BBM 101 – Introduction to Programming I

Fall 2014, Lecture 8

Aykut Erdem, Erkut Erdem, Fuat Akal

Today

- **Debugging**
  - Testing and debugging
  - Black box testing
  - Glass box testing
  - Integration testing and unit testing
  - Debugging approaches
- **Arrays**
  - Declaring Arrays
  - Examples
  - Passing Arrays to Functions
  - Sorting Arrays
  - Multi-Dimensional Arrays
  - Command Line Input

Today

- **Debugging**
  - Testing and debugging
  - Black box testing
  - Glass box testing
  - Integration testing and unit testing
  - Debugging approaches
- Arrays
  - Declaring Arrays
  - Examples
  - Passing Arrays to Functions
  - Sorting Arrays
  - Multi-Dimensional Arrays
  - Command Line Input

Testing and Debugging

- Would be great if our code always worked properly the first time we run it!
- But life ain't perfect, so we need:
  - Testing methods
    - Ways of trying code on examples to determine if running correctly
  - Debugging methods
    - Ways of fixing a program that you know does not work as intended

Slide credit: E. Grimson, J. Guttag and C. Terman

When should you test and debug?

- Design your code for ease of testing and debugging
  - Break program into components that can be tested and debugged independently
  - Document constraints on modules
    - Expectations on inputs, on outputs
    - Even if code does not enforce constraints, valuable for debugging to have description
  - Document assumptions behind code design

When are you ready to test?

- Ensure that code will actually run
  - Remove syntax errors
  - Remove static semantic errors
  - Both of these are typically handled by the C compiler
- Have a set of expected results (i.e. input- output pairings) ready

Testing

- Goal:
  - Show that bugs exist
  - Would be great to prove code is bug free, but generally hard
    - Usually can't run on all possible inputs to check
    - Formal methods sometimes help, but usually only on simpler code

"Program testing can be used to show the presence of bugs, but never to show their absence!"
– Edsger Dijkstra

Test suite

- Want to find a collection of inputs that has high likelihood of revealing bugs, yet is efficient
  - Partition space of inputs into subsets that provide equivalent information about correctness
    - Partition divides a set into group of subsets such that each element of set is in exactly one subset
  - Construct test suite that contains one input from each element of partition
  - Run test suite

Example of partition

```
/* Assumes x and y are ints
   returns 1 if x is less than y
   else returns 0*/
int isBigger(int x, int y);
```

- Input space is all pairs of integers
- Possible partition
  - – x positive, y positive
  - – x negative, y negative
  - – x positive, y negative
  - – x negative, y positive
  - – x=0,y=0
  - – x=0,y!=0
  - – x!=0,y=0

Why this partition?

- Lots of other choices
  - E.g., x prime, y not; y prime, x not; both prime; both not
- Space of inputs often have natural boundaries
  - Integers are positive, negative or zero
  - From this perspective, have 9 subsets
    - Split x = 0, y != 0 into x = 0, y positive and x =0, y negative
    - Same for x != 0, y = 0

Partitioning

- What if no natural partition to input space?
  - Random testing – probability that code is correct increases with number of trials; but should be able to use code to do better
  - Use heuristics based on exploring paths through the specifications – **black-box testing**
  - Use heuristics based on exploring paths through the code – **glass-box testing**

Black-box testing

- Test suite designed without looking at code
  - Can be done by someone other than implementer
  - Will avoid inherent biases of implementer, exposing potential bugs more easily
  - Testing designed without knowledge of implementation, thus can be reused even if implementation changed

Paths through a specification

```
float sqrt(float x, float eps)
/* Assumes x, eps floats
     x >= 0
     eps > 0
   returns res such that
     x-eps <= res*res <= x+eps */
```

- Paths through specification:
  - x = 0
  - x > 0
- But clearly not enough

Paths through a specification

- Also good to consider boundary cases
  - For numbers, very small, very large, “typical”

Example

- For our sqrt case, try these:
  - First four are typical
    - Perfect square
    - Irrational square root
    - Example less than 1
  - Last five test extremes
    - If bug, might be code, or might be spec (e.g. don’t try to find root if eps tiny)

x	eps
0.0	0.0001
25.0	0.0001
.05	0.0001
2.0	0.0001
2.0	1.0/pow(2.0,64.0)
1.0/pow(2.0,64.0)	1.0/pow(2.0,64.0)
pow(2.0,64.0)	1.0/pow(2.0,64.0)
1.0/pow(2.0,64.0)	pow(2.0,64.0)
pow(2.0,64.0)	pow(2.0,64.0)

Glass-box Testing

- Use code directly to guide design of test cases
- Glass-box test suite is **path-complete** if every potential path through the code is tested at least once
  - Not always possible if loop can be exercised arbitrary times, or recursion can be arbitrarily deep
- Even path-complete suite can miss a bug, depending on choice of examples

Example

```
/* Assumes x is an int
returns x if x>=0 and -x otherwise */
int abs(x)
{
  if (x<-1)
      return -x;
  else
      return x;
}
```

- Test suite of {-2, 2} will be path complete
- But will miss **abs(-1)** which incorrectly returns -1
  - Testing boundary cases and typical cases would catch this {-2 -1, 2}

Rules of thumb for glass-box testing

- Exercise both branches of all if statements
- Ensure each except clause is executed
- For each for loop, have tests where:
  - Loop is not entered
  - Body of loop executed exactly once
  - Body of loop executed more than once
- For each while loop,
  - Same cases as for loops
  - Cases that catch all ways to exit loop
- For recursive functions, test with no recursive calls, one recursive call, and more than one recursive call

Conducting tests

- Start with **unit testing**
  - Check that each module (e.g. function) works correctly
- Move to **integration testing**
  - Check that system as whole works correctly
- Cycle between these phases

Test Drivers and Stubs

- **Drivers** are code that
  - Set up environment needed to run code
  - Invoke code on predefined sequence of inputs
  - Save results, and
  - Report
- Drivers simulate parts of program that use unit being tested
- **Stubs** simulate parts of program used by unit being tested
  - Allow you to test units that depend on software not yet written

Good testing practice

- Start with unit testing
- Move to integration testing
- After code is corrected, be sure to do **regression testing**:
- Check that program still passes all the tests it used to pass, i.e., that your code fix hasn't broken something that used to work

"Most people, if you describe a train of events to them, will tell you what the result would be. They can put those events together in their minds, and argue from them that something will come to pass. There are few people, however, who, if you told them a result, would be able to evolve from their own inner consciousness what the steps were which led up to that result. This power is what I mean when I talk of reasoning backwards, or analytically." -- Sherlock Holmes (A Study in Scarlet, by Sir Arthur Conan Doyle)

Debugging

- The "history" of debugging
  - Often claimed that first bug was found by team at Harvard that was working on the Mark II Aiken Relay Calculator
  - A set of tests on a module had failed; when staff inspected the actually machinery (in this case vacuum tubes and relays), they discovered this:

9/9

0800 Anctan started

1000 stopped - anctan ✓

13" uc (032) MP - MC 2.130476415 4.615925059(-2)

(033) PRO 2 2.130476415

correct 2.130676415

Relays 6-2 in 033 failed special speed test

Relays changed

1100 Started Cosine Tape (Sine check)

1525 Started Mult + Adder Test.

1545 Relay #70 Panel F (moth) in relay.

First actual case of bug being found.

1630 antangent started.

1700 closed down.

Runtime bugs

- **Overt vs. covert:**
  - **Overt** has an obvious manifestation – code crashes or runs forever
  - **Covert** has no obvious manifestation – code returns a value, which may be incorrect but hard to determine
- **Persistent vs. intermittent:**
  - **Persistent** occurs every time code is run
  - **Intermittent** only occurs some times, even if run on same input

Categories of bugs

- Overt and persistent
  - Obvious to detect
  - Good programmers use **defensive programming** to try to ensure that if error is made, bug will fall into this category
- Overt and intermittent
  - More frustrating, can be harder to debug, but if conditions that prompt bug can be reproduced, can be handled
- Covert
  - Highly dangerous, as users may not realize answers are incorrect until code has been run for long period

Debugging skills

- Treat as a search problem: looking for explanation for incorrect behavior
- Study available data – both correct test cases and incorrect ones
- Form an hypothesis consistent with the data
- Design and run a repeatable experiment with potential to refute the hypothesis
- Keep record of experiments performed: use narrow range of hypotheses

Debugging as search

- Want to narrow down space of possible sources of error
- Design experiments that expose intermediate stages of computation (use print statements!), and use results to further narrow search
- Binary search can be a powerful tool for this

Some pragmatic hints

- Look for the usual suspects
- Ask why the code is doing what it is, not why it is not doing what you want
- The bug is probably not where you think it is – eliminate locations
- Explain the problem to someone else
- Don't believe the documentation
- Take a break and come back to the bug later

Today

- Debugging
  - Testing and debugging
  - Black box testing
  - Glass box testing
  - Integration testing and unit testing
  - Debugging approaches
- Arrays
  - Declaring Arrays
  - Examples
  - Passing Arrays to Functions
  - Sorting Arrays
  - Multi-Dimensional Arrays
  - Command Line Input

Arrays

- A block of variables grouped together
- Static entity – same size throughout program
- Dynamic data structures will be discussed next week

- declaration:

```
int score[5];
```


Arrays

- Group of consecutive memory locations
- Same name and type

- To refer to an element, specify
  - Identifier
  - Index

- Format:

  ***identifier*[*index*]**
  - First element at index 0
  - n element array named **c**:
    - **c[0],c[1]...c[n-1]**

Name of array (Note that all elements of this array have the same name, **c**)

c[0]	-45
c[1]	6
c[2]	0
c[3]	72
c[4]	1543
c[5]	-89
c[6]	0
c[7]	62
c[8]	-3
c[9]	1
c[10]	6453
c[11]	78

Position number (index) of the element within array **c**

Arrays

- Array elements are like normal variables

```
c[0] =  3;
printf("%d", c[0]);
```

  - Perform operations in subscript. If x equals 3

```
c[5-2] == c[3] == c[x]
c[x+1] == c[4]
c[x-1] == c[2]
```


Operator precedences – Revisited

Operators						Associativity	Type
[]	()					left to right	highest
++	--	!	*(type)*			right to left	unary
*	/	%				left to right	multiplicative
+	-					left to right	additive
<	<=	>	>=			left to right	relational
==	!=					left to right	equality
&&						left to right	logical and
||						left to right	logical or
?:						right to left	conditional
=	+=	-=	*=	/=	%=	right to left	assignment
,						left to right	comma


```
double x[8];
```

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
16.0	12.0	6.0	8.0	2.5	12.0	14.0	-54.5

```
i=5
printf("%d %.1f", 4, x[4]);      4 2.5
printf("%d %.1f", i, x[i]);      5 12.0
printf("%.1f", x[i]+1);          13.0
printf("%.1f", x[i]+i);          17.0
printf("%.1f", x[i+1]);          14.0
printf("%.1f", x[i+i]);          invalid
printf("%.1f", x[2*i]);          invalid
printf("%.1f", x[2*i-3]);        -54.5
printf("%.1f", x[(int)x[4]]);    6.0
printf("%.1f", x[i++]);          12.0
printf("%.1f", x[--i]);          12.0
```

May result in a run-time error
Display incorrect results

```
double x[8];
```

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
16.0	12.0	6.0	8.0	2.5	12.0	14.0	-54.5

```
i=5
x[i-1] = x[i]
x[i] = x[i+1]
x[i]-1 = x[i]
```

Illegal assignment statement!

Declaring Arrays

- When declaring arrays, specify
  - Name
  - Type of array
  - Number of elements

```
arrayType arrayName[ numberOfElements ];
```

Examples:

```
int c[ 10 ];
float myArray[ 3284 ];
```

- Defining multiple arrays of same type
  - Format similar to regular variables
  - Example:

```
int b[ 100 ], x[ 27 ];
```


Examples

- Initializers

```
int n[ 5 ] = { 1, 2, 3, 4, 5 };
```

  - If not enough initializers, rightmost elements become 0

```
int n[ 5 ] = { 0 }
```

    All elements 0
  - C arrays have no bounds checking

- If size omitted, initializers determine it

```
int n[ ] = { 1, 2, 3, 4, 5 };
```

  - 5 initializers, therefore 5 element array

Initializing an Array

```
1  /*
2     initializing an array */
3  #include <stdio.h>
4
5  /* function main begins program execution */
6  int main()
7  {
8     int n[ 10 ]; /* n is an array of 10 integers */
9     int i;       /* counter */
10
11    /* initialize elements of array n to 0 */
12    for ( i = 0; i < 10; i++ ) {
13       n[ i ] = 0; /* set element at location i to 0 */
14    } /* end for */
15
16    printf( "%s%13s\n", "Element", "Value" );
17
18    /* output contents of array n in tabular format */
19    for ( i = 0; i < 10; i++ ) {
20       printf( "%7d%13d\n", i, n[ i ] );
21    } /* end for */
22
23    return 0; /* indicates successful termination */
24
25 } /* end main */
```

```
Element        Value
      0            0
      1            0
      2            0
      3            0
      4            0
      5            0
      6            0
      7            0
      8            0
      9            0
```

Examples

- Reading values into an array

```
int i, x[100];
```

```
for (i=0; i < 100; i=i+1)
{
  printf("Enter an integer: ");
  scanf("%d",&x[i]);
}
```

- Summing up all elements in an array

```
int sum = 0;
for (i=0; i<=99; i=i+1)
   sum = sum + x[i];
```


Examples

- Finding the location of a given value (`item`) in an **`int`** array of size 100.

```
i = 0;
while ((i<100) && (x[i] != item))
   i = i + 1;

if (i == 100)
   loc = -1; // not found
else
   loc = i;  // found in location i
```


Examples

- Shifting the elements of an array to the left.

```
/* store the value of the first element in a
 * temporary variable
 */
temp = x[0];

for (i=0; i < 99; i=i+1)
   x[i] = x[i+1];

//The value stored in temp is going to be
the value of the last element:
x[99] = temp;
```


Examples Using Arrays

- Character arrays
  - String "**`first`**" is really a static array of characters
  - Character arrays can be initialized using string literals
    - **`char string1[] = "first";`**
    - Null character **`'\0'`** terminates strings
    - **`string1`** actually has 6 elements
      - equivalent to **`char string1[]={'f','i','r','s','t','\0' };`**
  - Can access individual characters

    `string1[3] is character 's'`
  - Array name is address of array, so & not needed for scanf

    `scanf( "%s", string2 );`
    - Reads characters until whitespace encountered
    - Can write beyond end of array, be careful

```c
1  /*
2     Initializing an array with an initializer list */
3  #include <stdio.h>
4
5  /* function main begins program execution */
6  int main()
7  {
8     /* use initializer list to initialize array n */
9     int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
10    int i; /* counter */
11
12    printf( "%s%13s\n", "Element", "Value" );
13
14    /* output contents of array in tabular format */
15    for ( i = 0; i < 10; i++ ) {
16       printf( "%7d%13d\n", i, n[ i ] );
17    } /* end for */
18
19    return 0; /* indicates successful termination */
20
21 } /* end main */
```

Element	Value
0	32
1	27
2	64
3	18
4	95
5	14
6	90
7	70
8	60
9	37


```c
1  /*
2     Initialize the elements of array s to the even integers from 2 to 20 */
3  #include <stdio.h>
4  #define SIZE 10
5
6  /* function main begins program execution */
7  int main()
8  {
9     /* symbolic constant SIZE can be used to specify array size */
10    int s[ SIZE ]; /* array s has 10 elements */
11    int j;         /* counter */
12
13    for ( j = 0; j < SIZE; j++ ) { /* set the values */
14       s[ j ] = 2 + 2 * j;
15    } /* end for */
16
17    printf( "%s%13s\n", "Element", "Value" );
18
19    /* output contents of array s in tabular format */
20    for ( j = 0; j < SIZE; j++ ) {
21       printf( "%7d%13d\n", j, s[ j ] );
22    } /* end for */
23
24    return 0; /* indicates successful termination */
25
26 } /* end main */
```

Element	Value
0	2
1	4
2	6
3	8
4	10
5	12
6	14
7	16
8	18
9	20


```c
1  /*
2     Compute the sum of the elements of the array */
3  #include <stdio.h>
4  #define SIZE 12
5
6  /* function main begins program execution */
7  int main()
8  {
9     /* use initializer list to initialize array */
10    int a[ SIZE ] = { 1, 3, 5, 4, 7, 2, 99, 16, 45, 67, 89, 45 };
11    int i;         /* counter */
12    int total = 0; /* sum of array */
13
14    /* sum contents of array a */
15    for ( i = 0; i < SIZE; i++ ) {
16       total += a[ i ];
17    } /* end for */
18
19    printf( "Total of array element values is %d\n", total );
20
21    return 0; /* indicates successful termination */
22
23 } /* end main */
```

Total of array element values is 383


```c
1  /*
2     Student poll program */
3  #include <stdio.h>
4  #define RESPONSE_SIZE 40 /* define array sizes */
5  #define FREQUENCY_SIZE 11
6
7  /* function main begins program execution */
8  int main()
9  {
10    int answer; /* counter */
11    int rating; /* counter */
12
13    /* initialize frequency counters to 0 */
14    int frequency[ FREQUENCY_SIZE ] = { 0 };
15
16    /* place survey responses in array responses */
17    int responses[ RESPONSE_SIZE ] = { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10,
18         1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7, 5, 6, 6,
19         5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };
20
```

```
21    /* for each answer, select value of an element of array responses
22       and use that value as subscript in array frequency to
23       determine element to increment */
24    for ( answer = 0; answer < RESPONSE_SIZE; answer++ ) {
25       ++frequency[ responses [ answer ] ];
26    } /* end for */
27
28    /* display results */
29    printf( "%s%17s\n", "Rating", "Frequency" );
30
31    /* output frequencies in tabular format */
32    for ( rating = 1; rating < FREQUENCY_SIZE; rating++ ) {
33       printf( "%6d%17d\n", rating, frequency[ rating ] );
34    } /* end for */
35
36    return 0; /* indicates successful termination */
37
38 } /* end main */
```

Rating	Frequency
1	2
2	2
3	2
4	2
5	5
6	11
7	5
8	7
9	1
10	3


```
1  /* Histogram printing program */
2
3  #include <stdio.h>
4  #define SIZE 10
5
6  int main()
7  {
8     int n[ SIZE ] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };
9     int i, j;
10
11    printf( "%s%13s%17s\n", "Element", "Value", "Histogram" );
12
13    for ( i = 0; i <= SIZE - 1; i++ ) {
14       printf( "%7d%13d        ", i, n[i]) ;
15
16       for ( j = 1; j <= n[ i ]; j++ )   /* print one bar */
17          printf( "%c", '*' );
18
19       printf( "\n" );
20    }
21
22    return 0;
23 }
```


Program Output

```
Element        Value        Histogram
      0           19        *******************
      1            3        ***
      2           15        ***************
      3            7        *******
      4           11        ***********
      5            9        *********
      6           13        *************
      7            5        *****
      8           17        *****************
      9            1        *
```



```
1  /*
2     Roll a six-sided die 6000 times */
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h>
6  #define SIZE 7
7
8  /* function main begins program execution */
9  int main()
10 {
11    int face;                     /* random number with value 1 - 6
*/
12    int roll;                     /* roll counter */
13    int frequency[ SIZE ] = { 0 }; /* initialize array to 0 */
14
15    srand( time( NULL ) ); /* seed random-number generator */
16
17    /* roll die 6000 times */
18    for ( roll = 1; roll <= 6000; roll++ ) {
19       face = rand() % 6 + 1;
20       ++frequency[ face ]; /* replaces 26-line switch of Fig. 5.8 */
21    } /* end for */
22
23    printf( "%s%17s\n", "Face", "Frequency" );
24
```

```
25    /* output frequency elements 1-6 in tabular format */
26    for ( face = 1; face < SIZE; face++ ) {
27       printf( "%4d%17d\n", face, frequency[ face ] );
28    } /* end for */
29
30    return 0; /* indicates successful termination */
31
32 } /* end main */
```

Program Output

```
Face      Frequency
   1           1029
   2            951
   3            987
   4           1033
   5           1010
   6            990
```



```
2     Treating character arrays as strings */
3  #include <stdio.h>
4
5  /* function main begins program execution */
6  int main()
7  {
8     char string1[ 20 ];                /* reserves 20 characters */
9     char string2[] = "string literal"; /* reserves 15 characters */
10    int i;                             /* counter */
11
12    /* read string from user into array string2 */
13    printf("Enter a string: ");
14    scanf( "%s", string1 );
15
16    /* output strings */
17    printf( "string1 is: %s\nstring2 is: %s\n"
18            "string1 with spaces between characters is:\n",
19            string1, string2 );
20
21    /* output characters until null character is reached */
22    for ( i = 0; string1[ i ] != '\0'; i++ ) {
23       printf( "%c ", string1[ i ] );
24    } /* end for */
25
26    printf( "\n" );
27
28    return 0; /* indicates successful termination */
29
30 } /* end main */
```


Program Output

```
Enter a string: Hello there
string1 is: Hello
string2 is: string literal
string1 with spaces between characters is:
H e l l o
```

How to read entire line into string1?

```
fgets (string1, 20, stdin);
```



```
1  /*
2     Static arrays are initialized to zero */
3  #include <stdio.h>
4
5  void staticArrayInit( void );    /* function prototype */
6  void automaticArrayInit( void ); /* function prototype */
7
8  /* function main begins program execution */
9  int main()
10 {
11    printf( "First call to each function:\n" );
12    staticArrayInit();
13    automaticArrayInit();
14
15    printf( "\n\nSecond call to each function:\n" );
16    staticArrayInit();
17    automaticArrayInit();
18
19    return 0; /* indicates successful termination */
20
21 } /* end main */
22
```

```
23 /* function to demonstrate a static local array */
24 void staticArrayInit( void )
25 {
26    /* initializes elements to 0 first time function is called */
27    static int array1[ 3 ];
28    int i; /* counter */
29
30    printf( "\nValues on entering staticArrayInit:\n" );
31
32    /* output contents of array1 */
33    for ( i = 0; i <= 2; i++ ) {
34       printf( "array1[ %d ] = %d  ", i, array1[ i ] );
35    } /* end for */
36
37    printf( "\nValues on exiting staticArrayInit:\n" );
38
39    /* modify and output contents of array1 */
40    for ( i = 0; i <= 2; i++ ) {
41       printf( "array1[ %d ] = %d  ", i, array1[ i ] += 5 );
42    } /* end for */
43
44 } /* end function staticArrayInit */
45
```



```
46 /* function to demonstrate an automatic local array */
47 void automaticArrayInit( void )
48 {
49    /* initializes elements each time function is called */
50    int array2[ 3 ] = { 1, 2, 3 };
51    int i; /* counter */
52
53    printf( "\n\nValues on entering automaticArrayInit:\n" );
54
55    /* output contents of array2 */
56    for ( i = 0; i <= 2; i++ ) {
57       printf("array2[ %d ] = %d  ", i, array2[ i ] );
58    } /* end for */
59
60    printf( "\nValues on exiting automaticArrayInit:\n" );
61
62    /* modify and output contents of array2 */
63    for ( i = 0; i <= 2; i++ ) {
64       printf( "array2[ %d ] = %d  ", i, array2[ i ] += 5 );
65    } /* end for */
66
67 } /* end function automaticArrayInit */
```


First call to each function:

Values on entering **staticArrayInit**:

array1[0] = 0   array1[1] = 0   array1[2] = 0

Values on exiting **staticArrayInit**:

array1[0] = 5   array1[1] = 5   array1[2] = 5

Values on entering **automaticArrayInit**:

array2[0] = 1   array2[1] = 2   array2[2] = 3

Values on exiting **automaticArrayInit**:

array2[0] = 6   array2[1] = 7   array2[2] = 8

Second call to each function:

Values on entering **staticArrayInit**:

array1[0] = 5   array1[1] = 5   array1[2] = 5

Values on exiting **staticArrayInit**:

array1[0] = 10   array1[1] = 10   array1[2] = 10

Values on entering **automaticArrayInit**:

array2[0] = 1   array2[1] = 2   array2[2] = 3

Values on exiting **automaticArrayInit**:

array2[0] = 6   array2[1] = 7   array2[2] = 8

Passing Arrays to Functions

- Passing arrays
  - To pass an array argument to a function, specify the name of the array without any brackets

    ```
    int myArray[24];
    myFunction(myArray, 24);
    ```

    - Array size usually passed to function
  - Arrays passed call-by-reference
  - Name of array is address of first element
  - Function knows where the array is stored
    - Modifies original memory locations
- Passing array elements
  - Passed by call-by-value
  - Pass subscripted name (i.e., `myArray[3]`) to function

Passing Arrays to Functions

- Function prototype

  ```
  void modifyArray( int b[], int arraySize );
  ```

  - Parameter names optional in prototype
    - `int b[]` could be written `int []`
    - `int arraySize` could be simply `int`

Example

```
int sum(int a[], int n)
{
  int j, s=0;
  for (j=0; j <n ; j++)
       s = s+ a[j];
  return s;
}
```

- *Note*: a[] is a notational convenience. In fact

  `int a[] ≡ int *a`

- Calling the function:

```
int total, x[100];

total = sum(x, 100);
total = sum(x, 88);
total = sum(&x[5], 50);
```



```
2  /* Passing arrays and individual array elements to functions */
3  #include <stdio.h>
4  #define SIZE 5
5
6  void modifyArray( int [], int );  /* appears strange */
7  void modifyElement( int );
8
9  int main()
10 {
11    int a[ SIZE ] = { 0, 1, 2, 3, 4 }, i;
12
13    printf( "Effects of passing entire array call "
14            "by reference:\n\nThe values of the "
15            "original array are:\n" );
16
17    for ( i = 0; i <= SIZE - 1; i++ )
18       printf( "%3d", a[ i ] );
19
20    printf( "\n" );
21    modifyArray( a, SIZE );  /* passed call by reference */
22    printf( "The values of the modified array are:\n" );
23
24    for ( i = 0; i <= SIZE - 1; i++ )
25       printf( "%3d", a[ i ] );
26
27    printf( "\n\n\nEffects of passing array element call "
28            "by value:\n\nThe value of a[3] is %d\n", a[ 3 ] );
29    modifyElement( a[ 3 ] );
30    printf( "The value of a[ 3 ] is %d\n", a[ 3 ] );
31    return 0;
32 }
```

```
33
34 void modifyArray( int b[], int size )
35 {
36    int j;
37
38    for ( j = 0; j <= size - 1; j++ )
39       b[ j ] *= 2;
40 }
41
42 void modifyElement( int e )
43 {
44    printf( "Value in modifyElement is %d\n", e *= 2 );
45 }
```

```
Effects of passing entire array call by
reference:

The values of the original array are:
  0  1  2  3  4
The values of the modified array are:
  0  2  4  6  8

Effects of passing array element call by
value:

The value of a[3] is 6
Value in modifyElement is 12
The value of a[3] is 6
```



```
1  /*
2     The name of an array is the same as &array[ 0 ] */
3  #include <stdio.h>
4
5  /* function main begins program execution */
6  int main()
7  {
8     char array[ 5 ]; /* define an array of size 5 */
9
10    printf( "    array = %p\n&array[0] = %p\n"
11       "   &array = %p\n",
12       array, &array[ 0 ], &array );
13
14    return 0; /* indicates successful termination */
15
16 } /* end main */
```

```
    array = 0012FF78
&array[0] = 0012FF78
   &array = 0012FF78
```



```
1  /*
2     Demonstrating the const type qualifier with arrays */
3  #include <stdio.h>
4
5  void tryToModifyArray( const int b[] ); /* function prototype */
6
7  /* function main begins program execution */
8  int main()
9  {
10    int a[] = { 10, 20, 30 }; /* initialize a */
11
12    tryToModifyArray( a );
13
14    printf("%d %d %d\n", a[ 0 ], a[ 1 ], a[ 2 ] );
15
16    return 0; /* indicates successful termination */
17
18 } /* end main */
19
20 /* in function tryToModifyArray, array b is const, so it cannot be
21    used to modify the original array a in main. */
22 void tryToModifyArray( const int b[] )
23 {
24    b[ 0 ] /= 2;    /* error */
25    b[ 1 ] /= 2;    /* error */
26    b[ 2 ] /= 2;    /* error */
27 } /* end function tryToModifyArray */
```

```
error C2166: l-value specifies const object
error C2166: l-value specifies const object
error C2166: l-value specifies const object
```


Sorting Arrays

- Sorting data
  - Important computing application
  - Virtually every organization must sort some data
- Bubble sort (sinking sort)
  - Several passes through the array
  - Successive pairs of elements are compared
    - If increasing order (or identical), no change
    - If decreasing order, elements exchanged
  - Repeat
- Example:
  - original: 3 4 2 6 7
  - pass 1: 3 2 4 6 7
  - pass 2: 2 3 4 6 7
  - Small elements "bubble" to the top

```
1  /*
2     This program sorts an array's values into ascending order */
3  #include <stdio.h>
4  #define SIZE 10
5
6  /* function main begins program execution */
7  int main()
8  {
9     /* initialize a */
10    int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
11    int i;    /* inner counter */
12    int pass; /* outer counter */
13    int hold; /* temporary location used to swap array elements */
14
15    printf( "Data items in original order\n" );
16
17    /* output original array */
18    for ( i = 0; i < SIZE; i++ ) {
19       printf( "%4d", a[ i ] );
20    } /* end for */
21
```



```
22    /* bubble sort */
23    /* loop to control number of passes */
24    for ( pass = 1; pass < SIZE; pass++ ) {
25
26       /* loop to control number of comparisons per pass */
27       for ( i = 0; i < SIZE - 1; i++ ) {
28
29          /* compare adjacent elements and swap them if first
30          element is greater than second element */
31          if ( a[ i ] > a[ i + 1 ] ) {
32             hold = a[ i ];
33             a[ i ] = a[ i + 1 ];
34             a[ i + 1 ] = hold;
35          } /* end if */
36
37       } /* end inner for */
38
39    } /* end outer for */
40
41    printf( "\nData items in ascending order\n" );
42
```

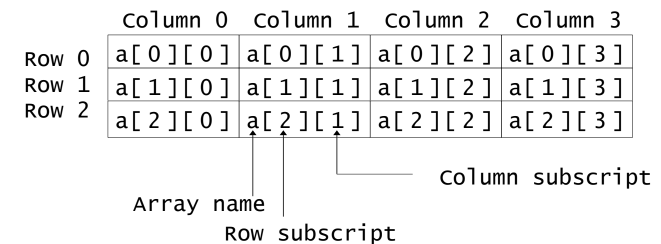


```
43    /* output sorted array */
44    for ( i = 0; i < SIZE; i++ ) {
45       printf( "%4d", a[ i ] );
46    } /* end for */
47
48    printf( "\n" );
49
50    return 0; /* indicates successful termination */
51
```

```
Data items in original order
  2  6  4  8  10  12  89  68  45  37
Data items in ascending order
  2  4  6  8  10  12  37  45  68  89
```


Multi-Dimensional Arrays

- Multiple subscripted arrays
  - Tables with rows and columns (m by n array)
  - Like matrices: specify row, then column



Multi-Dimensional Arrays

- Initialization
  - `int b[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };`
  - Initializers grouped by row in braces
  - If not enough, unspecified elements set to zero

    `int b[ 2 ][ 2 ] = { { 1 }, { 3, 4 } };`

1	2
3	4

1	0
3	4

- Referencing elements
  - Specify row, then column

    `printf( "%d", b[ 0 ][ 1 ] );`


```
1  /*
2     Initializing multidimensional arrays */
3  #include <stdio.h>
4
5  void printArray( const int a[][ 3 ] ); /* function prototype */
6
7  /* function main begins program execution */
8  int main()
9  {
10    /* initialize array1, array2, array3 */
11    int array1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } };
12    int array2[ 2 ][ 3 ] = { 1, 2, 3, 4, 5 };
13    int array3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } };
14
15    printf( "Values in array1 by row are:\n" );
16    printArray( array1 );
17
18    printf( "Values in array2 by row are:\n" );
19    printArray( array2 );
20
21    printf( "Values in array3 by row are:\n" );
22    printArray( array3 );
23
24    return 0; /* indicates successful termination */
25
26 } /* end main */
27
```



```
28 /* function to output array with two rows and three columns */
29 void printArray( const int a[][ 3 ] )
30 {
31    int i; /* counter */
32    int j; /* counter */
33
34    /* loop through rows */
35    for ( i = 0; i <= 1; i++ ) {
36
37       /* output column values */
38       for ( j = 0; j <= 2; j++ ) {
39          printf( "%d ", a[ i ][ j ] );
40       } /* end inner for */
41
42       printf( "\n" ); /* start new line of output */
43    } /* end outer for */
44
45 } /* end function printArray */
```

```
Values in array1 by row are:
1 2 3
4 5 6
Values in array2 by row are:
1 2 3
4 5 0
Values in array3 by row are:
1 2 0
4 0 0
```

```
1  /* Two-dimensional array example
2     */
3  #include <stdio.h>
4  #define STUDENTS 3
5  #define EXAMS 4
6
7  int minimum( int [][ EXAMS ], int, int );
8  int maximum( int [][ EXAMS ], int, int );
9  double average( int [], int );
10 void printArray( int [][ EXAMS ], int, int);
11
12 int main()
13 {
14    int student;
15    int studentGrades[ STUDENTS ][ EXAMS ] =
16       { { 77, 68, 86, 73 },
17         { 96, 87, 89, 78 },
18         { 70, 90, 86, 81 } };
19
20    printf( "The array is:\n" );
21    printArray( studentGrades, STUDENTS, EXAMS );
22    printf( "\n\nLowest grade: %d\nHighest grade: %d\n",
23            minimum( studentGrades, STUDENTS, EXAMS ),
24            maximum( studentGrades, STUDENTS, EXAMS ) );
25
26    for ( student = 0; student <= STUDENTS - 1; student++ )
27       printf( "The average grade for student %d is %.2f\n",
28               student,
29               average( studentGrades[ student ], EXAMS ) );
30
31    return 0;
32 }
```

Each row is a particular student, each column is the grades on the exam.

```
33
34 /* Find the minimum grade */
35 int minimum( int grades[][ EXAMS ],
36              int pupils, int tests )
37 {
38    int i, j, lowGrade = 100;
39
40    for ( i = 0; i <= pupils - 1; i++ )
41       for ( j = 0; j <= tests - 1; j++ )
42          if ( grades[ i ][ j ] < lowGrade )
43             lowGrade = grades[ i ][ j ];
44
45    return lowGrade;
46 }
47
48 /* Find the maximum grade */
49 int maximum( int grades[][ EXAMS ],
50              int pupils, int tests )
51 {
52    int i, j, highGrade = 0;
53
54    for ( i = 0; i <= pupils - 1; i++ )
55       for ( j = 0; j <= tests - 1; j++ )
56          if ( grades[ i ][ j ] > highGrade )
57             highGrade = grades[ i ][ j ];
58
59    return highGrade;
60 }
61
```



```
62 /* Determine the average grade for a particular exam */
63 double average( int setOfGrades[], int tests )
64 {
65    int i, total = 0;
66
67    for ( i = 0; i <= tests - 1; i++ )
68       total += setOfGrades[ i ];
69
70    return ( double ) total / tests;
71 }
72
73 /* Print the array */
74 void printArray( int grades[][ EXAMS ],
75                  int pupils, int tests )
76 {
77    int i, j;
78
79    printf( "                 [0]  [1]  [2]  [3]" );
80
81    for ( i = 0; i <= pupils - 1; i++ ) {
82       printf( "\nstudentGrades[%d] ", i );
83
84       for ( j = 0; j <= tests - 1; j++ )
85          printf( "%-5d", grades[ i ][ j ] );
86    }
87 }
```



```
The array is:
                 [0]  [1]  [2]  [3]
studentGrades[0] 77   68   86   73
studentGrades[1] 96   87   89   78
studentGrades[2] 70   90   86   81

Lowest grade: 68
Highest grade: 96
The average grade for student 0 is 76.00
The average grade for student 1 is 87.50
The average grade for student 2 is 81.75
```


Command Line

- Sometimes it is cleaner to read input from the command line

- `$ ./multiply 4 5`
  `4 times 5 is 20`

- The program, “multiply”, takes two integers and prints the result of multiplying them.

Slide credit: B. Huang

Command Line

```
/* multiply.c – Takes two integers as command line arguments
and displays their product */

#include <stdio.h>

int main(int argc, char *argv[]) /* arguments! */
{
 int a, b, c;

 sscanf(argv[1], "%d", &a);
 sscanf(argv[2], "%d", &b);
 c = a*b;
 printf("%d times %d is %d\n", a, b, c);
 return 0;
}
```

Slide credit: B. Huang

Command Line

- Only run the program "multiply" if **argc** is equal to 3. Otherwise, tell the user that there was a mistake.
  - "argc is equal to 3" is either true or false.

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    int a, b, c;
    if (argc == 3) {
        sscanf(argv[1], "%d", &a);
        sscanf(argv[2], "%d", &b);
        c = a*b;
        printf("%d times %d is %d\n", a, b, c);
    } else {
        printf("Input error\n");
    }

    return 0;
}
```

Slide credit: B. Huang