Language Design

COMS W4115

Katsushika Hokusai, *In the Hollow of a Wave off the Coast at Kanagawa*, 1827

Prof. Stephen A. Edwards
Fall 2003
Columbia University
Department of Computer Science
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
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
int gcd(int m, int n )
{
    int r;
    while ((r = m % n) != 0) {
        m = n;
        n = r;
    }
    return n;
}
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
Euclid on the PDP-11

.globl _gcd
.text
_gcd:
    jsr r5, rsave
L2:    mov 4(r5), r1
          r1 = n
    sxt r0
          sign extend
    div 6(r5), r0
          r0, r1 = m ÷ n
    mov r1, -10(r5)
          r = r1 (m % n)
    jeq L3
          if r == 0 goto L3
    mov 6(r5), 4(r5)
          m = n
    mov -10(r5), 6(r5)
          n = r
    jbr L2
L3:    mov 6(r5), r0
          r0 = n
    jbr L1
          non-optimizing compiler
L1:    jmp rretrn
          return r0 (n)
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
    jbr L1
L1:  jmp rretrn

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

```python
if x < 3:
    y = 2
    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:  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
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
\[ 1 \times 2 + 3 \times 4 \] means
\[ (1 \times 2) + (3 \times 4) \]

Associativity tells us
\[ 1 + 2 + 3 + 4 \] means
\[ ((1 + 2) + 3) + 4 \]
# C’s Operators in Precedence Order

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>f(r,r,...)</code></td>
<td><code>a[i]</code></td>
<td><code>p-&gt;m</code></td>
<td><code>s.m</code></td>
<td></td>
</tr>
<tr>
<td><code>!b</code></td>
<td><code>~i</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>++l</code></td>
<td><code>--l</code></td>
<td><code>l++</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>*p</code></td>
<td><code>&amp;l</code></td>
<td><code>(type) r</code></td>
<td><code>sizeof(t)</code></td>
<td></td>
</tr>
<tr>
<td><code>n * o</code></td>
<td><code>n / o</code></td>
<td><code>i % j</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>n + o</code></td>
<td></td>
<td><code>n - o</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>i &lt;&lt; j</code></td>
<td><code>i &gt;&gt; j</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>n &lt; o</code></td>
<td><code>n &gt; o</code></td>
<td><code>n &lt;= o</code></td>
<td><code>n &gt;= o</code></td>
<td></td>
</tr>
<tr>
<td><code>r == r</code></td>
<td></td>
<td><code>r != r</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>i &amp; j</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>i ^ j</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>`i</td>
<td>j`</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>b &amp;&amp; c</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>`b</td>
<td></td>
<td>c`</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>b ? r : r</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>l = r</code></td>
<td><code>l += n</code></td>
<td><code>l -= n</code></td>
<td><code>l * = n</code></td>
<td></td>
</tr>
<tr>
<td><code>l /= n</code></td>
<td><code>l %= i</code></td>
<td><code>l &amp;= i</code></td>
<td><code>l ^= i</code></td>
<td></td>
</tr>
<tr>
<td>`l</td>
<td>= i`</td>
<td><code>l &lt;&lt;= i</code></td>
<td><code>l &gt;&gt;= i</code></td>
<td></td>
</tr>
<tr>
<td><code>r1 , r2</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Declarators

Declaration: string of specifiers followed by a declarator

```
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).
Storage-Class Specifiers

- **auto**: Automatic (stacked), default
- **static**: Statically allocated
- **extern**: Look for a declaration elsewhere
- **register**: Kept in a register, not memory

C trivia: Originally, a function could only have at most three `register` variables, may only be `int` or `char`, can’t use address-of operator `&`.

Today, `register` simply ignored. Compilers try to put most automatic variables in registers.
Type Specifiers

int
char
float
double

struct { declarations }

struct identifier { declarations }

struct identifier
Declarators

*identifier*  
( *declarator*)  
*declarator ()*  
*declarator [ optional-constant ]*  
* *declarator*  
Grouping  
Function  
Array  
Pointer

C trivia: Originally, number and type of arguments to a function wasn’t part of its type, thus declarator just contained ().

Today, ANSI C allows function and argument types, making an even bigger mess of declarators.
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
```
Statements

expression ;
{ statement-list }
if ( expression ) statement else statement
while ( expression ) statement
do statement while ( expression );
for ( expression ; expression ; expression ) statement
switch ( expression ) statement
case constant-expression :
default :
break ;
continue ;
return expression ;
goto label ;
label :
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

```c
int f();
int f(a, b, c)
int a, b;
double c;
{
}
```

New-style

```c
int f(int, int, double);
int f(int a, int b, double 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 */
}
```
file1.c:
int foo()
{
    return 0;
}

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

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:

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

Replace a line with the contents of a file:

```
#include "filename"
```
C’s Standard Libraries

<table>
<thead>
<tr>
<th>Header File</th>
<th>Description</th>
<th>Example Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;assert.h&gt;</code></td>
<td>Generate runtime errors</td>
<td><code>assert(a &gt; 0)</code></td>
</tr>
<tr>
<td><code>&lt;ctype.h&gt;</code></td>
<td>Character classes</td>
<td><code>isalpha(c)</code></td>
</tr>
<tr>
<td><code>&lt;errno.h&gt;</code></td>
<td>System error numbers</td>
<td><code>errno</code></td>
</tr>
<tr>
<td><code>&lt;float.h&gt;</code></td>
<td>Floating-point constants</td>
<td><code>FLT_MAX</code></td>
</tr>
<tr>
<td><code>&lt;limits.h&gt;</code></td>
<td>Integer constants</td>
<td><code>INT_MAX</code></td>
</tr>
<tr>
<td><code>&lt;locale.h&gt;</code></td>
<td>Internationalization</td>
<td><code>setlocale(...)</code></td>
</tr>
<tr>
<td><code>&lt;math.h&gt;</code></td>
<td>Math functions</td>
<td><code>sin(x)</code></td>
</tr>
<tr>
<td><code>&lt;setjmp.h&gt;</code></td>
<td>Non-local goto</td>
<td><code>setjmp(jb)</code></td>
</tr>
<tr>
<td><code>&lt;signal.h&gt;</code></td>
<td>Signal handling</td>
<td><code>signal(SIGINT,&amp;f)</code></td>
</tr>
<tr>
<td><code>&lt;stdarg.h&gt;</code></td>
<td>Variable-length arguments</td>
<td><code>va_start(ap, st)</code></td>
</tr>
<tr>
<td><code>&lt;stddef.h&gt;</code></td>
<td>Some standard types</td>
<td><code>size_t</code></td>
</tr>
<tr>
<td><code>&lt;stdio.h&gt;</code></td>
<td>File I/O, printing</td>
<td><code>printf(&quot;%d&quot;, i)</code></td>
</tr>
<tr>
<td><code>&lt;stdlib.h&gt;</code></td>
<td>Miscellaneous functions</td>
<td><code>malloc(1024)</code></td>
</tr>
<tr>
<td><code>&lt;string.h&gt;</code></td>
<td>String manipulation</td>
<td><code>strcmp(s1, s2)</code></td>
</tr>
<tr>
<td><code>&lt;time.h&gt;</code></td>
<td>Time, date calculations</td>
<td><code>localtime(tm)</code></td>
</tr>
</tbody>
</table>
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