Inheritance

Inheritance allows us to use a previous class as a model for a new class. All functionality in the original class will be kept (without additional code), and we are allowed to add new functionality.

The class we use as a model is called the "base class" and the new class we create from this is called "derived class" or "subclass".
Inheritance can be done in many levels. One class may be derived from some class, and at the same time base class to an other class.

Inheritance syntax

The following syntax is used to create a subclass:

```
class <sub-class> : public <base-class>
{
   <sub-class>(<parameter-list>);
   <member-declarations>
};
```

It is common for the constructor of the derived class to call the constructor of the base class. This must be done with an initialization list:

```
<sub-class>::<sub-class>(<param-list>)
            : <base-class>(<argument-list>),
              <member-name>(<argument>)
{
   <constructor-code>
}
```

Initiating member variables with initialization list replace the corresponding assignment i the constructor.

Public inheritance

This rules apply for the normal public inheritance:
- **private members of the base class will not be accessible in the subclass, nor to anyone else**
- **protected members in the base class become protected also in the subclass, and behave as private to anyone else**
- **public members in the base class will be public in the derived class**

Inheritance exist in other flavours. The only difference is what part of the base class become available in the subclass. The table outline what protection a member receive with various inheritance mode:

```
baseclass   inheritance    subclass
-----------------------------------------
public    + public      => public
protected + public      => protected
private   + public      => not accessible

public    + protected   => protected
protected + protected   => protected
private   + protected   => not accessible

public    + private     => private
protected + private     => private
private   + private     => not accessible
```

We will only use public inheritance in the course, outlined in italic.

Inheritance example (classes)

```
class person
{
public:
   person(string const& n) : name(n) {}
   string get_name() { return name; }
protected:
   string name;
};

class parent : public person
{
public:
   parent(string const& n, int c)
         : person(n), children(c) {}

   int get_children() { return children; }
private:
   int children;
};

class senior : public parent
{
public:
   senior(string const& n, int c, int gc)
       : parent(n, c), grand_children(gc) {}

   int get_offspring()
   {
      return get_children() + grand_chidren;
   }
private:
   int grand_children;
};
```

Inheritance example (main)

```
int main()
{
   person author("Jerry Scott");
   person illustrator("Jim Borgman");
   parent pa("Walt Duncan", 2);
   /* Children: Chad and Jeremy */
   senior oldtimer("Rupert", 3, 9);

   cout << "Zits are written by "
        << author.get_name()
        << " and illustrated by "
        << illustrator.get_name()
        << endl;

   cout << pa.get_name() << " have "
        << pa.get_children() << " children."
        << endl;

   cout << oldtimer.get_name()
        << " is not a Zits character,"
        << " but he has "
        << oldtimer.get_offspring()
        << " decendants."
        << endl;
}
```

Polymorphism

When we in addition to inheritance use polymorphism (poly = many, morph = shifting) we can modify or customize the behavior of the base class. Thus we can have one class with behaviour that differ depending on which subclass it actually is.

The exact behaviour is not determined when compiling the program, but when the program runs (at runtime).

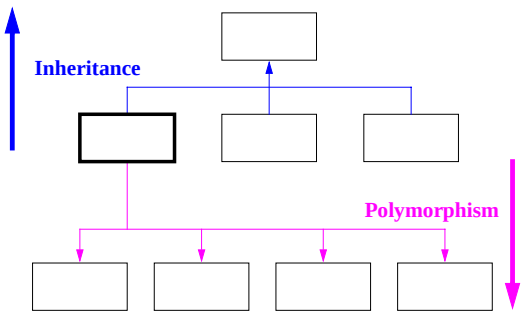
To enable polymorphism the base class must declare the morphing member functions as `virtual`.

Enabling polymorphism

A pointer (or reference) may refer to a object of the pointer (or reference) type, or to an object of any subclass (since a subclass is also the base class), or sub-sub-class etc.
This enables shifting behaviour, as the behavior will be determined by the actual class referred to. When a member function is called the base class will appear to shift its behavior according to the actual class the pointer (or reference) refer to.
The shifting behaviour is only visible when using a pointer (or reference) of a base class. The behaviour is determined at runtime.

No shifting behavior will take place when a function is called on an actual object, since the object can only be itself. It do not refer to some subclass with other behaviour. It is a super set of its base class(es), but that do not enable different behaviour, since the most specific member function (closest to the actual class) will always be used, and it is always the same. The behaviour is determined at compile time.

Polymorphism vs. Inheritance



Polymorphism example (classes)

```
class employee : public person
{
public:
    employee(string name, int salary);

    virtual int get_salary() { return salary; }

protected:
    int salary;
};

class manager : public employee
{
public:
    manager(string name, int salary, int bonus);

    int get_salary() { return salary + bonus; }

protected:
    int bonus;
};

class consultant : public employee
{
public:
    consultant(string name, int salary, int missions);

    int get_salary() { return salary * missions; }

protected:
    int missions;
};
```

Polymorphism example (main)

```
/* some details are left out */
int main()
{
   employee* worker[10];
   int pos = 0;

   while (cin >> type)
   {
      switch (type)
      {
      case EMPLOYEE:
         worker[pos] = read_employee(cin);
         break;
      case MANAGER:
         worker[pos] = read_manager(cin);
         break;
      case CONSULTANT:
         worker[pos] = read_consultant(cin);
         break;
      }
      pos = pos + 1;
   }

   for (int i = 0; i < pos; i = i + 1)
   {
         /* get_name() is inherited */
      cout << worker[i]->get_name() << " have "
         /* get_salary() is polymorph */
         << worker[i]->get_salary()
         << "SEK salary." << endl;
   }
}
```

Further topics

Abstract classes

```
class shape
{
public:
    shape();

    /* member function exist only in subclasses */
    virtual int get_area() = 0; /* abstract */
};
```

Casting

```
/* cast derived class to base class
 * cast base class to polymorph-specific
 */
dynamic_cast< new_type >(expression)

/* cast pointer to other pointer
   binary copy, no verification */
reinterpret_cast< new_type >(expression)

/* cast fundamental types */
static_cast< new_type >(expression)

/* cast to or from const of same type */
const_cast< new_type >(expression)
```

RTTI (Run Time Type Identification)

```
#include <typeinfo>

typeid(expression)

/* find type-name of actual class */
cout << typeid(worker[i]).name() << endl;
```