Formal Methods for C

Seminar – Summer Semester 2014

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

Content

- Brief history
- Comments
- Declarations and Scopes
  - Variables
  - Expressions and Statements
  - Functions
  - Scopes
- Pointers
- Dynamic Storage & Storage Duration
- Storage Class Specifiers
- Strings and I/O
- Tools & Modules
- Formal Methods for C
- Common Errors
Function Pointers

Functions in the System’s Memory

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

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

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

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

```c
void f() { return; }
f();
void (*p)() = &f;
```

'p' is a variable which stores the address of a function (here: of 'f')
Dereference Function Pointers

1. `void f() { return; }`
2. `f();`
3. `void (*p)() = &f;`
4. `(*p)();`

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

1. `void f() { return; }`
2. `f();`
3. `void (*p)() = &f;`
4. `(*p)();`

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

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

```
0x10 0x1001 0x1002 0x1003 0x1004 0x1005 0x1006 0x1007
      RET                0x10 0x01
    0x1008 0x1009 0x100A 0x100B 0x100C 0x100D 0x100E 0x100F
      CALL 0x10 0x01
    0x1010 0x1011 0x1012 0x1013 0x1014 0x1015 0x1016 0x1017
      MOV 0x10 0x04 R CALL R
```

Pointers vs. Arrays
Arrays

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

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

```
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

1. `char a[5] = { 'H', 'e', 'l', 'l', 'o' };`
2. `int i;`
3. `for (i = 0; i < 5; ++i)`
4. `a[i] = 'x';`

```
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

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

1. `char a[5] = {'H', 'e', 'l', 'l', 'o'};`
2. `int i;`
3. `for (i = 0; i < 5; ++i)`
4. `a[i] = 'x';`

Arrays
Arrays

1. `char a[5] = { 'H', 'e', 'l', 'l', 'o' };`
2. `int i;`
3. `for (i = 0; i < 5; ++i)`
4. `a[i] = 'x';`

Arrays vs. Pointers

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';`
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';
```
Arrays vs. Pointers

```
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';
```
Arrays vs. Pointers

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';`
**Integer Arrays**

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

2. `int i;`

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

---

**Integer Arrays**

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

2. `int i;`

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

---
Integer Arrays

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

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

**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

```c
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;
```

```
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 vs. Pointers

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

```
0x1000 0x1001 0x1002 0x1003 0x1004 0x1005 0x1006 0x1007
0x1008 0x1009 0x100A 0x100B 0x100C 0x100D 0x100E 0x100F
0x1010 0x1011 0x1012 0x1013 0x1014 0x1015 0x1016 0x1017
...```
**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++;
}
```
**Pointer to 'void'**

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

**Pointer to 'void'**

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

1. `int[3] a = { 10, 010, 0x1234 };`
2. `int* p = a;`
3. `void* q = a;`
4. `for (int i = 0; i < 3; ++i) {
   p++;
   q++;
}`
Pointer to ’void’

```c
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.

```
int a[2];
int* p = a;  // points to a[1]
++p;          // points to a[1]

void* q = a;
q += sizeof(int);  // points to a[1]
++q;    // may point into the middle
```

Pointer Arithmetic

```
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);
```
```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);
```

---

```
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);
```

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);
```
Dynamic Storage & Storage Duration

Dynamic Storage Allocation
A Linked List

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

Node c = {'C', 0};
Node b = {'B', &c};
Node a = {'A', &b};
```

Dynamic Storage Allocation

```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

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');
```

Dynamic Storage Allocation

```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
...
...
...
...
```
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');
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');

...
Dynamic Storage Allocation

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');
```

```
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

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;

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

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

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**

```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');
```

Data:
- 'B': next: 0x1013
- 'C': next: 0x0000
Dynamic Storage Allocation

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

Node* head = 0;

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

```c
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.”

- 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');
```

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>0x1001</td>
<td>0x1002</td>
<td>0x1003</td>
<td>0x1004</td>
<td>0x1005</td>
<td>0x1006</td>
</tr>
<tr>
<td>'B'</td>
<td>0x10</td>
<td>0x13</td>
<td>'A'</td>
<td>0x10</td>
<td>0x08</td>
<td></td>
</tr>
<tr>
<td>0x1010</td>
<td>0x1811</td>
<td>0x1812</td>
<td>0x1813</td>
<td>0x1814</td>
<td>0x1815</td>
<td>0x1816</td>
</tr>
<tr>
<td>'C'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

...
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 Storage Management Example

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

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

int tmp;   // tmp = pos.x;
tmp = (*p).x;
(*p).x = (*p).y;
(*p).y = tmp;

tmp = p->x;
p->x = p->y;
p->y = tmp;
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.”

---

Example: Anatomy of a Linux Program in Memory

![Diagram of memory layout](http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory)
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;
## Storage Duration “Automatic” (Simplified)

```c
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;
```

### Diagram:
- **`void h()`**
- **`void g()`**
- **`int* f()`**
- **`int a = 27, b, *p;`**
- **`p = f();`**
- **`b = *p;`**

---

**Storage Duration “Automatic” (Simplified)**

```c
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;
```

### Diagram:
- **`void h()`**
- **`void g()`**
- **`int* f()`**
- **`int a = 27, b, *p;`**
- **`p = f();`**
- **`b = *p;`**

### x no longer alive!
### Storage Duration “Automatic” (Simplified)

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

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

y no longer alive!
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(); 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(); 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
typedef char letter;

extern int x;
extern int f();

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

auto x; // "historic"
register y; // "historic"
Storage Class Specifiers: `extern` (6.7.1)

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

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

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

- modules, linking (later)
- usually only `extern` in headers (later)

---

Storage Class Specifiers: `static` (6.7.1)

```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

<table>
<thead>
<tr>
<th>0x1000</th>
<th>0x1001</th>
<th>0x1002</th>
<th>0x1003</th>
<th>0x1004</th>
<th>0x1005</th>
<th>0x1006</th>
<th>0x1007</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1008</td>
<td>0x1009</td>
<td>0x100A</td>
<td>0x100B</td>
<td>0x100C</td>
<td>0x100D</td>
<td>0x100E</td>
<td>0x100F</td>
</tr>
<tr>
<td>0x1010</td>
<td>0x1011</td>
<td>0x1012</td>
<td>0x1013</td>
<td>0x1014</td>
<td>0x1015</td>
<td>0x1016</td>
<td>0x1017</td>
</tr>
</tbody>
</table>

...
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
**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

- The compiler does not know, "memory is memory".

---

**Qualifiers: volatile (6.7.3)**

```c
1 volatile char* out = 0x1006;
2 volatile char* in = 0x1007;
3
4 out = 0x01; // switch lamp on
5
6 if (in & 0x01) { /* ... */ }
7
8 if ((in & 0x01) && (in & 0x01)) { /* ... */ }
```
Strings & Input/Output

Strings
Strings are **0-Terminated char** Arrays

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

![Diagram of strings and char arrays]

Strings are **0-Terminated char** Arrays

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

![Diagram of strings and char arrays]
# 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 − %i − %f
", "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.
- %
Modules

```c
#include <stdio.h>

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

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

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

Split into:

- `.h` (header): declarations
- `.c`: definitions, use headers to "import" declarations
### Modules

```c
#include <stdio.h>

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

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

int main() {
  printf("Hello World.
");
  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.`
Modules At Work

preprocess & compile:
- % gcc -c g.c f.c \ helloworld.c
- % ls *.o
- f.o g.o helloworld.o

link:
- % gcc g.o f.o helloworld.o

execute:
- % ./a.out
- Hello World.

fix and re-build:
- % gcc -c helloworld.c
- % gcc g.o f.o helloworld.o
- % ./a.out
- Hi!
## Preprocessing Directives (6.10)

```c
#include <stdio.h>
#include "battery.h"

#define PI 3.1415

#define DEBUG
#if DEBUG
    #define __pure __attribute__((pure))
#else
    #define __pure /* no pure */
#endif

extern int f() __pure;
```
Linking

- `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*,...), ...</td>
</tr>
<tr>
<td></td>
<td>Needs: libc.a</td>
</tr>
</tbody>
</table>

```
g.o
f.o
helloworld.o
```

```
a.out
```

**Compiler**

```
gcc [OPTION]... infile...
```

- `-E` – preprocess only
- `-c` – compile only, don’t link
  
  **Example:** `gcc -c main.c` — produces `main.o`

- `-o outfile` – write output to `outfile`
  
  **Example:** `gcc -c -o x.o main.c` — produces `x.o`

- `-g` – add debug information

- `-W`, `-Wall`, ... – enable warnings

- `-I dir` – add `dir` to include path for searching headers

- `-L dir` – add `dir` to library path for searching libraries

- `-D macro[=defn]` – define macro (to defn)
  
  **Example:** `gcc -DDEBUG -DMAGICNUMBER=27`

- `-l library` link against `liblibrary.{a,so}`, order matters
  
  **Example:** `gcc a.o b.o main.o -lxy`

→ 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;
}
```

1. gcc -g core.c
2. limit coredumpsizemax 0 kbytes
3. limit coredumpsizemax 1g
4. ./a.out
5. Segmentation fault (core dumped)
6. ls -lh core
7. -rw------- 1 user user 232K Feb 29 11:11 core
8. gdb a.out core
9. GNU gdb (GDB) 7.4.1-native
10. [...] Core was generated by './a.out'.
12. int main() {
13.     int *p;
14.     *p = 27;
15.     return 0;
16. }
```

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
- ...

Requirements
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)

```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 macro `NDEBUG` is defined when including `<assert.h>`, `expression` is not evaluated (thus should be side-effect free).

abort (7.20.4.1)

```c
#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 ) {
    #ifndef NDEBUG
        unsigned int* old_a = a, *old_b = b;
    #endif
    assert( a && b ); assert( a != b ); // precondition
    *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
- ...

**Outlook**

**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)*
Conformance (4)

- “A program that is
  - correct in all other aspects,
  - operating on correct data,
  - containing unspecified behavior

shall be a correct program and act in accordance with 5.1.2.3. (Program Execution)

- A conforming program is one that is acceptable to a conforming implementation.

- Strictly conforming programs are intended to be maximally portable among conforming implementations.

- An implementation [of C, a compiler] shall be accompanied by a document that defines all implementation-defined and locale-specific characteristics and all extensions.

Over- and Underflows
Over- and Underflows, Casting

• Not specific to C...

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

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

- “If an object that has automatic storage duration is not initialized explicitly, its value is indeterminate.”

```c
void f() {
    int a;
    printf("%i\n", a); // surprise...
}
```

**Unspecified Behaviour (J.1)**
Unspecified Behaviour (J.1)

Each implementation (of a compiler) documents how the choice is made.

For example

- whether two string literals result in distinct arrays (6.4.5)
- the order in which the function designator, arguments, and subexpressions within the arguments are evaluated in a function call (6.5.2.2)
- the layout of storage for function parameters (6.9.1)
- the result of rounding when the value is out of range (7.12.9.5, ...)
- the order and contiguity of storage allocated by successive calls to malloc (7.20.3)
- etc. pp.

```c
char a[] = "hello", b[] = "hello"; // a == b?
int g() { int a, b; } // &a > &b ?
int* p = malloc(sizeof(int));
int* q = malloc(sizeof(int)); // q > p?
```

Undefined Behaviour (J.2)
**Undefined Behaviour (3.4.3)**

“Behaviour, upon use of a non-portable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements.”

“Possible undefined behaviour ranges from

• ignoring the situation completely with unpredictable results,

• to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message),

• to terminating a translation or execution (with the issuance of a diagnostic message).”

“An example of undefined behaviour is the behaviour on integer overflow.”

---

**Undefined Behaviour (J.2)**

**More examples:**

• an identifier [...] contains an invalid multibyte character (5.2.1.2)
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)
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)
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)
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)
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 whole 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)
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 whole 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

```c
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

```c
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).
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).
Implementation-Defined Behaviour (J.3)

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

- J.3.2 Environment, e.g.
  The set of signals, their semantics, and their default handling (7.14).
- J.3.3 Identifiers, e.g.
  The number of significant initial characters in an identifier (5.2.4.1, 6.4.2).
- J.3.4 Characters, e.g.
  The number of bits in a byte (3.6).
- J.3.5 Integers, e.g.
  Any extended integer types that exist in the implementation (6.2.5).
- J.3.6 Floating Point, e.g.
  The accuracy of the floating-point operations […] (5.2.4.2.2).
- J.3.7 Arrays and Pointers, e.g.
  The result of converting a pointer to an integer or vice versa (6.3.2.3).
- etc. pp.

Locale and Common Extensions (J.4, J.5)

- J.4 Locale-specific behaviour

- J.5 Common extensions
  “The following extensions are widely used in many systems, but are not portable to all implementations.”
References