Fundamentals of Programming 1

Flow of Control

Arkadiusz Chrobot

Department of Information Systems

November 7, 2024

Outline

Control Statements

A statement is a high-level programming language instruction. *Control statements* or *control flow statements* are statements that change the flow of control in a program, and a crucial part of every programming language. They make it possible to change the order in which other statements are performed. Particularly, they decide, basing on the value of some condition (expression), whether to execute or repeat the execution of other statements. Control statements allow programmers to implement complex algorithms.

Block of Statements

A block of statements (or simply a block) in the C language is a group of statements inside a pair of curly brackets. The group can consist of several statements or just one statement or even it can be empty. The block is interpreted by the program as a single statement. Blocks can be nested i.e. a block can be inserted into another block. We already saw the usage of a block in the definition of the `main()` function, but it can be applied in other parts of source code, particularly in the control statements.

Conditional Statement

Description

The conditional statement (or simply the conditional) is a statement that performs an action depending on a some condition that is a part of it. The overall structure of such a statement is as follows:

```
if(condition)
    statement;
else
    alternative_statement;
```

If the `condition` is satisfied, than the `statement` is executed, otherwise the `alternative_statement` is carried out. When not needed the whole `else` branch can be omitted. In the C language any expression can be used as the condition.

Conditional statement

Remarks

The C language is very flexible in the terms of the structure of statements. For example in the `if` statement a programmer may not only skip the `else` branch but even the `statement`. This can be achieved by putting a semicolon right behind the closing parenthesis. Such a construction of the conditional, although correct, has no practical meaning. Using an assignment instruction (`=`) in the condition, instead of the equality operator (`==`) is a common mistake[1]. However, experienced programmers can use such a condition correctly for simplifying the program.

[1] The compiler will only warn the programmer that the condition should be inserted into another pair of parentheses.

Conditional Statements

An Example

```
if(a==b)
        a=5;
else
        b=5;
```

Some programmers recommend to use blocks even if only a single statement is used after the condition and/or the `else` keyword.

```
if(a==b) {
        a=5;
} else {
        b=5;
}
```

Conditional Statements

Nested Conditionals

It is possible to place a conditional inside another conditional. Such a statement is called a nested conditional:

```
if(a==3) {
        if(b==4)
                c=5;
}
```

However, such a construction can make the code illegible, so it is better to use a complex condition instead:

```
if(a==3 && b==4)
        c=5;
```

Beware, that converting the nested conditional to a simpler form is not always as obvious as in the example, due to the short-circuit evaluation of the expression in the condition.

Switch Statement

The switch statement is a kind of multiple choice statement that changes the flow of control. It performs other statements depending on the value of a variable which is called a *selector*. The variable can be of `int`, `char` or other integer type. The overall structure of `switch` statement is as follows:

```
switch(selector) {
    case value_1: statement_1;
                  break;
…
    case value_nth: statement_nth;
                    break;
    default: statement;
}
```

The number of cases (`case`) is limited only by the range of selector's data type.

Switch Statement

Remarks

More than one statement can be placed in a single case in the switch statement. It is only required that they should be followed by the `break` keyword. Using a block is not required. The `break` keyword, when reached by the program, finishes the execution of the switch statement. If the `case` is not terminated by the `break` then the program unconditionally carries out the next case. Sometimes this feature of switch statement is deliberately used by programmers, but often it is a mistake. If the value of the selector doesn't match any of the values in the cases then the statements in the `default` case are performed. The programmer may decide to not to use the `default` case at all.

Switch Statement

An Example

```
switch(a) {
        case 1: puts("One");
                break;
        case 2: puts("Two");
                break;
        case 3: puts("Three");
                break;
        default: puts("A different number.");
}
```

If the value of the `a` variable is `1` then the program will print the word `One` on the screen. If it is `2` then the word `Two` will be printed. Similarly the word `Three` will appear on the screen if the value of `a` is `3`. In case of any other value the sentence `A different number.` will be shown on the screen.

Iteration Statements

Iteration statements or simply loops repeat the execution of a statement or a block for a finite (sometimes infinite) number of times. The statement or the block is called a *body of the loop*. A single repetition of the body is called an *iteration*, hence the other name of the loops. Usually, the outcome of a single iteration is different from the outcomes of the other iterations, but sometimes is the same.

The `for` Loop

The `for` statement is a count-controlled loop which means that it repeats its body for a given number of times. It needs at least one variable that is known as a *loop counter* or a *control variable*. The overall structure of the `for` statement is as follows:

```
for(counter(s) initialization;condition;afterthought)
    body
```

The upper part of the `for` statement is called a *header*. In the `counter(s) initialization` part a loop counter or counters are given an initial value. If the `condition` is true, the `body` is repeated otherwise the loop terminates. The `afterthought` part describes what happens with the loop counter(s) after a single iteration. Usually the counters have identifiers that consist of a single letter. The loop counter can be a variable of any primary data type in the C language. The `for` loops can be nested.

The for Loop

Examples

```c
#include<stdio.h>

int a;

int main(void)
{
        for(a=0;a<5;a++)
                printf("%d\n",a);
        return 0;
}
```

The for Loop

Examples

```c
#include<stdio.h>

int a;

int main(void)
{
        for(a=0;a<5;a++) {
                printf("%d\n",a);
        }
        return 0;
}
```

The for Loop

Examples

```c
#include<stdio.h>

int a;

int main(void)
{
        for(a=1;a<=5;a++)
                printf("%d\n",a);
        return 0;
}
```

The for Loop

Examples

```c
#include<stdio.h>

int a;

int main(void)
{
        for(a=0;a<7;a+=2)
                printf("%d\n",a);
        return 0;
}
```

The for Loop

Examples

```c
#include<stdio.h>

int a;

int main(void)
{
        a=1;
        for(;a<=5;) {
                printf("%d\n",a);
                a++;
        }
        return 0;
}
```

The for Loop

Examples

```c
#include<stdio.h>

int a;

int main(void)
{
        for(a=7;a>0;a--)
                printf("%d\n",a);
        return 0;
}
```

The for Loop

Examples

```c
#include<stdio.h>

int i,j;

int main(void)
{
        for(i=7,j=0;i>j;j++,i--)
                printf("%d %d\n",i,j);
        return 0;
}
```

The for Loop

Examples

```
#include<stdio.h>

double x;

int main(void)
{
        for(x=0.0;x<0.5;x+=0.01)
                printf("%.10f\n",x);
        return 0;
}
```

The for Loop

Examples

```c
#include<stdio.h>

int a,i;

int main(void)
{
        for(i=0;i<5;i++) {
                a+=i;
                printf("%d\n",a);
        }
        return 0;
}
```

The for Loop

Examples

```
int a;
int main(void)
{
        for(a=0;a<5;a++)
                ;
        return 0;
}
```

The while loop

The **while** statement is a condition-controlled loop. It repeats the execution of its body as long as the condition that is a part of the loop is satisfied. The overall structure of such a loop is as follows:

```
while(condition)
    body;
```

The number of iterations of the **while** loop is not known in advance, therefore its body has to contain at least one expression that will eventually make the **condition** false and the loop will terminate. Otherwise the loop will continue to perform the body infinitely. The **while** loop can be nested or used inside the **for** loop. It is also possible to use the **for** loop inside the **while**.

The while Loop

Examples

```c
#include<stdio.h>

char a;

int main(void)
{
        while(a!='q')
                scanf(" %c",&a);
        return 0;
}
```

The while Loop

Examples

```c
#include<stdio.h>

char a;

int main(void)
{
        while(a!='q') {
                scanf(" %c",&a);
        }
        return 0;
}
```

The while Loop

Examples

```c
#include<stdio.h>

int x,y;

int main(void)
{
        while(y>=0) {
                scanf("%d",&y);
                x+=y;
        }
        return 0;
}
```

The do…while loop

The `do…while` statement is similar to the `while` loop. However, due to its construction the body is *always* executed at least once. The condition is checked after execution of the body. The overall structure of the loop is as follows:

```
do
    body
while(condition);
```

The do…while Loop

Examples

```c
#include<stdio.h>

char a;

int main(void)
{
        do
                scanf(" %c",&a);
        while(a!='q');
        return 0;
}
```

The do…while Loop

Examples

```c
#include<stdio.h>

char a;

int main(void)
{
        do {
                scanf(" %c",&a);
        } while(a!='q');
        return 0;
}
```

The do…while Loop

Examples

```c
#include<stdio.h>

int x,y=1;

int main(void)
{
        do {
                x+=1;
                y*=x;
        } while(x!=10);
        return 0;
}
```

The break Keyword

The **break** keyword may be applied not only in the **switch** statement but also inside a loop's body. In that case it is used with a conditional statement or a ternary operator. If the condition is met, than the **break** keyword terminates the loop. In other words it creates an additional exit point in the loop.

The continue keyword

The `continue` keyword can only be applied inside a loop body and is accompanied by a conditional or a ternary operator. If executed the `continue` keyword terminates the current iteration of the loop and stars the next one. Execution of all statements placed behind the keyword in the loop's body is in that case skipped.

The continue keyword

An Example

```c
#include <stdio.h>

int i;

int main(void)
{
    for(i=-5;i<=5;i++) {
        if(i==0)
            continue;
        printf("5 by %d is %f\n",i,5.0/i);
    }

    return 0;
}
```

The goto keyword

The goto keyword (statement) redirects control to any arbitrary place in the source code marked by a label. The label may be located above the spot in code where the goto is used or below or even in the same place. Although the goto keyword can be helpful its usage should be avoided. In the beginning of computer science it was overused by the programmers. That resulted in a messy, unreadable, unmaintainable and often incorrect code. The situation was so serious that Edsger Dijkstra, one of the pioneers of computer science has forbidden to use the goto statement at all. In the C language the goto keyword is usually applied by experienced programmers for exceptions handling and optimizing the performance of a program. **Other usages of that statement should be avoided at any cost.**

The goto keyword

An Example

```c
#include <stdio.h>

int i;

int main(void)
{

label_1: i++;
         printf("%d\n",i);
         if(i==15)
            goto label_2;
         goto label_1;
label_2:
    return 0;
}
```

Factorial

Description

In mathematics the factorial operation is defined for natural numbers as follows:
$0! = 1$
$1! = 1$
$n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot \ldots \cdot (n-1) \cdot n$
The program in the next slide calculates the factorial using a single `for` loop. The loop counter (the `i` variable) also serves for storing successive natural numbers that are multiplied. The final result is stored in the `factorial` variable, but the same variable is also used for storing the partial products. The argument of the factorial is entered by the user. However, the `do…while` loop limits user's choices to the natural numbers smaller that 21. The reason for that is the type of `factorial` variable. It simply cannot store factorials of greater numbers. Please observe, that the program correctly computes the 0!. In that case the `for` loop body is not executed. Not even once.

Factorial

The Code

```c
#include <stdio.h>

unsigned long long int factorial = 1;
unsigned char i,number;

int main(void)
{
    do {
       printf("Please enter a natural number that is less than 21, ");
       printf("for which You wish to calculate the factorial:\n");
       scanf("%hhu",&number);
    } while(number>20);

    for(i=1;i<=number;i++)
        factorial*=i;

    printf("Factorial of %hhu is %llu\n",number,factorial);

    return 0;
}
```

Factorial

The Code — a slightly different approach

```c
#include <stdio.h>

unsigned long long int factorial = 1;
unsigned char i,number;

int main(void)
{
    do {
       printf("Please enter a natural number that is less than 21, ");
       printf("for which You wish to calculate the factorial:\n");
       scanf("%hhu",&number);
    } while(number>20);

    for(i=1;i<=number;factorial*=i,i++)
        ;

    printf("Factorial of %hhu is %llu\n",number,factorial);

    return 0;
}
```

Greatest Common Divisor

Description

The next example is a program that calculates the Greatest Common Divisor. It uses a modified Euclid's Algorithm that was introduced in the first lecture. The names of the variables are preserved, but the program repeats the $m = n$ and $n = r$ statements just after the r becomes zero. It simplifies the code, but the result is stored in m instead of n. Furthermore, the GCD is calculated only for natural numbers. There is also introduced a `while` loop that prohibits the user from entering zero as the value of both m and n.

Greatest Common Divisor

The Code

```c
#include <stdio.h>

unsigned int r, n, m;

int main(void)
{
    puts("Please enter two natural numbers.");
    scanf("%u",&m);
    scanf("%u",&n);
    while(n==0 && m==0) {
        puts("The values cannot simultaneously be zero!");
        scanf("%u",&m);
        scanf("%u",&n);
    }
    if(n!=0) {
        do {
            r=m%n;
            m=n;
            n=r;
        } while(r!=0);
    }
    printf("The GCD for those numbers is: %u\n",m);
    return 0;
}
```

Quadratic equation

Description

Another example is a program that calculates the roots of a quadratic equation. It uses formulas that are immune to the accumulation of rounding errors, which is one of the issues of the floating-point arithmetics. For the "regular" formulas used for solving the quadratic equations those errors appear when $a \cdot c \ll b$ and the `float` data type is used. The "safe" formulas are as follows: $q \equiv -\frac{1}{2} \cdot [b + sgn(b) \cdot \sqrt{\Delta}]$, $x_1 = \frac{q}{a}$ and $x_2 = \frac{c}{q}$, where sgn is a *signum* function, that yields 1 if $b \geq 0$ or -1 if $b < 0$. The *signum* function is implemented with the use of the ternary operator. The program is protected, so the user cannot enter zero as a value of the a coefficient. The `sqrt()` function is a part of math library of the C language. It calculates the square root of a number. To use the function it is necessary to include the `math.h` header file to the program.

Quadratic equation

The Code

```c
#include<stdio.h>
#include<math.h>

float a,b,c,delta,q;

int main(void)
{
    puts("Please enter the quadratic equation coefficients:");
    do {
        printf("a= ");
        scanf("%f",&a);
        if(a==0.0)
            puts("The value of the 'a' coefficient cannot be 0! Please, enter it correctly:");
    } while(a==0.0);
    printf("b= ");
    scanf("%f",&b);
    printf("c= ");
    scanf("%f",&c);
    delta = b*b-4*a*c;
    if(delta>=0) {
        q = (b<0) ? -0.5*(b-sqrt(delta)) : -0.5*(b+sqrt(delta));
        if(delta!=0.0)
            printf("x1=%f x2=%f\n",q/a,c/q);
        else
            printf("x=%f\n",q/a);
    } else
        puts("This equation has no roots in the real number domain.");

    return 0;
}
```

Binary Numbers

Description

Sometimes it is necessary to display a decimal number in binary. Unfortunately, the C99 standard doesn't define a special formatting string for the `printf()` function to do that in a simple way. However, we should remember that any information in computer memory, including numbers, is represented in binary. The only issue is how to "take it" to the screen. That is what the next program does. It displays in binary an eight-bit number stored in a `char` variable using a single `for` loop. Inside the body of the loop each bit of the number, starting from the most significant one, is tested with the use of the masking operation. The second argument of this operation is an expression that shifts right the value of `MASK` constant (eight bits, the most significant one is set) by as many positions as it is indicated by the value of the loop counter.

Binary Numbers

The Code

```c
#include <stdio.h>

#define MASK 128 // 10000000

int i;
char number;

int main(void)
{
    puts("Please enter the number to be displayed in binary:");
    scanf("%hhi",&number);
    for(i=0;i<8*sizeof(number);i++)
        printf("%d",number&(MASK>>i)?1:0);
    return 0;
}
```

Prime Numbers

Description

A prime number is any natural number greater then one, that is divisible only by one and itself. Finding prime numbers is so time-consuming, that those numbers are applied in cryptography to construct ciphers. The next program displays all such numbers form 3 to almost the upper limit of the `unsigned long long int` type. The basic algorithm for finding a prime number is as follows: *Take a natural number and check if it is a prime by dividing it by all natural numbers grater than* 1 *and less than the number. If all reminders after those divisions are not zeros, the number is a prime, otherwise it is not.* Unfortunately, this algorithm is very inefficient. It is possible to slightly improve it, by making two modifications. In the program the numbers that should be checked are generated by an external `for` loop and all of them are odd. The internal `for` loop checks if a particular number is a prime by applying the above algorithm, but it stops checking when the divisor is greater then the square root of the tested number or when the reminder is equal zero. Please note the useage of the `prime` variable and the `break` keyword.

Prime Numbers

The Code

```c
#include <stdio.h>
#include <limits.h>
#include <math.h>
#include <stdbool.h>

unsigned long long int candidate, divisor;
bool prime;

int main(void)
{
    puts("Prime numbers greater than 2:");
    for(candidate=3;candidate<ULLONG_MAX;candidate+=2) {
        prime=true;
        for(divisor=2;divisor<=sqrt(candidate);divisor++)
            if(candidate%divisor==0) {
                prime = false;
                break;
            }
        if(prime)
            printf("%llu ",candidate);
    }

    return 0;
}
```

Cosine

Description

In the math library of the C language there is the `cos()` function that calculates the value of cosine for a given angle measured in radians. However, it is worth to know how to calculate such a value without the help of the `cos()` function. One of the possibilities is to apply the MacLaurian series. For the cosine it takes the following form:

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots + (-1)^k \cdot \frac{x^{2k}}{(2k)!} + \ldots$$

If we divide some of the series terms by their left neighbours, we will come to the conclusion that they differ by a factor of $-\frac{x^2}{(2i)(2i-1)}$, where i indicates the position of the term in series. We assume that for the $-\frac{x^2}{2!}$ term the value of i is 1.

Cosine

Description — follow-up

The program in the next slide computes the cosine for an angle of $\pi/3$ radians. In the body of the `while` loop all the necessary terms of MacLaurian series are calculated and stored in the `term` variable. The `cosinus` variable stores the sum of all calculated terms. The `i` variable stores the position of the current term. The loop should terminate when the value of `cosinus` is the same as of `cos()` function. However, we cannot compare them directly, because of the floating-point arithmetics properties. Those two numbers could always differ by a very small value and the loop would never stop. Instead, the program measures if the absolute error of those two numbers is less or equal to the `EPSILON` constant, which means that those values are the same with the respect to eleven digits after the decimal point. The absolute error is calculated by subtracting the already mentioned numbers and taking the absolute value of the result. The absolute value is calculated with the use of `fabs()` function from the math library of the C language.

Cosine

The Code

```c
#include<stdio.h>
#include<math.h>

#define EPSILON 1e-11

double cosinus = 1, term = 1, i = 1;
const double x = M_PI/3;

int main(void)
{
    while(fabs(cos(x)-cosinus)>EPSILON) {
        term *= -1.0*x*x/((2*i-1)*(2*i));
        cosinus += term;
        i++;
    }

    printf("The value of the cosine for the %f radians angle is %f\n",x,cosinus);

    return 0;
}
```

Natural Exponential Function

Description

The value of the natural exponential function, for a given exponent, can be calculated similarly to the cosine. The MacLaurian series for such a function takes the following form:

$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^k}{k!} + \ldots$

Using the same method as previously we can find out that the terms differ by a factor of $\frac{x}{i}$, where $i > 0$ is the position of a given term in the series. The program in the next slide calculates the value of the exponential function for an exponent entered by the user. It uses a similar algorithm as the program that calculates the cosine. The main difference between them is that the program for the exponential function uses the relative error to compare the value calculated with the use of the MacLaurian series and the value of the `exp()` function (also from the math library). The latter also calculates the value of the natural exponential function. The relative error can be applied in this program, instead of the absolute error, because the value of the exponential function is never zero.

Natural Exponential Function

The Code

```c
#include<stdio.h>
#include<math.h>

#define EPSILON 1e-11

double exponential = 1.0, x, i=1, term = 1;

int main(void)
{
    puts("Please enter the exponent:");
    scanf("%lf",&x);
    while(fabs((exp(x)-exponential)/exponential)>EPSILON) {
        term *= (x/i);
        exponential += term;
        i++;
    }

    printf("The value of e^x is: %f\n",exponential);

    return 0;
}
```

Thanks

Many thanks to Grzegorz Łukawski, PhD and Leszek Ciopiński, MSc for helping me to complete the Polish version of this slides.

Questions

?

The End

Thank You for Your attention!