CS 246 - Object Oriented Programming **Fall 2016**

Lecture 3: September 22, 2016

Lecturer: Brad Lushman *Notes By: Harsh Mistry*

Example 3.1 *Read all ints and echo to stdout until EOF. Also Skip all non-integer input.*

```
int main(){
    while(true){
        if(!(cin >> i)){
            if(cin.eof())break;
            cin.clear(); // clears the fail bit
            cin.ignore(); // ignore and throwaway current input character
        }
    }
    else cout << i << endl; // read was ok
}
```

3.1 Reading Strings

In C there is a type std::string which is included (`#include <string>`)

Example 3.2 *Basic Read Example*

```
int main(){
    string s;
    cin >> s;
    cout << s << endl;
}
```

- `cin` skips leading white space
- In addition, cin stops reading at white space (reads one word)
- `getline(cin,s)` can be used to read from new line to next new line into s

Example 3.3 *Printing a value out as hex decimal*

```
cout << hex << 95 << endl; //Prints 5f
```

- `hex` is a I/O manipulator, so all subsequent ints will be printed in hex
- `cout << dec` can be used to go back to basic decimal
- There are multiple I/O manipulators, refer to official site for full list. `#include<iomanip>` may be required.

3.2 Stream Abstraction

The concepts covered can be applied to other sources of data

3.2.1 Files

- `std::ifstream` - read file from a file
- `std::ofstream` - write to an file

Example 3.4 *File Access in C*

```c
int main(){
    char s[256];
    FILE *file=fopen("myfile.txt"; "r");

    while(true){
        fscanf(file, "%255", s);
        if (feof(file)) break;
        printf("%s\n",s);
    }
    fclose(file);
}
```

Example 3.5 *File Access in C++*

```cpp
1 #include <iostream>
2 #include <fstream>
3 #include <string>
4 using namespace std;
5
6 int main(){
7     ifstream file {"myfile.txt"};
8     string s;
9     while(file >> s){
10        cout << s << endl;
11    }
12 }
```

- *Declaring and Initializing the variable on line 7, opens the file*
- *File is closed when the ifstream variable goes out of scope*
- ***Anything** you can do with* `cin/cout`*, you can do with* `ifstream/ofstream`

3.2.2 Strings

You can attached a stream to a string and read from or write to it.

- You must include `#include<sstream>`
- `std::istringstream` - Read from string
- `std::ostringstream` - Write to string

Example 3.6 *Reading a value into a string using string streams*

```
int main(){
    int to = ??, hi = ??;
    ostring stream ss;
    ss << "Enter a number between" << 10 << "and" << hi;
    string s = ss.str();
}
```

Example 3.7 *Reading a value into a string and confirming it is a number using string streams*

```
int n;
while(true){
    cout << "Enter a number" << endl
    string s;
    cin > s;
    istringstream ss {s};
    if (ss >> n) break;
    cout << "I said " ;
}
cout << "You entered" << n << endl;
```

Example 3.8 *Example 3.1 Revjsted using String Streams*

```
int main(){
    string s;
    while(cin >> s){
        istringstream ss{s};
        int n;
        if (ss >> n) cout << n << endl;
    }
}
```

3.3 Strings

3.3.1 C vs C++ Strings

In C :

- array of characters (char * or char []) are terminated by a null terminator.
- In addition you must manage memory
- You must also get more memory as strings grow.
- Null terminators are also easy to overwrite

In C++ :

- Strings Grow as needed and no memory management is required
- Strings are safer to manipulate
- During Initialization, the value is a C string which is used to initialize a C++ string.

3.3.2 String operations

- Equality : `s1 == s2` or `s1 != s2`
- Comparison (Lexicographic Order) : `s1 <= s2`
- Length : `s.length`
- Get individual chars : `s[0], s[1], s[2]`
- Concatenate : `s3 = s1 + s2` or `s3 += s4`

3.4 Default Function Parameters

Example 3.9 *Read file function with default file*

```
void printWordsInFile (string name = "suite.txt"){
   ifstream file {name};
   string s;
   while (file >> s) cout << s << endl;
}

int main (){
   printWordsInFile("Suite2.txt");
   printWordsInfile(); //suite.txt
}
```

Note : Default Parameters must be last

3.5 Overloading

Example 3.10 *Functions to process different parameters in C*

```
int negInt(int m) {return n;}
bool negBool (bool b) {return b;)
```

Example 3.11 *Functions to process different parameters in C++*

- *Functions with different parameters lists can share the same name*

```
int neg(int m) {return n;}
bool neg(bool b) {return b;)
```

- *Compiler uses the number of types of arguments to decide which neg is called*
- *Overloads must differ in number or types of arguments. Functions cannot just differ on just return type*

Overloading explains how many functions are able to function. Functions such as `for #s, string, >>, <<, etc` rely on overloading

3.6 Structures

Example 3.12 *Structures in C++*

- *Structures are the same as C, except the struct keyword is not necessary*

```
struct Node{
    int data;
      Node *next; //Struct key word is not required
};
```

3.7 Constants

```
Const int maxGrade = 100; // must be initialized
```

Null Pointers in C++ : (`nullptr`) is the syntax for null pointer

```
Node n1={5,nullptr}; // Syntax for null pointer
//DO NOT SAY NULL or 0 !
```

Immutable Copies :

```
Const Node n2=n2}; // Can not change field
```

3.8 Parameter Passing

3.8.1 Review

```
void inc (int n) {++n}

int main (){
    int x = 5;
    inc (x)
    cout << x << endl; // prints 5
}
```

- Call by value : inc gets a copy of x, and increments the copy, so the original is unchanged.
- If a function need to modify a parameter, pass a pointer

3.8.2 References

C++ has another pointer like type, which is called a reference. Its why `cin >> x` is able to function without a address.

Example 3.13 - *Important*

```
int y = 10;
int &z = y; // z is an l value reference to int
            // like a const point
            // similar to int * const z = &y
```

- *References are like constant pointers with automatic dereferencing*

```
z=12; // (NOT *z = 12)
    // y is now equal to 12
```

```
int *p = &z; // gives the address of y
```

- *In all cases, z behaves exactly like y. Z is an alias for y*

3.8.2.1 Things You Can't Do

1. Leave them uninitialized : `int &x;`
   - must be initialized to something that has an address (an lvalue), since refs are pointers.
   - In short, values assigned to a reference MUST have an address
2. Create a pointer to a reference : `int &* x;`
   - References to pointers are OK : `int *&x = ______`
3. Create a reference to a reference : `int &&r;` (Means something different)
4. Create an array of references : `int &r[3] = {n, n , n};`

3.8.2.2 Things You Can Do

1. Pass references as function parameters : `void inc (int &n) {++n}`
   - This is why cin ¿¿ x works, as it takes in a reference
   - `istream &operator >> (istream &in, int&data);`
2. Pass-by-value : `int f(int n) {...} copies the argument`
   - If the argument is big, copying is expensive
   - `int f(reallyBig rb){...}` - Slow
   - `int g(ReallyBig &rb){...}` Fast, but you can't be sure that rb changes in the caller
   - `int h(const ReallyBig &rb){...}` Fast, no copy, and the parameter cannot be changed

Advice

- Prefer pass-by-constant-reference over pass-by-value for anything larger than a pointer.
- Unless the function needs to make a copy anyway, then ,maybe use pass-by-value
- Using pass-by-constant-reference can allow you to pass literal values to a reference, as the compiler has already been promised that the value will never change.
  - The compiler achieves this by creating a temporary location to hold the literal value, so a reference has somethings to point at.

3.9 Dynamic Memory Allocation

DO NOT USE malloc AND free IN C++

Instead use : `new` and `delete`, as they are type aware and less error prone.

Example 3.14 *Creating a heap object and deleting it with new/delete*

```
struct Node{
  int data;
  Node *next
}

Node *np = new Node;

...

delete np;
```

- All local variables reside on the stack
- Variables deallocated when they go out of scope (Stack is popped)
- Allocated memory resides on the heap
- Remains allocated until delete is called, if not deleted, memory leaks will occur.

Example 3.15 *Creating a array and deleting an array.*

```
Node *nArr = new Node[10];
...
delete [] nArr; //Special form of delete for arrays
```

Example 3.16 *Passing a pointer to heap data*

```
Node *getMeANode(){ //Returns a pointer to a heap data
    return new node;
}
```

3.10 Operation Overloading

Give meanings to c++ operators for our own types

Example 3.17 *Vector Operations using Operation Overloading*

```
struct vec{
    int x,y;
};

vec operator+(const Vec &v1, const vec &v2){
    vec v {v1.x + v2.x, x1.y + v2.y};
    return v;
}

```

```
vec operator*(const int k, const vec &v1){
    return {k * v1.x, k * v1.y} // ok because compiler knows its a vec based on return type
    //Handles 2*v, but not v*2
}

vec operator*(const vec &v1, const int k){ //different order tells compiler to use secondary function
    return k*v1;
}
```