The C abstract machine model

Systems Programming 2014-2015

Concepts

- Objects
  - Lifetimes
  - Storage
- Designators
  - Visibility, scope
  - Linkage
- Execution unit(s)
- System services
- Unspecified behavior
- Undefined behavior

Objects in C

Based on: ISO 9899:2011 section 6.2,
and "Semantics of C objects", Appendix F of "On the realizability of hardware microthreading"
http://hdl.handle.net/11245/2.109511

Storages & lifetime

- Static, thread, automatic, allocated
- Loosely answers "in which part of memory?"
- Storage determines lifetime
- The following properties are fixed during the lifetime of an object:
  - mutability: whether the object can be written to
  - addressability: whether its address is visible
  - its size in chars

Storages & lifetimes

Definition	Storage	Lifetime	Initialization
Defined with "static" in function or as global variable	static	Entire execution	Once, before startup
Defined with "_Thread_local"	thread	Thread's	At thread start
Local variable, no "static"	auto	Block or exp	none
...alloc(), sbrk(), mmap(), tss_create()	alloc	Until freed	none

Objects & designators

- Two ways to "make" objects and designators:
- **Declaration syntax** (eg. "int x"):
  - Definitions: new designator AND object
  - Declarations: new designator only
- **Expressions** (eg. "3+2", "foo()");
  - These always have "automatic" storage

Objects & designators

- There may be multiple designators to an object
  - After:
    int x; char *p = (char*)&x;
    void* foo() { return &x; }
  - Both "x", "*p" and "*foo()" designate the same object
- **Types are properties of designators**, not objects
- **Objects are just arrays of chars,** always (in C/C++)

Two kinds of designators

- **Primary designators**: at most one per object (perhaps zero)
  - Only for non-allocated objects
  - Derived from object definition
  - "Honest" about mutability - either **const** or not
- No primary designator implies object is mutable
- **Secondary designators**: everything else
  - Can lie about mutability

Linkage

- Linkage answers "When do two separate designators refer to the same object?" (only for objects with static storage)
  - **External linkage**: 1 object program-wide
    ```
    int x; // in global scope
    void foo() { static int z; }
    ```
  - **Internal linkage:** 1 object per translation unit
    ```
    static int x; // in global scope
    ```
  - **No linkage**: 1 object per definition/activation
- "**extern**": does not mean what you think it means
  - after a 1st declaration that specifies linkage: **same** linkage as 1st
  - in 1st declaration, or after declaration w/o linkage info: then **external**

Visibility

- Where is a designator visible? everything decided by scope (block structure { … })
- No surprise: same as Java, C++, C# etc
- Example:

```
int x = 1;
int foo() {
    int x = 2;
    if ( … ) { int x = 3 ; return x; }
}
```

3 different objects, 3 different designators,
Calling foo() returns 3

Visibility

- Another example:

```
int x = 1;
int foo() {
    int x = 2;
    if ( ... ) { extern int x; return x; }
}
```

2 different objects, 2 different designators,
Calling foo() returns 1

Mutability

- const int x = 3; // ok
- const int x; x = 3; // invalid
- int x; // x is mutable, "x" primary designator
  const int *p = &x; // p lies, "*p" secondary designator
  int *m = (int*) p; // m restores the truth
  *m = 3; // ok
- const int x = 3;
  int *p = (int*)&x; // p lies
  *p = 3; // UNDEFINED (x is really immutable)

Machine behavior

In a nutshell

- **Execution starts with main()**
  - or really **_start** on most POSIX systems
- When main() returns or exit() is called, all functions registered with **atexit(3)** are called in turn - Bypass with **_Exit** or terminating signal
- Execution may be arbitrarily preempted by **signal delivery**
  - control with signal(3) (ISO C)
    or sigaction(3) (POSIX, preferred)
- **1 initial thread**; new threads created with thrd_create()
  - Since ISO C 2011 only (POSIX has pthread_create)
  - Complex semantics wrt. shared data & signals - avoid if possible!

Undefined vs unspecified

- Three parties to decide what a program **means**: C standard, language implementation, run-time environment
- C standard defines / **specifies** most of it
- "Unspecified" means the C standard is silent, but implementation or environment decide something
- **"Undefined" means "HERE BE DRAGONS"**

Examples unspecified behavior

- The order function arguments are evaluated:
  foo(bar(), baz()); // is bar called first or baz?
- The value of uninitialized objects
  eg: int x; int z = x ^ x; // z always 0
- Read position in file after ungetc(3)
- Relative position or contiguity of objects allocated by malloc() etc

Examples
undefined behavior

- Accessing an object outside of its lifetime
- Accessing a position "outside" of an object
- Writing to an immutable object,
  eg: `const int x = 3; int *p = &x; *p = 4;`
- Exiting from a non-void function without a value, eg: `int foo() { }`

What really happens upon reaching undefined behavior?

Execution stops	rare
Signal is delivered	rare
Unrelated objects unpredictably modified	common
Execution continues from a predictable position	uncommon
Execution continues from an unpredictable position	common
If execution continues, further code changes meaning	common

Signal delivery

- **What happens when a signal occurs?**
- If signal is ignored (SIG_IGN), nothing happens
- Otherwise:
  - If in program or library code (incl libc): current **function pauses, signal handler runs**
    - Can't return for SIGSEGV, SIGILL or SIGFPE
  - If in system call (open, write, ...): it's complicated

Signal delivery

- Signal occurs in system call: two families
  - **"SysV" family,** including Linux:
    - system call completes, THEN signal is delivered
    - Program always sees complete system calls
  - **"BSD" family,** including MacOS X:
    - system call is interrupted!
    - Programs sees EINTR, must re-try

Signal delivery

- Example:

```
sz = read(0, buf, 10);
if (sz < 10) exit(1); // error
```

- This code may fail too often on BSD
- Rewrite as follows (or something similar):

```
do {
   sz = read(0, buf, 10);
} while(errno == EINTR);
```

System calls & errno

- write(2), open(2), etc are **wrappers** provided by a **POSIX-compliant C library**
- You can "roll your own" with inline assembly
- Different mechanism on each OS
- In POSIX, system calls return 2 things:
  - Their direct return value
  - An error code, which libc stores in "**errno**"
- "errno" looks&feels like a variable, but usually isn't

Variable argument lists

In a nutshell

- printf(const char *, ...)
- the "..." is called "ellipsis", means 0 or more arguments are accepted there
- Inside the function use the extra arguments with va_start, va_arg, va_end
  - Declared in stdarg.h
  - See manual pages for details!