Lec_29.md

# Lecture 29

Nov. 24/2020

# Inheritance

A property of **Object-oriented programming** and C++ mechanism that *facilitates code reuse*

## Name/Contact Inheritance Example

```
class Name {
  private:
    char * theName;
  public:
    Name();
    Name(const char* name);
    Name(Name & r);
    ~Name();
    void setName(constchar* newName);
    Name & operator=(Name & r);
    void print();
};
```

Notes:

1. Standard methods in this class

- `Name();`
  - Empty constructor
- `Name(const char* name);`
  - char array constructor
- `Name(Name & r);`
  - Copy constructor
- `~Name();`
  - Destructor

```
class Contact: public Name{
private:
  char * theAddress;
public:
  Contact();
  ~Contact();
  Contact(Contact & r);
  Contact(const char* newName, const char* newAddress);
  void setNameAddress(const char* newName,
  const char* newAddress);
  Contact & operator=(Contact & r);
  void print();
};
```

Notes:

1. `class Contact: public Name`

- This line reads as:
  - class Contact inherits Name publicly

2. Standard methods in this class

- `Contact();`
  - Empty constructor
- `Contact(const char* newName,const char* newAddress);`
  - char array constructor
  - takes both a newName string field and a newAddress string field
- `Contact(Contact & r);`
  - Copy constructor
- `~Contact();`
  - Destructor

## Sub Class Usage

```
#include "Name.h"
#include "Contact.h"

int main() {
  Name n;
  Contact c;

  n.setName("Tarek Abdelrahman");

  n.print();

  c.setName("Tarek Abdelrahman");

  c.setNameAddress("John Smith", "123 Main Street");

  n.setNameAddress("Tarek Abdelrahman","123 Main Street");

  c.print();

  return (0);
}
```

Notes:

1. `c.setName("Tarek Abdelrahman");`

- This *works*, since `Contact` inherits `Name`

2. `c.setNameAddress("John Smith", "123 Main Street");`

- This *works*, since `setNameAddress` defined in `Contact` class

3. `n.setNameAddress("Tarek Abdelrahman","123 Main Street");`

- This is a *compile-time error*
  - `Name` class has no method `setNameAddress`
  - **Inheritance** works one way
- **Inheritance** is a *one-way street*

4. `c.print();`

- This *works*
  - But the `print()` in `Contact` gets called, since it eclipses the `print()` in `Name`

### Contact Class Implementation

```
void Contact::setNameAddress(const char* newName,const char * newAddress) {
  setName(newName);
  delete [] theAddress;
  theAddress = new char[strlen(newAddress)+1];
  strcpy(theAddress, newAddress);
}
void Contact::print() {
  Name::print();
  cout << theAddress << endl;
}
```

Notes:

1. Can invoke and use member functions of `Name` as if they were member functions of `Contact`

- No need to define **objects**
- `setName(newName);`
  - Calling the `setName` function in `Name`
- `Name::print();`
  - Calling the `print()` function in `Name`
  - Notice the use of the **scope resolution operator** `::`

2. Functions in `Contact` *override* functions with the same **signature** in `Name`

- `Name::print();`
  - Calling `print();` by itself would call `Contact::print();`

## Aspects of Inheritance

General template for **inheritance**:

```
class Derived: public Base{
  ...
};
```

In general, the **derived** class inherits from the **base** class

- Questions we may have:
  - What does the **derived** class inherit?
  - What does the **derived** class *not* inherit?
  - What can the **derived** class add?
  - What happens when variables of **derived** class are created?
  - What happens when variables of **derived** class are destroyed?
- Special questions:
  - What is the relationship between objects of **base** and **derived**?
  - What is the relationship between **pointers** of **base** and **derived** objects?
  - What are **virtual** functions and **abstract classes**?

### What is Inherited

All the data members of the base class are inherited (both private and public)

- In `Contact`, `theName` is inherited from `Name`
  - `theName` is a private data member

However, private data members of base are **not accessible** in **derived**

- So how do we access private data members?
  - Use public functions inherited from superclass
  - Use `setName` and `print()` from `Name`
    - Specifically, `Name::print()`

Another exception: private function members of base are **not accessible** in **derived**

```
void Contact::print() {
  cout << theName << endl; //NOT VALID
  cout << theAddress << endl;
}
```

Instead, use

```
void Contact::print() {
  Name::print();
  cout << theAddress << endl;
}
```

```
void Contact::setNameAddress(const char * newName,const char * newAddress) {
  setName(newName); //can call Name::setName()
  delete [] theAddress;
  theAddress = new char[strlen(newAddress)+1];
  strcpy(theAddress, newAddress);
}
```

## What is NOT Inherited

The **constructors** and **destructors** are not inherited

- Think from procedural point of view
  - `Contact` object is an object *with* data members of `Name`
    - Need to instantiate (construct) `Contact` object
- `Contact()`, `~Contact()` must be defined
  - Default constructor is provided if none are defined

**Overloaded assignment operator** `operator=` is *not* inherited

- One is provided by default in `Contact`, as usual

**Friend** functions are *not* inherited

- Unfortunately, we can't steal (inherit) friends

## What can be added by Derived

**Derived** can have new data members (private and public)

- `Contact` adds `theAddress()`

**Derived** can add new function members (private and public)

- `Contact` adds `setNameAddress()`

## Object Creation

An object of type **derived** class is created

- By default, the default constructor of the **base** class is called
  - So if the **base** class has no default constructor, a *compile-time error* is thrown

Process (In Order):

1. **Derived** object is created
2. Default constructor of **base** class is called
3. Constructor of **derived** class is called

```
Contact::Contact() {
  theAddress = new char [1];
  theAddress[0] = '\0';
}
```

Notes:

1. The default constructor for the `Name` class is called

- More generally: You, the programmer, are responsible for the *new* data added by the **derived** class on the **base** class
- Assume `Name` part is already constructed

But what if we want a *non-default constructor* of the **base** class to be called?

- How to call `Name(const char* name)` ?
  - Use an **Initializer List**

## Initializer List

**Initializer lists** are used to initialize certain variables when a function is called

- Can be used to call the constructor of the **base** class

```
Contact::Contact(const char * newName,const char *newAddress):Name(newName) {
  theAddress = new char[strlen(newAddress)+1];
  strcpy(theAddress, newAddress);
}
```

Notes:

1. Notice in the function name line:

- `:Name(newName) { ..function body.. }`
- This is an **initializer list**
  - Invokes the `Name::Name(const char* name)` constructor
  - Instead of the default constructor for `Name`

## Object Destruction

The order for **object destruction** is the mirror of the **object creation** process

Process (In Order);

1. Destructor of **derived** class is called
2. Destructor of **base** class is called
3. The **derived** class is deleted

## Relationship between Base objects and Derived objects

This part is ***important***.

C++ is a **strongly typed language**

- Variables have a single, defined type
- `int` variables are *only* `int` 's, not strings, floats, or chars;
- Even `auto` derives a *single* type upon assignment.
  - The type of the variable *cannot* change

However, for **inheritance**, there is a special relationship between **base** and **derived** classes

1. Objects of type **derived** are ***also*** of type **base**

- The objects can simply *ignore* the new methods in the **derived** class
  - Pretend it is a **base** object
  - `Contact` can *ignore* new methods to appear as a `Name` class

2. Objects of type **base** are ***not*** of type **derived**

- `Name` class *cannot expand* to appear as a `Contact` class

### Implications of Derived and Base Objects

```
Name n;
Contact c;

//assume n.operator=(Name &) defined
//assume c.operator=(Contact &) defined
n = c;
c = n;
```

Notes:

1. Question: Are the following defined?

- Assuming `n.operator=(Name &)` is defined
  - Notice the parameter takes a `Name` object by ref
- Assuming `c.operator=(Contact &)` is defined
  - Notice the parameter takes a `Contact` object by ref

2. `n=c;`

- ***Yes***, `n=c` is defined
  - Since `c` is of type `Contact` , which is **derived** from `Name`
  - `c` can shrink (ignore new data members) to appear as a `Name` object
- `n=c` can *only* copy values defined in the `Name` class

3. `c=n;`

- ***No***, `c=n` is not defined
  - `n` is of type `Name` , which is the **base** for `Contact`

- Larger class ( `Contact` ) can shrink
- Smaller class ( `Name` ) cannot grow

In ***general***, one can use a **derived** object anywhere a **base** object can be used

```
Contact c;
bool foo(Name x);//for any function foo
val = foo(c);
```

Is **defined** (since `Contact` is **derived** from `Name` )