CS101 Introduction to computing

Array and Pointer

A. Sahu and A. Sahu

Dept of Comp. Sc. & Engg.

Indian Institute of Technology Guwahati

Outline

- Array Definition, Declaration, Use
- Array Examples
- Pointer
  - Memory access
  - Access using pointer
- Basic Pointer Arithmetic

Objectives

- Be able to use
  - arrays, pointers, and strings in C programs
- Be able to explain the
  - Representation of these data types at the machine level including their similarities and differences

Definition – *Array*

- A collection of objects of the *same type* stored contiguously in memory under one name
  - May be type of any kind of variable
  - May even be collection of arrays!
- For ease of access to any member of array
- Can be think as a group

Examples

- **`int A[10]`**
  - An array of ten integers
  - **`A[0]`**, **`A[1]`**, …, **`A[9]`**
- **`double B[20]`**
  - An array of twenty long floating point numbers
  - **`B[0]`**, **`B[1]`**, …, **`B[19]`**
- **Array indexes *always* start at zero in *C***

Examples

- **`int D[10][20]`**
  - An array of ten rows, each of which is an array of twenty integers
  - **`D[0][0]`**, **`D[0][1]`**, …, **`D[1][0]`**, **`D[1][1]`**, …, **`D[9][19]`**
  - Not used so often as arrays of pointers

Array Element

- May be used wherever a variable of the same type may be used
  - In an expression (including arguments)
  - On left side of assignment
- Examples:–

```
A[3] = x + y;
x = y – A[3];
z = sin(A[i]) + cos(B[j]);
```

Array Elements

- Generic form:–
  - *ArrayName*[*integer-expression*]
  - *ArrayName*[*integer-expression*] [*integer-expression*]
  - Same type as the underlying type of the array
- Definition:– *Array Index* – the expression between the square brackets

Array Elements

- Array elements are commonly used in loops
- E.g.,

```
for(i=0; i < max; i++)
    A[i] = i*i;
```

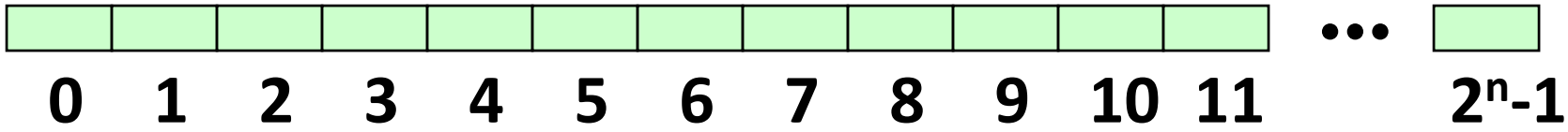
```
sum = 0;
for(j=0;j<max;j++) sum += B[j];
```

```
int count,sum = 0;
for (count=0; count<30; count++){
     scanf("%f", &A[count]);
     Sum += A[count];
}
```

Array: Initialization and Accessing

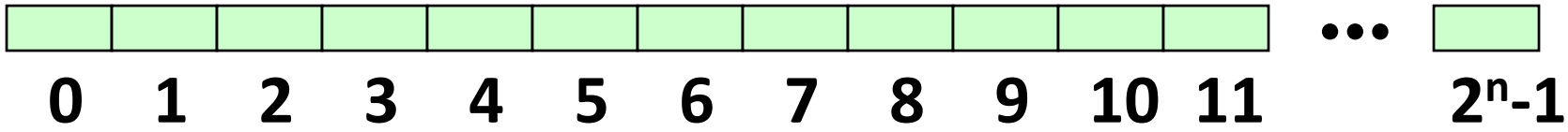
```
int A[5], i; //defining
//initializing
for (i=0; i<5; i++)
    A[i]=i;
//accessing the array
for (i=0; i<5; i++)
    printf("%d, ", A[i]);
```

Memory Organization



- All modern processors have memories organized as sequence of *numbered bytes*
  - Many (but not all) are linear sequences
- Definitions:–
  - *Byte:* an 8-bit memory cell capable of storing a value in range 0 … 255
  - *Address:* number by which a memory cell is identified

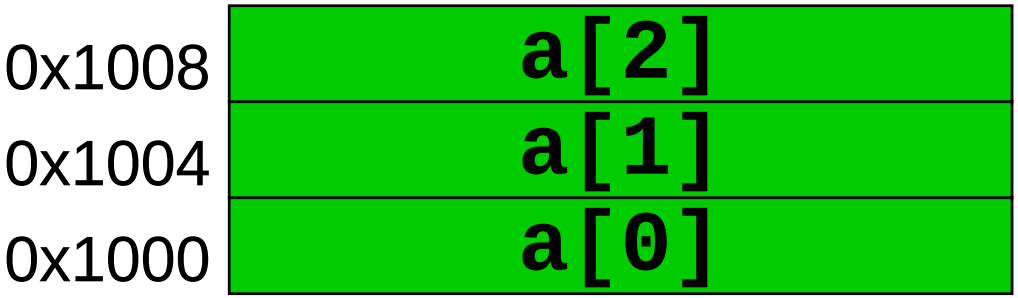
Memory Organization (continued)



- Larger data types are sequences of bytes
  - **`short int`** – 2 B, **`int`** – 4 B, **`long`** –8 B
  - **`float`** – 4 B, **`double`** – 8 B
- (Almost) always aligned to multiple of size in bytes
- Address is "first" byte of sequence
  - May be low-order or high-order byte
  - *Big endian* or *Little endian*

Array Representation

- Homogeneous: Each element same size –**s** bytes
  - An array of m data is a sequence of **m**×**s** bytes
  - Indexing: 0^{th} data at byte s×0, 1^{st} data at byte s×1, …
- **m** and **s** are not part of representation
  - s known by compiler – usually irrelevant to programmer
  - m often known by compiler – if not, must be saved by programmer




```
int a[3];
```

Array Sizes

```
int  A[10];
```

- Size of object/data : returns the size of an object in bytes
- What is `sizeof(A[3])`? 4
- What is `sizeof(A)`? 40

```
int A[10];
printf("%d, %d, %d ",
     sizeof(int), sizeof(A[3]),
     sizeof(A));
//outputs 4, 4, 40
```

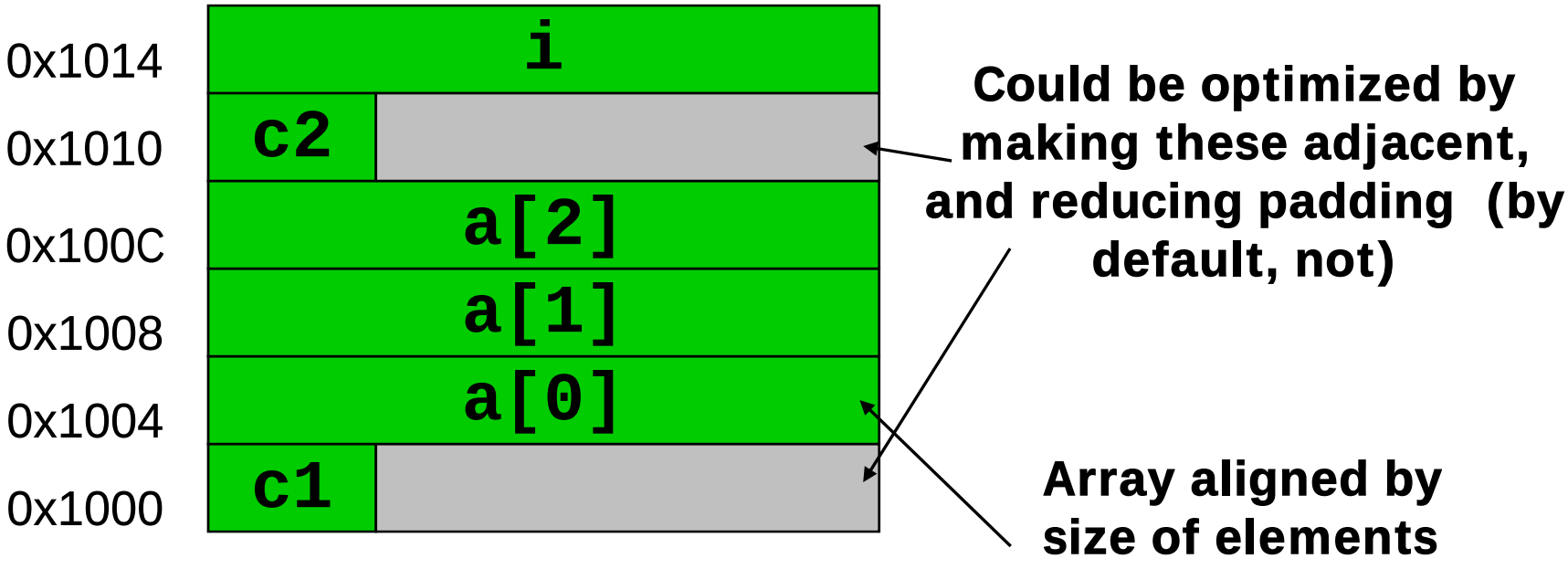
Array: Address, Element Address

- Scanf require element address
- Reading values from KBD and storing in array element

```
int A[10];
int Sum = 0;
for (count=0;count<10;count++){
     scanf("%f", &A[count]);
     Sum += A[count];
}
```

Array Representation

```
char    c1;
int     a[3];
char    c2;
int     i;
```



Arrays in C

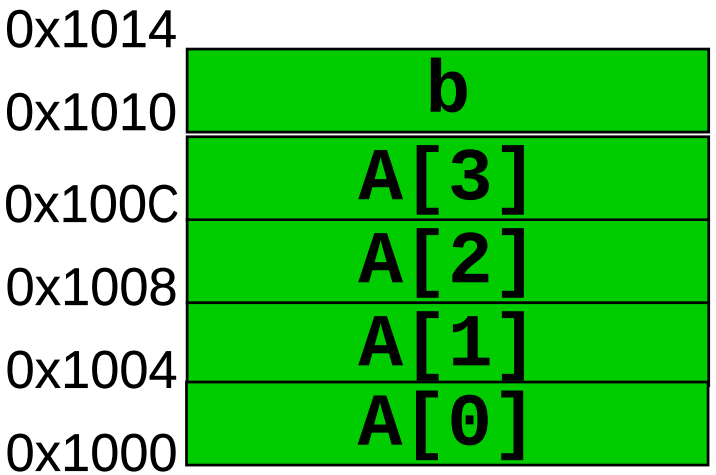
```
int A[4];
int b;


A[0]   = 3;
A[3]   = 4;
```

All elements of same type – homogenous

array size in declaration

First element (index 0)

Last element (index size - 1)



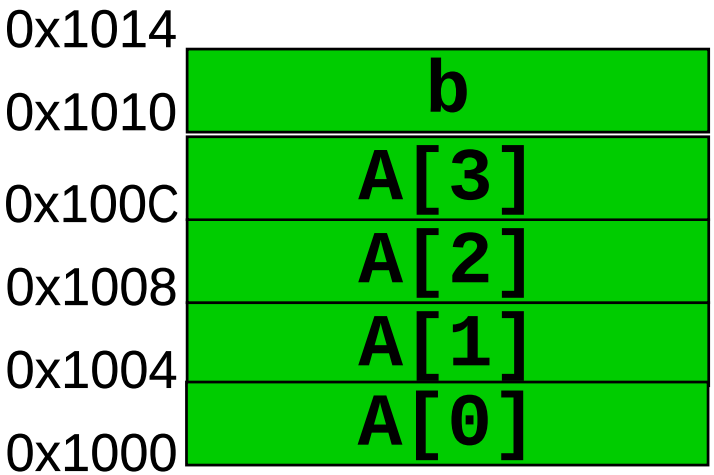
Arrays in C

```
int A[4];
int b;


A[0]   = 3;
A[3]   = 4;
A[4]  = 5;
A[-1]  = 6;
```

No bounds checking!

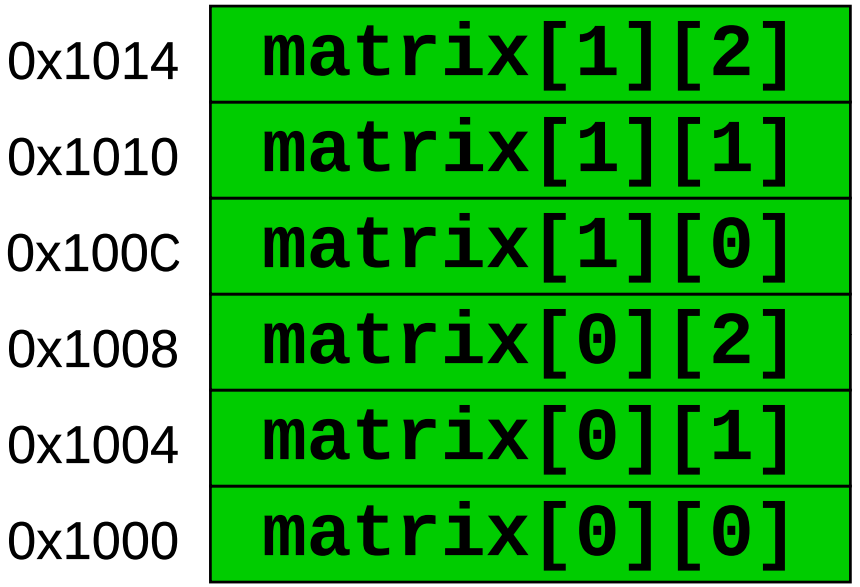
Allowed – usually causes no *obvious* error

A[4] may overwrite b



Multi-Dimensional Arrays

```
int  matrix[2][3];

matrix[1][0] = 17;
```



Recall: no bounds checking

What happens when you write:

```
matrix[0][3] = 42;
```

"Row Major" Organization

write at matrix[1][0] : **matrix+0*3+3** is same as **matrix+1*3+0**

Array Initialization

- `int A[5] = {2, 4, 8, 16, 32};`
- `int B[20] = {2, 4, 8, 16, 32};`
  - Unspecified elements are guaranteed to be zero
- `int C[4] = {2, 4, 8, 16, 32};`
  - Error — compiler detects too many initial values

2D array Initialization and access

```
int i, j;
int A[2][5] = {{20, 11, 22, 33, 44},
               {1, 12, 23, 34, 25} };
for (i=0; i<2; i++){
  for (j=0; j<5; j++) {
     printf("%d, ", A[i][j]);
  }
}
//prints 20, 11, 22, 33, 44, 1, 12, 23,
34, 25,
```

Implicit Array Size Determination

- `int days[] = {31,28,31,30,31, 30, 31,31,30,31,30,31};`
  - Array is created with as many elements as initial values
    - In this case, 12 elements
  - Values must be compile-time constants (for static arrays)
  - Values may be run-time expressions (for automatic arrays) : **will be discussed later**

Array Examples: Largest Element

```
int i, data[10], Largest;
printf("Enter 10 elements: ");
for(i=0; i < 10; ++i)
     scanf(“%d", &data[i]);
Largest=data[0];
for(i=1; i<10; ++i){
    if(Largest<data[i])
         Largest=data[i];
}
printf(“Largest Element=%d\n“,Largest);
```

Array Examples: Standard deviation

- SD σ = sqrt(($\sum (\mu - x_i)^2$) / N)

```
int i;
float data[10], sum=0.0, mean, SD=0.0;
printf("Enter 10 elements: ");
for(i=0; i < 10; ++i)
     scanf("%f", &data[i]);
for(i=0; i<10; ++i) sum += data[i];
mean = sum/10;
for(i=0; i<10; ++i)
   SD += (data[i]–mean)*(data[i]–mean);
SD = sqrt(SD/10);
printf("\nStandard Deviation=%f“,SD);
```

Caution! Caution! Caution!

- It is the programmer's responsibility to avoid indexing off the end of an array
  - *Likely* to corrupt data
  - May cause a *segmentation fault*
  - Could expose system to a *security hole!*
- *C* does NOT check *array bounds*
  - I.e., whether index points to an element within the array
  - Might be high (beyond the end) or negative (before the array starts)

Segmentation fault : GDB

- Finding Segmentation Fault using GDB
  - gcc -g test.c
  - ./a.out // Segmentaion fault core dump!
  - gdb ./a.out
  - gdb > run

```
int i, data[10], Largest=0;
for(i=0; i<50000; ++i){
    if(Largest<data[i])
        Largest=data[i];//i>=10
}
printf("Large Element=%d\n",Largest);
```

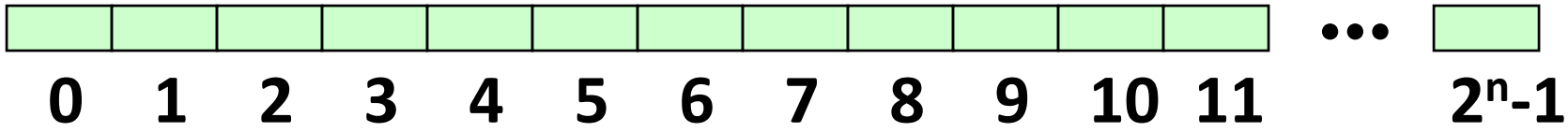
Pointers

- Special case of bounded-size natural numbers
  - Maximum memory limited by processor word-size
  - 2^{32} bytes = 4GB, 2^{64} bytes = 16 exabytes
- A pointer is just another kind of value
  - A basic type in C

```
int *ptr;
```

The variable “ptr” stores a pointer to an “int”.

Recall: Memory Organization



- All modern processors have memories organized as sequence of *numbered bytes*
  - Many (but not all) are linear sequences
- Definitions:–
  - *Byte:* an 8-bit memory cell capable of storing a value in range 0 … 255
  - *Address:* number by which a memory cell is identified