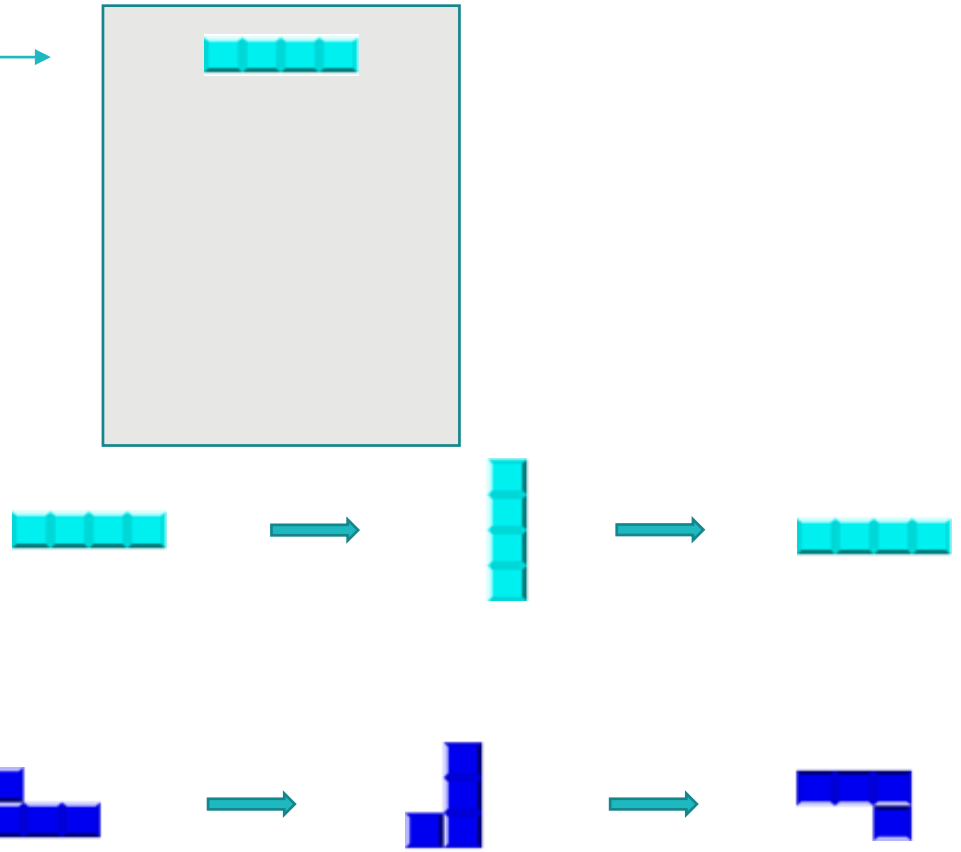
Lecture 10 - Polymorphism

Meng-Hsun Tsai
CSIE, NCKU

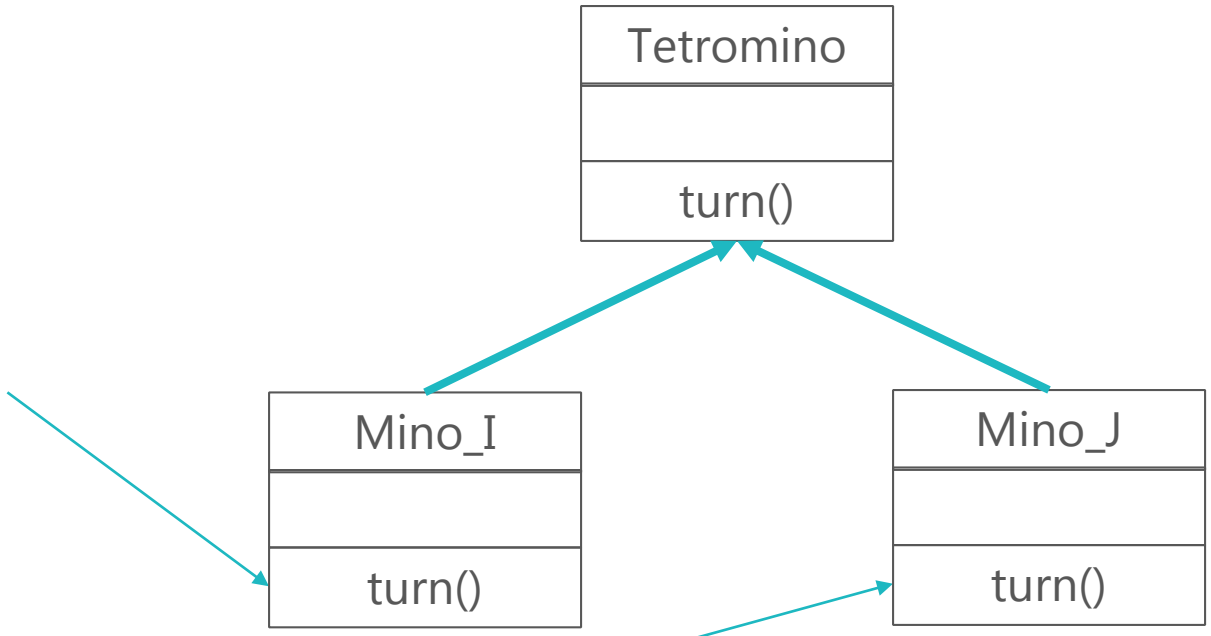
Polymorphism in Tetris Game

```
Tetromino * mino;
mino = new Mino_I ;
mino->turn( );
mino->turn( );
delete mino;
mino = new Mino_J ;
mino->turn( );
mino->turn( );
delete mino;
```



Polymorphism in Tetris Game (cont.)

```
Tetromino * mino;

mino = new Mino_I ;

mino->turn( );

mino->turn( );

delete mino;

mino = new Mino_J ;

mino->turn( );

mino->turn( );

delete mino;
```



Introduction

- We now continue our study of OOP by explaining and demonstrating polymorphism with inheritance hierarchies.
- Polymorphism enables us to "program in the general" rather than "program in the specific."
  - Enables us to write programs that process objects of classes that are part of the same class hierarchy as if they were all objects of the hierarchy's base class.
- Polymorphism works off base-class pointer handles and base-class reference handles, but not off name handles.
- Relying on each object to know how to "do the right thing" in response to the same function call is the key concept of polymorphism.

Introduction (cont.)

- With polymorphism, we can design and implement systems that are easily extensible.
  - New classes can be added with little or no modification to the general portions of the program, as long as the new classes are part of the inheritance hierarchy that the program processes generically.
  - The only parts of a program that must be altered to accommodate new classes are those that require direct knowledge of the new classes that you add to the hierarchy.
- The same message sent to a variety of objects has "many forms" of results—hence the term polymorphism.

Polymorphism Examples

- Polymorphism occurs when a program invokes a `virtual` function through a base-class pointer or reference.
  - C++ dynamically (i.e., at execution time) chooses the correct function for the class from which the object was instantiated.
- Polymorphism promotes extensibility: Software written to invoke polymorphic behavior is written independently of the types of the objects to which messages are sent. Thus new types of objects that can respond to existing messages can be incorporated into such a system without modifying the base system. Only client code that instantiates new objects must be modified to accommodate new types.

Relationships Among Objects in an Inheritance Hierarchy

- An object of a derived class can be treated as an object of its base class.
- Despite the fact that the derived-class objects are of different types, the compiler allows this because each derived-class object *is an* object of its base class.
- However, we cannot treat a base-class object as an object of any of its derived classes.
- The *is-a* relationship applies only from a derived class to its direct and indirect base classes.

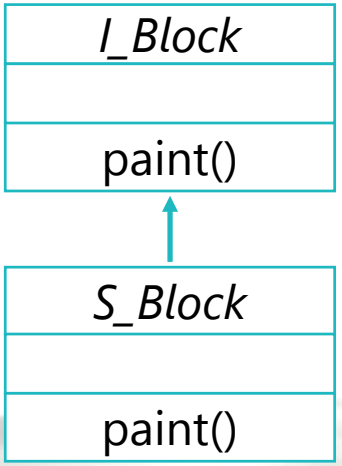
IBlock.h

```
#ifndef I_BLOCK_H
#define I_BLOCK_H
#include <iostream>
using namespace std;
char I_arr [2][4][4] = {{{'0','0','1','0'},
                        {'0','0','1','0'},
                        {'0','0','1','0'},
                        {'0','0','1','0'}},
                       {{'0','0','0','0'},
                        {'0','0','0','0'},
                        {'1','1','1','1'},
                        {'0','0','0','0'}} };
class I_Block{
   public:
      I_Block():x(0),y(0),rotate_index(0) {}
      I_Block& rotate(){
         rotate_index=(rotate_index>0)?
                        0:rotate_index+1;
         return *this;
      }
      I_Block& left() {x=(x>0)?(x-1):10;
                       return *this;}
      I_Block& right() {x=(x>10)?0:x+1;
                        return *this;}
      void paint() {
         for(int i=0;i<4;++i)
         {
            for(int j=0;j<x;++j) cout << ' ';
            for(int j=0;j<4;++j)
               cout << I_arr[rotate_index][i][j];
            cout << endl;
         }
         cout << endl;
      }
   public:
      int x, y;
      int rotate_index;
};
#endif
```

SBlock_inh.h

```
 1 #ifndef S_BLOCK_INH_H
 2 #define S_BLOCK_INH_H
 3 #include <iostream>
 4 #include "IBlock.h"
 5 using namespace std;
 6 char S_arr [2][4][4] = {{{'0','0','0','0'},
 7                          {'0','0','0','0'},
 8                          {'0','0','1','1'},
 9                          {'0','1','1','0'}},
10                         {{'0','0','0','0'},
11                          {'0','1','0','0'},
12                          {'0','1','1','0'},
13                          {'0','0','1','0'}} };
```

```
14 class S_Block: public I_Block{
15     public:
16         void paint() {
17             for(int i=0;i<4;++i)
18             {
19                 for(int j=0;j<x;++j) cout << ' ';
20                 for(int j=0;j<4;++j)
21                     cout << S_arr[rotate_index][i][j];
22                 cout << endl;
23             }
24             cout << endl;
25         }
26 };
27 #endif
```



Error: Assigning Address of Base Class Object to Pointer of Derived Class

tetris.cpp

```
 1 #include <iostream>
 2 #include "SBlock_inh.h"
 3 using namespace std;
 4 int main()
 5 {
 6    I_Block i, *ip = &i;
 7    S_Block s, *sp = &s;
 8    i.paint();
 9    ip->paint();
10    s.paint();
11    sp->paint();
12    sp = &i;
13    sp->paint();
14    return 0;
15 }
```

SBlock_inh.h

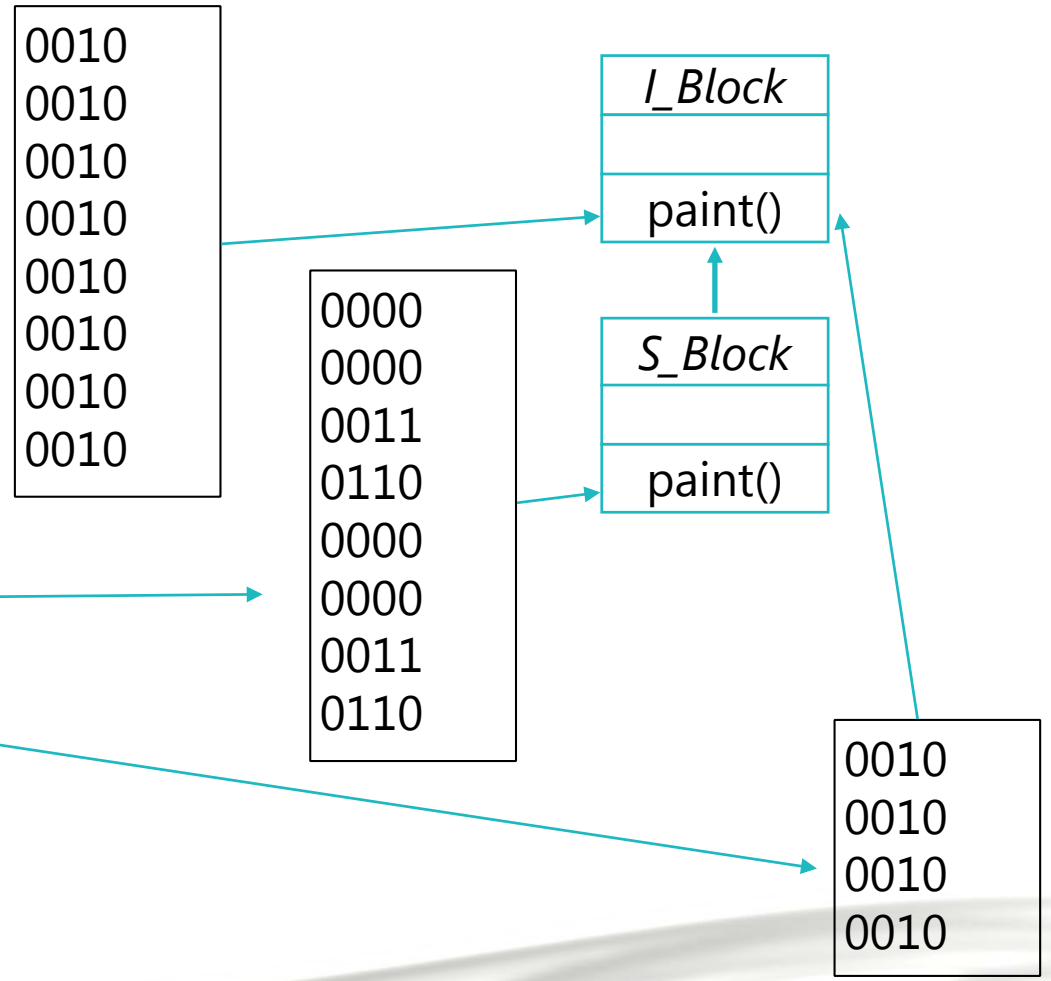
```
13 ...
14 class S_Block: public I_Block{
15 ...
```


```
tetris.cpp: In function 'int main()':
tetris.cpp:12: error: invalid conversion from 'I_Block*' to
'S_Block*'
*** [tetris.o] Error code 1
```

Assigning Address of Derived Class Object to Pointer of Base Class

tetris.cpp

```
 1  #include <iostream>
 2  #include "SBlock_inh.h"
 3  using namespace std;
 4  int main()
 5  {
 6     I_Block i, *ip = &i;
 7     S_Block s, *sp = &s;
 8     i.paint();
 9     ip->paint();
10     s.paint();
11     sp->paint();
12     ip = &s;
13     ip->paint();
14     return 0;
15  }
```



Error: Calling Derived-class-only Member Function From a Base-class Pointer

tetris2.cpp

```
 1  #include <iostream>
 2  #include "SBlock2.h"
 3  using namespace std;
 4  int main()
 5  {
 6     I_Block *ip;
 7     S_Block s;
 8     ip = &s;
 9     ip->paint();
10     ip->de_fun();
11     return 0;
12  }
```

SBlock2.h

```
   ...
17      void paint() {
18         I_Block::paint();
19         for(int i=0;i<4;++i)
20         {
21            for(int j=0;j<x;++j) cout << ' ';
22            for(int j=0;j<4;++j)
23               cout << S_arr[rotate_index][i][j];
24            cout << endl;
25         }
26      }
27      void de_fun() { }
   ...
```


```
tetris2.cpp: In function 'int main()':
tetris2.cpp:10: error: 'class I_Block' has no member named 'de_fun'
*** [tetris2.o] Error code 1
```

Calling Derived-class-only Member Function From a Casted Base-class Pointer

tetris3.cpp

```
 1 #include <iostream>
 2 #include "SBlock2.h"
 3 using namespace std;
 4 int main()
 5 {
 6    I_Block *ip;
 7    S_Block s;
 8    ip = &s;
 9    static_cast<S_Block *>(ip)
                    ->de_fun();
10    static_cast<S_Block *>(ip)
                    ->paint();
11    return 0;
12 }
```

SBlock2.h

```
   ...
17    void paint() {
18       I_Block::paint();
19       for(int i=0;i<4;++i)
20       {
21          for(int j=0;j<x;++j) cout << ' ';
22          for(int j=0;j<4;++j)
23             cout <<
                   S_arr[rotate_index][i][j];
24          cout << endl;
25       }
26    }
27    void de_fun() { }
   ...
```


```
0010
0010
0010
0010
0000
0000
0011
0110
```

Derived-Class Member-Function Calls via Base-Class Pointers

- Off a base-class pointer, the compiler allows us to invoke only base-class member functions.
- If a base-class pointer is aimed at a derived-class object, and an attempt is made to access a derived-class-only member function, a compilation error will occur.

```
10      ip->de_fun();
```

- The compiler allows access to derived-class-only members from a base-class pointer that is aimed at a derived-class object *if* we explicitly cast the base-class pointer to a derived-class pointer—known as downcasting.
- After a downcast, the program can invoke derived-class functions that are not in the base class.

```
9     static_cast<S_Block *>(ip)->de_fun();
```

Virtual Functions

- With `virtual` functions, the type of the object being pointed to, not the type of the handle, determines which version of a `virtual` function to invoke.
- To paint any block, we could simply use a base-class pointer to invoke function `paint` and let the program determine dynamically (i.e., at runtime) which derived-class `paint` function to use, based on the type of the object to which the base-class pointer points at any given time.
- To enable this behavior, we declare `paint` in the base class as a virtual function, and we override `paint` in each of the derived classes to paint the appropriate block.
- We declare a `virtual` function by preceding the function's prototype with the keyword `virtual` in the base class.

Virtual Functions (cont.)

- Once a function is declared `virtual`, it remains `virtual` all the way down the inheritance hierarchy from that point, even if that function is not explicitly declared `virtual` when a derived class overrides it.
- Even though certain functions are implicitly `virtual` because of a declaration made higher in the class hierarchy, explicitly declare these functions `virtual` at every level of the hierarchy to promote program clarity.
- When a derived class chooses not to override a `virtual` function from its base class, the derived class simply inherits its base class's `virtual` function implementation.

Virtual Functions (cont.)

- If a program invokes a `virtual` function through a base-class pointer to a derived-class object or a base-class reference to a derived-class object, the program will choose the correct derived-class function dynamically (i.e., at execution time) based on the object type—not the pointer or reference type.
  - Known as dynamic binding or late binding.
- When a `virtual` function is called by referencing a specific object by name and using the dot member-selection operator, the function invocation is resolved at compile time (this is called static binding) and the `virtual` function that is called is the one defined for (or inherited by) the class of that particular object—this is not polymorphic behavior.

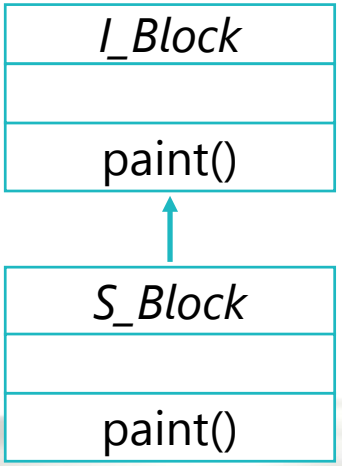
IBlock2.h

```
#ifndef I_BLOCK_H
#define I_BLOCK_H
#include <iostream>
using namespace std;
char I_arr [2][4][4] = {{{'0','0','1','0'},
                        {'0','0','1','0'},
                        {'0','0','1','0'},
                        {'0','0','1','0'}},
                       {{'0','0','0','0'},
                        {'0','0','0','0'},
                        {'1','1','1','1'},
                        {'0','0','0','0'}} };
class I_Block{
   public:
      I_Block(int xx=0,int yy=0,int ri=0):
            x(xx),y(yy),rotate_index(ri) {}   ;
      I_Block& rotate(){
        rotate_index=(rotate_index>0)?
                0:rotate_index+1;
      return *this;
      }
      I_Block& left() {x=(x>0)?(x-1):10;
                       return *this;}
      I_Block& right() {x=(x>10)?0:x+1;
                       return *this;}
      virtual void paint() {
         for(int i=0;i<4;++i)
         {
            for(int j=0;j<x;++j) cout << ' ';
            for(int j=0;j<4;++j)
               cout << I_arr[rotate_index][i][j];
            cout << endl;
         }
      }
   protected:
      int rotate_index;
      int x, y;
};
#endif
```

SBlock3.h

```
 1 #ifndef S_BLOCK_INH_H
 2 #define S_BLOCK_INH_H
 3 #include <iostream>
 4 #include "IBlock2.h"
 5 using namespace std;
 6 char S_arr [2][4][4] = {{{'0','0','0','0'},
 7                          {'0','0','0','0'},
 8                          {'0','0','1','1'},
 9                          {'0','1','1','0'}},
10                         {{'0','0','0','0'},
11                          {'0','1','0','0'},
12                          {'0','1','1','0'},
13                          {'0','0','1','0'}} };
```

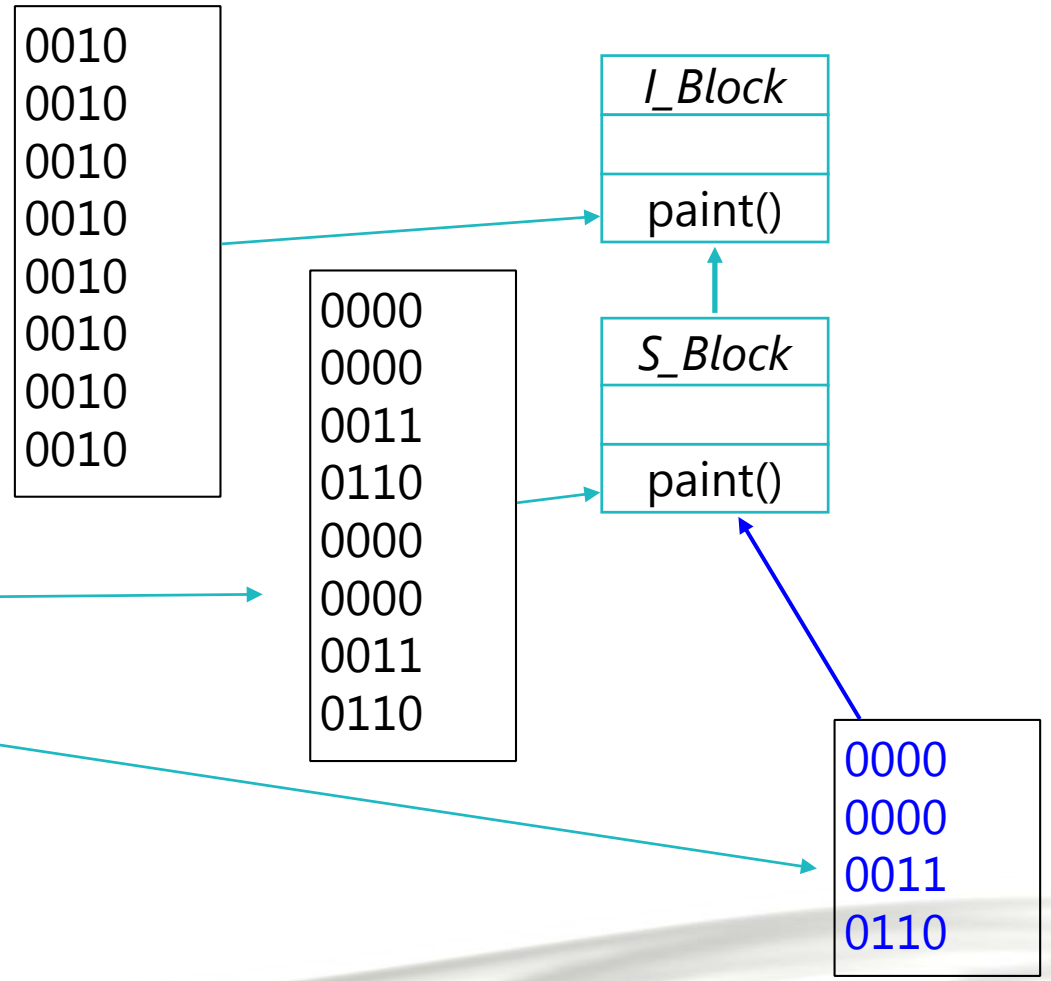
```
14 class S_Block: public I_Block{
15     public:
16         virtual void paint() {
17             for(int i=0;i<4;++i)
18             {
19                 for(int j=0;j<x;++j) cout << ' ';
20                 for(int j=0;j<4;++j)
21                     cout << S_arr[rotate_index][i][j];
22                 cout << endl;
23             }
24             cout << endl;
25         }
26 };
27 #endif
```



Calling a Virtual Function Through Pointer of Base Class

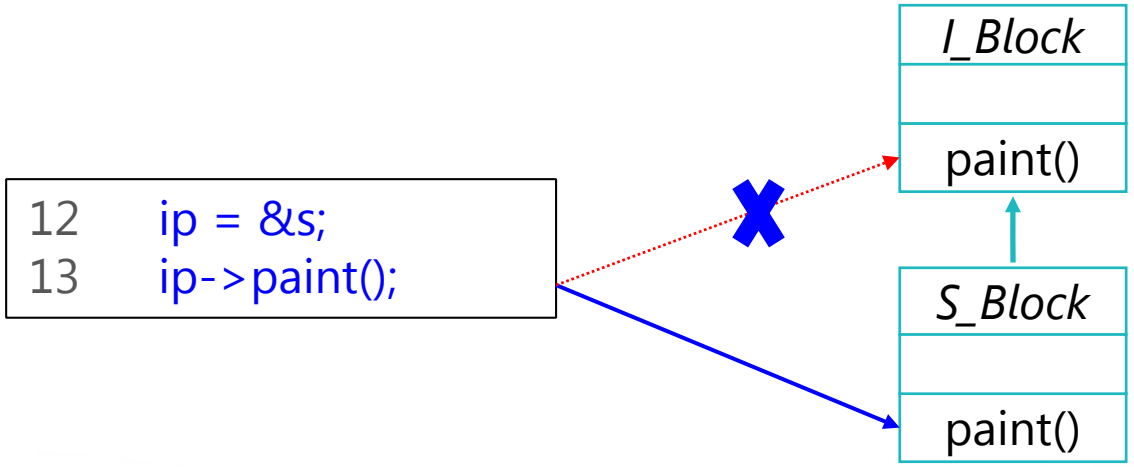
tetris4.cpp

```
 1  #include <iostream>
 2  #include "SBlock_inh.h"
 3  using namespace std;
 4  int main()
 5  {
 6     I_Block i, *ip = &i;
 7     S_Block s, *sp = &s;
 8     i.paint();
 9     ip->paint();
10     s.paint();
11     sp->paint();
12     ip = &s;
13     ip->paint();
14     return 0;
15  }
```



Calling a Virtual Function Through Pointer of Base Class (cont.)

- The only difference between these files is that we specify each class's `paint` member functions as `virtual`.
- Now, if we aim a base-class `I_Block` pointer at a derived-class `S_Block` object, and the program uses that pointer to call function `paint`, the `S_Block` object's corresponding function will be invoked.



Allowed Assignments Between Base-Class and Derived-Class Objects and Pointers

- We've discussed four ways to aim base-class pointers and derived-class pointers at base-class objects and derived-class objects:
  - Aiming a base-class pointer at a base-class object is straightforward—calls made off the base-class pointer simply invoke base-class functionality.
  - Aiming a derived-class pointer at a derived-class object is straightforward, too.
  - Aiming a base-class pointer at a derived-class object is safe, because the derived-class object *is an* object of its base class. This pointer can be used to invoke only base-class member functions.
  - Aiming a derived-class pointer at a base-class object generates a compilation error.

Abstract Classes and Pure `virtual` Functions

- There are cases in which it's useful to define classes from which you never intend to instantiate any objects. Such classes are called abstract classes.
- Because these classes normally are used as base classes in inheritance hierarchies, we refer to them as abstract base classes.
- These classes cannot be used to instantiate objects, because, as we'll soon see, abstract classes are incomplete—derived classes must define the "missing pieces."
- An abstract class provides a base class from which other classes can inherit.

Abstract Classes and Pure `virtual` Functions

- Classes that can be used to instantiate objects are called concrete classes. Such classes define every member function they declare.
- An inheritance hierarchy does not need to contain any abstract classes, but many object-oriented systems have class hierarchies headed by abstract base classes.
- In some cases, abstract classes constitute the top few levels of the hierarchy.

Abstract Classes and Pure `virtual` Functions (cont.)

- A class is made abstract by declaring one or more of its `virtual` functions to be "pure." A pure virtual function is specified by placing "`= 0`" in its declaration, as in
  - `virtual void paint() const = 0;`
- The "`= 0`" is a pure specifier.
- Pure `virtual` functions do not provide implementations.
- Every concrete derived class *must* override all base-class pure `virtual` functions with concrete implementations of those functions.

Abstract Classes and Pure `virtual` Functions (cont.)

- The difference between a `virtual` function and a pure `virtual` function is that a `virtual` function has an implementation and gives the derived class the option of overriding the function.
- By contrast, a pure `virtual` function does not provide an implementation and requires the derived class to override the function for that derived class to be concrete; otherwise the derived class remains abstract.
- Pure `virtual` functions are used when it does not make sense for the base class to have an implementation of a function, but you want all concrete derived classes to implement the function.

Abstract Classes and Pure `virtual` Functions (cont.)

- An abstract class defines a common public interface for the various classes in a class hierarchy.
- An abstract class has at least one pure `virtual` function. An abstract class also can have data members and concrete functions (including constructors and destructors).
- Attempting to instantiate an object of an abstract class causes a compilation error.

Error: Attempting to Instantiate an Abstract-class Object

```
class A {
        virtual int func() = 0;
};
int main()
{
        A objA;
        return 0;
}
```


```
$ g++ -o abstract_cls abstract_cls.cpp
abstract_cls.cpp: In function `int main()':
abstract_cls.cpp:10: error: cannot declare variable `objA' to be of type `A'
abstract_cls.cpp:10: error:   because the following virtual functions are abstract:
abstract_cls.cpp:5: error:  virtual int A::func()
```

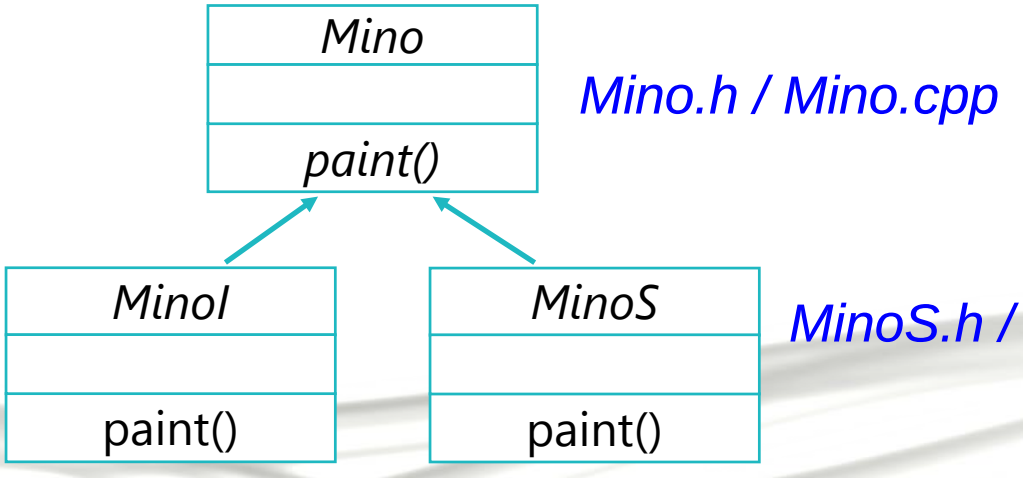
Abstract Classes and Pure `virtual` Functions (cont.)

- Although we cannot instantiate objects of an abstract base class, we can use the abstract base class to declare pointers and references that can refer to objects of any concrete classes derived from the abstract class.
- Programs typically use such pointers and references to manipulate derived-class objects polymorphically.

Case Study: Tetrominos in Tetris Game

In this example, there are 9 files:

- *Mino.h* and *Mino.cpp* for the abstract base class
- *MinoI.h* / *MinoI.cpp* and *MinoS.h* / *MinoS.cpp* for derived classes
- The interface *genMino.h* and *genMino.cpp* used by the client code.
- The client code *tetris5.cpp*



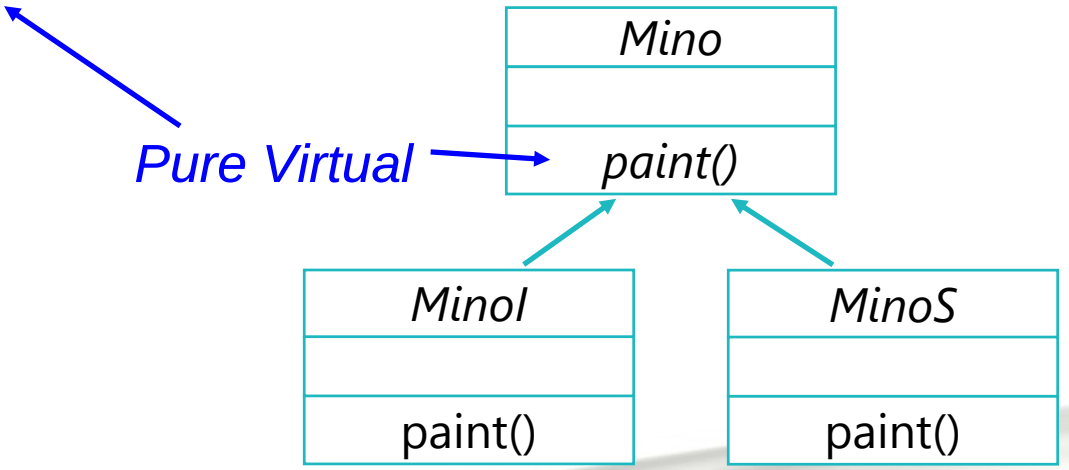
Mino.h and Mino.cpp

Mino.h

```
1  #ifndef MINO_H
2  #define MINO_H
3  #include <iostream>
4  using namespace std;
5  class Mino{
6     public:
7        Mino(int mri=1);
8        Mino& turn();
9        virtual void paint() =0;
10    protected:
11       int rotate_index, max_ri;
12 };
13 #endif
```

Mino.cpp

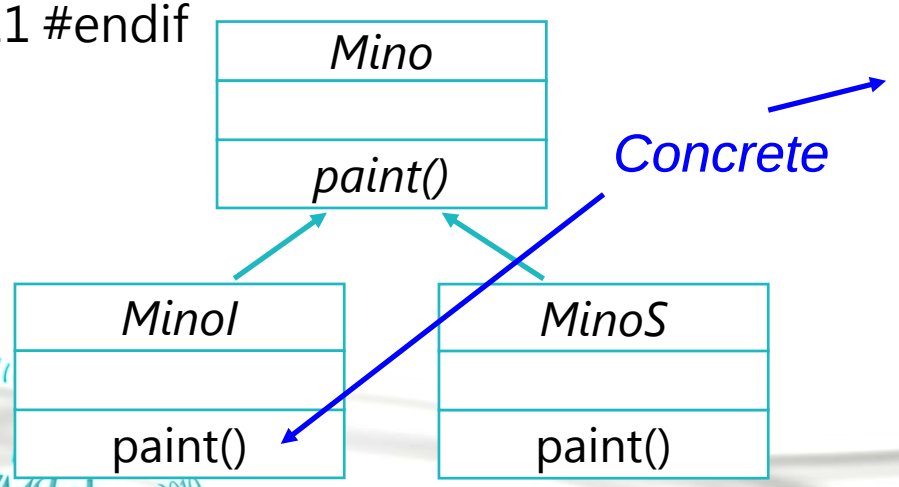
```
1  #include "Mino.h"
2  Mino::Mino(int mri):max_ri(mri) {};
3  Mino& Mino::turn()
4  {
5     rotate_index=(rotate_index>=max_ri)?
                        0:rotate_index+1;
6     return *this;
7  }
```



MinoI.h and MinoI.cpp

MinoI.h

```
1  #ifndef MINO_I_H
2  #define MINO_I_H
3  #include <iostream>
4  #include "Mino.h"
5  using namespace std;
6  class MinoI: public Mino{
7     public:
8        MinoI();
9        virtual void paint();
10 };
11 #endif
```



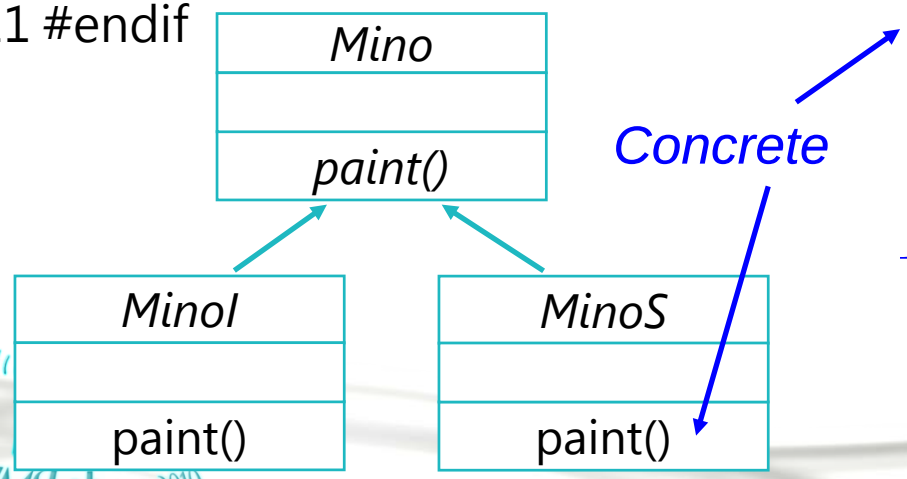
MinoI.cpp

```
1  #include "MinoI.h"
2  char I_arr [2][4][4] = {{{'0','0','1','0'},
3                           {'0','0','1','0'},
4                           {'0','0','1','0'},
5                           {'0','0','1','0'}},
6                          {{'0','0','0','0'},
7                           {'0','0','0','0'},
8                           {'1','1','1','1'},
9                           {'0','0','0','0'}} };
10 MinoI::MinoI():Mino(1){}
11 void MinoI::paint()
12 {
13    for(int i=0;i<4;++i)
14    {
15       for(int j=0;j<4;++j)
16          cout << I_arr[rotate_index][i][j];
17       cout << endl;
18    }
19
20 }
```

MinoS.h and MinoS.cpp

MinoS.h

```
1  #ifndef MINO_S_H
2  #define MINO_S_H
3  #include <iostream>
4  #include "Mino.h"
5  using namespace std;
6  class MinoS: public Mino{
7     public:
8        MinoS();
9        virtual void paint();
10 };
11 #endif
```



MinoS.cpp

```
1  #include "MinoS.h"
2  char S_arr [2][4][4] = {{{'0','0','0','0'},
3                           {'0','0','0','0'},
4                           {'0','0','1','1'},
5                           {'0','1','1','0'}},
6                          {{'0','0','0','0'},
7                           {'0','1','0','0'},
8                           {'0','1','1','0'},
9                           {'0','0','1','0'}} };
10 MinoS::MinoS():Mino(1){}
11 void MinoS::paint()
12 {
13    for(int i=0;i<4;++i)
14    {
15       for(int j=0;j<4;++j)
16          cout << S_arr[rotate_index][i][j];
17       cout << endl;
18    }
19
20 }
```

genMino.h and genMino.cpp

genMino.h

```
1  #ifndef GENMINO_H
2  #define GENMINO_H
3  #include "Mino.h"
4  #include "MinoS.h"
5  #include "MinoI.h"
6
7  Mino * genMino();
8  #endif
```

genMino.cpp

```
1 #include <cstdlib>
2 #include "genMino.h"
3 #define NUM_MINO 2
4 #define MINO_S 0
5 #define MINO_I 1
```

genMino.cpp

```
 6  Mino * genMino()
 7  {
 8    int mino_type;
 9    Mino * ptr;
10
11    mino_type = random() % NUM_MINO;
12
13    switch(mino_type) {
14      case MINO_S:
15        ptr = new MinoS;
16        break;
17      case MINO_I:
18        ptr = new MinoI;
19        break;
20    }
21    return ptr;
22  }
```

tetris5.cpp

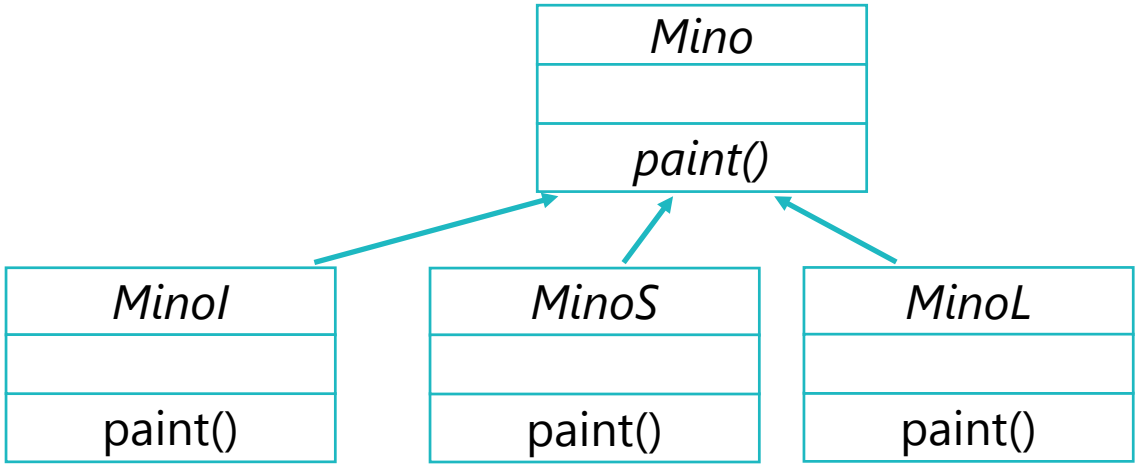
```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <ctime>
 4 #include "genMino.h"
 5 using namespace std;
 6 int main()
 7 {
 8   Mino * mino_ptr;
 9
10   srandom(time(NULL));
11   for(int i=0;i<3;++i)
12   {
13     mino_ptr = genMino();
14     mino_ptr->paint();
15     delete mino_ptr;
16   }
17   return 0;
18 }
```

```
0010
0010
0010
0010
0000
0000
0011
0110
0000
0000
0011
0110
```

Adding New Tetrominos

If we want to add a new tetromino MinoL, we only need to

- Add *MinoL.h* and *MinoL.cpp*
- Add corresponding codes in *genMino.h* and *genMino.cpp*, which must know something about MinoL.



MinoL.h / MinoL.cpp

MinoL.h and MinoL.cpp

MinoL.h

```
1  #ifndef MINO_L_H
2  #define MINO_L_H
3  #include <iostream>
4  #include "Mino.h"
5  using namespace std;
6  class MinoL: public Mino{
7     public:
8        MinoL();
9        virtual void paint();
10 };
11 #endif
```

MinoL.cpp

```
1  #include "MinoL.h"
2  char L_arr [4][4][4] = {{{'0','0','0','0'},
3                           {'0','1','0','0'},
4                           {'0','1','0','0'},
5                           {'0','1','1','0'}},
6                          {{'0','0','0','0'},
7                           {'0','0','0','0'},
8                           {'0','0','0','1'},
9                           {'0','1','1','1'}},
```

MinoL.cpp

```
10                         {{'0','0','0','0'},
11                          {'0','1','1','0'},
12                          {'0','0','1','0'},
13                          {'0','0','1','0'}},
14                         {{'0','0','0','0'},
15                          {'0','0','0','0'},
16                          {'0','1','1','1'},
17                          {'0','1','0','0'}} };
18 MinoL::MinoL():Mino(3){}
19 void MinoL::paint()
20 {
21    for(int i=0;i<4;++i)
22    {
23       for(int j=0;j<4;++j)
24          cout << L_arr[rotate_index][i][j];
25       cout << endl;
26    }
27 }
```

genMino.h and genMino.cpp

genMino.h

```
1  #ifndef GENMINO_H
2  #define GENMINO_H
3  #include "Mino.h"
4  #include "MinoS.h"
5  #include "MinoI.h"
6  #include "MinoL.h"
7
8  Mino * genMino();
9  #endif
```

genMino.cpp

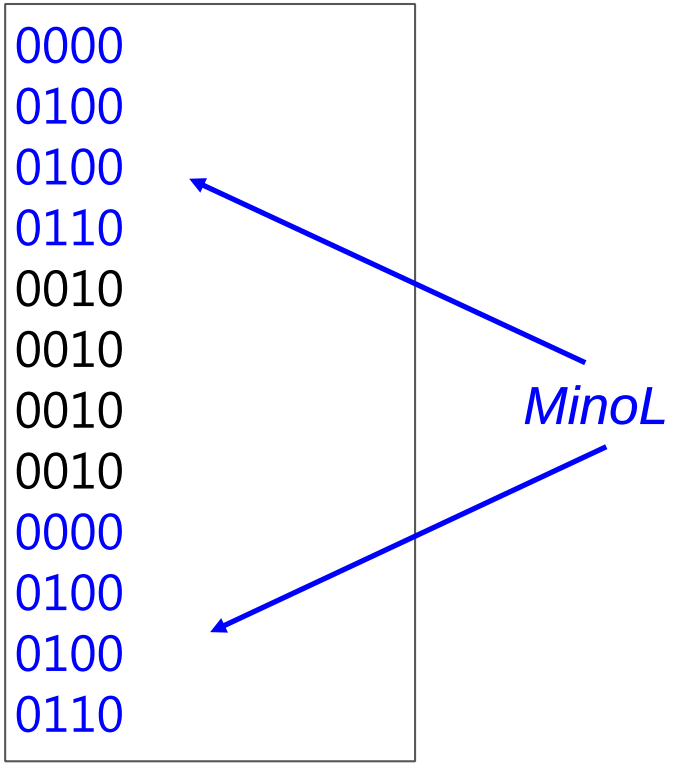
```
1  #include <cstdlib>
2  #include "Minos.h"
3  #define NUM_MINO 3
4  #define MINO_S 0
5  #define MINO_I 1
6  #define MINO_L 2
```

genMino.cpp

```
 7  Mino * genMino()
 8  {
 9    int mino_type;
10    Mino * ptr;
11
12    mino_type = random() % NUM_MINO;
13
14    switch(mino_type) {
15      case MINO_S:
16        ptr = new MinoS;
17        break;
18      case MINO_I:
19        ptr = new MinoI;
20        break;
21      case MINO_L:
22        ptr = new MinoL;
23        break;
24    }
25    return ptr;
26  }
```

tetris5.cpp (unmodified)

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 #include <ctime>
 4 #include   "genMino.h"
 5 using namespace std;
 6 int main()
 7 {
 8    Mino * mino_ptr;
 9
10    srandom(time(NULL));
11    for(int i=0;i<3;++i)
12    {
13        mino_ptr = genMino();
14        mino_ptr->paint();
15        delete mino_ptr;
16    }
17    return 0;
18 }
```



Virtual Destructors

- A problem can occur when using polymorphism to process dynamically allocated objects of a class hierarchy.
- If a derived-class object with a nonvirtual destructor is destroyed explicitly by applying the `delete` operator to a base-class pointer to the object, the C++ standard specifies that the behavior is undefined.
- The simple solution to this problem is to create a `virtual` destructor in the base class.
- This makes all derived-class destructors `virtual` *even though they do not have the same name as the base-class destructor*.

Polymorphism Without Virtual Destructors

```
class A {
public:  A() { cout << "A ctor" << endl; }
         ~A() { cout << "A dtor" << endl; }
};
class B: public A {
public:  B() { cout << "B ctor" << endl; }
          ~B() { cout << "B dtor" << endl; }
};
int main()
{
        A * aPtr ;
        B * bPtr ;
        aPtr = new A();
        delete aPtr;
        aPtr = new B();
        delete aPtr;
        bPtr = new B();
        delete bPtr;
        return 0;
}
```

Output:
A ctor
A dtor
A ctor
B ctor
A dtor
A ctor
B ctor
B dtor
A dtor

Polymorphism With Virtual Destructors

```
class A {
public:  A() { cout << "A ctor" << endl; }
         virtual ~A() { cout << "A dtor" << endl; }
};
class B: public A {
public:  B() { cout << "B ctor" << endl; }
         virtual ~B() { cout << "B dtor" << endl; }
};
int main()
{
        A * aPtr ;
        B * bPtr ;
        aPtr = new A();
        delete aPtr;
        aPtr = new B();
        delete aPtr;
        bPtr = new B();
        delete bPtr;
        return 0;
}
```

Output:
A ctor
A dtor
A ctor
B ctor
B dtor
A dtor
A ctor
B ctor
B dtor
A dtor

Notice on Virtual Constructor / Destructor

- If a class has `virtual` functions, provide a `virtual` destructor, even if one is not required for the class. This ensures that a custom derived-class destructor (if there is one) will be invoked when a derived-class object is deleted via a base class pointer.
- Constructors cannot be `virtual`. Declaring a constructor virtual is a compilation error.

```
$ g++ -o virtual_constructor virtual_constructor.cpp
virtual_constructor.cpp:6: error: constructors cannot be declared virtual
```