The C Programming Language

Part 2

Agenda

- Data Types
- Operators
- Statements
- I/O Facilities

Operators

Computers represent integers as bits
Arithmetic operations: +, -, *, /, etc.
Bit operations: and, or, xor, shift, etc.

Typical language design (1970s): provide abstraction so that one does not confuse integers with their representation

The C language design: no abstraction, revel in the “pun” between integers and their representation

Decisions

- Provide typical arithmetic operators: +, -, *, /, %
- Provide typical relational operators: == != < <= > >=
- Each evaluates to 0 ⇒ FALSE or 1 ⇒ TRUE
- Provide typical logical operators: ! && ||
- Each interprets 0 ⇒ FALSE, ≠0 ⇒ TRUE
- Each evaluates to 0 ⇒ FALSE or 1 ⇒ TRUE
- Provide bitwise operators: ~ & | ^ >> <<
- Provide a cast operator: (type)

Logical NOT (!) vs. bitwise NOT (~)

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>! 1 (TRUE) ⇒ 0 (FALSE)</td>
<td>! 1 00000000 00000000 00000000 00000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>&amp; 1 (TRUE) ⇒ -2 (TRUE)</td>
<td>2 00000000 00000000 00000000 00000000</td>
</tr>
</tbody>
</table>

Implication:

- Use logical NOT to control flow of logic
- Use bitwise NOT only when doing bit-level manipulation

Aside: Logical vs. Bitwise Ops

Logical AND (&&) vs. bitwise AND (&)

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>1</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>2 (TRUE) &amp;&amp; 1 (TRUE) ⇒ 1 (TRUE)</td>
<td>2 00000000 00000000 00000000 00000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>&amp; 1 (TRUE) ⇒ 0 (FALSE)</td>
<td>2 00000000 00000000 00000000 00000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>1</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>2 (TRUE) &amp; 1 (TRUE) ⇒ 0 (FALSE)</td>
<td>2 00000000 00000000 00000000 00000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>1</td>
<td>00000000 00000000 00000000 00000000</td>
</tr>
<tr>
<td>2 (TRUE) &amp; 1 (TRUE) ⇒ 0 (FALSE)</td>
<td>2 00000000 00000000 00000000 00000000</td>
</tr>
</tbody>
</table>
Aside: Logical vs. Bitwise Ops

Implication:
- Use logical AND to control flow of logic
- Use bitwise AND only when doing bit-level manipulation

Same for logical OR (||) and bitwise OR (|)

Assignment Operator

Typical programming language of 1970s:
Statements, Expressions
stmt :=
  exp :
  | if exp then stmt else stmt
  | while exp do stmt
  | begin stmtlist end
stmtlist ::= stmt | stmtlist ; stmt
exp :=
id | exp+exp | exp-exp | -exp
  | (exp) | ...

C language: assignment is an expression!
stmt :=
  exp :
  | { stmtlist }
  | if (exp) stmt else stmt
  | while (exp) stmt
stmtlist ::= stmt | stmtlist ; stmt
exp :=
id | exp+exp | exp-exp | -exp
  | id=exp | exp.exp | exp?exp:exp
  | (exp) | ...

Assignment Operator Examples

Examples
i = 0;
  /* Side effect: assign 0 to i.
     Evaluate to 0.
  */
j = i = 0; /* Assignment op has R to L associativity */
  /* Side effect: assign 0 to i.
     Evaluate to 0.
     Side effect: assign 0 to j.
     Evaluate to 0. */
while ((i = getchar()) != EOF) ...
  /* Read a character.
     Side effect: assign that character to i.
     Evaluate to that character.
     Compare that character to EOF.
     Evaluate to 0 (FALSE) or 1 (TRUE). */

Special-Purpose Assignment Operators

Decisions
- Provide special-purpose assignment operators:
  +-= -= *= /= ~= &= |= ^= <<= >>=

Examples
i += j  same as i = i + j
i /= j  same as i = i / j
i |= j  same as i = i | j
i >>= j same as i = i >> j

Design decision
- Is it worth mucking up the language definition with this feature?
  Does it really make programs any faster, or easier to read?

Examples
p->data[i+j*10].first->next += 1;
Special-Purpose Assignment Operators

Increment and decrement operators: ++ --
- Prefix and postfix forms

Examples

(1) i = 5;
   j = ++i;
(2) i = 5;
   j = i++;
(3) i = 5;
   j = ++i + ++i;
(4) i = 5;
   j = i++ + i++;

What is the value of i? Of j?

Memory allocation

Typical programming language of 1970s:
- Special program statement to allocate a new object
  stmt ::= new p
  This is not so different from Java's 
  p = new(MyClass)

Difficulties:
1. System standard allocator could be slow, or inflexible
2. What about deallocation?
   - Explicit "free" leads to bugs
   - Automatic garbage collection too expensive?

C language
Nothing built-in
- malloc, free functions provided in standard library
- allow programmers to roll their own allocation systems

Difficulties:
1. System standard allocator could be slow, or inflexible
   (but that's mitigated by roll-your-own)
2. Explicit "free" leads to bugs
   - Turns out, by now we know, automatic garbage collection isn't too expensive after all

SIZE OF OPERATOR

Malloc function needs to be told how many bytes to allocate
struct foo {int a, b; float c;} *p;
stmt ::= new p
This is not so different from Java's 
p = new(MyClass)

Issue: How can programmers determine data sizes?
Rationale:
- The sizes of most primitive types are unspecified
- Sometimes programmer must know sizes of primitive types
  - E.g. when allocating memory dynamically
- Hard code data sizes ⇒ program not portable
- C must provide a way to determine the size of a given data type programmatically

SIZE OF OPERATOR

Decisions
- Provide a sizeof operator
  - Applied at compile-time
  - Operand can be a data type
  - Operand can be an expression
  - Compiler infers a data type

Examples, on CourseLab
- sizeof(int) ⇒ 4
- When i is a variable of type int...
  - sizeof(i) ⇒ 4
  - sizeof(i++) ⇒ sizeof(i) + sizeof(void)
  - sizeof(i++) + sizeof(i) - 5

What is the value?

OTHER OPERATORS

Issue: What other operators should C have?

Decisions
- Function call operator
  - Should mimic the familiar mathematical notation
    - function(arg1, arg2, ..)
- Conditional operator: ?:
  - The only ternary operator
  - See King book
  - Sequence operator: ,
  - See King book
  - Pointer-related operators: & *
    - Described later in the course
  - Structure-related operators: . ,
    - Described later in the course

Operators Summary: C vs. Java

Java only
- >>>
  - right shift with zero fill
- new
  - create an object
- instanceof
  - is left operand an object of class right operand?
- p.f
  - object field select

C only
- p.f
  - structure field select
- *
  - dereference
- p->f
  - dereference then structure member select: (*p).f
- 
  - address of
- 
  - sequence
- sizeof
  - compile-time size of
Operators Summary: C vs. Java

Related to type boolean:
- Java: Relational and logical operators evaluate to type boolean
- C: Relational and logical operators evaluate to type int
- Java: Logical operators take operands of type boolean
- C: Logical operators take operands of any primitive type or memory address

Agenda

Data Types
 Operators
 Statements
 I/O Facilities

Sequence Statement

Issue: How should C implement sequence?
Decision
- Compound statement, alias block

Selection Statements

Issue: How should C implement selection?
Decisions
- if statement, for one-path, two-path decisions

Selection Statements

Decisions (cont.)
- switch and break statements, for multi-path decisions on a single integerExpr

Repetition Statements

Issue: How should C implement repetition?
Decisions
- while statement; test at leading edge
- for statement; test at leading edge, increment at trailing edge
- do...while statement; test at trailing edge

0 \Rightarrow \text{FALSE}
non-0 \Rightarrow \text{TRUE}
Declaring Variables

**Issue:** Should C require variable declarations?

**Rationale:**
- Declaring variables allows compiler to check spelling (compile-time error messages are easier for programmer than debugging strange behavior at run time!)
- Declaring variables allows compiler to allocate memory more efficiently

Where are variables declared?

**Typical 1960s language:**
- Global variables

**Typical 1970s language:**
- Global variables
- Local variables declared just before function body

**C language:**
- Global variables
- Local variables can be declared at beginning of any (block), e.g.,

```
{int i=6, j;
j=7;
if (i>j)
  [int x; x=i+j; return x;]
else {int y; y=i-j; return y;}
}  // scope of variable y ends
```

Repetition Statements

**Decisions (cont.)**
- Cannot declare loop control variable in for statement

```
{   for (int i = 0; i < 10; i++)
    /* Do something */
}  // Illegal in C (nobody thought of that idea in 1970s)
```

**Decisions:**
- Require variable declarations
- Provide declaration statement
- Programmer specifies type of variable (and other attributes too)

**Examples**
- int i;
- int i, j;
- int i = 5;
- const int i = 5; /* value of i cannot change */
- static int i; /* covered later in course */
- extern int i; /* covered later in course */

Other Control Statements

**Issue:** What other control statements should C provide?

**Decisions**
- **break** statement (revisited)
  - Breaks out of closest enclosing switch or repetition statement
- **continue** statement
  - Skips remainder of current loop iteration
  - Continues with next loop iteration
  - When used within for, still executes incrementExpr
- **goto** statement
  - Jump to specified label

```
{  int i;
  int j;
  /* Non-declaration stmts that use i. */
  i = i+1;
  int j;
  /* Non-declaration stmts that use j. */
  i = i+1;
  /* Non-declaration stmts that use i. */
  j = j+1;
}
```  // Illegal in C (nobody thought of that idea in 1970s)

```
{  int i;
  int j;
  /* Non-declaration stmts that use i. */
  i = i+1;
  /* Non-declaration stmts that use j. */
  i = i+1;
  /* Non-declaration stmts that use i. */
  j = j+1;
}
```  // Legal in C
Computing with Expressions

### Issue: How should C implement computing with expressions?

**Decisions:**
- Provide **expression statement**
  
  ```c
  expression ;
  ```

### Examples

```c
i = 5;  // Side effect: assign 5 to i. Evaluate to 5. Discard the 5. */

j = i + 1;  // Side effect: assign 6 to j. Evaluate to 6. Discard the 6. */

printf("hello");  // Side effect: print hello. Evaluate to 5. Discard the 5. */

i + 1;  // Evaluate to 6. Discard the 6. */

5;  // Evaluate to 5. Discard the 5. */
```

### Statements Summary: C vs. Java

#### Declaration statement:
- **Java**: Compile-time error to use a local variable before specifying its value
- **C**: Run-time error to use a local variable before specifying its value

#### Final and Const
- **Java**: Has `final` variables
- **C**: Has `const` variables

#### Expression statement
- **Java**: Only expressions that have a side effect can be made into expression statements
- **C**: Any expression can be made into an expression statement

#### Compound statement:
- **Java**: Declarations statements can be placed anywhere within compound statement
- **C**: Declaration statements must appear before any other type of statement within compound statement

#### If statement
- **Java**: Controlling `expr` must be of type `boolean`
- **C**: Controlling `expr` can be any primitive type or a memory address ( 0 → FALSE, non-0 → TRUE)

#### While statement
- **Java**: Controlling `expr` must be of type `boolean`
- **C**: Controlling `expr` can be any primitive type or a memory address ( 0 → FALSE, non-0 → TRUE)

#### Do-While statement
- **Java**: Controlling `expr` must be of type `boolean`
- **C**: Controlling `expr` can be of any primitive type or a memory address ( 0 → FALSE, non-0 → TRUE)

#### For statement
- **Java**: Controlling `expr` must be of type `boolean`
- **C**: Controlling `expr` can be of any primitive type or a memory address ( 0 → FALSE, non-0 → TRUE)

#### Loop control variable
- **Java**: Can declare loop control variable in `initexpr`
- **C**: Cannot declare loop control variable in `initexpr`

#### Break statement
- **Java**: Also has "labeled break" statement
- **C**: Does not have "labeled break" statement

#### Continue statement
- **Java**: Also has "labeled continue" statement
- **C**: Does not have "labeled continue" statement

#### Goto statement
- **Java**: Not provided
- **C**: Provided (but don’t use it!)
I/O Facilities

Issue: Should C provide I/O facilities?
(many languages of the 1960s / 1970s had built-in special-purpose commands for input/output)

Thought process
• Unix provides the file abstraction
  • A file is a sequence of characters with an indication of the current position
  • Unix provides 3 standard files
    • Standard input, standard output, standard error
  • C should be able to use those files, and others
  • I/O facilities are complex
  • C should be small/simple

Decisions
• Do not provide I/O facilities in the language
  • Instead provide I/O facilities in standard library
    • Constant: EOF
    • Data type: FILE (described later in course)
    • Variables: stdin, stdout, and stderr
    • Functions: ...

Reading Characters

Issue: What functions should C provide for reading characters?

Thought process
• Need function to read a single character from stdin
  • ... And indicate failure

Decisions
• Provide getchar() function*
  • Define getchar() to return EOF upon failure
    • EOF is a special non-character int
  • Make return type of getchar() wider than char
    • Make it int; that’s the natural word size

Reminder
• There is no such thing as “the EOF character”

*actually, a macro...

Writing Characters

Issue: What functions should C provide for writing characters?

Thought process
• Need function to write a single character to stdout

Decisions
• Provide putchar() function
  • Define putchar() to have int parameter
  • For symmetry with getchar()
**Reading Other Data Types**

**Issue:** What functions should C provide for reading data of other primitive types?

**Thought process**
- Must convert external form (sequence of character codes) to internal form
- Could provide `getshort()`, `getint()`, `getfloat()`, etc.
- Could provide parameterized function to read any primitive type of data

**Decisions**
- Provide `scanf()` function
  - Can read any primitive type of data
  - First parameter is a format string containing conversion specifications

**Writing Other Data Types**

**Issue:** What functions should C provide for writing data of other primitive types?

**Thought process**
- Must convert internal form to external form (sequence of character codes)
- Could provide `putshort()`, `putint()`, `putfloat()`, etc.
- Could provide parameterized function to write any primitive type of data

**Decisions**
- Provide `printf()` function
  - Can write any primitive type of data
  - First parameter is a format string containing conversion specifications
Other I/O Facilities

Issue: What other I/O functions should C provide?

Decisions

• fopen(): Open a stream
• fclose(): Close a stream
• fgetc(): Read a character from specified stream
• fputs(): Write a character to specified stream
• gets(): Read a line from stdin. Brain-damaged, never use this!
• fgets(): Read a line/string from specified stream
• fprintf(): Write data to specified stream

Described in King book, and later in the course after covering files, arrays, and strings

Summary

C design decisions and the goals that affected them

• Data types
• Operators
• Statements
• I/O facilities

Knowing the design goals and how they affected the design decisions can yield a rich understanding of C

Appendix: The Cast Operator

Cast operator has multiple meanings:

(1) Cast between integer type and floating point type:
  • Compiler generates code
  • At run-time, code performs conversion

\[
\begin{align*}
  f &= \text{11000000000000000000000000000000} \\
  \text{11111111111111111111111111111110} \\
  &\text{−27.375} \\
  i &= (\text{int})f \\
  \text{11111111111111111111111111111110} \\
  &\text{−27}
\end{align*}
\]

(2) Cast between floating point types of different sizes:
  • Compiler generates code
  • At run-time, code performs conversion

\[
\begin{align*}
  f &= \text{11000000000000000000000000000000} \\
  \text{11000000000000000000000000000000} \\
  &\text{−27.375} \\
  d &= (\text{double})f \\
  \text{11000000000000000000000000000000} \\
  \text{11000000000000000000000000000000} \\
  &\text{−27.375}
\end{align*}
\]

(3) Cast between integer types of different sizes:
  • Compiler generates code
  • At run-time, code performs conversion

\[
\begin{align*}
  f &= \text{11000000000000000000000000000000} \\
  \text{00000000000000000000000000000000} \\
  &\text{−27.375} \\
  i &= (\text{char})i \\
  \text{2} \\
  c &= (\text{char})i \\
  \text{2}
\end{align*}
\]

(4) Cast between integer types of same size:
  • Compiler generates no code
  • Compiler views given bit-pattern in a different way

\[
\begin{align*}
  f &= \text{11000000000000000000000000000000} \\
  \text{11000000000000000000000000000000} \\
  &\text{−27.375} \\
  u &= (\text{unsigned int})i \\
  \text{11111111111111111111111111111110} \\
  &\text{4294967294}
\end{align*}
\]