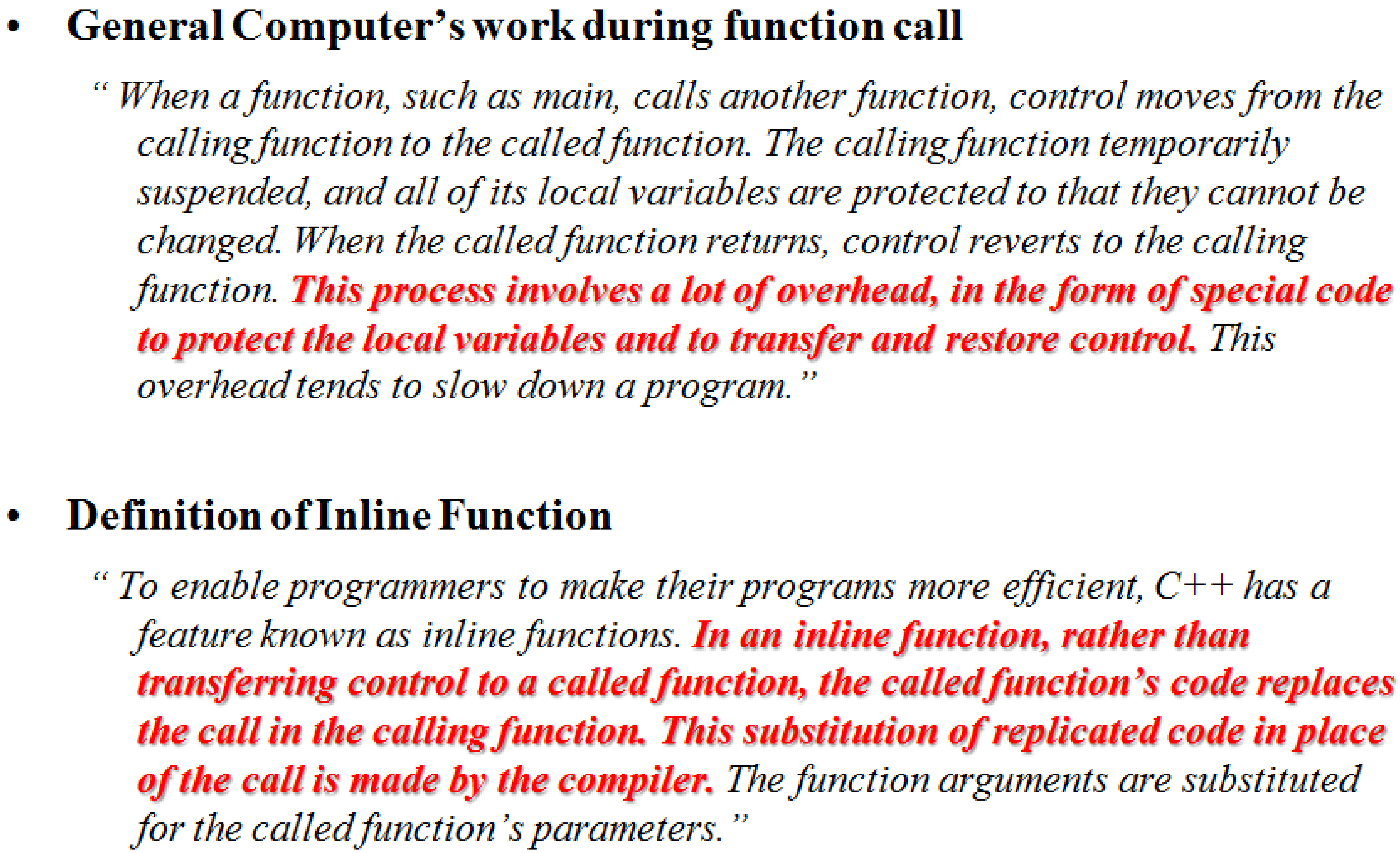
Advanced Object Oriented Programming

More class features and other types

Seokhee Jeon
Department of Computer Engineering
Kyung Hee University
jeon@khu.ac.kr

Inline function



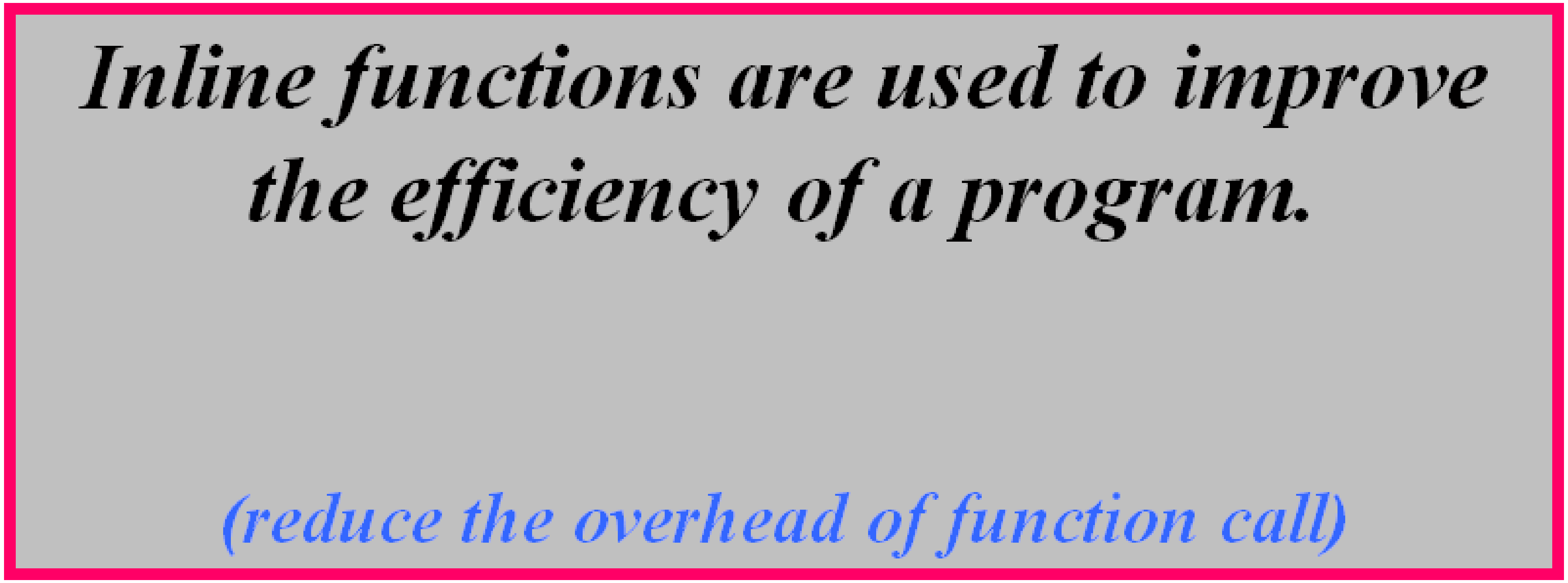
- **General Computer's work during function call**

  " When a function, such as main, calls another function, control moves from the calling function to the called function. The calling function temporarily suspended, and all of its local variables are protected to that they cannot be changed. When the called function returns, control reverts to the calling function. ***This process involves a lot of overhead, in the form of special code to protect the local variables and to transfer and restore control.*** *This overhead tends to slow down a program."*

- **Definition of Inline Function**

  " To enable programmers to make their programs more efficient, C++ has a feature known as inline functions. ***In an inline function, rather than transferring control to a called function, the called function's code replaces the call in the calling function. This substitution of replicated code in place of the call is made by the compiler.*** *The function arguments are substituted for the called function's parameters."*

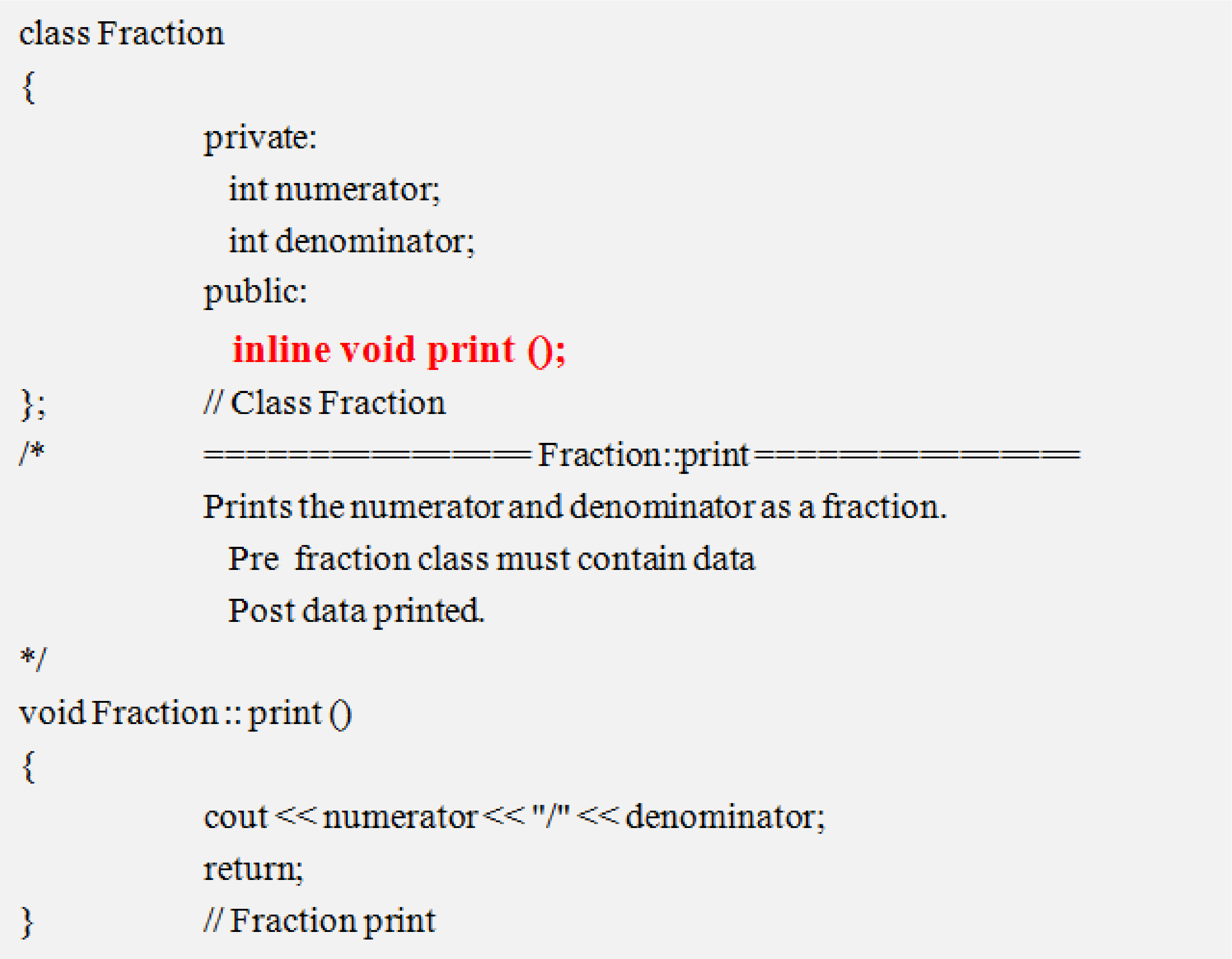
Usage of inline function



Inline functions are used to improve the efficiency of a program.

(reduce the overhead of function call)

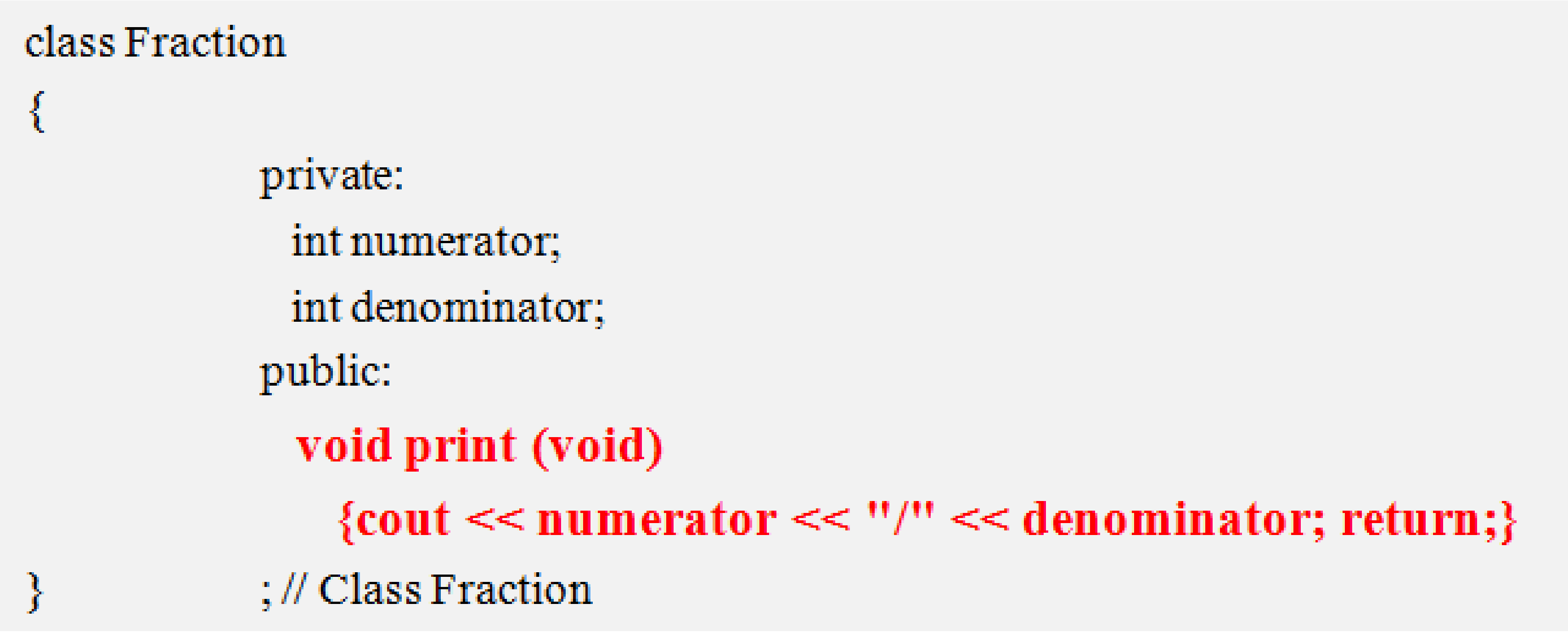
Explicit inline function declaration



```
class Fraction
{
            private:
              int numerator;
              int denominator;
            public:
              inline void print ();
};          // Class Fraction
/*          ============== Fraction::print ==============
            Prints the numerator and denominator as a fraction.
              Pre  fraction class must contain data
              Post data printed.
*/
void Fraction :: print ()
{
            cout << numerator << "/" << denominator;
            return;
}           // Fraction print
```

Implicit inline function declaration

- Defined in a class by coding it in the class declaration itself
- The function is defined without *inline*



```
class Fraction
{
          private:
            int numerator;
            int denominator;
          public:
            void print (void)
              {cout << numerator << "/" << denominator; return;}
}         ; // Class Fraction
```

Non-member Inline Functions

- Simply add the function specifier *inline* to its prototype declaration
- The inline designation does not appear in the function definition

```
//Prototype Declaration
inline void doIt (int num);
...
void doIt (int num)
{
   // Function Definition
   …
} // doIt
```

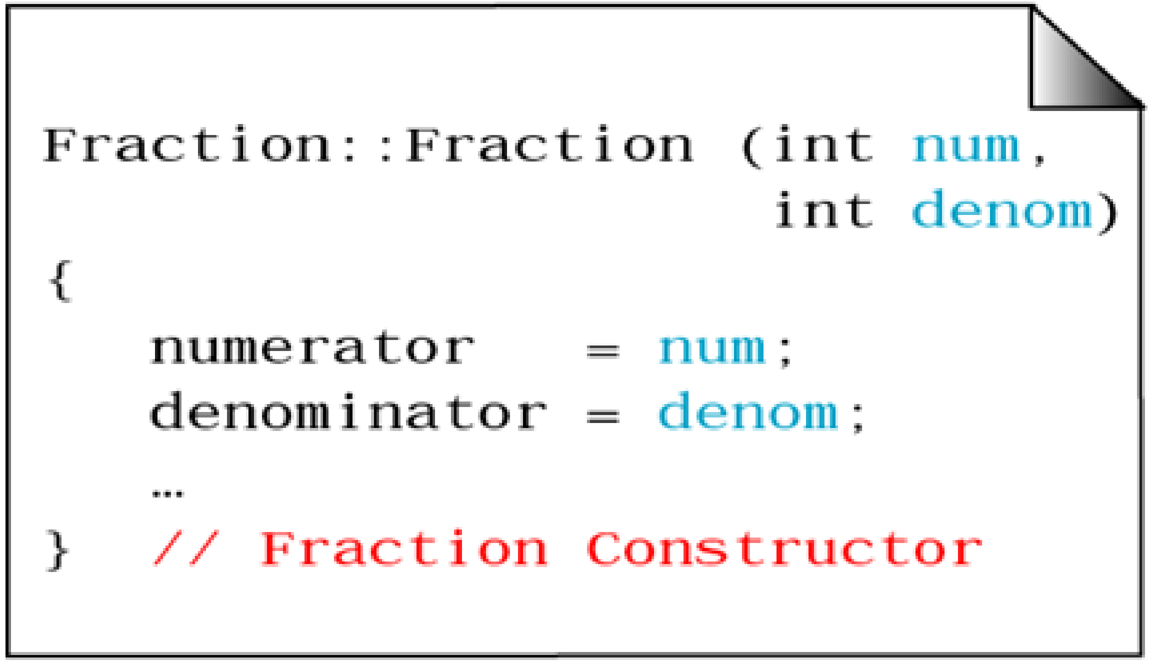
Shortcomings of Inline Functions

- A large function called from many different points increase the size of binary
- Recursive function
- Cache miss
- Compile overhead

* A rule of thumb: use inline functions only if the code is 3-8 lines long

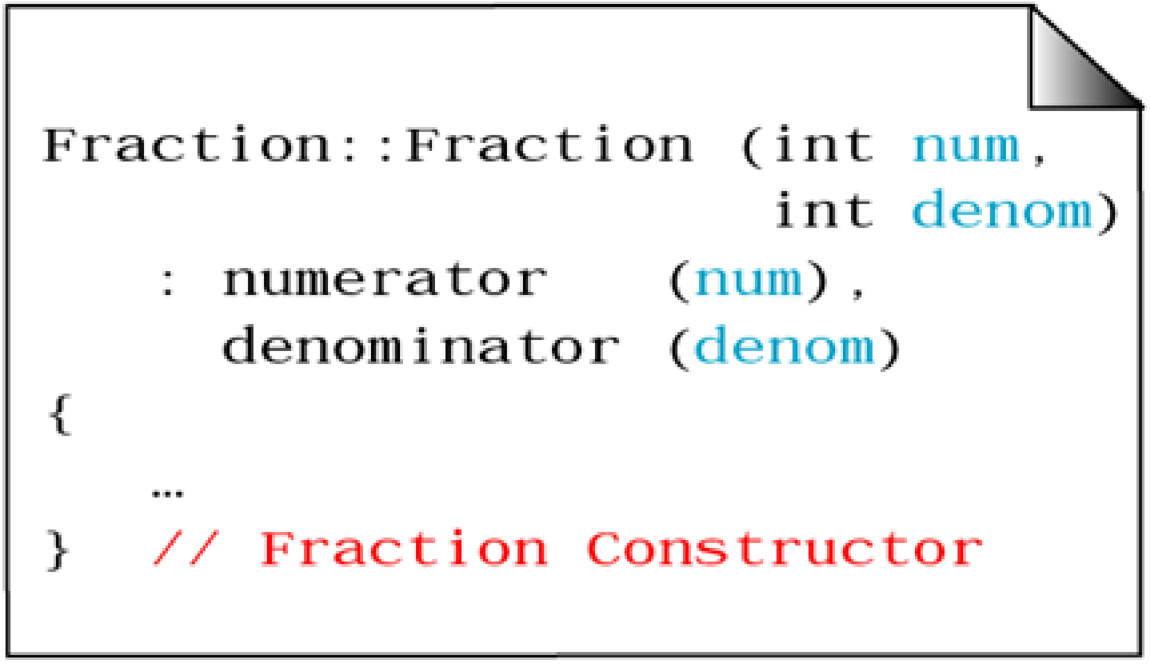
Initialization list

- Same result with different approach to initialize class variables



```
Fraction::Fraction (int num,
                    int denom)
{
    numerator   = num;
    denominator = denom;
    …
}   // Fraction Constructor
```

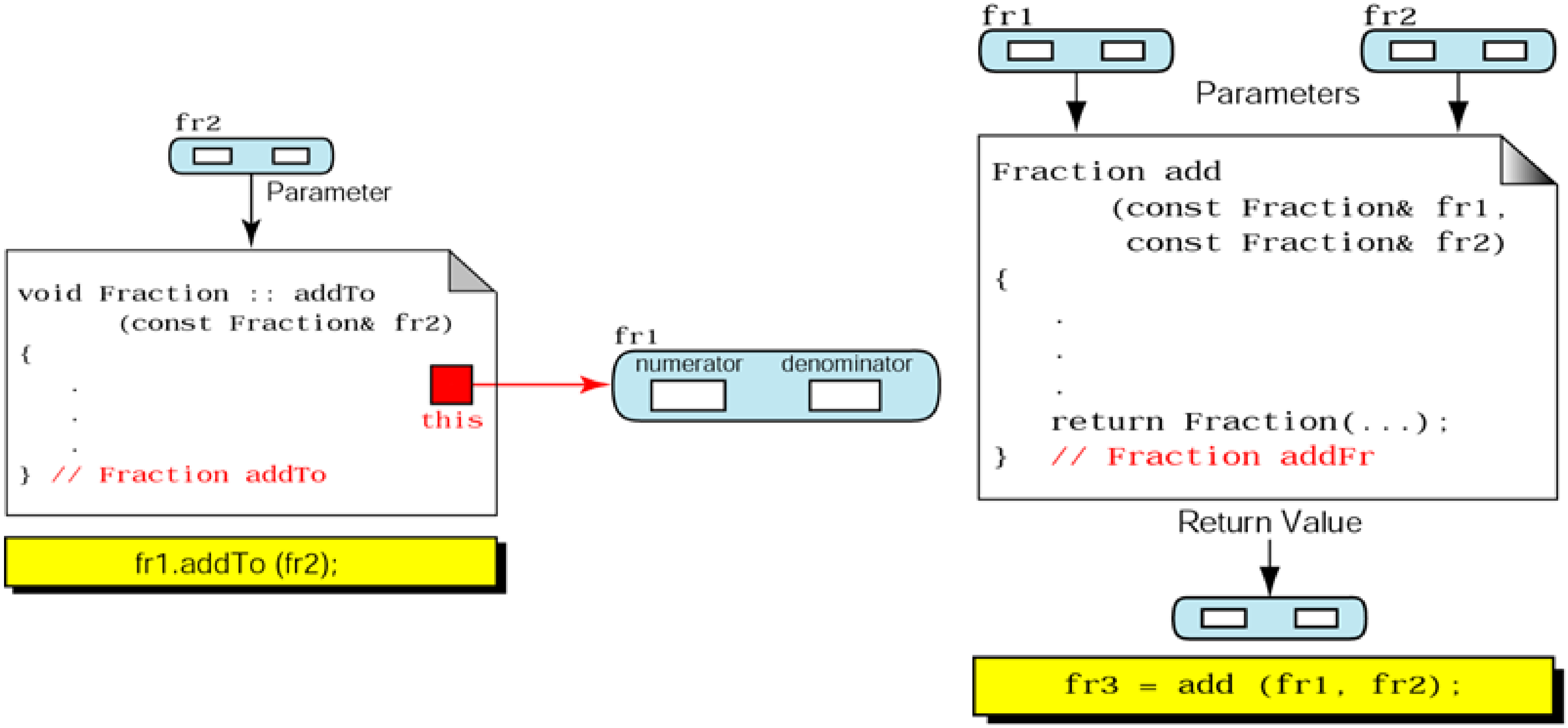
(a) Using the assignment operator



```
Fraction::Fraction (int num,
                    int denom)
    : numerator   (num),
      denominator (denom)
{
    …
}   // Fraction Constructor
```

(b) Using the initialization list

Remind – class Fraction (of chapter 10)



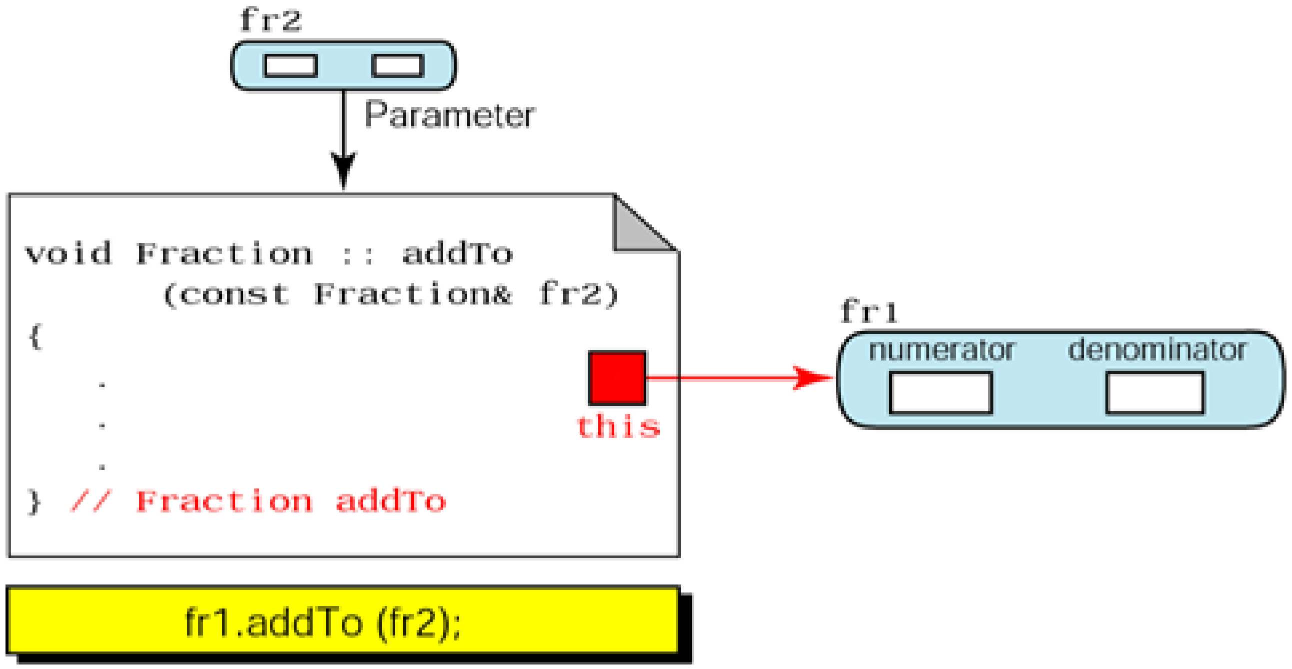
```
void Fraction :: addTo
        (const Fraction& fr2)
{
    .
    .
    .
} // Fraction addTo
```

```
fr1.addTo (fr2);
```

```
Fraction add
        (const Fraction& fr1,
         const Fraction& fr2)
{
    .
    .
    .
    return Fraction(...);
}   // Fraction addFr
```

```
fr3 = add (fr1, fr2);
```

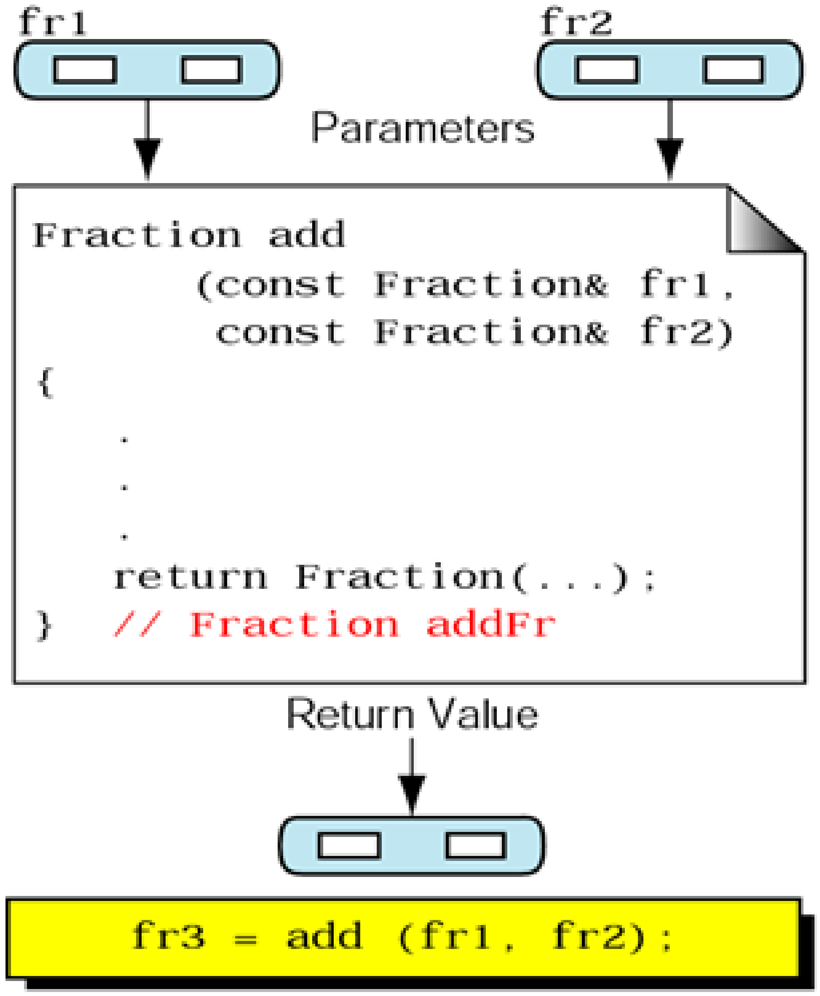
More attractive approach?



```
void Fraction :: addTo
        (const Fraction& fr2)
{
    .
    .
    .
} // Fraction addTo
```

fr1.addTo (fr2);

fr1.addTo(fr2); → fr1 += fr2;



```
Fraction add
        (const Fraction& fr1,
         const Fraction& fr2)
{
    .
    .
    .
    return Fraction(...);
}   // Fraction addFr
```

fr3 = add (fr1, fr2);

fr3=add(fr1,fr2); → fr3=fr1+fr2;

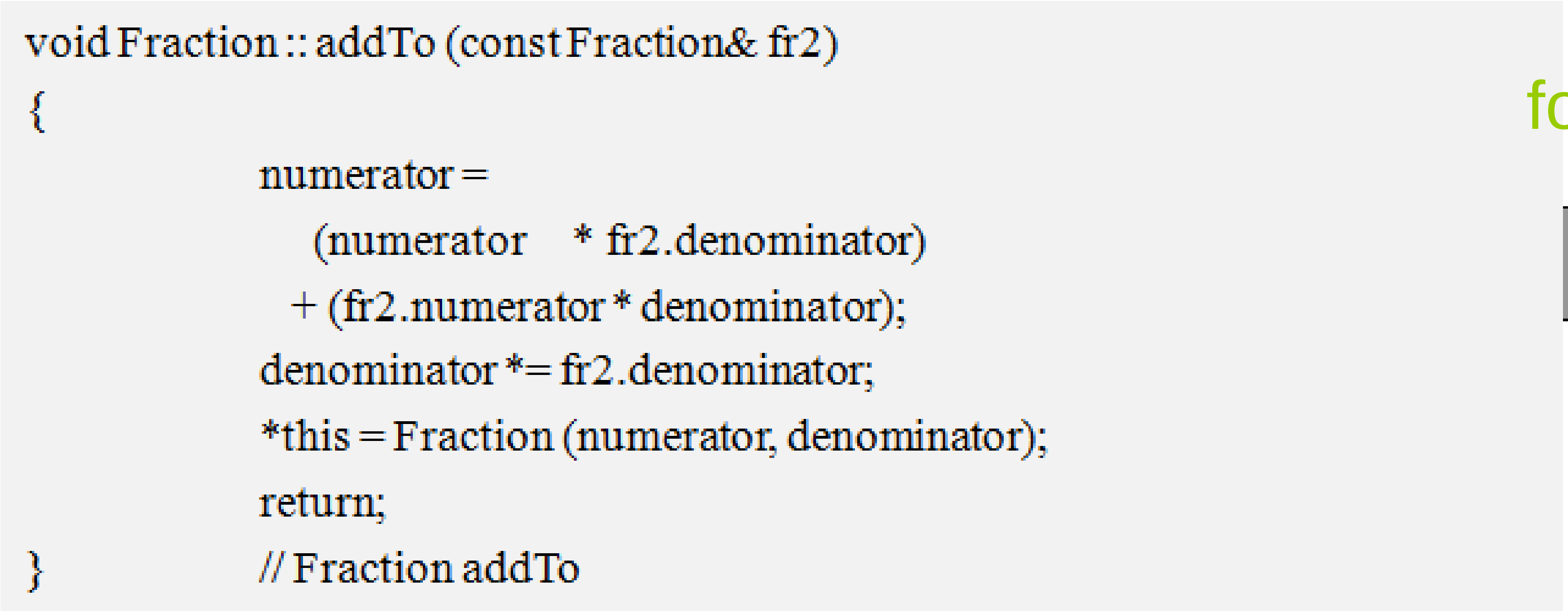
The arithmetic syntax is more natural than the function syntax

Overloading

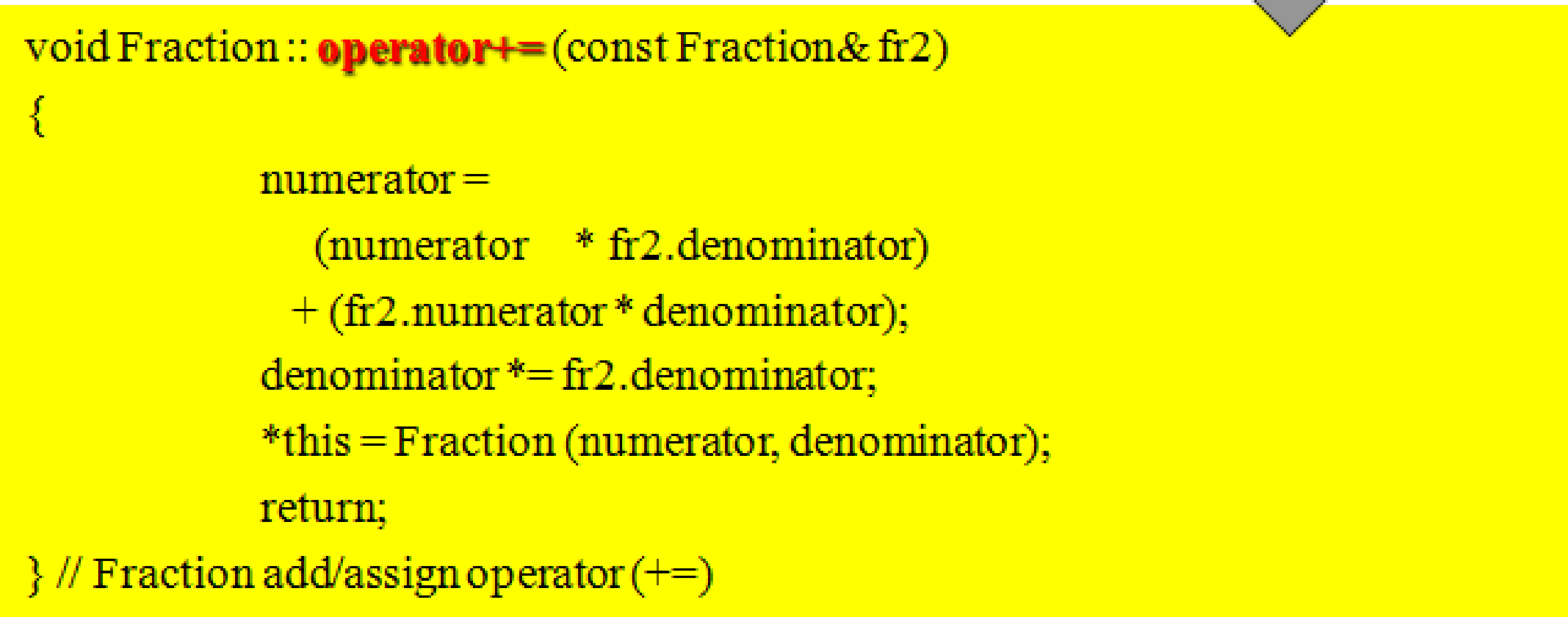
- Definition of two or more functions or operators within the same scope using the same identifier
  - Remind the overloaded constructors

Overloading ‘+=’ operator for class Fraction

HOWTO:
keyword *operator*
followed by the operator
we want to use

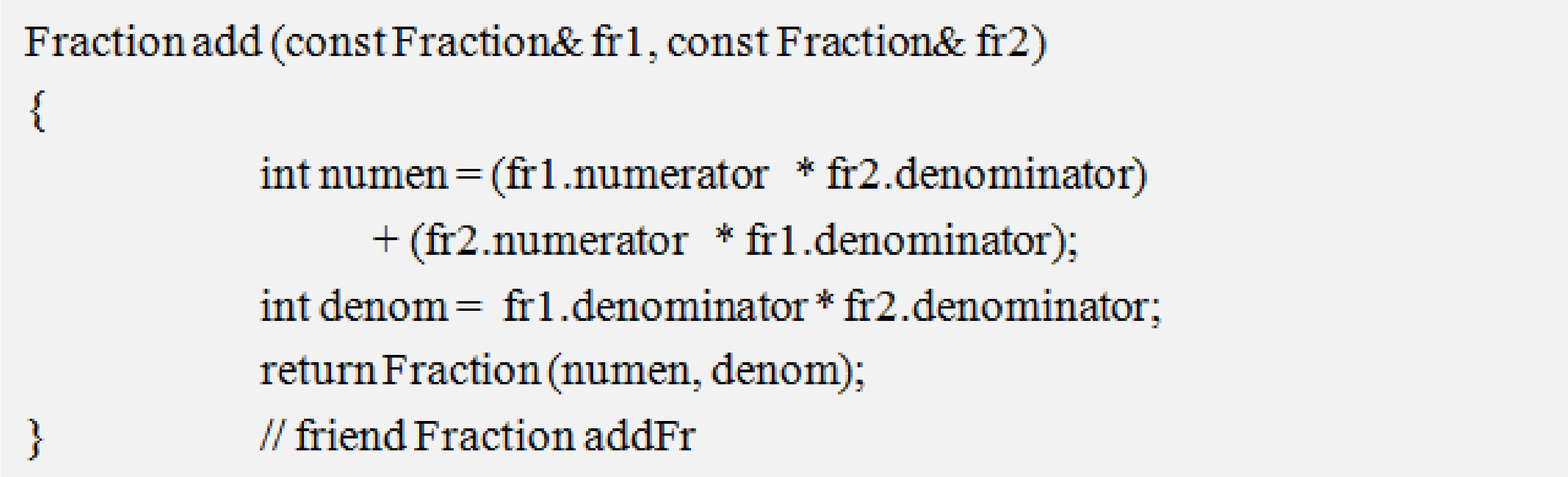


```
void Fraction :: addTo (const Fraction& fr2)
{
                numerator =
                    (numerator    * fr2.denominator)
                  + (fr2.numerator * denominator);
                denominator *= fr2.denominator;
                *this = Fraction (numerator, denominator);
                return;
}               // Fraction addTo
```

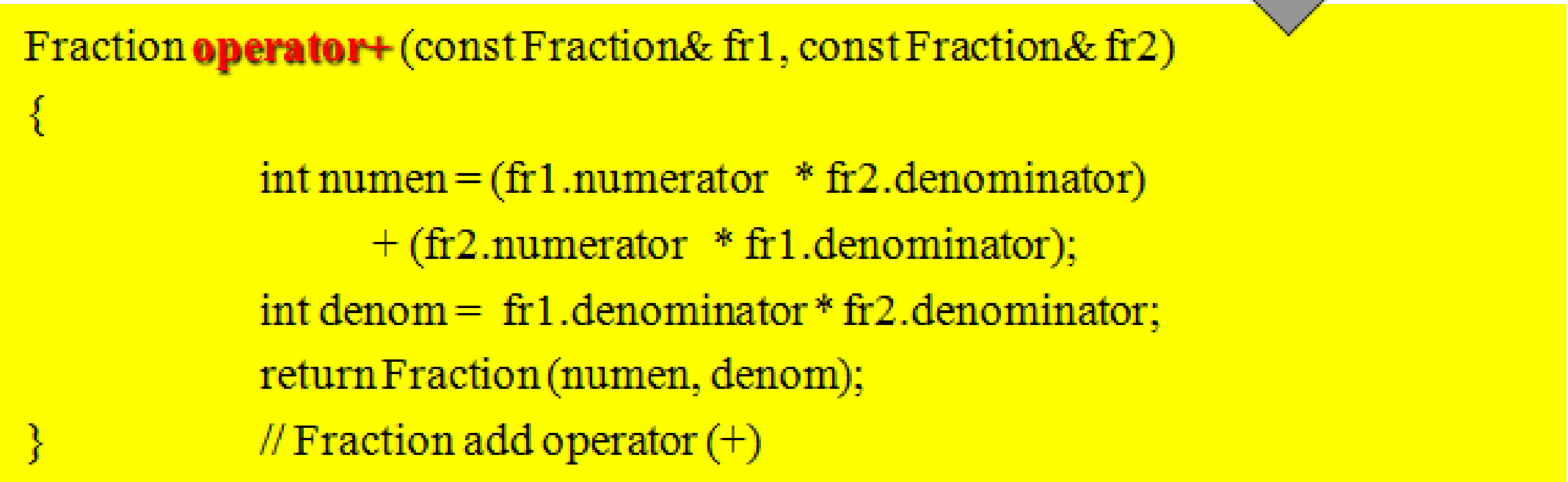


```
void Fraction :: operator+= (const Fraction& fr2)
{
                numerator =
                    (numerator    * fr2.denominator)
                  + (fr2.numerator * denominator);
                denominator *= fr2.denominator;
                *this = Fraction (numerator, denominator);
                return;
} // Fraction add/assign operator (+=)
```

Overloading ‘+’ operator for class Fraction

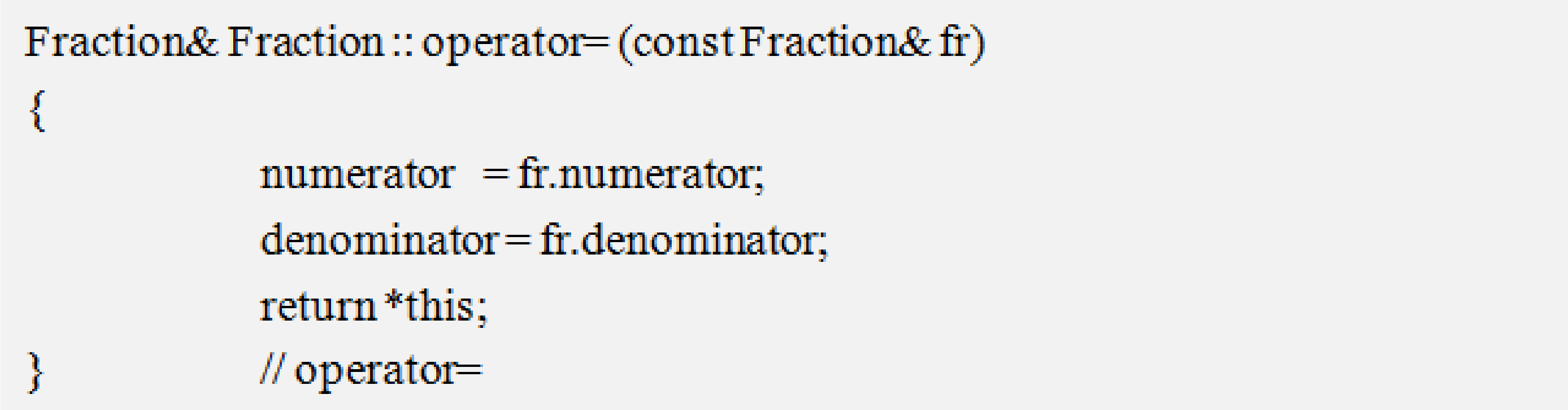


```
Fraction add (const Fraction& fr1, const Fraction& fr2)
{
        int numen = (fr1.numerator  * fr2.denominator)
            + (fr2.numerator  * fr1.denominator);
        int denom =  fr1.denominator * fr2.denominator;
        return Fraction (numen, denom);
}       // friend Fraction addFr
```

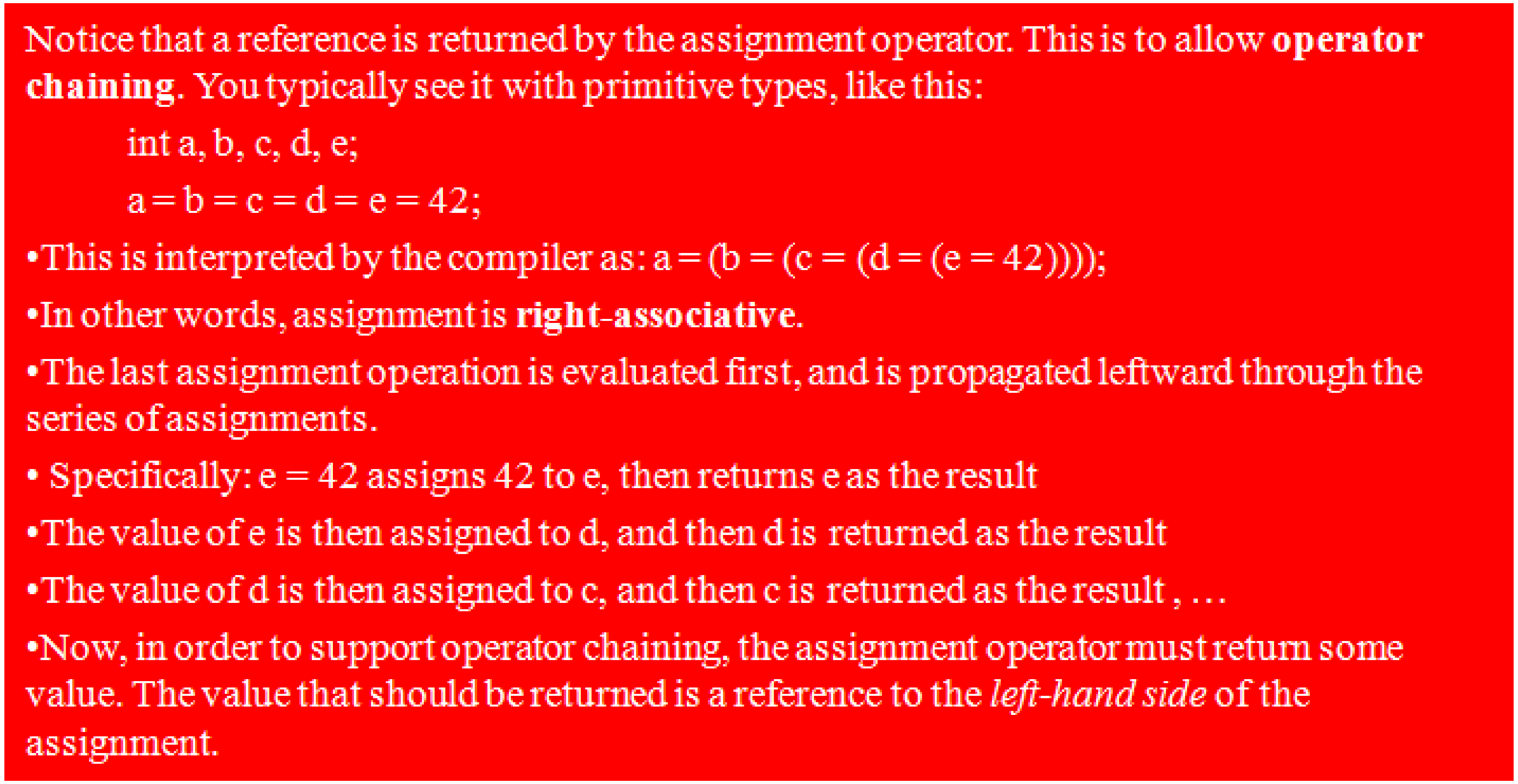


```
Fraction operator+ (const Fraction& fr1, const Fraction& fr2)
{
        int numen = (fr1.numerator  * fr2.denominator)
            + (fr2.numerator  * fr1.denominator);
        int denom =  fr1.denominator * fr2.denominator;
        return Fraction (numen, denom);
}       // Fraction add operator (+)
```

Overloading ‘=’ operator for class Fraction



```
Fraction& Fraction :: operator= (const Fraction& fr)
{
          numerator   = fr.numerator;
          denominator = fr.denominator;
          return *this;
}         // operator=
```

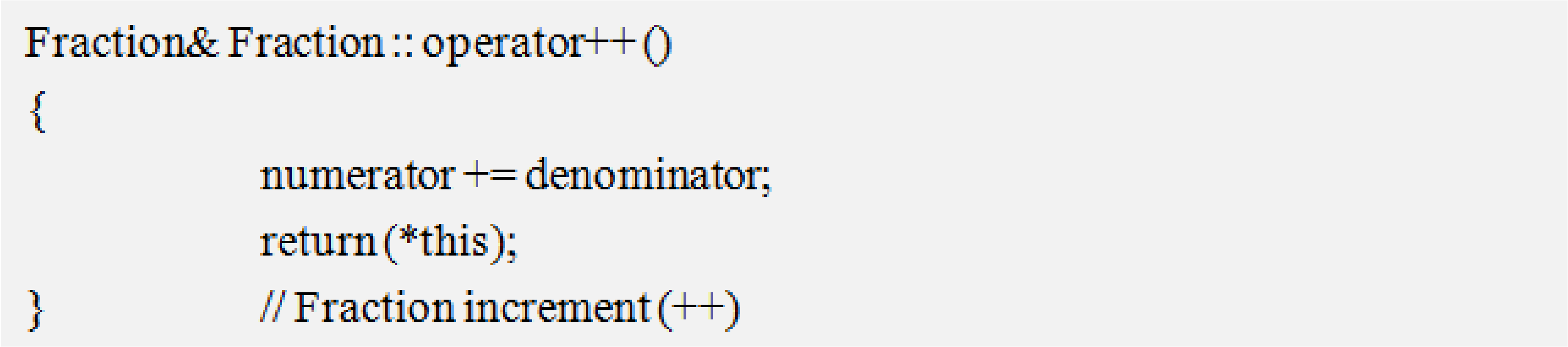


Notice that a reference is returned by the assignment operator. This is to allow **operator chaining**. You typically see it with primitive types, like this:

```
     int a, b, c, d, e;
     a = b = c = d = e = 42;
```

- This is interpreted by the compiler as: a = (b = (c = (d = (e = 42))));
- In other words, assignment is **right-associative**.
- The last assignment operation is evaluated first, and is propagated leftward through the series of assignments.
- Specifically: e = 42 assigns 42 to e, then returns e as the result
- The value of e is then assigned to d, and then d is returned as the result
- The value of d is then assigned to c, and then c is returned as the result , …
- Now, in order to support operator chaining, the assignment operator must return some value. The value that should be returned is a reference to the *left-hand side* of the assignment.

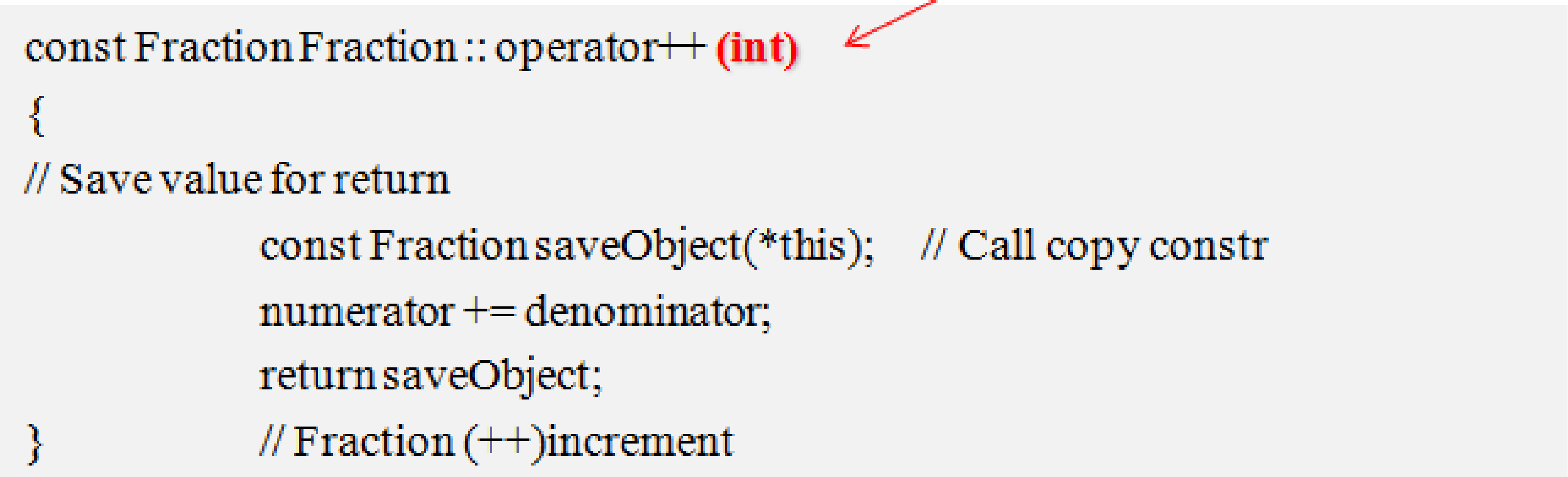
Overloading Prefix ‘++’ for class Fraction



```
Fraction& Fraction :: operator++ ()
{
                numerator += denominator;
                return (*this);
}               // Fraction increment (++)
```

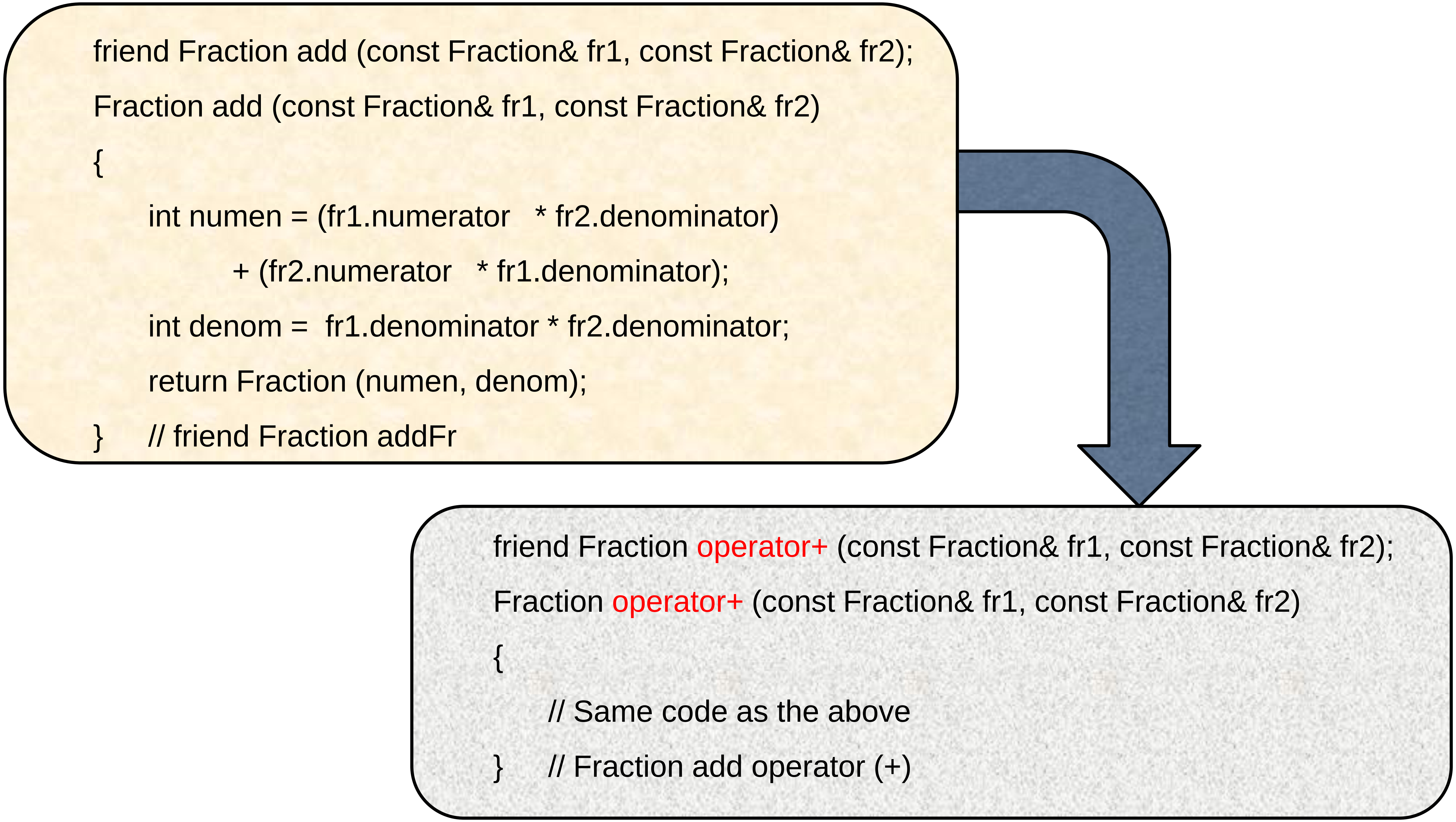
Overloading Postfix '++' for class Fraction

Have no meaning, Just for "Postfix operator" marking



```
const Fraction Fraction :: operator++ (int)
{
// Save value for return
              const Fraction saveObject(*this);    // Call copy constr
              numerator += denominator;
              return saveObject;
}             // Fraction (++)increment
```

Overloading Friend Functions



```
friend Fraction add (const Fraction& fr1, const Fraction& fr2);
Fraction add (const Fraction& fr1, const Fraction& fr2)
{
      int numen = (fr1.numerator   * fr2.denominator)
                + (fr2.numerator   * fr1.denominator);
      int denom =  fr1.denominator * fr2.denominator;
      return Fraction (numen, denom);
}     // friend Fraction addFr
```

```
friend Fraction operator+ (const Fraction& fr1, const Fraction& fr2);
Fraction operator+ (const Fraction& fr1, const Fraction& fr2)
{
      // Same code as the above
}     // Fraction add operator (+)
```

Default Assignment Operator

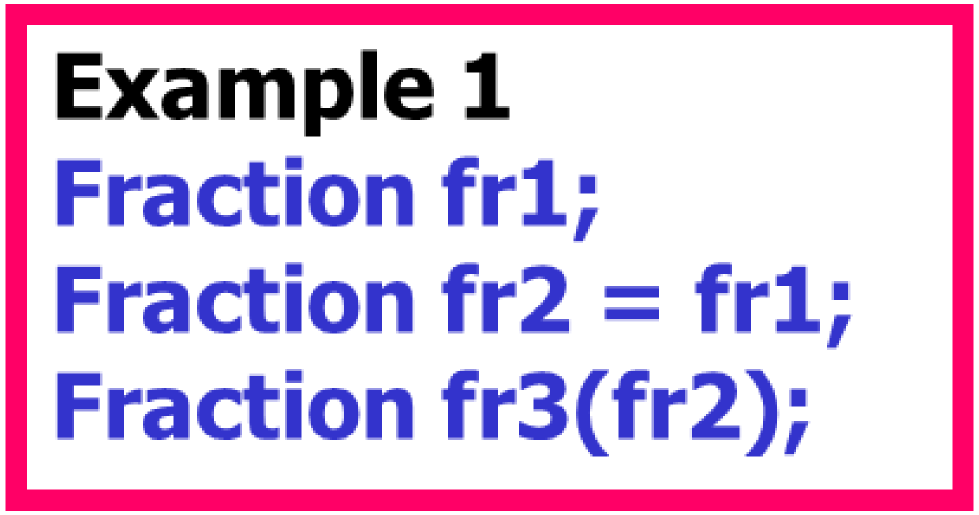
C++ provides a bitwise overloaded assignment operator if we don't overload it ourselves.

However, the default operator is a bitwise, not a logical, operator

(Remind "Bitwise versus Logical Copy Constructors")

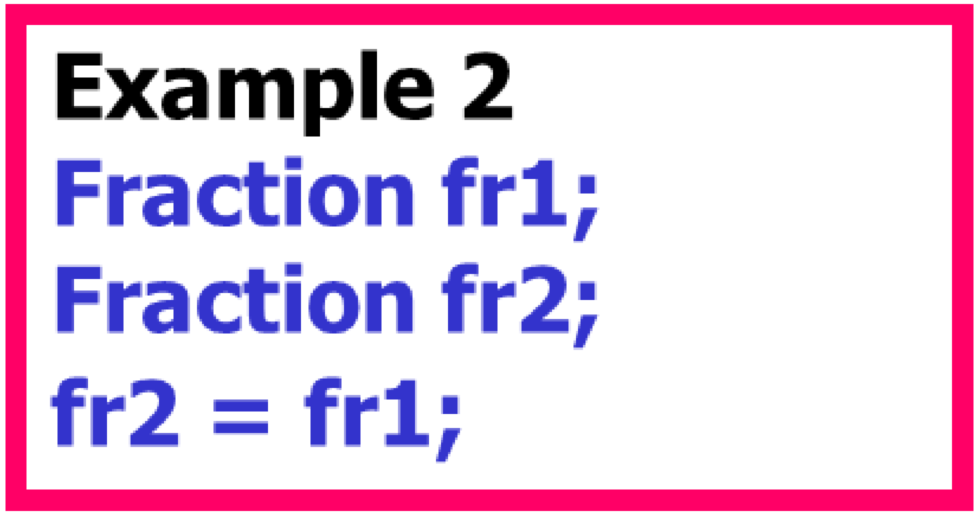
Assignment operator vs. Copy constructor

- **Copy constructor is called when a class object is created and initialized with an existing object**
  - **Includes passing/returning objects in function calls**
- **Assignment operator is called when the left-hand side and the right-hand side of an assignment statement are both objects of the same class**



Example 1

```
Fraction fr1;
Fraction fr2 = fr1;
Fraction fr3(fr2);
```



Example 2

```
Fraction fr1;
Fraction fr2;
fr2 = fr1;
```

Note:

When we need to write a copy constructor, we also need to write an assignment operator and a destructor.

Example:

If we need a logical copy because we are using dynamic memory, then we need a logical assignment and a destructor to recycle memory when the object is destroyed.

Overloading Cast Operators (Type Conversion)

From Standard Type to Object Type

- Implicit type conversion by using a constructor

```
fr1.addTo(45); // conversion from 45 to 45/1
```

- To prevent implicit type conversion, C++ provides an *explicit* constructor modifier

```
explicit Fraction (int numer);
```

- The explicit modifier is added only to the prototype; not coded in the function definition

Overloading Cast Operators (Type Conversion)

From Object Type to Standard Type

- Write a conversion operator for the source type

No return type: the converted data are automatically returned

the type to which we are converting

normally declared *const* to support constant invoking objects

```
Fraction :: operator float () const
{
   return (numerator / denominator);
} // operator float
```

```
Fraction fr (5, 4);
cout << fr << endl;        // Prints 1.25
```

Using overloaded operators

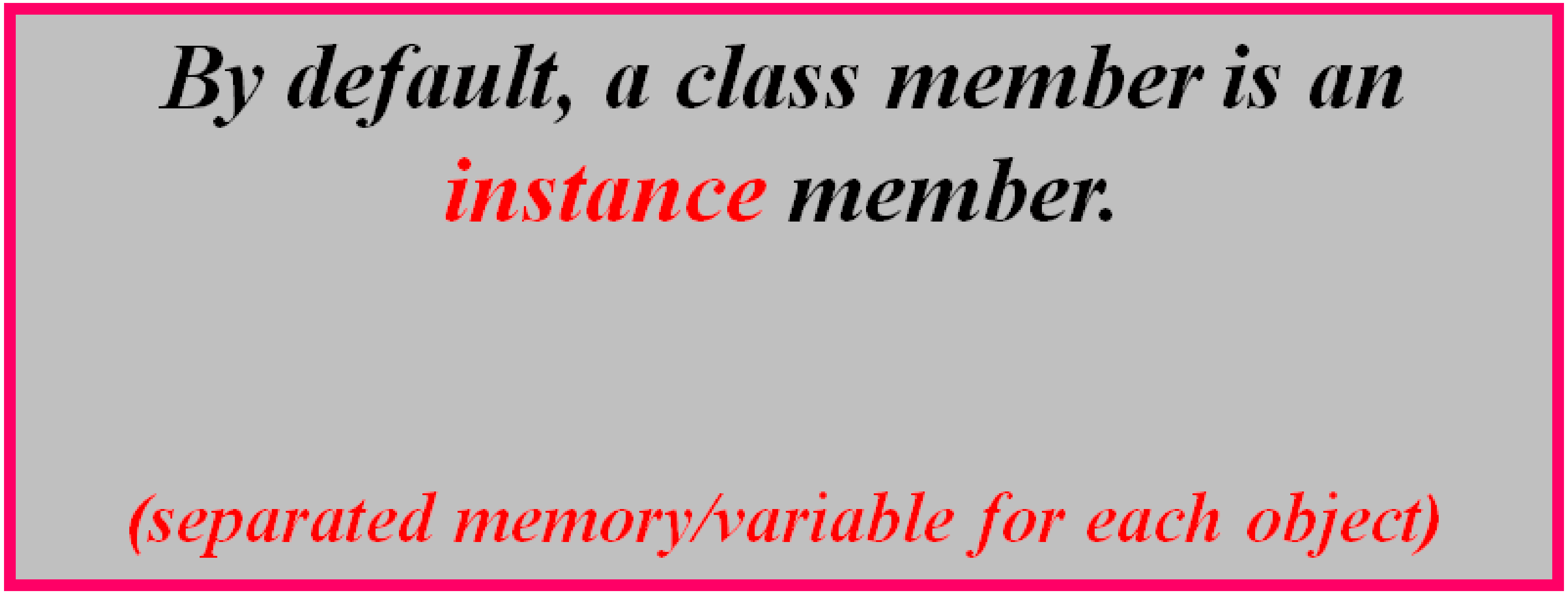
- The primary purpose of operator overloading is to enable us to write code in a more natural style
- Operators can also be used in the form of function calling
- Examples

```
++fr1;              vs.   fr1.operator++();
fr2 += fr1;          vs.  fr2.operator+= (fr1);
fr3 = fr1 + fr2;    vs.   fr3 = operator+ (fr1, fr2);
```

Overloaded Operator Limitation and Restriction

- Operators prohibited from overloading
    - Member operator (.)
    - Pointer to member operator (.*)
    - Scope resolution operator (::)
    - Conditional expression operator (? :)
    - sizeof operator (sizeof (int))
- Overloading does not change the precedence
- Overloading does not change the associativity
- Overloading does not change the commutativity
- Overloading does not change the arity
- The [bracket] and (parentheses) operators can be overloaded only as member operators
- Only objects in a class scope can be overloaded

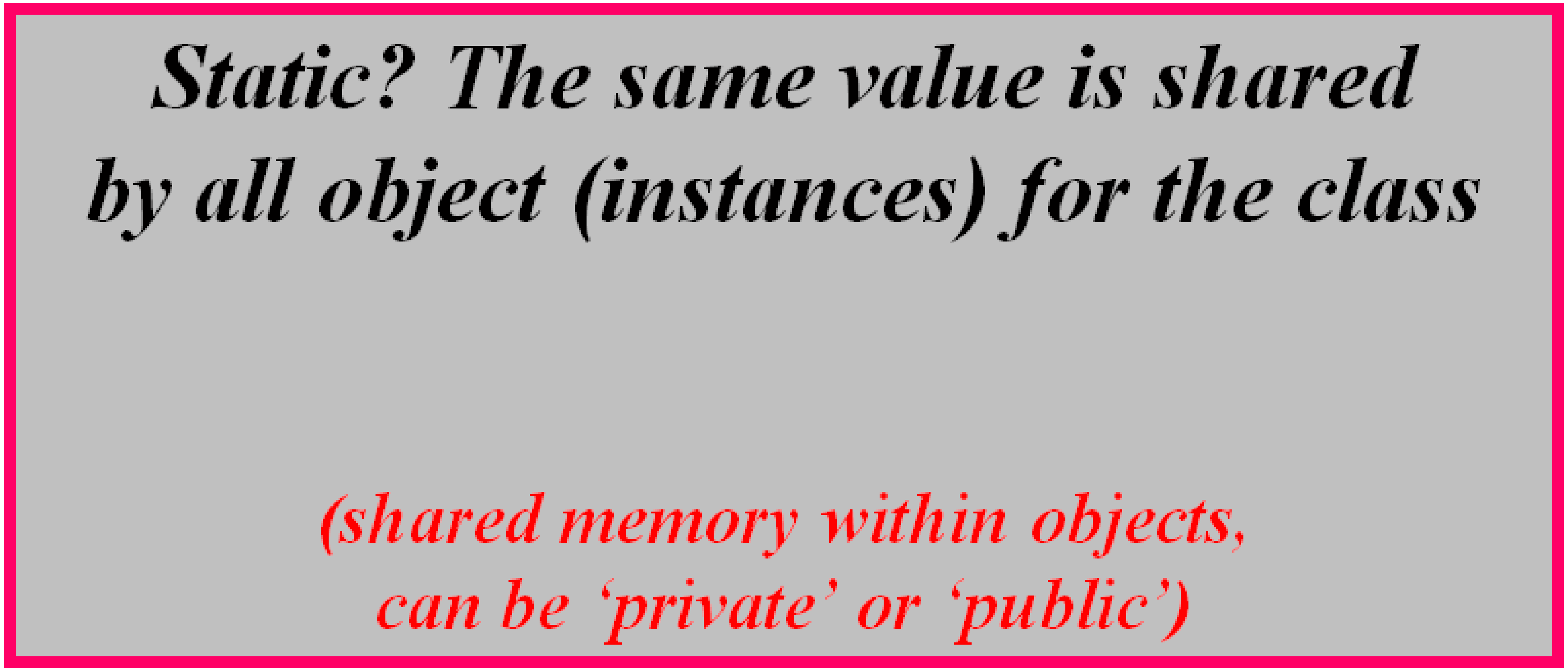
Instance member



By default, a class member is an instance member.

(separated memory/variable for each object)

Static member



Static? The same value is shared by all object (instances) for the class

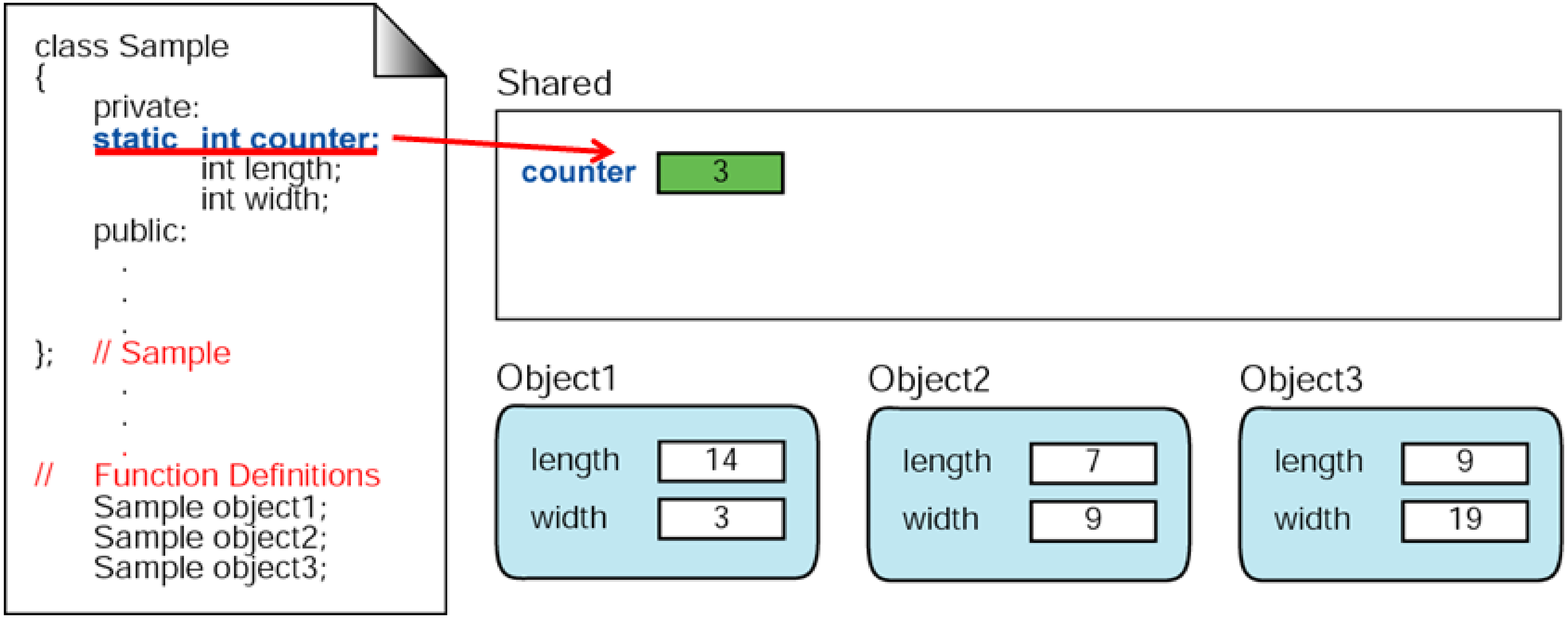
(shared memory within objects, can be 'private' or 'public')

Why static data member?

- Used when a data member applies to the class itself; all instances share the same value

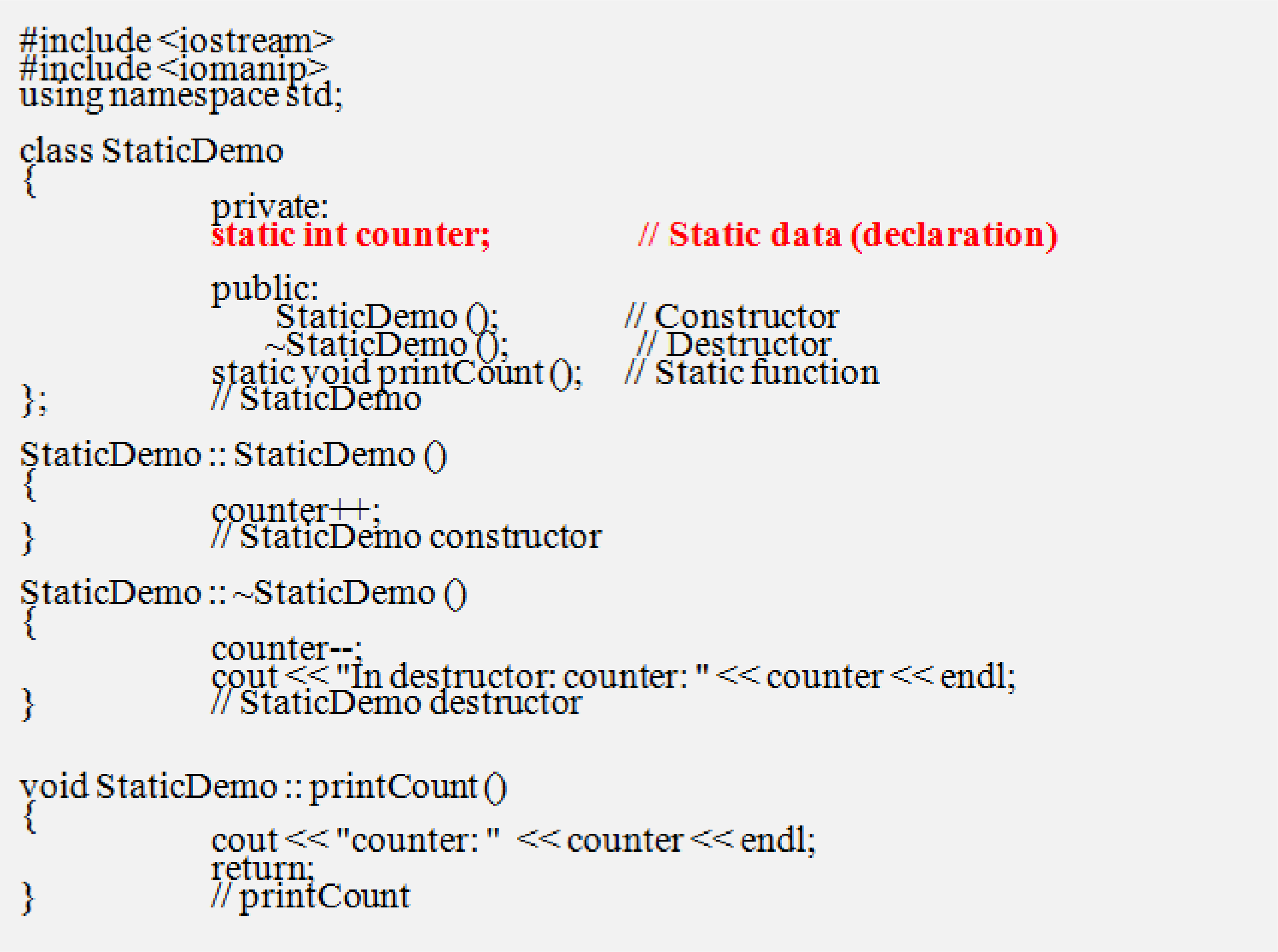
- Examples
  - An employee wage table or a table of tax rates
  - A counter of the number of occurrences of an class

Static data member



```
class Sample
{
      private:
      static  int counter;
              int length;
              int width;
      public:
        .
        .
        .
};    // Sample
        .
        .
        .
//    Function Definitions
      Sample object1;
      Sample object2;
      Sample object3;
```

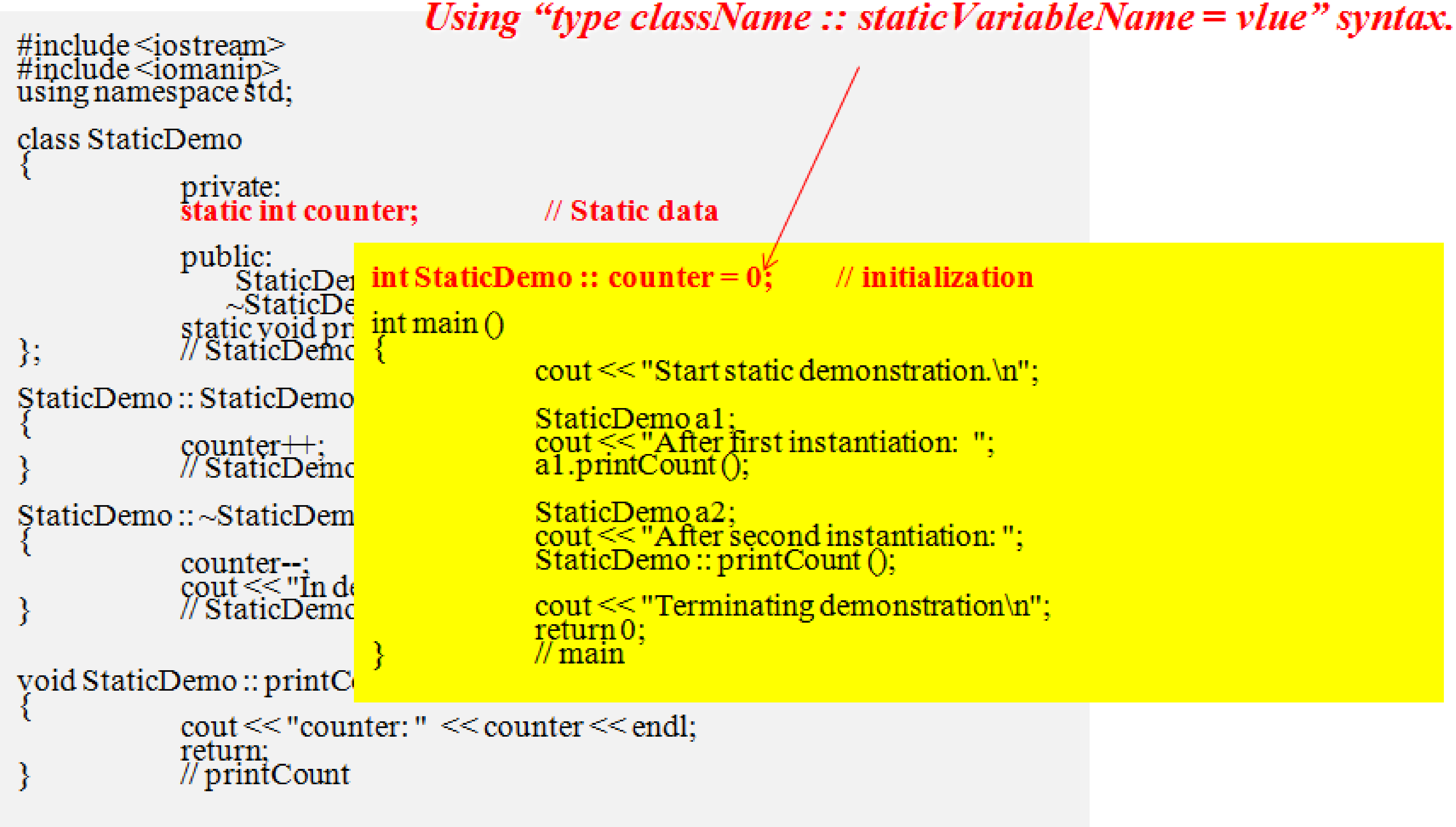
Static data member declaration



```
#include <iostream>
#include <iomanip>
using namespace std;

class StaticDemo
{
            private:
            static int counter;           // Static data (declaration)

            public:
                StaticDemo ();            // Constructor
               ~StaticDemo ();             // Destructor
            static void printCount ();    // Static function
};          // StaticDemo

StaticDemo :: StaticDemo ()
{
            counter++;
}           // StaticDemo constructor

StaticDemo :: ~StaticDemo ()
{
            counter--;
            cout << "In destructor: counter: " << counter << endl;
}           // StaticDemo destructor


void StaticDemo :: printCount ()
{
            cout << "counter: "  << counter << endl;
            return;
}           // printCount
```

Static data member definition and initialization

Should be defined and initialized out of class & global region of program.
Using "type className :: staticVariableName = vlue" syntax.



```
#include <iostream>
#include <iomanip>
using namespace std;

class StaticDemo
{
            private:
            static int counter;          // Static data

            public:
                StaticDe
               ~StaticDe
            static void pr
};          // StaticDemo

StaticDemo :: StaticDemo
{
            counter++;
}           // StaticDemo

StaticDemo :: ~StaticDem
{
            counter--;
            cout << "In de
}           // StaticDemo

void StaticDemo :: printC
{
            cout << "counter: "  << counter << endl;
            return;
}           // printCount
```

```
int StaticDemo :: counter = 0;      // initialization

int main ()
{
            cout << "Start static demonstration.\n";

            StaticDemo a1;
            cout << "After first instantiation:  ";
            a1.printCount ();

            StaticDemo a2;
            cout << "After second instantiation: ";
            StaticDemo :: printCount ();

            cout << "Terminating demonstration\n";
            return 0;
}           // main
```

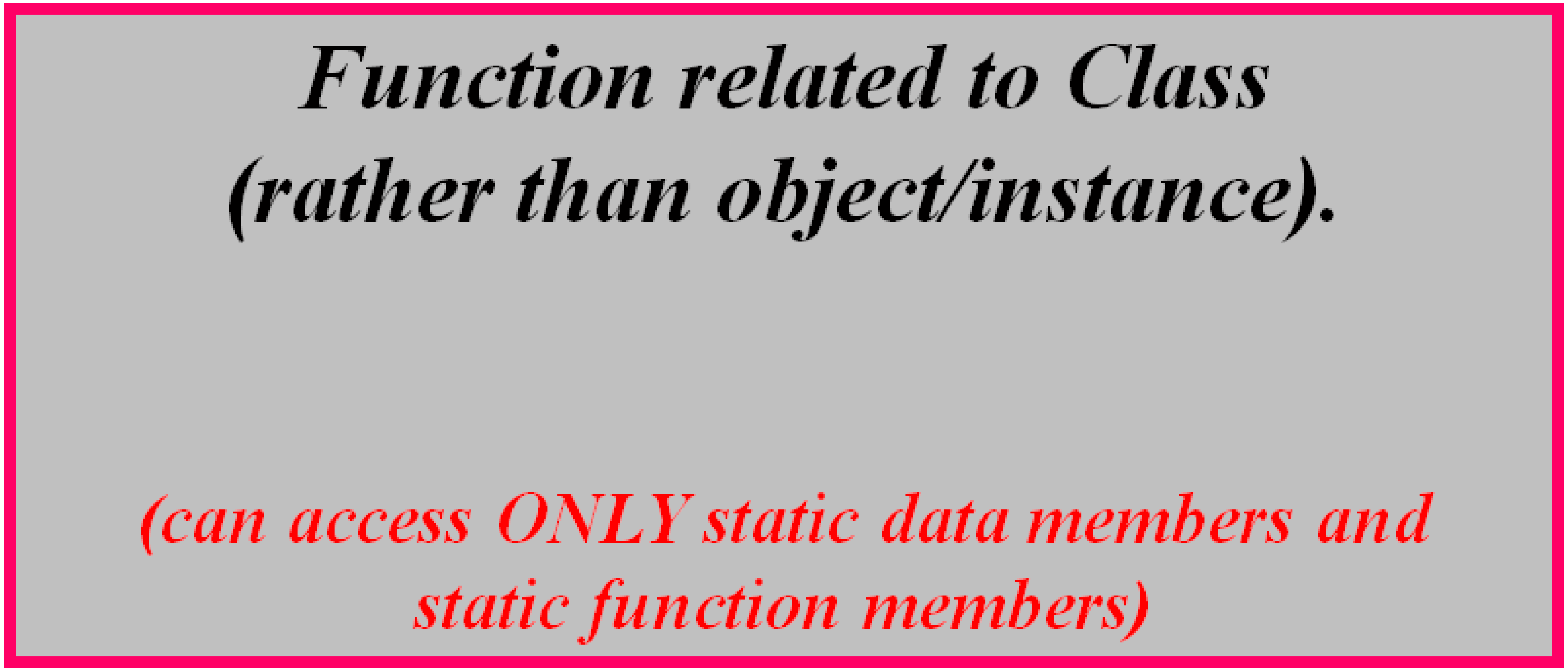
Note:

There are three static data members:

- ***explicit static data members***
- ***enumerated members***
- ***type defined members***

implicit static members

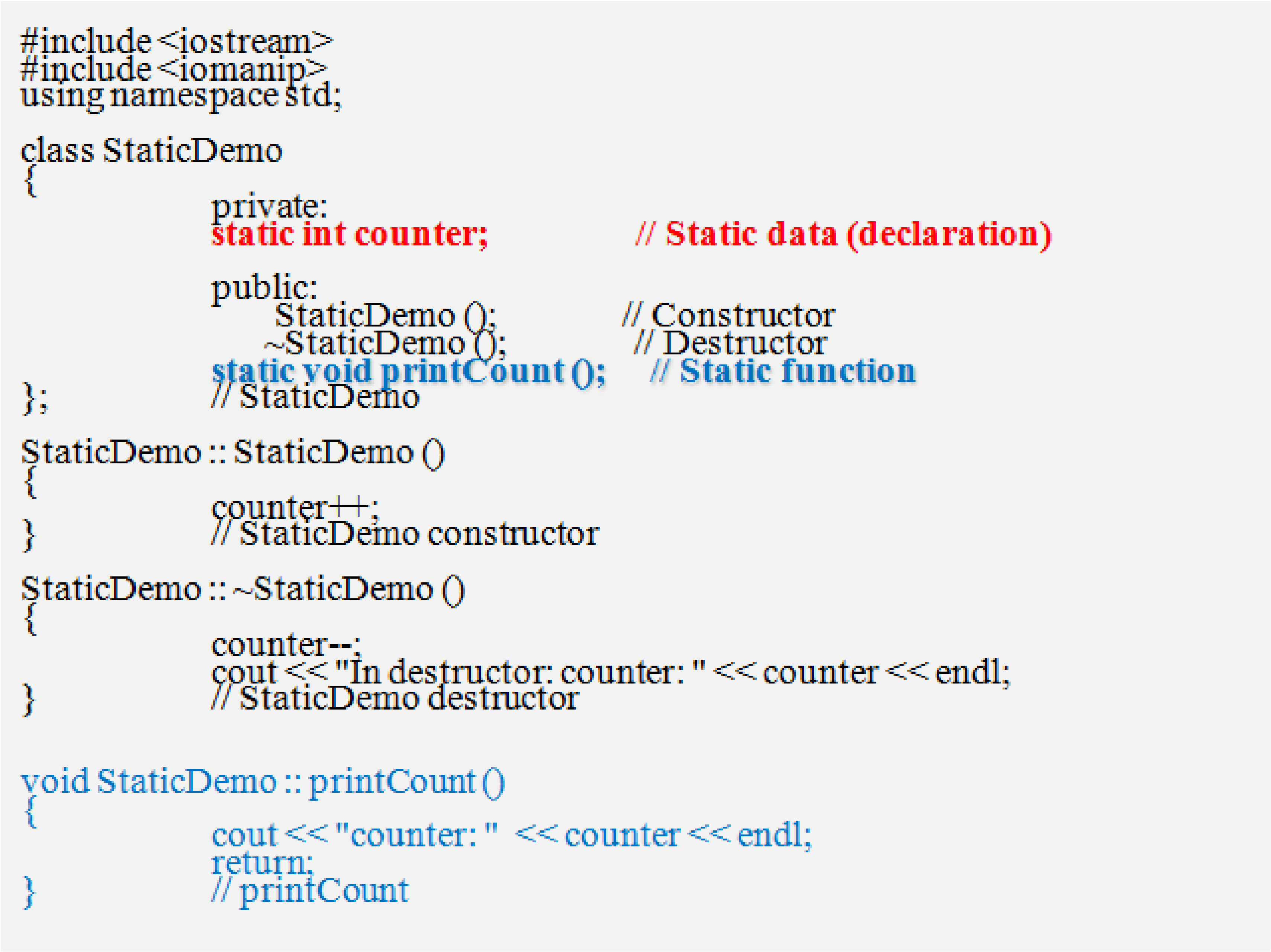
Static function member



Function related to Class
(rather than object/instance).

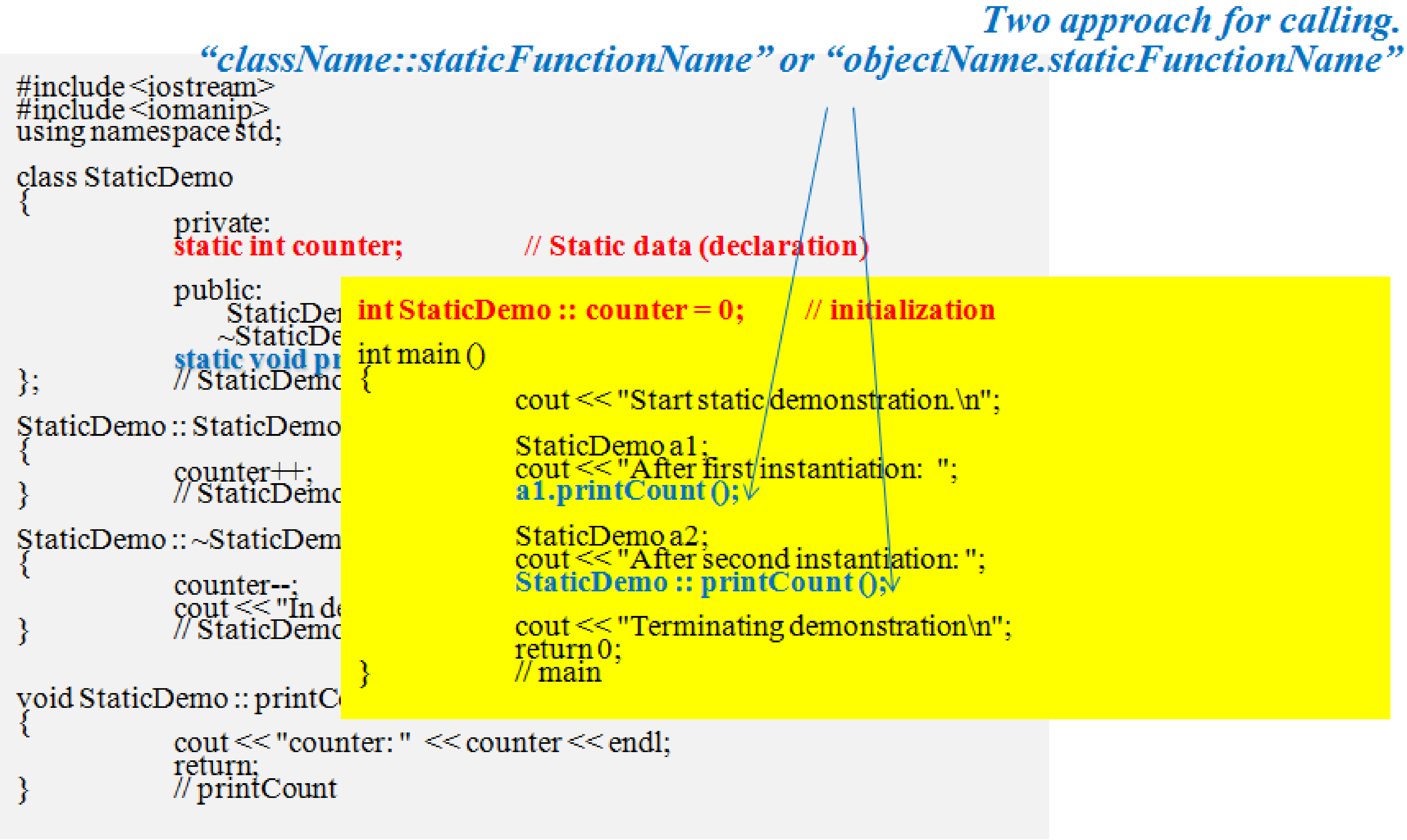
(can access ONLY static data members and static function members)

Static function member



```
#include <iostream>
#include <iomanip>
using namespace std;

class StaticDemo
{
                private:
                static int counter;             // Static data (declaration)

                public:
                     StaticDemo ();             // Constructor
                    ~StaticDemo ();              // Destructor
                static void printCount ();       // Static function
};              // StaticDemo

StaticDemo :: StaticDemo ()
{
                counter++;
}               // StaticDemo constructor

StaticDemo :: ~StaticDemo ()
{
                counter--;
                cout << "In destructor: counter: " << counter << endl;
}               // StaticDemo destructor


void StaticDemo :: printCount ()
{
                cout << "counter: "  << counter << endl;
                return;
}               // printCount
```

Static function member



Two approach for calling.
“className::staticFunctionName” or “objectName.staticFunctionName”

```
#include <iostream>
#include <iomanip>
using namespace std;

class StaticDemo
{
            private:
            static int counter;            // Static data (declaration)

            public:
                StaticDe
               ~StaticDe
            static void pr
};          // StaticDemo

StaticDemo :: StaticDemo
{
            counter++;
}           // StaticDemo

StaticDemo :: ~StaticDem
{
            counter--;
            cout << "In de
}           // StaticDemo


void StaticDemo :: printC
{
            cout << "counter: "  << counter << endl;
            return;
}           // printCount
```

```
int StaticDemo :: counter = 0;        // initialization

int main ()
{
            cout << "Start static demonstration.\n";

            StaticDemo a1;
            cout << "After first instantiation:  ";
            a1.printCount ();

            StaticDemo a2;
            cout << "After second instantiation: ";
            StaticDemo :: printCount ();

            cout << "Terminating demonstration\n";
            return 0;
}           // main
```

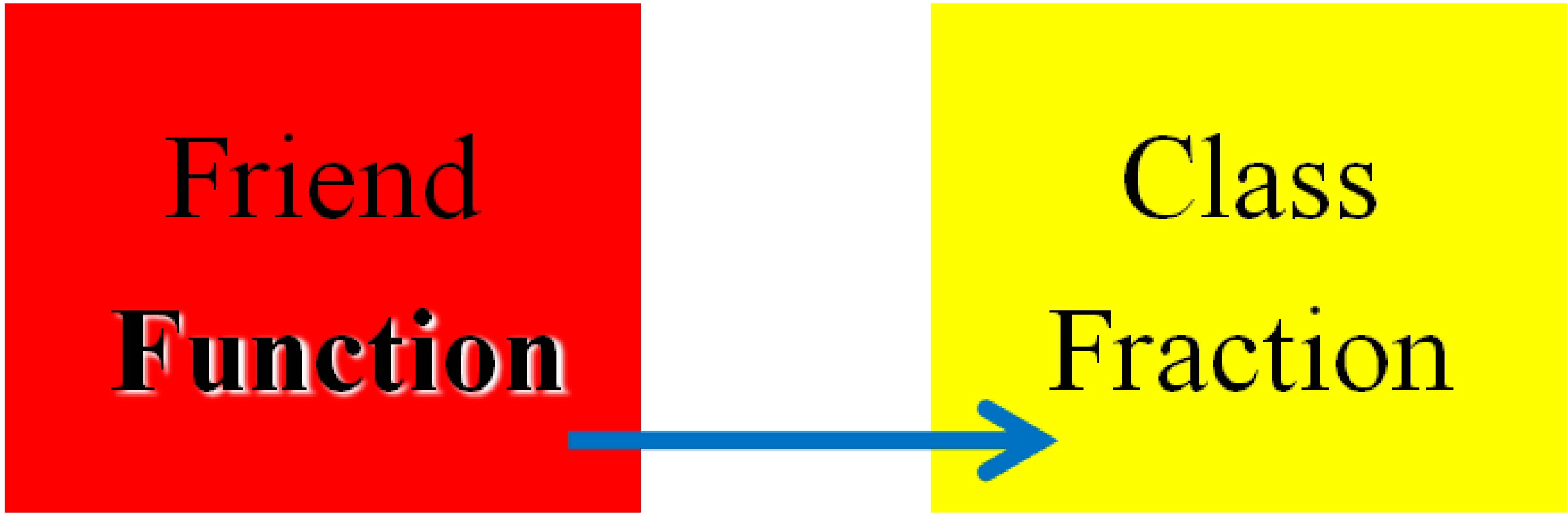
More about Static

- Refer “[REF] *More class features (Static).pdf*”

Remind! Friend function

Location: Outside of the class

(not a member of class)



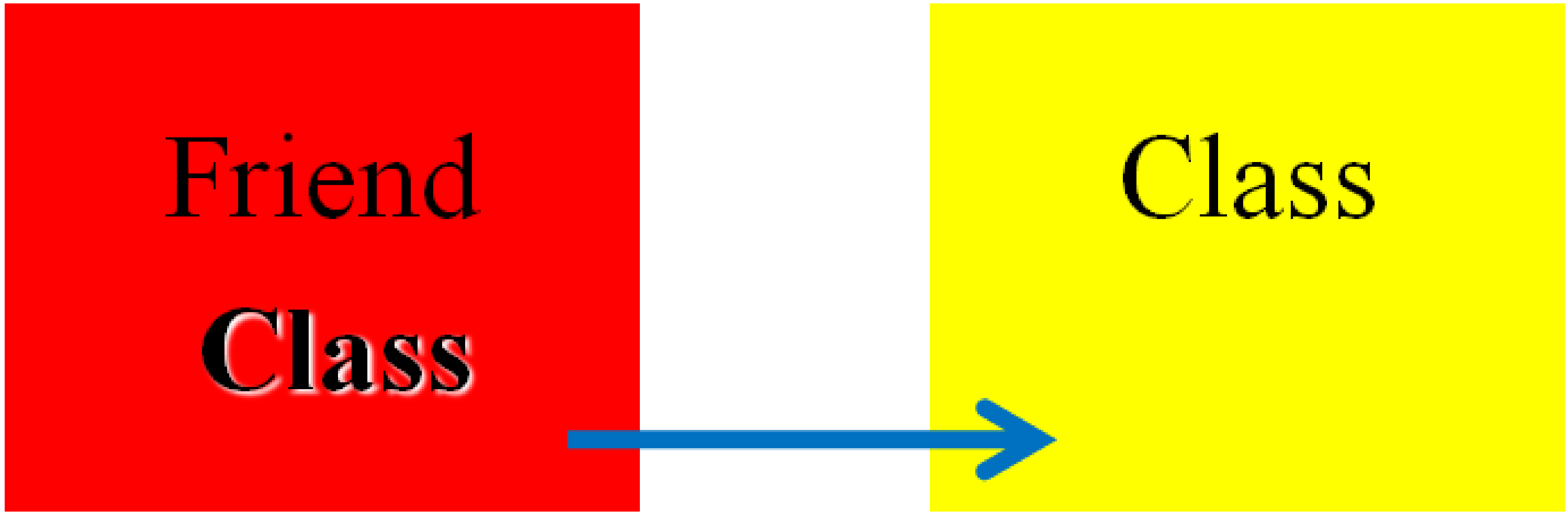
Friend **Function**

Class Fraction

Privilege: can access private members

Remind! Friend function

Location: Other Class



Friend Class

Class

Privilege: can access private members

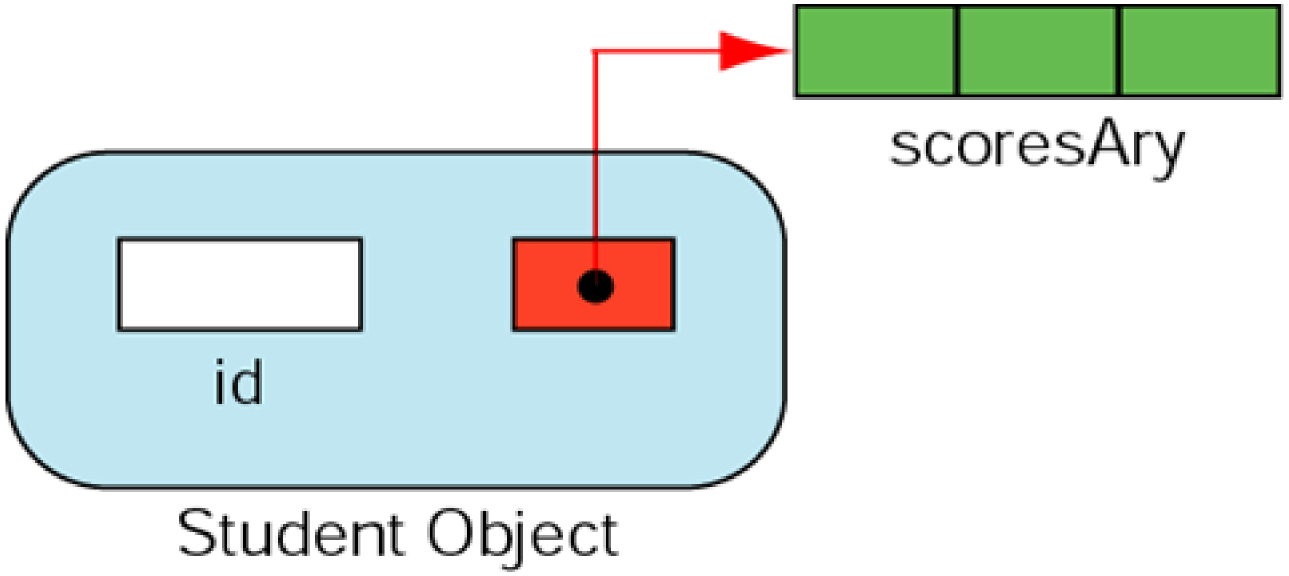
Friend Classes

- All member functions of the class granted friendship have unrestricted access to the members of the class granting the friendship
- Useful when two or more objects need to communicate with each other (e.g., the node and list objects used in a linked list)

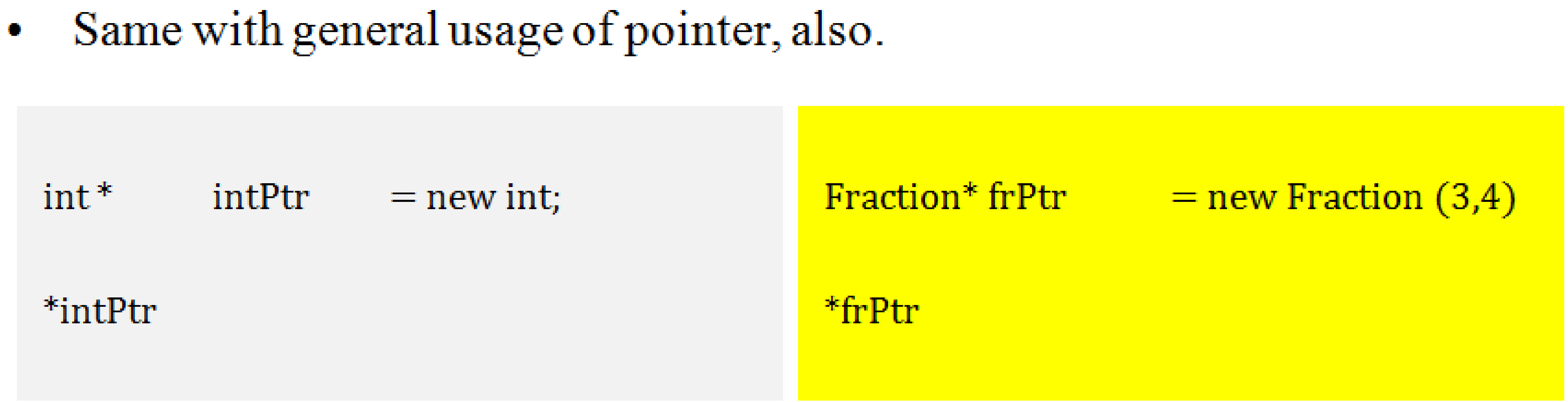
```
class First {
    private:
        …
    friend class Second;
    …
} // First
```

Pointer as a member of class

- Same with general usage of pointer



Pointer to object



- Same with general usage of pointer, also.

```
int *        intPtr       = new int;

*intPtr
```

```
Fraction* frPtr          = new Fraction (3,4)

*frPtr
```

Member function call using pointer to object

- Same with general usage of pointer, also.

```
int *       intPtr      = new int;

*intPtr
```

```
Fraction* frPtr         = new Fraction (3,4)

*frPtr
```

```
(*frPtr).store(2,5);
(*frPtr).print();
```

SAME

```
frPtr->store(2,5);
frPtr->print();
```

Class array

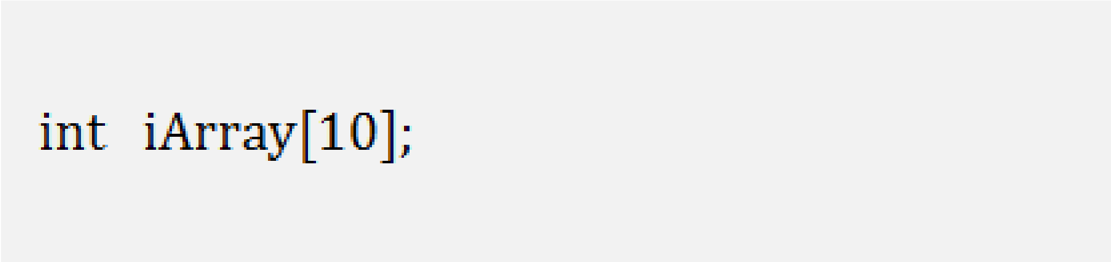
- Same with general usage of array

```
int   iArray[10];
```

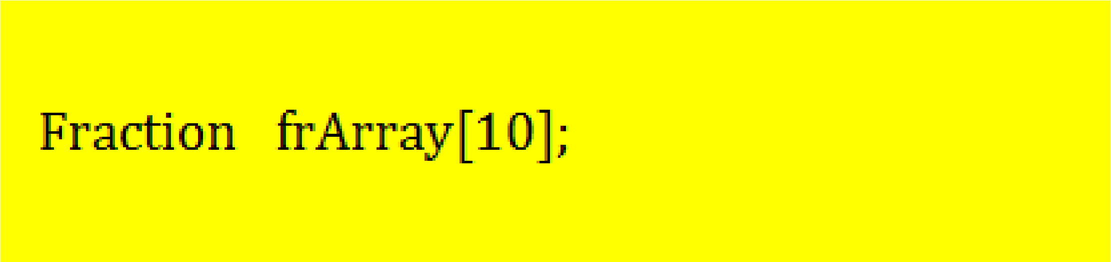
```
Fraction   frArray[10];
```

Class array initialization

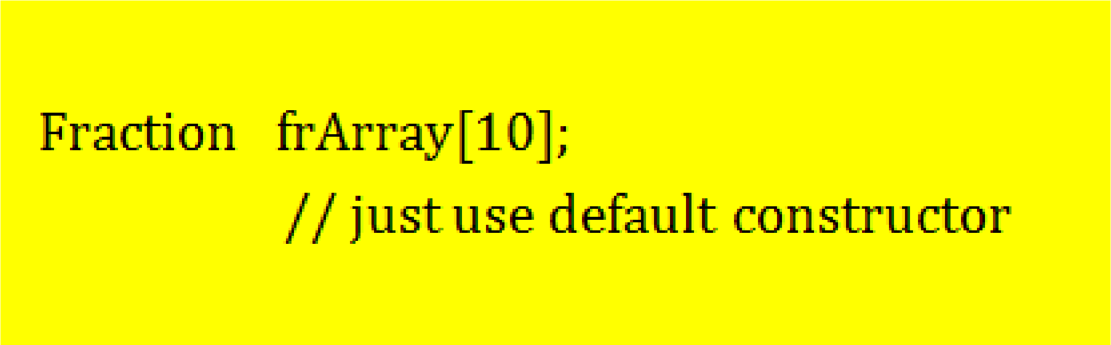
- Same with general usage of array



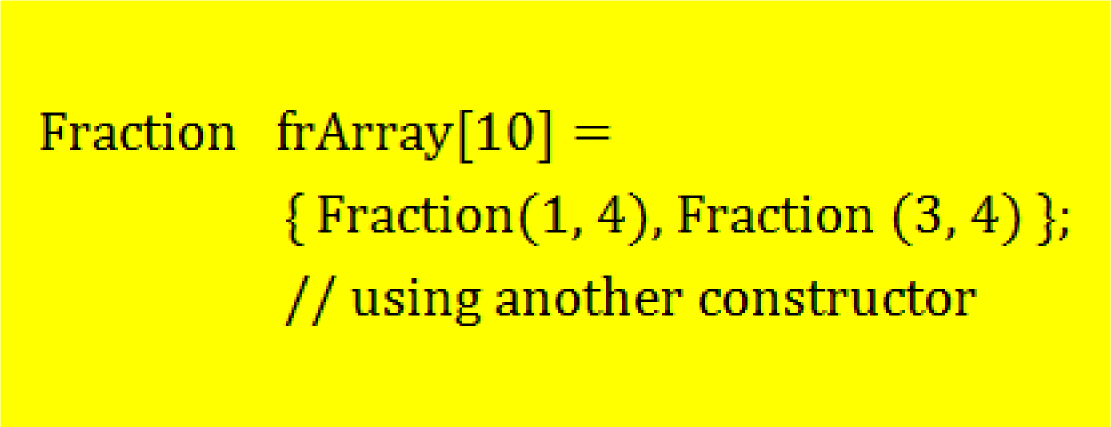
```
int   iArray[10];
```



```
Fraction   frArray[10];
```

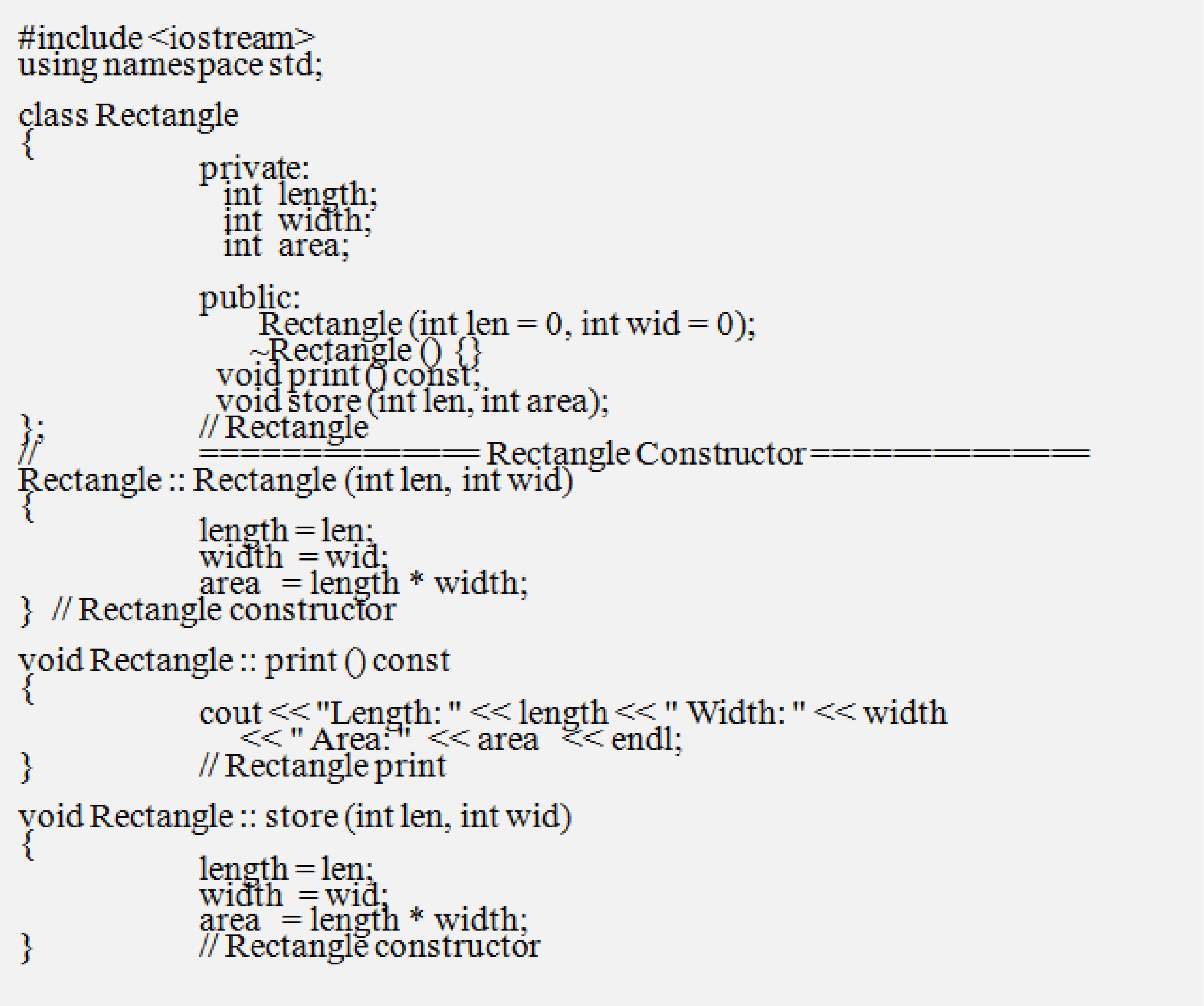


```
Fraction   frArray[10];
           // just use default constructor
```



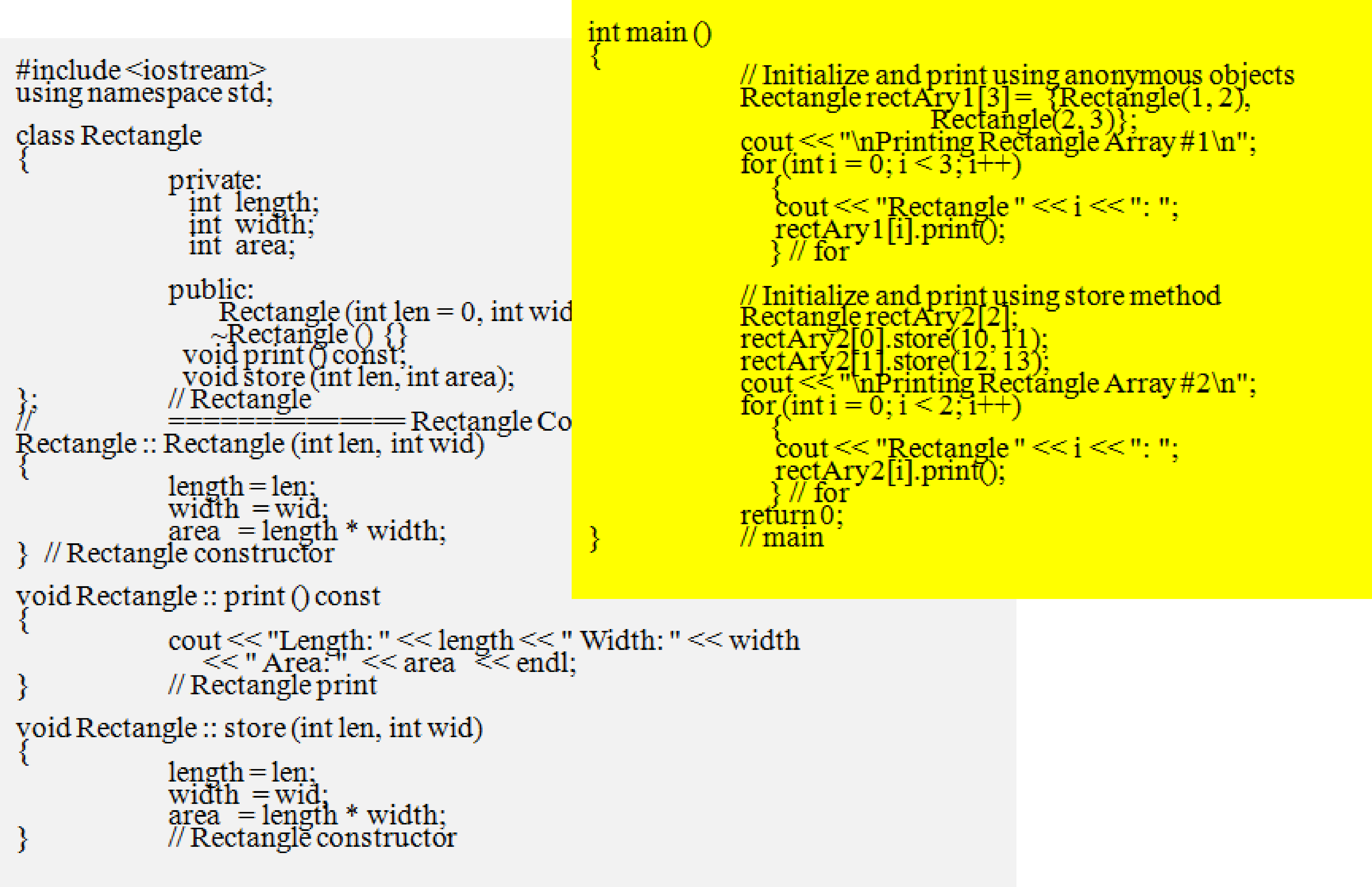
```
Fraction   frArray[10] =
           { Fraction(1, 4), Fraction (3, 4) };
           // using another constructor
```

Practice (Program 11-11)



```
#include <iostream>
using namespace std;

class Rectangle
{
              private:
                int  length;
                int  width;
                int  area;

              public:
                   Rectangle (int len = 0, int wid = 0);
                  ~Rectangle () {}
                void print () const;
                void store (int len, int area);
};            // Rectangle
//            ============ Rectangle Constructor ============
Rectangle :: Rectangle (int len, int wid)
{
              length = len;
              width  = wid;
              area   = length * width;
}  // Rectangle constructor

void Rectangle :: print () const
{
              cout << "Length: " << length << " Width: " << width
                  << " Area: "  << area   << endl;
}             // Rectangle print

void Rectangle :: store (int len, int wid)
{
              length = len;
              width  = wid;
              area   = length * width;
}             // Rectangle constructor
```

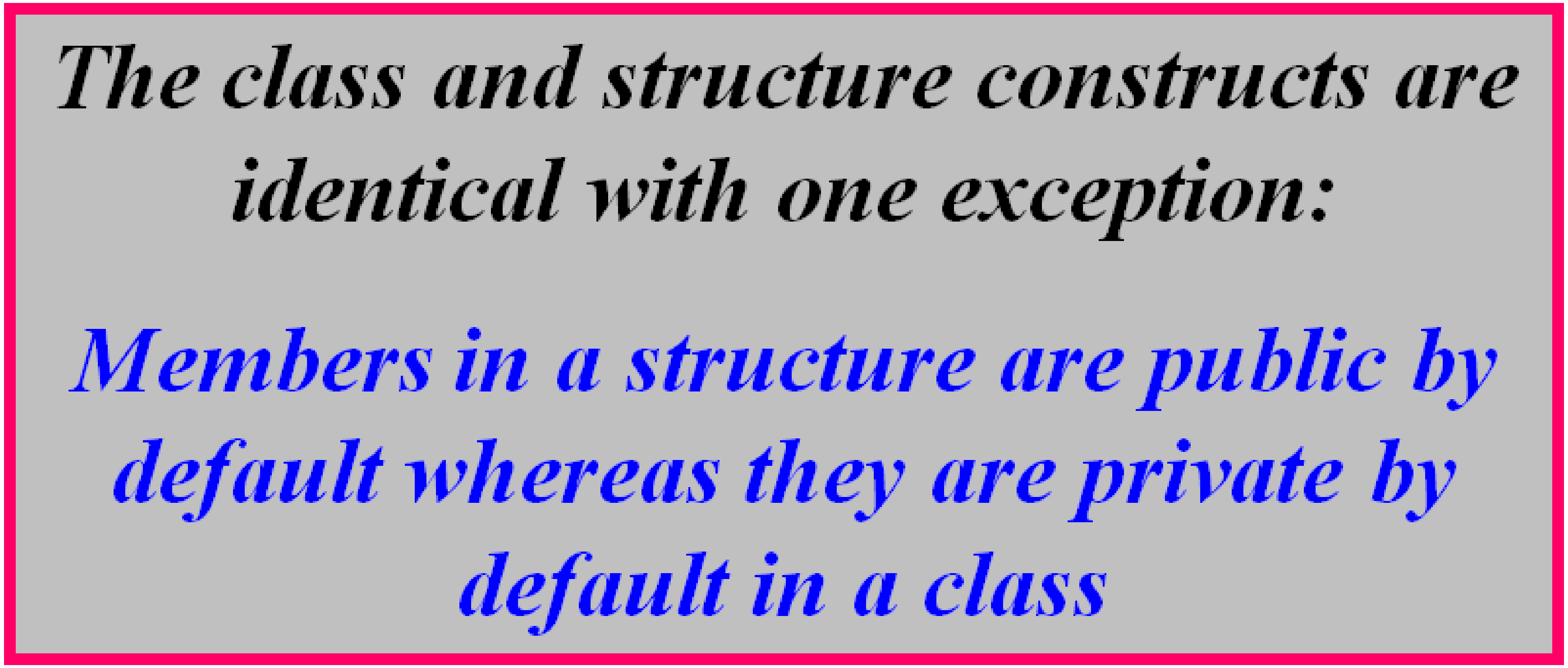
Practice (Program 11-11)



```
#include <iostream>
using namespace std;

class Rectangle
{
            private:
              int  length;
              int  width;
              int  area;

            public:
                  Rectangle (int len = 0, int wid
                 ~Rectangle () {}
             void print () const;
             void store (int len, int area);
};          // Rectangle
//          ============= Rectangle Co
Rectangle :: Rectangle (int len, int wid)
{
            length = len;
            width  = wid;
            area   = length * width;
}  // Rectangle constructor

void Rectangle :: print () const
{
            cout << "Length: " << length << " Width: " << width
               << " Area: "  << area   << endl;
}           // Rectangle print

void Rectangle :: store (int len, int wid)
{
            length = len;
            width  = wid;
            area   = length * width;
}           // Rectangle constructor
```

```
int main ()
{
            // Initialize and print using anonymous objects
            Rectangle rectAry1[3] = {Rectangle(1, 2),
                                     Rectangle(2, 3)};
            cout << "\nPrinting Rectangle Array #1\n";
            for (int i = 0; i < 3; i++)
               {
               cout << "Rectangle " << i << ": ";
               rectAry1[i].print();
               } // for

            // Initialize and print using store method
            Rectangle rectAry2[2];
            rectAry2[0].store(10, 11);
            rectAry2[1].store(12, 13);
            cout << "\nPrinting Rectangle Array #2\n";
            for (int i = 0; i < 2; i++)
               {
               cout << "Rectangle " << i << ": ";
               rectAry2[i].print();
               } // for
            return 0;
}           // main
```

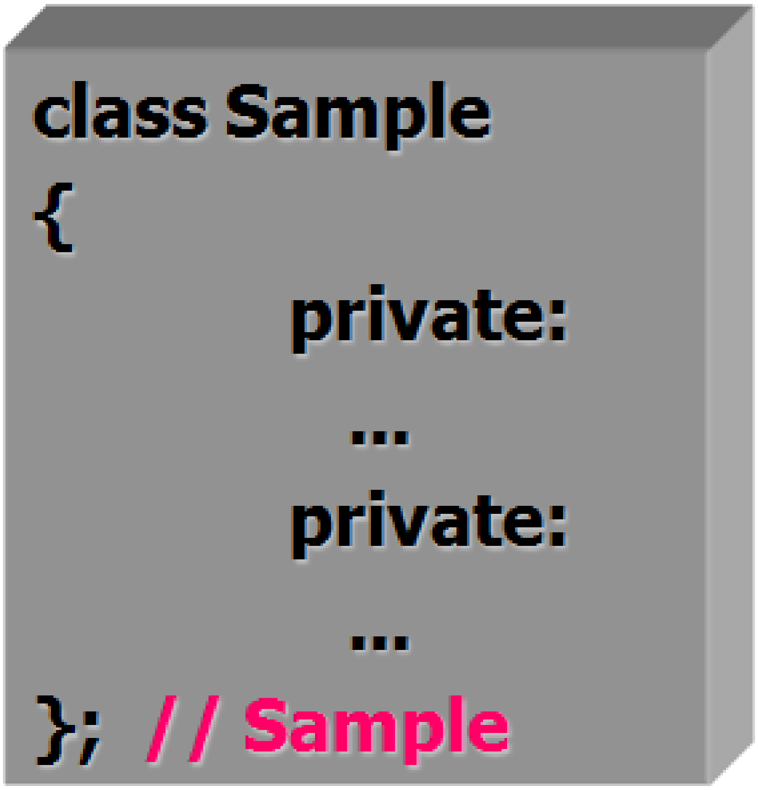
Structure?



The class and structure constructs are identical with one exception:

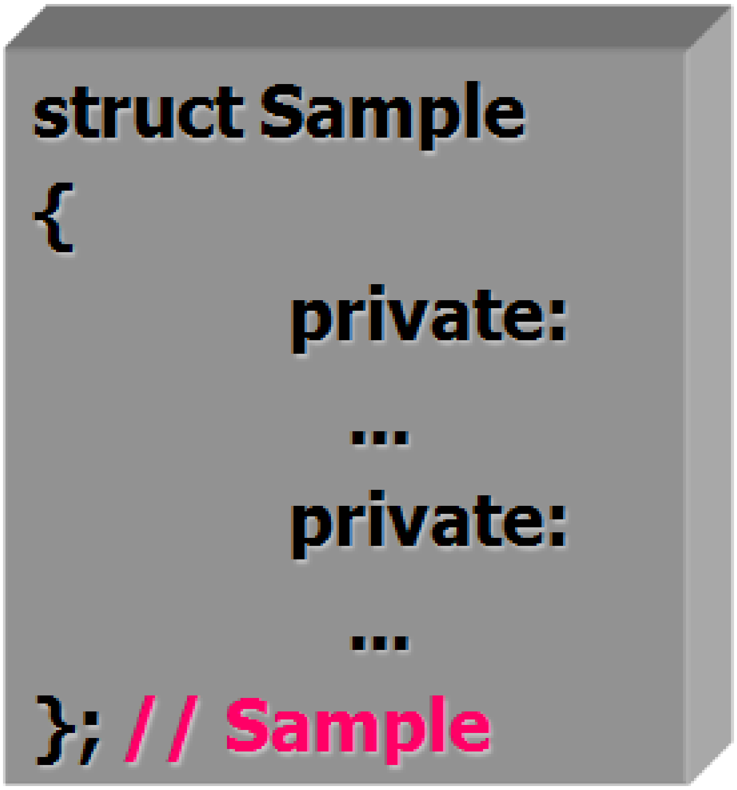
Members in a structure are public by default whereas they are private by default in a class

Class and structure declarations



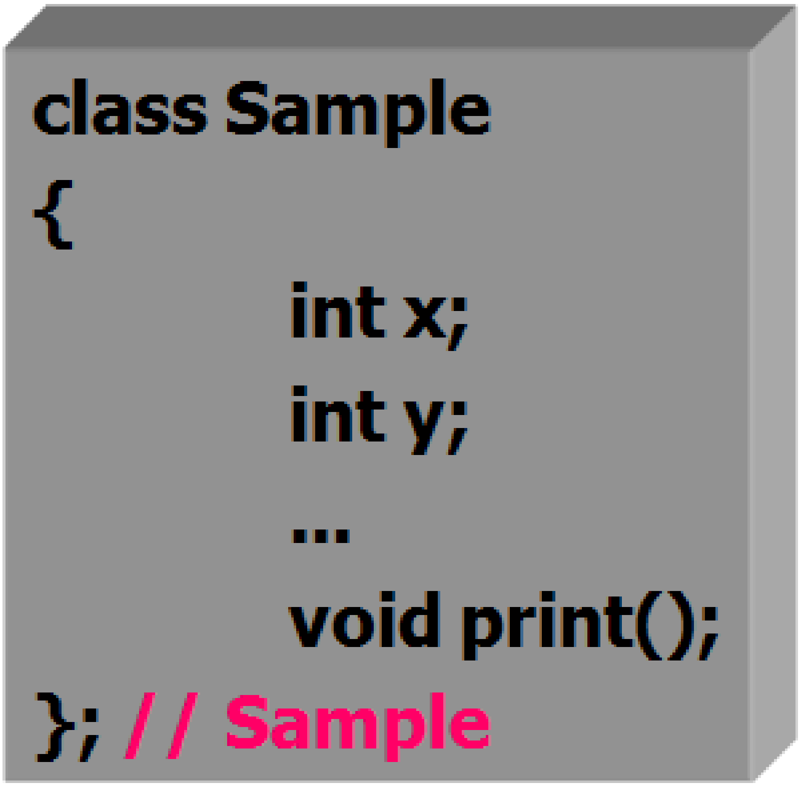
```
class Sample
{
          private:
            ...
          private:
            ...
}; // Sample
```

=



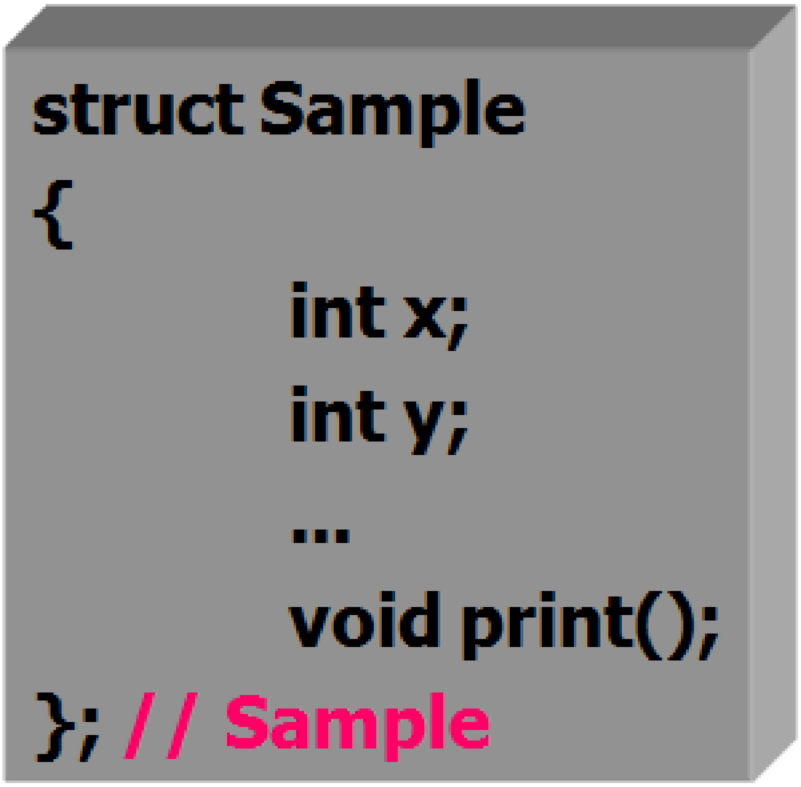
```
struct Sample
{
          private:
            ...
          private:
            ...
}; // Sample
```

Class and structure declarations



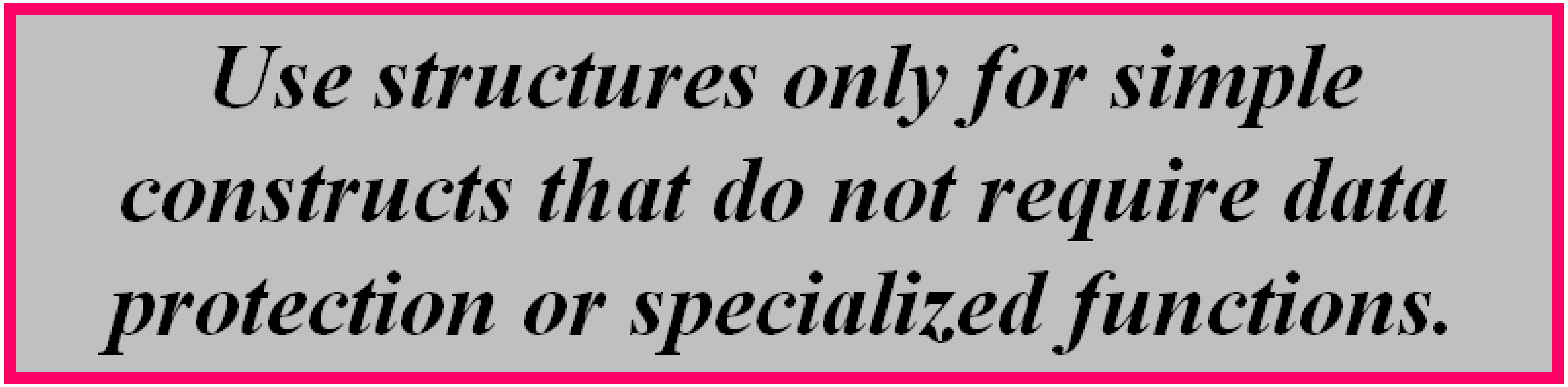
```
class Sample
{
        int x;
        int y;
        ...
        void print();
}; // Sample
```

≠



```
struct Sample
{
        int x;
        int y;
        ...
        void print();
}; // Sample
```

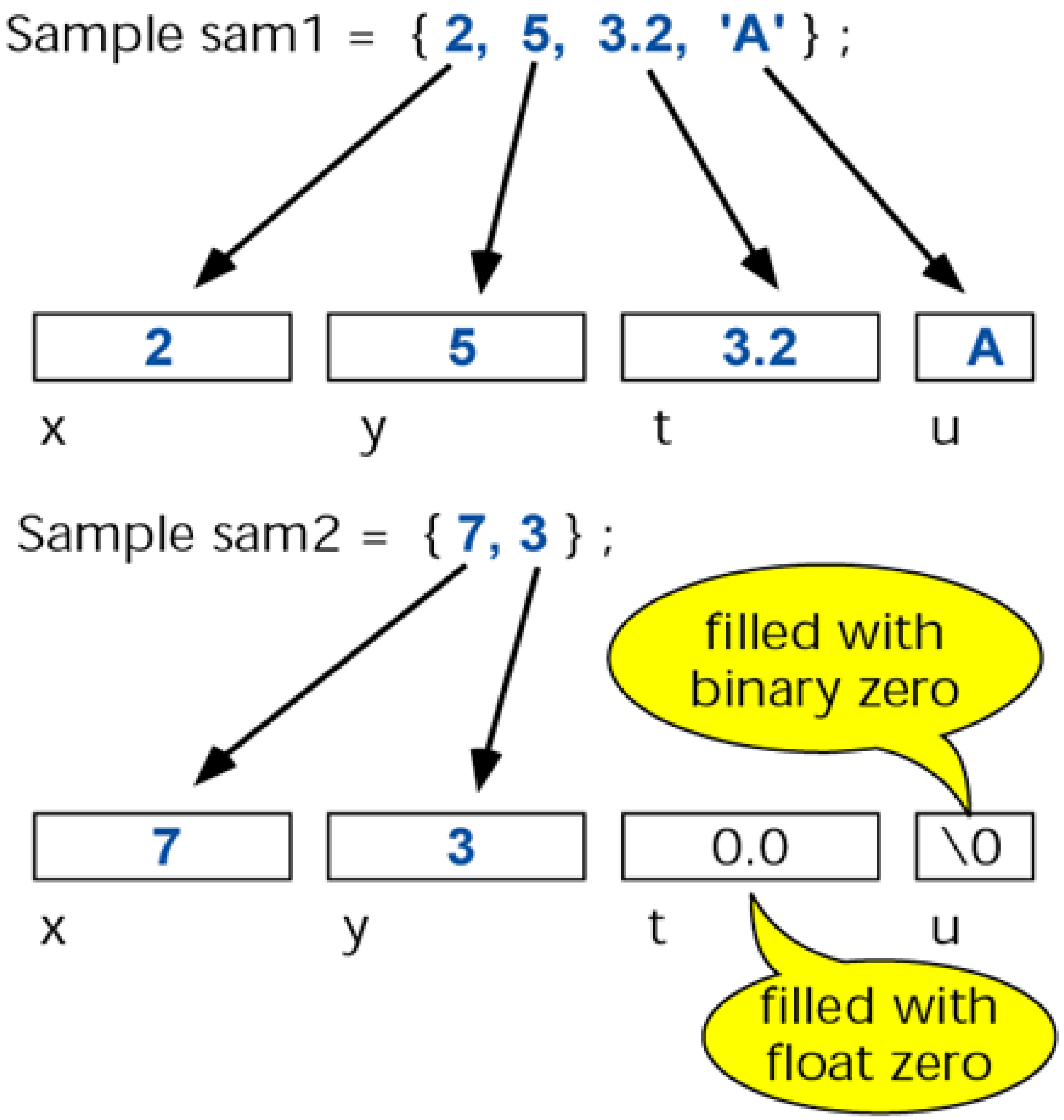
Usage of structure



Use structures only for simple constructs that do not require data protection or specialized functions.

Structure definition and initialization

```
struct Sample
  {
     int    x ;
     int    y ;
     float t ;
     char u ;
  } ;  //Sample
```



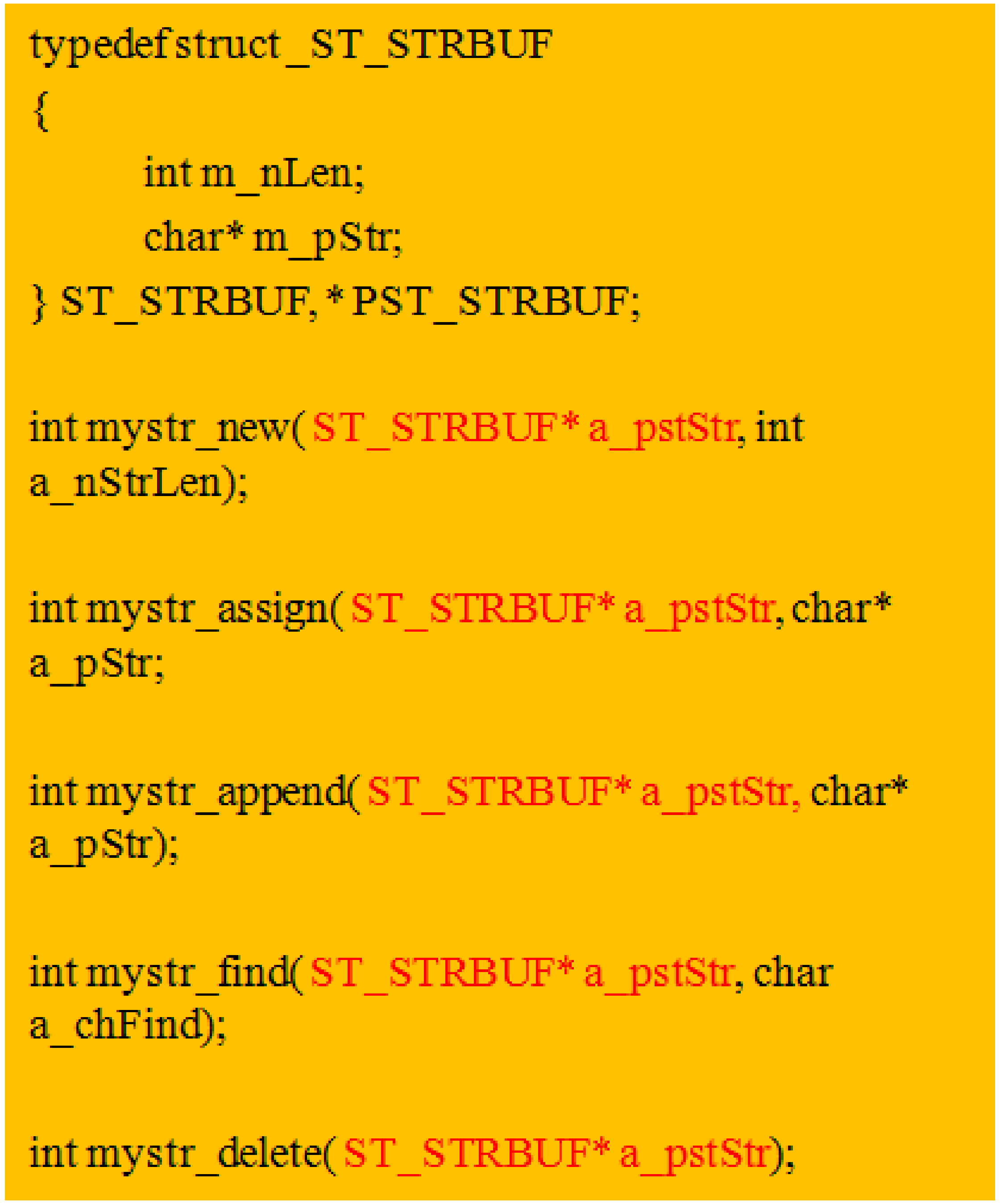
```
Sample sam1 = { 2, 5, 3.2, 'A' } ;
```

```
Sample sam2 = { 7, 3 } ;
```

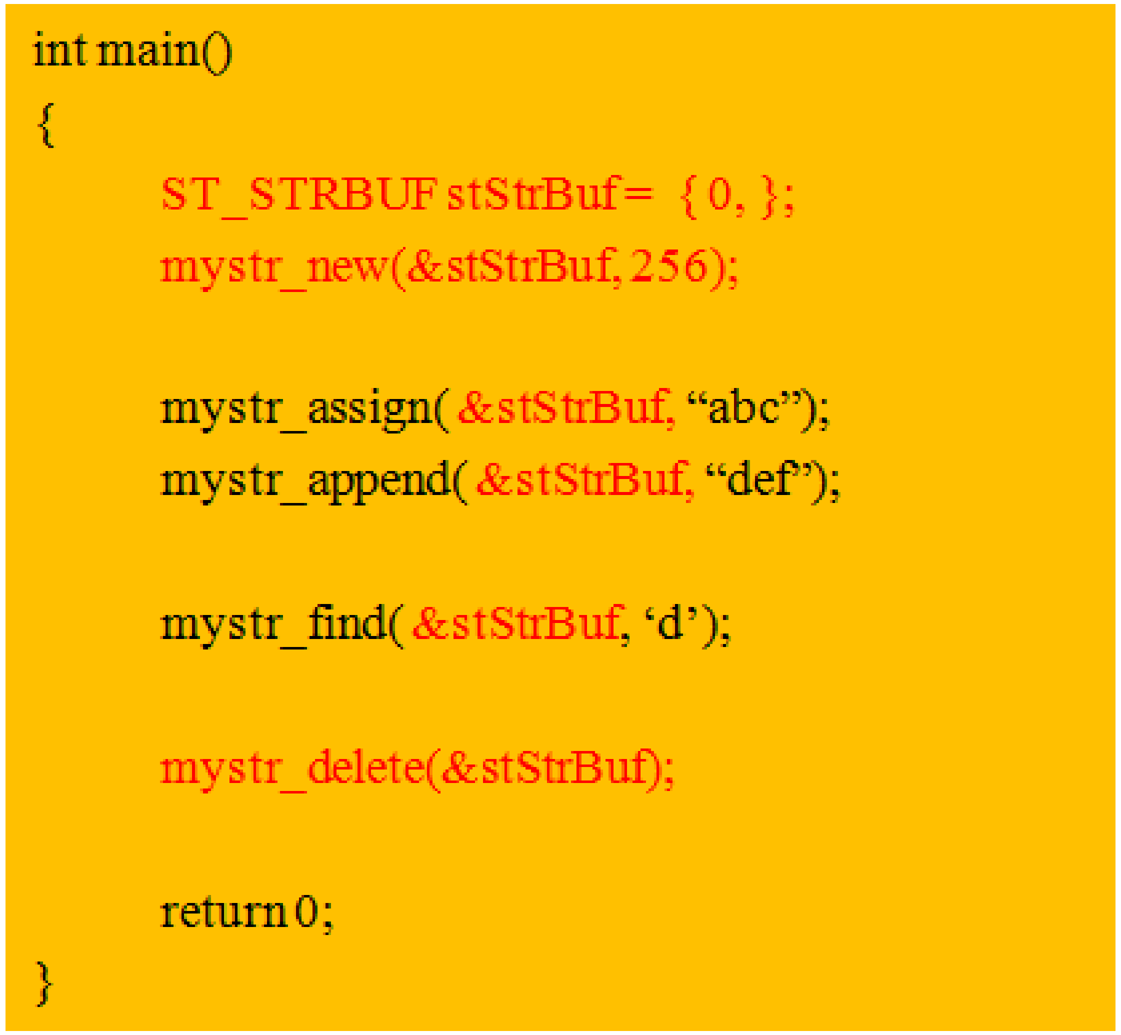
filled with binary zero

filled with float zero

String in C

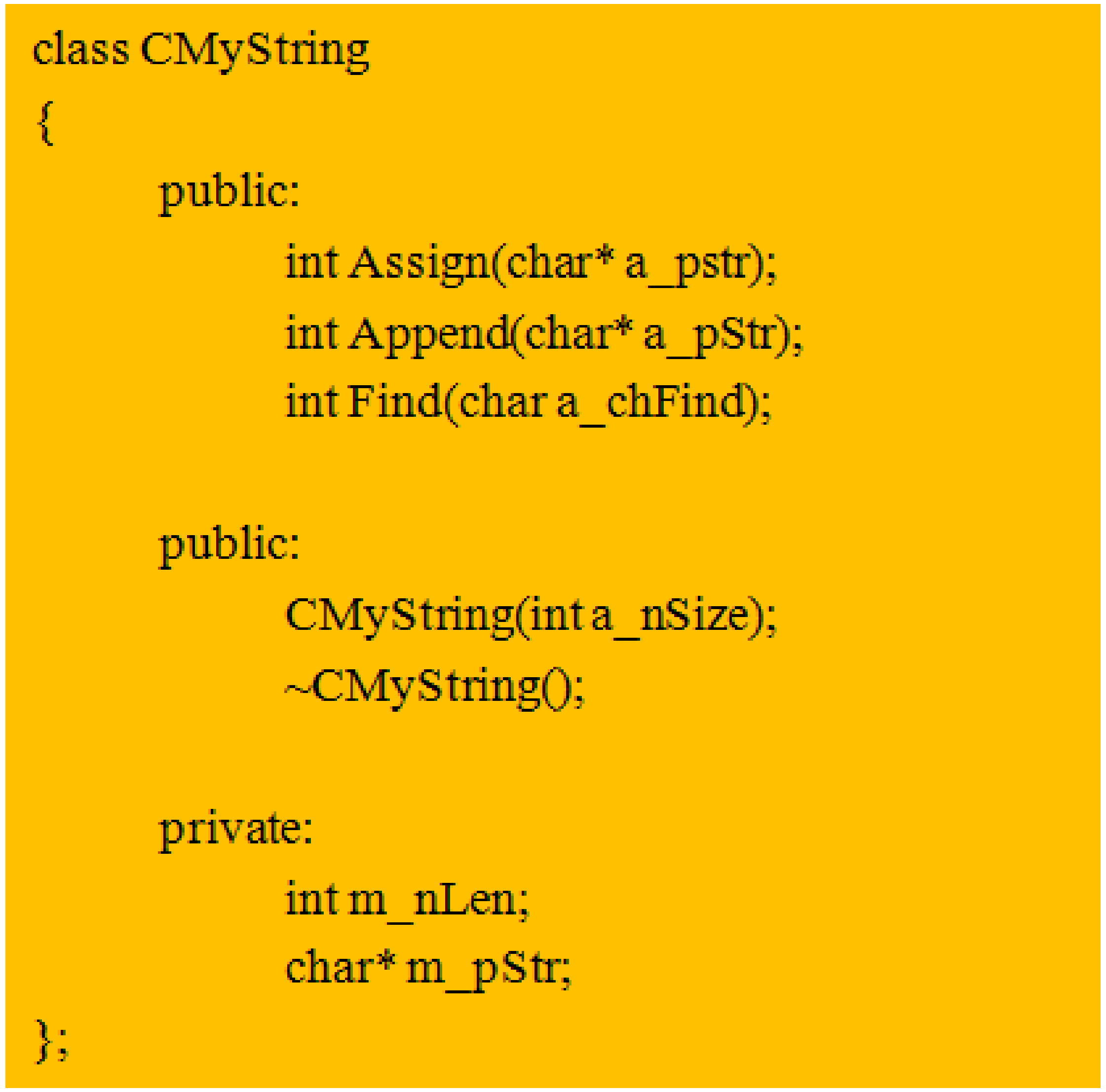


```
typedef struct _ST_STRBUF
{
        int m_nLen;
        char* m_pStr;
} ST_STRBUF, * PST_STRBUF;

int mystr_new( ST_STRBUF* a_pstStr, int
a_nStrLen);

int mystr_assign( ST_STRBUF* a_pstStr, char*
a_pStr;

int mystr_append( ST_STRBUF* a_pstStr, char*
a_pStr);

int mystr_find( ST_STRBUF* a_pstStr, char
a_chFind);

int mystr_delete( ST_STRBUF* a_pstStr);
```

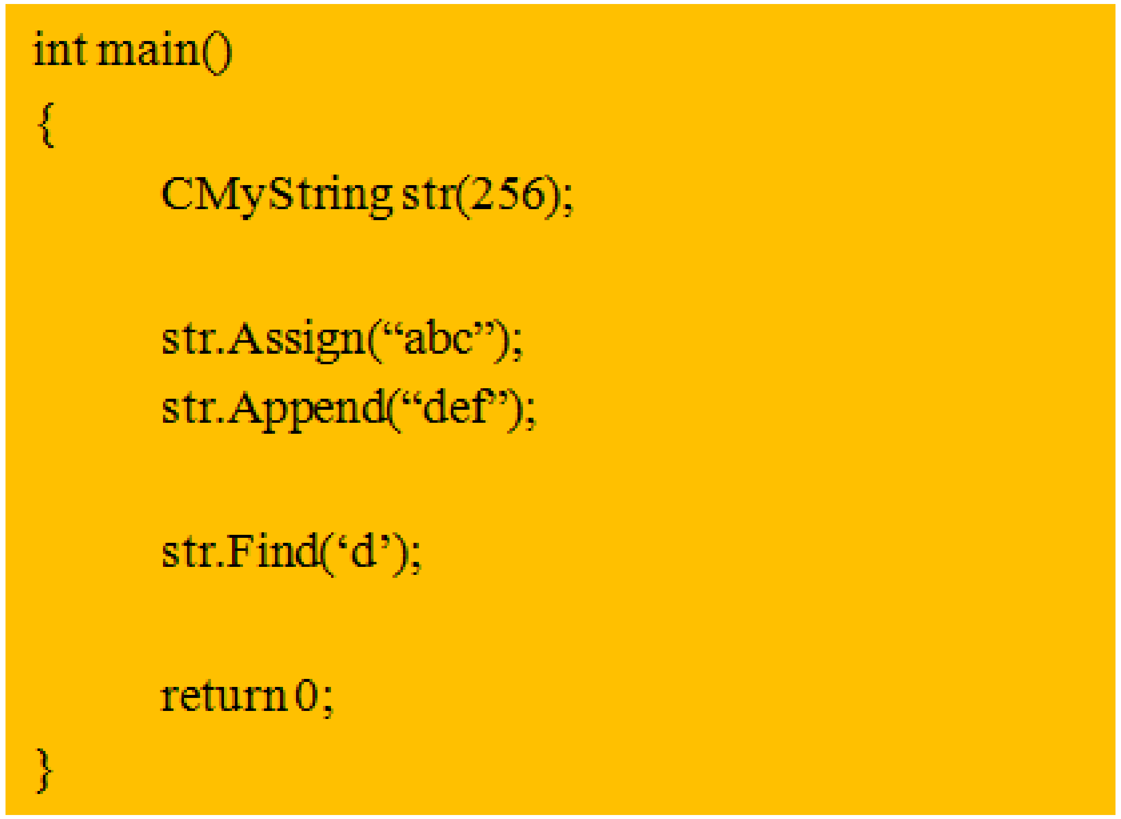


```
int main()
{
        ST_STRBUF stStrBuf = { 0, };
        mystr_new(&stStrBuf, 256);

        mystr_assign( &stStrBuf, “abc”);
        mystr_append( &stStrBuf, “def”);

        mystr_find( &stStrBuf, ‘d’);

        mystr_delete(&stStrBuf);

        return 0;
}
```

String in C++



```
class CMyString
{
        public:
                int Assign(char* a_pstr);
                int Append(char* a_pStr);
                int Find(char a_chFind);

        public:
                CMyString(int a_nSize);
                ~CMyString();

        private:
                int m_nLen;
                char* m_pStr;
};
```

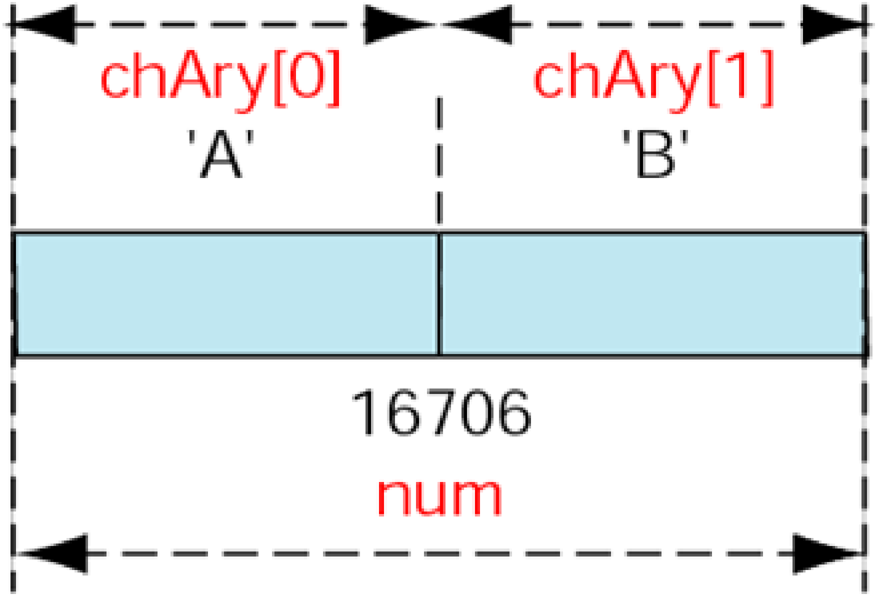


```
int main()
{
        CMyString str(256);

        str.Assign(“abc”);
        str.Append(“def”);

        str.Find(‘d’);

        return 0;
}
```

Union

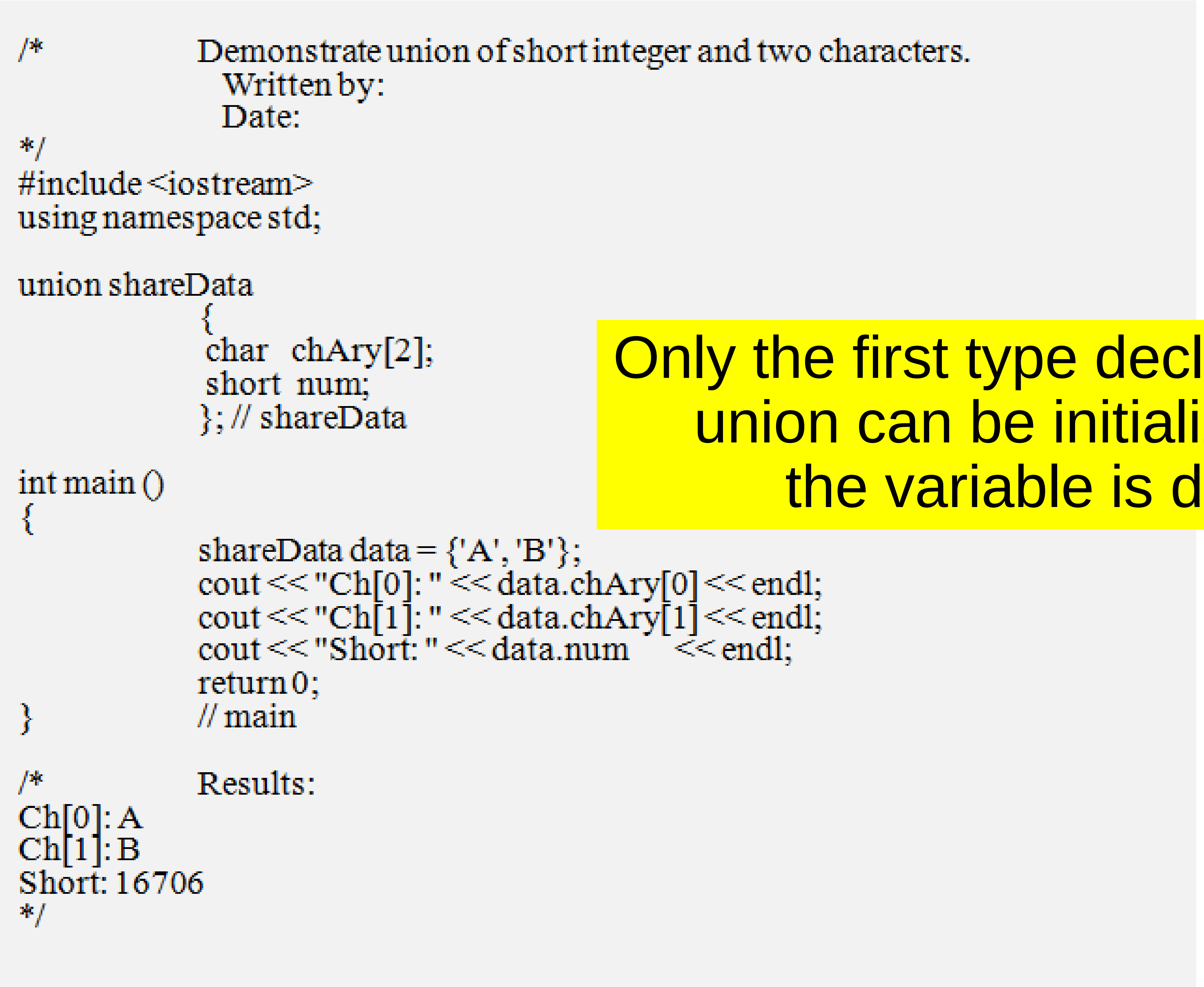
- Construct that allows memory to be shared by different types of data

```
union shareData
   {
    char    chAry[2];
    short   num;
   };
```



Both num and chAry start at the same memory address. chAry[0] occupies the same memory as the most significant byte of num.

Practice (Program 11-12)

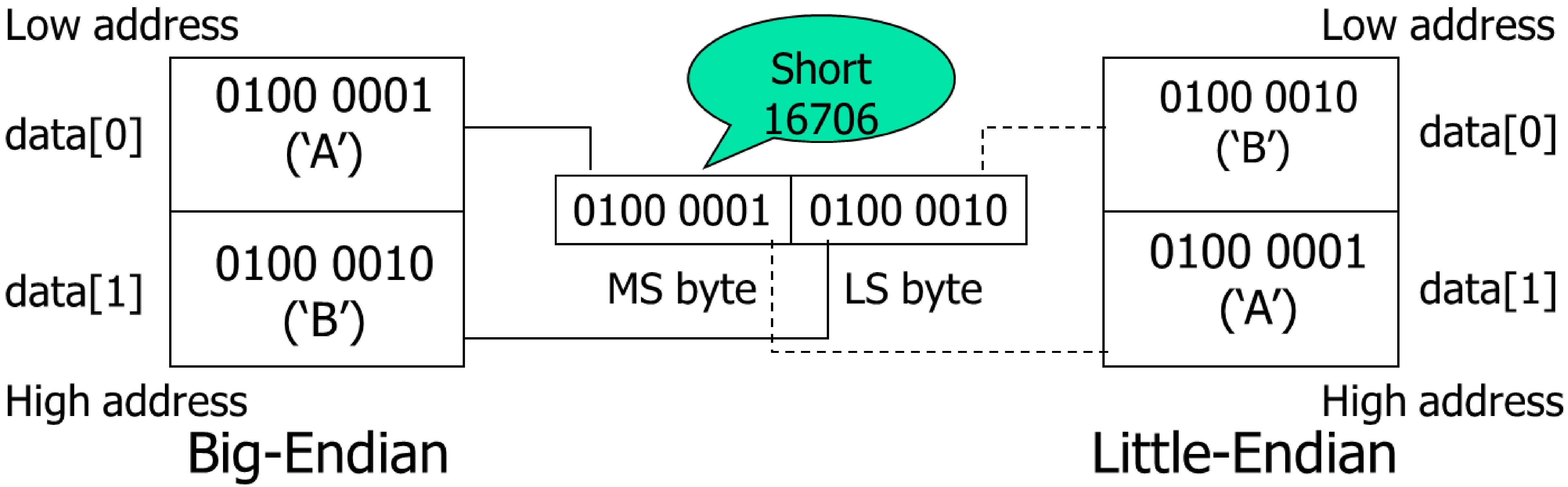


```
/*              Demonstrate union of short integer and two characters.
                  Written by:
                  Date:
*/
#include <iostream>
using namespace std;

union shareData
                {
                 char   chAry[2];
                 short  num;
                }; // shareData

int main ()
{
                shareData data = {'A', 'B'};
                cout << "Ch[0]: " << data.chAry[0] << endl;
                cout << "Ch[1]: " << data.chAry[1] << endl;
                cout << "Short: " << data.num     << endl;
                return 0;
}               // main

/*              Results:
Ch[0]: A
Ch[1]: B
Short: 16706
*/
```

Only the first type declared in the union can be initialized when the variable is defined

Big-and Little-Endian Computers

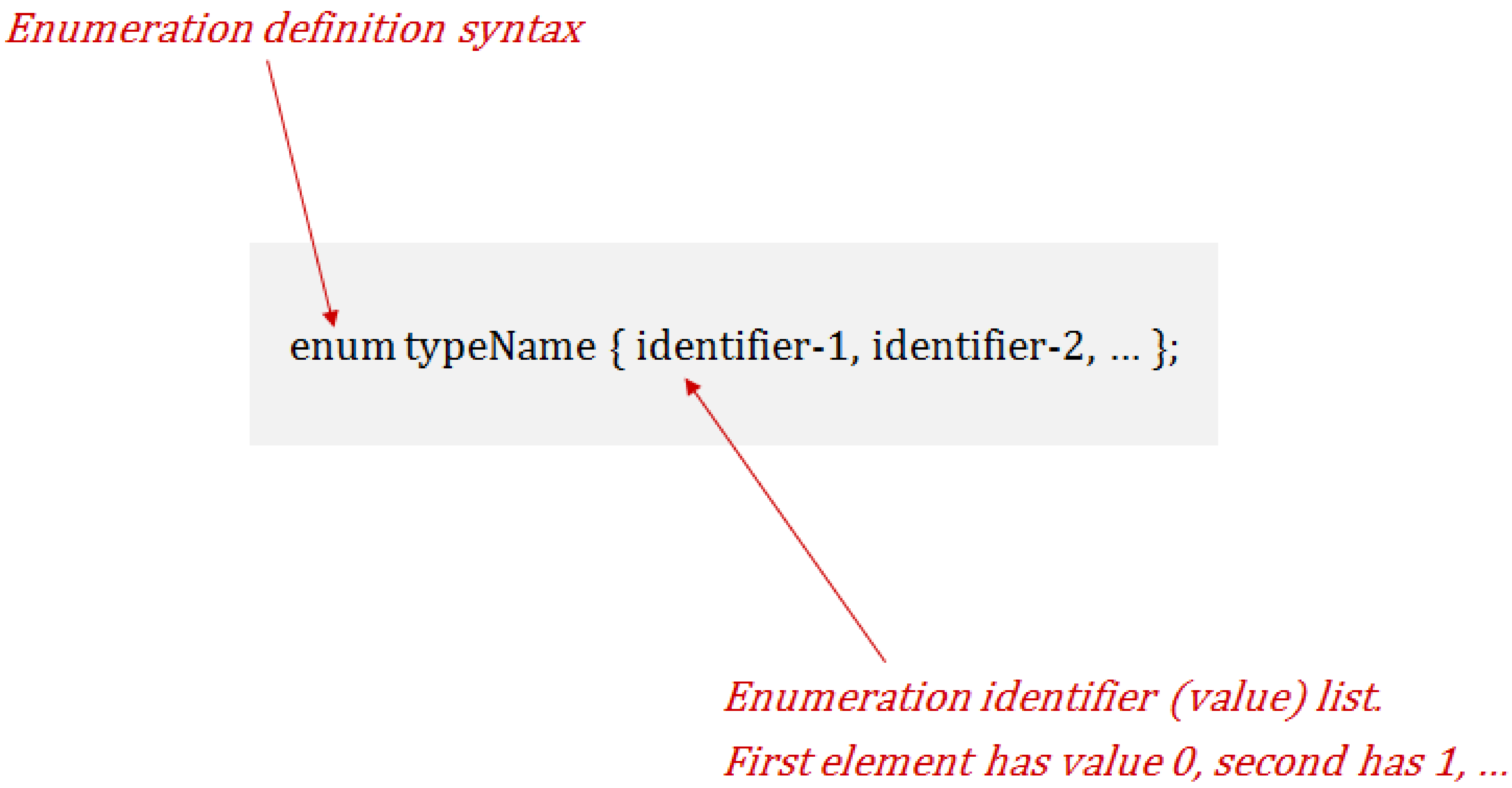
- The way to store the most significant byte (MSB) of a number varies according to computer hardware



Enumeration type

- user-defined type based on the standard integer type
- Each integer value is given an identifier called an *enumeration constant*
- Use the enumerated constants as symbolic names, which makes our programs much more readable

Enumeration type declaration



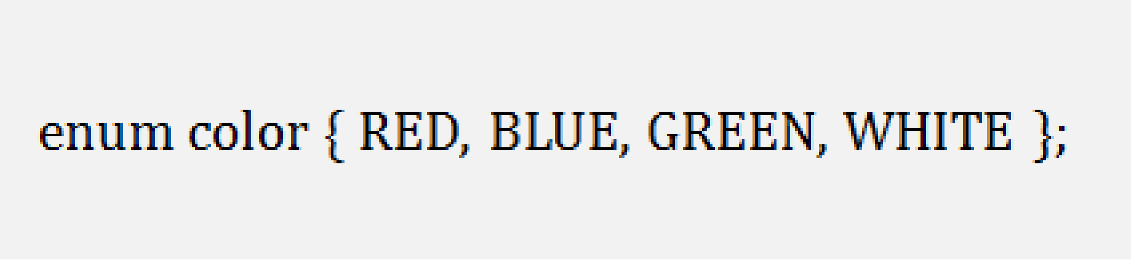
Enumeration definition syntax

```
enum typeName { identifier-1, identifier-2, ... };
```

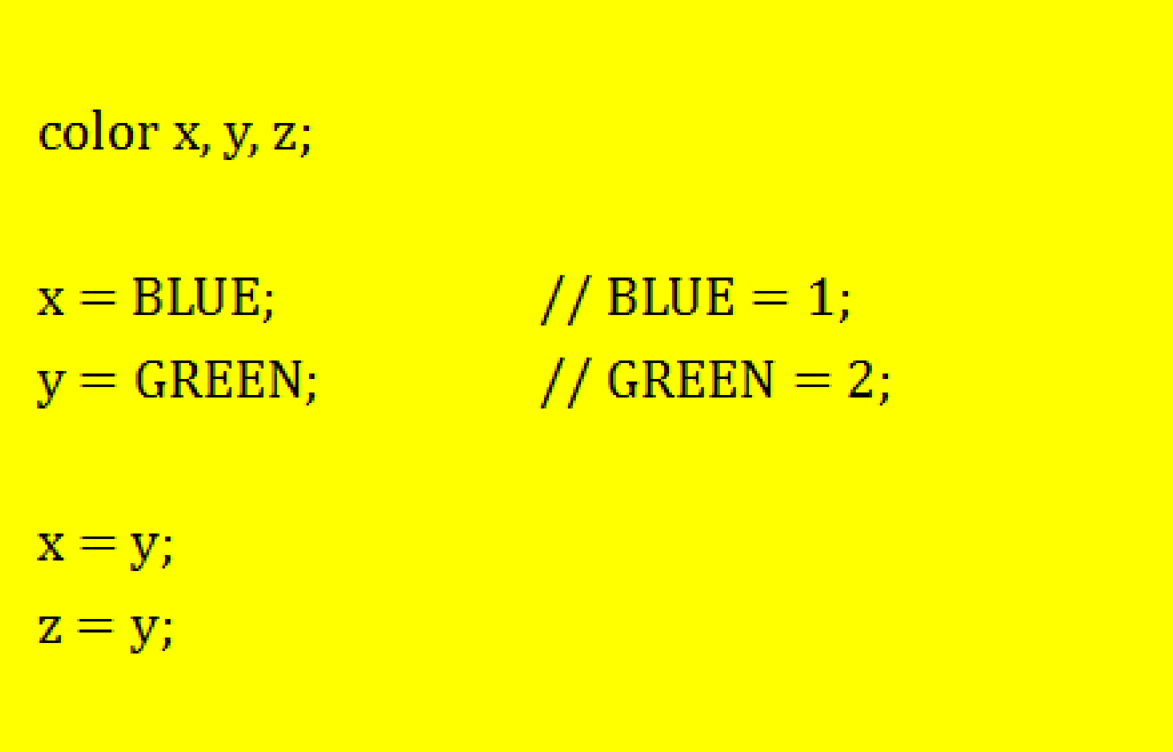
Enumeration identifier (value) list.

First element has value 0, second has 1, ...

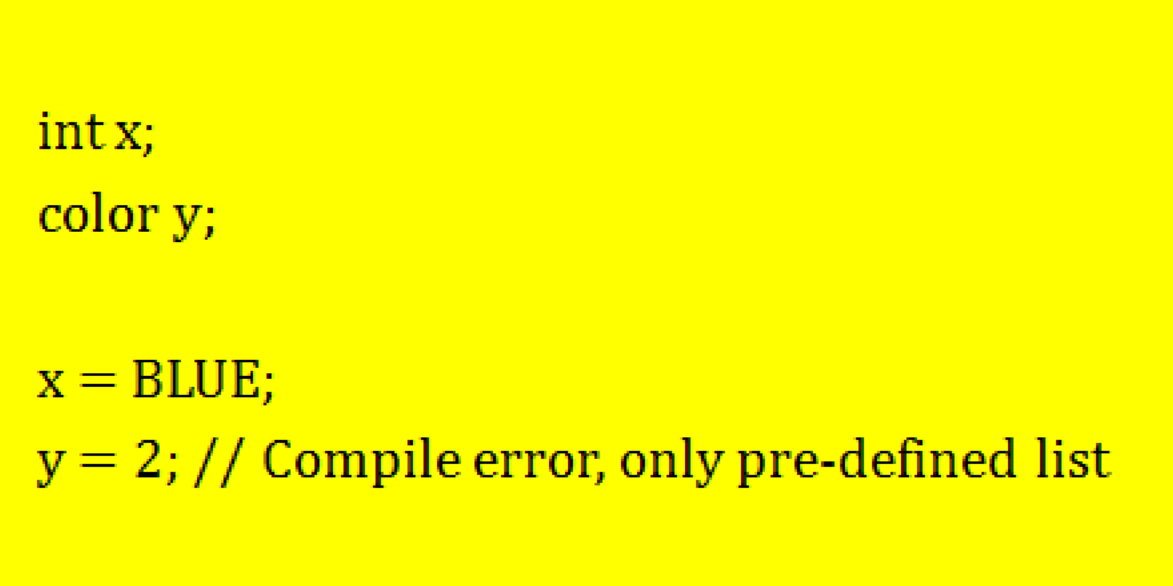
Enumeration example



```
enum color { RED, BLUE, GREEN, WHITE };
```

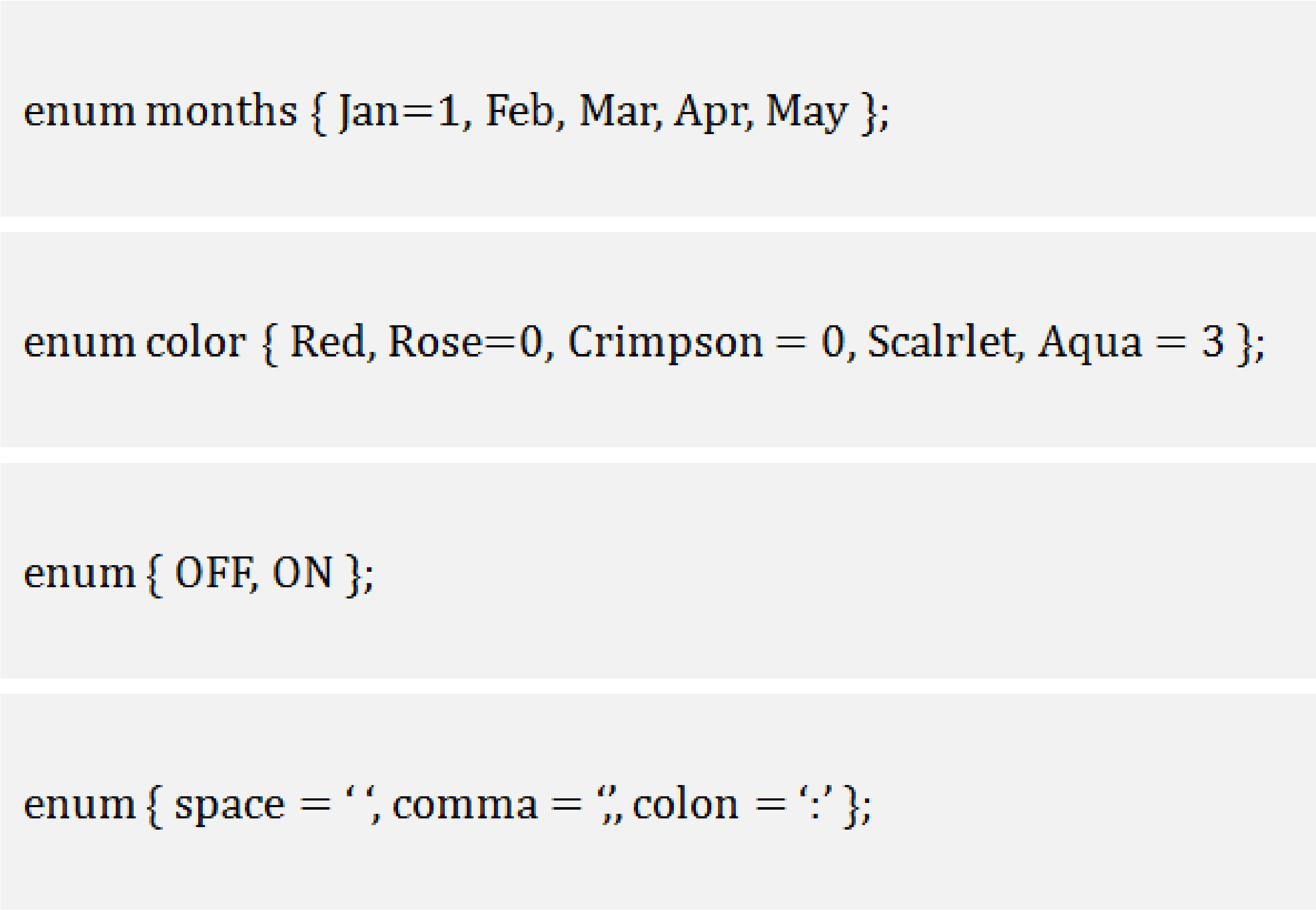


```
color x, y, z;

x = BLUE;            // BLUE = 1;
y = GREEN;           // GREEN = 2;

x = y;
z = y;
```



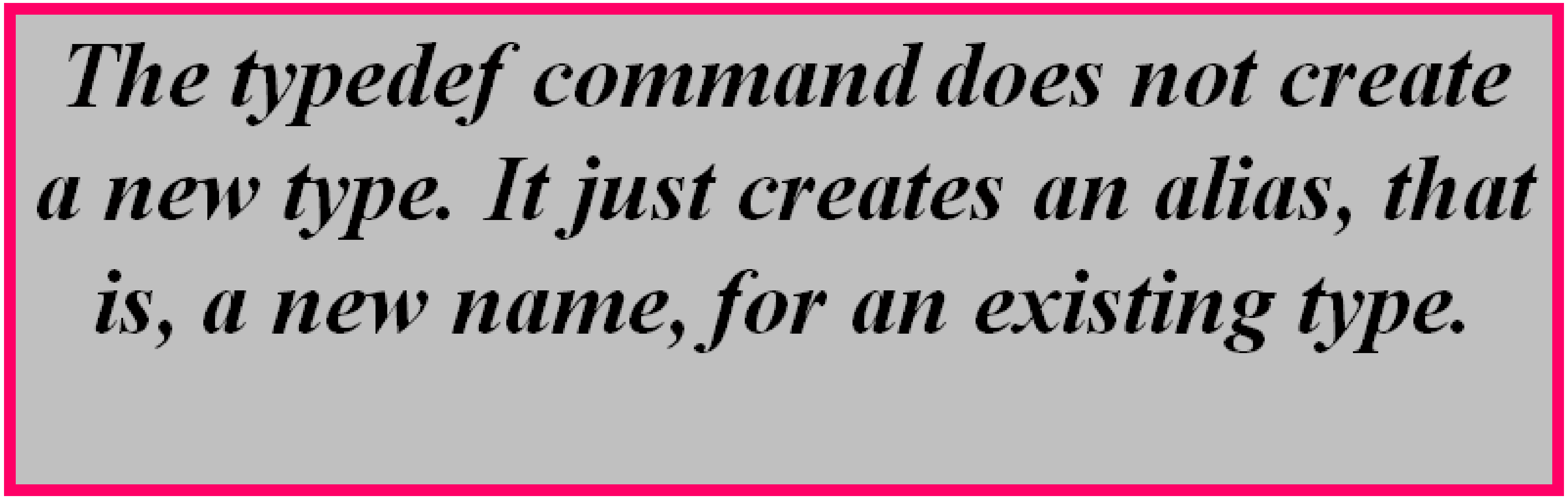
```
int x;
color y;

x = BLUE;
y = 2; // Compile error, only pre-defined list
```

More about enumeration



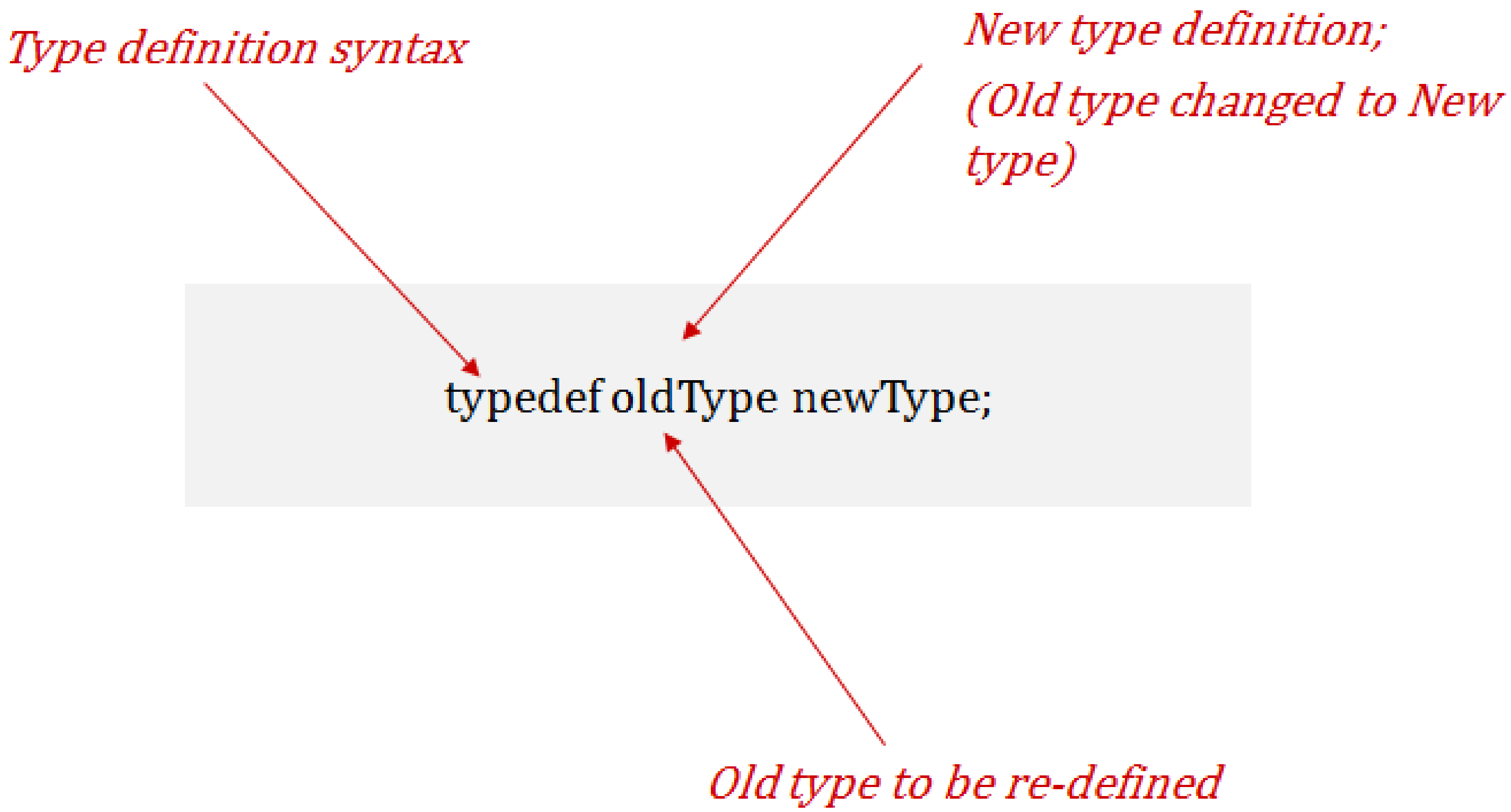
```
enum months { Jan=1, Feb, Mar, Apr, May };

enum color { Red, Rose=0, Crimpson = 0, Scalrlet, Aqua = 3 };

enum { OFF, ON };

enum { space = ‘ ‘, comma = ‘,’, colon = ‘:’ };
```

Type definition (typedef)



The typedef command does not create a new type. It just creates an alias, that is, a new name, for an existing type.

Type definition



Type definition syntax

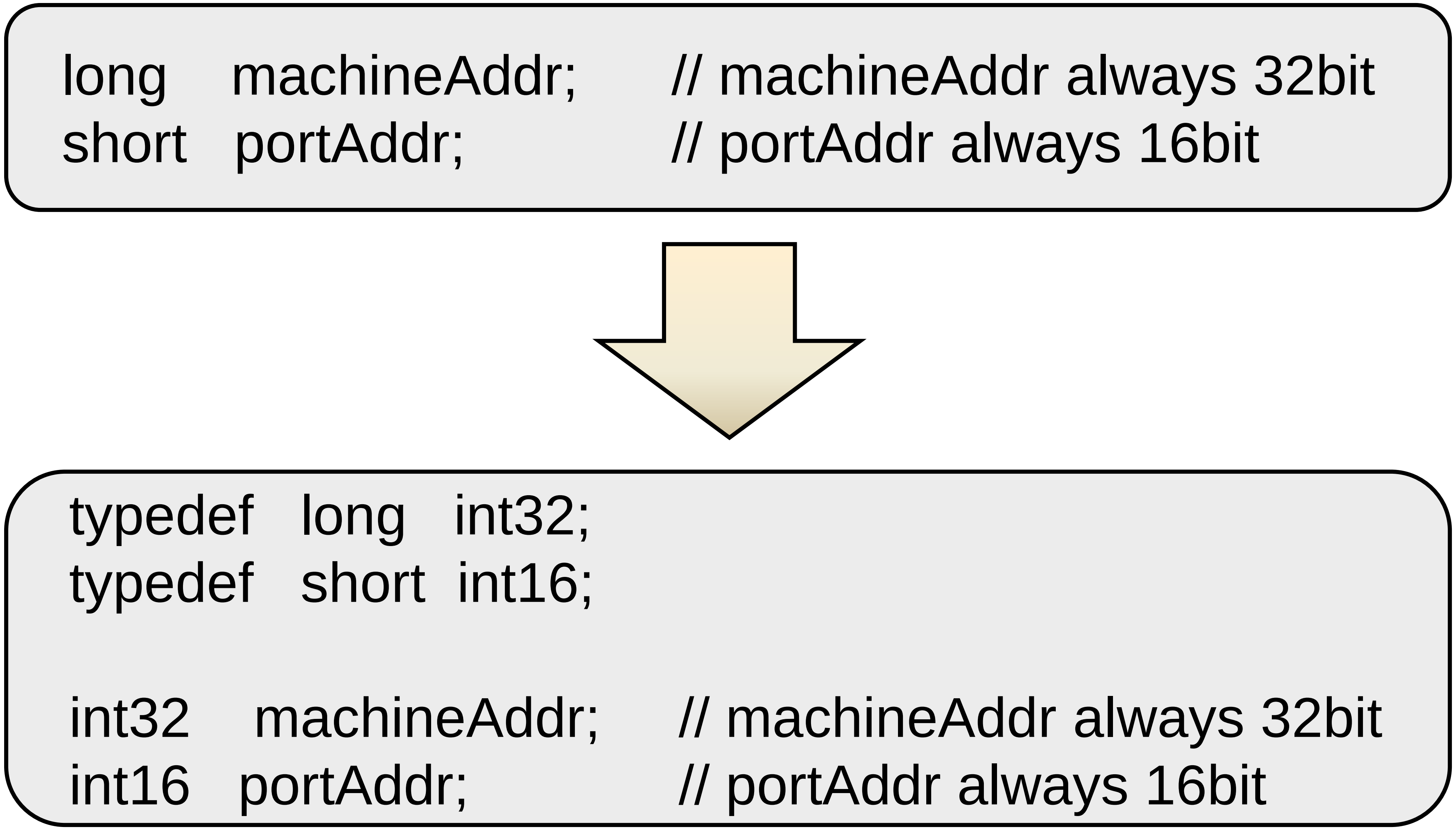
New type definition;

(Old type changed to New type)

```
typedef oldType newType;
```

Old type to be re-defined

typedef example 1:
writing portable programs



```
long    machineAddr;      // machineAddr always 32bit
short   portAddr;         // portAddr always 16bit
```

```
typedef   long   int32;
typedef   short  int16;

int32    machineAddr;     // machineAddr always 32bit
int16   portAddr;         // portAddr always 16bit
```

typedef example 2:
replacing complex declarations

```
Fraction* aryFraction[12];
```

```
typedef     Fraction*     pFraction;
pFraction   aryFraction[12];
```

```
int   aryFraction[NUM_CITIES][NUM_MONTHS];
```

```
typedef   int   TwoDimAry[NUM_CITIES][NUM_MONTHS];
TwoDimAry       highTemperatures;

highTemperatures[3][7] = 36;
…
```

Questions?