The C Programming Language

Part 2

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

Aside: Logical vs. Bitwise Ops

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 00000001</td>
</tr>
<tr>
<td>~ 1 (TRUE) ⇒ ~2 (TRUE)</td>
<td></td>
</tr>
</tbody>
</table>

<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 00000000 00000000 00000000 00000001</td>
<td></td>
</tr>
<tr>
<td>! 1 (TRUE) ⇒ 0 (FALSE)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>00000000 00000000 00000000 00000010</td>
</tr>
<tr>
<td>2 (TRUE) &amp; 1 (TRUE) ⇒ 0 (FALSE)</td>
<td></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 00000010</td>
</tr>
<tr>
<td>2 (TRUE) &amp; 1 (TRUE) ⇒ 0 (FALSE)</td>
<td></td>
</tr>
</tbody>
</table>

<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 00000000 00000000 00000000 00000000</td>
<td></td>
</tr>
</tbody>
</table>

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 00000010</td>
</tr>
<tr>
<td>2 (TRUE) &amp; 1 (TRUE) ⇒ 0 (FALSE)</td>
<td></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

Decisions
• Provide assignment operator =
  • Side effect: changes the value of a variable
  • Evaluates to the new value of the variable

Examples

Assignment Operator Examples

Examples

Special-Purpose Assignment Operators

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

Examples

Special-Purpose Assignment Operators

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

Examples
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

Sizeof Operator

Sizeof function needs to be told how many bytes to allocate
struct foo { int a, b; float c; } *p;
p = malloc(12); /* this is correct but not portable */

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

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

Sizeof 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+1)
  • sizeof(i++ * ++i - 5)

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: & *
  • Address of, dereference (described in precepts)
• Structure-related operators: . ->
  • Structure field select (described in precepts)

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

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 runtime)
- 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=0, j;
  if (i>7) {
    int x; x+=i; return x;
  } else {
    int y; y+=i; return y;
  }
  ```
  Scope of variable `y` ends at matching close brace

Repetition Statements

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

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

```c
int i;
for (i = 0; i < 10; i++)
  /* Do something */
```
Legal in C

Declaring Variables

Decisions (cont.):
- Declaration statements must appear before any other kind of statement in compound statement

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

```c
int i;
int j;
/* Non-declaration statements that use i, */
/* Non-declaration statements that use j. */
```
Legal in C

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

Computing with Expressions

Issue: How should C implement computing with expressions?

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

Computing with Expressions

Examples

```c
int 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

Statements Summary: C vs. Java

**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)

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

Statements Summary: C vs. Java

**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!)

Agenda

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

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

Reading Other Data Types

```
123
```

```
00000000000000000000000001111011
```

```
scanf("%d", &i);
```

```
123
```

See King book for 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

Writing Other Data Types

```
123
```

```
011000010110001001100011
```

```
printf("%d", i);
```

```
011000010110001001100011
```

See King book for 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
- `fputc()`: Write a character to specified stream
- `fgets()`: Read a line from stdin. Brain-damaged, never use this!
- `fputs()`: Write a line to specified stream
- `fscanf()`: Read data 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

  \[ f = 11000011101100000000000000 \]
  \[ i = (\text{int})f \]
  \[ i = 11111111111111111111111111100101 \]

\[ i \rightarrow -27.375 \]

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

  \[ f = 11000011101100000000000000 \]
  \[ d = (\text{double})f \]
  \[ d = 11000000001110110110000000000000 \]

\[ d \rightarrow -27.375 \]

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

  \[ i = 00000000000000000000000000000010 \]
  \[ c = (\text{char})i \]
  \[ c = 00000010 \]

\[ c \rightarrow 2 \]

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

  \[ i = 1111111111111111111111111111 \]
  \[ u = (\text{unsigned int})i \]
  \[ u = 4294967294 \]

\[ u \rightarrow -2 \]