The Design of C: A Rational Reconstruction (cont.)

Goals of this Lecture

- Recall from last lecture…
- Help you learn about:
  - The decisions that were available to the designers of C
  - The decisions that were made by the designers of C
  … and thereby…
  - C!
- Why?
  - Learning the design rationale of the C language provides a richer understanding of C itself
    - … and might be more interesting than simply learning the language itself !!!
  - A power programmer knows both the programming language and its design rationale
Integer Data Types (cont.)

• Decisions
  • Provide three integer data types: short, int, and long
  • Do not specify sizes; instead:
    • int is natural word size
    • 2 <= bytes in short <= bytes in int <= bytes in long

• Incidentally, on hats using gcc217
  • Natural word size: 4 bytes
  • short: 2 bytes
  • int: 4 bytes
  • long: 4 bytes

Integer Constants

• Issue: How should C represent integer constants?

• Thought process
  • People naturally use decimal
  • Systems programmers often use binary, octal, hexadecimal

• Decisions
  • Use decimal notation as default
  • Use "0" prefix to indicate octal notation
  • Use "0x" prefix to indicate hexadecimal notation
  • Do not allow binary notation; too verbose, error prone
  • Use "L" suffix to indicate long constant
  • Do not use a suffix to indicate short constant; instead must use cast

• Examples
  • int: 123, -123, 0173, 0x7B
  • long: 123L, -123L, 0173L, 0x7BL
  • short: (short)123, (short)-123, (short)0173, (short)0x7B
Unsigned Integer Data Types

- Issue: Should C have both signed and unsigned integer data types?

- Thought process
  - Must represent positive and negative integers
    - Signed types are essential
  - Unsigned data can be twice as large as signed data
    - Unsigned data could be useful
  - Unsigned data are good for bit-level operations
    - Bit-level operations are common in systems programming
  - Implementing both signed and unsigned data types is complex
    - Must define behavior when an expression involves both

Unsigned Integer Data Types (cont.)

- Decisions
  - Provide unsigned integer types: unsigned short, unsigned int, and unsigned long
  - Conversion rules in mixed-type expressions are complex
    - Generally, mixing signed and unsigned converts signed to unsigned
    - See King book Section 7.4 for details
**Unsigned Integer Constants**

- **Issue:** How should C represent unsigned integer constants?
  
- **Thought process**
  - "L" suffix distinguishes `long` from `int`; also could use a suffix to distinguish signed from unsigned
  - Octal or hexadecimal probably are used with bit-level operators

- **Decisions**
  - Default is signed
  - Use "U" suffix to indicate unsigned
  - Integers expressed in octal or hexadecimal automatically are unsigned

- **Examples**
  - `unsigned int`: `123U`, `0173`, `0x7B`
  - `unsigned long`: `123UL`, `0173L`, `0x7BL`
  - `unsigned short`: `(short)123U`, `(short)0173`, `(short)0x7B`

**Character Data Types**

- **Issue:** What character data types should C have?

- **Thought process**
  - The most common character codes are (were!) ASCII and EBCDIC
  - ASCII is 7-bit
  - EBCDIC is 8-bit

- **Decisions**
  - Provide type `char`
  - Type `char` should be one byte

- **Historical verdict**
  - Worked fine for Latin-derived alphabets
  - Unicode required library support for "wide character" type
Character Constants

• Issue: How should C represent character constants?

• Thought process
  • Could represent character constants as int constants, with truncation of high-order bytes
  • More readable to use single quote syntax ('a', 'b', etc.); but then...
  • Need special way to represent the single quote character
  • Need special ways to represent non-printable characters (e.g. newline, tab, space, etc.)

• Decisions
  • Provide single quote syntax
  • Use backslash to express special characters

Character Constants (cont.)

• Examples
  • 'a' the a character
  • (char) 97 the a character
  • (char) 0141 the a character
  • '\0141' the a character, octal character form
  • '\x61' the a character, hexadecimal character form
  • '\0' the null character
  • '\a' bell
  • '\b' backspace
  • '\f' formfeed
  • '\n' newline
  • '\r' carriage return
  • '\t' horizontal tab
  • '\v' vertical tab
  • '\\' backslash
  • '\'' single quote
Strings

• Issue: How should C represent strings?

• Thought process
  • String can be represented as a sequence of chars
  • How to know where char sequence ends?
    • Store length before char sequence?
    • Store special “sentinel” char after char sequence?
  • Strings are common in systems programming
  • C should be small/simple

Strings (cont.)

• Decisions
  • Adopt a convention
    • String consists of a sequence of chars terminated with the null (\0) character
    • Use double-quote syntax (e.g. "abc", "hello") to represent a string constant
    • Provide no other language features for handling strings
      • Delegate string handling to standard library functions
  • Examples
    • "abc" is a string constant
    • 'a' is a char constant (1 byte)
    • "a" is a string constant (2 bytes – 'a' and \0)
Logical Data Type

• Issue: How should C represent logical data?

• Thought process
  • Representing a logical value (TRUE or FALSE) requires only one \textbf{bit}
  • Smallest entity that can be addressed is one \textbf{byte}
  • Type \texttt{char} is one byte, so could be used to represent logical values
  • C should be small/simple

• Decisions
  • Don't define a logical data type
  • Represent logical data using type \texttt{char}, or any integer type
  • Convention: 0 \rightarrow FALSE, non-0 \rightarrow TRUE
  • Convention used by:
    • Relational operators (<, >, etc.)
    • Logical operators (!, &&, ||)
    • Statements (if, while, etc.)

Logical Data Type (cont.)

• Note
  • Using integer data to represent logical data permits shortcuts

```c
... int i;
...
if (i) /* same as (i != 0) */
    statement1;
else
    statement2;
...```

Less typing, but no real performance difference
Logical Data Type (cont.)

- Note
  - The lack of logical data type hampers compiler's ability to detect some errors with certainty

```c
... int i; ...
... i = 0;
... if (i = 5)
   statement1;
else
   statement2;
... 
```

Compiler may warn about this code

Floating-Point Data Types

- Issue: What floating-point data types should C have?
- Thought process
  - Systems programs use floating-point data infrequently
  - But some application domains (e.g. scientific) use floating-point data often
- Decisions
  - Provide three floating-point data types: `float`, `double`, and `long double`
  - bytes in `float` <= bytes in `double` <= bytes in `long double`
- Incidentally, on hats using gcc217
  - `float`: 4 bytes
  - `double`: 8 bytes
  - `long double`: 12 bytes
Floating-Point Constants

• Issue: How should C represent floating-point constants?

• Thought process
  • Convenient to allow both fixed-point and scientific notation
  • Decimal is sufficient; no need for octal or hexadecimal

• Decisions
  • Any constant that contains decimal point or "E" is floating-point
  • The default floating-point type is double
  • Append "f" to indicate float
  • Append "L" to indicate long double

• Examples
  • double: 123.456, 1E-2, -1.23456E4
  • float: 123.456F, 1E-2F, -1.23456E4F
  • long double: 123.456L, 1E-2L, -1.23456E4L

Feature 2: Operators

• Issue: What kinds of operators should C have?

• Thought process
  • Should handle typical operations
  • Should handle bit-level programming ("bit fiddling")

• 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)
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 expressions that involve 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 (cont.)

• Examples

```c
i = 0;
/* Assign 0 to i. Evaluate to 0.
 Discard the 0. */

i = j = 0;
/* Assign 0 to j. Evaluate to 0.
 Assign 0 to i. Evaluate to 0.
 Discard the 0. */

while ((i = getchar()) != EOF) …
/* Read a character. Assign it to i.
 Evaluate to that character.
 Compare that character to EOF.
 Evaluate to 0 (FALSE) or 1 (TRUE). */
```

Does the expressiveness affect clarity?
Special-Purpose Operators

• 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 expressive
  • Such operators would complicate the language and compiler

• Decisions
  • The convenience outweighs the complication
  • Increment/decrement operators (not for floats/double): \( ++ \), \( -- \)
  • Provide special-purpose assignment operators: \( += \), \( -= \), \( *= \), \( /= \), \( %= \)
  \( &= \), \( |= \), \( ^= \), \( <<= \), \( >>= \)

Sizeof Operator

• Issue: How can programmers determine the sizes of data?

• Thought process
  • The sizes of most primitive types are unspecified
  • C must provide a way to determine the size of a given data type programmatically

• Decisions
  • Provide a \texttt{sizeof} operator
    • Applied at compile-time
    • Operand can be a \texttt{data type}
    • Operand can be an \texttt{expression}, from which the compiler infers a data type

• Examples, on hats using gcc
  • \texttt{sizeof(int)} evaluates to 4
  • \texttt{sizeof(i)} evaluates to 4 (where \( i \) is a variable of type \texttt{int})
  • \texttt{sizeof(i+1)} evaluates to 4 (where \( i \) is a variable of type \texttt{int})
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
    • See King book
  • Sequence operator: ,
    • See King book
  • Pointer-related operators: & *
    • Described later in the course
  • Structure-related operators (. ->)
    • Described later in the course

Feature 3: 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, selection, repetition

Barry Boehm
Control Statements (cont.)

• Thought Process (cont.)
  • Dijkstra argued that any algorithm should be expressed using only those three control structures (GOTO Statement Considered Harmful paper)
  • The ALGOL programming language implemented control statements accordingly

• Decisions
  • Provide statements to implement those 3 control structures
  • For convenience, provide a few extras

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 & compound

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

    ```c
    if (integerExpr)
        statement1;
    else {
        if (integerExpr2)
            statement2;
        else
            statement3;
    }
    ```

Selection Statements (cont.)

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

    ```c
    switch (integerExpr) {
        case integerConstant1: 
            ... 
            break;
        case integerConstant2:
            ... 
            break;
        default: 
            ...
    }
    ```

Without break next case executes too

Remembering break is error-prone, so type it when you type case
Repetition Statements

• Issue: How should C implement repetition?

• Decisions
  • while statement, for general repetition, zero or more times
    
    ```c
    while (integerExpr)
    { 
    statement;
    }
    ```

  • for statement, for counting loops, zero or more times & init
    
    ```c
    for (initialExpr; integerExpr; incrementExpr)
    { 
    statement;
    }
    ```

  • do...while statement, for loops with test at trailing edge and one or more times
    
    ```c
    do 
    { 
    statement;
    while (integerExpr);
    }
    ```

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 with for, still executes incrementExpr
    • Can be difficult to understand; generally should avoid
  • goto statement and labels
    • Avoid!!! (as per Dijkstra)
Feature 4: Input/Output Facilities

• Issue: Should C provide I/O facilities?

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

• Decisions
  • Do not provide I/O facilities in C
  • Instead provide a standard library containing I/O facilities
    • Constants: EOF
    • Data types: FILE (described later in course)
    • Variables: stdin, stdout, and stderr
    • Functions: ...

Reading & Writing Characters

• Issue: What functions should C provide for reading & writing characters?

• Thought process
  • Need function to read a single character from stdin and indicate that no characters remain

• Decisions
  • Provide getchar() and putchar() functions
  • Make return type of getchar() wider than char
    • Make it int; that's the natural word size
    • Make putchar() take int for symmetry
  • Define getchar() to return EOF (a special non-character int) to indicate failure

• Note
  • There is no such thing as "the EOF character"
Reading/Writing 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 and vice-versa
  - Could provide `getshort()`, `getint()`, `getfloat()`, etc.
  - Could provide parameterized functions to read/write any primitive type of data
- Decisions
  - Provide `scanf()` / `printf()` functions
  - Can read/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’s design goals affected decisions concerning language features:
  - Data types
  - Operators
  - Control statements
  - I/O facilities

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