The C++ Class

The C++ class allows users to define their own data types

A class has the attributes:

- A class name
- Zero or more data members
- Zero or more member functions
- Levels of access for the members

Types, Classes and Objects

- In C++ (as in C) there are a number of **built-in types**.
  - **int** and **float** are examples of built-in types.

- A **class** is a **user-defined type**.
  - *Person* and *SavingsAccount* are possible C++ classes

- An **object** is a thing
  - Following the declaration
    ```
    int i, j, k;
    ```
  - **`i, j,`** and **`k`** are objects of type **`int`**

  - Following the declaration:
    ```
    Person fred(), mae(), inga();
    ```
  - **`fred, mae`** and **`inga`** are objects of type **`Person.`**

- Most classes correspond to types of things in the problem domain.
  - Exceptions are subsidiary classes introduced as implementation details.

Class is the most fundamental concept in Object Oriented Programming.

Declaration of a Class

- Defines **data members** which exist in every object of the class.
- Provides declarations (*prototypes or signatures*) of **member functions** which may be applied to objects of the class.
- Specifies **access level** (visibility) of both data members and member functions.

- Conventionally placed in a **header file** named *classname*.h.
  - This file must be `#include`'d in any source file which references this class.
  - Should contain directives to prevent accidental multiple inclusion of the header file.

Class Declaration Example

```
#ifndef MYCLASS_H            // If MyClass.h has already been
#define MYCLASS_H            //    included somewhere, skip this
                             //    declaration
class MyClass{
      int   u;               // Since no access specified, default
      float funct1( . . .); //    is private.
   public:
      int   v;               // v, funct2 access unrestricted
      int   funct2( . . .);
   protected:
      char  w;               // w, funct3 only accessible to member
      void  funct3( . . .); //    functions of MyClass or classes
                             //    which inherit from it
   private:
      float x;               // x, funct4 only accessible to member
      char  funct4( . . .); //    functions of MyClass
   public:                   // Access specifiers can appear in any
      . . . .                //    order, and may be repeated.
};
#endif
```

Member Access Level

- **`public`**
  - `public` member functions form the interface between a class and the rest of the application.
  - Conventionally these come first in the class declaration.

- **`protected`**
  - These will occur second in the class declaration.
  - This access level is the same as `private` except in the context of inheritance.
  - *(We will make no further reference to `protected` until we cover inheritance)*

- **`private`**
  - These normally occur last in the class declaration.
  - If no access level is specified the default is `private`. This provides protection in cases where the first specifier is accidentally omitted.
    Recommend that access levels are always specified explicitly.

Data Members - Access Level

- Data members (almost always) should not be `public`
  - Making data members accessible only via member functions has a number of advantages:
    - Errors due to accidental modification are eliminated.
    - Functions which modify data members (*mutator* functions) can perform validity checks on new values before the modifications are made.
    - Other parts of the application can take no advantage of the format within which data is stored.
      - Hence, this format can be changed (for efficiency or other reasons), without the danger of introducing errors elsewhere in the application.

- A data member that is `const` can safely be made `public`.

Data Members - Types

- Data members may be:
  - Elementary data types (e.g. **`int`**, **`char`** etc)
  - Standard composite types (i.e. `struct` or array)
  - Pointers to any of the previous – or pointers to pointers etc.
  - Objects of any class whose declaration completely precedes its use.
  - Pointers to objects of a type that is known to be a class.

Data Members - Example

In "person.h"

```
#ifndef PERSON_H
#define PERSON_H
#include "Car.h"       // contains complete declaration of Car
class Computer;        // a forward declaration of Computer
class Person{
   public:
      const long medicareNo; // const so safe to declare public
      . . . .
   private:
      int      age;
      string   name;
      Person   spouse; // illegal - Person decl'n not complete
      Person*  father; // OK
      Car      myCar;  // OK
      Computer hComp;  // illegal
      Computer*myComp; // OK
};
#endif
```

Member Functions

- Five categories of Class Member Functions deserve mention:
  - **Constructors**
    - Give initial values to data members when an object is created.
  - **Destructors**
    - Perform any necessary clean-up when an object ceases to exist.
  - **Accessor functions**
    - Read-only functions that do not alter values of any data members.
  - **Mutator functions**
    - Functions whose purpose is to alter data members.
  - **Custom functions**
    - Other special-purpose functions.
  - **Helper functions**
    - Functions only called by other member functions.

Member Functions (cont'd)

- Helper functions will usually be **private**, but other functions will be **public**.
  - Helper functions are implementation details. Of no concern to rest of the application

- Public member functions (and possibly **public const** data members) form the ***public interface*** of the class.
  - The public interface of the class, together with a description of the use of the public functions, constitute a

contract between the developer of the class and the users of the class

Member Functions - Declaration

- *In "person.h"*

```
#ifndef PERSON_H
#define PERSON_H
class Person{
   public:
            Person(char* nameIn, int ageIn);    // constructor
            ~Person();                          // destructor
      void  SetAge(int newAge);                 // mutator
      int   GetAge();                           // accessor
   private:
      int   CheckAge(int ageToCheck);           // helper
      char* name;
      int   age;
};
#endif
```

Member Function - Implementation

- *In "person.cpp"*

```
#include <string.h>
#include "person.h"
Person::Person(char*nameIn, int ageIn){
   name = new char[strlen(nameIn) +1];
   strcpy(name, nameIn);
   age = CheckAge(ageIn);
}
Person::~Person(){
   delete [] name;
}
void Person:: SetAge(int newAge){
   age = CheckAge(newAge);
}
int Person::GetAge(){
   return age;
}
int Person::CheckAge(int ageToCheck){
   if(ageToCheck <0 || ageToCheck > 150) return -1;
   return ageToCheck;        // if invalid return a garbage value
};
```

Object Creation and Destruction

- Objects may be created:
  - Globally
    - Memory is allocated by the compiler
  - Locally
    - Memory is allocated on the stack when the block in which they are declared is entered. Object is destroyed and memory released when execution leaves this block.
  - Dynamically
    - Objects are created via the verb **new**. They are explicitly destroyed by use of the verb **delete**.
- When an object is created:
  - Memory is allocated for a set of data members.
    A **constructor** is executed providing initialisation of the data members.
- When an object is destroyed:
  - The **destructor** is executed and then the memory is released.

Object Creation and Destruction - example

```
Person angie( . . . );          // Global allocation of angie

void SomeFunction( . . . ){
   Person bruce( . . . );       // Local allocation of bruce
   . . . .
}                                     // bruce destroyed

void main(){
   Person* david;
   david = new Person( . . . ) // Dynamic allocation of david
   . . .
   delete david;                      // david destroyed
}
                                      // angie destroyed
```

Operator *new* – built-in types

```
type* dataName = new type(expression);
```

dataName is a pointer to ***an object*** of a built-in type. The object has the initial value specified by *expression*.

```
type* dataName = new type[expression];
```

dataName is a pointer to an ***array of objects*** of a built-in type. The size of the array is given by *expression*.

e.g.

```
char* flag = new char('*');
```

flag is a pointer to a **char**, which has the initial value **'*'.**

```
char* address = new char[30];
```

address is a pointer to an uninitialised array of 30 characters.

Operator *new* – classes

```
ClassName* name = new ClassName(params for constructor);
```

name is a pointer to ***an object*** of a class **ClassName** . The parameters will be used by the constructor to initialise this object.

```
ClassName* arrName = new ClassName[expression];
```

arrName is a pointer to an ***array of objects*** of class **ClassName** . The size of the array is given by *expression*. Objects will be initialised by the constructor which takes no parameters (the ***default*** constructor).

e.g.

```
Person* fred = new Person("Fred");
```

fred is a pointer to a **Person**, whose constructor received the string **"Fred".**

```
Person* names = new Person[30];
```

names is a pointer to an array of 30 **Person** objects. Each will be initialised by the default constructor.

Operator *delete*

- The most common contexts for use of the operator **delete** are:

```
delete pointerName;
```

where **`pointerName`** is a pointer to an object **which has been created by the use of the operator `new`.**

The destructor for the object is executed, and then the space occupied is made available.

`pointerName` is now an invalid pointer – it is ***not*** automatically set to NULL.

```
delete [] pointerName;
```

where **`pointerName`** is a pointer to an array of objects **which have been created by the use of the operator `new`**.

The destructor for each object is executed in turn, and then the space occupied is made available.

`pointerName` is now an invalid pointer – it is ***not*** automatically set to NULL.

Operator *delete* - example

```
char* address = new char[30];
```

address is a pointer to an array of 30 characters.

```
delete [] address;
```

deletes the **whole array** pointed to by **address** .

The pointer is **not altered** by this operation

```
delete address;
```

deletes the **one character** pointed to by **address** .

The pointer is **not altered** by this operation.

```
char address1[30];
```

is an array of 30 characters.

```
delete [] address1; or
delete address1;
```

are errors, causing unspecified action – usually your application will **crash**.

delete** must only be used if objects have been created by **new.

Deletion of a NULL pointer

- Applying delete to an unitialised pointer has an unspecified effect. If you are lucky it will cause your program to crash.

- Applying delete to a pointer containing **`NULL`** (i.e. 0) is guaranteed to do nothing.

```
person* father;
delete father; // unspecified action

person* father = NULL;
delete father; // no problem!
```

- A pointer that points to nothing should be set to **`NULL`** ..

Input/Output

I/O is not defined as part of either the C or C++ languages.

In C functions *printf()* and *scanf()* etc are provided by the *stdio* library.
These are also available to C++ programmers.

In C++ additional capabilities are provided by the *iostream* library,
which is part of the ANSI Standard Library.

Input/Output

- Programs using the *iostream* library must have

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

  The *iostream* library
  - Provides the predefined stream objects
    `cout` which is tied to the standard output (screen)
    `cin` which is tied to the standard input (keyboard)
    `cerr` which is tied to the standard error (screen)
  - Overloads the operators
    `<<` and `>>`
    to provide output and input respectively
  - Provides member functions to provide additional functionality.
- The classes *ifstream*, *ofstream* and *fstream* allow
  a file to be used

  for *input*, *output* or *input*/*output* respectively.

<< Output Operator

`cout << value`

will display ***`value`*** on the screen, where ***`value`*** may be a *constant*, *variable* or *expression* of any of the standard types.

If ***`value`*** is of type:

`char*` (pointer to an array of chars)
Successive characters displayed until a NULL character is encountered.

`short, int, long, unsigned`
Number is displayed as a decimal integer using as many characters as needed.

`float, double`
Displays the number with a default precision of 6 significant figures.
Embedded decimal point used normally. Exponential format is used if this would require fewer characters

`pointer` (other than a pointer to an array of chars)
Address displayed in hexadecimal.

`endl`
Outputs a *newline* character.

`iostream` manipulators

Output formatting capability is provided by the *iomanip* library. Header used is

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

```
cout << setprecision(4);
```

Print 4 significant digits for floating point numbers (stays in force until changed).

```
cout << width(10);
```

The next (only) item output will occupy at least 10 characters.

```
cout << setf(ios::left);
cout << setf(ios::right);
```

Items will be output left (or right) justified in the field.

Further information on iostream manipulators is given at
http://www.cplusplus.com/ref/iostream/ios_base/

>> Input operator

`cin >> variable`

will read a value of the appropriate type and assign that value to *variable*.

If ***`variable`*** is of type:

`char*` (pointer to an array of chars)

white space characters will be skipped, and successive characters will be read into successive memory locations until *white space* is encountered.

`short, int, long, unsigned`

white space will be skipped, and successive decimal digits will be read until a non-digit character is encountered. These digits will be converted to a number and assigned to ***`variable`***.

`float, double`

white space will be skipped, and decimal digits, possibly including a decimal point and possibly followed by an exponent will be read until a character which could not be part of a floating point number is encountered. These characters will be converted to a number and assigned to ***`variable`***.

`white space —` any number of the characters *space*, *tab* or *newline* in any order.

Other Input Operations

- The overloaded operator >> will input the next *word* of a character string. Often this is inconvenient.
- There are functions of the `istream` (and its descendant `ifstream`) class providing different options:

```
istream& getline(char* buff, int len, char delim = '\n');
```

Extracts characters from input until

`delim` character is encountered, or

`len` characters have been read:

```
istream& get(char & ch);
```

Puts the next character (even if it is *white space*) into the reference variable `ch`.

Other Input Considerations

- The code sequence

```
int age; char name[50];
cout << "What is your age? ";
cin >> age;
cout << "What is your name? ";
cin.getline(name, 50);
```

will put an empty string into name, this is because the `cin >>` left the newline character in the input buffer.

The correct sequence is:

```
int age; char name[50];
cout << "What is your age? ";
cin >> age;
cin.ignore(50, '\n');
cout << "What is your name? ";
cin.getline(name, 50);
```

The above is a common error which leads to strange results

Checking for Incorrect Input

Consider the code sequence

```
cout << "What is your age? ";
cin >> age;
```

What happens if the user inputs the string `xyz` ? The system can not convert this to an integer.

To guard against this we need to do something like:

```
cout << "What is your age? ";
cin >> age;
while(cin.fail()){
   cin.clear()
   cout << "You must input a number";
   cin.ignore(50, '\n');
   cin >> age;
}
```

File Input/Output

- Programs using any file I/O must:

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

- To open a file for **output**

```
ofstream myOut("filename");
if(!myOut)
   cout << "File filename could not be opened";
```

`myOut` can now be used in the same way as `cout`, except that output will go to `filename`, rather that to the screen.

- To open a file for **input**

```
ifstream myIn("filename");
if(!myIn)
   cout << "File filename could not be opened";
```

`myIn` can now be used in the same way as `cin`, except that input will come from `filename`, rather that from the keyboard.

Input/Output

I/O is not defined as part of either the C or C++ languages.

In C functions *printf()* and *scanf()* etc are provided by the *stdio* library.
These are also available to C++ programmers.

In C++ additional capabilities are provided by the *iostream* library,
which is part of the ANSI Standard Library.

Input/Output

- Programs using the *iostream* library must have

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

  The *iostream* library
  - Provides the predefined stream objects
    `cout` which is tied to the standard output (screen)
    `cin` which is tied to the standard input (keyboard)
    `cerr` which is tied to the standard error (screen)
  - Overloads the operators
    `<<` and `>>`
    to provide output and input respectively
  - Provides member functions to provide additional functionality.
- The classes *ifstream*, *ofstream* and *fstream* allow
  a file to be used

  for *input*, *output* or *input*/*output* respectively.

<< Output Operator

`cout << value`

will display ***`value`*** on the screen, where ***`value`*** may be a *constant*, *variable* or *expression* of any of the standard types.

If ***`value`*** is of type:

`char*` (pointer to an array of chars)
Successive characters displayed until a NULL character is encountered.

`short, int, long, unsigned`
Number is displayed as a decimal integer using as many characters as needed.

`float, double`
Displays the number with a default precision of 6 significant figures.
Embedded decimal point used normally. Exponential format is used if this would require fewer characters

`pointer` (other than a pointer to an array of chars)
Address displayed in hexadecimal.

`endl`
Outputs a *newline* character.

`iostream` manipulators

Output formatting capability is provided by the *iomanip* library. Header used is

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

```
cout << setprecision(4);
```

Print 4 significant digits for floating point numbers (stays in force until changed).

```
cout << width(10);
```

The next (only) item output will occupy at least 10 characters.

```
cout << setf(ios::left);
cout << setf(ios::right);
```

Items will be output left (or right) justified in the field.

Further information on iostream manipulators is given at
http://www.cplusplus.com/ref/iostream/ios_base/

>> Input operator

`cin >> variable`

will read a value of the appropriate type and assign that value to *variable*.

If ***`variable`*** is of type:

`char*` (pointer to an array of chars)

white space characters will be skipped, and successive characters will be read into successive memory locations until *white space* is encountered.

`short, int, long, unsigned`

white space will be skipped, and successive decimal digits will be read until a non-digit character is encountered. These digits will be converted to a number and assigned to ***`variable`***.

`float, double`

white space will be skipped, and decimal digits, possibly including a decimal point and possibly followed by an exponent will be read until a character which could not be part of a floating point number is encountered. These characters will be converted to a number and assigned to ***`variable`***.

`white space —` any number of the characters *space*, *tab* or *newline* in any order.

Other Input Operations

- The overloaded operator >> will input the next *word* of a character string. Often this is inconvenient.
- There are functions of the `istream` (and its descendant `ifstream`) class providing different options:

```
istream& getline(char* buff, int len, char delim = '\n');
```

Extracts characters from input until

`delim` character is encountered, or

`len` characters have been read:

```
istream& get(char & ch);
```

Puts the next character (even if it is *white space*) into the reference variable `ch`.

Other Input Considerations

- The code sequence

```
int age; char name[50];
cout << "What is your age? ";
cin >> age;
cout << "What is your name? ";
cin.getline(name, 50);
```

will put an empty string into name, this is because the `cin >>` left the newline character in the input buffer.

The correct sequence is:

```
int age; char name[50];
cout << "What is your age? ";
cin >> age;
cin.ignore(50, '\n');
cout << "What is your name? ";
cin.getline(name, 50);
```

The above is a common error which leads to strange results

Checking for Incorrect Input

Consider the code sequence

```
cout << "What is your age? ";
cin >> age;
```

What happens if the user inputs the string `xyz` ? The system can not convert this to an integer.

To guard against this we need to do something like:

```
cout << "What is your age? ";
cin >> age;
while(cin.fail()){
   cin.clear()
   cout << "You must input a number";
   cin.ignore(50, '\n');
   cin >> age;
}
```

File Input/Output

- Programs using any file I/O must:

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

- To open a file for **output**

```
ofstream myOut("filename");
if(!myOut)
   cout << "File filename could not be opened";
```

`myOut` can now be used in the same way as `cout`, except that output will go to `filename`, rather that to the screen.

- To open a file for **input**

```
ifstream myIn("filename");
if(!myIn)
   cout << "File filename could not be opened";
```

`myIn` can now be used in the same way as `cin`, except that input will come from `filename`, rather that from the keyboard.

Laboratory Exercise 1

You may obtain a (free) copy of MS Visual C++ Compiler for use on your home computer.

Please use this tutorial to acquaint yourself with the MS Visual C++ compiler.

To get started:

1. Log in
   Enter your Username – *Login*
   From the bottom row of buttons select **gnome -> Windows XP**
   (Only now) Enter your Password – *Login*
   Create a folder (preferably on a memory stick) called "Lab1"

2. Enter the IDE
   Select **Visual Studio 2010**
   File -> New -> Project
   Visual C++ -> Win32 -> Win32 Console Application
   Enter **Name** as "tut1"
   Select **Location** as your Lab1 folder
   Select **Solution name** as "tut1"

3. Enter your program (Note that key-words are automatically **bold**).

4. Compile, link and execute your program.
   Select **Debug -> Start Debugging**
   If there are any errors, correct them and repeat

It is recommended that you use a USB memory device to transfer files between home and University.

1. Type and save this code through the IDE. Then compile it and execute it.

```
// Tutorial 1 - A simple program which utilises
// <iostream>.
// We will discuss this in more detail next week.

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

int main(int argc, char *argv[])
{
   char name[50];
   cout << "\n********************\n";
   cout << "Hello world!" << endl;
   cout << "Please enter your name: ";
   cin.getline(name, 50);
   cout << "\n********************" << endl;
   cout << endl;
   cout << "Hello " << name << " ! " << endl;
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

2. Modify the program so that it prints Hello World! three times.
   Do this with a for loop and then repeat the enhancement with the use of a while loop.
3. Enhance the program so that it displays a menu of options to perform different functions.
   Use a switch statement to evaluate which functions should be executed.

These <iostream> functions may prove useful.

cout << *string* sends strings and numeric values to standard output

cout << endl sends a newline character to standard output and flushes the output buffer

cin >> *string* accepts strings and values from standard input, ignoring leading whitespace, and terminating when a whitespace character is encountered .
(whitespace: any number of spaces, tabs or newlines)

cin.get(ch) extracts a character from the input stream into a character variable.

You may still use <stdio.h> but it is also worthwhile learning <iostream.h> capabilities.

Object-Oriented Programming in C++

Assignment 1

This is the first of the laboratory exercises. The exercise consists of implementing a Time class for which the public interface has been specified as follows:

```
class Time

{

    public:
      // Constructors and destructor
      Time();           // Default constructor
      Time(Time const& time);         // Copy constructor
      Time(long secondsAfterMidnight);
      Time(char const* tstring);   // String in hh:mm:ss
                                   // format (24 hour time).
      ~Time();                     // Destructor
      // Const (read-only) functions
      char* GetTime(bool military = false) const;
      // Return string representation of the time.
      //          If military then hh:mm:ss
      //             else hh:mm:ss am.
      int   GetHour() const;        // Get hour value.
      int   GetMinute() const;      // Get minute value.
      int   GetSecond() const;      // Get second value.
      bool  operator !() const;     // Returns true if time
                                    //     is NOT valid
      bool  IsAM() const;           // Is time AM?
      bool  operator ==(Time const& time) const; // Are times equal?
      Time  operator +(Time const& time) const;  // Add times.
      Time  operator -(Time const& time) const;  // Subtract times.
      // Non-const (read/write) functions
      void  SetTime(int hrs, int mins = 0, int secs = 0);
      // Set the time to the values supplied (in 24 hour format)
      void  AddHours(int hours);          // Add hours (which may be <0).
      void  AddMinutes(int minutes);      // Add minutes (which may be < 0).
      void  AddSeconds(int seconds);      // Add seconds (which may be < 0).
};
```

Implement the specified functionality using whatever private data members and member functions you require.

This should be completed by Week 6 (11th August) and should include :

1. The completed TIME.CPP and TIME.H files. The TIME.CPP should be well commented.

2. You should test most, if not all, the logic paths in the class. The test harness should take the form of a main program, preferably driven by options selected by the tester.

Please bring progressive code to class, when it can be checked in tutorial time to ensure that you are on the right track.

Object-Oriented Programming in C++

August 2012 - Assignment 1

```
class Time

{

    public:
      /* Constructors and destructor */

      /* This is the default constructor
      * This constructor takes no arguments and sets the time to 00:00:00
      * pm
      */

      Time();           // Default constructor

      /* This constructor takes a pre-existing Time object as
      * an argument (parameter) and it is passed by reference to the
      * constructor. It creates a copy of the pre-existing time object
      * which was passed in as an argument.
      */

      Time(Time const& time);         // Copy constructor

      /* This constructor takes as an argument a long integer which
      * represents seconds after midnight. The seconds argument is
      * converted to hours minutes and seconds. The argument is passed
      * by value
      */

      Time(long secondsAfterMidnight);

      /* This constructor takes as an argument a string which is
      * passed to the constructor by value. The argument string is in
      * the format hh:mm:ss. The string should be tested to see if it is
      * in the right format. If it is in the wrong format then the validity
      * flag should be set to false and the hours minutes and seconds
      * data items should be left uninitialized.
```

```
* If the argument string is in the right format, the hours minutes
* and seconds should be checked to see that they have a valid range.
* If the range is invalid e.g. hours greater than 24 or the minutes
* or seconds greater than 59, then the validity flag should be set to
* false and the hours minutes and seconds data items should be left
* uninitialized.
* Note that the description of the boundary conditions in the above
* scenario does not cover all cases where the argument string is
* invalid – can you think of some others?
* If the hours minutes and seconds from the argument string are
* valid, the valid flag should be set to true and the hours minutes
* and seconds should be initialised to the passed in values. If you
* are storing the time as seconds after midnight, then that value
* should be calculated and stored.
*/

Time(char const* tstring);   // String in hh:mm:ss
                             // format (24 hour time).


~Time();                     // Destructor

/* Const (read-only) functions
* The following functions are all "getter" functions that return a
* value.

* The GetTime function returns a string representation of the time
* The Boolean argument is optional. If a Boolean value is passed in
* as an argument to the function and its value is true then the
* string returned by the function is in 24 hour format.
* If military then hh:mm:ss else hh:mm:ss am.

char* GetTime(bool military = false) const;

/* The GetHour function returns the hour value in a Time object as
* an integer. There is no argument to the function.
*/
int   GetHour() const;        // Get hour value.

/* The GetMinute function returns the minute value in a Time object
```

```
* as an int. There is no argument to the function.
*/

int   GetMinute() const;      // Get minute value.

/* The GetSecond function returns the hour value in a Time object as
* an int. There is no argument to the function.
*/

int   GetSecond() const;      // Get second value.

/* The operator ! is used to determine if a time object
*  is valid. It returns the value of the valid field.
*/

bool  operator !() const;     // Returns true if time
                              //     is NOT valid

/* The IsAM() function takes no arguments and returns a Boolean
* value of true if the time object it refers to has an AM time
* If the time is PM or the time object is invalid the value of false
* is returned.
*/
bool  IsAM() const;           // Is time AM?

/* The == operator returns a value of true if all the stored
* attributes of the two objects are equal.
*/
bool  operator ==(Time const& time) const; // Are times equal?

/* The + operator returns a time object. The time stored in the
* returned object is the sum of the hours, the minutes and the
* seconds. Note the following :
* (1) The addition of the seconds may result in a seconds value
* greater than 60. In this case there is a carry over of 1 to the
* minutes addition.
* (2) Similarly to point (1) the addition of the minutes may result
* in a minutes value greater than 60. In this case there is a carry over
* of 1 to the hours addition.
* (3) The addition of the hours may result in a sum of hours greater
```

```
* than 24. This is an error condition known as an overflow.
*
* If either of the time objects being added together has a valid
* flag of false then the returned object is false and its hours
* minutes and seconds values are not initialised. If the returned object
* stores the seconds since midnight value it is not initialised and the
* valid flag is set to false..
*
* If there is an overflow in the result of the addition then the valid
* flag of the object returned is set to false and all other attributes
* of the returned object are uninitialized.
*/
Time  operator +(Time const& time) const;  // Add times.

/* The - operator returns a time object. The time stored in the
* returned object is the difference of the value of the hours, the
* minutes and the seconds in the first object minus the hours,
* minutes and seconds in the second object. Note the following :
* (1) The subtraction of the seconds may result in a seconds value
* less than 0. In this case there is a borrow of 1 from the
* minutes in the first time object.
* (2) Similarly to point (1) the subtraction of the minutes may result
* in a minutes value less than 0. In this case there is a borrow of 1
* from the hours column.
* (3) The subtraction of the hours may result in a value of hours
* less than 0. This is an error condition known as an underflow.
*
*
* If either of the time objects being subtracted has a valid
* flag of false then the returned object is false and its hours
* minutes and seconds values are not initialised. If the returned object
* stores the seconds since midnight value it is not initialised.
*
* If there is an underflow in the result of the subtraction then the
* valid flag of the object returned is set to false and all other
* attributes of the returned object are uninitialized.
*/
Time  operator -(Time const& time) const;  // Subtract times.

/* Non-const (read/write) functions
```

```
* The SetTime function returns no value. It takes 3 integer
* arguments, the hrs, mins and secs. If the function is called with
* only one argument it corresponds to the hrs argument, and the mins
* and secs
* arguments are set to their default values of 0. In the event of
* invalid values being passed in as arguments, the valid flag for the
* object is set to false and no changes will occur to the object's
* other data items.
*/

void  SetTime(int hrs, int mins = 0, int secs = 0);

/* The AddHours function takes one integer argument. The integer can have
* a range between -24 and +24. If the argument is outside this range it
* should set the valid flag of the object to false and not change the value
* of the hours data item.
* If the argument is positive and within the acceptable range, the hours
* argument is added to the existing hours data item in the object. If the
* argument is negative and within the acceptable range, it is subtracted
* from the existing hours argument.
* It is possible for overflow or underflow to occur where the resulting
* time is outside the range 00:00:00 to 24:00:00. In this case no change
* should occur in the hours, minutes or seconds data item and the valid
* flag should be set to false.
*
*/
void  AddHours(int hours);          // Add hours (which may be <0).

/* The AddMinutes function takes one integer argument. The integer can have
* a range between -1440 and +1440. There are 1440 minutes in a day. If the
* value passed in is outside this range, then underflow or overflow will
* occur. In this case you should set the valid flag of the object to false
* and not change the value of the minutes data item.
*
* If the argument is positive and within the acceptable range, the minutes
* argument is added to the existing hours data item in the object. If the
* argument is negative and within the acceptable range, it is subtracted
* from the existing hours argument.
* It is possible for overflow or underflow to occur in the addition or
```

```
      * subtraction operation. In many cases this will be acceptable and the
      * hours or seconds data item should also be altered to allow for the
      * overflow or underflow.
      * In the remaining cases of overflow or underflow, where the resulting time
      * is outside the range 00:00:00 to 24:00:00 no change should occur
      * in the hours, minutes or seconds data item and the valid flag should be
      * set to false.
      */

      void  AddMinutes(int minutes);      // Add minutes (which may be < 0).

      /* The AddSeconds function takes one integer argument. The integer can have
      * a range between -86400 and +86400. There are 86400 seconds in a day. If
      * the value passed in is outside this range, then underflow or overflow
      * will occur. In this case you should set the valid flag of the object to
      * false and not change the value of the minutes data item.
      *
      * If the argument is positive and within the acceptable range, the minutes
      * argument is added to the existing hours data item in the object. If the
      * argument is negative and within the acceptable range, it is subtracted
      * from the existing hours argument.
      * It is possible for overflow or underflow to occur in the addition or
      * subtraction operation. In many cases this will be acceptable and the
      * hours or seconds data item should also be altered to allow for the
      * overflow or underflow.
      * In the remaining cases of overflow or underflow, where the resulting time
      * is outside the range 00:00:00 to 24:00:00 no change should occur
      * in the hours, minutes or seconds data item and the valid flag should be
      * set to false.
      */

      void  AddSeconds(int seconds);      // Add seconds (which may be < 0).

   Private:
      /* the data items storing the hours, minutes and seconds or the seconds
      * since midnight go here. Any private functions you write will also go
      * in this section.
      */
};
```

To get a start on Assignment 1

Time.h

```
class Time
{
   public:
         // Constructors and destructor
      Time();                       // Default constructor
      Time(Time const& time);       // Copy constructor
      Time(long secondsAfterMidnight);
      Time(char const* tstring);
             // String in hh:mm:ss format (24 hour time).
      ~Time();                      // Destructor
          // Const (read-only) functions
      char*  GetTime(bool military = false) const;
          // Return string representation of the time.
          //     If military then hh:mm:ss
          //             else hh:mm:ss am.
      int    GetHour() const;             // Get hour value.
      int    GetMinute() const;           // Get minute value.
      int    GetSecond() const;           // Get second value.
      bool   operator !() const;
               // Returns true if time is NOT valid
      bool   IsAM() const;                    // Is time AM?
      bool   operator ==(Time const& time) const;
                                           // Are times equal?
      Time   operator +(Time const& time) const;  // Add times.
      Time   operator -(Time const& time) const;  // Subtract times.
       // Non-const (read/write) functions
      void   SetTime(int hrs, int mins = 0, int secs = 0);
             // Set the time to the values supplied (in 24 hour format)
      void   AddHours(int hours);      // Add hours (which may be <0).
      void   AddMinutes(int minutes);  // Add minutes (which may be < 0).
      void   AddSeconds(int seconds);  // Add seconds (which may be < 0).
   private:
      long sfm;
      static const int SEC_IN_HOUR = 3600;
      static const int SEC_IN_MINUTE = 60;
};
```

Time.cpp

```
#include "Time.h"

Time::Time(long secondsAfterMidnight){
   sfm = secondsAfterMidnight;
}

Time::~Time(){}

int Time::GetHour()const{
   return sfm / SEC_IN_HOUR;
}
```

main.cpp

```
#include <cstdlib>
#include <iostream>
#include "Time.h"

using namespace std;

int main(int argc, char *argv[])
{
    int secs = -1;
    Time* time;
    while(secs != 0){
       cout << "Give me some seconds (0 to finish)";
       cin >> secs;
       time = new Time(secs);
       cout << "Hours = " << time-> GetHour() << endl;
       delete time;
    }
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Functions - General

There are several considerations that apply to C++ functions, whether they are:

Global functions (as in C), or

Member functions of a class

Overloaded Function Names

- Two or more functions, defined within the same scope, may have the same name, provided that:
  - the number and/or
  - the types

  of the arguments are different.

  (A difference of return types is NOT sufficient.)

```
int someFunct();
int someFunct(int);
int someFunct(double);
int someFunct(int, int);
int someFunct(int, double);
```

all represent different functions, but.

```
char someFunct();
```

is an illegal redefinition..

*(We will see later that, while function overloading is often **useful**, it is sometimes **necessary**)*

Default Parameters

- If there are a large number of parameters required by a function, it is often possible to provide default values for some or all of these.
  - If a default argument is specified for one parameter, defaults must be specified for all **following** parameters.
  - If, in a call to the function, a value is provided for an argument, values must be provided for all **preceding** arguments.
  - Default values must be specified in the function **declaration** (or **prototype**), NOT in the **definition**.

e.g.

```
void SetFont(int size, int weight = 400, bool italic = false,
         bool underline = false);
main(){
   SetFont(18); // Weight, italic, underline take default values
   SetFont(18, 600, true); // underline takes default value
   SetFont(18, 400, false, true); //must specify values for
              // weight, italic even though defaults required
```


Default Parameters and Overloading

- Care must be taken to avoid ambiguity, when overloading a function with default parameters.

```
int myFunct(int x, int y = 0);
int myFunct(int x);
```

is not a legal overloading of `myFunct()`.

The compiler could not determine whether

```
y = myFunct(10);
```

is

- a call to the first function with default value as the second argument, or
- a call to the second function.

Argument Passing

- In C and C++ the standard is "pass-by-value", e.g.

```
void Swap(int a, int b){
   int c = a; a = b; b = c;
}
main(){
   int x, y;
   Swap(x, y); // x, y unaltered by function
}
```

- Pass by pointer – common in C, e.g.

```
void Swap(int* a, int* b){
   int c = *a; *a = *b; *b = c;
}
main(){
   int x, y;
   Swap(&x, &y); // x, y have been swapped
}
```


Argument Passing – pass by reference

- Pass by reference is new to C++

```
void Swap(int & a, int & b){
   int c = a; a = b; b = c;
}
main(){
   int x, y;
   Swap(x, y); // x, y have been swapped
}
```

Unspecified Arguments

- There are (admittedly rare) occasions when we wish to have a function, some of whose parameters are of
  - unspecified number and/or type.
- The most common examples of this are the library functions
  - printf(), scanf() and their variants.
- Such a function is declared by following the list of known parameters with an *ellipsis* (…), e.g.

```
int Add(int number, ...)or
int Add(int number ...)    // the comma is optional
```

- The unspecified parameters must follow all known parameters, of which there must be at least one. (Some Unix compilers allow none)
- The problem is
  How to reference these unspecified parameters in the body of the implementation.

Unspecified Arguments

- The following example function returns the sum of an unspecified number of integer arguments. The first parameter is the number of values to be added.

```
#include <stdarg.h>
int add(int number, ...){
   int total = 0;
   int next;
   va_list nxtArg; // Must declare a variable of type va_list
                   // Must now initialise this variable, giving
                   //    the name of the last specified parameter
   va_start(nxtArg, number);
   for(int i = 0; i < number; i++){
      next = va_arg(nxtArg, int);
                   // Returns the next argument
                   //   which I specify to be an int
      total += next;
   }
   va_end(nxtArg); // must close va_list variable
   return total;
}
```

Constant Arguments

- When a pointer or a reference to an object is passes to a function,
  without any intention of modifying it,
  declaring it to be const will ensure that the object is not accidentally altered by the function.

```
int SomeFunct(const char* array, const int& b, float val){
   array[3] = 's';           // ILLEGAL
   b += 5;                   // ILLEGAL
   val = (float) b;          // OK
   return array[b + 5];      // OK
}
```

Member Functions

There are considerations that apply only to member functions of a class

Some types of functions can only be used as member functions

Member Function Declaration

- Member functions are declared as part of a class declaration – conventionally in a .h file. The declaration includes:
  - The name of the function.
  - The return type.
  - Types of all the parameters to the function (names optional).
  - Default values for any of the parameters.
  - Pointer or reference parameters may be specified as **`const.`**
  - A pointer or reference return value may be specified as **`const.`**
  - The function itself may be declared as **`const.`**

Member Function Definition

- Functions are conventionally defined in a .cpp file. The definition includes:
  - The function name qualified by the name of the class of which it is a member

    e.g. `MyClass::FunctionName(parameters);`
  - Return type.
  - Types of all parameters. Names are required so that they may be referenced in the body.

  - Default values for parameters **are not repeated** in the definition.
  - If the function itself, or any of the parameters or the return type are specified as **`const`** in the declaration, this specification **must be repeated** in the definition.

  - Statements to implement the function.

Invoking Global Function

- A global function – one that is not a member function of any class – may be invoked by preceding its name by two colons (::).

- If no other function with the same name is in scope, the colons may be omitted

Invoking Global Function- Example

```
int Increase(int amount){
   return (amount + 1);
}
char ToUpper(char charIn){
   if('a' <= charIn && charIn <= 'z')
         return (charIn – 'a' + 'A');
   return charIn;
}
class Person{
   public:
         void Increase(int howMuch){
            for(int i = 0; i < howMuch; i++)
               salary = ::Increase(salary); // Want global, not local
         }
         void CapitaliseName(){
            name[0] = ToUpper(name[0]);    // No local with this name
         }
   private:
         char* name, int salary;
}
```


Invoking Member Function from Another Member Function

- A call from a member function to a function without any qualification will be a call to a member function of the same class. (if there is no member with that name, a global function will be invoked.)
- References to data members will be to the same set of data members accessable by the calling function.

```
class BankAccount{
   public:
         void EndOfMonth(){
                AddInterest();
                PrintStatement();
         }
   private:
         void PrintStatement(){ . . . }
         void AddInterest(){
                balance += balance * interestRate/100/12;
         }
         float balance, interestRate;
}
```

Invoking Member Function from Rest of the Application

- Given the name of, or a reference to, an object of the class within which the function is defined, the class object selector (.) is use to reference the function. E.g.
  - `Person personA;`
  - `int x = personA.GetAge();`
- Given a pointer to an object of the same class, the class pointer selector (->) is used
  - `Person* pPtr = new Person;`
  - `int x = pPtr->GetAge();`
- Any references within the function to

  data members of the class

  are applied to

  the set of data members associated with the object to which the function is applied.

Referencing Data Members

- Member functions of a class have unrestricted access to
  - public, protected *or* private

  data members of
  - the object to which the function is being applied, *or*
  - any other object of the same class to which it has access

```
class Person{
     public:
                    Person(char* nameIn, int ageIn);
            bool    IsOlder(Person& other);
     private:
            char*   name;
            int age;
};
bool Person::IsOlder(Person& other){
     return age > other.age
            // OK to access private data member of this
            //   Person and also of other Person
}
```

In-Line Member Functions

```
class Person{
   public:
        // an in-line function
      void  SetAge(int newAge){age = newAge;};
      int   GetAge();
   private:
      int   age;
};

     // an alternative specification of an in-line  function
     //    - this must be in the class declaration (.h) file
inline int Person::GetAge(){
   return age;
}
```


In-Line Member Functions

- Small, straight-line functions may be made **`inline`** to save the overhead of a function call.
- **`inline`** is only a recommendation to the compiler. It will be ignored if the function is too big or too complex (Compiler-dependent).
- The body of an **`inline`** function must be in the header (.h) file.
- Note that if a previously **`inline`** function is made non- **`inline`**, or vice-versa, all programs using this class must be re-compiled.

`const` Functions

- Declaring a member function to be **`const`** ensures that the data members of the object to which it is applied will not be altered.

```
class Person{
   public:
         Person();
      void SetParent(const Person* p);
            // p can't be altered by function
      Person* GetParent()const;
            // data members can't be altered by function
   private:
      Person* parent;
};
Person::Person(){
      parent = 0;
}
void Person::SetParent(const Person* p){
      parent = p;
}
Person* Person::GetParent()const{
      return parent;
}
```

- Note that **`const`,** for the function or parameters, **must be repeated** in the implementation.

Constructors and Destructor

These member functions are of critical importance in
Object-Oriented Programming

Constructor

- A constructor is a special type of class member function
  - Its name must be identical to the name of the class
  - NO return type can be specified – not even **`void`**.
  - Overloading is allowed. There can be, and usually are, many constructors – distinguished by their argument lists.
  - The constructor is never called explicitly.
- The purpose of a constructor is to initialise the data members of an object. When an object is created:
  - The system allocates sufficient memory for the data members.
  - The constructor is called automatically.
  - The address of the object is returned

Constructor (cont'd)

- The constructor is executed:
  - Global objects
    - Prior to execution commencing at `main().`
    - In the order of their declarations.
  - Local objects
    - When their declarations are encountered within a block of code.
  - Dynamic objects
    - During the execution of the operator **`new`**.

Destructor

- A destructor is another type of member function.
  - Its name is the name of the class, preceded by tilde (**~**)**.**
  - NO return type can be specified – not even **`void`**.
  - NO arguments are allowed for the destructor, so it can not be overloaded.
  - The destructor is rarely called explicitly.
- The purpose of a destructor is to clean up before an object is destroyed. When an object is destroyed

  either via delete or because it is going out of scope.
  - The destructor is called, if it exists.
  - Memory used by data members is released.

Destructor (when executed)

- A destructor is executed:
  - Static Objects
    - During program termination.
  - Local Objects
    - When execution leaves the block within which they are declared.
  - Dynamic Objects
    - During execution of the **`delete`** operator.

A destructor is normally only required
if some data members are pointers to objects which are not required
when this object is destroyed.

Constructor & Destructor Declaration

```
#ifndef PERSON_H
#define PERSON_H

class Person{
   public:
      Person();                          // Default constructor
      Person(const Person& other);       // Copy constructor
      Person(char* nameIn, int ageIn = 0)// Parameterised constr.
      ~Person();                         // Destructor
   private:
      char* name;
      int   age;
};

#endif
```

Constructor & Destructor Implementation

```
#include "Person.h"
#include <cstring>
using namespace std;
Person::Person(){                          // Default constructor
   name = 0;
   age = 0;
}
Person::Person(const Person& other){    // Copy constructor
   name = new char[strlen(other.name) + 1];
   strcpy(name, other.name);
   age = other.age;
}
Person::Person(char*nameIn, int ageIn){ // Parameterised constr.
   name = new char[strlen(nameIn) + 1];
   strcpy(name, nameIn);
   age = ageIn;
}
Person::~Person(){                         // Destructor
   delete [] name;
}
```


Using Constructor & Destructor

```
#include "Person.h"
Person archie("Archie");          // Parameterised constr. called
main(void){
   Person* pPtr1 = new Person("Fred");// Parameterised constr.
   {
      Person betty("Betty", 22);  // Parameterised constr. called
      Person anon;                // Default constructor called
      delete pPtr1;               // Destructor called for Fred
      Person betty2(betty);       // Copy constructor called
      Person betty3 = betty;      // Copy constructor called again
      Person* pPtr2 = new Person("Mary"); // Parameterised constr.
   }        // Destructor called for betty, anon, betty2, betty3
            // Pointer pPtr2 destroyed, but Mary remains as a
            //   memory leakage.
}           // Destructor called for archie
```

Constructor – problems (1)

```
// PersonDate.h
class Date{
   public:
      Date(int d, int m, int y){
         day = d; month = m; year = y;
      {
   private:
      int day, month, year;
};
class Person{
   public:
      Person(char* nameIn, long mCare, int daB, int moB, int yrB);
      const long MedicareNo;
   private:
      char* name;
      Date dateOfBirth;
};
```


Constructor – problems (2)

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

#include "PersonDate.h"
Person::Person(char* nameIn, long mCare, int daB, int moB, int yrB){
   name = new char[strlen(nameIn) + 1];
   strcpy(name, nameIn);
   MedicareNo = mCare;     // ILLEGAL MedicareNo is const
                           // What about dateOfBirth?
}
```

- This won't work
  - How can we give a value to a `const` data member?
  - How to create an object which is a data member?

Constructor – Initialisation List

```
#include <cstring>
using namespace std;
#include "PersonDate"
Person::Person(char* nameIn, long mCare, int daB, int moB, int yrB)
         // after colon comes initialisation list
      :  MedicareNo(mCare),         // initialise const data member
         dateOfBirth(daB, moB, yrB) // initialise member object
{
   name = new char[strlen(nameIn) + 1];
   strcpy(name, nameIn);
}
```

- Use initialisation list for:
  - Assigning a value to a `const` data member
  - Passing parameters to constructor for a data member which is an object of a class
  - Assigning values to simple data members (optional)
  - Passing arguments to parent constructor (covered later)

Automatic Copy Constructor

- If no **copy** constructor is provided and an object is to be initialised with another of the class, *memberwise initialisation* occurs.
  - A bit-wise copy of the supplied data members is made to the new object.

- This is fine *except* where a data member is a pointer.
  - If we had not supplied a copy constructor for the Person class, when we created another Person equal to an existing one, name would point to the same **char** array.
  - Changing the name of one object would change the name of the other.
  - Worse is that if one object is deleted, the destructor deletes the name array, leaving the other object with an invalid name pointer.

Automatic Default Constructor

- If *no constructors at all* are provided for a class, then a do-nothing **default constructor** is provided.
- If any constructors are provided but no default constructor is provided for a class, then parameters appropriate for some other constructor **must be provided** when an object of the class is created.

`this` pointer

the address of the *object* to
which a *member function* is being applied

`this` pointer

- Within a member function, the key-word this represents the *address of the object* to which the function is being applied.

```
class Employee {
    public:
        void AddHoursWorked(float ordT, float overT);
    private:
        float overTime, ordinaryTime;
};
    // The usual implementation
void Employee::AddHoursWorked(float ordT, float overT){
    overTime += overT;
    ordinaryTime += ordT;
}
    // An exactly equivalent implementation
void Employee::AddHoursWorked(float ordinaryTime,
                              float overTime ){
    this->overTime += overTime ;
    this->ordinaryTime += ordinaryTime ;
}
```

Returning `this`

- It is often useful to return `this` or `*this` from a function that would otherwise return void.
  Allows successive calls to different member functions for the same object, within the same expression.

```
class Employee {
   public:
      Employee&  AddHours(float ordT, float overT);
      float GetOverTime(){return overTime;};
   private:
      float overTime;
      float ordinaryTime;
};
Employee& Employee::AddHours(float ordT, float overT){
   overTime += overT;
   ordinaryTime += ordT;
   return *this;
}
main(){
   Employee  peter (. . .);
   float y = peter.AddHours(8, 6.3).GetOverTime();
}
```


Guarding against self-reference (1)

- It is sometimes essential to determine whether an object being passed as a parameter to a function is the same as the object to which the function is being applied.

```
class Person{
     public:
         Person(char* name, char* addr = 0);
         void SetAddrSameAs(Person& other);
     private:
         char* name;
         char* address;
};
 void Person::SetAddrSameAs(Person& other){
     delete [] address;
     address = new char[strlen(other.address)+ 1];
     strcpy(address, other.address);
 }
```

Guarding against self-reference (2)

This will cause havoc if it is used as follows

```
main(){
    Person* family[familySize];
    family[0] = new Person("Amy");
    family[1] = new Person("Brian");
    family[2] = new Person("Cecily", "Lot 7, WeeWaa");
    family[3] = new Person("David");
     . . . . . .
    for(int i = 0; i < familySize; i++)
        family[i]->SetAddrSameAs(*family[2]);
    . . . . . .
}
```

Why?

Guarding against self-reference (3)

- In the previous example, in the third iteration, we would be deleting the `address` from `family[2]` and then copying the `address` from `family[2]` to `family[2]` (and subsequently to the other array members).

  To correct this problem the function must be changed to

```
void Person::SetAddrSameAs(Person& other){
    if(this != &other){
        delete [] address;
        address = new char[strlen(other.address) + 1];
        strcpy(address, other.address);
    }
}
```

`static` Data Members & Member Functions

Data and Functions that belong to
a class as a whole
rather than to objects of the class

`static` Data Members

- There are times when data items are required, which relate to a class as a whole, rather than to an individual object of that class.
- Global data items could be used but these have two disadvantages:
  - Name space pollution
    - Care would need to be taken that the names chosen do not correspond to any global names chosen elsewhere in the application.
  - Protection
    - The only protection available to global data items is to make them `const`, but this is not always appropriate.
- The solution is to declare these as `static` data members of the class

`static` Data Members - Declaration

- Declaration
  - Static data members are declared as for other data members of a class.
    - the declaration is preceded by **static**.
  - Access level
    - static data members can be **public**, **protected** or **private**.
  - Constant static data members
    - static data members can be **const**

`static` Data Members – Decl'n Example

```
class someClass{
    public:
        int a;
        static double w;
        const static int x;
    protected:
        static char* y;
        char b;
    private:
        static const int z;
        char* c;
};
```

- Every object of class `someClass` will have **its own** data items *`a`*, *`b`* and *`c`*.
- **One** of each data item *`w`*, *`x`*, *`y`* and *`z`* will exist regardless of how many (including zero) objects of class `someClass` exist

Allocation of Space

- A class declaration does not allocate any space for any data items.
  - For non-`static` data members space is allocated
    - Global objects
      - By the compiler for global objects of the class
    - Local objects
      - When a declaration of an object is encountered within a code block.
    - Dynamic objects
      - Upon execution of `new`.
  - For **`static`** data items, space is allocated by means of a *global definition*.
    - the name of the data item must be qualified by the class if which it is a member.
  - e.g.

```
double someClass::w;
char*  someClass::y;
```

Note that the word `static` is NOT repeated in the definition

Initialisation of `static` Data Members

- After allocation of memory for the object (or before execution of main for global objects) a *constructor* is executed to enable the initialisation of non-`static` data members.
  - This is inappropriate for `static` data members since:
    - Repeated initialisation of the same members every time an object is created is probably not what is desired.
    - If a `static` member is accessed before any objects have been created, it would be uninitialised.
- `static` data members could be initialised by:
  - assigning values to them prior to the start of `main()`
    - possible for `public` non-`const` members only
  - calling a member function from main() to which initial values could be passed as parameters
    - possible for non-`const` members only
- Almost always static data members should be initialised by the declarations that allocate space for them. e.g.

```
const double someClass::e = 2.71828182845;
```

Initialisation/Declaration of `static` in .cpp file

- Statements allocating space for and assigning initial values to static members
  - Should NOT be placed in the file containing the class declaration (.h file)
    - This file must be #included with every program file referencing this class.
    - In a large project this is likely to lead to multiple declarations and definitions for these items.
  - They should be included in the (.cpp) file containing the definitions of the member functions of the class.
    - The object code from this file will be *linked* to the object code from other modules of the project.

Referencing `static` Data Members

- Within a member function of the class
  - **public**, **protected** or **private static** data members are referenced
    - without qualification just as are non-**static** members.
- In other parts of the application
  - **public** data members may be referenced in one of two ways:
    - As if they were data members of an object of the class
    - By qualifying the name with the name of the class

Referencing `static` Data Members - Example

```
                        /********* circle.h *********/
class Circle{
   public:
      Circle(float x, float y, float r): cX(x), cY(y), rad(r){};
      static const float pi;
      double Circum();
   private:
      float cX, cY, rad;
};
                        /******** circle.cpp ********/
#include "circle.h"
const float Circle::pi = 3.14159;
double Circle::Circum()
   {return pi * rad * 2;}     // pi in member function
                        /********* main.cpp *********/
#include "circle.h"
main(){
   Circle c(5.6, 7.3, 3.6);
   Circle* pc = new Circle(6, 8, 4.4);
   float x, y, z;
   x = c.pi;           // pi as a member of Circle object c
   y = pc->pi;         // pi as a member of *pc
   z = Circle::pi;     // pi as a member of Circle class
}
```


`static` Member Functions

- There are occasions when we wish to call a member function, even though no objects of the class may exist.

  This is clearly only sensible for a function that *does not* access *non-***static** data members.

  Such a function is a *Static Member Function*.

**** employee.h ****

```
class Employee{
    public:
        Employee();
        ~Employee();
        static int GetPop();
    private:
        static int pop;
};
```

**** employee.cpp ****

```
#include "employee.h"
int Employee:: pop = 0;
int Employee::GetPop(){
   return pop;
}
Employee::Employee(){pop ++;}
Employee::~Employee(){pop--;}
```

**** main.cpp ****

```
#include "employee.h"
main(){
    Employee Fred;
    int count1 =
      Fred.GetPop();
    int count2 =
      Employee::GetPop();
}
```

`static` Member Functions Rules

- Static member functions may NOT reference instance (non-static) data members or member functions.

- May be referenced :
  - via an object of the class, or
  - by using the function name qualified by the name of the class to which they belong.

Operator Overloading

The use of Arithmetic, etc. operators
with user-defined types

Operator Overloading

- Most programming languages provide arithmetical, logical and other operators which can take objects of the **built-in types** as operands.

- Some languages extend these operands to operate on **composite types** such as arrays and strings to provide e.g. matrix addition and string concatenation.

- C++ does not provide these extensions but allows the user to extend the use of operators to **user defined types** (i.e. classes).

Operaror Overloading - Restrictions

- Only the **predefined set** of C++ operators may be overloaded
  - We can NOT use ** to express exponentiation.

- The four operators **:: .* . ?:** may **not be overloaded**

- At least **one of the operands** for the overloaded operator must be an **object of a class**
  - We can NOT give a new meaning to the addition of two integers.

- The overloaded operators will have the same **precedence** as the built in equivalents
  - In the expression **a = b + c * d;**
    the operation * will be performed first, followed by **+** and then **=**.
    This is true whatever types of things **a**, **b**, **c** and **d** are.

- A **binary** operator (one taking two operands) can not be redefined to be **unary** (one taking one operand), and vice versa.
  - However the four operators which are predefined to be unary or binary , i.e.
    + - * &
    Can be redefined to be unary, binary or both.

Operator Overloading - member function

- An overloaded operator may be declared and defined as a member function of some class.
- An object of the class is assumed to be the Left Hand (or, in the case of a unary operator, the only) operand.
- The function parameter (if any) is the Right Hand operand.

Overloading Binary Operator

```
class Employee {
    public:
        Employee  operator+=(float hours);
    private:
        float hoursWorked;
};
void Employee::operator+=(float hours){
    hoursWorked += hours;
}
main(){
    Employee peter(. . .);
    peter += 3.5;
        // or equivalently
    peter.operator+=(3.5);
}
```

Overloading Unary Operator

```
class Person{
     public:
               Person(char* nameIn = 0, int ageIn = 0);
          bool  operator!();
     private:
          char* name;
          int   age;
};

bool Person::operator!(){
     return age <= 0;
}

void main(){
     Person p;
     . . . .
     if(!p) cout << "p has not been initialised" << endl;
}
```

Inheritance

Specialising library classes
Extending concept of code reuse

The foundation of **polymorphism**

Inheritance

- The principal factor differentiating object-oriented languages from other modular languages (e.g. Modula, Ada)

- The basis for polymorphism

- allows the"Open-Closed Principle" (Bertrand Meyer)
  - a module is "open" if it may be extended, by adding data items and/or new functions.
  - a module is "closed" if it can be compiled and stored in a library for use by others, with no need to access the source code

- When an object of the derived class is created, space is allocated for:
  - the data items of the base class(es) and
  - the data items specified for the derived class

Types of Inheritance

- Inheritance may be
  - single - a class has one parent class, or
  - multiple - a class has more than one parent

- Inheritance can also be
  - `public` – all public data or functions of parent are public data or functions of child, or
  - non-public (`protected` or `private`)

- Public inheritance is an implementation of an *is-a* relationship between two classes.
  - Non-public inheritance implements a *uses-a* relationship which becomes more of an implementation detail rather than a fundamental design concept.

- By far the most important of these is single, public inheritance.

Single Public Inheritance

- For example: we have a class called Person and we require a class called Employee.
  - Every employee **is a** person.
  - All information that we hold for a person we wish to hold for an Employee.
    - e.g. `name, address`
    - *but there is additional information that we require for an Employee*
      - e.g. `employeeNumber, hourlyRate`
  - Every operation that we may perform on a person we also wish to perform on an Employee
    - e.g. `getName(), ChangeAddress()`
    - *but there are some additional operations applicable to an employee*
      - e.g. `AddHoursWorked(), CalcPay()`

- This is an obvious case where Employee should inherit **publicly** from Person

Constructors Under Inheritance

- When an object of a particular class is created:
  - a block of memory is allocated:
    - for the data members specified for this class *and*
    - for the data members of any ancestor classes.

  - The constructor for this class is called.
    - If this class is a derived class, its constructor first makes an implicit call to the constructor of the parent class.
      - If the constructor of the parent class requires arguments, these are specified in the initialisation list for this class.
    - The other items in the initialisation list for the constructor is executed.
      - Inherited data members should not be initialised in the initialisation list - only data members specified for this class.
    - The body of the constructor is executed.

Constructor Ordering

- This means that, if the class of the object being created is at a low level in an inheritance hierarchy:
  - The constructor for the class at the **top** of the hierarchy is executed **first**
  - Then the constructor for its immediate child class is executed next, etc.
  - The constructor for the class of the **object** is executed **last**

Public Inheritance – parent class

in "person.h"

```
#ifndef PERSON_H
#define PERSON_H
class Person{
  public:
    Person(char* nameIn, int
      ageIn);
    ~Person();
    void SetName(const char*
      newName);
    int GetAge()const
       {return age;}
    const char* GetName()const
       {return name;};
  private:
    int   age;
    char* name;
};
#endif
```

in "person.cpp"

```
#include "person.h"
Person::Person(char* nameIn,
   int ageIn) : age(ageIn){
   name = 0;
   SetName(nameIn);
}
Person::~Person(){
   delete [] name;
}
void  Person::SetName(const
        char* newName){
   delete [] name;
   name = new
       char[strlen(newName)+1];
   strcpy(name, newName);
}
```

Public Inheritance – child class

in "employee.h"

```
#ifndef EMPLOYEE_H
#define EMPLOYEE_H
#include "person.h"
class Employee : public Person{
   public:
    Employee(char* nameIn,int
      ageIn, long noIn);
    ~Employee();
    void  SetSalary(float
      newSal);
    float GetSalary()const;
    long  GetEmpNo()const;
    float CalcPay()const;
   private:
     long  empNo;
     float salary;
};
#endif
```

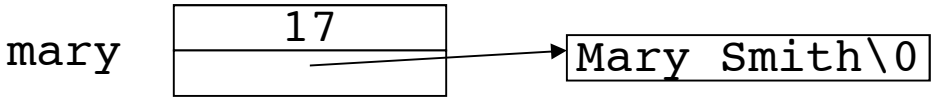
in "employee.cpp"

```
#include "employee.h"
Employee::Employee(char*nameIn,
   int ageIn, long noIn)
   : empNo(noIn), salary(0.0),
```

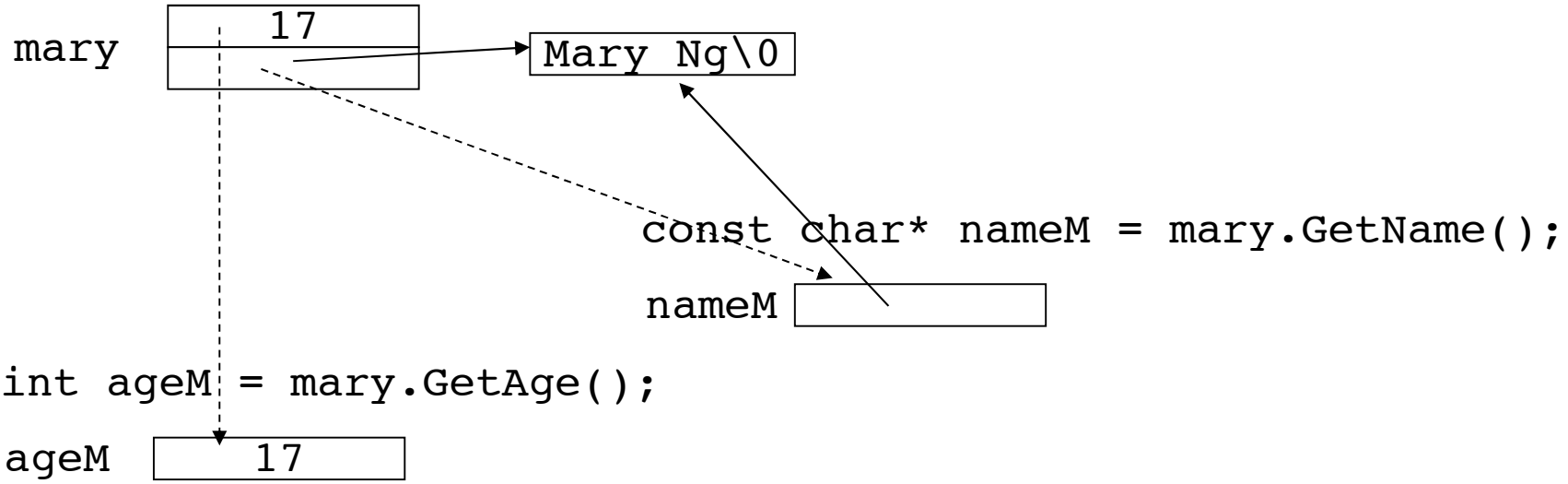
```
{}

void Employee::SetSalary(float
          newSal)
   {salary = newSal;};

float Employee::GetSalary()const
   {return salary;};

long Employee:: GetEmpNo()const
   {return empNo;};

float Employee::CalcPay()const
   {return salary/52;}
```

Parent Object

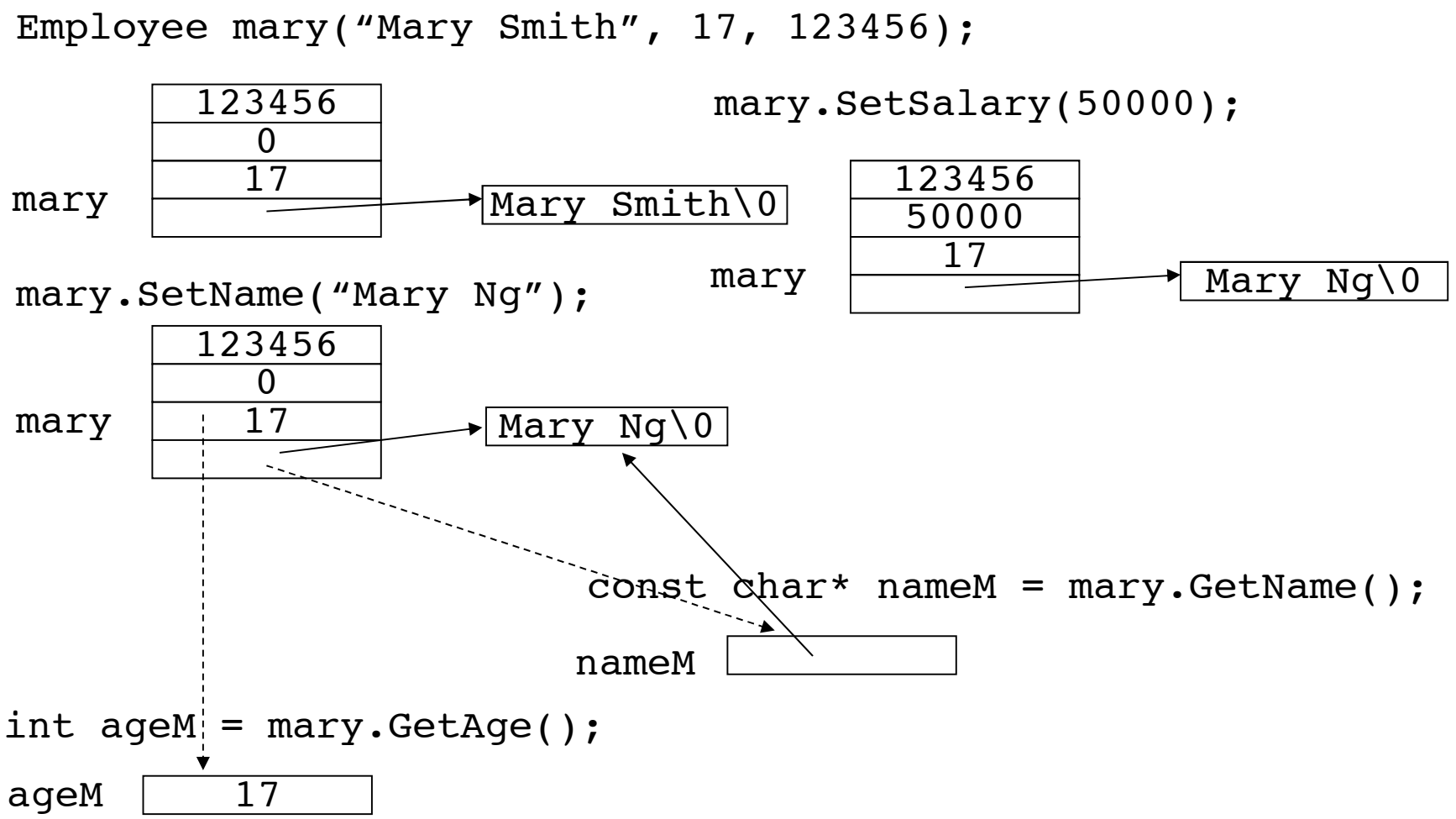
```
Person mary("Mary Smith", 17);
```




```
mary.SetName("Mary Ng");
```



Child Object



Public Inheritance

Every Person object has

- `name`
- `age`

To every Person object we can apply the member functions

- `SetName(char*)`
- `GetName()`
- `GetAge()`

Every Employee object has

- `name` `//inherited`
- `age` `//inherited`
- `empNo`
- `salary`

To every Employee object we can apply the member functions

- `SetName(char*)` `//inherited`
- `GetName()` `//inherited`
- `GetAge()` `//inherited`
- `GetEmpNo()`
- `SetSalary(float)`
- `CalcPay()`

Constructors Under Inheritance

- Arguments are passed to the constructor of the parent via the initialisation list, as in

```
Employee::Employee(char* nameIn,int ageIn,int noIn)
        : empNo(noIn), salary(0.0), Person(nameIn, ageIn)
{}
```

- If arguments for a constructor of the parent class are not given in the initialisation list of a class:
  - If the parent class has a default constructor (one taking no arguments), it is executed.
  - Otherwise if no constructors at all are specified for the parent class, a do-nothing constructor is executed.
  - Otherwise a compilation error occurs.

Constructors Under Inheritance = example

```
#include <stdlib.h>
#include <iostream.h>
class A{
    public:
       A(int x, int y) : a1(x), a2(y) {a3 = a2 + a1;};
    protected:
       int a1, a2, a3;
};
class B : public A{
    public:
       B(int p, int q, int r) : A(p, q), b1(5)  {b2 = a1 + r;};
    protected:
       int b1, b2;
};
class C : public B{
    public:
       C(int x, int y, int z) : B(x, y, z)  {c = b1 + a1;};
    `  Show(){
         cout <<"a1 = " <<a1 <<" a2 = "<< a2 <<" a3 = " << a3 <<
       endl;
         cout <<"b1 = " <<b1 <<"; b2 = "<< b2 << endl;
         cout << "c = " << c << endl;
       }
    protected:
       int c;
};

main(){
    C cThing(2, 3, 4);
    cThing.Show();
}
```

produces

```
a1 = 2; a2 = 3; a3 = 5
b1 = 5; b2 = 6
c = 7
```

Destructors Under Inheritance

- Since Employee inherits from Person:
  - All object of type **`Employee`** contain the data members:
  - **`age`** (inherited from Person)
  - **`name`** (inherited from Person)
  - **`empNo`**. (specific to Employee)
  - Since **`name`** is a **private** member of the **`Person`** (parent) class it is only accessible to member functions of the **`Person`** class.
  - A **destructor** is required for the **`Person`** class,
  - since the character array to which name points must be destroyed when
  - a **`Person`** is destroyed, *or*
  - an **`Employee`** is destroyed.

Order of Destruction

- When an object of a particular class is destroyed:
  - The destructor for this class is called.
    - The body of the destructor is executed.
    - If this class is a derived class, its destructor then makes an implicit call to the destructor of the parent class.
  - the block of memory allocated for the data members specified for this class and for any ancestor classes is then released.

- This means that, if the class of the object being created is at a low level in an inheritance hierarchy:
  - The destructor for the class of the **object** is executed **first**
  - Then the destructor for its immediate parent class is executed next, etc.
  - The destructor for the class at the **top** of the hierarchy is executed **last**

- If no destructor is defined for a class, any destructors up its inheritance tree will still be executed in order.

Order of Creation and Destruction

- The order of execution of
  - Constructors is DOWN the inheritance tree
  - Destructors is UP the inheritance tree

Constructors, Destructors - example

```
#include <stdlib.h>
#include <iostream.h>
class A{
   public:
      A() {cout << “A constructor”;};
      ~A() {cout << “A destructor”;};
};
class B : public A{
   public:
      B() {cout << “B constructor”;};
      ~B() {cout << “B destructor”;};
};
class C : public B{
   public:
      C() {cout << “C constructor”;};
      ~C() {cout << “C destructor”;};
};
```

Constructors, Destructors – order of execution

```
main(){
   C* cPtr;
   cout << endl << "Create C" << endl;
   cPtr = new C;
   {
      cout << endl << "entered inner block" << endl;
      C cThing;
      cout << endl << "leaving inner block" << endl;
   }
   cout << endl << "Delete C" << endl;
   delete cPtr;
}
```

produces

```
Create C
A constructor B constructor C constructor
entered inner block
A constructor B constructor C constructor
leaving inner block
C destructor B destructor A destructor
Delete C
C destructor B destructor A destructor
```

Visibility

of data members and member functions under public inheritance.

An object of a derived class contains all of the data members of all ancestor classes, but these may not be directly accessible.

Member access under Public Inheritance

- Member functions of the **parent** class have full access to
  - **public**, **protected** and **private** data members and member functions of the parent class

- Member functions of the **derived** class have full access to
  - **public** and **protected** data members and member functions of the *parent* (and *grandparent* etc) class, and
  - **public**, **protected** and **private** data members and member functions of the *derived* class

This applies to the data members of:

- The object to which the function is applied.
- Any object of the same class passed as a parameter to the function.
- Any object of the same class pointed to by either of the two preceding objects.

Data Name Hiding

- If a data member of the derived class has the same name as a data member of the parent class
  - every object of the derived class will have TWO data members with this name.
  - Member functions of the *parent* class will access the member defined in the *parent* class.
  - Member functions of the *derived* class will access the member defined in the *derived* class

  The derived class declaration *hides* the parent declaration from the derived class functions.

- If required, a derived class member function can explicitly access the hidden data member by qualifying the data name by the parent class name, e.g.

  ```
  parentClassName::dataMemberName
  ```

Data Name Hiding - example

```
class Person{
   public:
      Person( . . . ) { . . .};
      void  SetSSN(long SSN)
          {IDno = SSN;};
          // sets Person::IDno

   protected:
      long  IDno;
      . . . .
};
```

```
class Employee : public Person
   public:
     Employee(char* nm)
           :Person(nm){};
      void  SetEmpNo(int empNo)
              {IDno = empNo;};
         // sets Employee::IDno

      long  GetSSN()
         {return Person::IDno;}
          // explicit reference
          //   to Person::IDno

   protected:
      int   IDno;
          // hides Person::IDno
};
```

Pointers and Inheritance

- A variable that is declared as a pointer to a class (say *Class1*) may be assigned the address of:
  - An object of class *Class1*, or
  - An object of any class **publicly derived** from *Class1* (child class) or
  - An object of a class **publicly derived** from a class that is **publicly derived** from that *Class1* (grandchild class) , etc.

- The pointer can only be used to apply, to the object to which it points, the public member functions (or access any public data members)
  - of the class **for which the pointer is declared**
  - This **includes** member functions of any **ancestor** class.
  - This **does not include** data or function members of **derived** classes

Pointers and Inheritance - example

```
class Person{
   public:
           Person(..);
      const char* GetName();
   private:
        . . . . .
};

class Employee : public Person{
   public:
           Employee(..);
      void SetSalary(..);
   private:
      . . . . .
};
```

```
main(){
   Person*   perP;
   Employee* empP;
   Employee  fred(..);
   perP = &fred;
   empP = &fred; // OK an employee
                 //   IS A person
   const char* name =
      perP->GetName(); // OK GetName
         // is a member of Person
   const char* name =
      empP->GetName(); // OK GetName
         //  is an inherited member
         //  of Employee
   empP->SetSalary(..); // SetSalary
         //  IS a member of Employee
   perP->SetSalary(..);// NO!!
         //  SetSalary IS NOT a
         //  member of Person
}
```

Function Re-declaration

- If a class has two or more functions with the same name,
  - "Function Overloading" occurs provided that the parameter lists differ.

- If a member function of the derived class has the same name as a (*non-virtual*) member function of the parent class:
  - this provides a **re-declaration** of the function name in the base class.
  - this re-declaration **hides** all functions with this name in the parent class, no matter what parameter list this function has.
  - The function in the derived class can be overloaded in the derived class.

Function Re-declaration - example

```
class Person{
   public:
      Person( . . . ){ . . .};
      void SetID(long SSN)
           {IDno = SSN;};
      void SetID(char* SSN)
           {IDno = atol(SSN);};
       // function name SetID()
       //   overloaded
   protected:
      long  IDno;
      . . . .;
};
```

```
class Employee : public Person{
   public:
      Employee( . . . ){. . .};
      void  SetID(int empNo)
                {IDno = empNo;};
        // re-declaration hides
        //   both versions of
        //   Person::SetID()

      long  GetSSN()
         {return Person::IDno;}
   protected:
      int   IDno;
};
```

Using Re-declared Functions

```
main(){
   Person*   persPtr;
   Employee* empPtr;
   Employee  fred("Fred Bloggs");

   persPtr = &fred;
   empPtr  = &fred;

   persPtr->SetID(1234567890);
      // invokes
      //    Person::SetID(long)
      //    sets Person::IDno

   persPtr->SetID("1234567890");
      // invokes
      //   Person::SetID (char*)
      //    sets Person::IDno

   empPtr->SetID(12345);
      // invokes
      //    Employee::SetID(int)
      //    sets Employee::IDno

   empPtr->SetID("12345");
      // ILLEGAL - no such
      //   function
}
```

Virtual Functions
and
Polymorphism

Virtual Functions

With
- Function Overloading and
- Function Re-declaration,

the version of the function to be called is determined at
- Compile Time ("early binding")

But with
- **Virtual functions,**

the version of the function to be called is determined at
- Run Time ("late binding")

This provides us with **Polymorphism**

Polymorphism

- One of the most powerful mechanisms in Object Oriented programming.
- Allows major extensions to applications with minimal *changes* to existing code.
- An application may have to deal with objects of a number of classes which are similar but differ in details.

  The differences can be isolated in the member functions of the class.

  The rest of the application ignores these differences

Using Virtual Functions

- If, in the declaration of a class
  - a function declaration is preceded by the keyword **`virtual`**,
  - that function is a *virtual function*
- If, in a derived class a function is declared which
  - has the *same name, parameter list and return type* as a virtual function in an ancestor class
  - that function ***over-rides*** the virtual function in the parent class
  - the over-riding function is also a *virtual function*.
    - If the parameter list differs, the parent function is hidden (parent function is **re-declared**).
    - If only the return type differs, it is an error.
- A variable declared as a pointer to a base class may be assigned the address of an object of that class or of any of its descendant classes.

  If a **virtual function** is invoked via that pointer, the *appropriate function for the actual object being addressed* is called

Virtual Functions - example

```
class Employee : public Person{
   public:
            Employee(char* nm, int num):Person(nm), empNum(num){};
      void  SetRate(int rate){hrlyRate= rate;};
      virtual int CalcPay() {return hrlyRate * hoursWkd;};
   protected:
      int   hrlyRate, hoursWkd, empNum;
};
class Labourer : public Employee {
   public:
            Labourer(char* nm, int num):Employee (nm, num){};
       // inherits Employee::CalcPay
   protected:
};
class Salaried : public Employee {
   public:
            Salaried(char* nm, int num) :Employee(nm, num){};
      void  SetSal(int sal){salary = sal;};
      int   CalcPay(){return salary / 26;};
   protected:
      int   salary;
};
```


Using Virtual Functions - example

```
main(){
   Employee  mary("Mary", 1234);
   Labourer  nigel("Nigel", 2345);
   Salaried  irene("Irene", 3456);

   Person*   persPtr;
   Employee* empPtr;
   int p;

   persPtr = &mary;             // OK - an employee IS A Person
   p   = persPtr->CalcPay();    // NO - CalcPay is not a member of Person

   empPtr  = &mary;             // OK - mary is an Employee
   p   = empPtr->CalcPay();     // invoke Employee::CalcPay()

   empPtr  = &nigel;            // OK - nigel is an Employee
   p   = empPtr->CalcPay();     // Labourer inherits Employee::CalcPay()

   empPtr  = & irene;           // OK - irene is an Employee
   p   = empPtr->CalcPay();     // CalcPay() over-ridden in Salaried
                                //      so invoke Salaried::CalcPay()
}
```

Comments on example

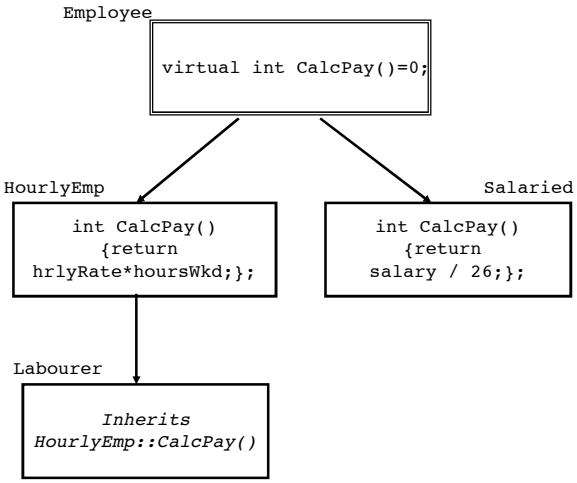
- The previous example was not entirely satisfactory:
  - Objects of class `Salaried` would have `hrlyRate` and `hoursWkd` data members although these are irrelevant for this class.
- A better design would be to have in the class `Employee`:
  - Only those data members that are relevant to ALL classes of employees.
    - *(It is likely that this set will NOT include ALL the data members needed by ANY class of employee.)*
  - Those member functions that are relevant to all employees and reference only the common set of data members.
  - **virtual** member functions for those operations that are required for all (or most) classes of employees, which may reference non-inherited data members.
  - If it is not appropriate to provide an implementation for the top-level of a virtual function:
    - Then this should be declared as a **Pure Virtual** function

Virtual Functions – example (revised)

```
class Employee : public Person{
    public:
        Employee(char* nm, int num):Person(nm),
                                empNo(num){...};
        virtual int  CalcPay()=0;
    // ***** pure virtual funct *****
    protected:
        int   empNo;
};
class HourlyEmp : public Employee{
    public:
        HourlyEmp(char* nm, int num)
                                :Employee(nm, num){};
        void  SetRate(int rate){hrlyRate= rate;};
        virtual int  CalcPay()
                {return hrlyRate * hoursWkd;};
    // ***** over-ride Employee::CalcPay() *****
     protected:
          int   hrlyRate, hoursWkd;
};
```

```
class Labourer : public HourlyEmp {
    public:
        Labourer(char* nm, int num)
                                HourlyEmp(nm, num){};
  // ***** Inherits HourlyEmp::CalcPay() *****
    protected:
};

class Salaried : public Employee {
    public:
        Salaried(char* nm, int num):
                                Employee(nm, num){};
         void  SetSal(int sal){salary = sal;};
        int   CalcPay(){return salary / 26;};
 // ***** over-ride Employee::CalcPay() *****
    protected:
        int   salary;
};
```

Revised Example



Using revised example

```
main(){
   Employee* staff[STAFF_SIZE];
   staff[0] = new HourlyEmp("Affris", 1234);
   staff[1] = new HourlyEmp("Barnes", 2345);
   staff[2] = new Salaried ("Chu",    3456);
   staff[3] = new HourlyEmp("Dana",   4567);
   staff[4] = new Labourer ("Eagles", 5678);
   . . . . . .
   for(int i = 0; i < STAFF_SIZE; i++){
        cout << staff[i]->Name() << " $" << staff[i]->CalcPay() << endl;
   }
```

- Which version of `CalcPay()` is executed on each time through the loop depends on the class of the object encountered
  - is determined at execution time.

- If a new class of employee is required:
  The only ***change*** to the existing application will be at the point at which a new employee is being added to the data base.
  New implementations of some or all of the virtual functions will be required, but these are ***additions*** to the application - not ***changes***

Abstract Class

- In the previous example, the base class Employee,
  - had no data fields that allow any sensible implementation of **`CalcPay().`**
- Since any Employee of whatever class must be paid,
  - it does not make sense to declare an object of class Employee.
- To declare unequivocally that **`CalcPay()`** will not be implemented for the Employee class, we make it a "pure virtual" function.
  - This is done by placing

    `=0`
  - between the closing parenthesis of the parameter list and the terminating semi-colon.
- A class containing one or more pure virtual functions is an "abstract class".
  - An object can not be created of an abstract class.
  - An abstract class exists only to be inherited from.
- If the parent of a class is an Abstract Class,
  - this class will also be an Abstract Class unless it over-rides **ALL** the Pure Virtual functions of its parent.

Function Relevant to few Sub-Classes

It will sometimes happen that there is an operation (say *RareFunct()*)that must be performed on only one (or very few) of the subclasses in an inheritance group. How to handle this?

- a) Declare

  ```
  virtual retType RareFunct(){};
  ```

  (i.e. a do-nothing function) as a member of the base class. Over-ride it in the class where it is relevant.

  Call this function regardless of the (sub)class of object.

  This is reasonably elegant and efficient, unless we wish to apply this function to ALL relevant objects and these make up a small fraction of the related objects.

- b) If the inefficiencies of a) are significant, the constructor for the unusual class could insert the object in a Linked List of objects of this class.

  The added complexity may be justified in the name of efficiency, in certain applications.

- c) If the base class is in a class library, or for some other reason it is not appropriate to add a virtual function to the base class, we may use a `dynamic_cast`.

  This approach is less efficient than approach a) above but may be necessary where the source code of the base class is unavailable.

Run-Time Type Identification (RTTI)

- RTTI allows retrieval of the actual type of an object referred to by a pointer (or a reference) to a base class.
  - Only "polymorphic classes" can use RTTI
    - A "polymorphic class" is one containing (or inheriting) at least one **`virtual`** function.

- Two operators are provided for RTTI
  - **`dynamic_cast`** operator allows the safe conversion of a pointer to a base class to a pointer to a derived class.
  - **`typeid`** operator allows the determination of the exact type of an object referred to by a reference or pointer to a base class

The use of RTTI should be minimised.

Polymorphism as provided by virtual functions is a more efficient and less error-prone solution in most circumstances.

RTTI – `dynamic_cast`

- dynamic_cast converts a pointer to a base class to a pointer to a derived class.
  dynamic_cast returns NULL if the object referenced by the base pointer is not of the desired class

```
class Employee : public Person{
   public:
      virtual int  CalcPay()=0;//virtual funct.
// Employee and all its derived classes are polymorphic classes
};
class HourlyEmp : public Employee{
   public:
      void  HourlyFunct(. . . );
};
class Salaried : public Employee {
   public:
      void  SalariedFunct(. . . );
};

main(){
   Employee*  emp;
   HourlyEmp* hrly;
   Salaried*  sal;

   if(hrly = dynamic_cast<HourlyEmp*>(emp))  hrly->HourlyFunct(. . . );
   else  if(sal = dynamic_cast<Salaried*>(emp))sal->SalariedFunct(. . . );
   else error();

}
```

RTTI – `typeid`

- `typeid` operator applied to an expression returns a reference to the object of type
  `const typeinfo`
  which is appropriate to the type of the expression

- The principal member function of the `typeinfo` class is
  `const char* name() const;`
  which returns the name of the type or class.

  Operators `==` and `!=` are overloaded for the `typeinfo` class.

RTTI – `typeinfo`

```
#include <typeinfo.h>

main(){
  Employee*  emp1;
  Employee*  emp2;
  emp1 = new Hourly(. . . );
  emp2 = new Salaried(. . . );
  if(typeid(*emp1) == typeid(*emp2))cout<<"Types are the same\n";
  else cout << "Types are different\n";
  cout << "*emp1 is a " << typeid(*emp1).name() << endl;
  cout << "*emp2 is a " << typeid(*emp2).name() << endl;
}

produces the output

        Types are different
        *emp1 is a Hourly
        *emp2 is a Salaried
```

Trouble with Destructor

Consider the following case

```
class Person{
   public:
      Person(char* nm){...};
      ~Person(){delete [] name;};
   protected:
      char* name;
};
class CarOwner: public Person{
   public:
      CarOwner(char* nm, char* make) :Person(nm){...};
      ~CarOwner(){delete [] car;};
   protected:
      char* car;
};
main(){
   CarOwner* pers1 = new CarOwner("John", "VW");
   Person*   pers2 = new CarOwner("Mary", "MG");
   delete pers1;    // calls ~CarOwner() which deletes "VW
                    //   and then ~Person() which deletes "John"
   delete pers2;    // calls ~Person() which deletes "Mary",
                    //   "MG" remains as a memory leakage
```


Virtual Destructor

- If an object of a derived class is destroyed via a **pointer to the derived class**, the destructor of the derived class is called, and then the parent class destructor is called.
- If an object of a derived class is destroyed via a **pointer to a parent class**, the destructor of the derived class is not called.
- To overcome this, the **destructor of the parent** class must be **virtual**.
- Virtual destructors are a special case of virtual functions
- If the destructor of a base class is virtual, the destructor of a derived class will override the base class destructor **even though its name is different**

Using Virtual Destructor

```
class Person{
      public:
           Person(char* nm){...};
            virtual ~Person(){delete [] name;};        // virtual destructor
      protected:
           char* name;
};
class CarOwner: public Person{
       public:
           CarOwner(char* nm, char* make):Person(nm){...};
           ~CarOwner(){delete [] car;};                // over-rides ~Person and is also virtual
      protected:
           char* car;
};

main(){
 CarOwner* pers1 = new CarOwner("John", "VW");
 Person*   pers2 = new CarOwner("Mary", "MG");
 delete pers1; // calls ~CarOwner() which deletes "VW"  and then ~Person() which deletes "John"
 delete pers2; // calls ~CarOwner() which deletes "MG" and then ~Person() which deletes "Mary"
}
```


Conversions

Converting an object of one class
to an object of another class

Standard Conversions under Inheritance

- In any context where
  - an *object* of a base class, or
  - a *reference* to a base class, or
  - a *pointer* to a base class

  is required, a corresponding item of a publicly derived class can be used.

- If we wish to use a pointer (or a reference) to a base class where a pointer (or a reference) to a derived class is required,
  - This is an **inherently dangerous operation**, so the pointer must be explicitly cast to a pointer to the derived class
  - Unpredictable results will occur if we use the pointer to apply a derived class function, and the object is not of that derived class.

Using Standard conversions

```
class Person{...};
class Employee : public Person{...};

fun1(Person){...}
fun2(Person&){...}
fun3(Person*){...}
fun4(Employee&){...}
fun5(Employee*){...}

main{
   Employee joe(...);
   Person mary(...);

   fun1(joe);          // OK
   fun2(joe);          // OK
   fun3(&joe);         // OK

   fun4(mary);         // NO!!!
   fun4(&mary);        // NO!!!
}
```

Implicit Construction

- If
  - a function requires, as a parameter, an object of a class, and
  - that class has a constructor with ONE parameter

  that function can be called with an object of the type of that parameter.

  The constructor will be called to create an object of the type and this will then be passed to the function.

Using Implicit Construction

```
class Person{
    public:
        Person(char* nm, int sal = 0); // Requires only 1 parameter
    private:
        char* name;
        int   salary;
};
Person::Person(char* nm, int sal){
    name = new char[strlen(nm) + 1];
    strcpy(name, nm);
    salary = sal;
}

funct(Person){...}

main(){
    funct("Joe Blow");        // A Person will be constructed and
                              //   passed to funct
}
```

Conversion Operator

- If we wish to convert an object of a class to some other type of object, we can state how that conversion is to be done.

  For example if we wish to convert a Person to an integer:

```
class Person{
    public:
        Person(char* nm, int sal = 0);
        operator int(){return salary;};
               // Note NO parameters, NO return type
    private:
        char* name;
        int   salary;
};
```


Using Conversion Operator

```
main(){
    Person fred("Fred", 1000);
    Person mary("Mary", 5000);
    int x = static_cast<int>(fred)  +  static_cast<int>(mary);
        // conversion called for explicitly. x now has value 6000
    int y = fred + mary;
        // conversion done implicitly.  y also now has value 6000
}
```

Syntax of Conversion Operators

- The general syntax of the conversion operator definition is:
  - `operator type();`

  The conversion function must:
  - be a member function (NOT a global function),
  - have NO return type,
  - have NO parameters.

  The *type* may be:
  - any predefined type such as **`int, char, float, unsigned int`** etc.,
  - the name of any class.

- The conversion operator may be invoked:
  - explicitly using the cast notation, e.g.
    - `static_cast<int>(Fred)`
  - or implicitly when
    - an object of one class is provided, but
    - an object of the other class or type is required

Type Conversion in C

In C the syntax for converting an object of one type or class to another is, for example:

```
double x = 15.3;
int j = (int)x; //  or
int j = int(x);
```

While, for compatibility reasons, these are legal in C++, the four new conversion operators are recommended.

Type Conversion in C++

- **`static_cast<`*`type`*`>(`*`expr`*`)`**
  Can be used to make explicit any cast that the compiler would perform implicitly.

- **`const_cast<`*`non-const ptr`*`>(`*`const ptr`*`)`**
  Usually used to cast **`this`** to a non-**`const`** pointer within a **`const`** member function.

- **`dynamic_cast<derived ptr>(base ptr)`**
  Return a pointer to the derived class if, indeed, the object pointed to is of that type. Else returns 0.
  `dynamic_cast<derived ref>(base ref)`
  Return a reference to the derived class if, indeed, the object referenced is of that type. Else throws **`std::bad_cast`** exception (requires header **`<typeinfo> ).`**

- **`reinterpret_cast<type>(expr)`**
  Allows a low-level re-interpretation of the bit pattern of its operand. Used, for example to examine the individual bits of a double. E.g.

```
char*   c;
double* d = new double(123.4567);
c = reinterpret_cast<char*>(d);
```

`friend`'s

Non-member functions
which have
unrestricted access
to private and protected members
of another class

Uses of `friend`'s

- A global function requires access to the non-public members of objects of a particular class which
  - are passed to it as a parameter
- A member function of one class requires access to the non-public members of objects of a second class which are
  - are passed to it as parameters, or
  - are data members of the first class
- **All** member functions of one class require access to the non-public members of objects of a second class which are
  - are passed to them as parameters, or
  - are data members of the first class

Objections to `friend`'s

- The use of friends violates the protection of data members.
- The use of friends is never necessary, since member functions can always give the desired access in a controlled fashion.

HOWEVER

- If the declarations are placed in the same *.h* file and the function definitions are in the same *.cpp* file, the violation of protection is more apparent than real.
- The provision of otherwise unnecessary access functions (especially *Set...* functions) is possibly a greater security threat than is the use of the friends mechanism.

Friend global function

- A common use of a friend global function is in overloading an operator where
  - an object of our class is to be the Right Hand operand,
  - an object of a library class or an inbuilt type is to be the Left Hand operand

For example

in "person.h"

```
class Person{
        friend ostream& operator<<(ostream& out, Person person);
    public:
                Person(.....);
    private:
        char*   name;
        char*   address;
};
```

in "person.cpp"

```
ostream& operator<<(ostream& out, Person person){
    out << person.name << endl << person.address << endl;
    return out;
}
```

Using Overloaded <<

```
#include "Person.h"

void main(){
     Person p( . . . . );
     cout  << p <<  " is the new Person " << endl;
}
```

- The value returned by
  `cout << p`
  (or actually `operator<<(cout, p)`)
  is `cout` which becomes the left-hand operand to
  `<< " is the new person"`

Complex class – `Complex.h`

```
#include <iostream>
class Complex{
      friend Complex operator+(float x, const Complex& y);
      friend ostream& operator<<(ostream& out,
                                 const Complex& y);
   public:
      Complex();
      Complex(float r, float i);
      Complex operator+(float y);
      Complex operator+(const Complex& y);
   private:
      float real;
      float imag;
};
```

- Default copy constructor OK since no data members are pointers

Complex class – `Complex.cpp`

```
ostream& operator<<(ostream& out, const Complex& y){
   out << "Real " << y.real << " Imag " << y.imag;
   return out;
}

Complex::Complex(): real(0), imag(0){
}

Complex::Complex(float r, float i): real(r), imag(i){
}
```


Complex class – `Complex.cpp` (cont'd)

```
Complex operator+(float x, const Complex& y){
   Complex temp(y);
   temp.real += x;            // A GLOBAL function
   return temp;
}

Complex Complex::operator+(float y){
   Complex temp(*this);
   temp.real += y;            // Left-hand operator is a Complex
   return temp;
}

Complex Complex::operator+(const Complex& y){
   Complex temp(*this);
   temp.real += y.real;       // Left-hand operator is a Complex
   temp.imag += y.imag;
   return temp;
}
```

Using *Complex* class – `main.cpp`

```
#include "complex.h"
#include <iostream>
using namespace std;
main(){
   Complex x(1.0, 2.0);
   Complex y(3.0, 4.0);
   cout << "x = " << x << endl;
   cout << "y = " << y << endl;
   Complex a;
   Complex b;
   Complex c;
   a = x + 10.0;
   b = x + y;
   c = 10.0 + y;
   cout << "x + 10.0 = " << a << endl;
   cout << "x + y  = " << b << endl;
   cout << "10.0 + y = " << c << endl;
   cout << "3.0 + x + y + 1.0 = " << (3.0 + x + y +1.0) << endl;
   system("PAUSE");
   return EXIT_SUCCESS;
}
```


Using *Complex* class – `main.cpp` (Output)

```
x = Real 1 Imag 2
y = Real 3 Imag 4
x + 10 = Real 11 Imag 2
x + y  = Real 4 Imag 6
10 + y = Real 13 Imag 4
3 + x + y + 1 = Real 8 Imag 6
```

Member function of another class as friend

- A member function of another class may be made a friend of a given class.

```
class SomeClass{
        friend RetType OtherClass::SomeFunction(. . .);
    public:
        . . . . . .
}
```

SomeFunction can now directly access the **protected** and **private** data members of an object of class *SomeClass*.

- ALL member functions of another class may be made friends of a given class.

```
class SomeClass{
        friend class OtherClass;
    public:
        . . . . . .
}
```

All member functions of *OtherClass* can now directly access the **protected** and **private** data members of an object of class *SomeClass*.

Access Level for `friends`

- `friend` declarations are made in the class declaration of the class granting the friend access.
- It is not relevant to specify an *access level* for a `friend` declaration - the access level for a function is specified wherever the function is declared.
- While a `friend` declaration can appear anywhere within a class declaration, it is conventional to place them *at the beginning* of the class declaration –
  Before any access specifier

Uses of `const`

`const` is used in a number of contexts.

It always denotes that something will not be changed.

While we have covered most of these uses, it seems desirable to summarise them here

Constant data item

- Constant data item - *the data item can not be changed*

```
const int int1 = 3;
```

The value of the integer `int1` can not be changed.

```
const Person fred("Fred Smith");
```

No other Person object may be assigned to `fred`..
No data members of `fred` can be altered.
Only `const` member functions may be applied to `fred`.

```
const int int1;
```

Declaring a `const` data item without specifying an initial value is only allowed within a **class** declaration. (Initial value must be specified in *constructor initialisation list*).

```
int const int1;
```

`const` can go before or immediately after *type* or *class name* without change of meaning.

Pointer to a constant data item

- Pointer to constant data item - *the data item that the pointer points to can not be changed via this pointer*

```
const int* pInt1;
```

The object to which pInt1 points can not be altered via the pointer *pInt1*. e.g.

```
*pInt1= 3;
```

is illegal.

```
const Person* fred = new Person("Fred Smith")
```

No data members of object to which *fred* points can be altered.
Only **const** member functions may be applied via *fred*.

```
fred->NonConstMemberFunct(...);
```

is illegal.

Pointer to a constant data item

- The address of a constant data item can not be assigned to a non-constant pointer.

```
char const* name = "fred";
char* chars = name;       // illegal

const int number = 321;
int* num = &number;       // illegal
```

Pointers to constant data

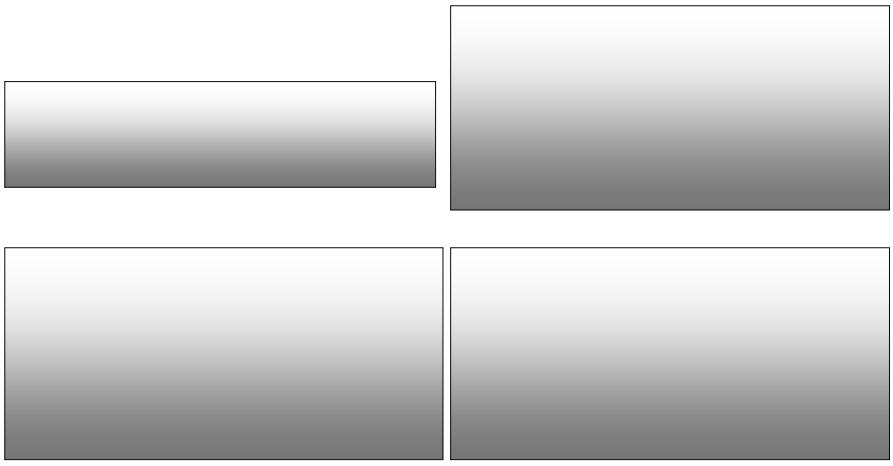
- Constant pointer to data item -
  the pointer can not be changed to point to something else - the thing it points to is not constant

```
int x, y;
int* const cpi = &x;            // must be initialised
x = 3;                          // OK
*cpi = 4;                       // OK
cpi = &y;                       // illegal
```

- Constant pointer to constant data item -
  the pointer can not be changed to point to something else - the thing it points to can not be changed via this pointer.

```
int x, y;
const int* const cpci = &x;     // must be initialised
x = 3;                          // OK
*cpci = 4;                      // illegal
cpci = &y;                      // illegal
```


Uses of `const` with data items



`const` argument to a Function

- Constant pointer or reference **argument** to a function (global or member function)- *the function code will not alter the object referenced by the argument*

```
void functA(const Person& person, const int iArr[]){
    person.f1();     // illegal if f1() is not a
                     //  const function
    iArr[1] = 3;      // illegal
}
```


`const` return type from a Function

- Constant **return type** from a function (global or member function)- *the recipient of the return value must be of the correct **const** type*

```
const char* GetName(){
    char* name = "My Name";
    return name;
}
. . . . .
char* hisName = GetName();       // illegal
const char* hisName = GetName(); // OK
```

`const` Member Function

- Constant member function -
  - *the function code will not alter the **non-static** data members of the object to which it is applied*

```
class myClass{
    public:
       void f1()const{ x = 3;};  // illegal
       void f2()const{ y = 3;};  // OK - y static
    private:
       int x;
       static int y;
}
```


Casting away `const`

- If we wish to give the user the appearance of `const` ness, but e.g. change the representation of data,
  - then we can "cast away `const` "

Change f1 above to

```
void f1()const{
      const_cast<myClass *>(this)->x = 3;
   };
```

and it is now legal

Virtual Functions
and
Polymorphism

Virtual Functions

With
- Function Overloading and
- Function Re-declaration,

the version of the function to be called is determined at
- Compile Time ("early binding")

But with
- **Virtual functions,**

the version of the function to be called is determined at
- Run Time ("late binding")

This provides us with **Polymorphism**

Polymorphism

- One of the most powerful mechanisms in Object Oriented programming.
- Allows major extensions to applications with minimal *changes* to existing code.
- An application may have to deal with objects of a number of classes which are similar but differ in details.

  The differences can be isolated in the member functions of the class.

  The rest of the application ignores these differences

Using Virtual Functions

- If, in the declaration of a class
  - a function declaration is preceded by the keyword **`virtual`**,
  - that function is a *virtual function*
- If, in a derived class a function is declared which
  - has the *same name, parameter list and return type* as a virtual function in an ancestor class
  - that function ***over-rides*** the virtual function in the parent class
  - the over-riding function is also a *virtual function*.
    - If the parameter list differs, the parent function is hidden (parent function is **re-declared**).
    - If only the return type differs, it is an error.
- A variable declared as a pointer to a base class may be assigned the address of an object of that class or of any of its descendant classes.

  If a **virtual function** is invoked via that pointer, the *appropriate function for the actual object being addressed* is called

Virtual Functions - example

```
class Employee : public Person{
   public:
            Employee(char* nm, int num):Person(nm), empNum(num){};
      void  SetRate(int rate){hrlyRate= rate;};
      virtual int CalcPay() {return hrlyRate * hoursWkd;};
   protected:
      int   hrlyRate, hoursWkd, empNum;
};
class Labourer : public Employee {
   public:
            Labourer(char* nm, int num):Employee (nm, num){};
       // inherits Employee::CalcPay
   protected:
};
class Salaried : public Employee {
   public:
            Salaried(char* nm, int num) :Employee(nm, num){};
      void  SetSal(int sal){salary = sal;};
      int   CalcPay(){return salary / 26;};
   protected:
      int   salary;
};
```


Using Virtual Functions - example

```
main(){
   Employee  mary("Mary", 1234);
   Labourer  nigel("Nigel", 2345);
   Salaried  irene("Irene", 3456);

   Person*   persPtr;
   Employee* empPtr;
   int p;

   persPtr = &mary;             // OK - an employee IS A Person
   p   = persPtr->CalcPay();    // NO - CalcPay is not a member of Person

   empPtr  = &mary;             // OK - mary is an Employee
   p   = empPtr->CalcPay();     // invoke Employee::CalcPay()

   empPtr  = &nigel;            // OK - nigel is an Employee
   p   = empPtr->CalcPay();     // Labourer inherits Employee::CalcPay()

   empPtr  = & irene;           // OK - irene is an Employee
   p   = empPtr->CalcPay();     // CalcPay() over-ridden in Salaried
                                //      so invoke Salaried::CalcPay()
}
```

Comments on example

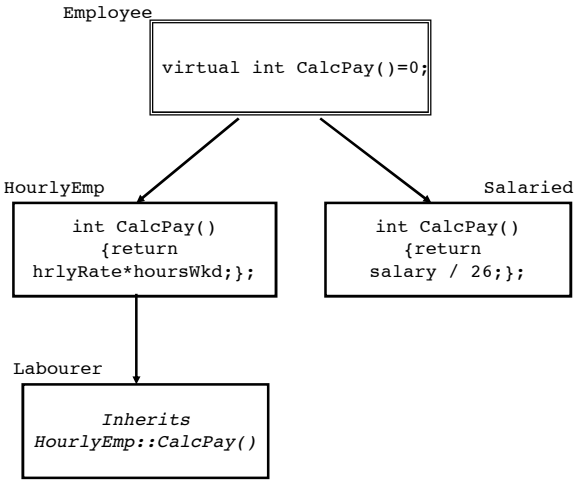
- The previous example was not entirely satisfactory:
  - Objects of class `Salaried` would have `hrlyRate` and `hoursWkd` data members although these are irrelevant for this class.
- A better design would be to have in the class `Employee`:
  - Only those data members that are relevant to ALL classes of employees.
    - *(It is likely that this set will NOT include ALL the data members needed by ANY class of employee.)*
  - Those member functions that are relevant to all employees and reference only the common set of data members.
  - **virtual** member functions for those operations that are required for all (or most) classes of employees, which may reference non-inherited data members.
  - If it is not appropriate to provide an implementation for the top-level of a virtual function:
    - Then this should be declared as a **Pure Virtual** function

Virtual Functions – example (revised)

```
class Employee : public Person{
    public:
        Employee(char* nm, int num):Person(nm),
                                empNo(num){...};
        virtual int  CalcPay()=0;
    // ***** pure virtual funct *****
    protected:
        int   empNo;
};
class HourlyEmp : public Employee{
    public:
        HourlyEmp(char* nm, int num)
                                :Employee(nm, num){};
        void  SetRate(int rate){hrlyRate= rate;};
        virtual int  CalcPay()
                {return hrlyRate * hoursWkd;};
    // ***** over-ride Employee::CalcPay() *****
     protected:
          int   hrlyRate, hoursWkd;
};
```

```
class Labourer : public HourlyEmp {
    public:
        Labourer(char* nm, int num)
                                HourlyEmp(nm, num){};
  // ***** Inherits HourlyEmp::CalcPay() *****
    protected:
};

class Salaried : public Employee {
    public:
        Salaried(char* nm, int num):
                                Employee(nm, num){};
         void  SetSal(int sal){salary = sal;};
        int   CalcPay(){return salary / 26;};
 // ***** over-ride Employee::CalcPay() *****
    protected:
        int   salary;
};
```

Revised Example



Using revised example

```
main(){
   Employee* staff[STAFF_SIZE];
   staff[0] = new HourlyEmp("Affris", 1234);
   staff[1] = new HourlyEmp("Barnes", 2345);
   staff[2] = new Salaried ("Chu",    3456);
   staff[3] = new HourlyEmp("Dana",   4567);
   staff[4] = new Labourer ("Eagles", 5678);
   . . . . . .
   for(int i = 0; i < STAFF_SIZE; i++){
        cout << staff[i]->Name() << " $" << staff[i]->CalcPay() << endl;
   }
```

- Which version of `CalcPay()` is executed on each time through the loop depends on the class of the object encountered
  - is determined at execution time.

- If a new class of employee is required:
  The only ***change*** to the existing application will be at the point at which a new employee is being added to the data base.
  New implementations of some or all of the virtual functions will be required, but these are ***additions*** to the application - not ***changes***

Abstract Class

- In the previous example, the base class Employee,
  - had no data fields that allow any sensible implementation of **`CalcPay().`**
- Since any Employee of whatever class must be paid,
  - it does not make sense to declare an object of class Employee.
- To declare unequivocally that **`CalcPay()`** will not be implemented for the Employee class, we make it a "pure virtual" function.
  - This is done by placing

    `=0`
  - between the closing parenthesis of the parameter list and the terminating semi-colon.
- A class containing one or more pure virtual functions is an "abstract class".
  - An object can not be created of an abstract class.
  - An abstract class exists only to be inherited from.
- If the parent of a class is an Abstract Class,
  - this class will also be an Abstract Class unless it over-rides **ALL** the Pure Virtual functions of its parent.

Function Relevant to few Sub-Classes

It will sometimes happen that there is an operation (say *RareFunct()*)that must be performed on only one (or very few) of the subclasses in an inheritance group. How to handle this?

- a) Declare

  ```
  virtual retType RareFunct(){};
  ```

  (i.e. a do-nothing function) as a member of the base class. Over-ride it in the class where it is relevant.

  Call this function regardless of the (sub)class of object.

  This is reasonably elegant and efficient, unless we wish to apply this function to ALL relevant objects and these make up a small fraction of the related objects.

- b) If the inefficiencies of a) are significant, the constructor for the unusual class could insert the object in a Linked List of objects of this class.

  The added complexity may be justified in the name of efficiency, in certain applications.

- c) If the base class is in a class library, or for some other reason it is not appropriate to add a virtual function to the base class, we may use a `dynamic_cast`.

  This approach is less efficient than approach a) above but may be necessary where the source code of the base class is unavailable.

Run-Time Type Identification (RTTI)

- RTTI allows retrieval of the actual type of an object referred to by a pointer (or a reference) to a base class.
  - Only "polymorphic classes" can use RTTI
    - A "polymorphic class" is one containing (or inheriting) at least one **`virtual`** function.

- Two operators are provided for RTTI
  - **`dynamic_cast`** operator allows the safe conversion of a pointer to a base class to a pointer to a derived class.
  - **`typeid`** operator allows the determination of the exact type of an object referred to by a reference or pointer to a base class

The use of RTTI should be minimised.

Polymorphism as provided by virtual functions is a more efficient and less error-prone solution in most circumstances.

RTTI – `dynamic_cast`

- dynamic_cast converts a pointer to a base class to a pointer to a derived class.
  dynamic_cast returns NULL if the object referenced by the base pointer is not of the desired class

```
class Employee : public Person{
   public:
      virtual int  CalcPay()=0;//virtual funct.
// Employee and all its derived classes are polymorphic classes
};
class HourlyEmp : public Employee{
   public:
      void  HourlyFunct(. . . );
};
class Salaried : public Employee {
   public:
      void  SalariedFunct(. . . );
};

main(){
   Employee*  emp;
   HourlyEmp* hrly;
   Salaried*  sal;

   if(hrly = dynamic_cast<HourlyEmp*>(emp))  hrly->HourlyFunct(. . . );
   else  if(sal = dynamic_cast<Salaried*>(emp))sal->SalariedFunct(. . . );
   else error();

}
```

RTTI – `typeid`

- `typeid` operator applied to an expression returns a reference to the object of type
  `const typeinfo`
  which is appropriate to the type of the expression

- The principal member function of the `typeinfo` class is
  `const char* name() const;`
  which returns the name of the type or class.

  Operators `==` and `!=` are overloaded for the `typeinfo` class.

RTTI – `typeinfo`

```
#include <typeinfo.h>

main(){
  Employee*  emp1;
  Employee*  emp2;
  emp1 = new Hourly(. . . );
  emp2 = new Salaried(. . . );
  if(typeid(*emp1) == typeid(*emp2))cout<<"Types are the same\n";
  else cout << "Types are different\n";
  cout << "*emp1 is a " << typeid(*emp1).name() << endl;
  cout << "*emp2 is a " << typeid(*emp2).name() << endl;
}

produces the output

        Types are different
        *emp1 is a Hourly
        *emp2 is a Salaried
```

Trouble with Destructor

Consider the following case

```
class Person{
   public:
      Person(char* nm){...};
      ~Person(){delete [] name;};
   protected:
      char* name;
};
class CarOwner: public Person{
   public:
      CarOwner(char* nm, char* make) :Person(nm){...};
      ~CarOwner(){delete [] car;};
   protected:
      char* car;
};
main(){
   CarOwner* pers1 = new CarOwner("John", "VW");
   Person*   pers2 = new CarOwner("Mary", "MG");
   delete pers1;    // calls ~CarOwner() which deletes "VW
                    //   and then ~Person() which deletes "John"
   delete pers2;    // calls ~Person() which deletes "Mary",
                    //   "MG" remains as a memory leakage
```


Virtual Destructor

- If an object of a derived class is destroyed via a **pointer to the derived class**, the destructor of the derived class is called, and then the parent class destructor is called.
- If an object of a derived class is destroyed via a **pointer to a parent class**, the destructor of the derived class is not called.
- To overcome this, the **destructor of the parent** class must be **virtual**.
- Virtual destructors are a special case of virtual functions
- If the destructor of a base class is virtual, the destructor of a derived class will override the base class destructor **even though its name is different**

Using Virtual Destructor

```
class Person{
      public:
           Person(char* nm){...};
            virtual ~Person(){delete [] name;};        // virtual destructor
      protected:
           char* name;
};
class CarOwner: public Person{
       public:
           CarOwner(char* nm, char* make):Person(nm){...};
           ~CarOwner(){delete [] car;};                // over-rides ~Person and is also virtual
      protected:
           char* car;
};

main(){
 CarOwner* pers1 = new CarOwner("John", "VW");
 Person*   pers2 = new CarOwner("Mary", "MG");
 delete pers1; // calls ~CarOwner() which deletes "VW"  and then ~Person() which deletes "John"
 delete pers2; // calls ~CarOwner() which deletes "MG" and then ~Person() which deletes "Mary"
}
```


Conversions

Converting an object of one class
to an object of another class

Standard Conversions under Inheritance

- In any context where
  - an *object* of a base class, or
  - a *reference* to a base class, or
  - a *pointer* to a base class

  is required, a corresponding item of a publicly derived class can be used.

- If we wish to use a pointer (or a reference) to a base class where a pointer (or a reference) to a derived class is required,
  - This is an **inherently dangerous operation**, so the pointer must be explicitly cast to a pointer to the derived class
  - Unpredictable results will occur if we use the pointer to apply a derived class function, and the object is not of that derived class.

Using Standard conversions

```
class Person{...};
class Employee : public Person{...};

fun1(Person){...}
fun2(Person&){...}
fun3(Person*){...}
fun4(Employee&){...}
fun5(Employee*){...}

main{
   Employee joe(...);
   Person mary(...);

   fun1(joe);          // OK
   fun2(joe);          // OK
   fun3(&joe);         // OK

   fun4(mary);         // NO!!!
   fun4(&mary);        // NO!!!
}
```

Implicit Construction

- If
  - a function requires, as a parameter, an object of a class, and
  - that class has a constructor with ONE parameter

  that function can be called with an object of the type of that parameter.

  The constructor will be called to create an object of the type and this will then be passed to the function.

Using Implicit Construction

```
class Person{
    public:
        Person(char* nm, int sal = 0); // Requires only 1 parameter
    private:
        char* name;
        int   salary;
};
Person::Person(char* nm, int sal){
    name = new char[strlen(nm) + 1];
    strcpy(name, nm);
    salary = sal;
}

funct(Person){...}

main(){
    funct("Joe Blow");        // A Person will be constructed and
                              //   passed to funct
}
```

Conversion Operator

- If we wish to convert an object of a class to some other type of object, we can state how that conversion is to be done.

  For example if we wish to convert a Person to an integer:

```
class Person{
    public:
        Person(char* nm, int sal = 0);
        operator int(){return salary;};
               // Note NO parameters, NO return type
    private:
        char* name;
        int   salary;
};
```


Using Conversion Operator

```
main(){
    Person fred("Fred", 1000);
    Person mary("Mary", 5000);
    int x = static_cast<int>(fred)  +  static_cast<int>(mary);
        // conversion called for explicitly. x now has value 6000
    int y = fred + mary;
        // conversion done implicitly.  y also now has value 6000
}
```

Syntax of Conversion Operators

- The general syntax of the conversion operator definition is:
  - `operator type();`

  The conversion function must:
  - be a member function (NOT a global function),
  - have NO return type,
  - have NO parameters.

  The *type* may be:
  - any predefined type such as **`int, char, float, unsigned int`** etc.,
  - the name of any class.

- The conversion operator may be invoked:
  - explicitly using the cast notation, e.g.
    - `static_cast<int>(Fred)`
  - or implicitly when
    - an object of one class is provided, but
    - an object of the other class or type is required

Type Conversion in C

In C the syntax for converting an object of one type or class to another is, for example:

```
double x = 15.3;
int j = (int)x; //  or
int j = int(x);
```

While, for compatibility reasons, these are legal in C++, the four new conversion operators are recommended.

Type Conversion in C++

- **`static_cast<`*`type`*`>(`*`expr`*`)`**
  Can be used to make explicit any cast that the compiler would perform implicitly.

- **`const_cast<`*`non-const ptr`*`>(`*`const ptr`*`)`**
  Usually used to cast **`this`** to a non-**`const`** pointer within a **`const`** member function.

- **`dynamic_cast<derived ptr>(base ptr)`**
  Return a pointer to the derived class if, indeed, the object pointed to is of that type. Else returns 0.
  `dynamic_cast<derived ref>(base ref)`
  Return a reference to the derived class if, indeed, the object referenced is of that type. Else throws **`std::bad_cast`** exception (requires header **`<typeinfo> ).`**

- **`reinterpret_cast<type>(expr)`**
  Allows a low-level re-interpretation of the bit pattern of its operand. Used, for example to examine the individual bits of a double. E.g.

```
char*   c;
double* d = new double(123.4567);
c = reinterpret_cast<char*>(d);
```

Containers

The Standard C++ Library provides C++ programmers with
a collection of container types

a library of common data structures
Linked lists, vectors, deques, sets, maps

a set of fundamental algorithms that operate on them.

Container Library Components

- Principal components of Container Library
  - **Containers** – data structures that manage a collection of objects.
  - **Iterators** - mechanisms for traversing and examining the elements in a container.
  - **Algorithms** - procedures that operate on the contents of a container.
- Other components of Container Library
  - **Function objects** - provide a more efficient mechanism than pointers to functions.
  - **Adaptors** - provide a new interface to a container or to an iterator.
  - **Allocators** - encapsulate the memory model for different machine architectures.

Types of Containers

- **Sequence** containers - containers that organise objects, all of the same type, into a strictly linear arrangement.
  - **Vector** - an array which can grow at one end
  - **List** - linked list
  - **Deque** - an array which can grow at either end
- **Associative** containers - provide for fast retrieval of objects via keys. The elements are sorted, so fast binary search is possible.
  - **Set** - supports unique keys (at most one of each key value) and provides fast retrieval of keys themselves.
  - **Multiset** - a set where equal keys (possibly multiple copies of the one key) are allowed.
  - **Map** - supports unique keys for fast retrieval of another type based on the key.
  - **Multimap** - a map where equal keys are allowed.

`vector, deque` Containers

- These containers use the basic one-dimensional array.
- The container may be subscripted for storing or retrieving items in random order - but use of an out-of-range subscript leads to unspecified results.
- Items can be inserted at the beginning, end or within the container. If insertions are at
  - *the end* - if there is space, the item is added after the last. If there is insufficient space, twice the existing space is allocated, the existing items are copied, and the new item is added.
  - *the middle* - items are moved towards the end to make room and the item is added. If necessary, the space is reallocated.
  - *the beginning* -
    - *vector* - as for the middle.
    - *deque* - unused space is divided between both ends, so insertion at the beginning is as efficient as at the end.
- Deletions are the inverse of insertions. If necessary, other items are moved to occupy the space of the deleted item.

`list` Containers

- The underlying structure is a doubly-linked list.
- Fast random access to items is not supported.
- Insertions are more efficient than with `vector` or `deque`.
- Memory is allocated and released for individual items as required

`set, multiset, map, multimap`

- The underlying data structure is the binary tree.
- Fast access is provided to individual items - based on content, not on position within the container.
- `operator<` must be overloaded for the class of item being stored.

Container Declaration

Elements held in a container can be

- Primitive language types, e.g. `int`, `double`…
- Pointer types.
- User defined types (classes).

  In this case, a default constructor and a copy constructor must be available for the class

Sequential Container Declaration

- Vector
  `vector<Person> people1;`
  – A Vector of default size with no elements.
  `vector<Person> people2(5);`
  – A Vector of size 5 with no elements.
  `vector<Person> people3(5, aPerson);`
  – A Vector, size 5, containing 5 copies of aPerson.

- Deque
  `deque<Person> people1;`
  – Etc as for Vector

- List
  `list<Person> people1;`
  – A List with no elements.
  `list <Person> people2(5);`
  – A List with 5 elements, each initialised by default constructor.
  `list <Person> people3(5, aPerson);`
  – A Vector, size 5, each element containing copy of aPerson.

`set` or `multiset` Declaration

- `Set`

  set<Employee, less<Employee> > empSet;
  set<Employee, greater<Employee> > empSet;

  A Set of Employees, whose ordering is defined by the less-than comparison operator (`operator<`)giving ascending order.

  – Alternatively the greater-than comparison operator (`operator>`) can be used to give descending order.

  – The function object requires the overloading of the appropriate comparison operator for the class being stored.

- If `multiset` is specified rather than `set`, objects that are equal to each other may be stored

`map` or `multimap` Declaration

- Pair data structure – used extensively by `map` and `multimap` (and in other contexts).
  – A `pair` is a `struct` with two components – `first` and `second`.
  (It is a template structure – *to be explained later* – so that the two components can be defined to be of any types .)

  ```
  map<int, Employee* , less<int> > empMap
  ```

- A `map` containing `pairs` – `first` an `int` and `second` a pointer to an Employee. The `int` is the key and the ordering is defined by the `operator<` defined over `int`s.

- `multimap` – As for a map except that duplicates are allowed

Iterators

- Iterators are a generalisation of pointers that allow a programmer to work with different data structures in a uniform manner.
- e.g. compare printing the contents of a vector

```
#include <vector>
#include <iostream>
using namespace std;
main(){
  vector <int> v(3);
  v[0] = 5; v[1] = 2; v[2] = 7;
  vector<int>::iterator
              vFirst = v.begin();
  vector<int>::iterator
              vLast = v.end();
      while(vFirst != vLast)
        cout << *vFirst++ << " ";
  system("PAUSE"):
  return EXIT_SUCCESS:
}
```

with printing the contents of a list

```
#include <list>
#include <iostream>
using namespace std;
main(){
  list <int> l;
  l.push_back(5); l.push_back(2);
  l.push_back(7);
  list<int>::iterator
            lFirst = l.begin();
  list<int>::iterator
            lLast = l.end();
  while(lFirst != lLast)
     cout << *lFirst++ << " ";
  system("PAUSE")
  return EXIT_SUCCESS;}
```

In each case the iterator moves from one element to the next - only the iterator cares whether via address arithmetic or a pointer.

Types of iterators

- *Input* and *output* iterators
  - Used with *istream* and *ostream* for I/O
  - Valid operations
    - `*` (dereference)
    - `++` (pre- or post-increment)
- *Forward* iterators
  - Usable with all containers
  - Valid operations as for *input* and *output*
- *Bidirectional* iterators
  - Usable with all containers
  - Valid operations as for *forward* plus
    - `--` (pre- or post-decrement)
- *Random access* iterators
  - Only usable with vector or deque
  - Valid operations as for *bidirectional* plus
    - `+(int), -(int)`

Operations on `vector`

- The following are the most important operations that may be applied to a Container. (Below `value_type` refers to the class or type of objects stored in the container.)

- `vector`

`iterator begin()`	Returns an iterator referring to the **first** element.
`iterator end()`	Returns an invalid iterator referring **beyond the last** element.
`value_type front()`	Returns the **first** element
`value_type back()`	Returns the **last** element
`iterator insert(iterator, value_type)`	Inserts the value_type **before** the specified iterator.
`void erase(iterator)`	Removes the element referred to by the iterator.
`void push_back(value_type)`	Adds the element to the **end** of the vector.
`void pop_back()`	Removes the **last** element.
`int size()`	Returns the number of elements currently in the vector.
`bool empty()`	Is the vector Empty?

Operations on `deque, list`

- `deque`
  - ***As for*** `vector` ***plus:***

`void push_front(value_type)`
Adds the element to the **front** of the `vector`.

`void pop_front()`
Removes the **first** element.

- `list`
  - ***As for*** `deque` ***plus:***

`void sort()`
Sorts list into ascending order.

Operations on `set, multiset`

- Set or Multiset

`iterator begin()` Returns an iterator referring to the **first** element.

`iterator end()` Returns an invalid iterator referring **beyond the last** element.

`int count(value_type)` The number of elements equal to `value_type`.

`void erase(iterator)` Remove the element pointed to by the iterator.

`int erase(value_type)`
Remove the element (or elements for Multiset) equal to `value_type`. Return the number of elements removed.

`iterator find(value_type)`
Find the element equal to `value_type` and return an iterator pointing to it – or `end()` if not found.

`pair<iterator, bool> insert(value_type)`
Insert an element equal to `value_type` and return an iterator to it and a `bool` that is `true` if the element was inserted, `false` otherwise. (In a `set` element not inserted if it matches one already in the set.)

Operations on `map, multimap`

- `map` or `multimap`

`iterator begin()` Returns an iterator referring to the **first** element.

`iterator end()` Returns an invalid iterator referring **beyond the last** element.

`int count(key_type)` The number of elements having the key `key_type` .

`void erase(iterator)` Remove the element pointed to by the iterator.

`int erase(key_type)` Remove the element (or elements for Multimap) having key `key_type`. Return the number of elements removed.

`iterator find(key_type)`
- Find the element having key `key_type` and return an iterator pointing to it – or `end()` if not found.

`pair<iterator, bool> insert(value_type)`
- Insert an element equal to `value_type` and return an iterator to it and a `bool` that is `true` if the element was inserted, `false` otherwise. (In a `map` element not inserted if it matches one already in the set.)
- For a `map` or `multimap` `value_type` is a `key/value` pair.

`value& operator[](key_type)`
- Use subscript-like syntax, providing a key as the subscript to return a reference to the associated value.

Container usage

- To illustrate the usage of some containers in the Container Library, the objects stored will be of the following class:

```
class Employee{
    friend ostream& operator<<(ostream& out, Employee& emp){
        out << emp.empNo << " " << emp.details << endl;
        return out;
    }
  public:
    Employee(int no = 0, char* dets = 0):empNo(no){
      if(dets != 0){
        details = new char[strlen(dets)+1];
        strcpy(details, dets);
      }
      else
        details = 0;
    }
    int operator<(const Employee& other)const{
           return empNo < other.empNo;}
     int GetEmpNo(){return empNo;};
  protected:
    int    empNo;
    char*  details;
};
```


Container usage (cont'd)

- These objects will be used in the following illustrations :

```
Employee empArr[10];
empArr[0] = Employee(123, "John");
empArr[1] = Employee(234, "Mary");
empArr[2] = Employee(100, "Elizabeth");
empArr[3] = Employee(765, "Natasha");
empArr[4] = Employee(222, "Hahai");
empArr[5] = Employee(12, "Sudesh");
empArr[6] = Employee(444, "Sujan");
empArr[7] = Employee(135, "Miyako");
empArr[8] = Employee(531, "Ulianov");
empArr[9] = Employee(333, "Do");
```

Using a `vector`

```
#include <vector.h>
#include "Employee.h"
main(){
   Employee empArr[10];
        // . . . .fill array . . . .
   vector<Employee> empVec(3);        // initial size 3
   empVec[0] = empArr[0];
   empVec[1] = empArr[1];
   empVec[2] = empArr[2];
   for(int i = 3; i < 10; i++)        // append to array –
        empVec.push_back(empArr[i]);  //   grows if necessary
   typedef vector<Employee>::iterator vIter;
                // the safe way to process container contents
   for(vIter vit = empVec.begin(); vit != empVec.end(); vit++)
        cout << *vit;
   for(int i = 0; i < 10; i++)        // if I goes outside limit,
        cout << empVec[i];            //   results undefined
}
```


Using a `vector` (output)

- Output from running previous program

```
123 John
234 Mary
100 Elizabeth
765 Natasha
222 Hahai
12 Sudesh
444 Sujan
135 Miyako
531 Ulianov
333 Do
```

Using a Set

```
#include <set.h>
#include "Employee.h"
main(){
   Employee empArr[10];
   // . . . .fill array . . . .
   set<Employee, less<Employee> > empSet;// less is "function object"
   for(int j = 0; j < 10; j++){     // items inserted in set in order
   if(!empSet.insert(empArr[j]).second)  //     specified by less
      cout << "Employee " << empArr[j] << "is a duplicate" << endl;
   }

   typedef set<Employee, less<Employee> > iterator sIter;
   for(sIter sit = empSet.begin(); sit != empSet.end(); sit++)
      cout << *sit;        // output all elements of set
   cout << endl;

   sIter sFind;               // find elements from set
   int nums[] = {123, 765, 888, 12};
   for(int k = 0; k < 4; k++){
      sFind = empSet.find(Employee(nums[k]));
      if(sFind != empSet.end())   cout << *sFind;
      else      cout << "No employee has number" << nums[k] << endl;
   }
}
```


Using a `set` (output)

- Output from running previous program (Employee number for Sujan changed to 234 to test duplicate insertion)

```
Employee 234 Sujan
Is a duplicate
12 Sudesh
100 Elizabeth
123 John
135 Miyako
222 Hahai
234 Mary
333 Do
531 Ulianov
765 Natasha

123 John
765 Natasha
No Employee has number 888
12 Sudesh
```

Using a map

```
#include <map.h>
#include "Employee.h"
main(){
   Employee empArr[10];
   // . . . .fill array . . . .
   map<int, Employee*, less<int> > empMap;
   for(int j = 0; j < 10; j++)   // items inserted in set in order
      empMap[empArr[j].GetEmpNo()] = &empArr[j]; //spec'd by less

   typedef map<int, Employee*, less<int> > :: iterator mIter;
   for(mIter mit = empMap.begin(); mit != empMap.end(); mit++)
      cout << *((*mit).second);     // output all elements of set
   cout << endl;

   mIter mFind;
   int nums[] = {123, 765, 888, 12};
   for(int k = 0; k < 4; k++){
        mFind = empMap.find(nums[k]) // find elements from set
        if(mFind != empMap.end())     cout << *((*mFind).second);
        else  cout << "No employee has number" << nums[k] << endl;
   }
}
```


Using a map (output)

- Output from running previous program

```
12 Sudesh
100 Elizabeth
123 John
135 Miyako
222 Hahai
234 Mary
333 Do
444 Sujan
531 Ulianov
765 Natasha

123 John
765 Natasha
No Employee has number 888
12 Sudesh
```

Exception Handling

A mechanism that allows
two separately developed program components to communicate
when a program anomaly
(an error condition - an *exception*)
is encountered during execution of a program.
.

Error Handling

- When a function, for some reason, is unable to perform its designated function it can
  - Ignore the problem, possibly causing garbage results, or an Operating System initiated abort (e.g. GPF).
    - This is what happens when an out-of-range subscript is used with a C-style array.
  - Return an error code to the calling program, which can then be tested.
    - Operator new returns the invalid value 0 if there is insufficient memory. Similarly, most Windows Application Program Interface calls return 0 if the operation requested can not be performed.
  - Abort with an error message specifying the source code line that detected the error
    - *The assert() macro provides this facility*.
  - ANSI Standard C++ Exception Handling provides a more flexible solution to this problem.
    - *This will be described in some detail*.

Assertions

- Assertions provide a fairly simple means of locating error conditions

```
//#define NDEBUG
#include <assert>
. . .
assert(int-expr);
. . .
```

The expression in the assert statement is evaluated.

If it returns a non-zero value, nothing happens.

If it returns zero, the program aborts with the message

`Assertion failed:`*`int-expr`*`, file `*`filename`*`, line `*`linenum`*

Where, *`filename`* and *`linenum`* identify the position of the failing `assert()` statement in the source code.

Uncommenting the *`#define NDEBUG`* statement has the same effect as commenting out all `assert()` statements in the program file.

Assertions example

- A common example of the use of assertions:

```
#include <assert>
main(){
   . . .
   Person* pPtr = new Person(...);
   assert(pPtr != 0);
   . . .
}
```

If there is insufficient memory to allocate a new object, there is no point in continuing execution.

This error is almost certainly due to programming errors resulting in "memory leakage" – failure to delete unused objects.

Limitations of Assertions

- Assertions are principally of use during debugging of programs.
- Assertions have limitations:
  - The first assertion to fail will abort the job - it is not possible to perform any processing after this failure.
  - Apart from indicating the source code position of the failing assertion, no other information may be passed to the user to indicate values of variables that caused the failure.
  - There is a significant run time penalty associated with evaluating the condition being asserted.
- For these reasons, assertions are not suitable for informing the rest of the application that an unusual, but recoverable, condition has occurred.

Exceptions

- Exceptions provide the most flexible means for a function to inform the calling program that an unusual condition has occurred.
- As much information as is appropriate may be passed to the calling program.
- On receipt of the exception, a special piece of code in the calling program may be executed to analyse the information passed back.

  If appropriate, the calling program may take corrective action and continue execution.

  Alternatively, the calling program may elect to abort after providing any information to the user.

Using Exceptions

- The basic use of exceptions may be illustrated as follows:

```
class MyExcept{
   public:
      MyExcept(. . . );
   . . .
}
void SomeFunct(. . .)throw(MyExcept){
   if(unusualCondition)
      throw(MyExcept(. . . ));        //throw the exception
   . . .
}
main(){
   . . .
   try{                          //a try block
      SomeFunct(. . . )
   }
       //a catch block - must immediately follow a try block
   catch(MyExcept exc){
      get information from exc
   }
   . . .
}
```


Throwing an Exception

Notes on previous code outline

- It is desirable (but not required) that a function declare the types of exception objects that it may throw.

  `retType FuncName(Params)throw(type1, type2, ..)`

  A warning diagnostic occurs if any other exception is thrown.

- When a function detects an abnormal situation, it creates an object of the appropriate type, passing any appropriate parameters to the constructor.

  `throw ClassName(params);`

  A `throw` operation with this object as operand is then executed. This terminates the function in the same manner as a `return`.

Responding to an Exception

- To process the exception, the statements which call the function (either directly or indirectly), are placed in a `try` block.

  ```
  try{ . . . . . }
  ```

  If the function call results in an exception, the remaining statements in the `try` block are by-passed.

- Immediately following the try block are one or more catch blocks.

  ```
  catch(type name){......}
  ```

  If an exception occurs within the try block, the catch block with the matching type is executed.

Exceptions – an example

```
#include <iostream>
class TooBig{
   public:
      TooBig(int v) : val(v) {};
      int val;
};
class TooSmall{
   public:
      TooSmall(int v) : val(v) {};
      int val;
};
void UseInt(int xx)throw(TooSmall,
   TooBig){
   if(xx > 10)
        throw(TooBig(xx));
   if(xx < 5)
        throw(TooSmall(xx));
}
```

```
main(){
   for(int i = 1; i < 15; i++){
      try{
         UseInt(i);
         cout << "i = " << i
              << endl;
       }
       catch(TooBig& exc){
          cout << "Too big: "
             << exc.val << endl;
       }
       catch(TooSmall& exc){
          cout <<"Too small: "
             << exc.val << endl;
       }
   }
}
```

Object-Oriented Software Development Using CRC Cards and UML

References:

Nancy M. Wilkinson. *Using CRC Cards - An Informal Approach to Object-Oriented Software Development*, AT&T Bell Laboratories, SIGS Books, New York, 1995

Martin Fowler with Kendall Scott. *UML Distilled, Second Edition - A Brief Guide to the Standard Object Modelling Language*, Addison-Wesley 2000

CRC Cards and UML

"Despite the buzzing about object-oriented methods and tools, the true measure of how successful someone will be with objects is the degree to which he or she intuitively grasps the underlying idea of the object-oriented approach.

Methodologies and tools cannot replace this gut-level understanding; they simply support the work done once the transition from process to object is made.

Using CRC cards is an excellent way for people to begin to make this transition."

Mary Wilkinson

"Now widely adopted as the de facto industry standard and sanctioned by the Object Management Group, the Unified Modelling Language (UML) is a notation all software developers need to know and understand.

One of the most valuable techniques for learning OO is CRC cards, which are not part of UML, although they can, and should, be used with it"

Martin Fowler

Classes, Responsibilities, Collaborators

class name	
subclasses	
superclasses	
responsibilities	collaborators

Suggested format for CRC card

CRC Cards

- **Class Definition (on back of card)**
  - A concise definition of the abstraction represented by the class.
- **Class Name**:
  - if chosen appropriately will greatly increase the understandability of the model.
- **Subclass, Superclass**
  - If they exist can be shown. Very probably these will be added late in the development cycle.
- **Responsibilities**
  - Knowledge that objects of a class maintains , and
  - Services that a class provides.
- **Collaborator**
  - Class(es) that provide services that are needed to allow fulfilment of responsibilities - listed on the same row as the relevant responsibility.

CRC Card Session

The CRC card technique is based on the use of small teams to perform analysis and design.

- Team Size
  - The ideal team size has been found to be 4 or 5.
  - A smaller group is unlikely to have the required range of problem domain and solution domain expertise.
  - A larger group tends to fragment, with greater interaction problems.

CRC Team

- Team Composition
  - **Domain Expert** — At least one participant should have "domain knowledge", ie should understand what the system should do, but may have little knowledge of how it may be accomplished.
  - **Solution Expert** — Clearly at least one member of the team should have expertise in object-oriented methodology, even if they have little knowledge of the problem domain.
  - **Facilitator** — The facilitator may also be the Solution Expert but may be a person who is not a member of the team but who uses her object-oriented expertise to keep the team on track. She may perform this role with more than one team concurrently.
  - **Group Dynamics** — Newly formed project teams sometimes find that the interactions in a team are a unifying experience, but " Don't try technological fixes for sociological problems" (Stroustrup). It is best to have a group who work well together, respect each other and are not dominated by one or a few of the members

CRC Tasks

Choosing a subset of the problem

It is essential that a CRC session focus on one fairly small and manageable portion of the system at a time.

e.g. the following example comes from Wilkinson:

The Problem

This application will support the operations of a technical library for an R&D organisation. This includes the searching for and lending of technical library materials, including books, videos, and technical journals. Users will enter their company IDs in order to use the system; and they will enter material ID numbers when checking out and returning items.

Each borrower can be lent up to five items. Each type of library item can be lent for a different period of time (books 4 weeks, journals 2 weeks, videos 1 week). If returned after their due date, the library user's organisation will be charged a fine, based on the type of item (books $1/day, journals $3/day, videos $5/day).

Materials will be lent to employees with no overdue lendables, fewer than five articles out, and total fines less than $100.

Possible Classes

A brainstorming session produces the following set of candidates for classes

- Library
- Librarian
- User
- Borrower
- Article
- Material
- Item
- Due Date
- Fine
- Lendable
- Book
- Video
- Journal

Filtering Classes

- Difference between Library and Librarian?
  - Librarian interacts with user and checks items in and out.
  - Hard to see role for Library - perhaps name of the application.
- Difference between Article, Item, Material and Lendable.
  - These all seem to refer to the same thing, so which name to use?
  - Lendable seems most expressive - discard other class names.
- Difference between User and Borrower?
  - Borrower represents information about library patrons - knows what books are out and which ones etc.
  - User represents a person using the library.
  - Retain both, at least tentatively.
- Due Date and Fine?
  - These are probably attributes of Lendable and Borrower respectively rather than independent objects.
  - But there is probably a need for a Date class since dates will need to be compared, added to etc. Due Date could be an instance of Date.

Tentative Classes

Librarian: the object that fufils user requests to check out check in and search for items

User: The human being that comes to use the library

Borrower: The set of objects that represent users who borrow from the library

Date: The set of objects that represent dates in the system

Lendable: The set of objects that represent items to be borrowed from the library

Book: The set of objects that represent books to be borrowed from the library

Video: The set of objects that represent DVDs and VideoTapes to be borrowed from the library

Journal: The set of objects that represent Technical journals to be borrowed

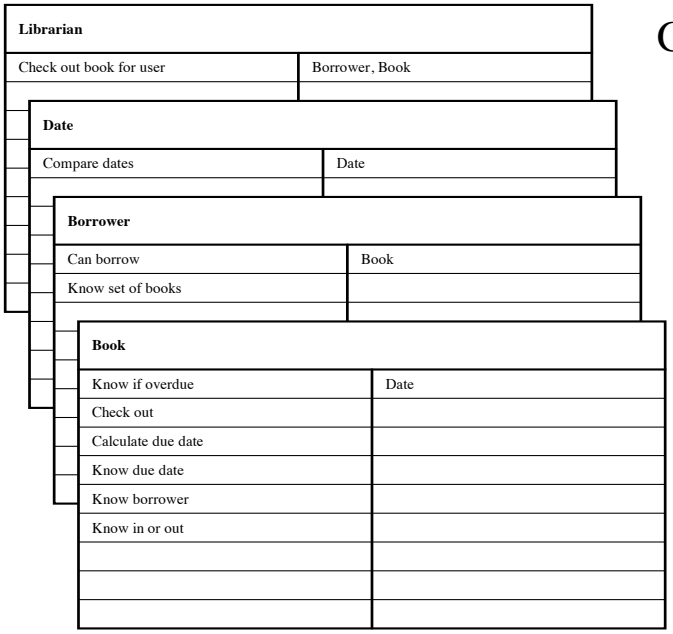
Check-out Scenario

"What happens when Barbara Stewart, who has no accrued fines
and one outstanding book, not overdue,
checks out a book entitled *Effective* C++ *Strategies*?"

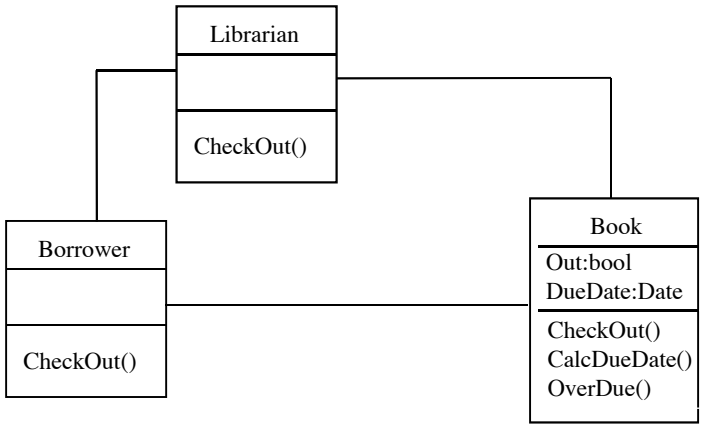
Check-out Scenario

- Librarian
  - needs to add “check out lendable”to list of responsibilities.
    - needs to collaborate with Book and Borrower to do this.
- Book
  - needs to add “know if overdue” to list of responsibilities.
  - needs to add “check out” to list of responsibilities. This requires updating:
    - in-or-out status, borrower, due date.
  - needs to add “calculate due date” to list of responsibilities.
    - needs to clllaborate with Date to do this.
- Borrower
  - needs to add “know set of lendables” as a responsibility.
- Date
  - needs to add “compare dates” to list of responsibilities.
  - needs to add “add days to date” to list of responsibilities.

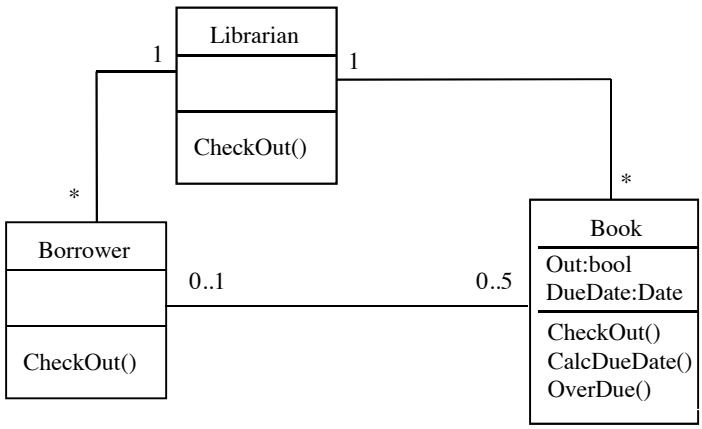
Cards after First Scenario



UML Class Diagram

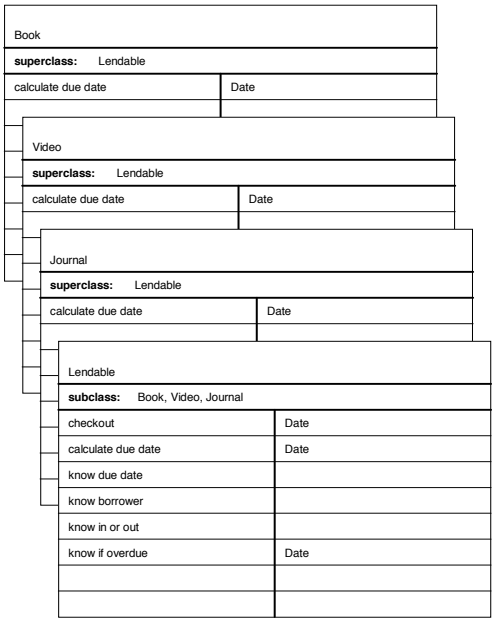


Adding Multiplicity



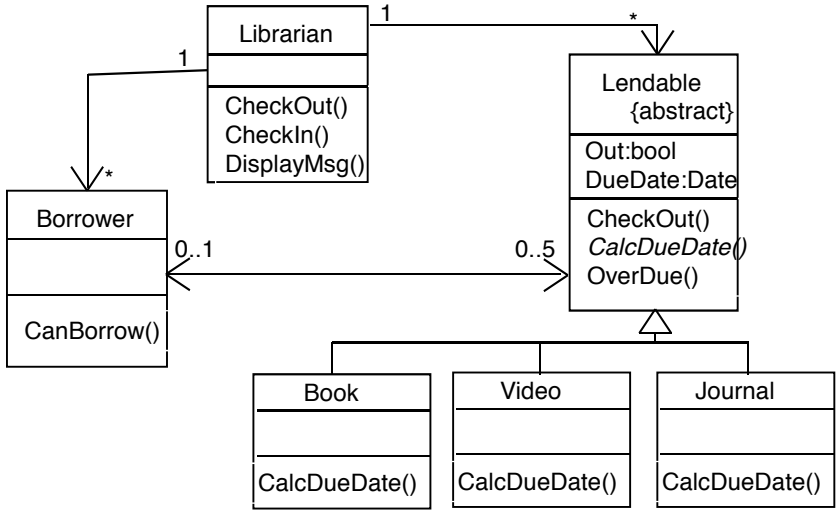
Superclass Discovery

- Checking out a Video or a Journal gives a very similar scenario to checking out a book.
- Lendable should be a superclass of Book, Video and Journal.
  - The difference between these lies in the way that due date is calculated
- Librarian should replace “check out book” by “check out lendable” in list of responsibilities
- Borrower should replace “know set of books” by “know set of lendables” in list of responsibilities



Cards after Superclass Discovery

Adding Inheritance and Navigability



Check-in Scenario

"What happens when Liz Flanagan,
returns the book, *Barely Managing O-O Projects*,
on time?"

Check-in Scenario

- **Librarian**
  - needs to add “check in lendable”to list of responsibilities.
  - needs to collaborate with Book and Borrower to do this
- **Book**
  - needs to add “check in” to list of
  - responsibilities of base class. This requires updating:
    - in-or-out status
    - borrower
- **Borrower**
  - already has “know set of lendables” as a responsibility

Exceptional Scenarios

“What happens when Gregg Vosander tries to check out a journal,
Genetic Programming in your spare time
when he already has 5 volumes of the same journal at home

- Borrower has “know set of lendables” responsibility, so knows that at lendable limit and can report this to Librarian.
- Librarian, since borrower is at lendable limit, needs to alert User that no more can be checked out, and end the session.
- Librarian needs to add “display message” to set of responsibilities.

Another Exceptional Scenario

"What happens when Garrett Ziegler returns a video, *Introduction to Interactive Television*, 3 days overdue

- Book has "know if overdue" as a responsibility so can report this to Librarian.
- Librarian needs to know fine to charge.
  - Book is obvious choice for this since each type of lendable does this differently. Book adds "calculate fine" to list of responsibilities.
- Librarian asks Borrower to record the amount of the fine.
  - Borrower adds "know fine amount" to list of responsibilities.
- Librarian needs to inform User of amount of fine.
  - Uses "display message" responsibility to do this.

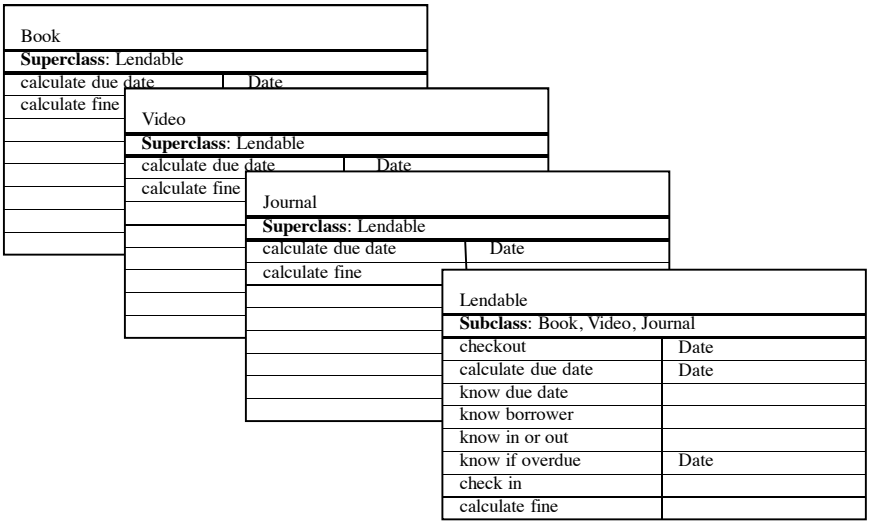
Cards after Exceptional Scenarios (1)

Date	
compare dates	Date
add days to date	

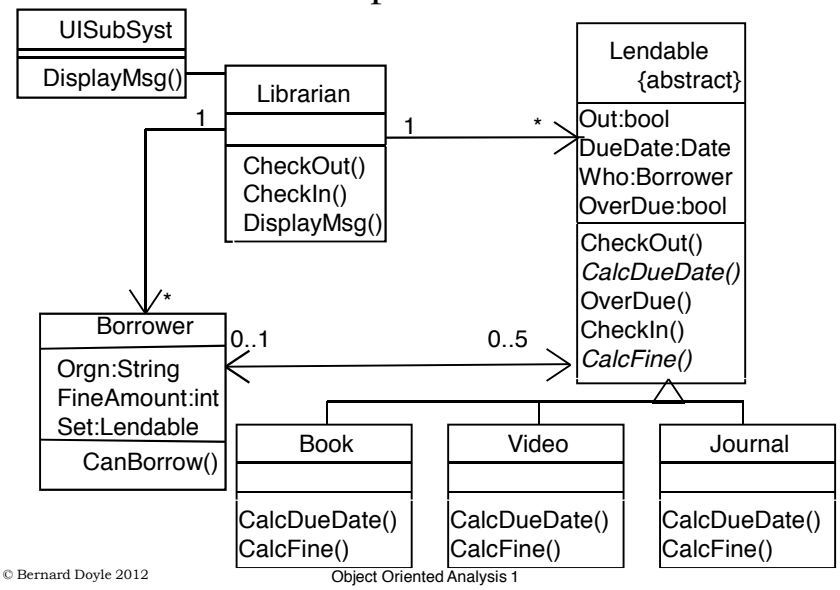
Borrower	
can borrow	Lendable
know set of lendables	
know fine amount	
know organisation	

Librarian	
check out lendable	Lendable, Borrower
check in lendable	Borrower, Lendable
display message	*UI Subsystem*

Cards after Exceptional Scenarios (2)



After Exceptional Scenarios



Discovering New Classes

"What happens when Ned Brooks comes to the Library
in search of a book entitled *The Mythical Mammoth*?"

- Librarian takes responsibility "search for lendable", but where to look?
- Need a Collection class that has responsibility "know set of lendables". Will need as collaborators, DB Subsystem and Book to get the information requested.
- This leads to a discussion of what Books and other lendables will need to support searching. Rather than have a "know" responsibility for each of these, represent this information a *attributes* on the back of eack card.

CRC Cards with Attributes on Back

Lendable: the set of objects that represent items to be borrowed

Attributes:

in/out status	due date
borrower	Dewey decimal #

Book: the set of objects that represent books to be borrowed

Attributes:

title	publication date
author	publisher

Journal: the set of objects that represent journals to be borrowed

Attributes:

name	volume
date	issue #

Video: the set of objects that represent videos to be borrowed

Attributes:

title	date
DVD or Tape	producer

Object Oriented Design

Design Constraints

- When passing from analysis to design, it is necessary to consider constraints that may exist.

  These include:
  - *Target Environment* - the hardware and software under which the application will run.
    - Is it important to isolate hardware/operating systems dependencies to facilitate porting to a different environment?
  - *Language* - what programming language will be used to implement the application? While it is desirable to use an OO language such as C++, a procedural language could be used.

Design Constraints (cont'd)

- *Supporting Software Components* - what packages or libraries will be utilised. These include
  - User Interface packages such as MFC, OWL, Motif etc.
  - Database Management - such as Oracle RDBMS or some OODBMS.
  - Class Libraries for Container classes, strings etc.
- *Performance Requirements* -Are performance requirements so tight that it may be necessary to sacrifice design elegance for speed.
- *Others*
  - Is memory likely to be severely limited.
  - What are the requirements on authenticating users of the system?

Enhancing CRC Syntax

- In the design stage, several enhancements can be made to the information on CRC Cards.

  These include:

  - Sub-responsibilities:
    - To perform a responsibility, it may be appropriate to break this up into steps.
  - Collaborating Responsibility
    - As well as showing the collaborator(s), it is useful to show which of their responsibilities will be called for.
  - Data Passed
    - The data sent to a collaborator to enable the fulfilling of a responsibility can also be shown

Sub-responsibilities

- To illustrate the recording of sub-responsibilities, the "check in Lendable" and "check out Lendable" responsibilities of the Librarian can be broken into discrete steps

Librarian	
check out Lendable	Borrower, Lendable
check in Lendable	Borrower, Lendable
search for Lendable	Collection
display message	*UI Subsystem*
get info from user	*UI Subsystem*

Librarian	
check out Lendable	
check Borrowers status	Borrower
check it out	Lendable
update Borrower	Borrower
check in Lendable	
check if overdue	Lendable
check it in	Lendable
update Borrower	Borrower
record fine	Borrower
search for Lendable	Collection
display message	*UI Subsystem*
get info from user	*UI Subsystem*

Collaborating Responsibility

- The Librarian card can be further enhanced to show which responsibilities of the Borrower and Lendable will be involved:

Librarian	
check out Lendable	
check Borrowers status	Borrower :can borrow
check it out	Lendable : check out
update Borrower	Borrower :know Lend
check in Lendable	
check if overdue	Lendable : know if overdue
check it in	Lendable : check in
update Borrower	Borrower : know Lend.
record fine	Borrower : know fine
search for Lendable	Collection : retrieve
display message	*UI Subsystem*
get info from user	*UI Subsystem*

Collaborating Responsibility (cont'd)

and also the data that will be passed:

Librarian	
check out Lendable	
check Borrowers status	Borrower :can borrow
check it out	Lendable : check out(Borr)
update Borrower	Borrower :know Lend(Lend).
check in Lendable	
check if overdue	Lendable : know if overdue
check it in	Lendable : check in
update Borrower	Borrower : know Lend.(Lend)
record fine	Borrower : know fine(fine)
search for Lendable	Collection : retrieve(info)
display message	*UI Subsystem*
get info from user	*UI Subsystem*

Interface Classes

- It was decided that, although Unix is being used now, a move to Windows or Macintosh is likely soon. Therefore:
  - Introduce a "User Interacter" class to get information from, and send messages to, the user.
  - This provides an interface to whatever user interface system is chosen.

- Also, it is at present uncertain what Data Base Management system will be used, so:
  - Introduce a "DB" class to interact with whatever DBMS is chosen.

Object Lifetimes

Certain questions are important regarding objects of each Class in the application:

- What is the lifetime of the object?
- Who is responsible for creating/destroying the object?
- What happens when it is created and destroyed?

Object Lifetimes – Persistent objects

- The lifetimes of some objects in the Library system, and what they must do on start-up and day's end, are reasonably clear:
  - DB
    - Created on start-up, when Data Base must be opened.
    - Destroyed at shut-down when Data Base closed.
  - Librarian
    - Created and destroyed at start-up and shut-down.
    - Need to add responsibility "wait for user".
  - User Interacter
    - Created and destroyed at start-up and shut-down.
    - Need to add "get employee ID" as responsibility.
      - The need to "get employee ID" raises the problem of verifying this ID. Best to put info about valid ID's with one object - "ID Verifier"

Object Lifetimes – Transient objects

With Lendable and its sub-classes and with Borrower:

- The information represented by these objects has a life much longer than any one run of the Library application.
- The number of things represented is very much greater than are likely to be referenced on any one day.

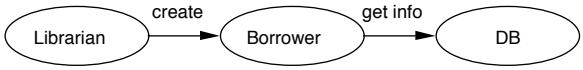
The lifetime of these objects will not exceed the time of interaction with one user.

- Once a User has been verified, a Borrower object must be constructed from information in the Data Base relating to this User ID.
- When the User leaves, the Data Base must be updated and the Borrower destroyed.
- During this period, several Lendable objects may also be created from Data Base information and then destroyed. If necessary the data base is updated before destruction.

When for example the Librarian wishes a Borrower object to be created/destroyed, who should have the responsibility?

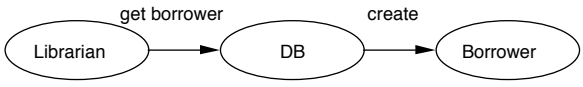
Object Creation/Destruction

- One possibility is for the Borrower class to have create/destroy responsibilities.
  - The constructor could be passed a valid User ID and the ask the DB object for the pieces of information needed.



  - Similarly, the destructor could send to the DB the information that needs to be saved.

- However, at times (e.g. when passed as parameters) temporary objects are created and destroyed without explicit programmer command.
  Better to have DB (which exists to provide interface between application and Data Base) accept responsibility for creating and destroying Borrower (and Lendable) objects.



Check-Out Scenario (1)

What happens when Barbara Stewart, who has one outstanding book, not overdue, checks out a book entitled *Effective C++ Strategies*?"

- *Librarian*: collaborates with User Interacter to "get employee ID".
- *Librarian*: collaborates with ID Verifier to "verify employee ID".
- *ID Verifier* : collaborates with DBMS to verify.
- *Librarian*: passes employee ID to DB to get a Borrower. Sets this as "current borrower".
- *DB*: using "get borrower" responsibility, "get data from DBMS" and collaborate with Borrower to "create borrower".
- *Borrower*: during creation needs a set of previously borrowed Lendables. Asks DB for these.
- *DB*: has set of IDs for the Borrower's Lendables. Creates a Book object to give to Borrower to add to set.

Check-Out Scenario (2)

- *Borrower*: uses "know set of lendables" to add book to set.
  - Decides to implement set of Lendables via a List class.
  - Breaks down "know set of lendables" into:
    - "add lendable to set"
    - "delete lendable from set"
    - "retrieve lendable from set"
    - "number of lendables"
- *Librarian*: uses User Interacter to get choice from user. Choice is "check out". Uses Borrower to "check user status".

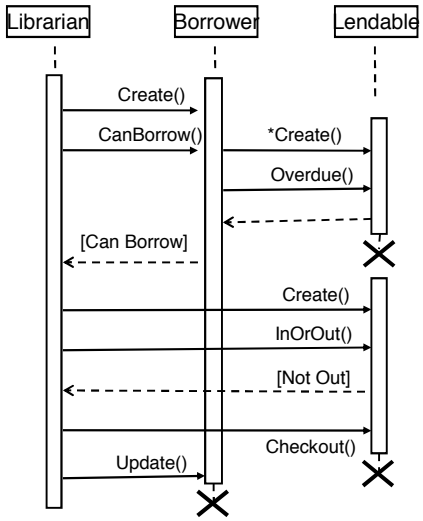
Check-Out Scenario (3)

- *Borrower*: uses “know fine amount” to see that total fine is less than $100. Uses “number of lendables” to see that has only one out. Then must “retrieve lendable from set” and collaborate with Book to see if overdue.
- *Book*: has “know if overdue” as responsibility. Should this be stored or worked out each time? Work it out. Send Due Date to Date
- *Date*: needs to overload operator<(), and can then compare Due Date with Today’s Date. Where is Today’s Date? Add responsibility “know today’s date”.
- *Book*: reports back to Borrower that not overdue.
- *Borrower*: reports to Librarian that can borrow.

Check-Out Scenario (4)

- *Librarian*: ask User Interacter for lendable ID.
- *User Interacter*: adds “get lendable ID” to responsibilities.
- *Librarian*: asks DB to get Lendable.
- *Librarian*: asks Book to check itself out.
- *Book*: set Borrower, set status to OUT. Gets Date to calculate Due Date.
- *Date*: adds responsibility “add days to date”.
- *Librarian*: asks Borrower to add Book to list of lendables. Asks DB to store Book.
- *DB*: stores Book via “put lendable”.
- *Librarian*: ask User Interacter for next request. Gets “exit”. Gives back Borrower to DB.
- *DB*: stores Borrower via “put borrower”.

Check Out Sequence Diagram



Check-In Scenario (1)

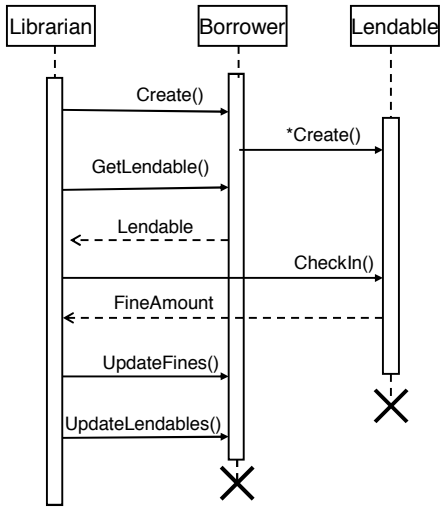
- *Librarian*: gets "current Borrower" as for Check-out. Gets "check in" from User Interacter as user's choice, and also Lendable ID.
  Collaborates with Borrower this time to get Lendable.
- *Borrower*: uses "retrieve lendable from set" to get Lendable.
- *Librarian*: could ask Book if it is overdue and what fine amount is.
  Instead ask Book to check itself in and return amount of Fine if any.
- *Book*: If overdue calculate Fine, change status to IN and clear borrower and due date attributes and return Fine to Librarian.
- *Librarian*: Tell Borrower to update Fine Amount and update its set of Lendables.

BUT what if person returning book is not the person who borrowed it?

Check-In Scenario (2)

- At this stage Librarian class has become very large.
- Add a new Transaction superclass with Check Out and Check In as subclasses and have these take over many of Librarian's responsibilities.
- Since Fines will need to be retrieved for billing it becomes clear that a Fines class is desirable and that the DB class should have responsibility for storing and retrieving these.

Check In Sequence Diagram



Updated Cards – Persistent classes

User Interacter	
get employee ID	GUI subsystem
get user action	GUI subsystem
get lendable ID	GUI subsystem
display message	GUI subsystem
disp. OK/Cancel	GUI subsystem

DB	
create	DBMS
get info DBMS	DBMS
get lendable	Lendable
get Borrower	Borrower, Lendable
put Lendable	Lendable
put Borrower	Borrower
put Fine	DBMS

Librarian	
wait for user	
get/verify emp. ID	User Interacter, ID Verifier
know curr. borrower	DB
create/exec. Trans.	User Interacter, Transaction
get lendable ID	User Interacter
display error mess.	User Interacter

ID Verifier	
verify	DB

Updated Cards – Transactions

Transaction
subclasses: Check In, Check Out
create
know current borrower
execute

Check Out

create	
know current borrower	
know current lendable	DB
execute	
check borrower status	Borrower
check out lendable	Lendable
update borrower	Borrower
display errors	User Interacter
destroy	
put borrower	DB
put Lendable	DB

Check In

create	
know current borrower	
know current lendable	Borrower
get alternate borrower	DB
execute	
display errors	User Interacter
check in lendable	Lendable
create and store fine	Fine, DB
update borrower	Borrower
destroy	
put borrower	DB
put lendable	DB

Updated Cards – Lendable

Lendable	
subclasses: Book, Video, Journal	
calculate due date	Date
calculate fine	Date
check out	Date
check in	
know if overdue	Date
know due date	
know borrower	
know in/out status	

Book, Video, Journal	
superclass: Lendable	
calculate due date	Date
calculate fine	

Updated Cards – Borower, etc.

Borrower	
create	List
can borrow	
know total fine amount	
know number of lendables	List
retrieve lendable from set	List
check overdue lendable	Lendable
add lendable to set	List
delete lendable from set	List

Fine
know borrower
know lendable
know fine amount
know due date
know returned date

Date
know today's date
subtract dates
add days to date

```
program matrix;

 type
  colvector = array[1..3] of real;
  matrix = array[1..3, 1..3] of real;
  determinant = array[1..2,1..2] of real;
  matname = string[20];

 var
  det : real;
  current:integer;
  m,n,o,minv : matrix;
  v, xyz : colvector;

 procedure pause;
  var
   c : char;
 begin
  write('Press return key to continue.......');
  readln(c);
  writeln;
 end;

 procedure getvector (p:matname; var c:colvector);
  var
   temp : colvector;
   r : integer;
 begin
  writeln(p);
  writeln;
  for r := 1 to 3 do
   begin
    write('Enter vector element no.', r : 1, '       ');
    readln(temp[r])
   end;
  c:= temp
 end;

 procedure printname (n :matname);
 begin
  if length(n) = 0 then
   writeln('unnamed')
  else
   writeln(n)
 end;

 procedure printcolvector (s1, s2, s3 :matname ;
         c : colvector);

 begin
```

```
  writeln;
  writeln(s1, c[1] : 10 : 2);
  writeln(s2, c[2] : 10 : 2);
  writeln(s3, c[3] : 10 : 2);
  writeln;
 end;


 procedure printmatrix (title : matname;
         m : matrix);
  var
   r, c : integer;
 begin
  writeln;
  writeln(title);
  for r := 1 to 3 do
   begin
    write('                    ');
    for c := 1 to 3 do
     write(m[r, c] : 10 : 2);
    writeln;
   end;
  writeln;
  writeln;
 end;

procedure minor(m:matrix;row,col:integer;var result:determinant);
var r,c:integer;
    dr,dc:integer;
    p:integer;

begin
     p:=2;
     for r:=1 to 3 do
     begin
         for c:=1 to 3 do
         begin
              if (r<>row) and (c<>col) then
              begin
                   p:=p+1;
                   dr:=p div 2;
                   dc:=p mod 2;
                   result[dr,dc]:=m[r,c]
              end;
          end;
     end;
end;

function evaldeterminant(d:determinant):real;
begin
```

```
     evaldeterminant:=d[1,1]*d[2,2]-d[1,2]*d[2,1]
end;


 procedure transpose (old : matrix; var new : matrix);
  var
   r, c : integer;
 begin
  for r := 1 to 3 do
   for c := 1 to 3 do
    new[c, r] := old[r, c];
 end;

 function determ3x3 (m : matrix) : real;
  var
   minors:array [1..3] of determinant;
   d:array[1..3] of real;
   i:integer;
   temp:real;

 begin
  for i:=1 to 3 do
      begin
      minor(m,1,i,minors[i]);
      d[i]:= evaldeterminant(minors[i]);
      end;
  temp:=0;
  for i:= 1 to 3 do
      begin
      if (odd(i)) then
         temp:=temp+m[1,i]*d[i]
         else
         temp:= temp-m[1,i]*d[i];
      end;
  determ3x3:=temp
 end;

procedure divmat (m : matrix;
         det : real; var result: matrix);
  var
   r, c : integer;
   temp : matrix;

 begin
 if det<>0 then
 begin
  for r := 1 to 3 do
   for c := 1 to 3 do
    temp[r, c] := m[r, c] / det;
  result := temp;
```

```
 end
 else
 writeln('Determinant is Zero. Unable to divide into matrix')
 end;

 procedure getmatrix (p : matname;
         var m : matrix);
  var
   r, c : integer;
 begin
  ClrScr;
  writeln(p);
  writeln;
  for r := 1 to 3 do
   for c := 1 to 3 do
    begin
     write('Enter Row', r : 1, '  Column ', c : 1, '                    ');
     readln(m[r, c])
    end;
  writeln;
  pause;
  ClrScr;
 end;

 procedure getminor (m : matrix; var result:matrix);
  var d:determinant;
      r,c:integer;
begin
  for r:= 1 to 3 do
      for c:= 1 to 3 do
      begin
           minor(m,r,c,d);
           result[r,c]:=evaldeterminant(d)
      end;
 end;


 procedure cofactor (var m,n : matrix);
  var
   r, c : integer;

 begin
      for r:= 1 to 3 do
          for c:= 1 to 3 do
          begin
               if (odd(r+c)) then n[r,c]:=-1*m[r,c]
               else
               n[r,c]:=m[r,c]
          end;
 end;
```

```
procedure mult3x3 (var f,s,q : matrix) ;
 var
  r, c, p : integer;
  sum : real;
  temp : matrix;
begin
 for r := 1 to 3 do
  for c := 1 to 3 do
   begin
    sum := 0;
    for p := 1 to 3 do
     sum := sum + f[r, p] * s[p, c];
    temp[r, c] := sum
   end;
 q := temp;
end;

procedure matxcolvec (m : matrix;
        c : colvector;var result:colvector);
 var
  r : integer;
  temp : colvector;
begin
 for r := 1 to 3 do
  temp[r] := m[r, 1] * c[1] + m[r, 2] * c[2] + m[r, 3] * c[3];
 result := temp
end;


procedure getinverse (m : matrix;
        var inverse : matrix);
 var
  det : real;
  tmp1,tmp2,tmp3: matrix;
begin
 printmatrix('A', m);
 getminor(m,tmp1);
 printmatrix('Minor', tmp1);
 pause;
 cofactor(tmp1,tmp2);
 printmatrix('Cofactor', tmp2);
 pause;
 transpose(tmp2,tmp3);
 printmatrix('Transpose', tmp3);
 pause;
 det := determ3x3(m);
 writeln;
 writeln('Determinant is:', det : 10 : 2);
 if det<>0 then
```

```
  divmat(tmp3, det, inverse)
  else
  writeln('No inverse exists for this matrix')
 end;


 procedure menu (var decision : integer);
  var
   option, x : integer;
 begin
  ClrScr;
  writeln('                              Menu');
  writeln('                              ____ ');
  writeln;
  writeln('Inverse of a matrix                                  1');
  writeln('Determinant of a matrix                              2');
  writeln('Multiply two 3x3 Matrices                            3');
  writeln('Solve set of simultaneous Equations                  4');
  writeln('Multiply 3x3 matrix by vector                        5');
  writeln('Quit                                                 6');
  writeln;
  write('Enter required option..........');
  readln(option);
  if (option > 6) then
   menu(x)
  else
   decision := option;
 end;

procedure doit;
var choice:integer;
begin
 menu(choice);
 case choice of
  1 :
   begin
    getmatrix('Enter matrix.......',m);
    getinverse(m,minv);
    printmatrix('Original Matrix', m);
    printmatrix('Inverse', minv);
    pause;
   end;
  2 :
   begin
    getmatrix('Enter matrix.......', m);
    printmatrix('Matrix is: ', m);
    det := determ3x3(m);
    writeln('Determinant is ', det : 1 : 2);
    pause;
   end;
```

```
  3 :
   begin
    getmatrix('Enter first matrix.........', m);
    getmatrix('Enter second matrix.........', n);
    mult3x3(m,n,o);
    pause;
    printmatrix('First Matrix', m);
    printmatrix('Second Matrix', n);
    printmatrix('Product is:-', o);
    pause;
   end;
  4 :
   begin
    getmatrix('Enter matrix of coefficients....', m);
    getvector('Enter values of RHS of linear equations.....',v);
    getinverse(m,minv);
    matxcolvec(minv,v,xyz);
    printcolvector('X = ', 'Y = ', 'Z = ', xyz);
    pause;
   end;
  5 :
   begin
    getmatrix('Enter 3x3 matrix.....',m);
    getvector('Enter elements of column vector......',v);
    matxcolvec(m,v,xyz);
    printcolvector('1st element = ', '2nd element = ', '3rd element = ',
xyz);
    pause;
   end;
  6 :writeln('Bye......');
 end;
 if choice<>6 then doit
end;

begin
     doit
end.
```

```
function noofrows(m:matrix):integer;
begin
     noofrows:=trunc(m[0,1])
end;

function noofcols(m:matrix):integer;
begin
     noofcols:=trunc(m[1,0])
end;

procedure entermatrix(var m:matrix);
var r,c:integer;
begin
     write('Enter no. of rows   : ');
     readln(input,m[0,1]);
     write('Enter no. of columns: ');
     readln(input,m[1,0]);
     for r:=2 to noofcols(m) do
         m[0,r]:=m[0,1];
     for c:=2 to noofrows(m) do
         m[c,0]:=m[1,0];
     for r:=1 to noofrows(m) do
     begin
         write('Enter row ',r:1,' :');
         for c:= 1 to noofcols(m) do
         begin
              read(input,m[r,c])
         end;
     end;
     writeln;
end;

procedure printmatrix(m:matrix);
var r,c:integer;
begin
     for r:= 1 to noofrows(m) do
         begin
         for c:= 1 to noofcols(m) do
             write(m[r,c]:8:2);
         writeln;
         end;
end;

procedure getrow(m:matrix;rownumber:integer; var r:row);
var c:integer;
begin
     for c:= 0 to noofcols(m) do
     r[c]:=m[rownumber,c];
end;
```

```
procedure getcolumn(m:matrix; colnumber:integer; var c:column);
var r:integer;
begin
     for r:= 0 to noofrows(m) do
     c[r]:=m[r,colnumber];
end;

procedure putrow(var m:matrix; rownumber:integer; r:row);
var c:integer;
begin
     for c:= 1 to noofcols(m) do
         m[rownumber,c]:=r[c];
end;

procedure putcolumn (var m:matrix; colnumber:integer; c:column);
var r:integer;
begin
     for r:= 0 to noofrows(m) do
         m[r,colnumber]:=c[r];
end;

procedure multrow(operand:row; multiplier:extended; var result:row);
var c:integer;
begin
     for c:= 1 to trunc(operand[0]) do
         result[c]:= operand[c]*multiplier;
     result[0]:=operand[0];
end;

procedure multcol (operand:column; multiplier:extended; var result:column);
var r:integer;
begin
     for r:= 1 to trunc(operand[0]) do
         result[r]:= operand[r]*multiplier;
end;

procedure subtractrow(first,second:row; var result:row);
var c:integer;
begin
     if trunc(first[0])<>trunc(second[0])
        then writeln('Unable to subtract.....')
        else
        begin
             result[0]:=first[0];
             for c:=1 to trunc(first[0]) do
                 result[c]:=first[c]-second[c];
        end;
end;

procedure multmatrix(first,second:matrix; var result:matrix);
```

```
var r,c,m:integer;
    temp:extended;
begin
     if noofcols(first)<>noofrows(second)
     then writeln('No product exists...')
     else
     begin
          result[1,0]:=noofcols(second)/1;
          result[0,1]:=noofrows(first)/1;
               for r:= 1 to noofrows(first) do
               begin
                    for c:=1 to noofcols(second) do
                    begin
                    temp:=0;
                    for m:=1 to noofcols(first) do
                    temp:=temp+first[r,m]*second[m,c];
                    result[r,c]:=temp;
                    end;
               end;
     end;
end;

procedure makeidentity(var I:matrix; t:extended);
var r,c:integer;
begin
     for r:=1 to trunc(t) do
         for c:= 1 to trunc(t) do
              if r=c then I[r,c]:=1
                 else I[r,c]:=0;
     for r:=1 to trunc(t) do
         begin
              I[0,r]:=t;
              I[r,0]:=t;
         end;
end;

procedure printrow(r:row);
var p:integer;
begin
     for p:= 1 to trunc(r[0]) do
         write(r[p]:5:2);
     writeln
end;

procedure exchangerows(var m:matrix; r1,r2:integer);
var first,second:row;
begin
     getrow(m,r1,first);
     getrow(m,r2,second);
     putrow(m,r1,second);
```

```
     putrow(m,r2,first)
end;

function nextnonzeroincol(m:matrix; r,c:integer):integer;
var found:boolean;
begin
     found:=false;
     r:=r+1;
     while (r<noofrows(m)) and (not found) do
     begin
          if m[r,c]<>0 then
             found:=true
             else
             r:=r+1;
     end;
     if r>noofrows(m)
     then nextnonzeroincol:=0
     else nextnonzeroincol:=r
end;

procedure pause;
var c:char;
begin
write('press return key to continue....');
readln(c)
end;

function poslargestincol(c:integer;m:matrix):integer;
var temp:extended;
    i,pos:integer;
begin
     pos:=1;
     temp:=m[1,c];
     for i:=1 to noofrows(m) do
     if m[i,c]>temp then
     begin
     pos:=i;
     temp:=m[i,c]
     end;
     poslargestincol:=pos
end;

procedure arrangecols(var m,n:matrix);
var temp:integer;
begin
     temp:=poslargestincol(1,m);
     if temp<>1 then
     begin
          exchangerows(m,1,temp);
          exchangerows(n,1,temp);
```

```
     end;
end;

procedure GaussInvert(m:matrix; var I:matrix);
var singular:boolean;
    r,c,tempint:integer;
    factor:extended;
    D:array[1..Maxcol] of extended;
    mrow,Irow,Icurrent,mcurrent:row;
begin
     singular:=false;
     makeidentity(I,m[0,1]);
     for c:= 1 to noofcols(m) do
     begin
         if m[c,c]=0.00000000 then
         begin
              tempint:=nextnonzeroincol(m,c,c);
              if tempint>0 then
              begin
                   exchangerows(m,tempint,c);
                   exchangerows(I,tempint,c)
              end
              else
              begin
              writeln('Matrix is singular. No Inverse exists');
              singular:=true;
              end;
         end;
         if not singular then
         begin
         for r:= 1 to noofrows(m) do
                  begin
                  if (r<>c) and (m[r,c]<>0.0000000000) then
                  begin
                       factor:=m[c,c]/m[r,c];
                       getrow(m,r,mrow);
                       getrow(I,r,Irow);
                       getrow(m,c,mcurrent);
                       getrow(I,c,Icurrent);
                       multrow(mrow,factor,mrow);
                       multrow(Irow,factor,Irow);
                       subtractrow(mrow,mcurrent,mrow);
                       subtractrow(Irow,Icurrent,Irow);
                       putrow(I,r,Irow);
                       putrow(m,r,mrow);
                   end;
                   end;
                   end;
                   end;
         if not singular then
```

```
          for r:= 1 to noofrows(I) do
          begin
               factor:=m[r,r];
               for c:= 1 to  noofcols(I) do
                       I[r,c]:=I[r,c]/factor;
          end;
end;
```

```
program solve;

{$E+}
{$N+}

Uses CRT;

{$I matext.def}

var M,I,C,R:matrix;


{$I matext.fun}

begin
     ClrScr;
     writeln;
     writeln('Solves up to 10 Simultaneous Linear Equations':60);
     writeln;
     writeln('Enter Matrix of Coefficients (i.e. LHS of equations) :-');
     entermatrix(M);
     writeln;
     writeln('Enter Column of constants (i.e. RHS of equations) :-');
     entermatrix(C);
     arrangecols(M,C);
     GaussInvert(M,I);
     multmatrix(I,C,R);
     writeln;
     writeln('Results are :- ');
     printmatrix(R);
     pause
end.
```

Templates

There are times when we wish to have

a FUNCTION which manipulates objects of different classes
in the same way, or

a CLASS which has data members of a class which is
specified by the user.

Template Functions

- There are certain operations such as sorting, taking the maximum etc. which we may wish to do
  - on different classes of objects, *and*
  - in the same sort of way.

  We will take the obtaining of the maximum of an group of objects, whose addresses are in an array, as an example.

- We could overload a function many times with one version for each class of object that we wish to operate on e.g.

```
Person* Maximum(Person* persons[], int number);
Car* Maximum(Car* cars[], int number);
```

  etc.

- The alternative is to write the function once, but as a Template Function.

Defining a Template Function

- A template function to find the maximum of an array of objects:

```
template <class S>
 S* Maximum(S* things[], int number){
    S* max = things[0];
    for(int i = 1; i < number; i++)
       if(*things[i] > *max)
          max = things[i];
    return max;
 }
```


Using a Template Function

A function call:

```
Person* greatest;
     Person* people[20];
     . . . . .
     greatest = Maximum(people, 20);
```

will cause the *instantiation (generation)* and *compilation* of the following function

```
 Person* Maximum(Person* things[], int  number){
    Person* max = things[0];
    for(int i = 1; i < number; i++)
       if(*things[i] > *max)
          max = things[i];
    return max;
 }
```

This function will then be called with parameters **`people`** and **`20`**.

Note that a compilation error will occur if the operation ">" has not been defined for class Person.

Using a Template Function

The following code segment (which assumes that operator **">"** has been defined for Person and Car classes):

```
Person* maxCust;
Person* maxEmp;
Car*    maxCar;
Person* customers[20];
Person* employees[30];
Car*    fleet[10];
. . . . .
maxCust = Maximum(customers, 20);
maxEmp  = Maximum(employees, 30);
maxCar  = Maximum(fleet, 10);
```

will cause the instantiation and compilation of TWO functions:

```
Person* Maximum(Person* things[], int  number);
Car*    Maximum(Car* things[], int  number);
```

The first two calls will reference the first function. The third call will reference the second function

Template Class

- A template class is used when we wish to declare a class which has *data members* which are objects of an unspecified class.
- The most common use for a template class is for "containers"
- We will illustrate the implementation of a template class by means of a Queue container class.
- The actual objects in the list will be contained in a QueueItem class.

(This example is adapted from C++ Primer 3rd Edition, by Lippman and Lajoie, Ch 16)

Declaration of QueueItem

```
        // In file Queue.h
#ifndef QUEUE_H
#define QUEUE_H
#include <iostream>
using namespace std;
#include <cstdlib>

template <class T> class Queue; //forward declaration

template <class Type>
class QueueItem {
   public:
      friend class Queue<Type>;
      friend ostream& operator<<(ostream &,const QueueItem<Type> &);
      friend ostream& operator<<(ostream &, const Queue<Type> &);
   private:
      QueueItem( const Type &t ) : item(t), next(0) { }
      Type item;
      QueueItem *next;
};
```


Declaration of QueueItem - notes

- Since `QueueItem` is only relevant to the `Queue` class, the constructor is made `private`. and the `Queue` class is made a `friend`.
  - Since the declaration of `QueueItem` precedes that of `Queue` , a forward declaration of `Queue` is required.
  - Now `QueueItem`s can only be created by member functions of the `Queue` class.
- The operator to output a `QueueItem` is declared as a `friend` in the normal way.
- Since the output operator for the `Queue` needs to follow the pointers of `QueueItem`s, this must also be declared as a `friend`.
- Note that, within the declaration of member functions or data members of `template<class Type> class QueueItem:`
  - the name `QueueItem` can be used without specifying the `template` parameter.
  - But reference to any other `template` class requires the `template` parameter.

Declaration of Queue

```
        // In file Queue.h

template <class Type>
class Queue {
     friend ostream& operator<<( ostream &, const
   Queue<Type> & );
   public:
      Queue() : front( 0 ), back ( 0 ) { }
      ~Queue();
      Type remove();
      void add( const Type & );
      bool is_empty() const {
         return front == 0;
      }
   private:
      QueueItem<Type> *front;
      QueueItem<Type> *back;
};
```


Declaration of Queue - notes

- The `Queue` has just two data members, pointers to the first (**`front`**) and last (**`back`**) `QueueItem<Type>`s in the `Queue`.
- **`front`** points to the first item to be removed, and **`back`** points to the last item added.
- As with other classes, simple member functions can be defined in the declaration and will become in-line functions.
- The null constructor (the only constructor), sets both pointers to 0 (NULL). If either of these pointers is 0, then the `Queue` must be empty.

Queue Member Functions

// In file Queue.h

```
template <class Type>
Queue<Type>::~Queue(){
   while ( ! is_empty() )
   remove();
}

template <class Type>
void Queue<Type>::add(
         const Type &val ){
   // allocate a new QueueItem
   //    object
   QueueItem<Type> *pt = new
      QueueItem<Type>( val );
   if ( is_empty() )
      front = back = pt;
   else{
      back->next = pt;
      back = pt;
   }
}
```

```
template <class Type>
Type Queue<Type>::remove(){
   if ( is_empty() ){
      cerr << "remove() on empty"
           << "queue\n";
      exit( -1 );
   }
   QueueItem<Type> *pt = front;
   front = front->next;
   Type retval = pt->item;
   delete pt;
   return retval;
}
```


Queue Member Functions - notes

- Note that the class declarations and the function implementations are in the same (.h) file. The functions can not be compiled until they are instantiated by specifying an explicit type for the template parameter.
- The source for the functions must be `#include`'d in any .cpp file referencing a `Queue`.

Output Operators

// In file Queue.h

```
template <class Type>
ostream& operator<<(
   ostream &os,
   const QueueItem<Type> &qi)
{
   os << qi.item;
   return os;
}
```

```
template <class Type>
ostream& operator<<(
   ostream &os,
   const Queue<Type> &q )
{
   os << "< ";
    QueueItem<Type> *p;
   for ( p = q.front; p != 0;
         p = p->next )
      os << *p << " ";
    os << " >";
   return os;
}
#endif
```


Using Queue

```
#include "Queue.h"
using namespace std;
int main() {
   Queue<int> qi;       // Queue instantiated
   cout << qi << endl;  // both output operators instant'd
   int ival;
   for ( ival = 0; ival < 10; ++ival )
      qi.add( ival );
    cout << qi << endl;
   int err_cnt = 0;
   for ( ival = 0; ival < 10; ++ival ) {
      int qval = qi.remove();
      if ( ival != qval ) err_cnt++;
   }
   cout << qi << endl;
   if ( err_cnt == 0 ) cout << "!! queue executed ok\n";
   else     cout << "?? queue errors: " << err_cnt << endl;
   return 0;
}
```

Using Queue - notes

- The `Queue` template class is instantiated with the template parameter type as `int`.
- The (empty) `Queue` is output.
  - This will cause the instantiation of the `Queue` output template function.
  - This uses the `QueueItem` output function so that this will also be instantiated.
  - In both cases the template parameter will be `int`, the same as the class to which they are `friend`s.
- The values 0 – 9 are the added to the `Queue`, which is then printed out again.
- All values are then removed from the `Queue` and compared with what the value should be. Any discrepancies would cause an error count to be incremented.
- The number of errors (which should be zero) is then reported.