4 Control Statements

Any interesting program will take decisions based on its input. This is done in C with the help of certain constructions we study in this section. These C statements will provide *branching* (e.g. `if/else`) or *looping* (`for`).

4.1 Relational and logical operators

True or false in C is expressed by an integer having value `0` (false) or different than zero (true), most commonly `1`. Relational and logical operators return true or false depending on their arguments.

The following is a table of C's relational operators:

Relational Operator	Meaning
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
!=	not equal to

Relational operators can be applied to integers and floating-point numbers. They have lower precedence than arithmetic operators.

IMPORTANT: Note that the relational operator '==' is not the same as the assignment operator '='!

Relational operators are lower in precedence than arithmetic operators. The expression:

```
10 + count > a + 12
```

is evaluated as

```
(10 + count) > (a + 12)
```

The following is a table of C's logic operators:

Logic Operator	Meaning
`&`	and
`||`	or
`!`	not

For generic operands `A` and `B`, logical operators are evaluated as in the following table.

`A`	`B`	`(A && B)`	`(A || B)`	`!A`	`!B`
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	1
1	1	1	1	0	0

Relational operators have higher precedence than the logical operators `&&` and `||`, except `!`. The following statement

```
a < b && b < c || !d
```

is evaluated as

```
(a < b) && (b < c) || (!d)
```

Here is a quick quiz: determine whether the variable `result` is true or false in the next statements:

```
int result;
double a = 5.6, b = 3.6, c = -4.;

result = a < 3. + c;
result = a < 8. && b >= 2.;
result = !(10. * a == b);
result = fabs(c) > 3. || c <= 2;
result = !( b == c || b == a );
result = sin(a) > 1.0;
```

4.2 The `if` and `if/else` statements

The C `if` statement allows conditional execution of statements in your program. The syntax is as follows

```
if (expression)
    statement;
```

When an `if` statement is encountered, `expression` is evaluated. If it is true, then the statement(s) following the test expression is executed. Otherwise the next following statement is executed.

Here is an example:

```
/* Program w3-1.c */

/* The following program reads in an integer and tests whether or not
it is greater than 50. If it is greater than 50, this fact is printed
on the screen.  */

#include <stdio.h>

main() {

  /* declare integer variable */
  int num;

  /* wait for integer input */
  printf("Enter an integer: ");
  scanf("%d", &num);

  /* prints if greater than 50 */
  if (num > 50)
    printf("num is greater than 50.\n");

  /* prints integer */
  printf("num = %d\n", num);

}
```

Note that in the following excerpt the assignment `a = 2` is executed no matter the value of a whereas the assignment `b= 1` is executed only if $a > 1$:

```
  if (a > 1)
    b = 1;
    a = 2;
  c = 3;
```

If more than one statement is to be conditionally executed it should be enclosed in braces.

```
  if (expression) {
    statement1;
    statement2;
    ...
  }
```

Now both assignments a = 2 and b= 1 are only executed if $a > 1$:

```
if (a > 1) {
  b = 1;
  a = 2;
}
c = 3;
```

One can use the keyword `else` to provide alternative execution paths using the syntax:

```
if (expression)
  statement1;
else
  statement2;
```

The `if/else` statement provides a two-way decision path. If `expression` evaluates to true, then `statement1` is executed and `statement2` is skipped. However, if `expression` evaluates false, `statement1` is skipped and `statement2` is executed.

As with the `if` statement, enclose multiple statements in braces if needed.

Here is an example:

```
/* Program w3-2.c */

/* In the following program, two integers are read from the keyboard.
If the second integer is not zero, the division is performed in
integer mode. */

#include <stdio.h>
main() {

  /* declare integer variables */
  int num1, num2;


  /* wait for integer input */
  printf("Enter integers num1 and num2: ");
  scanf("%d %d", &num1, &num2);

  /* if num2 == 0 */
  if (num2 == 0)
    /* print message */
    printf("Error: cannot divide by zero\n");
  else
    /* perform integer division */
```

```
    printf("answer is %d\n", num1 / num2);

}
```

where the integer division is only executed when the divisor (num2) is not zero.

IMPORTANT: A *very common mistake* is the use of the assignment operator '=' instead of the logical operator '=='. What do you think would happen if the if statement is mistakenly type as in the next excerpt?

```
  if (num2 = 0)
    /* print message */
    printf("Error: cannot divide by zero\n");
  else
    /* perform integer division */
    printf("answer is %d\n", num1 / num2);
```

Another thing to think about is what would the behavior be if you substitute for doubles instead of integers.

Note that if/else statements can be nested, as in the next example:

```
/* Program w3-4.c */

#include <stdio.h>

main() {

  /* declare integer variable */
  int flag;

  /* read terminal input */
  printf("Choose Padres (=1) or Yankees (=2): ");
  scanf("%d", &flag);

  /* produce output depending on input */
  if (flag ==1)
    printf("GO PADRES\n");
  else if (flag == 2)
    printf("GO YANKEES\n");
  else
    printf("Unrecognized team\n");

}
```

The C compiler will associate an else with the closest if within a block in nested if/else statements. It might be easier to visualize what goes on the previous code if we use better indentation:

```
if(flag == 1)
  printf("GO PADRES\n");
else
  if(flag == 2)
    printf("GO YANKEES\n");
  else
    printf("Unrecognized team\n");
```

or if we separate blocks with braces:

```
if(flag == 1)
  printf("GO PADRES\n");
else {
  if(flag == 2)
    printf("GO YANKEES\n");
  else
    printf("Unrecognized team\n");
}
```

Note that poor indentation may mislead one's interpretation the following piece of code:

```
if(x > y)
  if(y < z)
    k++;
else
  j++;
```

Consider the more complex example:

```
/* Program w3-5.c */

#include<stdio.h>

main() {

  /* declare variables */
  int j = 0, k = 0, m = 0;
  double x = 3., y = 1.5, z = 1.2;

  /* conditional statements */
  if(x > y)
    if(y < z)
      k++;  /* when x > y and y < z */
    else
      m++;  /* when x > y and y >= z */
  else
    j++;    /* when x <= y */

  /* print output */
```

```
  printf("j = %d, k = %d, m = %d\n", j, k, m);

}
```

When the number of `if/else` nests increases, the `switch` statement, which we will discuss later, offers an attractive alternative.

4.2.1 The conditional operator

C also offers a *ternary* conditional operator that is sometimes a convenient alternative to the use of the `if/then` statement. The syntax is

```
  expression1 ? expression2 : expression3
```

which returns `expression2` if `expression1` is true or `expression3` otherwise. It is useful with assignments and short function calls. For example, the excerpt:

```
  if (a > b)
    z = a;
  else
    z = b;
```

can be replaced by the compact

```
  z = (a > b) ? a : b;
```

Here is an example with a function call. The code

```
  y = sqrt( x < 0 ? -x : x )
```

will evaluate $\sqrt{|x|}$.

4.3 The `switch` statement

The `switch` statement allows for more flexibility in handling alternatives. While if is good for choosing between two alternatives, it may quickly become painful to handle several alternatives. The switch statement allows multiple selection in C. Switch only works with `int`'s (hence `char`'s as well). The syntax is as follows:

```
  switch (expression) /* must evaluate to int */
  {
    case constant1:
    statements;
    break; /* causes an immediate exit from the switch */

    case constant2:
    statement sequence;
```

```
    /* no break will continue to next case */

    case constant3:
    ... ;

    default:    /* an optional default, need not be at the end */
    statements;
  }
```

When a match is found, the statements associated with the case are executed until `break` or the end of the `switch` statement is encountered. If no match is found and `default` case is provided, it will be executed; The `default` case need not be the last case.

Here is an example:

```
/* Program w3-11.c */

#include <stdio.h>

main() {

  /* declare variables */
  int a, b, result;
  char ch;

  /* reads operator */
  printf("> Select one of the operators\n");
  printf("\t+ (plus)\n\t- (minus)\n\t* (multiply)\n\t/ (division)\n");
  printf("> ");
  ch = getchar();

  /* reads two integers */
  printf("> Enter two integers: ");
  scanf("%d %d", &a, &b);

  /* do you think if you read the integers first it will work? */

  /* process operator using switch */
  switch(ch) {

  case '+':
    result = a + b;
    break;

  case '-':
    result = a - b;
```

```
    break;

  case '*':
    result = a * b;
    break;

  case '/':
    if (b == 0) {
      printf("Error: division by zero\n");
      return 1;
    }
    result = a / b;
    break;

  default:
    printf("Error: unrecognized operator '%c'\n", ch);
    return 1;

  }

  printf("> You selected operator %c\n", ch);
  printf("  %d %c %d = %d\n", a, ch, b, result);

}
```

The next code is a variation of the previous example where one of the cases is not interrupted by a break:

```
/* Program w3-11a.c */

#include <stdio.h>

main() {

  /* declare variables */
  int a, b, result;
  char ch;

  /* reads operator */
  printf("> Select one of the operators\n");
  printf("\t+ (plus)\n\t- (minus)\n\t* (multiply)\n\t/ (division)\n");
  printf("> ");
  ch = getchar();

  /* reads two integers */
  printf("> Enter two integers: ");
  scanf("%d %d", &a, &b);
```

```
  /* process operator using switch */
  switch(ch) {

  case '-':
    b *= -1;
    ch = '+'; /*
                 this is not needed for continuing to next case!
                 we just use it to print the expression correctly
              */

  case '+':
    result = a + b;
    break;

  case '*':
    result = a * b;
    break;

  case '/':
    if (b == 0) {
      printf("Error: division by zero\n");
      return 1;
    }
    result = a / b;
    break;

  default:
    printf("Error: unrecognized operator '%c'\n", ch);
    return 1;

  }

  printf("> You selected operator %c\n", ch);
  printf("  %d %c %d = %d\n", a, ch, b, result);

}
```

4.4 The `while` loop

`while` loops work by repeating a statement as long as an expression is true. The basic syntax is as:

```
  while (expression)
  statement;
```

If `expression` is false, the statement is never executed. Enclose multiple statements in braces.

Here is an example:

```c
/* Program w4-5.c */

#include <stdio.h>

main() {

  /* declare variables */
  int i;

  /* will this print from 0 to 9, 1 to 10, 0 to 10 or 1 to 9? */
  i = 0;
  while ( i++ < 10 )
    printf("%2d ", i);
  printf("\n");

  /* will this print from 0 to 9, 1 to 10, 0 to 10 or 1 to 9? */
  i = 0;
  while ( i < 10 )
    printf("%2d ", i++);
  printf("\n");

}
```

HINT: If an executable code keeps running much longer than one expected it might be caught in an infinite loop. Hit <CONTROL>+C if you wish to terminate the program.

Can you figure out what the next example does?

```c
/* Program w4-5a.c */

#include <stdio.h>

main() {

  /* declare variables */
  char c;

  /* read characters from input */
  while ( (c = getchar()) != EOF )
    printf("%c ", c);

}
```

We will now revisit example `w3-11.c` using a `while` loop to eat extra characters:

```c
/* Program w4-7.c */

#include <stdio.h>

main() {

  /* declare variables */
  int a, b, result;
  char ch;

  /* reads two integers */
  printf("> Enter two integers: ");
  scanf("%d %d", &a, &b);

  /* we will now 'eat' the remaining characters in the line, include
     \n before asking for operator */
  while ( getchar() != '\n' );

  /* reads operator */
  printf("> Select one of the operators\n");
  printf("\t+ (plus)\n\t- (minus)\n\t* (multiply)\n\t/ (division)\n");
  printf("> ");
  ch = getchar();

  /* process operator using switch */
  switch(ch) {

  case '+':
    result = a + b;
    break;

  case '-':
    result = a - b;
    break;

  case '*':
    result = a * b;
    break;

  case '/':
    if (b == 0) {
      printf("Error: division by zero\n");
      return 1;
    }
    result = a / b;
```

```
    break;

  default:
    printf("Error: unrecognized operator '%c'\n", ch);
    return 1;

  }

  printf("> You selected operator %c\n", ch);
  printf("  %d %c %d = %d\n", a, ch, b, result);

}
```

4.5 The `break` and `continue` statements

The `break` statements provides a way to unconditionally exit the innermost enclosing loop (or `switch` statement as we have seen before). The `continue` statement provides a way to reinitiate the loop, skip the rest of the code in the loop and switching the program to the test condition. These two statements work with any type of loop construction, including `for` loops.

With `while` loops, it is very common to use `break` to exit from an infinite loop. The following example shows that:

```
/* Program w4-5b.c */

#include <stdio.h>

main() {

  /* declare variables */
  char c;

  /* start infinite loop: condition is always true! */
  while ( 1 ) {

    /* read characters from input */
    if ( (c = getchar()) == EOF )
      break;

    /* echo character */
    printf("%c", c);

    if ( c != '\n' )
      /* add space only if not new line */
```

```
      printf(" ");

  }

}
```

Here is the same example modified to use the `continue` statement:

```
/* Program w4-5c.c */

#include <stdio.h>

main() {

  /* declare variables */
  char c;

  /* start infinite loop: condition is always true! */
  while ( 1 ) {

    /* read characters from input */
    if ( (c = getchar()) == EOF )
      break;

    /* echo character */
    printf("%c", c);

    if ( c == '\n' )
      /* skip the rest of the code and return to the begining of the
         loop if \n is found */
      continue;

    /* add space */
    printf(" ");

  }

}
```

4.6 The `do/while` loop

The `do/while` is a variation of the `while` loop in which the condition is evaluated after all statements have been processed. Because of this arrangement, `do/while` loops always executes the enclosed statements at least once. The syntax is as follows:

```
  do
```

```
    statement;
  while (expression);
```

Enclose multiple statements in braces.

This is one of the previous example with a `do/while` loop:

```
/* Program w4-6.c */

#include <stdio.h>

main() {

  /* declare variables */
  int i;

  /* will this print from 0 to 9, 1 to 10, 0 to 10 or 1 to 9? */
  i = 0;
  do
    printf("%2d ", i);
  while ( i++ < 10 );
  printf("\n");

  /* will this print from 0 to 9, 1 to 10, 0 to 10 or 1 to 9? */
  i = 0;
  do
    printf("%2d ", i++);
  while ( i < 10 );
  printf("\n");

}
```

4.7 The `for` loop

`for` loops allow for complete control of the loop initialization and execution. The syntax is as follows:

```
  for (initialization; condition; increment)
    statement;
```

The three expressions in the parentheses control the behavior of the for-loop:

1. The first expression performs an arbitrary initialization; it is often used to initialize an integer counter;
2. The second is a test condition that is evaluated before each possible iteration of the loop;

3. The last expression is an expression that is evaluated after each loop, before testing the loop condition. It is often used to increment a counter.

4. The initialization and the increment condition can have more than one statement.

This is a simple program that uses a `for` loop to print numbers on the terminal:

```
/* Program w4-0 */

/* This program prints numbers 1 through 10 on the terminal */
#include <stdio.h>

main() {

  int num;

  /* version 1 */
  for (num = 1; num <= 10; num++)
    printf("%d ", num);

  printf("\n");

  /* version 2 */
  for (num = 0; num < 10; num++)
    printf("%d ", num + 1);

  printf("\n");

  /* version 3 */
  for (num = 0; num < 10; )
    printf("%d ", ++num);

  printf("\n");


}
```

A slightly more complex program is next:

```
/* Program w4-1.c */

#include <stdio.h>

main() {

  /* declare integer variables */
  int i, M, sum;
```

```
  printf("Enter maximum integer: ");
  scanf("%d", &M);
  printf("M = %d\n", M);

  /* sum all integers */
  for (i = 1, sum = 0; i <= M; i++)
    sum += i;
  printf("sum = %d\n", sum);

  /* sum all integers */
  for (i = 1, sum = 0; i <= M; sum += i, i++);
  printf("sum = %d\n", sum);

  /* sum all integers */
  for (i = 1, sum = 0; i <= M; sum += i++);
  printf("sum = %d\n", sum);

  /* sum all even integers */
  for (i = 0, sum = 0; i <= M; sum += i, i += 2);
  printf("odd sum = %d\n", sum);

  /* sum all odd integers */
  for (i = 1, sum = 0; i <= M; sum += i, i += 2);
  printf("even sum = %d\n", sum);

}
```

As before, enclose multiple statements in braces. The next program computes integer powers of an number using a `for` loop:

```
/* Program w4-2.c */

/* Compute the nth integer power */

#include <stdio.h>
main() {

  /* declare variables */
  double base, p;
  int n, k;

  /* input base and exponent */
  printf("Enter base: ");
  scanf("%lf", &base);
  printf("Enter an integer exponent: ");
  scanf("%d", &n);
```

```
  /* Compute the power */
  p = 1.;
  if (n > 0)

    for (k = 0; k < n; k++)
      p *= base; /* p = p * base */

  else {

    for (k = 0; k < -n; k++)
      p *= base;

    p = 1./p;

  }

  /* print power */
  printf("%f ^ %d = %g\n", base, n, p);

}
```

The previous two examples are clearly merely educational, as there would be much more efficient ways to compute the obtained results[3]!

The next example illustrates for loops which decrement the counter:

```
/* Program w4-3.c */

/* Compute the sum: 1/i^2 for i = 1 to n. */

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

main() {

  /* declare variables */
  double sum1, sum2, infsum;
  int i, n;

  /* input integer */
  printf("Enter integer n: ");
  scanf("%d",&n);

  if ( n < 0 )
    printf("n = %d$ must be positive\n", n);
```

[3]There is an explicit formula for the sum of a sequence of integers ($\sum_{i=0}^{n} i = n(n+1)/2$). For an algorithm to compute integer powers see Appendix A.1.

```
  for (i = 1, sum1 = 0.0; i <= n; i++)
    sum1 += 1./(i*(double)i);

  for (i = n, sum2 = 0.0; i > 0; i--)
    sum2 += 1./(i*(double)i);

  infsum =  4*atan(1.)*4*atan(1.)/6.;

  printf("Infinte sum \t= %.15f\n", infsum);

  printf("Finite sum 1 \t= %.15f\n", sum1);

  printf("Error 1 \t= %.15f%%\n", 100 * fabs(infsum - sum1) / infsum);

  printf("Finite sum 2 \t= %.15f\n", sum2);

  printf("Error 2 \t= %.15f%%\n", 100 * fabs(infsum - sum2) / infsum);

}
```

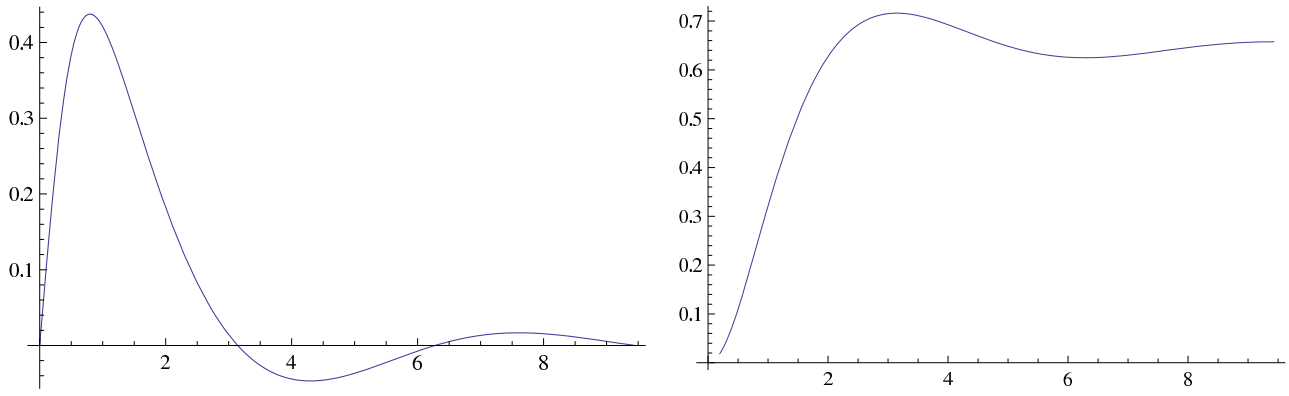
At last, we shall revisit an old friend using a `for` loop:

```
/* Program w4-5d.c */

#include <stdio.h>

main() {

  /* declare variables */
  char c;

  /* start infinite loop: condition is always true! */
  for ( ; (c = getchar()) != EOF; printf("%c ", c) )

    if ( c == '\n' )
      printf("!");

}
```

4.8 A simple application to basic numerical integration

The formulas used in this section are discussed in a bit more detail in the Appendix A.2. We shall present a simple code to perform the numerical integration of the univariate function

$$f(x) = \frac{\sin(x)}{1+x^2}$$



(a) $f(x)$ (b) $\int_0^x f(x)\,dx$

Figure 1: $f(x)$ and its integral.

on the closed interval $[a, b]$ using the trapezoidal rule. This function and its integral are shown in Figure 1. Note that f is an odd function. The interval extremes a and b and the number of segments to use n will be input from the terminal. The following program uses the approximate formula

$$\int_a^b f(x)\,dx \approx \frac{1}{n}(b-a)\left(\frac{1}{2}[f(a)+f(b)] + \sum_{i=1}^{n-1} f\left(a + \frac{i}{n}(b-a)\right)\right)$$

where the summation will be computed using a `for` loop:

```c
/* Program w4-10.c */

/* integrate f(x) = sin(x)/(1 + x^2) over a <= x <= b */

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

main () {

  /* declare variables */
  int n;
  double a, b, tmp,
    f, x, delta, sum;

  printf("Enter the extremes of the interval [a, b]: ");
  scanf("%lf %lf", &a, &b);

  printf("Enter the number of trapezoids (n): ");
  scanf("%d", &n);

  if ( a > b ) {
```

```
    /* swap a and b */
    printf("a > b, swaping a and b\n");
    tmp = a; a = b; b = tmp;
  }

  if ( n <= 0 ) {
    printf("'n' must be positive. Setting 'n = 100'\n");
    n = 100;
  }

  printf("Integrating\n  f(x) = sin(x)/(1+x^2)\non [%f, %f] "
         "using %d trapezoids...\n",
         a, b, n);

  delta = (b - a) / n;

  /* compute f(a) */
  x = a;
  f = sin(x) / (1. + (x * x));

  /* Initialize the sum */
  sum = f;

  /* compute f(b) */
  x = b;
  f = sin(x) / (1. + (x * x));

  /* add to the sum */
  sum += f;

  /* divide current sum = f(a) + f(b) by 2 */
  sum /= 2;

  /* compute the summation */
  for ( x = a + delta; x < (b - delta/2); x += delta )
    sum += sin(x) / (1. + (x * x));

  /* multiply by delta */
  sum *= delta;

 printf("Integral = %f\n", sum);

}
```

Note how the computation is spread over many small steps.