Introduction to Scientific and Engineering Computation

(BIL 102E)

LECTURE 6

Decision Making

The programs we have been writing so far were somewhat dull as they were always following the same sequence of instructions.

One of the main reasons why computers are such powerful tools is the fact that they can decide which instructions to follow depending on simple comparisons.

In C language, the **if** statement helps us to write segments of code that will be executed only under certain conditions.

The syntax for the simplest **if** statement is as follows:

if (*expression*) **{**
  [Block of C Statements]
}

Here *expression* can be *relational expressions*, *integer values* or *variables* or sets of smaller logical expressions that are connected by *logical operators*. In execution time, if *expression* is **true** (nonzero) then the block of C statements in the **if** construct are executed otherwise (*expression* evaluates to zero) they are skipped. For example,

```
if (x < 3) {
  printf("x is less than 3! \n");
}
```

prints the message "x is less than 3!" on the screen only when x <3.

Relational Expressions

Relational expressions have the following syntax:

expression1 **relation_operator** *expression2*

Here, *expression1* and *expression2* are both arithmetic expressions and **relation_operator** is one of the following:

Relation Operator	Meaning
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
!=	Not equal to

The result of a relational operation is an integer. If the relation is true then the result is (generally) 1, otherwise 0.

For example,

3 < 2*5-4 → 1, *true*

2.1 +3.0 == 5.0 → 0, *false*

- Note that a relation operation basically compares the results of two expressions.
- All arithmetic operators have higher priorities than the relational operators, hence arithmetic operations are done first.
- You can have more than one *relation operator* in the same *expression*. For example, the expression (0 < x <5) is **valid** in C. However, this may introduce confusion and should be avoided when possible.

For example,

3 < 2 < 1 → (3 < 2) < 1

→ 0 < 1

→ 1 *(true)*

Logical Operations

- It is possible to connect logical expressions using logical operations, which usually take the following syntax:
- *L1 <logic_operator> L2*
- Here L1 and L2 are logic expressions and the *logic operator* can be && (**and**), or || (**or).**
- The result of a && **(and)** operation is 1 (*true)* only if both logical expressions (L1 and L2) are nonzero (*true)*.
- The result of a || **(or)** operation is 1 (*true)* if one of the logical expressions (L1 or L2) is nozero (*true)*.

Truth tables for && and || operations:

0 && 0 → 0	0 || 0 → 0
0 && 1 → 0	0 || 1 → 1
1 && 0 → 0	1 || 0 → 1
1 && 1 → 1	1 || 1 → 1

Examples:

(3 < 5) **&&** (2.5 != 7.2)

→*1* **&&** 1

→*1*

(2 * 3 – 5 >= 0) || (2 – 7 / 4 > 1)

→ ((2*3) – 5 >= 0) || (2 – 7 / 4 > 1)

→ ((6-5) >= 0) || (2 – 7 / 4 > 1)

→ (1 >= 0) || (2 – 7 / 4 > 1)

→ 1 || (2 – (7 / 4) > 1)

→ 1 || ((2 – 1) > 1)

→ 1 || (1 > 1)

→ 1 || 0

→ 1

There is also a unary logic operator ! (**not)** which has the following syntax:

!L1

Here L1 is a logic expression and the result of ! (**not)** operation is the reverse of L1. That is, if L1 is nonzero (*true)* the result is 0 (*false)* and if otherwise the result is 1 (*true*).

For example:

!((2<3) && (3 < 1))
→ !(1 **.and.** 0)
→ !0
→ 1

Priorities

Arithmetic operators and relational operators have higher priorities than logical operators. The order of priorities among logical operators is as follows:

Operator	Priority
!	Higher
&& ||	Lower

It is possible to overwrite priorities by use of parentheses.

Write a program that asks the age of the user and if the user is between 13 and 19 it prints out “You are a teenager.”

```
#include <stdio.h>
main()
{
  int age;
  printf("Your age: ");
  scanf("%d",&age);
  printf("Age : %d\n", age);
  if((age>=13) && (age<=19)) {
        printf("You are a teenager!\n");
  }
}
```

```
Your age: 15
Age : 15
You are a teenager!
```


The use of **else** statement

Sometimes we want our program to do something under a certain condition and if that condition is **not** met we want our program to carry out some other tasks. The **else** statement can be used for this purpose. The syntax of the **if** statement in this case is as follows:

if (*expression*) **{**
 [First block of C statements]
}
else {
 [Second block of C statements]
}

Here, if the *expression* is nonzero (*true*) the first block otherwise the second block of C statements will be executed.

Example

The following program prints out "You are a teenager.", if the user is between 13 and 19, and if otherwise it prints out "You are NOT a teenager.".

```
#include <stdio.h>
main()
{
  int age;
  printf("Your age: ");
  scanf("%d",&age);
  printf("Age : %d\n", age);
  if((age>=13) && (age<=19))
    printf("You are a teenager!\n");
  else
    printf("You are NOT a teenager!\n");
}
```

Note that since there are only one statements in statement blocks we did not have to use the braces.

Nesting **if** Statements

It is possible to use **if** statements inside other **if** statements. For example,

```
#include <stdio.h>
main()
{
  int age;
  printf("Your age: ");
  scanf("%d",&age);
  printf("Age : %d\n", age);
  if(age<13)
       printf("You are a kid!\n");
  else
       if((age>=13) && (age<=19))
          printf("You are a teenager!\n");
        else
          printf("You are NOT a teenager!\n");
}
```


The following program prints out
You are a kid, if the user is younger than 13
You are a teenager, if the user is between 13 and 19
You are young, if the user is between 20 and 34
You are not young, if the user is older than 34

```
#include <stdio.h>
main()
{
  int age;
  printf("Your age: ");
  scanf("%d",&age);
  if(age<13)
       printf("You are a kid!\n");
  else if(age<20)
       printf("You are a teenager!\n");
  else if(age<35)
    printf("You are young!\n");
  else printf("You are not young!\n");
}
```

Note that in each run of the program only **one** block in the if statement is executed!

switch Statement

The logical expressions in nested **if** statements can become very messy when there are many decisions to make. The **switch** statement can be used to make decisions where to jump in the program depending on the value of an expression. The syntax is as follows:

switch (expression) {
case *case_expression_1:*
  [First Block of C Statements]
case *case_expression_2:*
  [Second Block of C Statements]
  ⋮
default:
  [Default Block of C Statements]
}

Here the execution of the program continues from the case expression corresponding to the value expression.

Case expressions are constant expressions evaluating to often an integer or character value.

Normally after jumping the proper block of C statements the execution continues for all the remaining program blocks. Using break statements at the end of each block it is possible to prevent this.

If there are no matching case expressions than the execution jumps to the default block.

```c
#include <stdio.h>
main()
{
  int category;
  printf("Enter category (1-3): ");
  scanf("%d",&category);
  switch(category) {
        case 1:
           printf("Category 1\n");
           break;
        case 2:
           printf("Category 2\n");
           break;
        case 3:
           printf("Category 3\n");
           break;
        default:
           printf("Wrong category number!\n");
  }
}
```

```
Enter category (1-3): 5
Wrong category number!
```

```
Enter category (1-3): 2
Category 2
```

*Without **break** statements the output would be:*
Category 2
Category 3
Wrong category number!

Using ++ or -- operators in **if** statements

Although not recommended, it is possible to use ++ or -- operators in if statements.

```c
#include <stdio.h>
main() {
  int i=1;
  if(i++ == 1)
        printf("%d \n", i);
  printf("%d \n", i);
}
```

First comparison then increment is done.
So, if block is executed.

```
2
2
```

```c
#include <stdio.h>
main() {
  int i=1;
  if(++i == 1)
        printf("%d \n", i);
  printf("%d \n", i);
}
```

First increment then comparison is done.
If block is **not** executed.

```
2
```