# ECE 2400 Computer Systems Programming
## Fall 2018
### Topic 12: Object-Oriented Programming

School of Electrical and Computer Engineering  
Cornell University  
revision: 2018-11-02-09-28

## 1 C++ Classes  

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1.</td>
<td>C++ Member Functions</td>
<td>7</td>
</tr>
<tr>
<td>1.2.</td>
<td>C++ Constructors</td>
<td>11</td>
</tr>
<tr>
<td>1.3.</td>
<td>C++ Operator Overloading</td>
<td>13</td>
</tr>
<tr>
<td>1.4.</td>
<td>C++ Rule of Three</td>
<td>16</td>
</tr>
<tr>
<td>1.5.</td>
<td>C++ Data Encapsulation</td>
<td>24</td>
</tr>
<tr>
<td>1.6.</td>
<td>C++ I/O Streams</td>
<td>25</td>
</tr>
</tbody>
</table>

## 2 C++ Inheritance  

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.</td>
<td>C++ Concrete Inheritance</td>
<td>30</td>
</tr>
<tr>
<td>2.2.</td>
<td>Transition from Concrete to Abstract Inheritance</td>
<td>34</td>
</tr>
<tr>
<td>2.3.</td>
<td>C++ Abstract Inheritance</td>
<td>42</td>
</tr>
<tr>
<td>2.4.</td>
<td>Revisiting Composition vs. Generalization</td>
<td>45</td>
</tr>
</tbody>
</table>

Copyright © 2018 Christopher Batten. All rights reserved. This handout was prepared by Prof. Christopher Batten at Cornell University for ECE 2400 / ENGRD 2140 Computer Systems Programming. Download and use of this handout is permitted for individual educational non-commercial purposes only. Redistribution either in part or in whole via both commercial or non-commercial means requires written permission.
Object-oriented programming

• Programming is organized around defining, instantiating, and manipulating objects which contain data (i.e., fields, attributes) and code (i.e., methods)
• Classes are the “types” of objects, objects are instances of classes
• Classes are nouns, methods are verbs/actions
• Classes are organized according to various relationships
  – composition relationship (“Class X has a Y”)
  – generalization relationship (“Class X is a Y”)
  – association relationship (“Class X acts on Y”)

Example class diagram for animals
Example class diagram for chess pieces
Example class diagram for shapes and drawings
1. C++ Classes

• Perfectly possible to use object-oriented programming in C

```cpp
typedef struct
{
    double x;
    double y;
}
point_t;

void point_translate( point_t* this,
                        double x_offset, double y_offset )
{
    this->x += x_offset; this->y += y_offset;
}

void point_scale( point_t* this, double scale )
{
    this->x *= scale; this->y *= scale;
}

void point_rotate( point_t* this, double angle )
{
    const double pi = 3.14159265358979323846;
    double s = std::sin((angle*pi)/180);
    double c = std::cos((angle*pi)/180);

    double x_new = (c * this->x) - (s * this->y);
    double y_new = (s * this->x) + (c * this->y);

    this->x = x_new; this->y = y_new;
}

void point_print( point_t* this )
{
    std::printf("(%.2f,%.2f)", this->x, this->y);
}
```
int main( void )
{
    point_t pt;
    pt.x = 1;
    pt.y = 2;
    point_translate( &pt, 1, 0 );
    point_scale ( &pt, 2 );
    return 0;
}
1.1. **C++ Member Functions**

- C++ allows functions to be defined *within* the struct namespace
- C++ struct has both member fields and member functions
- Member functions have an implicit this pointer
- Member functions which do not modify fields are const

```cpp
struct Point
{
    double x; // member fields
    double y; //

    // member functions

    void point_translate( double x_offset, double y_offset )
    {
        this->x += x_offset; this->y += y_offset;
    }

    void point_scale( double scale )
    {
        this->x *= scale; this->y *= scale;
    }

    void point_rotate( double angle )
    {
        ...
        double x_new = (c * this->x) - (s * this->y);
        double y_new = (s * this->x) + (c * this->y);
        this->x = x_new; this->y = y_new;
    }

    void point_print() const
    {
        std::printf("(%.2f,%.2f)", this->x, this->y );
    }
};
```
• Non-static member functions are accessed using the dot (.) operator in the same way we access fields

```cpp
text
int main( void )
{
    Point pt;
    pt.x = 1;
    pt.y = 2;
    pt.point_translate( 1, 0 );
    pt.point_scale( 2 );
    return 0;
}
```

• Recall object-oriented prog in C

```cpp
int main( void )
{
    point_t pt;
    pt.x = 1;
    pt.y = 2;
    point_translate( &pt, 1, 0 );
    point_scale( &pt, 2 );
    return 0;
}
```

`point_translate( &pt, 1, 0 ) ↔ pt.point_translate( 1, 0 )`

`foo( &bar, ... ) ↔ bar.foo( ... )`
• Member functions are in struct namespace  
• No need to use the point_ prefix  
• Member fields are in scope within every member function  
• No need to explicitly use this pointer

    struct Point 
    { 
        double x; // member fields  
        double y; //  

        // member functions

        void translate( double x_offset, double y_offset )
        { 
            x += x_offset; y += y_offset;
        }

        void scale( double scale )
        { 
            x *= scale; y *= scale;
        }

        void rotate( double angle )
        { 
            ... 
            double x_new = (c * x) - (s * y);
            double y_new = (s * x) + (c * y);
            x = x_new; y = y_new;
        }

        void print() const
        { 
            std::printf( "(%.2f,%.2f)" , x, y );
        }
    };
int main( void )
{
  Point pt;
  pt.x = 1;
  pt.y = 2;
  pt.translate( 1, 0 );
  pt.scale( 2 );
  return 0;
}

- A class is just a struct with member functions
- An object is just an instance of a struct with member functions
1.2. C++ Constructors

- We want to avoid the user from directly accessing member fields
- We want to ensure an object is always initialized to a known state
- In C, we used foo_construct
- In C++, we could add a construct member function

```cpp
int main( void )
{
    Point pt;
    pt.construct( 1, 2 );
    pt.translate( 1, 0 );
    pt.scale( 2 );
    return 0;
}
```

- What if we call translate before construct?
- What if we call construct multiple times?
- We want a way to specify a special “constructor” member function
  - *always* called when you create an object
  - cannot be called directly, can *only* be called during object creation

- C++ adds support for language-level constructors
  (i.e., special member functions)
  - no return type
  - same name as the class

- Can use function overloading to have many different constructors
```cpp
struct Point {
  double x;
  double y;

  // default constructor
  Point() {
    x = 0.0;
    y = 0.0;
  }

  // non-default constructor
  Point( double x_, double y_ ) {
    x = x_;  
    y = y_; 
  }

  ... 
};

int main( void ) {
  Point pt0;
  Point pt1( 1, 2 );
  pt0 = pt1;
  pt0.translate( 1, 0 );
  return 0;
}
```
• Constructors automatically called with new

```
Point* pt0_p = new Point; // constructor called
Point* pt1_p = new Point[4]; // constructor called 4 times
```

• Initialization lists initialize members before body of constructor
  – Avoids creating a temporary default object
  – Required for initializing reference members
  – Prefer initialization lists when possible

```cpp
struct Point
{
    double x;
    double y;

    // default constructor
    Point()
    { x = 0.0; y = 0.0; }

    // non-default constructor
    Point( double x_, double y_ )
    { x = x_; y = y_; }
};
```

1.3. C++ Operator Overloading

• C++ operator overloading enables using built-in operators
  (e.g., +, -, *, /) with user-defined types

• Applying an operator to a user-defined type essentially calls an
  overloaded function (either a member function or a free function)
Point operator+( const Point& pt0, const Point& pt1 )
{
    Point tmp = pt0;
    tmp.translate( pt1.x, pt1.y );
    return tmp;
}

int main( void )
{
    Point ptA(1,2);
    Point ptB(3,4);
    Point ptC = ptA + ptB;
    return 0;
}
Point operator*( const Point& pt, double scale )
{
    Point tmp = pt;
    tmp.scale( scale );
    return tmp;
}

Point operator*( double scale, const Point& pt )
{
    Point tmp = pt;
    tmp.scale( scale );
    return tmp;
}

Point operator%( const Point& pt, double angle )
{
    Point tmp = pt;
    tmp.rotate( angle );
    return tmp;
}

Point operator%( double angle, const Point& pt )
{
    Point tmp = pt;
    tmp.rotate( angle );
    return tmp;
}

• Operator overloading enables elegant syntax for user-defined types

Point pt0(1,2);
pt0.translate(5,3);
pt0.rotate(45);
pt0.scale(1.5);
Point pt1 = pt0;

Point pt0(1,2);
Point pt1 = 1.5 * ( ( pt0 + Point(5,3) ) % 45 );
1.4. C++ Rule of Three

- What if point coordinates are allocated on the heap?

```cpp
struct Point {
    double* x_p;
    double* y_p;

    Point() {
        x_p = new double;
        y_p = new double;
        *x_p = 0.0; *y_p = 0.0;
    }

    void translate( double x_offset, double y_offset )
    {
        *x_p += x_offset;
        *y_p += y_offset;
    }

    ...}
```

```cpp
int main( void )
{
    Point pt0;
    pt0.translate( 1, 0 );
    return 0;
}
```
C++ Destructors

- Destructors are special member functions for destroying an object

```
struct Point {
  double* x_p;
  double* y_p;

  Point() {
    x_p = new double;
    y_p = new double;
    *x_p = 0.0; *y_p = 0.0;
  }

  ~Point() {
    delete x_p;
    delete y_p;
  }

  ...
};

int main( void )
{
  Point pt0;
  pt0.translate( 1, 0 );
  return 0;
}
```
• What if we copy an object with dynamically allocated memory?

```cpp
struct Point {
    double* x_p;
    double* y_p;

    Point() {
        x_p = new double;
        y_p = new double;
        *x_p = 0.0; *y_p = 0.0;
    }

    ~Point() {
        delete x_p; delete y_p;
    }

    ...}

int main( void )
{
    Point pt0;
    Point pt1 = pt0;
    pt0.translate( 1, 0 );
    return 0;
}
```
C++ Copy Constructors

- Copy constructors are special member functions for constructing a new object from an old object

```cpp
struct Point {
    double* x_p;
    double* y_p;

    Point(const Point& pt) {
        x_p = new double;
        y_p = new double;
        *x_p = *pt.x_p; *y_p = *pt.y_p;
    }

    ~Point() {
        delete x_p; delete y_p;
    }

    //...
};

int main(void) {
    Point pt0;
    Point pt1 = pt0;
    pt0.translate(1, 0);
    return 0;
}
```
What if we assign an object with dynamically allocated memory?

```cpp
struct Point {
  double* x_p;
  double* y_p;

  Point() {
    x_p = new double;
    y_p = new double;
    *x_p = 0.0; *y_p = 0.0;
  }

  ~Point() {
    delete x_p; delete y_p;
  }
...}

int main( void ) {
  Point pt0;
  Point pt1;
  pt1 = pt0;
  pt0.translate( 1, 0 );
  return 0;
}
```
C++ Assignment Operators

- An overloaded assignment operator will be called for assignment

```cpp
struct Point
{
    double* x_p;
    double* y_p;

    Point& operator=( const Point& pt )
    {
        delete x_p; delete y_p;
        x_p = new double;
        y_p = new double;
        *x_p = *pt.x_p; *y_p = *pt.y_p;
        return *this;
    }

    ...;
};

int main( void )
{
    Point pt0;
    Point pt1;
    pt1 = pt0;
    pt0.translate( 1, 0 );
    return 0;
}
```
C++ Rule of Three

- Default destructor, copy constructor, and assignment operator will work fine for simple classes
- For a more complex class may need to define one of these ...
- ... and if you define one, then you probably need to define all three!

```cpp
struct Point
{
    double* x_p;
    double* y_p;

    Point()
    {
        x_p = new double;
        y_p = new double;
        *x_p = 0.0;
        *y_p = 0.0;
    }

    Point( const Point& pt )
    {
        x_p = new double;
        y_p = new double;
        *x_p = *pt.x_p;
        *y_p = *pt.y_p;
    }

    ~Point()
    {
        delete x_p;
        delete y_p;
    }

    Point( double x, double y )
    {
        x_p = new double;
        y_p = new double;
        *x_p = x;
        *y_p = y;
    }

    Point& operator=( const Point& pt )
    {
        delete x_p;
        delete y_p;
        x_p = new double;
        y_p = new double;
        *x_p = *pt.x_p;
        *y_p = *pt.y_p;
        return *this;
    }
...
};
```
C++ Exceptions and Destructors

- Destructors called automatically for all objects in scope when exception thrown

```cpp
struct Point {
    double* x_p;
    double* y_p;

    ~Point() {
        delete x_p;
        delete y_p;
    }

    void translate( double x_offset, double y_offset ) {
        if ( (x_offset > 100) || (y_offset > 100) )
            throw 42;
        *x_p += x_offset;
        *y_p += y_offset;
    }

    ...
};

int main( void ) {
    try {
        Point pt0;
        pt0.translate( 1e9, 0 );
    }
    catch ( int e ) {
        return e;
    }
    return 0;
}
```
1.5. C++ Data Encapsulation

- Recall the importance of separating interface from implementation
- This is an example of abstraction
- In this context, also called information hiding, data encapsulation
  - Hides implementation complexity
  - Can change implementation without impacting users

- So far, we have relied on a policy to enforce data encapsulation
  - Users of a struct could still directly access member fields

```c++
int main( void )
{
    Point pt(1,2);
    pt.x = 13; // direct access to member fields
    return 0;
}
```

- In C++, we can enforce data encapsulation at compile time
  - By default all member fields and functions of a struct are public
  - Member fields and functions can be explicitly labeled as public or private
  - Externally accessing an internal private field causes a compile time error

```c++
struct Point
{
    private:
    double x; double y;

    public:
    // default constructor
    Point() { x = 0.0; y = 0.0; }

    // non-default constructor
    Point( double x_, double y_ ) { x = x_; y = y_; }

};
```
• In C++, we usually use class instead of struct
  – By default all member fields and functions of a struct are public
  – By default all member fields and functions of a class are private
  – We should almost always use class and explicitly use public and private

```cpp
class Point // almost always use class instead of struct
{
  public:  // always explicitly use public ...
    private: // ... or private
};
```

• We are free to change how we store the point
• We could change point to store coordinates on the stack or heap
• Statically guaranteed that others cannot access this private implementation

### 1.6. C++ I/O Streams

• printf does not support user-defined types
• Need to encapsulate printing in a member function ...
• ... but this leads to very awkward syntax

```cpp
int main( void )
{
  Point pt0(1,2);
  Point pt1 = pt0;
  pt1.translate(2.0,2.0);

  std::printf("initial point = ");
  pt0.print();
  std::printf("\n");
  std::printf("translate by %.2f,%.2f\n", 2.0, 2.0 );
  std::printf("new point = ");
  pt1.print();
  std::printf("\n");
}
```
Clever use of operator overloading can provide cleaner syntax

```cpp
struct ostream
{
    // internal stream state
};

ostream& operator<<( ostream& os, int i )
{
    std::printf("%d",i); return os;
}

ostream& operator<<( ostream& os, double d )
{
    std::printf("%.2f",d); return os;
}

ostream& operator<<( ostream& os, const char* str )
{
    std::printf("%s",str); return os;
}

ostream& operator<<( ostream& os, const Point& pt )
{
    pt.print(); return os;
}

struct EndOfLine
{
};

EndOfLine endl;

ostream& operator<<( ostream& os, const EndOfLine& endl )
{
    std::printf("\n"); return os;
}
```
int main( void )
{
    Point pt0(1,2);
    Point pt1 = pt0;
    pt1.translate(2.0,2.0);

    cout << "initial point = " << pt0 << endl;
    cout << "translate by " << 2.0 << "," << 2.0 << endl;
    cout << "new point = " << pt1 << endl;
}

• The standard C++ library provides powerful streams
  – iostream: write/read standard I/O as streams
  – fstream: write/read files as streams
  – sstream: write/read strings as streams
2. C++ Inheritance

• Object-oriented design without inheritance

• Object-oriented design with inheritance

• Inheritance enables declaring a *derived* class such that it automatically includes all of the member fields and functions associated with a different *base* class

  – Derived class also called the child class or subclass

  – Base class also called the parent class or superclass
```cpp
class Pawn
{
public:

    Pawn(char col, int row)
        : m_col(col), m_row(row)
    {}

    int get_pts() const
    {
        return 1;
    }

    char get_col() const
    {
        return m_col;
    }

    int get_row() const
    {
        return m_row;
    }

    void move(char col, int row)
    {
        if (col != m_col || (row != (m_row + 1)))
            throw 42;

        m_col = col;
        m_row = row;
    }

    void print() const
    {
        std::printf("pawn%c%d", m_col, m_row);
    }

private:
    char m_col;
    int m_row;
};
```
```cpp
class Rook
{
public:

    Rook(char col, int row)
        : m_col(col), m_row(row)
    {}

    int get_pts() const
    {
        return 5;
    }

    char get_col() const
    {
        return m_col;
    }

    int get_row() const
    {
        return m_row;
    }

    void move(char col, int row)
    {
        if ((col != m_col) && (row != m_row))
            throw 42;

        m_col = col;
        m_row = row;
    }

    void print() const
    {
        std::printf("rook%c%d", m_col, m_row);
    }

private:
    char m_col;
    int m_row;
};
```
2.1. C++ Concrete Inheritance

- Object-oriented design with concrete inheritance
```cpp
class Piece
{
public:

    Piece( char col, int row )
    : m_col(col), m_row(row)
    { }

    char get_col() const
    {
        return m_col;
    }

    int get_row() const
    {
        return m_row;
    }

    void move( char col, int row )
    {
        m_col = col;
        m_row = row;
    }

    // derived classes cannot
    // access private data in
    // base class

protected:

    char m_col;
    int m_row;
};

class Rook : public Piece
{
public:

    Rook( char col, int row )
    : Piece( col, row )
    { }

    int get_pts() const
    {
        return 5;
    }

    void move( char col, int row )
    {
        if ( (col != m_col)
            && (row != m_row) )
            throw 42;

        Piece::move( col, row );
    }

    void print() const
    {
        std::printf( "rook%c%d",
                        m_col, m_row );
    }

};
```
class Piece {
    Piece( char col, int row )
        : m_col(col), m_row(row) { }

    char get_col() const
        { return m_col; }

    int get_row() const
        { return m_row; }

};

class Pawn : public Piece {

    Pawn( char col, int row )
        : Piece( col, row ) { }

    int get_pts() const
        { return 1; }

};

int main( void )
{
    Pawn p( 'a', 2 );
    int row = p.get_col();
    return 0;
}
```cpp
class Piece
{
    Piece( char col, int row )
    : m_col(col), m_row(row)
    {}
}

char get_col() const
{
    return m_col;
}

int get_row() const
{
    return m_row;
}
...
};

class Pawn : public Piece
{
    int get_pts() const { return 1; }
    ...
};

class Rook : public Piece
{
    int get_pts() const { return 5; }
    ...
};

int main( void )
{
    Pawn p('a',2);
    Rook r('h',3);
    int pts0 = p.get_pts();
    int pts1 = r.get_pts();
    int sum = pts0 + pts1;
    return 0;
}
```
2.2. Transition from Concrete to Abstract Inheritance

- **Concrete Inheritance**
  - really more of the composition relationship (“has a”)
  - focuses on refactoring implementation code into the base class
  - still only instantiate and manipulate the derived “concrete” classes
  - need to know all type information at compile time

- **Abstract Inheritance**
  - the key to leveraging the generalization relationship (“is a”)
  - focuses on enabling dynamic polymorphism (i.e., subtype polymorphism)
  - can manipulate objects using the base “abstract” class
  - do not need to know concrete type information at compile time
  - can use run-time type information

Reviewing and applying type conversion to class hierarchies

- Compiler uses type conversion if variables have different types
- Compiler must convert types so they match
- Potential loss in information

```cpp
// no issue, b == 2.0
int a = 2;
float b = a;

// careful! b == 2
float a = 2.5;
int b = a;
```

- The following example illustrates automatic type conversion

```cpp
int avg( int x, int y )
{
    int sum = x + y;
    return sum / 2;
}

int main( void )
{
    float a = 10;
    float b = 15;
    float c = avg( a, b );
    return 0;
}
```
• Instances of derived class can be implicitly converted to base class
• Instances of pointer to derived class can be implicitly converted to pointer to base class

```cpp
Rook r = Rook('h',3);
Piece p = r; // implicit type conversion
p.get_col(); // can only call methods defined in Piece

Rook r = Rook('h',3);
Piece* p_p = &r; // implicit type conversion
p_p->get_col(); // can only call methods defined in piece
```

• Consider the following code snippet from previous example

```cpp
int pts0 = p.get_pts();
int pts1 = r.get_pts();
int sum = pts0 + pts1;
```

• Consider refactoring this code into a separate function
  – we want our function to be able to work with any type of chess piece
  – this function exhibits dynamic polymorphism
  – polymorphism = condition of occurring in several different forms
  – dynamic = run-time

```cpp
int calc_pts( Piece p0, Piece p1 ) {
    int pts0 = p0.get_pts();
    int pts1 = p1.get_pts();
    return pts0 + pts1;
}
```

```cpp
int calc_pts( Piece* p0, Piece* p1 ) {
    int pts0 = p0->get_pts();
    int pts1 = p1->get_pts();
    return pts0 + pts1;
}
```
class Piece
{
    Piece( char col, int row ) { ... }
    char get_col() const { ... }
    int get_row() const { ... }
    void move( char col, int row ) { ... }
}

class Pawn : public Piece
{
    int get_pts() const { return 1; }
    ... 
}

class Rook : public Piece
{
    int get_pts() const { return 5; }
    ... 
}

int calc_pts( Piece* p0, Piece* p1 )
{
    int pts0 = p0->get_pts();
    int pts1 = p1->get_pts();
    return pts0 + pts1;
}

int main( void )
{
    Pawn p('a',2);
    Rook r('h',3);
    int sum = calc_pts( &p, &r );
    return 0;
}
2. C++ Inheritance

2.2. Transition from Concrete to Abstract Inheritance

- Add type field to base class
- Use type field to determine concrete type
- Cast a pointer from the base class to a pointer to the appropriate derived type
- Call the corresponding member function

```cpp
// enum for all possible concrete types
enum PieceType { PAWN, ROOK };

class Piece
{
    public:

        Piece( PieceType type,
               char col, int row )
            : m_type(type),
              m_col(col), m_row(row)
        {
        }

    PieceType get_type() const
    {
        return m_type;
    }

    char get_col() const
    {
        return m_col;
    }

    int get_row() const
    {
        return m_row;
    }

    void move( char col, int row )
    {
        m_col = col;
        m_row = row;
    }

    protected:

        // explicit type field
        PieceType m_type;

        char      m_col;
        int       m_row;
};

class Rook : public Piece
{
    public:

        Rook( char col, int row )
            : Piece( ROOK, col, row )
        {
        }

        int get_pts() const
        {
            return 5;
        }

        void move( char col, int row )
        {
            if ( (col != m_col)
                && (row != m_row) )
                throw 42;
            Piece::move( col, row );
        }

        void print() const
        {
            std::printf( "rook@%c%d",
                         m_col, m_row );
        }
};
```
Reviewing and applying type casting to class hierarchies

- Programmers can use type casting to explicitly convert types

```cpp
float avg( int x, int y )
{
    int sum = x + y;
    return ((float) sum) / 2.0;
}

int main( void )
{
    float a = 10;
    float b = 15;
    float c = avg( a, b );
    return 0;
}
```

- Instances of base class cannot be explicitly cast to derived class
- Instances of pointer to base class can be explicitly cast to pointer to derived class

```cpp
Rook r0 = Rook('h',3);
Piece p = r0; // implicit type conversion
Rook r1 = (Rook) p; // not allowed! // a Piece is not a Rook

Rook r = Rook('h',3);
Piece* p = &r; // implicit type conversion
Rook* r_p = (Rook*) &p; // explicit type cast
```
class Pawn : public Piece {
    int get_pts() const { return 1; }
    ...;
};

int calc_pts( Piece* p0, Piece* p1 )
{
    int pts0;
    if ( p0->get_type() == PAWN ) {
        Pawn* pawn_p = (Pawn*) p0;
        pts0 = pawn_p->get_pts();
    }
    else if ( p0->get_type() == ROOK ) {
        Rook* rook_p = (Rook*) p0;
        pts0 = rook_p->get_pts();
    }

    int pts1;
    if ( p1->get_type() == PAWN ) {
        Pawn* pawn_p = (Pawn*) p1;
        pts1 = pawn_p->get_pts();
    }
    else if ( p1->get_type() == ROOK ) {
        Rook* rook_p = (Rook*) p1;
        pts1 = rook_p->get_pts();
    }

    return pts0 + pts1;
}

int main( void )
{
    Pawn p('a',2);
    Rook r('h',3);
    int sum = calc_pts( &p, &r );
    return 0;
}
2. C++ Inheritance

2.2. Transition from Concrete to Abstract Inheritance

```cpp
enum PieceType { PAWN, ROOK };

class Piece {
    public:
        Piece( PieceType type, char col, int row )
            : m_type(type), m_col(col), m_row(row) {}

        int get_pts() const {
            if ( m_type == PAWN ) {
                Pawn* pawn_p = (Pawn*) this;
                return pawn_p->get_pts();
            } else if ( m_type == ROOK ) {
                Rook* rook_p = (Rook*) this;
                return rook_p->get_pts();
            }
        }

        char get_col() const {
            return m_col;
        }

        int get_row() const {
            return m_row;
        }

        void move( char col, int row ) {
            m_col = col;
            m_row = row;
        }

    protected:
        PieceType m_type;
        char m_col;
        int m_row;
};

- Add type field to base class
- Use type field to determine concrete type
- Cast a pointer from the base class to a pointer to the appropriate derived type
- Call the corresponding member function

- Example of dynamic dispatch

```
```cpp
class Piece {
    ...
    int get_pts() const {
        if ( m_type == PAWN ) {
            Pawn* pawn_p = (Pawn*) this;
            return pawn_p->get_pts();
        }
        else if ( m_type == ROOK ) {
            Rook* rook_p = (Rook*) this;
            return rook_p->get_pts();
        }
    }
};

class Pawn : public Piece {
    int get_pts() const { return 1; }
};

int calc_pts( Piece* p0, Piece* p1 ) {
    int pts0 = p0->get_pts();
    int pts1 = p1->get_pts();
    return pts0 + pts1;
}

int main( void ) {
    Pawn p('a',2);
    Rook r('h',3);
    int sum = calc_pts( &p, &r );
    return 0;
}
```
2.3. C++ Abstract Inheritance

- C++ includes language support for dynamic dispatch
- virtual keyword can be used with a base class member function
- Calling a virtual member function will dynamically dispatch to the appropriate derived class member function
- At least one virtual function, compiler generates implicit type field
- Compiler generates more optimized version of dynamic dispatch

```cpp
class Piece
{
public:

    Piece( char col, int row ) : m_col(col), m_row(row)
    {
    }

    // pure virtual function
    virtual int get_pts() const = 0;

    char get_col() const
    {
        return m_col;
    }

    int get_row() const
    {
        return m_row;
    }

    void move( char col, int row )
    {
        m_col = col;
        m_row = row;
    }

protected:
    char m_col;
    int m_row;
};

class Rook : public Piece
{
public:

    Rook( char col, int row ) : Piece( col, row )
    {
    }

    int get_pts() const
    {
        return 5;
    }

    void move( char col, int row )
    {
        if ( (col != m_col) && (row != m_row) )
            throw 42;

        Piece::move( col, row );
    }

    void print() const
    {
        std::printf( "rook%c%d", m_col, m_row );
    }
};
```
```cpp
class Piece
{
    virtual int get_pts() const = 0;
    ...
};

class Pawn : public Piece
{
    int get_pts() const { return 1; }
    ...
};

class Rook : public Piece
{
    int get_pts() const { return 5; }
    ...
};

int calc_pts( Piece* p0, Piece* p1 )
{
    int pts0 = p0->get_pts();
    int pts1 = p1->get_pts();
    return pts0 + pts1;
}

int main( void )
{
    Pawn p('a',2);
    Rook r('h',3);
    int sum = calc_pts( &p, &r );
    return 0;
}
```
Abstract base classes

- All member functions are pure virtual member functions
- Need to have a non-pure virtual destructor in base class

```cpp
class IPiece
{
public:
    virtual ~IPiece() = 0;
    virtual int get_pts() const = 0;
    virtual char get_col() const = 0;
    virtual int get_row() const = 0;
    virtual void move(char col, int row) = 0;
    virtual void print() const = 0;
};

class Rook : public IPiece
{
public:
    Rook(char col, int row) : Piece(col, row) { }
    ~Rook() { }

    int get_pts() const { return 5; }
    char get_col() const { return m_col; }
    int get_row() const { return m_row; }

    void move(char col, int row)
    {
        if ((col != m_col) && (row != m_row))
            throw 42;
        m_col = col;
        m_row = row;
    }

    void print() const
    {
        std::printf("rook%c%d", m_col, m_row);
    }

private:
    char m_col;
    int m_row;
};
```
2.4. Revisiting Composition vs. Generalization


class Position
{
public:
  Position( char col, int row )
  : m_col(col), m_row(row)
  { }

  char get_col() const
  {
    return m_col;
  }

  int get_row() const
  {
    return m_row;
  }

  void move( char col, int row )
  {
    m_col = col;
    m_row = row;
  }

  void print() const
  {
    std::printf( "%c%d",
                 m_col, m_row );
  }

private:
  char m_col;
  int m_row;
};

class Pawn : public IPiece
{
public:
  Pawn( char col, int row )
  : m_pos(col,row)
  { }

  ~Pawn()
  { }

  int get_pts() const
  {
    return 1;
  }

  char get_col() const
  {
    return m_pos.get_col();
  }

  int get_row() const
  {
    return m_pos.get_row();
  }

  void move( char col, int row )
  {
    int curr_col = m_pos.get_col();
    int curr_row = m_pos.get_row();
    if ( (col != curr_col)
         || (row != (curr_row + 1)) )
      throw 42;
    m_pos.move( col, row );
  }

  void print() const
  {
    std::printf("pawn@");
    m_pos.print();
  }

private:
  Position m_pos;
};