Low-Level Programming

M. Anton Ertl
TU Wien

High-level vs. low-level programming language

- Prevents some bugs
  ~~arbitrary overwrites~~
  ~~dangling references~~
- Prevents some exploits
  ~~Buffer overflow~~
- More convenient
  automatic memory reclamation
  arbitrary integers
- Limitations

- Access to bits and bytes
  across abstraction boundaries
- Systems programming
  run-time systems of HLLs
  Libraries
- Efficiency
  possible, not guaranteed
  knowledge useful for HLLs
- Less convenient
  fixed buffers or
  manual memory reclamation

Low-level programming languages

- Assembly language: machine-dependent, full power
- Forth: interactive, no type checking
- C
- C++
- Rust: elaborate type system
  tries to combine safety and low-level featurs

Debugging

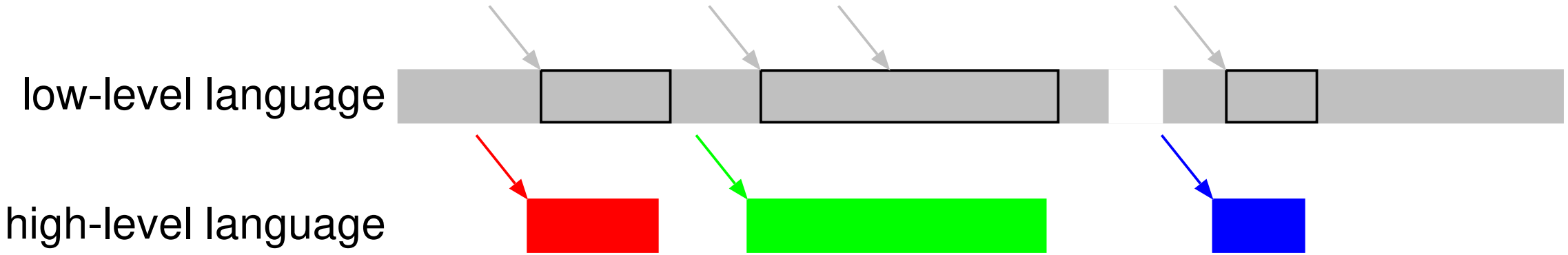
- interactive testing
  use small words
- Stepping Debugger
  `dbg word`
  Only works with `gforth-itc`
- Tracer (`printf` debugging)
  `~~`
- Backtrace
- insert `assert( ... )`
- insert conditional tracers

IDE features

- `locate word`
  `edit word`
- `help word`
- `where word`
  `nw bw ( u ) ww`

- after a Backtrace
  `nt bt ( u ) tt`
- Decompiler
  `see word`
  `simple-see word`
  `see-code word`

Memory



- Address space (per-process)
- Accessible (mapped) memory (`pmap pid`)
  Readable **W**ritable e**X**ecutable

Example: linked list

```
public class ListQueueSimple {
  private ListNode head;

  public void add(String v) {
    if (head == null) {
      head = new ListNode(v, null);
    } else {
      ListNode last = head;
      while (last.next() != null) {
        last = last.next();
      }
      last.setNext(new ListNode(v, null));
    }
  }

  public String poll() {
    if (head != null) {
      String result = head.value();
      head = head.next();
      return result;
    }
    return null;
  }
}
```

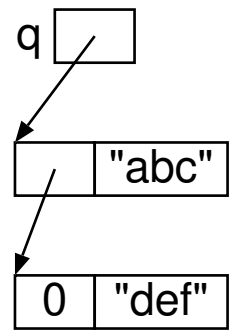
Example: linked list (cont.)

```
public class ListNode {
  private String value;
  private ListNode next;

  public ListNode(String v, ListNode n) {
    value = v;
    next = n;
  }

  public String value() {
    return value;
  }

  public ListNode next() {
    return next;
  }

  public void setNext(ListNode n) {
    next = n;
  }
}
```

Example: linked list (cont.)

```
0
  field:  list-next
  2field: list-value
constant listnode

: list-add {: c-addr u list-addr -- :}
  list-addr begin {: list-addr1 :}
    list-addr1 @ dup while
      list-next repeat
  drop
  listnode allocate throw {: node :}
  0 node list-next !
  c-addr u node list-value 2!
  node list-addr1 ! ;
```

```
: list-poll {: list-addr -- c-addr u :}
  list-addr @ {: node :}
  node if
    node list-next @ list-addr !
    node list-value 2@
    node free throw
  else
    0 0
  then ;

\ Example
variable q 0 q !
"abc" q list-add
"def" q list-add
q list-poll type
```



Storage management

- Static allocation
  `create allot`
  Small systems ($<$ 64KB)
- Dynamic allocation with explicit deallocation
  `allocate free` regions/arenas
  Keep track of allocations: Memory leaks and double `free`
  Medium systems
- Dynamic allocation with automatic deallocation
  `allocate` and garbage collection
  Large systems ($>$ 16MB)

Storage management and APIs

`read-line ( c-addr u1 fid -- u2 flag wior )`

Reads a line from fid into the buffer at c-addr u1. Gforth supports all three common line terminators: LF, CR and CRLF. A non-zero wior indicates an error. A false flag indicates that 'read-line' has been invoked at the end of the file. u2 indicates the line length (without terminator): u2<u1 indicates that the line is u2 chars long; u2=u1 indicates that the line is at least u1 chars long, the u1 chars of the buffer have been filled with chars from the line, and the next slice of the line with be read with the next 'read-line'. If the line is u1 chars long, the first 'read-line' returns u2=u1 and the next read-line returns u2=0.

`slurp-fid ( fid -- c-addr u )`

c-addr u is the content of the file fid

Types

Who knows the type of data?

		run-time system	
		no	yes
compiler	no	Forth	Python, Lisp
	yes	C, Pascal	C++, Java, Rust

- Type knowledge of the programmer (e.g., sorted array)
- uniformity (Lisp) vs. specialization (Java)

Types: Checking

```
'a' 5 *
```

- How do you find type errors in Forth? Testing!
- With a little experience type errors are easy to find.
  experience in interpreting program output
  experience in writing test cases
  experience in writing programs for easy testability
- More intensive testing $\Rightarrow$ you also find other errors

Types: Checking

As programmers learned C with Classes or C++, they lost the ability to quickly find the "silly errors" that creep into C programs through the lack of checking. Further, they failed to take the precautions against such silly errors that good C programmers take as a matter of course. After all, "such errors don't happen in C with Classes." Thus, as the frequency of run-time errors caused by uncaught argument type errors goes down, their seriousness and the time needed to find them goes up.
Bjarne Stroustroup

Types: Overloading

```
int n;       n*3          n @ 3 *
float r;     r*3.0        r f@ 3.0e f*
int n;       n<3          n @ 3 <
unsigned u;  u<3          u @ 3 u<
```

Types: Advantages and disadvantages of Forth

+ More uniformity: Better reusability
+ Extensions as libraries that would need type system extension in other languages (OOP, meta-programming)
+ Less complexity, e.g., for containers
– No type knowledge for garbage collection, marshalling etc.
– For changes affecting many lines static type checking would be useful
  Poor man's checking: Use new names for changed interfaces
  New names allow gradual changes

Object-oriented extension: features

- Dynamic Dispatch (virtual functions/methods)
- Instance Variables
- Single inheritance
- Static binding (C++: `A::m`)
- Basic class `object`
- `new`

Object-oriented extension: Usage (1)

```
object class
  cell var text
  cell var len
  cell var x
  cell var y
  method init
  method draw
end-class button

:noname ( o -- ) >r
 r@ x @ r@ y @ at-xy  r@ text @ r> len @ type ;
 button defines draw
:noname ( addr u o -- ) >r
 0 r@ x ! 0 r@ y ! r@ len ! r> text ! ;
 button defines init
```

Object-oriented extension: Usage (2)

```
button class
end-class bold-button

: bold   27 emit ." [1m" ;
: normal 27 emit ." [0m" ;

:noname bold [ button :: draw ] normal ; bold-button defines draw

button new Constant foo
s" thin foo" foo init
page
foo draw
bold-button new Constant bar
s" fat bar" bar init
1 bar y !
bar draw
```

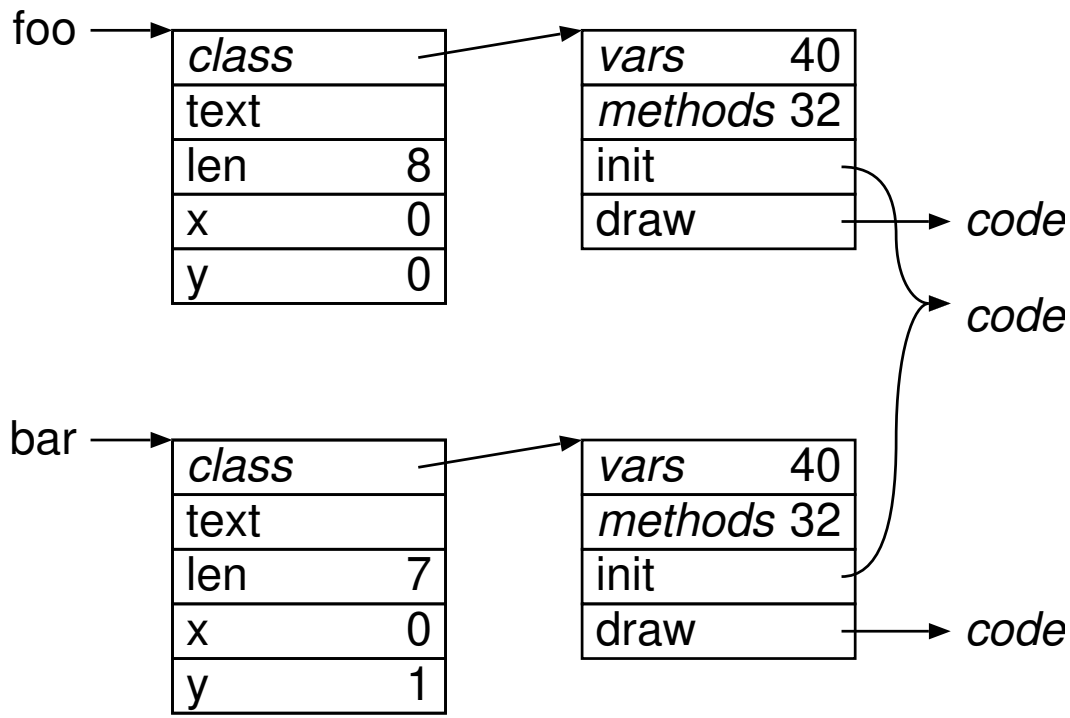
Object-oriented extension: source code

```
: method ( m v -- m' v ) Create  over , swap cell+ swap
  DOES> ( ... o -- ... ) @ over @ + @ execute ;
: var ( m v size -- m v' ) Create  over , +
  DOES> ( o -- addr ) @ + ;
: class ( class -- class methods vars ) dup 2@ ;
: end-class  ( class methods vars -- )
  Create  here >r , dup , 2 cells ?DO ['] noop , 1 cells +LOOP
  cell+ dup cell+ r> rot @ 2 cells /string move ;
: defines ( xt class -- ) ' >body @ + ! ;
: new ( class -- o )  here over @ allot swap over ! ;
: :: ( class "name" -- ) ' >body @ + @ compile, ;
Create object  1 cells , 2 cells ,
```

Explanation: `https://bernd-paysan.de/mini-oof.html`

Object-oriented extension: explanation

```
object class
  cell var text
  cell var len
  cell var x
  cell var y
  method init
  method draw
end-class button
button new Constant foo

button class
end-class bold-button
bold-button new Constant bar
```



Stateless control structures

```
... IF ... THEN
... IF ... ELSE ... THEN
BEGIN ... WHILE ... REPEAT
BEGIN ... UNTIL
CASE ... OF ... ENDOF ... OF ... ENDOF ... ENDCASE
?DO ... LOOP
```

Control structures: foundation

	forwards	backwards
Unconditional branch	`AHEAD ( -- orig )`	`AGAIN ( dest -- )`
Conditional branch	`IF ( -- orig )`	`UNTIL ( dest -- )`
branch target	`THEN ( orig -- )`	`BEGIN ( -- dest )`

```
: foo
  BEGIN ( C: dest )
    ... IF ( C: dest orig )
      ... THEN ( C: dest )
    ... UNTIL ( C: ) ;
```

Control Structures: ground floor

```
: ELSE ( compilation: orig1 -- orig2 ; run-time: -- ) \ core
    POSTPONE ahead
    1 cs-roll
    POSTPONE then ; immediate restrict

: WHILE ( compilation: dest -- orig dest ; run-time: f -- ) \ core
    POSTPONE if
    1 cs-roll ; immediate restrict

: REPEAT ( compilation: orig dest -- ; run-time: -- ) \ core
    POSTPONE again
    POSTPONE then ; immediate restrict
```

Control structures: Usage

```
: foo
  ... if ( C: orig1 )
    ...
  else ( C: orig2 )
    ... begin ( C: orig2 dest )
      ... while ( C: orig2 orig3 dest )
      ... repeat ( C: orig2 )
  then ;
```

Macros

```
: endif POSTPONE then ; immediate
: foo ... if ... endif ... ;

: (map) ( a n -- a a')   cells over + swap ;
: map<   postpone (map)  postpone ?do postpone i postpone @ ; immediate
: >map   1 cells postpone literal  postpone +loop ; immediate
: step  0  array 1000 map< + >map drop ;
```

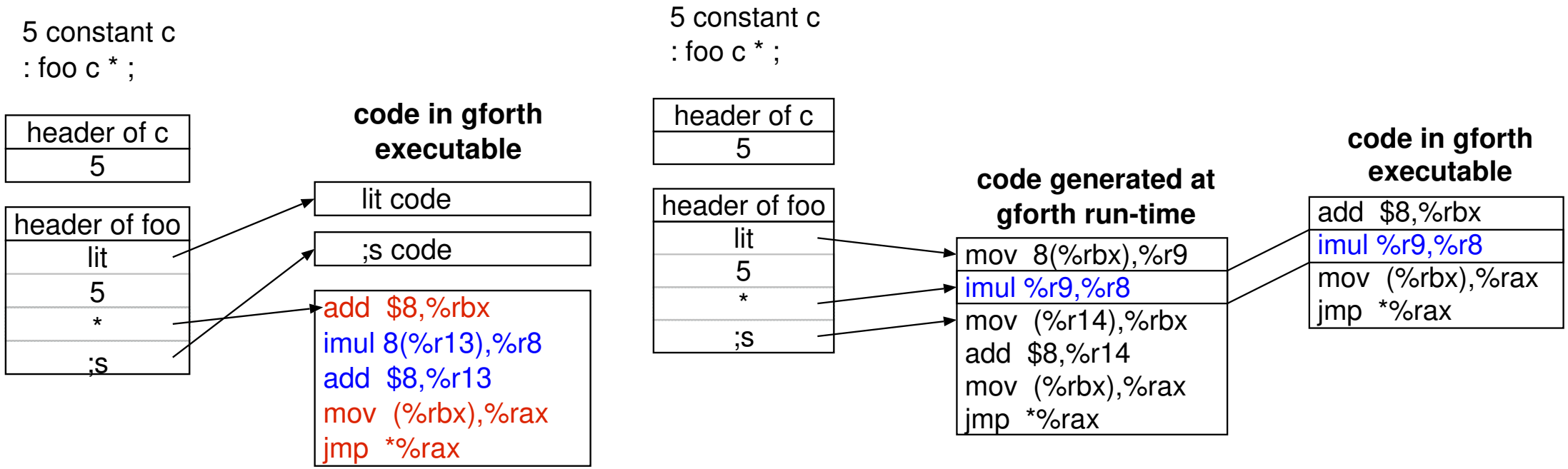
Code generation: LITERAL COMPILE,

```
: foo [ 5 cells ] literal ;
: ]cells ] cells POSTPONE literal ;
: foo [ 5 ]cells ;

: twice ( xt -- )
  dup compile, compile, ;
: 2+ [ ' 1+ twice ] ;
: 4* [ ' 2* twice ] ;
```

Code generation: Gray

```
: compile-test \ set -- )
 postpone literal
 test-vector @ compile, ;

: generate-alternative1 \ -- )
 operand1 get-first compile-test
 postpone if
 operand1 generate
 postpone else
 operand2 generate
 postpone endif ;
```

Implementation



- `see word`
- `simple-see word`
- `see-code word`