VE280 2021FA RC5

L10: I/O Streams

`cin` , `cout` & `cerr`

>> and <<

In C++, streams are **unidirectional**, which means you could only `cin >>` and `cout <<`.

If we look into `cin`, it's an object of class `istream` (input stream). `operator>>`(the extraction operator) is one of it's member function.

Check `std::istream::operator>>`, similar for `cout` & `cerr` (`ostream`)

```
istream& operator>> (bool& val);
istream& operator>> (short& val);
istream& operator>> (unsigned short& val);
istream& operator>> (int& val);
istream& operator>> (unsigned int& val);
istream& operator>> (long& val);
istream& operator>> (unsigned long& val);
istream& operator>> (long long& val);
istream& operator>> (unsigned long long& val);
istream& operator>> (float& val);
istream& operator>> (double& val);
istream& operator>> (long double& val);
istream& operator>> (void*& val);
```

Many type of parameter it takes -> it knows how to convert the characters into values of certain type

Return value also a reference of `istream` -> it can be cascaded like `cin >> foo >> bar >> baz;`

Some other useful functions:

```
istream& getline (istream& is, string& str);
std::ios::operator bool // member of istream -> if(cin), while(cin)
istream& get (char& c); // member of istream
```

What to notice is that `>>` will read until reaching the next space or `\n`, and the space and `\n` will still be left in the buffer. And space and `\n` won't be stored into the parameter. While `getline` would read a whole line and discard the `\n` at the end of the line directly. And `get()` reads a single character, whitespace or newlines.

Stream Buffer

`cout` and `cin` streams are buffered (while `cerr` is not).

You need to run `flush` to push the content in the buffer to the real output.

`cout << std::endl` is actually equivalent to `cout << '\n' << std::flush`

When the buffer becomes full, the program decides to read from `cin` or the program exits, the buffer content will also be wrriten to the output.

Cerr

There is another output stream object defined by the iostream library called `cerr`.

By convention, programs use the `cerr` stream for error messages.

This stream is identical in most respects to the `cout` stream; in particular, its default output is also the **screen**. Its output redirection can be used as `./program 2><filename>` e.g. `./program 2>test_error.txt`

File Stream

```
#include <fstream>

ifstream iFile; // inherit from istream
ofstream oFile; // inherit from ostream
iFile.open("myText.txt");  // if unsuccessful to open, iFile would be in failed state, if(iFile) returns false. But member function open() has void return type!!!


iFile >> bar;
while(getline(iFile, line)) // simple way to read in lines.
iFile.close(); // important

oFile << bar;
```

String Stream

```
#include <sstream>

istringstream iStream; // inherit from istream
iStream.str(line); // assigned a string it will read from, often used for getline
iStream >> foo >> bar;
iStream.clear(); // Sometimes you may find this useful for reusing iStream
iStream.close();

ostringstream oStream; // inherit from ostream
oStream << foo << " " << bar;
string result = oStream.str(); // method: string str() const;
```

L12: Exception

`try` & `catch`

1. the **first** catch block with the **same** type as the thrown exception object will handle the exception
2. `catch(...) {}` is the default handler that matches any exception type
3. You cannot write a catch block unless you have a try block before it
4. You can throw an exception from anywhere, instead of just within a try or catch block
5. It is always good coding styles to specify in the EFFECTS clause that when a function would throw an exception:

```
int factorial (int n);
    // EFFECTS: if n>=0, return n!; otherwise throws n
```

Backpropagation: locate the handler

1. browse through the remaining part of the function, if not found, proceed to step 2
2. browse through caller of the function, if not found, proceed to step 3
3. browse through caller of the caller... (until reach main function)

   Once found: run all the commands **in and after** the catch block **that catch the exception.**

```
void foo()
{
    try
    {
        throw 'a'; // try int and double
    }
    catch (char c)
    {
        cout << "Char " << c << " caught!" << endl;
    }
    cout << "Foo executed." << endl;
}

void bar()
{
    try
    {
        foo();
    }
    catch (int i)
    {
        cout << "Int " << i << " caught!" << endl;
    }
    cout << "Bar executed." << endl;
}

int main()
{
```

```
    try
    {
        bar();
    }
    catch (...)
    {
        std::cout << "Default catch!" << endl;
    }
    cout << "Main executed." << endl;
    return 0;
}
```

L13: Abstract Data Type

What is ADT?

An ADT provides an abstract description of values and operations.

The definition of an ADT must combine **both** some notion of **what** values that type represents, and **what** operations on values it supports.

Why we need ADT?

Abstraction hides detail and makes users' life simpler

Procedural abstraction: Function
A handy black box that do something (**what**) as specified (you don't need to worry about **how**)

Data abstraction: Abstract Data Type (ADT)
A handy object, stores data together with some specific operations

2 Advantages of ADT:

- Information hiding: same for procedural abstraction, one more thing is that the user don't know **how** the object is represented (is `IntSet` represented by array or linked list? The user don't know!)
- Encapsulation: combines both data and operation in one entity

Key Components

1. Specification
2. Member Type
3. Data Member
4. Methods

Specification

Comments to explain things

- **OVERVIEW**
  Short description of class
- **RME**
  What the function does

Member Type

Different view from user and developer (outside and inside)

- **Public**
  What users can see and use, no need to care about actual implementation
  "Insert 5 to the set! Easy!"
- **Private**
  What developers have to think of to meet users' requirement
  "The user wants to insert 5, I have to first check if 5 already exists! Ahh, and also the total number may change!"

Data Member

Variables to store information

- **Representation**
  "The user want a set, so it can be an array, or maybe a linked list."
- **Representation Invariant**
  What the variable stands for and can be taken as granted correct. And it must hold immediately before exiting each method of that implementation.
  "numElts always equals to the number of elements in the set, so size() can use this and doesn't need to count every time!"

Method

Functions to get things done

- **Maintenance**
  Constructor for initialization, Destructor for clean-up
- **Operation**
  For users to use
- **Helper**

  Useful because other methods may want to use the same functionality, **in most cases declared as `private`**

You may need to pay attention to:

- **Const Qualifier**
  const object, const member function, const this (And a const object can only call its const member functions, and a const member function can only call other const member functions)
- **Implementation**
  Detect error, get things done,
  repair invariant

An ADT Example: `IntSet`

```
const int MAXELTS = 100;
class IntSet {
    // OVERVIEW: a mutable set of integers
   public:
    IntSet();
```

```
    // MODIFIES: elts, numElts
    // EFFECTS: Default constructor
    void insert(int v);
    // MODIFIES: this
    // EFFECTS: insert v into the set
    void remove(int v);
    // MODIFIES: this
    // EFFECTS: delete v from the set
    bool query(int v) const;
    // EFFECTs: return true if v is in the set, return false otherwise
    int size() const;
    // EFFECTs: return the size of the set

   private:
    int elts[MAXELTS];
    int numElts;
    int indexOf(int v) const;
    // REQUIRES: v is in the set
    // EFFECTS: return the index of v in the set
};
```

L15: Subtypes and Inheritance

Subtype

Subtype relation is an “IS-A” relationship.

For examples, a Swan **is a** Bird, thus a class Swan is a subtype of class Bird. A bird can fly, can quake and can lay eggs. A swan can also do these. It might do these better, but as far as we are concerned, we don’t care.

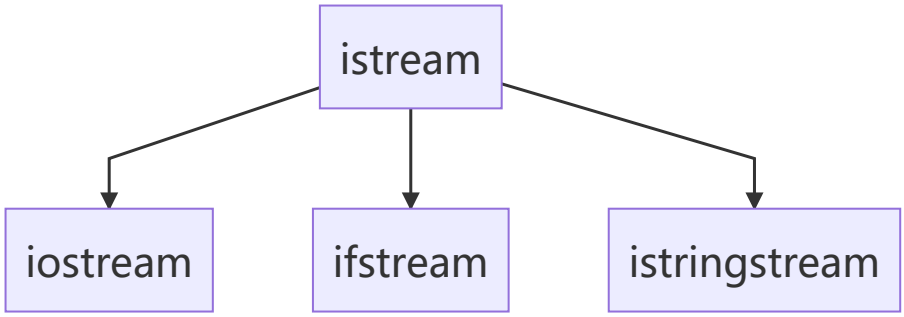
Reversely, Bird is the **supertype** of the Swan.

Substitution Principle

If S is a subtype of T or T is a supertype of S, written $S <: T$, then for any instance where an object of type T is expected, an object of type S can be supplied without changing the correctness of the original computation.

- Functions written to operate on elements of the supertype can also operate on elements of the subtype.
- Benefits: code reuse.

An example would be the input streams of c++:



Distinguish: Substitution vs. Type Convertion

Consider the following examples.

- Example 1: Substitution

  Can we use an `ifstream` where an `istream` is expected? Is there any type conversion happening in this piece of code?

  ```
  void add(istream &source) {
      double n1, n2;
      source >> n1 >> n2;
      cout << n1 + n2;
  }

  int main(){
      ifstream inFile;
      inFile.open("test.in")
      add(inFile);
      inFile.close();
  }
  ```

- Example 2: Type Convertion.

  Can we use an `int` where a `double` is expected? Is there ant type conversion happening in this piece of code?

  ```
  void add(double n1, double n2) {
      cout << n1 + n2;
  }

  int main(){
      int n1 = 1;
      int n2 = 2;
      add(n1, n2);
  }
  ```

Creating Subtypes

In an Abstract Data Type, there are three ways to create a subtype from a supertype:

1. Add operations.
2. Strengthen the postconditions of operations
    - Postconditions include:
        - The EFFECTS clause: we print an extra message to `cout` in the new class while returning the same thing
3. Weaken the preconditions of operations
    - Preconditions include:
        - The REQUIRES clause: we allow negative integers in the new class
        - The argument type

Inheritance Mechanism

When a class (called derived, child class or subclass) inherits from another class (base, parent class, or superclass), the derived class is automatically populated with almost **everything from the base class.**

- This includes member variables, functions, types, and even static members.
- The only thing that does not come along is friendship-ness (you will learn this later on).
- We will specifically discuss the constructors and destructors later.

The basic syntax of inheritance is:

```
class Derived : /* access */ Base1, Base2, ... {
private:
    /* Contents of class Derived */
public:
    /* Contents of class Derived */
};
```

Access Specifier

There are three choices of access specifiers, namely `private`, `public` and `protected`.

The accessibility of members are as follows:

specifier	private	protected	public
self	Yes	Yes	Yes
derived classes	No	Yes	Yes
outsiders	No	No	Yes

When declaring inheritance with access specifiers, the accessibility of (members that get inherited from the parent class) in the derived classes are as follows:

Inheritance \ Member	private	protected	public
private	inaccessible	private	private
protected	inaccessible	protected	protected
public	inaccessible	protected	public

When you omit the access specifier, the access specifier is assumed to be `private`, and the inheritance is assumed to be `private` as well.

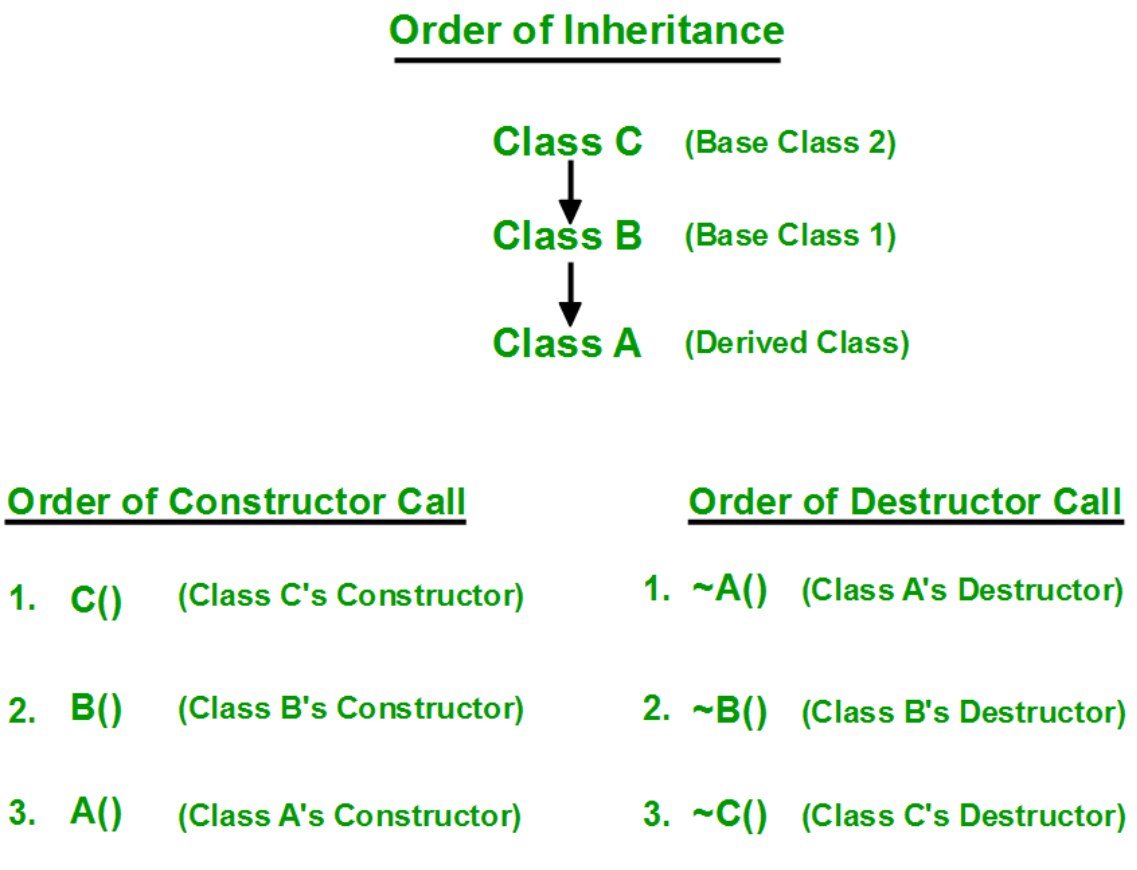
An example would be as follows. Which parts of the code does not compile? (Constructors and Destructors are omitted)

```
class X

{

public:

    int a;

protected:

    int b;

private:

    int c;

};

class Y : public X

{

    // a is public

    // b is protected

    // c is not accessible from Y

};

class Z : protected X

{

    // a is protected

    // b is protected

    // c is not accessible from Z

};

class T : private X    // 'private' is default for classes
```

```
{

    // a is private

    // b is private

    // c is not accessible from T

};
```

Constructors and Destructors in Inheritance

What would be the order of constructor and destructor call in an inheritance system? A short answer to remember would be:



Consider the following example:

```
class Parent {
public:
    Parent() { cout << "Parent::Constructor\n"; }
    virtual ~Parent() { cout << "Parent::Destructor\n"; }
};

class Child : public Parent {
public:
    Child() : Parent() { cout << "Child::Constructor\n"; }
    ~Child() override { cout << "Child::Destructor\n"; }
};
```

```
class GrandChild : public Child {
public:
    GrandChild() : Child() { cout << "GrandChild::Constructor\n"; }
    ~GrandChild() override { cout << "GrandChild::Destructor\n"; }
};

int main() {
    GrandChild gc;
}
```

The output would be:

```
Parent::Constructor
Child::Constructor
GrandChild::Constructor
GrandChild::Destructor
Child::Destructor
Parent::Destructor
```

Pointer and Reference in Inheritance

From the language perspective, C++ simply trusts the programmer that every subclass is indeed a subtype. We have the following rules.

- Derived class pointer is compatible to base class pointer, i.e., we could run `PosIntSet* p1, IntSet* p = p1`
- Derived class instance is compatible to base class reference, i.e., we could run `PosIntSet s, IntSet& p = s`
- You can assign a derived class object to a base class object, i.e., we could run `PosIntSet s, IntSet p = s`

The reverse is generally false. E.g., assigning a base class pointer to derived class pointers needs special casting.

Apparent type and Actual type

Apparent type: the declared type of the reference. (IntSet)

Actual type: the real type of the referent. (PosIntSet)

In default situation, C++ chooses the method to run based on its apparent type.

```
PosIntSet s; // Actual type: PosIntSet
IntSet& r  = s; // Apparent type: IntSet

try {
    r.insert(-1);
} catch (...){
    cout << "Exception thrown\n";
}
```

Result: we insert `-1` into `s`, and this breaks the abstraction of the set `s`, and that's why we need `virtual`.

Virtual Functions - Dynamic Polymorphism

`virtual` keyword

A way to tell C++ compiler to choose the **actual type at run-time** before execution.

Using the previous example:

```
class IntSet{
    ...
public:
    ...
    virtual void insert(int v);
    ...
};
```

The above syntax marks insert as a `virtual` function.

First, for each class with virtual functions, the compiler creates a **vtable** (or **virtual table**) with one function pointer for each virtual function initialized to the appropriate implementation. Then, each instance of a class with virtual methods has both the class' state, plus a pointer to the appropriate vtable.

Virtual methods are methods **overridable by subclasses.** When a method call is made, if the method you are calling is a virtual function (based on the apparent type), the compiler binds the method according to the **virtual table**. In this way, the function `insert` achieves dynamic polymorphism, the ability to change its behavior based on the actual type of the argument.

`override` keyword

The act of replacing a function is called overriding a base class method. The syntax is as follows.

```
class PosIntSet: public IntSet{
    ...
public:
    ...
    void insert(int v) override;
    ...
};
```

`override` cause the compiler to verify if a function is indeed overriding a base class method. If the base class method is not a virtual function, compiler will complain. The keyword is introduced in C++11. **It is considered good programming style to always mark override whenever possible.**

Comprehensive exercise for L13-L15

```
#include <iostream>
using namespace std;
class Foo {
```

```
    public:
    void f() { cout << "a"; };
    virtual void g() = 0;
    virtual void c() = 0;
};
class Bar : public Foo
{
    public:
    void f() { cout << "b"; };
    virtual void g() { cout << "c"; };
    void c() { cout << "d"; };
    virtual void h() { cout << "e"; };
};
class Baz : public Bar
{
    public:
    void f() { cout << "f"; };
    virtual void g() { cout << "g"; };
    void c() { cout << "h"; };
    void h() { cout << "i"; };
};
class Qux : public Baz
{
    public:
    void f() { cout << "j"; };
    void h() { cout << "k"; };
};
int main() {
    Bar bar;    bar.g();
    Baz baz;    baz.h();
    Qux qux;    qux.g();
    Foo &f1 = qux;
    f1.f();    f1.g();    f1.c();
    Bar &b1 = qux;
    b1.f();    b1.c();    b1.h();
    Baz &b2 = qux;
    b2.f();    b2.c();    b2.h();
    return 0;
}
```

The answer is shown below.

```
int main() {
    Bar bar;    bar.g(); // c
    Baz baz;    baz.h(); // i
    Qux qux;    qux.g(); // g
    Foo &f1 = qux;
    f1.f();    f1.g();  f1.c(); // a g h
    Bar &b1 = qux;
    b1.f();    b1.c();  b1.h(); // b h k
    Baz &b2 = qux;
    b2.f();    b2.c();  b2.h(); // f h k
    return 0;
}
```

Fun exercise for plugin

1. What is plugin?

From a simplified informal point of view, a plugin can be seen as a small piece of software that can be loaded to extend or bring in new features to a host application. For this to work, the host software must expose a plugin API (Application Programming Interface, defines how other applications can access data from this application). Then plugins can be developed independently from each others or the main application, i.e., the core software does not need them to compile and run properly. Plugins can hence be implemented following the plugin API, be compiled as shared libraries, and loaded at startup or even at run-time by the host software.

For instance when developing a music player one might want to introduce plugins to play various file types such as mp3, wav, and flac. In such a case one will want to have a generic `play_file()` function which would redirect the job to an appropriate function, for example `play_mp3()`, or `play_wav()` depending on the file type. Of course if a file type is not supported, e.g., `play_flac()` does not exist, the program should not crash but simply report that this file type is not supported. In particular this shows the necessity for each plugin to register itself and present some meta-information about itself to the main program.

2. How to use plugins in shell?

- Plugin files are files with extension .so . Such files are dynamic libraries. Dynamic libraries, often mentioned together with static libraries, will be discussed more in EECS 370, EECS/VE 482.
- When the program starts, it first scans through the folder, find all files with extension .so and load such files as plugins. If the program fails to load one of the plugin file, it will print an error message and continues on the next file.
- The program is a Read-Eval-Print Loop program. It is like a simple shell. You can also type some commands in this program. The program will execute your command and print a message.

Here are commands that you can use. For any other message you type, the program will repeat it.

```
load <.so filename>              load a plugin to the host program by its
filename
remove <plugin name>             remove a plugin from the host program by its
name
print <plugin name>              print the info of the plugin
exit                             exit the host program
```

To complie and obtain one plugin, you can use the following command. The source code and Makefile of the core applications are provided in the starter files, check for that for your understanding.

```
g++ <cpp file> -fPIC -shared -o <output_file_name.so> -Wall -Werror -std=c++11 -
I ./
```

Build with the host program by

```
make
# or
make 16
```

Execute with

```
./l6 <plugin directory>
```

And for most cases (also for this exercise), you will run with

```
./l6 plugins
```

3. Plugins provided for the exercise

Two plugins are provided and they will be loaded when the program starts.

Cat

This plugin watches for input in the format below.

```
dadsh > cat <number>
```

The number is the execution time (unit: second) of this command. Like for `cat 5` , the plugin will show a cat for 5 seconds. Its source files are in `plugins/cat` . The compiled dynamic library file is `plugins/cat.so`

Try the following commands by yourself.

```
dadsh > cat 3
# and then watch the cat for 3 seconds
dadsh > print cat
dadsh > remove cat
dadsh > cat 3
dadsh > load plugins/cat.so
dadsh > cat 3
```

Simple

This plugin watches for input in the format below.

```
dadsh > simple
```

It will print one message "1+1=2. Plugin is easy. You get it.". Its source files are in `plugins/simple`. The compiled dynamic library file is at `plugins/simple.so`. **Please make sure you understand this source file here before you start writing your code**.

Try the following commands by yourself.

```
dadsh > simple
dadsh > print simple
dadsh > remove simple
dadsh > simple
dadsh > load plugins/simple.so
dadsh > simple
```

4. TODO

Carefully read the source file `plugins/simple/simple.cpp` and `plugin.h`, understand what each part does.

Write a plugin yourself in `plugins/add/add.cpp`. Make sure that your plugin source codes are in the same directory as `plugin.h` . Namely, you should use `#include plugin.h`.

- **name**: `add`
- **author** : `<your name>`
- **description**: `add two integers`
- **help**: `usage: add <integer> <integer>`

It should output the sum of two integers.

Assume that all users follow the input format `add <integer> <integer>` and type valid integers.

When using `print add` , it should output as below.

```
Add operations is great!
```

extern "C": In C++, when used with a string, `extern` specifies that the linkage conventions of another language are being used for the declarator(s). In short, normally we cannot compile C++ source code and C source code together. But with `extern "C"` , we can first compile the C++ source code into a library or link file. Then we can compile such file with other C source code.

For matchRule, you can use `<regex>` header and the expression `"^(add)(\\s)+[0-9]+(\\s)+[0-9]+$"` to check if the input matches the requirement (usage seen in cat.cpp).

Reference

[1] Weikang, Qian. VE280 Lecture 10, 12-15.

[2] Changyuan, Qiu. VE280 Midterm Review Slides. 2020FA.

[3] Chengsong Zhang. VE280 Final Review PartI. 2021SU.

[4] VE280 Lab6 Manual. 2021SU.