Chapter 5: Boolean

Section 5.1: Using stdbool.h

Version ≥ C99

Using the system header file `stdbool.h` allows you to use `bool` as a Boolean data type. **`true`** evaluates to `1` and **`false`** evaluates to `0`.

```
#include <stdio.h>
#include <stdbool.h>

int main(void) {
    bool x = true;  /* equivalent to bool x = 1; */
    bool y = false; /* equivalent to bool y = 0; */
    if (x)  /* Functionally equivalent to if (x != 0) or if (x != false) */
    {
        puts("This will print!");
    }
    if (!y) /* Functionally equivalent to if (y == 0) or if (y == false) */
    {
        puts("This will also print!");
    }
}
```

`bool` is just a nice spelling for the data type `_Bool`. It has special rules when numbers or pointers are converted to it.

Section 5.2: Using #define

C of all versions, will effectively treat any integer value other than `0` as **`true`** for comparison operators and the integer value `0` as **`false`**. If you don't have `_Bool` or `bool` as of C99 available, you could simulate a Boolean data type in C using `#define` macros, and you might still find such things in legacy code.

```
#include <stdio.h>

#define bool int
#define true 1
#define false 0

int main(void) {
    bool x = true;  /* Equivalent to int x = 1; */
    bool y = false; /* Equivalent to int y = 0; */
    if (x) /* Functionally equivalent to if (x != 0) or if (x != false) */
    {
        puts("This will print!");
    }
    if (!y) /* Functionally equivalent to if (y == 0) or if (y == false) */
    {
        puts("This will also print!");
    }
}
```

Don't introduce this in new code since the definition of these macros might clash with modern uses of **`<stdbool.h>`**.

Section 5.3: Using the Intrinsic (built-in) Type _Bool

Version ≥ C99

Added in the C standard version C99, `_Bool` is also a native C data type. It is capable of holding the values `0` (for *false*) and `1` (for *true*).

```
#include <stdio.h>

int main(void) {
    _Bool x = 1;
    _Bool y = 0;
    if(x) /* Equivalent to if (x == 1) */
    {
        puts("This will print!");
    }
    if (!y) /* Equivalent to if (y == 0) */
    {
        puts("This will also print!");
    }
}
```

`_Bool` is an integer type but has special rules for conversions from other types. The result is analogous to the usage of other types in `if` expressions. In the following

```
_Bool z = X;
```

- If `X` has an arithmetic type (is any kind of number), `z` becomes `0` if `X == 0`. Otherwise `z` becomes `1`.
- If `X` has a pointer type, `z` becomes `0` if `X` is a null pointer and `1` otherwise.

To use nicer spellings `bool`, **`false`** and **`true`** you need to use **`<stdbool.h>`**.

Section 5.4: Integers and pointers in Boolean expressions

All integers or pointers can be used in an expression that is interpreted as "truth value".

```
int main(int argc, char* argv[]) {
  if (argc % 4) {
    puts("arguments number is not divisible by 4");
  } else {
    puts("argument number is divisible by 4");
  }
...
```

The expression `argc % 4` is evaluated and leads to one of the values `0`, `1`, `2` or `3`. The first, `0` is the only value that is "false" and brings execution into the `else` part. All other values are "true" and go into the `if` part.

```
double* A = malloc(n*sizeof *A);
if (!A) {
   perror("allocation problems");
   exit(EXIT_FAILURE);
}
```

Here the pointer `A` is evaluated and if it is a null pointer, an error is detected and the program exits.

Many people prefer to write something as `A == NULL`, instead, but if you have such pointer comparisons as part of other complicated expressions, things become quickly difficult to read.

```
char const* s = ....;   /* some pointer that we receive */
if (s != NULL && s[0] != '\0' && isalpha(s[0])) {
   printf("this starts well, %c is alphabetic\n", s[0]);
}
```

For this to check, you'd have to scan a complicated code in the expression and be sure about operator preference.

```
char const* s = ....;   /* some pointer that we receive */
if (s && s[0] && isalpha(s[0])) {
   printf("this starts well, %c is alphabetic\n", s[0]);
}
```

is relatively easy to capture: if the pointer is valid we check if the first character is non-zero and then check if it is a letter.

Section 5.5: Defining a bool type using typedef

Considering that most debuggers are not aware of `#define` macros, but can check **`enum`** constants, it may be desirable to do something like this:

```
#if __STDC_VERSION__ < 199900L
typedef enum { false, true } bool;
/* Modern C code might expect these to be macros. */
# ifndef bool
#  define bool bool
# endif
# ifndef true
#  define true true
# endif
# ifndef false
#  define false false
# endif
#else
# include <stdbool.h>
#endif

/* Somewhere later in the code ... */
bool b = true;
```

This allows compilers for historic versions of C to function, but remains forward compatible if the code is compiled with a modern C compiler.

For more information on `typedef`, see Typedef, for more on **`enum`** see Enumerations