CO517 C++ object oriented design

Department of Computing

Imperial College London

Personal Summary

Author:
Alexander Nederegger

March 22, 2020

Contents

List of Tables

List of Figures

List of Source Codes

1 C++ introduction to classes

The below illustrates the declaration, definition, and use of a basic classes. Also note that classes can do the following ...

- A class can be a member variable of another class.
- A class can be a formal function parameter.
- A function may return an object - i.e. the return value of a function may be an object type.

1.1 Classes basics

Declaration of class goes into separate header file or where globals are declared

```
Class Day_of_year // where Day_of_year is called the type qualifier
{
public:    //<< indictates that methods and objects have no restriction on them
   void output(); //<-- This is a member function (called methods)
   int month;    //<-- These are member variabes
   int day;
}; // don't forget the semicolon
```

Listing 1: Declaration of a class

Definitions of member functions of the class go into a separate file

For the function declaration you need to use the scope resolution operaotr *::*. Note that in member function definitions you do can directly refer to the member variables such as *month* or *day*.

```
void Day_of_year::output()
{
   cout << month;
   cout << day << endl;
}
```

Listing 2: Definition of class member functions

Create an instance of a class and refer to member variables

An instance of a class is like a variable name which has the *class name* as its underlying datatype.

```
int main()
{
   Day_of_year today, birthday //create two instances of a class

   //assign the member variables of the instance
   today.month = 10;
   today.day = 26;

   //call the method on the instance and print it out
   cout << today.output();
}
```

Listing 3: Create an instance of a class and refer to its members

1.2 Make all member variables and as many member functions as possible private

To do that you should write accessor and mutator functions. To ensure that member variables cannot be accessed by functions other than member functions (and friendly functions), use the *private* keyword.

```
Class Day_of_year
{
public:
   void input();
   void output();
   int get_month();
   int get_day();

   void set(int new_month, int new_day);
   // Reset date through arguments of function

private:
   void check_date();
   int month;
   int day;
};
```

Listing 4: Make member variables private and add accessor and mutator functions

Use of the assignment operator for class objects

Using the assignment operator on classes objects will set the member variables of one class equal to that of the other (this also applies for private member variables).

```
Day_of_year today, tomorrow;
today = tomorrow; // is legal

// equivalently you can write ...

today.month = tomorrow.month;
today.day = tomorrow.day;
```

1.3 Programming example: Bank account class

The below is a bank account class with a balance and interest rate. Class has member functions that compounds the balance and prints, prints out the values and sets the member variables to a value.

Interface

```
class BankAccount
{
public:
   //Sets the member variables to the values used as arguments
   void set(int dollars, int cents, double rate);

   //Updates the account balance according to the initial balance and the rate
   void update();

   //Returns the account balance
   double getBalance();

   //Returns the rate
   double getRate();


   void output(ostream& out);

private:
   double balance;
   double iRate;
   double fraction(double percent);
      //converts pct to fraction (e.g. fraction(50.3) returns 0.503)
};
```

Listing 5: Bank account class - interface

Implementation

```cpp
void BankAccount::set(int dollars, int cents, double rate)
{
   if (dollars < 0 || cents < 0 || rate < 0)
   {
      cout << "Illegal";
      return;
   }
   balance = dollars + 0.01 * cents;
   iRate = rate;
}

void BankAccount::update()
{
   balance += balance * fraction(iRate);
}

double BankAccount::fraction(double percent)
{
   return (percent / 100);
}

double BankAccount::getBalance()
{
   return balance;
}

double BankAccount::getRate()
{
   return iRate;
}

void BankAccount::output(ostream& out)
{
   out.setf(ios::fixed);
   out.setf(ios::showpoint);
   out.precision(2);
   out << "Account balance $" << balance;
   out << "Interest rate " << i_rate << "%" <<endl;
}
```

Listing 6: Bank account class - implementation

Use in main function

```cpp
int main()
{
   BankAccount account1, account2; //two instances

   account1.set(123, 99, 3.0); //normally use a constructor for this initialisation
   account1.output(cout); //prints it to the screen
   account1.update(); //adds the interest to the balance

   account2 = account1; //sets all member variables equal
   account2.output(cout);
}
```

Listing 7: Bank account class - used in a main function

1.4 Constructor for initialisation

- Is a member function that is used to initialise some or all member variables of an object at point of object declaration.
- The constructor is automatically called when an object is declared.
- You can overload the constructor (i.e. have more than one constructor).
- It is typical to have one default constructor (with no parameters).
- You can define a constructor like other member functions but note that
  1. Must have same name as class.
  2. Cannot return a value (not even void).
  3. Is in public space of the class definition (otherwise you cannot create an object).

Syntax when using a constructor

The below shows how to declare a constructor in the class definition and then defining the constructor in the function definition file used for the class. This version is not truly recommendable because this version does not truly initialise the member variables but rather assigns them some values. Hence, if you have a const member variable the use of this constructor would be illegal. After the below example you can see how to truly initialise member variables.

```cpp
//In header define the class interface
class BankAccount
{
public:
   //This is the constructor function declaration
   BankAccount(int dollars, int cents, int rate);
private:
   double balance;
   double iRate;
}

//In function definition file add the constructor definition
BankAccount::BankAccount(int dollars, int cents, int rate)
{
   if (dollars < 0 || cents < 0 || rate < 0)
   {
      cout << "Illegal";
      exit(1);
   }
   balance = dollars + 0.01*cents;
   iRate = rate;
}
```

Listing 8: Syntax when using a constructor

Having included the constructor you can now declare an instance using the following ...

```cpp
BankAccount account1(10, 50, 2.0);
BankAccount account2(500, 0, 4.5);
```

Constructor with real initialisation

The below shows two constructor definitions where the member variables are initialised in the initialisation section after the colon.

The initialisation section ...

- Consists of a : followed by a list of some or all member variables separated by commas.
- Each member variable is followed by its initialising value in parentheses.
- The initialising values can be given in terms of the constructor parameters.

```
//In class definition file
class BankAccount
{
public:
   //Default constructor that sets balance and rate to 0
   BankAccount();

   //Another constructor
   BankAccount(int dollars, int cents, double rate);

private:
   double balance;
   double iRate;
};

//In function definition file for the class
BankAccount::BankAccount() : balance(0), iRate(0) {//no need for body}

BankAccount::BankAccount(int dollars, int cents, double rate) :
   balance(dollars + 0.01*cents), iRate(rate)
{
   if (dollars < 0 || cents < 0 || rate < 0)
   {
      cout << "Illegal";
      exit(1);
   }
}
```

Listing 9: Syntax for constructor with initialisation section

Note that the constructor functions could also directly be defined at declaration in the class definition.

Use of constructor when declaring a dynamic variable

Initialisers can also be specified if the object is created as a dynamic variable:

```
BankAccount* my_account = new BankAccount(300, 3.2);
```

Constructor delegation

- Allows one constructor to call another one.
- E.g. set the default constructor to invoke the constructor with two parameters.

```
Coordinate::Coordinate() : Coordinate(99, 99) {}
```

Constructors with default arguments

Like with all other functions you can use default arguments in constructors. The default values for the member variables can, however, also be directly in the class definition when the member variables are declared – see two examples below.

The problem with the second constructor is that *occupancy* may not be initialised.

```
int getCapacityFromRegistry(char* serialNumber); // imagine you have a database somewhere
class Bus
{
   int capacity;
   float occupancy;
   char* serialNumber;
public:
        //Here in construct. declaration we set def. param of occupancy to 0
        Bus(int capacity, float occupancy = 0) : capacity(capacity), occupancy(occupancy){};
      //If you call Bus(4); then capacity is 4 and occupancy is 0
      //If you call Bus(4, 2); then capacity is 4 and occupancy is 2

        //In this constructor definition we set capacity to what we got rom the registry
        Bus(char* serialNumber) : serialNumber(serialNumber)
        {
      capacity = getCapacityFromRegistry(serialNumber);
        };
}
```

Listing 10: Constructor with default arguments 1/2

To make sure all member variables are initialised we can also define default values where the variables are declared.

```
int getCapacityFromRegistry(char* serialNumber); // imagine you have a database somewhere
class Bus
{
   //Set some default values here...
   int capacity = 99;
   float occupancy = 0;
   char* serialNumber = "unregistered";
public:
        Bus(int capacity, float occupancy = 0) : capacity(capacity), occupancy(occupancy){};

        Bus(char* serialNumber) : serialNumber(serialNumber) {};
            //Using this constructor then capacity and occupancy will be initialised
      //at 99 and 0 respectively.
}
```

Listing 11: Constructor with default arguments 2/2

1.5 Member initialisers using default values

- Allows to set default values for member variables.
- When object is created, member variables are automatically initialised to these specified values.

```
class Coordinate
{
public:
   Coordinate(); //default constructor
   Coordinate(int x);
   Coordinate(int x, int y);

```

```cpp
private:
   int x = 1; //Default value
   int y = 2; //Default value
};

Coordinate::Coordinate() {}
Coordinate::Coordinate(int x_val) : x(x_val) {}
```

With the above class if you call Coordinate c1; then x is 1 and y is 2 (the default values). If you call Coordinate c2(10), then x is 1 and y is 10.

1.6 A note on abstract data types

To define a class so that it is an ADT we need to separate the specification of how the type is used from the details of how the type is implemented. The separation should be complete such that you can change the implementation of the class w/o needing to make any changes in any program that uses the class ADT.

Follow these rules:

- Make all member variables private.
- Make the functions the class user needs to use public and sufficiently explain their use.
- Make any helping functions private member functions.
- Separate the interface from the implementation.

1.7 Brief introduction to inheritance

Inheritance means a class that inherits aspects of another (parent class). It allows you to define a general class and then later define more specialised classes that add some new details.

If class A is a derived class of some other class B, then class A has all the features of class B but also has added features. Class A is the child and class B the parent.

Defining a derived class

Add a colon : followed by the keyword *public* and the *name of the parent* class to specify a derived class.

The below is a derived class of the BankAccount class thus it inherits ALL member variables (balance, iRate) and all member functions from the BankAccount class.

```cpp
class SavingsAccount : public BankAccount
{
public:
   //Constructor
   savingsAccount(int dollars, int cents, double rate);

   void deposit(int dollars, int cents);
   void withdraw(int dollars, int cents);
private:
};

//The definition of the SavingsAccount constructor calls the BankAccount constructor
SavingsAccount::SavingsAccount(int dollars, int cents, double rate) :
   BankAccount(dollars, cents, rate) {}
```

Listing 12: Syntax for class inheritance

With an instance such as SavingsAccount account(100, 50, 5.5) you can call functions of the BankAccount class such as account.output().

We can go a step further and derive more specialised classes from the SavingsAccount class.

```cpp
class CDAccount : public SavingsAccount
{
public:
   CDAccount(int dollars, int cents, double rate);
   int getDaysToMaturity();
   void decrementDaysToMaturity();
private:
   int daysToMaturity;
};
```

Listing 13: Second level of class inheritance

The CDAccount class access to all member variables and functions of SavingsAccount and BankAccount.

2 Friend functions and operator overloading

2.1 Friend functions

A friend function of a class is not a member function but still has access to the private member variables of the class – and can also change these values. The concept may be useful at times but should be avoided (and is never required) – it is bad programming style.

To make a function a friend, list the function declaration in the definition of the class (in public section) and placing the keyword *friend* in front of the function declaration. The friend function *definition* does not require the keyword. This function can be used normally outside of the class.

```
//Friend function declaration inside public scope of class definition
public:
   //requires 'friend' keyword
   friend bool equal(Day_of_year today, Day_of_year tomorrow);

//Friend function definition outside class definition (as notmal function usual)
// ! Does not need the qualifier 'Day_of_year::' in function heading.
bool equal(Day_of_year today, Day_of_year tomorrow)
{
   //Function definition goes here
}
```

Listing 14: Syntax for using a friend function

Programming example: Money class, B.p. 662 - 668

The code below is not too hard but very useful exercise for using I/O streams as parameters in functions. See book for details but should be self evident.

Brief note on leading zeros in number constants

- Sometimes in C++ '09' and '9' are not interpreted in the same way.
- For some C++ compilers the use of leading zeros means that the number is written in base 8 rather than base 10.
- Since base 8 numerals do not use the digit 9 the constant 09 does not make sense in C++.
- However, for the GNU compiler it is ok.

2.2 The const parameter modifier

Const is a great safety measure to use if you don't want a function to change the value of a variable. If used in functions, then both the function declaration and function definition require the *const* modifier. *const* tells the compiler that this parameter should not be changed (if changed by a function you get an error).

In classes you should place the const modifier after the function definition and declaration if the functions do not change the member variables (e.g. for output member functions).

Parameters of a class type that are NOT changes by the function ordinarily should be constant call by reference parameters, rather than call-by-value parameters. If a method of a class does not change the value of its calling object, then mark the function by adding the const modifier to the function declaration and function definition. The const is placed at the end of the function declaration just before the final ;. Note that const is an all-or-nothing proposition.

I.e. if you use const for one parameter of a certain type, then you should use it for every other parameter that has the same type and that is not changed by the function call. Use const whenever it is appropriate for a class parameter and method of the class. If you do NOT use const every time it is appropriate for a class you should not use it at all.

```
// CLASS definition
class Sample
{
public:
   void input();
         void output() const;
         //place the const here as you do not wish that output
   //changes the value of its calling object
private:
         int stuff;
};

//FUNCTION definition then also requires 'const'
void Sample::output() const
{
   //some definition here
}
```

Listing 15: Using the const parameter modifier in class member functions

The below explanation is a bid more intricate:
If the parameter type you are using is a class, then you should use const for every method of the class that does not change the value of its calling object. An example for a redefinition of the class Money is given in book p. 675 - 676.

```
void guarantee(const Money& price) //where Money is a class and price is an object of type Money
{
   cout << price.get_value(); // where 'get_value' is a method of the class Money
   //Then in class 'get_value' should be 'const'
   //Even though 'get_value' does NOT change 'price' but the compiler thinks that
   //'get_value' might change price because the comiler will only know the declaration
   //of 'get_value' (not the definition) when calling 'guarantee()'
}
```

Listing 16: Using the const parameter modifier in function parameter where the type is a class

2.3 Overloading operators

An operator such as $+$, $-$, % etc. are just functions with a different syntax. These operators can be overloaded.

2.3.1 Overloading binary operators

You can overload most but not all operators. The operator does not need to be a *friend* of a class, but often you want them to be a friend.

The definition is the same as for member functions but instead you use the operator (e.g. +, -, = etc.) as the function name and precede it with the keyword *operator*.
Example of overloading '+' and '=='. Note that the friend keyword is not required and you could also overload the operators outside the class (but this is less common). A not of warning:

if you want to compare two pointers (using the equality operator '=') you can only do so if the pointers are initialised otherwise you will have a segmentation fault.

```
//CLASS DEFINITION
class Money
{
public:
        //Function that returns type 'Money'
        friend Money operator +(const Money& amount1, const Money& amount2);

        //Function that returns type 'bool'
        friend bool operator ==(const Money& amount1, const Money& amount2);
            //Note that friend keyword is NOT required for overloading the operator
      //Also you could overload the operator outside the class (but less common)

        //Constructors and other stuff go here
private:
        int all_cents;
};

//FUNCTION DEFINITION
Money operator +(const Money& amount1, const Money& amount2)
{
   Money temp;
   temp.all_cents = amount1.all_cents + amount2.all_cents;
   return temp;
}

bool operator ==(const Money& amount1, const Money& amount2)
{
   return (amount1.all_cents == amount2.all_cents);
}

//In MAIN we could write ...
int main()
{
   Money cost(1, 50), tax(0, 15), total;
   total = cost + tax;
   //we could now use '+' instead of having to call an 'add' method
}
```

Listing 17: Overloading the + and == operators

Rules for overloading binary operators:

- When overloading an operator, at least one argument of the resulting overloaded operator must be of a class type.
- An overloaded operator can be, but does not have to be, a friend of the class.
- You cannot create a new operator. All you can do is overload existing operators, such as +. -, *, /, %, and so forth.
- You cannot change the number of arguments that an operator takes. For example you cannot change % from a binary yo a unary operator when you overload % you cannot change ++ from a unary to a binary operator when you overload it.

- You cannot change the precedence of an operator.
- The following operators cannot be overloaded: the dot (.), the scope resolution (::), the (*), and (?) operator.
- Although the assignment operator = can be overloaded so that the default meaning of = is replaced by a new meaning, this must be done in a different way from what is described above. Overloading = is discussed a different section.
- Other operators such as [], -¿ must also be overloaded in a different way.

2.3.2 Constructors for automatic type conversion

!!! Not sure what this is .. check again in Book!!!

If we add an integer to an amount that is of another type (e.g. of the class Money) then we need a constructor in the class that initialises the integer we add to the appropriate type. The system first checks to see if the operator + has been overloaded for the combination of a value of the class type and value of an int type, if this is not the case, then the system next looks to see if there is a constructor that takes a single argument that is an int. It then uses this constructor to convert the integer to a value of the class type.

2.3.3 Overloading unary operators

Unary operators follow the same syntax as discussed above. Unary operators are . . .

- - can be a unary operator when used for negation (e.g. x = -y).
- Increment and decrement operators such as ++ or – (however note that you can only overload ++ and – with the aforementioned syntax if they are used in a *prefix* position but not for a postfix position.

2.3.4 Overloading >> and << operators

These are binary operators and their value returned must be a stream. The type for the value returned must have the & symbol at the end of the type name. The operator <<should return its first agrument, which is a stream of type ofstream.
Motivation is that instead of writing code like first alternative below, write it like the second alternative.

```
//Instead of writing this alternative 1
Money amount(100);
cout << "I have " << amount.output(cout);

//... you want to write
cout << "I have " << amount; //i.e. w/o having to call '.output(cout)'
```

Syntax . . .

```
//Class definition
class Name
{
public:
   friend ifstream& operator >>(ifstream& in, Name& parameter2);
   friend ofstream& operator <<(ofstream& out, const Name& parameter4);
   //use const here because outstream does not change the object values

};

//Overloaded operator definition
ifstream& operator >>(ifstream& in, Name& parameter2)
{
   //do something
   return in;
}

ofstream& operator <<(ofstream& out, const Name& parameter4)
{
   //do something
   return out;
}
```

Listing 18: Overloading the >> and << operators

Explanation of meaning of & in the return type ofstream& ...

- Whenever an operator of a function returns a stream you must add an & to the end of the name of the return type.
- When you add an & to the name of a returned type – you return a reference.
- This means that you are returning the object itself as opposed to the value of the object.
- Since you cannot return the value of a stream (i.e. the entire file) you want to return the stream itself rather than the value of the stream.

An example of overloading >> and << in the context of the Money class is given in book pages 689 to 692.

2.3.5 Overloading the assignment operator '='

Suppose we have the class StringVar (as in the next section) and we declare the two objects string1 and string both of type String_var. Then setting string1 = string2 may not result in the output we desire. More specifically it will set all member values of string1 to the same value as those of string2 - this may lead to problems if e.g. both have member variables that are pointers.

Any class that uses pointers and the *new* operator should overload the assignment operator for use with the class.

- Cannot overload = like we overloaded + or ==.
- When you overload the assignment operator it must be a member of the class (i.e. it CANNOT be simply a friend of a class).

The example below would set one string equal to another through operator overloading.

```
//In class definition
class StringVar
{
public:
   void operator =(const StringVar& rightSide);
};

//Function definition
void StringVar::operator = (StringVar& rightSide)
{
   int newLength = strlen(rightSide.value);
   if (newLength > maxLength)
   {
      delete [] value;
      maxLength = newLength;
      value = new char[maxLength + 1];
   }

   for (int i = 0; i < newLength; i++)
   {
      value[i] = rightSide.value[i];
   }
   value[newLength] = '\0';
}
```

Listing 19: Overloading the assignment operator ‘=’

A note of warning . . .

```
StringVar string1, string2;
string1 = stirng2; //This copies the object, however ...

StringVar* string1, *string2;
string1 = string2; //... this copies the pointer/address
```

2.3.6 Overloading conversions (i.e. casts)

- Conversions of base types are often called casts.
- Conversions of complex datatypes are often design flaws (sometimes we need them).

Motivation

```
struct Company
{
   char* name;
};
int get_phone_number(Company c);

struct Make
{
   char* manufacturer;
   char* model;
};

struct Bus
{
   int capacity;
   Make make;
};

//==============================

int main()
{
    Bus bus1; //get it from some database
    Company comp1;
    comp1.name = bus1.manufacturer.name;
    get_phone_number(comp1);
}
```

To make a one-liner out of the main part that is less error prone we require a 'conversion' in the Make class as follows:

```
struct Make
{
   char* manufacturer;
   char* model;

   //------ CASTNIG HERE ---------
   operator Company()
      {
      Company result;
            result.name = manufacturer;
         return result;
      }
      //-----------------------------
};

int main()
{
      get_phone_number(get_bus_form_database().make);
      //The outer function is then called with an object
      //of make
}
```

Listing 20: Overloading type conversions (i.e. casts)

3 Arrays and classes, destructor, and copy constructor

You can build arrays of structs or classes, or also structs or classes with arrays as member variables.

3.1 Arrays and classes

3.1.1 Arrays of classes

In case you want each indexed variable to contain items of different types - then make the array an array of structures or classes. For example if you want an array to hold 10 weather data points that each hold a data point for wind and velocity:

Note that when an array of classes is declared, the default constructor is called to initialise the indexed variables. Thus it is important to have a default constructor for any class that will be the base type of an array.

```
//Struct definition
struct WindInfo
{
   double velocity;
   char direction;
};

//Declare the array and fill it
WindInfo dataPoint[10];

for (int i = 0; i < 10; i++)
{
   cout << "Enter velocity: ";
   cin >> dataPoint[i].velocity;

   cout << "Enter direction: ";
   cin >> dataPoint[i].direction;
}
```

Listing 21: An array of structs – a short example of wind info

An example of using the class Monet is given in book page 696 to 697 (but it is self evident).

3.1.2 Arrays as class members

You can have a structure or class that has an array as a member variable.

For example, a speed swimmer wants a program to keep track of the practice times for various distances. You can use the structure my_best (the object) (of the type 'Data' (the structure)) to record a distance (member variable that is an integer) and the times (in seconds; member variable that is an array) for each of ten practice tries swimming that distance:

```cpp
//Define struct
struct Data
{
   int distance;
   int time[10];
   //an array that is a member of a struct which records times for one given distance
};

//In main
Data myBest;

myBest.distance = 20;
cout << "Enter then times (in seconds): ";
for (int i = 0; i < 10; i++)
{
   cin >> myBest.time[i];
}
```

Listing 22: An array as a class member variable

If you use a class rather than a structure type, then you can do all your array manipulations with member methods.

3.2 Programming example: A class for a partially filled array (not hard but interesting)

The program stores a list of temperatures in an array, adds them, and checks the size of the array. Interesting features include:

- Use of array in a class.
- Overloading <<as discussed earlier.
- Using const in member function.
- Use of friend function.

The full example is in book page 699 to 701 and extracts are . . .

Class interface

```cpp
class TemperatureList
{
public:
   //Initialises size at 0
   TemperatureList(double temperature);

   /*
   Adds a temperature to the list if the array is not full and increments size
   to know when the array is full.
   */
   void addTemperature(double temperature);

   /*
   Returns true if size is 50 (i.e. if array is full)
   */
   bool full() const;
```

```

    friend ofstream& operator <<(ofstream& out const TemperatureList& theObject);

private:
    double list[50]; //array of temperatures in celsius
    int size; //number of array positions filled --> initialised at 0
};
```

Class implementation

```
TemperatureList::TemperatureList() : size(0) {}

TemperatureList::TemperatureList(double temperature) : size(temperature) {}

void TemperatureList::addTemperature(double temperature)
{
    if (full())
    {
        cout << "Error, list is full!";
        exit(1);
    }

    else
    {
        list[size] = temperature;
        size++;
    }
}

bool TemperatureList::full() const
{
    return (size == 50);
}

ofstream& operator <<(ofstream& out const TemperatureList& theObject)
{
    for (int i = 0; i < theObject.size; i++)
    {
        out << theObject.list[i] << " C" << endl;
        return out;
    }
}
```

3.3 Classes and dynamic arrays

A dynamic array can have a base type that is a class. A class can have a member variable that is a dynamic array. An example of classes and dynamic arrays is given in the programming example at the end of this section (‘A string variable class’).

3.4 Destructor

A destructor is a member function that is called automatically when an object of the class passes out of scope. This means that if your program contains a local variable that is an object with a destructor, then when the function call ends, the destructor is called automatically. If the destructor is defined correctly, the call to delete eliminates all the dynamic variables created by the object.

Destructors are particularly useful if a dynamic variable is embedded in the implementation of a class where the programmer that uses the class does not know about the dynamic variable and cannot be expected to know that a call to delete is required (also normally member variables are 'private' thus he also could not make a call to delete).

Returning memory to the heap is the main job of the destructor.

The name of a constructor is the name of the class prefixed with ' '.

Example of a destructor

```
//Declare DESTRUCTOR in class definition
class StringVar
{
public:
   ~StringVar();
};

//Define DESTRUCTOR
StringVar::~StringVar()
{
   delete [] value;
}
```

Listing 23: Example of destructor

3.5 Copy constructor

Motivation – pitfall of pointers as call-by-value parameters (B.p. 708 - 709

Beware when calling a function with a pointer as parameter. Even when passed as value parameter, the pointer variable is still an address. When passing the address to a function and then change the value pointed to by the pointer (i.e. the address) you effectively also change the value in the main part of your code.

If the parameter type is a class that has member variables of a pointer type you could get surprising results. But for for class types you can avoid (and control) the effects of such by defining copy constructors.

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 parameter.
- Normally the parameter is preceded by const.
- In all other respects a copy-constructor is defined in the same way as other constructors.

The copy constructor is called automatically whenever an argument is 'plugged in' for a call-by-value parameter of the class type. Any class that used pointers and the new operator should have a copy constructor.

```
//Reminder of the copy constructor
//As needed, has a variable of same type as the class passed by reference
String_var::Stirng_var(const String_var& string_object) : max_length(string_object.length())
{
   value = new char[max_length + 1];
   strcpy(value, string_object.value);
}

//Assume code below ...
String_var line(20); //uses first constructor of string class example below
cout << "Enter a string of length 20 or less.";
line.input_line(cin); // assigns the 'line' string the value we have typed in

String_var temporary(line); // Initialized by copy constructor
    //temporary is a new object
    //Since line is of type String_var --> the copy constructor is invoked.
```

Listing 24: Example of copy constructor

So what happens exactly with the copy constructor?

- The object *temporary* is initialised by the constructor that has one argument of type const String_var&.
- The design of the copy constructor should be such that:
  - The object being initialised becomes an independent copy of its argument.
  - (i.e. in example that temporary becomes an independent copy of line)
    - (in this example) this is achieved by giving the object 'temporary' a new independent dynamic array variable.
    - We then copy the contents of 'line' onto 'temporary'.
  - This is called a deep copy where all changes we would make on 'temporary' would not be made on 'line'.

A copy constructor is also called automatically in certain other situations (whenever C++ needs to make a copy of an object it automatically calls the copy-constructor).

1. When a class object is declared and is initialised by another object of the same type (as in example).
2. When a function returns a value of the class type, the copy constructor is called automatically to copy the value specified by the return statement (but only if there is a copy constructor defined).
3. Whenever an argument of the class type is plugged in for a call-by-value parameter.

Book page 710 to 711 further illustrates the need for a copy constructor - if you do not create a deep copy you will end up having two pointers pointing to the same value.

Typically for classes that do not involve pointers or dynamically allocated memory you do not need copy constructors - a shallow copy performed by default will generally suffice.

The big three (copy constructor, the overloaded assignment operator =, and the destructor) are called big three because if you define one of them, you need to define all three.

3.6 Programming example: Own string class (B.p. 702)

Define a class *StringVar* whose objects are string variables, implemented using a dynamic array).

Class definition

```
class StringVar
{
public:
//Four constructors and one destructor
   StringVar();
      //Initialises object so it can accept string values up to '100'
      //and sets the value of the object equal to empty string
   StringVar(int size);
      //Initialises object so it can accept string values up to 'size'
      //and sets the value of the object equal to empty string
   StringVar(const char a[]);
      //Pre-condition: the array 'a' contains characters and is terminated with '\0'
      //Initialises object so its value is the string stored in 'a' and maxLength is
      //the length of 'a'
   StringVar(const StringVar& string_object);
      //Copy constructor (check again)
   ~StringVar();
      //Returns all dynamic memory to the heap

//Other member functions
   int length() const;
      //Returns length of current string value
   void inputLine(ifstream& ins);
      //Precondition: If 'ins' is a file input stream (i.e. not cin), then 'ins' has been
      //connected to a file
      //Action: The next text in in the input stream, up to '\n', is copied to the calling
      //object. If there is not sufficient room, then only as much as fit is copied
   friend ofstream& operator <<(ofstream& outs, const StringVar& theString);
      //Pre-condition: If outs is a file output stream, then outs has
      //already been connected to a file
   void operator =(const StringVar& rightSide);
      //Sets a string variable equal to rightSide

private:
   char* value; //pointer variable / dynamic array that holds the string
   int maxLength; //maximum length of value array
};
```

Listing 25: String class programming example 1/3

Member functions definition

```
#include <cstdlib>
#include <cstddef>
#include <cstring>

//---------- Constructors
StringVar::StringVar() : maxLength(100)
{
   value = new char[maxLength + 1];
   value[0] = '\0';
}

StringVar::StringVar(int size) : maxLength(size)
{
   value = new char[maxLength + 1];
   value[0] = '\0';
}

StringVar::StringVar(const char a[]) : maxLength(strlen(a))
{
   value = new char[maxLength + 1];
   strcpy(value, a);
}

//---------- Copy constructor
StringVar::StringVar(const StringVar& stringObject)
   : maxLength(stringObject.length())
{
   value = new char[maxLength + 1];
   strcpy(value, stringObject.value);
}

//---------- Destructor
StringVar::~StringVar()
{
   delete [] value;
}

//---------- Other member functions
int StringVar::length() const
{
   return strlen(value);
}

void StringVar::inputLine(ifstream& ins)
{
   ins.getline(value, maxLength + 1);
}

ofstream& operator <<(ofstream& outs, const StringVar& theString)
{
   outs << theString.value;
   return outs;
}

void StringVar::operator =(const StringVar& rightSide)
{
   int newLength = strlen(rightSide.value);
   if (newLength > maxLength)
   {
      delete [] value;
      maxLength = newLength;
      value = new char[maxLength + 1];
```

Illustrative program

```
int MAX_NAME_SIZE = 30;
int main()
{
   //Create two string objects
   StringVar yourName(MAX_NAME_SIZE); //use of second constructor
   StringVar ourName("Borg"); //uses third constructor

   cout << "What  is your name? " << endl;
   yourName.inputLine(cin);

   //This only works because we have overloaded the '<<' operator
   cout << "We are:  " << ourName << endl;
   cout << "We will meet again " << yourName << endl;
}
```

Listing 27: String class programming example 3/3

4 Separate compilation and namespaces

4.1 Separate compilation

- Define a class and place the definition and implementation in separate files
  - Definition of class into a header file (called interface) e.g. class_interface.h
  - Put the member function definitions into a cpp file (implementation) e.g. class_impl.cpp
  - Include the header file in the implementation file e.g. #include ”class_interface.h”
- Also put the main file and other functions into different files
  - e.g. functions.cpp
  - e.g. main.cpp
  - Include the class header file (and other header files) where needed (e.g. #include ”class_interface.h”).

Using #ifndef syntax

Use the syntax below when making a header file - ensures it is only read once.

```
#ifndef CLASS_HEADER_H
#define CLASS_HEADER_H

class Class_header{};

#endif
```

Listing 28: #ifndef syntax for header files

4.2 Namespaces

Scope of *using* directive

- The scope of a using directive is the block which it appears (i.e. from the location of the using directive to the end of the block).
- If the using directive is outside of all blocks, then it applies to all of the file that follows the using directive.

Creating and using a namespace

```
//Creation of namespace
namespace name_of_namespace
{
   //some code (e.g. class definition goes here)
}

//Using a namespace
using namespace name_of_namespace;
```

Listing 29: Creating and using a namespace

Qualifying names

Imagine the following situation ...

- You have two namespaces; ns1 and ns2.
- You want to use the function fun1 (defined in ns1) and fun2 (defined in ns2)/
- Complication: both namespaces also define a function myFunction (where no overloading applies).
- Now when 'using' both namespaces we would get conflicting definitions for myFunction.

Hence you need *using declarations* (i.e. that you will only use fun1 and fun2 from the two namespaces and nothing else). For that we use the scope resolution operator ::.

```
using ns1::fun1;
using ns2::fun2;
```

Listing 30: Using declaration with scope resolution operator

This form is often used when specifying a parameter type. For example the function declaration below the parameter inputStream is of type istream (which is defined in the std namespace.

```
int getNumber(std::istream inputStream);
```

A subtle difference between using directives and using declarations

1. A using declaration makes only one name in the namespace available while a using directive makes all the names in a namespace available.
2. A using declaration introduces a name (like cout) into your code so that no other use of the name can be made. However, a using directive only potentially introduces the names in the namespace.

Unnamed namespaces

Used in a scenario where you have:

- A class definition with member function declarations (interface file).
- An implementation file which defines the member functions.
  - But also in implementation file you DECLARE and later DEFINE some other functions that are no member functions.
  - To make these *outside* functions local to the compilation unit such that you can re-use their name again, you need to put them into the unnamed namespace.

Note that a compilation unit is the file you are in and all the text that is included through include directives.

To truly hide helping functions and make them local to the implementation file for a certain class we need to place them in a special namespace called the unnamed namespace. Otherwise we would not be able to define other functions (in another function file that uses the class) that has the names of some of the member functions of the class (which violates the principle of information hiding).

The unnamed namespace can be used to make a name definition local to a compilation unit (i.e. to a file and its included files). Each compilation unit has one unnamed namespace. All the identifiers defined in the unnamed namespace are local to the compilation unit. You place a definition in the unnamed namespace by placing it in a namespace grouping with no name:

```
namespace
{
   //Definition1
   //...
   //DefinitionN
}
```

Listing 31: Unnamed namespace

You can use any name in the unnamed namespace w/o a qualifier in the compilation unit.

Unnamed namespaces - illustrative example

Class interface (dtime.h)

```
#ifndef DTIME_H
#define DTIME_H

#include <iostream>
using namespace std;

//one grouping of the namespace (other is in implementation)
namespace dtimealexander
{
   class DigitalTime
   {
      //Class definition
   };
}

#endif
```

Listing 32: Interface file - unnamed namespace example

Class implementation file

```cpp
#include <iostream>
//some other libraries
#include "dtime.h"

//Grouping for unnamed namespace
namespace
{
   //The function declarations
   int digitToInt(char c);
   void readMinute();
   void readHour();
}

//Namespace of the class
namespace dtimealexander
{
   bool operator ==() {}
   DigitalTime::DigitalTime(){}
   //more member function definitions
}

//Another grouping for the unnamed namespace
//to define the functions we have declared before also in unnamed namespace
namespace
{
   int digitToInt(char c) {} //Definition goes in brackets
   void readMinute() {} //Definition goes in brackets
   void readHour() {} //Definition goes in brackets

   //The functions defined in the unnamed namespace are local
   //to this compilation unit (this file and all included files)
   //They can be used anywhere in this file but have
   //no meaning outside this compilation unit
}
```

Listing 33: Implementation file - unnamed namespace example

Main file

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

void read_hour(int& the_hour);
//New function declaration that has same name as one in the class
//implementation file but it is a different function

int main()
{
   using namespace std;
   using namespace dtimealexander;
   read_hour(23); // call to new function declared in MAIN
}

//Also need to define the new function here.
```

Listing 34: Main file - unnamed namespace example

Confusing the global namespace and the unnamed namespace

Both names in the global- and the unnamed namespace may be accessed w/o a qualifier. However, names in the global namespace have global scope (all the program files), while names in an unnamed namespace are local to a compilation unit.

5 Introduction to memory management

C++ does not, like some other programming languages (Java, Python) collect garbage (i.e. variables that cannot be reached from the stack) and does not automatically delete them. Hence, whenever dynamic memory (heap memory) is created these need to be deleted at some point.

The problem with automatic garbage collection (GC) is that it is tricky to get fast, given it is single threaded. GC might trigger at any moment which is bad for predictable runtime and predictable memory consumption.

Having said that, there are libraries for automatic GC in C++.

You can find out if you have memory leaks using *valgrind*.

5.1 Common memory management errors

Stack (local variables) memory errors

- Reading/writing to memory out of the bounds of a static array.
- Function pointer corruption: Invalid passing of function pointer and thus a bad call to a function (accessing a reference to a dead variable).

Heap memory errors

- Memory allocation error (malloc() & new; can return NULL).
- Attempting to free memory that was already freed or that was never allocated.
- Attempting to read/write memory already freed or that was never allocated (e.g. because an object is in automatic memory).
- Reading/writing to memory out of the bounds of a dynamically allocated array (leaking memory).

5.2 Simple examples where dynamic memory is not properly freed (w/o classes)

Mistake - try to access an already freed memory address

The below will not compile because you try to access an already freed address.

```cpp
void main()
{
   //Automatic memory
   Date tomorrow(6, 10, 2016);

   //Dynamic memory
   Date* today = new Date(5, 10, 2016);
   delete today; //Dynamic memory is cleaned here

   //---> Mistake: Try to access an already freed address
   today.day = 8;
}
```

Mistake - try to free a variable that in the incorrect scope

In the below when we try to free alsoToday, we cannot access the address because we are not in the right scope (i.e. it should have been deleted earlier). This code does note compile

```
void main()
{
   Date* today(6, 10, 2016);

   {  //This creates a syntactic scope
      Date* alsoToday = new Date(5, 10, 2016);
      today = alsoToday;
      //Note that after the above line, today will not be deletable, hence
      before assigning you need to delete
      //After reassigning today (after it was deleted) you can then delete
      //it again in the outer scope .. alsoToday would never have
      //to be deleted.
   }
   //---> Mistake: alsoToday is NOT visible in this scope
   delete alsoToday;

   delete today;
}
```

Mistake - try to free a variable that in the incorrect scope

Same problem as before but this code will compile but leak.

```
void main()
{
   Date* today(6, 10, 2016);

   {  //This creates a syntactic scope
      Date* alsoToday = new Date(5, 10, 2016);
      today = alsoToday;
      //Note that after the above line, today will not be deletable, hence
      before assigning you need to delete
      //After reassigning today (after it was deleted) you can then delete
      //it again in the outer scope .. alsoToday would never have
      //to be deleted.
   }
   //---> Mistake: Also today is not visible here hence you cannot access today
   //as well
   delete today;
}
```

Mistake - double free

Same code principle as before but trying to free the same address twice. This code will not compile.

```
void main()
{
   Date* today(6, 10, 2016);

   {  //This creates a syntactic scope
```

```
      Date* alsoToday = new Date(5, 10, 2016);
      today = alsoToday;
      delete alsoToday; //alsoToday refers to today
   }
   //---> Mistake: Also today has already been deleted and now you try again
   delete today;
}
```

Correct way

You need to delete today before reassigning it and then delete today again in the outer scope – alsoToday never needs to be deleted.

```
void main()
{
   Date* today(6, 10, 2016);
   {
      Date* alsoToday = new Date(5, 10, 2016);
      delete today; //needs to be deleted here before reassining
      today = alsoToday; //Now the ownership of alsoToday is transferred to
      //today (hence alsoToday does not need to be deleted)...
   }
   delete today; //... however, today needs to be deleted here again
}
```

5.3 Who is the owner of the dynamic memory

Ownership is conceptual – any pointer can be owning or not-owning, variable or member variable. Conventionally we say that references do not own.
Stack memory is easy: the function retains ownership.
Local variables are easy: upon exit ownership is decided (deleted or transferred).
Class member variables are tricky . . .

- Some member functions transfer ownership.
- Some member functions need to delete the objects.
- Some don't do anything.

5.4 Destructor based memory de-allocation

For the below examples we use the following class and function definitions.

```
int capacity = 99;

struct Passenger
{
   char const* name;
   //Constructor definition
   Passenger(char const* name) : name(name){}
};

struct Bus
{
   Passenger* occupants[capacity + 1]{};
   //Is an array of pointers to passenger instances called occupants
```

```
    //+1 because last is NULL (marks end of array)

    int findPassengerOrNullTerminator(Passenger* p)
    {
        int i = 0;
        while (occupants[i] != nullptr && occupants[i] != p)
        {
            i++;
        }
        return i;
    }

    void havePassengerEnter(Passenger* p)
    {
        int i = findPassengerOrNullTerminator(p);
        if (i < capacity)
        {
            occupants[i] = p;
        }
    }


    void havePassengerLeave(Passenger* p)
    {
        int i = findPassengerOrNullTerminator(p);

        //Shift everyone from found passenger left by one
        while (i < 99)
        {
            occupants[i] = occupants[i + 1];
            i++;
        }
    }
};
```

Implementation example 1 - code is dangerous

```
Bus* getFullBus()
{
    //Two dynamic busses
    Bus* b1 = new Bus;
    Bus* b2 = new Bus;

    //Three dynamic Passengers
    Passenger* holger = new Passenger("holger");
    Passenger* will = new Passenger("will");
    Passenger* fidelis = new Passenger("fidelis");

    //Let people enter the busses
    b1->havePassengerEnter(holger);
    b2->havePassengerEnter(will);
    b2->havePassengerEnter(fidelis);

    //Let Fidelis leave (no longer is in occupants array of the b2
    //but the dynamic variable fidelis is still in stack frame
    b2->havePassengerLeave(fidelis);

```

```
   //Free up memory
   delete b2; //deletes Bus instance (is it a problem that will is still in b2?)
   delete will;
   delete fidelis;

   //Return instance b1 (which has now ownership of 'holger').
   //To ensure objects owned by other objects are deleted - use destructors.
   return b1;
}
```

Enhance example with a destructor in the bus class

Not that a destructor is called when the instance of the class goes out of scope ore when you specifically delete the instance (if the instance was dynamic memory).

If you have an instance (b1) that is itself dynamic memory that itself holds variables of other instances which are also dynamic memory, then the destructor of b1 needs to delete the lower level instances (e.g. Passengers below) – see example below.

You do not need to explicitly delete b1 because if it goes out of scope the variables will be deleted anyway.

```
class Passenger;
class Bus
{
   Passenger* occupants[capacity + 1]{};
   ~Bus()
   {
      for (int i = 0; occupants[i] != nullptr; i++)
      {
         delete occupants[i];
      }
   }
}
//Now if you call delete an instance of Bus (e.g. b1) then the lifetime of
//the object ends and thus the constructor is called whereby the (as per code
//above) all instances of Passenger are that are in the occupants array are deleted.
```

Implementation example 2 - code is fine

At the end of the main scope b1 and b2 are out of scope and die. Because we have a good destructor definition, b1 and b2 take the passengers with them.

```
int main()
{
   //Two dynamic busses
   Bus* b1 = new Bus;
   Bus* b2 = new Bus;

   //Three dynamic Passengers
   Passenger* holger = new Passenger("holger");
   Passenger* will = new Passenger("will");
   Passenger* fidelis = new Passenger("fidelis");

   //Let people enter the busses
   b1->havePassengerEnter(holger);
   b2->havePassengerEnter(will);
```

```
    b2->havePassengerEnter(fidelis);

    //Let Fidelis leave (no longer is in occupants array of the b2)
    //but the dynamic variable fidelis is still in stack frame
    b2->havePassengerLeave(fidelis);
    b1->havePassengerEnter(fidelis);

    return 0;
} //no memory leaks: b1, b2 die here and take their passengers with them
```

Implementation example 3 - code is does not compile

Be careful when returning an address by value and using destructor based deallocation. Note that here the busses are not stored in heap (but in stackframe of the current function) hence you cannot really return their address. Normally this code does not even compile – but dependent on your compiler it actually may.

```
Bus getFullBus()
{
    //Two static buss variables
    Bus b1;
    Bus b2;

    //Three dynamic Passengers
    Passenger* holger = new Passenger("holger");
    Passenger* will = new Passenger("will");
    Passenger* fidelis = new Passenger("fidelis");

    //Let people enter the busses
    b1.havePassengerEnter(holger);
    b2.havePassengerEnter(will);
    b2.havePassengerEnter(fidelis);

    b2.havePassengerLeave(fidelis);
    b1.havePassengerEnter(fidelis);

    return b2;
    //b1 dies with fidelis and holger on board (this is fine)
    //b2 is copied - however the destructor is called which deletes will
    //the copy is returned with an invalid pointer to deleted will - MISTAKE

}
```

Implementation example 4 - code leaks

The function below lets fidelis leave but the question is who owns fidelis thereafter? The answer is that no-one owns fidelis.

```
void simulateBusses()
{
    //Two static buss variables
    Bus b1;
    Bus b2;

    //Three dynamic Passengers
    Passenger* holger = new Passenger("holger");
    Passenger* will = new Passenger("will");
    Passenger* fidelis = new Passenger("fidelis");

    //Let people enter the busses
    b1.havePassengerEnter(holger);
    b2.havePassengerEnter(will);
    b2.havePassengerEnter(fidelis);

    //---> LEAK: Let fidelis leave with a function
    b2.havePassenger Leave(fidelis);
    //If this function was executed then fidelis is not owned and lost
}
```

Implementation example 5 - code is dangerous

The function below may lead to problems because fidelis is in two busses at the same time. Hence it may lead to an incorrect double freeing (fidelis is still only one address).

```
void simulateBusses()
{
    //Two static buss variables
    Bus b1;
    Bus b2;

    //Three dynamic Passengers
    Passenger* holger = new Passenger("holger");
    Passenger* will = new Passenger("will");
    Passenger* fidelis = new Passenger("fidelis");

    //Let people enter the busses
    b1.havePassengerEnter(holger);
    b2.havePassengerEnter(will);
    b2.havePassengerEnter(fidelis);

    //---> Dangerous: after line below fidelis is in two busses
    b1.havePassengerEnter(fidelis);
}
```

Implementation example 6 - code is good

The solution to the aforementioned problem is to make explicit ownership transfer (i.e. let fidelis transfer properly to b1).

```
void havePassengerTransfer(Passenger* p, Bus& from, Bus& to)
{
    from.havePassengerLeave(p);
    to.havePassengerEnter(p);
}

void simulateBusses()
{
    //Two static buss variables
    Bus b1;
    Bus b2;

    //Three dynamic Passengers
    Passenger* holger = new Passenger("holger");
    Passenger* will = new Passenger("will");
    Passenger* fidelis = new Passenger("fidelis");

    //Let people enter the busses
    b1.havePassengerEnter(holger);
    b2.havePassengerEnter(will);
    b2.havePassengerEnter(fidelis);

    //---> Explicit ownership transfer (correct)
    //Ownership of fidelis is transferred from b2 to b1
    havePassengerTransfer(fidelis, b1, b2);
}
```

6 Inheritance

Inheritance involves a class being derived from another class and thus inherits all the member variables and member functions of the parent class. You can have multiple levels of class derivation (ancestor classes).

Note that private member variables of an ancestor class can only be accessed by the derived class through mutator and accessor functions that are publicly (or protectedly) defined in the ancestor class level.

Hence, private member functions are effectively *not inherited*. This is why private member functions in a class should just be used as helping functions, and thus their use limited to the class in which they are defined.

You can redefine a member function in the derived class if you wish to do so. This may be the case if you only want

6.1 Basic syntax of inheritance

6.2 Constructors in derived classes

6.3 Do not use private member variables form the base class

6.4 Use of *protected* qualifier (private member functions are effectively not inherited)

6.5 Redefinition of an inherited funciton

7 Exception handling

8 Templates

9 Standard template library and C++ 11