The Design of C: A Rational Reconstruction: Part 2
Continued from previous lecture
Agenda

Data Types
Operators
Statements
I/O Facilities
Issue: What kinds of operators should C have?

Thought process
- Should handle typical operations
- Should handle bit-level programming ("bit twiddling")
- Should provide a mechanism for converting from one type to another
Operators

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, non-0 => TRUE
  - Each evaluates to 0 => FALSE or 1 => TRUE
- Provide bitwise operators: ~ & | ^ >> <<
- Provide a cast operator: (type)
Aside: Logical vs. Bitwise Ops

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

• ! 1 (TRUE) => 0 (FALSE)

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

• ~ 1 (TRUE) => -2 (TRUE)

<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</td>
<td>11111111 11111111 11111111 11111110</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 (&)

- \(2 \text{ (TRUE)} \&& 1 \text{ (TRUE)} \Rightarrow 1 \text{ (TRUE)}\)

<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>&amp;&amp; 1</td>
<td>00000000 00000000 00000000 00000001</td>
</tr>
<tr>
<td>----</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>1</td>
<td>00000000 00000000 00000000 00000001</td>
</tr>
</tbody>
</table>

- \(2 \text{ (TRUE)} \& 1 \text{ (TRUE)} \Rightarrow 0 \text{ (FALSE)}\)

<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>&amp; 1</td>
<td>00000000 00000000 00000000 00000001</td>
</tr>
<tr>
<td>----</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>0</td>
<td>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

Issue: What about assignment?

Thought process

- Must have a way to assign a value to a variable
- Many high-level languages provide an assignment `statement`
- Would be more succinct to define an assignment `operator`
  - Performs assignment, and then evaluates to the assigned value
  - Allows assignment expression to appear within larger expressions
Assignment Operator

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

Examples

```c
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). */
```
Issue: Should C provide special-purpose assignment operators?

Thought process

• The construct \( i = i + 1 \) is common
• More generally, \( i = i + n \) and \( i = i \times n \) are common
• Special-purpose assignment operators would make code more compact
• Such operators would complicate the language and compiler
Special-Purpose Assignment Operators

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

Examples

\[
\begin{align*}
  i &= j \quad &\text{same as } i &= i + j \\
  i &= j \quad &\text{same as } i &= i / j \\
  i &= j \quad &\text{same as } i &= i | j \\
  i &= j \quad &\text{same as } i &= i >> j \\
\end{align*}
\]
Decisions (cont.)

- Provide 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`?
Sizeof Operator

**Issue:** How can programmers determine data sizes?

**Thought process**
- 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
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 nobel using gcc217
- `sizeof(int) => 4`

- When `i` is a variable of type int...
  - `sizeof(i) => 4`
  - `sizeof(i+1)`
  - `sizeof(i++ * ++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?

C only
- `->` structure member select
- `*` dereference
- `&` address of
- `,` sequence
- `sizeof` compile-time size of
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

```c
{ 
    statement1;
    statement2;
    ...
}
```
Selection Statements

Issue: How should C implement selection?

Decisions

- `if` statement, for one-path, two-path decisions

```c
if (expr)
    statement1;
else
    statement2;
```

0 => FALSE
non-0 => TRUE
Decisions (cont.)

• `switch` and `break` statements, for multi-path decisions on a single `integerExpr`

```java
switch (integerExpr)
{
  case integerLiteral1:
    ...
    break;
  case integerLiteral2:
    ...
    break;
  ...
  default:
    ...
}
```

What happens if you forget `break`?
Repetition Statements

Issue: How should C implement repetition?

Decisions

• **while** statement; test at leading edge

  ```c
  while (expr)
  
  statement;
  ```

• **for** statement; test at leading edge, increment at trailing edge

  ```c
  for (initialExpr; testExpr; incrementExpr)
  
  statement;
  ```

• **do...while** statement; test at trailing edge

  ```c
  do
  
  statement;
  
  while (expr);
  ```

0 => FALSE
non-0 => TRUE
Decisions (cont.)

- Cannot declare loop control variable in `for` statement

```c
{  
    ...  
    for (int i = 0; i < 10; i++)  
    /* Do something */  
    ...  
}  

{  
    int i;  
    ...  
    for (i = 0; i < 10; i++)  
    /* Do something */  
    ...  
}
```

Illegal in C

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**
Issue: Should C require variable declarations?

Thought process:
• Declaring variables allows compiler to check spelling
• Declaring variables allows compiler to allocate memory more efficiently
Declaring Variables

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 */`
 Declaring Variables

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

```c
{  
    int i;
    /* Non-declaration
        stmts that use i. */
    ...
    int j;
    /* Non-declaration
        stmts that use j. */
    ...
}
```

Illegal in C

```c
{  
    int i;
    int j;
    ...
    /* Non-declaration
        stmts that use i. */
    ...
    /* Non-declaration
        stmts that use j. */
    ...
}
```

Legal in C
Computing with Expressions

Issue: How should C implement computing with expressions?

Decisions:
• Provide expression statement
  expression ;
Computing with Expressions

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
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`)
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`
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
Issue: Should C provide I/O facilities?

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
I/O Facilities

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: …
**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”
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()`
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

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

See King book for conversion specifications

What is this ampersand? Covered later in course.
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

Decisions

• Provide `printf()` function
  • Can write any primitive type of data
  • First parameter is a *format string* containing *conversion specifications*
Writing Other Data Types

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

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/string from specified stream
- `fputs()`: Write a line/string 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

\[
\text{f} = \begin{array} {c}
11000001110110110000000000000000 \\
\end{array} = -27.375
\]

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

\[
\text{i} = \begin{array} {c}
111111111111111111111111100101 \\
\end{array} = -27
\]
Appendix: The Cast Operator

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

\[
\begin{align*}
\text{f} &= 11000001110110110000000000000000 \\
\text{d} &= (\text{double})\text{f} \\
\text{d} &= 110000000011011011000000000000000 \\
\end{align*}
\]

\(-27.375\)
Appendix: The Cast Operator

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

\[
\begin{array}{c}
i \quad 00000000000000000000000000000010 \\
c = \text{(char)}i \\
c \quad 00000010
\end{array}
\]
Appendix: The Cast Operator

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

\[
i = 11111111111111111111111111111110 \quad \text{−2}
\]

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

\[
u = 11111111111111111111111111111110 \quad 4294967294
\]