Lesson 7

Christian Schwarz, Jakob Krebs

9.12.2019

Contents

Sources and Solutions

- we publish all code written in this course at `https://github.com/jkrbs/c_lessons`
- we will publish example solutions of the tasks on same site
- send us questions or your solutions to c-lessons@deutschland.gmbh

Function Pointers

Function Pointers

- As we know, a pointer is really just a memory address
- The code for functions is also in memory, so it also has an address.
- As long as different functions have the same argument types, we can call them using a function pointer:

```
int foo(int x, int y){ puts("called foo"); }
<<<<<<< HEAD
int foo(int x, int y){ puts("called bar"); }
=======
int bar(int x, int y){ puts("called bar"); }
>>>>>>> master
// func_ptr is the newly created function pointer
int (*func_ptr)(int,int);
func_ptr = &foo; // we give it foo's address
func_ptr(1,2); // we call foo
func_ptr = bar; // the '&' is optional, think arrays
```

Function Pointer Typedefs

Since the above type declaration is slighly complicated, it's often a good idea to alias the function type:

```
// we can give the parameters names but it's optional
typedef int (*event_callback)(int event_id, void* context);

// now we can initialize a variable like this:
event_callback callback = &my_event_handler;
int result = callback(1, NULL);
```

Type Qualifiers

const

To give more information about a variable to the compiler, you can *qualify* its type.

The most common type qualifier is `const`. It prevents the qualified variable from being modified. If you try anyways you will get a compiler error.

```
// request that x can't be written to (after initialization)
int const x = 3;
x = 3; // error: assignment of read-only variable 'x'

void f(int *a); // forward declaration for f
f(&i); // warning: 'foo' [...] discards 'const' qualifier [...]
```

But this is C we're talking about, so of course there's a way: `*(int*)&x = 3;`

Compiles no problem. **But** what happens is undefined behaviour, so your progamm would no longer be valid.

From the west-const to the east-const

Normally, a qualifier refers to the type to its **left**, but the following is also valid (and more common!):

```
const int a;              // equal to ‘int const a‘
```

Watch complex types:

```
const int *foo;           // mutable pointer, constant integer
int const *foo;           // same as above
int * const foo;          // constant pointer, mutable integer
int const * const foo;    // everything constant
```

volatile

`volatile` prevents the compiler from doing aggressive optimizations on a variable. For example:

```
volatile bool interrupt_occured = false;
while(!interrupt_occured){
    // we don't change interrupt_occured, so the compiler
    // might assume that it can optimize away the check
}
```

This is mainly used in low-level programming:

- Hardware access (memory-mapped I/O)
- Threading (another thread modifies a value) (**!!** be very careful here)

volatile example C

```c
#include <stdio.h>

int main(void) {
    int i = 42;
    printf("%d\n", i);
}
```

```c
#include <stdio.h>

int main(void) {
    volatile int i = 42;
    printf("%d\n", i);
}
```

volatile example assembly

After compilation with *gcc -O3*:

```
.LC0:
    .string "%d\n"
main:
    sub     rsp, 8
    mov     esi, 42
    mov     edi, OFFSET FLAT:.LC0
    xor     eax, eax
    call    printf
    xor     eax, eax
    add     rsp, 8
    ret
```

```
.LC0:
    .string "%d\n"
main:
    sub     rsp, 24
    mov     edi, OFFSET FLAT:.LC0
    xor     eax, eax
    mov     DWORD PTR [rsp+12], 42
    mov     esi, DWORD PTR [rsp+12]
    call    printf
    xor     eax, eax
    add     rsp, 24
    ret
```

The compiler could not pass 42 to `printf` directly once we made `i` `volatile`.

restrict

Restrict guarantees to the compiler that nobody else is writing to the memory of a pointer (the pointer is not aliased). Therefore the compiler might do an optimization like this:

```c
void f(char *restrict p1, char *restrict p2) {
    for (int i = 0; i < 50; i++) {
        p1[i] = 4;
        p2[i] = 9;
    }
}
// optimized version, only valid if p1 and p2 don't overlap
void f(char *restrict p1, char *restrict p2) {
    memset(p1, 4, 50);
    memset(p2, 9, 50);
}
```

Since this is purely an optimization, `restrict` never changes the output of a valid program.

Parallelism

Executing code in parallel

Each program has a process associated with it. At program start, this process has exactly one thread executing your `main` function.

To achieve parallelism, you can

- create a new process running the same code
- call a function in a new thread

In Unix systems, processes are created with the `fork` system call.
The new process will have its own memory to work with.
For starting threads, libraries such as `p[osix]threads` are used.
All threads of a process share the same memory.

Use the fork

```
#include <unistd.h>
int main(void) {
    pid_t pid = fork();
    if (pid == 0) {
        /* do stuff in child process */
    } else if (pid > 0) {
        /* do stuff in parent process */
    } else {
        /* fork failed */
        return 1;
    }
    return 0;
}
```

Have a look at `man 2 fork` for further information.

pthreads

pthread_create

To execute a function in a new thread, use:

```
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine) (void *),
                   void *arg);
```

where

- * `thread` is where the thread's id will be stored
- * `attr` contains attributes for the thread (pass `NULL` for default)
- `start_routine` is the function to execute. Both the single argument and the return value must be `void *`.
- `arg` is passed to the function to be used as an argument

code example: threads

```c
#include <pthread.h>
#include <stdio.h>

void *hello_thread(void *tid) {
    printf("Hello, I am thread %d\n", *(int*) tid);
    pthread_exit(NULL);
}

int main(void) {
    pthread_t threads[5];
    for (int i=0; i < 5; ++i) {
        if (pthread_create(&threads[i], NULL,
                           hello_thread, (void *) i))
            return 1;
    }
    return 0;
```

How threads end

- `pthread_exit` is called

```c
void pthread_exit(void *retval);
```

- `pthread_cancel` is called from another thread

```c
int pthread_cancel(pthread_t thread);
```

- `exit` is called from any thread (ending the process)

Waiting for threads

To wait for a thread to finish, there is `pthread_join`

```c
int pthread_join(pthread_t thread, void **retval);
```

The thread passed to `pthread_join` must be joinable. The default is joinable, but one can disable this.

code example: joinable threads

```
int main(void) {
    pthread_t threads[5];

    for (int i=0; i < 5; ++i) {
        if (pthread_create(&threads[i], NULL,
                           hello_thread, (void *) i))
            return 1;
    }

    void *st;
    for (int i=0; i < 5; ++i) {
        if (pthread_join(thread[i], &st))
             return 1;
        printf("Thread %d finished with %d\n", i, *(int *) st);
    }

```

Mutual Exclusion

Mutexes

Threads can communicate with each other by manipulating global variables or the value behind the `arg` pointer we pass to `pthread_create`.
To avoid race conditions, the pthread library provides mutexes.

```c
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                       const pthread_mutexattr_t *restrict attr);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

A mutex is a datatype that can be locked before and unlocked after accessing a variable.

```c
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

mutex example

```c
struct stuff {
    unsigned a;
    unsigned b;
}
struct stuff global = {1, 2};
pthread_mutex_t mutex;
//initialize the mutex
pthread_mutex_init(&mutex, NULL);
//[...]
void *thread(void *tid) {
    pthread_mutex_lock(mutex);
    global.b = a;
    pthread_mutex_unlock(mutex);
    pthread_exit(NULL);
}
```

Deadlocks incoming

```c
void *thread_1(void *tid) {
    pthread_mutex_lock(mutex_1);
    pthread_mutex_lock(mutex_2);
    /* do stuff */
    pthread_mutex_unlock(mutex_1);
    pthread_mutex_unlock(mutex_2);
    pthread_exit(NULL);
}
void *thread_2(void *tid) {
    pthread_mutex_lock(mutex_2);
    pthread_mutex_lock(mutex_1);
    /* do stuff */
    pthread_mutex_unlock(mutex_2);
    pthread_mutex_unlock(mutex_1);
    pthread_exit(NULL);
}
```