Chapter 2

- Simple Programs
- Simple I/O
- Fundamental data types: Numbers, Strings
- Arithmetic operators
- Memory concepts
- “Decision-making”

Read: Textbook Chapter 2

- Style conventions
- Comments!
- What is a keyword?
- Note: int main()
- scanf( format, addresses );
  e.g.: scanf("%d", &temp);

C

- C is an imperative programming language
  — in a program, you “give orders to the computer”
- This is also called procedural programming
  — a program reflects the procedure how things are done
- Named (sub-)procedures are called functions in C
- Executing a program invokes its main function
- Comments /* ... */ are code-level documentation
- The textbook rightly promotes commenting a lot

Message Repetition Utility

```c
#include <stdio.h> /* message.c */

int main() {
    char message[100]; /* Message to be displayed */
    int count; /* Number of times message is to be displayed */
    int i; /* Loop counter */

    printf("Enter count: "); /* prompt */
    scanf("%d", &count); /* read an integer */
    printf("Enter message:"); /* prompt */
    scanf("%s", message); /* read the message */
    for (i = 1; i <= count; i++) {
        printf(message); /* print the message */
        printf("\n");
    }
    return 0;
}
```
Linux Output

$ ./message
Enter count: 2
Enter message: 5% rebate
5% rebate
5% rebate
$ ./message
Enter count: 2
Enter message: 10% discount
10-1073752496 discount
10-1073752496 discount
$ ./message
Enter count: 2
Enter message: ThisIs100% stupid
ThisIs100% stupid
ThisIs100% stupid
ThisIs100% stupid

Read: Textbook Chapter 2

- Review operator precedence!
- Laws for integer division and modulo:
  \[(n / k) \cdot k + n \% k = n\]
  \[\text{abs}(n \% k) < \text{abs}(k)\]

This is not a unique characterization!

Exercise: Write a C program to test the behaviour of integer division and modulo.
Run it on as many different platforms as you can!
For each kind of behaviour you observe, provide additional laws that uniquely characterize that behaviour.

Floating Point Arithmetic

- Associativity of mathematical addition: \((a + b) + c = a + (b + c)\)
- Floating point addition is not associative:

```c
#include <stdio.h>
/* addAssoc.c */
int main () {
    /* x + (x + y) =?= (x + x) + y */
    float x = 1e-16, y = 1.0; xx = x + x, xy = x + y;
    printf("%.18f =?= %.18f\n", x + xy, xx + y);
    double a = 1e-16, b = 1.0; aa = a + a, ab = a + b;
    printf("%.18f =?= %.18f\n", a + ab, aa + b);
    return 0;
}
```

2S> gcc -std=c99 -pedantic -Wall -o addAssoc addAssoc.c
2S> ./addAssoc
1.0000000000000000 =?= 1.0000000000000000
1.0000000000000000 =?= 1.0000000000000000
2S> gcc -ansi -pedantic -Wall -O3 -o addAssoc addAssoc.c
2S> ./addAssoc
1.0000000000000000 =?= 1.0000000000000000
1.0000000000000000 =?= 1.0000000000000000
```

Read: Textbook Chapter 2 — Arithmetic

- Style conventions
- Comments!
- What is a keyword?
- Note: `int` main()
- `scanf` (format, addresses);
  e.g.: `scanf/company/"%d", &temp);
- Format strings should be string constants!
  Otherwise, if values of variables are user-provided, format string attacks may become possible.
Chapter 3: Structured Program Development in C

• Refinement
• “single-selection statement”: if … then …
• “double-selection statement”: if … then … else …
• while-loops:
  – “counter-controlled repetition”
  – “sentinel-controlled repetition”
• Increment and decrement operators, updating assignment operators

Refinement Example

• Pseudocode can be written in a very abstract, high-level way
  – Read $r_0, \ldots, r_{2N-2}$
  – Construct the Toeplitz matrix
  – Print the Toeplitz matrix

Refinement

• Pseudocode can be written in a very abstract, high-level way
• Pseudocode can be refined by lowering the level of abstraction
  – Read $r_0, \ldots, r_{2N-2}$
    • Prepare array with indices 0 to $2N - 2$
    • For each index $i$ of this array, obtain $r_i$ from user
  – Construct the Toeplitz matrix
    • …
  – Print the Toeplitz matrix

Refinement Example

• Pseudocode can be written in a very abstract, high-level way
• Pseudocode can be refined by lowering the level of abstraction

Stepwise refinement is good software engineering practice
• Pseudocode at the same level as C is dangerous, since usually less precise.
• Refinement is usually performed on specifications
  — 2AA and more advanced courses.
Structured Programming (in C)

Composing programs from

- **expressions** and
- **primitive statements** (assignments, procedure calls)

using **control structures** at the level of:

- **sequencing**: $P_1; P_2$
- **conditional (selection)**: if (condition) $S_1$ else $S_2$
- **while-loops**: while (condition) $S_1$
- **for-loops**: for ($S_1$; condition; $S_2$) $S_3$
- **blocks (compound statements)**: { $P_1$ }

Unstructured Programming

Flowcharts — Control Flow Graphs

- Many decades ago, flowcharts were used in software design.
- A flowchart has **less structure** than the corresponding structured C program
- Therefore, flowcharts are **not useful** in design and refinement
- The flowcharts given in the book serve as **visualizations** of the control flow — they are **control flow graphs** derived from the programs

The Simplest Statement
The Simplest Program

/* empty.c */

void main () {}
The Use of the Conditions in Conditionals (ctd.)

Integer comparison function \(\text{intcmp}(x,y) = \begin{cases} 
1 & \text{if } x > y \\
0 & \text{if } x = y \\
-1 & \text{if } x < y 
\end{cases} \)

```c
int intcmp(int x, int y) {
    if (x > y) {
        assert (x > y); // the if-condition is true here!
        return 1;
    }
    else {
        assert (! (x > y)); // the negation of the if-condition is true here!
        assert (x <= y); // equivalent assertion
        if (x == y) {
            assert (x == y); // the inner if-condition is true here!
            return 0;
        }
        else {
            assert (x <= y && x != y); // both if-conditions are false here!
            assert (x < y); // equivalent assertion
            return -1;
        }
    }
}
```

Conditional Expressions (p. 62)

- Many (functional) programming languages have **conditional expressions**:
  ```c
  max (x,y) = if x > y then x else y
  fact n = if n == 0 then 1 else n * fact (n-1)
  ```
  - C has conditional expressions, too, but with a strange syntax:
    ```c
    condition ? expr_1 : expr_2
    ```

(There can be no one-way version of conditional expressions!)

Example:
```c
float floatMax(float x, float y) {
    return x > y ? x : y ;
}
```
### switch used as Multiple-Selection Statement

```c
switch (expr) {
    case A:
        stmtsA;
        break;
    case B:
        stmtsB;
        break;
    case C:
        stmtsC;
        break;
    case D:
        stmtsD;
        break;
    case E:
        stmtsE;
        break;
    default:
        stmtsDFT;
}
/* end switch */
```

### The while Repetition Statement

```c
while (condition) body
```

A single statement has to be given as body; often, body is a block, i.e., a sequence of statements enclosed in braces { }.

Intuitive meaning:

“While condition evaluates to true, execute execute.”

More precisely — operational semantics:

1. Evaluate condition
2. If result is false, execution of the while statement is finished
3. Otherwise, execute body
4. Resume from step (1)

### Loop Design

- Most loops are intended to terminate (at least in SE2S…)
- Each iteration makes progress towards termination

- Loops start in a controlled situation
- Each iteration expands the scope of this control — progress
- Each iteration maintains this control on the current scope — loop invariant

- Keywords: the loop “maintains”, “keeps”, “remembers”, …

- Formalisation into a C assert() is not always easy.

### Attention: it is easy to get mixed up here!

- indent carefully
- document case groups
Quiz 2005, Program Execution with Assertions

```c
#include <stdio.h>  
#include <assert.h>  // See Textbook 13.10
int square(int k) { return k * k; }
void main(void) {
    int n = 22;
    int k = 0, d = 1, s = 1;
    while (s <= n) {  // Loop invariant:
        assert(d == 2*k + 1 && s == square(k + 1));
        d = d + 2;
        s = s + d;
        k = k + 1;
        assert(d == 2*k + 1 && s == square(k + 1));
        printf("k = %d	d = %d	s = %d\n", k, d, s);
    }
    printf("The result is %d.\n", k);
}
```

Quiz, Question 2 — Factorial

```c
#include <stdio.h>

int fact(int n) { if n == 0 return 1; else return n * fact(n-1); }

void main ( void ) {
    int n = 5, r;
    int n0 = n;  // Initial value n₀ only used in assertions
    for (r = 1; n > 0; n--) {
        assert(fact(n) * r = fact(n0));  // Invariant: n! * r = n₀!
        printf("n = %d\tr = %d\n", n, r);
        r = r * n;
    }
    assert(n == 0 & r = fact(n0));  // Result: r = n₀!
    printf("The result is %d\n", r);
}
```

Quiz, Question 3 — Last Occurrence of Maximum

```c
int locate_max ( int n, int array[] ) {
    assert(n > 0);  // n is positive
    int m = array[0];  // legal since array non-empty
    int ind = 0;  // last time we have seen m
    int i;
    for (i = 1; i < n; i++) {  // start at 1 since we already looked at 0
        if (array[i] >= m) {  // found new maximum or later occurrence
            m = array[i];
            ind = i;
        }
    }
    return ind; }
```

“Sentinel-Controlled Repetition”

The Canadian Oxford Dictionary:

**sentinel** n. a sentry or lookout; a guard. v.tr. 1 station sentinels at or in 2 literary keep guard over or in.

Deitel & Deitel:

[--] sentinel value (also called a signal value, a dummy value, or a flag value)

**General Situation:**

- An operation can either succeed, producing a result of type *t*, or fail.
- Not all elements of type *t* are possible “good” results of the operation
- One element (or a class of elements) of type *t* is defined to indicate failure of the operation.
For loops where the body needs to be executed once before the condition is first tested:

\[
\begin{array}{c}
do \quad body \\
& \text{while ( condition )}
\end{array}
\]

\[
\begin{array}{c}
body \\
while ( condition )
\end{array}
\] \equiv

body has to be a single statement.

It is recommended

• to enclose body in braces, even when it is a primitive statement

• to indent the body

The invariant needs to hold only at the end of the body, not necessarily at the beginning of the first iteration.

Counter-Controlled Repetition

int k; /* Variable declaration at beginning of scope */

... k = 0; /* Variable initialisation immediately preceding loop */
while ( k \leq 10 ) /* Here, condition includes upper bound */
{
  ... printf("k = %d\n", k);
  ... k++; /* Incrementing the loop counter */
}

Writing the loop body as a block even when it is a single primitive statement is good practice.

for Loops

• Counter-controlled repetition occurs frequently
  – Variations: length or direction of step

• Specialised syntax for counter-controlled repetition:
  "A for statement specifies the repeated execution of a statement sequence while a progression of values is assigned to an integer variable called the control variable of the for statement."

• In “real” for loops,
  – the iteration bounds are calculated once, before the body is first executed
  – the body is not allowed to change the control variable

Such restricted for loops

– always terminate, if their body always terminates, but
– have less expressive power than while loops

• for statements in C are abbreviations for a certain kind of while loop.

for Statements in C

First explanation:

\[
\begin{array}{c}
\text{for ( init ; condition ; step )} \\
body
\end{array}
\] \equiv

\[
\begin{array}{c}
\text{while ( condition )} \\
\begin{array}{c}
body \\
step
\end{array}
\end{array}
\]

Typical use — “proper for loop”:

\[
\begin{array}{c}
\text{for ( k = start ; k \leq end ; k++)} \\
body
\end{array}
\]

One kind of atypical use — while loop:

\[
\begin{array}{c}
\text{for ( ; condition ; )} \\
body
\end{array}
\]
Chapter 4: C Program Control

- for, do ... while
- switch
- break, continue
- logical operators

"C-Truthy"

- Modern languages have a predefined datatype for truth values, also called Boolean values
- C allows values of any integral type (including characters and pointers) to be used in conditions.

Truth-value use of integral values:

<table>
<thead>
<tr>
<th>Integral</th>
<th>interpretation → Truth — translation → Integral</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>False — 0</td>
</tr>
<tr>
<td>non-zero</td>
<td>True — 1</td>
</tr>
</tbody>
</table>

Recommended:

```c
#include <stdbool.h>
/* provides bool, true, false */
/* only in C99 */
```

Better than nothing:

```c
#define bool int
#define true 1
#define false 0
```

Truth Tables

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x ∧ y</th>
<th>x ∨ y</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

"C-Truthy" Tables

<table>
<thead>
<tr>
<th>Integral — interpretation → Truth — translation → Integral</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integral — interpretation → Truth — translation → Integral</td>
</tr>
<tr>
<td>0 — False — 0</td>
</tr>
<tr>
<td>non-zero — True — 1</td>
</tr>
</tbody>
</table>

| x | y | x && y | x || y |
|---|---|--------|--------|
| 0 | 0 | 0      | 0      |
| 0 | non-zero | 0 | 1 |
| non-zero | 0 | 0 | 1 |
| non-zero | non-zero | 1 | 1 |
Conditional Evaluation

“An expression containing && or || operators is evaluated only until truth or falsehood is known.”

\[
x && y \equiv IF x THEN y ELSE FALSE ENDIF
\]

\[
x || y \equiv IF x THEN TRUE ELSE y ENDIF
\]

Therefore the following is safe:

```c
if (x != 0 && 12 / x < q)
{
    k = 7 / x;
}
else
{
    fprintf(stderr, "There is a problem.\n")
}
```

Increment and Decrement Operators

Used as statements,

```c
var++; and ++var; abbreviate var += 1;
var--; and --var; abbreviate var -= 1;
```

Assignment used as expression returns the assigned value:

```c
float x, y = 3.5; int k;
x = 4.2 + (k = y); /* k == 3 && x = 7.2 */
```

Used as expressions,

```c
++var abbreviates (var = var + 1)
var++ abbreviates (var0 = var, var += 1, var0)
```

where var0 is a “new” variable.

Updating Assignment Operators

```c
var += expr; abbreviates var = var + expr;
var -= expr; abbreviates var = var - expr;
var *= expr; abbreviates var = var * expr;
var /= expr; abbreviates var = var / expr;
var %= expr; abbreviates var = var % expr;
```

Compare readability:

\[myArray[3+k+i][j-5] += d + 2 \times k;\]

\[myArray[3+k+i][j-5] = myArray[3+k+i][j-5] + d + 2 \times k;\]