Formal Methods for C

Seminar – Summer Semester 2014

Daniel Dietsch, Sergio Feo Arenis, Marius Greitschus, Bernd Westphal

Content

• Brief history
• Comments
• Declarations and Scopes
  • Variables
  • Expressions and Statements
  • Functions
  • Scopes
• Pointers
• Dynamic Storage & Storage Duration
• Storage Class Specifiers
• Strings and I/O
• Tools & Modules
• Formal Methods for C
• Common Errors
Tools & Modules

Hello, Again

```
#include <stdio.h>

int g(int x) { return x/2; }

int f() { return g(1); }

int main() {
    printf( "Hello World. \n" );
    return f();
}
```

- % gcc helloworld.c
- % ls
- a.out helloworld.c
- % ./a.out
- Hello World.
- % echo $?
Zoom In: Preprocessing, Compiling, Linking

```c
#include <stdio.h>

int g(int x) { return x/2; }

int f() { return g(1); }

int main() {
    printf("HelloWorld.\n");
    return f();
}
```

- `% gcc -E helloworld.c > helloworld.i`
- `% gcc -c -o helloworld.i`
- `% ld -o helloworld [...] helloworld.o [...]`
- `% ./helloworld`
- Hello World.
- `%`

Modules

Split into:
- .h (header): declarations
- .c: definitions, use headers to "import" declarations
Modules At Work

preprocess & compile:
- % gcc -c g.c f.c \ helloworld.c
- % ls *.o
- f.o g.o helloworld.o

link:
- % gcc g.o f.o helloworld.o

execute:
- % ./a.out
- Hello World.

fix and re-build:
- % gcc -c helloworld.c
- % gcc g.o f.o helloworld.o
- % ./a.out
- Hi!
#include <stdio.h>
#include "f.h"

int main() {
    printf("Hello World\n");
    return f();
}

• % gcc -E helloworld.c -o helloworld.i

Preprocessing

Preprocessing Directives (6.10)

1 #include <stdio.h>
2 #include "f.h"
3
define PI 3.1415
4
#define DEBUG
5 #ifdef DEBUG
6 #ifdef DEBUG
7 printf(stderr, "honk\n");
8 #endif
9 #endif
10
# if __GNUC__ >= 3
11 # define __pure__ 
12 # define __attribute__ ((pure))
13 # else
14 # define __pure__
15 # define __attribute__ ((pure))
16 #endif
17
extern int f() __pure;
**Linking**

- `provides: int g(int)`
  - `needs: /g.o`

- `provides: int f()`
  - `needs: g.o`

- `provides: int main()`
  - `needs: int f(int)`
  - `int printf(const char*,...)`
  - `helloworld.o`
  - `provides: int printf(const char*,...)`
  - `...`
  - `needs: libc.a`

- `a.out`

**Compiler**

- `gcc [OPTION]... infile...`

  - `-E` – preprocess only
  - `-c` – compile only, don’t link

  **Example:** `gcc -c main.c` — produces `main.o`

  - `-o outfile` – write output to `outfile`

    **Example:** `gcc -c -o x.o main.c` — produces `x.o`

  - `-g` – add debug information

  - `-W, -Wall, ...` – enable warnings

  - `-I dir` – add `dir` to include path for searching headers

  - `-L dir` – add `dir` to library path for searching libraries

  - `-D macro[=defn]` – define macro (to `defn`)

    **Example:** `gcc -DDEBUG -DMAGICNUMBER=27`

  - `-l library` link against `liblibrary.{a,so}`, order matters

    **Example:** `gcc a.o b.o main.o -lxy` — produces `a.out`

→ cf. man `gcc`
• **Command Line Debugger:**
  
  `gdb a.out [core]`

• **GUI Debugger:**
  
  `ddd a.out [core]`

  (works best with debugging information compiled in `gcc -g`)

• **Inspect Object Files:**
  
  `nm a.o`

• **Build Utility:**
  
  `make`

See battery controller exercise for an example.

---

**Core Dumps**

• **Recall:** Anatomy of a Linux Program in Memory

• **Core dump:** (basically) this memory written to a file.

```c
int main() {
    int *p;
    *p = 27;
    return 0;
}
```

```bash
% gcc -g core.c
% limit coresize
coresize 0 kbytes
% limit coredumpsize 1g
% ./a.out
Segmentation fault (core dumped)
% ls -lh core
-rw------- 1 user user 232K Feb 29 11:11 core
% gdb a.out core
GNU gdb (GDB) 7.4.1 - debian
[...]
Core was generated by './a.out'.
Program terminated with signal 11, Segmentation fault. (core c:3)
(gdb)
```

1. int main() {
2.     int *p;
3.     *p = 27;
4.     return 0;
5. }

91/125
Correctness

- Correctness is defined with respect to a specification.
- A program (function, ...) is correct (wrt. specification $\varphi$) if and only if it satisfies $\varphi$.
- Definition of "satisfies": in a minute.

Examples:
- $\varphi_1$: the return value is 10 divided by parameter (if parameter not 0)
- $\varphi_2$: the value of variable $x$ is "always" strictly greater than 3
- $\varphi_3$: the value of $i$ increases in each loop iteration
- ...

Common Patterns

- **State Invariants:**
  "at this program point, the value of $p$ must not be NULL"
  "at all program points, the value of $p$ must not be NULL"
  (cf. sequence points (Annex C))

- **Data Invariants:**
  "the value of $n$ must be the length of $s$"

- **(Function) Pre/Post Conditions:**
  Pre-Condition: the parameter must not be 0
  Post-Condition: the return value is 10 divided by the parameter

- **Loop Invariants:**
  "the value of $i$ is between 0 and array length minus 1"
Diagnostics (7.2)

```c
#include <assert.h>

void assert( /* scalar */ expression );
```

- “The assert macro puts diagnostic tests into programs; [...]”

When it is executed, if `expression` (which shall have a scalar type) is false (that is, compares equal to 0), the assert macro

- writes information about the particular call that failed [...] on the standard error stream in an implementation-defined format.

- It then calls the `abort` function.”

Pitfall:

- If macro ` NDEBUG ` is defined when including `<assert.h>`, `expression` is not evaluated (thus should be side-effect free).
**abort (7.20.4.1)**

```c
#include <stdlib.h>

void abort();
```

- “The abort function causes abnormal program termination to occur, unless […]
- [...] An implementation-defined form of the status unsuccessful termination is returned to the host environment by means of the function call `raise(SIGABRT)`.”

(→ Core Dumps)

---

**Common Patterns with assert**

- **State Invariants:**
  - “at this program point, the value of `p` must not be NULL”
  - “at all program points, the value of `p` must not be NULL”
  (cf. sequence points (Annex C))

- **Data Invariants:**
  - “the value of `n` must be the length of `s`”

- **(Function) Pre/Post Conditions:**
  - Pre-Condition: the parameter must not be 0
  - Post-Condition: the return value is 10 divided by the parameter

- **Loop Invariants:**
  - “the value of `i` is between 0 and array length minus 1”
State Invariants with `<assert.h>`

```c
void f() {
  int* p = (int*)malloc(sizeof(int));
  if (!p)
    return;
  assert(p); // assume p is valid from here
  // ...
}

void g() {
  Node* p = find('a');
  assert(p); // we inserted 'a' before
  // ...
}
```

Data Invariants with `<assert.h>`

```c
typedef struct {
  char* s;
  int n;
} str;

str* construct(char* s) {
  str* x = (str*)malloc(sizeof(str));
  // ...
  assert((x->s == NULL && x->n == -1)
         || (x->n == strlen(x->s)));
  return x;
}
```
Pre/Post Conditions with `<assert.h>`

```c
int f(int x) {
    assert(x != 0); // pre-condition

    int r = 10/x;
    assert(r == 10/x); // post-condition
    return r;
}
```

---

Loop Invariants with `<assert.h>`

```c
void f(int a[], int n) {
    int i = 0;

    // holds before the loop
    assert(0 <= i && i <= n);
    assert(i < 1 || a[i-1] == 0);

    while (i < n) {
        // holds before each iteration
        assert(0 <= i && i <= n);
        assert(i < 1 || a[i-1] == 0);

        a[i++] = 0;
    }

    // holds after exiting the loop
    assert(0 <= i && i <= n);
    assert(i < 1 || a[i-1] == 0);
    return;
}
```
Old Variables, Ghost Variables

```c
void xorSwap( unsigned int* a, unsigned int* b ) {
  #ifndef NDEBUG
    unsigned int *old_a = a, *old_b = b;
  #endif
  assert( a && b ); assert( a != b ); // pre-condition
  *a = *a + *b;
  *b = *a - *b;
  *a = *a - *b;
  assert( *a == *old_b && *b == *old_a ); // post-condition
  assert( a == old_a && b == old_b ); // dition
}
```

Outlook

- Some verification tools simply verify for each `assert` statement:
  When executed, expression is not false.

- Some verification tools support sophisticated requirements specification
  languages like ACSL with explicit support for
  - pre/post conditions
  - ghost variables, old values
  - data invariants
  - loop invariants
  - ...
"The program has been verified." tells us not very much.

One wants to know (and should state):

- Which specifications have been considered?
- Under which assumptions was the verification conducted?
  - Platform assumptions: finite words (size?), mathematical integers, . . .
  - Environment assumptions, input values, . . .
  Assumptions are often implicit, “in the tool”!

And what does verification mean after all?

- In some contexts: testing.
- In some contexts: review.
- In some contexts: model-checking procedure.
  ("We verified the program!" – “What did the tool say?” – “Verification failed.”)
- In some contexts: model-checking tool claims correctness.
Distinguish

Most **generic errors** boil down to:

- specified but **unwanted behaviour**,  
  e.g. under/overflows
- **initialisation issues**  
  e.g. automatic block scope objects
- **unspecified behaviour** (J.1)  
  e.g. order of evaluation in some cases
- **undefined behaviour** (J.2)
- **implementation defined behaviour** (J.3)
Conformance (4)

- “A program that is
  - correct in all other aspects,
  - operating on correct data,
  - containing unspecified behavior

shall be a correct program and act in accordance with 5.1.2.3. (Program Execution)

- A conforming program is one that is acceptable to a conforming implementation.

- Strictly conforming programs are intended to be maximally portable among conforming implementations.

- An implementation [of C, a compiler] shall be accompanied by a document that defines all implementation-defined and locale-specific characteristics and all extensions.

---

Over- and Underflows
Over- and Underflows, Casting

- Not specific to C...

```c
void f(short a, int b) {
    a = b; // typing ok, but...
}

short a; // provisioning, implicit cast
if (++a < 0) { /* no */ }
if (++i > MAX_INT) {
    /* no */ }

int e = 0;
void set_error() { e++; }
void clear_error() { e = 0; }
void g() { if (e) { /* ... */ } }
```

Initialisation (6.7.8)
**Initialisation (6.7.8)**

- "If an object that has automatic storage duration is not initialized explicitly, its value is indeterminate."

```c
void f() {
    int a;
    printf("%i\n", a); // surprise...
}
```

**Unspecified Behaviour (J.1)**
Unspecified Behaviour (J.1)

Each implementation (of a compiler) documents how the choice is made.

For example

- whether two string literals result in distinct arrays (6.4.5)
- the order in which the function designator, arguments, and subexpressions within the arguments are evaluated in a function call (6.5.2.2)
- the layout of storage for function parameters (6.9.1)
- the result of rounding when the value is out of range (7.12.9.5, ...)
- the order and contiguity of storage allocated by successive calls to malloc (7.20.3)
- etc. pp.

```c
char a[] = "hello", b[] = "hello"; // a == b?
i = 0; f(++i, ++i, ++i); // f(1, 2, 3)?

int g() { int a, b; } // &a > &b ?

int* p = malloc(sizeof(int));
int* q = malloc(sizeof(int)); // q > p?
```

Undefined Behaviour (J.2)
**Undefined Behaviour (3.4.3)**

“Behaviour, upon use of a non-portable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements.”

**Possible undefined behaviour ranges from**
- ignoring the situation completely with **unpredictable results**,  
- to behaving during **translation or program execution** in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message),
- to terminating a **translation or execution** (with the issuance of a diagnostic message).”

“An example of undefined behaviour is the behaviour on **integer overflow**.”

**Undefined Behaviour (J.2)**

**More examples:**
- an identifier [...] contains an invalid multibyte character (5.2.1.2)  
- an object is referred to outside of its lifetime (6.2.4)  
- the value of a pointer to an object whole lifetime has ended is used (6.2.4)  
- conversion to or from an integer type produces a value outside the range that can be represented (6.3.1.4)  
- conversion between two pointer types produces a result that is incorrectly aligned (6.3.2.3)  
- the program attempts to modify a string literal (6.4.5)  
- an exceptional condition occurs during the evaluation of an expression (6.5)  
- the value of the second operand of the / or % operator is zero (6.5.5)  
- pointers that do not point into, or just beyond, the same array object are subtracted (6.5.6)  
- An array subscript is out of range [...] (6.5.6)  
- the program removes the definition of a macro whose name begins with an underscore and either an uppercase letter or another underscore (7.1.3)  
- etc. pp.
**Null-Pointer**

```
int main() {
    int *p;
    *p = 27;
    return 0;
}
```

- “An integer constant expression with the value 0, or such an expression cast to type `void*`, is called a **null pointer constant**. [...]”

- “The macro **NULL** is defined in `<stddef.h>` (and other headers) as a null pointer constant; see 7.17.”

- “Among the invalid values for dereferencing a pointer by the unary `*` operator are a null pointer, [...]” (6.5.3.2)

**Segmentation Violation**

```
int main() {
    int *p = (int*)0x12345678;
    *p = 27;
   (*(int*)((void*)p) + 1) = 13;
    return 0;
}
```

- Modern operating systems provide **memory protection**.
- Accessing memory which the process is not allowed to access is observed by the operating system.
- Typically an instance of “accessing an object outside its lifetime”.
- **But**: other way round does not hold, accessing an object outside its lifetime does not imply a segmentation violation.

- Some platforms (e.g. SPARC): unaligned memory access, i.e. outside word boundaries, not supported by hardware (“bus error”). Operating system notifies process, default handler: terminate, dump core.
Implementation-Defined Behaviour (J.3)

“A conforming implementation is required to document its choice of behavior in each of the areas listed in this subclause. The following are implementation-defined:”

- J.3.2 Environment, e.g.
  The set of signals, their semantics, and their default handling (7.14).
- J.3.3 Identifiers, e.g.
  The number of significant initial characters in an identifier (5.2.4.1, 6.4.2).
- J.3.4 Characters, e.g.
  The number of bits in a byte (3.6).
- J.3.5 Integers, e.g.
  Any extended integer types that exist in the implementation (6.2.5).
- J.3.6 Floating Point, e.g.
  The accuracy of the floating-point operations […] (5.2.4.2.2).
- J.3.7 Arrays and Pointers, e.g.
  The result of converting a pointer to an integer or vice versa (6.3.2.3).
- etc. pp.
**Locale and Common Extensions (J.4, J.5)**

- J.4 Locale-specific behaviour
- J.5 Common extensions

“The following extensions are widely used in many systems, but are not portable to all implementations.”

**References**