CS2002
Computer Systems
Lecture 4

Pointers

Jon Lewis (JC 0.26)

School of Computer Science

University of St Andrews

Overview

- Pointers
  - Memory addresses, obtaining an address in C
  - Declaring variables to contain pointers, dereferencing pointers
  - Pointers to arrays, pointer arithmetic
  - Pointers vs. arrays
  - Strings revisited

Memory Addresses

- Each memory location has an ***address***
- A 32-bit machine has a 32-bit ***address space***, ranging from 0 to 2^32-1 (`0xffffffff`). There may not be enough real (***physical***) memory for all addresses - 4GB.
- Each address usually refers to ***one byte*** of memory (the architecture is ***byte-addressable)***

- On Unix, each ***process*** has its own 32-bit (or 64-bit) virtual address space, mapped to real memory by hardware/software.

Addresses in C

- Every variable in C is stored at some memory location.
- Use the & operator to get a variable's address!

```
void show_addrs() {
  int i = 1;
  double d = 2.0;
  printf("i is %3i @ %p\n", i, &i);
  printf("d is %1.1f @ %p\n", i, &d);
}
```

One possible output is:

```
i is   1 @ 0x7fff5fbff73c
d is 2.0 @ 0x7fff5fbff730
```

Pointers - *

- An `int` variable stores an integer value.
- An `int*` variable stores the memory location of an `int`.
- These variables act exactly like other C variables:
  - Can be assigned to, compared.
  - Can be passed by value to functions.
- In general, X* stores the address of an object of type X.
  - `int*` stores the address of an `int`
  - `int**` stores the address of an `int*` !

Pointer Example

```
int main(void) {
  int i = 1025;
  int *ip = &i;
  printf("i is %d at %p", i, ip);
}
```

`int *ip` same as `int* ip`

FB30	FB31	FB32	FB33	FB34	FB35	FB36	FB37
0	0	4	1	0	0	FB	30

i ip

Dereferencing

- So far pointers are not very useful.
- We need the opposite of the '&' operator.

- Given a variable `var_name` of type X,
  - `&var_name` gives the address of the X-type variable, i.e. returns an X* or "pointer of type X"

- Given a variable `var_name` of type `X*`,
  - *var_name gives the value of type X that var_name points to (i.e. is "stored at the address in var_name")

Pointer Example

```
int main() {
  int i = 4, j = 6;
  int *ip = &i; // pointer definition and assignment (int *ip; ip = &i;)
  printf("i = %d @ %p, j = %d @ %p, ip = %p\n", i, &i, j, &j, ip);
  // i = 4 @ 0x7ffee0227a88, j = 6 @ 0x7ffee0227a84, ip = 0x7ffee0227a88

  *ip = 10; // pointer dereference
  printf("i = %d, j = %d. *ip = %d, ip = %p\n", i, j, *ip, ip);
  // i = 10, j = 6. *ip = 10, ip = 0x7ffee0227a88

  ip = &j; // pointer assignment
  printf("i = %d, j = %d. *ip = %d, ip = %p\n", i, j, *ip, ip);
  // i = 10, j = 6. *ip = 6, ip = 0x7ffee0227a84

  *ip = 20; // pointer dereference
  printf("i = %d, j = %d. *ip = %d, ip = %p\n", i, j, *ip, ip);
  // i = 10, j = 20. *ip = 20, ip = 0x7ffee0227a84

  return 0;
}
```

Pointers and arrays

```
int array[6];
int *ip;

// The following three lines are equivalent
ip = &(array[0]);
ip = &array[0];
ip = array;
```

Pointers vs. Arrays

Given:

```
int array[100];
int *p2array = array;
```

Arrays can be accessed as if they are variables of pointer to X!

```
array[i]  == *(array + i) == *(p2array + i) == p2array[i]
```

also,

```
&array[i] == array + i == p2array + i == &(p2array[i])
```

also,

```
&array[0] == array + 0   == array == p2array
                         == p2array + 0 == &(p2array[0])
```

However there is **no** variable associated with an array, thus assignments like

`p2array = array;` and `p2array++;` are legal

whereas these assignments

`array = p2array;` and `array++;` are **not legal**

sizeof

- `sizeof` gives the size of a C object.
  - `sizeof(char) == 1` (always)
  - `sizeof(int)  == 4` (usually)

- Size of a pointer is (usually) the 'width' of the processor
  - `sizeof(int*) == 8 (on 64-bit)`
  - `sizeof(int*) == 4 (on 32-bit)`

- This is the one place where arrays differ from pointers.

```
int array[5];
int* parray = array;
sizeof(int) == 4;
sizeof(parray) == 8; (on 64-bit, 4 on 32-bit)
sizeof(array) == 20; // 4 * 5;
```

sizeof(2)

- As soon as the array decays into a pointer, you cannot get the array back.
- Note that using: `int a[]` and `int *a` in a function declaration are interchangeable.

```
void print_size(int a[], int *b) {
  printf("%li, %li\n", sizeof(a), sizeof(b));
}
void someFunction() {
  int array[5];
  print_size(array, array); // 8, 8
}
```

Passing arrays to functions

- Arrays decay to a pointer (base address of array) when they are passed to a function
  - you can’t find out the size
  - so it is common to include an ***int length*** argument to the function so that the function knows how many elements are in the array
  - alternatively, the array must be terminated with a suitable value, e.g. ‘\0’ for character arrays (strings)

Array/Pointer example

```
//                     int ip[] is OK too
void init_int_array (int* ip, int length) {
    for (int i = 0; i < length; i++) {
      ip[i] = i;
    }
}

int main() {
  int ibuf1[10], ibuf2[5];
  init_int_array(&ibuf1[0], 10);
  init_int_array(ibuf2, 5);
  return 0;
}
```

Pointer Arithmetic

- Array indexing and pointer arithmetic work identically.

`a[i]`	is equivalent to	`*(a+i)`
`&a[i]`	is equivalent to	`a+i`

Pointer Arithmetic

- An array offset is calculated by adding the the offset value times the array element size, to the base address
- Pointer arithmetic goes in 'chunks' of sizeof(type). In general, this is what you want!

```c
int main() {
  int array[] = { 1, 4, 9, 16 };
  int *ap = array;
  int i;
  for(i = 0; i < 4; i++ )
    *(ap + i) -= 1;
  return 0;
}
```

array		
0	0x10000	&array[0] or (array + 0)
3	0x10004	&array[1] or (array + 1)
8	0x10008	&array[2] or (array + 2)
15	0x1000c	&array[3] or (array + 3)

Dangling Pointers

- Stack-based objects disappear whenever you get to the closing bracket after they were defined.
  - They disappear exactly where you can't refer to them by their original name any more.

- At this point, any pointers to these objects become invalid - trying to read or write to their values is undefined behaviour.
  - Re-entering a function makes new variables (unless they are static).

```
int f() {
  int i = 0;
  while(i++ < 10) {
    int j;
    j = ... //
  } // <- j out of scope here
} // <- i out of scope here
```

RETURNING DANGLING POINTER (NEVER DO THIS)

local_char_array_return.c on studres

Strings Revisited

- C strings are just arrays of chars!

```
char str[] = "Hello";

// makes a string lower case
void mklower(char *str) {
  while (*str != '\0') {
    *str = tolower(*str);
    str += 1;
  }
}
```

<string.h>

- Contains functions for working with strings

```
// the length of s (discounting \0)
int strlen(const char *s);

// compare s1 and s2, return 0 if equal
int strcmp(const char *s1, const char *s2);

// copy string s2 to s1 (like “s1 = s2”)
char *strcpy(char *s1, const char *s2);

// append s2 to s1 (like “s1 += s2”)
char *strcat( char *s1, const char *s2 );
```

<string.h>

- Remember, none of these functions do any memory allocation.

- Example: It is your job to make sure s1 points into a char array with enough space to store a copy of s2!

```
// copy string s2 to s1 (like “s1 = s2”)
char *strcpy(char *s1, const char *s2);
```

I/O with Strings

There are several I/O library functions for strings

```
char *gets(char *s);
```
Read input into s until \n or EOF read. Returns s.

gets is very dangerous, it can overflow your array!
Later we will learn about the much safer fgets.

```
int puts(const char *s);
```
print s followed by a newline. Same as printf("%s\n", s);

```
int sprintf(char *s, const char *format,…);
```
Like printf, but takes an extra first parameter 's' and writes to it.

```
int sscanf(char *s, const char *format,…);
```
Like scanf, but reads from s.

Example use

```
void show_path(char *dir, char *file) {
  char path[256]; // Lets choose a big number

  if(strcmp(dir,"") != 0) {// check if string is empty
   strcpy(path, dir); // path ← dir
   strcat(path, "/"); // path += '/’
  }

  strcat(path, file); // path += file
  printf("The full path is %s\n", path);
}
```

String Copying

`strcpy` copies its second argument *into* its first *in situ.*

Read like "s1 = s2"

s1 better have enough space to store s1!

```
void strcpy(char s1[], const char s2[]) {
 int i;

 for (i = 0; s2[i] != '\0'; i++) {
    s1[i] = s2[i];
 }

 s1[i] = '\0';
}
```

String Concat

`strcat` appends its second argument to its first *in situ.*
Read like "s1 += s2"
s1 better have enough space to store new s1 at the end!

```
void strcat(char s1[], const char s2[]) {
 int i;

 for (i = 0; s1[i] != '\0'; i++);

 for (int j = 0; s2[j] != '\0'; i++, j++) {
   s1[i] = s2[j];
 }

 s1[i] = '\0';
}
```