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

Lecture 4: September 29, 2016

Lecturer: Brad Lushman *Notes By: Harsh Mistry*

4.1 Overloading << and >>

Example 4.1 - *Formatting Output Using >> and << Operators*

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

struct Grade {
  int theGrade;
};

ostream &operator<<(ostream &out, const Grade &g) {
  out << g.theGrade << "%";
  return out;
}

istream &operator>>(istream &in, Grade &g) {
  in >> g.theGrade;
  if (g.theGrade < 0) g.theGrade = 0;
  if (g.theGrade > 100) g.theGrade = 100;
  return in;
}

int main () {
  Grade g;
  while (cin >> g) cout << g << endl;
}
```

4.2 The Preprocessor

Transforms the code before the compiler sees it. # ___________ is the prepocessor directive. e.g `#include`

4.2.1 Define

Defined constants are useful for conditional compilation

Example 4.2 *Using define to make programming from multiple operating systems simpler*

```
#define IOS 1
#define BBOS 2
#define OS IOS (or BBOS)

#if OS == IOS
    short int publickey
#elif OS == BBOS
    long long int public key
#end if
```

Special Case :

```
#if 0
    ...
#endif
```

- Never true and all inner text is suppressed before it gets to teh compiler
- Useful as a heavy-duty ”comment out” to comment out anything including other comments

Example 4.3 *Define symbols via compiler arguments*

Bash :

```
g++14 -DX=15 define.cc -O define
./define
```

Define.cc :

```
int main(){
    cout << x << endl
}
```

- # ifdef NAME : True is Name has been defined
- # ifndef Name : True if Name has not been defined

Example 4.4 *Using ifdef and ifndef to define a debug mode*

debug.cc :

```
int main(){
    #ifdef DEBUG
        cout << "setting x=1" << endl;
    #endif
    x=1;
    while (x < 10) {
        ++x;
        #ifdef DEBUG
            cout << "x = " << x << endl;
        #endif
    }
    cout << x << endl;
}
```

Enabling Debugging :

```
g++14 -DDEBUG debug.cc -o debig
```

4.3 Separate Compilation

Split programs into composable modules, with :

- Interface (.h) : type definitions. function prototypes
- implementation - full definitions for all provided functions

Recall

- declaration - asserts existence
- definition - full details and allocates space

Example 4.5 *Simple example of compilation*

vec.h :

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

Vec operator+(const vec &v1, const vec &v2);
```

main.cc :

```
#include "vec.h"

int main(){
    Vec v{1,2}'
    v = v + v:
}
```

vec.cc

```

#include "vec.h"

Vec operator+(const vec &v1, const vec &v2){
    // definition goes here
}
```

Recall :

An entity can be declared many times, but defined at most once.

4.3.1 Compiling a binary from multiple source files

Adding a `-c` to `g++14` compiles the files only and does not generate a executable. By doing this, the compiler will produce an object file(.o).

Once object files are generated for each source file, you can link the object files to generate a executable.

- Compile only : `g++14 -c filename.cc`
- Link Object Files : `g++14 filename.o filename.o -o binary name`

Example 4.6 *Including Header Files*

linalg.h :

```
#include "vec.h"
```

linalg.cc :

```
#include "linalg.h"
#include "vec.h"
```

main.cc :

```
#include "linalg.h"
#include "vec.h"
```

This Example will not compile

- *main.cc and linalg.cc include vec.h, linalg.h. In addition, linalg.h includes vec.h*
- *This causes 2 copies vec.h to be included, thus struct Vec gets defined twice*

4.3.2 Include Guard

To prevent issues when a header file is included multiple times, format header files to check for previous definition.

```
#ifndef VEC.H
#define VEC.H
    // File Contents
#endif
```

The example above will only define vec.h if it has not previously been defined. After the first definition, the rest of the file is suppressed. **Always ensure your header files have include guards.**

4.3.3 Stuff to Never Do !!!!

1. **Never** ever **EVER** compile header files **EVER**
2. **Never** include .cc files
3. **Never** put `using namespace std;` in header files
   - the using directive will be forced upon any client that includes the file
   - Always say `std::cin`, `std::string`, `std::istream`, etc in headers

4.4 Classes

A big idea of OOP is that you can put objects instead structures

Example 4.7 *Function instead of a Structure*

```
struct Student {
    int a, b , c;
    float grade(){
        return a * 0.4 + b * 0.2 + c * 0.4
    }
};

Student s {60,70,80}

cou << s.grade() << endl;
```

- Class - essentially a structure type that can contain functions.
  - C++ has a class keyword. *Since this is CS246, we will use it later*
- Object - an instance of a class
- Function grade - called a member function
- Methods take a hidden extra parameter called **this**
  - pointer to the object on which the method was involved
  - `billy.grade()` is a pointer to the object billy

4.5 Initializing Objects

In C : `Student billy {60,70,80};`

The c way is OK, but a better way would be to write a method that does initialization. This method is called a constructor. To create a constructor, simply create a method within the object that shares the same name.

Example 4.8 *Basic Constructor Example*

```
struct Student{
    int a, b, c;
    float grade (){ ... }
    Student (int a, int b, int c){
        this->a = a
        this->b = b
        this->c = c
    }
}

Student billy {60, 70, 80}; //better
Student billy = Student {60, 70, 80} // Also works
```

- *If a constructor is defined the values are passed as arguments*
- *If no constructor is defined, the values are used to initialize the individual fields of student.*

Heap allocation : `Student *Billy = new student {60,70,80}`

Advantages of Constructors : Default parameters, overloading, and sanity checks

Example 4.9 *Constructor Example With Default Values*

```
struct Student{
    int a, b, c;
    float grade (){ ... }
    Student (int a = 0, int b = 0, int c = 0){
        this->a = a
        this->b = b
        this->c = c
    }
}
```

Note

Every class comes with a default constructor, which just default constructs any fields that are objects

Example 4.10 *Default Constructor Example*

```
Vec v; //default constructor (Does nothing in this case)
```

- *The default constructor goes away if you define your own*

Example 4.11 *Proof of Default Constructor*

```
struct v {
    int x, y;
    Vec (int x, int y) {
        this->x = x;
        this->y = y;
    }
}

Vec v {1,2}; // Works
Vec v; .. // Error , thus there must have been a default constructor in ex 4.9
```

Example 4.12 *Constants and References in Structures*

```
int z;
Struct MyStruct(){
    const int myConst = 6;
    int &myref = z;
}
```

What happens when a object is created?

1. Space is allocated
2. Fields are constructed
3. Constructor Body Runs

Example 4.13 *Initializing Constants using the Member Initialization List (MIL)*

```
int z;
Struct MyStruct(){
    const int myConst;
    int a, b, c;
    MyStruct(int a, int b, int c):
      myConst{myConst}, a{a}, b{b}. c{c} {}
}
```

- *You can initialize any field using MIL, not just constants and references*
- *Fields are initialized in the order in which they are declared in the class, even if the MIL orders them differently.*
- *MIL is sometimes more efficient then setting fields in the body*
- ***Embrace MIL, or else!!. It is your friend.***

Example 4.14 *Field Initialized Inline and MIL*

```
struct Vec{
    int x=0, y=0;
    vec(int x):x{x} {}
};
```

- *In the example, MIL always takes precedence*

4.5.1 Every Class Comes With

1. Default Constructor : Default constructs all fields, but is lost if you define your own
2. Copy Constructor : Just copies all fields
3. Copy Assignment Operator
4. Destructor
5. Move Constructor
6. Move Assignment Operator

Example 4.15 *Writing your own copy constructor*

```
struct Student{
    student (const Student &char):
      a{other, a}, b{other, b}, c{other, c}{} // Same as Default
};
```

Example 4.16 *Example case where default copy constructor does not work*

```
struct Node{
    int data
    Node *next;
    Node (int data, Node *next) : data {data}, next{}{}
};
Node *n = node Node{1, New Node {2, new Node {3, nullptr}}};
Node m = *n;
Node *p = new Node(*n)
```

- *Since pointers are used, the copy constructor only copies the first node, this is called a shallow copy.*

Example 4.17 *Deepcopy Copy Constructor*

```
struct Student{
    int data
    Node *next;
    Node (const Node &other) : data {other, data}, next{other.next?new Node(*other.next):
        nullptr}{}
};
```