Pointers, Arrays, and Strings

CS449 Spring 2016

Pointers

- Pointers are important.
- Pointers are fun!

Pointers

Every variable in your program has a memory location.
This location can be accessed using **&** operator.

```
#include <stdio.h>
int main () {
    int  var1;
    printf("Address of var1 variable: %x\n", &var1  );
    return 0;
}
```

```
>> ./a.out
Address of var1 variable:
ffbff854
```

Pointers

A **pointer** is a variable whose value is the address of another variable.

Pointer **declaration**:
type *var-name;

Pointers

Pointer **declaration**:

```
type *var-name;
```

Declaration **examples**:

```
int *count;
double *ratio;
char *symbol;
```

Pointer Usage

```
#include <stdio.h>
int main () {
   int  var = 20;     /* actual variable declaration */
   int  *p;                 /* pointer variable declaration */
   p = &var;          /* store address of var in pointer variable*/

   printf("Address of var variable: %x\n", &var  );
   printf("Address stored in p variable: %x\n", p );
   printf("Value of *p variable: %d\n", *p );
   return 0;
}
```

```
>> ./a.out
Address of var variable: bffd8b3c
Address stored in p variable: bffd8b3c
Value of *p variable: 20
```

NULL Pointers

Good practice!

```
int  *ptr = NULL;
printf("The value of ptr is : %x\n", ptr  );
```

```
>> ./a.out
The value of ptr is 0
```

To Test:

```
if(ptr)     /* succeeds if p is not null */
if(!ptr)    /* succeeds if p is null */
```

Arrays

- Data type for a sequence of variables of a given type in consecutive memory

- Just as for pointers, there are array types for each primitive data type (e.g. char, int, float)

Running Example

```c
#include <stdio.h>

int main()
{
  int nums[5], i;                    /* declarations */
  printf("Enter nums: ");
  for(i=0; i<5; ++i) {
    scanf("%d", &nums[i]);     /* write to array */
  }
  printf("Your nums: ");
  for(i=0; i<5; ++i) {
    printf("%d ", nums[i]);       /* read from array */
  }
  return 0;
}
```

```
>> ./a.out
Enter nums: 10 20 30 40 50
Your nums: 10 20 30 40 50
```

Declaring Arrays

```c
#include <stdio.h>

int main()
{
 int nums[5], i;                    /* declarations */
 printf("Enter nums: ");
 for(i=0; i<5; ++i) {
  scanf("%d", &nums[i]);     /* write to array */
 }
 printf("Your nums: ");
 for(i=0; i<5; ++i) {
  printf("%d ", nums[i]);      /* read from array */
 }
 return 0;
}
```

- Syntax: <type> <name> [<length>]
  - E.g. “int num[5];”, “char c[10];”
- Combination of type and length tells the compiler amount of memory to reserve (sizeof(type) * length)
  - Length must always be declared (explicitly or implicitly)
- Array initializers
  - Handy way to initialize elements of array
  - E.g. “int num[3] = {1, 2, 3};”
  - If initializer shorter than length, rest initialized to 0 (e.g. “int num[3] = {1};”, “int num[3] = {};”)
  - Initializers implicitly gives size of array (e.g. “int num[] = {1, 2, 3};”
  - Initialized arrays can still be modified

Accessing Arrays

```c
#include <stdio.h>

int main()
{
  int nums[5], i;                    /* declarations */
  printf("Enter nums: ");
  for(i=0; i<5; ++i) {
    scanf("%d", &nums[i]);     /* write to array */
  }
  printf("Your nums: ");
  for(i=0; i<5; ++i) {
    printf("%d ", nums[i]);       /* read from array */
  }
  return 0;
}
```

- Syntax: <name> [<index>]
  - E.g. “nums[5]”, “c[10]”
- Index starts with 0
  - “nums[i]” accesses the “i+1 th” element
- Value of array name with no index (e.g. “nums”) is the address of first element of array (equivalent to “&nums[0]”)
  - Is interchangeable with pointer
  - “int *p = nums;” is perfectly valid
  - “*p == nums[0]”
- Difference between array and pointer
  - Array has allocated memory statically bound to the name at compile time
  - “nums” cannot point to new address (e.g. “nums = p” results in compile error)

Declaring Multidimensional Arrays

- Syntax: <type> <name> [<length1>] [<length2>] … [<lengthN>]
  - E.g. “int nums[2][3];”
- Array initializers
  - E.g. “int nums[2][3] = { {0, 1, 2}, {3, 4, 5} };”
- Conceptual layout

	Column 0	*Column 1*	*Column 2*
Row 0	0	1	2
Row 1	3	4	5

- Physical layout in linear memory

0	**1**	**2**	**3**	**4**	**5**
[0][0]	**[0][1]**	**[0][2]**	**[1][0]**	**[1][1]**	**[1][2]**

Accessing Multidimensional Arrays

- Syntax: <type> <name> [<index1>] [<index2>] … [<indexN>]
  - E.g. to access nums[1][2]; we offset (1 * 3 + 2) = 5 in linear memory

- If we do: **int (*p)[3] = nums;**
  - then p[1][2] == nums[1][2] == 5
- If we do: **int *p = nums[1];** then "p[2] == 5"

- "nums" cannot be target of assignment

Be careful:

int (*p)[3] is a pointer to a 3 column array

int *p[3] is an array of 3 pointers!!!

Strings

- There is no “string” data type in C
- Sequence of characters in consecutive memory ending with a null character (‘\0’);
  - Null character: character constant with ASCII value 0
  - In short, a string is a null-terminated character array
- String variables are declared as character arrays
  - e.g. “char s[10];” can hold a string 9 characters long (excluding the null character)
  - Character pointers (“char *”) can point to strings
  - “const char*” points to an immutable string

String Functions

- Defined in C standard library, declared in <string.h>

Prototype	Description
size_t strlen(const char *s);	Calculates the length of the string s, not including the terminating '\0' character.
int strcmp(const char *s1, const char *s2);	Compares the string pointed to by str1 to the string pointed to by str2. If return value < 0 then str1 is less than str2, if it is > 0 then str2 is less than str1, and if it is = 0 then str1 is equal to str2.
int strncmp(const char *s1, const char *s2, size_t n);	Same as above, except only compares the first (at most) n characters of s1 and s2

String Functions

Prototype	Description
char *strcpy(char *dest, const char *src);	Copies the string pointed to by src (including the terminating '\0' character) to the array pointed to by dest.
char *strncpy(char *dest, const char *src, size_t n);	Same as above, except that not more than n bytes of src are copied.
char *strcat(char *dest, const char *src);	Appends the src string to the dest string overwriting the '\0' character at the end of dest, and then adds a terminating '\0' character.
char *strncat(char *dest, const char *src, size_t n);	Same as above, except that it will use at most n characters from src.

- Always try to use the “n” version to prevent buffer overruns

String Functions

Prototype	Description
char *strchr(const char *s, int c);	Returns a pointer to the first occurrence of the character c in the string s.
char *strstr(const char *haystack, const char *needle);	Finds the first occurrence of the substring needle in the string haystack.

String Conversion Functions

- String to number conversion functions declared in <stdlib.h>

Prototype	Description
int atoi(const char *nptr);	Converts string pointed to by nptr to int.
double atof(const char *nptr);	Converts string pointed to by nptr to double.

- String formatting functions declared in <stdio.h>

Prototype	Description
int sprintf(char *str, const char *format, ...);	Same as printf except instead of writing to stdout formatted string is written to str
int snprintf(char *str, size_t size, const char *format, ...);	Same as above, except no more than size bytes are written to str

- Not an exhaustive list. Use your manpages. (e.g. “man string”, “man stdio”)

Pitfall 1: Arrays

- What's wrong with the following code?

```
int a[5];
a[5] = 0;
```

- "a[5]" is attempting to get a value from unallocated memory so result is undefined
- **Remember index always starts with 0**

Pitfall 2: String buffer allocation

- What's wrong with the following code?

```
char *s1;
char *s2 = "Some long string of characters";
strcpy(s1, s2);
```

- "s1" points to random unallocated memory
- Change pointer to array: "char s1[100];"
- Additionally, change strcpy to strncpy:

```
char s1[100];
strncpy(s1, s2, 100);  // to prevent buffer overflow
s1[99] = '\0';  // in case s2 is longer than 100 chars
```

Pitfall 3: String comparison

- What's wrong with the following code?

```
char s[100];
strcpy(s, "Hello");
if(s == "Hello") printf("Hello\n");
```

- The value of "s" is just the address &s[0]
- Should do instead:

```
if(strcmp(s, "Hello") == 0) printf("Hello\n");
```

Pitfall 2: String buffer allocation

- What's wrong with the following code?

```
char *s1;
char *s2 = "Some long string of characters";
strcpy(s1, s2);
```

- "s1" points to random unallocated memory
- Change pointer to array: "char s1[100];"
- Additionally, change strcpy to strncpy:

```
char s1[100];
strncpy(s1, s2, 100);  // to prevent buffer overflow
s1[99] = '\0';  // in case s2 is longer than 100 chars
```

Review of Data Types

- Primitive data types
  - integers: char, short, int, long, long (signed and unsigned)
  - reals: float, double, long double
- Pointers (derived type that points to another type)
  - char *p, int *p, int (*p)[3], int (*p)[2][3], int **p
- Arrays (derived type that is a sequence of a given type)
  - char a[3], int a[3], int a[2][3], int *a[3] (same as int *(a[3]))
- Given array int a[2][3][4], the following are valid
  - int (*p)[3][4] = a; (same as int (*p)[3][4] = &[a];)
  - int (*p)[4] = a[1]; (same as int (*p)[4] = &a[1][0];)
  - int *p = a[1][0]; (same as int *p = &a[1][0][0])
  - int p = a[1][0][2];

Pitfall 4: No L-Value

- What's wrong with the following code?

```
int n;
int **p = &(&n);
```

- &n has no storage location (a.k.a. l-value)
- The following code is valid:

```
int n;
int *p = &n;
int **p2 = &(p);
```

- By the same token, the following is invalid

```
int a[3], b[3];
a = b;  // a is not an l-value
```