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

1. `#include <stdio.h>`
2. `int g(int x)`
   ```c
   return x/2;
   ```
3. `int f()`
   ```c
   return g(1);
   ```
4. `int main()`
   ```c
   printf("Hello World.\n");
   return f();
   ```
Example needs `-l` library `lib`

```
#include <stdio.h>

int printf(const char*,...);
```

Example:

```
void main()
{
    printf("Hello World.");
}
```

Linking:

```
% ./a.out
Hello World.
```

```
% ls *.o
9 g.cf.c
1 helloworld.i
1 g.h
2 helloworld.o
1 f.h
2 g.o
```

```
% gcc -c g.c f.c
```

```
% gcc g.o f.o helloworld.o
```

```
% ls *.o
9 helloworld.i
1 g.cf.c
1 g.h
2 helloworld.o
```
• **CommandLineDebugger**:
  `gdb a.out [core]`

• **GUIDebugger**:
  `ddd a.out [core]` (works best with debugging information compiled in `gcc -g`)

• **InspectObjectFiles**:
  `nm a.o`

• **BuildUtility**:
  `make`

See battery controller exercise for an example.

---

**CoreDumps**

Recall: Anatomy of a Linux Program in Memory

Coredump: (basically) this memory written to a file.

```c
1 int main()
2 {
3    int *p;
4    *p = 27;
5    return 0;
6 }
```

1. `%gcc -gcore.c`
2. `%limitcoredumpsize 3 coredumpsize 0kbytes`
3. `%limitcoredumpsize 1g`
4. `%./a.out`
5. Segmentation fault (core dumped)
6. `%ls -lhcore`

<table>
<thead>
<tr>
<th>$1</th>
<th>(int *) 0x0</th>
</tr>
</thead>
</table>

14. `GNU gdb (GDB) 7.4.1 - debian`
15. `[...]
16. Core was generated by `./a.out`.
17. Program terminated with signal 11, Segmentation fault.
18. `#0 0x00000000004004b4 in main ( ) at core.c:3`

---

**Formal Methods for C**

Correctness and Requirements

**Correctness is defined with respect to a specification.**

A program (function, ...) is correct (wrt. specification $\phi$) if and only if it satisfies $\phi$.

**Definition of "satisfies":**

- **Examples:**
  1. $\phi_1$: the return value is 10 divided by parameter (if parameter not 0)
  2. $\phi_2$: the value of variable $x$ is "always" strictly greater than 3
  3. $\phi_3$: 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)

#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, compare 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).

• 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 (7.2)

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"

Data Invariants with <assert.h>

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) ) );
}
Pre/PostConditions with

```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

```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);
    //post-condition
    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);
}
```

Outlook

• Some verification tools simply verify each `assert` statement: When executed, the expression is not false.
• Some verification tools support sophisticated requirements specification languages like ACSL with explicit support for:
  - pre/postconditions
  - ghost variables, old values
  - data invariants
  - loop invariants
  ...

Dependable Verification (Jackson)

• "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 tools say?" – "Verification failed.")
      - Some contexts: model-checking tool claims correctness...
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:
• "A program that is correct in all other aspects, operating on correct data, containing unspecified behaviour 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, Casting

```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 setError() {
    e++;
} void clearError() {
    e = 0;
} void g() {
    if (e) {
        /* ... */
    }
}  ```

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

- 2014-04 – pitfalls –

114/125

• "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.
}
```

115/125

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)

116/125

**Unspecified Behaviour (J.1)**

117/125

"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 ignored, 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."

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.
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)