

Introduction to Programming using C++

Lecture Seven: Designing Classes

Carl Gwilliam

gwilliam@hep.ph.liv.ac.uk

<http://hep.ph.liv.ac.uk/~gwilliam/cppcourse>

delete in Objects

- If a class has data members on the heap, the memory associated to them must be freed when an object of that class goes out of scope.
- This is done by placing delete statements in the class's **destructor**.

“I don't need to bother with that. The compiler doesn't even warn me. Will it really affect the running of my program?”

```

:                                     [Point.cpp]
// constructor
Point::Point(float x, float y, float z) {
    // value placed on heap
    itsX = new float(x);
    :
}
:
// destructor: delete memory on heap
Point::~~Point() {
    delete itsX;
    :
}
:
float Point::getX() {
    //deref ptr to get val
    return *itsX;
}
:

class Point {
public:
    :
private:
    float* itsX;
    float* itsY;
    float* itsZ;
};                                     [Point.h]
```

Memory Leaks

- If memory used by object is not freed when pointer goes out of scope, we lose access to the object and the memory cannot be reused until program terminates.
- This is a **memory leak**.
- If `getArraySum()` is called frequently as part of a larger program then this memory leak becomes an important efficiency issue.

```
int main() {  
    int i = 0;  
    while (i < 500) {  
        // 500 000 memory addresses  
        // created on heap  
        float* result = getArraySum();  
        i++;  
        // no freeing of memory  
    }  
    return 0;  
}  
  
float* getArraySum() {  
    float* anArray = new float[100];  
    :  
    return anArray;  
}
```

More Memory Leaks

- Memory leaks can be introduced by reassigning a ptr before the original object has been deleted.
- The undeleted object on the heap can now longer be accessed as the pointer containing the object's address in memory has been reassigned => the memory the object is taking up cannot be freed until the program is terminated.

```
int main() {  
    float* anArray = new float[100];  
    :  
    // must delete memory  
    // associated to original array  
    delete anArray;  
    anArray = new float[200];  
    :  
    delete anArray;  
    return 0;  
}
```

- Memory leaks are easily introduced by misuse of pointers. But the damage from pointer mistreatment does not end here...

Stray Pointers

- An uninitialised pointer refers to a random memory address. This is called a **stray**, **wild** or **dangling** pointer.

```
int main() {  
    int carl = 27;  
    int* lecturer;  
    // ptr used before assigned  
    (*lecturer++);  
    :  
    lecturer = &carl;  
    return 0;  
}
```

```
int main() {  
    int carl = 27;  
    int* lecturer = NULL;  
    // compiler catches error  
    (*lecturer++);  
    :  
    lecturer = &carl;  
    return 0;  
}
```

- The stray pointer can be used to access and overwrite the value stored in a random memory address.
- The pointer can be made NULL (or 0) if a pointer has to be declared without being initialised.

Stray Pointers to the Free Store

- The meaning of **delete** is often misunderstood.
- The pointer to is not deleted, instead the memory on the free store that the pointer referred to has is freed. The pointer still exists, and can be misused:

```
int main() {  
    int andy = 27;  
    int* lecturer = new int(andy);  
    :  
    delete lecturer;  
    :  
    *lecturer = 28;           // error  
    return 0;  
}
```

Stray Pointers to the Free Store

- If the wild pointer is used after the memory has been freed there are two possible side-effects:
 - It can access an address in memory used by other processes on the computer.
 - It can overwrite the value in that address there by interrupting the process using that area of memory.
- Every time an object is deleted on the free store, ensure that
 - the pointer to that object will go out of scope after the delete, or
 - assign the pointer to NULL.

Writing a **Sample** Class

- A class **sample** can be constructed that has all the functionality of the **stats** program. It can also be extended to incorporate new routines.
- Data members:
 - An array to store the input sample
 - Size of the array storing the sample
- Class methods:
 - Calculate the mean of the sample
 - Calculate the standard deviation of the sample
 - Order the sample by lowest to highest value
- Constructors:
 - Declare an array to store the sample
 - Declare an integer to hold the size of the sample
 - Request the sample from the user
 - Request the size of the sample from the user

The Basic **Sample** Class

- Start with the basic interface to **Sample** class:

```
// sample class declaration [Sample.h]
class Sample {
public:
    Sample();    // empty constructor
    ~Sample();   // destructor
    // accesor methods (const and some inline)
    int getSize() const {return the_size;}
    double* getSample() const {return the_sample;}
    void showSample() const;
    double mean() const;
    double stdev() const;

private:
    int the_size;           // private data
    double* the_sample;     // members
    int setSize();          // setters called
    double* setSample(int size); // by constructor
};
```

Sample Constructor and Destructor

- Constructor initialises data members by calling **private** member funcs to request data interactively from the user of **Sample**.
- The array is placed on the heap rather than the stack.
- If **Sample** object only contains the size and address of array then size of object will be constant regardless of size of input sample.

```
:  
Sample::Sample() { //constructor  
    the_size = setSize();  
    the_sample = setSample(the_size);  
}  
Sample::~~Sample() { //destructor  
    delete the_sample;  
}  
:  
int Sample::setSize() {  
    cout << "How big is the sample?" << endl;  
    int size; cin >> size;  
    return size;  
}  
double* Sample::setSample(int size) {  
    double* sample = new double[size];  
    // read sample from keyboard  
    :  
    return sample;  
}
```

[Sample.cpp]

Additional Constructors

- Two more constructors can be added to the class:
 - First reads the sample from a given filename.
 - Second converts an array (and it's size) into a sample object.

```
// constructor to read sample from file    [Sample.cpp]  
Sample::Sample(char* filename) { ... }  
// constructor with sample input  
Sample::Sample(double* sampleArray, int size) { ... }
```

```
#include "Sample.h"  
int main() {  
    Sample height("heightsample"); //filenames are  
    Sample weight("weightsample"); //char arrays  
    double ageVal[5] = {7.2,18.6,21.3,34.7,45.6};  
    Sample age(ageVal,5);  
    :  
}
```

Ordering Routines

- The order routine from **stats** can now be included in the **sample** class. The order routine has two functions:
 - Return ordered sample back to user whilst keeping the original sample intact.
 - Use function call as to order sample inside the object.
- Can also extend to order from highest first as well as lowest first

```
class Sample {                                     [Sample.h]
public:
    :
    double mean() const;
    double stdev() const;
    double* order(int rank = 1) const;
    void selfOrder(int rank = 1);
```

```

:
};
double* Sample::order(int rank) const {
:
return new SampleArray;
}
void Sample::selfOrder(int rank) {

```

[.cpp]

```

}
#include "Sample.h"
int main() {
:
    double* newheight = height.order();
    weight.selfOrder(2);
}
[UseSample.cpp]

```

Correlation Class Method

- An input sample is now stored in a persistent way as long as the **sample** object remains in scope.
- Moreover, the functionality of the **sample** class can be extended to include interactions between **sample** objects.
- An example of this is a correlation method:

```
double Sample::correlation(const Sample& b) const {  
    :  
    return correlation;  
}  
[Sample.cpp]
```

- A **sample** object is passed (by reference) into a **sample** class method to calculate the correlation between the two samples.

Passing by Constant Reference

- In the **correlation** member function a reference to a **sample** object is passed in rather than the entire object.
- Passing by reference is a more efficient mechanism for objects because a local copy of the object does not have to be created.
- But by passing a reference to the object the function is able to modify the contents. To forbid modification the reference must be declared **const**. This is known as **passing by constant reference**.

```
double Sample::correlation(const Sample& b) const {  
    :  
    return correlation;  
}
```

[Sample.cpp]

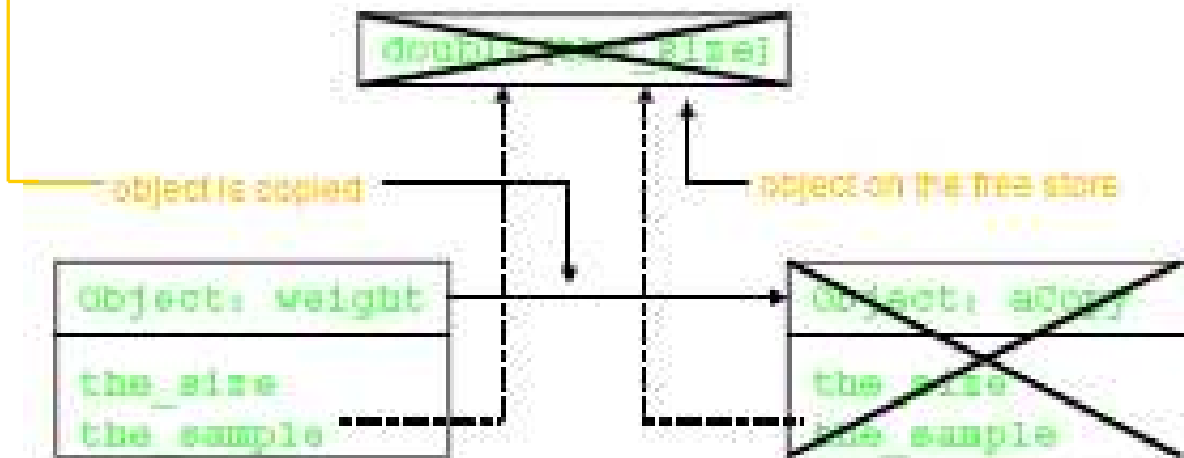
Copying Objects

How do I copy the contents of an existing **sample** object into a new **sample** object?

- The compiler provides a default **copy constructor** that performs a **memberwise** or **shallow** copy of an object.
- If the member data are pointers then the address held by a pointer is copied to the new object. This causes two problems:
 - The original and the copy have member data that both point to the same object. A change to the member data in one object would lead to a member data modification in the other.
 - If either the original or the copy go out of scope then the memory both point to is freed so the remaining object will contain a stray pointer.

Copying Objects

```
int main() {  
    :  
    Sample weight("weightsample");  
    if (makeCopy) {  
        Sample aCopy(weight); // shallow copy  
        :  
    } // copy goes out of scope => d'tor  
    : // frees memory containing the_sample  
}
```



Copy Constructor

- Default memberwise copying can be overwritten by writing a copy constructor which takes obj of same type.

```
Name::Name(const Name&) { ... }
```

- The copy constructor for the sample class:
 - copies the value of the sample size
 - creates a new array on the free store and copies the contents of the existing sample into the new array.

```
// copy constructor
Sample::Sample(const Sample& rhs) {
    the_size = rhs.the_size;           // assign size of array (straight copy)
    the_sample = new double[the_size]; // creat new array on heap and
    int iSample = 0;                   // copy every element (not address)
    while (++iSample <= the_size) {
        *(the_sample + (iSample-1)) = *(rhs.the_sample + (iSample-1));
    }
}
```

Overloading Operators

- How do I add two objects together
- What is meant by the above statement?
 - Add the contents of the sample array element-wise or concatenate the samples?
- The problem lies in the ambiguity of what it means to add two Sample objects together.
- The exact definition of addition can be included in the sample class. This is known as **operator overloading**

```
return_type operator sign (parameters) { ... }
```

```
Sample comb = weight + height
```

```
// is actually equivalent to
```

```
Sample comb = weight.operator+(height)
```

Adding Two Samples

- The addition operator is overloaded in the **sample** class:
- Almost all of the operators can be overloaded in an intuitive way, including
 - assignment operators, unary operators, relation operators ...

```
Sample Sample::operator+ (const Sample& rhs) {  
    // get total size  
    int newsize = this->the_size + rhs.the_size;  
    // declare new array of total size  
    double newSampleArray[newsize];  
    // copy array from lhs of + into new array, and  
    // copy array from rhs of + into new array  
    :  
    return Sample(newSampleArray,newsize);  
}
```

```
int main() {  
    :  
    Sample comb=  
    weight + height;  
    :  
}
```

The **this** Keyword

- The keyword **this** represents a pointer to the object whose member function is being executed. It is a pointer to the object **itself**.
- One of its uses can be to check if a parameter passed to a member function is the object itself (i.e. has the same address).

```
bool AClass::isSame(AClass& object)
{
    if (this == &object) return true;
    else return false;
}
```

Same

```
#include "AClass.h"
int main() {
    AClass a;
    AClass* b = &a;
    if (b->isSame(a)) cout << "Same\n"
    return 0;
}
```

- It is also used in overloading the assignment operator...

Overloading Assignment

- The compiler provides a default assignment operator which does a memberwise copy to object of same type.
- The assignment operator can be overloaded to ensure that pointers are treated properly (as with copy c'tors).

// Return by reference

```
Sample& Sample::operator= (const Sample& rhs) {  
    // if lhs & rhs are equal (same address) don't do anything  
    if (this == &rhs) return *this;  
    the_size = rhs.the_size; // get size  
    int iSample = 0;         // get sample  
    while (++iSample <= the_size) {  
        *(the_sample + (iSample-1)) =  
            *(rhs.the_sample + (iSample-1));  
    }  
    return *this;  
}
```

```
int main() {  
    :  
    Sample = height("heightsample");  
    Sample aCopy = height;  
    //Sample aCopy.operator=(height)  
    aCopy = aCopy;  
    :  
}
```

```

class Sample { // The sample class declaration
public:
    Sample(); // constructors
    Sample(char* filename);
    Sample(double* sample, int size);
    Sample(const Sample&); // copy constructor
    ~Sample(); // destructor
    Sample operator+ (const Sample& rhs); // overloaded operators
    Sample& operator= (const Sample& rhs);
    int getSize() const {return the_size;} // public member functions
    double* getSample() const {return the_sample;}
    void showSample() const;
    double mean() const;
    double stdev() const;
    double* order(int rank = 1) const;
    void selfOrder(int rank = 1);
    double correlation(const Sample& b);
private:
    int the_size; // private member data
    double* the_sample;
    int setSize(); // private member functions
    double* setSample(int size);
};

```