CS 300 Data Structures

Fall 2021

Bellevue College

Rule of Big Three

Memory Management

- The dynamic creation and destruction of objects is always one of the bugbears of C/C++.
- It required the programmer to (manually) control the allocation of memory for the object, handle the object's initialization, then ensure that the object was safely cleaned-up after use and its memory returned to the heap.

Resource Management

- The constructor/destructor pair can be used to create an object that automatically allocates and initialises another object (known as the *managed* object) and cleans up the managed object when it (the manager) goes out of scope. This mechanism is generically referred to as resource management.
- A small warning here: make sure the new and delete operators ‘match’: that is, if the resource is allocated with new, then use delete; if the resource is allocated as an array (new[]) make sure array delete is used (delete[]). Failure to do so will lead to ‘Bad Things’ happening.

Rule of Big Tree

- The Rule of The Big Three states that if you have to write *one* of the functions (below) then you have to have a policy for *all* of them.
- The Destructor
- The Assignment Operator
- The Copy Constructor

Example:

```
#ifndef INTCELL_H_
#define INTCELL_H_

class IntCell {
    int* x = new int;
public:
    IntCell(int=0);
    virtual ~IntCell();
    int read(){return *x;}
    void write(int _x){*x = _x;}
};

#endif /* INTCELL_H_ */
```

```
#include "IntCell.h"

IntCell::IntCell(int _x) {
    *x = _x;
}

IntCell::~IntCell() {
    delete x;
}
```

Example cont.

```
#include <iostream>
#include "IntCell.h"
using namespace std;

void f(IntCell cell){
     IntCell temp(11);
     temp = cell;
     cout<<temp.read()<<endl;
}
int main() {
     IntCell cell1(13);
     f(cell1);
     cout<<cell1.read()<<endl;

     return 0;
}
```

OUTPUT

```
13
bigthree(84054,0x7fffac88e380) malloc: *** error for object 0x7fd
*** set a breakpoint in malloc_error_break to debug
```

What is the problem?

because it does a shallow copy, not a deep copy

```
#include <iostream>
#include "IntCell.h"

using namespace std;

void f(IntCell& cell){ //call by reference
    cout<<cell.read()<<endl;
}

int main () {
    IntCell cell(13);
    f(cell);
    cout<<cell.read()<<endl;
    return 0;
}
```

```
Output:
13
13
```

```
#include <iostream>
#include "IntCell.h"

using namespace std;

void f(IntCell cell){ //call copy constructor
    cout<<cell.read()<<endl;
}

int main () {
    IntCell cell(13);
    f(cell);
    cout<<cell.read()<<endl;
    return 0;
}
```

```
Output:
13
13
malloc: *** error
```

```
#include <iostream>
#include "IntCell.h"

using namespace std;

void f(IntCell& cell){ //call by reference
    IntCell temp;
    temp = cell; //assignment operator
    cout<<temp.read()<<endl;
}

int main () {
    IntCell cell(13);
    f(cell);
    cout<<cell.read()<<endl;
    return 0;
}
```

```
Output:
13
13
malloc: *** error
```

Problem

- the default assignment operator performs a member-wise copy (shallow copy)
- When temp goes out of scope (at the end of f()) it is destroyed and it deletes its pointer – just as it should.
- When cell1 goes out of scope at the end of main() it, too, tries to delete its pointer. However, that region of memory has already be deleted (by temp) so we get a run-time error!

Solution

- **Recall:** The Rule of The Big Three states that if you have to write *one* of the functions (below) then you have to have a policy for *all* of them.
  - The Destructor
  - The Assignment Operator
  - The Copy Constructor

Write assignment operator, operator=

```
IntCell& IntCell::operator=(const IntCell& other){
      int* pTemp = new int(*(other.x));
      delete this->x;
      x = pTemp;
      return *this;
}
```

ASSIGNMENT OPERATOR

- Our assignment operator function now implements the correct class copy semantics ("deep copy").
- The assignment operator returns a reference to itself. This is so expressions like this work:
- mgr1 = mgr2 = mgr3;

Write copy constructor

- the compiler-supplied copy constructor does a member-wise copy of all the `IntCell`'s attributes
- need to write your own constructor, which overrides the compiler supplied one.

```
IntCell::IntCell(const IntCell& other){
    *x = NULL;
    if(other.x != NULL){
        x = new int(*(other.x));
    }
}
```

COPY CONSTRUCTOR

- Note the signature of the copy constructor - it takes a reference to a `const IntCell` object.

```
#ifndef INT_CELL
#define INT_CELL

class IntCell{
    private:
        int* x;
    public:
        IntCell(int);
        IntCell(const IntCell&); //copy constructor
        int read() const;
        void write(int);
        IntCell& operator=(const IntCell&); //copy assignment operator
        ~IntCell();
        operator<<
};
```

```cpp
#include "IntCell.h"
#include <iostream>
using namespace std;

IntCell::IntCell(int _x){
    cout<<"in constructor"<<endl;
    x = new int(_x);
}
IntCell::IntCell(const IntCell& other){

    x = NULL;
    if(other.x != NULL){
        int storedValue = *(other.x);
        x = new int(storedValue);
    }
        cout<<"in copy constructor"<<endl;
}
IntCell& IntCell::operator=(const IntCell& other){
    if(this != &other){  //self assignment
        delete x;
        x = new int(*(other.x));
    }
    return *this;
}
int IntCell::read() const{
    return *x;
}
void IntCell::write(int _x){
    *x = _x;
}
IntCell::~IntCell(){
    cout<<"in destructor"<<endl;
    delete x;
}
```

```cpp
#include <iostream>
#include "IntCell.h"

using namespace std;

void print(IntCell);

int main(){
    IntCell cell(20);
    IntCell cell2(10);

    cell = cell2; //assignment operator

    print(cell);

    return 0;
}

void print(IntCell c){
    cout<<c.read()<<endl;
}
```

output?

```
in constructor
in constructor
in copy constructor
10
in destructor
in destructor
in destructor
```

When is the copy constructor is invoked?

Four scenarios

- **Explicit copy construction:** The most explicit way to invoke the copy constructor on an object is to create said object, passing in another object (of the same type) as a parameter.
- **Object initialization:** C++ makes the distinction between *initialisation* and *assignment*. If an object is being initialised the compiler will call a constructor, rather than the assignment operator.
- **Pass-by-value parameters:** When objects are passed to functions by value a copy of the caller's object is made. This new object has a constructor called, in this case the copy constructor.
- **Function return value:** If a function returns an object from a function (by value) then a copy of the object is made. The copy constructor is invoked on return. Return value optimization: depending upon the compiler, constructor or copy constructor is invoked.

When is the copy constructor is invoked?

Four scenarios

- **Explicit copy construction**
  - `IntCell c1(11);`
  - `IntCell c2(c2);` //explicit copy construction
- **Object initialization**
  - `IntCell c1(11);`
  - `IntCell c2 = c2;` //calls the copy constructor, not `operator=`
- **Pass-by-value parameters**

```
void f(IntCell c) //note pass by value
{
   //…
}
int main(){
    IntCell cell(12);
    f(cell); //calls copy constructor
}
```

- **Function return value**

```
IntCell f() //note pass by value
{
   return IntCell(3);
}
```