Data Types in C
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
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 types of various sizes: signed 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 types not fully specified but constrained:
  - int was intended to be “natural word size”
  - $2 \leq \text{sizeof(short)} \leq \text{sizeof(int)} \leq \text{sizeof(long)}$

On CourseLab

- Natural word size: 8 bytes (“64-bit machine”)
- char: 1 byte
- short: 2 bytes
- int: 4 bytes (compatibility with widespread 32-bit code)
- long: 8 bytes

What decisions did the designers of Java make?
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

unsigned types: unsigned char, unsigned short, unsigned int, and unsigned long

- Conversion rules for mixed-type expressions
  (Generally, mixing signed and unsigned converts unsigned)
- See King book Section 7.4 for details
Unsigned Integer Literals

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
  • unsigned short:
    • (unsigned short)123, (unsigned short)0173, (unsigned short)0x7B
“Character” Data Type

The C `char` type

- `char` can hold an ASCII character
  - And should be used when you’re dealing with characters: character-manipulation functions we’ve seen (such as `toupper`) take and return `char`

- `char` might be signed or unsigned,
  - but since $0 \leq \text{ASCII} \leq 127$ it doesn’t really matter

- If you want a 1-byte type for *calculation*, you might (should?) 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):

<table>
<thead>
<tr>
<th>Character</th>
<th>ASCII Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>'a'</td>
<td>97, 01100001B, 61H</td>
</tr>
<tr>
<td>'\141'</td>
<td>the a character, octal form</td>
</tr>
<tr>
<td>'\x61'</td>
<td>the a character, hexadecimal form</td>
</tr>
<tr>
<td>'b'</td>
<td>98, 01100010B, 62H</td>
</tr>
<tr>
<td>'A'</td>
<td>65, 01000001B, 41H</td>
</tr>
<tr>
<td>'B'</td>
<td>66, 01000010B, 42H</td>
</tr>
<tr>
<td>'\0'</td>
<td>the null character (0, 00000000B, 0H)</td>
</tr>
<tr>
<td>'0'</td>
<td>the zero character (48, 00110000B, 30H)</td>
</tr>
<tr>
<td>'1'</td>
<td>the one character (49, 00110001B, 31H)</td>
</tr>
<tr>
<td>'\n'</td>
<td>the newline character (10, 00001010B, A H)</td>
</tr>
<tr>
<td>'\t'</td>
<td>the horizontal tab character (9, 00001001B, 9H)</td>
</tr>
<tr>
<td>'\'</td>
<td>the backslash character (92, 01011100B, 5CH)</td>
</tr>
<tr>
<td>'''</td>
<td>the single quote character (96, 01100000B, 60H)</td>
</tr>
</tbody>
</table>
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 length together with char sequence?
  • 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

How many bytes?

What decisions did the designers of Java make?
Arrays of characters

```c
char s[10] = {'H','e','l','l','o',0};
(or, equivalently)
char s[10] = "Hello";

char *p = s+2;

printf("Jello!");
```

prints `Jello!`
Back in the 1970s, English was the only language in the world\textsuperscript{[citation needed]}, so we only needed this alphabet:

ASCII: American Standard Code for Information Interchange

In the 21\textsuperscript{st} century, it turns out that there are other people and languages out there, so we need:
Modern Unicode

When Java was designed, Unicode fit into 16 bits, so `char` in Java was 16 bits long. Then this happened:

---

1988:
MY "UNICODE" STANDARD SHOULD HELP REDUCE PROBLEMS CAUSED BY INCOMPATIBLE BINARY TEXT ENCODINGS.

2018:
SENATOR ANGUS KING @SENANGUSKING
GREAT NEWS FOR MAINE — WE'RE GETTING A LOBSTER EMOJI!!! THANKS TO @UNICODE FOR RECOGNIZING THE IMPACT OF THIS CRITICAL CRUSTACEAN, IN MAINE AND ACROSS THE COUNTRY.
YOURS TRULY,
SENATOR 🦞👑
2/7/18 3:12PM

WHAT...WHAT HAPPENED IN THOSE THIRTY YEARS?
THINGS GOT A LITTLE WEIRD, OKAY?

https://xkcd.com/1953/
Unicode and UTF-8

Lots of characters in today’s Unicode
• 100,000+ defined, capacity for > 1 million

Can’t modify size of `char` in C

Solution: variable-length encoding (UTF-8)
• Standard ASCII characters use 1 byte
• Most Latin-based alphabets use 2 bytes
• Chinese, Japanese, Korean characters use 3 bytes
• Historic scripts, mathematical symbols, and emoji use 4 bytes
• This won’t be on the exam!
Logical Data Types

- No separate logical or Boolean data type
- Represent logical data using type `char` or `int`
  - Or any integer type
  - Or any primitive type!!!

- Conventions:
  - Statements (`if`, `while`, etc.) use 0 ⇒ FALSE, ≠0 ⇒ TRUE
  - Relational operators (`<`, `>`, etc.) and logical operators (`!`, `&&`, `||`) produce the result 0 or 1
Logical Data Type Shortcuts

Using integers to represent logical data permits shortcuts

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

It also permits some *really* bad code...

```c
i = (1 != 2) + (3 > 4);
```
Q: What is $i$ set to in the following code?

```c
i = (1 != 2) + (3 > 4);
```

A. 0  
B. 1  
C. 2  
D. 3  
E. 4
Logical Data Type Dangers

The lack of a 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

C specifies:

• Three floating-point data types: 
  \texttt{float}, \texttt{double}, and \texttt{long double} 

• Sizes unspecified, but constrained: 
  \texttt{sizeof(float)} \leq \texttt{sizeof(double)} \leq \texttt{sizeof(long double)}

On CourseLab (and on pretty much any 21\textsuperscript{st}-century computer using the IEEE standard)

• \texttt{float}: 4 bytes 
• \texttt{double}: 8 bytes 
• \texttt{long double}: 16 bytes (but only 10 bytes used on x86-64)
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: 123.456, 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