2

Control

Computation rests on two foundations: memory and control. Having developed a memory model in Chapter 1, this chapter extends the computational model with **control** statements. Such statements direct the flow of computation: the `if/else` construct enables conditional computation; and the `while` and `for` constructs facilitate iterative computation. Function calls, when combined with conditional statements, can yield even more complex control in the form of recursion.

2.1 Conditionals

The most basic control statement is the **conditional**. Consider this improvement of `divide`, which checks the input and either returns –1, indicating malformed input, or computes the quotient and remainder and returns 0, indicating a successful computation:[1]

```
1  int divide(int dividend, int divisor,
2             int * quotient, int * remainder) {
3    if (divisor <= 0 ||
4        quotient == NULL ||
5        remainder == NULL) {
6      // error: malformed input
7      return -1;
8    }
9    else {
10     *quotient = dividend / divisor;
11     *remainder = dividend % divisor;
```

[1] Returning a negative integer to indicate an error or 0 to indicate success is a custom based on this observation: "There are many ways of messing up, but only one way of getting it right." However, some libraries, including some standard C libraries, use other customs. For example, some functions may return 0 or 1 to indicate an error or successful completion, respectively.

A.R. Bradley, *Programming for Engineers*,
DOI 10.1007/978-3-642-23303-6_2,
© Springer-Verlag Berlin Heidelberg 2011

```
12      // successful computation
13      return 0;
14    }
15  }
```

Lines 6 and 12 are **comments**, which are ignored by the compiler but are intended to be useful to the reader. Lines 3–5 check if `divisor <= 0` or `quotient == NULL` or `remainder == NULL`. The operator `<=` is read as "less than or equal to" or "at most," while the operator `||` is read as "or." Because `=` is reserved for assignment, `==` is read as "equals." If any one (or more) of the predicates is true, then the **block** of code after the `if` is executed; otherwise, the block of code after the `else` is executed.

A caller could then check for an indication of an error:

```
1 int main() {
2   int q, r;
3   int errorCode = divide(7, 3, &q, &r);
4   assert (!errorCode);
5   return 0;
6 }
```

In this case, the error checking is minimal. The `!` operator is read as "not": `!0` is 1, while `!`n is 0 for any $n \neq 0$. Since an `assert` is triggered if its argument is false, which in C is 0, the assertion at line 4 is triggered precisely when `divide` returns -1, that is, when its input is malformed. While the overall effect in this particular use of `divide` is the same as in the previous chapter, the idea is that this new version of `divide` allows the caller to recover from an error if appropriate.

We have seen two **logical operators** so far: "or," `||`, and "not," `!`. The operator `&&` is read as "and." Using `&&`, `divide` can be implemented equivalently as follows:

```
1  int divide(int dividend, int divisor,
2             int * quotient, int * remainder) {
3    if (divisor > 0 &&
4        quotient != NULL &&
5        remainder != NULL) {
6      *quotient = dividend / divisor;
7      *remainder = dividend % divisor;
8      // successful computation
9      return 0;
10   }
11   else {
12     // error: malformed input
13     return -1;
14   }
15 }
```

Logical operators are also called **Boolean operators** after George Boole, whose contribution to mathematics includes the study of Boolean algebras. One particular Boolean algebra is the algebra of logical 0 and 1, also called "false" and "true," respectively. Here are some basic identities written using C syntax:

- `(!0) == 1`, `(!1) == 0`;
- when `x` is 0 or 1, `(!!x) == x`;
- `(x && y) == (y && x)`,
  `(x || y) == (y || x)`;
- `(x && (y && z)) == ((x && y) && z)`,
  `(x || (y || z)) == ((x || y) || z)`;
- `(0 && x) == 0`,
  `(1 && x) == x`;
- `(0 || x) == x`,
  `(1 || x) == 1`;
- `(!(x && y)) == (!x || !y)`,
  `(!(x || y)) == (!x && !y)`.

Developing an intuition for logical arithmetic is useful in programming because conditional statements are sometimes complex.

Exercise 2.1. Apply these identities to solve the following problems:

(a) Manipulate `!(x && (y || !z))` so that `!` is only applied to variables.
Solution. One application of the penultimate identity above, known as De Morgan's law, yields `!x || !(y || !z)`; an application of its dual, the final identity, yields `!x || (!y && !!z)`; and an application of the second identity yields `!x || (!y && z)`.
(b) Write an expression equivalent to `x || y || z` that uses only `!` and `&&`.
(c) Write your own logic manipulations and trade with your colleagues.

□

Conditional statements can extend beyond two options. Consider the following function, which computes the "sign" of an integer: it returns −1, 0, or 1 if the given integer is negative, 0, or positive, respectively:

```
1  int sign(int x) {
2    int s = 0;
3    if (x < 0)
4      s = -1;
5    else if (x == 0)
6      s = 0;
7    else
8      s = 1;
9    return s;
10 }
```

Notice that this code snippet does not use braces (`{` and `}`) for the conditional blocks. Braces are not required when a block consists of only one statement. However, one must be careful to avoid introducing bugs by accidentally omitting braces.

A function can have multiple `return` statements, a freedom that becomes relevant with control. The following is a functionally equivalent but more concise version of `sign`:

```
1 int sign(int x) {
2   if (x < 0)       return -1;
3   else if (x == 0) return 0;
4   else             return 1;
5 }
```

Notice that spacing can be used to clarify (or obscure) code.

Exercise 2.2. Modify the `swap` function of Exercise 1.19 so that it check its input and returns -1 if it is malformed and 0 otherwise.

Solution. Rather than asserting that neither `x` nor `y` is `NULL` as in Exercise 1.19, which causes the program to abort on bad input, we use an `int` return value to indicate whether the function executes successfully. If either is `NULL`, the function returns -1; otherwise, it executes normally and returns 0:

```
1  #include <assert.h>
2  #include <stdlib.h>
3
4  int swap(int * x, int * y) {
5    if (x == NULL || y == NULL) return -1;
6    int t = *x;
7    *x = *y;
8    *y = t;
9    return 0;
10 }
11
12 int main() {
13   int a = 0, b = 1;
14   int rv = swap(&a, &b);
15   assert (rv == 0);
16   assert (a == 1);
17   assert (b == 0);
18   rv = swap(&a, NULL);
19   assert (rv != 0);
20   assert (a == 1);
21   return 0;
22 }
```

The unit test implemented in `main` tests both normal and abnormal situations for `swap`. The second call to `swap` would cause the program to abort with the old version of `swap`. □

Exercise 2.3. Modify the `swap3` function of Exercise 1.20 so that it check its input and returns −1 if it is malformed and 0 otherwise. □

Exercise 2.4. Write a function that returns the absolute value of an integer variable. It should have the following prototype:

```
int abs(int a);
```

Write a unit test of `abs` in `main`.

Solution. We explore various equivalent ways of implementing this function. Given that this function is so simple, the variety in even this example suggests that, as we tackle ever more interesting programming problems, there will be ever greater freedom in the design and implementation choices.

The first implementation is verbose but straightforward:

```
#include <assert.h>

int abs(int a) {
  int x;
  if (a < 0) {
    x = -a;
  }
  else {
    x = a;
  }
  return x;
}

int main() {
  int x = -3;
  int y = abs(x);
  assert (x == y || -x == y);
  assert (y >= 0);
  x = abs(y);
  assert (y == x);
  return 0;
}
```

There are two tests in `main`: `abs` should return a nonnegative number that is equal in magnitude to the original number, and it should leave a positive number unchanged.

In this variation, we realize that we don't need a local variable:

```
int abs(int a) {
  if (a < 0)
    a = -a;
  return a;
}
```

In the final variant, we realize that we don't need to change the value of `a` at all but can instead use multiple `return` statements:

```
1 int abs(int a) {
2   if (a < 0) return -a;
3   return a;
4 }
```

□

Exercise 2.5. Write a function that computes the minimum and the maximum of two integer variables and returns them through call-by-reference parameters. It should have the following prototype:

```
1 int minmax(int a, int b, int * min, int * max);
```

Write a unit test of `minmax` in a `main` function. □

2.2 Recursion

According to the Church–Turing thesis, you have now learned all the tools necessary to compute anything that is theoretically computable—were memory and time unlimited. Does this statement surprise you? For that matter, have you ever thought about what is and is not computable? An entire branch of knowledge called **computability theory** has evolved from the pioneering work of Gödel, Church, Turing, and others.

To get a taste of just how powerful the combination of the stack, functions, and conditional statements are, let's implement a short function that computes the sum $1 + 2 + \cdots + n$, for a given positive integer n:

```
1  int sum(int n) {
2    int upto = 0;
3    // n must be positive
4    assert (n > 0);
5    if (n == 1)
6      // the sum of 1 is just 1
7      return 1;
8    else {
9      // the sum 1 + ... + n == (the sum 1 + ... + n-1) + n
10     upto = sum(n-1);
11     return upto + n;
12   }
13 }
```

Line 4 asserts that `n` is positive, which is according to the English specification of the function given above. Then, if `n == 1`, the function simply returns 1: the sum of 1 is 1. For the general case, we recognize that

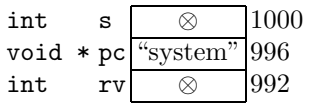
$$1 + \cdots + n = (1 + \cdots + (n-1)) + n \ ,$$

because addition is associative. Thus, to compute the sum $1 + \cdots + n$, `sum` simply needs to compute the sum $1 + \cdots + (n-1)$ and then add n, which is what lines 10–11 accomplish.

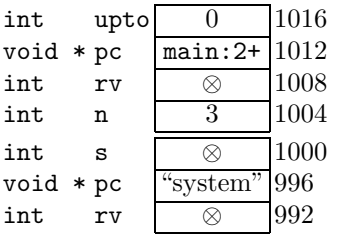
Let's trace through a call to `sum` arising in the following context:

```
1 int main() {
2   int s = sum(3);
3   return 0;
4 }
```

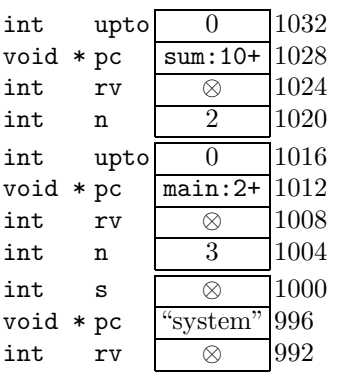
At entry, memory has the following configuration:



The call at line 2 causes `sum`'s stack frame to get pushed:



The location `main:2+` refers to the address of the machine instructions that must be executed after `sum` returns, which corresponds to the assignment of the return value to `s`. `sum(3)` executes. The second conditional block is executed because `3 != 1`. Line 10 of `sum` calls `sum` again, so that a second instance of `sum`'s stack frame is pushed:



Notice how, in the new instance, the parameter `n` is initialized to 2 and the program counter is set to be restored to line 10 of `sum` upon return.

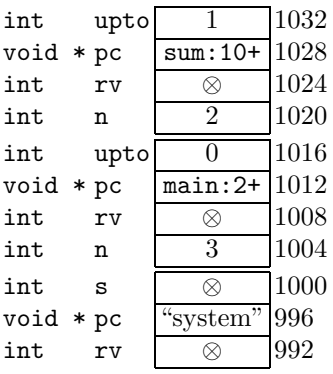
Once again, the second conditional block is executed because `2 != 1`, and another stack frame is pushed:

`int`	`upto`	0	1048
`void *`	`pc`	`sum:10+`	1044
`int`	`rv`	⊗	1040
`int`	`n`	1	1036
`int`	`upto`	0	1032
`void *`	`pc`	`sum:10+`	1028
`int`	`rv`	⊗	1024
`int`	`n`	2	1020
`int`	`upto`	0	1016
`void *`	`pc`	`main:2+`	1012
`int`	`rv`	⊗	1008
`int`	`n`	3	1004
`int`	`s`	⊗	1000
`void *`	`pc`	“system”	996
`int`	`rv`	⊗	992

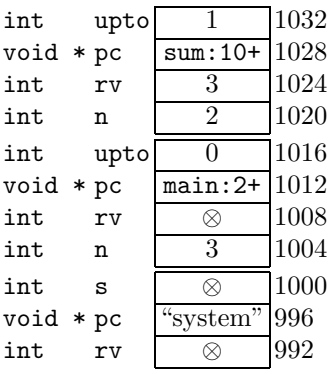
This time, the parameter is initialized to 1. Therefore, the first conditional block is executed so that the return value is set to 1:

`int`	`upto`	0	1048
`void *`	`pc`	`sum:10+`	1044
`int`	`rv`	1	1040
`int`	`n`	1	1036
`int`	`upto`	0	1032
`void *`	`pc`	`sum:10+`	1028
`int`	`rv`	⊗	1024
`int`	`n`	2	1020
`int`	`upto`	0	1016
`void *`	`pc`	`main:2+`	1012
`int`	`rv`	⊗	1008
`int`	`n`	3	1004
`int`	`s`	⊗	1000
`void *`	`pc`	“system”	996
`int`	`rv`	⊗	992

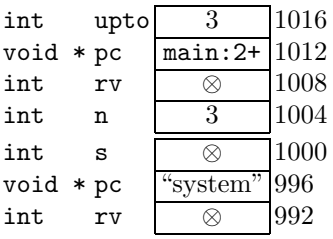
Then control returns to the calling context, where `upto` is set to the return value, and the expended stack frame is popped:



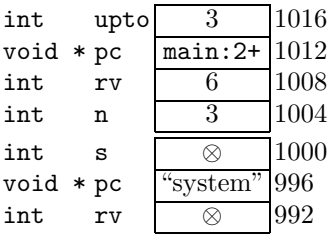
Control now proceeds to line 11, where the sum `upto + n` is computed and stored in the return value:



Then control returns to the calling context, where `upto` is set to the return value, and the expended stack frame is popped:



Control now proceeds to line 11, where the sum `upto + n` is computed and stored in the return value:



Finally, control returns to the calling context, where `s` is set to the return value, the expended stack frame is popped, and `main`'s `rv` is set to 0:

`int`	`s`	6	1000
`void *`	`pc`	"system"	996
`int`	`rv`	0	992

Execution of the program then completes.

Study this section until you understand precisely how the computer executes this program.

This example demonstrates **recursion**, which is the most powerful technique for writing programs that do an amount of work dependent on input.

Exercise 2.6. To make `sum` callable in any context, it would be best to remove the need for the assertion at line 4.

(a) Rename `sum` to `_sum`. Adding an underscore (_) at the beginning of a function name is a common naming convention to indicate that it is a function that is not intended to be called outside of a specific context.

(b) Write an entry function called `sum` with the following prototype:

```
1 int sum(int n, int * s);
```

The return value should be used to indicate whether the input is malformed, in particular if `n <= 0` or `s == NULL`. As usual, it should return 0 to indicate successful execution and a negative value to indicate an error. The sum itself should be returned via the reference `s`. After checking that the input is well formed, `sum` should call `_sum`, which should perform the main computation.

(c) Remove the protection in `_sum` to optimize the implementation.

Solution. The function `_sum` does the hard work. Unlike the original version of `sum` above, it does not protect itself against spurious input because it is not intended to be called outside of a context in which we can guarantee well formed input:

```
1 #include <assert.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 /* Helper function that computes the product.  Returns the
6  * sum 1 + 2 + 3 + ... + n.  Assumes that n > 0.
7  */
8 int _sum(int n) {
9   // Base case: the sum 1 is just 1.
10  if (n == 1) return 1;
11  // Recursive case: compute (1 + 2 + ... + (n-1)) + n.
12  return _sum(n-1) + n;
13 }
```

The function `sum` checks its input and invokes `_sum` if the input is well formed:

```
/* Interface for computing the sum
 *   1 + 2 + 3 + ... + n
 * Returns -1 if n <= 0 or s is NULL; otherwise, stores the
 * sum in the cell that s references and returns 0.
 */
int sum(int n, int * s) {
  if (n <= 0 || s == NULL)
    return -1;

  // We know that n > 0 at this point, so we can safely
  // call the helper function.
  *s = _sum(n);

  // success
  return 0;
}
```

Although the check that `n > 0` is simple, this pattern of separating the main computation from the external interface is common in situations in which the input check is more complex.

Finally, `main` tests `sum` with both well formed and malformed input. It uses the output function `printf`, which is discussed in depth in Chapter 5, to print the sum to the console:

```
int main() {
  // test the sum function
  int s, err;
  err = sum(5, &s);
  assert (err == 0);
  // print the result to the console
  printf("%d\n", s);
  // test bad input
  err = sum(-3, &s);
  assert (err != 0);
  return 0;
}
```

Compiling and running the program yields the expected output of 15:

```
$ gcc -Wall -Wextra -o sum sum.c
$ ./sum
15
```

□

Exercise 2.7. Write a function to compute the product $1 \times \cdots \times n$, for positive n. Write a `main` function to call it, and illustrate various interesting memory configurations during its execution. Use the protection and naming conventions of Exercise 2.6. □

2.3 Loops

While recursion is necessary for solving some important problems and the most natural looping structure in some widely used programming languages such as lisp and ocaml, the iteration exhibited in the `sum` example is better expressed—in C, anyway—through explicit looping control statements.

Let's revisit the problem of summing $1 + \cdots + n$, for positive integer n. This time we will use a `while` statement:

```
1 int sum(int n) {
2   assert (n > 0);
3   int i = 1, s = 0;
4   while (i <= n) {
5     s = s + i;
6     i = i + 1;
7   }
8   return s;
9 }
```

Line 2 declares a **loop counter**, `i`, that is incremented from `1` to `n` and an **accumulator**, `s`, that is initialized to `0`. Lines 4–7 execute iteratively, as long as `i <= n`. The effect is thus that every integer between `1` and `n` is added to `s` precisely once.

The stack is not the best way to visualize looping, or **iterative**, program behavior. Instead, we construct the following table for an input to `sum` of 5:

	n	i	s
	5	1	0
1	5	2	1
2	5	3	3
3	5	4	6
4	5	5	10
5	5	6	15

The first row of numbers indicates the variables' initial values. Subsequent rows indicate their values at the end of each iteration of the loop. Trace through the code and the table to verify your understanding of the computation. Explain to yourself why `sum(5)` returns `15`. What does `sum(8)` return? What about `sum(0)`?

Once again, we may not be satisfied with the possibility that calling `sum` with a nonpositive value could halt our program: such violent behavior compromises the modularity of the function. Instead, we write the following more modular and more robust function:

```
1 int sum(int n, int * s) {
2   int i = 1;
3
4   // check for well formed input
```

```
5   if (n <= 0 || s == NULL)
6     // indicate malformed input
7     return -1;
8
9   *s = 0;
10  while (i <= n) {
11    *s += i;  // short for *s = *s + i;
12    i++;      // short for i = i + 1;
13  }
14  // indicate successful execution
15  return 0;
16}
```

This implementation introduces new operators for accumulating sums. Loop counters are so prevalent in C that the language designers included the operator `++` to increment a variable by 1. Accumulation is also a frequent operation, and the `+=` operator provides a convenient shorthand. Similar operators exist for other arithmetic operations, including `--`, `-=`, `*=`, and `/=`.

Exercise 2.8. Write a version of `product` (see Exercise 2.7) that uses a `while` loop instead of recursion. Draw a table that illustrates values of its variables during execution for a reasonable input. □

The loop of `sum` follows a common pattern that motivates the `for` loop:

```
1 int sum(int n, int * s) {
2   int i;
3
4   // check for malformed input
5   if (n <= 0 || s == NULL) return -1;
6
7   *s = 0;
8   for (i = 1; i <= n; i++)
9     *s += i;
10
11  return 0;
12}
```

Lines 8–9 compile to exactly the same machine instructions as this loop:

```
1   i = 1;
2   while (i <= n) {
3     *s += i;
4     i++;
5   }
```

In general, a `for` loop of the form

```
1   for (<initialize>; <condition>; <increment>) {
2     <body>
3   }
```

is exactly the same as a `while` loop of the form

```
<initialize>
while (<condition>) {
  <body>
  <increment>
}
```

Programmer preference dictates when to use a `while` statement and when to use a `for` statement. Readability is the goal.

Exercise 2.9. Rewrite the `product` function of Exercise 2.8 using a `for` loop. □

Exercise 2.10. Write a function to compute the power a^n, where $n \geq 0$. It should have the following prototype:

```
/* Sets *p to the n'th power of a and returns 0, except
 * when n < 0 or p is NULL, in which case it returns -1.
 */
int power(int a, int n, int * p);
```

Write a unit test in a `main` function to test various values. The following code sequence illustrates how to use `printf` to provide informative output:

```
int x = 3, y = 5, pow;
power(x, y, &pow);
printf("%d^%d = %d\n", x, y, pow);
```

□

Exercise 2.11. Mathematical sequences can be computed using loops. Consider, for example, the following sequence:

$$a_0 = 1 \quad \text{and} \quad a_{i+1} = 2 \cdot a_i + 1 \text{ for } i > 0 \ ,$$

whose first elements are $1, 3, 7, 15, 31, 63, \ldots$. This function returns the nth element:

```
int seq(int n) {
  int i, a = 1;
  for (i = 1; i <= n; i++)
    a = 2*a + 1;
  return a;
}
```

For example, `seq(0)` returns 1, `seq(1)` returns 3, and `seq(4)` returns 31.

Write functions to compute the nth elements of the following sequences:

(a) $a_0 = 1$ and $a_{i+1} = 3 \cdot a_i + 2$ for $i > 0$.
(b) $a_0 = 59$ and $a_{i+1} = a_i/2 + 1$ for $i > 0$, where / denotes integer division; in C, use `/`. For example, $3/2 = 1$. The first elements of the sequence are $59, 59/2 + 1 = 29 + 1 = 30, 16, 9, 5, 3, \ldots$.

(c) $a_0 = 1$, $a_1 = 1$, and $a_{i+1} = a_{i-1} + a_i$ for $i > 1$. The first elements of the sequence, called the Fibonacci sequence, are $1, 1, 2, 3, 5, 8, \ldots$.
Solution. This function needs to remember the previous two values:

```
int seq(int n) {
  int i, a = 1, b = 1;
  for (i = 2; i <= n; i++) {
    int t = b;  // temporary variable
    b = a + b;
    a = t;
  }
  return b;
}
```

Verify that this function indeed returns the nth element of the sequence for various n.

(d) $a_0 = 0$, $a_1 = 2$, and $a_{i+1} = 2 \cdot a_{i-1} - a_i$ for $i > 1$.
(e) $a_0 = 7$, $a_1 = 11$, and $a_{i+1} = -a_{i-1} + a_i$ for $i > 1$.
(f) $a_0 = 1$, $a_1 = 1$, $a_2 = 1$, and $a_{i+1} = a_{i-2} + a_i$ for $i > 2$.

□

Exercise 2.12. Mathematical series can be computed using loops. Consider, for example, the following sequence:

$$a_0 = 1 \quad \text{and} \quad a_{i+1} = 2 \cdot a_i + 1 \text{ for } i > 0 \ .$$

The corresponding series is constructed by computing the partial sums:

$$a_0, \ \sum_{j=0}^{1} a_j, \ \sum_{j=0}^{2} a_j, \ \sum_{j=0}^{3} a_j, \ \ldots \ .$$

Since the first elements of the sequence are $1, 3, 7, 15, 31, 63, \ldots$, the first elements of the corresponding series are $1, 1+3 = 4, 1+3+7 = 11, 26, 57, 120, \ldots$. This function returns the nth element of the series:

```
int series(int n) {
  int i, a = 1, sum = 1;
  for (i = 1; i <= n; i++) {
    a = 2*a + 1;
    sum += a;
  }
  return sum;
}
```

For example, `series(0)` returns 1, `series(1)` returns 4, and `series(4)` returns 57. Write similar functions to compute the nth elements of series corresponding to the sequences of Exercise 2.11. □

More complex control patterns will come after we have studied more complex data structures. However, all control builds on conditionals, loops, and occasionally recursion.



http://www.springer.com/978-3-642-23302-9

Programming for Engineers
A Foundational Approach to Learning C and Matlab
Bradley, A.R.
2011, XIV, 238 p., Hardcover
ISBN: 978-3-642-23302-9