**Language Design Issues**

**Syntax:** how programs look
- Names and reserved words
- Instruction formats
- Grouping

**Semantics:** what programs mean
- Model of computation: sequential, concurrent
- Control and data flow
- Types and data representation

---

**C History**

Developed between 1969 and 1973 along with Unix
Due mostly to Dennis Ritchie
Designed for systems programming
- Operating systems
- Utility programs
- Compilers
- Filters
Evolved from B, which evolved from BCPL

---

**BCPL**

Martin Richards, Cambridge, 1967
Typeless
- Everything a machine word (n-bit integer)
- Pointers (addresses) and integers identical
Memory: undifferentiated array of words
Natural model for word-addressed machines
Local variables depend on frame-pointer-relative addressing: no dynamically-sized automatic objects
Strings awkward: Routines expand and pack bytes to/from word arrays

---

**Euclid’s Algorithm in C**

```c
int gcd(int m, int n )
{
    int r;
    while ((r = m % n) != 0) {
        m = n;
        n = r;
    }
    return n;
}
```

“New syle” function declaration lists number and type of arguments. Originally only listed return type. Generated code did not care how many arguments were actually passed, and everything was a word. Arguments are call-by-value

---

**Euclid on the PDP-11**

```assembly
.globl _gcd
.text
    jsr r5, rsave
L2: mov 4(r5), r1
    sxt r0
    div 6(r5), r0
    mov r1, -10(r5)
    jeq L3
    mov 6(r5), 4(r5)
    mov -10(r5), 6(r5)
    jbr L2
L3: mov 6(r5), r0
    jbr L1
L1: jmp rretrn
```

GPRs: r0–r7
r7=PC, r6=SP, r5=FP

Save SP in FP

r1 = n
r0 = f
r1 = m % n
r = r1 (m % n)
if r == 0 goto L3
m = n
n = r

---

**C History**

Original machine (DEC PDP-11) was very small:
24K bytes of memory, 12K used for operating system
Written when computers were big, capital equipment
Group would get one, develop new language, OS

---

**Euclid’s Algorithm in C**

```c
int gcd(int m, int n )
{
    int r;
    while ((r = m % n) != 0) {
        m = n;
        n = r;
    }
return n;
}
```

Automatic variable
Allocated on stack when function entered, released on return
Parameters & automatic variables accessed via frame pointer
Other temporaries also stacked

---

**C History**

Many language features designed to reduce memory
- Forward declarations required for everything
- Designed to work in one pass: must know everything
- No function nesting
PDP-11 was byte-addressed
- Now standard
- Meant BCPL's word-based model was insufficient
The Design of C
Taken from Dennis Ritchie's C Reference Manual
(Appendix A of Kernighan & Ritchie)

Lexical Conventions
Identifiers (words, e.g., foo, printf)
Sequence of letters, digits, and underscores, starting with a letter or underscore
Keywords (special words, e.g., if, return)
C has fairly few: only 23 keywords. Deliberate: leaves more room for users' names
Comments (between /* and */)
Most fall into two basic styles: start/end sequences as in C, or until end-of-line as in Java's //

Constants/Literals
Integers (e.g., 10)
Should a leading – be part of an integer or not?
Characters (e.g., 'a')
How do you represent non-printable or ' characters?
Floating-point numbers (e.g., 3.5e-10)
Usually fairly complex syntax, easy to get wrong.
Strings (e.g., "Hello")
How do you include a " in a string?

What's in a Name?
In C, each name has a storage class (where it is) and a type (what it is).
Storage classes:
1. automatic
2. static
3. external
4. register
Fundamental types:
1. char
2. int
3. float
4. double
Derived types:
1. arrays
2. functions
3. pointers
4. structures

Objects and lvalues
Object: area of memory
lvalue: refers to an object
An lvalue may appear on the left side of an assignment
a = 3; /* OK: a is an lvalue */
3 = a; /* 3 is not an lvalue */

Conversions
C defines certain automatic conversions:
- A char can be used as an int
- Floating-point arithmetic is always done with doubles; floats are automatically promoted
- int and char may be converted to float or double and back. Result is undefined if it could overflow.
- Adding an integer to a pointer gives a pointer
- Subtracting two pointers to objects of the same type produces an integer

Expressions
Expressions are built from identifiers (foo), constants (3), parenthesis, and unary and binary operators.
Each operator has a precedence and an associativity
Precedence tells us
1 * 2 + 3 * 4 means
(1 * 2) + (3 * 4)
Associativity tells us
1 + 2 + 3 + 4 means
((1 + 2) + 3) + 4
### C's Operators in Precedence Order

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>f(x, x, ...)</code></td>
<td>Function call</td>
</tr>
<tr>
<td><code>a[i]</code></td>
<td>Array access</td>
</tr>
<tr>
<td><code>p-&gt;m</code></td>
<td>Pointer dereference</td>
</tr>
<tr>
<td><code>s.m</code></td>
<td>Structure member access</td>
</tr>
<tr>
<td><code>!b</code></td>
<td>Logical NOT</td>
</tr>
<tr>
<td><code>~i</code></td>
<td>Bitwise NOT</td>
</tr>
<tr>
<td><code>-i</code></td>
<td>Arithmetic NEGATION</td>
</tr>
<tr>
<td><code>l++</code></td>
<td>Pre-increment</td>
</tr>
<tr>
<td><code>l--</code></td>
<td>Pre-decrement</td>
</tr>
<tr>
<td><code>*p</code></td>
<td>Dereference pointer</td>
</tr>
<tr>
<td><code>&amp;l</code></td>
<td>Address of</td>
</tr>
<tr>
<td><code>(type) r</code></td>
<td>Cast to type</td>
</tr>
<tr>
<td><code>sizeof(t)</code></td>
<td>Size of type</td>
</tr>
<tr>
<td><code>n * o</code></td>
<td>Multiplication</td>
</tr>
<tr>
<td><code>n / o</code></td>
<td>Division</td>
</tr>
<tr>
<td><code>n % o</code></td>
<td>Modulus</td>
</tr>
<tr>
<td><code>n + o</code></td>
<td>Addition</td>
</tr>
<tr>
<td><code>n - o</code></td>
<td>Subtraction</td>
</tr>
<tr>
<td><code>n &lt;&lt; j</code></td>
<td>Left shift</td>
</tr>
<tr>
<td><code>n &gt;&gt; j</code></td>
<td>Right shift</td>
</tr>
<tr>
<td><code>n &lt; o</code></td>
<td>Less than</td>
</tr>
<tr>
<td><code>n &gt; o</code></td>
<td>Greater than</td>
</tr>
<tr>
<td><code>n &lt;= o</code></td>
<td>Less than or equal to</td>
</tr>
<tr>
<td><code>n &gt;= o</code></td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td><code>r == r</code></td>
<td>Equality</td>
</tr>
<tr>
<td><code>r != r</code></td>
<td>Inequality</td>
</tr>
<tr>
<td><code>i &amp; j</code></td>
<td>Bitwise AND</td>
</tr>
<tr>
<td>`i</td>
<td>j`</td>
</tr>
<tr>
<td><code>i ^ j</code></td>
<td>Bitwise XOR</td>
</tr>
<tr>
<td><code>b &amp;&amp; c</code></td>
<td>Logical AND</td>
</tr>
<tr>
<td>`b</td>
<td></td>
</tr>
<tr>
<td><code>b ? r : r</code></td>
<td>Ternary operator</td>
</tr>
<tr>
<td><code>l = r</code></td>
<td>Assignment</td>
</tr>
<tr>
<td><code>l += n</code></td>
<td>Increment and assignment</td>
</tr>
<tr>
<td><code>l -= n</code></td>
<td>Decrement and assignment</td>
</tr>
<tr>
<td><code>l *= n</code></td>
<td>Multiplication and assignment</td>
</tr>
<tr>
<td><code>l /= n</code></td>
<td>Division and assignment</td>
</tr>
<tr>
<td><code>l %= n</code></td>
<td>Modulus and assignment</td>
</tr>
<tr>
<td><code>l &amp;= i</code></td>
<td>AND and assignment</td>
</tr>
<tr>
<td><code>l ^= i</code></td>
<td>XOR and assignment</td>
</tr>
<tr>
<td>`l</td>
<td>= i`</td>
</tr>
<tr>
<td><code>l &lt;&lt;= i</code></td>
<td>Left shift and assignment</td>
</tr>
<tr>
<td><code>l &gt;&gt;= i</code></td>
<td>Right shift and assignment</td>
</tr>
</tbody>
</table>

### Declarators

**Declaration:** string of specifiers followed by a declarator

- **Basic Type:**
  - `static unsigned int (*f[10])(int, char*)[10];`
    - specifiers
    - declarator

Declarator's notation matches that of an expression: use it to return the basic type.

Largely regarded as the worst syntactic aspect of C: both pre- (pointers) and post-fix operators (arrays, functions).

### Declaration Syntax

**Is int *f() a pointer to a function returning an int, or a function that returns a pointer to an int?**

Hint: precedence rules for declarators match those for expressions.

Parentheses resolve such ambiguities:

- `int *f();` Function returning pointer to int
- `int (*f)();` Pointer to function returning int

### External Definitions

“A C program consists of a sequence of external definitions”

Functions, simple variables, and arrays may be defined.

“An external definition declares an identifier to have storage class extern and a specified type”

Example:

```c
int max(a, b, c)
int a, b, c;
{
    int m;
    m = (a > b) ? a : b ;
    return m > c ? m : c ;
}
```
More C trivia

The first C compilers did not check the number and type of function arguments.

The biggest change made when C was standardized was to require the type of function arguments to be defined:

<table>
<thead>
<tr>
<th>Old-style</th>
<th>New-style</th>
</tr>
</thead>
<tbody>
<tr>
<td>int f();</td>
<td>int f(int, int, double);</td>
</tr>
<tr>
<td>int f(a, b, c)</td>
<td>int f(int a, int b, double c)</td>
</tr>
<tr>
<td>int a, b;</td>
<td></td>
</tr>
<tr>
<td>double c;</td>
<td></td>
</tr>
<tr>
<td>{}</td>
<td></td>
</tr>
</tbody>
</table>

Data Definitions

type-specifier init-declarator-list;

declarator optional-initializer

Initializers may be constants or brace-enclosed, comma-separated constant expressions. Examples:

```c
int a;

struct { int x; int y; } b = {1, 2};

float a, *b, c;
```

Scope Rules

Two types of scope in C:

1. Lexical scope
   - Essentially, place where you don’t get “undeclared identifier” errors

2. Scope of external identifiers
   - When two identifiers in different files refer to the same object. E.g., a function defined in one file called from another.

Lexical Scope

Extends from declaration to terminating } or end-of-file.

```c
int a;

int foo()
{
    int b;
    if (a == 0) {
        printf("A was 0");
        a = 1;
    }
    b = a; /* OK */
}

int bar()
{
    a = 3; /* OK */
    b = 2; /* Error: b out of scope */
}
```

External Scope

```c
file1.c:
int foo()
{
    return 0;
}

int bar()
{
    extern int foo();
    foo(); /* OK */
}
```

```c
file2.c:
int baz()
{
    foo(); /* Error */
}

extern int foo();

int baff()
{
    foo(); /* OK */
}
```

The Preprocessor

Violates the free-form nature of C: preprocessor lines must begin with #.

Program text is passed through the preprocessor before entering the compiler proper.

Define replacement text:

```c
#define identifier token-string
```

Replace a line with the contents of a file:

```c
#include "filename"
```

C's Standard Libraries

```
<assert.h>
<ctype.h>
<errno.h>
<float.h>
<limits.h>
<locale.h>
<math.h>
<setjmp.h>
<signal.h>
<stdio.h>
<stdlib.h>
<string.h>
<time.h>
```

```
assert(a > 0)
isalpha(c)
errno
FLT_MAX
INT_MAX
setlocale(...) sin(x)
setjmp(jb)
signal(SIGINT, &f)
va_start(ap, st)
size_t
printf("%d", i)
malloc(1024)
strcmp(s1, s2)
localtime(tm)
```

Language design

Language design is library design.

— Bjarne Stroustrup

Programs consist of pieces connected together.

Big challenge in language design: making it easy to put pieces together correctly. C examples:

- The function abstraction (local variables, etc.)
- Type checking of function arguments
- The #include directive