C Types

Kurt Schmidt
Skipjack Solutions

January 22, 2023
- Code examples might have an accompanying link
  - Follow link to step through the example at http://www.pythontutor.com
  - Does a nice job of graphically showing variables in memory, the heap, and the stack
- You will also find examples in the course Lectures/C and Labs/C directory, for your use
- Request other examples, or, create your own
Basic Types
Integers vs. Float Types

- Unlike math, integers and “real” (float) types are treated differently
- They are stored differently
- Operations are different
  - Float multiplication, division, and exponentiation is accomplished with logarithms
Integers

- Integers are stored exactly
- We can only store so many
  - Integers have a limited range
  - *overflow* is an issue
- Come in signed and unsigned flavors
Operations on Integers

- In C, arithmetic operations on integers yield integers
  - So, if \( i \) and \( j \) are integers, each of these expressions, e.g., is also an integer
- Be careful of division:
  - \( 15/4 \rightarrow 3 \)
- This can yield some subtle errors:
  - \( 1/2 \times a \times t \times t \rightarrow 0 \), for all values \( a \) and \( t \)
  - Try \( a \times t \times t/2 \)
These numbers are, generally, approximated
  
  • Powers of 2 are approximated well (exactly)
  • Irrationals ($\sqrt{7}$) are, of course, approximated
  • But even some rationals, like, $\frac{5}{13}$, and even 0.1

Numbers are stored similar to scientific notation
We can represent some very large (and very small) numbers
  • Overflow is rarely a concern
  • We now worry about precision
Operations on Floats

- Arithmetic operations on float types yield float types
- If an arithmetic operation has an integer type and a float type as operands, regardless of order, the integer is coerced to the float type
  - A coercion is a type conversion that the compiler does on its own, w/out being told
  - Ada, e.g., is more strict; it won’t do any coercions
  - Can’t add a float and an integer directly
  - One operand must be explicitly cast to the other’s type by the programmer
- Again, remember, float division and multiplication (and addition and subtraction, for that matter) use different algorithms
- There are functions to help conversions: ceil and floor
Enumerating Small Sets with Integers

- Integers and Floats is about all we have
- Everything else we represent is built on these ideas

- Simple, small sets can be enumerated using integers:
  - Names of weekdays
  - Characters, atoms of strings. (See the ASCII table for an example.)
  - `false` and `true`
  - The suits, on playing cards
Arrays as Tuples

- An array is a finite sequence of homogeneous values
- We can use fixed-length arrays to represent data in $n$ dimensions:
  - More general colors can be represented with 3 floats
  - A location on a plane or the surface of a globe can be represented with 2 floats
Aggregate Data – Structs

- C provides for a user-defined record type, the **struct**
  - A heterogeneous collection of named fields
- Some examples:
  - A student has a first and last name (strings), an ID (string), total credits (int), and quality points (int)
  - A staff member has a first and last name (strings), a department (enumeration), and an hourly wage.
Sequences as Arbitrary Collections

• We might use an array to store an arbitrary-length collection of data
  • The (growing) history of courses and grades a student has
  • The animals living at a zoo
• We will also discuss lists, later on
C Types
C has several types for integers, of varying sizes (ish)

- _Bool
- char
- short
- int
- long

- These come in signed and unsigned flavors

Here are the float types:

- float
- double
- long double

And that’s it. All other types are built upon these types

- with help from aggregation
Integers

• By default, `short`, `int` and `long` are signed
  • E.g., `unsigned long` or `signed short` (default)

• `char` is implementation dependent
  • The Basic ASCII table (including the printing characters) are all positive (0-127)

• The `_Bool` is unsigned
  • This barely matters
  • It is required only to represent 0 (false) and 1 (true)
Size of Integers

- Remember, there are 8 \textit{bits} to a \textit{byte}
- Absolute sizes are not defined
  - Rather, minimum sizes
  - Also, that each larger type is no smaller than its predecessor
  - E.g., for a given implementation, an \texttt{int} is never smaller than a \texttt{short}

<table>
<thead>
<tr>
<th>Type</th>
<th>Min Bytes</th>
<th>Size on Tux</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>_Bool</td>
<td>1</td>
<td>1</td>
<td>([-128, 127]]</td>
</tr>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>([-32,768, 32,767]]</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
<td>([0, 65,535]]</td>
</tr>
<tr>
<td>unsigned short</td>
<td>2</td>
<td>2</td>
<td>([0, 2^{32} - 1])</td>
</tr>
<tr>
<td>int</td>
<td>2</td>
<td>4</td>
<td>([-2^{31}, 2^{31} - 1]]</td>
</tr>
<tr>
<td>unsigned int</td>
<td>2</td>
<td>4</td>
<td>([0, 2^{32} - 1])</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
<td>8</td>
<td>([-2^{63}, 2^{63} - 1]]</td>
</tr>
</tbody>
</table>
Number of Distinct Values

If, e.g., a short is 4 bytes:

- It is stored in 32 bits
- There are, at most, $2^{32}$ distinct values we can store
- *Two’s Complement* is a strategy that yields all $2^{32}$
  - For the unsigned version, the numbers are $[0, 2^{32} - 1]
  - For the signed version we have $1/2$, or $2^{31}$ negative values, $[-2^{31}, -1]$, and $2^{31}$ non-negative numbers, $[0, 2^{31} - 1]$
Use `sizeof` to find the size of types

`sizeof` is a unary operator

- Takes a literal, variable, or type cast
- Returns the number of bytes (as a `size_t`)
  - Actually, returns the number of `char`-sized units
- Even works with arrays"sorta" and structs

```c
#include <stdio.h>
int main() {
    int i = 42;
    printf( "42 takes %zu bytes\n", sizeof 17 ) ;
    printf( "i takes %zu bytes\n", sizeof i ) ;
    printf( "int takes %zu bytes\n", sizeof int ) ;
    return 0 ;
}
```

"sorta"
size_t Type

Using size_t

- size_t represents the size of any type in bytes
- Used for array indexing
- Is an unsigned integral type
- Not built into the language
- Defined in various header files, including `<stdio.h>`, `<stdlib.h>`, `<string.h>`, and `<stddef.h>`
- Use the `%zu` format specifier in `printf`
• **Character literals** are *delimited* by single quotes
  • E.g., ’c’, ’Q’, ’7’

• **String literals** are delimited by double quotes
  • E.g., "abc123", "Heather Graham"
  • C doesn’t have dedicated storage for strings
  • C uses an *array* of *char*

• Numeric digits are contiguous
• In ASCII and UTF-\(n\) :
  • Upper-case letters are contiguous
  • Lower-case letters are contiguous
  • All letters and digits fit into the low 7 bits
ASCII (and UTF) character set allows these conveniences

- Decimal digits are contiguous
  - We can easily convert between a character and its ordinal
    \[ '4' - '0' = 4 \]
- We can play a similar game with letters of the alphabet
  \[ 'A' - 'A' = 0 \]
  \[ 'z' - 'a' = 25 \]
- Difference between ‘A’ and ‘a’ is 32 \( (2^5) \)
  - A single bit
  - To make a upper-case letter into lower-case, turn bit on:
    \[ c = c \mid 1 \ll 5 \]
  - To make a lower-case letter into upper-case, turn bit off:
    \[ c = c \& \neg (1 \ll 5) \]

\(^a\)I’ll make slides on bit operations some time
Boolean Type

• C treats any integer or float value of 0 as $false$
• All other values are $true$
• C99 added a _Bool
  • So name doesn’t break user-defined boolean types
  • Again, really just a small integer
• The stdbool.h header
  • Defines bool (using typedef) as _Bool
  • Also defines literals false (0) and true (1)
Float types

- Float types do not have unsigned flavors
- Remember, floats are approximations

<table>
<thead>
<tr>
<th>Type</th>
<th>Min Bytes</th>
<th>Size on Tux</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>10</td>
<td>16</td>
</tr>
</tbody>
</table>

- A 4-byte float has about 7-8 decimal digits of precision
- A 8-byte double has about 15 decimal digits of precision
Comparing Floats

Don’t Use `==` on Float Types

Is 3.141593 = 3.14159265358979?
- No one answer fits
- Depends on context

Is 0.1 + 0.2 == 0.3?
- No

You pick $\epsilon$ s.t. $|a - b| < \epsilon \implies a = b$
- Use `fabs(double)`

Similarly, use an integer type to store a money amount (whole pennies)
**Literals**

- A *literal* is a symbol which stands for itself

<table>
<thead>
<tr>
<th>Type</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>_Bool</td>
<td>false, true(^1)</td>
</tr>
<tr>
<td>char</td>
<td>'k', 's', '3', '\0', '\n'</td>
</tr>
<tr>
<td>short</td>
<td>28s, 496s</td>
</tr>
<tr>
<td>int</td>
<td>5, 12, 13</td>
</tr>
<tr>
<td>long</td>
<td>8l, 512l</td>
</tr>
<tr>
<td>float</td>
<td>8f, 8.f, .7f, 6.28f, 6.022094e23f</td>
</tr>
<tr>
<td>double</td>
<td>8., .7, 6.28, 6.022094e23, 3e12</td>
</tr>
</tbody>
</table>

- Numbers might be preceded by a + or a -

Please use 8.0 rather than 8., e.g.

\(^1\) If `stdbool.h` is included
## Escaped Character Literals

We use the `\` to represent some common non-printing characters:

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'\0'</td>
<td>Null-terminator (end of string)</td>
</tr>
<tr>
<td>'\t'</td>
<td>Tab</td>
</tr>
<tr>
<td>'\n'</td>
<td>Newline</td>
</tr>
<tr>
<td>'\r'</td>
<td>Carriage return</td>
</tr>
<tr>
<td>'\a'</td>
<td>Alert (terminal bell)</td>
</tr>
<tr>
<td>'\ooo'</td>
<td>Char at ordinal $ooo_{8}$</td>
</tr>
<tr>
<td>'\xhh'</td>
<td>Char at ordinal $hh_{16}$</td>
</tr>
<tr>
<td>'''</td>
<td>Single quote (handy inside single quotes)</td>
</tr>
<tr>
<td>'&quot;'</td>
<td>Double quote (handy inside double quotes)</td>
</tr>
<tr>
<td>'\'</td>
<td>Literal \</td>
</tr>
</tbody>
</table>

There are more.
Strings

• String literals are delimited by double quotes
  • E.g., "abc123", "Heather Graham"
• C doesn’t have dedicated storage for strings
  • C uses an array of char
• C concatenates adjacent string literals into a single string:

```c
#include <string.h>

int main() {
    char name[500] ;
    strcpy( name, /* Only 2 args to this function */
        "Rumpelstilzchen"
        " "
        "Smith"
    )
    ...
}
```
Other Integer Literals

C Understands Hex and Octal Integers

- Integral numbers that start with a 0 are taken to be octal
- Integral numbers that start with a 0x or 0X are taken to be hex

Try this, see what you get:

```c
#include <stdio.h>

int main() {
    int i = 012 ,
        j = 0x23 ;

    printf( "i is: %d\n", i ) ;
    printf( "j is: %d\n", j ) ;

    return 0 ;
}
```
Declarations
Declaring Variables

- C is a *statically-typed* language
- Variables must be *declared* with a type before use

```c
    type identifier ;
```

- This allows the compiler to check operations and assignments at compile time
- The type of a variable can not be changed
- Declaration statements can appear outside of a function
- Declarations can appear inside a function\(^1\)
  - These are *local* variables
  - Only visible inside this function, after the declaration

\(^1\)Anywhere, since C99
Initialisers

- Variables can be assigned values at declarations
  
  ```
  type identifier = value;
  ```

- This is different than assignment\(^1\)
- All `const` variables `must` be initialised when defined
- Do not assume that variables are initialised to 0\(^2\)

---

\(^1\)This would be more evident overloading operators for a class in C++

\(^2\)They might be, they might not be. Why guess?
Example – Declarations

```c
#include <stdio.h>

int i = 5 ,
    j = 12 ;  /* Both i & j are integers */

int main() {
    int i = 13 ,  /* This is a different i entirely */
        k = 42 ;

    printf( "i is: %d, j is: %d, k is: %d\n", i, j, k ) ;

    return 0 ;
}
```

Output:

```
i is: 13, j is: 12, k is: 42
```
Global vs. Local Variables

- Variables defined inside a function are *local* to that function
  - \( k \), above
  - Only visible in that function (local scope)
  - Disappears when function exits (*automatic* storage class)

- Variables defined outside all functions are *global*
  - \( j \), above
  - Visible to all functions (global scope)
  - Persists as long as program executes (*static* storage class)
  - Might be hidden, or *shadowed*, by local variables with the same name

- There are two different variables \( i \)
  - Entirely unrelated
  - Almost as though they had different names