The C Programming Language

CS 1025 Computer Science Fundamentals I

Stephen M. Watt

University of Western Ontario

The C Programming Language

- A high-level language for writing low-level programs
- Allows machine-independent systems programming

- Operators correspond to popular machine instructions
- Easy to provide efficient compilers

A Little Bit of History

- 1969-1972: Developed at AT&T Bell Labs by Dennis Ritchie
  - Predecessor B, used word-based addressing.
  - C used with development of Unix OS
- 1978: “The C Programming Language” book
  by Kernighan and Ritchie
- 1989: ANSI/ISO Standard C89
  - Function prototypes, void, enumerations, ...
- 1999: ANSI/ISO Standard C99
  - Inline functions, complex numbers, variable length arrays, ...

Comparison to Java

- No objects
- Structs, unions and enumerations instead.

- The three most important things in C: (1) pointers, (2) pointers, (3) pointers.
- Distinction between structure and pointer to structure.
- Pointer arithmetic. Array/pointer equivalence.
- Strings are arrays of small characters, null terminated.

- C pre-processor
- Setjmp/longjmp functions instead of try/throw.
- register, typedef, volatile, const
- Sizes of integers. Signed and unsigned arithmetic.

- Different IO
- Explicit memory management. Sizeof. Bit-fields.

Arithmetic

- There are several integer types providing different sizes of numbers and both signed and unsigned arithmetic.
- These are:

```
char        signed char    unsigned char       just a kind of integer
short int                  unsigned short int
int                        unsigned int
long int                   unsigned long int
long long int              unsigned long long int
```

  If the word “int” is left out, e.g. unsigned short, then it is assumed.

- The sizes of these are not specified. It is merely required that

```
sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long
```

- This allows integer sizes to match those provided by the particular machine.
- This illustrates C’s philosophy of using flexibility to gain efficiency.

No Objects: Struct-s Instead

- “struct”s allow values to be grouped together,
  but provide no constructors or methods.

```
struct farm_animal {
    int       number_of_feet;
    int       number_of_wings;
    double    weight;
};

struct farm_animal  rover, porky, daffy;

rover.number_of_feet  = 4;       daffy.number_of_feet  =
rover.number_of_wings = 0;       daffy.number_of_wings =
rover.weight          = 5.8;     daffy.weight          =
```

Unions

- “union”s allow the same area of memory to be used for different things at different times.

```
union some_number {
    int  n;
    double d;
};

union some_number myNum;

myNum.n = 2;
myNum.d = 3.7;    /* overwrites myNum.n */
```

- These are easy to mis-use.

Enums

- “enum”s allow symbolic names to be given to a set of integers.

```
enum colour {
    RED,
    GREEN,
    BLUE = 9
};
```

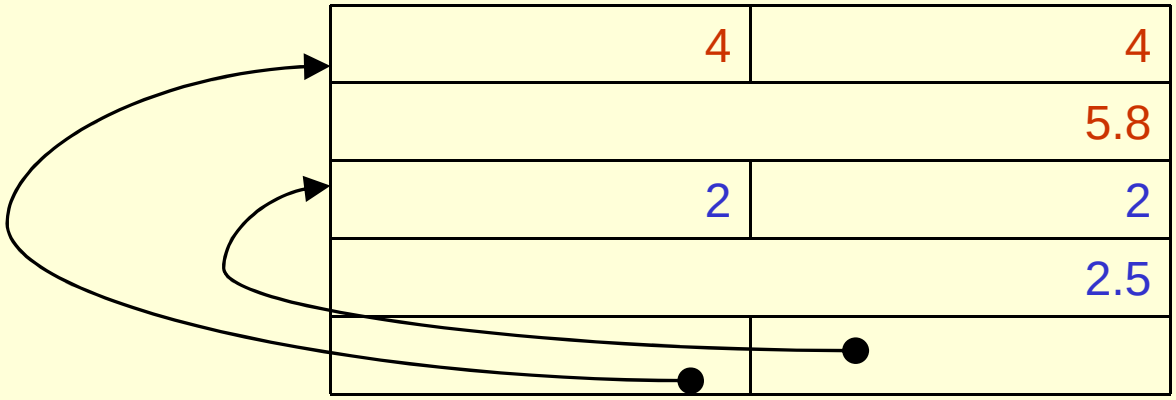
Then RED has value 0, GREEN value 1, BLUE value 9

The Thing *vs* A Pointer to the Thing

- The “&” operator gives the address (location in memory) of something.
- The “*” operator returns the contents of what is at a location.

- These are typed. A pointer to a double *vs* a pointer to a byte.
- A pointer to a value of type T is declared with “T *”

```
struct farm_animal *p_porky = &porky;
struct farm_animal *p_daffy = &daffy;
```

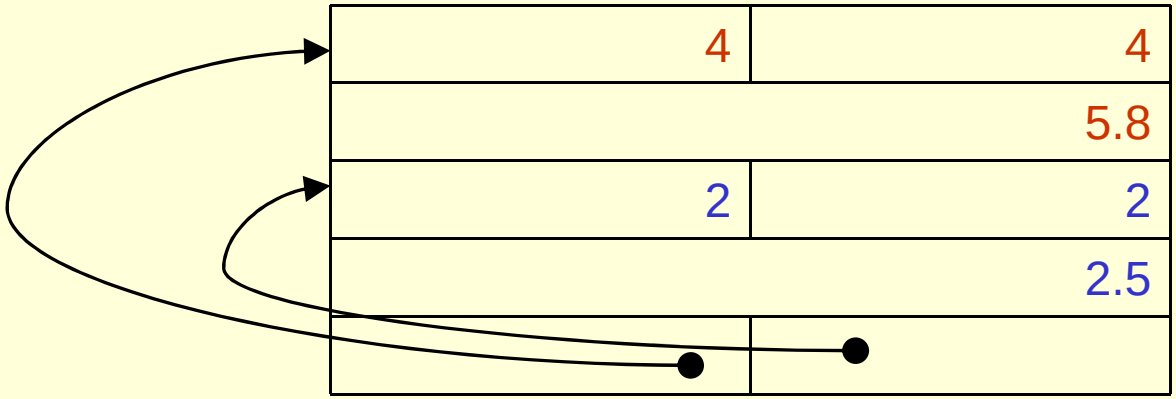


The Thing *vs* A Pointer to the Thing

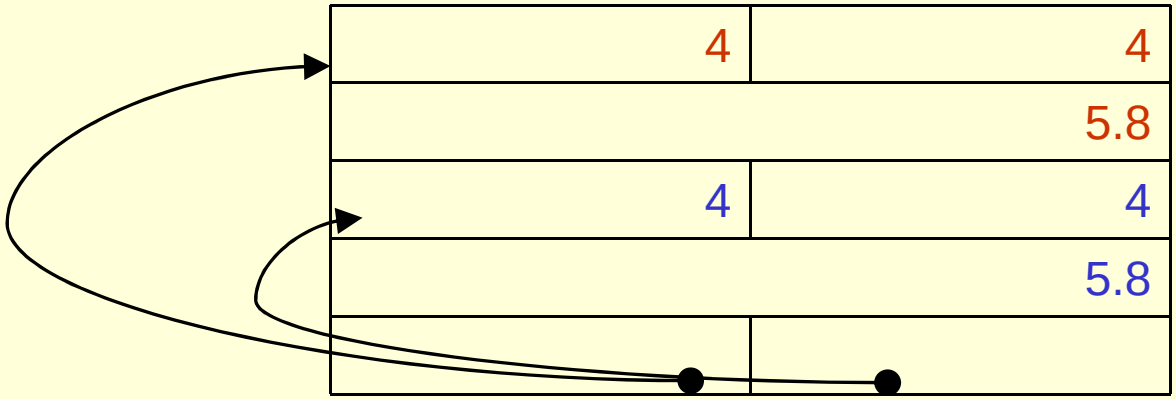
- Assignment of *structures* copies the data from the source *structure* to the destination *structure*.

```
daffy = porky;
```

Before:



After:

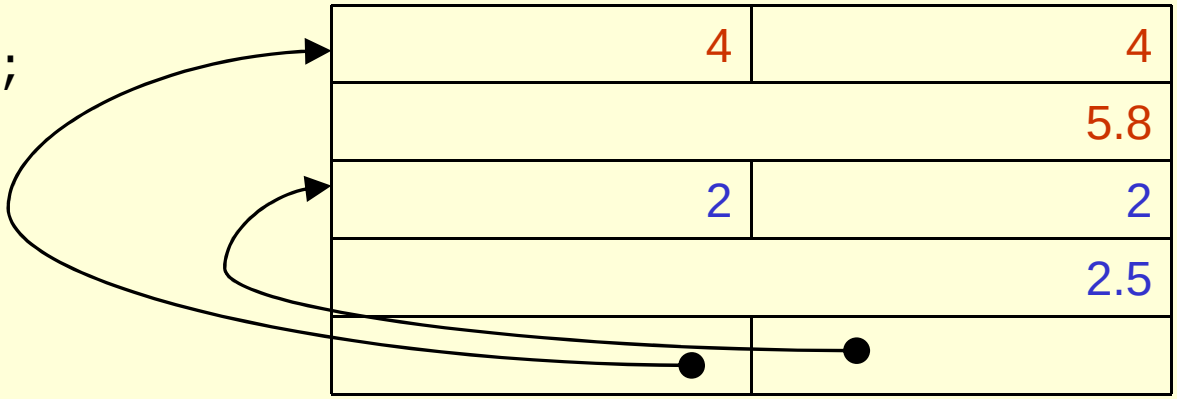


The Thing *vs* A Pointer to the Thing

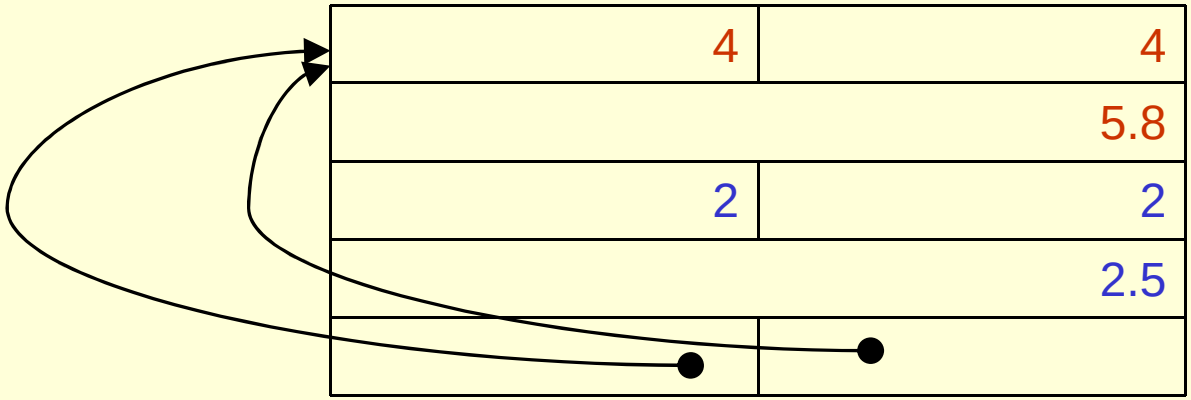
- Assignment of *pointers* copies the data from the source *pointer* to the destination *pointer.*

```
p_daffy = p_porky;
```

Before:



After:



The Thing *vs* A Pointer to the Thing

- In general, assignment copies data from the source to the destination, regardless of whether it is a pointer, structure, or other type.

- These may be used in combination.

```
p_porky = &daffy;    /* p_porky gets the address of daffy
porky   = *p_daffy;  /* porky gets what p_daffy points to
```

Arrays

- An array of a certain size can be declared using [ ].

```
int a[10];
```

- It is not necessary to call *new* to create this.

Pointer Arithmetic

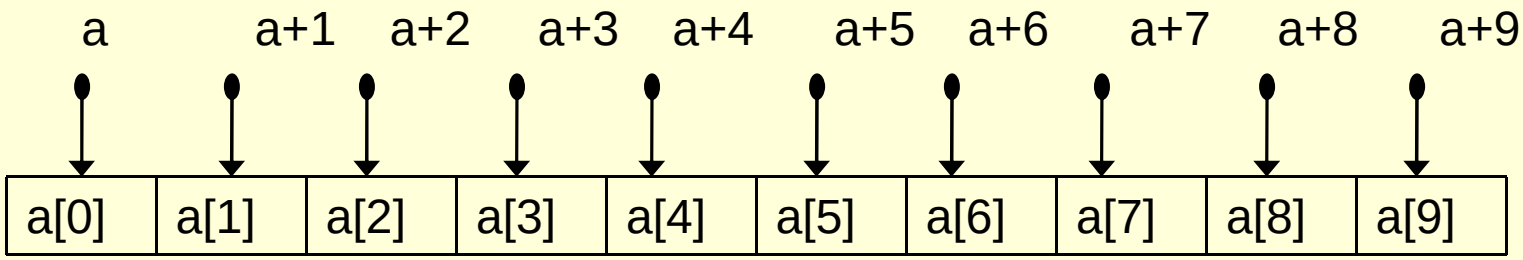
- The addresses of array elements are &(a[0]), &(a[1]), etc.

- You can add an integer to a pointer to get the location of another element.

```
&(a[3]) == &(a[0])  +  3
```

Pointer Arithmetic

- You can use the expression `a` in place of `&(a[0])`.
- Then `&(a[3]) == a + 3` .



Pointer Arithmetic

- Indeed, `a[i]` is really just a shorthand for `*(a+i)`
- You could write `*(a+2) = *(a+5)` instead of `a[2] = a[5]`

Pointer Arithmetic

- Since you can add integers to pointers, then `p++` and `p--` make sense.

  `p++` means “use the value of p, then afterward do p = p + 1”

Strings

- Strings in C are null-terminated arrays of characters.

  “hello” is the array [‘h’, ‘e’, ‘l’, ‘l’, ‘o’, 0]

- It is common to see code like

```
char *s = “hello”;
```

The C Preprocessor

- Because integers and bits in integers are used for so many things, and

  because compiler technology was not as advanced when C was defined,

  the language provides a textual-substitution “preprocessor” to allow symbolic names to be used where integers are needed.

```
#define READ_BIT  (1<<7)
#define MAX(a,b)  ((a) > (b) ? (a) : (b))

a |= READ_BIT;
c = MAX(sizeof(a), 12);
```

The C Preprocessor

- The C Preprocessor allows text from other files to be included in a program and conditional inclusion of text.

```
#include <stdio.h>

#if defined(MAXPATH)
static char pathname[MAXPATH];
#else
static char pathname[1000];
#endif
```

Type Definitions

- New type names may be introduced with “typedef”.

```
typedef  int            hash_code;
typedef  struct node    *pnode;


hash_code   h1, h2;
pnode       n1, n2, nodes[10];
```

- These are abbreviations and are equivalent to the old types.

Header Files

- Typically for each .c file or library one will have a .h header file containing
  - Function declarations
  - #define-s of related constants
  - typedef-s of related types

The Standard IO Library

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char **argv)
{
      FILE *fin, *fout;
      char c, *infile = “the-input”, *outfile = “the-output”;

      fin  = fopen(infile,  “r”);
      fout = fopen(outfile, “w”);
      if (fin == NULL || fout == NULL) {
            fprintf(stderr, “Cannot open %s or %s\n”, infile, outfile);
            exit(EXIT_FAILURE);
      }

      while ((c = getc (fin)) != EOF)
            putc(c, fout);

      fclose(fin);
      fclose(fout);
      return EXIT_SUCCESS;
}
```

The Standard IO Library – Part II

- Several functions. Most important:

```
printf(fmt, ...);
fprintf(FILE *, fmt, ...);
sprintf(char *, fmt, ...);

fgets(char *buf, int buf_size, FILE *infile);
```

- Output format codes:
  - %s : corresponding argument is a string
  - %c: corresponding argument is an integer to be displayed as a character
  - %d, %o, %x: corresponding argument is an integer to be displayed in decimal/octal/hex

Memory Management

- For dynamically managed structures use “malloc” and “free”

```
#include <stdlib.h>

void *malloc(size_t nbytes);
void free(void *);
```

Memory Management

```
struct cell { int value; struct cell  *next; };

struct cell *makeTheList(int start, int end) {
     struct cell *p = 0;

     for ( ; end >= start; end--) {
            struct cell *hd = (struct cell *) malloc(sizeof(struct cell
            if (hd == NULL) { .... }
            hd->value = end;   hd->next = p;
            p = hd;
     }
     return p;
}
void freeTheList(struct cell *p) {
     while (p) {
            struct cell *next = p->next;
            free(p);
            p = next;
     }
}
```

Conclusions

- C lets you write efficient, machine-dependent, low-level programs that use fixed size buffers.
- C also lets you write efficient, portable, safe, high-level programs that manage structures of arbitrary size.
- It is relatively easy to write efficient C code for small or medium-sized programs.
- It is relatively hard to write giant programs or maintain over time.
- If you think you understand C, check out http://www.ioccc.org/