

# Programming in C and C++

## Lecture 3: Pointers and Structures

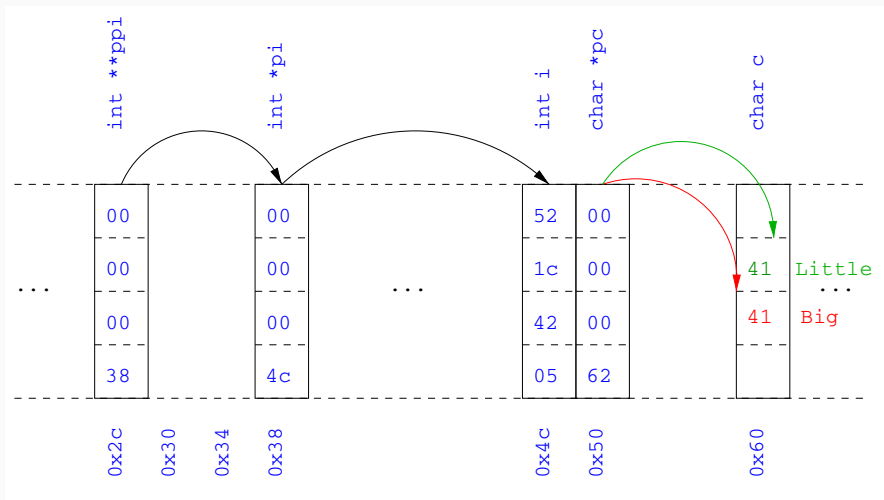
---

David J Greaves and Alan Mycroft  
(Materials by Neel Krishnaswami)

# 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



## 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

```
1  #include <stdio.h>
2
3  int main(void) {
4      int x=1,y=2;
5      int *pi;
6      int **ppi;
7
8      pi = &x; ppi = &pi;
9      printf("%p, %p, %d=%d=%d\n",ppi,pi,x,*pi,**ppi);
10     pi = &y;
11     printf("%p, %p, %d=%d=%d\n",ppi,pi,y,*pi,**ppi);
12
13     return 0;
14 }
```

## 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:

```
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:

```
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

```
1  #include <stdio.h>
2
3  int main(void) {
4      char str[] = "A string.";
5      char *pc = str;
6
7      printf("%c %c %c\n",str[0],*pc,pc[3]);
8      pc += 2;
9      printf("%c %c %c\n",*pc, pc[2], pc[5]);
10
11     return 0;
12 }
```



## 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)`:

```
1 void swp1(int x,int y)
2 {
3     int temp = x;
4     x = y;
5     y = temp;
6 }
```

```
1 void swp2(int *px,int *py)
2 {
3     int temp = *px;
4     *px = *py;
5     *py = temp;
6 }
```

## Arrays of pointers

- C allows the creation of arrays of pointers; for example

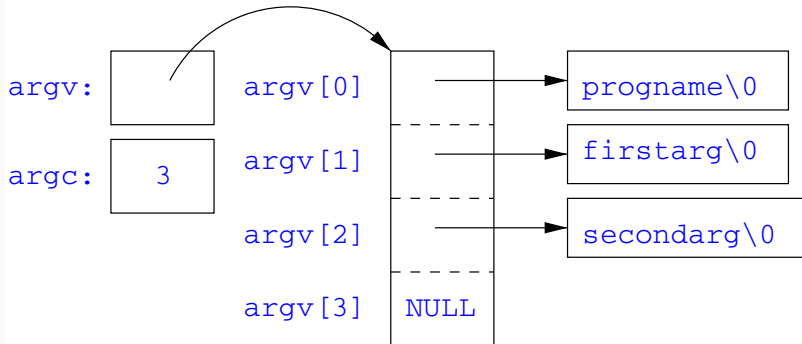
```
int *a[5];
```

- 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

```
1 void sort(int a[], const int len,  
2           int (*compare)(int, int)) {  
3     for(int i = 0; i < len-1; i++)  
4       for(int j = 0; j < len-1-i; j++)  
5         if ((*compare)(a[j],a[j+1])) {  
6           int tmp = a[j];  
7           a[j] = a[j+1], a[j+1] = tmp;  
8         }  
9     }  
10  
11 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

```
1  #include <stdio.h>
2  #include "example8.h"
3
4  int main(void) {
5      int a[] = {1,4,3,2,5};
6      unsigned int len = 5;
7      sort(a,len,inc); //or sort(a,len,&inc);
8
9      int *pa = a; //C99
10     printf("[");
11     while (len-->0) { printf("%d%s", *pa++, len?" ":""); }
12     printf("]\n");
13
14     return 0;
15 }
```



## 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 {                                1  struct link {
    int val;                                  2      int val;
    struct tree *left;                        3      struct link *next;
    struct tree *right;                       4  }
}
```

# 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)

```
1  struct { /* a compiler symbol table */
2      char *name;
3      struct {
4          unsigned int is_keyword : 1;
5          unsigned int is_extern : 1;
6          unsigned int is_static : 1;
7      } flags;
8      int utype;
9      union {
10         int ival; /* accessed as symtab[i].u.ival */
11         float fval;
12         char *sval;
13     } u;
14 } symtab[NSYM];
```