl;
  }
}
```

```
int main(void)
{
  char cary [2][3][4] =
      { { {'a','b','c','d'},
          {'e','f','g','h'},
          {'i','j','k','l'} },
        { {'m','n','o','p'},
          {'q','r','s','t'},
          {'u','v','w','x'} } };
  cout << "cary [0]: " << endl;
  printAry(3, cary [0]);
  cout << "cary [1]: " << endl;
  printAry(3, cary [1]);

  return 0;
}
```

```
cary [0]:
a b c d
e f g h
i j k l
cary [1]:
m n o p
q r s t
u v w x
```