Lec_9.md

Lecture 9 - C++ Input/Output (IO)

Sept.30/2020

Streams

What exactly is a stream

- A sequence of characters
- Allows connection between different hardware components
    - Keyboard
        - Input as a **stream**
    - Display
        - Output as a **stream**

However, there is disconnect between *input* and *output* streams

- Functionality of streams
    - Perform the necessary conversion between internal data representations and character **streams**
    - Act as agents to convert data for the hardware to interpret
    - Can send streams to different places
- Input Stream: `cin`
- Output Stream: `cout`
- Another output stream: `cerr`
    - Used for error messages

Input Stream

`cin` attempts to extract input values from input stream

main.cpp

```
int main(){
  int x;
  cin >> x;
}
```

Notes:

1. Input stream value placed inside variable `x`
2. A keyboard buffer is created before `cin` receives input stream

- Values are stored temporarily, s.t. they can be changed before the *enter* key is pressed and the stream is sent to `cin`
    - Imagine deleting a character before pressing enter

Extraction from Input Stream

For example, imagine typing the integers

> '1' '2' '3' ' ' '2' '0' '4'

into the executable for `main.cpp`

main.cpp

```
int main(){
  int x,y;
  cin >> x >> y;
}
```

Procedure for `cin` extracting from input stream:

1. `cin` reads '1'
2. `cin` moves onto next character
3. `cin` reads '2'
4. `cin` moves onto next character
5. `cin` reads '3'
6. `cin` moves onto next character
7. `cin` tries to read ' ', but since this is not an integer, it does not read this value into the integer
    - x becomes the *combined value* of the read values from the existing stream
        - in this case, `x=123`
8. `cin` moves onto next character
9. And the same process continues for `y=204`

The values of x,y are

```
x=123
y=204
```

Note: Streams are ***not flushed*** after the line of `cin` completes.

main.cpp

```
int main(){
  int x,y;
  cin >> x
  ...
  ...
  cin >> y;
}
```

And the same input is typed into the executable

> '1' '2' '3' ' ' '2' '0' '4'

The same behaviour holds as for the above case, so

```
x=123
y=204
```

Delimiters

Delimiters are values at which `cin` decides where a sub stream starts and ends

- **Delimiters** define where the values for `cin` assignment start/stop
    - For example, that 123 is an integer, but 12 3 is actually *two* integers

The value ' ','\n' and '\0' are called **Delimiters**.

- ' ' (the space character) is a very common delimiter.

Basically,

If there has not been a stream given to `cin`

`cin` will look for stream (by prompting command prompt input)

If `cin` reaches a **delimiter** before the stream is **exhausted** (e.g. before all of the characters in the stream are read)

`cin` will assign the value it read into the variable

However if `cin` does *not* exhaust the stream (e.g. there is a **delimiter** in the middle of the stream), it will *keep* the stream (as an internal variable) for future `cin` calls;

Int and Floats in Stream Extraction

For the below `main.cpp` :

main.cpp

```
int main(){
  int x;
  float y;
  cin >> x >> y;
}
```

If the input stream is:

> '1' '2' '3' ' ' '2' '.' '4'

Notes:

1. `cin` will read `x=123`
2. `cin` will read `y=2.4`

However, if the input stream is:

> '1' '2' '3' '2' '.' '4'

Notes:

1. `cin` will read `x=1232`
2. `cin` will read `y=0.4`

- In the first case, the ' ' character is the **delimiter**
- In the second case, the '.' character is the **delimiter**
    - This is because '.' is a valid **delimiter** for `int` values
    - Whereas '.' is *not* a **delimiter** for float values
        - The decimal point is a *part* of float values

For the below `main.cpp` :

main.cpp

```
int main(){
  int x,y;
  cin >> x >> y;
}
```

If the input stream is:

> '1' '2' '3' '.' '4'

Notes:

1. `cin` will read `x=123`
2. `cin` will ***not*** read `y`
    - In this case, `cin` tries to read from the '.' character but **fails**

Strings in Stream Extraction

main.cpp

```
int main(){
  string firstName;
  string lastName;

  cin >> firstName >> lastName;
}
```

If the input stream is:

> 'T' 'o' 'm' ' ' 'L' 'i'

Notes:

1. `cin` will read `firstName="Tom"`
2. `cin` will read `lastName="Li"`

What about errors in Input Stream?

`cin` fails silently

- Execution continues, leaving the variable and input stream unaffected;

main.cpp

```
int main(){
  int x = 9;
  cin >> x;
}
```

If the input stream is:

> 'T' 'e' 'n' ' ' '2'

`cin` will ***not*** affect the value of x, and fails silently without any indication

- On ECF computers, this is the behaviour of cin
- On *certain* compilers, `cin` will set the value of x to null

`cin` will not read space characters

- Since space characters (' ') are **delimiters**
- So how do we read strings with spaces?
  - Two ways:
    - Overwrite the `cin` function to treat ' ' not as **delimiters** but as **characters**
      - This is generally a bad idea. Think about it.
    - Use the `getline` function instead of `cin`

Flags (cin)

`cin` has certain **flags**

- **Flags** are set to true/false after every `cin` extraction operation
- These **flags** reflect what happened in the previous `cin` operation
- One specific flag is the `fail` flag
  - The `fail` flag is set to true if `cin` fails to extract
  - Otherwise, set to false
- Developer must check the flags
  - C++ `cin` fails silently (will *not* inform you)

The Fail flag

For the below code:

main.cpp

```
#include <iostream>
using namespace std;
int main(){
  int anInteger;
  cin >> anInteger;
  if(cin.fail()){
    cout << "bad input" << endl;
  }else{
    cout << "read from cin~!" << endl;
  }
  return 0;
}
```

Notes:

1. `cin.fail()` returns false if the line `cin >> anInteger` read successfully
   - `cin.fail()` returns true, the last call to `cin` failed
2. Notice that the `cin.fail()` uses the **field access operator**
   - So what is `cin` ??

Answer: `cin` is an object (instance) of **istream**

Remember that include statement we always use?

- `#include <iostream>`
  - iostream is split into two additional include files:
    - istream
    - ostream
- **istream** defines the `cin` object/instance

- **ostream** defines the `cout` object/instance

The Insertion Operator

So... what exactly is the **insertion operator** (this thing `>>`)

- It's an **operator**
- Acts as a function call
  - Calls with parameters (`cin` , `var`)
    - Assign the read characters from the stream into `var`