

Selection statements

Selection statements allow the program to select different execution paths depending on data. There are two selection statements in C: **if** and **switch**.

The **if** statement chooses execution path by testing a logical expression,

if (expression) statement else statement

Any valid statement is allowed, including a new **if** statement.

Example, computing $absx = |x|$:

```
if ( x > 0 )
    absx = x;
else
    absx = -x;
```

Marco Kupiainen
marcok@nada.kth.se



Logical expressions

Logical expressions are formulated using (a combination of) relational, equality, arithmetic and / or logical operators.

Each logical expression evaluates to either false (0) or true (1, nonzero).

Relational operators	
Symbol	Meaning
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

Logical operators	
Symbol	Meaning
!	logical negation
&&	logical AND
	logical OR



Marco Kupiainen
marcok@nada.kth.se

Logical expressions (cont.)

Equality operators	
Symbol	Meaning
==	equal to
!=	not equal to

Example: $(1 < n \leq 100 \text{ or } n = 150)$ can be written `(1<n && n<=100 || n==150)`.

It is important not to confuse equality (==) with assignment (=).

Mistakes can result in relevant data being overwritten and/or wrong execution path begin chosen.

`(i == j)` tests whether *i* is equal to *j*.

`(i = j)` assigns *i* the value of *j* and is evaluated as true if *j* is nonzero.



Marco Kupiainen
marcok@nada.kth.se

The switch statement

```
switch ( expression ) {
    case constant expression : statements
    :
    case constant expression : statements
    default : statements
}
```

All expressions must return integer (or character) values.

The constant expressions (or *case labels*) may not contain variables or function calls.

For simple use of **switch** the last statement in each case group should be **break**;

(To see what happens otherwise one needs to discuss jump statements which are not part of this course.)



Marco Kupiainen
marcok@nada.kth.se

The switch statement (cont.)

The switch statement can be used to avoid cascaded if statements.

Example: Solve polynomial equation of degree 1 or 2.

Cascaded if statements:

```
if (degree == 1)
    { /* Solve eqn of degree 1 */ }
else if (degree == 2)
    { /* Solve eqn of degree 2 */ }
else
    { /* Print error message */ }
```

switch statement:

```
switch (degree) {
    case 1: /* Solve degree 1 */; break;
    case 2: /* Solve degree 2 */; break;
    default: /* Print error message */;
}
```



do loops

```
do statement while ( expression )
```

do loops are essentially while loops, but the controlling expression is tested after the statement is executed.

Example, do loop written as while loop:

```
int i=1;
while ( i ) {
    statement
    i = expression;
}
```

Example, powers of 2 revisited:

```
int n=0;
int pow=1;
do {
    printf("%3d %3d\n", n, pow);
    n = n+1;
    pow *= 2;;
} while (pow < 100)
```



Iteration statements

Iteration statements (or loops) are used to repeatedly execute a statement. There are three iteration statements in C: while, do and for.

```
while ( expression ) statement
```

while executes the statement as long as the expression is true.

Example, computing powers of 2 smaller than 100:

```
int n = 0;
int pow = 1;
while (pow < 100) {
    printf("%3d %3d\n", n, pow);
    n = n + 1;
    pow *= 2;
}
```



for loops

```
for ( expr1 ; expr2 ; expr3 )
    statement
```

The syntax of for loops is more complicated than that of while and do loops, but compilers will often produce faster executables.

The expressions should be interpreted as follows:

- expr1 Executed before starting iteration.
- expr2 Iterate while expression is true.
- expr3 Executed at end of each iteration

Example, for loop written as while loop:

```
expr1;
while ( expr2 ) {
    statement
    expr3;
}
```



for loops (cont.)

Example, computing $n!$:

```
nfac = 1;
for (i=1 ; i<=n ; i++)
    nfac = nfac * i;
```

expr1 and *expr3* may be empty or contain several statements separated by commas.

Example, powers of 2 again:

```
for (n=0,pow=1 ; pow<100 ; n++,pow*=2)
    printf("%3d   %3d\n", n, pow);
```

A loop can be terminated by using the **break** statement within the loop body even if the controlling expression is true.



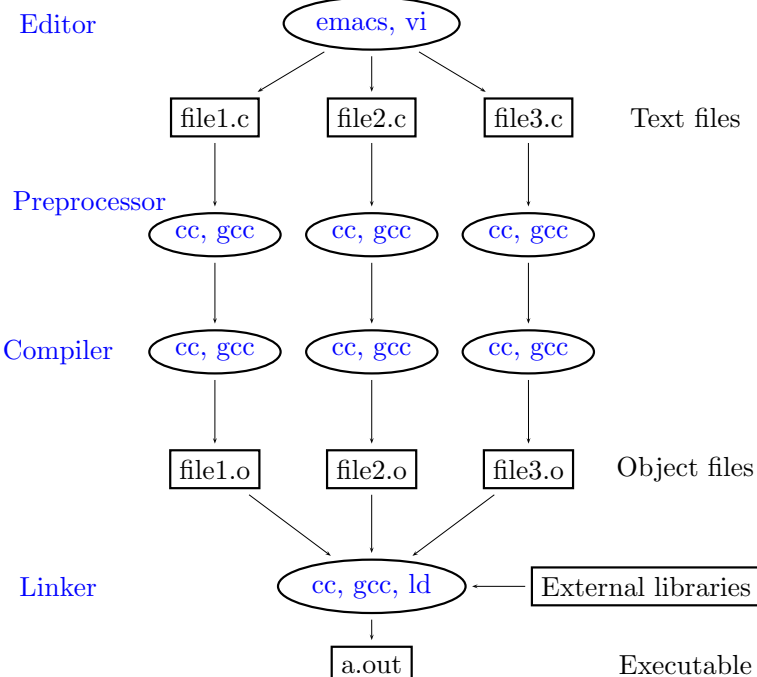
From problem to executable

Thinking	A problem is formulated, algorithms devised and data structures chosen.
Programming	The C program is written using an editor (emacs, xemacs, vi).
Preprocessing	Preprocessor directives are interpreted and a preprocessed C file generated (gcc -E, cc -E, cpp).
Compiling	The preprocessed source code is translated to object code (machine instructions) (gcc -c, cc -c).
Linking	The object code is linked with external libraries to produce an executable (gcc, cc, ld).
Debugging	Make sure the program works as expected. Correct possible mistakes (gdb).

The first step is the most important! By considering the problem carefully the debugging time can be significantly reduced.



Compiling & linking



Compiling & linking (cont.)

There are two different C compilers at Nada: **cc** (Sun) and **gcc** (GNU). Both can be used for preprocessing, compiling and linking small programs in a single command.

`gcc smallfile.c -o runme` generates an executable `runme` from the source code in `smallfile.c`.

The object files are linked with the most common libraries, including **stdio** and **stdlib**.

If compiling a program is tedious (several different files, many options, different programming languages, ...) it is convenient to use *makefiles*. These will be covered in lecture 9.



Compiler & linker options

Additional instructions to the compiler are given as command line options.

Option		
<code>gcc</code>	<code>cc</code>	Effect
<code>-c</code>	<code>-c</code>	Compile source files but do not link
<code>-llib</code>	<code>-llib</code>	Link with library <i>lib</i>
<code>-Ldir</code>	<code>-Ldir</code>	Add <i>dir</i> to library search path
<code>-Idir</code>	<code>-Idir</code>	Add <i>dir</i> to include file search path
<code>-otgt</code>	<code>-otgt</code>	Name executable <i>tgt</i> (default: <code>a.out</code>)
<code>-O2</code>	<code>-fast</code>	Compile with optimisation
<code>-Wall</code>	<code>-v</code>	Print “extra” warning messages

There are many more useful options.

Give the command `man gcc` (or `man cc`) at the prompt for a complete list.



Compiling – examples

Compile `program.c` and generate executable `target` with optimisation,

```
> gcc program.c -O2 -o target
```

Same example, but linked with math library

```
> gcc program.c -O2 -o target -lm
```

Compile files `main.c` and `function.c` and link to obtain executable `run`

```
> cc -c main.c
```

```
> cc -c function.c
```

```
> cc function.o main.o -o run -lm
```

Userdefined include files and libraries,

```
> cc program.c -L~/mylibraries
```

```
-I~/myincludefiles -lmylib
```

(*Note:* `>` is used to indicate that the command is given at the prompt, it should not be typed.)

