Language Design

COMS W4115

Prof. Stephen A. Edwards
Spring 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

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
```

```
return this
returnthis
```

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
```

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 **`double`**s; **`float`**s 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>
<tr><td>f(r,r,...)</td><td>a[i]</td><td>p->m</td><td>s.m</td></tr>
<tr><td>!b</td><td>~i</td><td>-i</td><td></td></tr>
<tr><td>++l</td><td>--l</td><td>l++</td><td>l--</td></tr>
<tr><td>*p</td><td>&l</td><td>(type) r</td><td>sizeof(t)</td></tr>
<tr><td>n * o</td><td>n / o</td><td>i % j</td><td></td></tr>
<tr><td>n + o</td><td>n - o</td><td></td><td></td></tr>
<tr><td>i << j</td><td>i >> j</td><td></td><td></td></tr>
<tr><td>n < o</td><td>n > o</td><td>n <= o</td><td>n >= o</td></tr>
<tr><td>r == r</td><td>r != r</td><td></td><td></td></tr>
<tr><td>i & j</td><td></td><td></td><td></td></tr>
<tr><td>i ^ j</td><td></td><td></td><td></td></tr>
<tr><td>i | j</td><td></td><td></td><td></td></tr>
<tr><td>b && c</td><td></td><td></td><td></td></tr>
<tr><td>b || c</td><td></td><td></td><td></td></tr>
<tr><td>b ? r : r</td><td></td><td></td><td></td></tr>
<tr><td>l = r</td><td>l += n</td><td>l -= n</td><td>l *= n</td></tr>
<tr><td>l /= n</td><td>l %= i</td><td>l &= i</td><td>l ^= i</td></tr>
<tr><td>l |= i</td><td>l <<= i</td><td>l >>= i</td><td></td></tr>
<tr><td>r1 , r2</td><td></td><td></td><td></td></tr>
</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).

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

```
int f();

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

New-style

```
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:

```
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()
{
  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

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