Plan

- **(15 min.)** Topic lottery: prepare lottery ticket with
  - name
  - first preference topic
  - second preference topic

(Common)

- **Introduction to C (1)**

- **(15 min.)** The VM (Marius)
Goals

**Educational Objectives:** Capabilities for following tasks/questions.

- Rough overview over concepts of the C programming language.
  - IOW: know, what to look for in books/manuals.
    (We try to stick with the names from of ISO/IEC 9899:1999.)
  - IOW: don’t be badly surprised from the examples.

- The concept of pointers.

- Basic work-flow, tool-usage (headers, sources, compiler, linker).

- Orthogonal: Rough overview over common sources of errors.

- Formal methods and C.
  
  ![Graph](image.png)

- **Not:** reference manual, each and every feature.

- **Not:** programming course.
#include <stdio.h>

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

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

int main() {
    printf( "Hello World.\n" );
    return f();
}
1972: Created by Dennis Ritchie († 2011) for Unix system programming.
1978: Brian W. Kernighan & Dennis Ritchie:
“The C Programming Language” – “K&R C”.
1989: ANSI X3.159-1989 – C89, C90 (still most widely used (?))

- Compilers for virtually every platform (CPU + operating system) available.
- Virtually every CPU vendor offers an own C compiler, in particular in the embedded domain (MSP 430, ARM, intel...).
- Still No. 1 programming language for embedded systems software, hardware drivers, performance critical applications, ...

- Preferred by many embedded programmers for “lack of surprises”:
(without optimisation) direct correspondence between C code and assembler.
- Resources widely controllable by programmer, downside: programmer needs to “know what one’s doing”

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
Comments

Comments (6.5.9)

- one line comment, until end of line: // ...
- generic comment, no nesting: /* ... */
- corner cases:

```c
"a/b"    // four-character string literal
#include "//e"    // undefined behavior
// /*    // comment, not syntax error
f = g/**/h;    // equivalent to f = g / h;
//\    // part of a two-line comment
i();    // part of a two-line comment
//
/ j();    // part of a two-line comment
#define glue(x,y) x##y
glue(/,/) k();    // syntax error, not comment
//*/*/ l();    // equivalent to l();
m = n/**/o + p;    // equivalent to m = n + p;
```
Declarations and Scopes

Variables
"[...] char is large enough to store any member of the basic exec. character set."

“A ‘plain’ int object has the natural size suggested by the architecture of the execution environment (large enough [for values] INT_MIN to INT_MAX).”

```
char c = 'a', d = 93;
int x = 027;  // octal!
long int y = 3L, z;
short int w = 0xBEEF;
```

char, short int, int, long int, long long int also as unsigned.

```
unsigned int x = 27U;
```

```
float f = 1;
double g = 314159e-5;
long double h;
```

-- 2014-04-01

---

**Bool (6.2.5)**

- Only introduced in C99.
- “An object declared as type _Bool
  is large enough to store the values 0 and 1.”
- `<stdbool.h>` (→ later) defines bool, true, false as macros (→ later).
- Before C99, and still very common:
  - Use scalar type (including pointers).
  - 0: false
  - everything else: true
  - values of boolean expressions: 0, 1

```
int y = 27;
int x = 13 && (y > 0);  // value of x becomes 1
```
Derived Types (6.2.5), (6.7)

- array types:
  ```c
  int a[10];
  char b[2][3];
  ```

- structured types:
  ```c
typedef struct {
  int n;
  double d;
} S; // declaration of type 'S'
S x; // declaration of variable 'x' of type 'S'
struct {
  int a; double b;
} y; // declaration of...
typedef struct { S[3] c; double d; } T;
```

**Note:** x and y are of different type!

- union types: not here
- function types, pointer types: later

Declaration Syntax

- ...takes a little bit getting used to:
  ```c
  int c, *p, a[3], *q[2], (*f)(int);
  ```

- is a shorthand notation for:
  ```c
  int c; // integer
  int *p; // pointer to integer
  int a[3]; // array of 3 integers
  int *q[2]; // array of 2 pointers to integer
  int (*f)(int); // function pointer...
  ```

- And what’s declared here, what is its type?
  ```c
  int (*g)( int (*)(int*[2])); // ...?
  ```
Expressions (6.5)

- An expression is a sequence of operators and operands that specifies
  computation of a value, or that designates an object or a function, or that
  generates side effects, or that performs a combination thereof.

- basically like Java:
  - postfix operators:
  - unary operators:
  - cast operators:
  - multiplicative, additive
  - relational, equality:
  - logical operators:
  - conditional operator:
  - assignment operator (are expressions!):
  - comma operator:
Boolean Logic (6.3.2.1, 6.5.13–6.5.15)

- “When any scalar value is converted to `Bool`, the result is 0 `false` if the value compares equal to 0; otherwise, the result is 1.” (6.3.2.1)
- `(a != 0)` and `'a'` are equivalent (if `a` is of scalar type), so are `(a == 0)` and `!a`
- for pointers (later): `p == NULL` and `!p` are equivalent

Bitwise Operators (6.5.3.3, 6.5.7, 6.5.10–12)

- Often used in hardware level programming:
  Communicate with “the hardware” via memory-mapped registers – single bits or groups of bits have particular, platform dependent meaning.
- Bitwise And, Or, Xor (6.5.10-12):
  \[
  \begin{align*}
  0101_2 & \& 1100_2 = 0000_2 \\
  0101_2 & \& 1100_2 = 0001_2
  \end{align*}
  \]
- Useful idioms (assuming 4-bit type):
  - Set the 3rd bit: `a |= 0100_2`
  - Clear the 2nd bit: `a &= 1101_2`
  - Test whether 2nd bit set: `a & 0010_2`
- Shift (6.5.7): `a << 2, a >> 2`
  (unsigned (!) filled up with 0 at left and right)
- Bitwise complement (6.5.3.3): `~a`

Be careful with signed types (bit-operations at best on unsigned):

`~(int(1)) == 0xFFFFFFFF // == -2`
"An lvalue is an expression with an object type or an incomplete type other than void;"

"The name "lvalue" [comes from] $E_1 = E_2$, in which the left operand $E_1$ is required to be a (modifiable) lvalue. What is sometimes called "rvalue" is in this International Standard described as the "value of an expression". An obvious example of an lvalue is an identifier of an object."

```
int x, a[3], *p;
x = 27;
a[1] = 0;
p = &x;
*p = 13;
&x = ...; // no
a = ...; // no, only as initializer
```
Statements (6.8)

- "A statement specifies an action to be performed. Except as indicated, statements are executed in sequence."

- also basically like Java:
  - Selection statements (6.8.4): if, else, switch
  - Iteration statements (6.8.5): while, do ... while, for
  - Jump statements (6.8.6): goto, continue, break, return

Functions
Function Definitions (6.9.1)

```c
int max(int a, int b)
{
    return a > b ? a : b;
}
```

- no nesting, no member functions
- all in file (global) scope (but: module scope possible (later))
- call-by-value semantics (call-by-reference: later)

Function declaration (vs. definition):

```c
int max(int a, int b);
// param. names just "decoration"
int max(int, int);
```

- “Zero or many declarations, exactly one definition.”

Scopes
Scopes of Identifiers (6.2.1)

- The same identifier can denote different entities at diff. points in the program.
- For each different entity that an identifier designates, the identifier is **visible** (i.e., can be used) only within a region of program text called its **scope**.

**Scopes of Identifiers (6.2.1)**

- **Different entities** designated by the **same identifier** either have different scopes, or are in different name spaces.
  
  There are **four kinds of scopes**:
  
  - function scope
  - file scope
  - block scope
  - function prototype

- A label name is the only kind of identifier that has function scope.

- Every other identifier has scope determined by the placement of its declaration (in a declarator or type specifier).

- **Declare before use:**
  
  Each identifier must be declared before (i.e. earlier in the source file) its first use in, e.g., an expression. (**Unlike Java**)

```c
int a;  // file scope (F)
struct { int a; } s;  // name-space
int f ( int a ) // block scope, block (A)
{  // uses a:(A)
   if ( a )  // uses a:(B)
      a = 0;   // uses a:(A)
   int a = 27;  // block scope, block (B)
   s.a = a;   // uses a:(B)
   return a;  // uses a:(A)
}

int main() { return f(a);  // uses a:(F)  }  // uses a:(F)
```
Scopes of Identifiers (6.2.1)

- "Different entities" designated by the same identifier
  either have different scopes, or are in different name spaces.
  There are four kinds of scopes:
  function, file, block, and function prototype."
- "A label name is the only kind of identifier that has function scope."
- "Every other identifier has scope determined by the placement of its
  declaration (in a declarator or type specifier)."

Declare before use:
each identifier must be declared before (i.e. earlier in the source file) its first
use in, e.g., an expression. (Unlike Java!)
- "Hidden" identifiers are not accessible:

```c
int a; /* (F) */

void f(int a) { /* (A) */
  a = 0; // uses a:(A).
  // a:(F) not accessible here, "hidden" by a:(A)
}
```

Pointers
### Variables in the System’s Memory

1. `char c = 127;`

   The compiler chose to store values of 'c' at memory cell with address 0x1001

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>0x7F</td>
</tr>
<tr>
<td>0x1001</td>
<td></td>
</tr>
<tr>
<td>0x1002</td>
<td>0x1003</td>
</tr>
<tr>
<td>0x1004</td>
<td>0x1005</td>
</tr>
<tr>
<td>0x1006</td>
<td>0x1007</td>
</tr>
<tr>
<td>0x1008</td>
<td>0x1009</td>
</tr>
<tr>
<td>0x100A</td>
<td>0x100B</td>
</tr>
<tr>
<td>0x100C</td>
<td>0x100D</td>
</tr>
<tr>
<td>0x100E</td>
<td>0x100F</td>
</tr>
</tbody>
</table>

### Assigning Variables = Update Memory

1. `char c = 127;`
2. `c = c + 1;`

   "c = c + 1" means: the new value of c is the old value plus 1; in assembler:
   - Read 0x1001, R;
   - Inc R;
   - Write R, 0x1001

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>0x7F</td>
</tr>
<tr>
<td>0x1001</td>
<td></td>
</tr>
<tr>
<td>0x1002</td>
<td>0x1003</td>
</tr>
<tr>
<td>0x1004</td>
<td>0x1005</td>
</tr>
<tr>
<td>0x1006</td>
<td>0x1007</td>
</tr>
<tr>
<td>0x1008</td>
<td>0x1009</td>
</tr>
<tr>
<td>0x100A</td>
<td>0x100B</td>
</tr>
<tr>
<td>0x100C</td>
<td>0x100D</td>
</tr>
<tr>
<td>0x100E</td>
<td>0x100F</td>
</tr>
</tbody>
</table>

   ...
Assigning Variables = Update Memory

1. `char c = 127;`
2. `c = c + 1;`

“c = c + 1” means: the new value of `c` is the old value plus 1; in assembler:
read 0x1001, R; inc R;
write R, 0x1001

A Pointer to ‘c’ (16-bit Architecture)

1. `char c = 127;`
2. `c = c + 1;`
3. `char* p = &c;`

the compiler chose to store values of variable ‘p’ at memory cells (!) with addresses 0x1002 and 0x1003
Dereferencing Pointers

1. `char c = 127;`
2. `c = c + 1;`
3. `char* p = &c;`
4. `*p = *p + 3;`

**Note:**
- `*p, rhs`: get the value of `p` (0x1001) and read the value at that address (at 0x1001, yields 0x80)
- `*p, lhs`: get the value of `p` again (0x1001), write the addition result (0x83) to that address (to 0x1001)

**Add:**
- Add 3 to the value just obtained (yields 0x83)
### Dereferencing Pointers

1. `char c = 127;`
2. `c = c + 1;`
3. `char *p = &c;`
4. `*p = *p + 3;`

*`p, rhs`: get the value of `p` (0x1001) and read the value at that address (at 0x1001, yields 0x80)

5. `char *q = p;`

*`p, lhs`: get the value of `p` again (0x1001), write the addition result (0x83) to that address (to 0x1001)

---

### Assigning Pointers

1. `char c = 127;`
2. `c = c + 1;`
3. `char *p = &c;`
4. `*p = *p + 3;`
5. `char *q = p;`

* assume the compiler choses to store values of variable 'q' at memory cells (!) with addresses 0x1004 and 0x1005
Assigning Pointers

1. `char c = 127;`
2. `c = c + 1;`
3. `char* p = &c;`
4. `*p = *p + 3;`
5. `char* q = p;`

Pointers to Pointers

1. `char c = 127;`
2. `c = c + 1;`
3. `char* p = &c;`
4. `*p = *p + 3;`
5. `char* q = p;`
6. `char** r = &q;`

assume the compiler choses to store values of variable 'r' at memory cells (!) with addresses 0x1006 and 0x1007
Pointers to Pointers

1. `char c = 127;`
2. `c = c + 1;`
3. `char* p = &c;`
4. `*p = *p + 3;`
5. `char* q = p;`
6. `char** r = &q;`

Using Pointers to Pointers

1. `char c = 127;`
2. `c = c + 1;`
3. `char* p = &c;`
4. `*p = *p + 3;`
5. `char* q = p;`
6. `char** r = &q;`
7. `*r = p;`
8. `**r = 5;`
Using Pointers to Pointers

1. `char c = 127;`
2. `c = c + 1;`
3. `char* p = &c;`
4. `*p = *p + 3;`
5. `char* q = p;`
6. `char** r = &q;`
7. `*r = p;`
8. `**r = 5;`
Pointers vs. Arrays

Arrays

```c
char a[5] = {'H', 'e', 'l', 'l', 'o'};
```
Arrays

reserve some space for 5 chars...

```c
char a[5] = {'H', 'e', 'l', 'l', 'o'};
```

...and let a point to that space

---

...and let a point to that space

```c
char a[5] = {'H', 'e', 'l', 'l', 'o'};
```

...and let a point to that space
Arrays

```c
char a[5] = {'H', 'e', 'l', 'l', 'o'};
int i;
for (i = 0; i < 5; ++i)
a[i] = 'x';
```
Arrays

```c
char a[5] = { 'H', 'e', 'l', 'l', 'o' };
int i;
for (i = 0; i < 5; ++i)
a[i] = 'x';
```
Arrays

```c
char a[5] = { 'H', 'e', 'l', 'l', 'o' };
int i;
for (i = 0; i < 5; ++i)
a[i] = 'x';
```
Arrays vs. Pointers

```c
char a[5] = { 'H', 'e', 'l', 'l', 'o' };
char* p = a; // not &a!
for (int i = 0; i < 5; ++i, ++p)
    *p = 'o';
```
Arrays vs. Pointers

```c
1 char a[5] = { 'H', 'e', 'l', 'l', 'o' };
2 char* p = a; // not &a !
3 for (int i = 0; i < 5; ++i, ++p)
4   *p = 'o';
```

```c
0x1000 0x1001 0x1002 0x1003 0x1004 0x1005 0x1006 0x1007
0x1008 0x1009 0x100A 0x100B 0x100C 0x100D 0x100E 0x100F
0x1010 0x1011 0x1012 0x1013 0x1014 0x1015 0x1016 0x1017
```

Arrays vs. Pointers

```c
1 char a[5] = { 'H', 'e', 'l', 'l', 'o' };
2 char* p = a; // not &a !
3 for (int i = 0; i < 5; ++i, ++p)
4   *p = 'o';
```

```c
0x1000 0x1001 0x1002 0x1003 0x1004 0x1005 0x1006 0x1007
0x1008 0x1009 0x100A 0x100B 0x100C 0x100D 0x100E 0x100F
0x1010 0x1011 0x1012 0x1013 0x1014 0x1015 0x1016 0x1017
```
Arrays vs. Pointers

```c
char a[5] = { 'H', 'e', 'l', 'l', 'o' };
char* p = a; // not &a!
for (int i = 0; i < 5; ++i, ++p)
    *p = 'o';
```
Arrays vs. Pointers

```c
char a[5] = { 'H', 'e', 'l', 'l', 'o' };
char* p = a; // not &a !
for (int i = 0; i < 5; ++i, ++p)
    *p = 'o';
```

Integer Arrays

```c
int a[3] = { 10, 010, 0x1234 };
```

...and let `a` point to that space
Integer Arrays

```c
int a[3] = { 10, 010, 0x1234 };
int i;
for (i = 0; i < 3; ++i)
a[i] = 0x27;
```

```
0x1000 0x1001 0x1002 0x1003 0x1004 0x1005 0x1006 0x1007
0x1008 0x1009 0x100A 0x100B 0x100C 0x100D 0x100E 0x100F
0x1010 0x1011 0x1012 0x1013 0x1014 0x1015 0x1016 0x1017
```

Integer Arrays

```c
int a[3] = { 10, 010, 0x1234 };
int i;
for (i = 0; i < 3; ++i)
a[i] = 0x27;
```

```
0x1000 0x1001 0x1002 0x1003 0x1004 0x1005 0x1006 0x1007
0x1008 0x1009 0x100A 0x100B 0x100C 0x100D 0x100E 0x100F
0x1010 0x1011 0x1012 0x1013 0x1014 0x1015 0x1016 0x1017
```
```c
int a[3] = { 10, 010, 0x1234 };
int i;
for (i = 0; i < 3; ++i)
    a[i] = 0x27;
```
### Integer Arrays vs. Pointers

1. ```
   int a[3] = { 10, 010, 0x1234 };
   int* p = a;
   for (int i = 0; i < 3; ++p)
     *p = 0x3421;
```
Integer Arrays vs. Pointers

```c
int a[3] = { 10, 010, 0x1234 }; int* p = a;
for (int i = 0; i < 3; ++p)
  *p = 0x3421;
```
Integer Arrays vs. Pointers

1 \texttt{int} \ a[3] = \{10, 010, 0x1234\};
2 \texttt{int} * \ p = \ a;
3 \texttt{for} ( \texttt{int} \ i = 0; i < 3; ++p)
4 \quad *p = 0x3421;

Pointers to 'void', Pointer Arithmetic
int[3] a = { 10, 010, 0x1234 };  
int* p = a;  
void* q = a;  
for (int i = 0; i < 3; ++i) {
    p++;  
    q++;  
}
```c
int [3] a = { 10, 010, 0x1234 };  
int* p = a;  
void* q = a;  
for (int i = 0; i < 3; ++i) {    
p++;  
q++;  
}
```
int[3] a = { 10, 010, 0x1234 };  
int* p = a;  
void* q = a;  
for (int i = 0; i < 3; ++i) {
    p++;  
    q++;  
}
Pointers: Observation

- A variable of pointer type just stores an address.
- So do variables of array type.
- Pointers can point to a certain type, or to void.
- “A pointer to void shall have the same representation and alignment requirements as a pointer to a character type.” (6.2.5.26)
- The effect of “incrementing” a pointer depends on the type pointed to.

```c
int a[2];
int* p = a;
++p; // points to a[1]
void* q = a;
q = sizeof(int); // points to a[1]
++q; // may point into the middle
```
```c
int[3] a = {10, 010, 0x1234}, i = 0;

int* p = a; // not &a!

if (a[0] == *p) i++;
if (a[1] == *(p+1)) i++;
if (a[2] == *(p+2)) i++;
if (&(a[2]) - p == 2) i++;

void* q = a;

if (a[2] == *((int*)(q + (2 * sizeof(int)))))) i++;

// i == 5
```

void as such does not have values, we need to cast 'q' here... note: void* can be casted to everything

Pointers for Call By Reference
Call By Reference with Pointers

```c
void f(int x, int y)
{
    x++, y++;
}

void g(int *p, int *q)
{
    (*p)++, (*q)++;
}

int a = 2, b = 5;

f(a, b);
g(&a, &b);
```

Call By Reference with Pointers
void f(int x, int y) {
    x++, y++;
}

void g(int *p, int *q) {
    (*p)++, (*q)++;  
}

int a = 2, b = 5;
f(a, b);
g(&a, &b);
Call By Reference with Pointers

```c
void f( int x, int y ) {
    x++, y++;
}

void g( int* p, int* q ) {
    (*p)++, (*q)++;  
}

int a = 2, b = 5;

f( a, b );
g( &a, &b );
```

Call By Reference with Pointers

```c
void f( int x, int y ) {
    x++, y++;
}

void g( int* p, int* q ) {
    (*p)++, (*q)++;  
}

int a = 2, b = 5;

f( a, b );
g( &a, &b );
```
Call By Reference with Pointers

```c
void f(int x, int y) {
    x++, y++;
}
void g(int* p, int* q) {
    (*p)++, (*q)++;
}
int a = 2, b = 5;
f(a, b);
g(&a, &b);
```

```
0x1000 0x1001 0x1002 0x1003 0x1004 0x1005 0x1006 0x1007
0x00 0x03 0x00 0x06
0x1008 0x1009 0x100A 0x100B 0x100C 0x100D 0x100E 0x100F
0x01 0x01 0x01 0x01 0x01 0x01 0x01 0x01
0x1010 0x1011 0x1012 0x1013 0x1014 0x1015 0x1016 0x1017
```

Call By Reference with Pointers

```c
void f(int x, int y) {
    x++, y++;
}
void g(int* p, int* q) {
    (*p)++, (*q)++;
}
int a = 2, b = 5;
f(a, b);
g(&a, &b);
```

```
0x1000 0x1001 0x1002 0x1003 0x1004 0x1005 0x1006 0x1007
0x00 0x03 0x00 0x06
0x1008 0x1009 0x100A 0x100B 0x100C 0x100D 0x100E 0x100F
0x01 0x01 0x01 0x01 0x01 0x01 0x01 0x01
0x1010 0x1011 0x1012 0x1013 0x1014 0x1015 0x1016 0x1017
```

48/120
Dynamic Storage & Storage Duration

Dynamic Storage Allocation
```c
typedef struct Node {
    char data;
    struct Node* next;
} Node;

Node c = { 'C', 0 };  // Node c
Node b = { 'B', &c };  // Node b
Node a = { 'A', &b };  // Node a
```
```c
typedef struct Node {
    char data; struct Node* next; } Node;

Node *head = 0, *hlp;

void insert( char d ) {
    hlp = (Node*)malloc( sizeof(Node) );
    hlp->data = d;
    hlp->next = head;
    head = hlp;
}

insert( 'C' );
insert( 'B' );
insert( 'A' );
```

Dynamic Storage Allocation

allocate some space for a `Node`, return its address;
may fail ("out of memory"),
`malloc(3)` yields 0 then

head
```c
typedef struct Node {
    char data; struct Node* next; } Node;

Node *head = 0, *hlp;

void insert(char d) {
    hlp = (Node*)malloc(sizeof(Node));
    hlp->data = d;
    hlp->next = head;
    head = hlp;
}

insert('C');
insert('B');
insert('A');
```

```
0x1000 0x1001 0x1002 0x1003 0x1004 0x1005 0x1006 0x1007
0x1008 0x1009 0x100A 0x100B 0x100C 0x100D 0x100E 0x100F
0x1010 0x1011 0x1012 0x1013 0x1014 0x1015 0x1016 0x1017
```

```
0x1000 0x1001 0x1002 0x1003 0x1004 0x1005 0x1006 0x1007
0x1008 0x1009 0x100A 0x100B 0x100C 0x100D 0x100E 0x100F
0x1010 0x1011 0x1012 0x1013 0x1014 0x1015 0x1016 0x1017
```

```
0x1000 0x1001 0x1002 0x1003 0x1004 0x1005 0x1006 0x1007
0x1008 0x1009 0x100A 0x100B 0x100C 0x100D 0x100E 0x100F
0x1010 0x1011 0x1012 0x1013 0x1014 0x1015 0x1016 0x1017
```


typedef struct Node {
    char data; struct Node* next; } Node;

Node *head = 0, *hlp;

void insert(char d) {
    hlp = (Node*)malloc(sizeof(Node));
    hlp->data = d;
    hlp->next = head;
    head = hlp;
}

insert('C');
insert('B');
insert('A');
```c
typedef struct Node {
    char data; struct Node* next; } Node;

Node *head = 0, *hlp;

void insert( char d ) {
    hlp = (Node*)malloc( sizeof(Node) );
    hlp->data = d;
    hlp->next = head;
    head = hlp;
}

insert( 'C' );
insert( 'B' );
insert( 'A' );
```

```c
... 0x1000 0x1001 0x1002 0x1003 0x1004 0x1005 0x1006 0x1007
    0x1008 0x1009 0x100A 0x100B 0x100C 0x100D 0x100E 0x100F
    0x1010 0x1011 0x1012 0x1013 0x1014 0x1015 0x1016 0x1017
    'C' 0x00 0x00
...```
```c
typedef struct Node {
    char data; struct Node* next; } Node;

Node* head = 0, *help;

void insert(char d) {
    help = (Node*)malloc(sizeof(Node));
    help->data = d;
    help->next = head;
    head = help;
}

insert('C');
insert('B');
insert('A');
```

---

```
0x1000 0x1001 0x1002 0x1003 0x1004 0x1005 0x1006 0x1007
0x1008 0x1009 0x100A 0x100B 0x100C 0x100D 0x100E 0x100F
0x1010 0x1011 0x1012 0x1013 0x1014 0x1015 0x1016 0x1017

... 'C' 0x00 0x00 ...
```
typedef struct Node {
    char data; struct Node* next; } Node;

Node *head = 0, *help;

void insert(char d) {
    help = (Node*)malloc(sizeof(Node));
    help->data = d;
    help->next = head;
    head = help;
}

insert('C');
insert('B');
insert('A');
typedef struct Node {
  char data; struct Node* next;
} Node;

Node *head = 0, *help;

void insert(char d) {
  help = (Node*)malloc(sizeof(Node));
  help->data = d;
  help->next = head;
  head = help;
}

insert('C');
insert('B');
insert('A');
```c
typedef struct Node {
    char data; struct Node* next; } Node;

Node *head = 0, *hlp;

void insert(char d) {
    hlp = (Node*)malloc(sizeof(Node));
    hlp->data = d;
    hlp->next = head;
    head = hlp;
}

insert('C');
insert('B');
insert('A');
```

...
```c
typedef struct Node {
    char data; struct Node* next; } Node;

Node *head = 0, *help;

void insert(char d) {
    help = (Node*)malloc(sizeof(Node));
    help->data = d;
    help->next = head;
    head = help;
}

insert('C');
insert('B');
insert('A');
```

```
0x1000 0x1001 0x1002 0x1003 0x1004 0x1005 0x1006 0x1007
0x1008 0x1009 0x100A 0x100B 0x100C 0x100D 0x100E 0x100F
'B' 0x10 0x08 0x10 0x13 0x10 0x10 0x10 'A' 0x.. 0x.. 0x08
0x1010 0x1011 0x1012 0x1013 0x1014 0x1015 0x1016 0x1017 'C' 0x00 0x00 0x10 0x.. 0x.. 0x10
...
typedef struct Node {
    char data; struct Node* next; } Node;

Node *head = 0, *hlp;

void insert(char d) {
    hlp = (Node*)malloc(sizeof(Node));
    hlp->data = d;
    hlp->next = head;
    head = hlp;
}

insert('C');
insert('B');
insert('A');

...
Dynamic Storage Management

Dynamic Storage Allocation:

- void* malloc( size_t size );
  
  “[...] allocates size bytes and returns a pointer to the allocated memory. The memory is not initialized. [...]”

- “On error, [this function] returns NULL.”

- void free( void* ptr )
  
  “[...] frees the memory space pointed to by ptr, which must have been returned by a previous call to malloc(), [...].”

- “Otherwise, or if free(ptr) has already been called before, undefined behavior occurs.”

- “If ptr is NULL, no operation is performed.”
Dynamic Storage Management

Dynamic Storage Allocation:

- **void** `malloc(size_t size);
  
  "[...] allocates size bytes and returns a pointer to the allocated memory. The memory is not initialized. [...]"
- "On error, [this function] returns NULL."

- **void** `free(void* ptr)
  
  "[...] frees the memory space pointed to by ptr, which must have been returned by a previous call to malloc(), [...]"
- "Otherwise, or if free(ptr) has already been called before, undefined behavior occurs."
- "If ptr is NULL, no operation is performed."

- **No garbage collection!**
  
  Management of dynamic storage is responsibility of the programmer. Unaccessible, not free’d memory is called memory leak.

Dynamic Storage Management Example

```c
void remove() {  
    if (hlp = head) {  
        head = hlp->next;  
        free(hlp);  
    }  
    insert( 'C' ); insert( 'B' ); insert( 'A' ); remove(); insert( 'X' );
```

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>'A'</td>
</tr>
<tr>
<td>0x1001</td>
<td></td>
</tr>
<tr>
<td>0x1002</td>
<td></td>
</tr>
<tr>
<td>0x1003</td>
<td></td>
</tr>
<tr>
<td>0x1004</td>
<td></td>
</tr>
<tr>
<td>0x1005</td>
<td></td>
</tr>
<tr>
<td>0x1006</td>
<td></td>
</tr>
<tr>
<td>0x1007</td>
<td></td>
</tr>
<tr>
<td>0x1008</td>
<td></td>
</tr>
<tr>
<td>0x1009</td>
<td>'B'</td>
</tr>
<tr>
<td>0x100A</td>
<td>'B'</td>
</tr>
<tr>
<td>0x100B</td>
<td>'B'</td>
</tr>
<tr>
<td>0x100C</td>
<td>'B'</td>
</tr>
<tr>
<td>0x100D</td>
<td>'B'</td>
</tr>
<tr>
<td>0x100E</td>
<td>'B'</td>
</tr>
<tr>
<td>0x100F</td>
<td>'B'</td>
</tr>
<tr>
<td>0x1010</td>
<td>'B'</td>
</tr>
<tr>
<td>0x1011</td>
<td>'B'</td>
</tr>
<tr>
<td>0x1012</td>
<td>'B'</td>
</tr>
<tr>
<td>0x1013</td>
<td>'B'</td>
</tr>
<tr>
<td>0x1014</td>
<td>'B'</td>
</tr>
</tbody>
</table>
```

...
void remove() {
  if (hlp = head) {
    head = hlp->next;
    free(hlp);
  }
}

insert('C'); insert('B'); insert('A');
remove();
insert('X');
```c
void remove() {
    if (hlp = head) {
        head = hlp->next;
        free(hlp);
    }
    insert('C'); insert('B'); insert('A');
    remove();
    insert('X');
}
```

```
void remove() {
    if (hlp = head) {
        head = hlp->next;
        free(hlp);
    }
    insert('C'); insert('B'); insert('A');
    remove();
    insert('X');
}
```
void remove() {
    if (hlp = head) {
        head = hlp->next;
        free(hlp);
    }
}
insert('C'); insert('B'); insert('A');
remove();
insert('X');
Dynamic Linked List Iteration

```c
Node* find( char d ) {
    hlp = head;
    while ( hlp ) {
        if ( hlp->data == d )
            break;
        hlp = hlp->next;
    }
    return hlp;
}
```

```c
insert( 'C' ); insert( 'B' ); insert( 'A' );
find( 'B' ); // yields 0x1008
find( 'O' ); // yields 0x0000, aka. NULL
```

Pointers to Struct/Union — ‘.’ vs. ‘->’

```c
typedef struct {
    int x;
    int y;
} coordinate;

coordinate pos = { 13, 27 };
coordinate* p = &pos;

int tmp;

tmp = (*p).x;
(*p).x = (*p).y;
(*p).y = tmp;

tmp = p->x;
p->x = p->y;
p->y = tmp;
```
Storage Duration of Objects

Storage Duration of Objects (6.2.4)

- **“static”** – e.g. variables in program scope:
  - live from program start to end
  - if not explicitly initialized, set to 0 (6.7.8)
- **“automatic”** – non-static variables in local scope:
  - live from block entry to exit
  - not automatically initialised: “initial value […] is indeterminate”
- **“allocated”** – dynamic objects:
  - live from `malloc` to `free`
  - not automatically initialised

“If an object is referred to outside of its lifetime, the behavior is undefined.

The value of a pointer becomes indeterminate when the object it points to reaches the end of its lifetime.”
Storage Duration “Automatic” (Simplified)

```c
void h() { int y; y++; }
void g() { int x = 5; x++; }
int* f() { int c = 3; g(); h(); h(); return &c; }
int a = 27, b, *p;
p = f();
b = *p;
```

- **Local variables** live here
- **malloc()/free()** work here
- **Uninitialised global variables**, set to 0, here
- **Initialised global variables** here
- **Program code** lives here

```
0x1000 0x1001 0x1002 0x1003 0x1004 0x1005 0x1006 0x1007
0x1008 0x1009 0x100A 0x100B 0x100C 0x100D 0x100E 0x100F
0x1010 0x1011 0x1012 0x1013 0x1014 0x1015 0x1016 0x1017
... 0x1019 0x101A 0x101B 0x101C 0x101D 0x101E 0x101F 0x1020
```

---
void h() { int y; y++; }
void g() { int x = 5; x++; }
int* f() { int c = 3; g(); h(); return &c; }

int a = 27, b, *p;
p = f();
b = *p;

stack pointer – stack ends at 0x1012 in this case; stack grows downwards (to smaller addr.)
void h() { int y; y++; }
void g() { int x = 5; x++; }
int* f() { int c = 3; g(); h(); return &c; }

int a = 27, b, *p;
p = f();
b = *p;
void h() { int y; y++; }
void g() { int x = 5; x++; }
int* f() { int c = 3; g(); h(); h(); return &c; }

int a = 27, b, *p;
p = f();
b = *p;

...
void h() { int y; y++; }
void g() { int x = 5; x++; }
int* f() { int c = 3; g(); h(); h(); return &c; }
int a = 27, b, *p;
p = f();
b = *p;

(now) y – not explicitly initialised, thus initial value is indeterminate
```
void h() { int y; y++; }
void g() { int x = 5; x++; }
int* f() { int c = 3; g(); h(); return &c; }
int a = 27, b, *p;
p = f();
b = *p;
```

Storage Duration “Automatic” (Simplified)
void h() { int y; y++; }
void g() { int x = 5; x++; }
int* f() { int c = 3; g(); h(); h(); return &c; }

int a = 27, b, *p;
p = f();
b = *p;

...
```c
void h() { int y; y++; }
void g() { int x = 5; x++; }
int* f() { int c = 3; g(); h(); h(); return &c; }

int a = 27, b, *p;
p = f();
b = *p;
```
void h() { int y; y++; }
void g() { int x = 5; x++; }
int* f() { int c = 3; g(); h(); h(); return &c; }

int a = 27, b, *p;
p = f();
b = *p;

p refers to a non-alive object, the behavior is undefined (everything may happen, from ‘crash’ to ‘ignore’).

Storage Classes and Qualifiers
Storage Class Specifiers (6.7.1)

```c
typedef char letter;

extern int x;
extern int f();

static int x; // two uses! (→ later)
static int f();

auto x; // "historic"

register y; // "historic"
```
• → modules, linking (later)
• usually only `extern` in headers (later)

```c
// not _defined_ here, "imported"
//
extern int x;
extern void f();

// declared _and_ defined here, "exported" ...
//
int y;

int g() {
  x = y = 27;
  f();
}
```

```c
// declared _and_ defined here,
// _not_ "exported" ...
//
static int x;
static void g();

int f() {
  static int a = 0;
  a++;
  printf( "%s\n", a );
}

f(); f(); f(); // yields 1, 2, 3
```
Qualifiers (6.7.3)

```c
int x;
const int y;
volatile int z;
int* restrict p; // aliasing
const volatile int a;
```
restrict:

• “[... lengthy formal definition ...]”

• “[... If these requirements are not met, then the behavior is undefined.”

• $→$ use extremely carefully (i.e. if in doubt, not at all)

---

Excursion: Memory Mapped I/O

• **Intuition:** some memory addresses are wired to hardware

  • writing to the address causes a pin to change logical value

  • reading the address gives logical value of a pin

```
0x1000 0x1001 0x1002 0x1003 0x1004 0x1005 0x1006 0x1007 0x00 0x00
0x1008 0x1009 0x100A 0x100B 0x100C 0x100D 0x100E 0x100F 0x00
0x1010 0x1011 0x1012 0x1013 0x1014 0x1015 0x1016 0x1017
...
```
Excursion: Memory Mapped I/O

- **Intuition**: some memory addresses are wired to hardware
- **writing** to the address causes a pin to change logical value
- **reading** the address gives logical value of a pin

```
0x1000  0x1001  0x1002  0x1003  0x1004  0x1005  0x1006  0x1007
0x1008  0x1009  0x100A  0x100B  0x100C  0x100D  0x100E  0x100F
0x1010  0x1011  0x1012  0x1013  0x1014  0x1015  0x1016  0x1017
```

Excursion: Memory Mapped I/O

- **Intuition**: some memory addresses are wired to hardware
- **writing** to the address causes a pin to change logical value
- **reading** the address gives logical value of a pin

```
0x1000  0x1001  0x1002  0x1003  0x1004  0x1005  0x1006  0x1007
0x1008  0x1009  0x100A  0x100B  0x100C  0x100D  0x100E  0x100F
0x1010  0x1011  0x1012  0x1013  0x1014  0x1015  0x1016  0x1017
```
• **Intuition**: some memory addresses are wired to hardware
  • **writing** to the address causes a pin to change logical value
  • **reading** the address gives logical value of a pin

• The compiler does not know, “memory is memory”.
Qualifiers: volatile (6.7.3)

```c
volatile char* out = 0x1006;
volatile char* in = 0x1007;

out = 0x01; // switch lamp on

if ((in & 0x01) && (in & 0x01)) { /* ... */ }
```

Strings & Input/Output
Strings are 0-Terminated char Arrays

```c
char* msg = "Hello";
char* str = msg;
```
Strings are 0-Terminated char Arrays

1. char* msg = "Hello";
2. char* str = msg;

String Manipulation (Annex B)

#include <string.h>

provides among others:
# include <string.h>

provides among others:

• size_t **strlen** ( const char* s )
  
  "[...] calculates length of string s, excluding the terminating null byte ('\0')."

• int **strcmp** ( const char* s1, const char* s2 )
  
  "[...] compares the two strings s1 and s2. It returns an integer less than, equal to, or greater than zero if s1 is found, respectively, to be less than, to match, or be greater than s2."
# include <string.h>

provides among others:

- `size_t strlen(const char* s)`
  
  "[...] calculates length of string s, excluding the terminating null byte (\'0\')."

- `int strcmp(const char* s1, const char* s2)`
  
  "[...] compares the two strings s1 and s2. It returns an integer less than, equal to, or greater than zero if s1 is found, respectively, to be less than, to match, or be greater than s2."

- `char* strcpy(char* s1, const char* s2)`
  
  "The strcpy() function copies the string pointed to by s2, including the terminating null byte (\'0\'), to the buffer pointed to by s1."

- `char* strncpy(char* s1, const char* s2, size_t n)`
  
  None of these functions allocates memory!
# include `<string.h>`

provides among others:

- `size_t strlen( const char* s )`
  “[...] calculates length of string `s`, excluding the terminating null byte (`\0`).”

- `int strcmp( const char* s1, const char* s2 )`
  “[...] compares the two strings `s1` and `s2`. It returns an integer less than, equal to, or greater than zero if `s1` is found, respectively, to be less than, to match, or be greater than `s2`.”

- `char* strcpy( char* s1, const char* s2 )`
  “The `strcpy()` function copies the string pointed to by `s2`, including the terminating null byte (`\0`), to the buffer pointed to by `s1`.”

- `char* strncpy( char* s1, const char* s2, size_t n )`

None of these functions allocates memory!

Allocate and copy: (not C99, but POSIX)

- `char* strdup( const char* s )`

---

**Input/Output**
#include <stdio.h>

printf("%s-%d-%f\n", "Hello", 27, 3.14);
Hello, Again

```c
#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.
• %

Zoom In: Preprocessing, Compiling, Linking

```c
#include <stdio.h>

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

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

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

• % gcc -E helloworld.c > helloworld.i
• % gcc -c -o helloworld.i
• % ld -o helloworld [...] helloworld.o [...]  
• % ./helloworld
• Hello World.
• %
#include <stdio.h>

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

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

int main() {
   printf("Hello World\n");
   return f();
}
#include <stdio.h>

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

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

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

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.
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!`
Preprocessing

Preprocessing Directives (6.10)

```c
#include <stdio.h>
#include "battery.h"
#define PI 3.1415
#define DEBUG
#define _pure __attribute__((pure))
#endif
#define __pure __attribute__((pure))
#endif
extern int f() __pure;
```
<table>
<thead>
<tr>
<th>Function</th>
<th>Provides</th>
<th>Needs</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>g(int)</code></td>
<td>int <code>g(int)</code></td>
<td>./g.o</td>
</tr>
<tr>
<td><code>f()</code></td>
<td>int <code>f()</code></td>
<td>int <code>g(int)</code></td>
</tr>
</tbody>
</table>
| `main()` | int `main()` | int `f(int)`
|          |          | int `printf(const char*,...)` |
|          |          | helloworld.o |
| `printf` | int `printf(const char*,...)` | ... |
|          |          | ... libc.a |

---

<table>
<thead>
<tr>
<th>Function</th>
<th>Provides</th>
<th>Needs</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>g(int)</code></td>
<td>int <code>g(int)</code></td>
<td>./g.o</td>
</tr>
<tr>
<td><code>f()</code></td>
<td>int <code>f()</code></td>
<td>int <code>g(int)</code></td>
</tr>
</tbody>
</table>
| `main()` | int `main()` | int `f(int)`
|          |          | int `printf(const char*,...)` |
|          |          | helloworld.o |
| `printf` | int `printf(const char*,...)` | ... |
|          |          | ... libc.a |
provides: int g(int)
needs: ./  g.o

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
Linking

<table>
<thead>
<tr>
<th>provides: int g(int)</th>
<th>needs: ./. g.o</th>
</tr>
</thead>
<tbody>
<tr>
<td>provides: int f()</td>
<td>needs: int g(int)</td>
</tr>
<tr>
<td>provides int main()</td>
<td>needs: int f(int) int printf(const char*,...) helloworld.o</td>
</tr>
<tr>
<td>provides:</td>
<td>needs:</td>
</tr>
<tr>
<td>int printf(const char*,...) ...</td>
<td>... libc.a</td>
</tr>
</tbody>
</table>

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 -llxy

→ 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.
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 coredumpsize 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 Segmentation fault.
#0 0x00000000004004b4 in main() at core.c:3
(gdb)
$p = 0x0
(gdb)
```

Formal Methods for C
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”

*Poor Man’s Requirements Specification aka. How to Formalize Requirements in C?*
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.”
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, 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)

```
#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)));
}
```

**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 ) {
    #ifdef 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
}
```
• 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)*
• “The program has been verified.” tells us not very much.
 Dependability

- “The program has been verified.” tells us not very much.
- One wants to know (and should state):
  - Which specifications have been considered?
• “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)
“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
• 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 > MAXINT) {
    /* no */
}

int e = 0;

void set_error() { e++; }
void clear_error() { e = 0; }

void g() { if (e) { /* ... */ } }
```

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...
}
```
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)**
“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.”

More examples:

- an identifier [...] contains an invalid multibyte character (5.2.1.2)
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)
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)
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)
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)
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)
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**

```c
int main() {
    int *p;
    *p = 27;
    return 0;
}
```
**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.
“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:"
“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).
“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).
“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).
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.”