Programming in C and C++

3. Pointers and Structures

Dr. Neel Krishnaswami
University of Cambridge
(based on notes from and with thanks to Anil Madhavapeddy, Alan Mycroft, Alastair Beresford and Andrew Moore)

Michaelmas Term 2016-2017
Pointers

- Computer memory is often abstracted as a sequence of bytes, grouped into words
- Each byte has a unique address or index into this sequence
- The size of a word (and byte!) determines the size of addressable memory in the machine
- A pointer in C is a variable which contains the memory address of another variable (this can, itself, be a pointer)
- Pointers are declared or defined using an asterisk (*); for example:
  ```c
  char *pc; or int **ppi;
  ```
- The asterisk binds to the variable name, not the type specifier; for example `char *pc, c;`
- A pointer does not necessarily take the same amount of storage space as the type it points to
Example
Manipulating pointers

- The value “pointed to” by a pointer can be “retrieved” or dereferenced by using the unary * operator; for example:
  
  ```c
  int *p = ...;
  int x = *p;
  ```

- The memory address of a variable is returned with the unary ampersand (&) operator; for example
  
  ```c
  int *p = &x;
  ```

- Dereferenced pointer values can be used in normal expressions; for example: *pi += 5; or (*pi)++
Example

```c
#include <stdio.h>

int main(void) {
    int x=1,y=2;
    int *pi;
    int **ppi;

    pi = &x; ppi = &pi;
    printf("%p, %p, %d=%d=%d\n",ppi,pi,x,*pi,**ppi);
    pi = &y;
    printf("%p, %p, %d=%d=%d\n",ppi,pi,y,*pi,**ppi);

    return 0;
}
```
Pointers and arrays

- A C array uses consecutive memory addresses without padding to store data.
- An array name (used in an expression without an index) represents the memory address of the first element of the array; for example:
  ```c
  char c[10];
  char *pc = c; is the same as
  char *pc = &c[0];
  ```
- Pointers can be used to “index” into any element of an array; for example:
  ```c
  int i[10];
  int *pi = &i[5];
  ```
Pointer arithmetic

- Pointer arithmetic can be used to adjust where a pointer points; for example, if \( pc \) points to the first element of an array, after executing \( pc+=3; \) then \( pc \) points to the fourth element.

- A pointer can even be dereferenced using array notation; for example \( pc[2] \) represents the value of the array element which is two elements beyond the array element currently pointed to by \( pc \).

- In summary, for an array \( c \), \( *(c+i) \equiv c[i] \) and \( c+i \equiv &c[i] \).

- A pointer is a variable, but an array name is not; therefore \( pc=c \) and \( pc++ \) are valid, but \( c=pc \) and \( c++ \) are not.
Example

```c
#include <stdio.h>

int main(void) {
    char str[] = "A string."
    char *pc = str;

    printf("%c %c %c\n", str[0], *pc, pc[3]);
    pc += 2;
    printf("%c %c %c\n", *pc, pc[2], pc[5]);

    return 0;
}
```
Pointers as function arguments

- Recall that all arguments to a function are copied, i.e. **passed-by-value**; modification of the local value does not affect the original.

- In the second lecture we defined functions which took an array as an argument; for example `void reverse(char s[])`.

- Why, then, does `reverse` affect the values of the array after the function returns (i.e. the array values haven’t been copied)?
  - because `s` is re-written to `char *s` and the caller implicitly passes a pointer to the start of the array.

- Pointers of any type can be passed as parameters and return types of functions.

- Pointers allow a function to alter parameters passed to it.
Example

Compare `swp1(a,b)` with `swp2(&a,&b)`: 

```c
1 void swp1(int x, int y) 1 void swp2(int *px, int *py)
2 { 2 {
3     int temp = x; 3     int temp = *px;
4     x = y; 4     *px = *py;
5     y = temp; 5     *py = temp;
6 } 6 }
```
Arrays of pointers

- C allows the creation of arrays of pointers; for example
  ```c
  int *a[5];
  ```
- Arrays of pointers are particularly useful with strings
- An example is C support of command line arguments:
  ```c
  int main(int argc, char *argv[]) { ... }
  ```
- In this case `argv` is an array of character pointers, and `argc` tells the programmer the length of the array
Example

argv:

argv[0] argv[3]

argv[2]

argv[1]

argv[0]: progname\0

argv[1]: firstarg\0

argv[2]: secondarg\0

argv[3]: NULL

argc: 3
Multi-dimensional arrays

- Multi-dimensional arrays can be declared in C; for example:
  ```c
  int i[5][10];
  ```

- Values of the array can be accessed using square brackets; for example: `i[3][2]`

- When passing a two dimensional array to a function, the first dimension is not needed; for example, the following are equivalent:
  ```c
  void f(int i[5][10]) { ... }
  void f(int i[][10]) { ... }
  void f(int (*i)[10]) { ... }
  ```

- In arrays with higher dimensionality, all but the first dimension must be specified
Pointers to functions

- C allows the programmer to use pointers to functions
- This allows functions to be passed as arguments to functions
- For example, we may wish to parameterise a sort algorithm on different comparison operators (e.g. lexicographically or numerically)
- If the sort routine accepts a pointer to a function, the sort routine can call this function when deciding how to order values
Example

```c
void sort(int a[], const int len,
          int (*compare)(int, int))
{
    int i,j,tmp;
    for(i=0;i<len-1;i++)
        for(j=0;j<len-1-i;j++)
            if ((*compare)(a[j],a[j+1]))
                tmp=a[j], a[j]=a[j+1], a[j+1]=tmp;
}

int inc(int a, int b) {
    return a > b ? 1 : 0;
}
```

Source of some confusion: either or both of the *s in *compare may be omitted due to language (over-)generosity.
Example

```c
#include <stdio.h>
#include "example8.h"

int main(void) {
    int a[] = {1,4,3,2,5};
    unsigned int len = 5;
    sort(a,len,inc); //or sort(a,len,&inc);

    int *pa = a; //C99
    printf("[");
    while (len--)
        printf("%d%s",*pa++,len?" ":");
    printf("]\n");

    return 0;
}
```
The void * pointer

- C has a “typeless” or “generic” pointer: `void *p`
- This can be a pointer to any object (but not legally to a function)
- This can be useful when dealing with dynamic memory
- Enables “polymorphic” code; for example:

```c
1 sort(void *p, const unsigned int len,
2     int (*comp)(void *,void *));
```

- However this is also a big “hole” in the type system
- Therefore `void *` pointers should only be used where necessary
Structure declaration

- A structure is a collection of one or more members (fields)
- It provides a simple method of abstraction and grouping
- A structure may itself contain structures
- A structure can be assigned to, as well as passed to, and returned from functions
- We declare a structure using the keyword `struct`
- For example, to declare a structure `circle` we write
  ```c
  struct circle {int x; int y; unsigned int r;};
  ```
- Declaring a structure creates a new type
Structure definition

- To define an instance of the structure `circle` we write
  ```c
  struct circle c;
  ```
- A structure can also be initialised with values:
  ```c
  struct circle c = {12, 23, 5};
  ```
- An automatic, or local, structure variable can be initialised by function call:
  ```c
  struct circle c = circle_init();
  ```
- A structure can be declared, and several instances defined in one go:
  ```c
  struct circle {int x; int y; unsigned int r;} a, b;
  ```
Member access

- A structure member can be accessed using `.` notation: `structname.member`; for example: `vect.x`
- Comparison (e.g. `vect1 > vect2`) is undefined
- Pointers to structures may be defined; for example:
  ```c
  struct circle *pc
  ```
- When using a pointer to a struct, member access can be achieved with the `.` operator, but can look clumsy; for example: `(*pc).x`
- Equivalently, the `->` operator can be used; for example: `pc->x`
Self-referential structures

- A structure declaration cannot contain itself as a member, but it can contain a member which is a pointer whose type is the structure declaration itself
- This means we can build recursive data structures; for example:

```c
struct tree {
    int val;
    struct tree *left;
    struct tree *right;
}
```

```c
struct link {
    int val;
    struct link *next;
}
```
Unions

- A union variable is a single variable which can hold one of a number of different types
- A union variable is declared using a notation similar to structures; for example: `union u { int i; float f; char c;};`
- The size of a union variable is the size of its largest member
- The type held can change during program execution
- The type retrieved must be the type most recently stored
- Member access to unions is the same as for structures (`.‘ and `->‘)
- Unions can be nested inside structures, and vice versa
Bit fields

- Bit fields allow low-level access to individual bits of a word
- Useful when memory is limited, or to interact with hardware
- A bit field is specified inside a struct by appending a declaration with a colon (`:`) and number of bits; for example:
  ```c
  struct fields { int f1 : 2; int f2 : 3;};
  ```
- Members are accessed in the same way as for structs and unions
- A bit field member does not have an address (no `&` operator)
- Lots of details about bit fields are implementation specific:
  - word boundary overlap & alignment, assignment direction, etc.
Example (adapted from K&R)

```c
struct { /* a compiler symbol table */
    char *name;
    struct {
        unsigned int is_keyword : 1;
        unsigned int isExtern : 1;
        unsigned int isStatic : 1;
        ...
    } flags;
    int utype;
    union {
        int ival; /* accessed as symtab[i].u.ival */
        float fval;
        char *sval;
    } u;
} symtab[NSYM];
```
Exercises

1. If \( p \) is a pointer, what does \( p[-2] \) mean? When is this legal?

2. Write a string search function with a declaration of
   \[
   \text{char *strfind(const char *s, const char *f);} \\
   \text{(and NULL otherwise)}
   \]
   which returns a pointer to first occurrence of the string \( s \) in the string \( f \)

3. If \( p \) is a pointer to a structure, write some C code which uses all the
   following code snippets: “++p->i”, “p+++i”, “*p->i”, “*p->i++”,
   “(*p->i)++” and “*p+++i”; describe the action of each code snippet

4. Write a program \texttt{calc} which evaluates a reverse Polish expression given on
   the command line; for example
   \[
   $ \text{calc 2 3 4 + *}$
   \]
   should print \texttt{14} (K&R Exercise 5-10)