Assorted Topics

In this lesson we will discuss a few assorted C++ topics and (annoying) pitfalls.

- Destructors
- Shallow Copies vs. Deep Copies
- Copy Constructors.

Destructors (L/L 14.3)

A **destructor** is a member function of a class that is called *when an object goes out of scope*.

In particular this means that it is called to destroy a local variable when the function it belongs to returns control. The destructor is automatically called as the last action of the function before the function call ends. It returns the memory used by the objects to the stack for reuse.

A class may have only one destructor. This destructor member function

- Has the same name as the class preceeded by a tilde (˜), e.g., `~ComplexClass()`
- Has no return type
- Has no parameters

The system automatically provides every class with a destructor. This default constructor

does not eliminate dynamically created variables.

Therefore, *if your object uses dynamically allocated variables you must write your own destructor to clean up after it.*

If you did not do this you might end up 'losing' memory.

```
//thing1.h
//Header file for thing class
//No copy constructor
class thing
{
public:
   thing(int val);
   thing();
   ~thing();  //destructor

   void   set(int val);
   int    get();

private:
   int* x;
};
```

```
//thing1.cpp
//Implementation file for thing class
//No copy constructor
#include <iostream.h>
#include <stddef.h>  // for NULL
#include "thing1.h"

thing::thing(int val)
{ cout << "Constructing new thing with val = " << val <<endl;
  x = new int;
  *x = val;}

thing::thing()
{ cout << "Constructing new thing using default constructor"
       << endl;
  x = new int;
  *x = 0;}

thing::~thing()
{ cout << "Deleting thing with val = " << *x << endl;
  delete x;
  x = NULL;
}

void thing::set(int val)
{ *x = val;}

int thing::get()
{ return *x;}
```

```
//Destructor.cpp
//Tests out destructor

#include <iostream.h>
#include "thing1.h"

void Do2(int level)
{ cout << "Entering Do2 with level = " << level << endl;
  thing TT(level);
}

void Do1(int level)
{ cout << "Entering Do1 with level = " << level << endl;
  thing TTT(level);
  Do2(level+1);
}

void main()
{ thing T(1);
  cout << endl;

  Do2(2);
  cout << endl;

  Do1(3);
  cout << endl;

  Do2(5);
  cout << endl;
}
```

```
Constructing new thing with val = 1

Entering Do2 with level = 2
Constructing new thing with val = 2
Deleting thing with val = 2

Entering Do1 with level = 3
Constructing new thing with val = 3
Entering Do2 with level = 4
Constructing new thing with val = 4
Deleting thing with val = 4
Deleting thing with val = 3

Entering Do2 with level = 5
Constructing new thing with val = 5
Deleting thing with val = 5

Deleting thing with val = 1
```

Shallow Copies

Recall that objects of class *thing* have one private member **int* x**. Now suppose that main() started with

```
A1:  thing T(1);
A2:  thing TT = T;
A3:  T.set(2);
```

Surprisingly, TT.get() now returns 2 and not 1. Why?
The commands are equivalent to performing

```
B1: *(T.x) = 1;
B2: TT.x   = T.x;
B3: *(T.x) = 2;
```

The second line means that

T.x and TT.x are pointers pointing to the same thing;

they **do not** point to two different locations containing the same value!

When constructing TT on line A2 the system made it *shallow copy* of T. This is a copy in which only the statically defined variables are copied but not the dynamically allocated ones.

A *deep copy* would also copy the dynamically allocated variables. For example, a deep copy would replace line B2 by

TT.x = new int; *(TT.x) = *(T.x);

To enable the making of deep copies during construction we introduce *copy constructor*s

```
//Shallow.cpp
//Example of Shallow Copy
#include <iostream.h>
#include "thing1.h"

void Do(thing Tcopy, int level)
{ Tcopy.set(level); }

void main()
{ thing T(1);
  thing TT = T;

  cout << "*(T.x) == "  <<  T.get()  << endl;
  cout << "*(TT.x) == " <<  TT.get() << endl << endl;

  T.set(2);
  cout << "*(T.x) == "  <<  T.get()  << endl;
  cout << "*(TT.x) == " <<  TT.get() << endl << endl;

  Do(T,3);
  cout << "After calling Do(T,3) *(T.x) == "<<T.get();
  cout << endl  << endl;
}
```

```
Constructing new thing with val = 1
*(T.x) == 1
*(TT.x) == 1

*(T.x) == 2
*(TT.x) == 2

Deleting thing with val = 3
After calling Do(T,3) *(T.x) == 3

Deleting thing with val = 3
Deleting thing with val = 3
```

Copy Constructor (Savitch. p.652)

A *copy constructor* is a constructor that has one parameter that is of the same type as the class. This parameter **must** be a call-by reference one and is frequently also a **const** one.

```
thing::thing(const thing& T)
{  x = new int;
  *x = *(T.x);
}
```

The copy constructor should be written so as to provide a deep copy.

Like every other constructor the copy constructor can be called explicitly, e.g.,

```
thing T(1);  thing A(T);
```

The copy constructor is also called *automatically* in three situations

- To copy an object that is passed by value to a a function

  ```
  int Do(thing Tcopy)
  { return 2*Tcopy.get(); }
  ```

- When an object is returned as the value of a function

  ```
  thing ReturnThing(int level)
  {thing localT;
   localT.set(level);
   return localT;}
  ```

- When an object is declared and initialized using =

  ```
  thing T(1);
  thing TT = T;  // copy constructor called
  ```

Note: Defining the copy constructor **does not** change the standard assignment operator $=$. This operator must be overloaded or else it will produce a shallow copy. For example

```
thing T(1);
thing TT;
TT = T;  // copy constr not called
         // Assignment (=) called
```

will make TT a shallow copy of T.

```
//thing2.h
//Header file for thing class

class thing
{
public:
   thing(const thing& T); //copy constructor
   thing(int val);
   thing();
   ~thing();

   void   set(int val);
   int    get();

private:
   int* x;
};
```

```
//thing2.cpp
//Implementation file for thing class
#include <iostream.h>
#include <stddef.h>  // for NULL
#include "thing2.h"

thing::thing(const thing& T)
{ cout << "Copy Constructing new thing with val = "  << *(T.x) <<endl;
  x = new int;
  *x = *(T.x);
}//end copy constructor

thing::thing(int val)
{ cout << "Constructing new thing with val = "  << val <<endl;
  x = new int;
  *x = val;
}


thing::thing()
{ cout << "Constructing new thing using default constructor" << endl;
  x = new int;
  *x = 0;
}

thing::~thing()
{ cout << "Deleting thing with val = " << *x << endl;
  delete x;
  x = NULL;
}

void thing::set(int val)
{ *x = val;}


int thing::get()
{ return *x;}
```

```
//Deep.cpp
//Example of Deep Copies
//Using copy instructor

#include <iostream.h>
#include "thing2.h"

void Do(thing Tcopy, int level)
{ Tcopy.set(level); }

thing ReturnThing(int level)
{thing localT;
 localT.set(level);
 return localT;
   }

void main()
{ thing T(1);
  thing A(T);   // copy constructor called
  A.set(5);
  thing TT = T;  // copy constructor called

  cout << "*(T.x) == "  <<  T.get()  << endl;
  cout << "*(TT.x) == " <<  TT.get() << endl << endl;;

  T.set(2);
  cout << "*(T.x) == "  <<  T.get()  << endl;
  cout << "*(TT.x) == " <<  TT.get() << endl << endl;

  Do(T,3);      // copy constructor called
  cout << "After calling Do(T,3) *(T.x) == "  <<  T.get();
  cout << endl  << endl;

  cout << "Returned value is";
  // copy constructor called
  cout << ReturnThing(4).get() << endl << endl;
}
```

```
Constructing new thing with val = 1
Copy Constructing new thing with val = 1
Copy Constructing new thing with val = 1
*(T.x) == 1
*(TT.x) == 1

*(T.x) == 2
*(TT.x) == 1

Copy Constructing new thing with val = 2
Deleting thing with val = 3
After calling Do(T,3) *(T.x) == 2

Returned value is
   Constructing new thing using default constructor
Copy Constructing new thing with val = 4
Deleting thing with val = 4
4

Deleting thing with val = 4
Deleting thing with val = 1
Deleting thing with val = 5
Deleting thing with val = 2
```