

COMP3062 Advanced Databases

99-03 Laboratory 2: C++ and ONTOS

1 Abstract class

An abstract class can be thought of as an incomplete class that is more or less finished with each subsequent derivation. It can also be thought of as a basic mechanism to protect you from creating “zombie” objects. As an example, entity is an abstract idea. Everything can be an entity, but keeping those entities in a program does not improve anything. It sounds like you can access many student identifiers but you will never know who they are. On the other hand, the idea of entity becomes concrete only when a subclass such as student is derived from entity.

The following is an example to define entity as an abstract class. An abstract class is specified when a virtual function, `id()`, is made pure by an initialiser, `= 0`. In `main()`, we first try to create an entity object.

```
#include <iostream.h>
#include <string.h>

class entity {
private:
    int entity_id;
public:
    entity(){
        virtual int id() = 0;
    };

main()
{
    entity e1;
}
```

- **Task 1:** Compile it. Can you compile it?
- **Task 2:** What does a pure virtual function mean?

2 Static Class Members

A class is a type, and each object of the class has its own copy of the data members of the class. However, some types are most elegantly implemented if all objects of that type share some data. In the following, we use an example to see why do we need static class members.

We first show the difficulties of creating unique identifiers without the help of a static class member. We define `student` as a subclass of `entity`, and add a constructor and a protected member function `getid()` in `entity` class. The `entity` class is still an abstract class. You can create `student` objects by using the `student` constructor, and pass the argument `sid` to `entity` class.

```
#include <iostream.h>
#include <string.h>

class entity {
private:
    int entity_id;
protected:
    int getid(){return entity_id;}
public:
    entity(int id){ entity_id = id; }
    virtual int id() = 0;
};

class student : public entity {
    char *name;
public:
    student(char* str, int sid);
    void print();
    int id(){return getid();}
};

student::student(char* str, int sid) : entity(sid)
{
    if (str != NULL){
        name = new char[strlen(str)+1];
        strcpy(name, str);
    } else name = NULL;
}

main()
{
    student s1("Albert", 10);
    cout << s1.id() << endl;
}
```

■ **Task 3:** Can you explain the whole initialization process for the following line?

```
student s1("Albert", 10);
```

Why do we need initialization list? Can you do the same thing without initialization list?

Looking at the above example, we find that it is strange because students have to assign identifier by themselves when they are created. That is to say, the application needs to keep a global

counter for generating student identifiers, and increase the counter when a student is created. However, it is the essential part of entity class to automatically create an identifier for an object. The question here is how to do it. The answer to the question is a static class member. A static data member acts as a global variable for objects of the class. A static data member is initialized outside the class definition in the same manner as a non-member variable. Compared with the previous example, what is the difference in the following example?

```
#include <iostream.h>
#include <string.h>

class entity {
private:
    static int current_id;
    int entity_id;
protected:
    int getid(){return entity_id;}
public:
    entity(){ entity_id = current_id; current_id++; }
    virtual int id() = 0;
};

class student : public entity {
    char *name;
public:
    student(char* str);
    void print();
    int id(){return getid();}
};

student::student(char* str) : entity()
{
    if (str != NULL){
        name = new char[strlen(str)+1];
        strcpy(name, str);
    } else name = NULL;
}

int entity::current_id = 0;

main()
{
    student s1("Albert");
    student s2("William");
    student s3("Steve");

    cout << s1.id() << endl;
    cout << s2.id() << endl;
    cout << s3.id() << endl;
}
```

- **Task 4:** How did we define a static data member in entity?
- **Task 5:** How did we initialize the static data member?
- **Task 6:** How did we assign an identifier to each object?
- **Task 7:** The `current_id` is defined as a static but private data member. Can we access it as follows?

```
student s1("Mark");
cout << s1.current_id;
```

- **Task 8:** Compile it, and check the identifiers assigned to each student.
- **Task 9:** Is it better than the previous example?

3 Multiple Inheritance

Suppose that we want to create `person` as a subclass of `entity`, and create `student` and `employee` as subclasses of `person`, and create `ta` as a subclass of both `student` and `employee` as shown in Fig 3. The following is an example for supporting those classes.

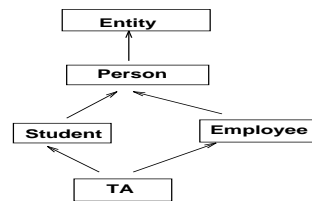


Figure 1: The Class Structure

```
#include <iostream.h>
#include <string.h>

/*
 * class definitions
 */
class entity {
private:
    static int current_id;
    int entity_id;
protected:
    int getid(){return entity_id;}
public:
    entity(){ entity_id = current_id; current_id++; }
    virtual int id() = 0;
```

```
};
int entity::current_id = 0;

class person : public entity {
    char *name;
    int tel;
public:
    person(char* str = NULL, int i = 0);
    int id(){return getid();};
    void virtual print();
};

class student : public person {
    char *degree;
public:
    student(char* str, char* d = NULL, int t = 0);
    void print();
};

class employee : public person {
    char* department;
public:
    employee(char*, char* d = NULL, int i = 0);
    void print();
};

class ta : public student, public employee {
    char* duty;
public:
    ta(char* n, char* deg = NULL, char* dept = NULL, char* work = NULL,
        int i = 0);
    void print();
};

/*
 * constructors and member functions
 */
person::person(char* str, int i) : entity()
{
    if (str != NULL){
        name = new char[strlen(str)+1];
        strcpy(name, str);
    } else name = NULL;
    tel = i;
}

void
```

```
person::print()
{
    cout << "name: " << name << endl;
    cout << "tel: " << tel << endl;
    cout << "id:  " << id() << endl;
}

student::student(char* n, char* d, int i) : person(n, i)
{
    if (d != NULL){
        degree = new char[strlen(d)+1];
        strcpy(degree, d);
    } else degree = NULL;
}

void
student::print()
{
    person::print();
    cout << "degree: " << degree << "\n";
}

employee::employee(char* n, char* d, int i) : person(n, i)
{
    if (d != NULL){
        department = new char[strlen(d)+1];
        strcpy(department, d);
    } else department = NULL;
}

void
employee::print()
{
    person::print();
    cout << "department: " << department << "\n";
}

ta::ta(char* n, char* deg, char* dept, char* work, int t)
    : student(n, deg, t), employee(n, dept, t*10)
{
    if (work != NULL){
        duty = new char[strlen(work)+1];
        strcpy(duty, work);
    } else duty = NULL;
}

void
ta::print()
```

```

{
    student::print();
    employee::print();
    cout << "duty: " << duty << "\n";
}

/*
 * main!
 */
main()
{
    person   p1("William", 101);
    student  s1("Paul", "Honour", 102);
    employee e1("Steve", "Dept. of CS", 103);
    ta       t1("Albert", "MS", "CS", "C53", 104);

    p1.print();
    s1.print();
    e1.print();
    t1.print();
}

```

Read/compile/run the above example, and think about the following issues.

- **Task 10:** How do you use initialisation list in a single/multiple inheritance?
- **Task 11:** How do you access the inherited data members?
- **Task 12:** How do you use the inherited member functions?
- **Task 13:** How do you initialise all the base classes of class `ta`.
- **Task 14:** What is ambiguity? How do you solve it generally? For example, can you add a single line as follows at the end of `main()`? If can't, why?

```
cout << t1.id() << endl;
```

Instead, can you do something as follows? If can, why?

```
cout << t1.student::id() << endl;
```

- **Task 15:** More problems? In the above example, we create `t1` as a teaching assistant. When we execute the following code

```
t1.print();
```

we find that two identifiers have been assigned to `t1`. Why does this happen? It is not because `print()` tries to print them out twice. It is because that two person objects have been created for a “`ta`” object. One is inherited from `student`, and the other is inherited from `employee`. The `person` constructor has been called twice for a single `ta` object. As you might know, here, we need to use virtual base class, in order to solve this problem.

- Change the following lines

```
class student : public person {
class employee : public person {

to
```

```
class student : public virtual person {
class employee : public virtual person {
```

What does “virtual” mean? Run it. What happens now? We found that the “ta” will have only one id to be assigned. But the name and the telephone number have not been initialised. Why?

- To solve it

```
ta::ta(char* n, char* deg, char* dept, char* work, int t)
    : student(n, deg, t), employee(n, dept, t*10)
```

to

```
ta::ta(char* n, char* deg, char* dept, char* work, int t)
    : student(n, deg, t), employee(n, dept, t*10), person(n, t)
```

Is this a good idea to do so?

- Can you use `t1.id()` at the `main()` now?

```
cout << t1.id() << endl;
```

Why?

- Here, you still print the same `t1`'s name and id twice. Find a general resolution to print name and id only once for `ta` objects, and print out the other information as usual (`person`, `student`, `employee`, `ta`).

4 What is ONTOS DB?

ONTOS DB is a multi-user, distributed object database with a C++ class library interface. As a database, its fundamental purpose is to provide a reliable persistent storage facility for C++ objects. To be more specific, it allows objects denoted by C++ program variables to have a lifetime that is longer than that of the program that created them, and it allows C++ programs to retrieve persistent objects (objects created by other programs) into their program variables.

The system also provides a set of tools for designing and developing ONTOS DB applications such as:

- **DBATool**: utility for configuring an ONTOS DB database.
- **classify**: schema compiler for loading schema based on C++ header files.
- **cplus**: utility for processing C++ code.

An ONTOS DB database is made up of files on disk. Each of these disk files is called an “area”. ONTOS provides a file called `OntosSchema`, which contains basic information including data about the classes provided with ONTOS. To create a new database, the `OntosSchema` file must

be copied into a file that will be the basis for the new database. This file is specified as the “kernel area” of the new database. A database is assigned a name – used to refer to the entire database – called a “logical database”.

ONTOS has been installed on iwaki!!!

5 A tutorialDB

Copy the introduction directory as follows.

```
iwaki% cp -r /dept/dcs/comp3062/public/examples/ontos/introduction ~
```

Then, change the directory into which you copied the files.

```
iwaki% cd ~/introduction
```

5.1 Logical database and areas

The way of setting up a kernel area and a database can be seen by run

```
iwaki% make -n db
```

With `-n` option, the `make` command doesn't execute any commands, but shows you what it will do without the `-n` option. It first copies the `OntosSchema`, then registers a kernel area and a new logical database for you. The name for the kernel area and the database is made up of your user name suffixed with `tutorialDB`. Here are some things you must remember when you use ONTOS.

- Each kernel area and database you use must be registered as a name prefixed with your user name.
- The kernel area must be registered on `iwaki`.
- Don't destroy other student's kernel-area/database information kept in DBATool.

If you are ready, then run

```
iwaki% make db
```

to create the kernel area and the database. The DBATool has an interactive mode as well as the command mode.

```
iwaki% DBATool
```

Note that, in the following, `>>` is a prompt used in DBATool. First, be familiar with help commands. The help commands show you the usages of commands. For example,

```
>> help
>> help database
>> help area
>> help show
>> help show database
>> help show kernel
>> help show area
>> help register
>> help register database
>> help register kernel
>> help register area
```

Next, be familiar with show commands.

```
>> show database
```

In order to see the details about your database, run the same command with your database name.

```
>> show kernel
```

Run this command to see details about your kernel area. You can quit from DBATool by

```
>> quit
```

Your kernel area and the registration information can be cleaned up by

```
iwaki% make clean-db
```

Clean up everything, then try to register the same kernel area and the same database by using the interactive mode of DBATool.

```
iwaki% DBATool
```

You have to know two commands in DBATool when you use interactive mode to register areas and databases. First, **sync** commits the changes made in the session's local copy to the Registry. Then the local buffer is refreshed with a new copy of the Registry. Second, **abort** is a command to abort.

5.2 Load Schema onto ONTOS

The following command

```
iwaki% make -n classify
```

show you how to load schema information onto ONTOS. The **+D** option specifies the database name. The **-I** option specifies the directory holding files to be included. The **+c** option specifies a control file holding further instructions for the classify utility. Load the schema onto ONTOS

```
iwaki% make classify
```

6 Compilation and Execution

To compile a command called `main`, run

```
iwaki% make chapter1
```

The above command causes `cplus` commands to be executed. To run `main`, you must define an environment variable called `DBNAME` as follows. The value of the `DBNAME` must be the same as the database name you have registered with `DBATool`.

```
iwaki% setenv DBNAME ${USER}tutorialDB  
iwaki% ./main
```

- **Task 17:** Read `main.cxx` file. Can you run the same command twice? Why?
- **Task 18:** What have we done in this lab?