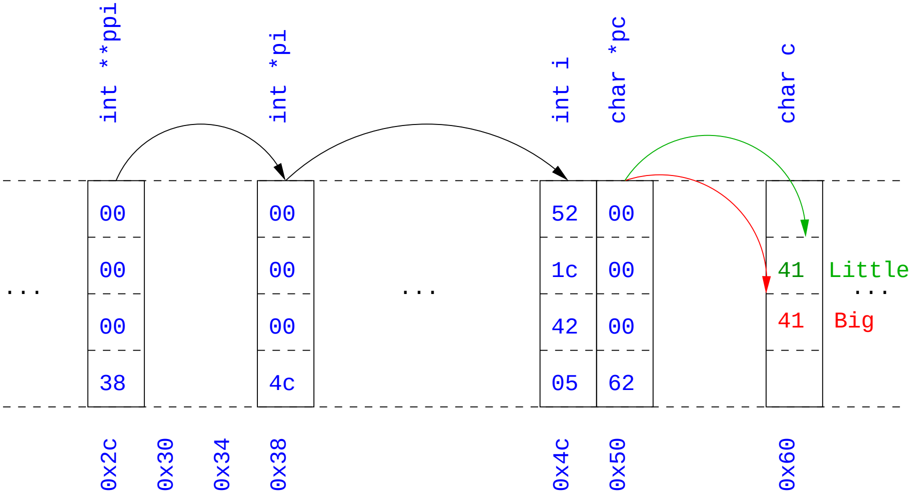
Programming in C and C++

4. Misc. Library Features, Gotchas, Hints and Tips

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

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

Uses of const and volatile

- Any declaration can be prefixed with `const` or `volatile`
- A `const` variable can only be assigned a value when it is defined
- The `const` declaration can also be used for parameters in a function definition
- The `volatile` keyword can be used to state that a variable may be changed by hardware or the kernel.
  - For example, the `volatile` keyword may prevent unsafe compiler optimisations for memory-mapped input/output
- The use of pointers and the `const` keyword is quite subtle:
  - `const int *p` is a pointer to a `const` int
  - `int const *p` is also a pointer to a `const` int
  - `int *const p` is a `const` pointer to an int
  - `const int *const p` is a `const` pointer to a `const` int

Example

```c
int main(void) {
  int i = 42;
  int j = 28;

  const int *pc = &i;        //Also: "int const *pc"
  *pc = 41;                  //Wrong
  pc = &j;

  int *const cp = &i;
  *cp = 41;
  cp = &j;                   //Wrong

  const int *const cpc = &i;
  *cpc = 41;                 //Wrong
  cpc = &j;                  //Wrong
  return 0;
}
```

Typedefs

- The `typedef` operator, creates a synonym for a data type; for example, `typedef unsigned int Radius;`
- Once a new data type has been created, it can be used in place of the usual type name in declarations and casts; for example, `Radius r = 5; ...; r = (Radius) rshort;`
- A `typedef` declaration does not create a new type
  - It just creates a synonym for an existing type
- A `typedef` is particularly useful with structures and unions:

```
typedef struct llist *llptr;
typedef struct llist {
  int val;
  llptr next;
} linklist;
```

Inline functions

- A function in C can be declared `inline`; for example:

```
inline int fact(unsigned int n) {
        return n ? n*fact(n-1) : 1;
}
```

- The compiler will then try to "inline" the function
  - A clever compiler might generate `120` for `fact(5)`
- A compiler might not always be able to "inline" a function
- An `inline` function must be <u>defined</u> in the same execution unit as it is used
- The `inline` operator does not change function semantics
  - the inline function itself still has a unique address
  - static variables of an inline function still have a unique address
- Both `inline` and `register` are largely unnecessary with modern compilers and hardware

That's it!

- We have now explored most of the C language
- The language is quite subtle in places; in particular watch out for:
  - operator precedence
  - pointer assignment (particularly function pointers)
  - implicit casts between `int`s of different sizes and `char`s
- There is also extensive standard library support, including:
  - shell and file I/O (`stdio.h`)
  - dynamic memory allocation (`stdlib.h`)
  - string manipulation (`string.h`)
  - character class tests (`ctype.h`)
  - ...
  - (Read, for example, K&R Appendix B for a quick introduction)
  - (Or type "`man` *function*" at a Unix shell for details)

Library support: I/O

I/O is not managed directly by the compiler; support in `stdio.h`:

- `FILE *stdin, *stdout, *stderr;`
- `int printf(const char *format, ...);`
- `int sprintf(char *str, const char *format, ...);`
- `int fprintf(FILE *stream, const char *format, ...);`
- `int scanf(const char *format, ...); // sscanf,fscanf`

- `FILE *fopen(const char *path, const char *mode);`
- `int fclose(FILE *fp);`
- ```
  size_t fread(void *ptr, size_t size, size_t nmemb,
               FILE *stream);
  ```
- ```
  size_t fwrite(const void *ptr, size_t size, size_t nmemb,
                FILE *stream);
  ```

```c
#include<stdio.h>
#define BUFSIZE 1024

int main(void) {
  FILE *fp;
  char buffer[BUFSIZE];

  if ((fp=fopen("somefile.txt","rb")) == 0) {
    perror("fopen error:");
    return 1;
  }

  while(!feof(fp)) {
      int r = fread(buffer,sizeof(char),BUFSIZE,fp);
      fwrite(buffer,sizeof(char),r,stdout);
  }

  fclose(fp);
  return 0;
}
```

Library support: dynamic memory allocation

- Dynamic memory allocation is not managed directly by the C compiler
- Support is available in `stdlib.h`:
  - `void *malloc(size_t size)`
  - `void *calloc(size_t nobj, size_t size)`
  - `void *realloc(void *p, size_t size)`
  - `void free(void *p)`
- The C `sizeof` unary operator is handy when using `malloc`:

  ```
  p = (char *) malloc(sizeof(char)*1000)
  ```
- Any successfully allocated memory must be deallocated manually
  - Note: `free()` needs the pointer to the allocated memory
- Failure to deallocate will result in a memory leak

Gotchas: operator precedence

```
#include<stdio.h>

struct test {int i;};
typedef struct test test_t;

int main(void) {

  test_t a,b;
  test_t *p[] = {&a,&b};
  p[0]->i=0;
  p[1]->i=0;
  test_t *q = p[0];

  printf("%d\n",++q->i); //What does this do?

  return 0;
}
```

Gotchas: `i++`

```c
#include <stdio.h>

int main(void) {

  int i=2;
  int j=i++ + ++i;
  printf("%d %d\n",i,j); //What does this print?

  return 0;
}
```

Expressions like `i++ + ++i` are known as grey (or gray) expressions in that their meaning is compiler dependent in C (even if they are defined in Java)

Gotchas: local stack

```c
#include <stdio.h>

char *unary(unsigned short s) {
  char local[s+1];
  int i;
  for (i=0;i<s;i++) local[i]='1';
  local[s]='\0';
  return local;
}

int main(void) {
  printf("%s\n",unary(6)); //What does this print?
  return 0;
}
```

Gotchas: local stack (contd.)

```c
#include <stdio.h>

char global[10];

char *unary(unsigned short s) {
  char local[s+1];
  char *p = s%2 ? global : local;
  int i;
  for (i=0;i<s;i++) p[i]='1';
  p[s]='\0';
  return p;
}

int main(void) {
  printf("%s\n",unary(6)); //What does this print?
  return 0;
}
```

Gotchas: careful with pointers

```
#include <stdio.h>

struct values { int a; int b; };

int main(void) {
 struct values  test2 = {2,3};
 struct values  test1 = {0,1};

 int *pi = &(test1.a);
 pi += 1; //Is this sensible?
 printf("%d\n",*pi);
 pi += 2; //What could this point at?
 printf("%d\n",*pi);

 return 0;
}
```

Gotchas: XKCD pointers

Tricks: Duff's device

```
boring_send(int *to, int *from, int count) {
  do {
    *to = *from++;
  } while(--count > 0);
}

send(int *to, int *from, int count) {
  int n = (count+7)/8;
  switch (count%8) {
  case 0: do{ *to = *from++;
  case 7:     *to = *from++;
  case 6:     *to = *from++;
  case 5:     *to = *from++;
  case 4:     *to = *from++;
  case 3:     *to = *from++;
  case 2:     *to = *from++;
  case 1:     *to = *from++;
          } while(--n>0);
  }
}
```

Assessed Exercise

See "Head of Department's Announcement"

- To be completed by noon on Monday 23 January 2017
- Viva examinations 1330-1630 on Thursday 26 January 2016
- Viva examinations 1330-1630 on Friday 27 January 2016
- Download the starter pack from:
  `http://www.cl.cam.ac.uk/Teaching/current/CandC++/`
- This should contain eight files:

`server.c`	`client.c`	`rfc0791.txt`	`rfc0793.txt`
`message1`	`message2`	`message3`	`message4`

Exercise aims

Demonstrate an ability to:

- Understand (simple) networking code
- Use control flow, functions, structures and pointers
- Use libraries, including reading and writing files
- Understand a specification
- Compile and test code
- Comprehending man pages

Task is split into three parts:

- Comprehension and debugging
- Preliminary analysis
- Completed code and testing

Exercise submission

- Assessment is in the form of a 'tick'
- There will be a short viva; remember to sign up!
- Submission is via email to `c-tick@cl.cam.ac.uk`
- Your submission should include seven files, packed in to a ZIP file called crsid`.zip` and attached to your submission email:

`answers.txt`	`client1.c`	`summary.c`	`message1.txt`
	`server1.c`	`extract.c`	`message2.jpg`

Hints: IP header

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Identification        |Flags|      Fragment Offset    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Time to Live |    Protocol   |         Header Checksum       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Source Address                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Hints: IP header (in C)

```c
#include <stdint.h>

struct ip {
  uint8_t hlenver;
  uint8_t tos;
  uint16_t len;
  uint16_t id;
  uint16_t off;
  uint8_t ttl;
  uint8_t p;
  uint16_t sum;
  uint32_t src;
  uint32_t dst;
};

#define IP_HLEN(lenver) (lenver & 0x0f)
#define IP_VER(lenver) (lenver >> 4)
```

Hints: network byte order

- The IP network is big-endian; x86 is little-endian; ARM can be either
- Reading multi-byte values requires possible conversion
- The BSD API specifies:
  - `uint16_t ntohs(uint16_t netshort)`
  - `uint32_t ntohl(uint32_t netlong)`
  - `uint16_t htons(uint16_t hostshort)`
  - `uint32_t htonl(uint32_t hostlong)`

  which encapsulate the notions of host and network and their interconversion (which may be a no-op)

Exercises

1. What is the value of `i` after executing each of the following:
    1.1 `i = sizeof(char);`
    1.2 `i = sizeof(int);`
    1.3 `int a; i = sizeof a;`
    1.4 `char b[5]; i = sizeof(b);`
    1.5 `char *c=b; i = sizeof(c);`
    1.6 `struct {int d;char e;} s; i = sizeof s;`
    1.7 `void f(int j[5]) { i = sizeof j;}`
    1.8 `void f(int j[][10]) { i = sizeof j;}`
2. Use `struct` to define a data structure suitable for representing a binary tree of integers. Write a function `heapify()`, which takes a pointer to an integer array of values and a pointer to the head of an (empty) tree and builds a binary heap of the integer array values. (Hint: you should `malloc()` and a binary tree data structure)
3. What other C data structure can be used to represent a heap? Would using this structure lead to a more efficient implementation of `heapify()`?