C Programming under Linux

P2T Course, Martinmas 2003–4
C Lecture 4

Dr Ralf Kaiser

Room 514, Department of Physics and Astronomy

University of Glasgow

r.kaiser@physics.gla.ac.uk

Summary

- More Operators
- Control Statements

`http://www.physics.gla.ac.uk/~kaiser/`

Operators for Shortcuts

- C includes a large number of special-purpose operators, some of which allow to use shortcuts for frequently used operations.
- The most popular one is the increment operator `++`. It allows to replace a statement like `counter = counter + 1;` with the shortcut `counter++`.
- Similar is the decrement operator `--` that decreases the value of a variable by 1.

Operator	Shortcut	Equivalent Statement
`++`	`x++ ;`	`x = x + 1;`
`--`	`x-- ;`	`x = x - 1;`
`+=`	`x += 2;`	`x = x + 2;`
`-=`	`x -= 2;`	`x = x - 2;`
`*=`	`x *+= 2;`	`x = x * 2;`
`/=`	`x /+= 2;`	`x = x / 2;`
`%=`	`x %+= 2;`	`x = x % 2;`

The Wonderful World of ++

- The increment and decrement operators come in two different flavours, the prefix form `++variable` and the postfix form `variable++`. (By now you also know where C++ comes from...)
- The two forms lead to different results; the prefix first increments and then evaluates the expression, the postfix first evaluates the expression and then increments:

```
number = 5;
result = number++;

result is 5
```

```
number = 5;
result = ++number;

result is 6
```

- Easier to read would be:

```
number = 5;
number++;
result = number;
```

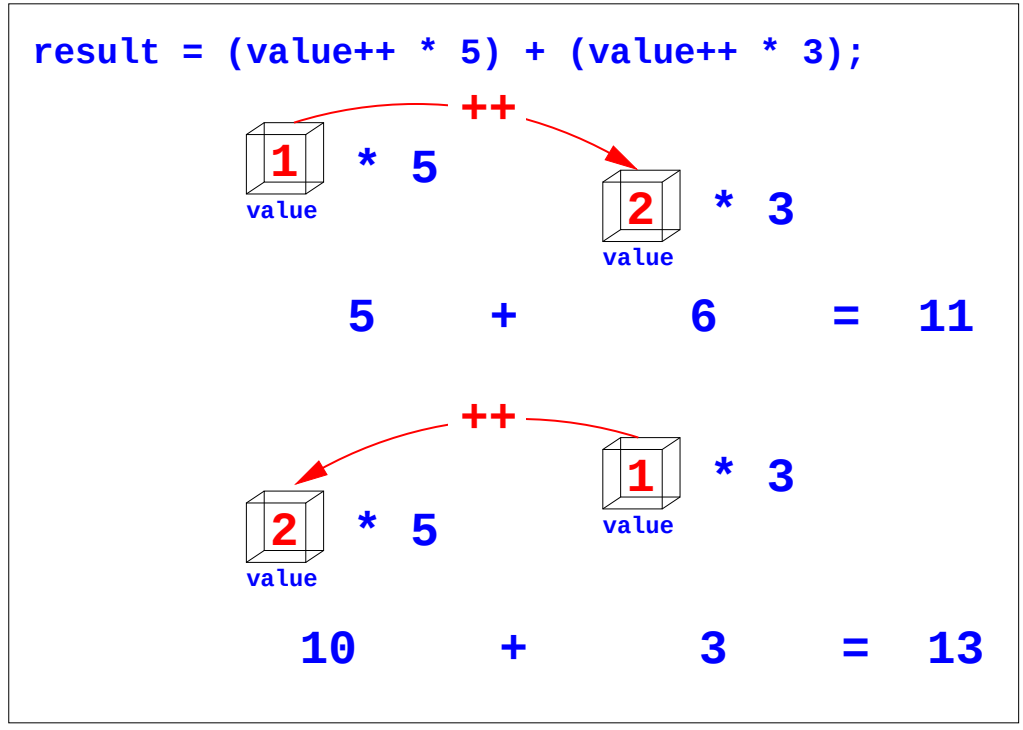
- Compact code is a holdover from the time when storage space cost a lot of money. And some people think it's cool.

The Wonderful World of ++

- Consider the code fragment

```
number = 1;
result = (number++ * 5) + (number++ * 3)
```

The result (11 or 13) actually depends on the compiler:



- Try to avoid using ++ in the middle of expressions, then you don't have to worry about this.

Control Statements

- So far our programs have been linear, i.e. they execute in a straight line, one statement after another.
- The statements we were dealing with were assignment statements.
- Now we are going to introduce control statements,i.e. branching statements and looping statements that change the control flow of a program.
- Branching statements cause a section of code to be executed or not.
- Looping statements are used to repeat a section of code a number of times or until some condition occurs.

The `if` Statement

simple `if` statement:

```
if (condition)
    statement;
```

`if - else if - else` statement:

```
if (condition1){
    statement1;
    statement2;
} else if (condition2){
    statement3;
    statement4;
} else
    statement5;
```

- If the condition is true (non-zero), the statement will be executed.
- If the condition is false (0), it will not be executed.
- Multiple statements may be in curly braces.
- `else` allows a statement to be executed if the condition is not fulfilled.

Relational Operators

- Now how do we formulate the condition for an `if` statement ? In principle, everything that returns a value that is 1 (true) or 0 (false) will work. (Actually, non-zero for true).
- Practically, this brings us to yet another set of operators, because we will mostly use relational operators:

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

- A simple `if` statement might look like this:

```
if (total_owed <=0)
    printf("You owe nothing.\n");
```

`if` - Example

Read in an amount and print out a message (`owe0.c`)

```
#include <stdio.h>
char  line[80];          /* input line */
int   balance_owed;      /* amount owed */

int main()
{
    printf("Enter number of dollars owed:");
    fgets(line, sizeof(line), stdin);
    sscanf(line, "%d", &balance_owed);

    if (balance_owed = 0)
        printf("You owe nothing.\n");
    else
        printf("You owe %d dollars.\n", balance_owed);

    return (0);
}
```

Output:

```
Enter number of dollars owed:42
You owe 0 dollars.
```

`if` - Example cont.

- The `fgets` and `sscanf` statements are alright, there is no problem with the length of the character array, so why does the program return `0` ?
- The program illustrates one of the most common errors in C programming.

  '=' is not the same as '=='.

  '=' is the assignment operator. '==' is a relational operator that compares two values and returns true/false.
- In the case of our program the statement `if (balance_owed = 0)` assigns the value 0 which is then always printed.
- Correct would be the statement `if (balance_owed == 0)`

How not to use `strcmp`

- The function `strcmp` compares two strings. It returns `0` if they are equal or non-zero if they are different. Obviously, it can be used in the condition of an `if` statement.

```
if (strcmp(string1, string2) == 0);
    printf("Strings equal\n");
else
    printf("Strings not equal\n");
```

- It may be tempting to write compact code and leave the `== 0` bit out:

```
if (strcmp(string1, string2));
    ...
```

  However, `strcmp` is counter-intuitive in this case, because it returns `0` (i.e. false, not true), when both string are equal.

The `while` Statement

- The `while` statement is used when a program needs to perform repetitive tasks. The general syntax is:
```
while (condition)
    statement;
```
- The program will repeat the execution of the statement inside the `while` until the condition becomes false (0).
- If the condition is initially false, it will never be executed.
- If the condition is always true, i.e. `while (1)`, the loop will loop forever (when nothing else happens...).
- As with `if`, multiple statements can be enclosed in curly braces.

`while` - Example

Print out all Fibonacci Numbers below 100 (`fib.c`).

```
#include <stdio.h>
int   old_number;     /* previous Fibonacci number */
int   current_number; /* current Fibonacci number */
int   next_number;    /* next number in the series */

int main()
{
    /* start things out */
    old_number = 1;
    current_number = 1;

    printf("1\n");    /* Print first number */

    while (current_number < 100) {

        printf("%d\n", current_number);
        next_number = current_number + old_number;
        old_number = current_number;
        current_number = next_number;
    }
    return (0);
}
```

`while` - Example cont.

- Fibonacci numbers are a series that is defined by

$$f_n = f_{n-1} + f_{n-2} \qquad f_0 = 1, f_1 = 1$$

  so the first Fibonacci numbers are 1, 1, 2, 3, 5, 8,.... They appear in nature e.g. in the number of left- and right-turning spirals on the bottom of some pine cones.

- In our C code, the above equation is implemented as

```
old_number = 1;
current_number = 1;

next_number = current_number + old_number;
old_number = current_number;
current_number = next_number;
   }
```

- We could also chose more 'mathematical' names, like `f_n`, `f_n_1` and `f_n_2` - the question is: What will be easier to understand for somebody else or after some time.

The Statements `break` and `continue`

- A `while` loop can be exited when the condition after the `while` becomes false (0). Alternatively, any loop can be exited at any point through the use of a `break` statement.
- Usually, `break` appears in combination with `if`:

```
if (condition)
    break;
```

- Often, `break` is used in an endless `while` loop, to exit once a condition is fulfilled:

```
while (1) {
    if (condition)
        break;
{
```

- The `continue` statement is similar to `break`, but instead of terminating the loop, `continue` starts re-executing the body of the loop from the top.

`break` - Example

Add numbers until '0' is entered, print total (`total.c`).
(Listing leaving out #include statements and variable declarations.)

```
int main()
{
    total = 0;
    while (1) {
        printf("Enter # to add \n");
        printf("  or 0 to stop:");

        fgets(line, sizeof(line), stdin);
        sscanf(line, "%d", &item);

        if (item == 0)
            break;

        total += item;
        printf("Total: %d\n", total);
    }
    printf("Final total %d\n", total);
    return (0);
}
```

Output:

```
Enter # to add
  or 0 to stop:20
Total: 20
Enter # to add
  or 0 to stop:1
Total: 21
Enter # to add
  or 0 to stop:678
Total: 699
Enter # to add
  or 0 to stop:0
Final total 699
```

`continue` - Example

Now only add positive numbers, count negative ones (`totalb.c`).
(Code fragment, only `while` loop and `printf` statements from.)

```
while (1) {
    printf("Enter # to add\n");
    printf("  or 0 to stop:");

    fgets(line, sizeof(line), stdin);
    sscanf(line, "%d", &item);

    if (item == 0)
        break;

    if (item < 0) {
        ++minus_items;
        continue;
    }
    total += item;
    printf("Total: %d\n", total);
}
printf("Final total %d\n", total);
printf("with %d negative items omitted\n",
                minus_items);
```

Output:

```
Enter # to add
  or 0 to stop:-2
Enter # to add
  or 0 to stop:22
Total: 22
Enter # to add
  or 0 to stop:1459890
Total: 1459912
Enter # to add
  or 0 to stop:0
Final total 1459912
with 1 negative items omitted
```

The `for` Statement

- The `for` statement allows to execute a block of code for a specified number of times. The general form of the `for` statement is:

```
for (initial-statement; condition; iteration-statement){
    statement-1;
    ...
    statement-n;
}
```

- This is equivalent to a `while` loop of the shape

```
initial-statement;
while (condition) {
    statement-1;
    ...
    statement-n;
    iteration-statement;
}
```

- The `iteration-statement` is most of the time a counter like `++counter`. It is conventional in C to start counters with 0 (like the numbering of array elements); i.e. count from 0 to 4, not from 1 to 5.

`for` - Example

Print Celsius to Fahrenheit conversion chart for 0 to 100 Celsius (fahrenheit.c).

```
#include <stdio.h>
/* the current Celsius temperature we are working with */
int celsius;
int main() {
    for (celsius = 0; celsius <= 100; ++celsius)
        printf("Celsius:%d Fahrenheit:%d\n",
            celsius, (celsius * 9) / 5 + 32);
    return (0);
}
```

What would happen if we accidentally add a semi-colon at the end of the `for`-statement ?

```
    for (celsius = 0; celsius <= 100; ++celsius);
```

Answer: The program would only print

```
Celsius:101 Fahrenheit:213
```

The `switch` Statement

The `switch` statement is similar to a chain of `if/else` statements. The general form of the `switch` statement is:

```
switch (expression) {
    case constant1:
        statement
        ...
        break;
    case constant2:
        statement
        ...
        /* Fall through */
    case constant3:
        statement
        ...
        break;
    default:
        statement
        ...
        break;
}
```

- `switch` evaluates the value of an expression and branches to one of the `case` labels. Duplicate labels are not allowed. The expression must evaluate an integer, character or enumeration.
- The `case` labels can be in any order and must be constants.
- The `default` label can be put anywhere in the `switch`.
- If no `case` matches and no `default` exists, the `switch` does nothing.

The `switch` Statement cont.

- A `break` statement inside a `switch` tells the computer to continue execution after the `switch`. If a `break` statement is not there, it will continue with the next statement, or fall through to the next statement.
- To make sure that this is intentional, it should be marked by a comment.
- The last `case` statement does not need a `break`, but should get one anyways.
- While a `default` is not necessary and if present can be anywhere, it should always be there and it should be the last statement. At least it should be present as an empty statement that contains only a `break`.

`switch` - Example

Select cases of different operators; code fragment only.

```
switch (operator) {
       case '+':
           result += value;
           break;
       case '-':
           result -= value;
           break;
       case '*':
           result *= value;
           break;
       case '/':
           if (value == 0) {
               printf("Error:Divide by zero\n");
               printf("   operation ignored\n");
           } else
               result /= value;
           break;
       default:
           printf("Unknown operator %c\n", operator);
           break;
       }
```

`break`, `continue` and `exit`

- `break` has two uses: Inside a `switch`, `break` causes the program to go to the end of the switch. Inside a `for` or `while` loop, `break` causes a loop exit.
- `continue` is valid only inside a loop and will cause the program to go to the top of the loop.
- In the case of a `continue` statement inside a `switch` that in turn is inside a loop, the `continue` will act on the loop.
- If a `break` statement is inside a `switch` that in turn is inside a loop, it will act on the `switch`.
- The `exit` function requires the inclusion of the header `<stdlib.h>`. It is normally called as `exit(0);` and causes the normal termination of the program, equivalent to reaching the closing curly brace of the `main` function. It will exit from anywhere in the program.