CSE 165/ENGR 140
Intro to Object Orient Program

Lecture 10 – References & Copy-Constructor
Lab #3 Due date:
  ◦ Problems 1 – 5: 2/19 (Thursday) at 11:59PM
  ◦ Problems 6 – 9: 2/27 (Thursday) at 11:59PM

Quiz on 2/26 during lecture

Reading assignment
  ◦ Ch. 7
Points to functions

- A function point is a variable that stores the address of a function.
- It allows a function to change its behavior when it is called separately.
  - The same sort function can either sort in an ascending or descending way.
  - A compare function can be passed as an argument.
- It enables “callback functions” or “event listener”.
  - In an graphic user interface, a function is called when a mouse click takes place.
void delete_func(void * pt) //A function to delete data pointed by a void pointer
{
    double * dpt = (double *)pt;
    cout<<"Deleting: " << *dpt << "\n";
    delete dpt;
}

In main()...

void (*del_func_ptr)(void * pt) = delete_func; //Define pointer to delete_func
doubleStack->setDeleteCallback(del_func_ptr); //Set function pointer in Stack.h to point to delete_func as well

In Stack.h...

void (*deletecb) (void *pt); //Declare a function pointer

setDeleteCallback ( void (*delcb) (void *pt) ) //Argument is a function pointer
{
    //Set pointer to point to delete_func using the input argument
}

Without (*) works as well

In cleanup()...

(*deletecb)(oldHead->data); //Delete data as appropriate type
delete oldHead;
References

- References are always tied to someone else’s storage.
  - When you change the value of a reference you are always changing the value of someone else’s variable/object.

- Similar to pointers, BUT:
  - References always manipulate someone else’s storage.
    - References cannot be null.
  - References must be initialized.
    - You cannot declare a reference without initialization.
  - A reference cannot be changed to refer to something else.
    - Assignment will assign contents, not make the reference to reference another object.
References

//: C11:FreeStandingReferences.cpp
#include <iostream>
using namespace std;

// Ordinary free-standing reference:
int y;
int& r = y;      // (1) When a reference is created, it must
                 // be initialized to an existing object.

const int& q = 12;  // (2) This is valid (note the const)

// References are always tied to someone else's storage:
int x = 0;
int& a = x;       // (3) a is a reference to x

int main() {
    a++;    // (4) we are actually incrementing x here
}
References in functions

- References are commonly used as function arguments and return values.
  - Any modification to the reference inside the function will cause changes to the argument outside the function.

```cpp
//: C11:Reference.cpp - Simple C++ references

int* f(int* x) { // 1) pointer passed to a function
    (*x)++;
    return x; // Safe, x is outside this scope
}

int& g(int& x) { // 2) reference passed to a function
    x++; // Same effect as in f()
    return x; // Safe, x is outside this scope
}
```
References in functions

//: C11:Reference.cpp - Simple C++ references

int& h() { // 3) function returning a reference
    int q;
    return q; // 3.1) this would generate an error since q is local
}

static int x; // 3.2) static makes x become a global variable
return x; // Safe, x lives outside this scope (even if not visible)
}

int main() {
    int a = 0;
    f(&a); // Sending a pointer to a to f: ugly (but explicit)
    g(a); // Sending a reference to a to g: clean (but hidden)
}
Passing a pointer by reference

#include <iostream>
using namespace std;

void increment(int*& i) { i++; } //Passing the reference of a pointer

int main() {
    int* i = 0; //i is a pointer
    cout << "i = " << i << endl;
    increment(i);
    cout << "i = " << i << endl;
} //://~

Output:

i = 0
i = 0x4
Passing objects as arguments

- Whenever possible, always pass an object to a function as a `const` reference.
- If the object has to be modified, then pass a simple (non-const) reference.
- Passing an object by value will include the overhead of constructor call and copy of contents, plus it is difficult to know when it will be a safe operation.
- Pointers are only helpful if you want the possibility of an optional object argument (since the pointer can be null)

```cpp
void process_event ( const Event& e, Event* newevent=0 );
```
Don’t do this!

//: C11:PassingBigStructures.cpp
struct Big {
    char buf[100];
    int i;
    long d;
} B, B2;

Big bigfun(Big b) { // Passing Big by value will copy 100 chars to
    // the new local Big b object!! Use a reference!

    b.i = 100; // Do something to the argument
    return b;
}

int main() {
    B2 = bigfun(B);
}
//: C11:HowMany.cpp - A class that counts its objects
#include <fstream>
#include <string>
using namespace std;
ofstream out("HowMany.out");

class HowMany {
    static int objectCount;
public:
    HowMany() { objectCount++; }

    static void print(const string& msg = "") {
        if(msg.size() != 0) out << msg << ": ";
        out << "objectCount = " << objectCount << endl;
    }

    ~HowMany() {
        objectCount--;
        print("~HowMany()");
    }
};

int HowMany::objectCount = 0;
// C11:HowMany.cpp - A class that counts its objects

// Pass and return BY VALUE:
HowMany f(HowMany x) {
    x.print("x argument inside f()"挝);
    return x;
}

int main() {
    HowMany h;
    HowMany::print("after construction of h");
    HowMany h2 = f(h);
    HowMany::print("after call to f()");
}

======Output:======
after construction of h: objectCount = 1
x argument inside f(): objectCount = 1
~HowMany(): objectCount = 0
after call to f(): objectCount = 0
~HowMany(): objectCount = -1
~HowMany(): objectCount = -2       !!!!!!!!
Fixing the problem:

```cpp
#include <fstream>
#include <string>
using namespace std;

ofstream out("HowMany2.out");

class HowMany2 {
    string name;  // Object identifier
    static int objectCount;

    public:
    HowMany2(const string& id = "") : name(id)
    { ++objectCount; print("HowMany2()"); }

    ~HowMany2() { --objectCount; print("~HowMany2()"); }

    HowMany2(const HowMany2& h) : name(h.name)  // ADDED: copy-constructor
    { name+= " copy"; ++objectCount; print("HowMany2(const HowMany2&)"); }

    void print(const string& msg = "") const {
        if(msg.size() != 0) out << msg << endl;
        out << '\t'<< name <<": "<<"objectCount = "<< objectCount << endl;
    }
};
```
Copy-construction

//: C11:HowMany2.cpp - Fixing the problem:

int HowMany2::objectCount = 0;

// Pass and return BY VALUE:
HowMany2 f(HowMany2 x) {
    x.print("x argument inside f()");
    out << "Returning from f()" << endl;
    return x;
}

int main() {
    HowMany2 h("h");
    out << "Entering f()" << endl;
    HowMany2 h2 = f(h);
    h2.print("h2 after call to f()");
    out << "Call f(), no return value" << endl;
    f(h);
    out << "After call to f()" << endl;
}
1) HowMany2()
2) h: objectCount = 1
3) Entering f()
4) HowMany2(const HowMany2&)
5) h copy: objectCount = 2
6) x argument inside f()
7) h copy: objectCount = 2
8) Returning from f()
9) HowMany2(const HowMany2&)
10) h copy copy: objectCount = 3
11) ~HowMany2()
12) h copy: objectCount = 2
13) h2 after call to f()
14) h copy copy: objectCount = 2
15) Call f(), no return value
16) HowMany2(const HowMany2&)
17) h copy: objectCount = 3
18) x argument inside f()
19) h copy: objectCount = 3
20) Returning from f()
21) HowMany2(const HowMany2&)
22) h copy copy: objectCount = 4
23) ~HowMany2()
24) h copy: objectCount = 3
25) ~HowMany2()
26) h copy copy: objectCount = 2
27) After call to f()
28) ~HowMany2()
29) h copy copy: objectCount = 1
30) ~HowMany2()
31) h: objectCount = 0
Preventing pass-by-value

“If I don’t make a copy-constructor, the compiler will create one for me. How do I know that an object will never be passed by value?”

- You can declare a private copy-constructor. You don’t even need to create a definition (unless one of your member functions or a friend function needs to perform a pass-by-value.)
Preventing pass-by-value

//: C11:NoCopyConstruction.cpp - Preventing copy-construction

class NoCC {
    int i;
    NoCC(const NoCC&); // No definition ok (just declaration)
public:
    NoCC(int ii = 0) : i(ii) {}  
};

void f(NoCC);  

int main() {  
    NoCC n;  
    f(n);      // 1) Error: copy-constructor called  
    NoCC n2 = n; // 2) Error: c-c called  
    NoCC n3(n); // 3) Error: c-c called  
}
Similar to pointers to functions, it is also possible to have “pointers to members”, however this is a less useful feature that is not commonly seen.

```cpp
//: C11:PointerToMemberFunction.cpp
#include <iostream>
using namespace std;

class Widget {
public:
    void f(int) const { cout << "Widget::f()\n"; }
    void g(int) const { cout << "Widget::g()\n"; }
    void h(int) const { cout << "Widget::h()\n"; }
    void i(int) const { cout << "Widget::i()\n"; }
};

int main() {
    Widget w;
    Widget* wp = &w;
    void (Widget::*pmem)(int) const = &Widget::h;
    (w.*pmem)(1);
    (wp->*pmem)(2);
}
```