Contents

1 Some Facts

- assignments associate from right to left.
- arithemetic operators associate left to right
- relational operators have lower precedence than arithmetic operators
- expressions connected by `&&` or `||` are evaluated left to right. `&&` has a higher precedence than `||`, both are lower than relational and equality operators. (higher than assignment operators?)

- printable characters are always positive
- The standard headers `<limits.h>` and `<float.h>` contain symbolic constants for all of the sizes of basic data types, along with other properties of the machine and compiler.
- a leading 0 (zero) on an integer constant means octal
- a leading `0x` or `0X` (zero x) means hexadecimal.
- you can use escape sequence to represent number. Check it at p51. The complete set of escape sequences are in p52.
- `strlen()` function and other string functions are declared in the standard header `<string.h>`.
- external and static variables are initialized to zero by default.
- for portability, specify `signed` or `unsigned` if non-character data is to be stored in `char` variables. (p58)
- to perform a type conversion:

  ```
  double a = 2.5;
  printf("%d", (int) a);
  ```

  result will be `2`.
- unary operator associate right to left (like `*`, `++`, `--`)

place holder.

2 headers

- `<stdio.h>`: contains input/output functions
- `<ctype.h>`: some functions regarding to characters

3 printf() formatting

Check p26-p27 of the textbook.

Use % with symbols to print the variables in different format. Example:

```
printf("%c", a)  //print a in format of character
printf("%s", a)  //print a in format of character string
printf("%nc", a)  //print a in format of character, using a character width
↪  of size n (at least)
printf("%f", a)  //print a in format of float
printf("%nf", a)  //print a in format of float, using a width of size n
printf("%n.0f", a)  //print a in format of float, using a character width
↪  of size n, with no decimal point and no fraction digits
printf("%n.mf", a)  //print a in format of float, using a character width
↪  of size n, with decimal point and m fraction digits
printf("%0.mf", a)  //print a in format of float, with decimal point and m
↪  fraction digits. The width is not constrained.
printf("%d", a)  //print a in format of integer
printf("%o", a)  //print a in format of octal integer
printf("%x", a)  //print a in format of hexadecimal integer
```

4 Symbolic Constants

A `#define` line defines a symbolic name or symbolic constants to be a particular string of characters. You use it like: `#define` *name replacement text*. You put this at the head of your code (outside scope of any function to make it globally). Example:

```
#include <stdio.h>

#define LOWER 0
#define UPPER 300
```

```c
#define STEP 20

int main() {

  for (int i = LOWER; i <= UPPER; i += STEP) {
    printf("%5d\t%20f", i, 5 * (i - 32) / 9.0);
    printf("\n");
  }

  return 0;
}
```

Pay attention that symbolic name or symbolic constants are not variables. They are conventionally written in upper case. No semicolon at the end of a `#define` line.

5 Character Input and Output

- `getchar()`: it reads the next input character from a text stream and returns that (from the buffer?).
- `putchar()`: it prints a character each time it is called and passed a char into.

Pay attention that a character written between single quotes represents an integer value equal to the numerical value of the character in the machine's character set. This is called a character constant. For example, `'a'` is actually `97`.

6 Arrays

The syntax is similar with C++. For example, to define an array of integers with a size of 100, you do:

```c
int nums[100];
```

Remember to initialize each slot:

```
for (int i = 0; i < 100; i++)
  nums[i] = 0;
```

You can also to use assignment operator and `{ }` to initialize the array when defining. For example, the following C-string is initialized when being defined:

```
int main() {
  char s[] = {'a', 'b', 'c' };
  printf("%s", s);
  return 0;
}
```

7 Enumeration constant

An enumeration is a list of constant integer values. For example:

```
enum boolean { NO, YES };
```

The first name in an `enum` has value 0, the next 1, and so on, unless explicit values are specified:

```
enum boolean { YES = 1, NO = 0 };
```

If not all values are specified, unspecified values continue the progression from the last specified value:

```
enum months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV,
↪  DEC };
// FEB is 2, MAR is 3, etc.
```

Names in different enumerations must be distinct. Values need not be distinct in the same enumeration. Enumeration works like using `#define` to associate constant values with names:

```
#define JAN 1
```

```
#define FEB 2
// etc
```

8 type-cast an expression

Explicit type conversions can be forced ("coerced") in any expression. For example:

```
int main() {
  int n = 2;
  printf("%f", (float) n);
  return 0;
}
```

In the above example, when being printed, the type of `n` has been modified to `float`. Notice that `n` itself is not altered. This is called a *cast*, it is an unary operator, has the same high precedence as other unary operators.

9 Bitwise operators

p62

There are 6 bitwise operators for bit manipulation. They may be applied to integral operands only.

They are:

- `&` : bitwise AND
- `|` : bitwise inclusive OR
- `^` : bitwise exclusive OR
- `«` : left shift
- `»` : right shift
- `~` : one's complement (unary)

The precedence of the bitwise operators `&`, `^` and `|` is lower than `==` and `!=`.

10 Operators can be used with assignment operators

p64

`+, -, *, /, %, «, », &, ^, |`

11 External Variables

If an external variables is to be referred to before it is defined, or if it is defined in a different source file from the one where it is being used, then an `extern` declaration is mandatory. For example, a function using external variables in a different source file can declare these variables in following manner:

```
int addNum(int a) {
  extern int ADDAMOUNT;  // variable ADDAMOUNT is in different source file

  return a + ADDAMOUNT;
}
```

Array sizes must be specified with the definition, but are optional with an `extern` declaration.

12 Command-line Arguments

p128 in CPL.

We can pass command-line arguments or parameters to a program when it begins executing. An example is the echo program. On the command prompt, you enter `ehco`, followed by a series of arguments:

```
$ echo hello world
```

then press enter. The command line window will repeat the inputed arguments:

```
$ echo hello world
$ hello world
```

The two strings "hello" and "world" are two arguments passed in echo program.

Basically, when `main()` is called, it is called with two arguments: `argc` and `argv`.

- `argc`: stands for argument count. It is the number of command-line arguments when the program was invoked (i.e. how many strings are there in the line that invoked the program). In the above echo example, `argc == 3`, the three strings are: "echo", "hello" and "world", respectively.
- `argv`: stands for argument vector. It is a pointer to an array of character strings that contain the actual arguments, one per string. You can imagine when you type in command line to invoke a program, what you typed in was stored somewhere in an array of character strings. Additionally, the standard requires that `argv[argc]` be a null pointer. In the echo example, you typed "echo hello world", and following array of characters was stored:

  ```
  ["echo", "hello", "world", 0]
  ```

Knowing this, we can write a program that mimic the `echo` function.