Classes

A simple cash register class in

```
//classregisterclass.h

#include <iostream>
#include <cmath>

using namespace std;


class Cashregister
{
public:

	Cashregister();
	float getdraweramt ();

	void sale (float );
	void returnitem (float);
private:
	float draweramt;
};

Cashregister::Cashregister()
{
	draweramt = 0;
}

float Cashregister::getdraweramt ()
{
	return draweramt;
}

void Cashregister::sale(float price)
{

	draweramt = draweramt + price;
}


void Cashregister::returnitem(float price)
{

	if (draweramt <= 0){
		cout << "Getting money from manager" << endl;
		draweramt = (fabs(draweramt) + draweramt) + 100;
	}
}
```

Driver for cash register class

```
//cashregisterdriver.cpp

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

int
main()
{
	Cashregister Walmart;

	Walmart.sale(25);

	cout << Walmart.getdraweramt() << endl;

	Walmart.returnitem(8.40);

	cout << Walmart.getdraweramt() << endl;

	Walmart.returnitem(128.40);

	cout << Walmart.getdraweramt() << endl;

	return 0;
}
```

When a class' method does not change any objects, it should be made constant

```
class Cashregister
{
public:

	//CONSTRUCTOR
	Cashregister();

	//CONSTANT METHODS
	float getdraweramt () const;

	//MODIFIER METHODS
	void sale (float );
	void returnitem (float);
private:
	float draweramt;
};
```

Constructors

Every class has a default constructor. If a specific initialization needs to be done, this is done in a user defined constructor.

- Constructors are called when a class variable is declared.
- The name of the constructor method is the name of the class.
- Constructors do not have any return values, not even void. Therefore no return data type is designated in the method's prototype.
- Initializes data members of an object.
- If a constructor is not supplied for a class the system will create a "do nothing" constructor. Good programming is to always include a constructor.

We can initialize data members using intializers in the constructor.

```
Stuff s (10, 2, 2010);

class Stuff
{
public:
      Stuff (int val1, int val2, int val3);
      void setmonth(int);
      void setday (int);
      void (setyear(int);
private:
      int month;
      int day;
      int year;
};

Stuff::Stuff (int val1, int val2, int val3)
{
      setmonth(val1);
      setday(val2);
      setyear(val3);
}

Stuff::Stuff (  )
{
      setmonth(1)
      setday(1)
      setyear(1900)
}
```

```
Letter l ('a', 'b', 'c');

Mixed m (5, 'd' 4.56);
```

syntax for initializers– enclosed in parentheses , to right of object name, before semicolon

Constructors can be defined in the class definition with default initializations.

Inline functions

Small methods can be implemented as inline functions. Inline functions add to code readability and an ad to efficiency. At compile time, the compiler replaces the inline function call with the function's compiled code. This saves execution time. The downside is that the resulting code contains multiple copies of the inline code.

Using Multiple Files To Implement Classes

Most classes are very long with a large number of methods. To make this manageable C++ has the ability to implement a class in two files, a header file, and an implementation file. The header file contains the class definition, and the implementation file contains code for the member functions.

```
//classregisterclass4.h

class Cashregister
{
public:
	//CONSTRUCTOR
	Cashregister();

	//CONSTANT METHODS
	float getdraweramt () const {return draweramt;}

	//MODIFIER METHODS
	void sale (float price) {draweramt = draweramt + price;}
	void returnitem (float);
private:
	float draweramt;
};
```

```cpp
//cashregister4.cpp
#include "cashregisterclass4.h"
#include <iostream>
#include <cmath>

using namespace std;

Cashregister::Cashregister()
{
	draweramt = 0;
}

void Cashregister::returnitem(float price)
{
	draweramt = draweramt - price;
	if (draweramt <= 0){
		cout << "Getting money from manager" << endl;
		draweramt = (fabs(draweramt) + draweramt) + 100;
	}

}

//cashregisterdriver.cpp

#include <iostream>
#include "cashregisterclass4.h"
using namespace std;

int
main()
{
	Cashregister Walmart;

	Walmart.sale(25);

	cout << Walmart.getdraweramt() << endl;

	Walmart.returnitem(8.40);

	cout << Walmart.getdraweramt() << endl;

	Walmart.returnitem(128.40);

	cout << Walmart.getdraweramt() << endl;

	return 0;
}
```

Sometimes in long programs, header files may be included more than once. To prevent a redefinition of objects, we use a macro guard.

```
//classregisterclass4.h

#ifndef CASHREGISTER_H
#define CASHREGISTER_H

class Cashregister
{
public:
	//CONSTRUCTOR
	Cashregister();

	//CONSTANT METHODS
	float getdraweramt () const {return draweramt;}

	//MODIFIER METHODS
	void sale (float price) {draweramt = draweramt + price;}
	void returnitem (float);
private:
	float draweramt;
};


#endif
```

Operator Overloading

We can extend C++ operators so that they can operate with new data types or classes.

Ex. << bitwise left shift and stream insertion
>> bitwise right shit and stream extraction

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


int
main()
{
	int y;
	int x;
	int i;

	cout << "Please enter an integer number: ";
	cin >> y;

	cout << "The number is: "<< y << endl;
	x = y << 1;
	cout << "The left shift number is: " << x << endl;

	y = 1;

	cout << "The number is: "<< y << endl;
	x = y >> 1;

	cout << "The right shift number is: " << x << endl;

	return 0;

}

/*Output:
Please enter an integer number: 4
The number is: 4
The left shift number is: 8
The number is: 1
The right shift number is: 0
Press any key to continue
*/
```

+, - works with float, integer, pointers, …

Operators such as +, −, ==, << , >> are actually functions. You can think of them as being function calls with the operators as the function parameters.

2 + 3 can be thought of as + (2, 3).

Thus, they can be overloaded in a similar manner to other overloading other types of functions.

We have already come across operators that are overloaded.

+ and − are used with integer, and float data types.

Syntax for operator overloading:

function definition is written as we previously have done. Function name becomes the keyword operator, followed by the symbol.

Ex. optional_return_type operator *op* (parameter list)
where *op* represents the operator that is overloaded such as the +, -, <, etc.

It is good programming to overload operators so that their new function is similar to their old function. For example we might want to overload the + operator to add two Cashregister objects.

To do this we can write an overloaded function that has two Cashregister objects in its parameter list and then returns a third Cashregister object.

```
Cashregister & operator + (Cashregister & A, Cashregister & B)
{
      float D;
      D =  A.getdraweramt() + B.getdraweramt();
      return Cashregister(D);
}
```

One subtlety of the above. Notice the return statement. Here, we called the Cashregister constructor function. A constructor function returns an object of the class. Our overloaded + function requires a Cashregister object to be returned. Therefore writing the return in this manner is perfectly valid. This is equivalent to writing the function like this:

```
Cashregister & operator + (Cashregister & A, Cashregister & B)
{
      float D;
      D =  A.getdraweramt() + B.getdraweramt();

      Cashregister C(D);
      return C;
}
```

We must be careful to include the function definition in the header file.

```
#include <iostream>
#include <cassert>


using std::iostream;
using std::ostream;


class Cashregister
{
public:

      Cashregister(){draweramt = 0;};
      Cashregister(float);
      float getdraweramt () {return draweramt;};
      void setdraweramt (float amt) { if (amt >= 0) draweramt = amt;
                              else draweramt = 0;};

      void sale (float );
      void returnitem (float);
private:
      float draweramt;
};

Cashregister & operator + (Cashregister &, Cashregister &);

inline Cashregister::Cashregister (float initcash)
{
      if (initcash < 0) draweramt = 0;
      else draweramt = initcash;

}
```

What if we wanted to include an overloaded operator inside the class definition? This means doing the function on the class itself. We need one less parameter in the parameter list since we can use the functions and data members of the class itself, as needed. For example, to add money to `draweramt` (in case we run low) we can overload the + operator as follows:

```
#include <iostream>
#include <cassert>


using std::iostream;
using std::ostream;

class Cashregister
{

public:
      Cashregister(){draweramt = 0;};
      Cashregister(float );
      float getdraweramt () {return draweramt;};
      void setdraweramt (float amt)
      {
            if (amt >= 0) draweramt = amt;
                              else draweramt = 0;\
      };
      Cashregister & Cashregister::operator + (Cashregister & B);
      void sale (float );
      void returnitem (float);
private:
};    float draweramt;



inline Cashregister::Cashregister (float initcash)
{
      if (initcash < 0) draweramt = 0;
      else draweramt = initcash;

}

#include "Cashregister3.h"
#include <iomanip>
#include <iostream>
#include <cmath>

using std::cout;
using std::endl;
using std::setprecision;
using std::setiosflags;
```

```
using std::ios;

void Cashregister::sale(float itemamt)
{
      float total;

      total = 0;
      total = itemamt + .075 * itemamt;

      draweramt = draweramt + total;
}


void Cashregister::returnitem(float itemamt)
{
      float total;

      total = 0;

      total = itemamt + .075 * itemamt;

      cout << total << endl;

      if (draweramt < 0){
            cout << "Getting money from manager" << endl;
            draweramt = (fabs(draweramt) + draweramt) + 100;
      }
}

Cashregister & Cashregister::operator + (Cashregister & B)
{
      return Cashregister(draweramt + B.getdraweramt( ));


}
```

All operators can be overloaded except for scope resolution operator (: :), member operator (.) and the dereference operator (*)

Some more rules:

1) Associate property of operator cannot be changed ex. y = 2 + 3 + 4;

+ is left to right associative. Any overloaded + has to be left to right associative. We can always change associative property using parentheses.

2) "arity" of an operator can't be changed (binary or unary) In other words we can't take a binary operator and make it unary ex. >> is binary , thus we can't define a unary >>

Those operators that are both unary and binary must have each of those versions overloaded separately.

3) We cannot overload an operator to change its meaning on built in data types. Ex. We cannot overload the + for integers, floats, etc. Overloaded operators must have a user defined data type such as a class as one if its operands.

4) Overloading one operator does not mean that another operator that is related to that operator is also overloaded.

Ex. Overloading + does not mean += is overloaded
Overloading == does not mean != is overloaded

Overloaded <<

```
Cashregister C(100):
cout <<  C;
```

Two operands are cout and C. The function call should return a value, but what is the data type of the returned value?

Consider the following statement:

```
cout << x << y;
```

what really happens is : `(cout << x) << y;`

Therefore << needs to return a `cout` object. `cout` is an object of the `ostream` class.

Thus we can write the overloaded function as this:

```
ostream & operator << (ostream & out, Cashregister & C)
{
      out << "The drawer amount is:  " << setprecision(2)
            << setiosflags(ios::fixed) <<  C.getdraweramt( )
      << endl; return out;
}
```

The >> and << operators cannot be overloaded as member functions. A member function for a binary operator must return an object of that class. Since we need an ostream object returned, the function needs to be defined outside of the class

```
using std::iostream;
using std::ostream;

class Cashregister
{
friend ostream & operator << (ostream &, Cashregister &);
public:

      Cashregister(){draweramt = 0;};
      Cashregister(int );
      const float getdraweramt () {return draweramt;};

      void sale (float );
      void returnitem (float);
private:
      float draweramt;
};

inline Cashregister::Cashregister (int initcash)
{
      if (initcash < 0) draweramt = 0;
      else draweramt = initcash;

}

#include "Cashregister.h"
#include <iomanip>
#include <iostream>
#include <cmath>

using std::cout;
using std::endl;

void Cashregister::sale(float itemamt)
{
      float total;

      total = 0;
      total = itemamt + .075 * itemamt;

      draweramt = draweramt + total;
}
```

```
void Cashregister::returnitem(float itemamt)
{
      float total;

      total = 0;

      total = itemamt + .075 * itemamt;

      cout << total << endl;

      if (draweramt < 0){
            cout << "Getting money from manager" << endl;
            draweramt = (fabs(draweramt) + draweramt) + 100;
      }
}

ostream & operator << (ostream & out, Cashregister & C)
{
      out << "The drawer amount is:  " << setprecision(2)
            << setiosflags(ios::fixed) <<  C.getdraweramt( )
      << endl; return out;
}
```

Operator functions as class members

1) The *this* pointer can be used to implicitly reference one of the class object arguments.
2) Assignment operator overloading has to be done as a member function
3) The leftmost or if only one argument that argument, has to be a class object.

Operator functions as nonmember functions

1) Class arguments are explicitly listed in the function call
2) Used when the left operand is an object of a different class or a built in type.
3) If the operator function needs to access private or protected class members then the function should be made a friend of the class
4) Operator functions need not be friends if the proper set and accessor functions are in the public part of the class.

Problem: Overload the >> operator to read data into the cashregister.