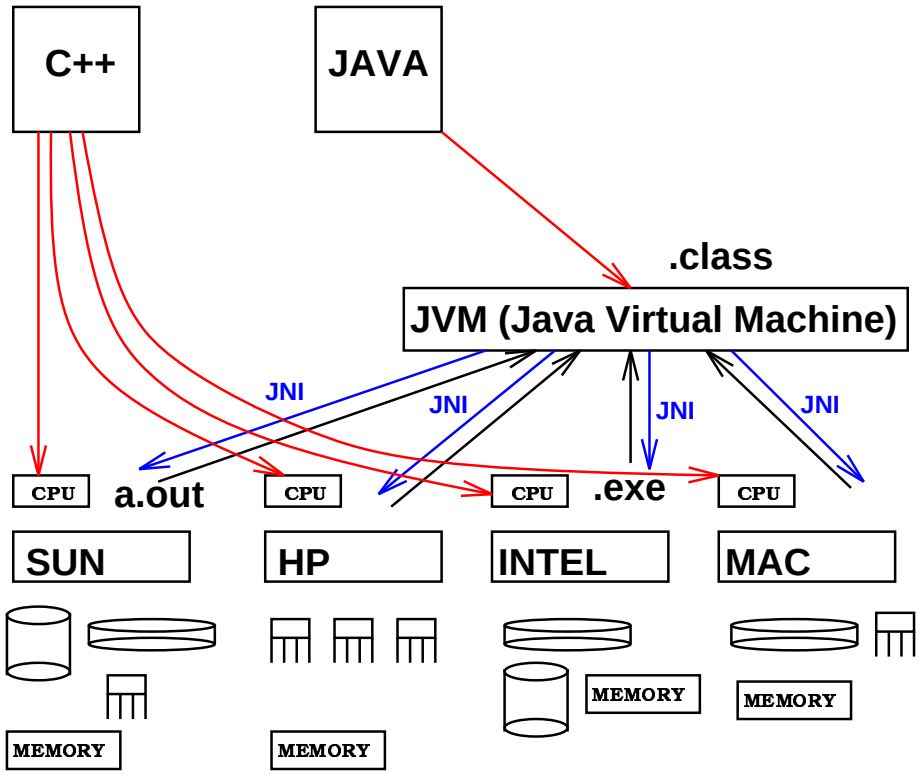
- C++ Computing & Operational Model (*v.s. Java*)
- Issues with C++
- Memory Model & Pointers
- Object Oriented Programming
- Classes & Inheritance
- Constructors, Destructor & Assignment Operator
- Operator Overloading
- Templates
- Streams

C++ Computing & Operational Model (*v.s. Java*)



Issues with C++

- portability (*even with ANSI*)
- native OS calls
- no standard libraries
- exception handling not enforced (*unlike Java*)
- device access
- explicit memory management

Memory Model & Pointers I

- arrays allocated by `new[]` and those statically declared are guaranteed to be contiguous
- access to addresses via pointers
- Storage
  - statically allocated variables in executable image
  - local variables on stack
  - dynamically allocated variables on heap

Memory Model & Pointers II

- pointers are **variables** which hold addresses
  - `ptr`: the address in the pointer
  - `&ptr`: the address of the pointer
- pointers to index arrays:

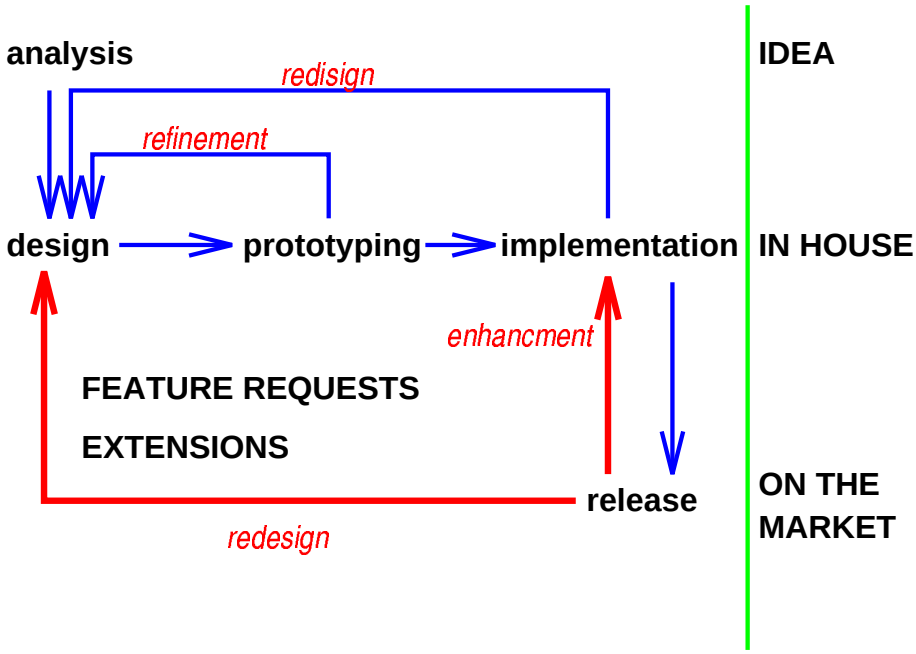
$$\texttt{ptr}[\texttt{n}] \equiv *(\texttt{ptr} + \texttt{n})$$

- dynamically allocated multi-dimensional structures

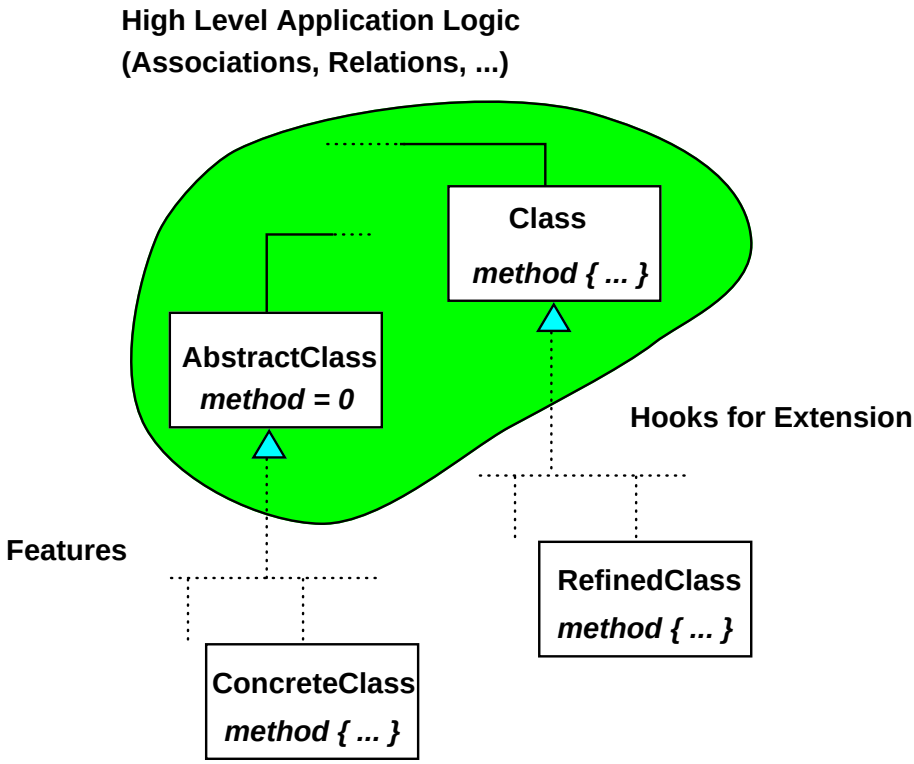
$$\texttt{ptr}[d_1]...[d_k] \equiv *(*(... * (\texttt{ptr} + d_1) + ...) + d_k)$$

- pointers can point to virtually anything:
  - primitive types (`int, char, ...`)
  - objects (*instances of classes*)
  - functions (`int (*) (int, char*)`)
  - methods (`int (A::*) (int, char*)`)
  - array elements (`&a[n]`)
- pointers are needed
  - for aliasing (*sharing memory*)
  - for run-time polymorphism to work
  - for dynamic memory allocation

Object Oriented Programming I

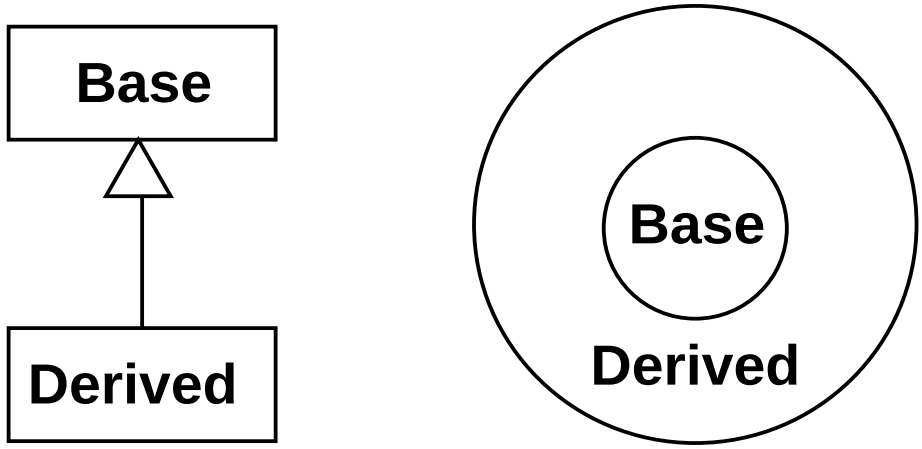


Object Oriented Programming II



Classes & Inheritance I

- instance variables: each instance has its own copy
- instance methods: have access to instance variables → needs an instance to be invoked
- all instance variables and methods inherit
- access mode
  - `private`: there but not visible in derived class
  - `protected`: like private but visible in derived class
  - `public`: the interface to the class
- methods which will be overloaded **must be** declared `virtual` in the base class
- destructors of classes which will have derived classes **must be** declared `virtual`


Base
Derived
Base
Derived

Constructors do not inherit!

Why? Suppose it inherits and we have `B` as a subclass of `A` and both have instance variables.

- if `B` does not implement the constructor, `A`'s constructor would only initialize `A`'s instance variables.
- if `B` implements the constructor, `A`'s constructor is overridden and hence `A`'s private variables **cannot** be initialized

Instead

> Constructors of the subclass first call the default constructor of the base class or another constructor explicitly specified in the initializer.

Constructors, Destructor & Assignment Operator II

```
class Base {
    int i;
  public:
    Base():i(-2) { }
    Base(int _i):i(_i) { }
};

class Derived : public Base {
    int j;
  public:
    Derived():j(-3) { }
           // Base() is implicitly called!!, i = -2
    Derived(int _j):j(_j) { }
           // Base() is implicitly called!!, i = -2
    Derived(int _i,int _j):Base(_i):j(_j) { }
           // Instead of Base() Base(int) is called
};
```

Constructors, Destructor & Assignment Operator III

The destructor does not inherit!

same reasons as the constructor: if it did, either instance variables of the derived class or the base class would not be destroyed.

The assignment operator inherits!

It is **not true** that the overloaded assignment operator calls the assignment operator of the base class!!!

```
class Base {
public:
   Base& operator=(const Base& b) { ...  return *this; }
};
class Derived: public Base {
public:
   Derived& operator=(const Derived& d) {
      Base::operator=(d); // explicit call
      ...
      return *this;
   }
};
```

Constructors, Destructor & Assignment Operator IV

What happens if not implemented?

- Constructor
  - verbatim copy of contents of instance variables using their own copy constructors
    for pointers it means aliasing!!!
  - constructors of the derived class implicitly call the default constructor of the base class, unless another specified explicitly
    if copy constructor not implemented in derived class then it calls the copy constructor of the base class!, as if it were implemented like this:

```
derived::derived(const derived& d):base(d) {
}
```

    this however calls the default constructor of `base`*!*

```
derived::derived(const derived& d) {
}
```

- Destructor
  - instance variables are destroyed by their own destructor
  - destructor of base is implicitly called
  - *if not implemented* `delete` *is not called on pointers!*

Operator Overloading

- overload it as an instance method
- overload it as a function

Usually binary operators are overloaded as functions so constants can be used on the left hand side!

```
class C {
   private:
     int x, y;
     ...
   public:
     ...
     int& operator[](int i) {
         return i<0 ? x : y;
     }
}
```

```
C operator+(const C& c1, const C& c2) {
  return C(c1.get_x()+c2.get_x(), c1.get_y() + c2.get_y());
}
```

Templates, Generic Programming

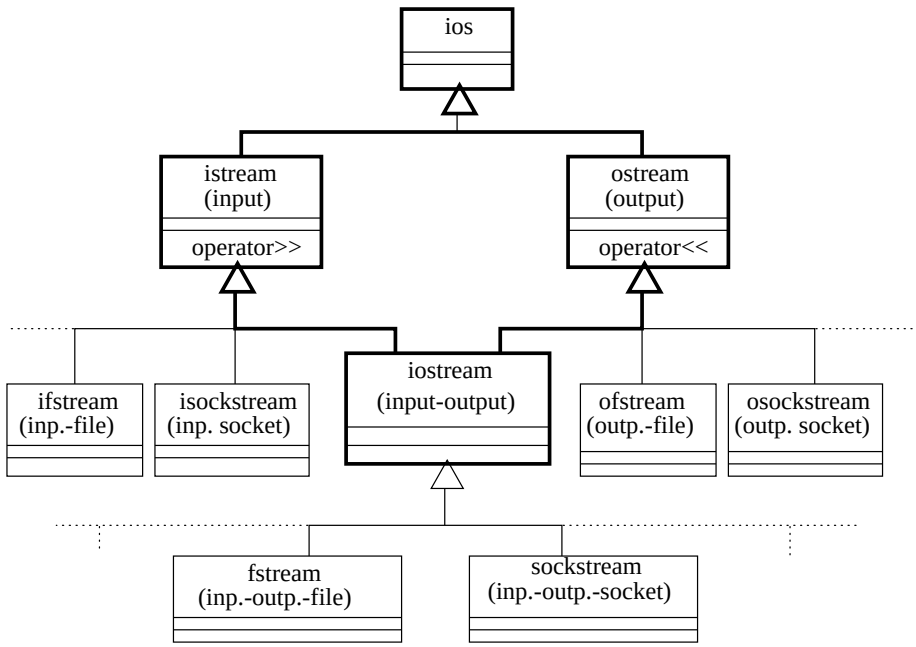
- ideal to implement container classes
- type is taken as a parameter:

```
list<int> L1; list<shape*> L2; list<double> L3;
```

- when implemented, syntax is

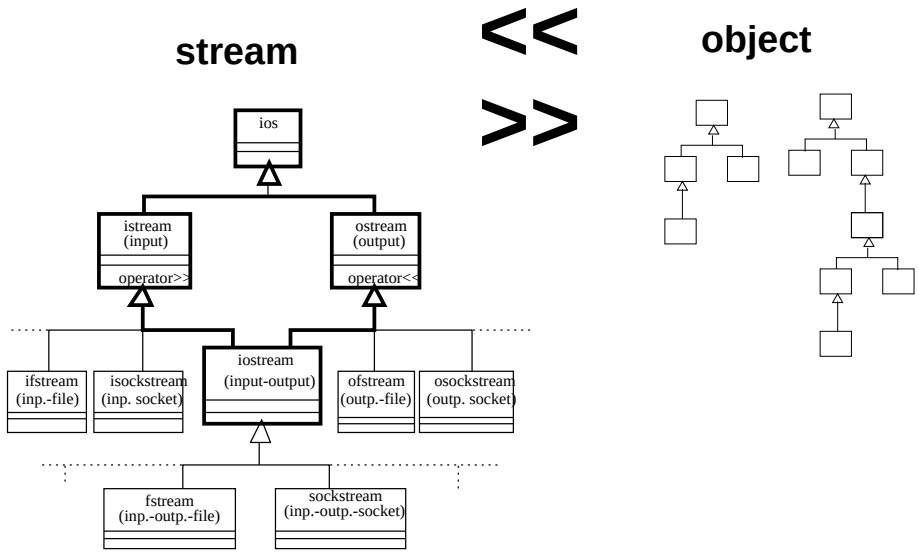
```
template<class T>
class list {

};

template<class T>
void list<T>::append(const T& e) {
   ...
}
```

Streams I


ios
istream
(input)
operator>>
ostream
(output)
operator<<
ifstream
(inp.-file)
isockstream
(inp. socket)
iostream
(input-output)
ofstream
(outp.-file)
osockstream
(outp. socket)
fstream
(inp.-outp.-file)
sockstream
(inp.-outp.-socket)

Streams II.

Serialization:



Final Exam

- closed book, 3 hours
- 5 questions
- motto: define simple classes, implement a few methods, implement assignment operator, constructor ... overload methods (*inheritance*), overload a few operators
  - implement a few methods for a simple container using templates (*much simpler than assignment # 3*)
  - implement methods and overload operators for simple classes
  - define simple classes
  - implement constructors, assignment operators and destructors (*some instance variables may be pointers to dynamically allocated memory!*)
  - output (*constructors and destructors with inheritance*)
  - model a simple problem with classes
- know the labs and your assignments!