Operators, Functions and Static Members

Lecture Objectives

- Apply C++'s extensions to the syntax and semantics of functions
  - Defining conversion functions using constructors
  - Overloading predefined operators
  - Using friends for restricted access to private members
- Use static members and functions to encapsulate global data

Type Conversion

- C and C++ provide implicit type conversion between predefined types
  - Sometimes called “mixed mode” arithmetic
  - “shorter” types are converted to “longer” types in an expression “a op b” to make a and b’s types match
  - “longer” types are truncated in an assignment
  - Anything non-zero is treated as false in a bool conversion

Type Conversion (cont.)

```
...
bool b;
char c = 'a';
int i;
float f = 3.6;

int main() {
  cout << c+i;
  c = 1; cout << (int)c;
  i = f; cout << i;
  b = f; cout << b;
  return 0;
}
```

Explicit type conversion to print c's ASCII integer value

Output is: 97 1 3 1

Type Conversion (cont.)

- Implicit type conversion between user-defined types can be useful too
- For example, we might want to implement an “arbitrary precision” decimal number, and mix it in expressions with predefined type

Type Conversion (cont.)

Implementation example:
123.6 stored as a vector of 4 `int`s, right to left, with `dp`, the decimal point location in digit as 1

```
// File decimal.h
#include <vector>
#include <string>
class Decimal {
  vector<int> digit;
  int dp;
public:
  Decimal(int);
  Decimal(float);
  Decimal(const std::string&);
  Decimal& addTo(const Decimal&);
  ...
} /
```

```
#include "decimal.h"
...
int main() {
  Decimal d1("3.2");
  Decimal d2("6.97");
  cout << d1.addTo(d2);
  ...
  return 0;
}
```

`addTo` has the meaning of +=
Output operators will be defined later

Output, and d1, should now be:
10.17

Type Conversion (cont.)

- In C++, a one argument constructor specifies an implicit conversion
  - In class C, a constructor C(D dobj), specifies an implicit conversion, by a constructor call from D to C, so cobj = dobj results in cobj = C(dobj)
  - This works in any expression
- Now we can write expressions that mix Decimals, strings, ints, etc.

Type Conversion (cont.)

```
// File decimal.h
#include <vector>
#include <string>
class Decimal {
  vector<int> digit;
  int dp;
public:
  Decimal& addTo(const Decimal&);
  Decimal(int);
  Decimal(float);
  Decimal(const std::string&);
  ...
} ;
```

3 constructors specify conversions from int, float, or std::string to Decimal

```
#include "decimal.h"
...
int main() {


  Decimal d1 = "3.2";
  cout << d1.addTo(6.97);
  ...
  return 0;
}
```

Same as `Decimal d1 = Decimal("3.2");`

Same as `d1.addTo(Decimal(6.97));`

Type Conversion (cont.)

- Implicit conversion can be unsafe
  - The conversions above are not “obvious” in the main() program above
  - Worse, unintended conversions can occur

```
// File polygon.h
#include "point.h"
class Polygon : public IGObject {
  Point* vlist;
  const int nv;
public:
  Polygon(int init_nv, Plist* init_v = 0);
  void draw();
};
```

Constructor specifies a conversions from `int` to `Decimal`

```
#include "polygon.h"
...
int main() {
  Polygon p3(...);
  ...
  Polygon p1 = 3;
  ...
  return 0;
}
```

We intended to write:
`Polygon p1 = p3` (calling a copy constructor)
Instead, `Polygon p1` is now initialized to a new `Polygon` with 3 points

Type Conversion (cont.)

- To prevent implicit conversions, constructors can be declared explicit
  - Now a cast or explicit constructor call is needed
  - Most single argument constructor should be declared explicit for safety

```
// File polygon.h
#include "point.h"
class Polygon : public IGObject {
  Point* vlist;
  const int nv;
public:
  explicit Polygon(int init_nv,
    Plist* init_v = 0);
  void draw();
};
```

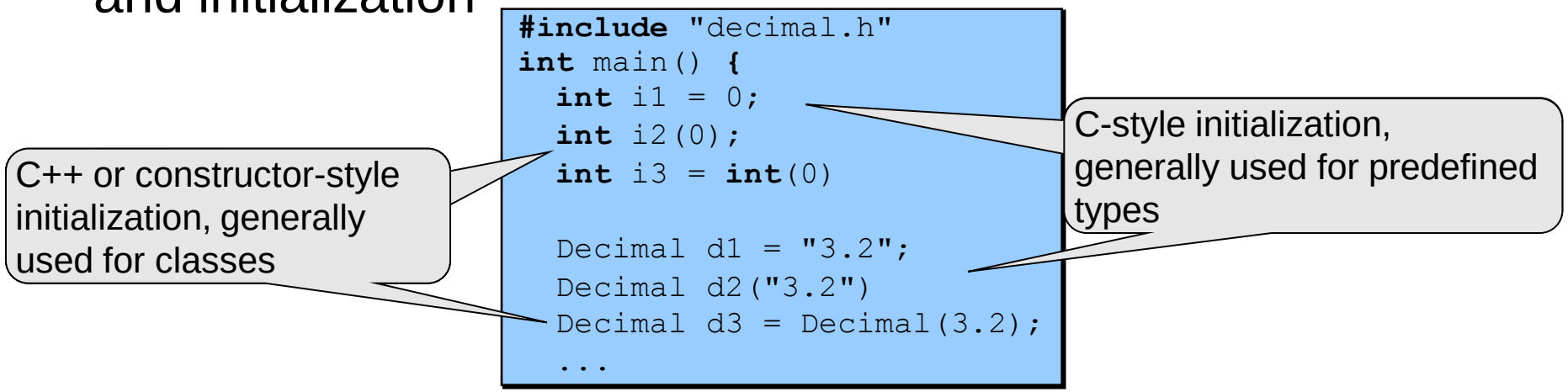
explicit keyword on constructor declaration

```
#include "polygon.h"
...
int main() {
  Polygon p3(...);
  ...
  Polygon p1 = 3;
  ...
  return 0;
}
```

Syntax error here now. If we want to call a constructor here it must be done explicitly:
`Polygon p1 = Polygon(3)`

Type Conversion (cont.)

- C++ provides several alternatives for constructors calls and initialization



- Constructors allows us to convert from predefined types to `Decimal`
  - But how can we convert from Decimal to predefined types? e.g. to allow `float f2 = d2;` or `float f3 = (float)d3;`

Type Conversion (cont.)

- A conversion operator converts from class C to T

```
C::operator T() { ... return tvalue; }
```

  - Now we can mix type conversions both ways between `Decimal` and predefined types

```
// File decimal.h
#include <vector>
#include <string>

class Decimal {
  vector<int> number;
  int point_loc;
public:
  Decimal& addTo(const Decimal&);
  Decimal(int);
  Decimal(float);
  Decimal(const std::string&);
  operator float() const {
    float result;
    ... // convert number
    return result;
  }
} ;
```

Note return type is after **operator** keyword, no declaration arguments allowed

```
#include "decimal.h"
...
int main() {

  Decimal d1 = "3.2"
  float f1 = 1.2 *
d1;
  cout << f1 << endl;
  cout << d1 << endl;
  ...
  return
}
```

Implicit conversion operator call; same as `1.2 * float(d1)`

Output operator not needed as and implicit conversion operator call will occur; same as `cout << float(d1)`

Type Conversion (cont.)

- The stream standard libraries provide a conversion operators to `void*`
  - Which returns non-zero if the stream state is good
  - Useful for control flow
- But the `std::string` class does not provide a conversion operator to `const char*`
  - Conversion operators, like implicit constructor calls, can be unsafe

Type Conversion (cont.)

```
...
int main() {
  char c;
  while (cin >> c) {
    ... // process input
        // until EOF
  }
  std::string s1;
  ...
  cout << s1.c_str();
  ...
}
```

Implicit conversion operator call;
same `((void*) (cin >> c))`
Conversion provided is to **`void*`** not **`bool`**, as **`void*`** is "safer" than **bool**

No conversion operator, instead an explicit member function call returns **`const char*`**

Friends

- Suppose that we wish to control allocation of `Polygon`s
  - So that they were allocated/deallocated only by a `SceneManager` class
- A class can declare its members
  - `public` - access by anyone
  - `protected` - access by child classes
  - `private` - no access by anyone
  - Unlike Java, there is no `package` or `namespace`, only access

Friends (cont.)

- To restrict access to a single class or function, C++ provides `friend` declarations
  - A class can declare one or more classes or functions as `friend`s
  - `friend`s can access all class members - `private`, `protected`, or `public`

Friends (cont.)

- The `Polygon` constructor can be private and a Scenemanager allocation function a friend

```
// File scenemanager.h
#include "point.h"
class SceneManager {
  std::list<IGObject*> objlist;
public:
  IGObject& AllocateGObject(
    const std::string&, Plist* init_p);
  ...
};
```

```
// File polygon.h
#include "point.h"
#include "scenemanager.h

class Polygon :
  public IGObject {
  Point* vlist;
  const int nv;
  friend IGObject& SceneManager::Allocate
    const std::string&, Plist* init_p);
  explicit Polygon(int init_nv, Plist* in
public:
  void draw();
};
```

Polygon declares `SceneManager`'s Function `AllocateGObject()` a friend so its definition can access constructor (and `vlist`, `nv`) function declarations must match friend declaration

Friends (cont.)

- Or `Polygon` can declare the entire `SceneManager` class a `friend`
  - Any `SceneManager` member can now access any `Polygon` member

```
// File scenemanager.h
#include "point.h"

class SceneManager {
  std::list<GObject*> objlist;
public:
  IGObject& AllocateGObject (
    const std::string&, Plist* init_p);
  void DeleteGObject(const std::string&)
  ...
};
```

```
// File polygon.h
#include "point.h"

class Polygon :
  public IGObject {
  Point* vlist;
  const int nv;
  friend class SceneManager;
  explicit Polygon(int init_nv, Plist* init_v = 0);
public:
  void draw();
};
```

Polygon declares `SceneManager` a **`friend`**, so `AllocateGObject()` and `DeleteGObject()` can access any `Polygon` member

Friends (cont.)

- `friend`s can compromise the encapsulation of a class, so are only used when necessary:
  - A nonmember (*global*) function needs access to private data
    - Sometimes operators are declared as non-members
  - A function needs to access the private data of more than one class
  - A class needs to allow another class access to private data

Friends (cont.)

- Friendship is not inherited
  - So if `GObject` is a friend of `SceneManager`, that does not make Polygon a friend of `SceneManager`
- Java has no `friend`s ☹
  - But friendship can be imitated in Java by package-level access modifiers

Operators

- An operator is just a convenient alternative syntax for functions of one or two arguments
  - For example, `a+b` could be implemented as a function `call add(a,b)`
  - That is exactly how the C++ compiler handles it
- C++ allows almost all the predefined operators to be overloaded using operator function declarations
  - Even `->`, and unary `*` and `&`!
  - But not the scope and member selection operators: `::` and `.`

Operators (cont.)

- At least one argument to an operator must be a class
  - You cannot redefine the meaning of “1+2”!
- Java does not support any operator overloading, for simplicity
  - .NET’s programming language C# provides operator overloading, but with far more restrictions than C++

Operators (cont.)

- It is often convenient to declare relational operators such as "=="
  - Assignment is the only operator predefined for classes
- For example, to assess "equality" of two movies (`Clip`s):

```
// File movie.cpp"
...
bool Movie::remove(const Clip& cp) {
  bool all_play = true;
  for (ClipSeq::iterator it = seq.begin(); it != seq.end(); ++it)
    if ((*it)->get_info() == cp.get_info()) {
      seq.erase(it);
      return true; // success
    }
  return false; // failure (to find Clip)
}
```

Implied definition of Clip equality as identical string info (calls string == operator)

Operators (cont.)

- It would be much cleaner to define the equality operator in the `Clip` class
  - Responsibility for defining "==" belongs in `Clip`, not clients!

Call `Clip::operator==` `*it` gets a `Clip*` from `seq`, then `**it` is a `Clip`

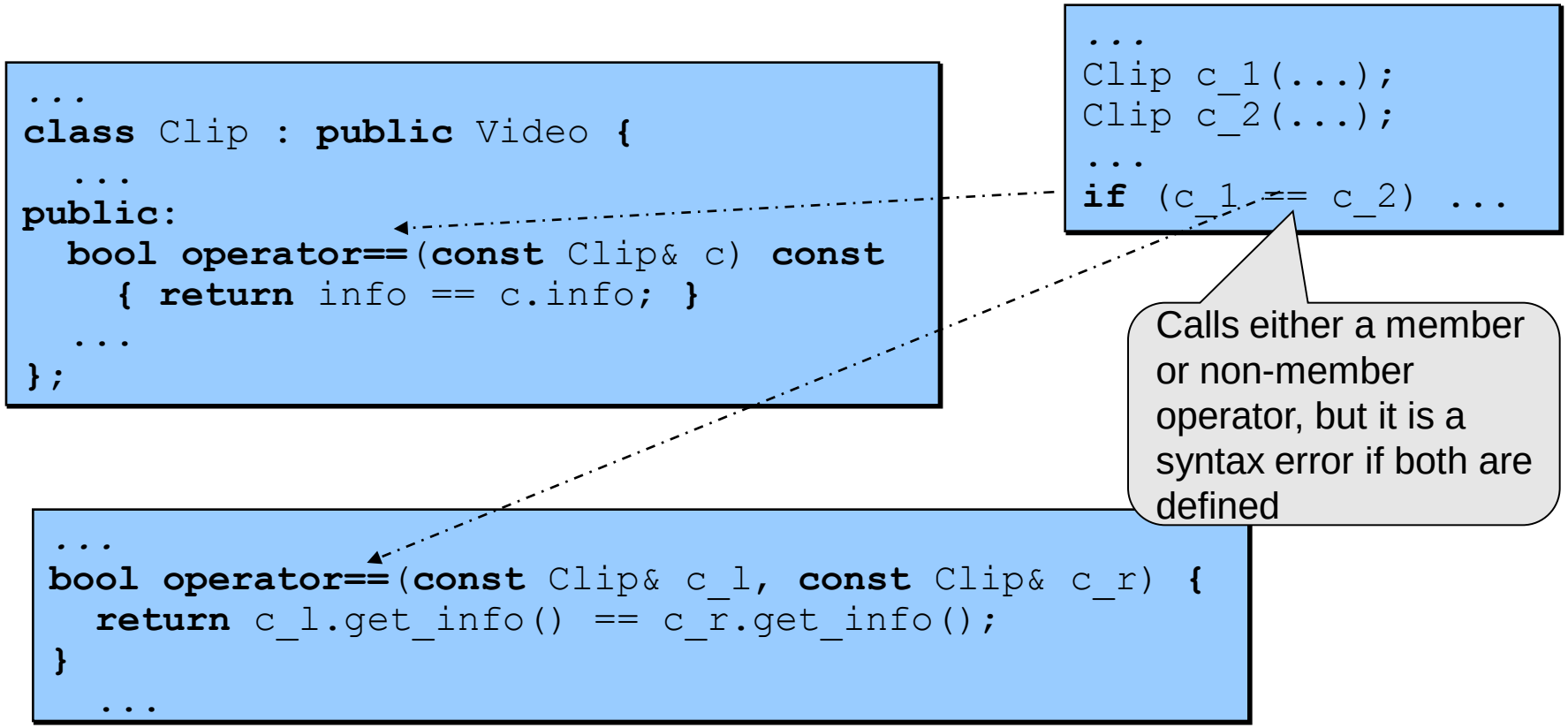
```
// File movie.cpp
...
bool Movie::remove(const Clip& cp) {
  bool all_play = true;
  for (ClipSeq::iterator it = seq.begin();
       it != seq.end(); ++it)
    if (**it == cp) {
      seq.erase(it); return true;// success
    }
  return false; // failure (to find Clip)
}
```

```
// File clip.h
#include "video1.h"
#include <string>
class Clip : public Video {
  std::string filename;
  std::string info;
public:
  bool play() const;
  const std::string& get_info() const { return info; }
  bool operator==(const Clip& c) const
    { return info == c.info; }
  Clip(const std::string& init_filename, const std::string& init_info);
};
```

Operators (cont.)

- Function call syntax distinguishes member from non-member functions
  - For example, "`object.doit()`" or "`doit(object)`"
- But operator function call syntax does not distinguish these
  - Most operators can be members or non-members

Operators (cont.)



Operators (cont.)

- Arithmetic operators can be very handy for numerical classes
  - Such as matrices, vectors, or our `Decimal` class
  - "`a += b`" is more readable than "`a.addTo(b)`"

```
// File decimal.h
#include <vector>
#include <string>
class Decimal {
  std::vector<int> digit;
  int dp;
public:
  ... // constructors
  Decimal& operator+=(const Decimal&);
  Decimal& operator++();
  bool operator==(const Decimal&) const;
  // other operators such as !=, *=, ...
  ...
} ;
```

Implementation wastes some storage as digit[i] is in the range 0..9, could be stored in a char. The "decimal point", dp, is the location of the decimal point, e.g. 2 in "1.63"

```
#include "decimal.h"
...
int main() {
  Decimal d1("3.2");
  Decimal d2("6.97");
  ++d2;
  d2 += d1;
  cout << d2;
  ...
  return 0;
}
```

Test ++ and += operators, output 11.17
Output operators will be defined later

Operators (cont.)

```
// File decimal.cpp
  ...
Decimal& Decimal::operator+=(const Decimal& rd) {
  std::vector<int> new_digit;
  int new_dp = (dp > rd.dp)? dp : rd.dp;
  int i = dp - new_dp;
  int ri = rd.dp - new_dp;
  int carry = 0;

  while (i < digit.size() || ri < rd.digit.size()) {
    int next_digit = carry;
    if (i >= 0 && i < digit.size()) next_digit += digit[i];
    if (ri >= 0 && ri < rd.digit.size())
      next_digit += rd.digit[ri];
    carry = (next_digit > 9) ? 1 : 0;
    new_digit.push_back(carry ? next_digit - 10 : next_digit);
  }
  if (carry) new_digit.push_back(1);
  this->digit = new_digit;
  this->dp = new_dp;
  return *this;
}
```

Implemention of operator +=

The “Right hand side Decimal”, rd and this Decimal, need to be “aligned” by the decimal point before addition

Operators (cont.)

- Why define the operator "`+=`" rather than the common operator "`+`"?
- The operator "**+**" returns an object "by value"
  - This is inefficient, except for small objects
  - For example: "`d = a+b+c`" would create 2 temporary objects:
    - "`(a+b)`" followed by "`(a+b)+c`" which is copied into `d`
    - Temporary value objects are automatically reclaimed, but inefficient
  - It is more efficient to write this using "`+=`":
    - "`d = a; d+= b; d += c`"
    - Now no temporary objects are allocated
- So standard library classes, such as `std::string`, do not implement "`+`"

Operators (cont.)

- Modern C++ style suggests avoiding use operator functions, except for
  - Classes where they are “natural”, such as `Decimal`
    - But not for our `Account`, `Video`, or `GraphicObject` classes
  - Using the standard library

Operators (cont.)

- The standard library
  - Provides operator functions:
    - Input/output operators: `>>`, `<<`
    - Increment and decrement operators for iterators: `++, --`
    - The dereference operator for iterators: `*`
    - The subscript operator for some containers: `[ ]`
  - Requires operator functions for application classes that use some Standard Library classes
    - Function call operator for *algorithms*: `()`
    - Comparison operators for containers: `==, <`
    - Assignment and copy constructor for containers

Operators (cont.)

- It is often useful to have input or output operators for application classes
  - But these cannot be member functions, as “a op b” must be declared as either “ClassA::operator op(ClassB)”
    or “operator op(ClassA, ClassB)”

```
#include "decimal.h"
...
int main() {
  Decimal d1("3.2");


  cout << "Decimal d1 is: " << d1 << '\n';
  ...
  return 0;
}
```

Left-Hand Side (LHS) of any output operator is an ostream&
return result is always an ostream&
(the same stream as on the LHS)

Operators (cont.)

- As they are non-members, input and output operator functions are usually friends

```
// File decimal.h
...
using namespace std;
class Decimal {
  std::vector<int> digit;
  int dp;
public:
  // constructors
  friend ostream& operator<<(ostream& os, const Decimal& d);
  ...
};
ostream& Decimal::operator<<(ostream& os, const Decimal& d) {
  for (int i=digit.size(); i>=0; --i) {
    os << digit[i];
    if (i-digit.size() == dp)
      os << '.';
  }
  return os;
}
```

friend prefix on the declaration in the class, not on the definition outside

Output digits, left to right, and the decimal point then return the same stream, os, that was on the RHS of the operator

Operators (cont.)

- The predefined prefix and suffix "`++`" operators are quite different
  - "`++i`" - increment `i`, return `i` by reference (so "`++i = j`" is legal)
  - "`i++`" - return `i` by value, then increment `i` (so "`i++ = j`" is illegal)
- Standard library operator functions behave like predefined operators

Operators (cont.)

- Application defined operator functions should mimic predefined syntax and semantics

```
...
class Decimal {
  ...
  Decimal& operator++() {
    // add one to digit[dp]
    return *this;
  }
  Decimal operator++(int) { // dummy int arg, C++ syntax
    Decimal old_value(*this); // copy this
    ++(*this);
    return old_value;
  }
};
```

Prefix operator++

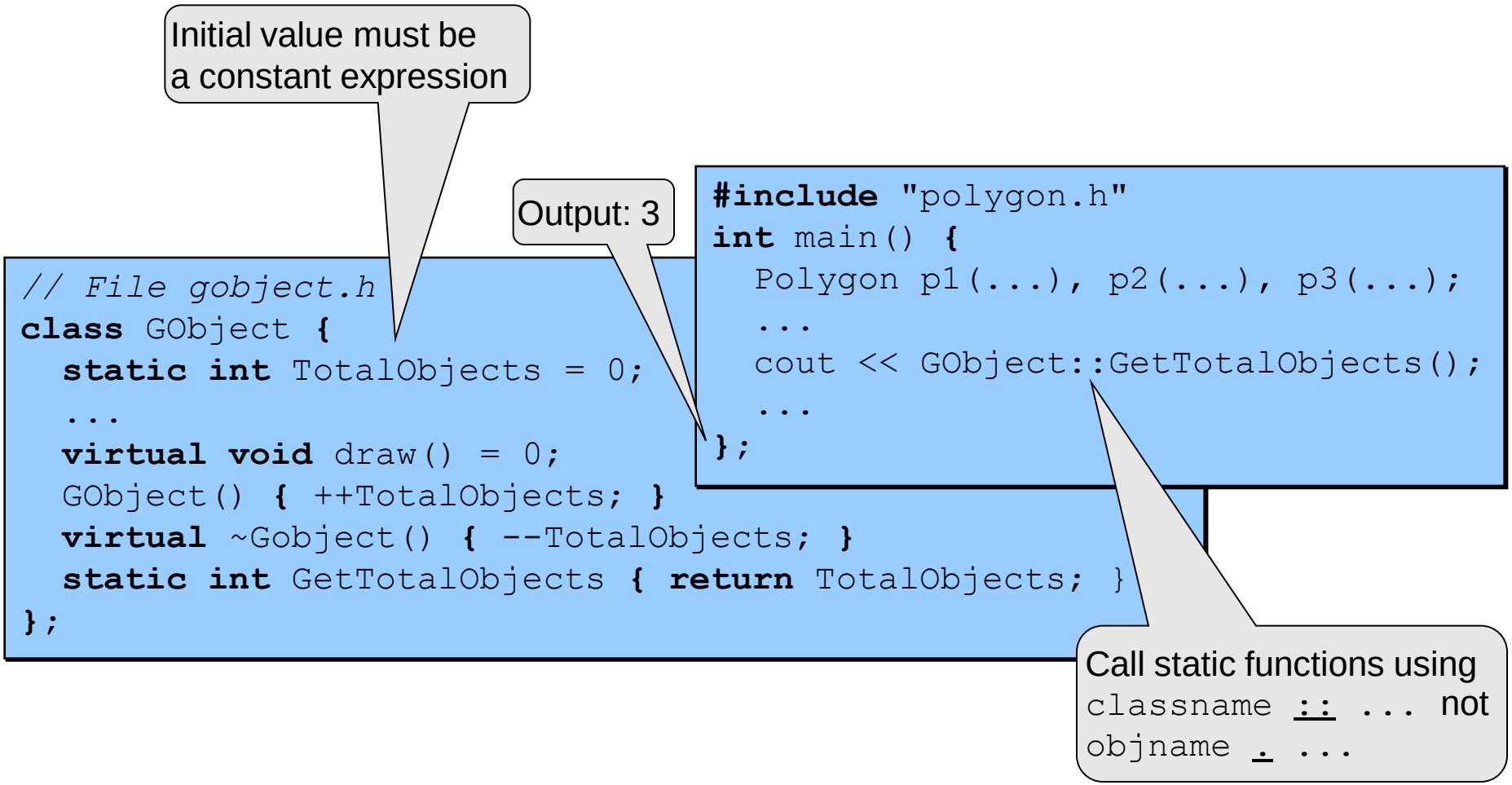
Postfix operator++

Call prefix operator ++ then return old value

Static Members

- Each object has its own data members
  - But sometimes data members are “shared” among class members
    - Such as the total number of `GraphicObject`s or the total amount on deposit in all `Account`s
- static members allow us to declare member data and functions that are shared by all class instances
  - Java and the .NET languages all support static, or “class” data

Static Members (cont.)



Static Members (cont.)

- Java, and .NET's, meaning of `static` differs subtly from C++
  - C++ static data is initialized at load time, whereas Java and .NET static data is initialized at run time when a class is first accessed
  - C++ static data that needs to be initialized to a run time expression often needs to a use coding idioms that mimics load time initialization

Static Members (cont.)

```
// File gobjectmanager.h
class GObjectManager {
  static GObjectManager S
  static int TotalObjects;...
  GObjectManager() {
    TotalObjects = loadObjectsfromFile();
    ...
  }
public:
  static int GetTotalObjects { return TotalObjects; }
  static GObject* GraphicObjectFactory(...);
...
};
```

```
// file gobjectmanager.cpp
#include "gobjectmanager.h"
int GObjectManager::TotalObjects = 0;
GObjectManager GObjectManager::Singleton;
```

If not defined in the .h file, static members must be defined in the .cpp file

GObjectManager's constructor called by Singleton initializes the other static data members

A "factory" is a function or class that encapsulates creation of objects of different types

Static Members (cont.)

- The example analysed is an instance of the singleton design pattern
  - Only one instance of the class `ObjectManager` exists
  - The `ObjectManager` singleton is implemented using a private constructor and a single private instance of the class `Singleton`
  - Singleton initializes all the other static data members of `ObjectManager`
    - The constructor call can include runtime checks that would be difficult if each static data member was initialized separately
- In the example above, we separated the static data out of the `GObject` class into a new class `GObjectManager`
- **Unrelated members in the same class should be put into distinct classes**