Software Languages Lab

Structuur van Computerprogramma's 2

dr. Dirk Deridder
Dirk.Deridder@vub.ac.be
http://soft.vub.ac.be/

Vrije Universiteit Brussel - Faculty of Science and Bio-Engineering Sciences - Computer Science Department - Pleinlaan 2 - 1050 Brussels - Belgium

Software Languages.Lab

Chapter 7 - Subtypes and Inheritance

Structuur van Computerprogramma's 2

Subtypes and Inheritance

Problem Description

A circular LAN consists of **nodes**.

Nodes process **packets** that are addressed to them; and may pass other packets on to the next node.

Besides "simple nodes" there are several more sophisticated types of nodes:
e.g., **workstations** may generate new packets, **fileservers** save their packets in a file, ...

Derived Classes

- Thus a workstation **is-a** node and a fileserver **is-a** node as well
- In OO jargon, we say that the class `Workstation` and the class `Fileserver` are **subclasses** of the class `Node`:
  - they **inherit** (data and function) members from the class `Node`
- In C++ we say that the user-defined types `Workstation` and `Fileserver` are **derived from** the type `Node`
- The scope of a derived class forms a subscope of the base class (private?)

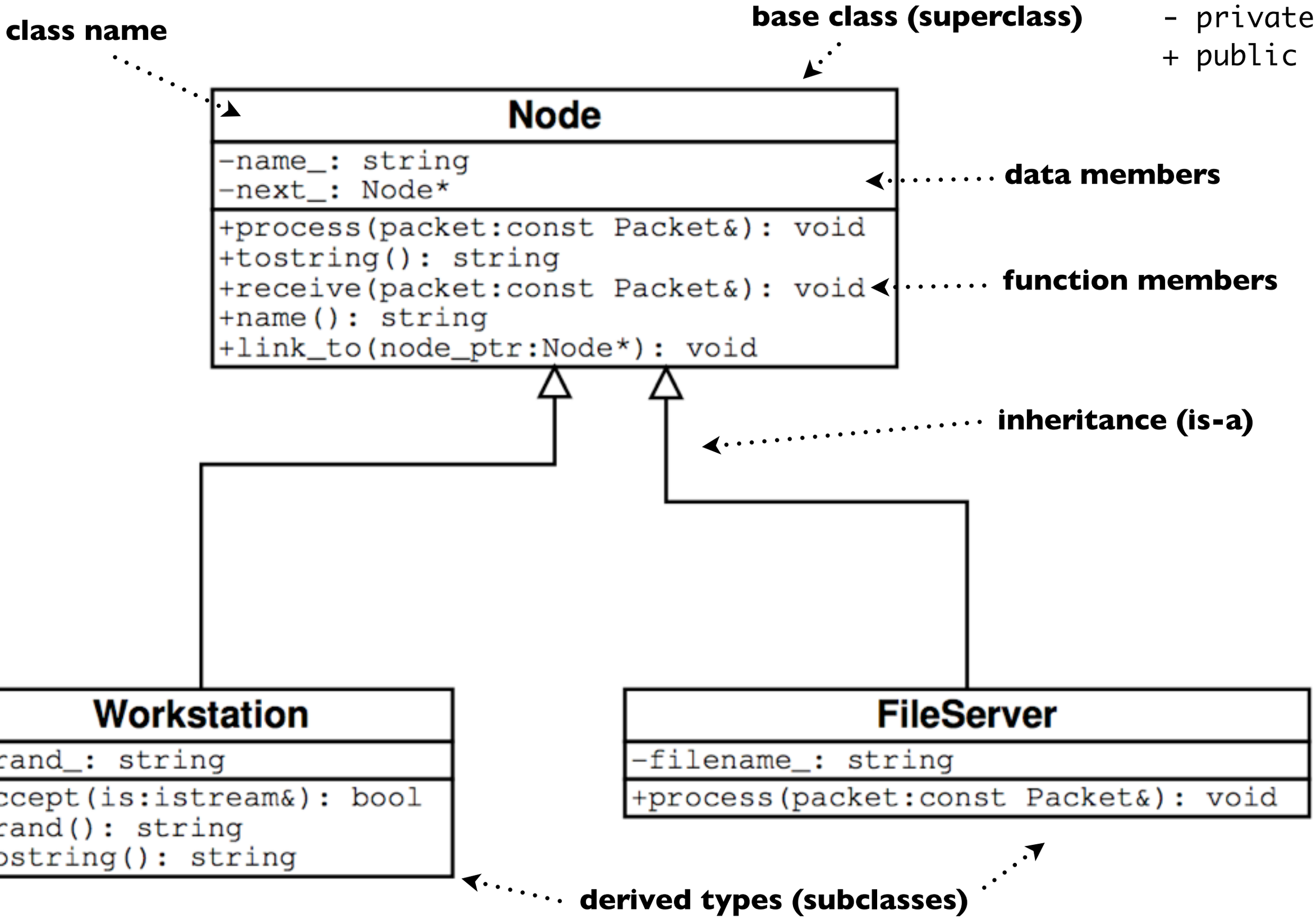
```
class Node;

// Workstation is derived from Node
class Workstation: public Node;

// Fileserver is derived from Node
class Fileserver: public Node;
```

```
class NameOfClass :
  public NameOfClass
{ ... };
```

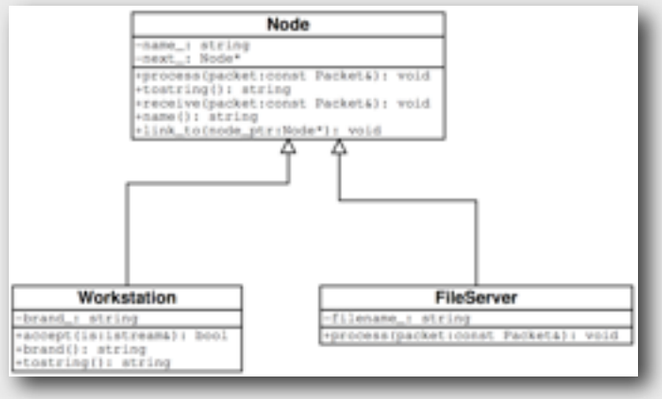
UML Class Diagram



packet.h

```
#ifndef PACKET_H
#define PACKET_H
#include <string>
#include <iostream>
class Packet { // ‘‘bare bones’’ implementation
public:
   Packet(const std::string& destination, const std::string& load) :
      destination_(destination), contents_(load) { }
   std::string destination() const { return destination_; }
   std::string contents() const    { return contents_; }
private:
   std::string destination_; // name of destination node
   std::string contents_;    // load of the packet to be passed
};
// Packets can be easily written to an ostream
inline std::ostream&
operator<<(std::ostream& os, const Packet& p) {
   return os << "[" << p.destination()
             << ", \"" << p.contents() << "\"]";
}
#endif
```

node.h




```
#ifndef NODE_H
#define NODE_H
#include <string>
#include "packet.h"
class Node {
public:
   Node(const std::string& name, Node* next = 0) :
      name_(name), next_(next) { }
// what we do with a packet we receive
   void receive(const Packet&);
// what we do with a packet destined for us
   void process(const Packet&);
   void link_to(Node* node_ptr) { next_ = node_ptr; }
   std::string name() const { return name_; }
   std::string tostring() const; // nice printable name
private:
   std::string name_;              // unique name of Node
   Node* next_;                    // in LAN (cf. list)
};
#endif
```

node.cpp

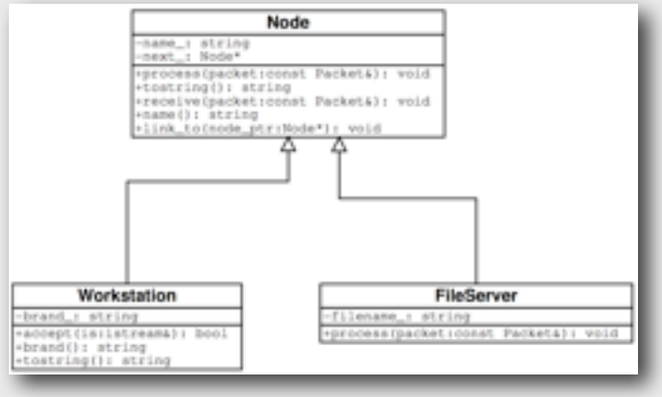
```
#include "node.h"

// what we do with a packet we receive
void Node::receive(const Packet& packet) {
   std::cout << tostring() << " receives " << packet << std::endl;
   if (packet.destination() == name())
      process(packet);
   else if (next_) // pass it on if there is a next node
      next_->receive(packet);
}

// what we do with a packet destined for us
void Node::process(const Packet& packet) {
   std::cout << tostring() << " processes " << packet << std::endl;
}


std::string Node::tostring() const {
   return "node " + name_; // explain?
}
```

workstation.h (a Specialized Node)




```
#ifndef WORKSTATION_H
#define WORKSTATION_H
#include <iostream>
#include "node.h"
// Workstation is publicly derived from Node
class Workstation: public Node {
public:
// constructor: note the use of the base class constructor
   Workstation(const std::string name, const std::string brand,
               Node* next = 0) :
      Node(name, next), brand_(brand) { }
// extra function: accept packet from cin and send it to next
   bool accept(std::istream& is);
// 'override' Node::tostring()
   std::string tostring() const;
// extra function: accessor for the brand data member
   std::string brand() const { return brand_; }
private:
   std::string brand_; // extra data member
};
#endif
```

workstation.cpp

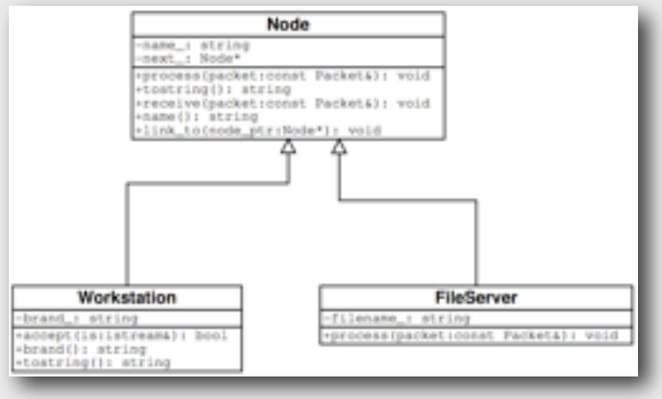
```
#include "workstation.h"

bool Workstation::accept(std::istream& is) {
// packet format: destination contents \n
   std::string destination;
   std::string contents;
   std::cout << tostring() << " accepts a packet.." << std::endl;
   is >> destination;
   std::getline(is, contents);
   if (destination.size() == 0) {
      std::cerr << "error: no destination" << std::endl;
      return false;
   }
   Packet packet(destination, contents);
   receive(packet); // ‘‘send’’ the packet to myself
   return true;
}

std::string Workstation::tostring() const {
   return Node::tostring() + " (" + brand() + ")";
}
```

We override the default version of `tostring()` for a `Workstation`

fileserver.h (another Specialized Node)

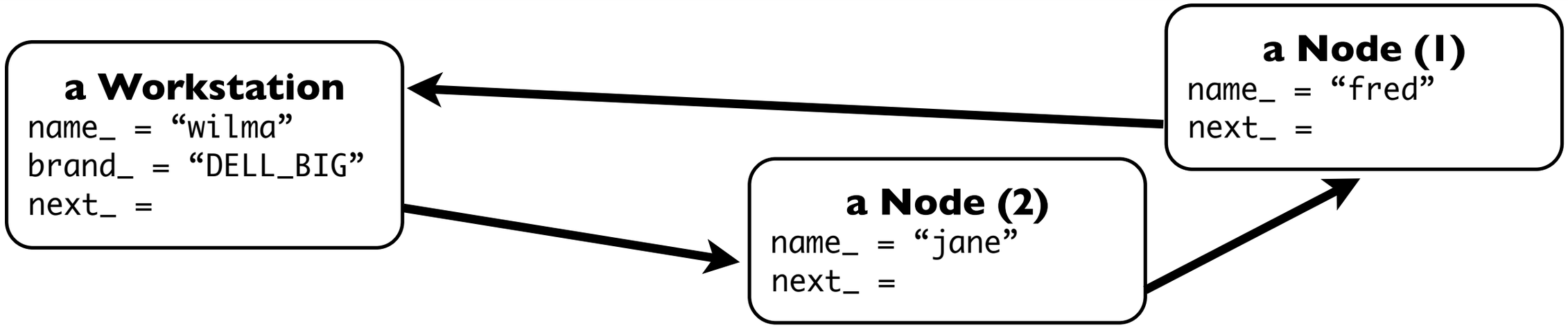



```
#ifndef FILESERVER_H
#define FILESERVER_H

#include <iostream>
#include "node.h"

// FileServer is publicly derived from Node
class FileServer: public Node {
public:
// constructor: simply delegate to base class constructor
   FileServer(const std::string name, Node* next = 0) :
      Node(name, next) { }
// override Node::process() to alter the way a Packet is processed
   void process(const Packet& packet) {
      std::cout << "FileServer " << tostring() << ": storing packet"
                << packet << std::endl;
   }
};
#endif
```

Program lan.cpp

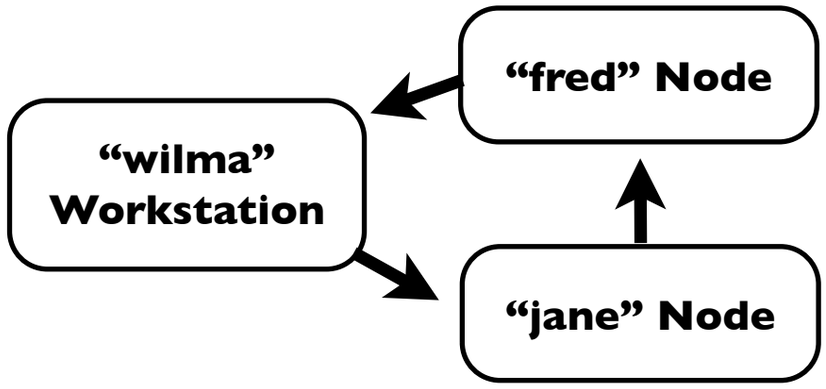


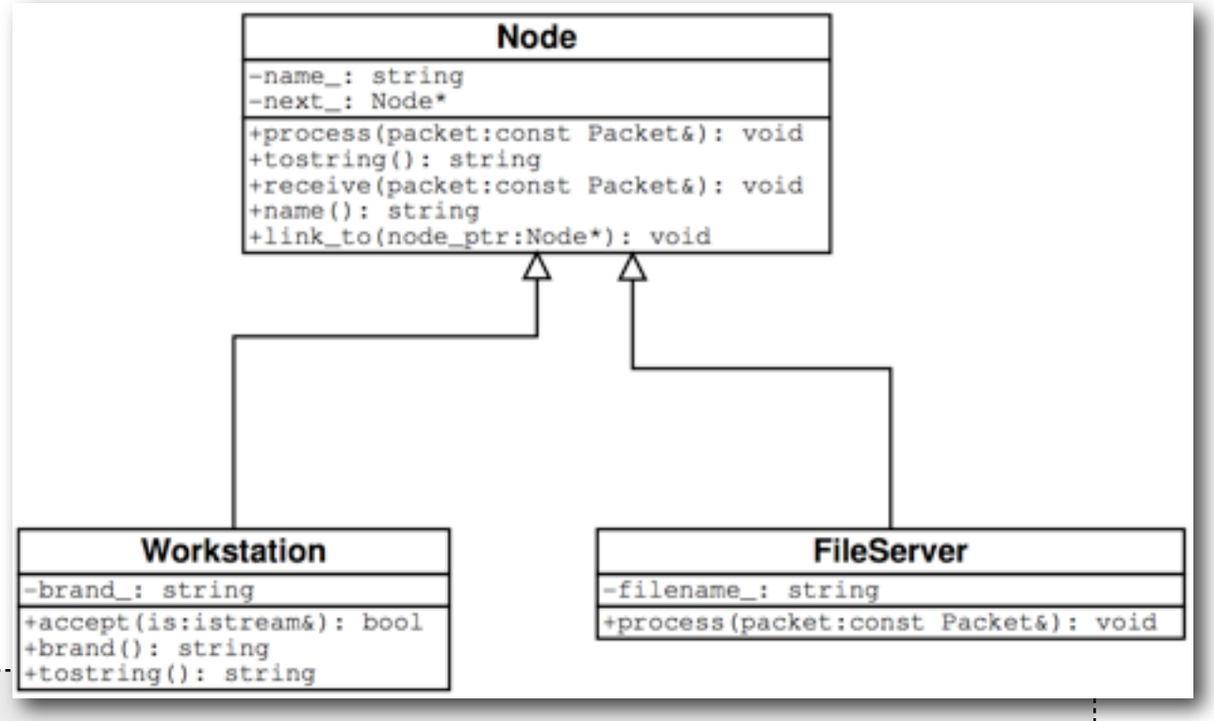

```
#include "node.h"
#include "workstation.h"

int main(int, char**) {
   Workstation wilma("wilma", "DELL_BIG");
   Node node1("fred", &wilma);     // node1 --> wilma
   Node node2("jane", &node1);     // node2 --> node1
   wilma.link_to(&node2);          // wilma --> node2
   // now we have a circular net:
   // wilma --> node2 --> node1 --> wilma

   while (wilma.accept(std::cin))
      ;
}
```

Running LAN



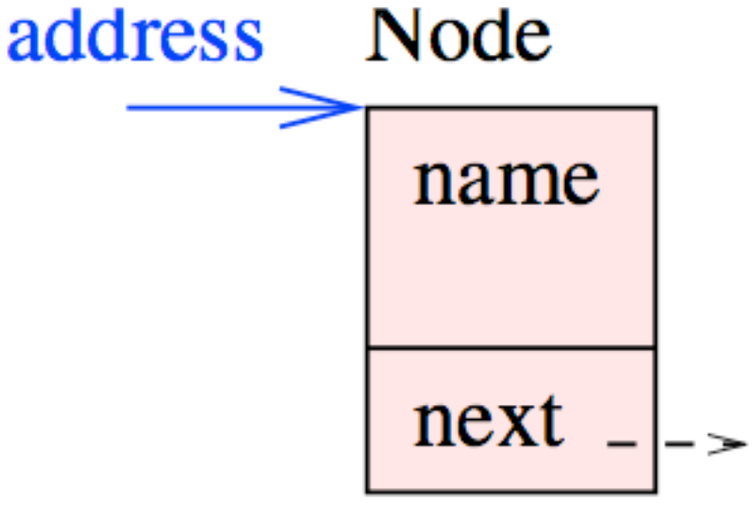


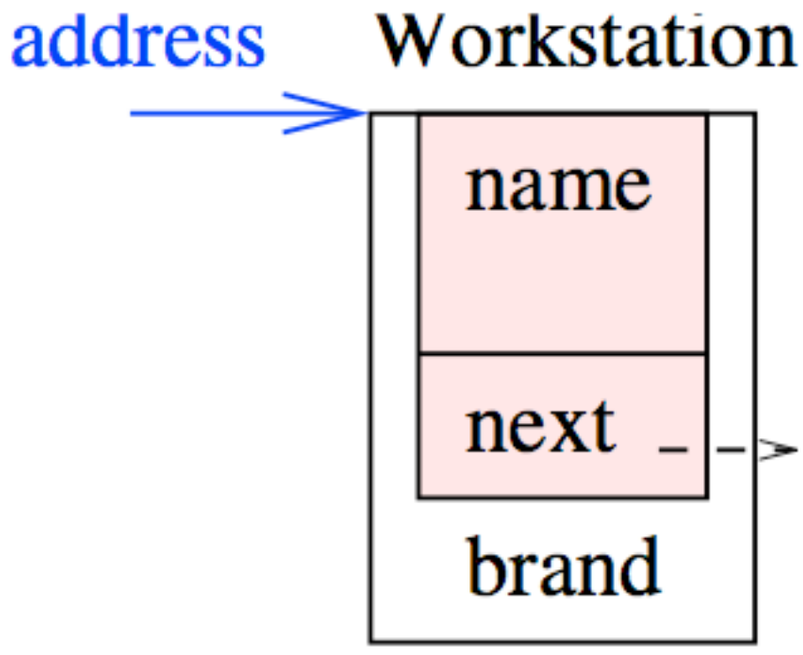

```
node wilma (DELL_BIG) accepts a packet..
fred hi there, fred
node wilma receives packet [fred, "hi there, fred"]
node jane receives packet [fred, "hi there, fred"]
node fred receives packet [fred, "hi there, fred"]
node fred processes [fred, "hi there, fred"]
node wilma (DELL_BIG) accepts a packet..
```

Explain these lines:

- node wilma (DELL_BIG) accepts a packet.
- node wilma receives packet [fred, “ hi there, fred”]

Derived Object Layout





- A **Workstation** contains a "**Node** part" which is initialized using a Node constructor
- The address of a **Workstation** is also the address of a **Node**
- The compiler can convert automatically:

```
Workstation*  →  Node*
Workstation&  →  Node&
Workstation   →  Node
```

Virtual Member Functions

Consider a pointer **nodeptr** (to a **Workstation**) with type **Node*** :

```
Workstation ws("wilma", "DELL_BIG");
Node* nodeptr(&ws); // ok, base/subclass substitutable
```

We would like **nodeptr->tostring()** to call **Workstation::tostring()**

This can be achieved by declaring **Node::tostring** to be **virtual**:

```
virtual std::string Node::tostring() const;
```

which will cause the compiler to generate code such that which function is actually called for **nodeptr->tostring()** is determined **at runtime** (**late binding**) and depends on the “**real**” class of ***nodeptr**

node2.h (virtual version)

```
#ifndef NODE_H
#define NODE_H

#include <string>
#include "packet.h"

class Node { // version with virtual functions
public:
   Node(const std::string& name, Node* next = 0) :
      name_(name), next_(next) { }
   virtual ~Node() { } // virtual destructor, see later
   void receive(const Packet&);
   virtual void process(const Packet&);  // for FileServer
   void link_to(Node* node_ptr) { next_ = node_ptr; }
   std::string name() const { return name_; }
   virtual std::string tostring() const;
private:
   std::string name_; // unique name of Node
   Node* next_;       // in LAN
};
#endif
```

Source with virtual functions

No change in `node.cpp`, `workstation.h`, `workstation.cpp`, `lan.cpp`, `packet.h`, `fileserver.h`

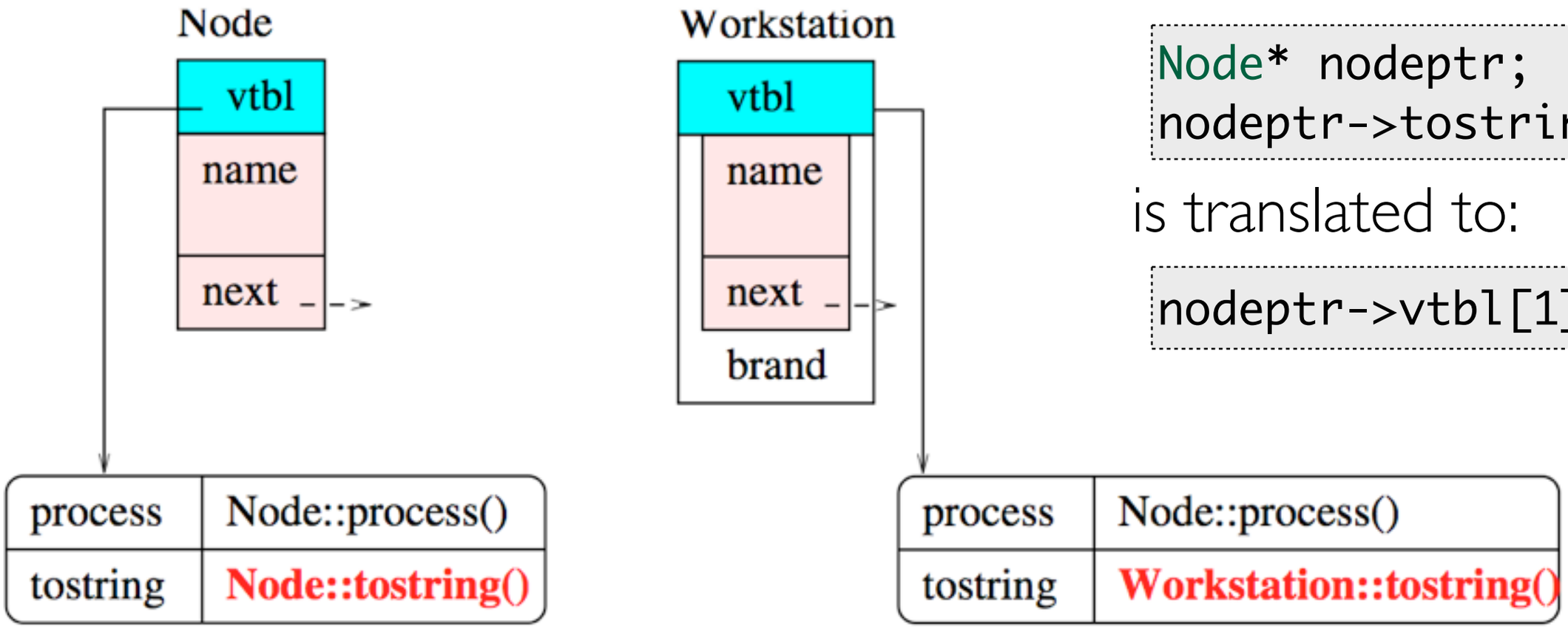
Running LAN (virtual version)

```
node wilma (DELL_BIG) accepts a packet..
fred hi there,fred
node wilma (DELL_BIG) receives packet [fred, "hi there,fred"]
node jane receives packet [fred, "hi there,fred"]
node fred receives packet [fred, "hi there,fred"]
node fred processes [fred, "hi there,fred"]
node wilma (DELL_BIG) accepts a packet..
```

Explain this line:

- `node wilma (DELL_BIG) receives packet [fred, " hi there,fred"]`

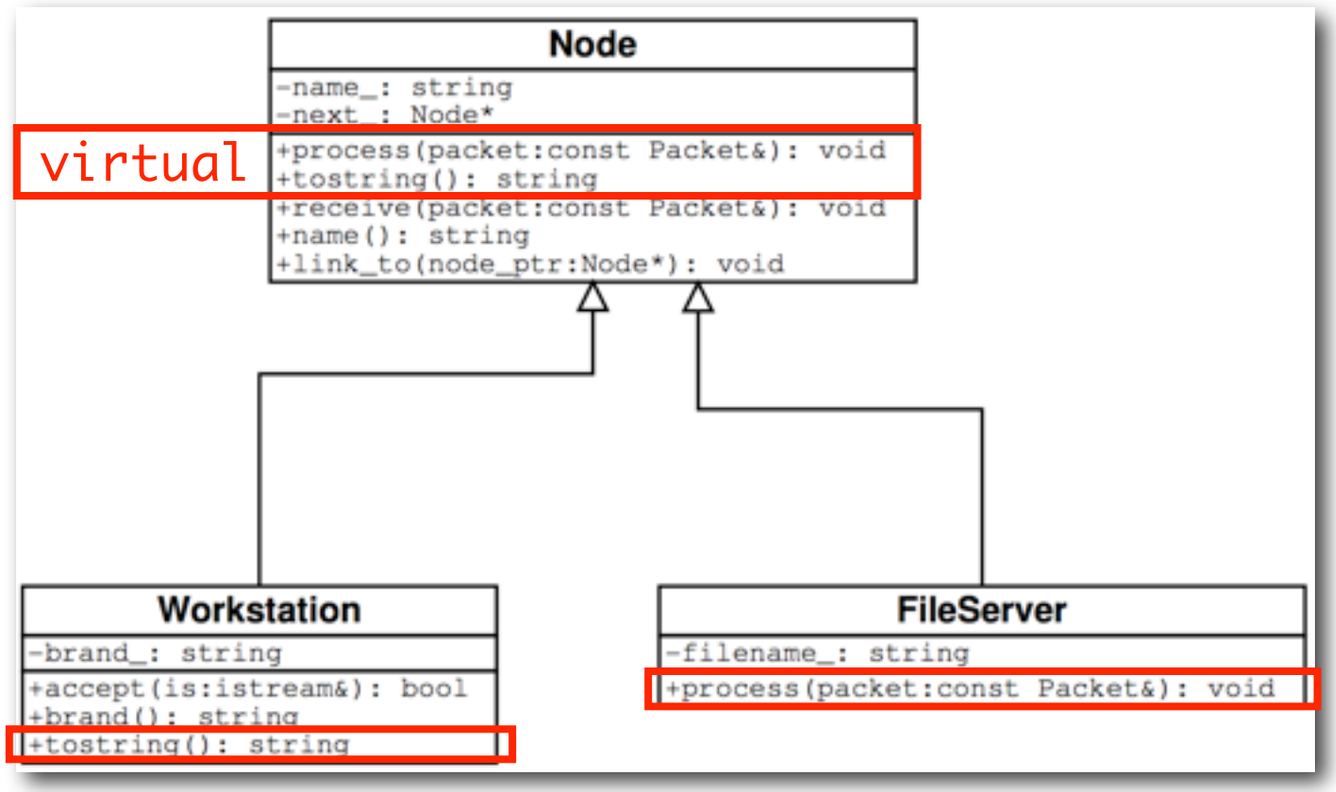
Object Layout with Virtual Functions: vtbls




```
Node* nodeptr;
nodeptr->tostring()
```

is translated to:

```
nodeptr->vtbl[1]()
```



Virtual Functions without Refs or Pointers

```
Workstation ws("wilma", "DELL");
Node node(ws);
node.tostring();
```

Which function is executed and why?

Pure Virtual Functions

```
class Animal {
public:
   virtual std::string sound() const = 0;
};

class Dog: public Animal {
   std::string sound() const { return "woef"; }
};

class Cat: public Animal {
   std::string sound() const { return "miauw"; }
};

std::set<Animal*> pets;
for (std::set<Animal*>::const_iterator a=pets.begin();
     a!=pets.end();
     ++a)
   std::cout << (*a)->sound() << std::endl;

Animal fred; // error, why?
```

Note how the specific animals 'know' how to respond to **sound()**. As a result no typechecks are needed (or a **switch** on an **enum**-''type-label'') !

Abstract Classes

Abstract classes are **specifications**; they define an **interface** that subclasses will have to implement:

```
#ifndef STACK_H
#define STACK_H
// an abstract class,
// aka interface in Java, subclassResponsibility in Smalltalk
template<typename T>
class Stack {
public:
    virtual T pop() = 0;
    virtual void push(const T&) = 0;
    virtual unsigned int size() const = 0;
// clear() only uses pure virtual functions
    void clear() {
        while (size() > 0)
            pop();
    }
    virtual ~Stack() { } // see further
};
#endif
```

Implementing Abstract Classes: a ListStack

```
#ifndef LISTSTACK_H
#define LISTSTACK_H

#include "stack.h"
#include <list>

// We implement a stack using a list; the top of the stack
// is represented by the front of the list. pop(), push(), and
// size() should be implemented as dictated by Stack
template<typename T>
class ListStack: public Stack<T> {
private:
    std::list<T> list_;
public:
    ListStack() : list_() { }
    T pop() { T t = list_.front();   list_.pop_front(); return t; }
    void push(const T& t) { list_.push_front(t); }
    unsigned int size() const { return list_.size(); }
};
#endif
```

Using ListStack

```
#include "liststack.h"

int main(int, char**) {
   Stack<int>* s = new ListStack<int> ();
   for (int i = 0; i < 10; ++i)
      s->push(i);

   while (s->size() > 0)
      std::cout << s->pop() << std::endl;
}
```

Virtual Destructors

```
class Person {
public:
   Person(const std::string& name) : name_(name) {  }
   std::string name() const { return name_; }
private:
   std::string name_;
};

class Image { // ... something *BIG*
};

class FamilyMember: public Person {
public:
   FamilyMember(const std::string& name, Image* im = 0) :
      Person(name), image_(im) { }
   ~FamilyMember() { if (image_) delete image_; }
   Image* image() const { return image_; }
private:
   Image* image_;
};
```

Virtual Destructors

```
#include "person.h"
Person* p(new FamilyMember("fred", im));
delete p; // which destructor will be called? why?
```

cf. why we needed virtual member functions in the first place

To solve it declare the destructor as virtual:

```
class Person {
public:
   Person(const std::string& name) : name_(name) { }
   virtual ~Person() { } // how will this solve the problem
   std::string name() const { return name_; }
private:
   std::string name_;
};
```

Private Derivation & Multiple Inheritance

```
#ifndef LISTSTACK_H
#define LISTSTACK_H
#include "stack.h"
#include <list>

// inheriting PRIVATELY from list<T> ensures that users of
// ListStack cannot access the underlying list<T>
// operations; this makes ListStack a properly encapsulated
// Abstract Data Type

template<typename T>
class ListStack: public Stack<T> , private std::list<T> {
public:
   ListStack() : std::list<T>() { }
   T pop() { T t = front(); pop_front(); return t; }
   void push(const T& t) { push_front(t); }
   unsigned int size() const { return std::list<T>::size();
};
#endif
```

Multiple Inheritance

Concrete implementations for pure virtual functions

Private Derivation & Multiple Inheritance

- Inherit publicly from the abstract base class (**interface**)
- Inherit privately for the **implementation**
- Keep in mind potential naming conflicts (scope resolution)!

Derivation	Access in Base	Access in Derived
public	private	none
public	protected	protected
public	public	public
protected	private	none
protected	protected	private
protected	public	protected
private	private	none
private	protected	private
private	public	private

Multiple and Virtual Inheritance

```
#include  <string>

struct Person {
   std::string name;
};

struct Staff: public virtual Person {
   int salary;
};

struct Student: public virtual Person {
   int rolnr;
};

struct Tutor: public Student, public Staff {
}; // how many name's does a Tutor have?
```

A derived class contains only a **single copy** of any **virtual base class.**

Inheritance and Operators

- All operators **except** (*why?*) constructors, destructors, and assignment are inherited
- Be careful with inherited **new** and **delete** operators: use **size_t** argument if necessary **and** virtual destructor
- What is the order of initialization for an object of a derived class?
- What if a derived class's constructor does not mention a base class constructor?
- What is the order of finalization for an object of a derived class?

Inheritance and Arrays

```
struct Base { // 4 bytes
   Base(int i = 0) : i(i) { }
   int i;
};

struct Derived: public Base { // 8 bytes
   Derived(int i = 0, int j = 0) : Base(i), j(j) { }
   int j;
};

void f(Base a[]) {
   std::cout << a[0].i << ", " << a[1].i << std::endl;
}

int main() {
   Derived da[] = { Derived(1, 2), Derived(3, 4) };
   f(da); // prints 1,2 instead of the expected 1,3
}
```