Reference

Stroustrup, Bjarne: "What is Object-Oriented Programming?", IEEE Software, Vol 5, No. 3, May 1988.

Slides: Courtesy of Bjarne Stroustrup, AT&T Bell Laboratories.

Benefits of Object-Oriented Programming

All the benefits of data abstraction.

Explicit representation of concepts and relations between concepts.

Greater modularity.

The ability to manipilulate objects of different, but similar, types through a single standard interface.

The ability to add new types to a system without modifying existing code.

Object-Oriented Programming Example

The problem:

> design a graphics system so that shapes can be manipulated without knowledge of exactly what kind of shape is manipulated.

Example:

```
// rotate all members of vector "v" of size "size"
//"angle" degrees

void rotate_all(shape* v[], int size, int angle)
{
    for (int i = 0; i < size; i++) v[i]->rotate(angle);
}
```

Object-Oriented Programming Example

```
class shape {
    point center;   // location
    color col;
    // ...
public:
    point where() { return center; }
    void move(point to) { center = to; draw(); }
    virtual void draw();
    virtual void rotate(int);
    // ...
};

class circle : public shape { //  circle is a shape
    int radius;
public:
    void draw(); // draw a circle - code elsewhere
    void rotate(int) {}      // yes, the null function
};

class triangle : public shape {// triangle is a shape
    point corner1; point corner2; //center is corner3
public:
    void draw(); //draw a triangle - code elsewhere
    void rotate(int); //rotate a triangle
};
```

Object-Oriented Programming Example

```
class shape {
    point center;
    color col;
    // ...
public:
    point where() { return center; }
    void move(point to) { center = to; draw(); }
    virtual void draw();
    virtual void rotate(int);
    // ...
};
```

Object-Oriented Programming Paradigm

Design:

Decide which classes you want; provide a full set of operations for each class; make commonality explicit using inheritence.

Key language features:

Mechanisms for defining new types,
data hiding mechanisms,
inheritance mechanisms,
access mechanisms,

Languages:

C++, Simula, Smalltalk

Summary of Problems with Data Abstraction

Must modify existing code, so the programmer

needs access
needs understanding
needs re-testing

No specific shape types, implying

large functions
no compile time checking
no help from tools

Problems With Data Abstraction

```
void shape::rotate(int a)
{
    switch (shape_type) {
    case circle:
            break;
    case triangle:
            // draw a triangle
            ...
    case square:
            ...
    }
}
```

Problems With Data Abstraction

```
// Data abstraction style solution:

class shape {
    point center;
    color col;
    kind shape_type;
    // ...
public:
    point where() { return center; }
    void move(point to) { center = to; draw(); }
    void draw();
    void rotate(int);
    // ...
};
```

Problems With Data Abstraction

The problem:

> design a graphics system so that shapes can be manipulated without knowledge of exactly what kind of shape is manipulated.

Example:

```
// rotate all members of vector "v" of size "size"
//"angle" degrees

void rotate_all(shape* v[], int size, int angle)
{
    for (int i = 0; i < size; i++) v[i]->rotate(angle);
}
```

Benefits of Data Abstraction

Allows the designer to work directly with application specific concepts.

Provides standard set of "natural" operations for users.

Enables change in implementation with out affecting users.

Eases debugging and maintanence by localizing information (and errors).

Data Abstraction Example

```
class char_stack {
      int size;
      char* top;
      char* s;
public:

      char_stack(int sz);
      ~char_stack();
      void push(char c);
      char pop();
};


char_stack s1(200);
void f(int x)
{
      char_stack s1(x);
      char_stack s2(x);

      s2.push('P');
      s1.push('C');
      char C = s1.pop();
      char P = s2.pop();
}
```

Fake Types

```
typedef int stack_id;

extern stack_id create_stack(int size);
extern destroy_stack(stack_id);

extern void push(stack_id, char);
extern char pop(stack_id);
```

Data Abstraction Paradigm

Design:

Decide which types you want;
provide a full set of operations
for each type

Key language features:

Mechanisms for defining new types,
data hiding and access mechanisms.

Languages:

Ada, Clu

Data Hiding Paradigm

```
#include "stack.h"

void some_function()
{
    push('c');
    char c = pop();
    if (c != 'c') error("impossible");
}
```

Data Hiding Paradigm

```
// declaration of the interface of module
//stack of characters
char pop();
void push(char);
const stack_size = 100;


#include "stack.h"
//"static" means local to this file/module
static char v[stack_size];
static char* p = v;   // the stack is initially empty

char pop()
{
    // check for underflow and pop
}

void push(char c)
{
    // check for overflow and push
}
```

Data Hiding Paradigm

Design:

Decide which modules you want; partition the program so that data is hidden in modules.

Languages:

Modula-2

Procedural Programming Example

C++:

```
double sqrt(double);     // declaration of sqrt

double sqrt(double d)    // definition of sqrt
{
    ...
}


double sqrt(double);
...
sqrt();          // compile time error
sqrt(2); // correct: 2 coerced to 2.0
sqrt("asdf");// compile time error
```

Procedural Programming Paradigm

Design:

Decide which procedures you want;
use the best algorithms you can find.

Key language features:

Procedures, functions, argument passing mechanisms, returning mechanism

Languages:

Algol, C, Fortran, Pascal, PL/1

Programming Paradigms

What is a paradigm?

A programming language supports a paradigm if programs can be written using that paradigm

without exceptional skill
without exceptional effort

A programming language can support more than one paradigm.

What Should Programs Reflect ?

The program should reflect the concepts of the application as directly as possible

Engineering:

complex, matrix, polynomial

Telephone Switching:

line, trunk, switch, digit_buffer

Graphics:

shape, circle, triangle, floor_plan

This is True

It is possible to write

truly awful

Object-Oriented Programs

C++ :

is a Better C

supports Data Abstraction

supports Object-Oriented Programming

Programming is more than Just Language Issues

Education

Methods Libraries

C++

Tools Language

Support

Typical C++ Application Areas

computer aided design
data base management
image processing
operating systems
networks
simulation
vlsi design
compilers
graphics
music systhesis
programming environment
robotics
switching

C++ is Designed for Both:

Management of Complexity (from Simula):

- classes
- hierarchies of classes
- strong (static) type checking

Efficiency (from C):

- run-time
- space
- access to hardware
- access to system resources

Origins of C++

Problem:

Event-driven simulator for software on distributed hardware

Simula version:

elegant
relatively easy to write,
easy to debug,
prohibitively slow

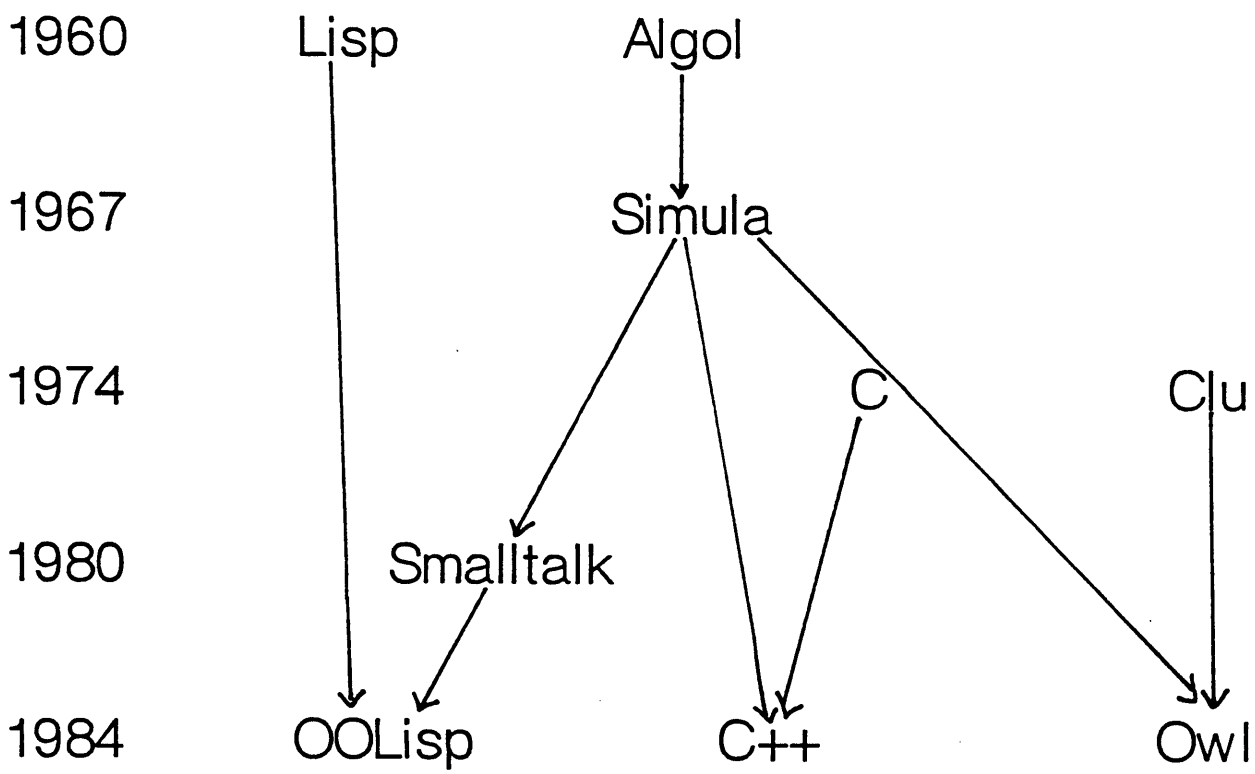
BCPL version:

ugly
hard to write,
very hard to debug,
fast

C++ ideal:

as elegant as Simula and
as fast as BCPL

Origins of Object-Oriented Programming Languages



Contents

Programming and Programming Languages

Programming Paradigms
- Procedural
- Data Hiding (Modules)
- Data Abstraction
- Object-Oriented Programming

Appendix A

Object-Oriented Programming in C++

Alternate Introduction