Lecture 10: Advanced C

Computer Architecture and Systems Programming (252-0061-00)

Timothy Roscoe

Herbstsemester 2012

Last time

- Memory layout
- Buffer overflow, worms, and viruses
- Program optimization
  - Removing unnecessary procedure calls
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions

Today

A tour of some advanced features, and techniques, available in C:

- Operators
- Function pointers
- Typedefs and structures
- **`goto`**
- Assertions
- Are arrays the same as pointers?
- **`setjmp()/longjmp()`**
- Coroutines

Assignment operators

- In many imperative languages

```
x = foo();
```

  - is an assignment **statement**.

- In C, it is an **expression!**
  - Value is the value being assigned

Post-increment and pre-increment

- **`i++`**
  - Value: current value of **`i`**
  - Effect: **`i`** ← **`i+1`**
- **`++i`**
  - Effect: **`i`** ← **`i+1`**
  - Value: new value of **`i`**
- Conversely **`i--`** and **`--i`**
- Works for any scalar type
  - Importantly: works for pointers!

Historical: Digital PDP computers had pre- and post- increment and decrement addressing modes

Common C idioms

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    FILE *f;
    ...
    if (!(f = fopen(argv[1], "r"))) {
        perror("Opening file");
        exit(-1);
    }
    ...
}
```

```
char *strcpy(char *dest, char *src)
{
    char *r = dest;
    while(*dest++ = *src++);
    return r;
}
```

Lots going on here!

- Assigment is an expression, not a statement
- Many C functions return NULL pointers to indicate failure
- Non-zero values evaluate to true, zero evaluates to false
- Strings are arrays of characters terminated by null bytes
- Post-increment operators bind more tightly that pointer dereference

Don't write C like this:

(Carl Shapiro 1985,
International Obfuscated C Code Contest
"Grand prize for most well-ro

An assignment is also an expression.

```
#define P(X)j=write(1,X,1)
#define C 39
int M[5000]={2},*u=M,N[5000],R=22,a[4],l[]={0,-1,C-1,-1},m[]={1,-C,-1,C},*b=N,
*d=N,c,e,f,g,i,j,k,s;main(){for(M[i=C*R-1]=24;f|d>=b;){c=M[g=i];i=e;for(s=f=0;
s<4;s++)if((k=m[s]+g)>=0&&k<C*R&&l[s]!=k%C&&(!M[k]||!j&&c>=16!=M[k]>=16))a[f++
]=s;if(f)f=M[e=m[s=a[rand()/(1+2147483647/f)]]+g];j=j<f?f:j;f+=c&-16*!j;M[g]=
c|1<<s;M[*d++=e]=f|1<<(s+2)%4;}else e=d>b++?b[-1]:e;}P(" ");for(s=C;--s;P("_")
)P(" ");for(;P("\n"),R--;P("|"))for(e=C;e--;P("_ "+(*u++/8)%2))P("| "+(*u/4)%2
);}
```

Subtle use of C's little-known comma operator!

Strings are pointers too!

- Although it has a certain kind of fascination…

Today

- Operators
- Function pointers
- Typedefs and structures
- **goto**
- Assertions
- Are arrays the same as pointers?
- **setjmp()/longjmp()**
- Coroutines

Function pointers

```
int (*func)(int *, char);
```

- "**func** is a pointer to a function which takes two arguments, a pointer to **int** and a **char**, and returns an **int**"
- Not often seen in OO languages, but c.f. C# "delegates".
- As with all types, can be used with **typedef**
- Basis for lots of techniques in Systems code

From the Linux kernel…

```
struct file_operations {
    ...
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ...
};
```

- Record (struct) type of function pointers
  - Sometimes called a vtable
  - Each member is a different function type
  - Implements a "method" (or "feature") of some "object"
    - In this case, a struct file
  - Provides polymorphism (e.g. files and sockets)
- Not really objects: no protection or hiding
  - But does provide the reuse benefits

Today

- Operators
- Function pointers
- Typedefs and structures
- **goto**
- Assertions
- Are arrays the same as pointers?
- **setjmp()/longjmp()**
- Coroutines

Struct tags and typedefs

Structure "tag"

```
struct list_el {
    unsigned long val;
    struct list_el *next;
};

struct list_el my_list;
```

Need the **struct** keyword to refer to a tag

Also applies to unions…

Struct tags and typedefs

- You can:

```
typedef struct list_el el_t;
```

- Or even:

```
typedef struct list_el {
    unsigned long val;
    struct list_el *next;
} el_t;

struct list_el my_list;
el_t my_other_list;
```

- Or *even:*

```
typedef struct list_el {
    unsigned long val;
    struct list_el *next;
} list_el;

struct list_el my_list;
list_el my_other_list;
```

Confusing.

Better to stay with always using tags.

C namespaces

1. Label names
   – (see goto later…)
2. Tags
   – one namespace for all **`struct`**, **`union`**, **`enum`**'s
3. Member names
   – One namespace for each **`struct`**, **`union`**, **`enum`**
4. Everything else (mostly)
   – Including **`typedef`**

Today

- Operators
- Function pointers
- Typedefs and structures
- **`goto`**
- Assertions
- Are arrays the same as pointers?
- **`setjmp()/longjmp()`**
- Coroutines

When to use **`goto`**

1. Don't.
   – Almost never a good idea
   – Usually argued on performance grounds
     - E.g. implementing state machines
   – Often fast, but other ways are often just as fast
     - E.g. switches and function pointers
2. Early termination of multiple loops
3. Nested cleanup code

Early termination of nested loops

```
int a[MAXROW][MAXCOL]

int i,j;
int found=0;
for(i=0; i< MAXROW; i++) {
    for(j=0; j < MAXCOL; j++) {
        if (a[i][j] == 0) {
            found=1;
            break;
        }
    }
    if (found)
        break;
}
if (found) {
    <do processing on i and j>
} else {
    <do processing when not found>
}
```

```
int a[MAXROW][MAXCOL]

  int i,j;
  for(i=0; i< MAXROW; i++) {
      for(j=0; j < MAXCOL; j++) {
          if (a[i][j] == 0) {
              goto found;
          }
      }
  }
  <do some processing when not found>
found:
  <do some processing on i and j>
```

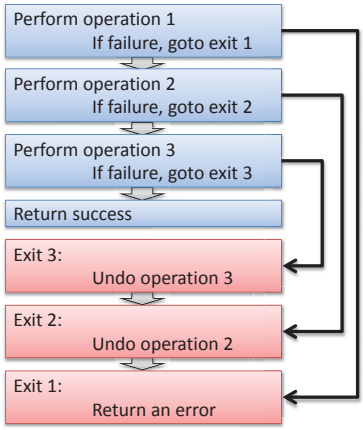
- But consider simply using **`return()`**
- Other languages have **`break(label)`**

Cleanup conditions

Goto is used for *recovery code*

- General idea:
  – Code performs a sequence of operations
  – Any one can fail
  – If one fails, all previous operations must be ***undone***
- Canonical example:
  – Malloc'ing a sequence of buffers for data
  – If one fails, must free all previous generated buffers
- Code is often (not always) auto-generated

Cleanup code



- Not easy to represent scalably in "structured programming"
- Best effort: nested "if" statements
  - But can get arbitrarily deep in a complex function
- "goto" version is highly stylized (undos can be paired with operations in reverse order)
- Can also be modified to handle cases where one "undo" deals with several operations…

Cleanup conditions

```
int nfs2_decode_dirent(struct xdr_stream *xdr, struct nfs_entry *entry, int plus)
{
        __be32 *p;
        int error;

        p = xdr_inline_decode(xdr, 4);
        if (unlikely(p == NULL))
                goto out_overflow;
        if (*p++ == xdr_zero) {
                p = xdr_inline_decode(xdr, 4);
                if (unlikely(p == NULL))
                        goto out_overflow;
                if (*p++ == xdr_zero)
                        return -EAGAIN;
                entry->eof = 1;
                return -EBADCOOKIE;
        }

        p = xdr_inline_decode(xdr, 4);
        if (unlikely(p == NULL))  goto out_overflow;
        entry->ino = be32_to_cpup(p);

        error = decode_filename_inline(xdr, &entry->name, &entry->len);
        if (unlikely(error))  return error;

        entry->prev_cookie = entry->cookie;
        p = xdr_inline_decode(xdr, 4);
        if (unlikely(p == NULL))  goto out_overflow;
        entry->cookie = be32_to_cpup(p);

        entry->d_type = DT_UNKNOWN;

        return 0;
out_overflow:
        print_overflow_msg(__func__, xdr);
        return -EAGAIN;
}
```

- From the Linux NFS2 kernel code
- Network marshalling code
  - canonical use-case for this style
- Often generated by a stub compiler
  - though not in this case

Today

- Operators
- Function pointers
- Typedefs and structures
- `goto`
- Assertions
- Are arrays the same as pointers?
- `setjmp()/longjmp()`
- Coroutines

Assertions

`assert( <scalar expression> );`

- At run time, evaluate the expression.
- If true, do nothing
- If false:
  - Print "*`file`*`.c:`*`line`*`: func: Assertion` `` `expr' failed.`"
  - Abort (dump core)

Assertion example

```
#include <assert.h>

// It's a bug to call this with null a or b
void array_copy(int a[], int b[], size_t count)
{
    int i;
    assert(a != NULL);
    assert(b != NULL);
    for(i=0; i<count; i++) {
        a[i] = b[i];
    }
}
```

```
int main(int argc, char *argv[])
{
    // This is not going to go well...
    array_copy(NULL, NULL, 0);
    return 0;
}
```

```
$ ./a.out
a.out: assert_text.c:8: array_copy: Assertion `a != ((void *)0)' failed.
Aborted
$
```


Assertions

- Almost a poor person's contracts
  - Checked at runtime, not compile time
  - Can be compiled out (**`-DNDEBUG`**)
    ⇒ no runtime overhead in finished(!) code
- It's a macro (why?)
  ⇒ Should not contain side-effects!
- Assertions **are *of no use to the user!***

Don't use assertions like this

```
#include <assert.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    assert( argc != 2 );

    ...

    FILE *f = fopen(argv[1]);
    assert( f != NULL );

    ...

}
```

These conditions are not bugs!

Assertions are for programmers to find bugs, not for programs to detect errors.

Today

- Operators
- Function pointers
- Typedefs and structures
- `goto`
- Assertions
- Are arrays the same as pointers?
- `setjmp()/longjmp()`
- Coroutines

Arrays and pointers

- An array name *in an expression* is treated as a pointer to the first element of the array…

```
int a[10];
assert(a == &(a[0]));
```

```
int get_1(int i)
{
    int *p = a;
    return p[i];
}
void set_1(int i, int v)
{
    int *p = a;
    p[i] = v;
}
```

```
int get_2(int i)
{
    int *p = a;
    return *(p+i);
}
void set_2(int i, int v)
{
    int *p = a;
    *(p+i) = v;
}
```

```
int get_3(int i)
{
    int *p = a+i;
    return *p;
}
void set_3(int i, int v)
{
    int *p = a+i;
    *p = v;
}
```


… except when

1. The array's address is taken with '&'

```
int a[10];
assert( &a == a )
```

2. The array is a string literal initializer
3. The array is an operand of sizeof()

```
int a[10];
assert( sizeof(a)     == 10*sizeof(int));
assert( sizeof(&a[0]) == sizeof(int *));
```


In fact…

- A[i] is *always* rewritten *(A+i) in the compiler

```
int a[10];
assert(a == &(a[0]));
assert(a[5] == 5[a]);
```

```
int get_2(int i)
{
    int *p = a;
    return i[p];
}
void set_2(int i, int v)
{
    int *p = a;
    i[p] = v;
}
```


An array name as a function parameter is a pointer

- The following are all precisely equivalent:

```
int arrfun( int *myarray )
{
    …
}
```

```
int arrfun( int myarray[] )
{
    …
}
```

```
int arrfun( int myarray[42] )
{
    …
}
```

Functions are also converted to pointers like this!

… and can be called in any of these ways:

```
int some_int;
int *some_ptr;
int some_array[10];

arrfun( &some_int );
arrfun( some_ptr );
arrfun( some_array );
arrfun( &some_array[i] );
```

… all of which turn into pointers.

You can't rename an array

- Compile-time error:

```
int array1[42], array2[42];
main(int argc, char *argv[])
{
    array1 = array2;
    return 0;
}
```

- But this is OK (it's a pointer):

```
void arrfun(int array[])
{
    array = array2;
}
```


Today

- Operators
- Function pointers
- Typedefs and structures
- `goto`
- Assertions
- Are arrays the same as pointers?
- **`setjmp()/longjmp()`**
- Coroutines

`setjmp()`

```
#include <setjmp.h>

int setjmp(jmp_buf env);
```

- **`setjmp(env)`**:
  - Saves the current stack state / environment in **env**
  - Returns 0.
- Why is this useful? Well…

`longjmp()`

```
#include <setjmp.h>

void longjmp(jmp_buf env, int val);
```

- **`longjmp(env,val)`**:
  - Causes *another* return to the point saved by **env**
  - This new return returns **`val`** (or 1 if **`val`** is 0)
  - This should only be done *once* for each **`setjmp()`**
  - It is invalid if the function containing the **`setjmp`** returns
- Very few programming languages have a construct like this… ☺

Toy example

(from wikipedia)

```
#include <stdio.h>
#include <setjmp.h>

static jmp_buf buf;

void second(void) {
    printf("second\n");
    longjmp(buf,1);
}

void first(void) {
    second();
    printf("first\n");          // does not print
}

int main() {
    if ( ! setjmp(buf) ) {
        first();                // when executed, setjmp returns 0
    } else {                    // when longjmp jumps back, setjmp returns 1
        printf("main\n");       // prints
    }

    return 0;
}
```

Output:
second
main

Today

- Operators
- Function pointers
- Typedefs and structures
- `goto`
- Assertions
- Are arrays the same as pointers?
- `setjmp()/longjmp()`
- Coroutines

What are coroutines?

```
/* Decompression code */
while (1) {
    c = getchar();
    if (c == EOF)
        break;
    if (c == 0xFF) {
        len = getchar();
        c = getchar();
        while (len--)
            emit(c);
    } else
        emit(c);
}
emit(EOF);
```

emit(c)

c=getchar()

```
/* Parser code */
while (1) {
    c = getchar();
    if (c == EOF)
        break;
    if (isalpha(c)) {
        do {
            add_to_token(c);
            c = getchar();
        } while (isalpha(c));
        got_token(WORD);
    }
    add_to_token(c);
    got_token(PUNCT);
}
```

Unfortunately, you can't do this...

c.f. Python, C++ iterators....

from Simon Tatham: http://pobox.com/~anakin/

Conventional approach: rewrite at least one of them

```
int decompressor(void) {
    static int repchar;
    static int replen;
    if (replen > 0) {
        replen--;
        return repchar;
    }
    c = getchar();
    if (c == EOF)
        return EOF;
    if (c == 0xFF) {
        replen = getchar();
        repchar = getchar();
        replen--;
        return repchar;
    } else
        return c;
}
```

Called repeatedly to return a character

Or

A bit clumsy!

Called repeatedly passing character as argument:

```
void parser(int c) {
    static enum {
        START, IN_WORD
    } state;
    switch (state) {
        case IN_WORD:
        if (isalpha(c)) {
            add_to_token(c);
            return;
        }
        got_token(WORD);
        state = START;
        /* fall through */

        case START:
        add_to_token(c);
        if (isalpha(c))
            state = IN_WORD;
        else
            got_token(PUNCT);
        break;
    }
}
```

from Simon Tatham: http://pobox.com/~anakin/

What we'd like...

```
/* Decompression code */
while (1) {
    c = getchar();
    if (c == EOF)
        break;
    if (c == 0xFF) {
        len = getchar();
        c = getchar();
        while (len--)
            emit(c);
    } else
        emit(c);
}
emit(EOF);
```

Decompressor calls parser when a new character is ready

Parser calls decompressor when it needs a new character

```
/* Parser code */
while (1) {
    c = getchar();
    if (c == EOF)
        break;
    if (isalpha(c)) {
        do {
            add_to_token(c);
            c = getchar();
        } while (isalpha(c));
        got_token(WORD);
    }
    add_to_token(c);
    got_token(PUNCT);
}
```


What we'd really like: *continuations*

- Decompressor runs until it has a character to emit
  - Saves its state (stack, variables, etc.)
  - Calls into the Parser
- Parser continues where it previously left off
  - Processes new character
  - Runs until it needs a new character
  - Calls back to the Decompressor
- Decompressor continues where it previously left off
- U.s.w.!

Languages like Scheme have a primitive for this: CallCC
This is hard to do in a stack-based language like C.
But: coroutines achieve something very close

Coroutines

(really minimal implementation)

```
#include <setjmp.h>

struct Coro {
        void *stack;
        jmp_buf env;
};

extern struct  Coro *Coro_new(void);
extern void Coro_free(struct  Coro *self);

typedef void (CoroSCB)(void *);

extern void Coro_start (struct  Coro *self, struct Coro *other,
                        void *context,  struct CoroSCB *callback);
extern void Coro_switchTo(struct Coro *self, struct  Coro *next);
```

Creation/deletion

```
struct Coro *Coro_new(void)
{
    struct Coro *self = (struct Coro *)calloc(1,
                                    sizeof(struct Coro));
    self->stack = (void *)calloc(1, 65536 + 16);
    return self;
}
```

```
void Coro_free(struct Coro *self)
{
    free(self->stack);
    free(self);
}
```


Switching

```
void Coro_switchTo(struct Coro *self,
                   struct Coro *next)
{
    if (setjmp(self->env) == 0)
    {
       longjmp(next->env, 1);
    }
}
```

- First return from **`setjmp()`**:
  ⇒ **`longjmp`** to the next coroutine
- Second return:
  ⇒ continue where we left off

Initialization (the hard bit)

```
struct CallbackBlock {
    void *context;
    CoroSCB *func;
};
```

This is a ***closure***: a function bundled with a set of arguments.

```
void Coro_Start_(struct CallbackBlock *block)
{
    (block->func)(block->context);
    printf("Scheduler error: returned from coro start!\n");
    exit(-1);
}
```


Initialization (the hard bit)

```
void Coro_start(struct Coro *self, struct Coro *other,
                void *context, struct CoroSCB *callback)
{
    struct CallbackBlock sblock;
    struct CallbackBlock *block = &sblock;
       block->context = context;
       block->func    = callback;

       setjmp(self->env);
       self->env[0].__jmpbuf[6] = ((unsigned long)(self->stack));
       self->env[0].__jmpbuf[7] = ((long)Coro_Start_);
       Coro_switchTo(self, other);
}
```

Machine-dependent hack:
6 ⇒ **`%esp`**
7 ⇒ **`%eip`**

Now: replace emit() and getchar() in original code with Coro_switchTo and you're done.

Coroutines and threads

- Coroutines are sometimes called:
  - Lightweight threads
  - Protothreads
  - Cooperative multitasking
  - Etc.
- You can make this look like threads:
  - Write a new function: Coro_yield()
  - Picks a new coroutine to run, and calls Coro_switchTo
  - Implement blocking I/O
    - coroutines become runnable or non-runnable
    - Etc. etc.

But this is misleading…

- There is no concurrency here
  - And no parallelism either!
- Coroutines are a generalization of subroutine call/return
  - More intuitive than raw continuations
  - Less powerful, but still important
  - Correspond to many OS-level scenarios
- Good example of "Systems Programming"
  - Rarely seen in, e.g. Computational Science code.

Summary

- Operators
- Function pointers
- Typedefs and structures
- **`goto`**
- Assertions
- Are arrays the same as pointers?
- **`setjmp()/longjmp()`**
- Coroutines

Next time: basic x86 processor design