Design and Computing 1

Introduction to Scientific Computing

Daniel Poole
Department of Aerospace Engineering
University of Bristol
d.j.poole@bristol.ac.uk

2017

LECTURE 2

Intro to Scientific Computing

- Previous lecture was an intro to the history of scientific computing
- Outlined the need to understand how to program and examples of its use

TODAY

- Basics of programming to solve engineering/scientific problems
- Introduction to the C language

The "Engineering Method"

- e.g. Describe the flow structure around a hovering rotor?

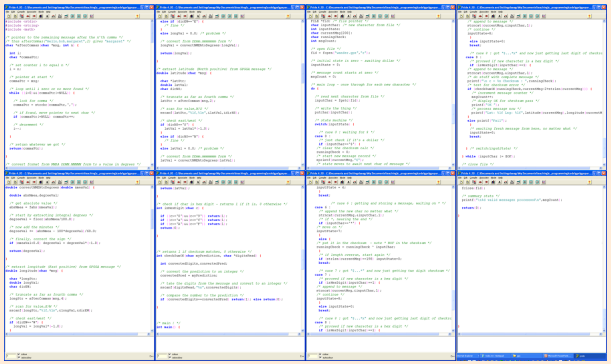
PROBLEM

Flow around rotor?

⇒

SOLVE

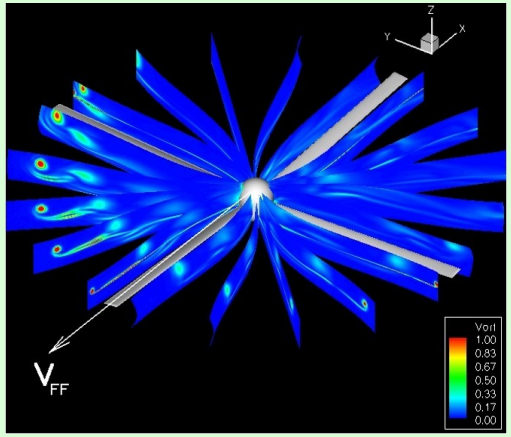
Programme computer to solve complex flow equations



⇒

REPORT

Scientific plotting software



C Program Anatomy

```
#include <stdio.h>

int main() {

   int a;
   float b;

   a=34*7;
   b=34.5/7.2;

   return(0);
}
```

Tells compiler to load some libraries

Start of the program

Memory allocation

Statements

End of the program

C Program Anatomy

`#include <stdio.h>`

Tells to compiler to load some standard commands for use in your program. For reference: `http://www.cplusplus.com/reference/cstdio/`. For mathematical operations, `math.h` is used.

Curly Brackets { }

These define the start and end of functions and commands in C. Their use will become obvious as we go through the course

Semi-colons ;

This defines the end of a line in C and must always be used. Often compile errors will be because of the lack of a trailing semi-colon.

`return(0)`

This is used later in the course when we start to talk about functions. For now, just add it to the end of all of your programs. It tells the computer to not expect any number back from the program.

Data Storage & Variables

```
#include <stdio.h>

int main() {

   int a;
   float b;

   a=34*7;
   b=34.5/7.2;

   return(0);
}
```

Tells compiler to load some libraries

Start of the program

Memory allocation

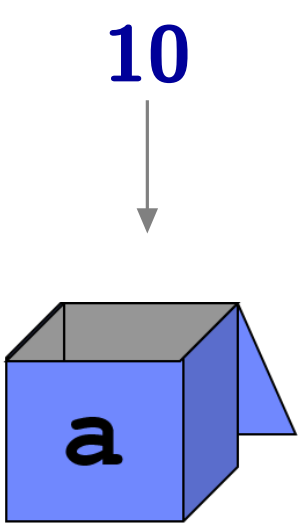
Statements

End of the program

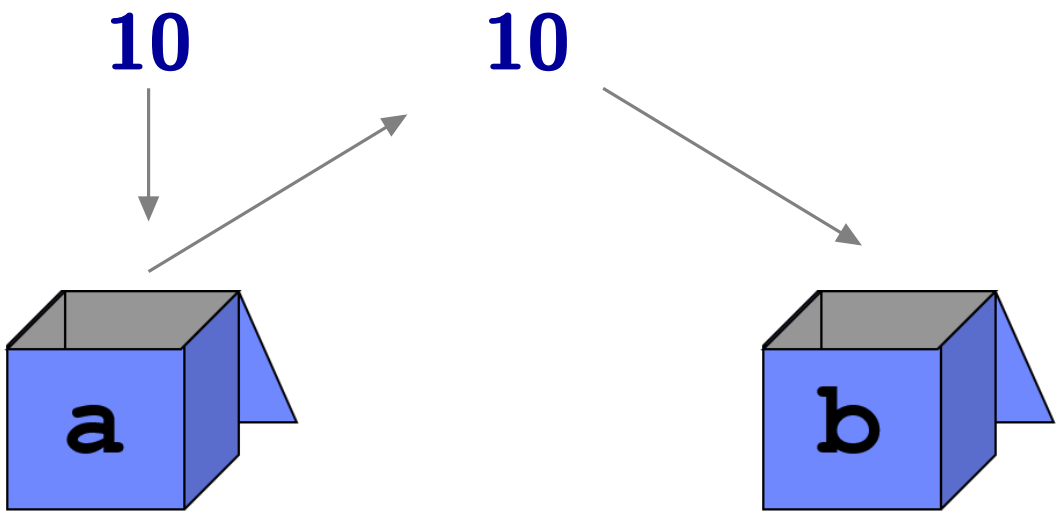
Data Storage & Variables

Remember that computers manipulate numbers in operations defined by us. As programmers, we need to place these numbers into the computer's memory such that we can tell the computer to manipulate these numbers in a way defined by the commands we will learn. Numbers and characters are stored as **VARIABLES**. Think of a box: a variable is a box with a number in it. If we assign a value to a variable, then:

```
a=10;
```




```
a=10;
b=a;
```



Data Storage & Variables

Each variable must also be defined by a type at the start of the program. In C the data types are:

`int`	Stores a 32-bit integer number	$\pm 2.1 \times 10^{9}$
`float`	Stores a 32-bit real number	$\pm 1.2 \times 10^{\pm 38}$ with precision 8
`double`	Stores a 64-bit real number	$\pm 2.3 \times 10^{\pm 308}$ with precision 16
`char`	Stores a single ASCII character	

For real numbers, assuming you have a 64-bit machine (any modern computer probably will be) then `double` should be the default data type for real numbers. `float` can be used if you have strict memory limits or major performance issues (graphical processing, which requires very high performance, where accuracy can be sacrificed, will commonly use floats over doubles).

Statements

```
#include <stdio.h>
int main() {
   int a;
   float b;

   a=34*7;
   b=34.5/7.2;

   return(0);
}
```

Tells compiler to load some libraries

Start of the program

Memory allocation

Statements

End of the program

Statements

Statements are the building blocks of any program. We will consider many statements throughout this course (IF, FOR, WHILE, etc. etc.) but for now, the simplest form of statement is an assignment, as we have seen before:

```
a=37*2;
```

```
b=a;
```

```
a=a*2;
```

Note that in this case '=' means take what is on the right and place assign it to the variable on the left. It does not mean equal to in a mathematical sense, which goes back to the example earlier where programs can't solve a statement of fact as it is, but they are a recipe to solve something.

We can also use other mathematical operations (assuming we have loaded `math.h`):

```
y=32*x/sin(z);
```

Statements

NOTE: Operator precedence will mean the operations are performed in the order of `()`, `/`, `*`, `+`, `-`. Innermost brackets are always evaluated first. If in doubt, just use brackets to be sure.

DEMO: Statements

Statements

Be careful of mixing data types in expressions to give false answers.

```
int main() {
  int a;
  float b;

  a=2;
  b=3/a;
  printf("%f\n",b);
  return(0);
}
```

```
int main() {
  int a;
  float b;

  a=2;
  b=3.0/a;
  printf("%f\n",b);
  return(0);
}
```

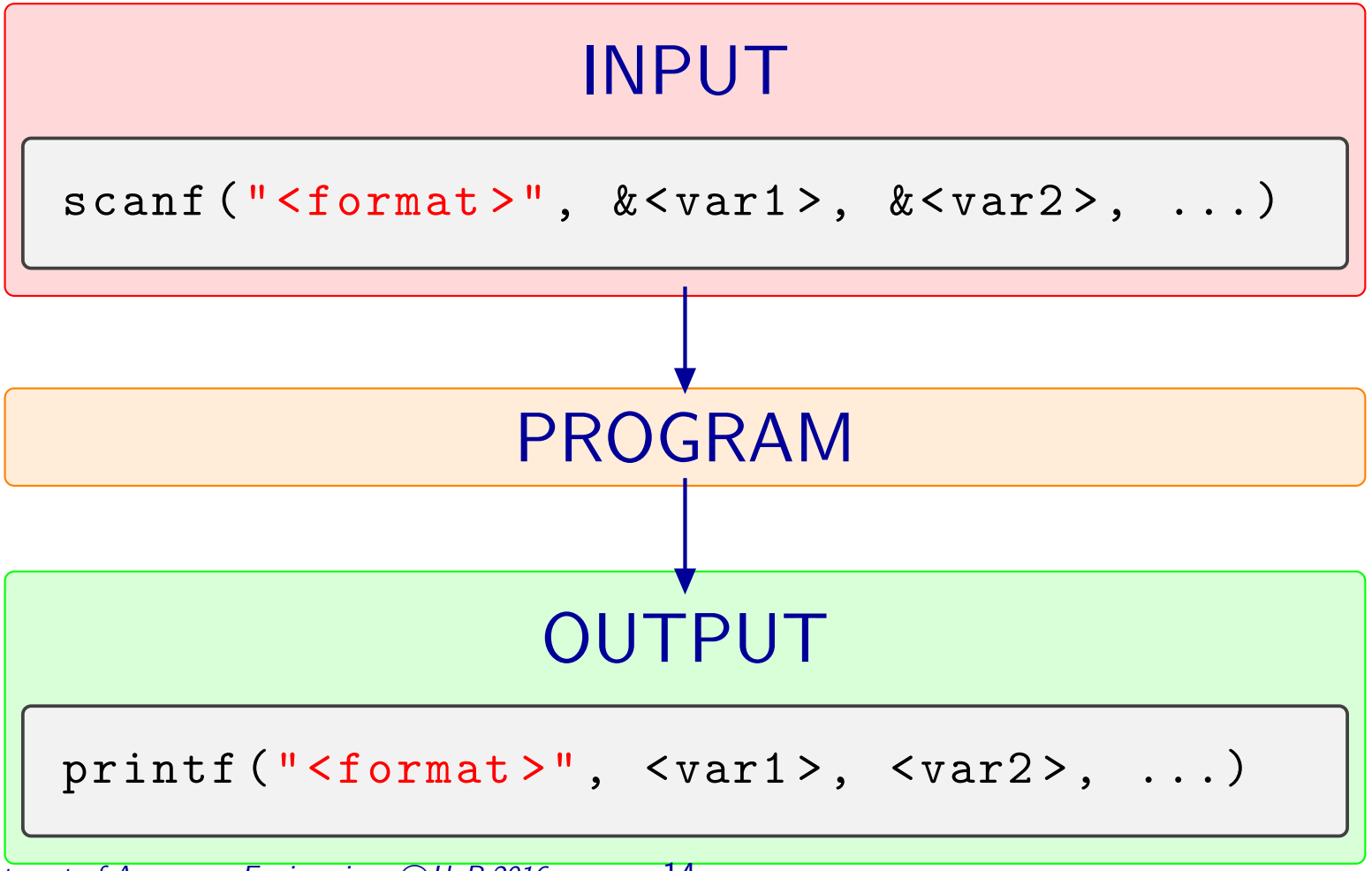
On the left, `b=1.000000` as we are dividing an `int` by an `int`, which returns an `int`, though is stored in a `float`.

On the right, `b=1.500000` as we are dividing a `float` by an `int` so then `int` gets 'upgraded' and we return a `float`.

DEMO: Mixed data types

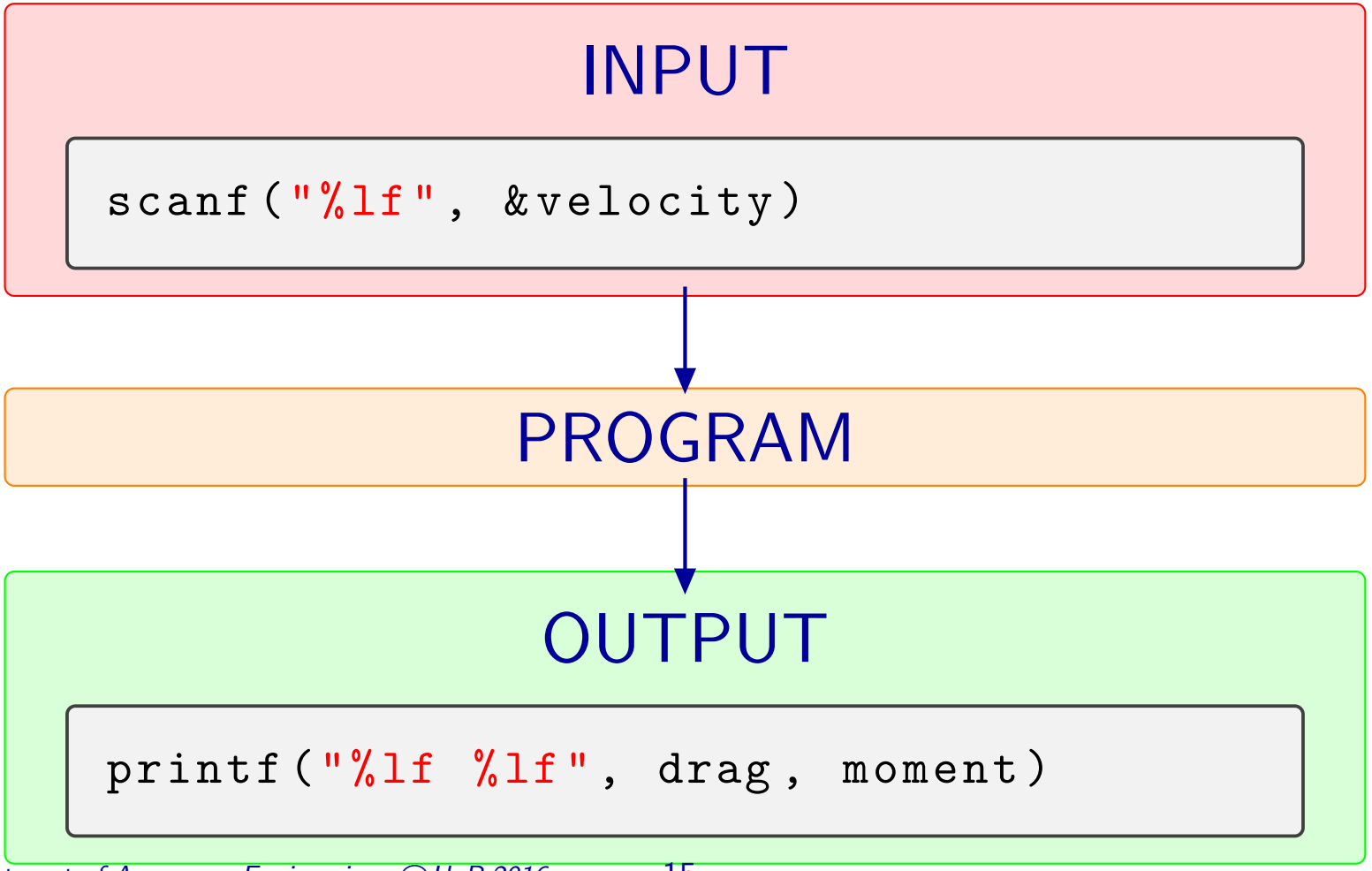
Data I/O

Once operations have been performed, solutions need to be printed to the screen. Also, there may be variables in your code that you want to have as user inputs. Later in the course we will deal with file input and output, but for now, the screen based commands are:



Data I/O

Once operations have been performed, solutions need to printed to the screen. Also, there may be variables in your code that you want to have as user inputs. Later in the course we will deal with file input and output, but for now, the screen based commands are:



Data I/O

- `<format>` is a string containing information about how the variables are printed. Within this string, C expects %`<spec>`, which is replaced by the variable in the order it is written (see below)

- The & in `scanf` is because we are writing to the variables, instead of reading from them.

INFO: The & in `scanf` is because of the use of pointers, which will be covered later in this course. A pointer is a variable that contains the address of a variables, rather than the value of the variable itself. Some programming languages such as FORTRAN avoid pointers altogether. It is difficult to avoid pointers in C (but possible).

Data I/O

%X.Yf for floats (and sometimes doubles) - X is total number of units printed, Y is number printed behind decimal (units greater than asked for are replace with blanks and any trailing digits should be zero if stored correctly).

```
a=1.2;
printf("%3.1f--%4.1f--%4.2f--%5.2f"\
,a,a,a,a);
```

```
1.2--  1.2--1.20--  1.20
```

%X.Ylf for doubles

%X.Ye for print your double or float in exponent form

```
a=0.00013;
printf("%3.1e",a);
```

```
1.3e-04
```

%Xi for integers - X has same rules as above

%c for characters

Writing Code

The principle of writing good code is something that is difficult to explain, but is gained through experience and doing coding. However, there are a few concepts that are particularly important to understand (indents can be with tab or space).

Debugging

Debugging is one of the most important skills you will learn in coding and is an exercise in pure logical deduction. Debugging is the process of making your code work properly and is a skill that you get better at with practice.

One can use complex debugging software that is designed for this sort of thing, but the 'old-fashioned'/simplest methods for debugging are:

- Test your code as you write it
- Use `printf` commands to output variable values to screen

Writing Code

The principle of writing good code is something that is difficult to explain, but is gained through experience and doing coding. However, there are a few concepts that are particularly important to understand (indents can be with tab or space).

Comments

Often you are not the only people who will read your code. For other users of your code it is important to explain what is going on, hence comments become useful. Often you will need comments to remind yourself about what is going on in your code if you come back to it at a later date. Comments are lines of the code that are ignored by the computer. They can therefore contain anything, but must be encased within '/*' and '*/'.

```
/* Allocate memory */
double a,b;

/* Ask for a user input */
printf("Type a number:");
scanf("%f",&a);

/* Multiply by two */
b=2.0*a;

/* Print the output */
printf("%f\n",b);
```

Writing Code

The principle of writing good code is something that is difficult to explain, but is gained through experience and doing coding. However, there are a few concepts that are particularly important to understand (indents can be with tab or space).

Indents

Properly indenting code makes it much easier to follow. This will become much more important in the next lecture when we consider branching and looping

```
/* Loop over n sections */
for (ii=1; ii<=n; ii++) {

    /* Print ii */
    printf("%i\n",ii);

}
```

Writing Code

The principle of writing good code is something that is difficult to explain, but is gained through experience and doing coding. However, there are a few concepts that are particularly important to understand (indents can be with tab or space).

Variable Names

When naming your variables, you can chose almost any combination of basic characters (except any keywords that C would be expecting) so to make your code easier to write, debug, follow and document, you should always give your variables names that describe the data that it is holding. For example, avoid using `a`, `d`, `z` to represent variables, and use names similar to those here, for example. Often counters in loops are given variable names that have `i` or `j` in them such as `icount` or `ii`.

```
double lengthA, lengthB, angle;
lengthA=1.0;
lengthB=2.0;
angle=acos(lengthA/lengthB);
```

NOTE: C is case specific (`A`≠`a`)

Writing Code

The principle of writing good code is something that is difficult to explain, but is gained through experience and doing coding. However, there are a few concepts that are particularly important to understand (indents can be with tab or space).

Continuation Character

You may often not be able to fit all of a single line of code on one line, especially in the variable declaration lines. In this case, the continuation character \ can be used. This tells the compiler to read the next line and treat it as though it belongs to the end of this line.

```
/* This single line ... */
double a, b, c, d;
```

```
/* ... can be over multiple lines */
double a, b, \
c, \
d;
```

Summary

- Have introduced the concept of computer programming
- Have considered basic concepts of programming
- Have introduced the C programming language

LAB

- Writing, compiling and running code
- Using variables and basic statements to do calculations