Chapter Two

Numbers In, Numbers Out

Lecture slides prepared by Alistair Moffat

Chapter 2

Chapter 2 – Concepts

- Identifiers, variables and constants.
- Type hierarchy.
- Arithmetic operators and expression types.
- Precedence.
- Input using `scanf` and format descriptors.
- Output using `printf` and format descriptors.

2.1 Identifiers

An identifier is a name for a value.

- Identifiers must begin with a letter or underscore (“_”) character
- They can then use any combination of letters, digits, and underscore
- They may not contain any other characters.

Use meaningful identifiers, so that names reflect the quantities being manipulated.

2.1 Identifiers

Some words are reserved because they have special significance.

For example, `int` and `while` were used in `addnumbers.c`.

There are about two dozen such special words in C.

2.2 Constants

Constants are fixed values that do not change.

They are introduced using the "#define" facility:

```
#define SEC_PER_MIN 60
#define MIN_PER_HOUR 60
#define EXAM_PERCENTAGE 60
#define PROJECT_PERCENTAGE (100 - EXAM_PERCENTAGE)
#define KM_PER_MILE 1.609
```

Use symbolic constants for all fixed values, to improve readability and modifiability. Use different names for different concepts, even if they have the same value.

2.2 Variables

Variables can change value during program execution.

Assignment statements store into variables the values generated by a corresponding expression.

```
nitems = nitems+1;
miles = km/KM_PER_MILE;
tot_seconds = (hours*MIN_PER_HOUR + mins)*SEC_PER_MIN
              + secs;
final_mark = (exam_mark*EXAM_PERCENTAGE
              + project_mark*PROJECT_PERCENTAGE)/100;
```

2.2 Exercise 1

Using suitable identifiers, write assignment statements to compute (a) the total surface area and (b) the total edge length, of a rectangular prism of edge lengths a, b, and c.

2.2 Variables

Variables must be declared before they are used.

Each variable in a program has a type associated with it.

Variables must then be assigned values before they can be used in expressions.

Use of uninitialized variables is a common error.

2.2 Integers

Variables of type `int` store integer-valued numbers in a constrained range, often $-2^{31} = -2{,}147{,}483{,}648$ to $+2^{31} - 1 = +2{,}147{,}483{,}647$.

These bounds are a consequence of 32-bit words.

- `overflow.c`

Beware. Integer overflow in C programs results in incorrect values propagating through a computation without warning messages being produced.

2.2 Doubles

The type `double` can store fractional parts, and numbers over a wider range, but may not store them exactly.

- `rounding.c`
- `addorder.c`

Care must be taken that rounding errors do not affect the usefulness of computed values.

2.2 Int versus double

Constants also have types. If there is a "`.`" or an "`e`" (exponent part), it is a `double`. If not, it is an `int`.

Label each of these constants with its type. Then put them in to order, from smallest to largest.

`-1.5e6`, `3.14e1`, `-66.7e-1`, `6.67e1`

`-1.5`, `30e0`, `-30`, `667`, `1e4`, `100000`

2.2 Constants and variables

Each subexpression of an expression has a type dictated by the types of the operands. Use constants that match the variables they are being combined with.

```
double energy, mass, velocity;
/* BEWARE -- incorrect code */
energy = (1/2)*mass*velocity*velocity;
```

2.3 Operators and expressions

The precedence rules specify operator order.

In the absence of parentheses, (all) the multiplicative operators are evaluated first.

The expression `2+3*4` is `14`, which might (or might not be) what your calculator gives.

The remainder (or modulus) operator “`%`”, which must have two `int` arguments, is also a multiplicative operator.

2.3 Exercise 2

What are the values of:

```
6*7-8*9/10

2*3*4+5*6%7

5*6/4%3*5 ?
```

2.3 Operators and expressions

When both operands are of type `int`, the result is an `int`.

If either operand is of type `double`, the result is a `double`.

```
int n_pass, class_size;
double pass_percent;
/* BEWARE -- incorrect code */
pass_percent = n_pass/class_size*100;
```

2.3 Operators and expressions

Can use the type rules to ensure a correct calculation:

```
pass_percent = 100.0*n_pass/class_size;
```

Or can use an explicit cast:

```
pass_percent = (double)n_pass/class_size*100;
```

2.3 Operators and expressions

When doubles are assigned to int variables the value is truncated.

To get rounding, add 0.5 just before the assignment occurs:

```
int n_pass, class_size, pass_percent;
pass_percent = (int)(0.5 + 100.0*n_pass/class_size);
```

2.3 Exercise 3

What are the types and values of:

```
(1+8/3*4.0/5+6)/2

(2.0*5*6/3)*(2/3)

(int)(5*(double)6/8) ?
```

2.3 Program layout

Program layout is free-form. All whitespace is removed during the compilation process, except within strings.

Semi-colons are used to terminate statements.

Long statements may be broken over multiple lines.

Thoughtful choice of identifiers makes programs readable.

Comments should explain each block of code.

2.3 Divide by zero

Integer division by zero usually results in program termination, with an error message like “Floating exception”.

Floating point division by zero may result in the value `inf` being generated, or an arbitrary value.

C systems might also make use of the double value `nan` (not a number), for example, when taking the square root of a negative number.

Be sure you know what your C system does.

2.4 Numbers in

To read numbers into a program, use `scanf`:

`scanf(` *control string* `,` *list of variable addresses* `)`

The control string specifies how the values are to be read:

```
int n_pass, class_size, pass_percent;
double mass, velocity, force;
scanf("%d%d", &n_pass, &class_size);
scanf("%lf%lf%lf", &mass, &velocity, &force);
```

2.4 Numbers in

To read an `int`, use “`%d`”.

To read a `double`, use “`%lf`”.

To read a `float` (reduced precision floating point), use “`%f`”.

To read an `char`, use “`%c`”.

Note the use of operator “`&`” prior to each variable name.

2.4 Numbers in

For the numeric types, leading white space characters are skipped.

If an input character cannot form part of the current variable, it is assumed to mark the beginning of the next variable.

If that input character cannot be part of the next variable, reading halts and no further variables are assigned values.

2.4 Numbers in

The number of variables successfully assigned is returned by the `scanf` and can be captured and used:

```
num = scanf("%d %f %c", &n, &z, &c);
printf("There were %d values successfully read\n",
    num);
```

It is easy to make mistakes with `scanf`. Always echo back values after reading.

2.5 Numbers out

General structure:

`printf(` *control string* `,` *list of expressions* `)`

Both `double` and `float` values are printed using "`%f`" – the "`%lf`" format descriptor is used only for input.

The descriptor "`%s`" is used to print strings.

2.5 Numbers out

Field widths control the layout.

Positive values give right-aligned values, negative give left-aligned output.

- `format.c`

2.6 Assignment statements

The general form of the assignment statement is:

variable = *expression*

Shorthand forms: `sum=sum+next` can be shortened to `sum+=next`, and `n=n+1` to `n+=1`. Similarly, `n/=2` divides `n` by `2`.

The postincrement and postdecrement operators go one step further: `n++` and `n--` add and subtract one from `n`.

2.6 Assignment statements

Variables of type `char` are a restricted form of `int`, and character values like `'a'` are `int` constants.

The ASCII mapping dictates the correspondence between integers and characters.

The lowercase letters form a contiguous block starting at 97. The uppercase letters are a block starting at 65; the digits (from '0') at 48.

2.6 Assignment statements

If an expression has a different type to the receiving variable, it is cast on assignment.

- `mixedvals.c`

2.7 Case study

Write a program that reads in the radius of a sphere, in meters, and calculates and outputs the volume of that sphere, in cubic meters. The volume of a sphere of radius r is given by $\frac{4}{3}\pi r^3$.

- `spherevol.c`

2.7 Case study

Some general advice when writing programs:

- Echo the input data
- Keep it simple
- Keep it neat
- Build it incrementally
- Test it at every stage.

Programs should always be written presuming that the user will either deliberately or accidentally abuse them.

2.7 Exercise 4

Write a program that reads in a weight in pounds, and outputs the corresponding weight in kilograms. One kilogram is 2.2046 pounds.

What is the metric equivalent of a 90-pound weakling?

Chapter 2 – Summary

- Use `#define` for *all* constants.
- declare and initialize all variables.
- `int` arithmetic will overflow silently
- `int` *op* `int` yields `int`, even if *op* is division.
- With `float` and `double`, operation order and rounding can affect the result
- Each type has corresponding input and output format descriptors. For `double`, `%lf` on input, and `%f` on output.