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

Euclid's Algorithm in 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

Euclid on the PDP-11

```
.globl _gcd
.text
_gcd:
    jsr r5, rsave
L2: mov 4(r5), r1
     sxt r0
     div 6(r5), r0
     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
sign extend
r0, r1 = m ÷ n
if r == 0 goto L3
m = n
n = r
r0 = n
non-optimizing compiler
return r0 (n)
The Design of C

Taken from Dennis Ritchie's *C Reference Manual*
(Appendix A of Kernighan & Ritchie)

Lexical Conventions

C is a *free-form* language where whitespace mostly serves to separate tokens. Which of these are the same?

```
1+2  return this
1 + 2  returnthis
foo bar
foobars
```

Space is significant in some language. Python uses indentation for grouping, thus these are different:

```
if x < 3:
    y = 2
    z = 3
```
```
if x < 3:
    y = 2
    z = 3
```

What's in a Name?

In C, each name has a *storage class* (where it is) and a *type* (what it is).

```
Storage classes:  Fundamental types:  Derived types:
1. automatic  1. char  1. arrays
2. static  2. int  2. functions
3. external  3. float  3. pointers
4. register  4. double  4. structures
```

Convertions

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(r, r, ...)</code></td>
<td>Function call</td>
</tr>
<tr>
<td><code>a[i]</code></td>
<td>Array subscript</td>
</tr>
<tr>
<td><code>p-&gt;m</code></td>
<td>Member access</td>
</tr>
<tr>
<td><code>s.m</code></td>
<td>Member access</td>
</tr>
<tr>
<td><code>!b</code></td>
<td>Logical negation</td>
</tr>
<tr>
<td><code>¬i</code></td>
<td>Bitwise negation</td>
</tr>
<tr>
<td><code>l++</code></td>
<td>Increment</td>
</tr>
<tr>
<td><code>l--</code></td>
<td>Decrement</td>
</tr>
<tr>
<td><code>*p</code></td>
<td>Dereference</td>
</tr>
<tr>
<td><code>&amp;l</code></td>
<td>Address-of</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>i % j</code></td>
<td>Modulo</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>i &lt;&lt; j</code></td>
<td>Left shift</td>
</tr>
<tr>
<td><code>i &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</td>
</tr>
<tr>
<td><code>n &gt;= o</code></td>
<td>Greater than or equal</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><code>i ˆ j</code></td>
<td>Bitwise XOR</td>
</tr>
<tr>
<td>`i</td>
<td>j`</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>
</tbody>
</table>

Declarators

Declaration: string of specifiers followed by a declarator

- `int` (basic type)
- `static` (storage class specifier)
- `unsigned` (type specifier)
- `*` (pointer)
- `()` (function)
- `[]` (array)

Example:

```
static unsigned int (*f[10])(int, char*)[10];
```

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).

Type Specifiers

- `int`
- `char`
- `float`
- `double`
- `struct`
- `union`

Declarator 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”

Function definitions

```
type-specifier declarator ( parameter-list )

type-decl-list
{
  declaration-list
  statement-list
}

Example:
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:

Old-style                   New-style
int f();                   int f(int, int, double);
int f(a, b, c)             int f(int a, int b, double c)
int a, b;                  { double c;  }
struct { int x; int y; } b = { 1, 2 };
float a, *b, c;

Data Definitions

type-specifier init-declarator-list ;
deaclarator optional-initializer

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

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.

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

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

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

C’s Standard Libraries

<assert.h> Generate runtime errors
<ctype.h> Character classes
<errno.h> System error numbers
<float.h> Floating-point constants
<limits.h> Integer constants
<locale.h> Internationalization
<math.h> Math functions
<setjmp.h> Non-local goto
<signal.h> Signal handling
<stdio.h> Variable-length arguments
<stdlib.h> Some standard types
<stdio.h> File I/O, printing,
<string.h> Miscellaneous functions
	.Size
<string.h> String manipulation
	.Strcmp
<time.h> Time, date calculations

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:
#define identifier token-string
Replace a line with the contents of a file:
#include "filename"

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