Arrays and Strings
ENEE 140

Prof. Tudor Dumitraș
Assistant Professor, ECE
University of Maryland, College Park

http://ter.ps/enee140

Today’s Lecture

• Where we’ve been
  – Scalar data types (int, long, float, double, char)
  – Integer and floating point arithmetic
  – Basic control flow (while and if)
  – Functions
  – Random number generation

• Where we’re going today
  – Vector data types: arrays, strings, enums
  – Composite data types: struct
  – Defensive programming and assert()
  – Coding style
  – Project 1 Q&A

• Where we’re going next
  – Complex programs
Scalar vs. Vector Data Types

- We’ve seen
  char, int, long, float, double
  — These are scalar data types: a variable holds a single value

- Vector data types: hold a series of scalar variables of the same type
  — Must specify the size \( N \) of the array

  ```c
  int a[10];  // int array with N=10 elements
  long b[10];  // long array with N=10 elements
  float c[10];  // float array with N=10 elements
  double d[10];  // double array with N=10 elements
  char e[10];  // string with up to 9 characters (!)
  ```

  — Accessing array elements: index between 0 and \( N-1 \)

  ```c
  a[0] = 0;  // first element
  a[9] = 0;  // last element
  ```

Strings

- Strings are character arrays, with some special rules
  — You can initialize strings using string literals (use double quotes)

  ```c
  char s[] = "Hello world\n";  // size of s[] is implicit
  ```

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>S[]</td>
<td>H</td>
<td>e</td>
<td>l</td>
<td>l</td>
<td>o</td>
<td>w</td>
<td>o</td>
<td>r</td>
<td>l</td>
<td>d</td>
<td>\n</td>
<td>\0</td>
<td></td>
</tr>
</tbody>
</table>

  — The character ‘\0’ indicates the end of the string

  ```c
  char s[10];  // must account for ‘\0’ => can only store 9 chars
  ```

  — You can read and write strings using scanf and printf

  - Use the \%s format modifier

  ```c
  char s[] = "Hello world\n";
  printf("The string is: %s", s);
  ```
Initialization vs. Assignment

• Arrays and strings can be initialized, but **can not assigned**
  
  ```
  char s1[] = "ENEE 140";  // s1 is declared and initialized
  char s2[10];            // s2 is declared but not initialized
  s2 = "ENEE 140";        // error! (cannot assign strings)
  ```

• Instead, arrays **can be copied**
  
  ```
  #include <string.h>  // needed for strncpy
  
  strncpy(s2, "ENEE 140", 10);  // must specify the size of s2[]
  ```

Reading Strings

• **scanf**: input string stops at whitespace or at the max field width
  
  ```
  char s[10];
  scanf("%9s", s);  // specify field width 9 to allow for \0 terminator
  ```

• **fgets**: read whole line up to specified size – 1
  
  ```
  fgets(s, 10, stdin);  // stdin is the standard input stream
  ```
  
  – The `\n` character will be included in s[]
  – **fgets()** returns NULL on EOF or error

• Read input line-by-line, until EOF is encountered
  
  ```
  while (fgets(s, 10, stdin) != NULL) { ... }
  ```

• Use a string as input source
  
  ```
  sscanf(s, "%d", &i);  // read integer i from string s[]
  ```
Writing Strings

• printf: use %s format specifier
  ```
  char str[] = "world";
  printf("Hello %s\n", str);
  ```

• fputs: print only the string
  ```
  fputs(str, stdout);
  ```
  stdout is the standard output stream

• Use a string as output:
  ```
  sprintf(str, "%3d", i);
  ```
  write integer i into str[]
  – Important: Must be careful not to exceed the size of str[]!

Common Programming Mistakes

• Accessing or modifying elements outside the array bounds
  – Incorrect
  ```
  int a[10];
  a[-1] = 0;  // index out of bounds
  a[10] = 0;  // index out of bounds
  ```

  – Correct
  ```
  char s[10];
  scanf("%s", s);  // read string of infinite length
  ```

  – This is one of the most common security vulnerabilities in software!!
Defensive Programming

- Good programming practice:
  - Think about relationships among the variables in your program
  - Determine conditions (e.g. \( a == b+1 \)) that must be true at various steps, if your program is correct
  - Force the program to stop when these conditions are violated, then test the program with a variety of inputs to make sure that this doesn’t happen
  - This approach is called “defensive programming”

- Assert: a tool for defensive programming

  ```c
  #include <assert.h>
  assert(condition);
  ```

  exits the program if condition is false

  - Use assert() liberally
  - Assertions allow you to diagnose mistakes in your program
  - They also make your assumptions clear to other programmers who will read your code

Defensive Programming – Example

- Use defensive programming to prevent common mistakes related to arrays and strings

  ```c
  #include <assert.h>

  int a[10];
  assert(i>=0 && i<10);  // exits before accessing index out of bounds
  a[i]=0;
  ```

- Turn off all assertions at compile time

  ```bash
  gcc -DNDEBUG myprogram.c
  ```
The sizeof Operator for Vector Data Types

- Yields the number of bytes required to store the array or string
  - Array dimension x size of base type

```c
char a[10];
int b[10];
sizeof(a) 10
sizeof(b) 40
```

String Functions

- Convenient operations on strings
  ```c
  #include <string.h>
  ```
  ```c
  strlen(s); \hspace{1cm} \text{length of s}
  strncpy(dst, src, n); \hspace{1cm} \text{copy up to n characters from src to dst}
  strncat(dst, src, n); \hspace{1cm} \text{concatenate dst and src}
  strncmp(s1, s2, n); \hspace{1cm} \text{compare s1 and s2}
  ```

- Common programming mistake
  - Using `strcpy`, `strcat`, `strcmp`, etc.
  - These functions do not allow you to specify the size of the destination string
  - **Always use the strn* functions instead of the str* functions!**
enum

- Enumeration constant: list of constant enumeration values
  
  ```
  enum answer {NO, YES};  
  ```
  variables of type answer can take 2 values: NO or YES

  ```
  enum months {JAN=1, FEB, MAR, APR, 
               MAY, JUN, JUL, AUG, 
               SEP, OCT, NOV, DEC}; 
  ```
  FEB is 2, MAR is 3, etc.

  ```
  int current_month = FEB; 
  ```

---

Composite Data Types

- Structures: encapsulate multiple variables
  - May have different types
    ```
    struct cartesian_coord {
    int x; 
    int y; 
    };
    struct polar_coord {
    int radius; 
    float angle; 
    };
    ```

    ```
    struct cartesian_coord a; 
    struct polar_coord b; 
    ```
    variables of composite type

    ```
    b.radius = 1; 
    b.angle = M_PI_2;  
    a.x = b.radius * cos(angle);  
    a.y = b.radius * sin(angle); 
    ```
    accessing members

    π / 2
    0
    1
Coding Style

• Programs are meant to be read by humans
  – Code reviews are a common practice in the industry

• Good coding style makes programs more readable
  – Examples of what not to do: http://www.ioccc.org/

• There is no “right” coding style
  – Choose a style and be consistent

Coding Style: Examples

• Explain what the program does in a comment at the top
• Explain what each function does in comments before the function definition
• Use concise, meaningful names for variables and constants
  – If you have many variables, also add short comments describing the purpose of some of the variables
• Follow normal English rules when possible for better readability of your code
  – Write complete sentences in your comments
  – Leave a space after each comma and semicolon (e.g. in printf(), scanf(), for)
  – Leave a space on each side of a binary operator (e.g. =, ==, +)
• Indent code consistently
  – CLion and Eclipse try to do this automatically
• If you have long, nested {...} blocks, add a comment after the enclosing bracket
  – Indicate which block you are closing (the while block, the if block, etc.)
Coding Style: Examples
• Place braces {} in a consistent manner:
  \[
  \text{for ( } i = 0; \ i < 100; \ i++ \text{) }
    \begin{array}{l}
      \text{statements;  } \\
      \text{OR}
    \end{array}
  \]
  \[
  \text{for ( } i = 0; \ i < 100; \ i++ \\
  \begin{array}{l}
    \text{statements;  } \\
    \end{array}
  \]
  \]
• When you prompt the user for input, first print out a message describing what is expected
• Check for errors and corner-cases throughout the program (more about this later)
• Use simple statements as much as possible
  – Avoid statements like \text{sum = a++ + -b*2}

Review of Lecture
• What did we learn?
  – Arrays
  – Strings
  – String I/O (\texttt{printf} and \texttt{scanf} with \%s)
  – \texttt{enum}
  – \texttt{struct}
  – Coding style
  – Defensive programming
• Next lecture
  – Complex programs
• Assignments for this week
  – Read \texttt{K&R Chapters 4.3, 4.4, 4.5, 4.6, 4.8, 4.9, 4.11}
  – Weekly challenge: \texttt{trim_strings.c}
  – Homework: \texttt{lab07.pdf} (on \url{http://ter.ps/enee140}), due on March 11 at 11:59 pm
  – Quiz 6, due on Monday (after Spring Break) at 11:59 pm
  – Reminder: Project 1 due on Monday, March 21 (after the Spring Break)