Object oriented Programming

This lecture covers the paper

What is "Object-Oriented Programming"?
by
Bjarne Stroustrup

It is available via

`http://www.cl.cam.ac.uk/Teaching`
`/1998/CompPrLs/bjarne.ps`

or

`http://cm.bell-labs.com/cm/cs/cstr.html`

Language Design Philosophy

- All features must be cleanly and elegantly integrated into the language.
- It must be possible to use features in combination to achieve solutions that would otherwise have requied extra separate features.
- There should be as few spurious and "special purpose" features as possible.
- A feature should be such that its implematation does not impose significant overheads on programs that do not require it.
- A user need only know about the subset of the language explicitly used to write a program.

Programming Paradigms

Procedural Programming

```
double sqrt(double arg) {
  // code for calculating a square root
}

void some_function() {
  double root2 = sqrt(2);
  // ...
}
```

Data Hiding

The file stack.h

```
char pop();
void push(char);
const stack_size = 100;
```

The file stack.cpp

```
#include "stack.h"
static char v[stack_size]; // local to module
static char* p = v;

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

The file prog.cpp

```
#include "stack.h"

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

Data Abstraction

The file stack.h

```
class stack_id; // stack_id is a type

stack_id create_stack(int size);
void destroy_stack(stack_id);
char pop(stack_id);
void push(stack_id, char);
```

The file prog.cpp

```
#include "stack.h"

void some_function() {
  stack_id s1;
  stack_id s2;
  s1 = create_stack(200);
  // Oops: forgot to create s2
  push(s1, 'a');
  char c1 = pop(s1);
  if (c1 != 'a') error("impossible");
  push(s2, 'b');
  char c2 = pop(s2);
  if (c2 != 'b') error("impossible");
  destroy_stack(s2);
  // Ooops: forgot to detroy s1
}
```

Data Abstraction

The above scheme is nearly good enough when no more than one object of a type is needed.

```
class complex {
  double re, im;
public:
  complex(double r, double i) { re=r; im=i; }
  complex(double r) { re=r; im=0; }

  friend complex operator+(complex, complex);
  friend complex operator-(complex, complex);
  friend complex operator-(complex);
  friend complex operator*(complex, complex);
  friend complex operator/(complex, complex);
};
```

The definition of `complex +` could be:

```
complex operator+(complex, complex) {
  return complex(a1.re+a2.re, a1.im+a2.im);
}
```

Typical use:

```
complex a = 2.3;
complex b = 1/a;
complexc = a+b*complex(1, 2.3);
c = - (a/b)+2;
```

Problems with Data Abstraction

```
class point { ... };
class circle { ... };

enum kind { circle, triangle, square };

class shape {
  point center;
  color col;
  kind k;
  // representation of shape
public:
  point where() { return center; }
  void move{point to} { center=to; draw(); }
  void draw();
  void rotate(int);
  // more operations
};

void shape::draw()
  switch (k) {
    case circle:   // draw a circle
                   break;
    case triangle: // draw a circle
                   break;
    case square:   // draw a circle
                   break;
  }
}
```

Object-Oriented Programming

```
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);
  ...
};
```

Typical use of class `shape` is:

```
void rotate_all(shape* v, int size, int angle) {
// rotate all members of a vector by given angle
  for (int i=0; i<size; i++) v[i].rotate(angle);
}
```

Typical definition of a particular shape.

```
class circle : public shape {
  int radius;
public:
  void draw() { ... }
  void rotate() {}  // yes, the null function
}
```

Initialization and Cleanup

```
class vector {
  int sz;
  int* v;
public:
  void init(int size);
  ...
};


vector v;
// don't use yet
v.init(10);
// ok to use v now
..
```

Initialization and Cleanup

It is better to use constructors and destructors.

```
class vector {
  int sz;
  int* v;
public:
  vector(int);                  // constructor
  ~vector();                    // destructor
  int& operator[] (int index); // subscript operator
```

Typical definitions:

```
vector::vector(int s) {
  if (s<=0) error("bad vector size");
  sz = s;
  v = new int[s]; // allocate an array
}

vector::~vector() {
  delete v;  // deallocate the vector
}
```

Assignment and Initialization

It is possible to control all copy operations in C++.

```
class vector {
  int sz;
  int* v;
public:
  vector(int);                      // constructor
  ~vector();                        // destructor
  int& operator[] (int index);      // subscript operator
  void operator=(const vector&);  // assignment
  vector(const vector&);            // initialization
};
```

Typical definitions:

```
vector::operator=(const vector& a) {
  // check size and copy elements
  if (sz != a.sz) error("bad vector size for =");
  for (int i=0; i,sz; i++) v[i]=a.v[i];
}

vector::vector(const vector& a) {
  // initialize a vector from another vector
  sz = a.sz;
  v = new int[sz];
  for (int i=0; i,sz; i++) v[i]=a.v[i];
}
```

Parameterized Types

It would be more useful if the writer of the `vector` class did not know what the element type it is to use.

```
template<class T> class vector { // element type T
  T* v;
  int sz;
public:
  vector(int) {
    if (s <= 0) error("bad vector size");
    v = new T[sz=s];
  }
  T& operator[] (int index);   // subscript operator
  int size() { return sz; }
  ...
};
```

A template specifies a family of types.

```
vector<int> v1(100);
vector<complex> v2(200);

v2[i] = complex(v1[x], v1[y]);
```

Usually, only one version of `size()` is needed.

Iterators

Iterators can be provided by overloading the function call operator.

```
class vector_iterator {
  vector& v;
  int i;
public:
  vector_iterator(vector& r) { i=0; v=r; }
  int operator()() {
    return i<v.size() ? v.elem(i++) : 0;
  }
};
```

It can now be used, as in:

```
void f(vector& v) {
  vector_iterator next(v);
  int i;
  while (i=next()) print(i);
}
```

Iterators (alternative)

The iterator mechanism can be put in the "container" type, if preferred.

```
class vector {
  int* v;
  int i;
  int current;
public:
  int next() { return (++current < sz) ? v[current] : 0; }
  int curr() { return v[current];
  int prev() { return (0 <= --current) ? v[current] : 0; }
};
```

It can now be used, as in:

```
void f(vector& v) {
  vector v(sz);
  int i;
  while (i=v.next()) print(i);
}
```

Multiple Implementations

```
template<class T> class stack {
public:template<class T> class stack {
  virtual void push(T) = 0; // pure virtual function
  virtual T pop() = 0;
};
```

This declares an abstract class. It can be used, but not created.

```
stack<cat> s; // error: stack is abstract

void some_fn(stack<cat> s, cat kitty) { // ok
  s.push(kitty);
  cat c2 = s.pop();
  ...
}
```

Multiple Implementations

```
template<class T> class aStack : public stack<T> {
  // actual representation of a stack object
  // in this case an array
  ...
public:
  aStack(int size);
  ~aStack();
  void push(T);
  T pop();
};
```

Elsewhere we can create a list based stack.

```
template<class T> class lStack : public stack<T> { ...
};
```

We can now create stacks of both sorts

```
void g() {
  lstack<cat> s1(100);
  astack<cat> s2(100);

  cat ginger;
  cat snowball;

  some_fn(s1, ginger);
  some_fn(s2, snowball);
}
```

Multiple Inheritance

Multiple inheritance is when a class inherits from more than one base class. C++ allows multiple inheritance many languages (eg Java) do not.

```
class my_displayed_task : public displayed, public task {
  // my stuff
};

class my_task : public task { // not displayed
  // my stuff
};

class my_displayed : public displayed { // not a task
  // my stuff
};
```

Using only single inheritance leads to code duplication and loss of flexibility – typically both.

Ambiguity

Multiple inheritance cause ambiguity when two parents have methods with the same name. Such ambiguity can be solved at compile time.

```
class A { public: void f(); ... }
class B { public: void f(); ... }
class C : public A, public B { ... }

void g(C* p) {
  p->f(); // error: ambiguous
}
```

A solution is the following:

```
class C : public A, public B {
  ...
public:
  void f();
  ...
}

void C::f()
{ // C's own stuff
  A::f();
  B::f();
}
```