Section 2: Imperative C

2.1: I/O and output side effects

Functional Programming

In CS 135 we used the ***functional programming paradigm***:

- functions are “pure” (*a.k.a.* “mathematical”):
  - functions **only return values**
  - return values **only depend** on argument values
- only **constants** are used

Imperative Programming

In this course we use the ***imperative programming paradigm***:

- functions may be “impure”
- **variables** and **constants** are used
- a **sequence of instructions** (or “statements”) are *executed*. A block {} is formally known as a ***compound statement***, which is simply a **sequence of statements**† (to be executed in order). A program is mostly a sequence of statements to be executed and *control flow* is used to change the order of statements.
- ***side effects*** are used. In fact, this is the most significant difference between functional and imperative programming. Functional programming does not use side effects.

I/O

I/O (Input/Output) is the term used to describe how programs interact with the “real world”. We use text-based I/O.

To display text output in C, we use the `printf` function with a "string" parameter. The first parameter of `printf` is always a "string", but if we want to format the output, we will use a ***format specifier*** (the **f** in `printf`) within the string and will provide an additional argument. The `%d` format specifier is for an integer in decimal form.

There are some special circumstances where you will need to add an extra character to get the output you want. Here are some common ones:

- to print a percent sign (%), you use use two (`%%`)
- to print a backslash (), use two (`\`)
- to print a quote ("), add an extra backslash (`"`)

Example:

```
int main(void) {
  printf("2 plus 2 is: %d\n", 2 + 2);
  printf("%d plus %d is: %dCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", 2, 10 / 5, 2 + 2);
  printf("I am %d%% sure you should watch your", 100);
  printf("spacing!CODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS");
  printf("4 digits with zero padding: %04dCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", 42);    // format and alignment
}
```

Side Effects and State(Introduction)

A programming *side effect* is when the **state** of something changes. State is the value of some data or information at a moment in time.

The `printf` function has a side effect: it changes the output (or “display”). Add an **effects**: section to document any side effects.

Example:

```
// noisy_sqr(n) computes n^2
// effects: produces output
int noisy_sqr(int n) {
  printf("Yo! I'm squaring %d!CODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", n);
  return n * n;
}
```

void functions

A function may *only have a side effect*, and **not return a value**. The **void** keyword is used to indicate a function returns "nothing". In a void function, the return is **optional** and has no expression (when the end of a void function is reached, it returns automatically).

Surprisingly, `printf` is **not** a void function (it returns an int) and `printf` returns the number of characters printed. So, for example, `printf("hello!CODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS")` returns 7 (5 for the letters in the word, 1 for the ! and 1 for CODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS , since CODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS counts as a single character).

Expression Statements

An ***expression statement*** is an expression with a semicolon (;). The **value** of an expression statement is **discarded** after it is executed. The purpose of an expression statement is to generate side effects. Imperative programming is, in essence, "programming by side effects".

Example:

```
// display_score(score, max) displays the player score
// effects: produces output
void display_score(int score, int max) {
  printf("your score is %d out of %d.CODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", score, max);
  return; // optional
}

// sqr(n) computes n^2
int sqr(int n) {
  return n * n;
}

int main(void) {
  display_score(97, 100);
  11;                                      // throws an edX warning
  10 + 1;                                  // throws an edX warning
  sqr(6) - sqr(5);                         // throws an edX warning
  printf("expressionCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS");
  printf("fiveCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS") + 6;                    // throws an edX warning

  // Tracing unusual expression statements
  trace_int(11);
  trace_int(10 + 1);
  trace_int(sqr(6) - sqr(5));
  trace_int(printf("expressionCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS"));
  trace_int(printf("fiveCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS") + 6);
}
```

Statements

There are only three types of C statements:

- **compound statements (blocks)** `{}` a sequence of statements (to be executed in order)
- **expression statements** for generating side effects (values are discarded)
- **control flow statements** control the order in which other statements are executed (e.g., return, if and else)

2.2: Mutation side effects

Variable

Variables store values. To define a variable in C, we need (in order):

- the **type**
- the **identifier** or "name"
- We should also have the initial value

The equal sign (=) and semicolon (;) complete the syntax. When we set an initial value, it is called (*initialization*). Although the syntax looks like any other C statement, variable *definitions* are not considered to be *statements*. They are, in fact, two different "things" (syntactic units).

Mutation

When the value of a variable is changed, it is known as ***mutation***. As with initialization, in C, mutation is achieved with the ***assignment operator*** (`=`). To accomplish this mutation:

- The "right hand side" (RHS) must be an *expression* that produces a **value** with the same *type* as the LHS.
- The LHS **must** be the name of a variable

- The LHS variable is changed (mutated) to store the **value** of the RHS expression. In other words, the RHS value is *assigned* to the variable.
- This is a *side effect*: the state of the variable has changed.

The use of the equal sign (=) can be misleading since the assignment operator is not symmetric. In other words, `x = y;` is **not the same** as `y = x;`.

In addition to the mutation side effect, the assignment operator (=) also produces the right hand side value. This is occasionally used to perform multiple assignments, such as `x = y = z = 0;`, which is the same as `(x = (y = (z = 0)));`.

Avoid having more than one side effect per expression statement.

Initialization

Always initialize variables. Both initialization and assignment use the equal sign (`=`), but they have different semantics. The `=` used in *initialization* is **not** the assignment operator.

Example:

```
int main(void) {
  int my_variable = 7;      // initialized
  int another_variable;     // uninitialized (BAD!)

  int n = 5;                // initialization syntax
  n = 6;                    // assignment operator

  int x = 0, y = 2, z = 3;  // bad style
  int a, b = 0;             // a is uninitialized (BAD!)
}
```

More assignment operators

The *compound* addition assignment operator (`+=`) combines the addition and assignment operator (for convenience).

Additional compound operators include: `-=`, `*=`, `/=`, `%=`.

There are also *increment* and *decrement* operators that increase or decrease a variable by one (either *prefix* or *postfix*).

Example:

```
int main(void) {
  int x = 0;
  int j = 0;
  x += 4;                                         // x = x + 4;
  printf("After line 8: x = %dCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", x);
  x -= 2;                                         // x = x - 2;
  printf("After line 10: x = %dCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", x);
  x++;                                            // x += 1;
  printf("After line 12: x = %dCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", x);
  x--;                                            // x -= 1;
  printf("After line 14: x = %dCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", x);

  x = 5;
  j = x++;                                        // j = 5, x = 6 (BAD STYLE!)
  printf("After line 18: x = %d and j = %dCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", x, j);

  x = 5;
  j = ++x;                                        // j = 6, x = 6 (BAD STYLE!)
  printf("After line 22: x = %d and j = %dCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", x, j);
}
```

Constants

A ***constant*** is a "variable" that is **immutable** (not mutable). In other words, the value of a constant cannot be changed. To define a C *constant*, we add the `const` keyword to the type.

Global and local variables

Variables are either ***global*** or ***local***. *Global* variables are defined *outside* of functions (at the "top level"), whereas *local* variables are defined *inside* of functions.

Example:

```
const int my_global_constant = 42;
int my_global_variable = 7;
```

```
void f(void) {
  const int my_local_constant = 22;
  int my_local_variable = 11;
  trace_int(my_local_constant);
  trace_int(my_local_variable);
  trace_int(my_global_variable);
}

int main(void) {
  trace_int(my_global_constant);
  trace_int(my_global_variable);
  f();
}
```

2.3: More Mutation

Variable Scope

The `scope` of a variable is the region of code where it is “accessible” or “visible”.

For global variables, the scope is anywhere *below* its definition.

Local variables have ***block scope***, which means their *scope* extends from their definition to the *end of the block* they are defined in.

Example:

```
// Demonstrating scope

#include "cs136.h"

void f(int n) {
                        // b OUT of scope
  if (n > 0) {
                        // b OUT of scope
  int b = 19;
                        // b IN scope
  trace_int(b);
  }
                        // b OUT of scope
  // ...
}

                        // g OUT of scope
int g = 1;
                        // g IN scope

int main(void) {
                        // g IN scope
  trace_int(g);
  f(2);
  // shadowing example:
  trace_int(g);         // g => 1
  int g = 2;
  trace_int(g);         // g => 2
  {
    int g = 3;
    trace_int(g);       // g => 3
  }
  trace_int(g);       // g => 2
}
```

“Impure” functions

Recall that the functional paradigm requires “pure” functions:

- functions **only return values** (no side effects)
- return values **only depend** on argument values

For example, the noisy_sqr function is “impure” because it has a side effect (produces output):

```
int noisy_sqr(int n) {
  printf("Yo! I'm squaring %d!CODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", n);
  return n * n;
}
```

Mutating global variables

A function that mutates a global variable has a *mutation side effect* (which makes it "impure").

Example:

```
int counter = 0;       // global variable

// increment() returns the number of times it has been called
// effects: modifies counter
int increment(void) {
  counter += 1;
  return counter;
}

int main(void) {
  assert(increment() == 1);
  assert(increment() == 2);
}
```

Mutating local variables

Mutating a **local** variable does **not** give a function a side effect since it does not affect state *outside* of the function (global state). It only affects state *inside* of the function (local state).

Example:

```
// add1(n) calculates n + 1
int add1(int n) {
  int k = 0;
  k += 1;
  return n + k;
}

int main(void) {
  assert(add1(3) == 4);
}
```

Mutating Parameters

Parameters are nearly *indistinguishable* from local variables, and can also be mutated.

Example:

```
// add1(n) calculates n + 1
int add1(int n) {
  n += 1;
  return n;
}

int main(void) {
  assert(add1(3) == 4);
}
```

Global dependency

A "pure" function only depends on its argument values whereas a function that depends on a global *mutable* variable is "impure" even if it has **no** side effects.

Global *mutable* variables are almost always poor style and should be avoided. As mentioned earlier, unless otherwise specified, you are **not allowed** to use global *mutable* variables on your assignments.

Example:

```
int n = 10;

// addn(k) returns n + k
int addn(int k) {
  return k + n;
}
```

```
int main(void) {
  assert(addn(5) == 15);
  n = 100;
  assert(addn(5) == 105);
}
```

On the other hand, global *constants* are great style and strongly encouraged. We will revisit this topic in more detail in a later section of the course.

2.4: Input side effects

read helper functions

We have provided some helper functions to make reading in input easier until we introduce `scanf`, which is the converse of `printf`. When C reads in integer values using our `read_int` function, it skips over any whitespace (newlines and spaces).

Example:

```
// read_int() returns either the next int from input
//   or READ_INT_FAIL
// effects: reads input

// the constant READ_INT_FAIL is returned by read_int() when:
// * the next int could not be successfully read from input, or
// * the end of input (e.g., EOF) is encountered
```

Testing Harness

To summarize, our function testing strategies are as follows:

- **return values**: use asserts (e.g., in **main**)
- **input and output**: [Run with Tests] (.in and .expect files)

There is an *alternate* approach for testing *return values*. To test a function f, we can write a dedicated test function that reads in argument values from input, passes those values to f, and then prints out the corresponding return values.

This strategy is known as a ***testing harness*** *and it is useful for testing both "pure" and "impure" functions. On some assignment questions, we may build a testing harness for you (later, you may be expected to build your own).*

Example:

```
// sqr(n) computes n^2
int sqr(int n) {
  return n * n;
}

// test_sqr() is an I/O testing harness for sqr
//   it continuously reads in argument values (e.g., n)
//   and then prints out sqr(n)
// effects: reads input
//          produces output
void test_sqr(void) {
  int n = read_int();
  if (n != READ_INT_FAIL) {
    printf("%dCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", sqr(n));
    test_sqr(); // recurse
  }
}

int main(void) {
  test_sqr();
}
```

Testing Terminology

Keyword	**Definition**
Assertion Testing	Using assertions to test our code

I/O-Driven Testing	Uses input and expected output
Testing Harness	Using input call functions
White Box Testing	You can see the code being tested
Black box testing	You can not see the code being tested
Unit Testing	Testing one piece at a time
Regression Testing	Rerunning all tests to check if everything still works after a change to part of the code to ensure that changes made don't introduce new bugs.

I/O Tools Documentation (cs136.h library)

```
/*****************************************************************************
  I/O TOOLS
 ****************************************************************************/

// the constant READ_INT_FAIL is returned by read_int() when either:
// a) the end of input (e.g., EOF) is encountered, or
// b) the next int could not be successfully read from input
//    (e.g., the input is not a properly formatted int)
extern const int READ_INT_FAIL;

// read_int() returns either the next int from input or READ_INT_FAIL
// effects: reads from input
int read_int(void);


// the constant READ_CHAR_FAIL is returned by read_char when either:
// a) the end of input (e.g., EOF) is encountered, or
// b) the next char in input is an invalid (unprintable) char, where
//    valid characters are in the range ' '...'~' or newline 'CODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS'
extern const char READ_CHAR_FAIL;

// the constants READ_WHITESPACE and IGNORE_WHITESPACE are used
//   as arguments to read_char to control its behaviour
extern const bool READ_WHITESPACE;
extern const bool IGNORE_WHITESPACE;

// read_char(ws_behaviour) returns the next valid char from input
//   or READ_CHAR_FAIL (see above)
// note: ws_behaviour determines how read_char handles whitespace and must
//         be either READ_WHITESPACE or IGNORE_WHITESPACE
// effects: reads from input
char read_char(bool ws_behaviour);
```

Symbol Tools Documentation (cs136.h library)

```
/*****************************************************************************
  SYMBOL TOOLS
****************************************************************************/

// symbols follow the same naming convention as identifiers ("names") in C:
//   - they can only contain letters, underscores and numbers
//   - they must start with a letter
//   - they must be <= 63 characters
// at most there can be 255 symbols defined

// when reading or looking up symbols, they are assigned an int ID
```

```c
// the constant INVALID_SYMBOL is returned by lookup_symbol & read_symbol when:
// a) the next symbol in the input or the parameter is invalid, or
// b) the end of the input (e.g., EOF) is encountered (read_symbol only), or
// c) a new symbol is being defined and 255 symbols have already been defined
extern const int INVALID_SYMBOL;


// read_symbol(void) returns the ID for the next valid symbol from input
//   (which may be a new or existing ID) or INVALID_SYMBOL
// effects: reads from input
int read_symbol(void);


// lookup_symbol(symbol_string) returns the ID for symbol_string
//   (which may be a new or existing ID) or INVALID_SYMBOL
int lookup_symbol(const char *symbol_string);


// print_symbol(symbol_id) displays the symbol corresponding to symbol_id
// requires: symbol_id is a valid ID
// effects: displays a message
void print_symbol(int symbol_id);
```