CO658 bonus material: C for Java programmers (part two)

Stephen Kell

s.r.kell@kent.ac.uk

A more interesting bit of code

```
// tree.c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

struct bst {
    struct bst *left;
    int num;
    struct bst *right;
};
static struct bst *root;
static
void bst_insert(int n, struct bst *node) {
    struct bst *new_node = calloc(1,
       sizeof (struct bst));
    new_node->num = n;
    if (!node) { // empty tree
        assert(!root);
        root = new_node;
    } else if (n < node->num && !node->left) {
      node->left = new_node;
    } else if (n >=node->num && !node->right){
        node->right = new_node;
    } else if (n < node->num) {
        bst_insert(n, node->left);
    } else {
        assert(n >= node->num);
        bst_insert(n, node->right);
    }
}
```

- indirection (reference) is explicit, using *pointers:*

`T*`	is a pointer to type `T`
`*p`	means “follow `p`”
`p->f`	means `(*p).f`
`&n`	means “get me a pointer to `n`”

- i.e. no ‘value types’ vs ‘reference types’
  - ... can point to any type `T`
  - ... can *nest* any `T` in struct `S`

Pointers and arrays in C

```
int x = 42;          // x is an int
int *px;             // px is a pointer to int (uninitialized!)
px = &x;             // set px to point to x (“= address of x”)
*px = 43;            // assign to the thing px points to

// continuing...

int xs[] = { 2, 3, 5 };  // xs is an array of 3 ints
xs[0] = 42;
px = &xs[1];                 // now px points to… what?
printf(“The value is: %d\n”, *px);
```

`*px` is *following* a pointer... we say the pointer is *dereferenced*

Pointers and arrays in C

```
int xs[] = { 2, 3, 5 }; // xs is an array of 3 ints
px = &xs[1];            // now px points to xs[1]
++px;                   // now px points to xs[2]
--px;                   // now?
px = px – 1;            // now?
```

We’ve just seen *pointer arithmetic*.
This just means pointers can be used as “array iterators”.
Conceptually, pointers always point into an array
... if pointing at a single value, pretend it’s an array of one

```
int x; int *p = &x;     // as if “array of one”
```

Pointers and arrays in C

```
int xs[] = { 2, 3, 5 }; // xs is an array of 3 ints
px = &xs[0];            // now px points to xs[0]
px = xs;                // same effect
```

If we name an array where a pointer is required,
an implicit conversion is done
yielding a pointer to the first element of the array.

It is not the case that “arrays and pointers are the same”!

```
printf(“Array size: %d\n”, sizeof xs);    // 12 (...)
printf(“Pointer size: %d\n”, sizeof px);  // 8 (...)
```

String literals themselves are passed using this implicit conversion: “Hi!” is a char[4], but `printf` gets a pointer.

Pointers and arrays in C

It is <u>not</u> the case that “arrays and pointers are the same”! Pointers are iterators over arrays.

However, we can do *indexing* on pointers. It accesses the array that they are ranging over.

```
int xs[] = { 2, 3, 5 }; // xs is an array of 3 ints
int *px = &xs[1];        // now px points to xs[1]
printf(“Indexing px at 1 gave: %d\n”, px[1]);
```

Pointers and arrays in C

Pointer indexing is just a notation for combining pointer arithmetic and pointer dereferencing.

`px[1]` is the same as `*(px + 1)`

Alternatively, pointer arithmetic is just a notation for combining pointer indexing and address-taking!

`px + 1` is the same as `&px[1]`

Pointers and arrays in C

We can think of pointers as combining:

- a reference to an array (modulo “pretend, if it’s just one value”)

... with:

- a position in that array.

Value and reference in C

“Pointers as iterators” is a useful feature, but also dangerous, when we want to pass around arrays, as arguments or return values.

How does parameter-passing work in C?

Short answer: we pass everything by value. E.g.

```
struct point { double x; double y; };

struct point scale(struct point p, double factor) {
   p.x *= factor; p.y *= factor; return p;
}

struct point mypoint = { 1.0, 3.0 };
struct point scaled = scale(mypoint, 2.0);
```

`mypoint` is not modified! *Copies* are passed/returned.

Value and reference in C

“Pointers as iterators” is a useful feature, but also dangerous, when we want to pass around arrays, as arguments or return values.

So how do we pass arrays? <u>We can’t.</u> (Why not?)

You have to pass a pointer.

```
void scale_two(double *p, double factor) {
   p[0] *= factor; p[1] *= factor;
}

double mypoint[2] = { 1.0, 3.0 };
scale_two(mypoint, 2.0);
```

In this version, `mypoint` <u>is</u> modified!
We passed the pointer by value… it *references* the array.

Value and reference in C

There are three common idioms for passing arrays around.

- explicit length

```
void sort_n(int *arr, unsigned int n);
```

- “sentinel” a.k.a. pass the ‘end pointer’

```
void sort_between(int *begin, int *end);
```

- terminator element

```
int puts(const char *string); // simpler printf
```

A “terminated array” means a special value denotes the end of the sequence... here a null (zero) character.
-> no need to pass a length or end pointer

Closing: some real C snippets

(from musl libc, http://musl-libc.org/)

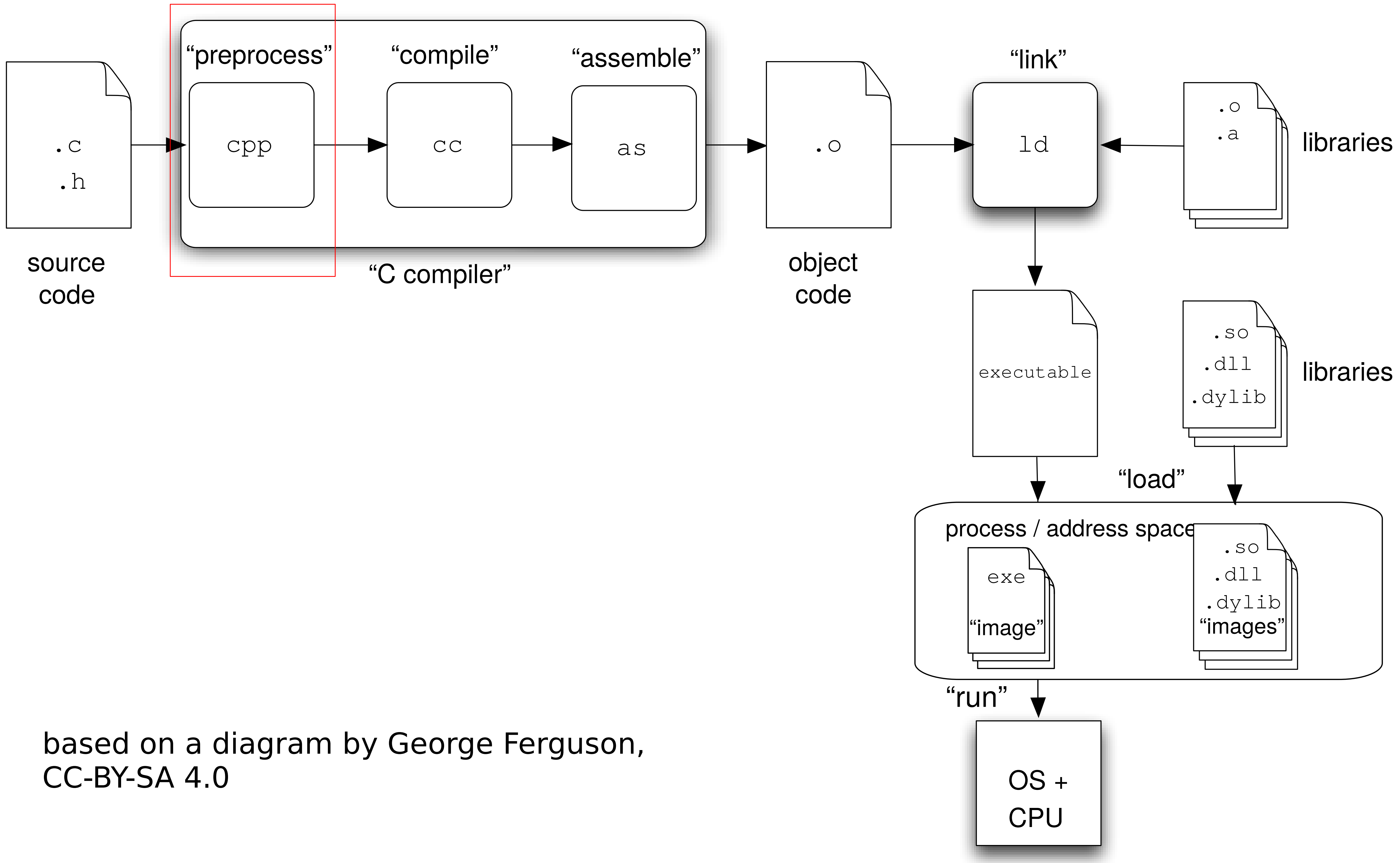
```
// duplicate the string pointed to by s
char *strndup(const char *s, size_t n)
{
        size_t l = strnlen(s, n);  // get length of string, max n
        char *d = malloc(l+1); // allocate space for the copy
        if (!d) return NULL;       // check malloc didn’t fail
        memcpy(d, s, l);           // copy ‘l’ bytes from *s to *d
        d[l] = 0;                  // write a zero terminator
        return d;
}
```

Closing: some real C snippets
(from musl libc, http://musl-libc.org/)

```
// “find the first token in the string *stringp that is delimited
// by one of the bytes in sep”
char *strsep(char **str, const char *sep)
{
    char *s = *str, *end;
    if (!s) return NULL;            // here strcspn finds the
    end = s + strcspn(s, sep); // offset of the first char in ‘s’
    if (*end) *end++ = 0;       // that is present in string ‘sep’
    else end = NULL;
    *str = end;                         // update the caller’s pointer
    return s;
}
```

Notice the first argument: this “passes a pointer by reference”, by passing a pointer to the pointer. This is so that the function can *modify* a pointer provided by the caller.

Preprocessing



based on a diagram by George Ferguson, CC-BY-SA 4.0

```
// tree.c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

struct bst {
    struct bst *left;
    int num;
    struct bst *right;
};
static struct bst *root;
static
void bst_insert(int n, struct bst *node) {
    struct bst *new_node = calloc(1,
       sizeof (struct bst));
    new_node->num = n;
    if (!node) { // empty tree
        assert(!root);
        root = new_node;
    } else if (n < node->num && !node->left) {
      node->left = new_node;
    } else if (n >=node->num && !node->right){
        node->right = new_node;
    } else if (n < node->num) {
        bst_insert(n, node->left);
    } else {
        assert(n >= node->num);
        bst_insert(n, node->right);
    }
}
```

- Preprocessor directives begin with a ‘#’

- They are not part of the C language per se!

- They are processed, and thereby eliminated, before code is compiled

- Here #include will paste the contents of the named file (found at some path known to the compiler) into the current file...

- ... recursively (i.e. that file may #include others)

```
// tree.c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

struct bst {
    struct bst *left;
    int num;
    struct bst *right;
};
static struct bst *root;
static
void bst_insert(int n, struct bst *node) {
    struct bst *new_node = calloc(1,
        sizeof (struct bst));
    new_node->num = n;
    if (!node) { // empty tree
        assert(!root);
        root = new_node;
    } else if (n < node->num && !node->left) {
       node->left = new_node;
    } else if (n >=node->num && !node->right){
        node->right = new_node;
    } else if (n < node->num) {
        bst_insert(n, node->left);
    } else {
        assert(n >= node->num);
        bst_insert(n, node->right);
    }
}
```

These '.h' *header* files contain declarations:

- function signatures for library code (not code itself; just its declaration) e.g.
`int printf(const char *s, ...);`

- declarations of global variables, e.g.
`char** environ;`

- type definitions used in the above (e.g. `struct _IO_FILE`)

... in short, defining the *interface* of library code

```
// tree.c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

struct bst {
     struct bst *left;
     int num;
     struct bst *right;
};
static struct bst *root;
static
void bst_insert(int n, struct bst *node) {
     struct bst *new_node = calloc(1,
         sizeof (struct bst));
     new_node->num = n;
     if (!node) { // empty tree
          assert(!root);
          root = new_node;
     } else if (n < node->num && !node->left) {
        node->left = new_node;
     } else if (n >=node->num && !node->right){
          node->right = new_node;
     } else if (n < node->num) {
          bst_insert(n, node->left);
     } else {
          assert(n >= node->num);
          bst_insert(n, node->right);
     }
}
```

In a more realistic example, some of the code on the left would actually appear in a .h file, not a .c file

e.g. the struct definition

and signatures for the "public" functions of the module

If you want a file to export code to other files, create a header for it, and include that header from the other files. (<u>And</u> from the current file, to save duplication.)

Conditional compilation

```
struct sigaction action = {
        .sa_handler = &handle_sigill,
        .sa_flags = SA_RESTORER | SA_NODEFER
    #ifndef __FreeBSD__
        , .sa_restorer = restore_rt
    #endif
};
```

Sometimes to make code portable, we have to build in per-platform variations.

In this example, FreeBSD defines a struct slightly differently to other OSes, lacking a particular field.

Usually the condition is whether a given *preprocessor macro* is defined (#ifdef)

- here __FreeBSD__
- this is a built-in macro which is defined by the FreeBSD preprocessor but not others

Conditional compilation

```
#if __STDC_VERSION__ >= 201112L
/* Use the new threading library. */
#include <threads.h>
#else
/* Hope we have the pthreads library */
#include <pthread.h>
#endif
```

As well as #ifdef, there is also
#if
which can evaluate simple conditional expressions in a C-style syntax

Remember: this isn't C!

It runs *before* compile time. Program variables cannot be used.

__STDC_VERSION__ is a *macro* that is defined with a value (at least 201112 in a C11-conforming compiler)

Symbolic constants

In the old days, there was no way to declare constants in C. So we used the preprocessor instead.

```
#define PI 3.14159

double circle_area(double r) {
   return PI * r * r;
}

// .. becomes..
```

By the time the compiler gets to see this code…

```
double circle_area(double r) {
   return 3.14159 * r * r;
}
```

… PI has been macro-expanded away by the preprocessor, and it looks like this.

```
// can nowadays write...
const double PI = 3.14159;
// ... but less easily optimised
```

It’s still useful to define constants this way. For reasons to do with *linking*, the compiler can’t assume this value is fixed yet.

#include guards

```
// myheader.h
#ifndef MYHEADER_H_
#define MYHEADER_H_

int myfunc(int x);
struct mystruct { int y; };
// etc

#endif // i.e. “else say nothing!”
```

```
// myotherheader.h
#ifndef MYOTHERHEADER_H_
#define MYOTHERHEADER_H_

#include “myheader.h”
struct myotherstruct {
    struct mystruct nested;
    double z;
};

#endif
```

```
// program.c
#include “myheader.h”
#include “myotherheader.h”
...
```

It’s idiomatic to use some preprocessor directives at the beginning and end of header files...

... mainly to help speed up compilation, by preventing the same things from being included two or more times

Here our client doesn’t need to know that myotherheader includes myheader – but we still avoid sending two copies to the compiler

#define for *function-like* macros

Macros can be used to perform some function-like substitutions.

Again: this isn’t C! Macros consume tokens and generate tokens.

Useful to avoid repetition…

… or more generally *generate* snippets of code (metaprogramming).

Historically used to gain the effect of *inline* functions

```
// we previously wrote:
struct bst *new_node = calloc(1,
        sizeof (struct bst));

// but if we define
#define NEW_ARRAY(len, t) \
    calloc((len), sizeof (t))

// ... we can instead write
struct bst *new_node
 = NEW_ARRAY(1, struct bst));

// or doing more:
#define ARR_DECL_ALLOC( \
  len, name, t) \
    t *name = calloc((len), sizeof (t))

// then our whole calloc line can be:
ARR_DECL_ALLOC(new_node, 1,
 struct bst);
```

Preprocessor summary

The C preprocessor is a fairly general-purpose "level of syntactic abstraction".

It is a "Swiss Army Knife" sort of tool...

... albeit fairly crude.

Used well, it helps keep code concise, portable and maintainable.

Used badly, it creates confusion.

It does not know C! It just works blindly by crunching files and tokens

The last chunk: gotchas and demos

I’ll do a walk-through covering some common gotchas:

- containment recap
- meaning of const
- function pointers
- “declaration reflects use”
- typedef
- lvalues and rvalues
- passing pointers up the stack

... and demoing some tools

- gdb, a.k.a. how to debug your code
- Godbolt’s “Compiler Explorer”

References

As last time, plus...

Kernighan. “Why Pascal is not my favourite programming language”
(an insight into why arrays in C are as they are)
http://www.eprg.org/computerphile/pascal.pdf