Data Abstraction and Object Orientation

Data abstraction is association of a name with a program fragment that represents information about data.

Class is data abstraction with behavior to manipulate data.

Instance of a class is an **object**.

Three key concepts

- **Inheritance** allows new abstractions to be defined as refinements or extensions to existing ones.
- **Encapsulation** enables grouping of data and subroutines that operate on them together in one place, and to hide irrelevant details from the users of an abstraction.
- **Dynamic method binding** allows a new version of an abstraction to display newly refined behavior, even when used in a context that expects an earlier version.

List Node Abstraction (1)

//C++

Encapsulation of **data members** (attributes, fields) and **subroutine members** (methods)

Visibility

- **public** - class members accessible to anybody
- **protected** - class members accessible to members of this class or derived classes (for C++, also friend classes)
- **private** – class members accessible just to members of this class (for C++, also friend classes)

Constructor as initiliazation subroutine, usually automatically invoked on object creation

```
class list_err {                                  // exception
    public:
    char *description;
    list_err (char *s) {description = s;}
};

class list_node {
    list_node* prev;
    list_node* next;
    list_node* head_node;
public:
    int val;                                      // the actual data in a node
    list_node () {                                // constructor
        prev = next = head_node = this;           // point to self
        val = 0;                                  // default value
    }
    list_node* predecessor () {
        if (prev == this || prev == head_node) return 0;
        return prev;
    }
    list_node* successor () {
        if (next == this || next == head_node) return 0;
        return next;
    }
    bool singleton () {
        return (prev == this);
    }
```

List Node Abstraction (2)

//C++ (cont.)

Destructor as finalization subroutine, automatically invoked on object destruction by either

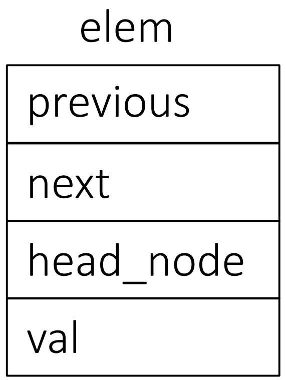
- Explicit programmer action, or
- Return from subroutine in which it was declared

```
    void insert_before (list_node* new_node) {
        if (!new_node->singleton ())
            throw new list_err ("attempt to insert node already on list");
        prev->next = new_node;
        new_node->prev = prev;
        new_node->next = this;
        prev = new_node;
        new_node->head_node = head_node;
    }
    void remove () {
        if (singleton ())
            throw new list_err ("attempt to remove node not currently on list");
        prev->next = next;
        next->prev = prev;
        prev = next = head_node = this;     // point to self
    }
    ~list_node () {                         // destructor
        if (!singleton ())
            throw new list_err ("attempt to delete node still on list");
    }
};
```

Object Creation

Static or **automatic allocation** on stack using a declaration statement

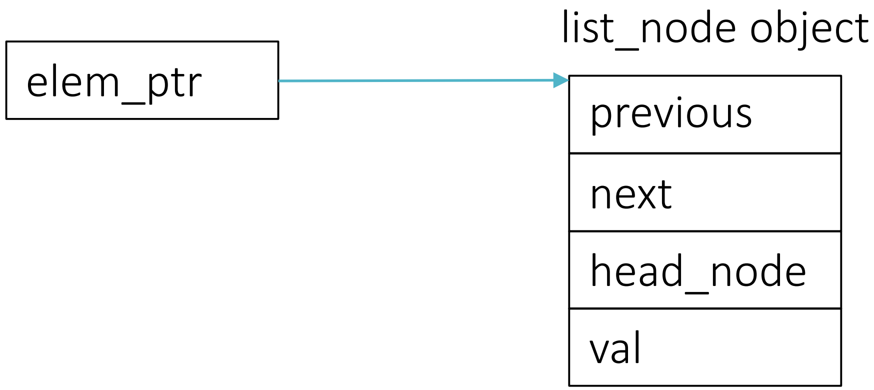
- Space is allocated when the block containing the variable creation is entered, and is released when the block is exited.




```
list_node elem;       //C++
```

Dynamic allocation on heap using an explicit operator

- Space is usually referenced by a pointer variable and is released manually, or automatically by garbage collection.




```
list_node* elem_ptr = new list_node;      //C++
```

Exercise: Object Creation and Destruction

What is printed when procedureA() is called?

```
//C++
class Trace {
public:
  Trace (string t): text(t)
         {cout << “entering “ << text << endl;}
  ~Trace () {cout << “exiting “ << text << endl; }
private:
  string text;
};
…
void procedureA () {
 Trace dummy(“procedure A”);
 cout << “processing procedure A” << endl;
}
```

……………………………………………………………………………………

Reuse by Composition

//C++

Whole-part or **has-a** relationship

```
class list {
    list_node header;
public:
    // no explicit constructor required;
    // implicit construction of 'header' suffices
    int empty () {
        return (header.singleton ());
    }
    list_node* head () {
        return header.successor ();
    }
    void append (list_node *new_node) {
        header.insert_before (new_node);
    }
    ~list () {                          // destructor
        if (!header.singleton ())
            throw new list_err ("attempt to delete non-empty list");
    }
};
```

Reuse by Inheritance

//C++

Is-a relationship

All fields and methods of the base class are inherited.

Derived class can define extra fields and methods that the base class lacks.

Derived class can redefine methods of base class.

C++ has no single root class unlike others (e.g. Object in Java and Smalltalk, object in C#)

```
class queue : public list {                    // derive from list
public:
    // no specialized constructor or destructor required
    void enqueue (list_node* new_node) {
        append (new_node);
    }
    list_node* dequeue () {
        if (empty ())
            throw new list_err ("attempt to dequeue from empty queue");
        list_node* p = head ();
        p->remove ();
        return p;
    }
    int head() {
        if (empty())
            throw new list_err("attempt to peek at head of empty queue");
        return list::head()->val;
    }
};
```

Execution Order

//C++

Calling base class constructor before derived class constructor to ensure inherited fields are in consistent state

Default base class constructor is called in this case . (Or define

`Derived::Derived(Derived_params) : Base(Base_arguments) {…}`

to specify base class constructor.)

```
class gp_list_node {
    gp_list_node* prev;
    gp_list_node* next;
    gp_list_node* head_node;
public:
    gp_list_node ();          // assume method bodies given separately
    gp_list_node* predecessor ();
    gp_list_node* successor ();
    int singleton ();
    void insert_before (gp_list_node* new_node);
    void remove ();
    ~gp_list_node ();
};

class int_list_node : public gp_list_node {
public:
    int val;                  // the actual data in a node
    int_list_node () {
        val = 0;
    }
    int_list_node (int v) {
        val = v;
    }
};
```

Dynamic Method Binding (1)

A derived class has all data and subroutines members of its base class.

An object of derived class can be allowed to use in any context that expects an object of base class.

In the code, the choice of the method to be called depend on the types of the variables x and y, or on the classes of the objects s and p to which x and y refer?

```
class person { ...
class student : public person { ...
class professor : public person { ...


student s;
professor p;

person *x = &s;
person *y = &p;


void person::print_mailing_label () { ...
...
s.print_mailing_label ();   // student::print_mailing_label (s)
p.print_mailing_label ();   // professor::print_mailing_label (p)

x->print_mailing_label ();  // ??
y->print_mailing_label ();  // ??
```

We redefine student's print_mailing_label() to include student's year.
We redefine professor's print_mailing_label() to include professor's department.

Dynamic Method Binding (2)

Static method binding if the method call is resolved at compile time. The type of the reference is used.

Dynamic method binding if the method call is resolved at run time. The class of the object to which the reference refers may be used.

Dynamic method binding imposes run time overhead to determine the type of the object referred to by the reference.

Smalltalk, Python, Ruby, and Objective-C use dynamic method binding for all methods.

Java uses dynamic method binding by default, but allows methods to be labeled final, in which case they cannot be overridden by derived classes.

C++ and C# use static method binding by default, but allow the programmer to specify dynamic method binding when desired.

Virtual Method

In C++, calls to **virtual methods** are dispatched to the appropriate implementation at run time, based on the class of the object rather than the type of the reference.

```
class person {
public:
  virtual void print_mailing_label();
  ...
  student s;
  person *x = &s;
  x->print_mailing_label();        //dynamic method binding, student's version
```

But if the method in base class is **not declared virtual**, or the method is invoked on a **statically allocated object**

```
        student s;
        person x = s;                //static allocation
        x.print_mailing_label();     //static method binding, person's version
```

Exercise: Method Binding and Reference Variable

Do the four method calls compile OK? Do they use static or dynamic binding and which version of the method is called?

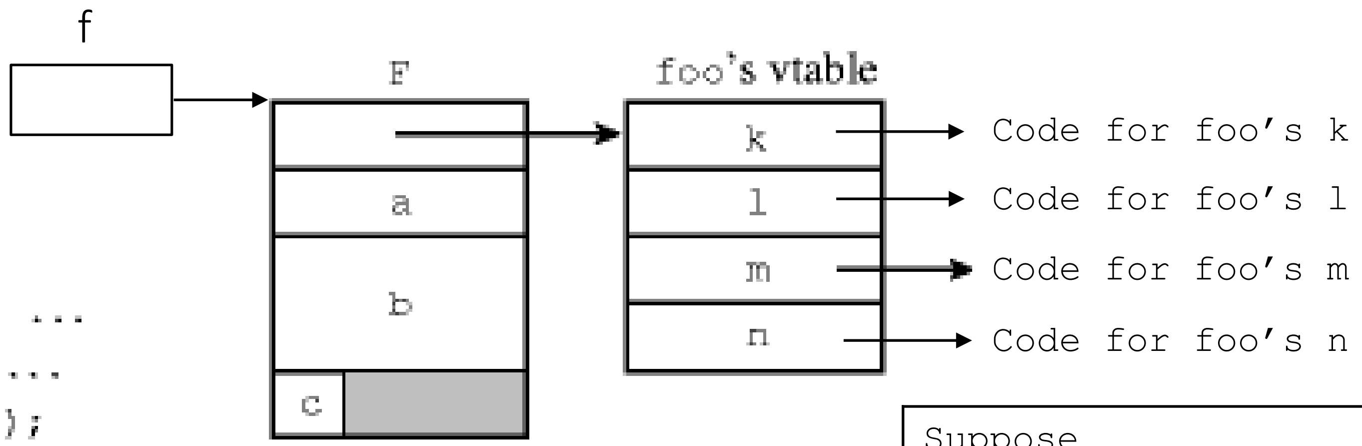
```
class person {
public:
   virtual void print_mailing_label();
   ...
  Person x;   Student s;   Person* pt;
  Person &r1 = s;                          //r1 is the reference variable of s
  r1.some_student_method();                ………………………………………………………………………………………….
  r1.print_mailing_label();                ………………………………………………………………………………………….

  Person &r2 = x;
  r2.print_mailing_label();                ………………………………………………………………………………………….

  Person* &r3 = pt;
  r3 = &s;
  pt->print_mailing_label();               ………………………………………………………………………………………….
```

Member Lookup at Run Time (1)

With dynamic method binding, the object referred to by a reference or pointer variable contains sufficient info in a **virtual method table (vtable)** for the object's class.

Each entry is the address of the code of each virtual method of the class. All objects of a given class share the same vtable.

```
class foo {
    int a;
    double b;
    char c;
public:
    virtual void k ( ...
    virtual int l ( ...
    virtual void m ();
    virtual double n( ...
    ...
} F;
```

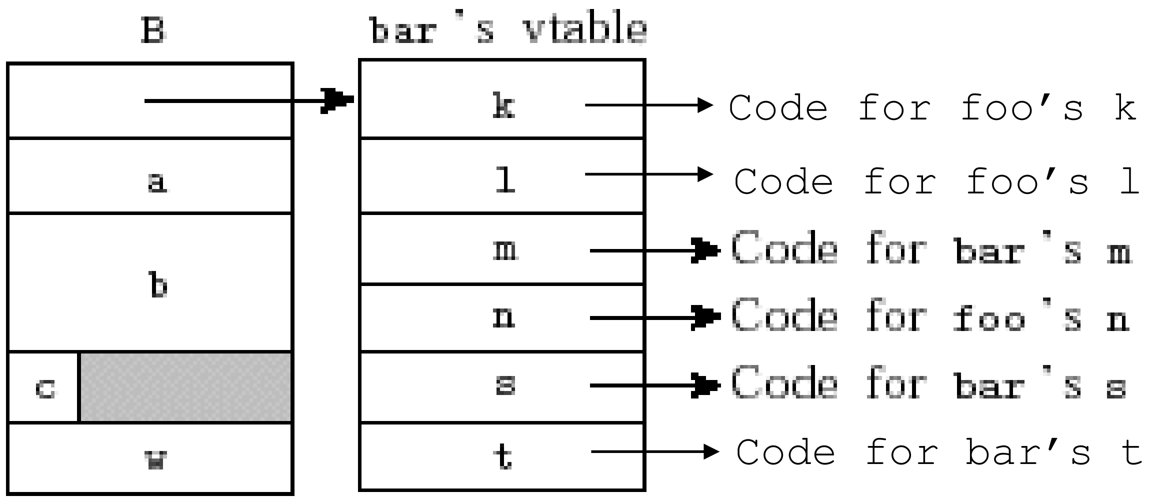



```
Suppose
foo* f; f = &F;
f->m();

r1=f
r2=*r1       --vtable addr
r2=*(r2+(3-1)x4)   --sizeof(addr)=4
call *r2
```

Member Lookup at Run Time (2)

If bar is derived from foo, copy foo’s vtable, replace overridden virtual method entries, and append bar’s virtual method entries.

```
class bar : public foo {
    int w;
public:
    void m (); //override
    virtual double s ( ...
    virtual char *t ( ...
    ...
} B;
```



Member Lookup at Run Time (3)

```
class foo { ...
class bar : public foo { ...
...
foo F;
bar B;
foo* q;
bar* s;
...
q = &B;         // ok; references through q will use prefixes
                // of B's data space and vtable
s = &F;         // static semantic error; F lacks the additional
                // data and vtable entries of a bar
```

C++ allows backward assignment which performs dynamic semantic check. Dynamic_cast is allowed only on pointers and references of polymorphic types (they have vtables).

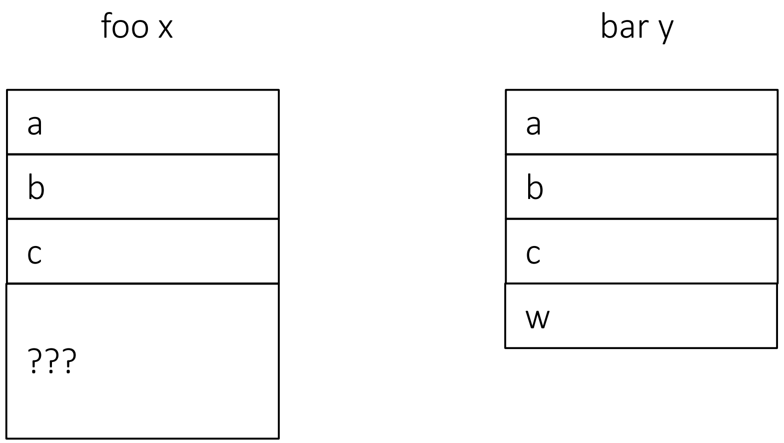
s = dynamic_cast<bar*>(q);   //run time check, s is null if failed

s = (bar*) q;   //C-style cast is permitted but no run time check

Space Allocation for Polymorphic Variable (1)

How much space to set aside for a variable of base class if it can hold a value of derived class?

```
//C++
foo x;          //foo is base class
bar y;          //bar is derived class
x = y;
```

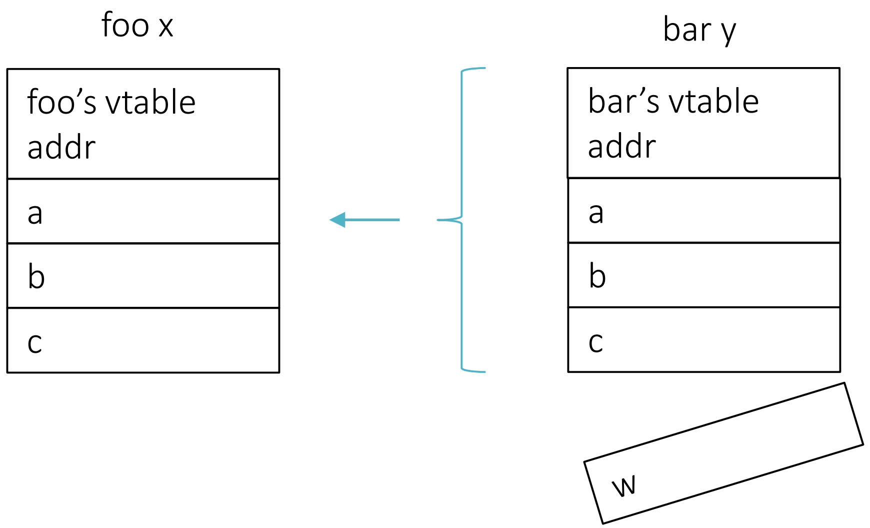


Space Allocation for Polymorphic Variable (2)

If variable is declared normally

- Use minimum static space allocation and slicing.
- Assignment changes the values of derived class into the values of base class.
- Method binding has to be static (by static class of variable), regardless of whether the method is declared virtual or not.

```
//C++
foo x;        //foo is base class
bar y;        //bar is derived class
x = y;
x.m();
```



Space Allocation for Polymorphic Variable (3)

If variable is declared as a pointer

- Method binding can be dynamic (by dynamic class of variable, if the method is declared as virtual).
- Assignment changes pointer to the dynamically allocated area of the derived class object.

```
//C++
foo* x;               //foo is base class
bar* y = new bar();   //bar is derived class
x = y;
x -> m();
```