HIGH-LEVEL PROGRAMMING I

Arrays

by Prasanna Ghali

Reference

- Chapter 6 of text book

Aggregate Types

- Scalar type: can only store single data item
  - Every type studied so far: `char`, `short`, `int`, `long`, `long long`, `float`, `double`, `long double`
- Aggregate type: collection of values
  - Array: collection of homogeneous data
  - Structure: collection of heterogeneous data

Why Arrays?

- Problem: Computing summary statistics for 100s of students!!!
- Problem: Given date in form of day/month, print out day of year – see code on course page

Defining Arrays (1/6)

Size of array or number of elements in array.
Arrays have <u>fixed</u> size.
This means array size must be *constant integer expression*, so that compiler can know array size and <u>fix</u> *contiguous* storage during compilation!!!

name for collection of 5 `double` elements

element type

```
double grades[5];
```

1 2 3

`grades` is "array of 5 `double`s"

Defining Arrays (2/6)

- Array size must be <u>known</u> at compile time!!!

```
int grades[5];
```

```
int N;
N = 5;
int grades[N];
```

array size must be *constant integer expression*

```
#define SIZE (5)
int grades[SIZE];
```

```
int N, grades[];
scanf("%d", N);
grades[N];
```

Variable length arrays are non-standard in C++ and will compile with gcc and clang: must use `-Werror=vla` option to disable!!!

Defining Arrays (3/6)

- When you define an array, you can optionally specify initial values to array elements
- Lots of rules to remember!!!

```
double grades[5] = {11.1, 22.2, 33.3, 44.4, 55.5};
```

optional list of initial values

Defining Arrays (4/6)

- You can initialize all elements to `0.0`

```
double grades[100] = {0.0};
```

- You can partially initialize array: 1st two elements initialized to `10.1` and `20.2` and all other elements to `0.0`

```
double grades[100] = {10.1, 20.2};
```

- Obvious compile-time error if there're too many initializers

```
double grades[3] = {10.1, 20.2, 30.3, 40.4};
```

Defining Arrays (5/6)

- You can omit array size if you supply initial values
  - In this case, size is set to number of initial values

ok to omit size if initial values are given!!!

```
double grades[] = {11.1, 22.2, 33.3, 44.4, 55.5};
```

compiler will count initial values
to determine array size.
five initializers means array size is 5

Defining Arrays (6/6)

`int nums[10];`	Array of ten `int` elements with each element having garbage values.
`#define SIZE (10)` `int nums[SIZE];`	Good idea to use macro for size.
`int size = 10;` `int nums[size];`	Works in C99 but doesn't work in C++!!! Use gcc and clang option `-Werror=vla` to curb variable length arrays.
`int squares[5] = {0, 1, 4, 9, 16};`	Array of 5 `int`s, with initial values.
`int squares[] = {0, 1, 4, 9, 16};`	Array size can be omitted if initial values are supplied. Size is set to number of initial values.
`int squares[5] = {0, 1, 4};`	If fewer initial values than the size are specified, remaining values are set to `0`. This array has values: `0, 1, 4, 0, 0`.
`char initials[3];`	Array of three uninitalized characters.

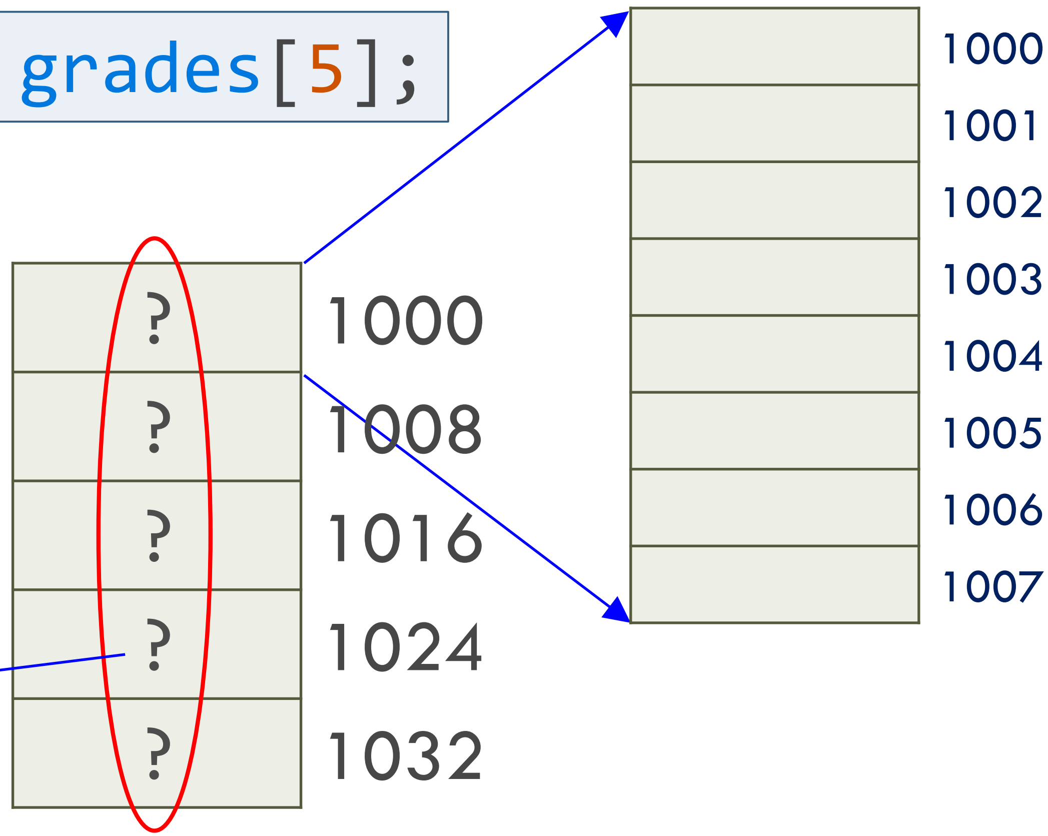
Arrays: Memory Storage (1/2)

- Arrays have linear structure - elements are given contiguous memory storage

```
double grades[5];
```

Memory block is 40 bytes which is equivalent to `sizeof(double)*5`

No initializers during definition; thus, memory contents are garbage!!!



Arrays: Memory Storage (2/2)


```
double grades[5] = {11.1, 22.2, 33.3, 44.4, 55.5};
```

1) *Base type* `double` means each element is 8 bytes
2) Size 5 means contiguous memory block is 40 bytes
3) Compiler will fix *base address* for array, say 1000
4) From compiler's perspective, name `grades` means base address 1000
5) Visualization

11.1	1000
22.2	1008
33.3	1016
44.4	1024
55.5	1032

Accessing Array Elements: Subscript Operator [ ] (1/6)

- Individual array elements are *anonymous* variables that don't have names
  - Accessed thro' *subscript* operator [ ]:

    array-name[*integer-expr*]
  - Can be used *anywhere* variable of similar type can be used

```
double grades[5];
grades[0] = 70.0;
grades[1] = grades[0] + 10.0;
grades[2] = ++grades[0];
++grades[1];
// and so on ...
```

Accessing Array Elements: Subscript Operator `[]` (2/6)

- Although elements are anonymous, they've *subscripts* in array: *subscript*, *index* are synonyms
- `grades[i]`: variable of type `double` offset by `i` elements from base address of array `grades`

subscript is `0`	11.1	`grades[0]`
subscript is 1	22.2	`grades[1]`
subscript is 2	33.3	`grades[2]`
subscript is 3	44.4	`grades[3]`
subscript is 4	55.5	`grades[4]`

subscripted variable of type `double` offset by `0` elements from base address of array `grades`

subscripted variable of type `double` offset by 3 elements from base address of array `grades`

```
double grades[5] = {11.1, 22.2, 33.3, 44.4, 55.5};
```

Accessing Array Elements: Subscript Operator [] (3/6)

- `grades[i]`: variable of type `double` offset by `i` elements from base address of array `grades`
- `grades[i]` can now be used anywhere a `double` variable can be used

11.1	grades[0]
22.2	grades[1]
33.3	grades[2]
44.4	grades[3]
55.5	grades[4]

```
grades[0] = grades[1]*2.0;
```

```
++grades[1];
```

```
grades[3] += grades[2];
```

```
double grades[5] = {11.1, 22.2, 33.3, 44.4, 55.5};
```

Accessing Array Elements: Subscript Operator [ ] (4/6)

- **No runtime boundary checking occurs when reading from and writing to array!!!**
- Reading/writing past array bounds results in undefined behaviour

```c
#include <stdio.h>

int main(void) {
  double grades[5] = {11.1, 22.2, 33.3, 44.4, 55.5};
  int i = 5000;
  // this call to printf crashes the program on my PC ...
  printf("grades[%d]: %f\n", i, grades[i]);
  return 0;
}
```

Accessing Array Elements: Subscript Operator [ ] (5/6)

- Notice that compiler doesn't provide any help in detecting reading/writing past array bounds

```
#include <stdio.h>

int main(void) {
  double grades[5] = {11.1, 22.2, 33.3, 44.4, 55.5};
  int i = 5;
  // writing to and reading from outside array ...
  grades[i] = 66.6;
  printf("grades[%d]: %f\n", i, grades[i]);
  return 0;
}
```

Accessing Array Elements: Subscript Operator [ ] (6/6)

- Another insidious error …

insidious because some other variable is inadvertently getting updated …

```
#include <stdio.h>

int main(void) {
  double grades[5] = {11.1, 22.2, 33.3, 44.4, 55.5};
  int i = -5;
  // writing to and reading from outside array ...
  grades[i] = 66.6;
  printf("grades[%d]: %f\n", i, grades[i]);
  return 0;
}
```

Arrays and Loops (1/2)

- Increase scores by two points

```
double grades[5] = {11.1, 22.2, 33.3, 44.4, 55.5};
for (int i = 0; i < 5; ++i) {
  // increase grades by 2 points
  grades[i] += 2.0;
}
```

- Here, loop is used to decrement scores by one point

```
double grades[5] = {11.1, 22.2, 33.3, 44.4, 55.5};
// decrement each element of grades by 1 point
for (int i = 0; i < 5; ++i) {
  --grades[i];
}
```

Arrays and Loops (2/2)

- Store values from standard input in array

```
double grades[5];
for (int i = 0; i < 5; ++i) {
  scanf("%lf", &grades[i]);
}
```

`grades[i]` is like any other variable. Therefore, `&grades[i]` will pass to `scanf` the memory address of subscripted variable `grades[i]`

- Compute sum of array elements

```
double sum = 0.0;
int i = 0;
while (i < 5) {
  sum += grades[i++];
}
```

equivalent

```
sum += grades[i]; i++;
```

wrong!!!

```
sum += grades[++i];
```

Off-By-One Error

- Store values from standard input in array

```
double grades[5];
for (int i = 0; i <= 5; ++i) {
  scanf("%lf", &grades[i]);
}
```

causes undefined behavior!!!

Can’t Assign to Array!!!

- Recall: memory location of array is fixed!!!
- Therefore, array name cannot be on left-side of assignment operator

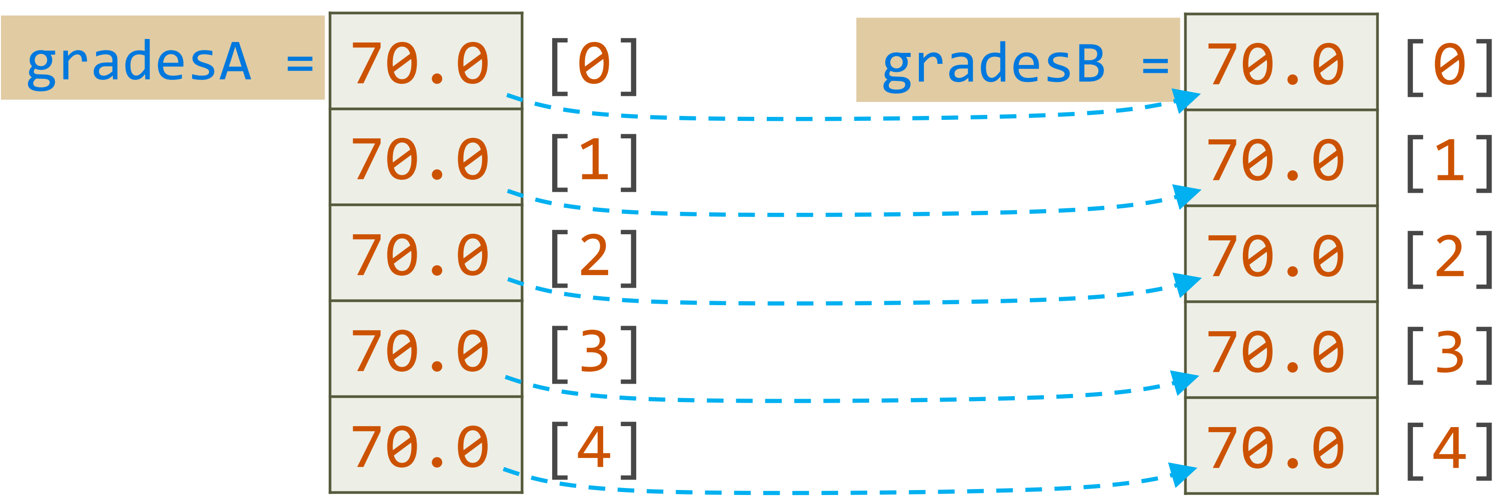
```
double gradesA[5], gradesB[5];
// everybody in section A passes ...
for (int i = 0; i < 5; ++i) {
  gradesA[i] = 70.0;
}

// assign an array to another array ...
gradesB = gradesA; // error
```

Copying Arrays (1/2)

- Must copy element-wise – one element at a time

```
double gradesA[5], gradesB[5];
// everybody in A passes the course
for (int i = 0; i < 5; ++i) {
  gradesA[i] = 70.0;
}
// everybody in B also passes the course
```



Copying Arrays (2/2)

- Must copy element-wise – one element at a time

```
double gradesA[5], gradesB[5];
// everybody in A passes the course
for (int i = 0; i < 5; ++i) {
  gradesA[i] = 70.0;
}

// copy values from gradesA to gradesB ...
for (int i = 0; i < 5; ++i) {
  gradesB[i] = gradesA[i];
}
```

`sizeof` Operator (1/3)

`sizeof(expr)` **OR** *`sizeof(type)`*

- Returns `unsigned long` value representing number of bytes of storage required for variable or object that operand evaluates to

```
printf("sizeof(char):   %lu\n", sizeof(char));
printf("sizeof(int) :   %lu\n", sizeof(int));
printf("sizeof(10ULL):  %lu\n", sizeof(long long));
printf("sizeof('a'):    %lu\n", sizeof('a'));
printf("sizeof(double): %lu\n", sizeof(double));
```

sizeof Operator (2/3)

- For array names, sizeof returns number of bytes of storage for all array elements
- Type of value returned by sizeof: size_t
  - size_t is *largest unsigned integral type* – most implementations use unsigned long int

```
int    racers[5];
double prizes[10];
char   inits[1000];

printf("sizeof(racers): %lu\n", sizeof(racers));
printf("sizeof(prizes): %lu\n", sizeof(prizes));
printf("sizeof(inits):  %lu\n", sizeof(inits));
```

Notice that array name is operand to sizeof operator

sizeof Operator (3/3)

- Determining array size …

```c
int racers[] = {1,2,3,4,5};
// how to find size of array racers?
// sizeof(racers)     == 20 and
// sizeof(racers[0]) == 4  and, therefore
// sizeof(racers)/sizeof(racers[0]) == 5
unsigned int usz = sizeof(racers)/sizeof(racers[0]);
printf("size of array racers: %u\n", usz);
```

Recap: Pass-by-Value Convention of Functions (1/2)

this variable is called *formal parameter* or just *parameter*

```
int myabs(int number) {
  return number < 0 ? -number : number;
}
```

client calls function `myabs` using function call operator `()`

```
int num = 10;
num = myabs(-num)
```

this expression is called *function argument*

1) At runtime, argument (expression) `-num` is evaluated
2) Result of evaluation is used to initialize parameter `number`
3) Changes made to parameter `number` are localized to function `myabs`
4) Function `myabs` terminates by returning value of type `int`
5) When function `myabs` terminates, variable `number` ceases to exist

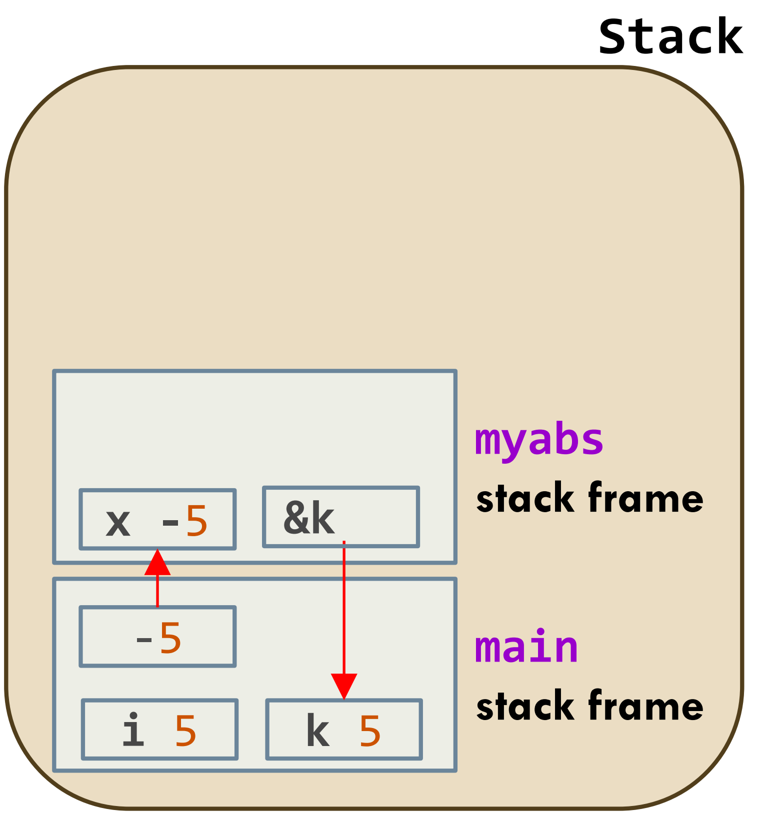
Recap: Pass-by-Value Convention of Functions (2/2)

- Visualization

```c
#include <stdio.h>

int myabs(int x) {
  return x > 0 ? x : -x;
}

int main(void) {
  int i = 5;
  int k;
  k = myabs(-i);
  printf("abs(-%d): %d\n", i, k);
  return 0;
}
```



Array and Functions

- Pass-by-value convention is also used to pass array to a function
  - Inefficient to pass entire array - every array element must be copied from caller to callee
  - Instead, what is passed by value is array's base address
- However, base address doesn't say anything about array bounds
- Therefore, array size must be passed too!!!

Array and Functions: Syntax

- Code visualizer

empty brackets specify parameter arr is array base address

```
void print_dbl_arr(double arr[], int size) {
  for (int i = 0; i < size; ++i) {
    printf("%.2f\n", arr[i]);
  }
}

double grades[5] = {11.1, 22.2, 33.3, 44.4, 55.5};
print_dbl_arr(grades, 5);
```

arr has copy of value in argument grades

argument evaluates to base address of array grades

Using `const` to Protect Array Elements (1/3)

- `print_dbl_arr` only wants to read elements of array parameter, not change their values

```
void print_dbl_arr(double arr[], int size) {
  for (int i = 0; i < size; ++i) {
    printf("%.2f\n", arr[i]);
  }
}
```

Using `const` to Protect Array Elements (2/3)

- Author can document function wont modify array passed to function by adding *type qualifier* `const` to parameter's declaration

```
void print_dbl_arr(double const arr[], int size);
```

- Now contract established between author and clients:
  - Array elements of parameter `arr` will not be changed by function `print_dbl_arr`
  - Compiler will enforce this contract
- Visualizer code

Using const to Protect Array Elements (3/3)

- Following declarations are equivalent since:
  - type specifier (const) and type qualifiers (such as int, double, and so on) can be written in any order
  - names of parameter can be omitted

```
void print_dbl_arr(double const [], int size);
void print_dbl_arr(const double [], int size);
void print_dbl_arr(double const arr[], int size);
void print_dbl_arr(const double arr[], int size);
void print_dbl_arr(double const [], int);
void print_dbl_arr(const double [], int);
void print_dbl_arr(double const arr[], int);
void print_dbl_arr(const double arr[], int);
```

Passing Partial Array

- Common practice to pass size smaller than actual number of elements in array
- Visualizer

```
void print_dbl_arr(double const arr[], int size) {
  for (int i = 0; i < size; ++i) {
    printf("%.2f\n", arr[i]);
  }
}
#define SIZE (5)
double grades[SIZE] = {11.1, 22.2, 33.3, 44.4, 55.5};
printf("first 3 elements: ");
print_dbl_arr(grades, 3);
printf("last %d elements: ", SIZE-1);
print_dbl_arr(&grades[1], SIZE-1);
```

Common Array Algorithms: Computing Sum of Elements

- Visualizer

```
double dsum(double const arr[], double size) {
  double sum = arr[0];
  for (int i = 1; i < size; ++i) {
    sum += arr[i];
  }
  return sum;
}
```

```
#define SIZE (5)
double grades[SIZE] = {11.1, 22.2, 33.3, 44.4, 55.5};
double sum = dsum(grades, SIZE);
printf("Sum: %.2f | Avg: %.2f\n", sum, sum/(double)SIZE);
```

Common Array Algorithms: Maximum Value

- Strategy is to remember, using a variable, the largest value seen so far …
- Visualizer code

```
double dmax(double const arr[], int size) {
  double max_val = arr[0];
  for (int i = 1; i < size; ++i) {
    max_val = arr[i] > max_val ? arr[i] : max_val;
  }
  return max_val;
}
```

Common Array Algorithms: Minimum Value

- Strategy is to remember, using a variable, the smallest value seen so far …
- Visualizer code

```
double dmin(double const arr[], int size) {
  double min_val = arr[0];
  for (int i = 1; i < size; ++i) {
    min_val = arr[i] < min_val ? arr[i] : min_val;
  }
  return min_val;
}
```

Common Array Algorithms: Linear Search (1/2)

- *Linear search* inspects elements in sequence until match is found
- To search for specific value, visit elements and stop when a match is encountered

Common Array Algorithms: Linear Search (2/2)

- Visualizer code

```
int lin_int_search(int const arr[], int size, int val) {
  for (int pos = 0; pos < size; ++pos) {
    if (val == arr[pos]) {
      return pos;
    }
  }
  return -1; // no element in arr having value val ...
}
```

Summary

- Array is aggregate data structure that contains several elements of the same type
- Each element is accessed using an index or subscript in square brackets
- Arrays begin with *position* or *subscript* or *offset* or *index* 0
- C has no built-in boundary checking for arrays!
- Array is passed to function by passing array’s base address