Plan

- Basic data types: declaration, assignment
- Arithmetic expressions
- Formatted input and output
- Relational and logical expressions
- Selection statements
- Iteration statements
- Compiling & linking

A (simple) generic C program

```
preprocessor directives

return-type function_1(parameters) {
  declarations
  statements
}

/*   .
     . (This is a C comment)
     .
*/
return-type function_n(parameters) {
  declarations
  statements
}

int main(int argc, char **argv) {
  declarations
  statements
}
```

Statements

The major part of a C program consists of *statements*. A statement is a command which is executed when the program runs.

All statements must be terminated by a semi-colon, e.g.

```
area = pi * radius * radius;
```

A statement can span several lines,

```
area = pi *
       radius * radius;
```

Several statements can be collected in a *compound statement* which is treated as one statement by the compiler. This is accomplished by braces,

```
{
  area = pi * radius * radius;
  circumference = 2 * pi * radius;
}
```

Basic types

All variables in C must have a type, which specifies what kind of data it will hold. A variable must be declared before use.

Some basic types:

Type	**Description**
`int`	Integer values
`float`	Floating point numbers
`double`	Floating point numbers (higher accuracy, larger numbers)
`char`	Character

There are different variants of the basic types, e.g. `unsigned int` which can only hold non-negative integers, and `long int` which can hold larger integers.

In addition to the basic types C allows the user to specify his/her own data types.

Declaration of variables

Each program / function should begin with a declaration of variables. For example

```
main() {
  int   i, j;
  double f;

  /* statements */
}
```

This declaration defines two integers, `i` and `j`, and a floating point number `f` which can be used in the program.

Note: C is case-sensitive.

A variable which is not supposed to change value can be declared as a constant, e.g.

```
const double pi = 3.14;
```

This improves efficiency and simplifies debugging.

Arithmetic expressions

A variable is assigned a value using the operator `=`.

C supports all basic arithmetic operators.

- `a = -b;`
- `a = b + c;`
- `a = b - c;` (equivalent to `a=b+-c;`)
- `a = b * c;`
- `a = b / c;`

Operator precedence as in mathematics. The statement
`a = (1 + 2) * 3 - 4;`
will result in `a` being assigned the value 5.

Additional mathematical functions declared in `math.h` (trigonometric, exponential, power, ...).

Compound assignment

When one is only interested in updating the value of a variable it is possible to use *compound assignment*.

`a += b;` (`-= , *= , /=`)

is equivalent to

`a = a + b;` (`- , * , /`).

`i=i+1` can be conveniently written using the *increment operator* `++`. `i=i+1` can be written `i++` (postfix) and `++i` (prefix).

Prefix:
```
i = 1; j = 1; k = j + ++i;
i ← 2 , k ← 3 (=1+2)
```
`i` updated before evaluation

Postfix:
```
i = 1; j = 1; k = j + i++;
k ← 2 (=1+1) , i ← 2
```
`i` updated after evaluation

Decrement operator: `--`

Type conversion

Since all values in C have a specific type some mathematical operations can introduce ambiguities. For example, what type should the sum of an `int` and a `double` have?

These questions are resolved by type conversion. Implicit type conversion is done by the compiler when

- type of expressions on left and right hand sides differ (assignment)
- operands in expression of different type

Since a type can not be changed during execution the first case is handled by converting the value of the right hand side to the type on the left hand side before assignment.

Type conversion (cont.)

The second case is handled by "safe conversion". If one type can be expressed in the other (e.g. `int` special case of `double`) that type is converted to obtain a more accurate result.

There are some pathological cases.

Sometimes it is advisable to do an explicit type cast,

```
(type) expression
```

which converts the value of the expression to the given type.

Example:

```
int   i , j;
double f;
i = 3 ; j = 2;
f = i / j;           /* f will hold 1.0 */
f = (double) (i / j);/* f will hold 1.0 */
f = (double) i / j;  /* f will hold 1.5 */
f = i / (double) j;  /* f will hold 1.5 */
```

Marco Kupiainen
marcok@nada.kth.se

Formatted output (`stdio.h`)

Output in C is written to *output streams*. The two most common are `stdout` (for standard output, usually screen) and `stderr` (for error messages).

Output written to `stderr` is shown as soon as the statement is executed.

Output written to `stdout` is buffered.

A programmer can define new streams.

```
int fprintf(FILE *stream,
            const char *format, ...);
 int printf(const char *format, ...);
```

`fprintf` allows the programmer to specify the output stream. `printf` writes to `stdout`.

The appearance of the output is specified by the format string.

The ellipsis ,..., should be replaced by the values to print.

Marco Kupiainen
marcok@nada.kth.se

Formatted output (cont.)

A simple example:

```
printf("Area = %f\n", pi*r*r);
```

`"Area = %f\n"` is a format string instructing the computer to write the string `"Area = "` followed by a floating point number (`pi*r*r` evaluated) and a linebreak to `stdout`.

The format string must contain a conversion specification for each value to print.

Some common conversion specifiers:

Specifier	Displays
`%f`	Floating point numbers
`%e`	– " – , exponential form
`%g`	Combination of two above
`%d`	Integers
`%c`	Characters
`%s`	Strings

Marco Kupiainen
marcok@nada.kth.se

Formatted output (cont.)

The basic specifiers can be modified. The specification `%`p`.`$q$$X$ prints a value of type X with precision q and a minimum field width p (for tables).

The meaning of "precision" depends on X. Typically the number of decimals ($X = f$ `or` $X = e$) or significant digits ($X = g$ `or` $X = d$).

Escape sequences instructs the computer to print special characters. Some useful examples:

Sequence	Displays
`\n`	New line
`\t`	Tab
`\"`	Double quote (")
`\\`	Backslash (\)
`\?`	Question mark (?)

Marco Kupiainen
marcok@nada.kth.se

Input/output – an example (cont.)

Execution of example program:

```
c2m2-20>a.out
Enter radius of circle: 1
Area = 3.140000e+00   ,  Circumference = 6.28000

c2m2-20>a.out
Enter radius of circle: 4.5
Area = 6.358500e+01   ,  Circumference = 28.26000

c2m2-20>a.out
Enter radius of circle: 2.3e1
Area = 1.661060e+03   ,  Circumference = 144.44000
```

Marco Kupiainen
marcok@nada.kth.se

Marco Kupiainen
marcok@nada.kth.se

Formatted input

Input in C is read from *input streams*. The most common is `stdin` (standard input, usually keyboard).

```
int fscanf(FILE *stream,
           const char *format, ...);
int scanf(const char *format, ...);
```

The format string is basically the same as for `fprintf`.

The ellipsis should be replaced by addresses to memory locations where the values are to be stored.

Addresses to variables can be obtained by the address operator `&`.

`&x` returns the address of the variable `x` in memory.

Note: To read a `double` the format specification `%lf` must be used, `%f` will not work.

Input/output – an example

```
#include <stdio.h>

main() {
  const double pi = 3.14;
  double radius;

  printf("Enter radius of circle: ");
  scanf("%lf", &radius);

  printf("Area = %e   ,  Circumference = %.5f\n\n",
         pi*radius*radius, 2*pi*radius);
}
```

Marco Kupiainen
marcok@nada.kth.se