The Design of C:
A Rational Reconstruction

1

Goals of this Lecture

- Help you learn abou
  - The decisions t … designers of C
  - The decisi … rs of C
- Why?
  - Learni … rovides a richer
    unde
    - … language itself
  - A po … language and its
    des

2

Goals of C

Designers wanted C to support:

- **Systems programming**
  - Development of Unix OS
  - Development of Unix programming tools

But also:

- **Applications programming**
  - Development of financial, scientific, etc. applications

Systems programming was the primary intended use

3

The Goals of C (cont.)

The designers of wanted C to be:

- Low-level
  - Close to assembly/machine language
  - Close to hardware

But also:

- Portable
  - Yield systems software that is easy to port to differing hardware
  - E.g. Unix, written in C, much more portable than previous OSes

- These goals are conflicting
  - So compromises needed to be made

4

The Goals of C (cont.)

The designers wanted C to be:

- Easy for **people** to handle
  - Easy to understand
  - **Expressive**
    - High (functionality/sourceCodeSize) ratio

But also:

- Easy for **computers** to handle
  - Easy/fast to compile
  - Yield efficient machine language code

Commonality:

- Small/simple

- These sets of goals are also conflicting
  - Understandable and expressive
  - Understandable and easy to compile efficiently

5

Design Decisions

In light of those goals…

- What design decisions did the designers of C **have**?
- What design decisions did they **make**?

Consider a few language features, from simple to complex…

6

Feature 1: Data Types

- Remember:
  - Bits can be combined into bytes
  - Our interpretation of a collection of bytes gives it meaning
    - A signed integer, an unsigned integer, a RGB color, etc.
- A **data type** is a well-defined interpretation of a set of bytes
- A high-level language should provide primitive data types
  - Facilitates abstraction
  - Facilitates manipulation via well-defined operators associated with the data types
  - Enables compiler to check for mixing of types, inappropriate use of types, etc.

7

Primitive Data Types

- Issue: What primitive data types should C provide?
- Thought process
  - C should handle:
    - **Integers**
    - **Characters**
    - Character **strings**
    - **Logical** (alias **Boolean**) data
    - **Floating-point** numbers
  - C should be small/simple
- Decisions
  - Provide **integer**, **character**, and **floating-point** data types
  - **Do not** provide a character **string** data type (More on that later)
  - **Do not** provide a **logical** data type (More on that later)

8

Integer Data Types

- Issue: What integer data types should C provide?
- Thought process
  - For flexibility, should provide integer data types of various sizes
  - For portability at **application** level, should specify size of each data type
  - For **systems** programming, should define integral data types in terms of **natural word size** of computer
  - Primary use will be **systems** programming

Why?

Why?

9

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 nobel using gcc217
  - Natural word size: 4 bytes
  - **`short:`** 2 bytes
  - **`int:`** 4 bytes
  - **`long:`** 4 bytes

10

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

17

Character Constants (cont.)

- Examples
  - `'a'` **the a character**
  - `(char)97` the a character
  - `(char)0141` the a character
  - `'\o141'` 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**

18

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

Advantages/disadvantages?

19

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
  - `"a"` is a string constant

How many bytes?

20

Feature 2: Operators

- A high-level programming language should have **operators**
- Operators combine with constants and variables to form expressions
  - E.g. x + 5
- C provides a number of arithmetic, logical, relational, bitwise and type-casting operators

27

Assignment

- 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

29

Assignment Operator (cont.)

- Examples

```
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?

30

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 `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 nobel using gcc217
  - `sizeof(int)` evaluates to 4
  - `sizeof(i)` evaluates to 4 (where `i` is a variable of type `int`)
  - `sizeof(i+1)` evaluates to 4 (where `i` is a variable of type `int`)

33

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

34

Feature 3: Control Statements

- A programming language must provide **statements**
- Some statements must affect flow of control

35

Control Statements

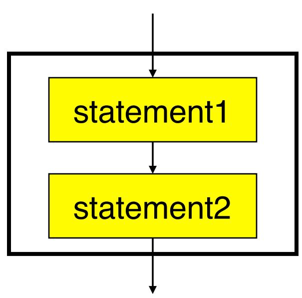
- Issue: What control statements should C provide?

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

36

Control Statements (cont.)

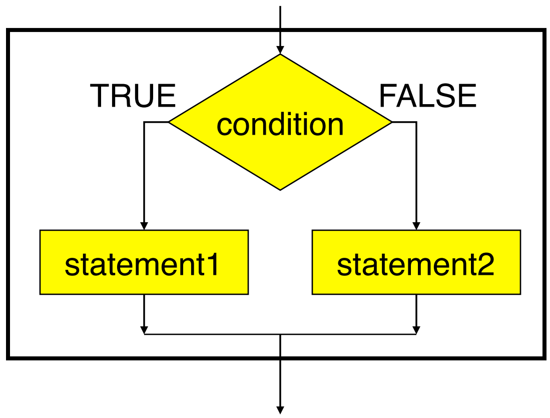
(1) Sequence



37

Control Statements (cont.)

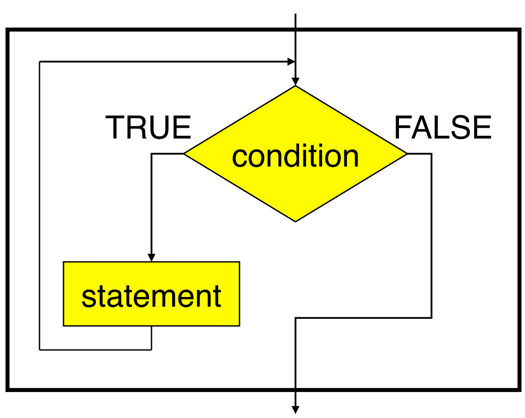
(2) Selection



38

Control Statements (cont.)

(3) Repetition



39

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

Edsgar Dijkstra

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

40

Sequence Statement

- Issue: How should C implement sequence?

- Decision
  - **Compound** statement, alias **block**

```
{
    statement1;
    statement2;
     …
}
```

41

Selection Statements

- Issue: How should C implement selection?

- Decisions
  - **if** statement, for one-path or two-path decisions

```
if (integerExpr)
   statement1;
```

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

42

Selection Statements (cont.)

- Decisions (cont.)
  - **switch** and **break** statements, for multi-path decisions

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

What if these **break** statements are omitted?

Was that use of **break** a good design decision?

43

Repetition Statements

- Issue: How should C implement repetition?
- Decisions
  - **while** statement, for general repetition

```
while (integerExpr)
    statement;
```

  - **for** statement, for counting loops

```
for (initialExpr; integerExpr; incrementExpr)
    statement;
```

  - **do…while** statement, for loops with test at trailing edge

```
do
    statement;
while (integerExpr);
```

44

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
    - Can be difficult to understand; generally should avoid
  - **goto** statement and labels
    - Avoid (as per Dijkstra)

45

Feature 4: Input/Output

- A programming language must provide facilities for reading and writing data
- Alternative: A programming **environment** must provide such facilities

46

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: …

47

Reading types beyond characters

- 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

50

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

51

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

52

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

53

You're getting there …

54