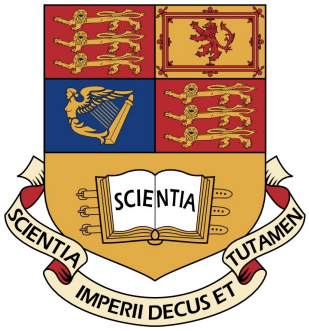
CO517 C++ object oriented design

Department of Computing



Imperial College
London

Personal Summary

Author:
Alexander Nederegger

April 8, 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

```
1 Class Day_of_year // where Day_of_year is called the type qualifier
2 {
3 public:    //<< indictates that methods and objects have no restriction on them
4    void output(); //<-- This is a member function (called methods)
5    int month;    //<-- These are member variabes
6    int day;
7 }; // 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*.

```
1 void Day_of_year::output()
2 {
3    cout << month;
4    cout << day << endl;
5 }
```

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

```
1  class BankAccount
2  {
3  public:
4     //Sets the member variables to the values used as arguments
5     void set(int dollars, int cents, double rate);
6
7     //Updates the account balance according to the initial balance and the rate
8     void update();
9
10    //Returns the account balance
11    double getBalance();
12
13    //Returns the rate
14    double getRate();
15
16
17    void output(ostream& out);
18
19 private:
20    double balance;
21    double iRate;
22    double fraction(double percent);
23       //converts pct to fraction (e.g. fraction(50.3) returns 0.503)
24 };
```

Listing 5: Bank account class - interface

Implementation

```
1  void BankAccount::set(int dollars, int cents, double rate)
2  {
3     if (dollars < 0 || cents < 0 || rate < 0)
4     {
5        cout << "Illegal";
6        return;
7     }
8     balance = dollars + 0.01 * cents;
9     iRate = rate;
10 }
11
12 void BankAccount::update()
13 {
14    balance += balance * fraction(iRate);
15 }
16
17 double BankAccount::fraction(double percent)
18 {
19    return (percent / 100);
20 }
21
22 double BankAccount::getBalance()
23 {
24    return balance;
25 }
26
27 double BankAccount::getRate()
28 {
29    return iRate;
30 }
31
32 void BankAccount::output(ostream& out)
33 {
34    out.setf(ios::fixed);
35    out.setf(ios::showpoint);
36    out.precision(2);
37    out << "Account balance $" << balance;
38    out << "Interest rate " << i_rate << "%" <<endl;
39 }
```

Listing 6: Bank account class - implementation

Use in main function

```
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.

```
//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 ...

```
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);

```

```
8  private:
9     int x = 1; //Default value
10    int y = 2; //Default value
11 };
12
13 Coordinate::Coordinate() {}
14 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.

```
1  class SavingsAccount : public BankAccount
2  {
3  public:
4     //Constructor
5     savingsAccount(int dollars, int cents, double rate);
6
7     void deposit(int dollars, int cents);
8     void withdraw(int dollars, int cents);
9  private:
10 };
11
12 //The definition of the SavingsAccount constructor calls the BankAccount constructor
13 SavingsAccount::SavingsAccount(int dollars, int cents, double rate) :
14    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.

```
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.

```
1  //Friend function declaration inside public scope of class definition
2  public:
3     //requires 'friend' keyword
4     friend bool equal(Day_of_year today, Day_of_year tomorrow);
5
6  //Friend function definition outside class definition (as notmal function usual)
7  // ! Does not need the qualifier 'Day_of_year::' in function heading.
8  bool equal(Day_of_year today, Day_of_year tomorrow)
9  {
10    //Function definition goes here
11 }
```

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.

```
1  //CLASS DEFINITION
2  class Money
3  {
4  public:
5          //Function that returns type 'Money'
6          friend Money operator +(const Money& amount1, const Money& amount2);
7
8          //Function that returns type 'bool'
9          friend bool operator ==(const Money& amount1, const Money& amount2);
10             //Note that friend keyword is NOT required for overloading the operator
11       //Also you could overload the operator outside the class (but less common)
12
13         //Constructors and other stuff go here
14 private:
15         int all_cents;
16 };
17
18 //FUNCTION DEFINITION
19 Money operator +(const Money& amount1, const Money& amount2)
20 {
21    Money temp;
22    temp.all_cents = amount1.all_cents + amount2.all_cents;
23    return temp;
24 }
25
26 bool operator ==(const Money& amount1, const Money& amount2)
27 {
28    return (amount1.all_cents == amount2.all_cents);
29 }
30
31 //In MAIN we could write ...
32 int main()
33 {
34    Money cost(1, 50), tax(0, 15), total;
35    total = cost + tax;
36    //we could now use '+' instead of having to call an 'add' method
37 }
```

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.

```
1 //Instead of writing this alternative 1
2 Money amount(100);
3 cout << "I have " << amount.output(cout);
4
5 //... you want to write
6 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.

```
1  //Struct definition
2  struct WindInfo
3  {
4     double velocity;
5     char direction;
6  };
7
8  //Declare the array and fill it
9  WindInfo dataPoint[10];
10
11 for (int i = 0; i < 10; i++)
12 {
13    cout << "Enter velocity: ";
14    cin >> dataPoint[i].velocity;
15
16    cout << "Enter direction: ";
17    cin >> dataPoint[i].direction;
18 }
```

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:

```
//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

```
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;
```

```
17
18    friend ofstream& operator <<(ofstream& out const TemperatureList& theObject);
19
20 private:
21    double list[50]; //array of temperatures in celsius
22    int size; //number of array positions filled --> initialised at 0
23 };
```

Class implementation

```
1  TemperatureList::TemperatureList() : size(0) {}
2
3  TemperatureList::TemperatureList(double temperature) : size(temperature) {}
4
5  void TemperatureList::addTemperature(double temperature)
6  {
7     if (full())
8     {
9        cout << "Error, list is full!";
10       exit(1);
11    }
12
13    else
14    {
15       list[size] = temperature;
16       size++;
17    }
18 }
19
20 bool TemperatureList::full() const
21 {
22    return (size == 50);
23 }
24
25 ofstream& operator <<(ofstream& out const TemperatureList& theObject)
26 {
27    for (int i = 0; i < theObject.size; i++)
28    {
29       out << theObject.list[i] << " C" << endl;
30       return out;
31    }
32 }
```

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

```
1  //Declare DESTRUCTOR in class definition
2  class StringVar
3  {
4  public:
5     ~StringVar();
6  };
7
8  //Define DESTRUCTOR
9  StringVar::~StringVar()
10 {
11    delete [] value;
12 }
```

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 ::.

```
1 using ns1::fun1;
2 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.

```
1 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.

```
1  void main()
2  {
3     //Automatic memory
4     Date tomorrow(6, 10, 2016);
5
6     //Dynamic memory
7     Date* today = new Date(5, 10, 2016);
8     delete today; //Dynamic memory is cleaned here
9
10    //---> Mistake: Try to access an already freed address
11    today.day = 8;
12 }
```

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
```

```
6        Date* alsoToday = new Date(5, 10, 2016);
7        today = alsoToday;
8        delete alsoToday; //alsoToday refers to today
9     }
10    //---> Mistake: Also today has already been deleted and now you try again
11    delete today;
12 }
```

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.

```
1  void main()
2  {
3     Date* today(6, 10, 2016);
4     {
5        Date* alsoToday = new Date(5, 10, 2016);
6        delete today; //needs to be deleted here before reassining
7        today = alsoToday; //Now the owmership of alsoToday is transferred to
8        //today (hence alsoToday does not need to be deleted)...
9     }
10    delete today; //... however, today needs to be deleted here again
11 }
```

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.

```
1  int capacity = 99;
2
3  struct Passenger
4  {
5     char const* name;
6     //Constructor definition
7     Passenger(char const* name) : name(name){}
8  };
9
10 struct Bus
11 {
12    Passenger* occupants[capacity + 1]{};
13    //Is an array of pointers to passenger instances called occupants
```

```
14      //+1 because last is NULL (marks end of array)
15
16      int findPassengerOrNullTerminator(Passenger* p)
17      {
18          int i = 0;
19          while (occupants[i] != nullptr && occupants[i] != p)
20          {
21              i++;
22          }
23          return i;
24      }
25
26      void havePassengerEnter(Passenger* p)
27      {
28          int i = findPassengerOrNullTerminator(p);
29          if (i < capacity)
30          {
31              occupants[i] = p;
32          }
33      }
34
35
36      void havePassengerLeave(Passenger* p)
37      {
38          int i = findPassengerOrNullTerminator(p);
39
40          //Shift everyone from found passenger left by one
41          while (i < 99)
42          {
43              occupants[i] = occupants[i + 1];
44              i++;
45          }
46      }
47  };
```

Implementation example 1 - code is dangerous

```
1   Bus* getFullBus()
2   {
3       //Two dynamic busses
4       Bus* b1 = new Bus;
5       Bus* b2 = new Bus;
6
7       //Three dynamic Passengers
8       Passenger* holger = new Passenger("holger");
9       Passenger* will = new Passenger("will");
10      Passenger* fidelis = new Passenger("fidelis");
11
12      //Let people enter the busses
13      b1->havePassengerEnter(holger);
14      b2->havePassengerEnter(will);
15      b2->havePassengerEnter(fidelis);
16
17      //Let Fidelis leave (no longer is in occupants array of the b2
18      //but the dynamic variable fidelis is still in stack frame
19      b2->havePassengerLeave(fidelis);
20
```

```
21     //Free up memory
22     delete b2; //deletes Bus instance (is it a problem that will is still in b2?)
23     delete will;
24     delete fidelis;
25
26     //Return instance b1 (which has now ownership of 'holger').
27     //To ensure objects owned by other objects are deleted - use destructors.
28     return b1;
29 }
```

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.

```
1  class Passenger;
2  class Bus
3  {
4     Passenger* occupants[capacity + 1]{};
5     ~Bus()
6     {
7        for (int i = 0; occupants[i] != nullptr; i++)
8        {
9           delete occupants[i];
10       }
11    }
12 }
13 //Now if you call delete an instance of Bus (e.g. b1) then the lifetime of
14 //the object ends and thus the constructor is called whereby the (as per code
15 //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.

```
1  int main()
2  {
3     //Two dynamic busses
4     Bus* b1 = new Bus;
5     Bus* b2 = new Bus;
6
7     //Three dynamic Passengers
8     Passenger* holger = new Passenger("holger");
9     Passenger* will = new Passenger("will");
10    Passenger* fidelis = new Passenger("fidelis");
11
12    //Let people enter the busses
13    b1->havePassengerEnter(holger);
14    b2->havePassengerEnter(will);
```

```
15    b2->havePassengerEnter(fidelis);
16
17    //Let Fidelis leave (no longer is in occupants array of the b2)
18    //but the dynamic variable fidelis is still in stack frame
19    b2->havePassengerLeave(fidelis);
20    b1->havePassengerEnter(fidelis);
21
22    return 0;
23 } //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.

```
1  Bus getFullBus()
2  {
3     //Two static buss variables
4     Bus b1;
5     Bus b2;
6
7     //Three dynamic Passengers
8     Passenger* holger = new Passenger("holger");
9     Passenger* will = new Passenger("will");
10    Passenger* fidelis = new Passenger("fidelis");
11
12    //Let people enter the busses
13    b1.havePassengerEnter(holger);
14    b2.havePassengerEnter(will);
15    b2.havePassengerEnter(fidelis);
16
17    b2.havePassengerLeave(fidelis);
18    b1.havePassengerEnter(fidelis);
19
20    return b2;
21    //b1 dies with fidelis and holger on board (this is fine)
22    //b2 is copied - however the destructor is called which deletes will
23    //the copy is returned with an invalid pointer to deleted will - MISTAKE
24
25 }
```

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.

```
1  void simulateBusses()
2  {
3     //Two static buss variables
4     Bus b1;
5     Bus b2;
6
7     //Three dynamic Passengers
```

```
8     Passenger* holger = new Passenger("holger");
9     Passenger* will = new Passenger("will");
10    Passenger* fidelis = new Passenger("fidelis");
11
12    //Let people enter the busses
13    b1.havePassengerEnter(holger);
14    b2.havePassengerEnter(will);
15    b2.havePassengerEnter(fidelis);
16
17    //---> LEAK: Let fidelis leave with a function
18    b2.havePassenger Leave(fidelis);
19    //If this function was executed then fidelis is not owned and lost
20 }
```

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).

```
1  void simulateBusses()
2  {
3     //Two static buss variables
4     Bus b1;
5     Bus b2;
6
7     //Three dynamic Passengers
8     Passenger* holger = new Passenger("holger");
9     Passenger* will = new Passenger("will");
10    Passenger* fidelis = new Passenger("fidelis");
11
12    //Let people enter the busses
13    b1.havePassengerEnter(holger);
14    b2.havePassengerEnter(will);
15    b2.havePassengerEnter(fidelis);
16
17    //---> Dangerous: after line below fidelis is in two busses
18    b1.havePassengerEnter(fidelis);
19 }
```

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).

```
1  void havePassengerTransfer(Passenger* p, Bus& from, Bus& to)
2  {
3     from.havePassengerLeave(p);
4     to.havePassengerEnter(p);
5  }
6
7  void simulateBusses()
8  {
9     //Two static buss variables
10    Bus b1;
11    Bus b2;
```

```
12
13      //Three dynamic Passengers
14      Passenger* holger = new Passenger("holger");
15      Passenger* will = new Passenger("will");
16      Passenger* fidelis = new Passenger("fidelis");
17
18      //Let people enter the busses
19      b1.havePassengerEnter(holger);
20      b2.havePassengerEnter(will);
21      b2.havePassengerEnter(fidelis);
22
23      //---> Explicit ownership transfer (correct)
24      //Ownership of fidelis is transferred from b2 to b1
25      havePassengerTransfer(fidelis, b1, b2);
26  }
```

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). Also, a class can be derived from two parent classes, i.e. would then inherit the member variables and functions from both parent classes. An example could be a class *Person* with two derived classes *Faculty* and *Student*. Finally a class *TeachingAssistent* which would be a derived class from Student and Faculty. Hence, the class TA would inherit all member variables and functions from Person, Student, and Faculty.

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

What is also important to note is that every object of the child class can be used anywhere an object of the parent class can be used. For example you can use an argument of type *Child* when a function requires an argument of type *Parent*. You can assign an object of the class *Child* to a variable of type *Parent* but NOT vice versa.

More generally, an object of a class can be used anywhere that an object of any of its ancestor classes can be used.

6.1 Basic syntax of inheritance

When using inheritance you will, ideally split the program across the following files:

- An interface file (class definition) for the parent class (header file).
- An implementation file for the parent class (ccp file) that *includes* the header file of this class.
- An interface file for *each* derived class that *includes* the header file of its direct parent(s).
- An implementation file for *each* derived class that *includes* only its own header file (because the parent headers are already included in the header file of the derived class).

The syntax is as follows ...

```
//For all the below use the identical namespace definition

//--- 1) Define the parent class in a separate file
#ifndef PARENT_H
#define PARENT_H
namespace parentnederegger
{
   class Parent
   {
      // 1) All member variables should be private
      //     and write accessor and mutator functions.

      // 2) All private member functions are essentially
      //     not inherited, hence only make them private if they
      //     are only helper functions and only used within the class.

      // 3) Note that you should declare and define your desired
      //      constructors. These should initialise ALL member variables.
      //      When defining the constructors of the child class, the child it
      //      should call the constructor of the parent class to initialise the
      //      inherited member variables and then also initialise the new member
      //      member variables of the child class.

      int aFunction();
   };
} //parentnederegger
#endif


//--- 2) Implement the parent class in a separate file
#include "parent.h"

namespace parentnederegger
{
   Parent::Parent() //some definitions of constructor
   //More function definitions
} //parentnederegger
```

Listing 35: Basic syntax for inheritance 1/2

```
//--- 3) Define the child class in a separate file
//         - Note the public keyword with the name of the parent clas
#ifndef CHILD_H
#define CHILD_H

#include "parent.h"
namespace parentnederegger
{
   class Child : public Parent
   {
      //NOTE: if 'Child' was a child class also of 'Parent2' then you
      //would declare it with: 'class Child : public Parent, Parent2'

      //some class definition
      int aFunction(); //This would be a redefinition of the function
   };
} //parentnederegger

#endif



//--- 4) Implement the child class (standard as any other class) in separate file
//         - NOTE: Speciality regarding the constructor (see later)
#include "child.h"
namespace parentnederegger
{
   Child::Child() : //some definition
   //Other function definitions
} //parentnederegger
```

Listing 36: Basic syntax for inheritance 2/2

Assume that the parent class has the member variable *int age* and the child class has the member variable *char* name*. Also assume you create an instance of the Child class called child1. You can now access both variables from this instance.

```
Child child1;

//If the member variables were public you can access them...
child.age;
child.name;
//However this should ALWAYS be private hence you would access them
//using accessor functions

child.getAge(); //getAge should be a public member function of the Parent
child.getName(); //getName should be public member function of the Child
```

6.2 Constructors in derived classes

A constructor in a base class is not inherited in the derived class, but you can (and should) invoke a constructor of a the base class within the definition of a derived class constructor.

A constructor for the base class initialise all the data inherited from the base class. Thus, a constructor for a derived class begins with an invocation of a constructor for the base class.

You should ALWAYS include an invocation of one of the base class constructors in the initialisation section of a derived class constructor. If you do not include one, then the default constructor version of the base class constructor will be invoked automatically (but preferably you should explicitly state it for clarity).

If you have three classes, A, B, and C. Where B is a descendant of A and C is a descendant of B. Then, if you declare an instance of C then the constructor of A is called first, then a constructor for B is called, and finally the remaining actions of the C constructor are taken.

Example of constructor syntax

Assume you have a parent class called *Employee*, with two member variables name, number, and a child class called *HourlyEmployee* with two more member variables called hourlyRate and hours.

Note that you initialise the inherited member variables by calling the constructor of the parent class first in the initialisation section. After that you initialise the new member variables (still also in the initialisation section).

```
1 HourlyEmployee::HourlyEmployee(string name, int number, int rate, int theHours)
2    : Employee(name, number), hourlyRate(rate),  hours(theHours)
3 {
4    //deliberately left blank as all variables are initialised in the initialisation section
5    //The inherited member variables are initialised by calling the constructor of
6    //the parent class, and the new member variables are declared thereafter.
7 }
```

Listing 37: Constructor in inherited class

6.3 Private member variables and functions

Write accessor and mutator functions for private member variables of parent

Note that you cannot access the private member variables of the parent class by name - they are also private for descendent/child classes. You need to write public accessor and mutator functions in the parent class to access these variables.

Private member functions are effectively not inherited

A private member function of a parent is not accessible from the child. Hence, they should only be used as helping functions.

The *protected* qualifier

If you use the *protected* qualifier instead of public/private in the class definition for member variables of member functions these variables are then accessible/changeable by name from child classes / descendent classes - however, not by any other classes.

Also note, that member variables that are protected in the base class act as though they were also marked protected in any derived class.

It is, however, bad programming style to use protected member variables.

6.4 Redefinition of an inherited function

If a derived class requires a different implementation for an inherited member function, the function may be redefined in the derived class. When a member function is redefined, you must list its declaration in the definition of the derived class. If you do not wish to redefine a member

function that is inherited from the base class, then it is not listed in the definition of the derived class.

Note that if a function has the same name in a derived class as in the base class but has a different signature, then this is overloading and not redefining.

Access the base function of a redefined function

Note that if you have redefined a function (e.g. myFunction()), and you wish to access its parent definition on an object of the child class. If the parent class is called Parent and the child class is called Child then you can access the definition of the function in the base class on a child object using: *childInstance.Parent::myfunction()*.

6.5 Inheritance details

Note that the following functions are not inherited if they are defined in the base class:

- Constructors
- Destructors
- Copy-constructors
- An overloaded assignment operator '='

Overloaded assignment operator '='

If Child is a class derived from Paremt, then the definition of the overloaded assignment operator for the class Child would typically begin with something like the following:

```
1 Child& Child::operator =(const Child& rightSide)
2 {
3    //First call to parent overloaded assignment operator
4    Parent::operator =(rightSide);
5
6    //Here you set new member variables of the Child class
7 }
```

Listing 38: Overloaded assignment operator in derived class

Hence you first have a call to the overloaded assignment operator of the Parent class which takes care of the inherited member variables and their data. The definition of the overloaded assignment operator would then go on to set new member variables that were introduced in the definition of the class Child.

Copy-constructor

A similar situation holds for the copy-constructor. Use the copy-constructor of the Parent class to set up the inherited member variables. Then the new member variables of the Child class are initialised thereafter.

```
1 Child::Child(const Child& object)
2    : Base(object), <more initialisation here>
3 {}
```

Listing 39: Copy constructor in derived class

Note that on the above, 'object' is of type Child and thus also of type Parent. Hence 'object' is a legal argument to the copy constructor for the Parent class.

Destructor in derived class

When the destructor for the derived class is invoked, it automatically invokes the destructor of the base class. The derived class destructor therefore need only worry about using delete on the member variables (and any data they point to) that are added in the derived class. It is the job of the base class destructor to invoke delete on the inherited member variables.

If class B is derived from class A and class C is derived from class B, then when an object of the class C goes out of scope, first the destructor for the class C is called, then the destructor for class B is called, and finally the destructor for class A is called. Note that the order in which destructors are called is the reverse of the order in which constructors are called.

6.6 Polymorphism and virtual functions

Polymorphism refers to the ability to associate multiple meanings to one function name by means of the mechanism of late binding. Late binding is a technique of waiting until run-time to determine the implementation of a procedure.

Late binding

A virtual function is one that may be used before it is defined. When you make a function virtual, you are telling the compiler that you do not know how this function is implemented and to wait until it is used in a program, and then get the implementation from the object instance.

If you redefine a virtual function in a child class we call this overriding and not redefining.

The problem of making a function virtual is that it decreased efficiency of the program - which is why we do not make every function virtual.

In essence you require a virtual function if you have ...

- A Parent and Child class.
- A function, F, of Parent is used by another function, M, of the class.
- The function F is redefined by the child.
- M is NOT redefined but you use the inherited function definition.
- However, M always requires the definition of F of the current class.

To make a function virtual:

- Use the reserved word *virtual* to the function declaration.
- Do not use *virtual* in the function definition.

This is illustrated in the example below.

Illustrative example of virtual function

Assume a record-keeping program for an automobile parts store for which you have different type of sales such as a normal sale, a discount sale, or also mail-order sales. The standard sale class will be the parent to the other classes and has the member variable 'price'. With the program you also want to compute daily gross sales which is the sum of all individual sales.

You have the following classes to start with:

- Sale – which is the parent.

- 'Sale' has a member function called 'bill()' which essentially is a 'getPrice()' function.
- 'bill()' is used by other functions and is redefined by the descendent classes.
- Hence 'bill()' needs to be virtual.
- DiscountSale – which is derived from Sale

Interface for *Sales*

```
1  //Note you should put it into a separate namespace
2
3  class Sale
4  {
5  public:
6     //The two constructors just set `price'
7     Sale();
8     Sale(double thePrice);
9
10    virtual double bill() const;
11       //Bill returns price (needs to be virtual)
12
13    double savings(const Sale& other) const;
14       //Returns the savings if you buy other instead of the calling object
15       //Includes a call to `bill()'
16 protected:
17    double price;
18 };
19
20 //Overloaded `<' operator:
21 bool operator <(const Sale& first, const Sale& second);
22    //Compares two sales to see which is larger
23    //Includes a call to `bill()'
```

Listing 40: Example: Virtual functions - interface for 'Sales'

Implementation of *Sales*

```cpp
#include "sale.hpp"
//Note: you should put it into a separate namespace

Sale::Sale() : price(0)
{}

Sale::Sale(double thePrice) : price(thePrice)
{}

//Note that virtual is just needed in the declaration (not in definition)
double Sale::bill() const
{
   return price;
}

double Sale::savings(const Sale& other) const
{
   return (bill() - other.bill());
}

bool operator <(const Sale& first, const Sale& second)
{
   return (first.bill() < second.bill());
}
```

Listing 41: Example: Virtual functions - implementation for 'Sale'

Interface of *DiscountSale*

```cpp
#include "sale.hpp"
//Note: also put in a namespace as before

class DiscountSale : public Sale
{
public:
   Discount();
   DiscountSale(double thePrice, double theDiscount);

   virtual double bill() const;
      //Note: the keyword virtual is not needed here but it is good
      //practice to include it
protected:
   double discount; //hence the function has a price (inherited) and a discount
};
```

Listing 42: Example: Virtual functions - interface for 'DiscountSale'

Implementation of *DiscountSale*

```cpp
#include "discountsale.hpp"
//Note: also put in a namespace as before

DiscountSale::DiscountSale() : Sale(), discount(0)
{}

DiscountSale::DiscountSale(double thePrice, double theDiscount)
   : Sale(thePrice), discount(theDiscount)
{}

double DiscountSale::bill() const
{
   double fraction = discount/100;
   return (1 - fraction) * price;
   //Note that you can reference to price by name because it is a protected member
   //in the parent class (and not merely private).
}
```

Listing 43: Example: Virtual functions - implementation for 'DiscountSale'

As explained earlier, DiscountSale requires a different definition for its version of the function 'bill()'. But, when the member function savings and the overloaded operator ¡ are used with an object of the class DiscountSale, they will use the version of the function definition for 'bill()' that was given with the class DiscountSale.

This is indeed a pretty fancy trick for C++ to pull off. Consider the function call d1.savings(d2) for objects d1 and d2 of the class DiscountSale. The definition of the function savings (even for an object of the class DiscountSale) is given in the implementation file for the base class Sale, which was compiled before we ever even thought of the class DiscountSale. Yet, in the function call d1.savings(d2), the line that calls the function bill knows enough to use the definition of the function bill given for the class DiscountSale.
Example usage: main function

```cpp
#include <iostream>
#include "sale.hpp"
#include "discountsale.hpp"

using namespace std;

int main()
{
   Sale simple(10.00);
   DiscountSale discount(11.00, 10);

   cout.setf(ios::fixed);
   cout.setf(ios::showpoint);
   cout.precision(2);

   if (discount < simple)
   {
      cout << "Discounted item is cheaper." << endl;
      cout << "Savings is $" << simple.savings(discount) << endl;
   }
   else
   {
      cout << "Discounted item is not cheaper." << endl;
   }
   return 0;
}
```

Listing 44: Example: Virtual functions - sample usage in a main function

Virtual functions and extended type compatibility and the slicing problem

To deal with the slicing problem (explained below) in C++ inheritance you will use dynamic pointers to class objects. For that please note the following important concepts:

- If the domain type of the pointer pParent is a base class for the domain type of the pointer pChild, then you can assign: $pParent = pChild$ ... w/o losing any of the data members or member functions of the dynamic variable being pointed to by pChild.
- Although all extra fields of the dynamic variable are there, you will need *virtual* member functions to access them.

Assume you have two classes, Pet and Dog where Dog is a descendent of Pet.

- Pet has the member variable 'name'.
- Dog has an additional member variable 'breed'.

In C++ it i possible to assign all variables of an instance of Dog to an instance of Pet. However, even though this assignment is allowed the value that is assigned to the instance of Pet loses its 'breed' field. This is called the slicing problem.
Short illustration of the slicing problem

```
//Class of Pet
class Pet
{
public:
   virtual void print();
      //Prints out the name
   string name;
};

//Class of Dog
class Dog : public Pet
{
public:
   virtual void print(); //keyword 'virtual' not needed but good style
      //Prints out the name + the breed (thus needs to be virtual)
   string breed;
};

//Assume the below is embedded in a proper main function
Dog vDog;
Pet vPet;

vDog.name = "Tiny";
vDog.breed = "Great Dane";

//!!!!!
vPet = vDog;
//Note: this assignment is legit, but you lose the breed field in vPet
```

Listing 45: Short illustration of the slicing problem

But C++ offers a way to treat a Dog as a Pet w/o throwing away the name of the breed. To so so we use *pointers to dynamic object instances*. In the below you can still access the breed field from pPet. Also the 'print()' function would work because it is a virtual function. Most importantly if you call print() on pPet or pDog it will print out the same thing even though originally pPet has a different print() function. This is because pPet is essentially a Dog.

However, you can only access 'breed' through pPet using a virtual function and NOT directly by name (please see below).

```
Pet* pPet;
Dog* pDog = new Dog;

pDog->name = "Tiny";
pDog->breed = "Great Dane";

pPet = pDog;
//Note: this is legit and we do not lose the breed field in the pDog instance.

//The below is NOT legitimate...
cout << "name: " << pPet->name << "breed: " << pPet->breed;
//... this is because breed is not a member function of pPet.
//You can only make a call yo 'breed' through pPet using a virtual function.
```

Listing 46: Solution to slicing problem using pointers to dynamic object instances

Pitfall: Attempting to compile w/o defining all virtual member functions

Usually it is recommended to develop incrementally and test in between. However, if you have a class with a virtual function make sure you fully define all virtual member functions before compiling – otherwise you may get strange errors which are hard to identify.

Good programming tip: Make destructors virtual

If you are dealing with inheritance it is good practice to make destructors virtual, particularly if you make

Note that if you mark a destructor of a parent class as virtual, then ALL destructors of the child classes are virtual as well (w/o even specifying it).

Suppose you have the following two classes, Parent and Child:

- Parent which has a member variable 'pP' of pointer type.
  - The constructor creates a dynamic variable pointed to by 'pP'.
  - The destructor deletes the dynamic variable pointed to by pP.
- Child which has a member variable 'pC' of a pointer type.
  - The constructor creates a dynamic variable pointed to by 'pC'.
  - The destructor deletes the dynamic variable pointed to by 'pC'.

Now if you have the following:

```
1 Parent* pParent = new Child;
2 //... some code ...
3 delete pParent;
```

No suppose that the destructor of 'Parent' is NOT marked as virtual, then only the destructor for 'Parent' will be invoked. This will return to the heap the memory for the dynamic variable pointed to by pP, but the memory for the dynamic variable pointed to by pC will never be returned to the heap (until the program ends).

However (and this is why you should make destructors virtual), if the destructor for the Parent class were marked virtual, then when delete is applied to pParent, the destructor for the class Child would be invoked (since the object pointed to is of type Child). The destructor for the class Child would delete the dynamic variable pointed to by pC and then automatically invoke the destructor for the class Parent, and that would delete the dynamic variable pointed to by pP.

7 Exception handling

7.1 Basics - try-throw-catch statements

This is the basic mechanism for throwing and catching exceptions. It generally simplifies otherwise complex if-else statements.

The throw statement throws the exception (a value). The catch block catches the exception (the value). When an exception is thrown, the try block ends and then the code in the catch block is executed. After the catch block is completed, the code after the catch block(s) is executed (provided the catch block has not ended the program or performed some other special action).

If no exception is thrown, then the catch block(s) are ignored.

throw statement

When the throw statement is executed, the execution of the enclosing try block is stopped. If the try block is followed by a suitable catch block, then flow of control is transferred to the catch block. A throw statement is almost always embedded in a branching statement, such as an if statement. The value thrown can be of any type.

The syntax is: *throw ExpressionForValueToBeThrown;*; see example.

catch-block parameter

The catch-block parameter is an identifier in the heading of a catch block that serves as a placeholder for an exception (a value) that might be thrown. When a value is thrown in the preceding try block, that value is plugged in for the catch-block parameter. In example we use 'e' but you can also use any other non-reserved word.

Syntax

```
try
{
   Some_statements
   //Either some code with a throw statement or a
   //function invocation that might throw an exeption
   Some_more_statements
}
catch(TypeName e)
{  //Note that 'e' is the value that is thrown in the try block if some conditions
   //are met (see example below

   //Code to be performed if a value of the catch-block
   //parameter type is thrown in the try block
}
```

Listing 47: Basic try-throw-catch syntax

Short example

Note the example below is so simple that you would not use a try-throw-catch syntax for such a simple code - but it is good for illustration.

```
int main()
{
   int donuts, milk;
   double donutsPerGlass;

   try
   {
      cout << "Enter number of donuts: " << endl;
      cin >> donuts;
      cout << "Enter number of glasses: " << endl;
      cin >> milk;

      //The below is the condition whether to make a throw
      if (milk <= 0)
      {
         throw donuts;
         //If a throw is encountered, the throw is thrown value
         //becomes the value of the parameter in the catch block (i.e. the 'e')
      }
      //... if a throw was made, than the execution of try stops at
      //the throw and continues at the catch

      donutsPerGlass = donuts/static_cast<double>(milk);
      cout << "Donuts for each glass of milk: " << donutsPerGlass;
   }
   catch(int e)
   {
      //'e' is the thrown value from the try block
      cout << e << "donuts, and no milk!";
   }
   return 0;
}
```

Listing 48: Simple example of using try-throw-catch

7.2 Defining your own exception classes

A common thing is to define a class whose objects can carry the precise kind of information you want to throw to the catch block. It is just a standard class used in a particular way. An instance of the class would only be created in the try block if and only if you would enter a throw statement.

For example for the previous example you could define the following exception class.

```
class NoMilk
{
public:
   NoMilk(int howMany);
   //Sets 'count' to 'howMany'

   int getDonuts();
   //Returns 'count'
private:
   int count; //--> number of donuts
};

//The rest of the program will be exactly the same apart from:
if (milk <= 0)
{
   throw NoMilk(donuts);
   //This creates an instance of an object which
   //is thrown to the catch block
}

catch(NoMilk e)
{
   //Here 'e' is the instance of NoMilk and you refer to the number of donuts
   //through the getDonut function (i.e. 'e.getDonuts()').
}
```

Listing 49: Exception class

7.3 Multiple throws and catches

One try block can throw multiple and different exceptions. For each exception you would make one exception class. When catching multiple exceptions, the order of the catch blocks can be important.

When an exception value is thrown in a try block, the following catch blocks are tried in order, and the first one that matches the type of the exception thrown is the one that is executed.

Example of throwing multiple exceptions

In the example below we have two exception classes with but three throw blocks and two catch blocks.
Definition of two exception classes:

```
//Two exception classes
class NegativeNumber
{
public:
   NegativeNumber(string toCatchBlock);
      //Sets the message to 'toCatchBlock'
   string getMessage();
      //Returns the message
private:
   string message;
};

class DivideByZeor
{};
```

Listing 50: Exception class

main function:

```
int main()
{
   int jemHadar = 5, klingons = 10;
   double portion;

   try
   {
      if (jemHadar < 0)
         throw NegativeNumber("JemHadar");
      if (klingons < 0)
         throw NegativeNumber("Klingons");
      if (klingons = 0)
         throw DivideByZero();
      else
      {
         portion = jemHadar/static_cast<double>(klingons);
      }
   }

   //First catch block
   catch(NegativeNumber e)
   {
      cout << "Cannot have a negative number of ";
      cout << e.getMessage() << endl;
   }

   //Second catch block
   catch(DivideByZero)
   {
      cout << "Send for help." < endl;
   }
}
```

Listing 51: Exception class

Note that you can have a catch block that catches a thrown value of any type (see below). You would literally type the '...'.

```
catch(...)
{
   //Some code
}
```

Exception classes can be trivial

This refers to a class with no member variables and functions (other than the default constructor) such as the *DivideByZeo* class above.

Throwing an object of the class DivideByZero can activate the appropriate catch block. When using a trivial exception class, you normally do not have anything you can do with the exception (the thrown value) once it gets to the catch block. The exception is just being used to get you to the catch block. Thus, you can omit the catch-block parameter. (You can omit the catch-block parameter anytime you do not need it, whether the exception type is trivial or not.)

7.4 Throwing an exception in a function

Often it is good practice to throw an exception inside a function call. You can then decide to catch the exception inside the function that throws the exception – or you can also catch the exception outside the function. Put in another wording:

- You could define a function in which you define a try-block, throw an exception, and catch it all inside the function.
- Alternatively you can define a function that only throws an exception. You would call the function inside a try block. Here you could decide whether to catch the exception INSIDE the function or OUTSIDE the function (straight after the try block in which the function was called) – as in example below.

If you throw an exception inside a function you require an *exception specification*. This is a *throw* statement next the the function declaration and definition (see example and section below).

```
double safeDivide(int top, int bottom) throw (DivideByZero, int, double, char)
{  //NOTE: inside the brackets of the trow statement you only include the types of
   //exceptions that can be thrown separated by comma (this can be one or more).
   //You would not include variable names.

   //some function definition
}
```

```
class DivideByZero
{};

double safeDivide(int top, int bottom) throw (DivideByZero)
{
   if (bottom == 0)
      throw DivideByZero();

   return top/static_cast<double>(bottom);
}

int main()
{
   int numerator = 10, denominator = 20;
   double quotient;

   try
   {
      quotient = safeDivide(numerator, denominator);
      //The throw statement is in function call
   }

   //The catch statement is outside the function call
   catch(DivideByZero)
   {
      cout << "Error: Division by zero!" << endl;
      exit(0);
   }

   cout << "Quotient: " << quotient << endl;
   return 0;
}
```

Listing 52: Throwing exception inside a function

7.5 Exception specification

If you throw an exception inside a function you require an *exception specification*. This is a *throw* statement next the the function declaration and definition.

- Inside the brackets of the throw keyword you include all possible exception types separated by comma.
- You would not include variable names – only the types.

You use this as a warning for programmers that use the function that the function might possibly throw an exception. If there are exceptions that might be thrown, but not caught in the function definition, then those exception types should be listed in an exception specification.

If an exception is not caught inside a function then the next suitable catch statement outside the function will catch the exception.

If an exception is thrown in a function but is not listed in the exception specification, and not caught inside the function), then it will not be caught by any catch block. Instead your program will end.

However, if there is no specification list at all, not even an empty one, then it is the same as if all exceptions were listed in the specification list, and so throwing an exception will not end the program.

Keep in mind that the exception specification is for exceptions that 'get outside' the function. If they do not get outside the function, they do not belong in the exception specification. If they get outside the function, they belong in the exception specification no matter where they originate. If a function definition includes an invocation of another function and that other function can throw an exception that is not caught, then the type of the exception should be placed in the exception specification of the outer function.

Exception specification in a derived class

Keep in mind that an object of a derived class1 is also an object of its base class. So, if D is a derived class of class B and B is in the exception specification, then a thrown object of class D will be treated normally, since it is an object of class B and B is in the exception specification.

However, no automatic type conversions are done. If double is in the exception specification, that does not account for throwing an int value. You would need to include both int and double in the exception specification.

Note that when you redefine or override a function definition, you cannot add any exceptions to the exception specification (but you can delete some exceptions if you want). This makes sense, since an object of the derived class can be used anyplace an object of the base class can be used, and so a redefined function must fit any code written for an object of the base class.

One final warning: Not all compilers treat the exception specification as they are supposed to. Some compilers essentially treat the exception specification as a comment, and so with those compilers, the exception specification has no effect on your code. It is still good practice to include them should the compiler recognise them as such.

7.6 Best practices for exception handling

Firstly, its best to use the throwing of exceptions rarely as if often overcomplicates the code. You can catch exceptions in completely different places of your program as to where they are thrown and thus may make your code hard to follow.

When to throw an exception

- Separate throwing an exception and catching the exception into separate functions. Mostly you would ...
  - Include any throw statement within a function definition.
  - List the exception in the exception specification for the function.
  - Place the catch clause in a *different* function.
- Exceptions should be reserved for situations in which the way the exceptional condition is handled depends on how and where the function is used.
- If the way that the exceptional condition is handled depends on how and where the function is invoked, then the best thing to do is to let the programmer who invokes the function and handle the exception.
- If an exception is thrown but not caught anywhere, your program will end.
- Avoid creating nested try-catch blocks - but if you do, exceptions not caught in the inner block are caught in the outer block.

Exception class hierarchies

It can be very useful to define a hierarchy of exception classes. For example, you might have an 'ArithmeticError' exception class and then define an exception class 'DivideByZeroError' that is a derived class of ArithmeticError. Since a DivideByZeroError is an ArithmeticError, every catch block for an ArithmeticError will catch a DivideByZeroError. If you list ArithmeticError in an exception specification, then you have, in effect, also added DivideByZeroError to the exception specification, whether or not you list DivideByZeroError by name in the exception specification.

Testing for available memory when creating dynamic pointers

Since *new* will throw a *bad_alloc* exception when there is not enough memory to create the variable, you can check for running out of memory as follows:

```
1 try
2 {
3    NodePtr pointer = new Node;
4 }
5
6 catch(badAlloc)
7 {
8    cout << "Ran out of memory!" << endl;
9 }
```

Listing 53: Testing for available memory using exception handling

Rethrowing and exception

It is legal to throw an exception within a catch block. In rare cases, you may want to catch an exception and then, depending on the details, decide to throw the same or a different exception for handling farther up the chain of exception-handling blocks.

8 Templates

Templates allow you to design functions that can be used with arguments of different types and to define classes that are much more general than typical ones. For example if you have a simple 'swap' function that swaps two values, you could only use it with a certain type (e.g. int). However, the more general form would be the creation of a template which would allow you to define the function and it would work with any type as its inputs (e.g. int, char, double, someClass).

8.1 Templates for algorithm abstraction (function templates)

- The definition and the function declaration begin with the *template prefix*: 'template¡class T¿'. This tells the compiler that the definition or function declaration that follows is a template and the T is a type parameter. You could use any non-reserved word instead of 'T' – but it is typically 'T'.
- The 'class' keyword is has nothing to do with creating a class.
- You can use more than one type parameter: e.g. 'template¡class T1, class T2¿'
- The function is defined normally.
- You call the function with a type you use in your program (e.g. int, double, someType, etc.).
- Usually it is best not to separate the declaration and definition of template functions. That is to just define them in one go. Otherwise you may have compiler complications.
  - Thus do not use template function declarations – only definitions.
  - Ensure that the function template definition appears in the same file in which it is used and appears before the function template is called.
  - However, the function template definition can appear via an #include directive. Hence, define it in a .cpp or .hpp file and include it where the file before the template function(s) is/are called.

```
1 template<class T>
2 void swapValues(T& variable1, T& variable2)
3 {
4    T temp;
5
6    temp = variable1;
7    variable1 = variable2;
8    variable2 = temp;
9 }
```

Listing 54: Definition of a template function

8.2 Templates for data abstraction (class templates)

You can also create template classes for which the template type (usually 'T') is used for some class member variables (can also use multiple types T1, Tn - I assume).

The class definition and the definitions of the member functions are prefaced with:

```
template<class Type_parameter> //e.g. you can use 'T'
```

The class and member function definitions are then the same as for any ordinary class, except that the 'Tpye_parameter' can be used in place of a type.

The following is important to note:

- Member functions need to be itself defined as template functions (this also holds for constructors and destructors).
- When creating an instance of the class you need to specify which 'type' the variables need to be.

An example of a template class could be:

```
//Definition of the class
template<class T>
class Pair
{
public:
   Pair();
   Pair(T firstValue, T secondValue);
   T getElement(int position) const;
private:
   T first;
   T second;
};

//Definition of constructors
template<class T>
Pair<T>::Pair(T, firstValue, T secondValue)
   : first(firstValue), second(secondValue)
{}

//Definition of destructors
template<class T>
Pair<T>::~Pair()
{
   delte [] item;
}

//Definition of the member functions
template<class T>
void Pair<T>::getElement(int position) const
{
   //some function definition
}

//Create an instance of the class
Pair<int> score;
   //The 'int' then replaces all member variables types that have type 'T'
Pair<char> seats;
   //The 'char' then replaces all member variable tpyes that have type 'T'
```

Listing 55: Template class syntax

Type definitions

You can define a new class type name that has the same meaning as a specialised class template name, such as Pair¡int¿. The syntax for such a defined class type name is as follows:

```
1 typedef Pair<int> PairOfInt;
```

The type name PairOfInt can then be used to declare objects of type Pair¡int¿, as in the following example:

```
1 PairOfInt pair1, pair2;
```

9 Standard template library and C++ 11

The C++ standard template library (STL) is not really part of the C++ syntax but it is pretty much used as such. Essentially this involves a namespace ('std') in which certain template classes and template functions are defined. These include common data structures (i.e. containers) such as vectors, stack, queues, etc, as well as functions and variables types to use them. For example most 'containers' include *iterators*. Iterators are data types which are used like pointers but have some predefined functions that help navigate particular 'containers' with loops.

9.1 Iterators

An iterator is an object that can be used with a container to gain access to elements in the container. An iterator is a generalisation of the notion of a pointer, and the operators ==, !=, ++, −−and behave the same for iterators as they do for pointers. Every container class has their own iterators.

The basic outline of how an iterator can cycle through all the elements in a container is

```
1 STL_Container<type>::iterator p;
2 for (p = container.begin(); p != container.end(); p++)
3 {
4    //do something
5 }
```

Listing 56: Basic declaration and use of an interator, syntax

STL_Container is the name of the container class (for example, vector) and type is the data type of the item to be stored. The member function begin() returns an iterator located at the first element. The member function end() returns a value that serves as a sentinel value one location past the last element in the container.

Once declared and initialised you can typically manipulate iterators using the following overloaded operators:

- Pre- and postfix increment and decrement operators (++, and –), for moving the iterator to the next or previous data item.
- Equal and unequal operators, == and !=, to test whether two iterators point to the same data location.
- A dereferencing operator to give access to the data located where the pointer points to. But for some STL container classes, *p produces read-only access.

Most container classes (see later section), such as the 'vector' template then have member functions that point to special data elements in the data structure.

- containerObject.begin() – returns an iterator for the container that points to the first data item in the container.
- container.end() – returns an iterator for the container.

Note that instead of explicitly stating the type you could also use *auto* to infer the type.

```
1 //Instead of ...
2 vector<int>::iterator p = v.begin();
3
4 //... you could also write
5 auto p = v.begin();
```

Example of using iterators

```
#include <vector>
using std::vector;

int main()
{
   vector<int> container; //--> just a vector called 'container'
   for (int i = 1; i <= 4; i++)
   {
      container.push_back(i);
   }

   //Increment each element of the container by 1
   vector<int>::iterator p;
      //This means p is a variable and its type is iterator
      //iterator is the type named iterator that is defined in the class vector<int>...
      //...which in turn is defined in the std namespace
      //If we had not included the using directive 'using std::vector' we would require here
      //using std::vector<int>::iterator;
   for (p = container.begin(); p != container.end(); p++)
   {
      //Here p is an iterator and you assign it to the beginning of the container
      //and then iterate through it.
      *p += 1;
   }
}
```

Listing 57: Example for using iterators

9.1.1 Kinds of iterators

Different containers have different kinds of iterators – the three main kinds are:

- Forward iterators: ++ works on the iterator (e.g. slist).
- Bidirectional iterators: Both ++ and – work on the iterator (e.g. list).
- Random access iterators: ++, –, and random access all work with the iterator (e.g. vector).

For random access iterators (e.g. vectors) you can use the following expressions *vectorName[2]; *(iteratorName + 2); and iteratorName[2]* are all the same. It is important to note that these expressions itself leave the iterator where it is, however they literally mean 'two locations beyond the location of p' wherever p may be – p may not be at the beginning of the container. Hence it may lead to an issue if p is actually not at the beginning of the container and you think it is (perhaps you had used p++ before).

9.1.2 Mutable, constant, and reverse iterators

An iterator can usually be of the following types:

- *iterator*: mutable and non-revers. This means you can change the value of the variable the iterator is pointing to and you move through the container in a classic order (++ advances and – goes back).

- *const_iterator*: non-mutable and non-reverse: This means the variable the iterator is pointing to is 'read-only' and you move through the container in a classic order (++ advances and – goes back).
- *reverse_iterator*: mutable and revers. This means you can change the value of the variable the iterator is pointing to and you move through the container in reverse order (– advances and ++ goes back; see example).
- *const_reverse_iterator*: non-mutable and revers. This means the variable the iterator is pointing to is 'read-only' and you move through the container in reverse order (– advances and ++ goes back; see example).

```
1  //In the below it might not be 'vector' but could be any other container type
2  //and it might not be 'int' but it could also be any other type
3
4  //Classic iterator
5  std::vector<int>::iterator iteratorName;
6
7  //Read-only iterator
8  std::vector<int>::const_iterator iteratorName;
9
10 //Reverse iterator
11 std::vector<int>::reverse_iterator iteratorName;
12
13 //Read-only reverse iterator
14 std::vector<int>::const_reverse_iterator iteratorName;
```

Listing 58: Declare other iterator types

Note that for reverse iterators you need to pay attention how you move through the container – illustrated in example below. For reverse iterators ++ and – are interchanged, and you start from containerName.rbegin() and end at containerName.rend().

```cpp
#include <vector>

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

int main()
{
   //Create a vector called container that holds A, B, C
   vector<char> container;
   container.push_back('A');
   container.push_back('B');
   container.push_back('C');

   //Create reverse iterator to move through vector in backward fashin
   //- Here ++ means we are moving backward
   //- Also note you start from .rbegin() and not .begin()
   //  and end at .rend() and not .end()
   vector<char>::reverse_iterator rp;
   for (rp = container.rbegin(); rp != container.rend(); rp++)
   {
      cout << *rp << " ";
   }
   cout << endl;
}
```

Listing 59: Example of using reverse iterators

9.2 Containers

We have different kinds of data structures which are implemented as container templates in the STL. Among these are (most importantly): linked lists, vectors, stacks, and queues.

9.2.1 Sequential containers

Sequential containers are vectors, doubly linked lists, singly linked lists, and deques. In the STL only double linked lists (i.e. that have pointers pointing to previous and next node) is implemented singly linked lists are not.
STL basic sequential containers and container member functions:

Template name	**Iterators**	**Library header file**
list	mutable bidirectional, const bidirectional, and both in reverse	<list>
vector	mutable random access, const random access, and both in reverse	<vector>
deque	mutable random access, const random access, and both in reverse	<deque>
slist (may not be available)	mutable and constant forward	<slist>

Table 1: STL basic sequential containers

Member function (c is an object	**Meaning**
c.size()	Returns the number of elements in the container.
c.begin()	Returns an iterator located at the first element in the container.
c.end()	Returns an iterator located one beyond the last element in the container.
c.rbegin()	Returns an iterator located at the last element in the container. Is used with reverse iterators. Not a member function of slist.
c.rend()	Returns an iterator located at one before the first element in the container. Is used with reverse iterators. Not a member function of slist.
c.push_back(Element)	Inserts the Element at the end of the sequence. Not a member of slist
c.push_front(Element)	Insert the Element at the front of the sequence. Not a member of vector.
c.insert(Iterator, Element)	Insert a copy of Element before the location of 'Iterator'.
c.erase(Iterator)	Removes the element at location Iterator. Returns an iterator at the location immediately following. Returns c.end() if the last element is removed.
c.clear()	A void function that removes all the elements in the container.
c.front()	Returns a reference to the element in the front of the sequence which is equivalent to writing *(c.begin()).
c1 == c2	True if both are the same size AND each element of c1 is equal to the corresponding element of c2.
c1 != c2	!(c1 == c2).

Table 2: STL basic container member functions

Note that deque stands for doubly ended queue. A deque is a kind of super queue. With a queue you add data at one end of the data sequence and remove data from the other end. With a deque you can add data at either end and remove data from either end. The template class deque is a template class for a deque with a parameter for the type of data stored.
Basic example of using the *list* container:

```cpp
#include <iostream>
#include <list>
using std::cout;
using std::endl;
using std::list;

int main()
{
   //Create a doubly linked list object that contains integers
   //and fill the nodes
   list<int> listObject;
   for (int i = 1; i <= 3; i++)
   {
      listObject.push_back(i);
   }

   //Create a mutable iterator for the list object
   list<int>::iterator listIterator;
   for (listIterator = listObject.begin(); listIterator != listObject.end(); listIterator++)
   {
      *listIterator = 0;
   }
   return 0;
}
```

Listing 60: Basic example of using the 'list' container

9.2.2 Container adapters – *stack* and *queue*

These are stack, queue, and priority queue. These adapter template classes have a default container class on top of which they are built but you can also choose the a different underlying container.
Stack:

- Type name: stack<TYPE>or stack<TYPE, Underlying_container>for a stack of elements of type TYPE.
- Library header: <stack>
- Defined types: value_type, size_type.
- No iterators.

Member functions for *stack*

Member function (s is an object	**Meaning**
s.size()	Returns the number of elements in the stack.
s.empty()	Returns true if the stack is empty; otherwise false.
s.top()	Returns a mutable reference to the top member of the stack.
s.push(Element)	Inserts a copy of Element at the top of the stack.
s.pop()	Removes the top element of the stack (a void function).
s1 == s2	True if they are of equal size and each element of s1 is equal to the corresponding element of s2.

Table 3: Stack common member functions

Queue:

- Type name: queue<TYPE>or queue<TYPE, Underlying_container>for a queue of elements of type TYPE.
- For efficiency reasons, the Underlying_Container cannot be a vector.
- Library header: <queue>
- Defined types: value_type, size_type.
- No iterators.

Member function (q is an object	**Meaning**
q.size()	Returns the number of elements in the queue.
q.empty()	Returns true if the queue is empty; otherwise false.
q.front()	Returns a mutable reference to the front member of the queue
q.back()	Returns a mutable reference to the last member of the queue.
q.push(Element)	Inserts Element at the back of the queue.
q.pop()	Removes the front element of the queue (void function).
q1 == q2	True if they are of equal size and each element of q1 is equal to the corresponding element of q2.

Table 4: Queue common member functions

For short programs that use the stack templates check the book pages 1016 – 1017.

9.2.3 Associative containers – *set* and *map*

Set Stores a set of elements of a certain type where each element is only stored once – that means if you store an element twice, only one version is actually in it, hence all elements in the set are unique.

- Type name: set<TYPE>or set<TYPE, Ordeing>.
- The ordering is used to sort elements for storage. If no ordering is given the ordering is the binary operator <.
- Library header <set>.
- Defined types include: value_type, size_type.
- Iterators are both const and reverse and all are bidirectional.
- .begin(), .end(), .rbegin(), and .rend() have the expected behaviour.
- Adding or deleting elements does not affect iterators, except for an iterator located at the element removed.

Member function (s is an object	**Meaning**
s.size()	Returns the number of elements in the set.
s.insert(Element)	Inserts a copy of Element in the set given it is not already in the set.
s.erase(Element)	Removes Element from the set given it is in the set.
s.find(Element)	Returns a mutable iterator located at the copy of Element in the set. If Element is not in the set, then s.end() is returned
s.erase(Iterator)	Erases the element at the location of the Iterator.
s.empty()	Returns true if the set is empty and false otherwise.
s1 == s2	Returns true if the sets contain the same elements and false otherwise

Table 5: Set common member functions

```cpp
#include <iostream>
#include <set>
using std::cout;
using std::endl;
using std::set;

int main()
{
   //Declare a set called someSet and fill it
   set<char> someSet;
   someSet.insert('A');
   someSet.insert('D');
   someSet.insert('B');
   someSet.insert('D'); //--> this will have no effect given set already includes 'D'

   //Declare an iterator for set and print out each element
   set<char>::iterator setIterator;
   for (setIterator = someSet.begin(); setIterator != someSet.end(); setIterator++)
   {
      cout << *setIterator << endl;
   }

   //Remove 'B'
   someSet.erase('B');

   return 0;
}
```

Listing 61: Short example of using a set template class

Map A map stores pairs of elements which can be of different type (similar to a python dictionary). In essence it is a simple database where one variable is associated to another. For example you could a SSN as 'key' and a name as variable that is associated to it.

- Type name: map<keyType, assocType>or map<keyType, assocType, Ordeing>for a map that associates ('maps') elements of type keyType to elements of type assocType.
- The ordering is used to sort elements by key value for efficient storage. If no ordering is given, the ordering used is the binary operator <.
- Library header: <map>.
- Defined types include: keyType (for the type of the key values), assocType (for the type of the values mapped to the key), and size_type.
- Iterators are const and also reverse and all are bidirectional. However note that the iterators not including 'const_' are neither constant nor mutable. You change the key value but not the value of type assocType.
- .begin(), .end(), .rbegin(), .rend() have the expected behaviour.
- Adding or deleting elements does not affect iterators, except for an iterator located at the element removed.
- You can access a mapped value using array notation like:
  - someMapName["someKeyValue"]

  - This can also be used to redefine a the mapped value that is stored at a certain key or you can add a new value to the map.

- Use the 'this' pointer to access either a key or the value mapped to the key (you use it on the a mapIterator:
  - To access the key: mapIterator->first;
  - To access the value mapped: mapIterator->second;

Member function (m is an object	**Meaning**
m.size()	Returns the number of pairs in the map.
m.insert(Element)	Inserts Element in the map. Element is of type pair<keyType, assocType>. Returns a value of type pair<iterator, bool>. If the insertion is successful, the second part of the returned pair is true nd the iterator is located at the inserted element
m.erase(Target_Key)	Removes the element with the key Target_key.
m.find(Target_key)	Returns an iterator located at the element with key value Target_key. Returns m.end() if there is no such element.
m.empty()	Returns true if the map is empty and false otherwise.
m1 == m2	Returns true if the maps contains the same pairs and false otherwise.

Table 6: Map common member functions

```
#include <iostream>
#include <map>
#include <string>
using std::cout;
using std::endl;
using std::map;
using std::string;

int main()
{
   //Declare a map where a string is associated to a string (call map 'planets')
   map<string, string> planets;

   //Fill the map with entries
   planets["Mercury"] = "Hot planet";
   planets["Venus"] = "Atmosphere of sulphuric acid";
   planets["Earth"] = "Home";

   //Use map in certain ways
   cout << "Entry of Mercury: " << planets["Mercury"] << endl;
   if (planets.find("Mercury") != planets.end())
   {
      cout << "Mercury is in the map." << endl;
   }

   //Declare a map iterator and iterate through the map
   map<string, string>::const_iterator mapIterator;
   for (mapIterator = planets.begin(); mapIterator != planets.end(); mapIterator++)
   {
      //Here you first print out the key and then the mapped value
      cout << mapIterator->first << " is the key for " << mapIterator->second << endl;
   }
   return 0;
}
```

Listing 62: Short example of using a map template class

Tip: use initialisation, ranged for, and auto with containers You can initialise your container objects using the uniform initialiser list format, which consists of initial data in curly braces. You can also use auto and the ranged for loop to easily iterate through a container. Consider the following two initialised collection objects (one map and one set):

```
map<int, string> personIDs =
{
   {1, "Walt"},
   {2, "Kenrick"}
};

set<string> colors = {"red", "green", "blue"};
```

We can iterate through each container conveniently using a ranged for loop and auto:

```
1  //personIDs and colors are defined in code block above
2  for (auto p : personIDs)
3  {
4     cout << p.first << " " << p.second << endl;
5  }
6
7  for (auto p : colors)
8  {
9     cout << p << " ";
10 }
```

Listing 63: Easy code to iterate over containers

9.3 Generic algorithms

9.4 C++ is evolving

```
1
```

Listing 64: Definition of a template function