Passing Arrays to Functions

Concept of Pass-by-Value & Pass-by-Reference

Outline

- Passing a 1-D array to a function
- An array is passed by reference
- Passing a 2-D array to a function

1. Passing a 1-D Array to a Function

- Why do we mention specifically about passing arrays to functions?
- **Short Answer:** There are both syntactical and conceptual issues.
  - – Let's start with syntax first

1. Passing a 1-D Array to a Function

```c
void printArray(int array [], int arraySize) {
    int i;
    for (i = 0; i < arraySize; i++)
        printf("%d ", array[i]);
    printf("\n");
}

int main(void) {
    int A[5] = { 1, 2, 3, 4, 5 }, B[10] = { 0 };

    printArray( A, 5 );   // Output "1 2 3 4 5 "
    printArray( B, 10 );  // Output "0 0 0 0 0 0 0 0 0 0 "
    printArray( A, 3 );   // Output "1 2 3 "

    return 0;
}
```

1. 1-D Arrays as Formal Parameters

```c
void printArray(int array [], int arraySize) {
    int i;
    for (i = 0; i < arraySize;
        printf("%d ", array[i])
    printf("\n");
}

int main(void) {
    int A[5] = { 1, 2, 3, 4, 5 }, B[10] = { 0 };

    printArray( A, 5 );   // Output "1 2 3 4 5 "
    printArray( B, 10 );  // Output "0 0 0 0 0 0 0 0 0 0 "
    printArray( A, 3 );   // Output "1 2 3 "

    return 0;
}
```

Needs a pair of **[ ]** after the parameter name.

Number inside **[ ]**, if any, is ignored.

1. 1-D Arrays as Actual Parameters

```
void printArray(int array [], int arraySize) {
    int i;
    for (i = 0; i
        printf("%
    printf("\n");
}

int main(void) {
    int A[5] = { 1, 2, 3, 4, 5 }, B[10] = { 0 };

    printArray( A, 5 );   // Output "1 2 3 4 5 "
    printArray( B, 10 );  // Output "0 0 0 0 0 0 0 0 0 0 "
    printArray( A, 3 );   // Output "1 2 3 "

    return 0;
}
```

The actual parameter can be a 1-D array of the same data type of ANY size.

The array name, "A" or "B", already represents an "array of `int`".

1. Indicating Array Size via a Parameter

```c
void printArray(int array [], int arraySize) {
    int i;
    for (i = 0; i < arraySize;
        printf("%d ", array[i])
    printf("\n");
}

int main(void) {
    int A[5] = { 1, 2, 3, 4, 5 }, B[10] = { 0 };

    printArray( A, 5 );    // Output "1 2 3 4 5 "
    printArray( B, 10 );   // Output "0 0 0 0 0 0 0 0 0 0 "
    printArray( A, 3 );    // Output "1 2 3 "

    return 0;
}
```

The function is ***unaware*** of the size of the actual parameter; the size is usually indicated using a separate parameter.

2. A Curious Example

```c
void clear(int A[], int size, int B) {
    int i;
    for (i = 0; i < size; i++)
        A[i] = 0;
    B = 0;
}

int main(void) {
    int C[3] = { 1, 2, 3 };
    int D = 10;
    clear(C, 3, D);
    printf("%d %d %d %d\n",C[0],C[1],C[2],D);

    return 0;
}
```

Can you dry run the program and tell me your expected output?

2. A Curious Example

```c
void clear(int A[], int size, int B) {
    int i;
    for (i = 0; i < size; i++)
        A[i] = 0;
    B = 0;
}

int main(void) {
    int C[3] = { 1, 2, 3 };
    int D = 10;
    clear(C, 3, D);
    printf("%d %d %d %d\n",C[0],C[1],C[2],D);

    return 0;
}
```

The actual output is:

```
0 0 0 10
```

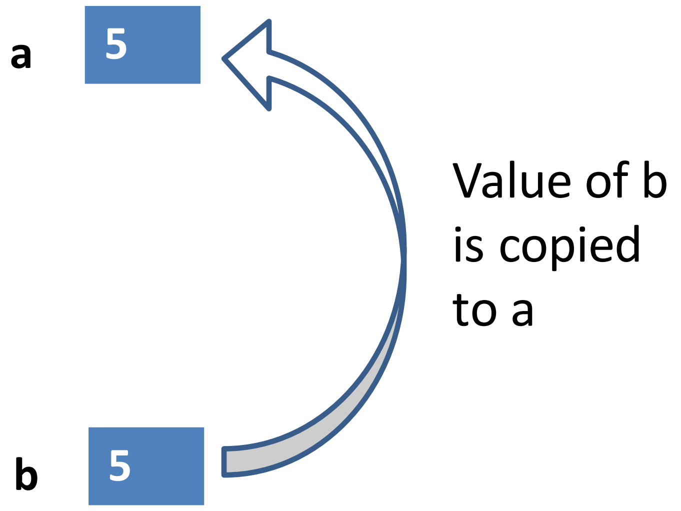
Why would that be?

2. Array is passed to a function by reference

- When an array is passed to a function via a parameter, the array is *passed by reference* (an ordinary variable is *passed by value*).
- When a parameter is passed by reference, modifying the formal parameter has the same effect on the actual parameter.
- That is, an array parameter is SHARED between two functions, as an actual parameter at the caller side AND as a formal parameter at the callee.

2. When an `int` parameter is passed by value

```
1   void clear(int a) {
2       a = 0;
3   }
4
5   int main(void) {
6       int b = 5;
7
8       clear(b);
9       // After line 8 is executed,
10      // b remains 5
11
12      return 0;
13  }
14
15
16
```



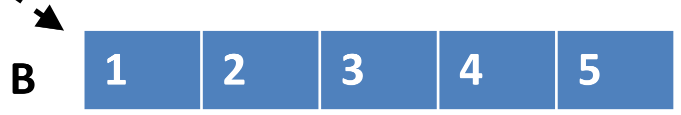
When a parameter is passed by value, the value of the actual parameter is copied to the formal parameter; they have their own storage space.

If you print the addresses of **a** and **b**, they are obviously **two different memory locations**

2.When a parameter is passed by reference

```
1   void clear(int A[], int N) {
2       int i;
3       for (i = 0; i < N; i++)
4           A[i] = 0;
5   }
6
7   int main(void) {
8       int B[5] = { 1, 2, 3, 4, 5 };
9
10      clear(B, 5);
11      // After line 10 is executed,
12      // all elements of B will
13      // become 0's
14
15      return 0;
16  }
```

When a parameter is passed by reference, the formal parameter becomes an *alias* of the actual parameter during the function call.



In this example, the formal parameter **A** and the actual parameter **B** refer to the same array during the function call.

2. When a parameter is passed by reference

```
void clear(int A[], int N) {
    int i;
    for (i = 0; i < N; i++)
        A[i] = 0;
    printf("%p\n",&A[0]);
}

int main(void) {
    int B[5] = { 1, 2, 3, 4, 5 };

    clear(B, 5);
    printf("%p\n",&B[0]);

    return 0;
}
```

If you print the addresses of **A** and **B**, you will find that they have the **same address**.

3. Pass-by-value vs. Pass-by-reference

- In most cases we would prefer pass-by-value for our functions.
  - In C, we generally do not expect to change our actual parameters when we pass them into a function
- Pass-by-reference is most useful when you <u>wish to update the actual parameter passed in</u>.
- At the end of the course you will get to learn how to pass ordinary variables (non-arrays) by reference (more accurately, to emulate).

3. Example: Using array to pass data from a callee to a caller

```c
void readIntegers(int A[], int N) {
    int i;
    for (i = 0; i < N; i++)
        scanf("%d", &A[i]);
}

int main(void) {
    int B[100];

    // Pass an array to the function to store 100 input.
    readIntegers(B, 100);

    return 0;
}
```

4. Passing a 2-D Array to a Function

```c
void foo(int array[][64], int rows) {
  // array should be treated in this function as a rowsx64 2D-array
  ...
}

int main() {
  int a1[24][64], a2[100][64], a3[10][2];

  foo( a1, 24 );    // OK; process row 0-23
  foo( a2, 100 );   // OK; process row 0-99
  foo( a2, 10 );    // OK; process row 0-9

  foo( a3, 10 );    // Compile-time error;  different 2nd dimension
  ...
}
```

4. 2-D Arrays as Formal Parameters

```
void foo(int array[][64], int rows) {
   // array should be treated in this function as a rowsx64 2D-array
   ...
}

int main() {
   int a1[24][64], a2[100]

   foo( a1, 24 );     // OK; process row 0-23
   foo( a2, 100 );    // OK; process row 0-99
   foo( a2, 10 );     // OK; process row 0-9

   foo( a3, 10 );     // Compile-time error;  different 2nd dimension
   ...
}
```

The size of the 1st dimension, if any, is ignored, but the size of 2nd dimension is required.

4. 2-D Arrays as Actual Parameters

```
void foo(int array[][64], int rows) {
   // array should be treated in this function as a rowsx64 2D-array
   ...
}

int main() {
   int a1[24][64], a2[100][64], a3[10][2];

   foo( a1, 24 );     // OK; process row 0-23
   foo( a2, 100 );    // OK; process row 0-99
   foo( a2, 10 );     // OK; process row 0-9

   foo( a3, 10 );     // Compile-time error;  different 2nd dimension
   ...
}
```

The actual parameter can be a 2-D array of the same data type in which its <u>second dimension must match</u>.

4. Indicating the 1st Dimension Size via a Parameter

```
void foo(int array[][64], int rows) {
   // array should be treated in this function as a rowsx64 2D-array
   ...
}

int main() {
   int a1[24][64], a2[100][64

   foo( a1, 24 );     // OK; process row 0-23
   foo( a2, 100 );    // OK; process row 0-99
   foo( a2, 10 );     // OK; process row 0-9

   foo( a3, 10 );     // Compile-time error;  different 2nd dimension
   ...
}
```

The function is unaware of the size of the first dimension of the actual parameter; the size is usually indicated using a separate parameter.

Reading Assignment

- C: How to Program, 8th ed, Deitel and Deitel
- Chapter 6 C Arrays
  - Sections 6.7: Passing Arrays to Functions
  - Sections 6.9: A Case Study
  - Sections 6.11: Multidimensional Arrays