Object-Oriented Programming
Why and How to Start Doing It in Fortran, C++, and Python

Karl D. Hammond

September 11, 2020
Programming Paradigms

There are three common methods in computer programming

• Procedural programming
  • Based on functions, subroutines, arguments
  • Focus is on the sequence of operations

• Modular programming
  • Based on collections of subroutines, functions
  • Emphasis on organizing like things into manageable groups

• Object-oriented programming
  • Based on collections of data
  • Subroutines and functions are associated with the data they are intended to operate upon

Nearly all modern programs are combinations of these three paradigms, and it is virtually impossible to disentangle them.
History of Programming

1822  Charles Babbage proposes the “difference machine”
1837  Babbage proposes the “analytical engine,” the first general-purpose computer
1842  A. Ada King, Countess of Lovelace (“Ada Lovelace”) writes in her notes the first computer program, intended for Babbage’s analytical engine

1843–1946  The ENIAC, the first electronic general-purpose computer, is constructed at U. Penn for the U.S. Army.

1953  An estimated 100 computers exist world-wide

1954–1957  The first compiler and the first high-level language, FORTRAN (“FORmula TRANslation”), developed by John Backus’s team at IBM

1959–1961  COBOL is developed by Grace Hopper (US Navy) and others to write financial software

1958  ALGOL (“ALGOrithmic Language”) is written (predecessor to Pascal, Simula, BCPL, B, C)

1960  Lisp, a language for “symbolic computation” (think Mathematica) is developed at MIT

1962  The first computer game, Spacewar!, is developed at MIT

1962  Simula, the first object-oriented language, is introduced

1965  BASIC is developed by Dartmouth College to teach programming
1967 The first language standard is published, FORTRAN 66

1967 Simula 67

1967–1971 Pascal written by Niklaus Wirth (ETH Zürich) to teach structured programming

1970 UNIX is developed at AT&T Bell Labs

1969–1973 Dennis Ritchie of Bell Labs develops C
  • C is a descendant of the B programming language
  • B is a descendant BCPL (Basic Combined Programming Language)
  • BCPL is derived from CPL, which was (loosely) derived from ALGOL (which was loosely derived from FORTRAN)
  • C was designed to write operating system components

1976 Modula (first modular language) developed by Niklaus Wirth

1976 The Apple I personal computer is created by Steve Wozniak; it was merely a motherboard

1977 AWK is written by Aho, Weinberger, and Kernighan at Bell Labs; the Bourne shell (sh) is written by Stephen Bourne of the same

1977–1985 Modula 2 developed by Niklaus Wirth
History of Computing (...continued)

1978  The FORTRAN 77 standard is released
1979  Ada is released to standardize U.S. Defense Department programming
1981  The IBM 5150, the first IBM personal computer, is released
1984  C++ is developed by Bjarne Stroustrup at Bell Labs to bring object-oriented programming to C
1989  The C language is standardized (C89, or ANSI C)
1991  Python 1.0 released by Guido van Rossum at CWI (Netherlands), with the goal of emphasizing readability and conciseness
1992  A major revision of FORTRAN, called Fortran 90, is released; most vestiges of the punched card method of program input are deprecated
1995  Java is developed by James Gosling at Sun Microsystems to write platform-independent software (such as Web applications)
1997  The C++ language is standardized (C++97)
1999  A minor revision of the C language standard, dubbed C99, is published; features include inline functions, variable-length arrays, and single-line comments
2000  Python 2.0 released
2003  C++03, a minor C++ standard revision, is published; all changes are relevant for implementers only
2004  Fortran 2003 standard brings object-oriented features and C interoperability to Fortran programs
2008  Python 3.0 released. This is a major update to the language, which is no longer backward-compatible with Python 2. Some Python 3 features are back-ported to Python 2, and Python 2 is later frozen except for bug fixes.
2010  A minor update to the Fortran standard, Fortran 2008, is approved; most updates are aimed at parallel computing.
2011  A minor revision of the C language standard, dubbed C11, is published; main addition is a memory model to support threaded execution better
2011  C++11 is approved
2018  Fortran 2018 standard published (a minor revision)
2014  C++14 is approved
2017  C++17 is approved
2020  Python 2.7.18 is released as the last Python 2 update; no further development of Python 2 is planned.
What is Object-Oriented Programming?

- Idea started with Simula (1962), a superset of ALGOL 60, which introduced the ideas of classes, inheritance, subclasses, virtual procedures, and coroutines; also featured garbage collection.

- Thought is to create a collection of data that all talk about the same thing, all of which are operated on by a set of procedures designed to manipulate that particular set of data.

- Most OOP languages involve type-bound procedures or bound methods, which are subroutines or functions intended to operate on the data to which they are bound.

- Example: a data structure called fruit that is intended to be “eaten” by some subroutine, which we will call eat, and which can be “thrown in the trash” by some other subroutine, which we will call trash. Fruit can also be “thrown” via the subroutine toss.

- The advantage of OOP is that the data bound to the object are automatically available to all of the object’s bound procedures, without passing them as arguments.
Advantages and Disadvantages

Advantages of object-oriented programming include:
- Encapsulation: you can add data to the object without modifying existing function/subroutine calls and without modifying any code outside the object.
- “global” variables are less necessary—data are passed as a group.
- Argument lists can be one or two long instead of dozens.
- Can prevent accidental/unanticipated modification of objects by an unwary user.
- Can declare new objects that are similar to old ones without re-programming all subroutines that act on them.

Disadvantages of object-oriented programming include:
- Not immediately obvious to new programmers.
- Can create lots of overhead (memory requirements).
- Can become an obsession (i.e., it’s not always the best paradigm!)
- Requires significant planning to implement: must have an overall “code design” to implement.
OOP Concepts

**Class** Descriptions of data structures, how they are organized, and the definitions of the procedures that act upon those data.

**Object** Instances of classes (the data themselves, not the description).

**Composition** A class whose members are themselves objects (collections of data/procedures).

**Inheritance** A class whose definition starts with that of another class, with all of its procedures; those procedures not re-defined by the subclass are inherited from the parent.

**Extensibility** One class extended to form another, with the subclass adding data, procedures, or both to do things the base class does not do.

**Polymorphism** An object that can be interpreted as several different classes, all of which are extensions of a base class. For example, a pointer to an object of class `Shape` can point to an object of class `Circle` as defined above.

**Encapsulation** Binding data and the methods for manipulating them behind an interface, preventing “private” data or methods from being accessed outside the class definition.
Objects’ Data

An object’s members, both data and functions, are accessed by the member operator, which varies by language:

**C++**  object.member
  If `object` is a pointer to the object rather than the object itself, you need to dereference the pointer, which can be either
  `(*object).member`
  OR
  `object->member` *(note: object -> member also works)*

**Python**  object.member*(object . member also works)*

**Fortran**  object%member *(note: object % member works, too)*
Simple Example (C++): Compare the following

```cpp
#include <cstdio>
#include <cmath>
using namespace std;

enum {CIRCLE, SQUARE, RECTANGLE};
double area (int type, double a, double b) {
    switch (type) {
    case CIRCLE :
        printf("%f
", M_PI * a * a); break;
    case SQUARE :
        printf("%f
", a * a); break;
    case RECTANGLE :
        printf("%f
", a * b); break;
    default :
        fprintf(stderr, "I don't know how to print " "that shape.
");
        return NAN;
    }
}

int main() {
    double r = 3.0, a = 4.0, a1 = 4.0, a2 = 3.0;
    printf("%f
", area(CIRCLE, r, 0.0));
    printf("%f
", area(SQUARE, a, 0.0));
    printf("%f
", area(RECTANGLE, a1, a2));
    return 0;
}
```

Use this `area` function to compare the area of different shapes. The function `area` takes three parameters: `type`, `a`, and `b`. The `type` parameter is an integer that represents the type of shape (CIRCLE, SQUARE, or RECTANGLE). The `a` and `b` parameters represent the dimensions of the shape.

Here is another version of the same function, but this time it uses classes and virtual functions to achieve the same goal.

```cpp
#include <cstdio>
#include <cmath>
using namespace std;

// These would typically be in a header (for modularity)
struct Shape { Shape (double, double); virtual double area () { return -1.0; } virtual double perimeter () { return -1.0; } };

class Circle : Shape {
    double r;
public:
    Circle (double);
    double area ();
    double perimeter ();
};

class Rectangle : Shape {
    double a, b;
public:
    Rectangle (double, double); // constructor
    double area ();
    double perimeter ();
};

struct Square : Rectangle {
    Square (double);
};

// These would be in another .cpp file
Circle::Circle (double r) {
    this->r = r;
}

double Circle::area () {
    return M_PI * r * r;
}
double Circle::perimeter () {
    return 2.0 * M_PI * r;
}

Rectangle::Rectangle (double a, double b) :
    this->a = a, this->b = b;

double Rectangle::area () {
    return a * b;
}
double Rectangle::perimeter () {
    return 2*a + 2*b;
}

Square::Square (double a) :
    Rectangle (a, a) {};

int main () {
    struct Circle circle(3.0);
    struct Square square(4.0);
    struct Rectangle rect(4.0, 3.0);
    printf("%f
", circle.area());
    printf("%f
", square.area());
    printf("%f
", rect.area());
    return 0;
}
```
Polymorphism

Try this without OOP

```c
int main() {
    double r = 3.0, a = 4.0,
    a1 = 4.0, a2 = 3.0;
    for (int i = 0; i < 3; i++) {
        switch (i) {
            case CIRCLE :
                print_area (i, r, r);
                break;
            case SQUARE :
                print_area (i, a, a);
                break;
            case RECTANGLE :
                print_area (i, a1, a2);
                break;
        }
    }
    return 0;
}
```

```c
int main() {
    struct Circle circle(3.0);
    struct Square square(4.0);
    struct Rectangle rect(4.0,3.0);
    struct Shape *shape[3];
    shape[0] = &circle;
    shape[1] = &square;
    shape[2] = &rect;
    for (int i = 0; i < 3; i++)
        shape[i]->print_area();
    return 0;
}
```
#include <cstdio>
using namespace std;

struct Shape {
  double a, b;
  Shape (double, double);
virtual double area () { return 0.0; }
virtual double perimeter () { return 0.0; }
void print_area ();
void print_perimeter ();
};

struct Rectangle : Shape {
  Rectangle (double, double);
  double area ();
  double perimeter ();
};

struct Square : Rectangle {
  Square (double);
};

Shape::Shape (double A, double B) {a = A; b = B;}
void Shape::print_area () { printf("%f\n", area()); }
void Shape::print_perimeter () {
  printf("%f\n", perimeter());
}

Rectangle::Rectangle (double a, double b) : Shape (a, b) {}
double Rectangle::area () { return a*b;}
double Rectangle::perimeter () { return 2.0*a+2.0*b;}

Square::Square (double a) : Rectangle (a, a) {}

int main () {
  Rectangle myrect(3.0,4.0);
  Square mysquare(4.0);
  myrect.print_area();
  mysquare.print_area();
  return 0;
}

Notes:

- Rectangle::print_area and Square::print_area are inherited from Shape::print_area; same for ::print_perimeter
- Square::area and Square::perimeter are inherited from Rectangle::area and Rectangle::perimeter
OOP in Fortran

- Fortran (then FORTRAN) was first written in 1954 to replace Assembly and related low-level languages. Programs were a few hundred lines long at most and had to be input on punched cards.
- 1966: FORTRAN 66, the first language standard (for any language) is published. This allowed various vendors to agree on features that had to work the same way on any vendor’s compiler.
- 1977: FORTRAN 77 introduces character arrays and proper strings, allows lower-case letters in strings (but not key words!); this is the FORTRAN most people mean when they make fun of it.
- 1992: Fortran 90 standard is published. Introduces free-form code (fixed form was from the punch card era), introduces end do in place of continue and other features to improve code structure and readability, creates pointers and allocatable arrays, and many other features. Introduces modules and releated features.
- 1997: Minor update with Fortran 95 adds vectorization features, pure and elemental procedures; some features made obsolescent
- 2004: Fortran 2003 standard adds object-oriented features, procedure pointers, C interoperability, and other features
- 2010: Fortran 2008 is approved; adds submodules, coarrays, and several features intended for multiprocessor programs
- 2018: Fortran 2018 (formerly 2015) introduces further C interoperability and parallel features
Our Shape class in Fortran

```fortran
module geometry
  implicit none
  private
  public :: geometric_shape, circle, square, rectangle
  real, parameter :: pi = acos(-1.0)

  type :: geometric_shape
    double precision :: a, b
  contains
    procedure :: area => default_area
    procedure :: perimeter => default_perimeter
  end type geometric_shape

  type, extends(geometric_shape) :: circle
  contains
    procedure :: area => circle_area
    procedure :: perimeter => circumference
  end type circle

  interface circle
    procedure :: new_circle
  end interface circle

  type, extends(geometric_shape) :: rectangle
  contains
    procedure :: area => rectangle_area
    procedure :: perimeter => rectangle_perimeter
  end type rectangle

  interface square
    procedure :: new_square
  end interface square

  contains

  function new_circle (r) result (newcircle)
    real, intent(in) :: r
    type(circle) :: newcircle
    newcircle%a = r
    newcircle%b = 0.0
  end function new_circle

  function new_square (a) result (newsquare)
    real, intent(in) :: a
    type(square) :: newsquare
    newsquare%a = a
    newsquare%b = a
  end function new_square

  real function default_area (self) result (area)
    class(geometric_shape), intent(in) :: self
    area = -1.0
  end function default_area
```

Our Shape class in Fortran (continued)

real function default_perimeter (self) result(perimeter)
   class (geometric_shape), intent(in) :: self
   perimeter = -1.0
end function default_perimeter

real function circle_area (self) result (area)
   class (Circle), intent(in) :: self
   area = pi * self%a**2
end function circle_area

real function circumference (self)
   class (Circle), intent(in) :: self
   real, parameter :: pi = acos(-1.0)
   circumference = 2.0 * pi * self%a
end function circumference

real function rectangle_area (self) result (area)
   class (Rectangle), intent(in) :: self
   area = self%a**2
end function rectangle_area

real function rectangle_perimeter (self) result (perimeter)
   class (Rectangle), intent(in) :: self
   perimeter = self%a * self%b
end function rectangle_perimeter

end module geometry

program oopdemo
   use geometry
   implicit none

type :: shape_array
   class (geometric_shape), pointer :: ptr
end type shape_array

type (circle), target :: mycircle

type (square), target :: mysquare

type (rectangle), target :: myrectangle

type (shape_array), dimension(3) :: myshapearray

integer :: i

mycircle = Circle(3.0)
mysquare = Square(4.0)
myrectangle = Rectangle(4.0,3.0)

print *, mycircle%area()
print *, myrectangle%area()
print *, mysquare%area()

myshapearray(1)%ptr => mycircle
myshapearray(2)%ptr => myrectangle
myshapearray(3)%ptr => mysquare

do i = 1, 3
   print *, myshapearray(i)%ptr%area()
end do
end program oopdemo
Python treats *everything* as an object.¹ For example, the number 3 has numerous procedures bound to it:

```python
>>> dir(3)
['_abs__', '_add__', '_and__', '_bool__', '_ceil__', '_class__', '_delattr__',
'_dir__', '_divmod__', '_doc__', '_eq__', '_float__', '_floor__', '_floordiv__',
'_format__', '_ge__', '_getattribute__', '_getnewargs__', '_gt__', '_hash__',
'_index__', '_init__', '_init_subclass__', '_int__', '_invert__', '_le__',
'_lshift__', '_lt__', '_mod__', '_mul__', '_ne__', '_neg__', '_new__', '_or__',
'_pos__', '_pow__', '_radd__', '_rand__', '_rdivmod__', '_reduce__', '_reduce_ex__',
'_repr__', '_rfloordiv__', '_rlshift__', '_rmod__', '_rmul__', '_ror__',
'_round__', '_rpow__', '_rrshift__', '_rshift__', '_rsub__', '_rtruediv__',
'_rxor__', '_setattr__', '_sizeof__', '_str__', '_sub__',['_subclasshook__',
'_truediv__', '_trunc__', '_xor__', 'as_integer_ratio', 'bit_length', 'conjugate',
'denominator', 'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
```

¹Ultra-feminists: I’m not talking about that kind of object.
Integer Objects in Python

So what do all of those do? Some implement intrinsic procedures; for example, 
(-3).__abs__() and abs(-3) are equivalent. dir(3) and (3).__dir__() are equivalent. 
Others implement basic operations; (3).__mod__(2) is the same as 3 % 2
Why, you ask? (1) It’s just how Python is, and (2) The methods can be redefined by a class that 
inherits the base class.
You can extend base classes pretty easily; a stupid example:

class Whole (int) :
    def __init__ (self, val) :
        super().__init__()
        self.stupid = True

a = Whole(4)
b = 4
print ("a is", a, "; b is", b)
print ("type(a) is", type(a), "; type(b) is", type(b))
print ("isinstance(a,int) is", isinstance(a,int),
      "; isinstance(b,int) is", isinstance(b,int))
print ("hasattr(a,'stupid') is", hasattr(a,'stupid'),
      "; hasattr(b,'stupid') is", hasattr(b,'stupid'))
class Shape () :
    def area (self) :
        return None
    def perimeter (self) :
        return None

class Rectangle (Shape) :
    def __init__ (self, a, b) :
        self.a = a
        self.b = b
    def area (self) :
        return self.a * self.b
    def perimeter (self) :
        return 2*(self.a + self.b)

class Square (Rectangle) :
    def __init__ (self, a) :
        super().__init__(self, a, a)

from math import pi
class Circle (Shape) :
    def __init__ (self, r) :
        self.a = r
    def area (self) :
        return pi*self.a**2
    def perimeter (self) :
        return 2.0*pi*self.a

shapes = (Circle(3.0), Square(4.0), Rectangle(4.0,3.0))
for shape in shapes :
    print (shape.area())
    print (shape.perimeter())
OOP Can Simplify Interfaces and Function Calls

```fortran
logical function is_bigger (shape1, a1, b1, nedge1, &
nvert1, shape2, a2, b2, nedge2, nvert2)
select case (shape1)
  case (0) :
    select case (shape2)
      case (0) :
        is_bigger = ( circle_area(shape1, a1, b1) &>
                      circle_area(shape2, a2, b2) )
      case (1,2) :
        is_bigger = ( circle_area(shape1, a1, b1) &>
                      rectangle_area(shape2, a2, b2) )
      case default
        write (ERROR_UNIT,'(A)') 'ERROR: Unknown shape2'
  case (1,2) :
    select case (shape2)
      case (0) :
        is_bigger = ( rectangle_area(shape1, a1, b1) &>
                      circle_area(shape2, a2, b2) )
      case (1,2) :
        is_bigger = ( rectangle_area(shape1, a1, b1) &>
                      rectangle_area(shape2, a2, b2) )
      case default
        write (ERROR_UNIT,'(A)') 'ERROR: Unknown shape2'
  end select
end function is_bigger
```

Note that the above could actually be bound to the type itself if earlier in the module we added one line. That would allow us to do something like this:

```fortran
if ( mysquare%is_bigger_than(mycircle) )
call do_something (mysquare)
```

Questions to ponder:

1. What would it take to add another shape—say, a triangle—to this program?
2. Would I have to change the function is_bigger in either case?
3. Which would be easier to maintain?
Comments

There are fundamental differences between how C++, Fortran, and Python handle objects, data types, and so forth

- Python is weakly-typed, meaning things like polymorphism are simple, but accidentally reassigning the wrong variable name to something is very easy to do
- C++ and Fortran require strong typing, and polymorphism must be explicit
  - C++ handles this via type casts and pointers, with polymorphic types being resolved by the dereferencing operator
  - Note that in C++, obj.ptr accesses the member ptr, which we’ll assume is a pointer to another object; the object it points to can be accessed via either *(obj.ptr) or obj->ptr
  - Fortran automatically dereferences pointers (you must explicitly ask for the address of a pointer), so obj%ptr points to the object ptr points to; things pointed to must be targets
  - Fortran uses the class key word to set off polymorphic objects, which are typical in function calls and similar places where inheritance is likely
- OOP is not inherently better or worse than other paradigms; it is on the programmer(s) to make the structure evident and maintainable
class LinkedList {
    public:
        int item;
        class LinkedList *prev;
        class LinkedList *next;
    LinkedList (int);
    LinkedList *append (LinkedList*);
    LinkedList *prepend (LinkedList*);
};

LinkedList::LinkedList (int i) {
    item = i;
    prev = NULL;
    next = NULL;
}

LinkedList* LinkedList::append (LinkedList *link) {
    next = link;
    link->prev = this;
    return next;
}

LinkedList* LinkedList::prepend (LinkedList *link) {
    prev = link;
    link->next = this;
    return prev;
}

int main () {
    LinkedList list(1);
    LinkedList *li = &list;
    li = li->append(new LinkedList(3));
    li = li->append(new LinkedList(5));
    li = li->append(new LinkedList(-3));
    li = &list;
    while (li != NULL) {
        printf("%d\n", li->item);
        li = li->next;
    }
    return 0;
}
How LAMMPS Is Organized

Basic structure

class LAMMPS {
public:
    class Memory *memory; // memory allocation functions
    class Error *error; // error handling
    class Universe *universe; // universe of processors
    class Input *input; // input script processing
    class Atom *atom; // atom-based quantities
    class Update *update; // integrators/minimizers
    class Neighbor *neighbor; // neighbor lists
    class Comm *comm; // inter-processor communication
    class Domain *domain; // simulation box
    class Force *force; // inter-particle forces
    class Modify *modify; // fixes and computes
    class Group *group; // groups of atoms
    class Output *output; // thermo/dump/restart
    class Timer *timer; // CPU timing info

    MPI_Comm world; // MPI communicator
    FILE *infile; // infile
    FILE *screen; // screen output
    FILE *logfile; // logfile

    double initclock; // wall clock at instantiation

    char *suffix,*suffix2; // suffixes to add to style names
    int suffix_enable; // 1 if suffixes enabled, 0 if disabled
    char *exename; // pointer to argv[0]
    char ***packargs; // arguments for cmdline pkg commands
    int num_package; // number of cmdline pkg commands
    int cite_enable; // 1 if generating log.cite, 0 else

    LAMMPS(int, char **, MPI_Comm);
    ~LAMMPS();
    void create();
    void post_create();
    void init();
    void destroy();
    void print_config(FILE *);

private:
    struct package_styles_lists *pkg_lists;
    void init_pkg_lists();
    void help();
    LAMMPS(){}; // can't use default constructor
    LAMMPS(const LAMMPS &){}; // can't use copy constructor
};
How LAMMPS Is Organized

Basic structure

class Atom : protected Pointers {
public:
    char *atom_style;
    class AtomVec *avec;

    // atom counts
    bigint natoms; // total atoms in system, could be 0
    // natoms may not be current if atoms lost
    int nlocal,nghost; // # of owned+ghost atoms on this proc
    int nmax; // max # of owned+ghost on this proc
    int tag_enable; // 0/1 if atom ID tags are defined
    int molecular; // 0 = atomic, 1 = standard molecule,
    // 2 = molecule template system
    bigint nellipsoids; // number of ellipsoids
    bigint nlines; // number of lines
    bigint ntris; // number of triangles
    bigint nbodies; // number of bodies
    bigint nbonds,nangles,ndihedrals,nimpropers;
    int ntypes,nbondtypes,nangletypes,ndihedraltypes,
    nimpropertypes;
    int bond_per_atom,angle_per_atom,dihedral_per_atom,
    improper_per_atom;
    int extra_bond_per_atom,extra_angle_per_atom;
    int extra_dihedral_per_atom,extra_improper_per_atom;

    // per-atom arrays
    // customize by adding new array
    tagint *tag;
    int *type,*mask;
    imageint *image;
    double **x,**v,**f;
    tagint *molecule;
    int *molindex,*molatom;
    double *q,**mu;
    double **omega,**angmom,**torque;
    double *radius,*rmass;
    int *ellipsoid,*line,*tri,*body;
...

    // per-type arrays
    double *mass;
    int *mass_setflag;
...

    // functions
    Atom(class LAMMPS *);
    ~Atom();
    void settings(class Atom *);
    void create_avec(const char *, int, char **, int);
    virtual class AtomVec *new_avec(const char *, int, int &);
    void init();
...