Formal Methods for C

Overview

1. Brief history
2. Comments
3. Declarations and Scopes
4. Variables
5. Expressions and Statements
6. Functions
7. Scopes
8. Pointers
9. Dynamic Storage & Storage Duration
10. Storage Class Specifiers
11. Strings and I/O
12. Tools & Modules
13. Formal Methods for C
14. Common Errors

Function Pointers

FunctionsintheSystem'sMemory

1. void f()
   return;

CallingFunctions

1. void f()
   return;
2. f();

The compiler chose to store the machine code of 'f' at memory cell with address 0x1001.

Calling 'f' means machine operation CALL with address of callee (here: f, address 0x1001).
A Pointer to `f` (16-bit Architecture)

```c
void f()
{
  return;
}
```

```c
f();
```

```c
void (*p)() =&f;
```

...and then machine operation CALL with R, calls address stored in R.

Dereference Function Pointers

calling via `p` means read value of `p` (here: 0x1001) into register R...

Pointers vs. Arrays

A Pointer to `f` (16-bit Architecture)
char a[5] = { 'H', 'e', 'l', 'l', 'o' };

reserves some space for 5 characters... and let a point to that space...

\begin{verbatim}
0x1000 0x1001 0x1002 0x1003 0x1004 0x1005 0x1006 0x1007 0x1008 0x1009 0x100A 0x100B 0x100C 0x100D 0x100E 0x100F 0x1010 0x1011 0x1012 0x1013 0x1014 0x1015 0x1016 0x1017
\end{verbatim}

...and for i = 0 to 4, a[i] = 'x';

\begin{verbatim}
0x1000 0x1001 0x1002 0x1003 0x1004 0x1005 0x1006 0x1007 0x1008 0x1009 0x100A 0x100B 0x100C 0x100D 0x100E 0x100F 0x1010 0x1011 0x1012 0x1013 0x1014 0x1015 0x1016 0x1017
\end{verbatim}
Array vs Pointers

```c

for (int i = 0; i < 5; i++)
    a[i] = 'H' + i;
```

//not &a!

```c

p = a;
```

```c

*p = 4;
```

```c

p = ++p;
```

```c

for (int i = 0; i < 5; i++)
    *(p + i) = 'o';
```

```c

Array vs Pointers

//not &a!

```c

p = a;
```

```c

*p = 4;
```

```c

p = ++p;
```

```c

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

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

for (int i = 0; i < 5; i++, p++)
  ∗p = 'o';

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

for (int i = 0; i < 3; i++)
  a[i] = 0x27;
1
int a[3] = {10, 0x1234};

2
int i;

3
for (i = 0; i < 3; ++i)
        a[i] = 0x27;

4

p

IntegerArrays vs. Pointers
– 2014 -04 – pointers –
int [3] a = {10, 0x10, 0x1234};

int * p = a;

void * q = a;

for (int i = 0; i < 3; i++) {
    p++;
    q++;
}
1
int [3]a = {10, 010, 0x1234};
2
int * p = a;
3
void * q = a;
4
for (int i = 0; i < 3; i++)
5
    p++;
6
    q++;

...
Call By Reference with Pointers

```c
void f(int a, int b)
{
    int x, y;
    x = a + b;
    y = a + b;
    x++; y++;
    int p, q;
    p = a + b;
    q = a + b;
    p++; q++;
    printf("%d, %d\n", x, y);
    printf("%d, %d\n", p, q);
}

g(&a, &b);

int a = 2, b = 5;

f(a, b);
```
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

Dynamic Storage Allocation

Dynamic Storage Allocation

Dynamic Storage Allocation
Dynamic Storage Allocation:

- Management of dynamic storage is performed using a linked list.
- The memory is not initialized.
- No operation is performed.

```c
void* malloc(size_t size) {
    // Allocate memory
    // Check if memory was returned by a previous call to malloc()
    // Then allocate new memory
    // Otherwise, or if memory has been returned
    // Perform another allocation
}
```

```c
void free(void* ptr) {
    // Free memory
    // Check if memory was returned
    // Then release memory
    // Otherwise, or if memory has been returned
    // Perform another deallocation
}
```

```c
void* realloc(void* ptr, size_t size) {
    // Reallocate memory
    // Check if memory was returned
    // Then allocate new memory
    // Otherwise, or if memory has been returned
    // Perform another reallocation
}
```
Dynamic Storage Management Example

```c
void insert(char ch) {
    // Insert 'X';

    next = 0x0000;
}

char head = hlp;

if (head != NULL) {
    // Remove(

    free(hlp);

    data = 'B';
}

head = hlp;
```

Dynamic Storage Management Example

Dynamic Storage Management Example

Dynamic Storage Management Example
### Dynamic Storage Management Example

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

```plaintext
insert('C');insert('B');insert('A');
remove();
insert('X');
```

### Dynamic LinkedList Iteration

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

```plaintext
insert('C');insert('B');insert('A');
find('B'); //yields0x1008
find('O'); //yields0x0000,aka.NULL
```

### Pointer to Struct/Union—'.' vs. '->'

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

```plaintext
coordinate pos={13,27};
coordinate ∗p=&pos;
it = (p->x);
(p->x)=(p->y);
(p->y)=it;
```

### Storage Duration of Objects

- **"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 initialized: "initial value [...] is indeterminate"

- **"allocated"**—dynamic objects:
  - live from malloc to free
  - not automatically initialized

"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."

### Example: Anatomy of a Linux Program in Memory

```
Example: Anatomy of a Linux Program in Memory

- local variables
- malloc()/free()
- uninitialised
- global variables, set to 0
- initialised global variables
- program code
```
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;

...
StorageClassSpecifiers:

- `extern` (6.7.1)

```
// not defined here, "imported"

extern int x;
extern void f();
```

- `extern` declared and defined here, "exported"

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

- `static` declared and defined here, "not exported"

```
static int x;
static void g();
```

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

Qualifiers (6.7.3)

```
int x;
const int y;
volatile int z;
int *restrict p;
```

```
const volatile int a;
```

```
restrict:
• "[...]lengthy formal definition..."
• "[...]If these requirements are not met, 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 0x00 0x1007 0x00
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

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

out = 0x01; // Switch lamp on
```

If ((in & 0x01) && (in & 0x01)) {
  // ...}
```
Strings are 0-Terminated char Arrays

```c
char *msg = "Hello ";
char *str = msg;
```

- **strlen**
  - 
  - **Purpose**: Calculates the length of a string, excluding the terminating null byte ('\0').
  - **Example**: `strlen("Hello");` returns 5.

- **strcmp**
  - 
  - **Purpose**: Compares 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 greater than `s2`.
  - **Example**: `strcmp("Hello", "Hello")` returns 0; `strcmp("Hello", "World")` returns -1.

- **strcpy**
  - 
  - **Purpose**: Copies a string pointed to by `s2`, including the terminating null byte ('\0'), to the buffer pointed to by `s1`.
  - **Example**: `strcpy(dest, "Hello");` copies "Hello" into dest.

- **strncpy**
  - 
  - **Purpose**: Copies the first `n` characters of a string pointed to by `s2` to the buffer pointed to by `s1`, including the terminating null byte ('\0').
  - **Example**: `strncpy(dest, "Hello", 5);` copies the first 5 characters of "Hello" into dest.

None of these functions allocates memory!

Allocate and copy: (not C99, but POSIX)

- **strdup**
  - 
  - **Purpose**: Copies `s`, including the terminating null byte ('\0'), into a new buffer.
  - **Example**: `char *dup = strdup("Hello");` creates a copy of "Hello".

---

**Printing**

```c
#include <stdio.h>

void printf("Hello", 27, 3.14);
```

---

**Tools & Modules**

---

**String Manipulation ( Annex D)**
Modules At Work

Hello, Again
#include <stdio.h>  

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

int g(int) {  
    return x;  
}  

int main() {  
    printf(helloworld.o
        fprintf(stderr,"honk
    return 0;
}

% gcc -E helloworld.c -o helloworld.i
% gcc -c g.c f.c
% ls *.o
% gcc -l -o a.out helloworld.o g.o
% ./a.out

Hi!
Hello World.
Linking provides:
\[ \text{int } g(\text{int}) \]
needs:
\[ \text{...} \]
provides:
\[ \text{int } f() \]
needs:
\[ \text{int } g(\text{int}) \]
provides:
\[ \text{int } main() \]
needs:
\[ \text{int } f(\text{int}) \text{int printf(const char*...,...} \]
provides:
\[ \text{int printf(const char*,...)} \]
...needs:
\[ \text{...} \]

g.of.o
helloworld.o
libc.a

Compiler—produces:
\[ \text{main.o} \]
\[ \text{x.o} \]
\[ \text{main.o} \]
\[ \text{x.o} \]
\[ \text{a.out} \]

Example:
\[ \text{gcc -c main.c} \]
—produces
\[ \text{main.o} \]
\[ \text{gcc -c -o x.o main.c} \]
—produces
\[ \text{x.o} \]

- \text{gdb(a.out [core])} \hspace{1cm} \text{ddd(a.out [core])} \hspace{1cm} \text{nm(a.o)} \hspace{1cm} \text{make for an example.} \]

Core Dumps

\[ \text{c} \hspace{1cm} \text{⃝ GustavoDuarte2009, used by permission.} \hspace{1cm} \text{http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory} \]

- \text{Recall: Anatomy of a Linux Program in Memory}
Recall: Anatomy of a Linux Program in Memory

A core dump (basically) is this memory written to a file.

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

1. `%gcc -gcore.c`
2. `%limitcoredumpsize`
3. `coredumpsize 0kbytes`
4. `%limitcoredumpsize 1g`
5. `%./a.out`
6. Segmentation fault (core dumped)
7. `%ls -lhcore`
8. `-rw------- - - - - - - 1 user user 232K Feb 29 1 1:11 core`
9. `%gdba.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
17. $1 = (int *) 0x0
18. (gdb)
```

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:

1. \( \varphi_1 \): the return value is 10 divided by parameter (if parameter not 0)
2. \( \varphi_2 \): the value of variable \( x \) is "always" strictly greater than 3
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?
#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 the macro NDEBUG is defined when including <assert.h>,
  expression is not evaluated (thus should be side-effect free).

#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>
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) ) );
}

int f(int x) {
  assert(x != 0);
  // pre-condition
  int r = 10 / x;
  assert(r = 10 / x);
  // post-condition
  return r;
}

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;
}

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);
}

Some verification tools simply verify each assert statement:
When executed, 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
...
"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.
• Review.
• Model-checking procedure.

("We verified the program!" – "What did the tools say?" – "Verification failed.")

In some contexts: model-checking tool claims correctness.
Most generic errors boil down to:

- Specified but unwanted behaviour, e.g., over/underflows
- 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:

```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)
        {
            /*...*/
        }
    }
}
```
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.

```
char a[ ]="hello ", b[ ]="hello ";
// a==b?
int i=0; f(+ +i, + +i, + +i);
// f(1,2,3)?
int g()
{
    int a,b;
// &a>
}
```

```c
int *p = malloc(sizeof(int));
// q>p?
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)
- 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)
More examples:

• an identifier \([\ldots]\) 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)

• pointersthatdonotpointinto,orjustbeyond,thesamearrayobjectare subtracted (6.5.6)
• An arraysubscriptisoutofrange\([\ldots]\)(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., etc.
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)

Modern operating systems provide memory protection.

• Accessing memory which the process is not allowed to access is observed by the operating system.

• Typically an instance of “accessing an object outside its lifetime”.

• But: other way round does not hold, accessing an object outside its lifetime does not imply a segmentation violation.

• Some platforms (e.g. SPARC): unaligned memory access, i.e. outside word boundaries, not supported by hardware ("bus error"). Operating system notifies process, default handler: terminate, dump core.

Implementation-Defined Behaviour (J.3)

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

• Precedence of operators
• Precedence of operators
• Precedence of operators
• Precedence of operators
• Precedence of operators
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).
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).

...and so on.

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

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

References