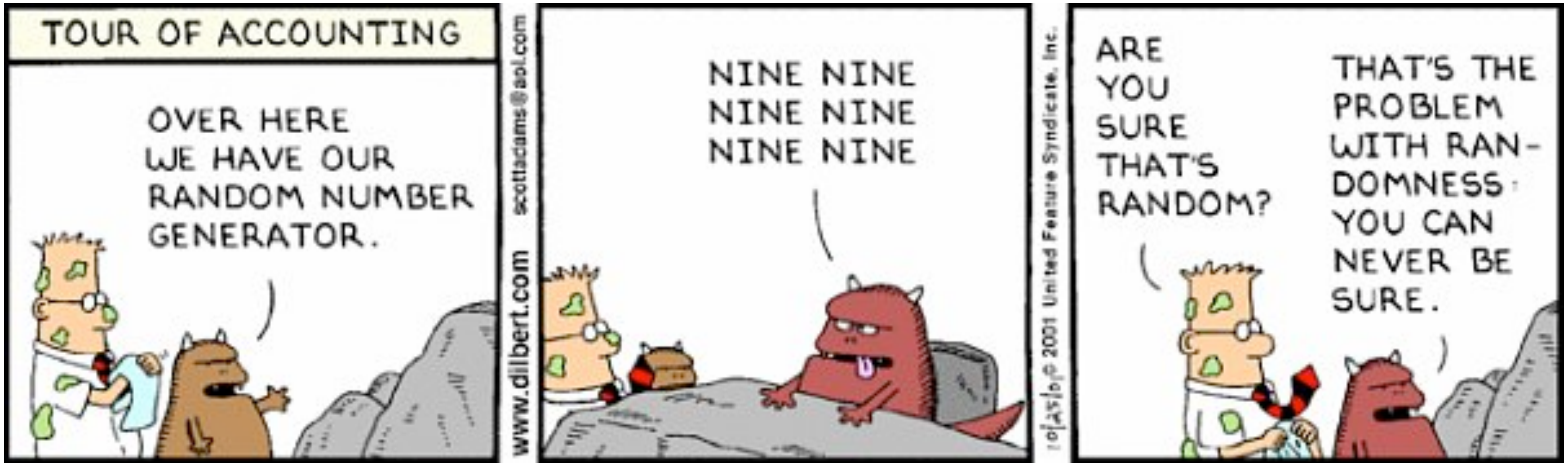
C Programming I: Lecture III

Andreas Heinz
FYD400 - HT 2018


TOUR OF ACCOUNTING
OVER HERE WE HAVE OUR RANDOM NUMBER GENERATOR.
NINE NINE NINE NINE NINE NINE
ARE YOU SURE THAT'S RANDOM?
THAT'S THE PROBLEM WITH RANDOMNESS: YOU CAN NEVER BE SURE.
www.dilbert.com
scottadams@aol.com
© 2001 United Feature Syndicate, Inc.

Recap questions

IN CASE YOU
MISSED IT

- **What is an algorithm?**
- **Where can I find out what the largest number is that I can assign to a variable of type long int?**
- **Why do I need enumerations and how do they work?**
- **What is conditional compiling?**

Lecture 3 Contents

- Functions from the standard library (standard functions)
- Expressions
- Operators and operator precedence

(VtC, Chapter 4)

Standard Functions: *printf*

- **printf is a call of a *standard function* of the *standard library.***
- **The declaration of printf in stdio.h, the code is included in the program during compilation.**

conversion specifier

escape sequence

MACRO; defined in limits.h

```
# include <limits.h>
# include <stdio.h>
int main (void)
{
    printf (“Largest number for a short int: %d\n”,SHRT_MAX);
    return 0;
}
```

Note: “%i” would have worked, too.

Standard Functions: *printf*

- **common conversion specifiers:**

%i	integer	%o	unsigned octal
%d	integer	%x	hexadecimal
%f	float as decimal	%p	pointer
%e	float as exponential	%g	float either as e or f
%c	single character	%u	unsigned integer
%s	text string	%%	prints % sign

=> %10.2f: 10 digits; 2 digits behind the decimal point

```
printf(“%d %d\n”, i);       // WRONG!!!
printf(“%d\n”, i, j);       // WRONG!!!
printf(“%f %d\n”, i, x);    // Correct, if i is a float and x an integer.
printf(”%f”,a*b);           // Correct! printf() evaluates expressions.
```

Printing strings

```
/* Printing strings */
#include <stdio.h>
#include <utility.h>
#define BLURB "Authentic imitation!” // define a macro

int main(void)
{
        printf("[%s]\n",BLURB);
        printf("[%2s]\n",BLURB);
        printf("[%24s]\n",BLURB);
        printf("[%24.5s]\n",BLURB);
        printf("[%-24.5s]\n",BLURB);

        while(!KeyHit());
        return 0;
}
```

Conversion specifier modifer

Printing strings

```
/* Printing strings */
#include <stdio.h>
#include <utility.h>
#define BLURB "Authentic imitation!” // define a macro

int main(void)
{
        printf("[%s]\n",BLURB);
        printf("[%2s]\n",BLURB);
        printf("[%24s]\n",BLURB);
        printf("[%24.5s]\n",BLURB);
        printf("[%-24.5s]\n",BLURB);

        while(!KeyHit());
        return 0;
}
```

```
Output:
[Authentic imitation!]
[Authentic imitation!]
[    Authentic imitation!]
[                   Authe]
[Authe                   ]
```

Conversion specifier modifier => look up

C: Standard Functions

- **scanf()** **(usage similar to printf)**
    - **reads numbers or text strings into variables**
    - **looks for a variable of the right type/format**
    - **the & operator gives the memory address of a variable**
        - **no discussion about the & operator now – see lecture 4**

```
/* Program converts int to hex */
# include <stdio.h>
int main (void)
{
        int number;
        printf (“Get a number:\n”);
        scanf (“%d”, &number);
        printf (“Hexadecimal: %x\n”, number);
        return 0;
}
```

C: Standard Functions

- **scanf()** **(compare printf)**
  - **reads numbers or text strings into variables**
  - **looks for a variable of the right type/format**
  - **the & operator gives the memory address of a variable**
    - **no discussion about the & operator now – see lecture 4**

```
/* Program converts int to hex */
# include <stdio.h>
int main (void)
{
        int number;
        printf (“Get a number:\n”);
        scanf (“%d”, &number);
        printf (“Hexadecimal: %x\n”, number);
        return 0;
}
```

Standard Functions: Try it out

```
# include <stdio.h>
int main (void)
{
        char ch;
        int ascii = 0, i = 0;
        printf ("Give a character: \ n");
        scanf (”%c", &ch);
        ascii = (int) ch;
        while (i < ascii) {  // loop
        printf (".");
            i++;            // means i = i+1
        }
        return 0;
}
```

What is the output of this program? Discuss with your neighbor

"How much exercise would you say you skip each week?"

Standard Functions

```
# include <stdio.h>
int main (void) / * This program does ???  * /
{
        char ch;                                // declaration
        int ascii = 0, i = 0;                   // declaration and initialization
        printf ("Give a character: \ n");   // standard function
        scanf ("%c", &ch);                      // read a character
        ascii = (int) ch;                       // Type conversion, explicit
        while (i < ascii) {                     // Loop
        printf (".");                           // another standard function
          i++;                                  // means i = i + 1
        }
        return 0;
}
```

Standard Functions

```
# include <stdio.h>
int main (void) / * characters in and "Braille-ASCII" out * /
{
          char ch;                                  // declaration
          int ascii = 0, i = 0;                     // declaration and initialization
          printf ("Give a character: \ n"); // standard function
          scanf (”%c", &ch);                        // read a character
          ascii = (int) ch                          // Type conversion, explicit
          while (i < ascii) {                       // Loop
          printf (".");                             // Prints letter in “Baillie”
             i++;
          }
          return 0;
}
```

C: Standard Functions

getchar() and putchar() – an alternative to scanf() and printf()

- **ends reading when reading a End-of-File, called EOF (crtl + z)**
- **read/write a single character fast**

Example (see also chapter 10 in VtC):

```
# include <stdio.h> //  getchar and putchar are in <stdio.h>
int main (void)
{                                    not true
          int number  = 0;           /
          while (getchar () != EOF)          // EOF = End-of-File (ctrl+z)
                    ++ number;               // number = number +1
          printf (“Number of characters: %d”, number);
          return 0;
}
```

Exercise

Take a piece of paper and write down the C code for the a program that reads a float number and that prints it afterwards in exponential form (3 digits behind the comma).

This is not an exam and no one but you will see what you wrote.

"I did a 30-minute workout today: 15 minutes looking for my sneakers, 10 minutes looking for my sweat pants and 5 minutes on the treadmill."

Exercise

```
/* scanf and printf exercise */
/* reads a float and prints it in exponential form */

# include <stdio.h>
#include <utility.h>

int main (void) {
        float number = 0;                        // declaration and initialization
        printf (”Enter a float number: \ n");
        scanf (”%e", &number);                         // read
        printf (“Number: %8.3e”, number);       // print
        while(!KeyHit());
        return 0;
}
```

Expressions and Operators

- expressions – change the value of one or more variables with the help of operators (depending on the variable type)
- operators – have 1 -3 operands
- arithmetic expressions and operators
  - four operations – as usual: +, -, *, /
  - integer division – may be the result of automatic type casting:

```
float a;
a = 3/2; // a is assigned 1 – result of integer division!!
         // Note: 3/2 = a is never possible!!!
```

  - modulus (remainder of…), e.g. *a = 10* % *3* // a = 1
  - if uncertain about the order of application after compiling: use ( ).
  - math.h (stdlib.h) // => access to mathematical functions

```
float x;
y = sin(x); // use <math.h> ; x is not an int; x is in radian
```

Increment and Decrement Operators

- **++ and --**
  - ***i = i + 1;*** **is very common**
    - **can be done easier with *i*++ (post) or ++*i* (pre)**
  - **post: increase after the variable is calculated**
  - **pre: increase before the variable is calculated**
  - **can be used only on variables:**

```
int x = 1, y = 1, z;
z = (x + y)++;     // z = ?
z = x + (y++);     // z = ?
```

Example:

```
int x, n = 5;
x = n++;     // x = ?
x = ++n;     // x = ?
```

Increment and Decrement Operators

- **++ and --**
  - ***i = i + 1;* is very common**
    - **can be done easier with *i*++ (post) or ++*i* (pre)**
  - **post: increase after the variable is calculated**
  - **pre: increase before the variable is calculated**
  - **can be used only on variables:**

Beware of "side effects"!

```
int x = 1, y = 1, z;
z = (x + y)++;     // error – not a variable
z = x + (y++);     // bracket around y does not work: z == 2
```

Example:

```
int x, n = 5;
x = n++;     // x is assigned the value of the expression before
            // the increase; x = 5
x = ++n;    // x = 7 because n increased with ++…
```

Relational Operators

Relation and equality operators

- compare two operands (>, <, <=, >=, ==, !=)
- true gives 1; false gives 0 (int) – boolean type in C99!
- note the == and !=
  - no "=" /* assignment */ and no "=!" /*syntax error*/

Example:

```
int a=12, b=14, i=1;

if (a < b)
                printf("a smaller\n"); // true
else
                printf ("a larger\n");
if (i != 0)
                printf ("i not 0\n"); // true
```

Logical Operators

AND, OR, NOT

- &&, ||, !
- **two, two, or one operand**
- **anything which is not false (0) is true (1) – again of type int**

Example:

```
int a = 12, b = 1, c = 4;
if ((a < 10) && (2 * b < c))   // result?
            printf (“true”);       // is this printed?
else
            printf (“false”);      // or is this printed?
```

Logical Operators

AND, OR, NOT

- &&, ||, !
- two, two, or one operand
- anything which is not false (0) is true (1) – again of type int

Example:

```
int a = 12, b = 1, c = 4;
if ((a < 10) && (2 * b < c))   // false (0) && true (1) give false (0)
        printf (“true”);       // only the first expression is evaluated!!
else
        printf (“false”);      // therefore this is the output
```

Conditional Operator

- **test (expression) ? a (use if test == true) : b (use if test == false)**
- **test is true (!=0) or false (==0)**
- **creates a choice – compare to *if* (later today), “?” is not as common as *if* but effective!**

Example:

```
d = c == 10 ? 11 : 4; // if c is 10, set d to 11, otherwise to 4
d = k > j ? k : j;       // assigns the larger value of j, k to d
                         // does not need brackets!
```

Bitwise Operators

- strength of C: manipulation of data at bit-level -> fast and efficient (compilers, operating systems, encryption, compression, graphics, ...)
- only integer types (char, int, signed or unsigned, ...)
- logic on bit-level:
  - ~ (tilde) bitwise complement - unary operator
  - & bitwise AND (comparable to &&)
  - | bitwise inclusive OR (comparable to ||)
  - ^ bitwise exclusive OR (there is no equivalent*)

```
x = 1, y = 2;    // 01 and 10 in the binary system
z = x & y;       // z = 0 (00)
z = x | y;       // z = 3 (11)
```

*The exclusive-or ^operation takes two inputs and returns a 0 if both bits are 1 or 0, but returns a 1 otherwise.

Bitwise Operators: ~A

Bit n from A	Bit n from ~A
0	1
1	0

```
a = 9;              // Binary: a = 0000 1001
b = ~a;             // Binary: b = 1111 0110
```

Bitwise complement: used to invert all bits.

Bitwise Operators: A & B

Bit n from A	Bit n from B	Bit n from A&B
0	0	0
0	1	0
1	0	0
1	1	1

```
 0 & 1; // Binary: 0 & 1              Result: 0
14 & 1; // Binary: 1110 & 0001        Result: 0000
```

Bitwise AND: used to set bits to zero.

Bitwise Operators: A | B

Bit n from A	Bit n from B	Bit n from A&B
0	0	0
0	1	1
1	0	1
1	1	1

```
 0 | 1;   // Binary: 0 | 1             Result: 1
14 | 1;   // Binary: 1110 | 0001       Result: 1111
```

Bitwise inclusive OR: used to set bits in a binary number to one.

Bitwise Operators: A ^ B

Bit n from A	Bit n from B	Bit n from A&B
0	0	0
0	1	1
1	0	1
1	1	0

```
 0 ^ 1; // Binary: 0 ^ 1           Result: 1
14 ^ 1; // Binary: 1110 ^ 0001   Result: 1111
14 ^ 3; // Binary: 1110 ^ 0011   Result: 1101
        // Bit 0 and bit 1 of number 14 are exchanged
```

Bitwise exclusive OR: used to invert individual bits.

Bitwise Operators

```
i = 21;         // i is 21              (binary 0000000000010101)
j =56;          // j is 56              (binary 0000000000111000)
k = ~i;         // k is now 65514       (binary 1111111111101010)
k = i & j;      // k is now 16          (binary 0000000000010000)
k = i | j;      // k is now 61          (binary 0000000000111101)
k = i ^ j;      // k is now 45          (binary 0000000000101101)
```

- ~ (tilde) bitwise complement - unary operator
- & bitwise AND (comparable to &&)
- | bitwise inclusive OR (comparable to ||)
- ^ bitwise exclusive OR (there is no equivalent)

Highest precedence ~ > & > ^ > | lowest precedence – use of brackets is possible (and often useful)

Bitwise Operators

Bit shift operators : “<<“ and “>>”

// do not mix up with printing commands in C++ !

```
int i,k = 12;      // …00001100 or 0xC in hexadecimal
i = k ^ 0x1f;      // 0x1f: … 00011111, i becomes ...00010011 (19)

k = k | 0x20;      // 0x20: 00100000 sets the 6th bit to 1
                   // regardless of the value of k (here k = 12 + 32),

k = k | (1 << 5); // 00000001 shifts 5 steps to 00100000, see above.
```

see also Ch 4.6 in VtC!

```
number << n; // multiplies number by 2^n
number >> n; // divides number by 2^n (unless n is negative)
```

Assignment Operators

We had already a lot of examples of “=“

- **=…**
- **compound assignment operators**
  - ***=, /=, +=, -=**
- **assign an expression which has a value and can be compared**

Example of a compound operator:

```
a = a – b;          // a = a op b
                    // can be simplified to
a -= b;             // a op= b
```

Beware: i *= j + k is not the same as i = i*j+k because

of operator precedence!

Sizeof Operator

Size of an object (variable) or of its type: very useful.

What is its size in “c-bytes”? -> see limits .h and float.h (Lecture 1)

Example:

```
char q;
m = sizeof(q);      // q is a character variable, it needs 1 byte of
                   // memory by definition, m =1
int a=5;
b = sizeof(a);      // a is an int variable, it needs ? bytes of memory
                   // e.g. 4 bytes, b = 4
```

print(“%zd”,sizeof(int)); // gives direct access to the type

- ⇒ **very interesting, when you work with fields and pointers …**
  evaluated (usually) by the compiler
- ⇒ **works for variables, expressions, or types**

Sizeof Operator and *size_t*

```
/* Prints sizes of different data types */
# include <stdio.h>
# include <utility.h>          /* 1 “C-byte” is the length of a char in bytes */

int main(void){
        size_t x;              // special return type for the sizeof operator
        short a=5;
        int b=5;
        long long c=5;
        printf("Size of a short int: %zd\n”,sizeof(a));   // %zd for size_t in C99
        printf("Size of a int: %zd\n",sizeof(b/2));
        printf("Size of a long long int: %zd\n",sizeof(c/3));
        printf("Size of a size_t: %zd\n”,sizeof(x));
        while(!KeyHit());
        return 0;
}                              // Try this one out yourself and see what you get!
```

Operator Precedence

- **study Table 4.5 in VtC!**
- **usually the priority is fairly intuitive, but sometimes ( ) are necessary**
- **left- or right-associative?**
- **Unary operators (one operand) are always first!**
- **actual priority of calculations is determined by the compiler**
  - **the same variable may be in several places...**
  - **type conversion**
  - **which variable has the values 0 and 1, which ones have other values?**

Example:

```
int i = 2;
i == !!i;    // in fact false because left variable =2
             // and right variable =1
```

Be aware of the the difference between *logic* and *bitwise* operators!

Summary of Lecture 3

- standard functions: printf() and scanf() for formatted I/O by using conversion specifiers, escape sequences and conversion specification modifiers => very useful!
- getchar() and putchar() as alternative
- operators
  - +,-,*,/,%
  - increment and decrement operators (side effects!)
  - relational and logic operators
  - bitwise operators
  - sizeof operator
  - operator precedence