Object Oriented Programming
The C++ programming language

Programming evolution

Machine code

- programs in binary code, executed directly by the processor

Assembly languages

- still low level programming, replaced machine code functions with mnemonics and memory addresses with symbolic labels

Procedural programming

- decompose programs into procedures/functions

Modular Programming

- decompose programs into modules

Object Oriented Programming

- decompose program into a set of objects
- objects interact with each other to form a system

Object Oriented Paradigm

Provide flexible and powerful abstraction

- Allow programmers to think in terms of the structure of the problem rather than in terms of the structure of the computer.
- Decompose the problem into a set of objects
- Objects interact with each other to solve the problem
- create new type of objects to model elements from the problem space

An object is an entity that:

- has a local state
- able perform some operation (behavior)

It may be viewed as a combination of:

- data (attributes)
- procedural elements (methods)

Object Oriented Programming is a method of implementation where:

- objects are fundamental building blocks
- each object is an instance of some type (class)
- classes are related to each others by inheritance

Fundamental concepts and properties

Concepts:

- object
- class
- method (message)

Properties:

- encapsulation
- inheritance
- polymorphism

C++ Programming Language

Why C++:

- widely used, both in industry and in education
- is a high level programming language
- hybrid language, implements all the concepts needed for object oriented programming
- many programming languages are based on C++ (Java, C#). Knowing C++ makes learning other programming languages easier

Advantages:

- **Conciseness**: allow us to express common sequences of commands more concisely.

- **Performance:** program written in c++ usually run faster than programs written in other languages

- **Maintainability**: modifying code is easier when it entails just a few text edits,instead of rearranging hundreds of processor instructions.

- **Portability**: can be used to write programs for nearly any processor

- **Productivity**: The goal of C++ is improved productivity (over C).

Compiled Languages. The compilation Process

A program goes from text files(or source files)to processor instructions as follows:

Source file - text file, contain the program in a programming language

| - compiler, parse the source files and create object files

Object File - intermediate files that represent an incomplete copy of the program

| - linker, combine multiple object files and create the program that can be executed

Executable

| - operating system, load the executable file into the memory and execute the program

Program in memory

In C++, all these steps are performed ahead of time, before you start running a program. In some languages, they are done during the execution process, which takes time. This is one of the reasons C++ code runs far faster than code in many more recent languages.

Python (Interpreted) vs C++ (compiled)

Integrated Development Environment for C++

Compiler

MinGW - - Minimalist GNU for Windows, a native Windows port of the GNU Compiler Collection (GCC)
MinSYS - is a collection of GNU utilities such as bash, make, gawk and grep

Eclipse IDE

Eclipse CDC provides a fully functional C and C++ Integrated Development Environment based on the Eclipse platform.

C++ Project

Hello World sample application

Lexical elements

C++ is case sensitive (a <> A)

Identifier:

- Sequence of letters and digits, start with a letter or “_”(underline).
- Names of things that are not built into the language
- Ex. i, myFunction, rez,

Keywords (reserved words):

- Identifier with a special purpose
- Words with special meaning to the compiler
- int, bool, for,

Literals:

- Basic constant values whose value is specified directly in the source code
- Ex. “Hello”, 72, 4.6, ‘c’

Operators:

- Mathematical or logical operations
- +,-, <<

Separators:

- Punctuation defining the structure of a program
- ; { } , ( )

Whitespace:

- Spaces of various sorts; ignored by the compiler
- space, tab, new line

Comments:

- ignored by the compiler
- // this is a single line comment

- ```
  /*   This is a
   *    multiline comment
  */
  ```
  ```

Data types

A **type** is a domain of values and a set of operations defined on these values.

Built in data types:

Name	Description	Size	Range
char	Character or small integer.	1byte	signed: -128 to 127 unsigned: 0 to 255
short int (short)	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
int	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	Boolean value. It can take one of two values: true or false.	1byte	true or false
float	Floating point number.	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	Double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	Long double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
wchar_t	Wide character.	2 *or* 4 bytes	1 wide character

- operations: +, -, *, /, %
- relations: <, >, <=, >=, ==, !=
- short/long can be used to change the domain of the type
- signed/unsigned can be used to change how the value is interpreted (the sign bit)
- operations can be performed only on compatible types. Ex. Can't take the remainder of an integer and a floating-point number

Arrays

If T is an arbitrary basic type:

- T[n] - is an array of length n with elements of type t
- indexes are from 0 to n-1
- indexing operator: [ ]
- compare 2 arrays by comparing the elements
- multidimensional arrays: t[n][m]

Record

struct name{
  type1 field1;
  type2 field2
}
	struct car{
	  int year;
	  int nrKm;
	}
		car c;
		c.year = 2010
		c.nrKm = 30000;

Variable declaration

- Introduce a name into the program and associate a type to the name.
- Memory is allocated according to the type of the variable.
- Variable is a named location in memory
- The type tell the compiler how much memory to reserve for it and what kinds of operations may be performed on it
- The value is undefined until the variable is initialized
- Can combine (recommended) the declaration with the initialization
- Use meaningful names for the variables
- Always initialize the variables with a meaningful value

<type> <identifier>
Ex. int i
     long rez
     bool val
     int j=7

Constants

- numeric constants: 1, 12, 23.5
- string constants: “Hello World”
- character: ‘c’

Statements

A statement is a unit of code that does something –a basic building block of a program.
An expression is a statement that has a value.
Every statement is ended by: ; (except the compound statement)

Empty statement: ;

Compound statement:

```
{
//multiple statements here
}
```

Assignment

- Assignment operator: =
- Assign a value to a variable (initialize or change the value of a variable)

Conditional: if, if-else and else if

```
if (condition){
    //statements executed only if the condition is true
}
```

The condition is some expression whose value is being tested. If the condition resolves to a value of true, then the statements are executed before the program continues. Otherwise, the statements are ignored. If there is only one statement, the curly braces may be omitted.

```
if (condition){
    //statements executed if the condition is true
} else {
   //statements executed only if the condition is not true
}
```

```
if (condition1){
    //statements executed if the condition1 is true
} else if (condition2){
   //statements executed only if condition1 is not true and the condition2 is true
}
```

- condition, condition1, condition2 - are expressions
- any expression has a numeric value
- a value equal with 0 is false, any other value is true

Conditional: switch-case

```
switch(expression)
{
   case constant1:
         statementA1
         statementA2
         ...
    break;
    case constant2:
        statementB1
        statementB2
         ...
        break;
     ...
   default:
       statementZ1
       statementZ2
       ...
}
```

The switch evaluates expression and, if expression is equal to constant1, then the statements beneath case constant 1: are executed until a break is encountered. If expression is not equal to constant1, then it is compared to constant2. If these are equal, then the statements beneath case constant 2: are executed until a break is encountered. If not, then the same process repeats for each of the constants, in turn. If none of the constants match, then the statements beneath default: are executed

Loops

Loops execute certain statements while certain conditions are met.

while and do-while

```
while(condition)
{
   statement1
   statement2
   …
}
```

As long as condition holds, the block of statements will be repeatedly executed. If there is only one statement, the curly braces may be omitted

```
do
{
   statement1
   statement2
   …
}
while(condition);
```

The block of statements is executed and then, if the condition holds, the program returns to the top of the block. Curly braces are always required.

for loop

```
for(initialization; condition; incrementation)
{
  //body
}
```

initialization - initialise one or more variables
incrementation - executed after each iteration
The for loop is designed to allow a counter variable that is initialized at the beginning of the loop and incremented (or decremented) on each iteration of the loop. As long as condition holds, the block of statements will be repeatedly executed.
A for loop can be expressed as a while loop and vice-versa.

for loop	while loop

for(*initialization*; *condition*; *incrementation*)
{
  statement1
  statement2
  …
}
	initialization
	while(*condition*)
	{
	 statement1
	 statement2
	 …
	 incrementation
	}

Functions

A function is a group of related instructions (statements) which together perform a particular task. The name of the function is how we refer to these statements.
The *main* function is the starting point for every C++ program

Declaration

<result type> name (<parameter list>);

<result-type> - is the type of the result, and can be any data type. *void* if the function does not return a value
<name> - the name of the function
<parameter-list> - definitions of the variables involved as parameters, separated be comma (formal parameters)

The function body is not part of the declaration.

Definition

```
<result type> name(<parameter list>){
//statements - the body of the function
}
```

- **return** <exp> gives the expression value as the result of the function and unconditionally gets out from the function
- a function that returns a result (not void) must include at least one return statement.
- The declaration need to match the function definition (parameter name may differ)

Specification

- meaningful name for the function
- short description of the function (the problem solved by the function)
- meaning of each input parameter
- conditions imposed over the input parameters (precondition)
- meaning of each output parameter
- the relation between the input and output parameters (postcondition)

```
/*
* Verify if a number is prime
* nr - a number, nr>0
* return true if the number is prime (1 and nr are the only dividers)
*/
bool isPrime(int nr);
```

A **precondition** is a condition that must be true just prior to the execution of some section of code.
A **postcondition** is a condition that must be true just after the execution of some section of code.

Function invocation

name (<parameter list>);

- All argument expressions are evaluated before the call is attempted
- The list of actual parameters need to match the list of formal parameters (types)
- function declaration need to occur before invocation

Overloading

- Multiple functions with the same name, but different arguments
- The function called is the one whose arguments match the invocation

Visibility scope

Scope: where a variable was declared, determines where it can be accessed from

A compound statement ({ }) have its own scope: variables, functions declared inside a compound statement will be visible only inside the compound statement

- function have their own scopes (variables defined inside the function will be visible only in the function, destroyed after the function call
- Loops and if/else statements also have their own scopes
- Cannot access variables that are out of scope
- Variables defined outside of any function are global variables. Can be accessed from any function.

Pass by value vs Pass by reference

Parameters with the type one of the built in type are passed by value.

- on function call c++ makes a copy of the actual parameter;
- changes to the variable within the function don't occur outside the function
- If you want to modify the original variable as opposed to making a copy, pass the variable by reference (**int &a** instead of **int a**)

Arrays are passed by reference.

Functions

A function is a group of related instructions (statements) which together perform a particular task. The name of the function is how we refer to these statements.
The *main* function is the starting point for every C++ program

Declaration (Function prototype)

<result type> name (<parameter list>);

<result-type> - is the type of the result, and can be any data type. *void* if the function does not return a value
<name> - the name of the function
<parameter-list> - definitions of the variables involved as parameters, separated be comma (formal parameters)

The function body is not part of the declaration.

```
/**
 * Computes the greatest common divisor of two positive integers.
 * a, b integers, a,b>0
 * return the the greatest common divisor of a and b.
 */
int gcd(int a, int b);
```

Definition

<result type> name(<parameter list>){
//statements - the body of the function
}

- **return** <exp> gives the expression value as the result of the function and unconditionally gets out from the function
- a function that returns a result (not void) must include at least one return statement.
- The declaration need to match the function definition (parameter name may differ)

```
/**
 * Computes the greatest common divisor of two positive integers.
 * a, b integers, a,b>0
 * return the the greatest common divisor of a and b.
 */
int gcd(int a, int b) {
	if (a == 0 || b == 0) {
		return a + b;
	}
	while (a != b) {
		if (a > b) {
			a = a - b;
		} else {
			b = b - a;
		}
	}
	return a;
}
```

Specification

- meaningful name for the function
- short description of the function (the problem solved by the function)
- meaning of each input parameter
- conditions imposed over the input parameters (precondition)
- meaning of each output parameter
- the relation between the input and output parameters (postcondition)

```
/*
* Verify if a number is prime
* nr - a number, nr>0
* return true if the number is prime (1 and nr are the only dividers)
*/
bool isPrime(int nr);
```

A **precondition** is a condition that must be true just prior to the execution of some section of code.
A **postcondition** is a condition that must be true just after the execution of some section of code.

Function invocation

name (<parameter list>);

- All argument expressions are evaluated before the call is attempted
- The list of actual parameters need to match the list of formal parameters (types)
- function declaration need to occur before invocation

```
int d = gcd(12, 6);
```

Overloading

- Multiple functions with the same name, but different arguments
- The function called is the one whose arguments match the invocation

```
int g() {
	return 10;
}

int g(int a) {
	return a * 10;
}
```

Variable scope and Lifetime

Scope: where a variable was declared, determines where it can be accessed from

Local variables

A compound statement or block ({ }) have its own scope: variables, functions declared inside a block will be visible only inside the compound statement

- function have their own scopes (variables defined inside the function will be visible only in the function, destroyed after the function call
- Loops and if/else statements also have their own scopes
- Cannot access variables that are out of scope (compiler will signal an error)
- A variable lifetime begins when it is declared and ends when it goes out of scope (destroyed)

Global Variables

- Variables defined outside of any function are global variables. Can be accessed from any function.
- The scope is the entire application
- Not use global variales unless you have a very good reason to do so (usually you can find better alternatives)

Pass by value vs Pass by reference

Parameters with the type one of the built in type are passed by value (default).

- on function call c++ makes a copy of the actual parameter, the original variable is not affected by the change made inside the function
- changes to the variable within the function don't occur outside the function
- If you want to modify the original variable as opposed to making a copy, pass the variable by reference (**int &a** instead of **int a**)
- the function call is the same
- You can only pass variables by reference (not expressions or constats)

Procedural version for Calculator

Problem statement

A teacher needs a program for students who learn or use rational numbers.
The program shall help students to make basic arithmetic operations.

Arrays

An array is a fixed number of elements of the same type stored sequentially in memory

```
type arrayName[dimension];
```

When you declare an array you have to provide the size
The array size(dimension) will not change automatically to accomodate more elements

Use a constant for the size of the array (will be easier to change the code)
Array index start from 0 until nrElements-1

After an array declaration the elements of the array are un-initialised

The elements of an array must be initialized before they can be used; otherwise we will almost certainly get unexpected results in our program

```
const int SIZE = 10;
int a[SIZE];
//initialize the array
for (int i=0;i<SIZE;i++){
   a[i] = 0;
}
```

Multidimensional arrays

type arrayName[dimension1][dimension2];

```
const int M = 4;
const int N = 2;
int matrix[M][N];
for (int i = 0; i < M; i++) {
      for (int j = 0; j < N; j++){
            matrix[i][j] = i + j;
      }
}
cout << matrix[3][1];
```

```
int matrix2[M][N] = {{1,2},{3,4},{5,6},{7, 8}};
for (int i = 0; i < M; i++) {
      for (int j = 0; j < N; j++) {
            cout<<matrix2[i][j];
      }
}
```

Arrays can also be passed as arguments to functions
Array function parameters are passed by reference (address)

Strings

String literals are actually represented by C++ as a sequence of characters in memory.
A string is simply a character array and can be manipulated as such.

```
char str[] ={'H','e','l','l','o','\0'};
cout<<str;
cout<<str[4];
```

```
char str2[] = "Hello";
cout<<str2;
cout<<str2[4];
```

The character array helloworld ends with a special character known as the null character.
This character is used to indicate the end of the string

Modular programming in C++.

Module is a collection of functions and variables that implements a well defined functionality

Header files. Libraries.

Function prototypes (function declaration) are grouped into a separate file called header file.

Libraries/modules are generally distributed as the header file containing the prototypes, and a binary .dll/.so file containing the (compiled) implementation

- Don’t need to share the source code (.cpp)

Library user only needs to know only the function prototypes (in the header file), not the implementation source code (in the .cpp file)
Separates the specification, definition of the function from its implementation

The Linker takes care of locating the implementation of functions in the .dll file at compile time

C++ header file conventionally has a file name extension of ‘h’, such as headerfile.h

Include

To include a header file, use the **#include** preprocessor directive.
This tells the preprocessor to open the named header file and insert its contents where the #include statement appears.

A #include may name a file in two ways: in angle brackets (< >) or in double quotes:
#include <header> // search the file in a way that "include search path" in the system directories
#include "local.h" //search for the file relative to the current directory

Create modular programs

The code of an C++ program is split into a number of source files: .h and .cpp

- **.h** file containing the definition (the interface)
- **.cpp** file containing the implementation

Advantage: the .cpp files can be compiled separately (for error checking and testing)

Whenever a .h file is changed all the files that include it (directly or indirectly) must be recompiled.

The header file is a CONTRACT between the developer and the client of the library that describes the data structures and states the arguments and return values for function calls

The compiler enforces the contract by requiring the declarations for all structures and functions before they are used => include the header file

Multiple declaration

If a declaration belongs to a header file, it is possible for the file to be included more than once in a complex program (eg: iostreams) ->error (multiple declaration)

Solution: use the pre-processor directives

Pre-processor directives: #ifdef, #ifndef, #define, #endif

The label can be tested by the pre-processor to see if it has been defined

```
#ifndef RATIONAL_H_   /* verify if RATIONAL_H_ is already defined, the rest
                          (until the #endif will be processed only if RATIONAL_H_ is
                          not defined*/
#define RATIONAL_H_   /* define RATIONAL_H_ so next time the prepocessor will not
                        include this */

/**
 * New data type to store rational numbers
 */
typedef struct {
      int a, b;
} Rational;

/**
 * Compute the sum of 2 rational numbers
 * a,b rational numbers
 * rez - a rational number, on exit will contain the sum of a and b
 */
void sum(Rational nr1, Rational nr2, Rational &rez);

#endif /* RATIONAL_H_ */
```

Modular version of the Calculator

Pointers and references.

A variable declaration will associate a name with a type and a memory location.
When the variable is used somewhere in the code the compiler:
- Lookup the address that the variable name corresponds to
- Go to that location in memory and retrieve or set the value it contains
- will use the type to figure out the domain, memory size, semantics of the bits and the possible operations that can be performed

Reference operator ('&' ampersand)

&x evaluates to the address of x in memory.

```
int x = 7;
cout << x << " memory address is:" << &x;
cout << " size is:" << sizeof(x) << "\n";
```

```
int v[3] = { 1, 2, 3 };
cout << v[0] << " memory address is:" << &v[0];
cout << " size is:" << sizeof(v[0]) << "\n";
cout << v[1] << " memory address is:" << &v[1];
cout << " size is:" << sizeof(v[1]) << "\n";
cout << v[2] << " memory address is:" << &v[2];
cout << " size is:" << sizeof(v[2]) << "\n";
```

Dereference operator ('*' star)

*(&x) takes the address of x and dereferences it – it retrieves the value at that location in memory. *(&x) thus evaluates to the same thing as x

```
int x = 7;
cout << x << " memory address is:" << &x;
cout << x << " the value is:" << *(&x) << "\n";
```

Pointers

Pointers are just variables storing integers representing memory addresses
Memory addresses, or pointers, allow us to manipulate data much more flexibly; manipulating the memory addresses of data can be more efficient than manipulating the data itself

A pointer that stores the address of some variable x is said to point to x. We can access the value of x by dereferencing thepointer

Pointer declaration

data_type *pointer_name;

pointer_name variable can hold a memory adress.
We can access the value by dereferencing it using the * operator
Without the * operator, the identifier x refers to the pointer itself, not the value it points to.

```
int x = 7;
//declare and initialise the pointer p
//p and x refer the same memory location
int* p = &x;
//print the value stored in the memory location
cout << *p << "\n";
//print the address of the memory location
cout << p << "\n";
//modify the value through x
x = 8;
cout << *p << "\n";
//modify the value through p
*p = 10;
cout << *p << "\n";
return 0;
```

Null pointer

Any pointer set to 0 is called a **null pointer**, and since there is no memory location 0, it is an invalid pointer.
Pointers are often set to 0 to signal that they are not currently valid.
We should generally check whether a pointer is null before dereferencing it.

Dangling pointer

Some pointers do not point to valid data; dereferencing such a pointer is a runtime error.
Dereferencing pointers to data that has been erased from memory usually causes runtime errors.
dangling pointer (points to memory whose contents are undefined)

```
int* f() {
	int localVar = 7;
	cout << localVar << "\n";
	return &localVar;
}

int main() {
	int* badP = f();
	cout << *badP << "\n";
}
```

References

`data_type &reference_name;`

```
int y = 7;
int &x = y; //make x a reference to, or an alias of, y
```

Changing x will change y and vice versa, they are just 2 names for the same memory location
References are really similar with pointers:
References are pointers that are dereferenced automatically when you use them
Can not change the location to which a reference points.

Pointers and arrays

An array is actually a pointer to the first element in the array

- arrays are passed by reference. Passing an array is realy passing a pointer
- the array indices start at 0. The first element is 0 away from the start of the array. array[3] – the compiler computes the element that is 3 away from the starting element.

Pointer aritmethic

is a way of using subtraction and addition of pointers to move around between locations in memory.

```
int t[3] = { 10, 20, 30 };
int *p = t;
//print the first elem
cout << *p << "\n";
//move to the next memory location (next int)
p++;
//print the element (20)
cout << *p << "\n";
```

p++ the compiler will go to the next memory location, depending on the type the compiler will use the appropiate step size.

Dynamic allocation

Whenever we declare a new local variable (int x), memory is allocated on a region of memory called **stack**, the memory is freed up by the compiler when the variable goes out of scope.

```
int f(int a) {
   if (a>0){
      int x = 10; //memory for x is allocated on the stack
   }
   //here x is out of scope and the memory allocated for x is no longer reserved
   //the memory can be reused
   return 0;
}
```

```
int f(int a) {
   int *p;
   if (a>0){
      int x = 10;
      p = &x;
   }
   //here p will point to a memory location that is no longer reserved
   cout << *p; //undefined behavior, the program may crash
   return 0;
}
```

The compiler will take care of the allocation/dealocation.

Dynamic allocation

type_name *variable_name = **new** type_name;

new T;

- memory is allocated to store a value of type T
- it returns the address of the memory location
- the return value has type T*
- the memory is allocated on the **Heap**
- **the memory remain allocated until you manually de-allocate it**

delete p;

- precondition: p is of type T* and p had been obtained through new;
- the memory space alocated to the variable p is free;
- always deallocate everything that has ever been allocated.

```
int *p1 = new int; //value from address p1 is not initialised
int *p2 = new int(3); //value from address p2 is 3
int *p3 = new int[4]; //a vector of 4 integer elements is alocated

//free the memory
delete p1;
delete p2;
delete[] p3;
```

Memory leak

Memory leak: when a program allocates memory but is unable to de-allocate it

```
void f() {
	int *p;
	for (int i = 0; i < 10; i++) {
		p = new int;//allocate memory for an int on the heap
		*p = i * 2;
		cout << *p;
	}
	delete p;//de-allocate memory (WRONG)
}
void f() {
	int *p;
	for (int i = 0; i < 10; i++) {
		p = new int; //allocate memory for an int on the heap
		*p = i * 2;
		cout << *p;
		delete p; //de-allocate memory
	}
}
```

Pointers and references.

A variable declaration will associate a name with a type and a memory location.
When the variable is used somewhere in the code the compiler:
- Lookup the address that the variable name corresponds to
- Go to that location in memory and retrieve or set the value it contains
- will use the type to figure out the domain, memory size, semantics of the bits and the possible operations that can be performed

Reference operator ('&' ampersand)

&x evaluates to the address of x in memory.

```
int x = 7;
cout << x << " memory address is:" << &x;
cout << " size is:" << sizeof(x) << "\n";
```

```
int v[3] = { 1, 2, 3 };
cout << v[0] << " memory address is:" << &v[0];
cout << " size is:" << sizeof(v[0]) << "\n";
cout << v[1] << " memory address is:" << &v[1];
cout << " size is:" << sizeof(v[1]) << "\n";
cout << v[2] << " memory address is:" << &v[2];
cout << " size is:" << sizeof(v[2]) << "\n";
```

Dereference operator ('*' star)

*(&x) takes the address of x and dereferences it – it retrieves the value at that location in memory. *(&x) thus evaluates to the same thing as x

```
int x = 7;
cout << x << " memory address is:" << &x;
cout << x << " the value is:" << *(&x) << "\n";
```

Pointers

Pointers are just variables storing integers representing memory addresses
Memory addresses, or pointers, allow us to manipulate data much more flexibly; manipulating the memory addresses of data can be more efficient than manipulating the data itself

A pointer that stores the address of some variable x is said to point to x. We can access the value of x by dereferencing thepointer

Pointer declaration

`data_type *pointer_name;`

pointer_name variable can hold a memory adress.
We can access the value by dereferencing it using the * operator
Without the * operator, the identifier x refers to the pointer itself, not the value it points to.

```
int x = 7;
//declare and initialise the pointer p
//p and x refer the same memory location
int* p = &x;
//print the value stored in the memory location
cout << *p << "\n";
//print the address of the memory location
cout << p << "\n";
//modify the value through x
x = 8;
cout << *p << "\n";
//modify the value through p
*p = 10;
cout << *p << "\n";
return 0;
```

Null pointer

Any pointer set to 0 is called a **null pointer**, and since there is no memory location 0, it is an invalid pointer.
Pointers are often set to 0 to signal that they are not currently valid.
We should generally check whether a pointer is null before dereferencing it.

Dangling pointer

Some pointers do not point to valid data; dereferencing such a pointer is a runtime error.
Dereferencing pointers to data that has been erased from memory usually causes runtime errors.
dangling pointer (points to memory whose contents are undefined)

```
int* f() {
	int localVar = 7;
	cout << localVar << "\n";
	return &localVar;
}

int main() {
	int* badP = f();
	cout << *badP << "\n";
}
```

References

data_type &reference_name;

```
int y = 7;
int &x = y; //make x a reference to, or an alias of, y
```

Changing x will change y and vice versa, they are just 2 names for the same memory location
References are really similar with pointers:
References are pointers that are dereferenced automatically when you use them
Can not change the location to which a reference points.

Pointers and arrays

An array is actually a pointer to the first element in the array

- arrays are passed by reference. Passing an array is realy passing a pointer
- the array indices start at 0. The first element is 0 away from the start of the array. array[3] – the compiler computes the element that is 3 away from the starting element.

Pointer aritmethic

is a way of using subtraction and addition of pointers to move around between locations in memory.

```
int t[3] = { 10, 20, 30 };
int *p = t;
//print the first elem
cout << *p << "\n";
//move to the next memory location (next int)
p++;
//print the element (20)
cout << *p << "\n";
```

p++ the compiler will go to the next memory location, depending on the type the compiler will use the appropiate step size.

Const Pointers

1 Changeable pointer to constant data

```
int j = 100;
const int* p2 = &j;
```

The value cannot be changed through thispointer but
the pointer maybe changed to point to a different constant value

```
const int* p2 = &j;
cout << *p2 << "\n";
//change the memory address (valid)
p2 = &i;
cout << *p2 << "\n";
//change the value (compiler error)
*p2 = 7;
cout << *p2 << "\n";
```

2 Constant pointer to changeable data

```
int * const p3 = &j;
```

value can be changed through this pointer, but the pointer may not be changed to point to a different memory location

```
int * const p3 = &j;
cout << *p2 << "\n";
//change the memory address (compiler error)
p3 = &i;
cout << *p3 << "\n";
//change the value (valid)
*p3 = 7;
cout << *p3 << "\n";
```

3 Constant pointer to constant data

```
const int * const p4 = &j;
```

Dynamic allocation

Whenever we declare a new local variable (int x), memory is allocated on a region of memory called **stack**, the memory is freed up by the compiler when the variable goes out of scope.

```
int f(int a) {
   if (a>0){
       int x = 10; //memory for x is allocated on the stack
   }
   //here x is out of scope and the memory allocated for x is no longer reserved
   //the memory can be reused
   return 0;
}
```

```
int f(int a) {
   int *p;
   if (a>0){
       int x = 10;
        p = &x;
   }
   //here p will point to a memory location that is no longer reserved
   cout << *p; //undefined behavior, the program may crash
   return 0;
}
```

The compiler will take care of the allocation/dealocation.

Dynamic allocation

Another way to allocate memory, where the memory will remain allocated until you manually de-allocate it

type_name ***variable_name = **new** type_name;

new T;

- memory is allocated to store a value of type T
- it returns the address of the memory location
- the return value has type T*
- the memory is allocated on the **Heap** (memory region)
- **the memory remain allocated until you manually de-allocate it**

delete p;

- De-allocates memory that was previously allocated using new
- precondition: p is of type T* and p had been obtained through new;
- the memory space alocated to the variable p is free;
- always deallocate everything that has ever been allocated.

```
int *p1 = new int; //value from address p1 is not initialised
int *p2 = new int(3); //value from address p2 is 3
int *p3 = new int[4]; //a vector of 4 integer elements is alocated

//free the memory
delete p1;
delete p2;
delete[] p3;
```

Memory leak

Memory leak: when a program allocates memory but is unable to de-allocate it

```
void f() {
	int *p;
	for (int i = 0; i < 10; i++) {
		p = new int;//allocate memory for an int on the heap
		*p = i * 2;
		cout << *p;
	}
	delete p;//de-allocate memory (WRONG)
}
void f() {
	int *p;
	for (int i = 0; i < 10; i++) {
		p = new int; //allocate memory for an int on the heap
		*p = i * 2;
		cout << *p;
		delete p; //de-allocate memory
	}
}
```

ADT – Abstract data types

Definition: An ADT:

- exports a name (type)
- defines the domain of possible values
- establishes an interface to work with objects of this type (operations)
- restricts the access to the object components through the operations defined in its interface
- hides the implementation.

ADT implemented in C++

<adt.h> + <adt.cpp>

interface implementation

Any program entity that satisfies the requirements from the ADT definition is considered to be an implementation of the ADT.

ADT - Set imlplemented using modular programming

Object oriented programming

OOP provide flexible and powerful abstraction

- Allow programmers to think in terms of the structure of the problem rather than in terms of the structure of the computer.
- Decompose the problem into a set of objects
- Objects interact with each other to solve the problem
- create new type of objects to model elements from the problem space

Primary OOP features:

Encapsulation: grouping related data and functions together as objects and defining an interface to those objects

Inheritance: allowing code to be reused between related types

Polymorphism: allowing a value to be one of several types, and determining at runtime which functions to call on it based on its type

Classes and objects in C++

Class: A user-defined datatype which groups together related pieces of information
Class is defined in a header file. Will contain field and function declarations
Syntax:

```
/**
 * Represent rational numbers
 */
class Rational {
public:
	//methods
	/**
	 * Add an integer number to the rational number
	 */
	void add(int val);
	/**
	 * multiply with a rational number
	 * r rational number
	 */
	void mul(Rational r);
private:
	//fields (members)
	int a;
	int b;
};
```

Method definition

In a separate cpp file we define the methods declared in the class using the scope operator **::**

```
/**
 * Add an integer number to the rational number
 */
void Rational::add(int val) {
	a = a + val * b;
}
```

We can define the methods right in the class (header file). These methods will be **inline methods.**

```
class Rational {
public:
	/**
	 * Return the numerator of the number
	 */
	int getNumerator() {
		return a;
	}
	/**
	 * Get the denominator of the fraction
	 */
	int getDenominator() {
		return b;
	}
private:
	//fields (members)
	int a;
	int b;
```

Only works for simple methods (no loops).
The compiler will inline the code for every invocation.

An **instance** is an occurrence of a class.
Different instances can have their own set of values in their fields

Object declaration

<class_name> <identif>;

- Memory is allocated to store the object
- The object is initialised using the default constructor
- if the class provide constructors with arguments we may initialise the object using these

'

```
Rational r1 = Rational(1, 2);
Rational r2(1, 3);
Rational r3;
cout << r1.toFloat() << endl;
cout << r2.toFloat() << endl;
cout << r3.toFloat() << endl;
```

Access to the object fields

Inside a class method

```
int getDenominator() {
    return b;
}
```

```
int getNumerator() {
      return this->a;
}
```

Useful if the method argument has the same name as the field.
this: a pointer to the current instance. Every method has this pointer, is implicitly passed to the method to have a reference to the current instance.

In other (non member) function (if the field is visible)
Using **'.' object.field** operatior or
Using **'->' object_reference->field** is a shorthand for **(*object reference).field**

Members. Access modifiers.

Access modifiers: Define where your fields/methods can be accessed from

public: can be accessed from anywhere

private: can only be accessed within the class

Use getters to allow read-only access to private fields.

```
class Rational {
public:
	/**
	 * Return the numerator of the number
	 */
	int getNumerator() {
		return a;
	}
	/**
	 * Get the denominator of the fraction
	 */
	int getDenominator() {
		return b;
	}
private:
	//fields (members)
	int a;
	int b;

};
```

Constructors

Constructor: Method that is called when an instance is created. (declaring a local variable or creating an object using **new**)

is a special method. The name is the class name
no return type
the constructor should allocate memory and initialise the fields

```
class Rational {
public:
	Rational();
private:
	//fields (members)
	int a;
	int b;
};
```

```
Rational::Rational() {
	a = 0;
	this->b = 1;
}
```

It is called automatically whenever a class object is created – it is impossible to create an object without a constructor being called;

A class must have at least one constructor function (if you don't declare one an implicit constructor is automatically created);

May have 0 or more parameters; a constructor with no parameters is called the default constructor;

When making an array of objects, default constructor (no argument constructor) is invoked on each element

Constructor with parameters

```
Rational::Rational(int a, int b) {          Rational r2(1, 3);
       this->a = a;
       this->b = b;
}
```

We can have multiple constructors with different parameter-list

Copy constructor

Constructor invoked when a copy of the current object is needed

- assigning one class instance to another
- passing object as arguments (pass by value)
- returning a value from a function.

```
Rational::Rational(Rational &ot) {
       a = ot.a;
       b = ot.b;
}
```

There is a default copy constructor (even if you dont write one) that copy all the fields but this not always provide the expected behavior

Allocating instances using new

new can be used to allocate a class instance on the heap.

```
Rational *p1 = new Rational;
Rational *p2 = new Rational(2, 5);
cout << p1->toFloat() << endl;
cout << (*p2).toFloat() << endl;
delete p1;
delete p2;
```

Destructors

Destructor is called when the class instance gets de-allocated

- If allocated with new, when delete is called
- If stack-allocated, when it goes out of scope

<pre>DynamicArray::DynamicArray() {
 capacity = 10;
 elems = new Rational[capacity];
 size = 0;
}</pre>
	<pre>DynamicArray::~DynamicArray() {
	 delete[] elems;
	}</pre>

Operators' overloading.

Define operator semantic for user defined datatypes.

```
/**
 * Overloading the + to add 2 rational numbers
 */
Rational Rational::operator +(Rational r) {
	Rational rez = Rational(this->a, this->b);
	rez.add(r);
	return rez;
}
```

The list of overloadable operators:
+, -, *, /, +=, -=, *=, /=, %, %=, ++, –, =, ==, < >, <=, >=, !, !=, &&, ||, <<, >>, <<=, >>=, &, ^, |, &=, ^=, |=, ~, [] ,, () , ->*, →, new , new[] , delete, delete[],

Object oriented programming

OOP provide flexible and powerful abstraction

- Allow programmers to think in terms of the structure of the problem rather than in terms of the structure of the computer.

- Decompose the problem into a set of objects

- Objects interact with each other to solve the problem

- create new type of objects to model elements from the problem space

Primary OOP features:

Encapsulation: grouping related data and functions together as objects and defining an interface to those objects

Inheritance: allowing code to be reused between related types

Polymorphism: allowing a value to be one of several types, and determining at runtime which functions to call on it based on its type

Classes and objects in C++

Class: A user-defined datatype which groups together related pieces of information
Class is defined in a header file. Will contain field and function declarations
Syntax:

```
/**
 * Represent rational numbers
 */
class Rational {
public:
	//methods
	/**
	 * Add an integer number to the rational number
	 */
	void add(int val);
	/**
	 * multiply with a rational number
	 * r rational number
	 */
	void mul(Rational r);
private:
	//fields (members)
	int a;
	int b;
};
```

Method definition

In a separate cpp file we define the methods declared in the class using the scope operator **::**

```
/**
 * Add an integer number to the rational number
 */
void Rational::add(int val) {
	a = a + val * b;
}
```

We can define the methods right in the class (header file). These methods will be **inline methods.**

```
class Rational {
public:
	/**
	 * Return the numerator of the number
	 */
	int getNumerator() {
		return a;
	}
	/**
	 * Get the denominator of the fraction
	 */
	int getDenominator() {
		return b;
	}
private:
	//fields (members)
	int a;
	int b;
```

Only works for simple methods (no loops).
The compiler will inline the code for every invocation.

An **instance** is an occurrence of a class.
Different instances can have their own set of values in their fields

Object declaration

<class_name> <identif>;

- Memory is allocated to store the object
- The object is initialised using the default constructor
- if the class provide constructors with arguments we may initialise the object using these

'

```
Rational r1 = Rational(1, 2);
Rational r2(1, 3);
Rational r3;
cout << r1.toFloat() << endl;
cout << r2.toFloat() << endl;
cout << r3.toFloat() << endl;
```

Access to the object fields

Inside a class method

```
int getDenominator() {
    return b;
}
```

```
int getNumerator() {
       return this->a;
}
```

Useful if the method argument has the same name as the field.
this: a pointer to the current instance. Every method has this pointer, is implicitly passed to the method to have a reference to the current instance.

In other (non member) function (if the field is visible)
Using **'.' object.field** operatior or
Using **'->' object_reference->field** is a shorthand for **(*object reference).field**

Members. Access modifiers.

Access modifiers: Define where your fields/methods can be accessed from

public: can be accessed from anywhere

private: can only be accessed within the class

Use getters to allow read-only access to private fields.

```
class Rational {
public:
	/**
	 * Return the numerator of the number
	 */
	int getNumerator() {
		return a;
	}
	/**
	 * Get the denominator of the fraction
	 */
	int getDenominator() {
		return b;
	}
private:
	//fields (members)
	int a;
	int b;

};
```

Constructors

Constructor: Method that is called when an instance is created. (declaring a local variable or creating an object using **new**)

is a special method. The name is the class name
no return type
the constructor should allocate memory and initialise the fields

```
class Rational {
public:
	Rational();
private:
	//fields (members)
	int a;
	int b;
};
```

```
Rational::Rational() {
	a = 0;
	this->b = 1;
}
```

It is called automatically whenever a class object is created – it is impossible to create an object without a constructor being called;

A class must have at least one constructor function (if you don't declare one an implicit constructor is automatically created);

May have 0 or more parameters; a constructor with no parameters is called the default constructor;

When making an array of objects, default constructor (no argument constructor) is invoked on each element

Constructor with parameters

```
Rational::Rational(int a, int b) {        Rational r2(1, 3);
      this->a = a;
      this->b = b;
}
```

We can have multiple constructors with different parameter-list

Copy constructor

Constructor invoked when a copy of the current object is needed

- assigning one class instance to another
- passing object as arguments (pass by value)
- returning a value from a function.

```
Rational::Rational(Rational &ot) {
      a = ot.a;
      b = ot.b;
}
```

There is a default copy constructor (even if you dont write one) that copy all the fields
but this not always provide the expected behavior

Allocating instances using new

new can be used to allocate a class instance on the heap.

```
	Rational *p1 = new Rational;
	Rational *p2 = new Rational(2, 5);
	cout << p1->toFloat() << endl;
	cout << (*p2).toFloat() << endl;
	delete p1;
	delete p2;
```

Destructors

Destructor is called when the class instance gets de-allocated

- If allocated with new, when delete is called
- If stack-allocated, when it goes out of scope

DynamicArray::DynamicArray() { capacity = 10; elems = new Rational[capacity]; size = 0; }	DynamicArray::~DynamicArray() { delete[] elems; }

```
DynamicArray::DynamicArray() {
	capacity = 10;
	elems = new Rational[capacity];
	size = 0;
}
```

```
DynamicArray::~DynamicArray() {
	delete[] elems;
}
```

Operators' overloading.

Define operator semantic for user defined datatypes.

```
/**
 * Overloading the + to add 2 rational numbers
 */
Rational Rational::operator +(Rational r) {
	Rational rez = Rational(this->a, this->b);
	rez.add(r);
	return rez;
}
```

The list of overloadable operators:
+, -, *, /, +=, -=, *=, /=, %, %=, ++, –, =, ==, < >, <=, >=, !, !=, &&, ||, <<, >>, <<=, >>=, &, ^,
|, &=, ^=, |=, ~, [] ,, () , ->*, →, new , new[] , delete, delete[],

Memory leaks dangling pointers

Destructor is invoked:

- if the instance is allocated with new and we invoke delete
- if is allocated on the stack, when it goes out of scope
- implement destructor and deallocate any resource !!!!

Constructor is invoked:

- when a new stack-allocated variable is declared
- if we allocate instance using new (on the heap)
- when a copy of the instance is required
    - assigmnet
    - argument passing by value
    - return an object from a method (by value)
- write your own copy constructor !!!

Dynamic array sample

Pointer void *

Syntactically, void is a basic type
void is used for functions that doesn't return any value

```
void f() {

}
```

- a variable cannot have the type void;
- we may define pointers to void

```
void* p;
int *i;
*i = 1;
p = i; //correct
//i = p; // incorect
int j = 1;
p = &j;
```

Can use *void to create data structures that manage any element type
Problems:

- test for equality
- copy 2 elements

Pointer to function

```
void (*funcPtr)();  // a pointer to a function
void *funcPtr();    // a function that returns a pointer
```

```
void func() {
      cout << "func() called..." << endl;
}
int main() {
      void (*fp)(); // Define a function pointer
      fp = func; // Initialise it
      (*fp)(); // Dereferencing calls the function
      void (*fp2)() = func; // Define and initialize
      (*fp2)(); // call
}
```

Solutions for void* containers: functions parameters

```
typedef elem (*copyPtr)(elem&, elem);

typedef int (*equalsPtr)(elem, elem);
```

Templates

- allow to work with generic types
- rather than repeating function code for each new type we wish to accommodate, we can create functions that are capable of using the same code for different types
- provides a way to reuse source code

Function template:

```
template <class identifier> function_declaration;
```

or

```
template <typename identifier> function_declaration;
```

```
int sum(int a, int b) {
	return a + b;
}

double sum(double a, double b) {
	return a + b;
}
```

```
template<typename T> T sum(T a, T b) {
	return a + b;
}

int sum = sumTemp<int>(1, 2);
cout << sum;
double sum2 = sumTemp<double>(1.2, 2.2);
cout << sum2;
```

- T is the template parameter, a type argument for the template
- the process of generating an actual function from a template function is called instantiation: `int sum = sumTemp<int>(1, 2);`

Class template:

A template can be seen as a skeleton or macro or framework
when specific types are added to this skeleton (like double) then the result is an actual C++ class

```
template<typename Element>
class DynamicArray {
public:
	/**
	 * Add an element to the dynamic array to the end of the array
	 * e - is a generic element
	 */
	void addE(Element r);
	/**
	 * Delete the element from the given position
	 * poz - the position of the elem to be deleted, poz>=0;poz<size
	 * returns the deleted element
	 */
	Element deleteElem(int poz);

	/**
	 * Access the element from a given position
	 * poz - the position (poz>=0;poz<size)
	 */
	Element get(int poz);
	/**
	 * Give the size of the array
	 * return the number of elements in the array
	 */
	int getSize();
	/**
	 * Clear the array
	 * Post: the array will contain 0 elements
	 */
	void clear();
private:
	Element *elems;
	int capacity;
	int size;
};
```

Using OOP to implement Abstract Data Types

ADT

- separate interface from implementation
- provide abstract specification
- hide implementation details (data protection)

Class

- header file: will contain the class / method declarations
- each method will be specified
- using private access modifier, the representation (fields) can not be accessed/modified from outside of the class

Sample data structure: Generic Dynamic array (for any type of elements)

- typedef Telem = <type name>
    - can not have different list (integers, Rational) in the same programm

- implemented using void*
    - can not add literal values (only references to some variables)
    - have to use casts (code not so easy to read)

- implemented using templates

Dynamic Array

variable-size array data structure that allows elements to be added or removed
elements are accessed by indexes
sequencial representation (elements are stored in succesive memory location)
the underlying array will be resized automatically if needed

Static elements (members and methods).

the reference to the variable is performed using the scope resolution operator

Static members belongs to the class not to the instances
the variables declared as static are characteristic to the CLASS, they do NOT represent object state

“global variables, but defined in classes” – they are global for all objects of the class
Keyword : static is used

```
/**                                                     Rational::nrInstances
 * New data type to store rational
numbers
 * we hide the data representation
 */
class Rational {
public:
	/**
	 * Get the nominator
	 */
	int getUp();
	/**
	 * get the denominator
	 */
	int getDown();
private:
	int a;
	int b;
	static int nrInstances = 0;
};
```

Friend elements.

- want to allow a function that is not a member of a given class to access the private fields/methods of that class
- placing the signature of the function inside the class, preceded by the word **friend**;
-
- A friend function is a function that is NOT a member of a class, but has ACCESS to the private members of that class.
-
- A class B is called friend class of class A if class B has ACCESS to the private members of A.

```
template<typename E>
class Set {
	friend class Set_iterator<E> ;
….
```

Iterators

Provide a generic (abstract) way to access the elements of a container.
An iterator will contain:
  a reference to the current element
  a reference to the container
Allow access to the elements of a container without exposing the internal representation (implementation hiding)

UML diagrams for classes (members, accessibility).

- UML (Unified Modeling Language)
- UML is the industry-standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems
- UML is the standard notation for software architecture
- Language Independent

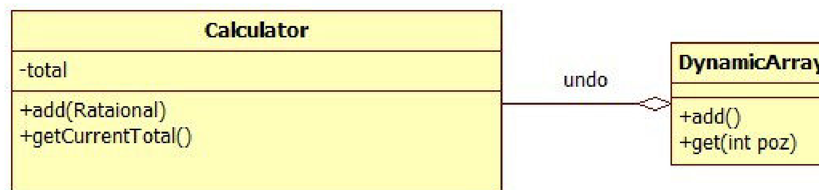
UML Class diagram

specifies the entities in a program and the relations between them

contains and specifies:

- class name
- variables - (name + type)
- methods - (name + parameter types + return type)

private members are denoted by “-”,
public members are denoted by “+”



Simple Feture Driven Development process

- Identity and outline requirements
- Build a feature list from the problem statement
- Plan iterations, an iteration including a single feature
- For each iteration
    - Design feature
    - Implement/test feature

Assert

```
#include <assert.h>
void assert (int expression);
```

expression - Expression to be evaluated. If this expression evaluates to 0 (false), this causes an assertion failure that terminates the program.

The specifics of the message shown depend on the specific implementation in the compiler

it shall include:
the expression whose assertion failed,
the name of the source file,
and the line number where it happened.

We will use assert to build simple automated tests.

Primary OOP features:

Encapsulation: grouping related data and functions together as objects and defining an interface to those objects

Inheritance: allowing code to be reused between related types

Polymorphism: allowing a value to be one of several types, and determining at runtime which functions to call on it based on its type

Inheritance

Inheritance is the property that allows us to define a new class (subclass) using the definition of another class (superclass) – or classes.

If A and B are two classes, we say that B inherits from A (B is derived from A or B is a specialization of A) if:

- class B has all variables and methods of class A
- class B may redefine methods of class A
- class B may add new members besides the ones inherited from A.

```class Person {```	```class Student: public Person {```

<pre>class Person {
public:
 Person(string cnp, string name);
 const string& getName() const {
  return name;
 }
 	
 const string& getCNP() const {
  return cnp;
 }
 string toString();
protected:
 string name;
 string cnp;
 	
};</pre>
	<pre>class Student: public Person {
	public:
	 Student(string cnp, string name,
	string faculty);
	 const string& getFaculty() const {
	  return faculty;
	 }
	 string toString();
	private:
	 string faculty;
	};</pre>

Simple inheritance. Derived classes.

If class B inherits from class A then:

- an object of class B includes all member variables of class A
- the member functions of class A can be applied to objects of class B (unless they are hidden)
- class B can add as many extra member variables or functions as it requires

```
class A:public B{
….
}
```

class A = superclass, base class, parent class

class B = subclass, derived class, descendent class

inherited member (function, variable) = a member defined in A, and used unchanged in B

redefined member (overridden) = defined in A and B

added member (new) = defined only in B

Access control

`public` means that inherited public members from class B are also public members of class A;

```
class A:public B{
….
}
```

`private` means that inherited public members of class B are private members of class A;

```
class A:private B{
…
}
```

`protected` means that any public or protected members inherited from class B will be protected members of class A.

```
class A:protected B{
…
}
```

Members access modifiers.

Access modifiers: Define where your fields/methods can be accessed from

`public`: can be accessed from anywhere

`private`**:** can only be accessed within the class

`protected`**:** members that are accessible only by the defining class and classes derived from it. `protected` acts just like `private`, with the exception that an inheriting class has access to protected members, but not private members.

Destructors and Constructors for derived classes

- Constructors and destructors are not automatically inherited
- Constructors and destructors in the derived class need to invoke a constructor/destorctor from the base class.

```
Student::Student(string cnp, string name, string faculty) :
            Person(cnp, name) {
      this->faculty = faculty;
}
```

- If the constructor is not explicitly invoked, the default constructor from the base class is invoked automatically
- if there are no default constructor defined we get a compiller error

```
Student::Student(string cnp, string name, string faculty) {
      this->faculty = faculty;
}
```

The destructor of the base class (Person) is also invoked

```
Student::~Student() {
      cout << "destroy student\n";
}
```

Member initializer syntax.

When defining a constructor, you sometimes want to initialize certain members,particularly const members,even before the constructor body.
put a colon before the function body, followed by a comma-separated list of items of the form dataMember(initialValue).

```
Person::Person(string c, string n) :
            cnp(c), name(n) {
}
```

Object creation and destruction in derived classes

Creation

- allocate memory for member variables from base class
- allocate memory for member variables from derived class
- a constructor is selected and called to initialize the variables from the base class
- a constructor is selected and called to initialize the variables from the derived class

Destruction

- destructor call for derived class
- destructor call for base class

Substitution principle.

An object of the derived class can be used in any context expecting an object of the base class. (upcast is implicit!)

```
Person p = Person("1", "Ion");
cout << p.toString() << "\n";

Student s("2", "Ion2", "Info");
cout << s.toString() << "\n";

Teacher t("3", "Ion3", "Assist");
cout << t.getName() << " " << t.getPosition() << "\n";

p = s;
cout << p.getName() << "\n";

p = t;
cout << p.getName() << "\n";

s = p;//not valid, compiler error
```

Pointers

```
Person *p1 = new Person("1", "Ion");
cout << p1->getName() << "\n";

Person *p2 = new Student("2", "Ion2", "Mat");
cout << p2->getName() << "\n";

Teacher *t1 = new Teacher("3", "Ion3", "Lect");
cout << t1->getName() << "\n";

p1 = t1;
cout << p1->getName() << "\n";

t1 = p1;//not valid, compiler error
```

Is a vs Has a

- a Sale has 1 or more SaleItem
- SaleItem has a Product

UML Associations: Describe a relationship of structural dependency between classes

An association may have:

- a name
- a multiplicity
- a role name
- navigability (uni/bi-directional)

Association types

- Association
- Agregation (composition) (whole-part relation)
- Dependency
- Inheritance

There are basically two ways to describe that some class A is depending on some other class B
has a:

- Every object A has an object B.
- Every SaleItem has a Product. Every Person has a name (string)
- implemented by declaring data members

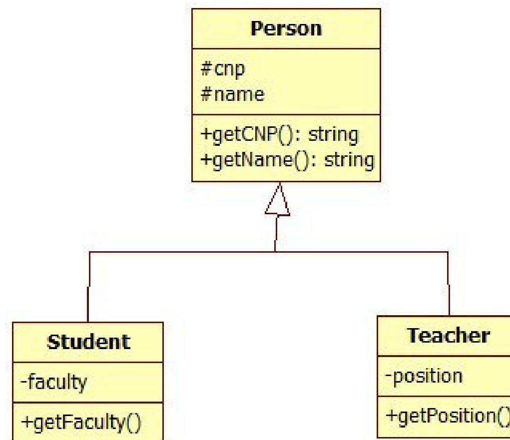
is a (is like a):

- Every instance of A is instance of B.
- Every Student is a person
- implemented using inheritance

Only use inheritance to implement **is a** relationships

Specialization/generalization relation - UML representation.

Inheritance allows us to define hierarchies of related classes
UML description:



Student is a Person with some aditional attributes/methods
Student inherit (fields and methods) from Person
Student is derived from Person. Person is the base class Student is the derived class

Person is generalization of Student
Student is a specialization of Person

Same for Teacher

Method overriding.

A base class may redefine (override) some methods of the base class

`string Person::toString() {`
`    return "Person:" + cnp + " " + name;`
`}`
	`string Student::toString() {`
	`  return "Student:" + cnp + " " + name + " " + faculty;`
	`}`

`Person p = Person("1", "Ion");` `cout << p.toString() << "\n";`	`Student s("2", "Ion2", "Info");` `cout << s.toString() << "\n";`

In defining derived classes, we only need to specify what's different about them from their base classes. (programming by difference)

Inheritance allows only overriding methods and adding new members and methods. We cannot remove functionality that was present in the base class.

Overriding ≠ Overloading

```
string Person::toString() {
       return "Person:" + cnp + " " + name;
}

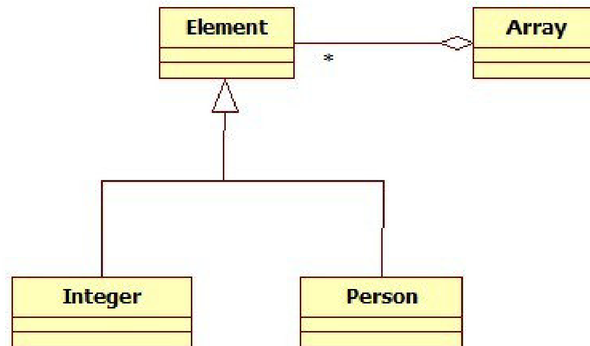
string Person::toString(string prefix) {
       return prefix + cnp + " " + name;
}
```

toString method is overloaded (`toString()`,`toString(string prefix)`)

Person List sample

- sample LIST ADT
- a list that can hold any Person
- we can add to the list: Person, Student, Teacher
- provide iterator to access the elements

More generic:



Polymorphism

Definitions:

Polymorphism is the property of an entity to react differently depending on its state.

Polymorphism is the property that allows different entities to behave in different ways to the same action.

Polymorphism allows different objects to respond in different ways to the same message.

```
string Person::toString() {
      return "Person:" + cnp + " " + name;
}
string Student::toString() {
      return "Student:" + cnp + " " + name + " " + faculty;
}
string Teacher::toString() {
      string rez = Person::toString();
      return "Teacher " + rez;
}
Student s("2", "Ion2", "Info");
Person* aux = &s;
cout << aux->toString() << "\n";
Person p = Person("1", "Ion");
aux = &p;
cout << aux->toString() << "\n";
```

- Person,Student,Teacher has a toString , so there'll be different versions of the method toString in each subclass.
- The system must dynamically determine the code to be executed when the method toString is called;
- The decision is based on the actual type of the object.
- This important feature (present in almost any OOL) is called dynamic binding (late binding, runtime binding).
- Dynamic binding means that the decision of which function body to execute is made at runtime, unlike static binding which means that the decision is made at compile time (before running).
- Dynamic binding only works with non-value types: references and pointers

Dynamic binding.

The binding between the name of a function and its code is done:
at the compilation time => static binding
at the execution time   => dynamic binding

Dynamic Binding (Late Binding)
When you send a message to an object, the code being called isn't determined until runtime.
(only for references and pointers)
Virtual methods allow dynamic binding

Virtual methods.

Dynamic binding in c++: Using virtual functions
If a function is declared virtual in a base class:
`virtual` <function-signature>

- and then overridden in a derived class, dynamic binding is enabled
- now, the actual function called depends on the type of the object.

- Constructors can NEVER be virtual!
- Destructor may be declared virtual!

```
class Person {
protected:
	string name;
	string cnp;

public:
	Person(string cnp, string name);
	virtual ~Person();

	const string& getName() const {
		return name;
	}

	const string& getCNP() const {
		return cnp;
	}
	virtual string toString();
	string toString(string prefix);
};
```

C++ Mechanism

the object itself must store some information about its actual type

Based on this type the call is dispatched to the appropriate implementation of the method. In fact, the object may store a pointer to a dispatch table through which virtual methods are called.

Every class that has at least one virtual method (a polymorphic class) has a hidden member that is a pointer to a virtual table (VTABLE).

The virtual table contains the addresses of the virtual member functions for that particular class.

When a call is made through a pointer or reference to such object, the compiler generates code that dereferences the pointer to the object's virtual table and makes an indirect call using the address of a member function stored in the table.

Virtual destructors

- the destructor responsibility is to dealocate resources (memory)
- if you have a class hierarchy then most probably you want a polymorphic behavior for the destructor (invoke the destructor based on the actual type of the object, not the type of the pointer holding the reference to the object)

Multiple inheritance.

Unlike many object-oriented languages, C++ allows a class to have multiple base classes

```
class Car : public Vehicle , public InsuredItem {

};
```

The class will inherit all the members from all the base classes.
Multiple inheritance can be dangeuros

- you can inherit the same field/method from different classes
- some of the base classes may have a common base class

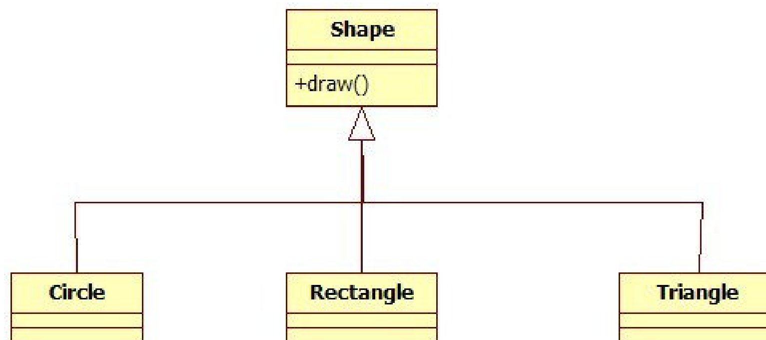
In general avoid multiple inheritance

Pure virtual functions

We can omit the definition of a method from a class by making the function pure virtual.
Using pure virtual funtions, we make sure all the derived (concrete) classes will define the method

```
class Shape {
public:
	Shape();
	virtual ~Shape();
	virtual void draw() = 0;//pure virtual
};
```

The =0 indicates that no definition will be given. This implies that one can no longer create an instance of Shape.
Shape is an abstract class - defines only an interface, but doesn't actually implement it, and therefore cannot be instantiated.



Abstract classes

An abstract class serves as a base class for a collection of related derived classes;
It provides:

- a common public interface (or pure virtual member functions)
- any shared representation
- any shared member functions

an abstract class does NOT have instances
pure virtual function (at least one!):
virtual <return-type> <name> (<parameters>) = 0;

you tell the compiler to reserve a slot for a function in the VTABLE, but not to put an address in that particular slot. Even if only one function in a class is declared as pure virtual, the VTABLE is incomplete .

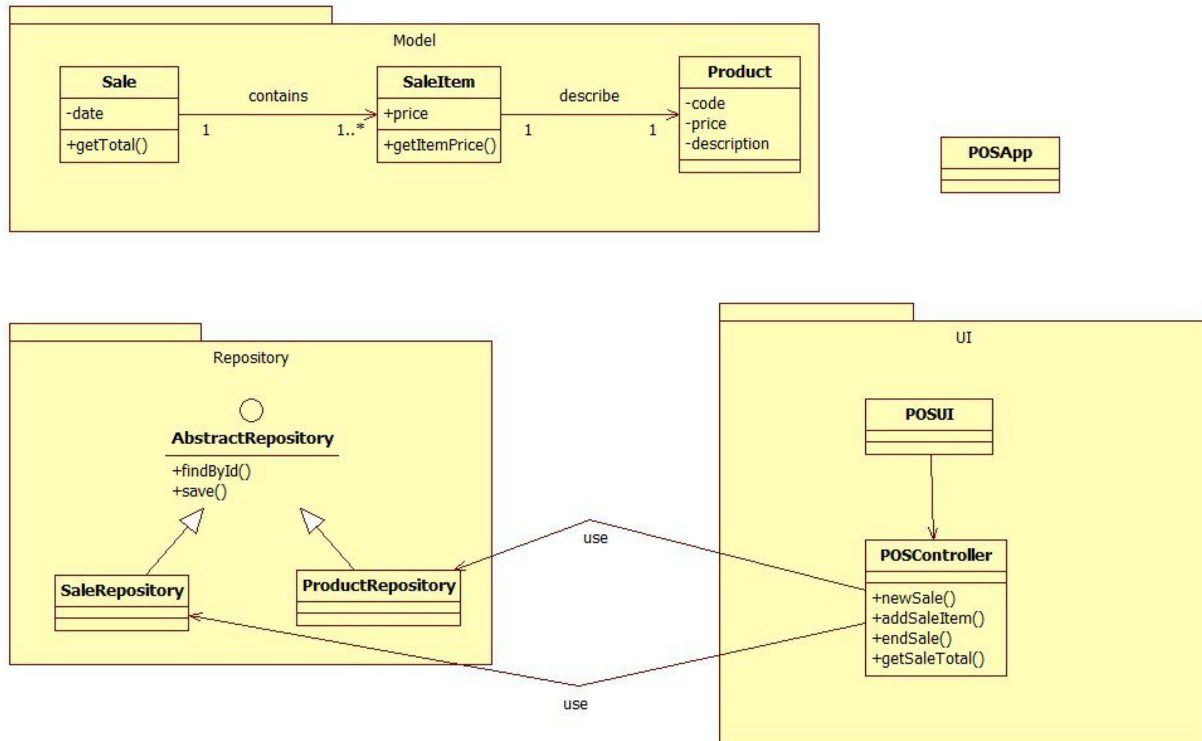
pure abstract class = class has nothing but pure virtual functions
pure abstract class = interface

UML representation for abstract entities: italic fonts

Concrete classes that extends an abstract class

- A concrete class is derived from an abstract class, inheriting its public interface
- a concrete class is expected to have instances
- override inherited abstract functions to provide concrete implementation specific to its representation (otherwise it will also be abstract classes)

POS – Point of sale



Input/output operation

Input/Output in C

<stdio.h> -> **getchar**(), **scanf**(), **printf**(), **getc**(), **putc**(), **open**(), **close**(), **fgetc**(), etc.

- C io functions are not extensible
- are designed to handle the four basic data types in C (**char**, **int**, **float**, **double** and their variations).
- Not part of the standard library => Implementation may differ (ANSI standard)
- every time you add a new class, you could add an overloaded **printf**() and **scanf**() function (and their variants for files and strings)
- Overloaded functions must have different types in their argument lists and the printf() family hides its type information in the format string and in the variable argument list.

For C++ a new IO library -> goal is to be able to easily add new data types

I/O streams. I/O Hierarchies of classes.

The **iostream** library is an object-oriented library that provides input and output functionality using streams.

A family of class templates and supporting functions in the C++ Standard Library that implement stream-based input/output capabilities

Elements of the standard iostream library:

Basic class templates

The base of the iostream library is the hierarchy of class templates. The class templates provide most of the functionality of the library in a way that they can be used to operate with any type of elements.

Class template instantiations

The library incorporates two standard sets of instantiations of the iostream class template hierarchy: one, narrow-oriented, to manipulate **char** elements (commonly used) and another one, wide-oriented, to manipulate **wchar_** elements.

Standard objects

Within the iostream library, declared in the**<iostream>** header file there are certain objects that are used to perform input and output operations on the standard input and output devices.

Types

contains types used in the io library like: streampos, streamoff and streamsize (used to represent positions, offsets and sizes, respectivelly)

Manipulators

Manipulators are global functions that modify properties and formatting information of streams

Stream

A stream is an abstraction that represents a device on which input and ouput operations are performed.

A stream is a flow of data from the set of sources (input – keyboard, file or memory zone) to a set of destinations (output – display, file or memory zone).

Streams are generally associated to a physical source or destination of characters, like a disk file, the keyboard, or the console, so the characters gotten or written to/from our abstraction called stream are physically input/output to the physical device.

For example, file streams are C++ objects to manipulate and interact with files; Once a file stream is used to open a file, any input or output operation performed on that stream is physically reflected in the file.

The essential characteristic of stream processing is that data elements must be sent to or received from a stream one at a time or in a serial fashion.

Buffer

buffer is a memory block that acts as an intermediary between the stream and the physical source or destination

Each time the member function **put** (writes a single character) is called, the character is not written directly to the physical destination (ex. file) with which the stream is associated. Instead of that, the character is inserted in that stream's intermediate buffer.

When the buffer is flushed, all the data contained in it is written to the physical medium (if it is an output stream) or simply freed (if it is an input stream). This process is called synchronization and takes place under any of the following circumstances:

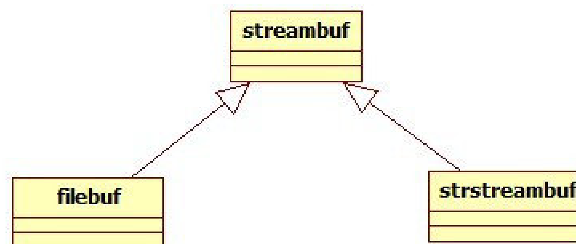
- When the file is closed: before closing a file all buffers that have not yet been flushed are synchronized and all pending data is written or read to the physical medium.

- When the buffer is full: Buffers have a certain size. When the buffer is full it is automatically synchronized.

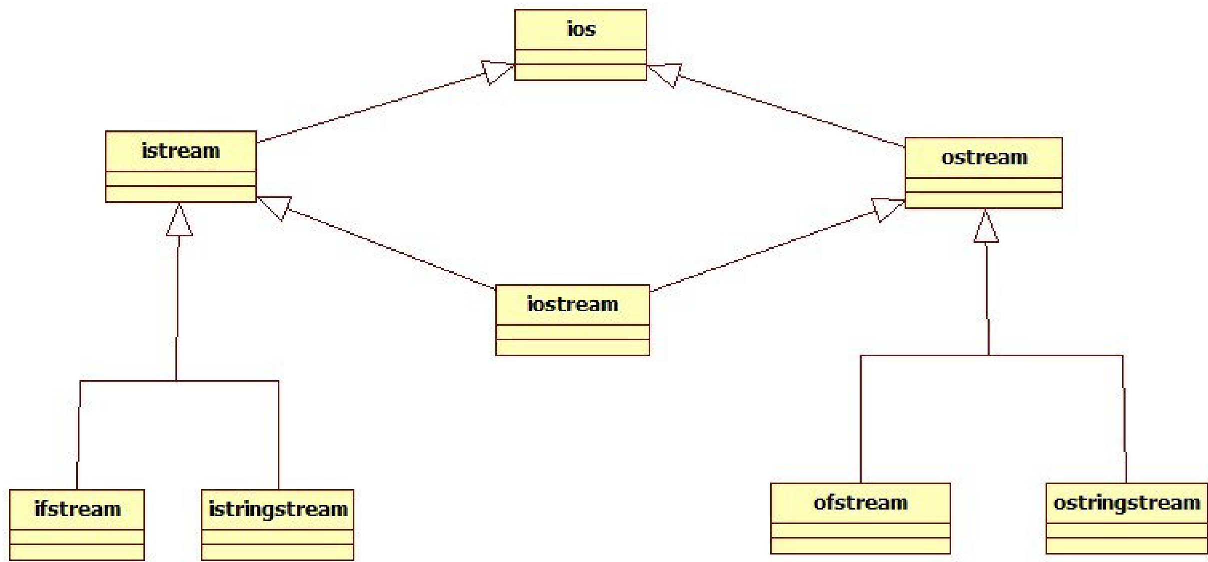
- Explicitly, with manipulators: flush and endl.

- Explicitly, with member function sync()

- Each **iostream** object contains a pointer to some kind of **streambuf**



IO Class hierarchy



IOStream header files

The classes of the input/output library reside in several headers.

- <ios> that manage formatting information and the associated streambuffer.
- <istream> implements formatted input
- <ostream> implements formatted output
- <**iostream**> implements formatted input and output
- <fstream> implement formatted input, output and input/output on file streams.
- <sstream> implement formatted input, output and input/output on string-based streams.
- <iomanip> contains formatting manipulators.
- <iosfwd> contains forward declarations of all classes in the input/output library.

Standard streams defined in <iostream>

cin - corespond to the standard file stdin, type **istream**

cout - corespond to the standard file stdout , type **ostream**

cerr - - corespond to the standard file stderr, type **ostream**

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

void testStandardIOStreams() {
	cout << "!!!Hello World!!!" << endl; // prints !!!Hello World!!! to the console
	int i = 0;
	cin >> i; //read an int from the console
	cout << "i=" << i << endl; // prints !!!Hello World!!! to the console

	cerr << "Error message";//write a message to the standard error stream
}
```

Output (Insertion operator)

- writing operations on a stream (the standard output, in a file or in a memory zone) are performed through the operator “**<<**“, called **insertion operator**

- the operand from the left hand side of the operator << must be an object of class ostream (or of an derived class). For standard output the object cout is used

- the operand from the right hand side can be an expression.

- The << operator is overloaded for standard types, and for abstract types the programmer must overload it.

```
void testWriteToStandardStream() {
	cout << 1 << endl;
	cout << 1.4 << endl << 12 << endl;
	cout << "asdasd" << endl;
	string a("aaaaaaaa");
	cout << a << endl;

	int ints[10] = { 0 };
	cout << ints << endl; //print the memory address
}
```

- since the << operator returns a reference to the current class, it may be chained and the evaluation of the expressions is performed in reversed order to the writing

Input (Extraction operator)

- reading operations are performed through the operator “>>“, called **extraction operator**
- the operand from the left hand side of the operator >> must be an object of class istream (or of an derived class). For standard output the object cin is used
- the operand from the right hand side can be an expression. The << operator is overloaded for standard types.
- for abstract types the programmer must overload it.

```
void testStandardInput() {
	int i = 0;
	cout << "Enter int:";
	cin >> i;
	cout << i << endl;
	double d = 0;
	cout << "Enter double:";
	cin >> d;
	cout << d << endl;
	string s;
	cin >> s;
	cout << s << endl;
}
```

Overload <<, >> for user defined types

- Is similar with any operator overloading
- <<. >> are a binary operators
- on the left hand we have a stream (like cin, cout) ont the right we have the user defined object.

```
class Product {
public:
	Product(int code, string desc, double
price)
	~Product();
	double getCode() const {
		return code;
	}
	double getPrice() const {
		return price;
	}
	const string& getDescription() const {
		return description;
	}

	friend ostream& operator<< (ostream&
stream, const Product& prod);

private:
	int code;
	string description;
	double price;
};
```

```
ostream& operator<<(ostream& stream, const
Product& prod) {
  stream << prod.code << " ";
  stream  << prod.description << " ";
  stream  << prod.price;
  return stream;
}
```

```
void testStandardOutputUserType() {
	Product p = Product(1, "prod", 21.1);
	cout << p << "\n";
	Product p2 = Product(2, "prod2", 2.4);
	cout << p2 << "\n";
}
```

Output formatting

width(int x)	minimum number of characters for next output
fill(char x)	character used to fill with in the case that the width needs to be elongated to fill the minimum.
precision(int x)	sets the number of significant digits for floating-point numbers

```
void testFormatOutput() {
	cout.width(5);
	cout << "a";
	cout.width(5);
	cout << "bb" << endl;
	const double PI = 3.1415926535897;
	cout.precision(3);
	cout << PI << endl;
	cout.precision(8);
	cout << PI << endl;
}
```

Manipulators.

- Manipulators are functions specifically designed to be used in conjunction with the insertion (<<) and extraction (>>) operators on stream objects

- They are still regular functions and can also be called as any other function using a stream object as argument

- Manipulators are used to change formatting parameters on streams and to insert or extract certain special characters.

- the member data x_flags declared in ios refers to the format of I/O operations, the bits of x_flags corresponding to conversion can be set using manipulators

- manipulators are defined in file **iostream.h** (endl, dec, hex, oct, etc) and **iomanip.h** (setbase(int b),setw(int w),setprecision(int p))

```
void testManipulators() {
	cout << oct << 9 << endl << dec << 9 << endl;
	oct(cout);
	cout << 9;
	dec(cout);
}
```

Flags

Flags to indicate the internal state of the stream:

Flag	Description	Verify flag
fail	Invalid data	fail()
badbit	Phyisical error	bad()
goodbit	OK	good()
eofbit	End of file detected	eof()

```
void testFlags(){
	cin.setstate(ios::badbit);
	if (cin.bad()){
		//something wrong
	}
}
```

Control flags:

```
cin.setf(ios::skipws);  //Skip white space. (For input; this is the default.)
cin.unsetf(ios::skipws);
```

Files.

Files are data structures that are stored on a disk device.

To work with files you must connect a stream to the file on disk

fstream provides an interface to read and write data from files as input/output streams.

<fstream.h>

- ifstream (input file stream)
- ofstream (output file stream)

The file to be associated with the stream can be specified either as a parameter in the constructor or by calling member **open**.

After all necessary operations on a file have been performed, it can be closed (or disassociated) by calling member **close**. Once closed, the same file stream object may be used to open another file.

The member function **is_open** can be used to determine whether the stream object is currently associated with a file.

Output File Stream

```
#include <fstream>

void testOutputToFile() {
  ofstream out("test.out");
  out << "asdasdasd" << endl;
  out << "kkkkkkk" << endl;
  out << 7 << endl;
  out.close();
}
```

- If a file named “test.out” exists on disk, it is opened for output and connected to the output stream out_file. If any data are in file when it is opened, the data are erased from file.
- Otherwise, a file named “test.out” is created, opened for output, and connected to the output stream out_file.

Input File Stream

<table>
<tr>
<td>

```
void testInputFromFile() {
	ifstream in("test.out");
	//verify if the stream is opened
	if (in.fail()) {
		return;
	}
	while (!in.eof()) {
		string s;
		in >> s;
		cout << s << endl;
	}
	in.close();
}
```

</td>
<td>

```
void testInputFromFileByLine() {
	ifstream in;
	in.open("test.out");
	//verify if the stream is opened
	if (!in.is_open()) {
		return;
	}
	while (in.good()) {
		string s;
		getline(in, s);
		cout << s << endl;
	}
	in.close();
}
```

</td>
</tr>
</table>

- If a file named "test.out" exists on disk, it is opened for input and connected to the input stream in_file.
- Otherwise, a file named "test.out" does not exist on disk. This may lead to an error
- for some C++ implementation a new file is created, opened for input, and connected to the input stream in_file.

Open mode

In order to open a file with a stream object we use its member function open():

open (filename, mode);

filename is a null-terminated character sequence of type **const char *** (the same type that string literals have) representing the name of the file to be opened
mode is an optional parameter with a combination of the following flags:

ios::in	Open for input operations.
ios::out	Open for output operations.
ios::binary	Open in binary mode.
ios::ate	Set the initial position at the end of the file. If this flag is not set to any value, the initial position is the beginning of the file.
ios::app	All output operations are performed at the end of the file, appending the content to the current content of the file. This flag can only be used in streams open for output-only operations.
ios::trunc	If the file opened for output operations already existed before, its previous content is deleted and replaced by the new one.

flags can be combined using the bitwise operator OR (|).

Read/write user defined objects

```
void testWriteReadUserObjFile() {

	ofstream out;
	out.open("test2.out", ios::out | ios::trunc);
	if (!out.is_open()) {
		return;
	}
	Product p1(1, "p1", 1.0);
	out << p1 << endl;
	Product p2(2, "p2", 2.0);
	out << p2;
	out.close();

	//read
	ifstream in("test2.out");
	if (!in.is_open()) {
		cout << "Unable to open";
		return;
	}
	Product p(0, "", 0);
	while (!in.eof()) {
		in >> p;
		cout << p << endl;
	}
	in.close();
}
```

Exception handling in c++

The exception mechanism allows programs to signal unexpected problems.

- The **try block** marks the code you believe might run into difficulty.
- The **catch block** immediately follows the try block and holds the code that deals with the problem.
- The **throw** statement is how the code with a problem notifies the calling code.

```
void testTryCatch() {
	// some code
	try {
		//code that may throw an exception
		throw 12;
		//code
	} catch (int error) {
		//error handling code
		cout << "Error ocurred." << error;
	}
}
```

Exceptions - execution flow

- Place the source code that you want guarded against errors inside a try block.
- Then construct a catch program block that acts as the error handler.
- If the code in the try block (or any code called from the try block) throws an exception, the try block immediately transfer the control to the exception handler (catch block).

```
void testTryCatchFlow(bool throwEx) {
	// some code
	try {
		cout << "code before the exception" << endl;
		if (throwEx) {
			cout << "throw (raise) exception" << endl;
			throw 12;
		}
		cout << "code after the exception" << endl;
	} catch (int error) {
		cout << "Error handling code " << endl;
	}
}
```

```
testTryCatchFlow(0);
testTryCatchFlow(1);
```

- The catch program block doesn't have to be in the same function as the one in which the exception is thrown.
- When an exception is thrown, the system starts "unwinding the stack," looking for the nearest catch block.
- If the catch block is not found in the function that threw the exception, the system looks in the function that called the throwing function.
- This search continues up the function-call stack. If the exception is never caught, the program halts.

Exception objects

- The exception object thrown can be just about any kind of data structure you like.

- Is a good practice to create an exception class for certain kinds of exceptions that occur in your programs

```
class POSError {
public:
	POSError(string message) :
			message(message) {
	}
	const string& getMessage() const {
		return message;
	}
private:
	string message;
};
```

```
class ValidationError: public POSError {
public:
	ValidationError(string message) :
			POSError(message) {
	}
};
```

```
void Sale::addSaleItem(double quantity, Product* product) {
	if (quantity < 0) {
		throw ValidationError("Quantity must be positive");
	}
	saleItems.push_back(new SaleItem(quantity, product));
}
```

```
try {
	pos->enterSaleItem(quantity, product);
	cout << "Sale total: " << pos->getSaleTotal() << endl;
} catch (ValidationError& err) {
	cout << err.getMessage() << endl;
}
```

Exception specification

Exception specifications are used to document what exceptions are thrown from a function

If a function does not have an exception specification, that function is allowed to throw any type of exception

```
//can throw any type of exceptions
void f1() {

}
```

```
//can not throw exceptions
void f3() throw (){

}
```

```
/**
 * create ,validate and store a product
 * code - product code
 * desc - product description
 * price - product price
 * throw ValidatorException if the product is invalid
 * throw Repository exception if the code is already used by a product
 */
Product Warehouse::addProduct(int code, string desc, double price)
             throw (ValidatorException, RepositoryException) {
      //create product
      Product p(code, desc, price);
      //validate product
      validator->validate(p);
      //store product
      repo->store(p);
      return p;
}
```

Advantages of exceptions over adhoc error handling (control flags, error codes,etc)

- using exception handling it is possible to separate the error handling code from the normal flow of control
- different types of errors can be handled in one place (handler for an exception base class, handle any kind of exception(...)
- using exceptions the programm can not ignore the error. The program will terminate unless there is a handler (a catch statement) that can handle the exception.
- Functions will need fewer arguments and return values, this makes them easier to use and understand
- any amount of information can be passed with the exception
- if you do not have any way of recovering from an error reported as an exception you can ignore it and it will be propagated up allong the call chain

When to throw exceptions

- Only throw exceptions when a function fails to perform what is expected to do
- use exceptions to signal unexpected errors
- too frequent use of exceptions will make the control flow difficult to follow
- we may throw an exception to signal precondition violation.
    - Normally the user of the function is responsable to satisfy the preconditions iposed by the function.
- Throwing an exception for the sole purpose of changing the control flow is therefore not recommended.
- Constructors/ destructors should not throw exceptions

Handling multiple types of exceptions

```
try {
	Product p = wh->addProduct(code, desc, price);
	cout << "product " << p.getDescription() << " added." << endl;
} catch (RepositoryException& ex) {
	cout << "Error on store:" << ex.getMsg() << endl;
} catch (ValidatorException& ex) {
	cout << "Error on validate:" << ex.getMsg() << endl;
} catch (...) {
	cout << "unknwn exception";
}
```

- If the code in the **try** block throw an exception the appropiate handler code will be executed
- the execution flow will jump to the first catch clause that match the type of the exception (is the same type or is a derived type)
- **(...)** match any exception type

Exception class hierarchy

- Exception handling makes it possible to localize error handling to fewer places in the code.
- The number of try blocks should not have to grow exponentially with the size of the program.
- Exception classes should be organized in hierarchies to minimize the number of exception handlers.
- Group related exception types by using inheritance.
- Exception hierarchies allow for object-oriented error handling (dynamic binding, the same handler used for diffrent type of exceptions)

```
try {
	Product p = wh->addProduct(code, desc, price);
	cout << "product " << p.getDescription() << " added." << endl;
} catch (WarehouseException& ex) {
	cout << "Error on store:" << ex.getMsg() << endl;
}
```

`catch (WarehouseException& ex)` – will be executed for any exception of type `WarehouseException` or derived from `WarehouseException` (`ValidatorException`, `RepositoryException`)

Qt Toolkit

Qt is a cross-platform application and UI framework in C++.

Using Qt, you can write GUI applications once and deploy them across desktop, mobile and embedded operating systems without rewriting the source code.

Qt is supported on a variety of 32-bit and 64-bit platforms (Desktop, embedded, mobile).

- Windows (MinGW, MSVS)
- Linux (gcc)
- Apple Mac OS
- Mobile / Embedded (Windows CE, Symbian, Embedded Linux)

binding are available in C# ,Java, Python(PyQt), Ada, Pascal, Perl, PHP(PHP-Qt), Ruby(RubyQt)

Qt is available under GPL v3, LGPL v2 and commercial license.

Real applications developed using Qt:
Google Earth, KDE (desktop envirmoent for Unix-like OS), Adobe Photoshop Album, etc

Resources: qt.nokia.com

QT Tools and modules

- **Qt Library -** C++ class library, provides set of application building blocks needed to build cross-platform applications
- **Qt Creator** - integrated development environment (IDE) for developing QT applications
- **Qt Designer** – tool for designing and building GUI's from QT components
- **Qt Assistant** – tool for on-line documentation
- **Qt Linguist** – support for translating applications into local languages
- **qmake** – tool for the build process across diffrent platforms.
- **Qt Eclipse integration** – eclipse plugin to create c++ application and use the Qt Framework

Download install Qt, eclipse plugin

Prerequisite: Working Eclipse CDC (MinGW)

1) Download install Qt Library

http://qt.nokia.com/downloads → for windows **Qt libraries 4.8.1 for Windows (minGW 4.4, 319 MB)**

Provide:

- Qt SDK (library)
- Qt Designer, Assistant, Linguist, Samples and demos

2) Download and install the eclipse Qt plugin

Eclipse plugin can be downloaded from: http://qt.nokia.com/products/eclipse-integration/

Provide:

- Wizards for creating new Qt projects and classes
- Automated build setup
- Integrated form editor (QT Designer)

NOTE:

- plugin currentlly not work in Eclipse Indigo 64 bit (use 32 bit Indigo or Helios)
- the current Qt installer will issue a warning if you have a MinGW version different than 3.13. If you have a newer version of MinGW you should just ignore the warning and continue the installation.

Qt Hello World

Create a QT Eclipse Project: File → New → Qt GUI Project

Edit the main.cpp file.

```
int main(int argc, char *argv[]) {
	QApplication app(argc, argv);
	QLabel *label = new QLabel("hello world");
	label->show();
	return app.exec();
}
```

Build the project: Project → Build Project

Run the application

QApplication

The QApplication class manages the GUI application's control flow and main settings.

QApplication contains the main event loop, where all events from the window system and other sources are processed and dispatched.

For any GUI application using Qt, there is exactly one QApplication object (no matter whether the application has 0, 1, 2 or more windows at any given time)

Responsibility:

- initializes the application with the user's desktop settings
- performs event handling, it receives events from the underlying window system (x11, windows) and dispatches them to the relevant widgets.
- knows about the application's windows
- defines the application's look and feel

For non-GUI Qt applications, use QCoreApplication instead.

```
app.exec();
```

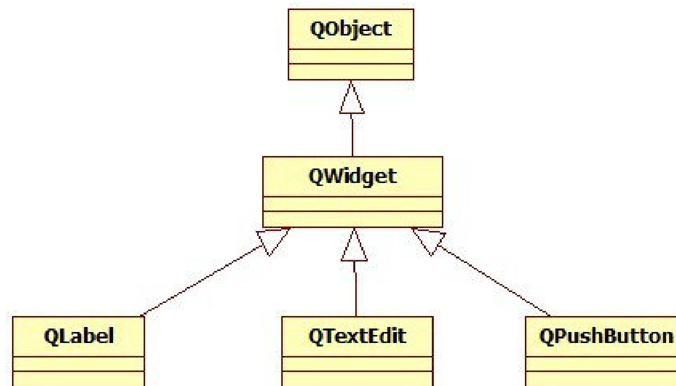
- makes the QApplication enter its event loop. When a Qt application is running, events are generated and sent to the widgets of the application.

Qt GUI Components (widgets)

Widgets are the basic building blocks for graphical user interface (GUI) applications built with Qt

- buttons, labels, text editor, etc

A GUI component (widget) can be placed on the user interface window or can be displayed as independent window.



A widget that is not embedded in a parent widget is called a window.

Windows provide the screen space upon which the user interface is built.

Windows separate applications visually from each other and usually provide a window decoration (show a title bar, allows the user to resize, position, etc)

Widget examples: Label, button, textbox, list

QLabel

- QLabel is used for displaying text or an image. No user interaction functionality is provided
- A QLabel is often used as a label for an interactive widget. For this use QLabel provides a useful mechanism for adding an mnemonic that will set the keyboard focus to the other widget (called the QLabel's "buddy").

```
QLabel *label = new QLabel("hello world");
label->show();


QLineEdit txt(parent);
QLabel lblName("&Name:", parent);
lblName.setBuddy(&txt);
```

QPushButton

The QPushButton widget provides a command button.

- Push (click) a button to command the computer to perform some action
- Push buttons display a textual label, and optionally a small icon, shortcut key can be specified by preceding the preferred character with an ampersand

```
QPushButton btn("&TestBTN");
btn.show();
```

Widget examples: Label, button, textbox, list

QLineEdit

- The QLineEdit widget is a one-line text editor.
- A line edit allows the user to enter and edit a single line of plain text with a useful collection of editing functions, including undo and redo, cut and paste, and drag and drop.

```
QLineEdit txtName;
txtName.show();
```

A related class is QTextEdit which allows multi-line, rich text editing.

QListWidget

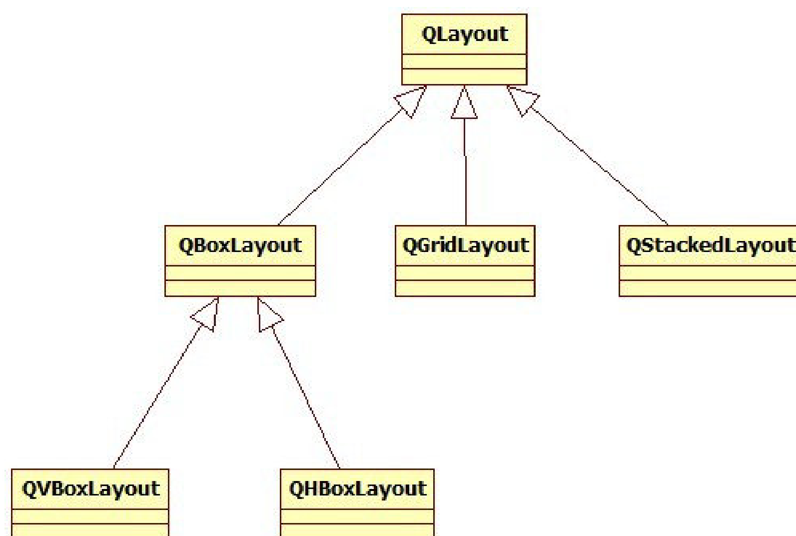
The QListWidget class provides an item-based list widget. The widget present a list of items to the user.

```
QListWidget *list = new QListWidget;
new QListWidgetItem("Item 1", list);
new QListWidgetItem("Item 2", list);
QListWidgetItem *item3 = new QListWidgetItem("Item 3");
list->insertItem(0, item3);
list->show();
```

For a more flexible list view widget, use the QListView class with a standard model.

Layout management

- any QWidget can have child widgets
- The Qt layout system provides way of automatically arranging child widgets within a widget to ensure that they make good use of the available space.
- Qt includes a set of layout management classes that are used to describe how widgets are laid out in an application's user interface.
- These layouts automatically position and resize widgets when the amount of space available for them changes, ensuring that they are consistently arranged and that the user interface as a whole remains usable.



Layout management

`QWidget *wnd = new QWidget;`
`QHBoxLayout *hLay = new QHBoxLayout();`
`QPushButton *btn1 = new QPushButton("Bt &1");`
`QPushButton *btn2 = new QPushButton("Bt &2");`
`QPushButton *btn3 = new QPushButton("Bt &3");`
`hLay->addWidget(btn1);`
`hLay->addWidget(btn2);`
`hLay->addWidget(btn3);`
`wnd->setLayout(hLay);`
`wnd->show();`
	`QWidget *wnd2 = new QWidget;`
	`QVBoxLayout *vLay = new QVBoxLayout();`
	`QPushButton *btnn1=new QPushButton("B&1");`
	`QPushButton *btnn2= new QPushButton("B&2");`
	`QPushButton *btnn3= new QPushButton("B&3");`
	`vLay->addWidget(btnn1);`
	`vLay->addWidget(btnn2);`
	`vLay->addWidget(btnn3);`
	`wnd2->setLayout(vLay);`
	`wnd2->show();`

GUI can be created by nesting multiple widgets and using different layouts

```
QWidget *wnd3 = new QWidget;
QVBoxLayout *vL = new QVBoxLayout;
wnd3->setLayout(vL);
//create a detail widget
QWidget *details = new QWidget;
QFormLayout *fL = new QFormLayout;
details->setLayout(fL);
QLabel *lblName = new QLabel("Name");
QLineEdit *txtName = new QLineEdit;
fL->addRow(lblName, txtName);
QLabel *lblAge = new QLabel("Age");
QLineEdit *txtAge = new QLineEdit;
fL->addRow(lblAge, txtAge);
//add detail to window
vL->addWidget(details);
QPushButton *store = new QPushButton("&Store");
vL->addWidget(store);
//show window
wnd3->show();
```

Layout management

addStretch() call tells the layout manager to consume space at that point in the layout (adds a stretchable space)

```
QHBoxLayout* btnsL = new QHBoxLayout;
btns->setLayout(btnsL);
QPushButton* store = new QPushButton("&Store");
btnsL->addWidget(store);
btnsL->addStretch();
QPushButton* close = new QPushButton("&Close");
btnsL->addWidget(close);
```

The layout manager will put any excess space between the Store and Close buttons

Layout management

Common pattern to build GUI:

- instantiate the required Qt widgets
- set properties if required
- add widgets to a layout (the layout manager will take care of the position and size)
- connect widgets using the signal and slot mechanism

Key benefits of using layout managers:

- provide a consistent behavior across different screen sizes and styles, handles resize

- provides sensible defaults for every type of widgets

- adapt automatically to different fonts and platforms: If the user changes the system's font settings, the application's forms will respond immediately, resizing themselves if necessary.

- Adapt automatically to different languages: if you the application's user interface translated to other languages, the layout classes take into consideration the widgets' translated contents to avoid text truncation.

- add a widget to a layout or remove a widget from a layout, the layout will automatically adapt to the new situation (same for *show() hide()* of a widget)

Absolute positioning

```
/**
 * Create GUI using absolute positioning
 */
void createAbsolute() {
	QWidget* main = new QWidget();
	QLabel* lbl = new QLabel("Name:", main);
	lbl->setGeometry(10, 10, 40, 20);
	QLineEdit* txt = new QLineEdit(main);
	txt->setGeometry(60, 10, 100, 20);
	main->show();
	main->setWindowTitle("Absolute");
}
/**
 * Create the same GUI using form layout
 */
void createWithLayout() {
	QWidget* main = new QWidget();
	QFormLayout *fL = new QFormLayout(main);
	QLabel* lbl = new QLabel("Name:", main);
	QLineEdit* txt = new QLineEdit(main);
	fL->addRow(lbl, txt);
	main->show();
	main->setWindowTitle("Layout");
	//fix the height to the "ideal" height
	main->setFixedHeight(main->sizeHint().height());
}
```

Absolute positioning disadvantages:

- The user cannot resize the window.
- Some text may be truncated (large font or change in the labels).
- The widgets might have inappropriate sizes for some styles.
- The positions and sizes must be calculated manually (error-prone, hard to maintenance)

Reference documentation

- Qt's reference documentation covers every class and function in Qt
- Documentation is available in HTML format in Qt's doc/html directory and can be read using any web browser
- Qt Assistant, the Qt help browser, which has searching and indexing features that make it quicker and easier to use than a web browser
- Documentation is available online at http://doc.trolltech.com/
- For every class : Detailed description, members (methods, attributes), signals, slots
- Inherited functions are documented in the base class

Signals and slots

- Signals and slots are used for communication between objects

- The signals and slots mechanism is a central feature of Qt

- the part that differs most from the features provided by other GUI frameworks.

- when we change one widget, we often want another widget to be notified.
  Ex. if a user clicks a **Close** button, we probably want the window's **close()** function to be called.

- Other toolkits achieve this kind of communication using callbacks.

- A callback is a pointer to a function, so if you want a processing function to notify you about some event you pass a pointer to another function (the callback) to the processing function.
- The processing function then calls the callback when appropriate.

- Callbacks in c++ have two fundamental flaws:
  - they are not type-safe (never be certain that the processing function will call the callback with the correct arguments)
  - the callback is strongly coupled to the processing function since the processing function must know which callback to call.

Signals and slots

- A **signal** is emitted when a particular event occurs. (ex.: cliked())
- Qt's widgets emit signals to indicate that a user action or a change of state has occurred
- A **slot** is a function that is called in response to a particular signal.
- A **signal** can be connected to a function (called a **slot**), so that when the signal is emitted, the slot is automatically executed

```
QPushButton *btn = new QPushButton("&Close");
QObject::connect(btn,SIGNAL(clicked()),&app,SLOT(quit()));
btn->show();
```

- The signature of a signal must match the signature of the receiving slot. (In fact a slot may have a shorter signature than the signal it receives because it can ignore extra arguments.)
- **Slots** can be used for receiving signals, but they are also normal member functions. Just as an object does not know if anything receives its signals, a slot does not know if it has any signals connected to it. This ensures that truly independent components can be created with Qt.
- Qt's widgets have many predefined signals, but we can subclass widgets to add new signals to them.
- Qt's widgets have many pre-definedd slots
- it is common practice to subclass widgets and add your user-defined slots that can handle the signals

Connect signals and slots

To connect a signal and a slot we use `QObject::connect` and the SIGNAL and SLOT macros

```
QWidget* createButtons(QApplication &a) {
	QWidget* btns = new QWidget;
	QHBoxLayout* btnsL = new QHBoxLayout;
	btns->setLayout(btnsL);
	QPushButton* store = new QPushButton("&Store");
	btnsL->addWidget(store);
	QPushButton* close = new QPushButton("&Close");
	btnsL->addWidget(close);
	//connect the clicked signal from close button to the quit slot (method)
	QObject::connect(close,SIGNAL(clicked()),&a,SLOT(quit()));
	return btns;
}
```

A **slot** is called when a **signal** connected to it is emitted. **Slots** are normal C++ functions and can be called normally; their only special feature is that signals can be connected to them.

several **slots** can be connected to one **signal**, the **slots** will be executed one after the other, in the order they have been connected, when the **signal** is emitted

the signature of a **signal** must match the signature of the receiving **slot.**
a **slot** may have a shorter signature than the signal it receives because it can ignore extra arguments.

Connect signals and slots

```
QSpinBox *spAge = new QspinBox();
QSlider *slAge = new QSlider(Qt::Horizontal);
```

```
//Synchronise the spinner and the slider
//Connect spin box - valueChanged to slider setValue
QObject::connect(spAge, SIGNAL(valueChanged(int)),slAge,SLOT(setValue(int)));
//Connect - slider valueChanged to spin box setValue
QObject::connect(slAge, SIGNAL(valueChanged(int)),spAge,SLOT(setValue(int)));
```

If the the user change the value in the Spin Box:

- the spinbox will **emits** the **signal** valueChanged(int) with an int argument, the current value of the spinner
- because the spinner and slider are connected, the setValue(int) method of the slider will be invoked. The argument from the valueChanged method (the current value of the spinner) will be passed to the setValue method of the slider
- the slider update itself to reflect the new value, and emits a valueChanged **signal** because its own value changed
- because the slider is also connected with the spinner, the setValue slot of the spinner is invoked, in response to the slider's signal.
- The setValue of the spinner will not emit any signal because the current value is the same as the value provided in the setValue. (prevents infinite loop)

Subclassing QWidget

- create separate class for the GUI
- make it an independent, self-contained component, with its own signals and slots

```
/**
 * GUI for storing Persons
 */
class StorePersonGUI: public QWidget {
public:
	StorePersonGUI();
private:
	QLabel *lblName;
	QLineEdit *txtName;
	QLabel *lblAdr;
	QLineEdit *txtAdr;
	QSpinBox *spAge;
	QLabel *lblAge2;
	QSlider *slAge;
	QLabel *lblAge3;
	QPushButton* store;
	QPushButton* close;
	/**
	 * Assemble the GUI
	 */
	void buildUI();
	/**
	 * Link signals and slots to define the behaviour of the GUI
	 */
	void connectSignalsSlots();
};
```

The class extends Qwidget, QMainWindow, QDialog etc.

QMainWindow

- A main window provides a framework for building an application's user interface.
- QMainWindow has its own layout to which you can add QToolBars, QDockWidgets, a QMenuBar, and a QstatusBar.

QMainWindow layout:

- Menu – on the top

```
QAction *openAction = new QAction("&Load", this);
QAction *saveAction = new QAction("&Save", this);
QAction *exitAction = new QAction("E&xit", this);
fileMenu = menuBar()->addMenu("&File");
fileMenu->addAction(openAction);
fileMenu->addAction(saveAction);
fileMenu->addSeparator();
fileMenu->addAction(exitAction);
```

- Toolbars

```
QToolBar* fileToolBar = addToolBar("&File");
fileToolBar->addAction(openAction);
fileToolBar->addAction(saveAction);
```

- Central widget - in the center of the window

```
    middle = new QTableWidget(10, 10, this);
    setCentralWidget(middle);
```

- Status bar - on the bottom

```
statusBar()->showMessage("Status Message ....");
```

Qt Build system

A standard c++ application consist of header (.h) and source (.cpp) files

The build process for a c++ application:

- compile the cpp files using a compiler (the source files include the header files) → object files (.o)
- link edit the object files using a linker → executable file (.exe)

Qt add some additional steps:

The **meta-object compiler** (moc)
- the meta-object compiler takes all classes starting with the Q_OBJECT macro and generates a moc_*.cpp C++ source file. This file contains information about the class being moc'ed such as class name, inheritance tree, etc, but also implementation of the signals. This means that when you emit a signal, you actually call a function generated by the moc.

The **user interface compiler**
– The user interface compiler takes designs from Designer and creates header files. These header files are then included into source files as usual, making it possible to call setupUi to instantiate a user interface design.

The **Qt resource compiler**
– It makes it possible to embed images, text files, etc into your executable, but still to access them as files. We will look at this later, I just want to include it in this picture where it belongs.

Qt Build steps – command line

Run :

- qmake -project
  - generate a qt project file (.pro)
- qmake
  - based on a project file creates a make file
- make
  - execute the make file generated by qmake. This will invoke the tools required to build a qt project (meta-object compiler, user interface compiler, resource compiler, c++ compiler, linker)

User defined signals and slots - Q_OBJECT

We may define custom slots and signals.

```
class Notepad : public QMainWindow
{
    Q_OBJECT
...
```

The Q_OBJECT macro must be first in the class definition, and declares the class as a QObject
The Q_OBJECT macro at the beginning of the class definition is necessary for all classes that define signals or slots.

Qt introduce a mechanism called the **meta-object system** this provides two key services:

- signals–slots
- introspection.

The introspection functionality is necessary for implementing signals and slots, and allows application programmers to obtain “meta-information” about QObject subclasses at run-time, including the list of signals and slots supported by the object and its class name.

The moc tool (moc.exe, meta object compiler) parses Q_OBJECT class definitions and makes the meta-information available through C++ functions. The moc tool will generate all the c++ code that allow introspection (in a separate file called *.moc)

User defined signals and slots - slots

Custom slots are declared using the *slots* keyword (is actually a macro). Qt (moc utility) need this for generating meta-information about the available slots
The slot is just a regular method.

```
class Notepad : public QMainWindow
{
    Q_OBJECT

public:
    Notepad();

private slots:
    void open();
    void save();
    void quit();
```

```
void Notepad::save()
{
...
}
```

User defined signals and slots - signals

Using the *signals* macro we can define custom signals.

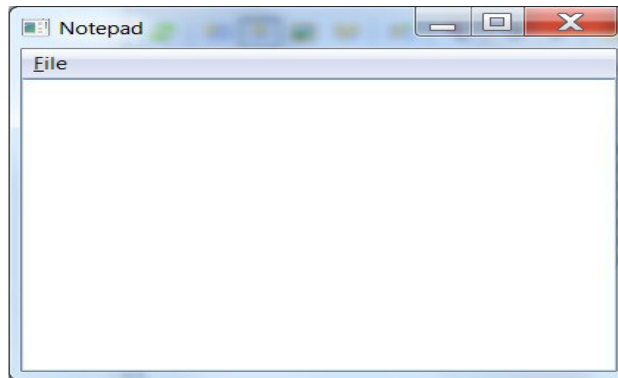
```
private signals:
     storeRq(QString* name,QString* adr );
```

Using the **emit** keyword we can emit signals

```
emit storeRq(txtName->text(),txtAdr->text());
```

The user-defined signals and slots can be connected just like any signal, slot defined by the standard Qt widgets

Notepad example




```
class Notepad : public QMainWindow
{
    Q_OBJECT

public:
    Notepad();

private slots:
    void open();
    void save();
    void quit2();
```

```

    openAction = new QAction(tr("&Load"), this);
    saveAction = new QAction(tr("&Save"), this);
    exitAction = new QAction(tr("E&xit"), this);

    connect(openAction, SIGNAL(triggered()), this, SLOT(open()));
    connect(saveAction, SIGNAL(triggered()), this, SLOT(save()));
    connect(exitAction, SIGNAL(triggered()), this, SLOT(quit2()));
```

```
void Notepad::open()
{
    QString fileName = QFileDialog::getOpenFileName(this, tr("Open File"), "",
        tr("Text Files (*.txt);;C++ Files (*.cpp *.h)"));

    if (fileName != "") {
        QFile file(fileName);
        if (!file.open(QIODevice::ReadOnly)) {
            QMessageBox::critical(this, tr("Error"), tr("Could not open file"));
            return;
        }
        QTextStream in(&file);
        textEdit->setText(in.readAll());
        file.close();
    }
```

Working with QtDesigner in Eclipse

Eclipse Qt GUI Project

File ->New->Qt GUI Project

- generate the qt project structure (set the includes, directory, etc)
- .ui – user interface file
  - the uic (user interface copiller) utility will transform the .ui file into a c++ (ui_<name>.h)
- a GUI component class (.h, .cpp) - extends Qwidget or some subclass (QDialog, QmainWindow). Here you can add user defined slots, signals
- main cpp file

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    ProductRep w;
    w.show();
    return a.exec();
}
```

Obs. The current verision of the Qt eclipse plugin is not handling well the interactive code analysis when you mix standard c++ classes (cout,iostream, etc) with Qt library classes.

Deactivate the Code Analysis (Project->Properties->c/c++ general->Code Analyses uncheck all).

Errors will be reported after you build the project.

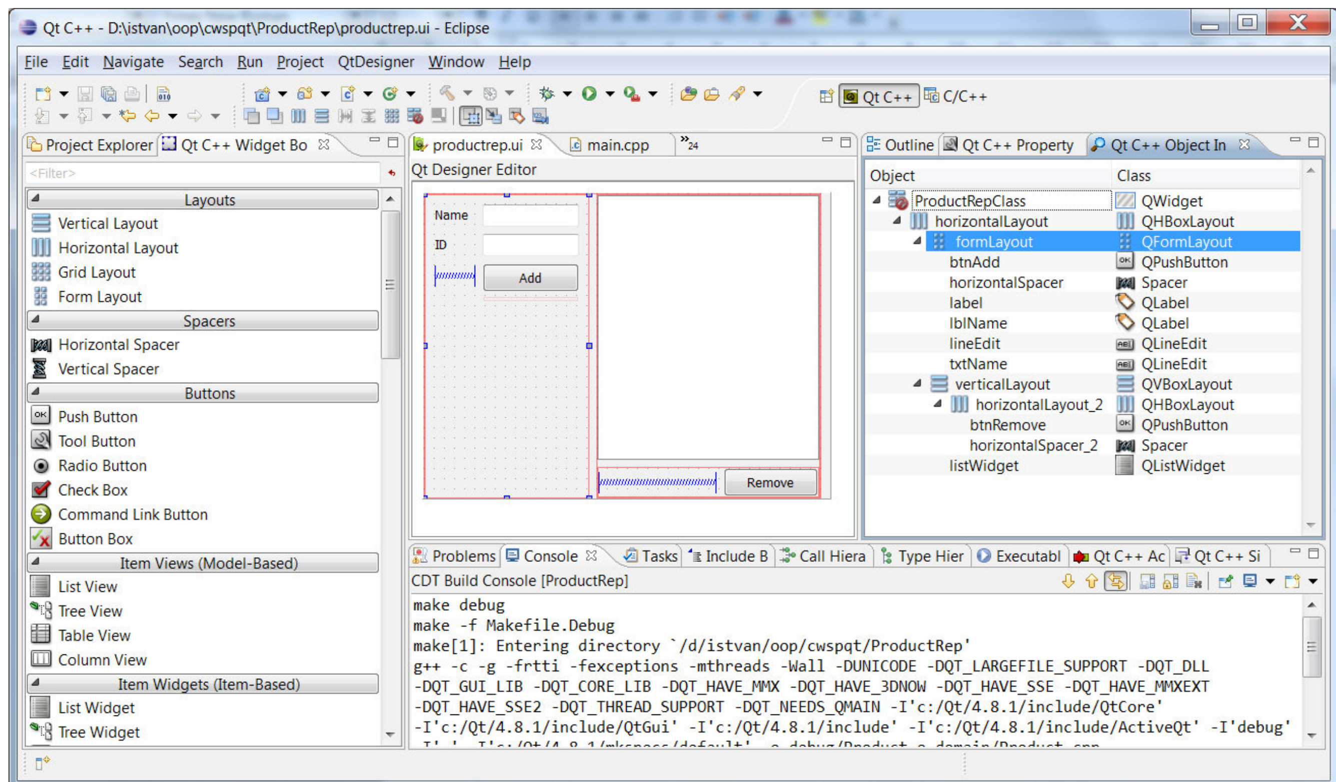
Visual (drag & drop) design of graphic user interfaces

Eclipse Qt plugin provide a visual approach for designing forms

- is not unusual for Qt programmers to create Qt applications purely by writing C++
- can be faster than hand-coding
- experiment with different design quickly

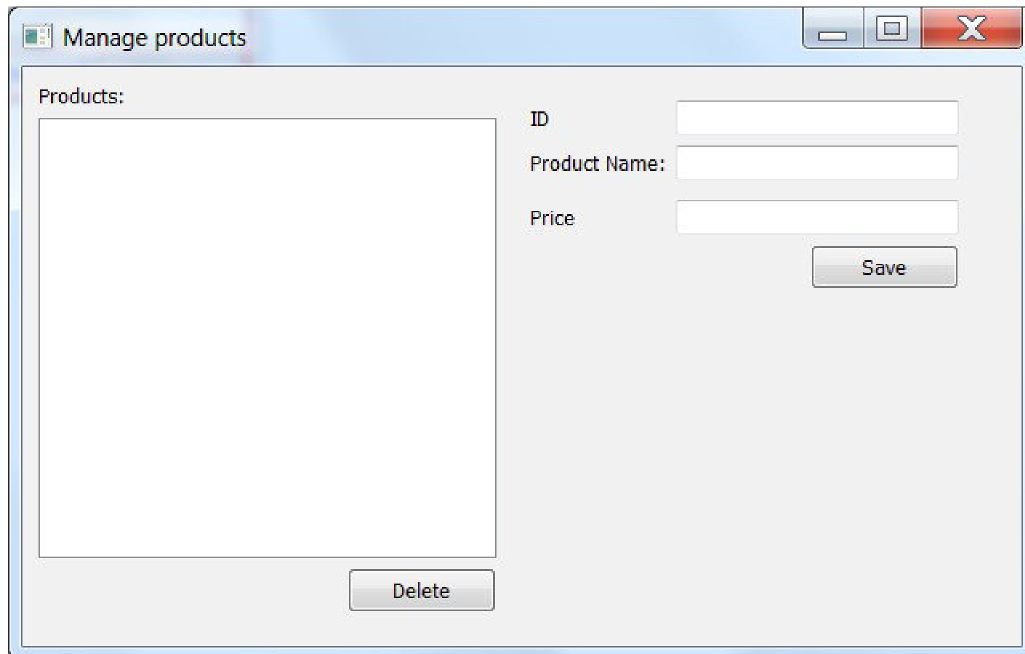
Eclipse Qt editor/views:

- Qt Designer editor – design the GUI
- Qt C++ Widget Box - can drag widgets to the form
- Qt C++ Object Inspector – show the widgets hierarchy
- Qt C++ Property Editor – set properties of widgets



Master detail – Product

CRUD (Create Read Update Delete) for Product



User defined slots

<table>
<tr>
<td>

```
class testSlots: public QWidget
{
Q_OBJECT

public:
	testSlots(Warehouse* wh,
QWidget *parent = 0);
	~testSlots();

private:
	Ui::testSlotsClass ui;
	Warehouse* wh;
	void connectSS();
	void reloadList();

private slots:
	void save();
	void productSelected();
};
```

</td>
<td>

```
/**
 * Save products
 */
void testSlots::save() {
 int id = ui.txtID->text().toInt();
 double price = ui.txtPrice->text().toDouble();
 string desc = ui.txtName->text().toStdString();
 try {
  wh->addProduct(id, desc, price);
  reloadList();
  QMessageBox::information(this, "Info", "Product
saved...");
 } catch (WarehouseException ex) {
   QMessageBox::critical(this, "Error",
              QString::fromStdString(ex.getMsg()));
 }
}
```

</td>
</tr>
</table>

QString

- provide an ADT implementation for character sequences (Unicode character string)
- used frequently in the Qt classes

Create QString

```
QString s1 = "Hello";
QString s2("World");
```

QString and string (STL)

```
string str = "Hello";
QString qStr = QString::fromStdString(str);
string str2 = qStr.toStdString();
```

Numbers and QString

```
QString s3 = QString::number(2);
QString s4 = QString::number(2.5);
QString s5 = "2";
int i = s5.toInt();
double d = s5.toDouble();
```

QListWidget – load, itemSelectionChanged()

```
private slots:
	void save();
	void productSelected();

/**
 * Save product
 */
void testSlots::save() {
	int id = ui.txtID->text().toInt();
	double price = ui.txtPrice->text().toDouble();
	try {
		wh->addProduct(id, ui.txtName->text().toStdString(), price);
		reloadList();
		QMessageBox::information(this, "Info", "Product saved...");
	} catch (WarehouseException ex) {
		QMessageBox::critical(this, "Error",
				QString::fromStdString(ex.getMsg()));
	}
}


/**
 * Load the products into the list
 */
void testSlots::reloadList() {
	ui.lstProducts->clear();
	DynamicArray<Product*> all = wh->getAll();
	for (int i = 0; i < wh->getNrProducts(); i++) {
		string desc = all.get(i)->getDescription();
		QListWidgetItem *item = new QListWidgetItem(
				QString::fromStdString(desc), ui.lstProducts);
		item->setData(Qt::UserRole, QVariant::fromValue(all.get(i)->getCode()));
	}
}
```

Qt ItemView classes

QListWidget, QTableWidget , QTreeWidget

Item widgets are populated with the entire content of a data set (items: QListWidgetItem, QTableWidgetItem, QTreeWidgetItem).

Searches, edits are performed on the data held in the widgets

the data need to be synchronized, write back to the data source (file, database, network)

Pros:

- simple to understand
- simple to use

Cons:

- do not scale well to very large data sets
- do not work if we have multiple views of the same data set
- requires data duplication

Model-View-Controller

Flexibile approach to visualizing large data sets

the **model:** represents the data set, is responsible for:

- fetching the data that is needed for view
- writing back any changes

the **view**: presents data to the user.

- Even if we have a large dataset, only a limited amount of data is visible. That is the only data that is requested by the view

the **controller**: mediates between the user and the view

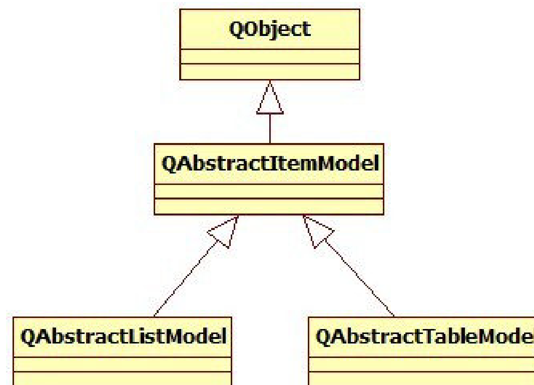
- convert user actions into requests (to navigate or edit the data)

Model/View architecture

- Model/View is a technology used to separate data from their visual representation (views)
- handle large data sets, complex data items, database integration, multiple data views
- Qt 4 > provide a set of model/view classes (list, table, tree)
- Model/View architecture from Qt is inspired from the MVC Pattern (Model-View-Controller), but instead of controller, Qt use a different abstraction called **delegate**
- **delegate** is used to provide fine control over how items are rendered and edited
- Qt provides a default delegate for every type of view (sufficient for most applications)
- Qt Item Views : QListView, QTableView, QTreeView and associated model classes

Implementing custom models

- create a new class for the model (list model, table model)
- extend the appropriate Qt class



QAbstractItemModel – class representing the model for any Qt Item View Class.
Able to represent table data (row, columns) or hierarchical data (tree structure child, parent)
The underlying data is exposed as a hierarchy of tables.

Every item has a number of data elements associated with different roles
Data elements can be retrieved by specifying the role

```
void testSlots::reloadList() {
	ui.lstProducts->clear();
	DynamicArray<Product*> all = wh->getAll();
	for (int i = 0; i < wh->getNrProducts(); i++) {
		string desc = all.get(i)->getDescription();
		QListWidgetItem *item = new QListWidgetItem(
				QString::fromStdString(desc), ui.lstProducts);
		item->setData(Qt::UserRole, QVariant::fromValue(all.get(i)->getCode()));
	}
}
/**
 * Load the selected product into the detail panel
 */
void testSlots::productSelected() {
	QList<QListWidgetItem*> sel = ui.lstProducts->selectedItems();
	if (sel.size() == 0) {
		return;
	}
	QVariant idV = sel.first()->data(Qt::UserRole);
	int id = idV.toInt();
	const Product *p = wh->getByCode(id);
	ui.txtID->setText(QString::number(id));
	ui.txtName->setText(QString::fromStdString(p->getDescription()));
	ui.txtPrice->setText(QString::number(p->getPrice()));
	QMessageBox::information(this, "Info", "Product selected...");
}
```

Implementing custom models

```
class MyTableModel: public QAbstractTableModel {
public:
	MyTableModel(QObject *parent);
	/**
	 * number of rows
	 */
	int rowCount(const QModelIndex &parent = QModelIndex()) const;
	/**
	 * number of columns
	 */
	int columnCount(const QModelIndex &parent = QModelIndex()) const;
	/**
	 * Value at a given position
	 */
	QVariant data(const QModelIndex &index, int role = Qt::DisplayRole) const;

};
```

```
MyTableModel::MyTableModel(QObject *parent) :
		QAbstractTableModel(parent) {
}

int MyTableModel::rowCount(const QModelIndex & /*parent*/) const {
	return 100;
}

int MyTableModel::columnCount(const QModelIndex & /*parent*/) const {
	return 2;
}

QVariant MyTableModel::data(const QModelIndex &index, int role) const {
	if (role == Qt::DisplayRole) {
		return QString("Row%1, Column%2").arg(index.row() + 1).arg(
				index.column() + 1);
	}
	return QVariant();
}
```

We can create models that only fetch the data that is actually needed for display.

Using predefined models

Predefined models:

- **QStringListModel** - Stores a list of strings
- **QStandardItemModel** - Stores arbitrary hierarchical data
- **QDirModel** - Encapsulates the local file system
- **QSqlQueryModel** - Encapsulates an SQL result set
- **QSqlTableModel** - Encapsulates an SQL table
- **QSqlRelationalTableModel** - Encapsulates an SQL table with foreign keys
- **QSortFilterProxyModel** - Sorts and/or filters another model

```
void createTree() {
      QTreeView *tV = new QTreeView();
      QDirModel *model = new QDirModel();
      tV->setModel(model);
      tV->show();
}
```

Control the text appearance

enum Qt::ItemDataRole	Meaning	Type
Qt::DisplayRole	text	QString
Qt::FontRole	font	QFont
Qt::BackgroundRole	brush for the background of the cell	QBrush
Qt::TextAlignmentRole	text alignment	enum Qt::AlignmentFlag
Qt::CheckStateRole	suppresses checkboxes with QVariant(), sets checkboxes with Qt::Checked or Qt::Unchecked	enum Qt::ItemDataRole

```
QVariant MyTableModel::data(const QModelIndex &index, int role) const {
	int row = index.row();
	int column = index.column();
	if (role == Qt::DisplayRole) {
		return QString("Row%1, Column%2").arg(row + 1).arg(column + 1);
	}
	if (role == Qt::FontRole) {
		QFont f;
		f.setItalic(row % 4 == 1);
		f.setBold(row % 2 == 1);
		return f;
	}
	if (role == Qt::BackgroundRole) {
		if (column == 1 && row % 2 == 0) {
			QBrush bg(Qt::red);
			return bg;
		}
	}
	return QVariant();
}
```

Table headers

The model also controls the headers (column, row headers) for a table

Re-implement the `QVariant headerData(int section, Qt::Orientation orientation,int role)` method

```
QVariant MyTableModel::headerData(int section, Qt::Orientation orientation,
		int role) const {
	if (role == Qt::DisplayRole) {
		if (orientation == Qt::Horizontal) {
			return QString("col %1").arg(section);
		}else {
			return QString("row %1").arg(section);
		}
	}
	return QVariant();
}
```

Synchronize the model and the view

If the underlying data in the model is changed the view need to be repainted

The view is automatically connected (in the view.setModel method) to the **dataChanged** signal.

If the model is changed we need to emit the dataChanged signal

```
/**
 * Slot invoked by the timer
 */
void MyTableModel::timerTikTak() {
      QModelIndex topLeft = createIndex(0, 0);
      QModelIndex bottomRight = createIndex(rowCount(), columnCount());
      emit dataChanged(topLeft, bottomRight);
}
```

Multiple views for the same model

Multiple views attached to the same model allow the user to interact with the data in different ways

Qt automatically keeps multiple views in sync, reflecting changes in the model

```
QTableView* tV = new QTableView();
MyTableModel *model = new MyTableModel(tV);
tV->setModel(model);
tV->show();

QListView *tVT = new QListView();
tVT->setModel(model);
tVT->show();
```

Edit model values

Redefine methods in the model:

```
bool MyTableModel::setData(const QModelIndex & index, const QVariant & value, int role)
```

```
Qt::ItemFlags MyTableModel::flags(const QModelIndex & /*index*/)
```

When the model changed do not forget to emit a dataChanged signal to notify all the views

```
/**
 * Invoked on edit
 */
bool MyTableModel::setData(const QModelIndex & index, const QVariant & value,
             int role) {
      if (role == Qt::EditRole) {
             int row = index.row();
             int column = index.column();
             //save value from editor to member m_gridData
             m_gridData[index.row()][index.column()] = value.toString();
             //make sure the dataChange signal is emitted so all the views will be
notified
             QModelIndex topLeft = createIndex(row, column);
             emit dataChanged(topLeft, topLeft);
      }
      return true;
}

Qt::ItemFlags MyTableModel::flags(const QModelIndex & /*index*/) const {
      return Qt::ItemIsSelectable | Qt::ItemIsEditable | Qt::ItemIsEnabled;
}
```

Design Pattern

- Design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context

- General, reusable, solution to a commonly occurring problem within a given context in software design

- Object-oriented design patterns show relationships and interactions between classes or objects

- Christopher Alexander: "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"

- Design Patterns: Elements of Reusable Object-Oriented Software – 1994

- Gang of Four (GoF)- Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides

- Introduce the principles of design patterns and then offer a catalog of such patterns

Design patterns-types:

- **Creational**
  - abstracts the instantiation process
  - Abstract Factory, Builder, Factory Method, Prototype, Singleton

- **Structural**
  - are concerned with how classes are composed to form larger structures
  - Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy

- **Behavioral**
  - concerned with algorithms and the assignment of responsibilities between objects
  - Chain of responsibility ,Command Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template method, Visitor

Design pattern elements

- pattern name
    - describe in a word or two the design problem, its solution and consequences
    - part of the software developer vocapbulary

- problem
    - describe when to apply the pattern. Describe the problem and its context.

- solution
    - describe the elements that make up the design, their relationships, responsibilities and collaborations
    - provide and abstract descriprion of a design problem and how general arrangement of elements (classes and objects) solves it

- consequences
    - describe the results and trade-offs of applying the pattern.

Standard Template Library (STL)

- The Standard Template Library, or STL, is a C++ library, part of the C++ Standard Library
- Provides many of the fundamental algorithms and data structures needed to develop programs
- The STL is a generic library, meaning that its components are heavily parameterized: almost every component in the STL is a template.
- The Standard Template Library is designed such that programmers create components that can be composed easily without losing any performance. (style of programming called generic programming)
  - moves the effort that is done at run-time in object-oriented programming (dynamic binding) to compile-time, using templates
- STL contains classes for:
  - containers
  - iterators
  - algorithms
  - function objects
  - allocators

Containers

A container is a holder object that stores a collection of other objects (its elements). They are implemented as class templates, which allows a great flexibility in the types supported as elements.

The container manages the storage space for its elements and provides member functions to access them, either directly or through iterators (reference objects with similar properties to pointers).

Containers are share some common functionality (methods):

- allow element access (ex.: [])
- methods to manage capacity (ex.: size())
- modifiers (ex.: insert, clear)
- iteratior (begin(), end()
- operations (ie: find)

The decision of which type of container to use for a specific need depends:

- on the functionality offered by the container
- the efficiency of some of its members (complexity).

Container class templates

- Sequence containers: **vector<T>**, **deque<T>**, **list<T>**
- Container adaptors: **stack<T, ContainerT>**, **queue<T, ContainerT>**, **priority_queue<T, ContainerT, CompareT>**
- Associative containers: **set<T, CompareT>**, **multiset<T, CompareT>**, **map<KeyT, ValueT, CompareT>**, **multimap<KeyT, ValueT, CompareT>**, **bitset<T>**

Sequence containers:

Common functionality:

Sequence containers, offer different trade-offs in complexity between inserting/removing elements and accessing them.

- Vector (Dynamic Array): like array + capacity is handled automatically
  - Elements stored in sequential memory block
  - Vector is good for:
    - Accessing individual elements by their position index (constant time).
    - Iterating over the elements in any order (linear time).
    - Add and remove elements from its end (constant amortized time).

- Deque (double ended queue)
  - Elements in a deque can be divided in several chunks of storage
  - Deque is good for:
    - Elements can be efficiently added and removed from any of its ends (either the beginning or the end of the sequence).

- List
  - implemented as double-linked lists
  - List is good for:
    - Efficient insertion and removal of elements anywhere in the container (constant time).
    - Efficient moving elements and block of elements within the container or even between different containers (constant time).
    - Iterating over the elements in forward or reverse order (linear time).

Container operations complexity

`#include <vector>`
`void sampleVector() {`
` vector<int> v;`
` v.push_back(4);`
` v.push_back(8);`
` v.push_back(12);`
` v[2] = v[0] + 2;`
` int lg = v.size();`
` for (int i = 0; i < lg; i++)`
` {`
`   cout << v.at(i) << " ";`
` }`
`}`
	`#include <deque>`
	`void sampleDeque() {`
	` deque<double> dq;`
	` dq.push_back(4);`
	` dq.push_back(8);`
	` dq.push_back(12);`
	` dq[2] = dq[0] + 2;`
	` int lg = dq.size();`
	` for (int i = 0; i < lg; i++)`
	` {`
	`   cout << dq.at(i) << " ";`
	`  }`
	`}`
		`#include <list>`
		`void sampleList() {`
		` list<double> l;`
		` l.push_back(4);`
		` l.push_back(8);`
		` l.push_back(12);`
		` while (!l.empty()) {`
		`   cout << " " << l.front();`
		`   l.pop_front();`
		` }`
		`}`

Vector : constant time O(1) random access; insert/delete at the end

Deque: constant time O(1) insert/delete at the either end

List: constant time O(1) insert / delete anywhere in the list

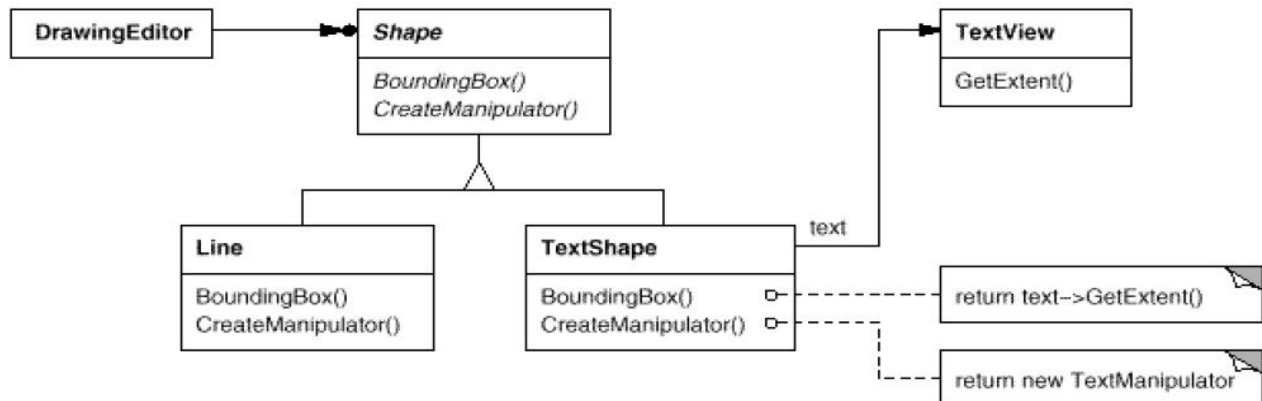
Vector vs Deque

- Random access to elements is more efficient on vector, than deque
- Deque allows insertion/removal of elements from arbitrary positions more efficient than the vector, but not in constant time
- For large sequences of elements vectors will allocate large memory blocks while deque will allocate many small memory blocks – more efficient from operating systems point of view

Adapter pattern (Wrapper)

Intent: Convert the interface of a class into another interface clients expects. Adapter lets classes work together that couldn't otherwise becouse of incompatible interface.

Motivation: Sometimes a toolkit class that's designed for reuse isn't reusable only because its interface doesn't match the domain-specific interface an aplication requires.
Ex. Draw Editor (Shape: lines, polygons, etc) Add TextShape. Solution Adapt the existing TextView class. TexShape adapts the TextView interface to Shape's

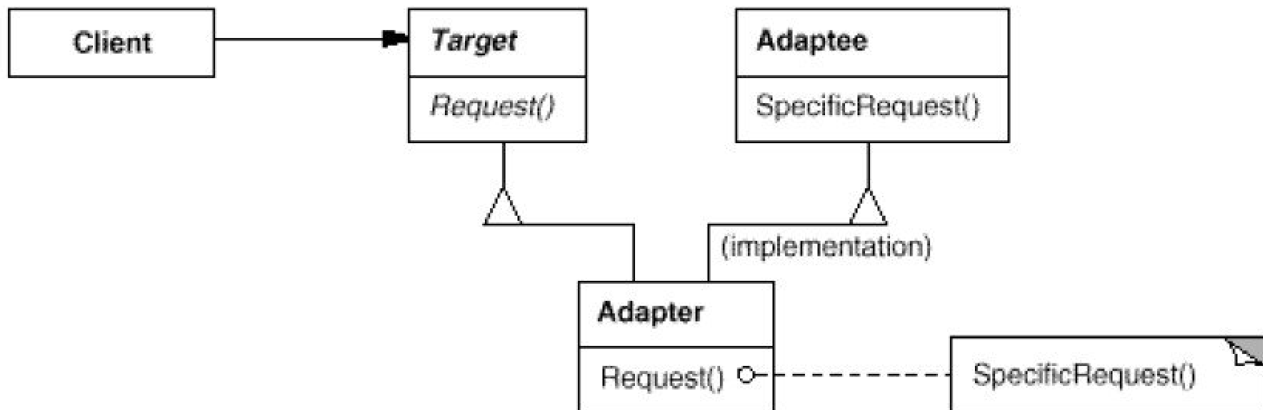


Applicability:

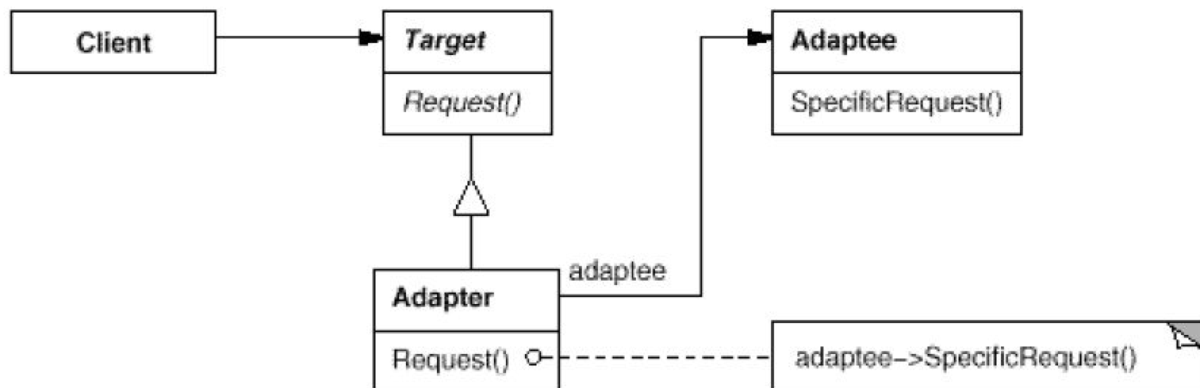
- want to use an existing class, and its interface does not match the one you need
- want to create a reusable class that cooperates with unrelated classes (don't have compatible interfaces)
- need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

Adapter pattern structure

Class adapter use multiple inheritance to adapt one interface to another



Object adapter use composition



Participants:

- Target: defines the domain-specific interface that Clinet uses.
- Client: collaborates with objects conforming to the Target interface.
- Adaptee: defines an existing interface that needs adapting.
- Adapter: adapts the interface of Adaptee to the Target interface.

Adapter

Collaborations:

- Clients call operations on an Adapter instance. In turn, the adapter calls Adaptee operations that carry out the request.

Consequences:

A class adapter:

- wont work when we want to adapt a class and all its subclasses
- lets Adapter override some of Adaptee's behavior
- introduce only one object, and no aditional pointer indirection is needed to get to the adaptee

An object adapter:

- lets a single Adapter work with many Adaptees.
- Makes it harder to override Adaptee behavior (will require subclassing Adaptee and making Adapter refer to the subclass)

Adapter pattern used in STL: Container adadpters, Iterator adapters

Container adaptors

Containers adaptors, are classes that use an encapsulated object of a specific container class as its underlying container, providing a specific set of member functions to access its elements

Use the adapter pattern: Stack, Queue, Priority Queue is an adapter. Has a template parameter of type Sequence Container and only exports the operataions that are alowed on Stack, Queue, Priority Queue

- Stack: LIFO (last in first out) strategy for insert/extract elements
  - Elements are pushed/popped from the "back" of the specific container, which is known as the top of the stack.
  - Operations: empty(), push(), pop(), top()
  - `template < class T, class Container = deque<T> > class stack;`
    - T: the type of elements
    - Container: Type of the underlying container object used to store and access the elements

- queue: FIFO (first in first out) strategy for insert/extract elements
  - Elements are pushed into the "back" of the specific container and popped from its "front"
  - operations: empty(), front(), back(), push(), pop(), size();
  - `template < class T, class Container = deque<T> > class queue;`

- priority_queue: extract elements based on their priority
  - similar to a heap where only the max heap element can be retrieved (the one at the top in thepriority queue) and elements can be inserted indefinitely.
  - operations: empty(), top(), push(), pop(), size();
  - ```
    template < class T, class Container = vector<T>,
     class Compare = less<typename Container::value_type> > class priority_queue;
    ```
    ```

Container adaptor samples

```cpp
#include <stack>
void sampleStack() {
 stack<int> s;
 //stack<int,deque<int>> s;
 //stack<int,list<int> > s;
//stack<int,vector<int>> s;
 s.push(3);
 s.push(4);
 s.push(1);
 s.push(2);
 while (!s.empty()) {
    cout << s.top() << " ";
    s.pop();
 }
}
```

```cpp
#include <queue>
void sampleQueue() {
//queue<int> s;
//queue<int,deque<int>>s;
queue<int, list<int> > s;

 s.push(3);
 s.push(4);
 s.push(1);
 s.push(2);
 while (!s.empty()) {
  cout << s.front() << " ";
  s.pop();
 }
}
```

```cpp
#include <queue>
void samplePriorQueue() {
//priority_queue<int> s;
//priority_queue<int,deque<int>> s;
//priority_queue<int,list<int>> s;

priority_queue<int,vector<int> > s;
 s.push(3);
 s.push(4);
 s.push(1);
 s.push(2);
 while (!s.empty()) {
   cout << s.top() << " ";
   s.pop();
 }
}
```

Associative containers:

Associative containers are containers especially designed to be efficient accessing its elements by their key (unlike sequence containers, which are more efficient accessing elements by their relative or absolute position).

- set
  - container that stores unique elements, and in which the elements themselves are the keys
  - no two elements in the set can compare equal to each other
  - typically implemented as binary search trees.

- multiset
  - ame properties as set containers, but allowing for multiple keys with equal values.

- Map
  - stores elements formed by the combination of a key value and a mapped value
  - no two elements in the map have keys that compare equal to each other
  - Each element is composed of a key and a mapped value

- multimap
  - like map containers, but allowing different elements to have the same key value.

- Bitset
  - special container class that is designed to store bits (elements with only two possible values: 0 or 1,true or false, ...).

```
void sampleMap() {
	map<int, Product*> m;
	Product *p = new Product(1, "asdas", 2.3);
	//add code <=> product
	m.insert(pair<int, Product*>(p->getCode(), p));

	Product *p2 = new Product(2, "b", 2.3);
	//add code <=> product
	m[p2->getCode()] = p2;

    //lookup
	cout << m.find(1)->second->getName()<<endl;
	cout << m.find(2)->second->getName()<<endl;
}
```

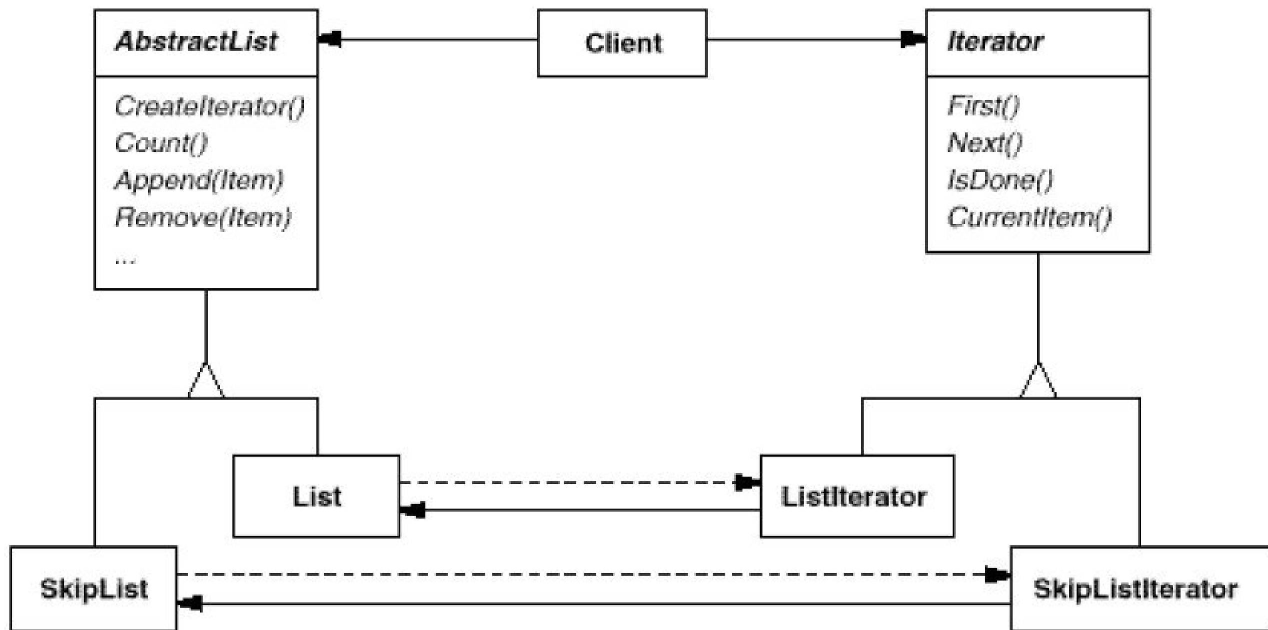
Iterator (Cursor) design pattern

Intent: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation

Motivation:

- An agregate (such as a list) should provide a way to access its elements without exposing its internal structure.
- Should allow to traverse the object in diffrent ways

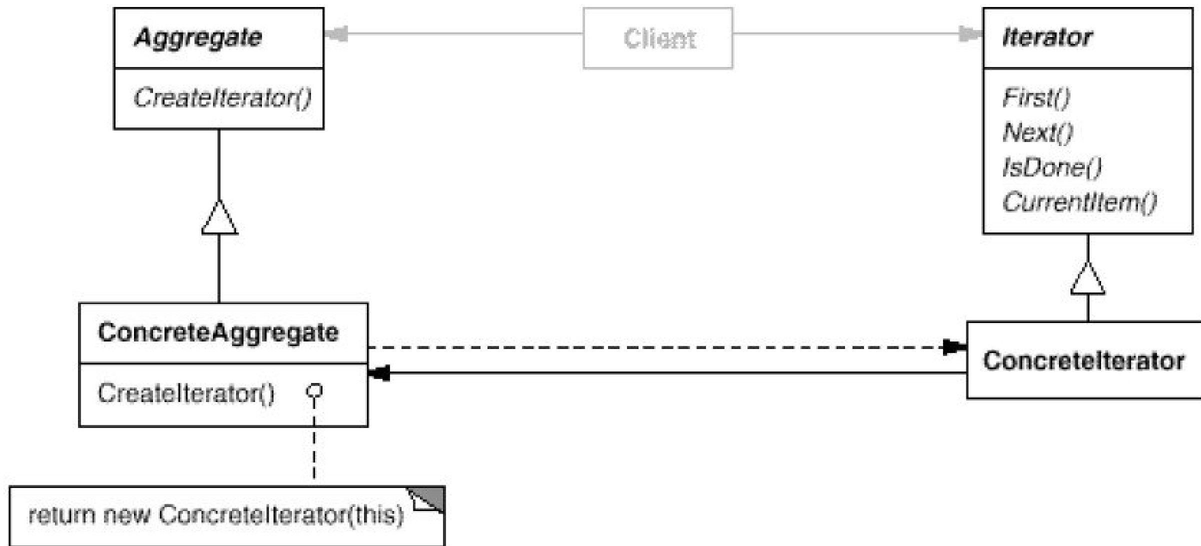
Example: List, SkipList, Iterator



Applicability:

- to access an agregate object's withput exposing its internal representation
- to support multiple traversals of aggregate objects
- to provide an uniform interface for traversing diffrent aggregate structures (support polymorphic iteration)

Iterator design pattern structure



Participants:

Iterator: defines an interface for accessing and traversing elements

ConcreteIterator: implements the Iterator interface, keeps track of the current position in the traversals

Aggregate: defines an interface for creating Iterator object

ConcreteAggregate: implements the Iterator creation interface to return an instance of proper ConcreteIterator

Consequences:

- support variations in the traversal. Complex aggregates may be traversed in many ways
- simplify the aggregate interface.
- More than one traversal can be pending on an aggregate.

Iterators in STL

Iterator: an object that keeps track of a location whitin an associated STL container object, providing support for traversal (increment/decrement), dereferencing and container bounds detection.

Iterators are fundamental in many of the STL algorithms and are necessary tool for making good use of the STL container library.

Each STL container type includes member functions begin() and end() which effectively specify iterator values for the first and the “first - past - last” element.

```
void sampleIterator() {
	vector<int> v;
	v.push_back(4);
	v.push_back(8);
	v.push_back(12);
	//Obtain an the start of the iteration
	vector<int>::iterator it = v.begin();
	while (it != v.end()) {
		//dereference
		cout << (*it) << " ";
		//go to the next element
		it++;
	}
	cout << endl;
}
```

Iterators are the mechanism that makes it possible to decouple algorithms from containers

There are several kinds of iterators:

- input/output iterators (istream_iterator, ostream_iterator)
- forward iterators, bidirectional iterators, random access iterators
- reverse iterators

Iterator adaptors

Iterators can iterate over streams, either to read or to write

A stream like cin has the right functionality for an input iterator (provide access to a sequence of elements).

Cin has the wrong interface for an iterator => STL provide adaptors (adapter design pattern)

istream_iterator – useful adaptor that transform the istream interface to the iterator interface

```
//create a istream iterator using the standard input
istream_iterator<int> start(cin);
istream_iterator<int> end;

//the vector where the values are stored
vector<int> v;
back_insert_iterator<vector<int> > dest(v);

//copy elements from the standard input into the vector
copy(start, end, dest);
```

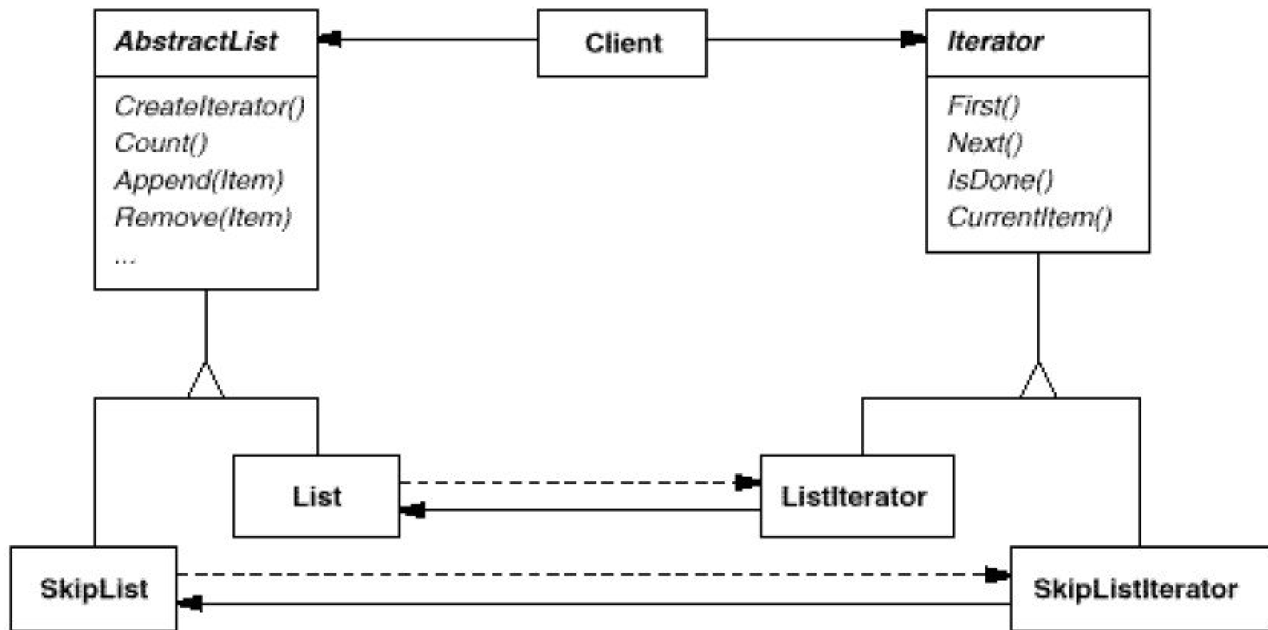
Iterator (Cursor) design pattern

Intent: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation

Motivation:

- An agregate (such as a list) should provide a way to access its elements without exposing its internal structure.
- Should allow to traverse the object in diffrent ways

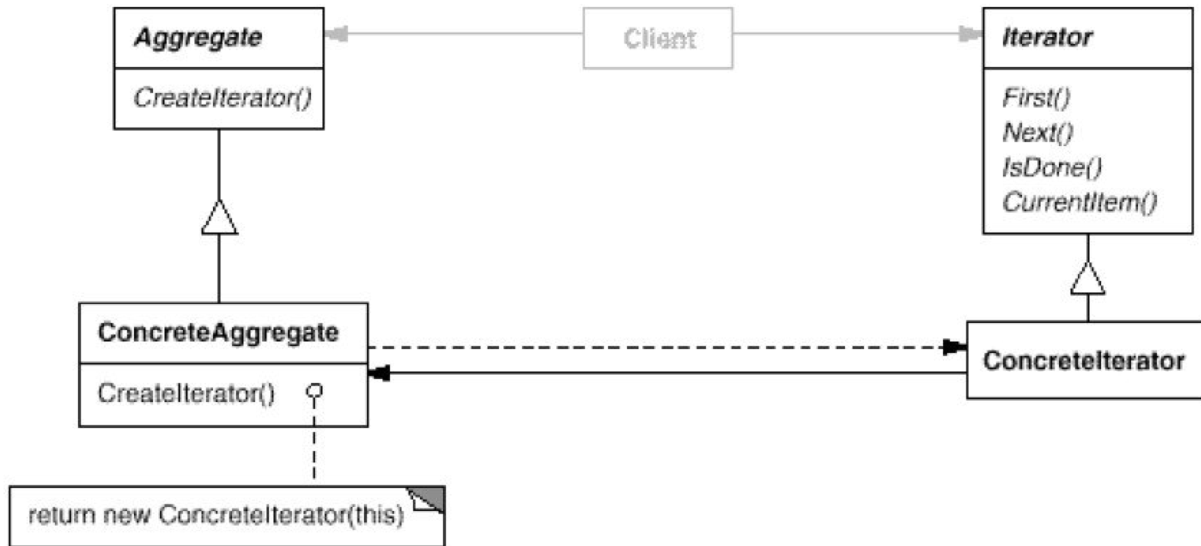
Example: List, SkipList, Iterator



Applicability:

- to access an agregate object's withput exposing its internal representation
- to support multiple traversals of aggregate objects
- to provide an uniform interface for traversing diffrent aggregate structures (support polymorphic iteration)

Iterator design pattern structure



Participants:

Iterator: defines an interface for accessing and traversing elements

ConcreteIterator: implements the Iterator interface, keeps track of the current position in the traversals

Aggregate: defines an interface for creating Iterator object

ConcreteAggregate: implements the Iterator creation interface to return an instance of proper ConcreteIterator

Consequences:

- support variations in the traversal. Complex aggregates may be traversed in many ways
- simplify the aggregate interface.
- More than one traversal can be pending on an aggregate.

Iterators in STL

Iterator: an object that keeps track of a location whitin an associated STL container object, providing support for traversal (increment/decrement), dereferencing and container bounds detection.

Iterators are fundamental in many of the STL algorithms and are necessary tool for making good use of the STL container library.

Each STL container type includes member functions begin() and end() which effectively specify iterator values for the first and the “first - past - last” element.

```
void sampleIterator() {
      vector<int> v;
      v.push_back(4);
      v.push_back(8);
      v.push_back(12);
      //Obtain an the start of the iteration
      vector<int>::iterator it = v.begin();
      while (it != v.end()) {
            //dereference
            cout << (*it) << " ";
            //go to the next element
            it++;
      }
      cout << endl;
}
```

Iterators are the mechanism that makes it possible to decouple algorithms from containers

There are several kinds of iterators:

- input/output iterators (istream_iterator, ostream_iterator)
- forward iterators, bidirectional iterators, random access iterators
- reverse iterators

Iterator types

Iterator Type	Behavioral Description	Operations Supported
random access (most powerful)	Store and retrieve values Move forward and backward Access values randomly	`* = ++ -> == != --` `+ - [] < > <= >= += -=`
bidirectional	Store and retrieve values Move forward and backward	`* = ++ -> == != --`
forward	Store and retrieve values Move forward only	`* = ++ -> == !=`
input	Retrieve but not store values Move forward only	`* = ++ -> == !=`
output (least powerful)	Store but not retrieve values Move forward only	`* = ++`

Iterator type provided by the STL containers

Container Class	Iterator Type	Container Category
vector	random access	sequential
deque	random access	
list	bidirectional	
set	bidirectional	associative
multiset	bidirectional	
map	bidirectional	
multimap	bidirectional	

– the container adaptors (stack, queue, priority_queue) do not provide iterator

STL Algorithms

- A collection of template functions that work on ranges defined by iterators.

- A range is any sequence of objects that can be accessed through iterators or pointers, such as an array or an instance of some of the STL containers.

STL algorithms header files: <algorithm> or <numeric>.

- algorithms from <numeric> tend to be based more on computational programming
- functions from <algorithm> tend to be more general-purpose.
- If you receive compiler errors about undefined functions, make sure you've included both these headers.

Functions in STL:

- Non-modifying sequence operations: **accumulate**, **count**, **find**, **count_if**, etc
- Modifying sequence operations: **copy**, **transform**, **swap**, **reverse**, **random_shuffle**, etc
- Sorting: **sort**, **stable_stort**, etc
- Binary search (operating on sorted ranges): **binary_search**, etc
- Merge (operating on sorted ranges): **merge, set_union, set_intersect,** etc
- Min/max: **min**, **max**, **min_element**, etc
- Heap: **make_heap, sort_heap**, etc

STL Algorithms - accumulate

```
    vector<int> v;
    v.push_back(3);
    v.push_back(4);
    v.push_back(2);
    v.push_back(7);
    v.push_back(17);
   //compute the sum of all elements in the vector
    cout << accumulate(v.begin(), v.end(), 0) << endl;

    //compute the sum of elements from 1 inclusive, 4 exclusive [1,4)
    vector<int>::iterator start = v.begin()+1;
    vector<int>::iterator end = v.begin()+4;
    cout << accumulate(start, end, 0) << endl;
```

– accumulate : compute the sum of elements in a range

– STL algorithms – performing huge tasks in a tiny space

```
/**
 *  @brief  Accumulate values in a range.
 *
 *  Accumulates the values in the range [first,last) using operator+().  The
 *  initial value is @a init.  The values are processed in order.
 *
 *  @param  first  Start of range.
 *  @param  last  End of range.
 *  @param  init  Starting value to add other values to.
 *  @return  The final sum.
 */
template<typename _InputIterator, typename _Tp>
  accumulate(_InputIterator __first, _InputIterator __last, _Tp __init)
```

STL Algorithms – copy

Copy a range of elements

```
template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last, OutputIterator result)
{
  while (first!=last) *result++ = *first++;
  return result;
}
```

Parameters:

- first, last - Input iterators to the initial and final positions in a sequence to be copied. The range used is [first,last), which contains all the elements between first and last, including the element pointed by first but not the element pointed by last.
- Result - Output iterator to the initial position in the destination sequence. This shall not point to any element in the range [first,last).

```
vector<int> v;
v.push_back(3);
v.push_back(4);
v.push_back(2);
v.push_back(7);
v.push_back(17);

//make sure there are enough space in the destination
//allocate space for 5 elements
vector<int> v2(5);

//copy all from v to v2
copy(v.begin(), v.end(), v2.begin());
```

STL Algorithms – sort

- Sort elements in range: Sorts the elements in the range [first,last) into ascending order.
- The elements are compared using **operator <**
- Elements that would compare equal to each other are not guaranteed to keep their original relative order.

Parameters:

- first, last : The range used is[first,last), which contains all the elements between first and last, including the element pointed by firstbut not the element pointed by last.
- Random-Access iterators to the initial and final positions of the sequence to be sorted.
- Comp: Comparison function object that, taking two values of the same type than those contained in the range, returns true if the first argument goes before the second argument in the specific strict weak ordering it defines, and false otherwise.

```
	vector<int> v;
	v.push_back(3);
	v.push_back(4);
	v.push_back(2);
	v.push_back(7);
	v.push_back(17);

	//sort
	sort(v.begin(), v.end());
```

Provide a comparison function:

```
bool asc(int i, int j) {
	return (i < j);
}
```

```
bool desc(int i, int j) {
	return (i > j);
}
```

```
void testSortCompare() {
	vector<int> v;
	v.push_back(3);
	v.push_back(4);
	v.push_back(2);
	v.push_back(7);
	v.push_back(17);

	//sort
	sort(v.begin(), v.end(), asc);

	//print elements
	for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
		cout << *it << "  ";
	}
	cout << endl;
}
```

STL Algorithms – for_each, transform

Applies function to each of the elements in the range [first,last).

```
void testForEach() {
	vector<int> v;
	v.push_back(3);
	v.push_back(4);
	v.push_back(2);
	v.push_back(7);
	v.push_back(17);

	for_each(v.begin(), v.end(), print);
	cout << endl;
}
```

```
void print(int elem) {
	cout << elem << " ";
}
```

Transform:applies a function to all the elements in the input range ([first1,last1)) and stores each returned value in the range beginning at result.

```
int multiply3(int el) {
	return 3 * el;
}

void testTransform() {
	vector<int> v;
	v.push_back(3);
	v.push_back(4);
	v.push_back(2);
	v.push_back(7);
	v.push_back(17);

	vector<int> v2(5);
	transform(v.begin(), v.end(), v2.begin(), multiply3);

	//print the elements
	for_each(v2.begin(), v2.end(), print);
}
```

STL Function Objects (Functors)

- Classes which have an operator ()can be used as functions
- More powerful than plain functions, because they can store additional data

A function object (or functor) is simply any object of a class that provides a definition for operator()

if you declare an object **f** of the class in which this **operator()** is defined you can subsequently use that object **f** just like you would use an "ordinary" function.

For example, you could have an assignment statement like
someValue = f(arg1, arg2);
which is the same as
someValue = f.operator()(arg1, arg2);

```
class MyClass {
public:
	bool operator()(int i, int j) {
		return (i > j);
	}
};

sort(v.begin(), v.end(), MyClass());
```

Iterator adaptor for insertion (insert iterator, inserters)

- permit algorithms to operate in insert mode rather than overwrite mode (default)
- solve the problem that arise when an algorithm tries to write elements to a destination container not already big enough to hold them, by making the destination grow as needed.

kind of inserters:

- The back_inserter(), which can be used if the recipient container supports the push_back() member function.
- The front_inserter(), which can be used if the recipient container supports the push_front() member function.
- The inserter(), which can be used if the recipient container supports the insert() member function.

```
deque<int> v;
v.push_back(4);
v.push_back(8);
v.push_back(12);

deque<int> v2;
//    back_insert_iterator<deque<int> > dest(v2);
front_insert_iterator<deque<int> > dest(v2);

//copy elements from v to v2
copy(v.begin(), v.end(), dest);
```

Input/Output Iterator adaptors

Iterators can iterate over streams, either to read or to write

A stream like cin has the right functionality for an input iterator (provide access to a sequence of elements).

Cin has the wrong interface for an iterator => STL provide adaptors (adapter design pattern)

istream_iterator – useful adaptor that transform the istream interface to the iterator interface

```
//create a istream iterator using the standard input
istream_iterator<int> start(cin);
istream_iterator<int> end;

//the vector where the values are stored
vector<int> v;
back_insert_iterator<vector<int> > dest(v);

//copy elements from the standard input into the vector
copy(start, end, dest);
```

Simple sort application

```
const int size = 1000; // array of 1000 integers
int array[size];

cout << "Elements (Nr elements <1000):";

int elem;
int n = 0;
while (cin >> elem) {
        array[n++] = elem;
}

//sort the array
qsort(array, n, sizeof(int), cmpInt);

cout << endl;
//print the array
for (int i = 0; i < n; i++)
        cout << array[i] << " ";
cout << endl;
```

```
ifstream inFile("in.txt");
//create a istream iterator using the file
istream_iterator<int> start(inFile);
istream_iterator<int> end;

//the vector where the values are stored
vector<int> v;
back_insert_iterator<vector<int> > dest(v);

//copy from the standard input into the vector
copy(start, end, dest);
inFile.close();

sort(v.begin(), v.end());

copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));
cout << endl;

//copy to file
ofstream outFile("out.txt");
copy(v.begin(), v.end(), ostream_iterator<int>(outFile, " "));
outFile.close();
```

Why STL Algorithms

- simplicity: use existing code instead of writing the code from scratch
- correctness: know to be corect, tested
- performance: generally perform better than a hand writed code.
  - Use sophisticated algorithms (performs better than any of the code writen by an average c programmer)
  - Use advanced techniques like template specialization and template metaprogramming, STL algorithms are transparently optimized to work as fast as possible.
  - The functions may exploit some of the internal details of the underlying containers.

- clarity: you can immediately tell that a call to accumulate adds up numbers in a range (with a for loop that sums up values, you have to read each line of code to understand)
- Maintainability:
  - Algorithm calls often yield code that is clearer and more straightforward than the corresponding explicit loops.
  - Copy paste is avoided.
  - when can replace low-level words like **for**, **while**, and **do** with higher-level terms like **insert**, **find**, and **for_each**, we raise the level of abstraction in our software and thereby make it easier to write, document, enhance, and maintain.

String class

String objects are a special type of container, specifically designed to operate with sequences of characters.

Unlike traditional c-strings, which are mere sequences of characters in a memory array, C++ string objects belong to a class with many built-in features to operate with strings in a more intuitive way and with some additional useful features common to C++ containers.

Member functions

Iterators:
begin, end - Return iterator to beginning/end

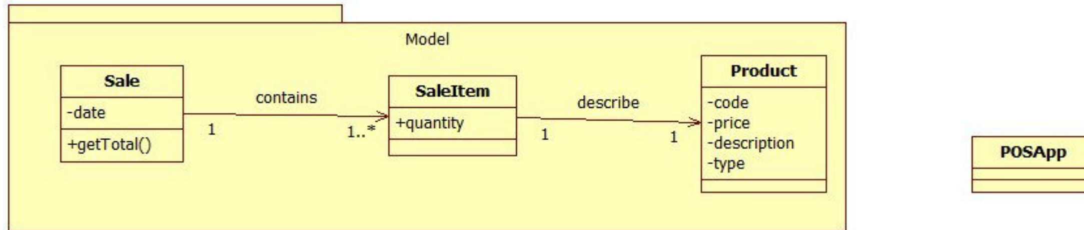
Capacity:
size - Return length of string
clear - Clear string
empty - Test if string is empty

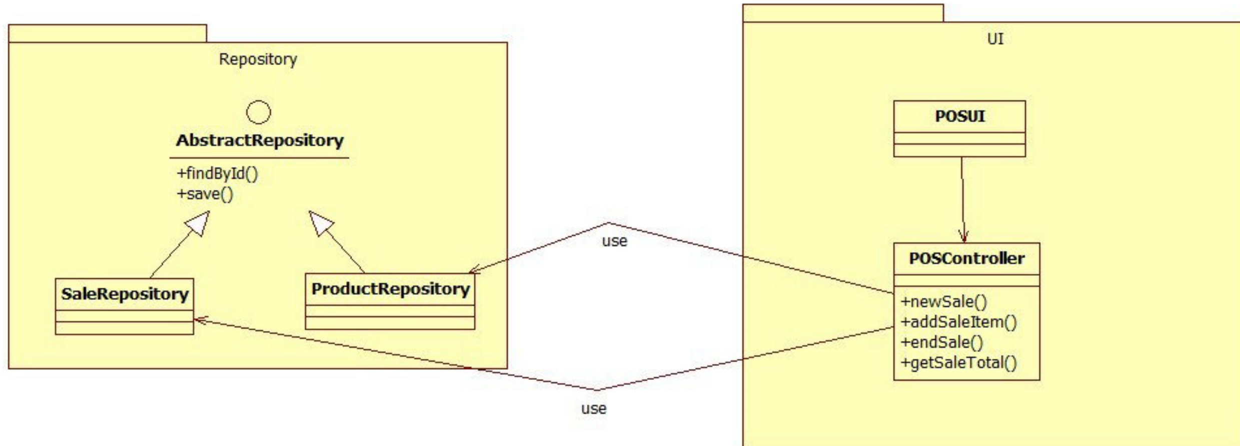
Element access:
operator[] - Get character in string
at - Get character in string

Modifiers:
operator+= - Append to string
append - Append to string
push_back - Append character to string
insert - Insert into string
erase - Erase characters from string
replace - Replace part of string

String operations:
find - Find content in string
substr - Generate substring
compare - Compare strings

POS application






```
/**
 * Compute the total price for this sale
 * return the total for the items in the sale
 */
double Sale::getTotal() {
	double total = 0;
	for (int i = 0; i < items.size(); i++) {
		SaleItem sIt = items[i];
		double price = sIt.getQuantity() * sIt.getProduct().getPrice();
		total += price;
	}
	return total;
}
void testSale() {
	Sale s;
	assert(s.getTotal()==0);

	Product p1(1, "Apple", "food", 2.0);
	s.addItem(3, p1);
	assert(s.getTotal()==6);

	Product p2(1, "TV", "electronics", 2000.0);
	s.addItem(1, p2);
	assert(s.getTotal()==2006);
}
```

POS – application

Requirements

- Give a 2% discount if the payment is made using a credit card
- If the client buy 3 ore more item from the same product there is a 10% discount
- On Monday there is a discount of 5% for food
- Frequent buyer discount
- ...

```
/**
 * Compute the total price for this sale
 * isCard true if the payment is by credit card
 * return the total for the items in the sale
 */
double Sale::getTotal(bool isCard) {
	double total = 0;
	for (int i = 0; i < items.size(); i++) {
		SaleItem sIt = items[i];
		double pPrice;
		if (isCard) {
			//2% discount
			pPrice = sIt.getProduct().getPrice();
			pPrice = pPrice - pPrice * 0.02;
		} else {
			pPrice = sIt.getProduct().getPrice();
		}
		double price = sIt.getQuantity() * pPrice;
		total += price;
	}
	return total;
}
void testSale() {
	Sale s;
	assert(s.getTotal(false)==0);

	Product p1(1, "Apple", "food", 2.0);
	s.addItem(3, p1);

	assert(s.getTotal(false)==6);

	Product p2(1, "TV", "electronics", 2000.0);
	s.addItem(1, p2);

	assert(s.getTotal(false)==2006);

	//total with discount for cars
	assert(s.getTotal(true)==1965.88);
}
```

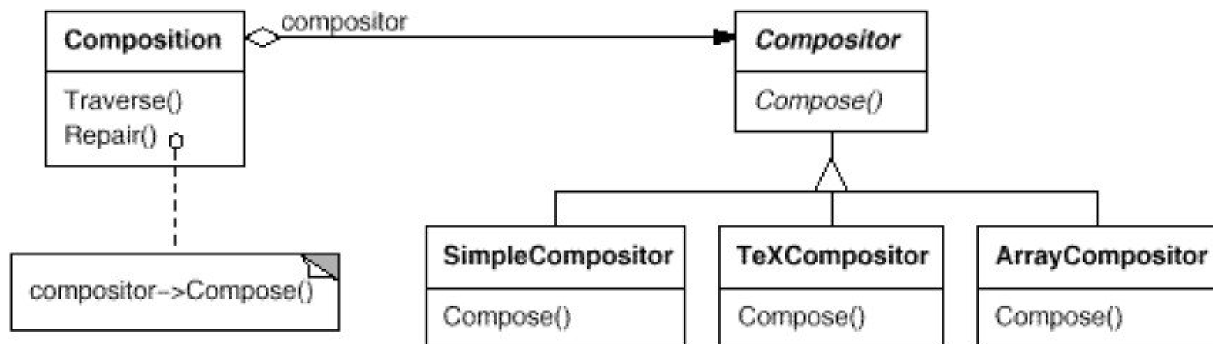
This approach will lead to complicated total calculation (hard to maintain, extend, understand)

Strategy (policy) – design pattern

Intent: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Motivation:

In a document editor application, suppose a Composition class is responsible for maintaining and updating the line-breaks of text displayed in a text viewer. Many algorithms exist for breaking a stream of text into lines.
Different algorithms will be appropriate at different times.



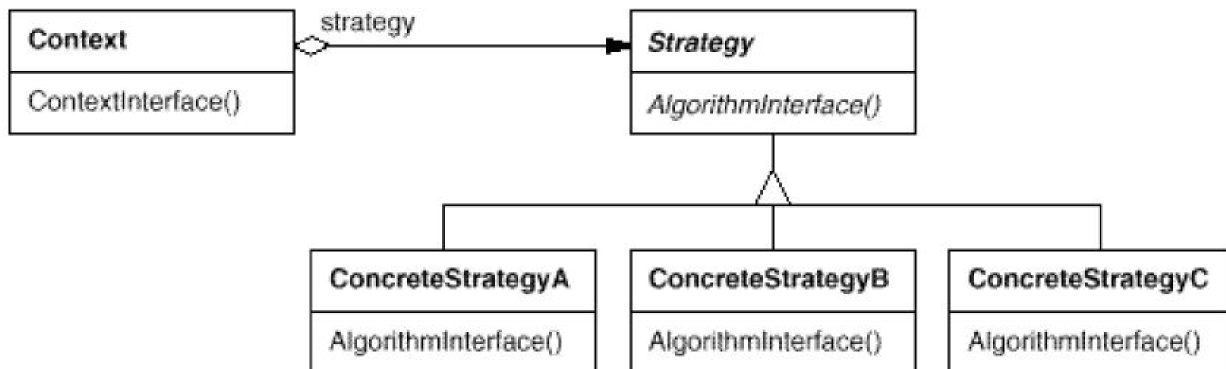
Linebreaking strategies are implemented separately by subclasses of the abstract Compositor class (not the **Composition** class) .
Compositor subclasses implement different strategies:

- **SimpleCompositorimplements** a simple strategy that determines linebreaks one at atime.
- **TeXCompositor** implements the TeX algorithm for finding linebreaks. This strategytries to optimize linebreaks globally, that is, one paragraph at a time.
- **ArrayCompositor** implements a strategy that selects breaks so that each row has a fixednumber of items. It's useful for breaking a collection of icons intorows, for example.

Strategy (Policy)

Applicability:

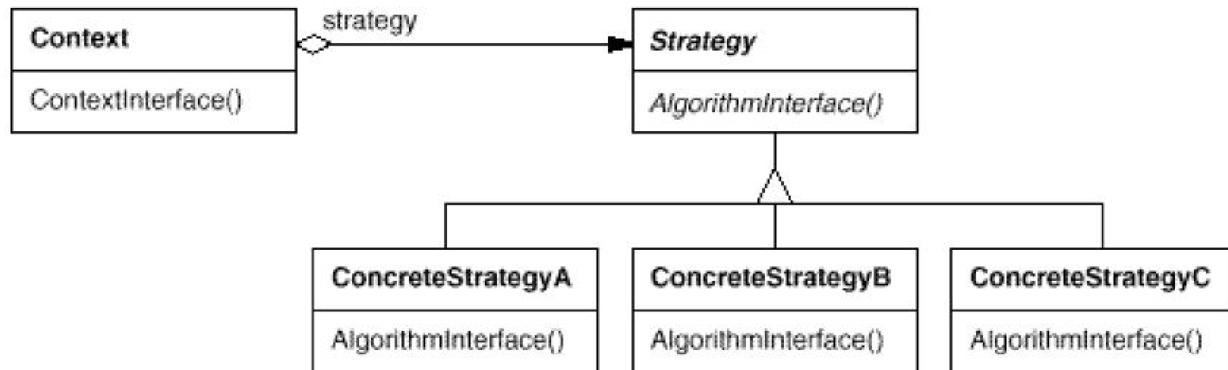
- many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
- you need different variants of an algorithm. For example, you might definealgorithms reflecting different space/time trade-offs. Strategies can be used when these variants are implemented as a classhierarchy of algorithms.
- an algorithm uses data that clients shouldn't know about. Use theStrategy pattern to avoid exposing complex, algorithm-specific datastructures.
- a class defines many behaviors, and these appear as multipleconditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.



Participants:

- **Strategy** (Compositor): declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.
- **ConcreteStrategy** (SimpleCompositor, TeXCompositor,ArrayCompositor) implements the algorithm using the Strategy interface.
- **Context** (Composition)
  - is configured with a ConcreteStrategy object
  - maintains a reference to a Strategy object.
  - may define an interface that lets Strategy access its data.

Strategy



Collaborations:

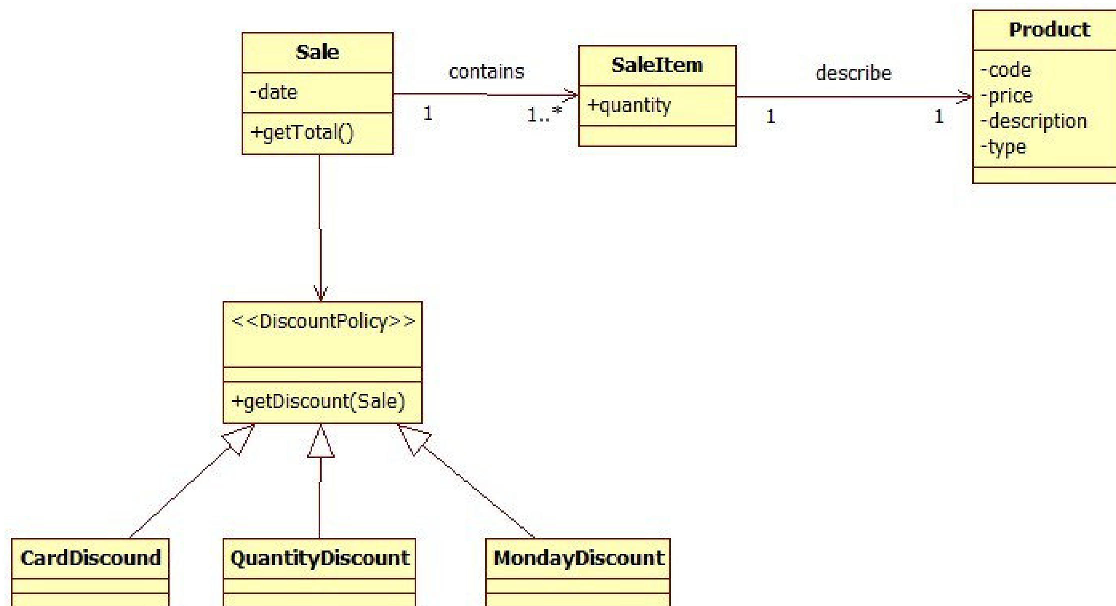
- Strategy and Context interact to implement the chosen algorithm. A context may pass all data required by the algorithm to the strategy when the algorithm is called. Alternatively, the context can pass itself as an argument to Strategy operations. That lets the strategy call back on the context as required.
- A context forwards requests from its clients to its strategy. Client susually create and pass a ConcreteStrategy object to the context;
- Clients interact with the context exclusively. There is often a family of ConcreteStrategy classes for a client to choose from.

Consequences:

- Families of related algorithms. Hierarchies of Strategy classes define a family of algorithms or behaviors for contexts to reuse. Inheritance can help factor out common functionality of the algorithms.
- An alternative to subclassing. Inheritance offers another way to support a variety of algorithms or behaviors
- Strategies eliminate conditional statements. The Strategy pattern offers an alternative to conditional statements forselecting desired behavior
- Clients must be aware of different Strategies
- Communication overhead between Strategy and Context
- Increased number of objects. Strategies increase the number of objects in an application.

Discount Policy for POS

Factor the varying part (discount) of a process (computing the total) into a separate "strategy" object in the model.
Factor apart a rule and the behavior it governs.
Implement the rule or substitutable process following the STRATEGY design pattern.
Multiple versions of the strategy object represent different ways the process can be done.



Controll the behavior of getTotal using diffrent **DiscountPolicy** objects

Easy to add new discount schemas.

The logic behind the discount is isolated (Protected variation)

The Protected Variations GRASP pattern: protect elements from the variations on other elements (objects, systems, subsystems) by wrapping the focus of instability with an interface and using polymorphism to create various implementations of this interface.

Discount Policy for POS

```
class DiscountPolicy {
public:
	/**
	 * Compute the discount for the sale item
	 * s - the sale, some discount may based on all the products in te sale, or other attributes of the sale
	 * si - the discount amount is computed for this sale item
	 * return the discount amount
	 */
	virtual double getDiscount(const Sale* s, SaleItem si)=0;
};
/**
 * Apply 2% discount
 */
class CreditCardDiscount: public DiscountPolicy {
public:
	virtual double getDiscount(const Sale* s, SaleItem si) {
		return si.getQuantity() * si.getProduct().getPrice() * 0.02;
	}
};
```

```
/**
 * Compute the total price for this sale
 * return the total for the items in the sale
 */
double Sale::getTotal() {
	double total = 0;
	for (int i = 0; i < items.size(); i++) {
		SaleItem sIt = items[i];
		double price = sIt.getQuantity() * sIt.getProduct().getPrice();
		//apply discount
		price -= discountPolicy->getDiscount(this, sIt);
		total += price;
	}
	return total;
}
```

```
void testSale() {
	Sale s(new NoDiscount());
	Product p1(1, "Apple", "food", 2.0);
	Product p2(1, "TV", "electronics", 2000.0);
	s.addItem(3, p1);
	s.addItem(1, p2);
	assert(s.getTotal()==2006);

	Sale s2(new CreditCardDiscount());
	s2.addItem(3, p1);
	s2.addItem(1, p2);
	//total with discount for card
	assert(s2.getTotal()==1965.88);
}
```

How to combine multiple discounts?

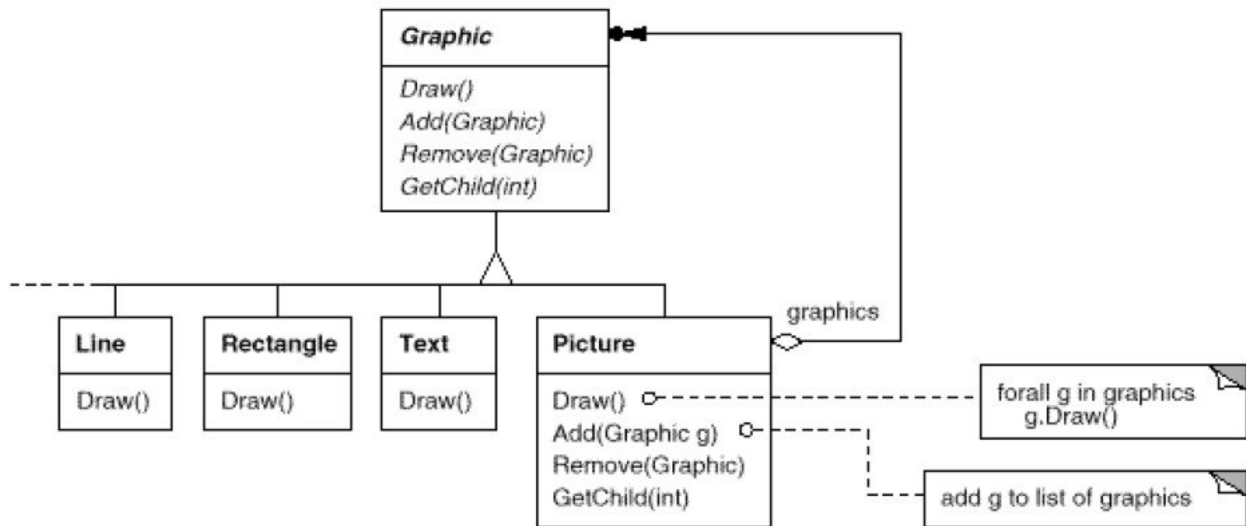
Composite design pattern

Motivation:

Compose objects into tree structures to represent part-whole hierarchies.
Composite lets clients treat individual objects and compositions of objects uniformly.

Graphic application for building complex diagrams out of simple components.
The user can group components to form larger components, which in turn can be grouped to form still larger components.
A simple implementation could define classes for graphical primitives such as Text and Lines plus other classes that act as containers for these primitives

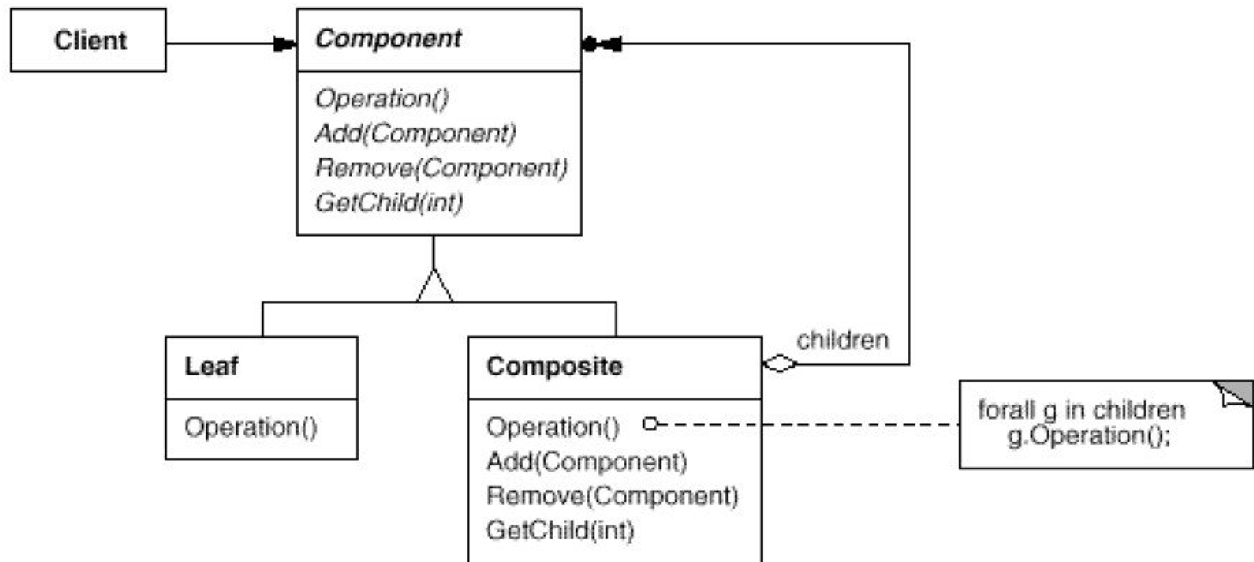
Code must treat primitive and container objects differently this makes the application more complex.



Applicability:

- you want to represent part-whole hierarchies of objects.
- you want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

Composite design pattern



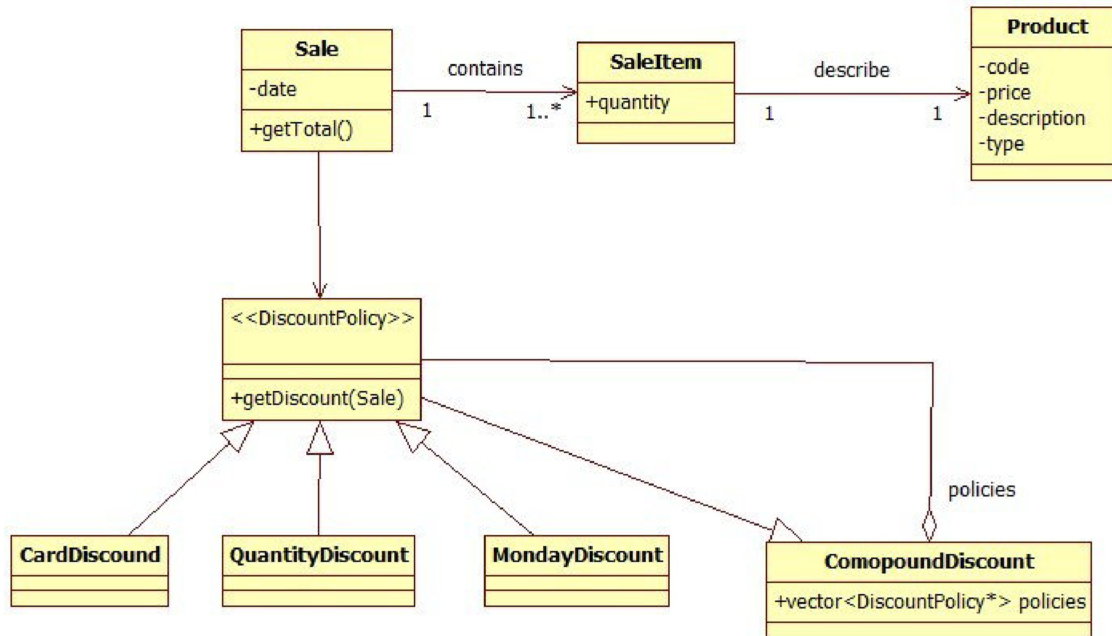
Participants:

- Component (Graphic)
  - declares the interface for objects in the composition.
  - implements default behavior for the interface common to all classes, as appropriate.
  - declares an interface for accessing and managing its child components.
  - (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.
- · Leaf (Rectangle, Line, Text, etc.)
  - represents leaf objects in the composition. A leaf has no children.
  - defines behavior for primitive objects in the composition.
- Composite (Picture)

Consequences

- Primitive objects can be composed into more complex objects, which in turn can be composed, and so on recursively. Wherever client code expects a primitive object, it can also take a composite object.
- makes the client simple. Clients can treat composite structures and individual objects uniformly.
- makes it easier to add new kinds of components. Clients don't have to be changed for new Component classes.
- can make your design overly general. it makes it harder to restrict the components of a composite.

POS – Combine discount policies




```
/**
 * Combine multiple discount types
 * The discounts will sum up
 */
class CompoundDiscount: public DiscountPolicy {
public:
	virtual double getDiscount(const Sale* s, SaleItem si);

	void addPolicy(DiscountPolicy* p) {
		policies.push_back(p);
	}
private:
	vector<DiscountPolicy*> policies;
};
```

```
/**
 * Compute the sum of all discounts
 */
double CompoundDiscount::getDiscount(const Sale* s, SaleItem si) {
	double discount = 0;
	for (int i = 0; i < policies.size(); i++) {
		discount += policies[i]->getDiscount(s, si);
	}
	return discount;
}
```

POS – Combine discount policies

```
Sale s(new NoDiscount());
Product p1(1, "Apple", "food", 10.0);
Product p2(2, "TV", "electronics", 2000.0);
s.addItem(3, p1);
s.addItem(1, p2);
assert(s.getTotal()==2030);

CompoundDiscount* cD = new CompoundDiscount();
cD->addPolicy(new CreditCardDiscount());
cD->addPolicy(new QuantityDiscount());

Sale s2(cD);
s2.addItem(3, p1);
s2.addItem(4, p2);
//total with discount for card
assert(s2.getTotal()==7066.4);
```

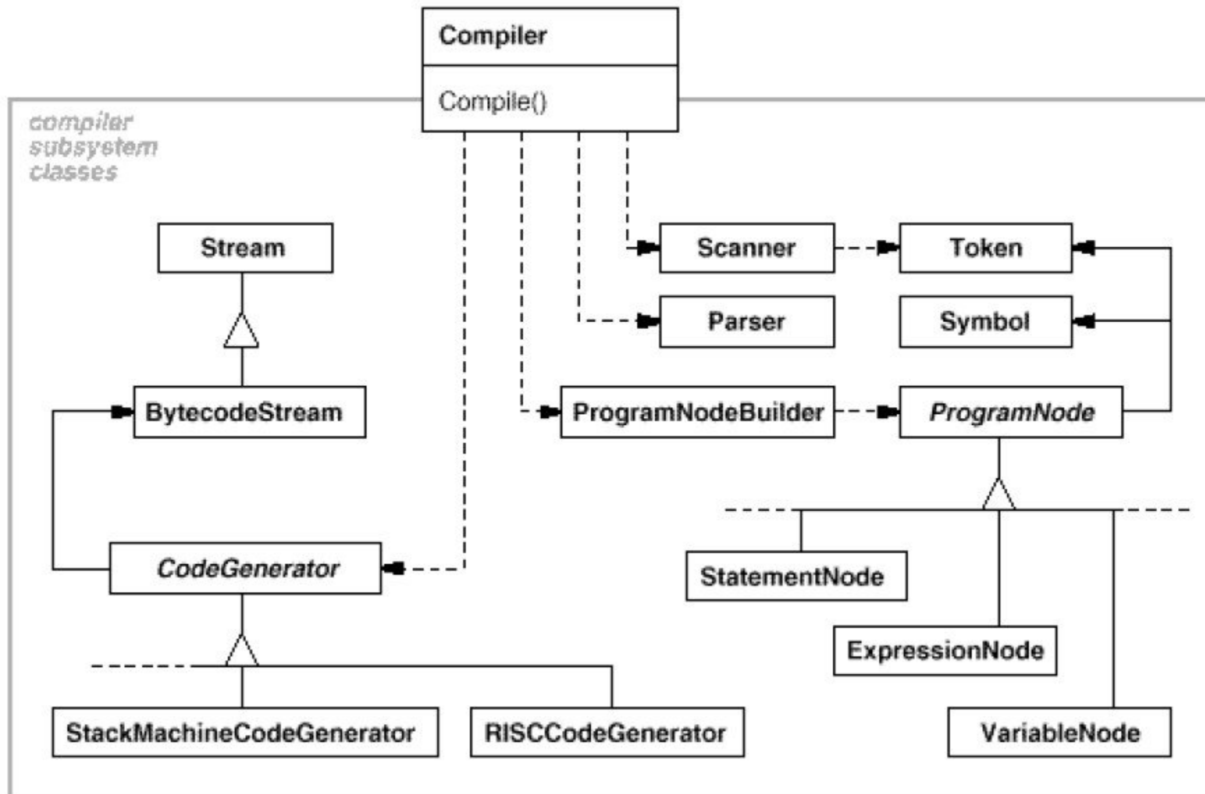
How can we express requirements like:

The “Frequent buyer discount” and the Monday Food discount can not be combined, only one of the discount will be aplied (the largest discount)

Facade

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

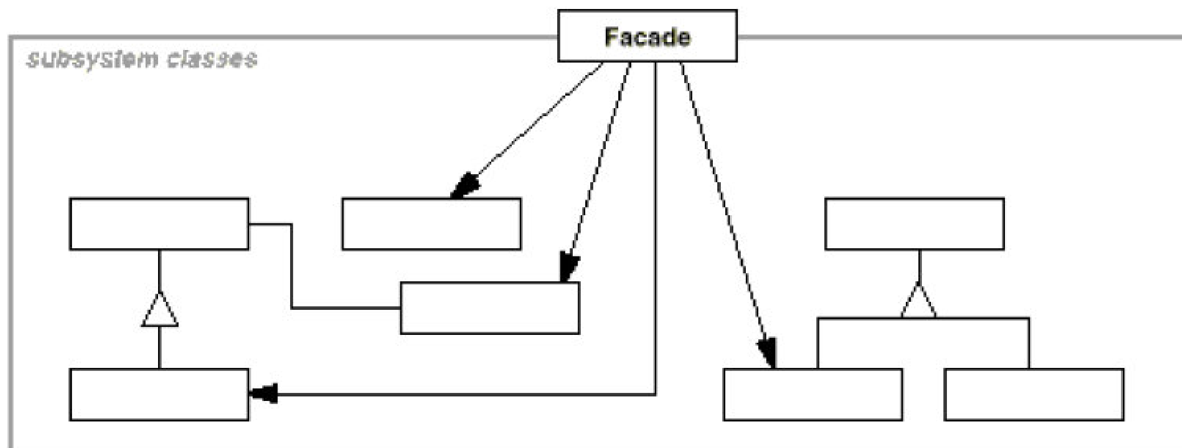
Motivation:
Structuring a system into subsystems helps reduce complexity



Applicability

- you want to provide a simple interface to a complex subsystem. Subsystems often get more complex as they evolve. Most patterns, when applied, result in more and smaller classes. A facade can provide a simple default view of the subsystem that is good enough for most clients. Only clients needing more customizability will need to look beyond the facade.
- there are many dependencies between clients and the implementation classes of an abstraction. Introduce a facade to decouple the subsystem from clients and other subsystems, thereby promoting subsystem independence and portability.
- you want to layer your subsystems.

Facade



Participants

- Facade (Compiler)
  - knows which subsystem classes are responsible for a request.
  - delegates client requests to appropriate subsystem objects.
- subsystem classes (Scanner, Parser, ProgramNode, etc.)
  - implement subsystem functionality.
  - handle work assigned by the Facade object.
  - have no knowledge of the facade; that is, they keep no references to it.

Consequences

- It shields clients from subsystem components, thereby reducing the number of objects that clients deal with and making the subsystem easier to use.
- It promotes weak coupling between the subsystem and its clients.
- It doesn't prevent applications from using subsystem classes if they need to. Thus you can choose between ease of use and generality.