Lec_12.md

Lecture 12 - C++ Input/Output (IO)

Oct.7/2020

Output Streams

The **output stream** `ostream` acts as an agent that transfers data between your screen and the program

- Similar to the **output file stream** `ofstream`, but ***different***
- `cout` writes to **output stream**

Output Stream Manipulators

`cout` has **output manipulator** functions that allow for formatting of output

main.cpp

```
#include <iostream>
#include <iomanip>

using namespace std;
int main(){
  int myVar = 503;
  cout << myVar << endl;
  cout << setw(8) << myVar << endl;
  cout << myVar << endl;
  cout << setfill('0') << setw(8) << myVar << endl;
  cout << setw(8) << myVar << endl;
  return 0;
}
```

Notes:

1. `#include <iomanip>`
   - header file containing `cout` **output manipulator** functions
2. `setw(8)`
   - By default, the output stream is **left aligned/justified**
     - This means each line starts from the *left* side of the console/terminal
   - `setw(x)` adds x amount of blank/fill spaces
     - Can use to format `cout` output
   - `setw(8)` adds 8 fill spaces
3. `setfill('0')`
   - `setfill('0')` replaces ' ' characters generated by `setw` with '0'
   - `setfill` is **persistent**
     - Calling `setfill` once will replace all blank space (fill characters) with given parameter
       - Will replace fill chars for all future `setw` function calls

main.cpp

```
#include <iostream>
#include <iomanip>

using namespace std;
int main(){
  float myVar = 3.2876891;
  cout << myVar << endl;
  cout << setprecision(2) << myVar << endl;
  return 0;
}
```

Notes:

1. `setprecision(2)`
   - `setprecision(x)` sets the **precision** of floating point numbers to x
   - `setprecision(2)` , will set **precision** to 2
     - `cout` will print "3.3"
   - Note that `setprecision` ***will round*** the output number, ***not truncate***

I/O Redirection

I/O streams can be redirected to files so you can I/O from files instead of standard input (keyboard) and standard output (screen)

- This works at the OS level (outside the program executable)
- Useful for debugging your code

Can be done at the command prompt:

> % myprog.exe > outfile

all output (`cout`) goes to outfile

> % myprog.exe < infile

all input (`cin`) taken from infile

> % myprog.exe < infile > output

all input (`cin`) taken from infile and all output (`cout`) goes to outfile

> % myprog.exe >& outfile

all output (`cout` and `cerr`) goes to outfile

Pointers, Scopes, and Arrays

Pointers! and Dynamic Allocation!

Pointers

How are variables stored?

- Stored in **memory**
  - But what does memory look like???

address	memory contents	symbol

address	memory contents	symbol
0x0000AB00	5	x
0x0000AB04	3.8	y
0x0000AB08		
0x0000AB0c		
0x0000AB10		
0x0000AB14		
0x0000AB18		

Something like this ^. Note:

- Each address (**hexadecimal number**) differs by 4
  - Each address is of size 4 bytes
    - Or, 32 bits, since
      - 4 bytes * 8bits/byte = 32 bits
- Addresses in most computers are **byte addressable**
  - Meaning that the smallest unit of space that can be **addressed** is a **byte**

When you write `int x;`, a space in memory is reserved

- How much space?
  - Enough to hold an `int` variable

main.cpp

```
int main(){
  int x;
  float y;
  x = 5;
  y = 3.8;
  cout << x << endl;
  cout << y << endl;
}
```

address	memory contents	symbol
0x0000AB00	5	x
0x0000AB04	3.8	y
0x0000AB08		
0x0000AB0c		
0x0000AB10		
0x0000AB14		
0x0000AB18		

Pointers are used to **address** other variables of the same type

- **Pointers** are variables that contain the address of other variables

Dereference Operator

main.cpp

```
int main(){
  int x;
  float y;
  x = 5;
  y = 3.8;
  cout << x << endl;
  cout << y << endl;
  ...
  int* px;
  float* py;
  px = &x;
  py = &y;
}
```

address	memory contents	symbol
0x0000AB00	5	x
0x0000AB04	3.8	y
0x0000AB08		
0x0000AB0c		
0x0000AB10		
0x0000AB14	0x0000AB00	px
0x0000AB18	0x0000AB04	py

Notes:

1. `int* px;`
   - Define **pointer** variable of type `int`
2. `float* py;`
   - Define **pointer** variable of type `float`
3. `&` operator
   - The **ampersand** `&` is the **reference** operator
     - Get the address of the thing the operator is operating on
   - **reference** operator is an **overloaded** operator
     - This means that the functionality of the `&` operator depends on the *context* on which it is called
4. `px = &x;`
   - Set value of `px` *to* **address** of x
   - px, x of type `int*`, `int` respectively
5. `py = &y;`
   - Set value of `py` *to* **address** of y
   - py, y of type `float*`, `float` respectively

Reference Operator

main.cpp

```
int main(){
  int x;
  float y;
  x = 5;
  y = 3.8;
```

```
    cout << x << endl;
    cout << y << endl;
    ...
    int* px;
    float* py;
    px = &x;
    py = &y;
    *px = 3;
    *py = 5.2;
}
```

address	memory contents	symbol
0x0000AB00	3	x
0x0000AB04	5.2	y
0x0000AB08		
0x0000AB0c		
0x0000AB10		
0x0000AB14	0x0000AB00	px
0x0000AB18	0x0000AB04	py

Notes:

1. `*` operator
   - The **asterisk** `*` is the **dereference** operator
     - Access the value at the address of the thing the operator is operating on
2. `*px = 3;`
   - Essentially saying:
     - Change the value at the address of `px` to `3`
   - Sets the value of x to 3
3. `*py = 5.2;`
   - Essentially saying:
     - Change the value at the address of `py` to `5.2`
   - Sets the value of y to 5.2

Pointer Schematics

Just like schematics in circuits (ECE212), **pointer schematics** describe the *mapping* between pointers and variables