# CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 5
September 12, 2018

# Functions and Methods

# Call by value

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

```
void f( int y ) { ... y=4; ... };
// Calling context
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

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 with corresponding pointer parameters.

```
void g( int* p ) { ... (*p)=4; ... };
// Calling context
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; ... };
// Calling context
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 inadvertant 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

A parameter that is expected to be changed by the function is called an **output parameter**. (This is distinct from the function return value.)

Both call by reference and call by pointer work for output parameters.

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**. This 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; }
   ...
};
// Calling context
MyExample ex;
ex.advance(3);
```

Increments `ex.count` by 3.

# `this` keyword

The implicit argument is passed by pointer.

It can be referenced directly from within a member function using the special keyword `this`.

In the call `ex.advance(3)`, the implicit argument is `ex`, and `this` acts like a pointer variable of type `MyExample*` that has been initialized to `&ex`.

Within the body of `advance()`, the variable name `count` and the expresssion `this->count` are synonymous. Both refer to the private data member `count`.

# 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 Base {
public:
    int x;
    ...
};
class Deriv : public Base {
    int y;
    ...
};
```

`Base` is the base class; `Deriv` is the derived class.

`Deriv` inherits the members from `Base`.

# 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:                 // public API
    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`.

# Objects of Class Types

# 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.

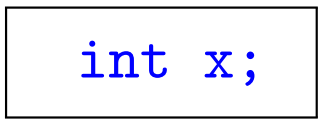
Example:

```
class Deriv :  Base { ... };
Deriv myObj;
```

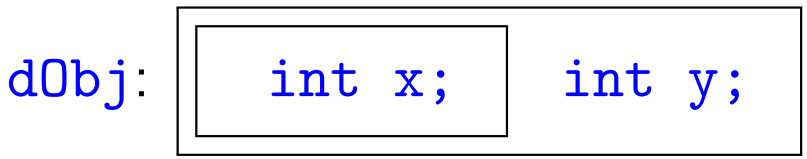
Then "inside" of `myObj` is a `Base`-instance!

# Example object of a derived class

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




The declaration `Deriv dObj` creates a variable of type `Deriv` and storage size large enough to contain all of `Base`'s data members plus all of `Deriv`'s data members.




The inner box denotes a `Base`-instance.

# Referencing a composed object

Contrast the previous example to

```
class Deriv { Base bObj; ...};
Deriv dObj;
```

Here `Deriv` composes `Base`.

The variable `x` from the embedded `Base` object can be referenced using `bObj.x` .

# Referencing a base object

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

Example:

```
class Base { public: int x; int y; ...};
class Deriv :  Base { int y; ...};
Deriv dObj;
```

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