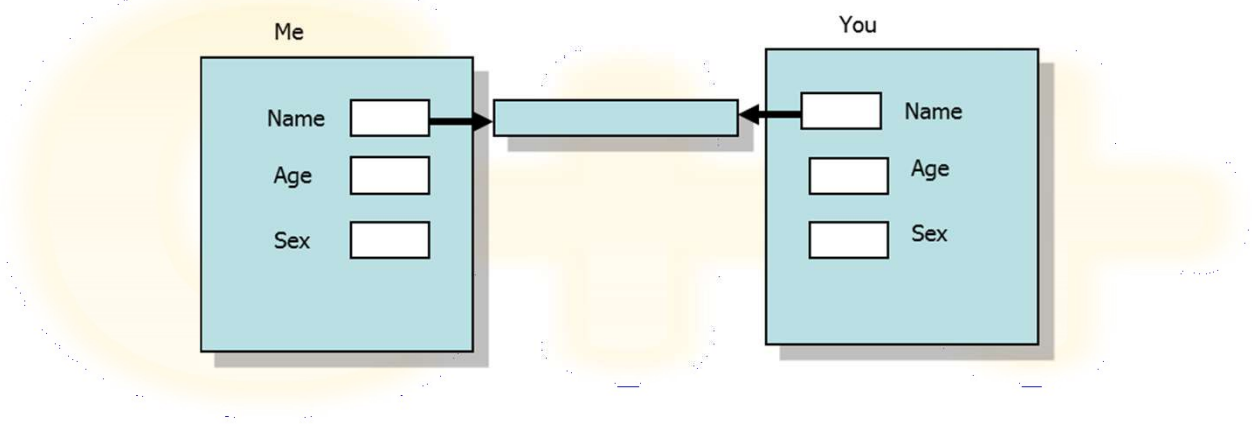
Advanced Constructors
&
Operator Overloading

Contents

- Objectives
  - To be able to use operator overloading
- Contents
  - Copy Constructors
  - Move Constructors
  - Conversion Constructors
  - Operator Overloading
  - Operator =
  - Move Assignment
  - Operator +
  - Operator ++
  - Boolean Operators
  - Global Operators

Copy Constructors



3

If you do not specify your own copy constructor, the compiler will generate one of its own. This will perform a shallow copy of me to you. The compiler copies the char * name pointer, but not the memory at that pointer.

Copy Constructors

```
 8 public:
 9     Person(void);
10     Person(const char* name, int age, char gender);
11     Person(const Person& p);
```

```
 5 Person::Person(void)
 6 {
 7     name = new char[10];
 8     strcpy(name,"Anonymous");
 9     age = 0;
10     gender = '?';
11     cout << "Person default constructor\n";
12 }
13 Person::Person(const char* n, int a, char g)
14 {
15     name = new char[strlen(n) + 1];
16     strcpy(name, n);
17     age = a;
18     gender = g;
19     cout << "Person overloaded constructor\n";
20 }
21 Person::Person(const Person& p)
22 {
23     name = new char[strlen(p.name) + 1];
24     strcpy(name, p.name);
25     age = p.age;
26     gender = p.gender;
27     cout << "Person copy constructor\n";
28 }
```

4

The solution to this problem is to code your own copy constructor which is simply another overloaded constructor. In this case the copy constructor will take a Person object as a parameter declared & implemented in the Person class as shown above. The copy constructor takes care of creating a new memory area for the name and copying all parts of the object being copied.

lvalues & rvalues

```
printReference (const String& str)
{
        cout << str;
}

printReference (String&& str)
{
        cout << str;
}
```

```
string me( "alex" );
printReference(  me );       // calls the first printReference
                             //function, taking an lvalue reference

printReference( getName() );      // calls the second
                                  // printReference function,
                                  // taking a mutable rvalue
                                  // reference
```

5

Copying can take place in all sorts of situations. Sometimes copying is vital, at other times it is undesirable, and in still other situations code could be made more efficient if it could be avoided in some way.

C++11 introduced the concept of move semantics to reduce copying. What are move semantics? They are a way of reducing copying with temporary objects.

In C++, there are rvalues and lvalues. An lvalue is an expression whose address can be taken, a locator value essentially, an lvalue provides a (semi)permanent piece of memory. You can make assignments to lvalues. Rvalues are… well, rvalues are not lvalues. An expression is an rvalue if it results in a temporary object.

Move Constructor

```
1 #pragma once
2 class Copy
3 {
4 public:
5     Copy();                 // default constructor
6     ~Copy();                // destructor
7     Copy(const Copy& clone);// copy constructor
8     Copy(Copy&& clone);     // move constructor
9 private:
10    int* largeMemBlock;
11    int memBlockSize;
12 };
```

```
1 #include "Copy.h"
2 #include <iostream>
3 #include <cstring>
4 using namespace std;
5
6 Copy::Copy()                // default constructor
7 {
8     largeMemBlock = new int[1000000];
9     memBlockSize = 1000000;
10 }
11 Copy::~Copy()               // destructor
12 {
13     if (largeMemBlock != nullptr)
14     {
15         delete[] largeMemBlock;
16     }
17 }
18 Copy::Copy(const Copy& clone)   // copy constructor
19 {
20     cout << "Copy constructor called - Copying largeMemBlock" << endl;
21     largeMemBlock = new int[clone.memBlockSize];
22     memcpy(largeMemBlock, clone.largeMemBlock, clone.memBlockSize * 4);
23     memBlockSize = clone.memBlockSize;
24 }
25 Copy::Copy(Copy&& clone)        // move constructor
26 {
27     cout << "Move constructor called - Moving largeMemBlock" << endl;
28     largeMemBlock = clone.largeMemBlock;
29     clone.largeMemBlock = nullptr;
30     memBlockSize = clone.memBlockSize;
31 }
```

6

The most common pattern you'll see when working with rvalue references is to create a move constructor and move assignment operator (which follows the same principles). A move constructor, like a copy constructor, takes an instance of an object as its argument and creates a new instance based on the original object. However, the move constructor can avoid memory reallocation because we know it has been provided a temporary object, so rather than copy the fields of the object, we will move them.

Conversion Constructors

```
 4 #include "Employee.h"
 5 using namespace std;
 6
 7 class Person
 8 {
 9 public:
10     Person(void);
11     Person(const char* name, int age, char gender);
12     Person(const Person& p);
13     Person(const Employee& e);
14     char* GetName() const;
15     int GetAge() const;
```

```
29 Person::Person(const Employee& e)
30 {
31     name = new char[strlen(e.GetName()) + 1];
32     strcpy(name, e.GetName());
33     age = e.GetAge();
34     gender = e.GetSex();
35     cout << "Person constructed from Employee\n";
36 }
```

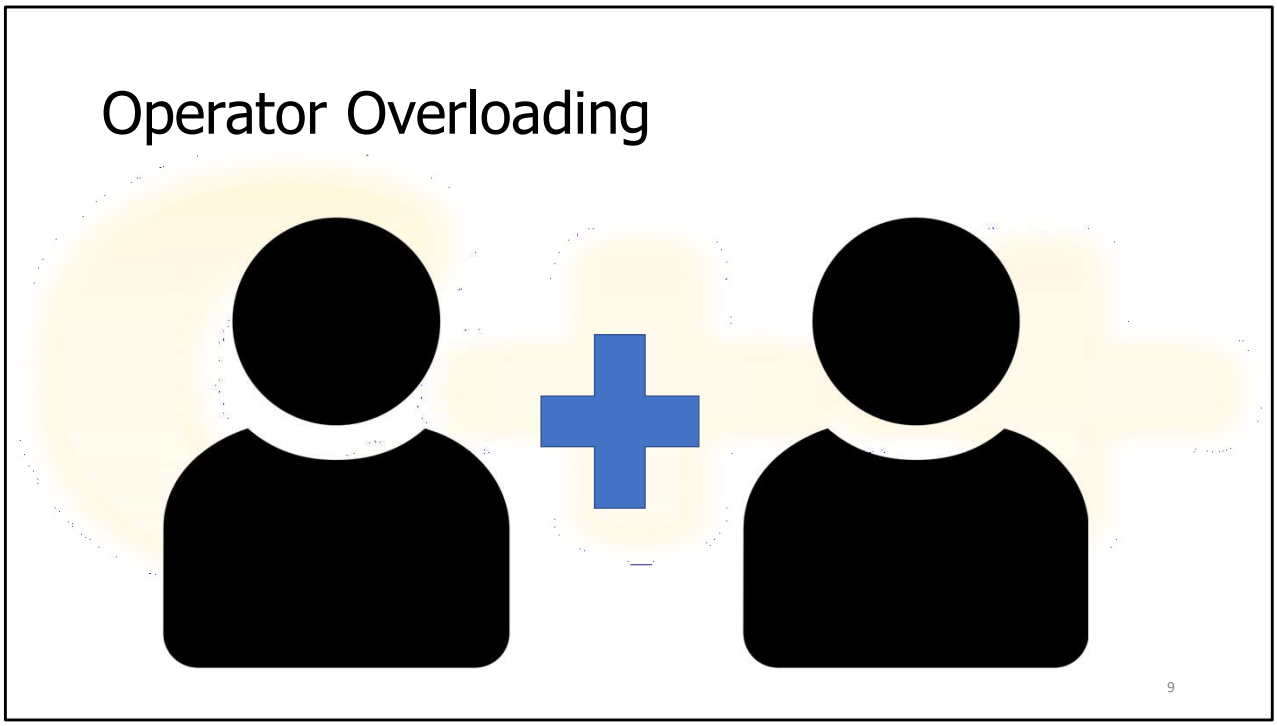
7

Sometimes it is desirable to be able to construct one type of object from another. For example, let's suppose our project has an employee class & a person class (not part of a hierarchy!). If the employee is similar to the person, it may make sense to be able to construct a person from an employee and maybe vice-versa. However, the compiler cannot automatically generate code to do this as we have two different types & it does not know how to do the conversion. You can, however, code a constructor to do that for you.

Explicit

8

This keyword is a declaration specifier that can only be applied to in-class constructor declarations. Constructors declared explicit will not be considered for implicit conversions. Explicit conversions can only be performed by means of a cast.



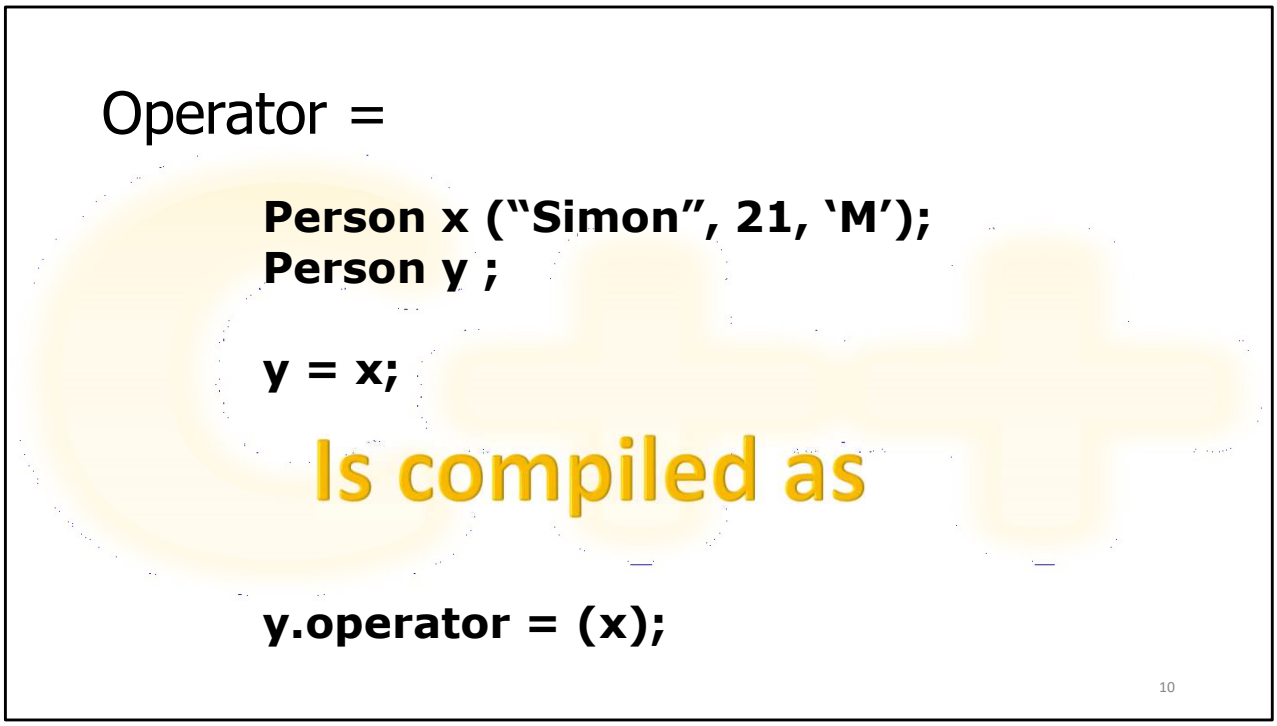
What is the effect of adding 2 Person objects together? Once all the sniggers havedied down you might come to realise that in computing terms it has no meaning. The compiler does not know how to add 2 Person objects (or any other objects for that matter!) This si where operator overloading comes in.

As well as being able to overload functions and class member functions you can also overload most of the standard operators for objects.

Operator overloading is subject to a set of rules. The operators that you cannot overload are the scope resolution operator ::, the ternary conditional operator ?:, the member selection operator ., the sizeof operator and the dereference pointer to class member operator .*.

In addition you cannot create new operators. For example, you might think it a good idea to implement an operator ** to raise a number to a power, forget it, you can't.

Apart from that the rest of the operators are fair game for overloading.



It stands to reason that if we can create one object as a copy of another (see the copy constructor discussed earlier) we should also be able to assign one object to another. Once again though we come across the problems of shallow copies not copying the memory at a pointer. To overcome this problem we must implement an overloaded assignment operator to perform the copying for us. Generally speaking it is a de-facto standard that if you implement either a copy constructor or an overloaded assignment operator you should also implement the other one too.

When dealing with operators the compiler does some behind the scenes conversion of the code. For example, the statement:

X = Y;

Can be thought of as:

X.operator = (Y);

In fact, this almost gives us the prototype for the operator = function.

Operator =

```
 6 class Person
 7 {
 8 public:
 9     Person(void);
10     Person(const char* name, int age, char gender);
11     Person(const Person& p);
12     Person& operator= (const Person& p);
13     char* GetName() const;
```

```
29 Person& Person::operator= (const Person& rhs)
30 {
31     delete [] name;
32     name = new char[strlen(rhs.name) + 1];
33     strcpy(name, rhs.name);
34     age = rhs.age;
35     gender = rhs.gender;
36     return *this;
37 }
```

11

There are a couple of interesting points in this code. First of all the initial delete of the name pointer. Since we are assigning one person to another, the name already points to some memory & it must be deleted before the new name memory is allocated and copied. Secondly the return value of the function. C++ allows you to type expressions of the form:

X = Y = Z;

Move Assignment

```
1 #pragma once
2
3 class Copy
4 {
5 public:
6     Copy();                              // default constructor
7     ~Copy();                             // destructor
8     Copy(const Copy& clone);             // copy constructor
9     Copy(Copy&& clone);                  // move constructor
10    Copy& operator=(const Copy& rhs);    // Assignment operator
11    Copy& operator=(Copy&& rhs);         // Move Assignment operator
12 private:
13    int* largeMemBlock;
14    int memBlockSize;
15 };
```

```
42 Copy& Copy::operator=(Copy&& rhs)   // Move Assignment operator
43 {
44     delete[] largeMemBlock;
45     cout << "Move assignment operaotor called" << endl;
46     largeMemBlock = rhs.largeMemBlock;
47     memBlockSize = rhs.memBlockSize;
48     rhs.largeMemBlock = nullptr;
49     rhs.memBlockSize = 0;
50     return *this;
51 }
```

12

Earlier we looked at how move constructors can be used to make code more efficient when dealing with temporary objects or rvalues. The same logic can be applied to move assignment. In this example a class Copy has been created that has an assignment operator implemented and also a move assignment operator. The full code for this example is in the MoveAssignment folder on the media accompanying this course.

Operator +

```
10      void SetPence(int pence);
11      int GetPounds()const;
12      int GetPence() const;
13      Currency operator+(const Currency& rhs);
14  private:
```

```
33 Currency Currency::operator+(const Currency& rhs)
34 {
35     Currency temp;
36     temp.pounds = this->pounds + rhs.pounds;
37     temp.pence = this->pence + rhs.pence;
38     if (temp.pence >= 100)
39     {
40         ++temp.pounds;
41         temp.pence -= 100;
42     }
43     return temp;
44 }
```

13

the operators that can be overloaded follow the conventions mentioned in the previous section. In order to better illustrate operator overloading a new class will be used. Adding 2 persons together does not make much sense so we will be using a Currency class. The code for this class is defined in the OperatorOverloading example on the course media.

It makes sense that currency amounts can be added together. In fact, it makes sense that currencies should be able to use all the arithmetic operators +, -, * & /. * could be used when purchasing multiple items at the same price to get a total cost & / could be used to divide a restaurant bill between a number of people.

This example implements the + operator.

Question

How would you implement Operator* that multiples a currency value by n where n is an integer?

Answer

How would you implement Operator* that multiples a currency value by n where n is an integer?

```
Currency operator*(const int rhs);

Currency Currency::operator*(const int rhs)
{
        double tempamt = this->pounds + (this->pence /100.0);
        tempamt *=rhs;
        return Currency(int(tempamt), (tempamt-int(tempamt))*100);
}
```

15

This raises an interesting question. If you can do currency *5, can you do 5 * currency??? This will be tackled shortly.

Operator ++

```
12      int GetPence() const;
13      Currency operator+(const Currency& rhs);
14      Currency operator++(int postfix);   // postfix has dummy argument
15      Currency& operator++();             // prefix
16  private:
17      int pounds;
```

```
45  Currency Currency::operator++(int postfix)      // postfix has dummy argument
46  {
47      Currency temp(*this);
48      ++this->pounds;
49      return temp;
50  }
51  Currency& Currency::operator++()                // prefix
52  {
53      ++this->pounds;
54      return *this;
55  }
```

16

The ++ (and --) operator is a special case for two reasons. Firstly, it is a unary operator. There is no right hand side to the expression. Secondly, it can take two forms – prefix and postfix. Strictly speaking if you overload any of the prefix / postfix ++ or – operators you should also implement the others. For simplicity we will only overload the prefix & postfix ++ operators for the Currency class. When we use ++ on a floating point number it adds one to the whole part leaving the decimal part unchanged. This is what we will do with our Currency example. i.e. add one whole pound to the Currency.

The operator is ++ regardless of whether it is used in prefix or postfix form. There is no way of distinguishing between the two forms, but they operate in quite different ways. To get round this problem the postfix form is given a dummy int parameter by the compiler as shown in this example.

Boolean Operators

```
15     Currency& operator++();              // prefix
16     bool operator==(const Currency& rhs);
17     bool operator<(const Currency& rhs);
18 private:
```

```
56 bool Currency::operator==(const Currency& rhs)
57 {
58     return ((this->pounds == rhs.pounds) && (this->pence == rhs.pence));
59 }
60 bool Currency::operator<(const Currency& rhs)
61 {
62     return ((this->pounds < rhs.pounds)||
63             ((this->pounds == rhs.pounds) && (this->pence < rhs.pence)));
64 }
```

17

The boolean operators include operator ==, operator !=, operator < etc. They all have similar prototypes so only operator== & operator< will be implemented here. The main difference between these operators & the other operators that we have already looked at is that the return type will be bool.

Operator== & operator< for our Currency class are demonstrated on this slide.

Operator << & >>

```
19     bool operator<(const Currency& rhs);
20     friend ostream& operator<<(ostream& stream, const Currency& c);
21     friend istream& operator>>(istream& stream, Currency& c);
22 private:
```

```
67 ostream& operator<<(ostream& stream, const Currency& c)
68 {
69     stream << "£" << c.pounds << "." <<  c.pence;
70     return stream;
71 }
72 istream& operator>>(istream& stream, Currency& c)
73 {
74     stream >>  c.pounds >>  c.pence;
75     return stream;
76 }
```

18

It would be nice if we could simply cout a Currency value. Unfortunately, the compiler does not know how to generate code to do this. We must overload the << & >> operators to make this work. However, therein lies a problem. The statement:

cout << cash;

would be translated to:

cout.operator << (cash);

This implies that we must add a new operator to the cout object. We could implement the << operator using the get accessors of the Currency class, however, that assumes that the accessors are public & available for us to use.

Once again C++ provides a solution in the form of friends. You can declare a function or an entire class as a friend. Doing so allows the friend function access to all the private members of the class. In other words it breaks the encapsulation rules albeit in a controlled manner. For this reason friend functions / classes should be kept to a bare minimum.

Friend functions are not members of a class. The private / protected / public access specifiers do not apply to them. They are ordinary global functions

with special privileges. Further, because they do not belong to a class / object they do not have a this pointer.

Functors or Function Objects

```
1 #pragma once
2 #include <iostream>
3 using namespace std;
4
5 class Add1
6 {
7 public:
8     // overload function call operator accept two integer arguments
9     // return their sum
10    int operator() (int a, int b);
11 };
12
13 class Add2
14 {
15 private:
16    int storedNumber;
17 public:
18    Add2(int n) : storedNumber(n) {}    // Overloaded Constructor
19    // overload function call operator add argument to stored state
20    int operator() (int a);
21 };
22
23 class Compare
24 {
25 public:
26    bool operator()(int a, int b);
27 };
```

19

A functor is pretty much just a class which defines the operator(). That lets you create objects which "look like" a function. Their real power will become more apparent when we deal with the Standard Template Library algorithms.

Functors or Function Objects

```
 1 #include <iostream>
 2 #include "Classes.h"
 3 using namespace std;
 4
 5
 6 int Add1::operator() (int a, int b)
 7 {
 8    return a + b;
 9 }
10
11 int Add2::operator() (int a)
12 {
13    return storedNumber + a;
14 }
15
16 bool Compare::operator()(int a, int b)
17 {
18     return (a == b);
19 }
```

20

Operator () is overloaded just like any other operator

Functors or Function Objects

```
 1 #include <iostream>
 2 #include "Classes.h"
 3 using namespace std;
 4
 5 int main()
 6 {
 7     Add1 a1;
 8
 9     int total = a1(15,21);
10     cout << total << endl;
11
12     Add2 a2(21);
13     cout << a2(10) << endl;
14     cout << a2(15) << endl;
15
16     Compare c;
17
18     cout << boolalpha << c(10, 10) << endl;
19     cout << boolalpha << c(10, 20) << endl;
20
21     return 0;
22 }
```

```
g++ -g -Wall -o functors Classes.cpp Functors.cpp
[sbailey@localhost Functors]$ ./functors
36
31
36
true
false
```

21

The above code defines 3 classes, each with a functor. Add1 defines a functor with 2 arguments. The functor simply adds the 2 arguments & returns the sum. The Add2 class takes things one step further. This time, the class contains sored state. The state is set using an overloaded constructor. When the functor is called the argument is added to the stored value & the result returned. Finally, class Compare compares 2 values & returns true or false. This is simply to demonstrate that a functor can return any type of value.

Global Operators

- Operators that are required to be binary symmetric operators can only be overloaded as global operators.
- Operators that you wish to subject to accessor limitations (public, private, protected) must be member operators.
- Operators which only modify the left hand operand may be declared as member operators.
- Certain functions can only be overloaded as member functions. For example, operator =.
- Member function operator overloading may be preferable if the data that the operator requires access to is complex or not easily accessible or secure.
- Certain operators can only be overloaded as global operators for example operator<< & operator>>

22

All but one of the operators that we have looked at so far have been members of a class. The exception was the output operator operator <<. This was a global operator. Some operators cannot be overloaded as member operator functions, for example the input & output operators. While others can only be overloaded as member functions for example the (), [], -> and assignment operators. The rest can be either global or member operators. This poses the question "which operators are best overloaded as global operators?"

Lets return to the question I set earlier of 5 * Currency. We cannot implement this as a member operator because the compiler always works on the left hand side & passes the right hand side for member operators. 5 is a numeric literal not an object.

Global Operators

```
22 private:
23     int pounds;
24     int pence;
25 };
26
27 Currency operator+(const Currency& c, int pence);
28 Currency operator+(int pence,const Currency& c);
```

```
80 Currency operator+(const Currency& c, int pence)
81 {
82     Currency temp;
83     int pennies;
84     pennies = c.GetPence()+pence;
85     temp.SetPounds(c.GetPounds()+(pence/100));
86     temp.SetPence(pennies%100);
87     return temp;
88 }
89 Currency operator+(int pence,const Currency& c)
90 {
91     Currency temp;
92     int pennies;
93     pennies = c.GetPence()+pence;
94     temp.SetPounds(c.GetPounds()+(pence/100));
95     temp.SetPence(pennies%100);
96     return temp;
97 }
```

23

The code on this slide illustrates the overloading of operator+ as a binary symmetric operator for the Currency class used earlier. In this implementation I have used it to add a number of pence to the currency i.e Currency +5 or 5 + Currency.

Note that because the functions are not members of the Person class we have no access to the private data members of a Person so we must either use friend functions as described earlier, or accessor functions.

C++ Classes as Value Types

```
16 class MyRefType
17 {
18 private:
19     MyRefType& operator=(const MyRefType&) = delete;
20     MyRefType(const MyRefType&) = delete;
21 public:
22     MyRefType() {}
23     virtual ~MyRefType() {}
24 };
```

Reference type with operator= & copy constructor deleted

24

Value types are sometimes viewed from the perspective of memory and layout control, whereas reference types are about base classes and virtual functions for polymorphic purposes. By default, value types are copyable, which means there's always a copy constructor and a copy assignment operator. (See earlier in this chapter). For reference types, you make the class non-copyable (disable the copy constructor and copy assignment operator) and use a virtual destructor, which supports their intended polymorphism. The copy constructor & operator = can be disabled wit he c++11 delete keyword.. Value types are also about the contents, which, when they're copied, always give you two independent values that can be modified separately. Reference types are about identity - what kind of object is it? For this reason, "reference types" are also referred to as "polymorphic types".

One of the constraints on value objects is that the values of the instances of the objects never change after they are set in a constructor. This also implies that all operations on value objects must return a new object. Value objects are immutable. Immutability is an important requirement. The values of a value object must be immutable once the object is created. Therefore, when the object is constructed, you must provide the required values, but you must not allow them to change during the object's lifetime.

C++ Classes as Value Types

```
class Currency
{
public:
    Currency& operator=(const Currency&);
    Currency(const Currency&);
    Currency(const char* name, const double amnt);
    void Display()const;
    ~Currency();
private:
    char* currencyName;
    double amount;
};
```

Value type with operator= & copy constructor

25

Exercise