The Design of C

“C is quirky, flawed, and an enormous success. While accidents of history surely helped, it evidently satisfied a need for a system implementation language efficient enough to displace assembly language, yet sufficiently abstract and fluent to describe algorithms and interactions in a wide variety of environments.”

-- Dennis Ritchie
Goals of this Lecture

Help you learn about:

• The decisions that were made by the designers* of C
• Why they made those decisions
  … and thereby…
• The fundamentals of C

Why?

• Learning the design rationale of the C language provides a richer understanding of C itself
• A power programmer knows both the programming language and its design rationale

* Dennis Ritchie & members of standardization committees
## Goals of C

<table>
<thead>
<tr>
<th>Designers wanted C to:</th>
<th>But also:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support system programming</td>
<td>Support application programming</td>
</tr>
<tr>
<td>Be low-level</td>
<td>Be portable</td>
</tr>
<tr>
<td>Be easy for people to handle</td>
<td>Be easy for computers to handle</td>
</tr>
</tbody>
</table>

- Conflicting goals on multiple dimensions!
- Result: different design decisions than Java
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, 1 ⇒ TRUE
• Provide typical logical operators: ! && ||
  • Each interprets 0 ⇒ FALSE, non-0 ⇒ TRUE
  • Each evaluates to 0 ⇒ FALSE, 1 ⇒ TRUE
• Provide bitwise operators: ~ & | ^ >> <<
• Provide a cast operator: (type)
Logical vs. Bitwise Ops

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

• 2 (TRUE) && 1 (TRUE) => 1 (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 (TRUE) & 1 (TRUE) => 0 (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>

Implication:

• Use **logical** AND to control flow of logic
• Use **bitwise** AND only when doing bit-level manipulation
• Same for OR and NOT
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 expressive to define an assignment operator
  • Performs assignment, and then evaluates to the assigned value
  • Allows assignment to appear within larger expressions

Decisions

• Provide assignment operator: =
• Define assignment operator so it changes the value of a variable, and also evaluates to that value
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). */
```
Special-Purpose Assignment

Issue: Should C provide tailored assignment operators?

Thought process

- The construct `a = b + c` is flexible
- The construct `i = i + c` is somewhat common
- The construct `i = i + 1` is very common
- Special-purpose operators make code more expressive
  - Might reduce some errors
  - May complicate the language and compiler

Decisions

- Introduce `+=` operator to do things like `i += c`
- Extend to `-= *= /= ~= &= |= ^= <<= >>=`
- Special-case increment and decrement: `i++ i--`
- Provide both pre- and post-inc/dec: `x = ++i; y = i++;`
iClicker Question

Q: What are $i$ and $j$ set to in the following code?

```java
i = 5;
j = i++;
j += ++i;
```

A. 5, 7
B. 7, 5
C. 7, 11
D. 7, 12
E. 7, 13
sizeof Operator

Issue: How to determine the sizes of data?

Thought process
- The sizes of most primitive types are un- or under-specified
- Provide a way to find size of a given variable programmatically

Decisions
- Provide a `sizeof` operator
  - Applied at compile-time
  -Operand can be a data type
  -Operand can be an expression, from which the compiler infers a data type

Examples, on courselab using gcc217
- `sizeof(int)` evaluates to 4
- `sizeof(i)` evaluates to 4 (where i is a variable of type int)
Q: What is the value of the following `sizeof` expression on the courselab machines?

```c
int i = 1;
sizeof(i + 2L)
```

A. 3  
B. 4  
C. 8  
D. 12  
E. error
Other Operators

Issue: What other operators should C have?

Decisions

• Function call operator
  • Should mimic the familiar mathematical notation
  • `function(param1, param2, ...)`
• Conditional operator: `?:`
  • The only ternary operator: “inline if statement”
  • Example: `(i < j) ? i : j` evaluates to min of i and j
  • See King book for details
• 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
**Control Statements**

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

- **Thought process**
  - *Boehm* and *Jacopini* proved that any algorithm can be expressed as the nesting of only 3 control structures:

  - **Sequence**
    - `statement1`
    - `statement2`

  - **Selection**
    - `condition`
    - `statement1`
    - `statement2`

  - **Repetition**
    - `condition`
    - `statement`
Control Statements (cont.)

- Thought Process (cont.)
  - Dijkstra argued that any algorithm should be expressed using only those control structures (GOTO Statement Considered Harmful paper)

- Decisions
  - Provide statements to implement those 3 control structures
  - For convenience, provide a few extras
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;
```
Selection Statements

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);
```
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 grudgingly provided
  - Jump to specified `label`
Declaring Variables

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

**Thought process:**
- Declaring variables allows the compiler to check spelling.
- Declaring variables allows the 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.):

• Unlike Java, declaration statements must appear before any other kind of statement in compound statement.

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

Illegal in C

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

Legal in C
Decisions (cont.)

• Similarly, cannot declare loop control variable in `for` statement

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

Illegal in C
```

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

Legal in C
```
Java only

- Declarations anywhere within block
- Declare immutable variables with `final`
- Conditionals of type `boolean`
- “Labeled” `break` and `continue`
- No `goto`

C only

- Declarations only at beginning block
- Declare immutable variables with `const`
- Conditionals of any type (checked for zero / nonzero)
- No “labeled” `break` and `continue`
- `goto` provided (but don’t use it)
What does the following code print?

```c
int i = 1;
switch (i++) {
    case 1: printf("%d", ++i);
    case 2: printf("%d", i++);
}
```

A. 1  
B. 2  
C. 3  
D. 22  
E. 33
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**: ...
Reading Characters

**Issue:** What functions should C provide for reading characters from standard input?

**Thought process**
- Need function to read a single character from `stdin`
- Function must have a way to indicate failure, that is, to indicate that no characters remain

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

**Note**
- There is no such thing as "the EOF character"
Writing Characters

Issue: What functions should C provide for writing a character to standard output?

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

Decisions
• Provide a `putchar()` function
• Define `putchar()` to accept one parameter
  • For symmetry with `getchar()`, parameter should be an `int`
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 one 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

See King book for details
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 one 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

See King book for details
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 (last time)
- 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

\[
\begin{align*}
    f & \quad 1100000111011011000000000000000000 \\
    i &= \text{(int)} f \\
    i & \quad 11111111111111111111111111100101
\end{align*}
\]

\[-27.375 \quad -27\]
Appendix: The Cast Operator

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

\[
\begin{align*}
  f &= 1100000011101101100000000000000000 \\
  \text{d} &= (\text{double}) f \\
  \text{d} &= 11000000000111011011011000000000000000000 \\
              & \quad 000000000000000000000000000000000000000000 \\
  & \quad -27.375
\end{align*}
\]
Appendix: The Cast Operator

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

\[
\begin{array}{cccc}
\text{i} & 00000000000000000000000000000010 & 2 \\
\text{c} & \text{=} \ (\text{char})\text{i} \\
\text{c} & 00000010 & 2 \\
\end{array}
\]
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

\[
i \quad \begin{array}{c}
11111111111111111111111111111110 \\
\end{array} \quad -2
\]

\[
u = \text{(unsigned int)}i \\
u \quad \begin{array}{c}
11111111111111111111111111111110 \\
\end{array} \quad 4294967294
\]