Overloading

C++ extends overloading to constructs not usually treated as operators:

- **Indexing**: `operator [ ] ( )`
- **Selection**: `operator -> ( )`
- **Function call**: `operator ( ) ( )`
- **All assignment operators**: `+=, *=, etc.`
- `new, delete and delete[ ]`


Operator Overloading: indexing

class four_vect {
    double stor[4];  // private member, actual contents of vector.
    ...
    double& operator[ ] (int j) {
        if (j < 0 || j > 3) throw index_error; // defined elsewhere
        return stor[j];
    }
};

Note: return type is a reference, so can be used on l.h.s
Extending the meaning of subscripting

class Assoc {
    // a map from strings to numbers
    struct Pair {
        // an inner class
        char* name;
        double val;
        Pair (char* n = "", double v = 0): name(n), val(v) { }; }
        pair * contents;  // Assoc is a set of pairs
    public:
        Assoc () { }  // default constructor
        double& operator[ ] (const char *);  // map string => number
};
Efficiency vs. privacy

Linear algebra package: vector class for 4-vectors, matrix class for 4-by-4 matrices

- Vector class redefines [ ] to do check index
- Matrix class redefines [ ] to check both indices

Want to implement matrix * vector:

\[ v_1[j] = \sum m[j][k] \times v[k]; \]

- 4 range-checks for every component, all redundant
Relaxing privacy: friends

friend function can access private members of a class
a friend is declared in the class (cannot sneak in)

```cpp
class vector {
    friend vector operator* (const matrix&, const vector&);
}
```
a function can be the friend of several classes

```cpp
class matrix {
    friend vector operator* (const matrix&, const vector&);
}
```
A class can be a friend, so all its function members are.

```cpp
class quaternion { friend class matrix; …}
```

- A friend is not a class member
class matrix { vector row [4]; public: ... };

vector operator * (const matrix&m, const vector v) {
    vector res;
    for (int i = 0; i < 4; i ++) {
        res.stor[i] = 0; // stor is private data in a vector
        for (int j = 0; j < 4; j ++)
            res.stor[j] += m.row[i].stor[j] * v.stor[j]; // row is private
    }
    return res;
}
Packages and private information

In Ada, declare both types in same package. Package body has access to full declarations for both:

```ada
package linear_algebra is
  type vector is private;
  type matrix is private;
  function "*" (m : matrix; v : vector) return vector;
private
  type vector is array (1..4) of long_float;
  type matrix is array (1..4) of vector;
  -- code in body can use array representation of both.
end linear_algebra;
```
Inheritance

General mechanism for extending existing classes.

Specialization: an X is an A

A mammal is a vertebrate
- a feline is a mammal
  - a cat is a feline
  - a persian is a cat

- A persian has all the attributes of a feline, a mammal, and a vertebrate

A class hierarchy implements a sequence of Is-A
Advantages of inheritance

- **Factorization**: Common properties are grouped in a single class
- **code reuse**: Descendant class (derived class) is defined incrementally over parent.
- **incremental programming**: New derived classes can be added without changing existing parents and siblings.
- **Polymorphism**: Can describe collections of diverse objects belonging to a set of related classes
Derivation

class point {
    int x, y;

public:
    point () { x = 0; y = 0; } // default constructor
    void move (int dx, int dy) { x += dx; y += dy; }
    void display ( ); // put black dot on the screen.
}
class color_point: public point { // derived class
    int R, G, B; // in addition to hidden members x, y

public:
    color_point ( ):point ( ) {R = 50; G = 50; B = 50;}; // call parent constr.
    void lighten (int w) { R -= w; G -=w; B -= w;};
    void display ( ); // put colored dot on the screen
    // move is inherited, applies to color_points as well
};
Substitution

An object from a derived class can be used wherever an object from the base class is expected.

```c++
point* p1 = new color_point();
```
works because a descendant has all the attributes of the parent (possibly with different meanings).

```c++
color_point* wrong = new point();
// error: incomplete object
```
but p1 can be coerced (cast) to a color_point, because **it is** one.
inheritance and privacy

Private members of the parent class are not visible in (the methods of) the derived class.

```cpp
int color_point::abcisa ( ) {return x; };  // error
```

Constructor for parent is used to initialize parent data.

```cpp
Derived (…) : Parent (…) { …};  // rest of construction
```

- **Protected** members are visible in descendants, but not outside.
Polymorphism

Because of substitution rule, can have collection of various kinds of points:

```cpp
    point* figure [100];
```

To display collection, apply the appropriate display method to each:

```cpp
    point::display ()
    color_point::display ()
    blinkling_point::display ()
```

Could add discriminant to each object, to recognize its class

- Best to use virtual methods.
Virtual methods

class parent {
    virtual void change (int x) …
}
class child: public parent {
    virtual void change (int x) … // overrides parent operation
}
class grandchild: public child {
    virtual void change (int x)… // overrides child operation
}

parent* someone = … // can be any member of family
someone-> change () // the proper one for someone’s class
Dynamic dispatching

If M is virtual, the call `ptr -> m(..)` must determine the actual nature of `ptr`, and invoke the proper method.

Each class is described by a table of virtual methods (the vtable).

Each object carries a pointer to the vtable.

Each derived class inherits the table, each overriding modifies an entry in the table.

- Dynamic dispatching is an indirect call through an entry at a known position in the vtable.
Overloading and dynamic dispatching

Overloading resolution is a compile-time process that applies to an expression in a context:

\[
\text{that} = \text{this\_one} + \text{the\_other}; \quad // \text{find right numeric type}
\]

dynamic dispatching is a run-time activity that applies to a reference:

\[
\text{some\_relative}\rightarrow\text{display}(); \quad // \text{indirect call}
\text{tommy}\cdot\text{display}(); \quad // \text{static resolution}
\]

overloading resolution may fail if call is ambiguous
dynamic dispatching succeeds because derived classes have all the virtual methods of ancestors.
The profile of an overriding method

class parent {
    virtual void method (int x);
    virtual void relation (parent x);
}
class child: public parent {
    virtual void method (int x); // overrides inherited method
    virtual void method (double x); // different method
    virtual void relation (child x); // does not override parent relation
    virtual void relation (parent x); // overrides
Abstract classes

Base class may not have enough attributes to create meaningful object:

```cpp
class shape {
    point anchor;
    int color [3];
    virtual void display (); // can't write anything here
};
shape blob; // can't do anything with it
```

If methods are declared **abstract**, cannot create objects of class:

```cpp
virtual void display ( ) = 0; // must be overridden
```
A useful pattern: the factory

Need to create class hierarchy.
Need different versions of hierarchy on different platforms (e.g. Windows vs. Linux)
Need to minimize dependence on platform.
Solution: encapsulate “constructors” into an object:

```cpp
class maker {
    virtual class1* build_class1 (int) = 0;
    virtual class2* build_class2 (int) = 0;
};
```
Everything can be solved with indirection

```cpp
class win_maker: public maker {
    virtual class1* build_class1 (int){ ...};  // implementation
    virtual class2* build_class2 (int){...};  // on win XP
};
class linux_maker: public maker {
    virtual class1* build_class1 (int){ ...};  // implementation
    virtual class2* build_class2 (int){...};  // on RedHat 9
};
maker* factory = ..  // build the proper one
class1* thing1 = maker -> build_class1 (5);
```
Multiple inheritance

```cpp
class abstract_stack { // algebraic definition
  public:
    void push (int x) = 0;
  ...
};
class vector {...}; // sequential allocation
class stack: public abstract_stack, public vector {
  public:
    void push (int x) { .. }; // use vector primitives
}
```
Multiple inheritance and overloading

class A { ...
    void draw ( );
};
class B { ...
    void draw ( );
};
class C: public A, public B { ...
    void display ( ) {
        draw ( );        // ambiguous: 2 inherited methods
        A:: draw ( );    // OK
    }
};
Replicated base classes

class Root { … // some data members

class A: public Root { … // inherit data members
class B: public Root { … // inherit data members

class hybrid: public A, public B { …
    // two copies of data members of Root?

class C: virtual public Root { …
    // single copy in children
Other complexities

Additional run-time structures for dispatching:

Data members are concatenation of data members of parents
Vtabl is concatenation of Vtabl’s of parents.
Each operation needs to compute offset to find parent data members in descendant object.

Multiple inheritance best when it is interface inheritance, or when implementation is inherited from only one parent:

Java model: one parent, several interfaces.
Object-Oriented Programming in Ada

Type extensions: different syntax, same semantics
Single inheritance only
No distinguished object: call has a controlling argument
All primitive operations are virtual, but call need not be indirect if context is not dispatching
The family of types derived from T is the set $T\text{’Class}$
Type extensions

```haskell

 type Object is tagged record
    X_Coord: Float;
    Y_Coord: Float
end record;

...

 type Circle is new Object with record
    Radius: Float;
end record;

...

Obj : Object := (1.0, 1.0); -- regular aggregate
Circ1 : Circle := (Obj with Radius => 4.5); -- extension aggregate
```
Primitive operations and inheritance

All operations declared in the same package, with a parameter or return type that mentions T, are primitive operations of T

```plaintext
function "=" (Left, Right : T) return Boolean;
function Build (Parm : Integer) return T;
function Capacity (Thing : access T) return Integer;
procedure Transform (Thing : in out T);
```

- Usual inheritance and overriding rules apply:
  ```plaintext
  function "=" (Left, Right: Better_T) return Boolean;
  ```
Dynamic Dispatching

classwide
type obj_ref is access all Object’class;
procedure display_figure (Group: Obj_List) is
  Ptr : Obj_Ref := First (Group);  -- assume some iterator
  Envelope : Circle;
begin
  Envelope := Find_Boundary (Group);
  while Ptr /= null loop
    Display (Ptr.All);  -- Object’class: dispatch
    Display (Envelope);  -- Static call: display a circle
    Ptr := Next (Ptr);
  end loop;
end;
Multiple controlling parameters

Dispatching can only depend on one type: cannot have primitive operations in more than one type.

**procedure** Combine (This: Circle; That: Square);  --  Illegal

**procedure** Combine (This: Circle; That: figure’class);
--  OK, dispatches on This, other parameter is any figure.

**procedure** Combine (This, That: Figure’class);
--  OK, no dispatching
Abstract types

Same semantics as in C++: introduce root type for a class, with abstract operations:

```plaintext
type Root_Object is abstract tagged null record;
function Display (It : Root_Object) is abstract;
```

Type extension is abstract if it does not override Display

An extension of a concrete type can be abstract: make overriding of some operation abstract.