

Lecture 11

Inheritance and abstract types

With the programming tools make and cvs, we now have the infrastructure to actually do programming more efficiently.

Now let's start working on our original project, namely to make sound of different sorts. To do this, we need better mechanism for abstraction.

References:

- 3rd Edition, section 2.5, chapter 12.

Output to sound card instead of WAV file

Now suppose we want to output sound directly, instead of making a sound file. Naturally we cannot use WavFile to do that. So we refer to the documentation of our computer systems, and rewrite the class.

In many Unix system, sound is managed by a piece of software called OSS (Open Sound System). So we should read the documentation there:

- Open Sound System™ Programmer's Guide, chapter on Audio Programming, available from <http://www.opensound.com/pguide/>.
- You will also need access to the low level file operation: man pages of **open(2)**, **close(2)**, **write(2)** and **ioctl(2)** describes them.

At the end, you can modify the class WavFile to do what you want to do. You can see how this is done in the source of this lecture.

PMOOP(0396A)

PMOOP(0396A)-11.1

The resultant header

We then come up with another pair of files OSS.cc and OSS.h. The OSS class looks like this:

```
class OSS {
public:
    OSS();
    ~OSS();
    void set_sample_rate(int sr);
    int get_sample_rate();
    void set_sample_bits(int sb);
    void set_num_channels(int n);
    void write_sample(float amp);
    void write_sample(float left_amp, float right_amp);
private:
    ...
};
```

Interface is purposely made to be exactly the same as WavFile.

Using the resultant class

To use the new class instead of the WavFile class, only a few things need to be changed in the main program:

```
#include "OSS.h"
...
int main() {
    OSS wf;
    ...
}
```

We then add OSS.h and OSS.cc into the repository, and change the Makefile rule to add the OSS.o object, and recompile.

Now if you run the program, we hear some sound from the speaker of the computer, with no WAV file generated. Everything is easy, since internal change of the library does not require changing the user code.

PMOOP(0396A)-11.2

PMOOP(0396A)-11.3

New challenge: combining the two

Now that we have a library generating a WAV file and another generating sound directly, we would like our program to be able to do both. Let's make our program so that the following command output sound directly:

```
note2wav
```

and the following command output sound to WAV file *filename*:

```
note2wav filename
```

The arguments can be obtained by the main() function, through its argument. We can write the main() function like this:

```
int main(int argc, char *argv[])
```

Here **argc** is the number of "arguments" received by the program, while **argv** is the arguments themselves. An argument is a word in the above command, like "note2wav" or "filename".

A simple program that illustrate the idea

To understand the idea about command-line arguments, try the following program:

```
#include <iostream>
using namespace std;
int main(int argc, char *argv[]) {
    for (int i = 0; i < argc; ++i)
        cout << "argv[" << i << "] = " << argv[i] << endl;
}
```

If you compile the program and output it as **arguments**, a run of the program looks like this:

```
>./arguments foo "foo bar"
argv[0] = ./arguments
argv[1] = foo
argv[2] = foo bar
```

PMOOP(0396A)-11.4

PMOOP(0396A)-11.5

The real problem comes: if-else code

But the real problem is not the command line, but the different classes. See what we have to do to achieve the effect we described.

```
OSS *oss = 0;
WavFile *wf = 0;
if (argc==1) {
    oss = new OSS();
} else {
    wf = new WavFile(argv[1]);
}
auto_ptr<OSS> osac(oss);
auto_ptr<WavFile> wfac(wf);
```

```
int rate = 22050;
if (oss != 0) {
    oss->set_sample_rate(rate);
    oss->set_sample_bits(16);
    oss->set_num_channels(2);
} else if (wf != 0) {
    wf->set_sample_rate(rate);
    wf->set_sample_bits(16);
    wf->set_num_channels(2);
}
```

Note the ugliness of the code: although we know for sure that everything is the same in the two versions, we have to write it twice. If we have 10 ways to output, we will have to write things 10 times.

PMOOP(0396A)-11.6

What we really want: polymorphism

Of course, this is stupid. Can we write in a more elegant way, so that we write only once instead of many times?

Before answering that, let's see what we really need.

- We want to have a way for some value to be **either a WavFile or a OSS** class object. (Polymorphic behaviour #1)
- We need a way to **tell the compiler that the two types are "compatible"**, and the degree in which they are compatible.
- We need to be able to call a member function of such an object, in such a way that **the member function called depends on the actual type**. (Polymorphic behaviour #2)

In general, we call this polymorphism. The real type of a variable, or the real meaning of a function, is known only when the program actually get executed, not when it is compiled.

PMOOP(0396A)-11.7

Polymorphism: dictionary meanings

polymorphism (WordNet):

1. (chemistry) the existence of different kinds of crystal of the same chemical compound [syn: pleomorphism]
2. the existence of two or more forms of individuals within the same species (independent of sex differences)

polymorphism (Free On-line Dictionary of Computing):

A concept first identified by Christopher Strachey (1967) and developed by Hindley and Milner, allowing types such as list of anything.

...
In object-oriented programming, the term is used to describe variables which may refer at run-time to objects of different classes.

PMOOP(0396A)-11.8

The mechanism: inheritance

A class, say **X**, can be **derived** from another class, say **Y**:

```
class X: public Y {
    ...
};
```

- **X** is called the derived class, **Y** is called the base class. The word "public" means that the public members of **Y** is available to everybody.
- Whenever a value of type **Y*** or **Y&** is expected, we can supply a value of type **X*** or **X&** instead. (Polymorphic behaviour #1)
- Note that polymorphism is only done through pointers and references. An object of type **X** can never be a **Y**.
- The derived class **X inherits** all properties of its base class **Y**. That is, **X** contains all data members of **Y**, and supply all member functions of **Y**.

PMOOP(0396A)-11.9

Classic example

A classic example is something like this:

```
class Employee {
public:
    void print();
private:
    int employee_no;
};

class Manager: public Employee {
public:
    void print();
private:
    vector<Employee *> managed;
};
```

An employee has a data member `employee_no`. A manager, being an employee, also has an `employee_no`. But they also manage some Employee's, stored in `managed`.

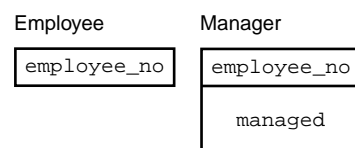
Since `managed` is a vector of pointers, it is polymorphic: a Manager can manage Managers.

Employee defines `print()`, and Manager override it. If we have **Manager m**, `m.print()` calls `Manager::print()`, not `Employee::print()`.

PMOOP(0396A)-11.10

Data organization with inheritance

For each class object in the memory, a place is allocated for each data member. Inheritance simply extends that idea. For example, an Employee object and a Manager object looks like this in memory.



Note that the memory layout of the first part of a Manager object is exactly the same as an Employee object.

Thus if Employee has a member function, say `get_employee_no`, it can also be used for a Manager object (if Manager does not override it).

PMOOP(0396A)-11.11

Polymorphic functions

Pointers and references of `Employee` are polymorphic (Polymorphic behaviour #1). But its member functions are not polymorphic (Polymorphic behaviour #2). For example:

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

This always execute `Employee::print()`, even if `*emp` is a `Manager`.

Why we have this behaviour in C++ (it is not this way in Java)?
Answer: it is a C++ design decision.

If `print()` is polymorphic, the program must decide whether to call `Employee::print()` or `Manager::print()` at run-time: program is slower.

C++ cares a lot about efficiency. By default functions are not polymorphic, so people won't get slow programs by accident.

PMOOP(0396A)-11.12

Virtual functions

We mark a function as polymorphic by the keyword **virtual**:

```
class Employee {
public:
    virtual ~Employee();
    virtual void print();
private:
    int employee_no;
};

class Manager: public Employee {
public:
    ~Manager();
    void print();
private:
    vector<Employee *> managed;
};
```

Points to note:

- `Manager::print()` is virtual since `Employee::print()` is virtual. Similarly, `Manager::~~Manager()` is virtual.
- Destructors **chains** automatically: at the end of the destructor `Manager::~~Manager()`, `Employee::~~Employee()` is called.

PMOOP(0396A)-11.13

Trivial Example

A simple example shows how polymorphic behaviour works.

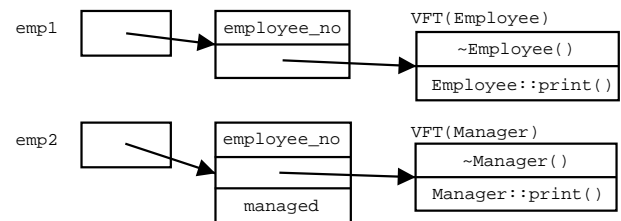
```
int main() {
    Employee *emp1 = new Employee();
    Employee *emp2 = new Manager(); // Okay: Polymorphic #1
    emp1->print(); // Calls Employee::print()
    emp2->print(); // Calls Manager::print(): Polymorphic #2
    delete emp2; // delete the managed vector: virtual destructor
    delete emp1;
}
```

The compiler usually doesn't know whether `*emp1` and `*emp2` are of type `Employee` or `Manager` (or some other derived class of `Employee`).

Instead, each `Employee` has a pointer to a per-class structure called the *Virtual function table* (VFT). A VFT answers questions like "what function should be called for `print()`?"

PMOOP(0396A)-11.14

Program internals: Virtual Function Table (VFT)



With virtual function tables, it becomes possible to generate code for polymorphic function calls like `emp1->print()`:

- find `emp1->(VFT)`
- find the `print()` function of that VFT
- call that function (giving `emp1` to it as the **this** pointer).

PMOOP(0396A)-11.15

Two types of classes

There are two types of classes in C++:

- Some are **designed for efficiency**. They have no virtual function, and thus no VFT. They target for efficiency rather than polymorphism.
- Even though polymorphic behavior #1 is still there, we should really not use it.
- Some are **designed for polymorphic behaviour**. Certain functions are marked as virtual and are intended for extension. Other functions should not be overridden.
- Since polymorphic behavior means dereferencing VFTs, they are considerably slower than the other type of classes.
- Polymorphic classes should have virtual destructors** (even if empty). Otherwise, derived classes cannot be destructed correctly through a pointer to the base class. (**See exercise.**)

PMOOP(0396A)-11.16

Solving our problem

At this point we have enough understanding to solve our earlier problem.

- We want `OSS` and `WavFile` to have a **common base class**.
- It is no good for either `OSS` or `WavFile` to be the common base class. The data members of the two types are quite different from each other.
- Let's make a third type, called `SoundOutput`, that is the base class of both `OSS` and `WavFile`.
- The main program creates a `OSS` object or a `WavFile` object depending on the command-line arguments. Then it uses a `SoundOutput` pointer to refer to either of them. I.e., **SoundOutput is polymorphic**.
- We then use the `SoundOutput` pointer to do all the operations. The correct member function will be called: the compiled code will have the VFT to do the trick.

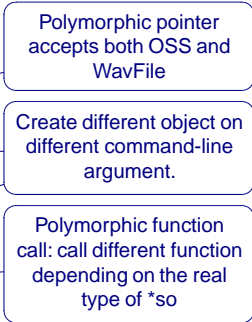
PMOOP(0396A)-11.17

User code

With this idea of inheritance, we can write our main function like this:

```
#include "SoundOutput.h"
#include "WavFile.h"
#include "OSS.h"

int main(int argc, char *argv[]) {
    SoundOutput *so = 0;
    if (argc==1)
        so = new OSS();
    else
        so = new WavFile(argv[1]);
    auto_ptr<SoundOutput> soac(so);
    int rate = 22050;
    so->set_sample_rate(rate);
    ...
}
```



Where to define SoundOutput: SoundOutput.h

SoundOutput is a polymorphic class. It must define all the member functions that need polymorphic behaviour to be virtual.

```
#ifndef _SOUNDOUTPUT_H
#define _SOUNDOUTPUT_H

class SoundError { ... };
class SoundOutput {
public:
    SoundOutput() {};
    virtual ~SoundOutput() {};
    virtual void set_sample_rate(int sr) {};
    ...
    virtual void write_sample(float left_amp, float right_amp) {};
};

#endif
```

OSS.h and WavFile.h: using SoundOutput

The changes needed for OSS.h and WavFile.h are minimal: we just remove all SoundError reference to it, and then add the base classes:

```
#ifndef _OSS_H
#define _OSS_H

#include "SoundOutput.h"

class OSS: public SoundOutput {
public:
    OSS();
    ~OSS();
    ...
};

#endif

#ifndef _WAVFILE_H
#define _WAVFILE_H

#include "SoundOutput.h"

class WavFile: public SoundOutput {
public:
    WavFile();
    ~WavFile();
    ...
};

#endif
```

Abstract classes

- Conceptually, the **SoundOutput class only specifies what is expected from a derived class of SoundOutput**. It does not make sense to have an object of type SoundOutput.
- We call such classes "**abstract class**": it defines the interface of the class without defining also the implementation.
- The SoundOutput functions are not meaningful: they are just place-holders. It is better to mark it as unmeaningful, so that other classes cannot inherit them.
- C++ supports this concept by allowing virtual member functions to be **pure**, i.e., does not have implementation. Derived classes can override them, thus making them non-pure.
- A class with at least one pure virtual function is said to be **abstract**. Abstract classes cannot be instantiated, i.e., making an object of an abstract class will cause a compile-time error.

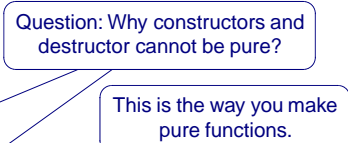
SoundOutput.h with pure functions

SoundOutput is best abstract. It makes sure that a SoundOutput cannot be created, and that OSS and WavFile provide all the needed functions.

```
#ifndef _SOUNDOUTPUT_H
#define _SOUNDOUTPUT_H

class SoundError { ... };
class SoundOutput {
public:
    SoundOutput() {};
    virtual ~SoundOutput() {};
    virtual void set_sample_rate(int sr) = 0;
    ...
    virtual void write_sample(float left_amp, float right_amp) = 0;
};

#endif
```



Miscellaneous 1: Chaining calls

- At some times, a member function of the derived class might want to call the member function of the base class, before/after its own work.
- This can be achieved with format **ClassName::function_name()**.
- For example, we might want SoundOutput to manage the sample_rate:

```
class SoundOutput {
public:
    virtual void set_sample_rate(int sr) {
        { sample_rate = sr; };
    }
    int get_sample_rate() {
        { return sample_rate; };
    }
private:
    int sample_rate;
};

class WavFile: public SoundOutput {
public:
    void set_sample_rate(int sr) {
        if (init)
            throw SoundError(...);
        SoundOutput::set_sample_rate(sr);
    };
};
```

Miscellaneous 2: Chaining constructors

We already know that destructors are chained at the end automatically. Constructors are just the reverse: it chains at the beginning.

```
class A {
public:
    A(int n)
    { cout << n << endl; }
}

class B: public A {
public:
    B(int p, int q): A(p)
    { cout << q << endl; }
}
```

If we construct an object of class B with `new B(2, 3)`, `B::B(2,3)` is executed. The first thing to do in the constructor is to call `A::A(2)`. So we get the following:

2
3

Note that we chain the constructor just in the same way as initializing data member. If we do not specify how to chain, it uses the default constructor.

PMOOP(0396A)-11.24

Miscellaneous 3: Access of members across types

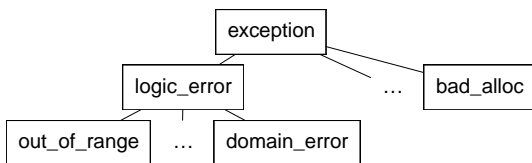
- We have known that **public** members can be used outside a class, while **private** members can only be used within the class defining it.
- There is a third access type: **protected**. It is very similar to private, except that the member is also accessible by a derived type.
- This is usually done for functions that derived classes may need to override, but are never called from user code. Making it protected instead of private allows the derived class to make chained calls.
- On the other hand, data members are seldom made protected. This is because it gives you the same problem as public data member, i.e., you can never change the implementation.

PMOOP(0396A)-11.25

Miscellaneous 4: Inheritance and exceptions

An exception of a derived type can be caught by a **catch** clause of a base type. So we can have a hierarchy of types for exception catching.

And the standard library actually do this. In the `<stdexcept>` header, we have exceptions of this hierarchy:



Thus if an exception of type **out_of_range** is thrown (e.g., by a **vector** operation), a `catch(logical_error &)` clause will catch it. We can use this mechanism to catch a group of exceptions.

PMOOP(0396A)-11.26

Miscellaneous 5: Cost of polymorphism

We have mentioned that polymorphism is not very efficient. The efficiency problems come in several parts:

- **Space cost:** we need 4 bytes for VFT pointer for each object. So it is certainly unsuitable if you need a million objects of a few bytes.
- **Redirection cost:** we need to dereference VFT to call functions. So if you need to call small functions a huge number of times (e.g., for complex numbers arithmetics), don't use polymorphism.
- **Inlining cost:** polymorphic calls are impossible to inline. So if you don't want to spend time calling functions, or need statements to be optimized between function calls, don't think of polymorphism.

If any of these stops you from a necessary polymorphism, it usually means that polymorphism is too fine-grained. Try to redesign the library.

PMOOP(0396A)-11.27

Conclusion

- Polymorphism in C++ allows a pointer or reference of one type to refer to an object of another type.
- The former type must be a base class of the latter. The compiler assumes only the interface of the base class.
- We can mark member functions as virtual, which allows them to be called from a base-class pointer or reference. At run-time, the program find out what function to call depending on the type of the actual object.
- A virtual member function can be marked as pure, i.e., undefined. Classes with pure virtual member functions cannot be instantiated.
- To achieve polymorphism, the compiler generate a virtual function table for each class. Each object has a pointer to one of these tables.
- Polymorphism is not very efficient. When you need the best efficiency, polymorphism should be avoided.

PMOOP(0396A)-11.28

Exercise

Read the following program and predict its behaviour. Try to compile and run it to see whether your predictions are correct, and try to reconcile and difference found.

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

```
class Base {
public:
    Base(int myid) { id = myid; }
    ~Base() { cout << "Base destructed: " << id << endl; };
    virtual void f() { cout << "Base::f() called: " << id << endl; };
    void g() { cout << "Base::g() called: " << id << endl; };
    int id;
};

class Component {
```

PMOOP(0396A)-11.29

```

public:
    virtual ~Component() { cout << "Component destructed." << endl; };
};

class Derived: public Base {
public:
    Derived(int myid): Base(myid) {};
    ~Derived() { cout << "Derived destructed: " << id << endl; };
    void f() { cout << "Derived::f() called: " << id << endl; }
    virtual void g() { cout << "Derived::g() called: " << id << endl; }
private:
    Component x;
};

int main() {
    Base a(1);
    Derived b(2);
    Base *c = new Base(3);
    Base *d = new Derived(4);
}

```

```

Derived *e = new Derived(5);
a.f();
b.f();
c->f();
d->f();
e->f();
a.g();
b.g();
c->g();
d->g();
e->g();
delete e;
delete d;
delete c;
}

```