Arrays and Strings

Lecture 6
CS 113 – Fall 2007

Announcements

- Assignment 2 posted, due next Friday

Arrays

- To declare an array, use [ ], e.g.:

```
// create an array with 5 integer elements
int A[5];
```

  - Arrays in C are fixed size: their size can't be changed
  - The number between the brackets must be a constant

- You can give initial values for array elements, e.g.:

```
// create an array with 5 integer elements
int A[] = {3, 7, -1, 4, 6};
```

Arrays

- Array indices in C are *zero-based*.
  - e.g. A[0], A[1], A[2], A[3], A[4]

- Example:

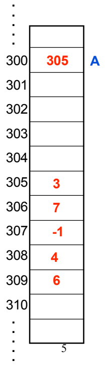
```
int main(void)
{
  int A[] = {3, 7, -1, 4, 6};
  int j;
  double mean = 0;

  // compute mean of values in A
  for(j=0; j<5; j++)
    mean += A[j];

  mean /= 5;
  return 0;
}
```

Array and pointers

- Pointers and arrays are closely related
  - An array variable is actually just a pointer to the *first element in the array*

```
// create an array with 5 integer elements
int A[] = {3, 7, -1, 4, 6};
```

- You can access array elements using array notation or pointers
  - **`A[0]`** is the same as **`*A`**
  - **`A[1]`** is the same as **`*(A+1)`**
  - **`A[2]`** is the same as **`*(A+2)`**
  - etc.



300	305	A
301		
302		
303		
304		
305	3	
306	7	
307	-1	
308	4	
309	6	
310		

Arrays and pointers

- Accessing array elements using pointers:

```
int main(void)
{
  int A[5] = {3, 7, -1, 4, 6};
  int j;
  double mean = 0;

  // compute mean of values in A
  for(j=0; j<5; j++)
    mean += *(A+j);

  mean /= 5;
  return 0;
}
```

Some examples

```
// create an array with 5 integer elements
int A[] = {3, 7, -1, 4, 6};
```

- Q: How to access the integer at index 0 of A?
- A: `A[0]` or `*A`

- Q: How to access the integer at index 3 of A?
- A: `A[3]` or `*(A+3)`

- Q: What is the address of the first element of A?
- A: `A` or `&(A[0])`

- Q: What is the address of the second element of A?
- A: `A+1` or `&(A[1])`

Bounds checking

- What happens when you run this code?

```
int A[5] = {3, 7, -1, 4, 6};
A[28] = 5;
A[-3] = 12;
```

- Unlike most languages, C makes *no attempt* to check for out-of-bounds errors
  - These checks would add overhead at runtime
  - C's philosophy is to generate code that is as fast as possible

Out of bounds error example

```
#include <stdio.h>

int main()
{
   int b = 4;
   int A[]={1,2,3};

   A[7] = 12;
   printf("%d", b);
   return 0;
}
```

```
12
```

Arrays aren't necessary!

- Array syntax is just *syntactic sugar*
  - It's never necessary: you can always use pointers instead
  - But often array syntax is easier to read

- `A[B]` is translated by the compiler into `*(A+B)`
  - e.g. `A[12]` becomes `*(A+12)`
  - either `A` or `B` must be a pointer

- This allows for some unusual expressions
  - `12[A]` is the same as `A[12]`
  - `12[(int *)100]` is the same as `*((int *)112)`
  - avoid these unusual expressions in practice

Passing arrays to functions

```
#include <stdio.h>

int change_and_sum( int *a, int size ) {
  int i, sum = 0;
  a[0] = 100;
  for( i = 0; i < size; i++ )
    sum += a[i];
  return sum;
}

int main() {
  int a[5] = { 0, 1, 2, 3, 4 };
  printf( "Sum of a: %d\n", change_and_sum( a, 5 ));
  printf( "Value of a[0]: %d\n", a[0] );
  return 0;
}
```

Pointer arithmetic

- C lets you perform arithmetic on pointers
  - pointer + integer, pointer – integer
  - pointer++, pointer--
  - Arithmetic is performed based on the type of the pointer
    - e.g. `((char *) 100) + 2` evaluates to address 102, but `((int *) 100) + 2` evaluates to address 108

- Also allowed: pointer comparisons
  - pointer1 < pointer2
  - pointer1 == pointer2
  - etc.

- Multiplication, division not allowed

Pointer arithmetic example

```
#include <stdio.h>

int count( int *a, int size, int target ) {
  int *last, c=0;

  for(last = a + size - 1; a <= last; a++)
    if( *a == target)
      c++;

   return c;
}

int main() {
  int a[5] = { 0, 1, 2, 3, 4 };
  int c = count(a, 5, 4);
  printf("%d\n", c);
  return 0;
}
```

Function pointers

- In C, you can also take the address of a function
  - I.e. the memory location of the function's machine code
- Example of declaring a function pointer:

```
int (*fctnptr)(int, int);
```

  - This is a pointer to a function that takes two parameters of type `int` and returns an integer
- Use the `&` operator to take the address of a function, and the `*` operator to dereference a function pointer

Uses of function pointers

- Systems programming
  - e.g. setting up interrupt vector tables
- Writing generic code
  - A function can take a function as a parameter
  - Example: a generic function to compute integrals

```
double compute_integral(double a, double b, double (*f)(double))
  { /* lots of code */ }

double x_squared(double x) { return x * x; }
double x_cubed(double x) { return x * x * x; }

int main() {
  compute_integral(0, 1, x_squared);
  compute_integral(5, 9, x_cubed);
}
```

Function pointer example

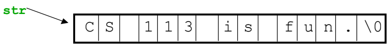
```
#include <stdio.h>

void change_array( int *a, int size, int (*f)(int))
{
  int j;
  for( j=0; j < size; j++)
    a[j] = f(a[j]);
}

int add_one(int x) { return x + 1; }
int square(int x) { return x * x; }

int main()
{
  int a[5] = { 0, 1, 2, 3, 4 };

  change_array(a, 5, add_one); // increment every element
  change_array(a, 5, square);  // square every element
  return 0;
}
```

Other examples

- `qsort()`, `heapsort()`, `mergesort()` are standard library functions for generic sorting
  - defined in `stdlib.h`
  - They take a comparison function as a parameter
  - They can sort *any* type of array, as long as an appropriate comparison function is given
- Also available:
  - `lsearch()` : linear search
  - `bsearch()` : binary search through a sorted array

Strings

- There is no string type in C!
  - Instead, strings are implemented as arrays of characters
  - By convention, strings in C are *zero-terminated*
    - The last character of a string has ASCII code zero ('\0')
  - String constants are written in double quotes
    - C adds the zero automatically
- A string is a pointer to the beginning of a char array
  - And terminated by a zero character
  - e.g. `char *str = "CS 113 is fun."` is stored as:



C	S		1	1	3		i	s		f	u	n	.	\0

String example

```
#include <stdio.h>

// change uppercase letters in str to lowercase
void to_lowercase ( char *str)
{
  for( ; *str ; str++)
    *str = (*str >= ‘A’ && *str <= ‘Z’) ? *str-’A’ : *str;
}

int main()
{
  char message[] = “Five Hundred Twenty-Five Thousand”;

  printf(“%s\n”, message);
  to_lowercase(message);
  printf(“%s\n”, message);
  return 0;
}
```

Built-in string functions

- `string.h` has functions for manipulating strings, e.g.
  - `strlen(char *s)` : returns length of s
  - `strlen(char *s1, char *s2)` : appends s2 to s1
    - s1 must point to enough space to hold the result!
  - `strcpy(char *s1, char *s2)` : copies s2 into s1
    - Again, s1 must point to enough space
  - `strcmp(char *s1, char *s2)`: compares s1 & s2
    - returns 0 if the two strings are equal
    - returns a positive integer if s1 is lexicographically greater than s2
    - returns a negative integer if s1 is lexicographically less than s2

String headaches

- Remember that you are responsible for allocating enough space for strings!
  - Unlike most other languages

```
// BAD code
int main() {
  char s1[] = “Any person, ”;
  char s2[] = “any study.”

  strcat(s1, s2);
  printf(“%s\n”, s1);
}
```

String headaches

- Remember that you are responsible for allocating enough space for strings!
  - Unlike most other languages

```
// still bad code
int main() {
  char s1[] = “Any person, ”;
  char s2[] = “any study.”
  char s3[1024];

  strcat(s3, s1);
  strcat(s3, s2);
  printf(“%s\n”, s3);
}
```

String headaches

- Remember that you are responsible for allocating enough space for strings!
  - Unlike most other languages

```
// better code
int main() {
  char s1[] = “Any person, ”;
  char s2[] = “any study.”
  char s3[1024];

  strcpy(s3, s1);
  strcat(s3, s2);
  printf(“%s\n”, s3);
}
```

More string headaches

- Idea: what if we create a wrapper function for strcat?
  - What goes wrong here?

```
char *my_strcat(char *s1, char *s2) {
  char s3[1024];
  strcpy(s3, s1);
  strcat(s3, s2);
  return s3;
}

int main() {
  char s1[] = “hello”, s2[] = “world”;
  char *result = my_strcat(s1, s2);
  printf(“%s\n”, result);
  return 0;
}
```

String I/O functions

- printf() and scanf() have a **%s** placeholder for string I/O

```
#include <stdio.h>

int main()
{
  char string[1024];

  scanf(“%s”, string);
  printf(“You entered: %s\n”, string);

  return 0;
}
```

  - Note: no **&** before string in argument to scanf

Buffer overruns

- What if someone enters more than 1024 characters?
  - Remember: C doesn’t check for array out-of-bounds errors
  - scanf copies the input to memory, past the end of the space allocated for the string

- What happens is undefined. It could:
  - write to another program’s memory (the O/S will kill it)
  - change the values of other variables
  - change the program’s machine code

Buffer overruns == security catastrophe

- An “evildoer” can purposely enter a clever string, that
  - is too long for the buffer
  - contains machine code

- The program will then start running the new machine code!

- Example: Morris Internet worm, 1988
  - Servers expected to be sent a name of not more than 512 characters
  - But they were written in C and didn’t check for buffer overruns
  - The Internet worm took advantage of this

- Other worms: Code Red (2001), Blaster (2003), SQL Slammer (2003)
  - Tens of billions of dollars of damage!

A simple fix

- Make sure buffers are big enough!
  - e.g. specify a maximum width to scanf

```
#include <stdio.h>

int main()
{
  char string[1024];

  scanf(“%1023s”, string);
  printf(“You entered: %s\n”, string);

  return 0;
}
```