Consider the following equation:

$$F - 32 = 9\,C / 5$$

Meaning:

If we subtract 32 from the temperature in degrees Fahrenheit, we get the same value as when we multiply 9 by the same temperature expressed in degrees Celsius and divide by 5.

To convert equation into a formula useful for converting a temperature in °F to °C, make C the only symbol on left hand side.

$$C = 5(F - 32) / 9$$

So

Given a Fahrenheit temperature F, subtract 32, then multiply by 5 and divide by 9. The result

is the equivalent degrees Celsius.

Here's the C++ program.

```
#include <iostream>
using namespace std;

int main()
{
     float F, C;

     cout << "Enter Fahrenheit Temp to convert: ";
     cin >> F;
     C = 5.*(F-32.)/9.;
     cout << F << " in Fahrenheit is " << C
          << " in Celsius\n";
     return 0;
}
```

You may notice the difference between many of the programs shown in the notes and here.

For example

```
cout << F << " in Fahrenheit is " << C  << " in Celsius\n";
```

becomes

```
cout << F << " in Fahrenheit is " << C
     << " in Celsius\n";
```

A statement can be broken over multiple lines at any point other than within a quote string.

$$C = 5(F - 32) / 9$$

became

```
C = 5.*(F-32.)/9.;
```

Note the changes, especially the use of the symbol `*` to mean multiplication.

The remaining changes are required C++ syntax.

Here is another way of writing the same C++ program:

```
#include <iostream>
using namespace std;int main(){float F,
C;cout<<"Enter Fahrenheit Temp to convert: ”
;cin>>F;C=5.*(F-32.)/9.;cout <<F<<" in \
Fahrenheit is "<<C<" in Celsius\n";return 0;}
```

Which do you prefer?
Only changes are removal of 'unnecessary' white space – blanks, newlines and tabs.

C++ compilers ignore unnecessary white space – but we can't.

Spaces, newlines and tabs make programs readable – by humans.

We must be able to understand what the program does so that we are in control of the programming process.

Especially important is the location of syntax errors – some can be avoided while entering the program into the computer.

Programming Style

- not part of language syntax
- can be personal
- develop your own style
- learners should follow someone else's
- once syntax is mastered you can develop your own style

We will propose some tenets of good style – follow them.

Style Rule 1:

short programs with little white space are unreadable

Each line of code should be readable.

- instant understanding of purpose
- if not obvious, add a comment in English, explaining the line
- don't overdo it

```
#include <iostream>   // defines all I/O
using namespace std;  // selects namespace
int main()     // the start of program proper
{
     float F, C;   // variables to hold the temps
/* following line prints a prompt */
     cout << "Enter Fahrenheit Temp to convert: ";
     cin >> F;   // here enter the Temp to convert
     C = 5.*(F-32.)/9.    // the calculation
/*  now the output */
     cout << F << " in Fahrenheit is " << C
          << " in Celsius\n";
     return 0;    // we're finished
}
```

OVERKILL

Comments should only be used if the meaning of a line is not obvious.

A person who doesn't know C++ is not going to be helped by an unnecessary comment.

Can't see the programming for the comments.

Comments

Form 1:

any sequence of characters between the pairs

```
/*      and      */
```

are ignored by the compiler.

- can be written over many lines
- especially useful to describe large segment of code (not just one line)

Example:
it is common practice to describe the purpose of whole program and give the name of the author in a comment at head of program.

```
/*
     This program requests a temperature
     in degrees Fahrenheit calculates
     the equivalent temperature in
     degrees Celsius and prints out the
     two temps.
     Written by Peter Castle.
*/
```

Form 2:

any sequence of characters between the pair

```
    //
```

and the end of a line are ignored.

e.g.

```
C = 5.*(F-32.)/9.;    // the calculation
```

Use of white space

- some spaces are required by the syntax
- others just make the code readable

```
float F,C;
```

- space after `t` is required

```
float F, C;
```

- space after `,` is style

Sequences of spaces can be entered using the tab key.

There are pre-defined tab stops every 8 characters – entering a tab means text jumps to next tab stop.

We can alter the 8 in editors, but Linux (and Unix) uses 8 most everywhere.

```
#include <iostream>
using namespace std;

int main()
{
        float F, C;

        cout << "Enter Fahrenheit Temp to convert: ";
        cin >> F;
        C = 5.*(F-32.)/9.;
        cout << F << " in Fahrenheit is " << C
             << " in Celsius\n";
        return 0;
}
```

Indentation

Alignment

white space (tabs)

VERY IMPORTANT

The concepts of style described here are NOT for the purposes of producing a final neat program.

They are an aid to writing CORRECT code and for DEBUGGING your programs.

Use style while typing in your program.

In particular, use indenting when entering your program.

Do not use spaces for indenting.
Use a tab.

Most editors will automatically continue an indent on successive lines. Just backspace to 'de-indent'.

Lines can be indented in groups using the Shift Left and Shift Right commands.

Some more style

Names of entities mean little to the compiler – just a label for a memory address.

BUT they have a major part to play in documenting your code, so that others can understand your program.

- names must be meaningful
  and give insight into their purpose

- balance readability with minimising
  typing – if in doubt use longer name
  3 to 10 characters should be enough

e.g. our example used `C` and `F`.
Perhaps `Celsius` and `Fahrenheit` ?

```
      Celsius = 5.*(Fahrenheit-32.)/9.;
```

or maybe `degC` and `degF`?

- a symbol should be used for one purpose only – any time it appears in a program it should represent the same entity.

- use comments freely BUT the use of identifiers can reduce the need

  ```
  answer = number + INCREMENT;
  ```

  does not need a comment like

  ```
  // add increment to number giving answer
  ```

- use blank lines, indenting & aligning

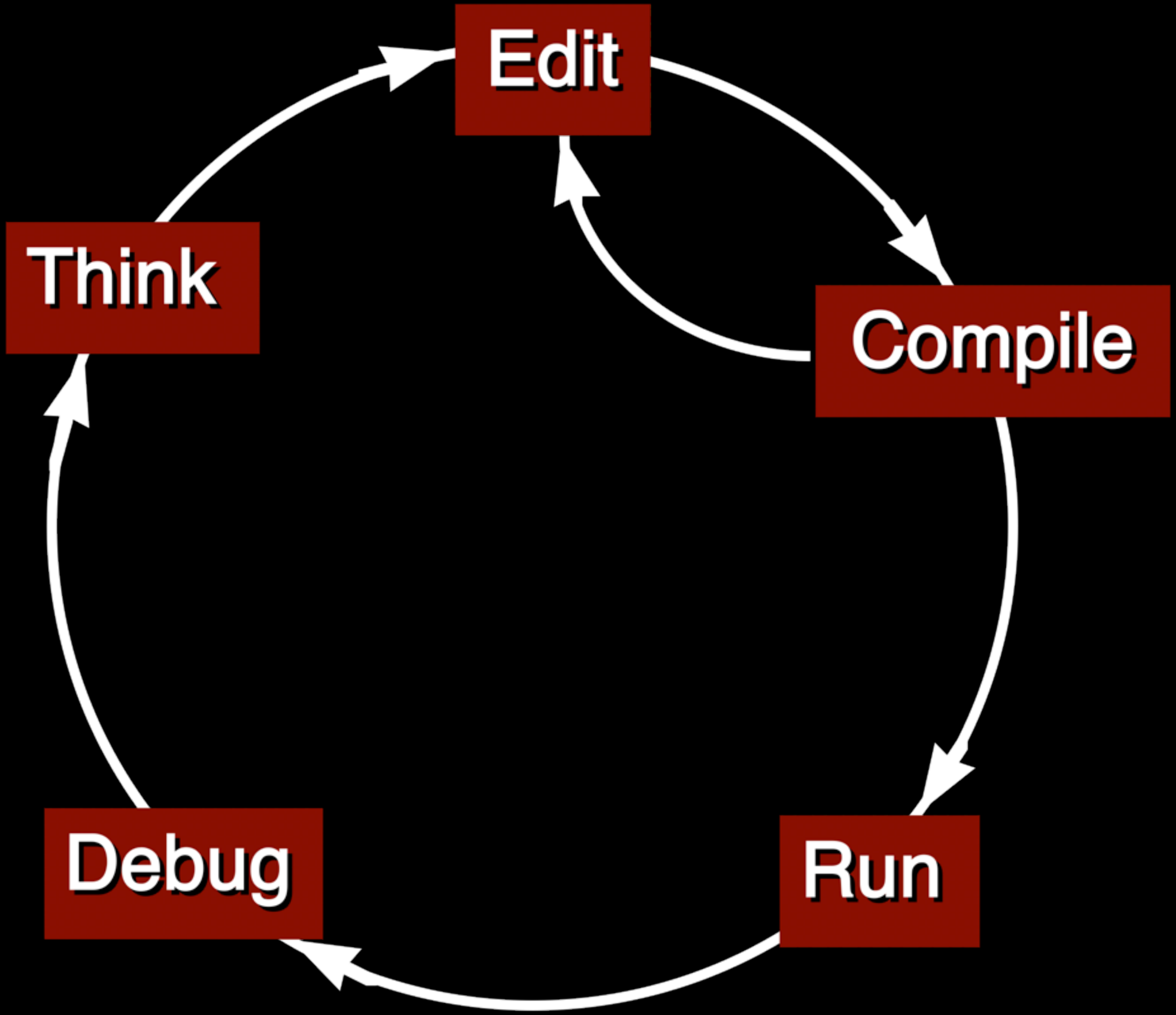
Finding syntax errors

- programs usually don't work first time
- there will be mistakes
- some will be blunders – typos.
- some will be incorrect characters by choice

- the compiler doesn't care how the incorrect characters appeared in the code – it will report a syntax error

Some common syntax errors

- mistyped identifiers
  - misspelling
  - different case
- incorrect punctuation
  - ; instead of ,
  - leaving out a ;
- forgetting vital spaces


Edit
Compile
Run
Debug
Think

Consider the following problem

What is the (numerically) largest and smallest telephone number in the alphabetic subscriber section of the New York Telephone White Pages?

How do we start?
How can we be sure we don't miss any numbers?
What is the set of rules to follow?
What is the algorithm?

- open the telephone book at the first page of the listing

- take two sheets of paper and write LARGEST on one and SMALLEST on the other

- look at the first number in the list.
  It is the largest seen so far. Write it on the paper marked LARGEST.
  It's also the smallest so far. Write it on that piece as well.

LOOP

- read the next number
- compare it to the number on LARGEST. If the number we've just read is larger than the largest seen so far, it must be the largest so far, so cross out the number on the LARGEST page and put the new number on it.
- compare it to the number on SMALLEST. If it is smaller, replace the number on the page.

Continue LOOP until end of listing.

At the end of the process, the two pages will show the LARGEST and SMALLEST in the whole book.

This is an algorithm – a description of the steps to be carried out to solve a problem.

Let's look at some of the properties of the description.

What is always written on the piece of paper headed LARGEST?

The largest of all the numbers read so far.

At the outset, the blank sheet also shows the largest so far.

At any point in the process, it holds the largest so far.

What is always written on the piece of paper headed SMALLEST?

The smallest of all the numbers read so far.

At the outset, the blank sheet also shows the smallest so far.

At any point in the process, it holds the smallest so far.

Do we have to do the second test (is the number the smallest so far?) every time?

No. If the number is the largest so far, it cannot be also the smallest.

The first number is treated as a special case, where the same number appears on both pages.

So we can modify our algorithm.

LOOP
- read the next number
- compare it to the number on LARGEST. If the number we've just read is larger than the largest seen so far, cross out the number on the LARGEST page and put the new number on it.

OTHERWISE
- compare it to the number on SMALLEST. If it is smaller, replace the number on the page.

Continue LOOP until end of listing.

How do we stop?

When there are no more.

When we've reached the end of the section – marked by the page headed Using Your Telephone Service – there are no more numbers.

A computer program would also need to have some way of stopping.

Alternatives

- count how many numbers there are and loop the exact number of times

- pick a number that is not a valid phone number and put it at the end of the valid ones – a sentinel – zero or negative.

(Is the sentinel part of the data to be processed?
NO. Otherwise it would be the smallest.)

Here's a C++ program to perform a similar task.

```
#include <iostream>
using namespace std;

/*
     Program designed to read a set of
     integers and to find the largest and
     smallest numbers in the set.
     The program is to stop when it reads a
     non-positive number. There will always
     be at least one positive number in the
     data.
     Written by Peter Castle.
*/

int main()
{
      int number, maximum, minimum;

      cout << "Type in integers (<=0 to end)\n";
      cin >> number;
      maximum = number;
      minimum = number;
      while(number > 0)
      {
            if (number > maximum)
                  maximum = number;
            else if (number < minimum)
                  minimum = number;
            cin >> number;
      }
      cout << "Largest number found was "
            << maximum << endl;
      cout << "Smallest number found was ”
            << minimum << endl;
      return 0;
}
```

Basic structure:

```
READ the first number
WHILE the number is valid
    PROCESS the number
    READ the next number
WRITE the results
```

We set maximum and minimum to the first value read in.

Why don't we just set minimum to some huge number that must be bigger than the smallest possible value? And maximum to, say, -1.

Because we have to know beforehand that such values are invalid.

AND

Because

`minimum`

represents the smallest number found so far.

It cannot represent an invalid value, one that isn't even in the data set.

Now that we've seen one program let's start learning some C++.