Programming in C and C++

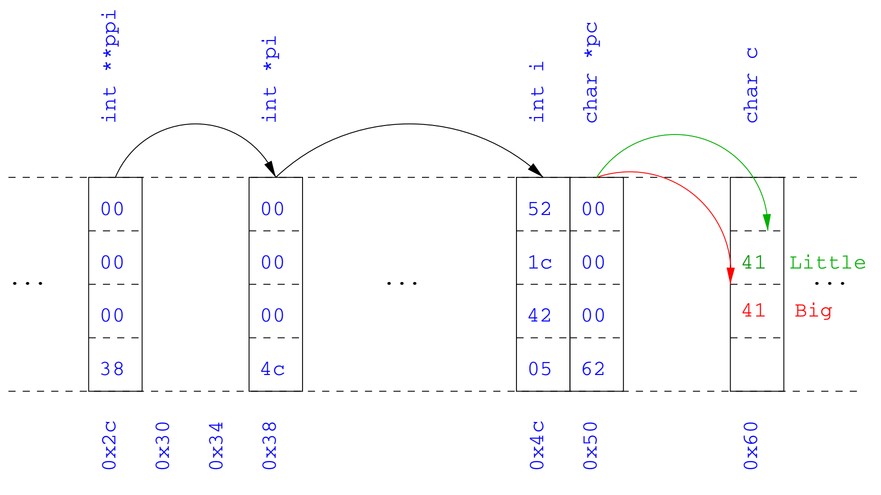
Lecture 3: Pointers and Structures

Neel Krishnaswami and Alan Mycroft

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: `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



int **ppi
int *pi
int i
char *pc
char c
...
00
00
00
38
00
00
00
4c
...
52
1c
42
05
00
00
00
62
41
Little
41
Big
...
0x2c
0x30
0x34
0x38
0x4c
0x50
0x60

Manipulating pointers

- The value pointed to by a pointer can be retrieved or dereferenced by using the unary * operator; for example:

```
int *p = ...
int x = *p;
```

- The memory address of a variable is returned with the unary ampersand (&) operator; for example `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;       // This is the same
char *pc = &c[0];   // as this
```

- 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) == c[i]` and `c+i == &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

Pointer Arithmetic 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 havent 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)`:

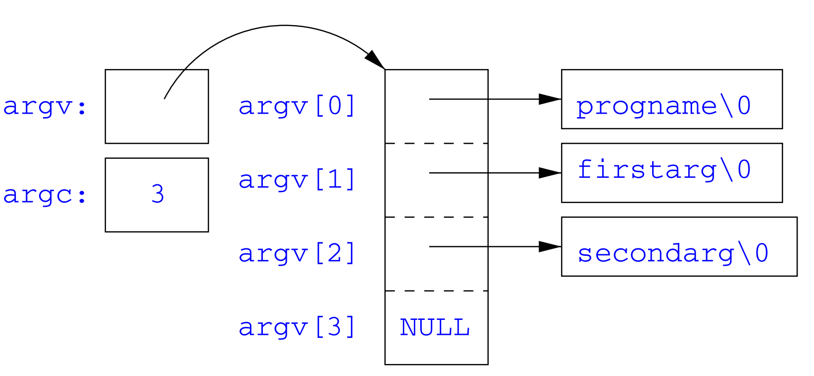
```
void swp1(int x,int y)
{
  int temp = x;
  x = y;
  y = temp;
}
```

```
void swp2(int *px,int *py)
{
  int temp = *px;
  *px = *py;
  *py = temp;
}
```

Arrays of pointers

- C allows the creation of arrays of pointers; for example
  `int *a[5];`
- Arrays of pointers are particularly useful with strings
- An example is C support of command line arguments:
  `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

Diagram of Argument List Layout



Multi-dimensional arrays

- Multi-dimensional arrays can be declared in C; for example: `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:

```
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

Function Pointer Example

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

Using a Higher-Order Function in C

```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:

  ```
  sort(void *p, const unsigned int len,
  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 `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
  `struct circle c;`
- A structure can also be initialised with values:
  ```
  struct circle c = {12, 23, 5};
  struct circle d = {.x = 12, .y = 23, .r = 5}; // C99
  ```
- An automatic, or local, structure variable can be initialised by function call: `struct circle c = circle_init();`
- A structure can be declared and several instances defined in one go:
  ```
  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: `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:

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

```
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; e.g.:
  `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)

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