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
- **pointer** 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)

For “under the hood” details, stay tuned for “number systems” lecture next week
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”
  - \(2 \leq \text{sizeof}(\text{short}) \leq \text{sizeof}(\text{int}) \leq \text{sizeof}(\text{long})\)

On ArmLab:
- 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

• Holds only non-negative integers
• Conversion rules for mixed-type expressions
  (Generally, mixing signed and unsigned converts to 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
The C **char** type

- **char** is designed to 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 (-128..127) or unsigned (0..255)
  - But since 0 ≤ ASCII ≤ 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>Description</th>
<th>ASCII Code</th>
<th>Binary Code</th>
<th>Hexadecimal Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>'a'</td>
<td>the a character</td>
<td>(97, 01100001&lt;sub&gt;B&lt;/sub&gt;, 61&lt;sub&gt;H&lt;/sub&gt;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>'\141'</td>
<td>the a character, octal form</td>
<td>(97, 01100001&lt;sub&gt;B&lt;/sub&gt;, 61&lt;sub&gt;H&lt;/sub&gt;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>'\x61'</td>
<td>the a character, hexadecimal form</td>
<td>(97, 01100001&lt;sub&gt;B&lt;/sub&gt;, 61&lt;sub&gt;H&lt;/sub&gt;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>'b'</td>
<td>the b character</td>
<td>(98, 01100010&lt;sub&gt;B&lt;/sub&gt;, 62&lt;sub&gt;H&lt;/sub&gt;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>'A'</td>
<td>the A character</td>
<td>(65, 01000001&lt;sub&gt;B&lt;/sub&gt;, 41&lt;sub&gt;H&lt;/sub&gt;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>'B'</td>
<td>the B character</td>
<td>(66, 01000010&lt;sub&gt;B&lt;/sub&gt;, 42&lt;sub&gt;H&lt;/sub&gt;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>'\0'</td>
<td>the null character</td>
<td>(0, 00000000&lt;sub&gt;B&lt;/sub&gt;, 0&lt;sub&gt;H&lt;/sub&gt;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>'0'</td>
<td>the zero character</td>
<td>(48, 00110000&lt;sub&gt;B&lt;/sub&gt;, 30&lt;sub&gt;H&lt;/sub&gt;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>'l'</td>
<td>the one character</td>
<td>(49, 00110001&lt;sub&gt;B&lt;/sub&gt;, 31&lt;sub&gt;H&lt;/sub&gt;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>'\n'</td>
<td>the newline character</td>
<td>(10, 00001010&lt;sub&gt;B&lt;/sub&gt;, 0AH)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>'\t'</td>
<td>the horizontal tab character</td>
<td>(9, 00001001&lt;sub&gt;B&lt;/sub&gt;, 09H)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>'\'</td>
<td>the backslash character</td>
<td>(92, 01011100&lt;sub&gt;B&lt;/sub&gt;, 5CH)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>''</td>
<td>the single quote character</td>
<td>(96, 01100000&lt;sub&gt;B&lt;/sub&gt;, 60H)</td>
<td></td>
<td></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

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

char *p = s+2;

printf("Je%s!", p);

p is a pointer: it contains the address of another variable

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

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>!</td>
<td>#</td>
<td>$</td>
<td>%</td>
<td>&amp;</td>
<td>( )</td>
<td>*</td>
<td>+</td>
<td>,</td>
<td>-</td>
</tr>
<tr>
<td>/</td>
<td>:</td>
<td>;</td>
<td>&lt;</td>
<td>=</td>
<td>&gt;</td>
<td>?</td>
<td>@</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
</tr>
<tr>
<td>M</td>
<td>N</td>
<td>O</td>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>V</td>
</tr>
<tr>
<td>W</td>
<td>X</td>
<td>Y</td>
<td>Z</td>
<td>[ \ ]</td>
<td>^</td>
<td>_</td>
<td>`</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td>i</td>
<td>j</td>
<td>k</td>
<td>l</td>
</tr>
<tr>
<td>m</td>
<td>n</td>
<td>o</td>
<td>p</td>
<td>q</td>
<td>r</td>
<td>s</td>
<td>t</td>
<td>u</td>
<td>v</td>
</tr>
<tr>
<td>w</td>
<td>x</td>
<td>y</td>
<td>z</td>
<td>{</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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:
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 $0 \Rightarrow \text{FALSE}, \neq 0 \Rightarrow \text{TRUE}$
  • Relational operators (`<`, `>`, etc.) and logical operators (`!`, `&&`, `||`) produce the result 0 or 1
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?

\[
i = (1 \neq 2) + (3 > 4);
\]

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

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

C specifies:

- Three floating-point data types: `float`, `double`, and `long double`
- Sizes unspecified, but constrained:
  \[
  \text{sizeof}(\text{float}) \leq \text{sizeof}(\text{double}) \leq \text{sizeof}(\text{long double})
  \]

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

- `float`: 4 bytes
- `double`: 8 bytes

On ArmLab (but varying a lot across architectures)

- `long double`: 16 bytes
Floating-Point Literals

How to write a floating-point number?
- Either 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
Java only
  • boolean, byte

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

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