Language Design

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

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

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

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

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

new style
declaration lists
number and type of
arguments.
Generated code did
care how many
arguments were
actually passed,
and everything was
a word.
Arguments are
call-by-value
Euclid on the PDP-11

```
.globl _gcd
.text
_gcd:
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 rreturn
```

Very natural mapping from C into PDP-11 instructions. Complex addressing modes make frame-pointer-relative accesses easy. Another idiosynrasy: registers were memory-mapped, so taking address of a variable in a register is straightforward.

Lexical Conventions

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

```
1+2
1 + 2
foo bar
foobar
```

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

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

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 (fo0), 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>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>f(r, r, ...)</td>
<td>1</td>
</tr>
<tr>
<td>a[i]</td>
<td>2</td>
</tr>
<tr>
<td>p-&gt;m</td>
<td>3</td>
</tr>
<tr>
<td>*=</td>
<td>4</td>
</tr>
<tr>
<td>&amp;</td>
<td>5</td>
</tr>
<tr>
<td>(type) r</td>
<td>6</td>
</tr>
<tr>
<td>sizeof(t)</td>
<td>7</td>
</tr>
<tr>
<td>n * o</td>
<td>8</td>
</tr>
<tr>
<td>n / o</td>
<td>8</td>
</tr>
<tr>
<td>n + o</td>
<td>8</td>
</tr>
<tr>
<td>n - o</td>
<td>8</td>
</tr>
<tr>
<td>i &lt; j</td>
<td>9</td>
</tr>
<tr>
<td>i &gt;&gt; j</td>
<td>9</td>
</tr>
<tr>
<td>n &gt; o</td>
<td>10</td>
</tr>
<tr>
<td>n &lt;= o</td>
<td>10</td>
</tr>
<tr>
<td>n &gt;= o</td>
<td>10</td>
</tr>
<tr>
<td>r = r</td>
<td>11</td>
</tr>
<tr>
<td>r != r</td>
<td>11</td>
</tr>
<tr>
<td>i &amp; j</td>
<td>12</td>
</tr>
<tr>
<td>i ^ j</td>
<td>12</td>
</tr>
<tr>
<td>i</td>
<td>j</td>
</tr>
<tr>
<td>b &amp; i &amp; c</td>
<td>14</td>
</tr>
<tr>
<td>b ? r : r</td>
<td>15</td>
</tr>
<tr>
<td>i = r</td>
<td>16</td>
</tr>
<tr>
<td>i += n</td>
<td>16</td>
</tr>
<tr>
<td>i -= n</td>
<td>16</td>
</tr>
<tr>
<td>i *= n</td>
<td>16</td>
</tr>
<tr>
<td>i /= n</td>
<td>16</td>
</tr>
<tr>
<td>i ^= i</td>
<td>17</td>
</tr>
<tr>
<td>i &amp;= i</td>
<td>17</td>
</tr>
<tr>
<td>i ^= i</td>
<td>17</td>
</tr>
<tr>
<td>i</td>
<td>= i</td>
</tr>
<tr>
<td>i &lt;&lt;= i</td>
<td>17</td>
</tr>
<tr>
<td>i &gt;&gt;= i</td>
<td>17</td>
</tr>
<tr>
<td>r1, r2</td>
<td>18</td>
</tr>
</tbody>
</table>

Declarators

Declaration: string of specifiers followed by a declarator

<table>
<thead>
<tr>
<th>Specifiers</th>
<th>Declarator</th>
</tr>
</thead>
<tbody>
<tr>
<td>basic type</td>
<td>int (*f[10])(int, char *)[10];</td>
</tr>
</tbody>
</table>

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-x operators (arrays, functions).

Declarator syntax

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

```c
type-specifier declarator ( parameter-list )
{
  declaration-list
  statement-list
}
```

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;</td>
<td>int a, b;</td>
</tr>
<tr>
<td>double c;</td>
<td>double c;</td>
</tr>
<tr>
<td>{}</td>
<td>{}</td>
</tr>
</tbody>
</table>

Data Definitions

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

```c
file2.c:
int baz()
{
    foo(); /* Error */
}
int bar()
{
    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:
```
#define identifier token-string
```

Replace a line with the contents of a file:
```
#include "filename"
```

C’s Standard Libraries

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

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