CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 5
September 14, 2016

Functions and Methods

Call by value (recall)

Like C, C++ passes explicit parameters by value.

```
void f( int y ) { ... y=4; ... };
...
int x=3;
f(x);
```

- `x` and `y` are independent variables.
- `y` is created when `f` is called and destroyed when it returns.
- At the call, the *value* of `x` (=3) is used to initialize `y`.
- The assignment `y=4;` inside of `f` has no effect on `x`.

Call by pointer (recall)

Like C, pointer values (which I call **reference values**) are the things that can be stored in *pointer variables*.
Also like C, references values can be passed as arguments to functions having corresponding pointer parameters.

```
void g( int* p ) { ... (*p)=4; ... };
...
int x=3;
g(&x);
```

- `p` is created when `g` is called and destroyed when it returns.
- At the call, the *value* of `&x`, a reference value, is used to initialize `p`.
- The assignment `(*p)=4;` inside of `g` changes the value of `x`.

Call by reference

C++ has a new kind of parameter called a *reference* parameter.

```
void g( int& p ) { ... p=4; ... };
...
int x=3;
g(x);
```

- This does same thing as previous example; namely, the assignment `p=4` changes the value of `x`.
- Within the body of `g`, `p` is a synonym for `x`.
- For example, `&p` and `&x` are *identical* reference values.

I/O uses reference parameters

- The first argument to `<<` has type `ostream&`.
- `cout << x << y;` is same as `(cout << x) << y;`.
- `<<` returns a reference to its first argument, so this is also the same as

```
cout << x;
cout << y;
```

How should one choose the parameter type?

Parameters are used for two main purposes:

- To send data to a function.
- To receive data from a function.

Sending data to a function: call by value

For sending data to a function, call by value copies the data whereas call by pointer or reference copies only an address.

- If the data object is large, call by value is expensive of both time and space and should be avoided.
- If the data object is small (eg., an `int` or `double`), call by value is cheaper since it avoids the indirection of a reference.
- Call by value protects the caller's data from being inadvertantly changed.

Sending data to a function: call by reference or pointer

Call by reference or pointer allows the caller's data to be changed.
Use `const` to protect the caller's data from inadvertane change.

Ex: `int f( const int& x )` or `int g( const int* xp )`.

Prefer call by reference to call by pointer for input parameters.

Ex: `f( 234 )` works but `g( &234 )` does not.

Reason: 234 is not a variable and hence can not be the target of a pointer.
(The reason `f( 234 )` *does* work is a bit subtle and will be explained later.)

Receiving data from a function

An output parameter is expected to be changed by the function.

Both call by reference and call by pointer work.

Call by reference is generally preferred since it avoids the need for the caller to place an ampersand in front of the output variable.

Declaration: `int f( int& x )` or `int g( int* xp )`.

Call: `f( result )` or `g( &result )`.

The implicit argument

Every call to a class member function has an *implicit argument*, which is the object written before the dot in the function call.

```
class MyExample {
private:
   int count;    // data member
public:
   void advance(int n) { count += n; }
   ...
};
...
MyExample ex;
ex.advance(3);
```

Increments `ex.count` by 3.

this

The implicit argument is passed by pointer.

In the call `ex.advance(3)`, the implicit argument is `ex`, and a pointer to `ex` is passed to `advance()`.

The implicit argument can be referenced directly from within a member function using the keyword `this`.

Within the definition of `advance()`, `count` and `this->count` are synonymous.

Derivation

Class relationships

Classes can relate to and collaborate with other classes in many ways.

We first explore **derivation**, where one class modifies and extends another.

What is derivation?

One class can be *derived* from another.

Syntax:

```
class A {
public:
    int x;
    ...
};
class B : public A {
    int y;
    ...
};
```

`A` is the base class; `B` is the derived class.

`B` inherits the members from `A`.

Instances

A base class instance is contained in each derived class instance.

Similar to composition, except for inheritance.

Function members are also inherited.

Data and function members can be overridden in the derived class.

Derivation is a powerful tool for allowing variations to a design.

Some uses of derivation

Derivation has several uses.

- To allow a family of related classes to share common parts.
- To describe abstract interfaces à la Java.
- To allow generic methods with run-time dispatching.
- To provide a clean interface between existing, non-modifiable code and added user code.

Example: Parallelogram

```
class Parallelogram {
protected:              // allows access by children
    double base;        // length of base
    double side;        // length of side
    double angle;       // angle between base and side
public:
    Parallelogram() {}          // null default constructor
    Parallelogram(double b, double s, double a);
    double area() const;        // computes area
    double perimeter() const;   // computes perimeter
    ostream& print( ostream& out ) const;
};
```

Example: Rectangle

```
class Rectangle : public Parallelogram {
public:
   Rectangle( double b, double s ) {
      base = b;
      side = s;
      angle = pi/2.0; // assumes pi is defined elsewhere
   }
};
```

Derived class `Rectangle` inherits `area()`, `perimeter()`, and `print()` functions from `Parallelogram`.

Example: Square

```
class Square : public Rectangle {
public:
   Square( double b ) : Rectangle(b, b) {} // uses ctor
   bool inscribable( Square& s ) const {
      double diag = sqrt( 2.0 )*side;  // this diagonal
      return side <= s.side && diag >= s.side;
   }
   double area() const { return side*side; }
};
```

Derived class `Square` inherits the `perimeter()`, and `print()` methods from `Parallelogram` (via `Rectangle`).

It overrides the method `area()`.

It adds the method `inscribable()` that determines whether this square can be inscribed inside of its argument square `s`.

Notes on `Square`

Features of `Square`.

- The **ctor** `:Rectangle(b, b)` allows parameters to be supplied to the `Rectangle` constructor.
- The method `inscribable()` **extends** `Rectangle`, adding new functionality.
  It returns `true` if this square can be inscribed in square `s`.
- The function `area` overrides the less-efficient definition in `Parallelogram`.

Construction, Initialization, and Destruction

Structure of an object

A simple object is like a `struct` in C.
It consists of a block of storage large enough to contain all of its data members.

An object of a derived class contains an instance of the base class followed by the data members of the derived class.

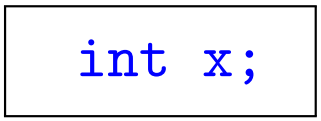
Example:

```
class B : A { ...};
B bObj;
```

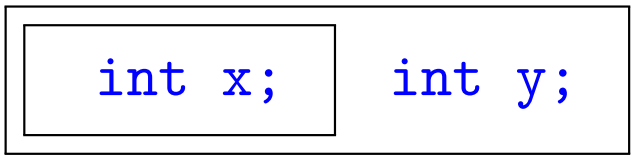
Then "inside" of `bObj` is an `A`-instance!

Example of object of a derived class

The declaration `A aObj` creates a variable of type `A` and storage size large enough to contain all of `A`'s data members (plus perhaps some padding).

`aObj`: 

The declaration `B bObj` creates a variable of type `B` and storage size large enough to contain all of `A`'s data members plus all of `B`'s data members.

`bObj`: 

The inner box denotes an `A`-instance.

Referencing a composed object

Contrast the previous example to

```
class B { A aObj; ...};
B bObj;
```

Here `B` composes `A`.

The embedded `A` object can be referenced using data member name `aObj`, e.g., `bObj.aObj`.

Referencing a base object

How do we reference the base object embedded in a derived class?

Example:

```
class A { public: int x; int y; ...};
class B : A { int y; ...};
B bObj;
```

- The data members of `A` can be referenced directly by name.
  `x` refers to data member `x` in class `A`.
  `y` refers to data member `y` in class `B`.
  `A::y` refers to data member `y` in class `A`.
- `this` points to the whole object.
  Its type is `B*`.
  It can be coerced to type `A*`.

Initializing an object

Whenever a class object is created, one of its constructors is called.

This applies not only to the "outer" object but also to all of its embedded objects.

If not specified otherwise, the default constructor is called.
This is the one that takes no arguments.

If you do not define the default constructor, then the null constructor (which does nothing) is used.

Construction rules

The rule for constructing an object of a simple class is:

1. Call the constructor/initializer for each data member, in sequence.
2. Call the constructor for the class.

The rule for constructing an object of a derived class is:

1. Call the constructor for the base class (which recursively calls the constructors needed to completely initialize the base class object.)
2. Call the constructor/initializer for each data member of the derived class, in sequence.
3. Call the constructor for the derived class.

Destruction rules

When an object is deleted, the destructors are called in the opposite order.

The rule for an object of a derived class is:

1. Call the destructor for the dervied class.
2. Call the destructor for each data member object of the derived class in reverse sequence.
3. Call the destructor for the base class.

Constructor ctors

Ctors (short for constructor/initializors) allow one to supply parameters to implicitly-called constructors.

Example:

```
class B : A {
  B( int n ) : A(n) {};
      // Calls A constructor with argument n
};
```

Initialization ctors

Ctors also can be used to initialze primitive (non-class) variables.

Example:

```
class  B {
  int x;
  const int y;
  B( int n ) : x(n), y(n+1) {}; // Initializes x and y
};
```

Multiple ctors are separated by commas.

Ctors present must be in the same order as the construction takes place – base class ctor first, then data member ctors in the same order as their declarations in the class.

Initialization not same as assignment

Previous example using ctors is not the same as writing

```
B( int n ) { y=n+1; x=n; };
```

- The order of initialization differs.
- `const` variables can be initialized but not assgined to.
- Initialization uses the constructor (for class objects).
- Initialization from another instance of the same type uses the copy constructor.

Copy constructors

- A copy constructor is automatically defined for each new class `A` and has prototype `A(const A&)`. It initializes a newly created `A` object by making a shallow copy of its argument.
- Copy constructors are used for call-by-value parameters.
- Assignment uses `operator=()`, which by default copies the data members but does not call the copy constructor.
- The results of the implicitly-defined assignment and copy constructors are the same, but they can be redefined to be different.

Move constructors

C++ 11 introduced a move constructor. Its purpose is to allow an object to be safely moved from one variable to another while avoiding the "double delete" problem.

We'll return to this interesting topic later, after we've looked more closely at dynamic extensions.