Overview on Today's Lecture

- Last Lecture
- Schedule
- Type Conversion
- Operators
- Friends
- Inheritance
- Templates

Last Lecture

- Constructor, Descturctor for building safe dynamic objects
- Assignment vs. Copying
- Operators

Schedule Options

Two possible options:

- Another Lecture: 17/01/2000 same location same time and two small exercises
- Hemtentamen: announced this week and to be submitted latest 23/12/2000

Type Conversion

There are two kind of type conversion:

- using constructors for conversion
- using conversion operators

Example of Constructor Conversion

```
Class X{
 public:
  X(int);
  X(double);
  X operator+(const X&);
}
main()
{
  X a,b;
  b = a + 2;
  b = a +1.0;
}
```


Example for Operator Conversion

```
class Z7{
  char v;
  void assign(int i);
public:
  Z7(int i=0) {assign(i);}
  Z7& operator=(int i){assign(i); return *this;}
  operator int() const {return v;}
};
void Z7::assign(int i)
{
  int z=i%7;
  if (z<0) z+=7;
  v = z;
}
```



```
int main(){
  Z7 a,b,c;

  a = -3;                              // a=4
  b = 5;                               // b=5

  c= a*b;                              // result c=6

  cout << a << ' ' << b << ' ' << c << '\n';
  return 0;
}
```


Remarks

- The first conversion method transform already defined types to an object of the kind needed for an operation.
- The second converts an object to a basic type or to a previously defined type.

Operator<< for Output

Some objects like algebraic objects have also a representation which we would like to output.

For example, after performing computation with a class Vector we would like to see the result of the computation.

We can do this by using the class ostream. For this class << means that something is written to the output.

cout is an ostream which writes to the standard output.

Similarly there exists the class *istream* which reads from the input. In this case *cin* is an istream which reads from the standard input.

For defining the behaviour of an *istream* you have to overload operator>>.

Example

```
ostream& operator<<(ostream& out, const Vector& v)
{
  for(int i=0; i<v.dim(); i++)
    out << v.member(i) << '' '';
}

main()
{
  Vector v, w;
  cout << v+w << ''\n'';                    // cout is a ostream
}
```


Subscripting Operator []

With the operator[] we can define subscripts for objects.

For instance, we can write instead of $vec.member(i)$ $vec[i]$ for a variable of type vector.

We also can design assignments of the form $vec[i] = 5$.

Note that the subscript argument can be of any type.

Example

```
int& Vector::operator[](const int& pos)
{
  if ((pos>=0) && (pos<size)){
    return elem[pos];
  }
}

main(){
  Vector x(10);
  for (int i=0; i<10; i++)
    x[i]=0;
}
```

Friends

Motivation: Sometimes it is necessary for functions which have to be declared outside the scope the class to access the private part. Declaring these functions as *friend* gives this feature.

For ordinary member function following three properties are valid:

1. the function can access the private part of the class declaration
2. the function is in the scope of the class
3. the function must be invoked on an object

Declaring a function as *static* means that only the first two properties hold.

Declaring a function as *friend* means that only the first property holds.

Example

```
class Matrix;

class Vector{
  //...
  friend Vector operator*(const Matrix&, const Vector&);
};
class Matrix{
  //...
  friend Vector operator*(const Matrix&, const Vector&);
};
Vector operator*(const Matrix& m, const Verctor& v)
{
  // can access the inner reprsentation of both objects
}
```


Remarks

- It does not matter whether a *friend* declaration is private or public
- A member function of one class can be friend of some other class.
- You can also declare a whole class as a friend of another.
- In general friends should only be used to describe closely related objects.
- A friend class must be previously declared in an enclosing scope or defined in the non-class-scope immediately after the class

Examples

```
class X{
  // ...
  int do_something();
};

class Y{
  friend int X::do_something();
  // ...
};

class Z{
  friend class X;
  // ...
};
```

```
class A {...};                          // friend of D

namespace N{
  class D{
    friend class A;
    friend class B;
    friend class C;
  }
  class B{...};                         // friend of D
}
class C{...};                           // not a friend of D
```


Large Objects

Using operators for large objects can lead to a large overhead each time the copy constructor is called.

In order to solve this problem we can define operators by using constant references to its parameters, i.e. the operator uses a reference to the parameter but is not allow to change the parameter.

Note that pointers cannot be used for defining operator functions since you cannot redefine the meaning of an operator applied to a pointer.

Example

```
class Vector
{
  int *elem;
  int size;
  friend Vector operator-(const Vector&, const Vector&);
};
```


Some Advice for Operator Overloading

- Operators should be used as they are intentionally defined.
- For dealing with big objects it is recommended to use constant references for parameters.
- If possible then use the default copy operations.
- If the default copy operations are not appropriate you should either redefine or prohibit them.
- Member functions are preferable to non-members for operations that need access to the default representation.
- Non member functions are preferable for operations that need not to access the default representation.

Inheritance

Before we learnt how to represent concepts as classes and how to create safe objects.

Now we will have a basic concept and we will built new classes which use our basic concept.

The class representing the basic concept is called *base class* while the new class derived from our basic class is called *inheritance*.

Similar to the circle-shape example (lecture 1) we could construct a relation between an employee and a manager, i.e. a manager can be described as an inherited concept of an employee.

Two approaches for the manager concept

```
struct Employee {
  string first_name, family_name;
  char middle_initial;
  short department;
  // ...
}
```

```
struct Manager {
  Employee emp;
  set<Employee*> group;
  short level;
};
```

```
struct Manager : public Employee {
  set<Employee*> group;
  short level;
   // ...
};
```


Due to the second approach the compiler knows that it can deal the class Manager as an employee.

This can be helpful when we would like to use the Manager class in data structures which where designed for the base class.

For example we can store the manger in a list of pointers on employees:

```
void f(Manager man, Employee emp)
{
  list<Employee*> elist;
  elist.append(&man);
  elist.append(&emp);
}
```

Note that this will not work vice versa.

Member Functions

Members which are declared within a derived class can access all *public* members of its base class.

Note that they do not have access to the private part of the base class.

Why? If this was not the case we could easily destroy the concept of private and public members by deriving for any class a new class and then accessing the private part.

What to do if we would like to have a member only accessible for derived classes?

We can declare the member as *protected* in the base class.

Constructors

Some of the derived classes will need constructors.

The default constructor of a base class will be automatically invoked.

If a constructor with arguments should be used, then this constructor must be explicitly called.

Example

```
class Employee{
  string first_name, family_name;
  short department;
  // ...
public:
  Employee(const string& n, int d);
  // ...
}

class Manager : public Employee {
  set<Employee*> group;
  // ...
}
```


Example

```
class Employee{
  string first_name, surname; // cannot be accessed by Manager
  // ...
protected:
  int special                 // can only be accessed by derived
                              //  classes
public:
  void print();               // can be accesed by any class
  string full_name();
  // ...
}
```


Example II

```
class Manager : public Employee {
  // ...
public:
  void print(){
    Employee::print();
    // ...
  }
  // ...
};
```

```
Employee::Employee(const string& n, int d)
  : family_name(n), department(d)
{
  // ...
}

Manager::Manager(const string& n, int d, int lvl)
  : Employee(n,d), level(lvl)
{
  // ...
}
```


Virtual Functions

Assume we have a pointer to a base class e.g. to our example class employee. Then, we might like to know to which derived class this pointer belongs. For example we would like to output dependent on our derived class a different message.

For these kind of problems C++ provides virtual functions, for which the compiler guarantees that for each such function the right representation is chosen.

A function which is declared virtual in a base class can be overridden by a declaration of a derived class.

Example

```
class Employee{
  string first_name, family_name; short department;
public:
  Employee(const string& n, int d);
  virtual void print() const;
  // ...
}
class Manager : public Employee {
  set<Employee*> group;
public:
  void print() const;
  // ...
}
```



```
void Employee::print() const
{
  cout << family_name << ' ' << department << "\n";
  // ...
}

void Manager::print() const
{
  Employee::print();
  cout << "More information \n";
}

void f(Employee *emp)
{
  (*emp)->print();
}
```

Abstract Classes

Some classes just make sense for deriving an inheritance out of it, but the object itself is not useful.

If this is the case we can built the class out of *pure virtual functions*. These are virtual functions which are initialised by 0.

For example:

```
void draw() = 0
```

in class shape.

It is not possible to create an object out of an abstract class.