Programovanie v operačných systémoch

03 - Resources

Jozef Šiška

Department of Applied Informatics
Comenius University in Bratislava

2018/2019

Resource Management

- acquire
    - allocate memory, open file, lock mutex
- release
    - release memory, close file, unlock mutex

- leaks (not releasing resources)
- ownership

Resource management in C, take 1

```
int do_something()
{
    int fd_in, fd_out;
    char *buffer;

    if ((fd_in = open(.....)) == -1)
        return -1;
    if ((fd_out = open(....)) == -1)
        return -1; // !!!

    if ((buffer == malloc(...)) == NULL)
        return -1; // !!!

    // do something

    free(buffer);
    close(fd_out);
    close(fd_int);

    return 0;
}
```

Resource management in C, take 2

```
int do_something()
{
    int fd_in, fd_out;
    char *buffer;

    if ((fd_in = open(.....)) == -1)
        return -1;
    if ((fd_out = open(....)) == -1)
        close(fd_in);
        return -1;

    if ((buffer == malloc(...)) == NULL)
       close(fd_out);
       close(fd_in);
        return -1;

    // more resources ?!

    free(buffer);
    close(fd_out);
    close(fd_int);

    return 0;
}
```

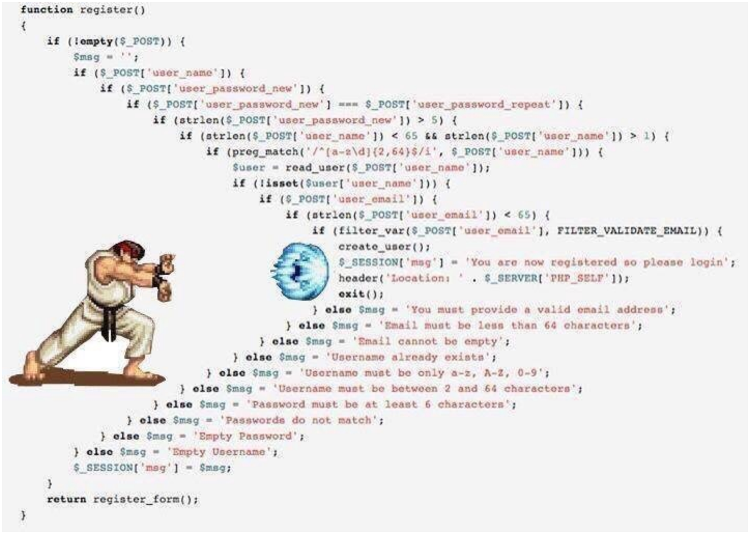
Resource management in C, take 3

```
int do_something()
{
    int fd_in, fd_out, ret = -1;;
    char *buffer;

    if ((fd_in = open(.....)) != 1) {
        if ((fd_out = open(....)) != -1) {
            if ((buffer == malloc(...)) != NULL) {
                // do something...
                // ret = 0;
                free(buffer);
            }
            close(fd_out);
        }
        close(fd_in);
    }

    return ret;
}
```

—-> arrow functions



```
function register()
{
    if (!empty($_POST)) {
        $msg = '';
        if ($_POST['user_name']) {
            if ($_POST['user_password_new']) {
                if ($_POST['user_password_new'] === $_POST['user_password_repeat']) {
                    if (strlen($_POST['user_password_new']) > 5) {
                        if (strlen($_POST['user_name']) < 65 && strlen($_POST['user_name']) > 1) {
                            if (preg_match('/^[a-z\d]{2,64}$/i', $_POST['user_name'])) {
                                $user = read_user($_POST['user_name']);
                                if (!isset($user['user_name'])) {
                                    if ($_POST['user_email']) {
                                        if (strlen($_POST['user_email']) < 65) {
                                            if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {
                                                create_user();
                                                $_SESSION['msg'] = 'You are now registered so please login';
                                                header('Location: ' . $_SERVER['PHP_SELF']);
                                                exit();
                                            } else $msg = 'You must provide a valid email address';
                                        } else $msg = 'Email must be less than 64 characters';
                                    } else $msg = 'Email cannot be empty';
                                } else $msg = 'Username already exists';
                            } else $msg = 'Username must be only a-z, A-Z, 0-9';
                        } else $msg = 'Username must be between 2 and 64 characters';
                    } else $msg = 'Password must be at least 6 characters';
                } else $msg = 'Passwords do not match';
            } else $msg = 'Empty Password';
        } else $msg = 'Empty Username';
        $_SESSION['msg'] = $msg;
    }
    return register_form();
}
```

https://medium.com/@mena.meseha/how-to-refactor-the-arrow-type-code-8c40c21db85f

Resource management in C, take 4

```
int do_something()
{
    int fd_in, fd_out, ret = -1;
    char *buffer;

    if ((fd_in = open(.....)) == -1)
        goto err_fd_in;
    if ((fd_out = open(....)) == -1)
        goto err_fd_out;

    if ((buffer == malloc(...)) == NULL)
        goto err_buffer;

    // do something
    ret = -1;

    free(buffer);
err_fd_buffer:
    close(fd_out);
err_fd_out:
    close(fd_int);
err_fd_in:
    return ret;
}
```

Resource management in C++ (RAII)

```
int doSomething() // returns -1 or throws std::bad_alloc on errors
{
    ifstream inf(...);
    if (!inf.good()) return -1;

    ofstream outf(...);
    if (!outf.good()) return -1; // ifstream destructor releases resource

    Buffer buffer(...); // throws std::bad_alloc when a problem occurs

    // do something

    return 0;
}
```

So... how does this work?

Resource management in C++ (RAII)

```
int doSomething() // returns -1 or throws std::bad_alloc on errors
{
    ifstream inf(...);
    if (!inf.good()) return -1;

    ofstream outf(...);
    if (!outf.good()) return -1; // ifstream destructor releases resource

    Buffer buffer(...); // throws std::bad_alloc when a problem occurs

    // do something

    return 0;
}
```

So... how does this work?

- RAII – Resource acquisition is initialization (1984)
- CADRe – Constructor Acquires, Destructor Releases
- SBRM – Scope-based Resource Management
- Context managers in other languages

RAII

```
Buffer::Buffer(size_t size) { data = new char[size]; }
Buffer::~Buffer() { delete[] data; }


File::File(const char *name, int flags)
{
    fd = open(name, flags);
    if (fd == -1) throw IOError("Error openning file");
}
File::~File() { close(fd); }


Lock::Lock(Mutex *mutex) { mutex->lock(); }
Lock::~Lock() { mutex->unlock(); }
```

Resource management in C++ (RAII)

```
void doSomething() // throws something on errors
{
    File inf(...);
    File outf(...); // throws IOException when open fails?
    Buffer buffer(...);

    // do something
}
```

Resource management in C++ (RAII)

```
void doSomething() // throws something on errors
{
    File inf(...);
    File outf(...); // throws IOException when open fails?
    Buffer buffer(...);

    // do something
}
```

So, why are RAII + exceptions not used all the time...

- need to go all the way...
- ... so people consider C++ exceptions problematic
- interop with c / "bad" libraries (need to wrap everything etc)

- Note: RAII can be used also without exceptions (with a bit of checking for valid objects)

Exception Safety

- When an exception is thrown (and possibly catched), no resources must be leaked!
- Transactional behaviour: either an operation completes successfully, or no side effects appear at all.

Other languages – Python

```
def doSomething():
        try:
                inFile = open(inFileName)
                try:
                        outFile = open(outFileName)
                        process(inFile, outFile)
                finally:
                        close(outFile)
        finally:
                close(inFile)
```

Actually wrong ;-)

Other languages – Python

```
def doSomething():
        inFile = open(inFileName)
        try:
                outFile = open(outFileName)
                try:
                        process(inFile, outFile)
                finally:
                        close(outFile)
        finally:
                close(inFile)
```

Other languages – Python

```
def doSomething():
        inFile = open(inFileName)
        try:
                outFile = open(outFileName)
                try:
                        process(inFile, outFile)
                finally:
                        close(outFile)
        finally:
                close(inFile)
```

Python 2.5 - context managers

```
def doSomethin():
        with
                lock,
                open(inFileName) as inFile,
                open(outFileName) as outFile:
                        return process(inFileName, outFileName)
```

Other languages – Java

```
int doSomething()
{
        FileInputStream inStream = new FileInputStream(inFileName);
        try {
                FileOutputStream outStream = new FileOutputStream(outFileName);
                try {
                        process(inStream, outStream);
                } finally {
                        outStream.close();
                }
        } finally {
                inStream.close();
        }
}
```

Other languages – Java

```
int doSomething()
{
        FileInputStream inStream = new FileInputStream(inFileName);
        try {
                FileOutputStream outStream = new FileOutputStream(outFileName);
                try {
                        process(inStream, outStream);
                } finally {
                        outStream.close();
                }
        } finally {
                inStream.close();
        }
}
int doSomething()
{
        try (
                FileInputStream inStream = new FileInputStream(inFileName);
                FileOutputStream outStream = new FileOutputStream(outFileName);
        ) {
                process(inStream, outStream);
        }
}
```

Memory management

- Allocation
  - kernel: brk, mmap
  - C/C++: malloc, realloc, free, mmap, new, delete
- "Management"
  - pairing alloc/release, memory leaks
  - ownership, passing between functions etc. (size?)
  - dangling pointers
  - `std::unique_ptr`, `std::shared_ptr`
- Reference counting
  - RAII, immediate release, cycles?
  - implicit sharing, COW
- Garbage collection
  - when will it happen? price of detection?

Reference counting

- `std::shared_ptr`
- immediate "release", RAII similar to other resources
- cheap / fast (at least relatively: large object "trees" can take a while to release, which can be noticable in realtime apps)
- slight space (refcount/control block) / speed (inc/dec) overhead
- memory access - one or two dereferences
- synchronization (atomic refcount)
- cycles!
- breaking cycles: weak references
  - can become dangling
  - reference "zeroing"
    - keep track of both weak and strong references
    - when "strong" refcount becomes zero, data is released and weak references can't be used anymore to access data
    - `std::shared_ptr` + `std::weakt_ptr`

Garbage collection

- doesn’t combine nicely with management of other resources... (Java `finalize()`)
- "unpredictable", performance...

- reference counting + cycle detection (Python)
- tracing - find objects not reachable from "root" objects

Memory leaks

So... how to avoid memory leaks in C / bad C++ / ...?

- release reources before each **`return`**
- **`goto`** solutions
- with exceptions in C++, everything is a possible **`return`**!
- valgrind (memcheck)
- (and other tools...)

... and is Java really safe?

- "hidden" references: registering listeners, observers,...

Copy on write (COW)

- reference counting on steroids
- cheap pass by value even for very large objects
- don't make copies when not needed
- shared data - every data class is basically a shared (refcounted) pointer
- Copy on change
  - C++: const vs non-const methods
  - when refcount > 1
- might not be always possible/feasible (`std::string` in C++11?)
- unpredictable/unintuitive complexity/efficiency

  ```
  String s2 = s1; s2[0] = 'a';
  ```