C-Programming

Brandon LIN

October 1, 2023

Contents

Chapter 1

Operators and Expressions

1.1 L-values and R-values

Definition 1.1.1: L-values and R-values

An **L-value** is something that can appear on the left side of an equal sign. An **R-value** is something that can appear on the right side of an equal sign.

Proposition 1.1.1 L-values and R-values

1. An **L-value** identifies a specific location,
   - where a result can be stored
   - that we can refer to later in the location
2. An **R-value** designates a value.

Example 1.1.1

```
b + 25 = a; // Wrong, we can't predict where the result will be.

int a[30];
a[ b + 10 ] = 0; // True, we do know where *(a + b + 10) would be !

int a, *pi;
pi = &a; // True, the expression specifies the location to be modified.
```

The value in the pointer `pi` is the address of a specific location in memory, and the `*` operator directs the machine to that location.

- When used as an **L-value**, this expression specifies the location to be modified.
- When used as an **R-value**, it gets the value currently stored as that location.

Proposition 1.1.2 Meaning of exp1=exp2

Here, we take the **address** of `exp1` **(L-value)** and the **value** of `exp2` **(R-value)**.
We pass the **value** `exp2` into the **value** which **position** `exp1` possess.

1.2 Operator precedence

Chapter 2

Pointers

2.1 Pointer Variables

Definition 2.1.1: Address

The values of variables are stored in the computer's memory, each at a particular location. Each location is identified and referenced with an **address**.

Definition 2.1.2: Pointer, Pointer variable

Pointer variable is a variable whose value is the address of some other memory location.

2.2 Memory and Addresses

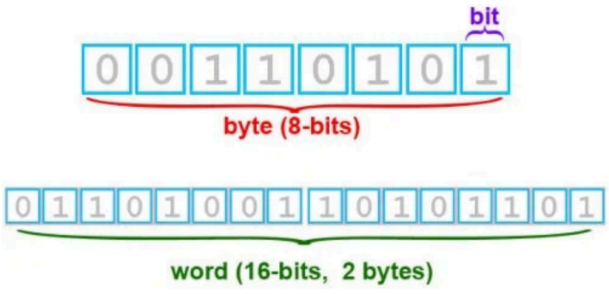

Figure 2.1: Bits and Bytes

Definition 2.2.1: Bits, Bytes, Words

Computer memories are composed of

- **bits**, each capable of holding either the value 0 or the value 1.
- **bytes**, grouping together **bits** and treated as a unit in order to store a wider range of values. Usually, each **bytes** contains eight **bits**, storing unsigned integers from 0 to 255 or signed integers from -128 to 127.
- **words**, taking two or more **bytes** and treat them as if they were a single, larger unit of memory, in order to store even larger values.

Note : Even though a word contains 4 bytes, it has only one address.

Proposition 2.2.1 Location in memory

1. Each location in memory is identified by a unique address.
2. Each location in memory contains a value.

Proposition 2.2.2 Address versus contents

- As it is difficult to remember the real addresses, the high-level languages provide the ability to refer to memory locations by name (**variables**) rather than by address. This is done by the *compiler*.
- The *hardware* still accesses memory locations using addresses.

2.3 Declaration and Initialization

2.3.1 Passing the address to the pointer, Indirection the pointer

Proposition 2.3.1 Declaration

See the code :

```
int *a; // Declaration : a pointer to an integer.
float *a;
```

Proposition 2.3.2 Initialization of a pointer variable

Pointers are initialized with the *addresses of other variables*. `&` operator produces the memory address of its operand.

Definition 2.3.1: Indirection or dereferencing the pointer

The process of following a pointer to the location to which it points to obtain the value is called **Indirection** or **dereferencing** the pointer.
The operator is the unary `*`.

When declaring two or more pointers:

```
int *a, *b, *c; // Correct
int *a, b, c; // Wrong, two integers with one pointer
```

Declaration with initialization :

```
int a;
int *d = &a;

char *message = "Hello world!";
// The value is assigned to message (the pointer variable) rather than *message
// It equals to :
char *message;
message = "Hello world!";
```

2.3.2 Values and their Types

Proposition 2.3.3 Type of a value and the value itself

The type of a value is not something inherent in the value itself but depends on how it is used. The type of a value cannot be determined simply by examining its bits.

Thus, declaring a variable to be a `float` causes the compiler to generate *floating-point instructions* when accessing it.

```
// Two ways of pointing according to the type of the value.
int   a = 112, b = -1;
int  *c = &a;

float  d = 3.14;
float *e = &d;
```

2.3.3 Uninitialized and Illegal Pointer

A very common error :

```
int *a;
...
*a = 12;
/* We're goint to assign the value 12 in the location where a points to.
 * But where does a points to exactly ? */
```

Don't access a location that is outside of the memory allocated to your program. (**Bus error**)

2.3.4 Null Pointer

Definition 2.3.2: NULL pointer

A **NULL pointer** is a pointer value that does not point to anything at all.

To make a pointer variable `NULL` assign the value zero.
To test a pointer variable is `NULL` you compare it to zero.

2.4 Contents of a Pointer Value, Indirection Operator

```
float *e = &d;
/* 1. Produce the memory address of d
 * 2. Pass the value to which we dereference a pointer */
```

Proposition 2.4.1 Contents of a pointer variable

The contents of pointers are addresses rather than integers or floating-point numbers.
It is important to distinguish between the address of the variable and its contents.