Distributed and Parallel Technology

C Revision (Part I)

Hans-Wolfgang Loidl
`http://www.macs.hw.ac.uk/~hwloidl`

School of Mathematical and Computer Sciences
Heriot-Watt University, Edinburgh

---

[0] No proprietary software has been used in producing these slides

---

Introduction

- C is a *strict, strongly typed, imperative system programming language*
- combines high-level constructs with low level access to type representations and memory
- origins in BCPL & Fortran
- system programming language for Unix
- much wider use: Unix descendants e.g. Linux; Apple e.g. OS X
- evolution
    - C++: Object-oriented extension
    - C#: Advance programming language concepts; built on top of Microsoft .Net
- Reference: B. Kernighan & D. Ritchie, The C Programming Language (2nd Ed), Prentice-Hall, 1988

Source Code in Red Hat 7.1

Source code in Red Hat Linux 7.1[1]:

Language	*Source lines of Code SLOC (in %)*
C	*21461450 (71.18%)*
C++	*4575907 (15.18%)*
Shell (Bourne-like)	*793238 (2.63%)*
Lisp	*722430 (2.40%)*
Assembly	*565536 (1.88%)*
Perl	*562900 (1.87%)*
Fortran	*493297 (1.64%)*
Python	*285050 (0.95%)*
Tcl	*213014 (0.71%)*
Java	*147285 (0.49%)*

---

[1] From an article on slashdot (`http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html`)

Basic C Usage

- Put program text in `name1.c`

```
% gcc -o name2 name1.c
```

- this compiles `name1.c` using the GNU C compiler and puts the executable in `name2`

```
% ./name2
```

- run compiled program in `name2`

```
% ./name2 arg1 . . . argN
```

- run `name2` with command line arguments `arg1` . . . `argN`

```
% gcc -p -o name2 name1.c
```

- display profile information after running `name2`

Compiling with Optimisation

```
% gcc . . . -O . . .
```

- this generates *optimised code*

```
% gcc -c name1.c . . . nameN.c
```

- this generates object files `name1.o` . . . `nameN.o` from `name1.c` . . . `nameN.c` but not executables

```
% gcc -o name name1.o . . . nameN.o
```

- link object files `name1.o` . . . `nameN.o` and put executable in name

```
% man gcc
```

- displays Unix manual entry for GNU C compiler
- *Aside:* can use cc instead of gcc, as proprietary C compiler for host OS

Compiling for Debugging

```
% gcc . . . -g . . .
```

- this generates *code with debugging information*

```
% gdb name2
```

- this starts the GNU debugger on this program

```
% run arg1 . . . argN
```

- this excutes the program within the debugger

```
% man gdb
```

- check the man pages for commands, such as setting breakpoints, in the debugger
- *Aside:* the 1 page gdb cheat sheet is a very useful summary

Program Layout

```
1. #include ...
2. #define ...
3. extern ...
4. declarations
5. function declarations
6. main(int argc,char ** argv)
{ ... }
```

1. (textually) import files:
    - `#include "..."` from current directory
    - `#include <...>` from system directory
    - eg. `#include <stdio.h>`
2. macro and constant definitions
3. names/types of variables/functions used in this file but declared in linked files
4. declare all variables, used later on
5. declare all functions, used later on
6. main function with optional command line argument count and array

Basic C Types

- The following *variable declaration* allocates space for the variable `identifier` of type `type` on the stack

  ```
  type identifier;
  ```

- the variable remains in existence within the current {. . .} block
- To read the memory *address* for the start of a variable (“lvalue”) use

  ```
  &identifier;
  ```

- To read the *contents* of a variable, whose address is denoted by expression (“rvalue”) use

  ```
  *expression;
  ```

Sizes of Data Structures

- To get the size of a data structure (in byte) use this function

```
int sizeof(type);
```

- *NB:* the result may depend on OS & CPU & compiler
- basic types and their sizes

int	4 bytes	char	1 byte
short	2 bytes	float	4 bytes
long	4 bytes	double	8 bytes

- *NB:* int is stored from most significant to least significant byte, e.g.

```
int a;
a = 0x01234567;
```

is stored as `67 45 23 01`

- *NB:* the size of a pointer depends on architecture (32-bit means 4 byte pointers)

Structured Types

- Array declaration:

```
type identifier [int];
```

- this allocates the array on the stack
- *NB:* `identifier` is an alias for the address, not a variable in the usual sense, eg.

```
int a[3];
printf("a: %x; &a: %x",a,&a);

==> a: 80497fc; &a: 80497fc
```

- *Aside:* %x means, print as hexadecimal
- to access an array element use

```
identifier[exp]
```

- same as: `*(identifier + exp * size for type)`
- i.e. read the contents of offset for exp elements of type from address of 1st byte

---

Multi-dimensional Arrays

- To allocate a multi-dimensional array, use

```
type identifier[int1][int2];
```

- this allocates space for `int1` arrays of `int2 * size for type`
- to read the value of an array element use

```
identifier[exp1][exp2]
```

- this is the same as: `*(identifier + (int1*exp1 + exp2) * size for type)`
- i.e. skip `exp1` rows of length `int1` and then skip `exp2` columns

Structures

- To allocate a structure, use

```
struct {type1 id1; ... typeN idN;} identifier;
```

- allocate size of `type1` + . . . + size for `typeN` on stack
- `identifier` is name of variable made up of all these bytes
- `&identifier` is address of 1st byte in sequence
- stucture fields are allocated in the given order
- To define only the structure type, use

```
struct identifier1 { type1 id1; ... typeN idN }
```

- the name of the type is `struct identifier1`
- does not allocate space!

Structures

- Both forms can be combined, eg
  `struct identifier1 identifier2;`
- this associates `identifier2` with 1st byte of new sequence of type `struct identifier1`
- to access a field in a structure, use `identifier.idi;`
- same as `*(&identifier + size for type1 ... + size for typeiI-1)`
- i.e. read the contents of offset of preceding fields from start of structure
- *NB:* for `struct { ... } identifier;`, we have `identifier != &identifier`, eg

```
printf("m: %x; &m: %x", m, &m);

==> m: 64636261; &m: 8049808
```

- 61 == ASCII ‘a’ in hex; 62 == ASCII ‘b’ in hex ...
- *NB:* struct fields held left to right but printing struct as hex coerces to int and accesses bytes right to left as most to least significant

---

Recursive Structures

- NB: we cannot directly define recursive structs

```
struct node  { int val; node next; }
struct node list;
```

- list is allocated space for a struct node
  - space for an `int`
  - space for a `struct node`
    - space for an `int`
    - space for a `struct node`
- solution: use indirect recursion via pointers

Pointers

- To declare a pointer, use

```
type *identifier;
```

- `identifier` holds address for byte sequence for `type`
- allocates space for address but *does not create instance*

```
struct node { int val; node *next; }
```

- `node` needs space for `int` and space for pointer to `node`
- To declare a variable `list` as a pointer to `node`, use

```
struct node *list;
```

- must use `malloc` to allocate space for node
- for structure field access via pointers, use

```
identifier->idI
```

- same as: `*(identifier + size for type1 ... + size for typeI-1)`
- i.e. read contents of offset of preceding elements from byte sequence that identifier points at
  (empty pointer: `NULL`)

Dynamic Space Allocation

- To dynamically allocate memory in the heap, use

```
malloc(int)
```

- allocates int bytes on heap
- returns address of 1st byte
- like `new` in C#/Java
- returns `char *`; use coercion to convert type
- to de-allocate memory, use an explicit `free`

```
free(void *)
```

- returns space to heap
- space must have been allocated by `malloc`
- *NB:* does not recursively return space
- Example:

```
list = (node *)malloc(sizeof(node));
```

- this allocates space for `int` and space for pointer to `node`

```
list->val = 0;
list->next = NULL;
```

Example: Generating a list

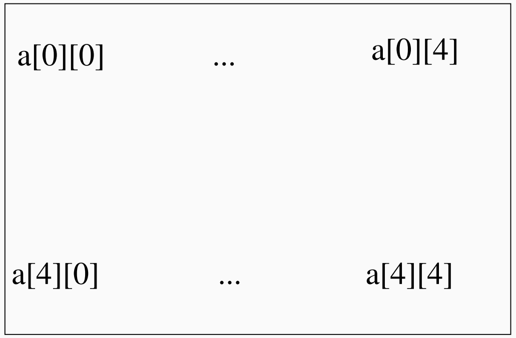
```
/* types */
typedef struct _node { int value; struct _node *next; } node;
/* generate a list from an array */
node *mkList(int len, int *arr) {
  int i;
  node *curr, *last, *root;
  if (len>0) {
    last = (node*) malloc(1*sizeof(node));
    last->value = arr[0];
    root = last;
  } else {
    return NULL;
  }
  for (i=0; i<len-1; i++) {
    curr = (node*) malloc(1*sizeof(node));
    curr->value = arr[i+1];
    last->next = curr;
    last = curr;
  }
  last->next = NULL;
  return root;
```

Pointers vs Arrays

- For the difference between arrays and pointers, consider
  `type identifier[int1][int2]`
- this allocates `int1` * `int2` array of `type`
- 2nd dimension all length `int2`
- actually allocated as `int1` * `int2` continuous locations for type
- *BUT:*
  `type * identifier[int]`
- allocates *array of int pointers* to `type`
- must use `malloc` to allocate 2nd dimension
- arrays in 2nd dimension can be any sizes
- *AND:*
  `type ** identifier`
- allocates pointer to pointer to `type`
- must use `malloc` to allocate 1st and 2nd dimension
- 1st and 2nd dimension can be any size
- in all cases, use `identifier[exp1][exp2]` to access an element

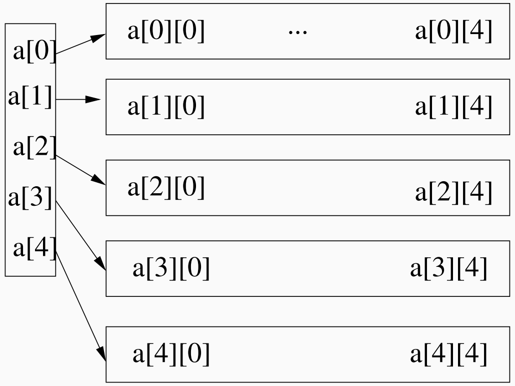
Array data layout

`int a[5][5]`



Array data layout

`int *a[5]`



Pointer Arithmetic

- For pointer arithmetic, consider:
  `type *identifier`
- arithmetic on identifier is in units of the size for type

```
identifier+exp ==
identifier = identifier + exp * size for type
```

- i.e. pointer has moved on exp elements in sequence

```
identifier-exp ==
identifier = identifier - exp * size for type
```

- i.e. pointer has moved back exp elements in sequence

```
identifier++ ==
identifier = identifier + size for type
```

- i.e. pointer has moved on one element

```
identifier-- ==
identifier = identifier - size for type
```

- i.e. pointer has moved back one element

Example: memcpy

A typical example of such pointer arithmetic is this function to copy a block of memory from the location pointed to by `p1` to the location pointed to by `p2`.

```
void memcpy (int *p1, int *p2, int n) {
    int *p = p1;
    int *q = p2;
    for (int i = 0; i<n; i++) {
        *q++ = *p++;
    }
}
```

Type Coercions

- To coerce an expression to a type, use

  `(type)expression;`
- this evaluates expression to value
- then treats value as if of type `type`
- *does not physically transform* expression
- as if overlaid template for type on value

Example for Type Coercions

```
int x;  char * c;
x = 0x01234567
c = (char *)(&x);
printf("%x %x %x %x",
       c[0],c[1],c[2],c[3]);

==> 67 45 23 01
```

- `x` is 4 hex bytes `01 23 45 67`
- stored from most significant to least significant
- `&x` returns address of 1st byte of `int`
- `(char *)` coerces address of 1st byte of int to address of 1st byte of array of char
- `c[0]` is 1st byte of `x`, `c[1]` is 2nd byte of x etc
- coercions very important for inter-process communication
- if space for data allocated continuously then can:
  - coerce arbitrary type to *sequence of char* for transmission coerce back to type on reception
  - coerce back to type on reception

Exercises

- Write a function `node *append(node *x, node *y)` that appends the elements of the the second list `y` to the end of the first list `x` (i.e. `append([1,2],[3,4]) ==> [1,2,3,4]`).
- How does this affect the list `x`?
- Write a second version that does not modify the input lists.
- Under which condition is it safe to use the first version?
- Write a function `node *reverse(node *x)` that reverses the elements in the list (i.e. `reverse([1,2,3]) ==> [3,2,1]`)