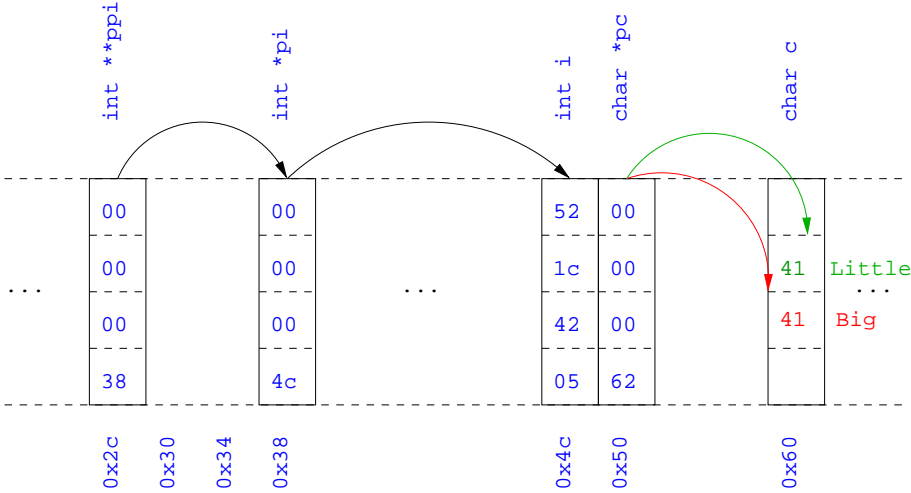# Programming in C and C++

4. Misc. — Library Features — Gotchas — Hints 'n' Tips

Dr. Anil Madhavapeddy

University of Cambridge
(based on previous years –
thanks to Alan Mycroft, Alastair Beresford and Andrew Moore)

Michaelmas Term 2014–2015

Example

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

```
1 int main(void) {
2   int i = 42;
3   int j = 28;
4
5   const int *pc = &i;        //Also: "int const *pc"
6   *pc = 41;                  //Wrong
7   pc = &j;
8
9   int *const cp = &i;
10  *cp = 41;
11  cp = &j;                   //Wrong
12
13  const int *const cpc = &i;
14  *cpc = 41;                 //Wrong
15  cpc = &j;                  //Wrong
16  return 0;
17 }
```

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 <u>not</u> create a new type
  ▶ It just creates a synonym for an existing type

▶ A `typedef` is particularly useful with structures and unions:

```
1 typedef struct llist *llptr;
2 typedef struct llist {
3   int val;
4   llptr next;
5 } linklist;
```

Inline functions

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

```c
1 inline int fact(unsigned int n) {
2     return n ? n*fact(n-1) : 1;
3 }
```

- ▶ 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 defined 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
1 #include<stdio.h>
2 #define BUFSIZE 1024
3
4 int main(void) {
5   FILE *fp;
6   char buffer[BUFSIZE];
7
8   if ((fp=fopen("somefile.txt","rb")) == 0) {
9     perror("fopen error:");
10    return 1;
11   }
12
13   while(!feof(fp)) {
14       int r = fread(buffer,sizeof(char),BUFSIZE,fp);
15       fwrite(buffer,sizeof(char),r,stdout);
16   }
17
18   fclose(fp);
19   return 0;
20 }
```

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

```
1 #include<stdio.h>
2
3 struct test {int i;};
4 typedef struct test test_t;
5
6 int main(void) {
7
8   test_t a,b;
9   test_t *p[] = {&a,&b};
10  p[0]->i=0;
11  p[1]->i=0;
12  test_t *q = p[0];
13
14  printf("%d\n",++q->i); //What does this do?
15
16  return 0;
17 }
```

Gotchas: `i++`

```
1 #include <stdio.h>
2
3 int main(void) {
4
5   int i=2;
6   int j=i++ + ++i;
7   printf("%d %d\n",i,j); //What does this print?
8
9   return 0;
10 }
```

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.)

```
1 #include <stdio.h>
2
3 char global[10];
4
5 char *unary(unsigned short s) {
6   char local[s+1];
7   char *p = s%2 ? global : local;
8   int i;
9   for (i=0;i<s;i++) p[i]='1';
10  p[s]='\0';
11  return p;
12 }
13
14 int main(void) {
15   printf("%s\n",unary(6)); //What does this print?
16   return 0;
17 }
```
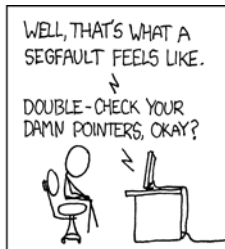
Gotchas: careful with pointers

```
1 #include <stdio.h>
2
3 struct values { int a; int b; };
4
5 int main(void) {
6  struct values  test2 = {2,3};
7  struct values  test1 = {0,1};
8
9  int *pi = &(test1.a);
10  pi += 1; //Is this sensible?
11  printf("%d\n",*pi);
12  pi += 2; //What could this point at?
13  printf("%d\n",*pi);
14
15  return 0;
16 }
```

Gotchas: XKCD pointers

Tricks: Duff's device

```
1 boring_send(int *to, int *from, int count)
2 {
3   do {
4     *to = *from++;
5   } while(--count > 0);
6 }
7
8 send(int *to, int *from, int count)
9 {
10   int n=(count+7)/8;
11   switch(count%8){
12   case 0: do{ *to = *from++;
13   case 7:     *to = *from++;
14   case 6:     *to = *from++;
15   case 5:     *to = *from++;
16   case 4:     *to = *from++;
17   case 3:     *to = *from++;
18   case 2:     *to = *from++;
19   case 1:     *to = *from++;
```

Assessed exercise

See "Head of Department's Announcement"

- ▶ To be completed by noon on Monday 9th February 2015
- ▶ Viva examinations 1300-1600 on 12th February 2015
- ▶ Viva examinations 1300-1600 on 13th February 2015
- ▶ Download the starter pack from:
  http://www.cl.cam.ac.uk/Teaching/current/CandC++/
- ▶ This should contain eight files:
  ```
  server.c   rfc0791.txt   message1   message3
  client.c   rfc0793.txt   message2   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     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
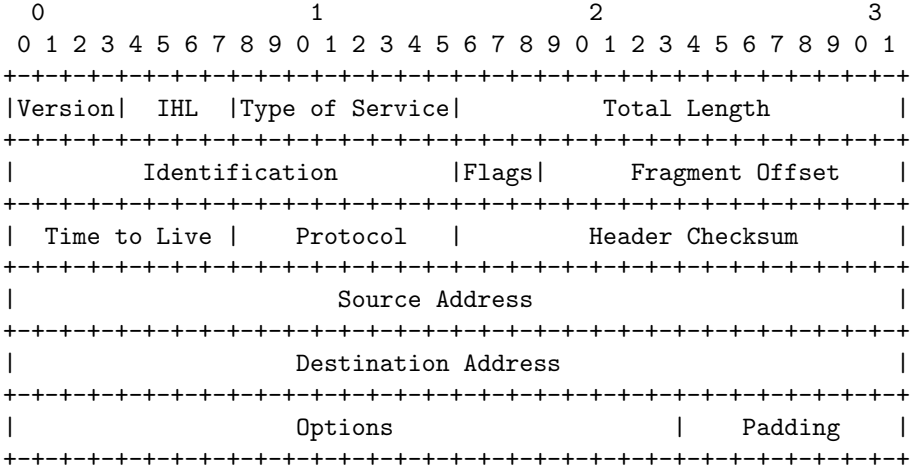
Hints: IP header (in C)

```c
1  #include <stdint.h>
2
3  struct ip {
4    uint8_t hlenver;
5    uint8_t tos;
6    uint16_t len;
7    uint16_t id;
8    uint16_t off;
9    uint8_t ttl;
10   uint8_t p;
11   uint16_t sum;
12   uint32_t src;
13   uint32_t dst;
14 };
15
16 #define IP_HLEN(lenver) (lenver & 0x0f)
17 #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'll need to use `malloc()`)

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()`?