Programming in C and C++

Lecture 4: Miscellaneous Features, Gotchas, Hints and Tips

Neel Krishnaswami and Alan Mycroft

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, j = 28;
3
4    const int *pc = &i;          // Also: "int const *pc"
5    *pc = 41;                    // Wrong
6    pc = &j;
7
8    int *const cp = &i;
9    *cp = 41;
10   cp = &j;                     // Wrong
11
12   const int *const cpc = &i;
13   *cpc = 41;                   // Wrong
14   cpc = &j;                    // Wrong
15   return 0;
16 }
```

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:

```
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:

```
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 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

Thats it!

- We have now explored most of the C language
- The language is quite subtle in places; especially beware of:
  - operator precedence
  - pointer assignment (particularly function pointers)
  - implicit casts between ints of different sizes and chars
- 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

```c
#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: Increment Expressions

```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;
}
```

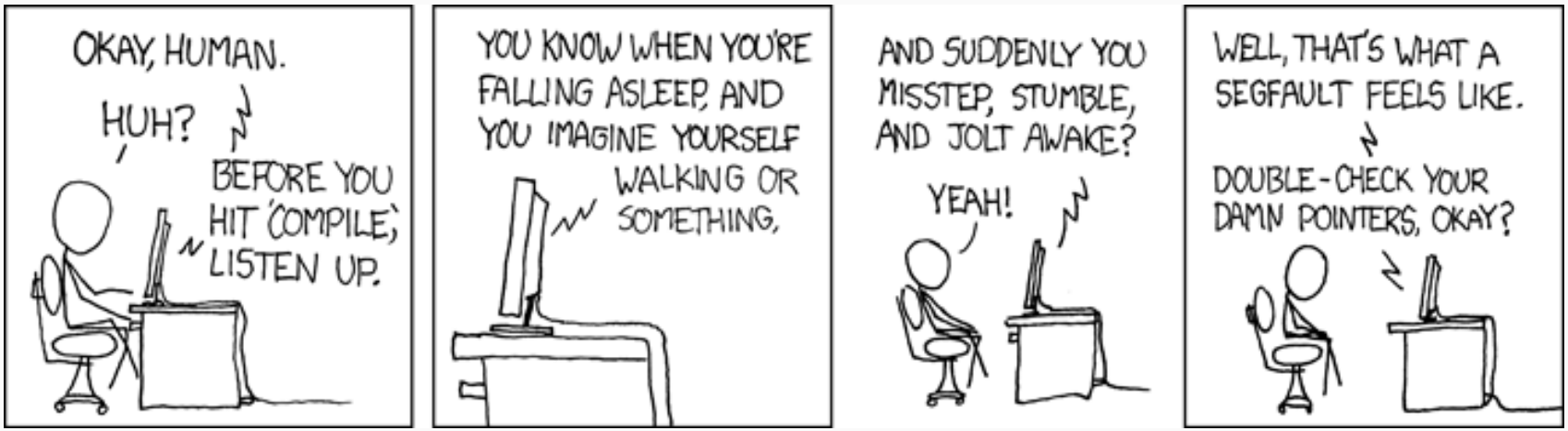
Gotchas: local stack (contd.)

```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

```c
#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: Duffs device

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

```
1  boring_send(int *to, int *from,
2              int count) {
3    do {
4      *to = *from++;
5    } while(--count > 0);
6  }
```

Assessed Exercise

See Head of Departments Announcement

- To be completed by noon on Monday 21 January 2019
- Viva examinations 1330-1630 on Thursday 24 January 2019
- Viva examinations 1330-1630 on Friday 25 January 2019
- Download the starter pack from:
  `http://www.cl.cam.ac.uk/Teaching/1819/ProgC/`
- 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:

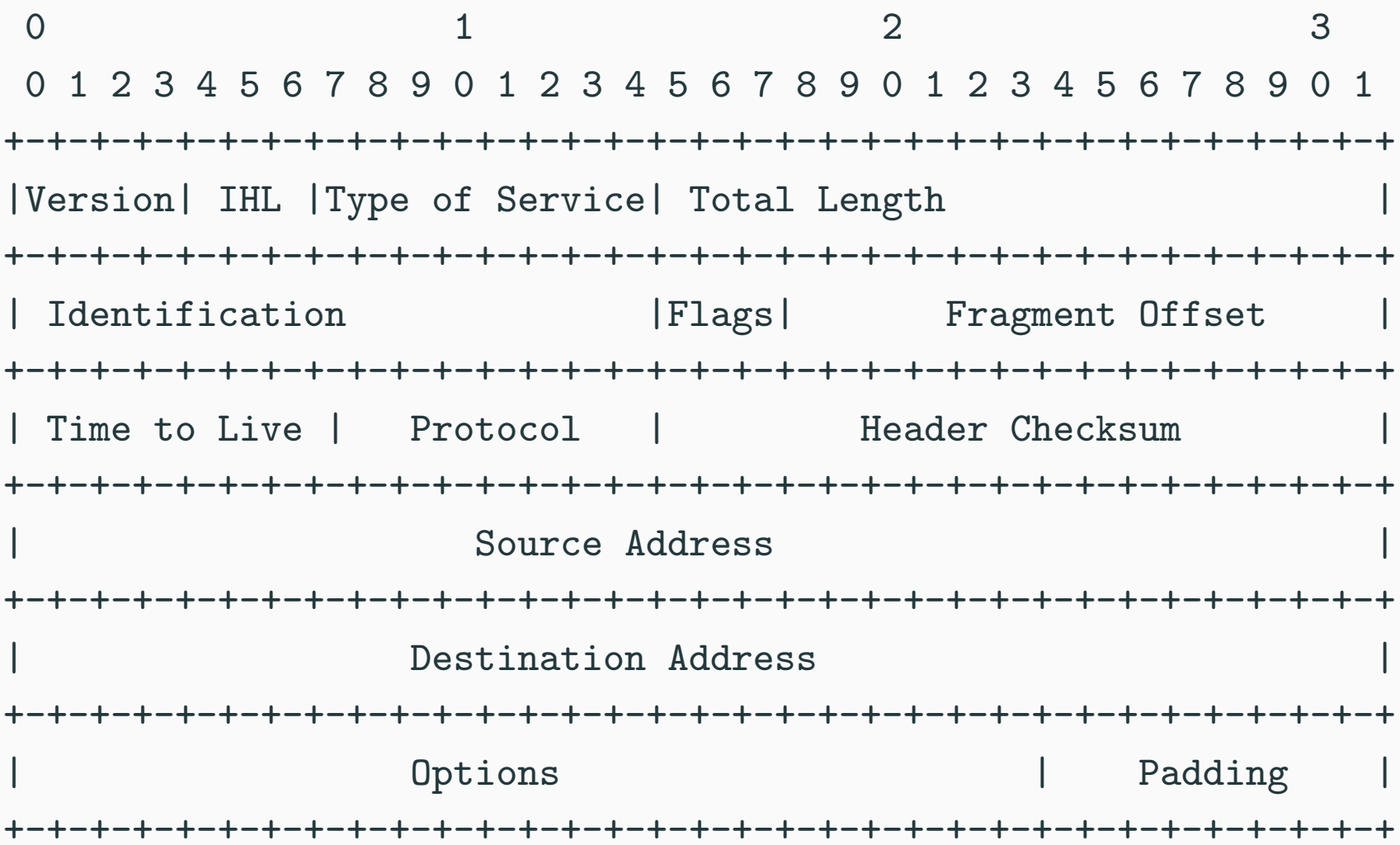
- Comprehension and debugging
- Preliminary analysis
- Completed code and testing

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



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)