# A *very* brief introduction to programming in C

Mike Holenderski

November 10, 2014

A C program can be modeled as a state machine. The state is defined by the values assigned to the *variables* declared in the program. The program code specifies how its state should change during program execution. C belongs to the class of imperative programming languages, which have four fundamental primitives for specifying the state transitions: *assignment*, *sequential composition*, *branching*, and *loop*. Sometimes a program will contain repetitive code. To make the program code more readable and concise, the primitives can be combined into *functions*, which can be "called" from other places in the program.

Section 2 describes how the program state is specified. Sections 3 and 4 introduce the four basic primitives. Functions are presented in Section 5.

## 1 Comments

Programs are meant to be read by other people, not just machines. To help others understand the code one should describe using plain language what it does and any assumptions the code relies on. Such descriptions are places in comments between the `/*` and `*/` signs. For example,

```
/* This is short a comment */
```

## 2 Variables

The program state is defined by the values assigned to all variables in the program. Each variable is of a particular type, which describes what kind of values the variable can assume. We distinguish the following types:

### 2.1 Simple types

The most common simple types are

1. `char`: an 8-bit integer number[1]
2. `int`: a 16-bit integer number[2]
3. `float`: a 16-bit rational number[2]

[1] `char` used to represent a letter character, hence the name.

[2] The exact bit-size of a variable type depends on the underlying processor architecture. These values hold for a 16-bit controller.

Before a variable can be used, it has to be declared, specifying its type. For example,

```
int i;
```

means that variable `i` is of type `int` (i.e. it is an integer number).

In C, boolean variables (i.e. those which can be either *true* or *false*) are represented using the type `char` or `int`, where 0 represents a *false* value, and anything else represents a *true* value. It is customary to use 1 to represent *true*.

The integer data types `char` and `int` can be either `signed` or `unsigned`. Signed types can represent both positive and negative values, whereas unsigned types can only represent values greater or equal to 0. For example,

```
unsigned int j;
signed int k;
```

declares an unsigned integer variable `j` and a signed integer variable `k`. Note that integer types are signed by default, so the previously declared variable `i` is a signed integer.

### 2.2 Arrays

Arrays represent fixed size sequences of values of the same type. An array variable is defined by the type of the values it contains and its size. For example,

```
int a[3];
```

declares an array variable called `a` which contains 3 integer values.

### 2.3 Structures

A structure (or a tuple) is a collection of data types which you want to treat as a single entity. For example,

```
struct {
  int x;
  int y;
} point;
```

declares a variable `point`, which may represent a point in a two dimensional grid with integer coordinates `x` and `y`.

### 2.4 Type definition

It may be convenient to introduce aliases for certain types, in particular structures. For example,

```
typedef struct {
  int x;
  int y;
} Point;
```

declares a `Point` to be a structure type, containing two integer fields `x` and `y`. The `Point` type can then be used to declare other variables. For example,

```
Point p1;
Point p2;
```

declares two structure variables `p1` and `p2`, each containing an `x` and `y` field.

### 2.5 Pointers (Advanced)

A pointer points to another variable. Pointers are commonly used in combination with structure types. For example,

```
Point* p;
```

declares a pointer variable `p` which points to a `Point` structure.

### 2.6 Functions (Advanced)

A variable can also be of a function (see Section 5) type. A function declaration specifies a function's signature, i.e. the types of all its arguments, and the type of its return value. For example,

```
char (*f)(Point*, int);
```

declares a variable `f`, which is a pointer to a function which takes two arguments: the first of type `Point*` and the second of type `int`, and returns a value of type `char`.

A C function, besides returning a value, can also change the program state. A `void` return type means that the function does not return any value. Such a function only changes the program state. Similarly, a `void` argument list means that the function does not accept any arguments. For example,

```
void (*g)(void);
```

declares a function variable, which does not accept any arguments nor does it return any value.

Function types can also be defined. For example,

```
typedef char (*PointFunction)(Point*, int);
```

defines a function type `PointFunction`. All functions of this type take one `Point*` and one `int` argument, and return a value of type `char`.

### 2.7 Initial values

When declaring a variable one can also specify its initial value, which the variable will hold at the moment that the program starts executing. For example,

```
int i = 42;
```

declares an integer `i` which will be initialized to 42.

The value of an uninitialized variable is undefined.

## 3 Assignment

An assignment accomplishes a state change during program execution, by assigning a new value to a variable. An assignment has the following syntax:

```
<variable> = <expression>;
```

where `<variable>` is the name of the variable which will acquire the value expressed by the `<expression>`. An `<expression>` is comprised of constant values (such as numbers), variable names, function calls (see Section 5), and operators (such as "`==`" or "`<`"). A variable name inside of the `<expression>` represents the current value held by the corresponding variable.

C provides the following basic operators:

| | |
|---|---|
| `a == b` | check if `a` and `b` are equal |
| `a != b` | check if `a` and `b` are not equal |
| `a <= b` | check if `a` $\leq$ `b` |
| `a >= b` | check if `a` $\geq$ `b` |
| `a \|\| b` | check if `a` or `b` is true (i.e. not 0) |
| `a && b` | check if `a` and `b` are both true |

### 3.1 Simple types

For example,

```
i = i + 1;
```

increments the value of the `i` variable by 1. Another example,

```
k = i > 10;
```

compares the value of `i` to 10, and if it is larger than 10, then `k` is assigned 1 (representing *true*), otherwise it is assigned 0 (representing *false*).

### 3.2 Arrays

Individual elements of an array variable can be accessed using the bracket notation. For example,

```
a[2] = a[1] + 20;
```

takes the value of the second element of array `a`, adds 20 to it, and assigns the result to the third element of `a`.

Note that arrays are indexed starting at 0, i.e. an array declared with

```
int a[3];
```

will contains elements `a[0]`, `a[1]`, and `a[2]`.

### 3.3 Structures

Individual fields of a structure variable can be be accessed using the dot notation. For example,

```
p2.y = p1.x + 2;
```

assigns to the `p2.y` field a value which is equal to the value of `p1.x` plus 2. We can also assign the value of an entire structure to another structure variable. For example,

```
p2 = p1;
```

copies the field values from `p1` to `p2`, i.e. after the assignment, `p2.x` will have acquired the value of `p1.x`, and `p2.y` will have acquired the value of `p1.y`.

### 3.4 Constants

A constant provides a name for a fixed value which does not change during runtime. Constants are useful for maintaining the code, when the same value is used in several places in the program: the value assigned to a constant needs to be changed only in one place in the code. For example,

```
#define N 1000
```

will make sure that any occurrence of `N` in the code will be substituted with 1000. Note that it is a textual substitution. For example,

```
#define e 1+1
int x = e*2;
```

will evaluate to `x` $= 1 + 1 * 2 = 3$, rather than `x` $= (1 + 1) * 2 = 4$.

### 3.5 Pointers (Advanced)

A pointer can be assigned a reference to another variable. For example,

```
p = &p1;
```

assigns to the pointer `p` a reference to the structure `p1`. A pointer can be dereferenced, returning the value of the variable it is pointing to. For example,

```
p2 = *p;
```

assigns to the fields of the structure `p2` the values of the structure pointed to by the `p` pointer.

A pointer to a structure, e.g. the `p` pointer, must be dereferenced to access the individual fields. For example,

```
k = (*p).x;
```

assigns the value of the `x` field of the `Point` structure pointed to by the `p` pointer.

There is an alternative notation for accessing a filed of a structure pointed to by a pointer. For example,

```
k = p->x;
```

is equivalent to `k = (*p).x;`

### 3.6 Functions (Advanced)

A function variable can be assigned a particular function definition (see Section 5). For example,

```
char LongerThan(Point*, int);
PointFunction f;
```

declares a `LongerThan` function, and a function variable `f` of type `PointFunction`. We can assign a particular function to a function variable, as long as their signatures match. For example,

```
f = LongerThan;
```

assigns the `LongerThan` function to the function variable `f`.

## 4 Control flow

Control flow describes how to combine assignments to create programs. The three essential control flow primitives are: *sequential composition*, *branching*, and *loop*.

### 4.1 Sequential composition

Assignments and declarations can be combined into sequences using the ";" sign, meaning that the joined assignments should be executed one after the other. For example,

```
Point point;
point.x = 1;
point.y = point.x + 2;
```

will declare variable `point` and then assign 1 to `point.x`, and subsequently 3 to `point.y`.

Note that in C the ";" sign must also follow the last assignment.

### 4.2 Branching

Sequential composition on its own can express only the most basic programs and is not sufficient for most programming tasks. A conditional branch can be introduced into the sequence using the *if-then-else* statement. For example,

```
if (i > 42) {
  /* then branch: we know that i > 42 */
  point.x = i;
  point.y = 2*i;
} else {
  /* else branch: we know that i <= 42 */
  point.x = 0;
  point.y = 0;
}
```

checks if `i` is greater than 42, and if so, executes the `point.x = i; point.y = 2*i;` branch. Otherwise, it executes the `point.x = 0; point.y = 0;` branch. The

curly braces "{" and "}" enclose the code comprising a branch. Note that only one of the branches will be executed.

It is also possible to skip the *else* branch. For example,

```
if (i > 42) {
  point.x = i;
  point.y = 2*i;
}
```

checks if `i` is greater than 42, and if so, executes `point.x = i; point.y = 2*i;`. Otherwise, it does nothing.

### 4.3 Loop

A conditional loop will keep executing a piece of code until its guard is false. For example,

```
while (i > 42) {
  point.x = point.x - i;
  point.y = point.y + i;
  i = i + 1;
}
```

checks if `i` is greater than 42, and keeps executing its *body* (enclosed by curly braces) until the condition is false (i.e. `i` is less or equal to 42).

## 5 Functions

To enable code reuse and recursion, parts of the code can be grouped into functions. For example, imagine that our program needs to compute a lot of dot products. We would like to introduce a function which will compute the dot product of two free vectors. We can *declare* a function

```
int DotProduct(Point, Point);
```

which takes two `Point` arguments and returns a value of type `int`. We can then *define* this function as

```
int DotProduct(Point a, Point b) {
  int dotProduct = a.x*b.x + a.y*b.y;
  return dotProduct;
}
```

This defines function `DotProduct`, which takes two `Point` arguments `a` and `b`, and returns the dot product of the vectors pointed to by `a` and `b`.

A function can be called (or executed) by instantiating its arguments. For example,

```
k = DotProduct(p1, p2);
```

calls the function `DotProduct`, instantiating the `a` argument with `p1` and the `b` argument with `p2`, and assigns the result to variable `k`. After this function call `k` will be equal to the dot product of vectors `p1` and `p2`.

### Local vs. global variables

The scope of a variable determines from where it can be accessed. A *local* variable is declared inside of a function and can be accessed only within that function. A *global* variable is declared outside of functions (on the "global" level) and can be accessed within any function. For example,

```
int i;

int Factorial(int x) {
  int y = 1;
  int k = 1;
  while (k <= x) {
    y = y * k;
    k = k + 1;
  }
  return y;
}

int main(void) {
  int k = 10;
  int x;
  i = 0;
  while (i < k) {
    x = Factorial(i);
    i = i + 1;
  }
}
```

declares a global variable `i`, which is accessible form `Factorial()` and `main()`. Both functions declare their own local variable `k`, i.e. the `k` declared inside `Increment()` does not interfere with the `k` declared in `main()`, and vice-versa. The same hods for variable `x`, since function arguments are treated as local variables.

*Important: all local variables must be declared before any assignments or function calls!*

## References


[1] Niklaus Wirth, *Algorithms & data structures*, Prentice Hall, 1985

[2] B. Kernighan, D. Ritchie, *The C Programming Language*, 2nd Edition, Prentice Hall, 1988