Lecture Three

A Closer Look at Classes

Ref: Herbert Schildt, Teach Yourself C++, Third Ed^{n} (Chapter 3)

© **Dr. M. Mahfuzul Islam**

Professor, Dept. of CSE, BUET

Assigning Objects

- One object can be **assigned** to another provided that both objects are of the **same type**.
- By default, when one object is assigned to another, a **bitwise copy** of all the data members is made.

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

class myclass {
   int  a, b;
public:
   void set(int i, int j) { a = i; b = j; }
   void show() { cout << a << ‘ ‘ << b << “\n”;}
};

class yourclass {
    int a, b;
public:
    void set(int i, int j) { a = i;  b= j; }
    void show() { cout << a << ‘ ‘ << b << “\n”;}
}
```

```
int main(){
    myclass O1, O2;
    yourclass O3;

    O1.set(10,4);

    O2 = O1;     // OK
    O3 = O1;  // Error,  Objects- not same type

    O1.show();
    O2.show();

    return 0;
}
```

Problem with Assigning Objects

- When an object **pointing to dynamic memory allocation** is assigned to another object
  - both object **share** the **same memory**.
  - **Destroying** one object **release the common memory** and possibly cause **program crash**.

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

```
class strtype {
   char  *p;
   int len;
public:
   strtype(char *ptr);
   ~strtype() { cout << “Freeing....”; free(p);}
   void show();
};
```

```
void strtype::show(){
     cout << p << “ – length” << len  << “\n”;
}
```

```
strtype::strtype( char *ptr){
     len = strlen(ptr);
     p = (char *) malloc(len+1);
     if (!p) { cout << “Allocation error\n”;  exit(1); }
     strcpy(p, ptr);
}
```

```
int main(){
    strtype  s1(“This is a test.”), s2(“I like C++.”);

    s1.show();
    s2.show();
    s2 = s1;
    s1.show();
    s2.show();

    return 0;
}
```

Passing Object into Function

- **Parameter passing,** by default, is **called by value**. That is a bitwise copy is made.
- New object created in the function **does not call constructor,** but **destructor is called**.

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

class samp {
        int i;
public:
       samp(int n);
      ~samp();
       void seti(int n) { i = n;}
       int geti() { return i; }
};

samp::samp(int n){
     i= n;
     cout << “Constructing.....\n”;
}
samp::~samp(){
     cout << “Destructing.....\n”;
}
```

```
void sqr_it(samp o) {
     o.seti(o.geti() * o.geti());
     cout << “Copy: value of a:” << o.geti() << ‘\n’;
}

int main() {
     samp a(10);

     sqr_it(a);
     cout << “Main: value of a:” << a.geti() << ‘\n’;

     return 0;
}
```

OUTPUT:

```
Constructing….
Copy: value of a: 100
Destructing….
Main: value of a: 10
Destructing….
```

Problem in Passing Object into Function

- If the **object** used as the arguments **allocates dynamic memory** and **free** the memory then the destructor function is called and the **original object** is **damaged**.

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

```
class dyna {
        int *p;
public:
       dyna(int i);
      ~dyna() {free(p); cout << “Freeing…\n”;}
       int get () { return *p}
};
```

```
dyna:: dyna(int i){
       p = (int *) malloc(sizeof(int));
       if (!p) {
            cout << “Allocation problem\n”;
            exit(1);
       }
      *p = i;
}
```

```
void neg (dyna ob) {
      return -ob.get();
}
```

```
int main() {
     dyna o(-10);

     cout << o.get() << ‘\n’;
     cout << neg(o) << ‘\n’;
     cout << o.get() << ‘\n’;

     return 0;
}
```

OUTPUT:

```
-10
Freeing…   // when o is passed to neg(),
           // copy is created and
           // destroyed in neg()
10
NULL pointer assignment  // o.get()
NULL pointer assignment  // free(p) in destructor
Freeing…..//when o is destroyed
```

Solution to Passing Object into Function

- To pass the **address of the object**, not the object itself, i.e., **call by reference**
- A special type of constructor called “**copy constructor**” is used (Chap 5/Lecture 5).

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

```
class samp {
        int i;
public:
      samp(int n) { i = n; }
      void seti (int n) { i = n; }
      int geti() { return i; }
};
```

```
Void sqr_it(samp *o) {
     o.seti( o->geti() * o->geti() );
     cout << “copy: value of i: ” << o->geti() << ‘\n’;
}
```

```
int main() {
     samp a(10);

     sqr_it(&a);
     cout << “main: value of i: ” << a.geti() << ‘\n’;

     return 0;
}
```

OUTPUT:

```
copy: value of i: 100
main: value of i: 100
```

Returning Object from Function

- When an object is **returned** by a function, a **temporary object** is **created** which holds the return value. This object is return by the function.
- After the **value** has been **returned**, this object is **destroyed**.
- The **destruction** of this **temporary object** may cause **unexpected side effects**.

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

```
class samp {
     char *s;
public:
     samp() { s = ‘\0’; }
    ~samp() { if (s) free(s); count << “Freeing S\n”;}
    void show() {cout << s << ’\n’;}
    void set (char *str);
};
```

```
void samp::set(char *str) {
     s = (char *) malloc(strlen(str)+1);
     if(!s) { cout << “Allocation error\n”; exit(1); }
     strcpy(s, str);
}
```

```
samp input() {
      char s[80];
      samp str;
      cout <<”Enter a string: ”;
      cin >> s;
      str.set(s);
      return str;
}
```

```
int main() {
     samp ob;

     ob = input();
     ob.show();

     return 0;
}
```

Friend Function

- A friend function is **not a member** of a class but still has **access** to its **private elements**.

- **Three uses** of friend functions (1) to do **operator overloading**; (2) **creation** of certain types of **I/O functions**; and (3) one function to **have access** to the **private members** of **two or more different classes**.

- A friend function is a **regular non-member** function

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

```
class car {
      int passengers, speed;
public:
      car(int p, int s) { passengers = p; speed = s; }
      friend int sp_greater(car c, truck t);
};
```

```
class truck {
      int weight, speed;
public:
     car(int w, int s) { weight = w; speed = s; }
     friend int sp_greater(car c, truck t);
};
```

```
int  sp_greater (car c, truck t) {
     return c.speed – t.speed;
}
```

```
int main() {
      int t;
      car c(6, 55);
      truck t(2000, 72);

      t = sp_greater(c, t);
      if (t > 0) cout << ”Faster.\n”;
      else if (t==0) cout << “Equal.\n”;
              else count << “slower.\n”;
      return 0;
}
```

Forward declaration:
class truck;

Friend Function

- A **friend function** can be a **member of one class** and a **friend of another**.

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

```
class car {
      int passengers, speed;
public:
      car(int p, int s) { passengers = p; speed = s; }
      int sp_greater(truck t);
};
```

```
class truck {
      int weight, speed;
public:
      car(int w, int s) { weight = w; speed = s; }
      friend int car::sp_greater(truck t);
};
```

```
int  car::sp_greater (truck t) {
      return speed – t.speed;
}
```

```
int main() {
     int t;
     car c(6, 55);
     truck t(2000, 72);

     t = c. sp_greater(t);
     if (t > 0) cout << ”Faster.\n”;
     else if (t==0) cout << “Equal.\n”;
            else count << “slower.\n”;

      return 0;
}
```

t = c.sp_greater(t);

can be written by the scope resolution operator (::) as

t =c.car::sp_greater(t);

but this is unnecessary.