

Formal Methods for C

Seminar – Summer Semester 2014

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

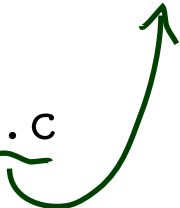
Content

- 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

```
1  #include <stdio.h>
2
3  int g( int x ) { return x/2; }
4
5  int f() { return g(1); }
6
7  int main() {
8      printf( "Hello World.\n" );
9      return f();
10 }
```

- % gcc helloworld.c
 - % ls
 - a.out helloworld.c
 - % ./a.out
 - Hello World.
 - % echo \$?
- 

Zoom In: Preprocessing, Compiling, Linking

```
1  #include <stdio.h>
2
3  int g( int x ) { return x/2; }
4
5  int f() { return g(1); }
6
7  int main() {
8      printf( "Hello World.\n" );
9      return f();
10 }
```

preprocess

compile

link

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

Modules

```
1 #include <stdio.h>
2
3 int g( int x ) {
4     return x/2;
5 }
6
7 int f() {
8     return g(1);
9 }
10
11 int main() {
12     printf( "Hello World.\n" );
13     return f();
14 }
```

g.h

```
1 #ifndef G_H
2 #define G_H
3
4 extern int
5     g( int x );
6 #endif
```

f.h

```
1 #ifndef F_H
2 #define F_H
3
4 extern int
5     f();
6 #endif
```

g.c

```
1 #include "g.h"
2
3 int g( int x ) {
4     return x/2;
5 }
```

f.c

```
1 #include "g.h"
2 #include "f.h"
3
4 int f() {
5     return g(1);
6 }
```

Split into:

- .h (header): declarations
- .c: definitions, use headers to “import” declarations

helloworld.c

```
1 #include <stdio.h>
2 #include "f.h"
3
4 int main() {
5     printf( "Hello World.\n" );
6     return f();
7 }
```

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.

g.h

```
1 #ifndef G_H
2 #define G_H
3
4 extern int
5     g( int x );
6 #endif
```

f.h

```
1 #ifndef F_H
2 #define F_H
3
4 extern int
5     f();
6 #endif
```

g.c

```
1 #include "g.h"
2
3 int g( int x ) {
4     return x/2;
5 }
```

f.c

```
1 #include "g.h"
2 #include "f.h"
3
4 int f() {
5     return g(1);
6 }
```

helloworld.c

```
1 #include <stdio.h>
2 #include "f.h"
3
4 int main() {
5     printf( "Hello World.\n" );
6     return f();
7 }
```

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!

g.h

```
1 #ifndef G_H
2 #define G_H
3
4 extern int
5     g( int x );
6 #endif
```

f.h

```
1 #ifndef F_H
2 #define F_H
3
4 extern int
5     f();
6 #endif
```

g.c

```
1 #include "g.h"
2
3 int g( int x ) {
4     return x/2;
5 }
```

f.c

```
1 #include "g.h"
2 #include "f.h"
3
4 int f() {
5     return g(1);
6 }
```

helloworld.c

```
1 #include <stdio.h>
2 #include "f.h"
3
4 int main() {
5     printf( "Hi!\n" );
6     return f();
7 }
```


Preprocessing

helloworld.i

helloworld.c

```
1 #include <stdio.h>
2 #include "f.h"
3
4 int main() {
5     printf( "Hello World.\n" );
6     return f();
7 }
```

PREPROCESS

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

```
1 # 1 " helloworld.c"
2 # 1 "<command-line>"
3 # 1 " helloworld.c"
4 # 1 "/usr/include/stdio.h" 1 3 4
5 # 28 "/usr/include/stdio.h" 3 4
6 # 1 "/usr/include/features.h" 1 3 4
7 # 323 "/usr/include/features.h" 3 4
8 # 1 "/usr/include/x86_64-linux-gnu/bits/predefs.h" 1 3 4
9 # 324 "/usr/include/features.h" 2 3 4
10 # 356 "/usr/include/features.h" 3 4
11 # 1 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 1 3 4
12 # 359 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 3 4
13 # 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
14 # 360 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
15 # 357 "/usr/include/features.h" 2 3 4
16 # 388 "/usr/include/features.h" 3 4
17 # 1 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 1 3 4
18
19
20
21 # 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
22 # 5 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 2 3 4
23
24 [...]
25
26 extern int ftrylockfile (FILE *__stream) __attribute__ ((__nothrow__));
27
28
29 extern void funlockfile (FILE *__stream) __attribute__ ((__nothrow__));
30 # 936 "/usr/include/stdio.h" 3 4
31
32 # 2 " helloworld.c" 2
33 # 1 " f.h" 1
34
35
36
37 extern int
38     f();
39 # 3 " helloworld.c" 2
40
41 int main() {
42     printf( "Hello World.\n" );
43     return f();
44 }
```

Preprocessing Directives (6.10)

```
1 #include <stdio.h>
2 #include "battery.h"
3
4 #define PI 3.1415
5
6 #define DEBUG
7 #ifdef DEBUG
8     fprintf( stderr , "honk\n" );
9 #endif
10
11 #if __GNUC__ _ >= 3
12 # define __pure          __attribute__ _ ((pure))
13 #else
14 # define __pure          /* no pure */
15 #endif
16
17 extern int f() __pure;
```

macro name

M(p)

Linking

provides: int g(int)

needs: ./. g.o } *object code*

provides: int f()

needs: int g(int)

f.o

provides int main()

needs:
int f(int)
int printf(const char*,...)

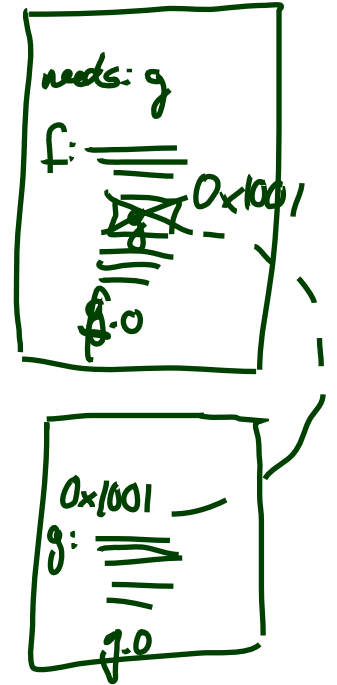
helloworld.o

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`

static ↑
dynamic ↑

→ cf. `man gcc`

gdb(1), ddd(1), nm(1), make(1)

- **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
```

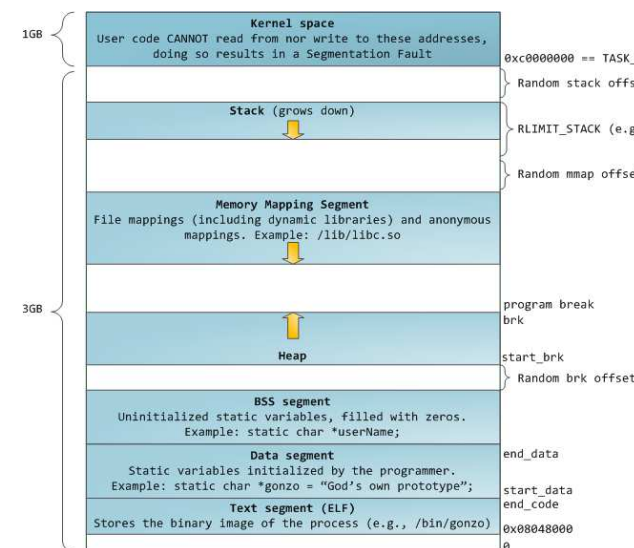
- **Build Utility:**

```
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.



©Gustavo Duarte 2009, used by permission.
<http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory>

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

```
1 % gcc -g core.c  
2 % limit coredumpsize  
3 coredumpsize 0 kbytes  
4 % limit coredumpsize 1g  
5 % ./a.out  
6 Segmentation fault (core dumped)  
7 % ls -lh core  
8 -rw----- 1 user user 232K Feb 29 11:11 core  
9 % gdb a.out core  
10 GNU gdb (GDB) 7.4.1-debian  
11 [...]  
12 Core was generated by './a.out'.  
13 Program terminated with signal 11, Segmentation fault.  
14 #0 0x00000000004004b4 in main() at core.c:3  
15 3*****p==27;  
16 (gdb) p p  
17 $1 = (int *) 0x0  
18 (gdb) q
```

Formal Methods for C

Correctness and Requirements

Correctness

- Correctness is defined **with respect to** a specification.
- A program (function, ...) is **correct** (wrt. specification φ)
if and only if it satisfies φ .
- Definition of “satisfies”: **in a minute**.

Examples:

- φ_1 : the return value is 10 divided by parameter (if parameter not 0)
- φ_2 : the value of variable x is “always” strictly greater than 3
- φ_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”

*Poor Man's Requirements Specification
aka. How to Formalize Requirements in C?*

Diagnostics (7.2)

```
1 #include <assert.h>
2 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)

```
1 #include <stdlib.h>
2
3 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>`

```
1 void f() {
2     int* p = (int*)malloc(sizeof(int));
3
4     if (!p)
5         return;
6
7     assert(p); // assume p is valid from here
8     // ...
9 }
10
11 void g() {
12     Node* p = find('a');
13
14     assert(p); // we inserted 'a' before
15     // ...
16 }
```

Data Invariants with `<assert.h>`

```
1  typedef struct {
2      char* s;
3      int n;
4  } str;
5
6  str* construct( char* s ) {
7      str* x = (str*)malloc( sizeof(str) );
8      // ...
9      assert( (x->s == NULL && x->n == -1)
10             || (x->n == strlen( x->s ) ) );
11 }
```


Pre/Post Conditions with `<assert.h>`

```
1  int f( int x ) {
2      assert( x != 0 ); // pre-condition
3
4      int r = 10/x;
5
6      assert( r == 10/x ); // post-condition
7
8      return r;
9  }
```

Loop Invariants with `<assert.h>`

```
1 void f( int a[], int n ) {
2     int i = 0;
3
4     // holds before the loop
5     assert( 0 <= i && i <= n );
6     assert( i < 1 || a[i-1] == 0 );
7
8     while ( i < n ) {
9         // holds before each iteration
10        assert( 0 <= i && i <= n );
11        assert( i < 1 || a[i-1] == 0 );
12
13        a[i++] = 0;
14    }
15    // holds after exiting the loop
16    assert( 0 <= i && i <= n );
17    assert( i < 1 || a[i-1] == 0 );
18
19    return;
20 }
```

Old Variables, Ghost Variables

```
1 void xorSwap( unsigned int* a, unsigned int* b ) {
2 #ifndef NDEBUG
3     unsigned int *old_a = a, *old_b = b;
4 #endif
5     assert( a && b ); assert( a != b ); // pre-condition
6
7     *a = *a + *b;
8     *b = *a - *b;
9     *a = *a - *b;
10
11     assert( *a == *old_b && *b == *old_a ); // post-con-
12     assert( a == old_a && b == old_b ); // dition
13 }
```

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
 - ...

Dependable Verification (Jackson)

Dependability

- “**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**.

Common Errors

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)

↑
the compiler

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. (*← compiler*)
- 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...

```
1 void f( short a, int b ) {
2     a = b; // typing ok, but...
3 }
4
5 short a; // provisioning, implicit cast
6 if (++a < 0) { /* no */ }
7
8 if (++i > MAX_INT) {
9     /* no */ }
10
11
12 int e = 0;
13
14 void set_error() { e++; }
15 void clear_error() { e = 0; }
16
17 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.”

```
1 void f() {  
2     int a;  
3  
4     printf( "%i\n" , a ); // surprise ...  
5 }
```

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.

```
1  char a[] = "hello", b[] = "hello"; // a == b?
2
3  i = 0; f( ++i, ++i, ++i ); // f(1,2,3)?
4
5  int g() { int a, b; } // &a > &b ?
6
7  int* p = malloc( sizeof(int) );
8  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 whose 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

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

- “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

```
1  int main() {
2      int *p = (int*)0x12345678;
3      *p = 27;
4
5      *(int*)((void*)p) + 1) = 13;
6      return 0;
7  }
```

- 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)

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

[ISO, 1999] ISO (1999). Programming languages – C. Technical Report ISO/IEC 9899:1999, ISO. Second edition, 1999-12-01.