The C Programming Language

Part 1

For Your Amusement

“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, then later, members of standardization committees

Historical context - 1972

Operating systems were programmed in assembly language (i.e., in machine instructions)
[Efficient; expressive; easy to translate to machine language; but not portable from one computer instruction set to another; hard to write programs, hard to debug, maintain…]

Application programs were in “high-level” languages such as Algol, COBOL, PL/1, (newly invented) Pascal

Goals of these languages: Ease of programming, expressiveness, structured programming, safety, data structures, portability

Not fully achieved: safety, expressiveness, portability

Not even attempted: modularity

Goals for C language - 1972

Program operating-systems in a “high-level” language

Need: ease of programming, (reasonable) expressiveness, structured programming, data structures, modularity, compilable on a 64-kilobyte computer

Don’t even attempt: safety

When possible, have a bit of: portability

Goals for Java language - 1995

(reasonable) ease of programming, (reasonable) expressiveness, structured programming, data structures, safety, portability, automatic memory management

It’s not that Java was particularly innovative (in these respects). By 1995, decades of computer-science research had made it straightforward to achieve all these goals at once. In 1972, nobody knew how.
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>Run fast</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!

Agenda

Data Types
Operators
Statements
I/O Facilities

Primitive Data Types

- integer data types
- floating-point data types
- no character data type (use small integer types instead)
- no character string data type (use arrays of small ints instead)
- no logical or boolean data types (use integers instead)

Integer Data Types

- integer data types: char, short, int, long
  - char is 1 byte
  - Number of bits per byte is unspecified! (but in the 21st century, pretty safe to assume it's 8)
  - sizes of other integer type is not fully specified but constrained:
    - int is natural word size
    - $2 \leq \text{sizeof(short)} \leq \text{sizeof(int)} \leq \text{sizeof(long)}$

On CourseLab

- Natural word size: 4 bytes (but not really!)
  - char: 1 byte
  - short: 2 bytes
  - int: 4 bytes
  - long: 8 bytes

Integer Literals

- Decimal: 123
- Octal: 0173 = 123
- Hexadecimal: 0x7B = 123
- Use "L" suffix to indicate long literal
- No suffix to indicate short literal; instead must use cast

Examples

- int: 123, 0173, 0x7B
- long: 123L, 0173L, 0x7BL
- short: (short)123, (short)0173, (short)0x7B

Unsigned Integer Data Types

Both signed and unsigned integer data types
### Unsigned Integer Data Types

Both signed and unsigned integer data types:
- Signed integer types: `int`, `short`, `long`
- Unsigned integer types: `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long`
- `char` might mean `signed char` or `unsigned char`
- Define conversion rules for mixed-type expressions
- Generally, mixing signed and unsigned converts signed to unsigned
- See King book Section 7.4 for details

What decisions did the designers of Java make?

### Unsigned Integer Literals

#### Decisions
- Default is signed
- Use “U” suffix to indicate unsigned literal

#### Examples
- `unsigned int`:
  - 123U, 0173U, 0x7BU
  - 123, 0173, 0x7B will work just fine in practice; technically there is an implicit cast from signed to unsigned, but in these cases it shouldn't make a difference.
- `unsigned long`:
  - 123UL, 0173UL, 0x7BUL

### Character Data Types

Back in 1972, some computers had 6-bit bytes, some had 7-bit bytes, some had 8-bit bytes; the C language had to accommodate all these.

By 1985, pretty much all computers had 8-bit bytes:
- The ASCII character code fits in 7 bits
- One character per byte
- It would be a very strange 21st-century C compiler that supported other than 8-bit bytes

The C character type:
- `char` can hold an ASCII character
- `char` might be signed or unsigned, but since 0 ≤ ASCII ≤ 127 it doesn’t really matter
- If you’re using these for arithmetic, you might care to specify `signed char` or `unsigned char`

### Character Literals

- Single quote syntax: ‘a’
- Use backslash (the escape character) to express special characters

Examples (with numeric equivalents in ASCII):

```plaintext
'a'    the a character (97, 01100001B, 61H)
'\0141' the a character, octal character form
'b'    the b character (98, 01100010B, 62H)
'\x61' the a character, hexadecimal character form
'B'    the B character (66, 01000010B, 42H)
'\0'    the null character (0, 00000000B, 0H)
'O'    the zero character (48, 00110000B, 30H)
'\1'    the one character (49, 00110001B, 31H)
'\"'   the double quote character (34, 00110100B, 22H)
'\\'   the backslash character (92, 01011100B, 5CH)
'\'    the single quote character (39, 00110011B, 27H)
```

### Strings and String Literals

Issue: How should C represent strings and string literals?

Rationale:
- Natural to represent a string as a sequence of contiguous chars
- How to know where char sequence ends?
- Store special “sentinel” char after char sequence?
**Strings and String Literals**

**Decisions**
- Adopt a convention
- String is a sequence of contiguous chars
- String is terminated with null char (`\0`)
- Use double-quote syntax (e.g. "hello") to represent a string literal
- Provide no other language features for handling strings
- Delegate string handling to standard library functions

**Examples**
- `'a'` is a char literal
- "abcd" is a string literal
- "a" is a string literal

**What decisions did the designers of Java make?**

**Logical Data Types**

- no logical or Boolean data type
- Represent logical data using type `char`
  - Or any integer type
  - Or any primitive type!!
- Convention used by:
  - Relational operators (<, >, etc.)
  - Logical operators (!, &&, ||)
  - Statements (if, while, etc.)

**Aside: Logical Data Type Shortcuts**

**Note**
- Using integer data to represent logical data permits shortcuts

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

**Aside: Logical Data Type Dangers**

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

```java
int i;
i = 0;
if (i = 5)
    statement1;
```

**Unicode and UTF-8**

**Back in 1970s,** English was the only language in the world, so we only needed this alphabet:

<table>
<thead>
<tr>
<th>ASCII</th>
<th>Unicode</th>
<th>UTF-8</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>!</td>
<td>!</td>
</tr>
<tr>
<td>( )</td>
<td>( )</td>
<td>( )</td>
</tr>
<tr>
<td>,</td>
<td>,</td>
<td>,</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>;</td>
<td>;</td>
<td>;</td>
</tr>
<tr>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>=</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>@</td>
<td>@</td>
<td>@</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>E</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>G</td>
<td>G</td>
<td>G</td>
</tr>
<tr>
<td>H</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>I</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>J</td>
<td>J</td>
<td>J</td>
</tr>
<tr>
<td>K</td>
<td>K</td>
<td>K</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>M</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>Q</td>
<td>Q</td>
<td>Q</td>
</tr>
<tr>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>S</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>V</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>W</td>
<td>W</td>
<td>W</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Z</td>
<td>Z</td>
<td>Z</td>
</tr>
</tbody>
</table>

**ASCII:** American Standard Code for Information Interchange

**In the 21st century,** it turns out that there are other people and languages out there, so we need:

**Solution:** UTF-8 encoding of Unicode

http://www.cprogramming.com/tutorial/unicode.html

(This won’t be on the exam…)

**Problem 1:** Then, "\n"=0x0a might not mean newline (if it’s one of the bytes of a 3-byte sequence)

**Problem 2:** wastes a lot of space for English text

**But Unicode characters are 24 bits; how to encode them in 8-bit bytes?**

**Obvious solution:** 3 bytes per char.

**Aside:** Logical Data Type Shortcuts

**Note**
- Using integer data to represent logical data permits shortcuts

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

**Aside:** Logical Data Type Dangers

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

```java
int i;
i = 0;
if (i = 5)
    statement1;
```
Floating-Point Data Types

Back in 1972, each brand of computer had a different (and slightly incompatible) representation of floating-point numbers. This was standardized in 1985; now practically all computers use the IEEE 754 Floating Point standard, designed by Prof. William Kahan of the Univ. of California at Berkeley.

• three floating-point data types:
  - float: 4 bytes
  - double: 8 bytes
  - long double: 16 bytes

On CourseLab (and on pretty much any 21st-century computer)

Floating-Point Literals

• fixed-point or “scientific” notation
• Any literal 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: 1.23456, 1E-2, -1.23456E4
- float: 123.456F, 1E-2F, -1.23456E4F
- long double: 123.456L, 1E-2L, -1.23456E4L

Data Types Summary: C vs. Java

Java only
• boolean, byte

C only
• unsigned char, unsigned short, unsigned int, unsigned long

Sizes
• Java: Sizes of all types are specified, and portable
• C: Sizes of all types except char are system-dependent

Type char
• Java: char is 2 bytes (to hold all 1995-era Unicode values)
• C: char is 1 byte

Continued next lecture