C++ Programming:
Classes

2019년도 2학기

Instructor: Young-guk Ha
Dept. of Computer Science & Engineering

Contents

- Classes: foundation of C++ programming
  - Classes and objects
    - Declarations and definitions
    - Access modifiers
    - Methods and object references
    - Overloaded methods
  - Pointers to objects
  - Constructors and destructors
  - Class members

Class Declarations

- Classes are like data types
  - Class declaration = Data type definition
  - Object creation = Variable declaration

    E.g., a built-in class **`string`** acts like a data type

- Class declaration must come before object creation

```
// class declaration
class Person {
   ...     // data members and methods definitions go here
};         // *** semicolon is REQUIRED ***

Person maryLeakey;              // creation of an object
Person korean[49000000];        // an array of objects
```

Defining Methods

- A method can be defined either
  - *Inside* of the class declaration, or
  - *Outside* of the class declaration

```
class Person {
 public:
   // methods defined inside
   //   of class declaration
   void setAge(unsigned n)
                  { age = n;  }
   unsigned getAge()
                  { return age; }
 private:
   unsigned age;
};
```

```
class Person {
 public:
   void setAge(unsigned n);
   unsigned getAge();
 private:
   unsigned age;
};

// methods defined outside
void Person::setAge(unsigned n)
                  { age = n; }

unsigned Person::getAge()
                  { return age; }
```

Defining Inline Methods

- A method defined inside the declaration is said to be **`inline`** implicitly

```
class Person {
public:
  // if defined inside of class
  //  implicitly declared inline
  void setAge(unsigned n)
                  { age = n;   }
  unsigned getAge()
                  { return age; }
private:
  unsigned age;
};
```

=

```
class Person {
public:
 // explicitly declared inline
 inline void setAge(unsigned n);
 inline unsigned getAge();
private:
 unsigned age;
};

void Person::setAge(unsigned n)
                  { age = n; }

unsigned Person::getAge()
                  { return age; }
```

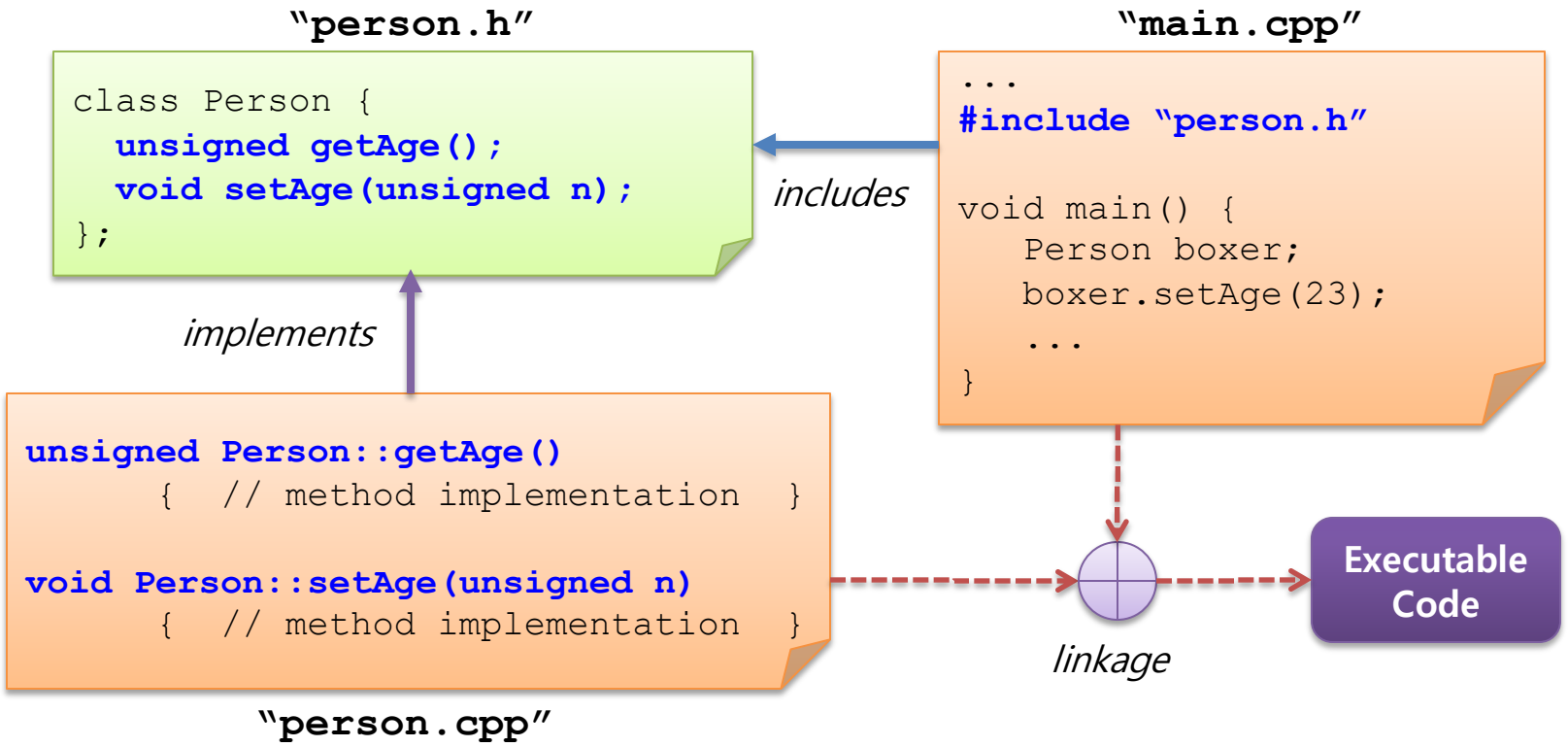
Accessing Members

- We can access object members using ***member selection operator*** "**.**"
  - The member selection operator is also used for object references
  - Note that we must use ***class indirection operator*** "**->**" for object pointers

```
Person p1;
cout << p1.getAge() << '\n';

Person& p2;
cout << p2.getAge() << '\n';

Person* p3;
cout << p3->getAge() << '\n';
```

Writing Classes into Files

- Normally, class declarations are placed in header files and then included whenever needed
  - Method definitions are placed in separate CPP files (note **`inline`** methods are commonly defined in header files)



C++ Access Modifiers

- C++ provides the following access modifiers for data members and methods of class
  - **`public`**
    - Used to expose data and methods
    - I.e., public members of class are visible to other classes and functions (called ***public scope***)
  - **`private`**
    - Used to hide data and methods
    - I.e., private members of class are visible only in the same class (***class scope***) or **`friend`** functions (to be discussed later)
  - **`protected`**
    - Allowing classes in the same inheritance hierarchy to access members (to be discussed later)
    - I.e., protected members of a class are accessible only from its derived classes (subclasses)

Access Modifier Examples

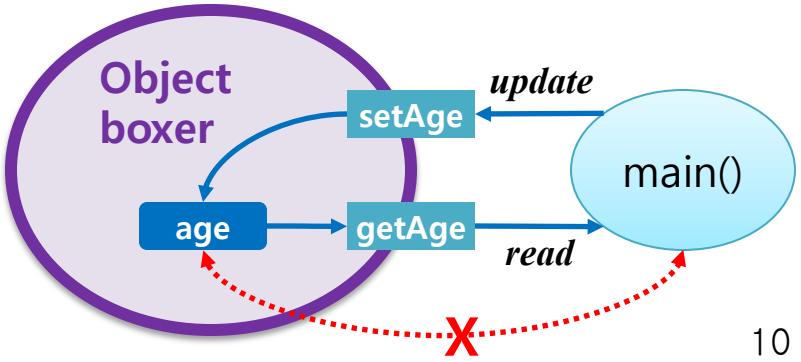
```
// preferable style:
//   put the public members
//   (interfaces) first,
//   this adds clarity to
//   the declaration
class Person {
 private:
   unsigned age;
 public:
   void setAge(unsigned n);
   unsigned getAge();
};
```

```
// not a good style:
//   public and private
//   members are intermixed
//   within the class
//   declaration
class Person {
 public:
   void setAge(unsigned n);
 private:
   unsigned age;
 public:
   unsigned getAge();
};
```

Encapsulation with `private` Modifier

```
class Person {
 // hidden from outside
 private:
   unsigned age;
 public:
   void setAge(unsigned n);
   unsigned getAge();
};

// interface to hidden members
void Person::setAge(unsigned n)
{
   age = n;      // update age
}
unsigned Person::getAge()
{
   return age;   // read age
}
```

```
void main() {
  Person boxer;

  boxer.age = 27;
  cout << boxer.age << '\n";
       // error: age is private
       //        (not accessible)

  boxer.setAge(27);
  cout << boxer.getAge() << '\n';
}
```



C++ Default Access Scope

- Recall that a class can be defined with either keyword **`class`** or **`struct`** in C++

- For each case, data members and methods of the class have different default access scopes
  - If **`class`** is used, all the members default to **`private`**
  - If **`struct`** is used, all the members default to **`public`**

```
class C {
   int x;
   // private by default
 public:
   void setX(int x);
};
```

=

```
struct C {
   void setX(int x);
   // public by default
 private:
   int x;
};
```

❖ To emphasize the OO principle of ***information hiding***, it is preferable to use **`class`**, where all the members default to **`private`**

Passing and Returning Objects

- Like other variables, objects can be passed to or returned from functions or methods
  - Call/return by value (default)
  - Call/return by reference (as pointer or reference type)
- Objects required to be passed or returned **by reference**
  - Passing or returning objects by value could be inefficient due to **copying objects**, which may cause waste of storage and time
  - References are easier to use than pointers because no dereferencing is required
- When returning local objects
  - The returned object must be **`static`**, otherwise, the invoker may receive a nonexistent object
  - Note that once initialized, **`static`** objects last the program's whole execution time (similar to global objects except for naming scope)

Passing and Returning Objects by Reference

```
class C {
 private:
   int num;

 public:
   void set(int n) {
      num = n;
   }

   int get() {
      return num;
   }
}
```

```
-999
123
```

```
void f(C& c) {
   c.set(-999);
}

C& g() {
   static C c;
   c.set(123);
   return c; // storage for c will
             //  remain even after return
}

void main() {
   C c1, c2;
   f(c1);
   cout <<< c1.get() <<< '\n';
   c2 = g();
   cout << c2.get() << '\n';
}
```

Passing Object Arrays by Reference

- By default, object arrays are passed by reference

```
// E.g., passing an array of C objects by reference
//       (reference of the 1st element of the array)

void useArray(C arr[])
{
   ...
   arr[0].set(123);
   ...
   arr[6].set(789);
   ...
}

C c_arr[10];
useArray(c_arr);
```

`const` Object References and `const` Methods

- Keyword **`const`** can be used in method headings in conjunction with object references
  - Keyword **`const`** is placed in front of the parameter type
    - This means that the parameter is **`const`**
    - I.e., prevented from being changed by unwanted write operations within the method body
  - Keyword **`const`** is placed in front of the return type
    - This means that the return value is **`const`**
    - I.e., prevented from being changed by unwanted write operations (placed on the left-hand side of "=") within the caller method or function
  - Keyword **`const`** is placed in front of the method body
    - This means that the method itself is **`const`**
    - I.e., not to change any data member of the object either directly or indirectly by invoking other methods that change the data members
    - Thus, a **`const`** method can invoke only other **`const`** methods

Examples of Using const

```
class C1 {
 private:
  static string name;

 public:
  // const parameter
  void set(const string& n) {
    name = n;
    n = "noname";   }   // error

  // const return
  const string& get() {
    return name;    }
};
                ...
C cobj;
cobj.set("test1");
string n = cobj.get();
cobj.get() = "test2";  // error
```

```
class C2 {
 private:
  int dm;

 public:
  // const method
  void m1(int x) const {
     dm = 10;  // error
     m2(x);
       // error: m2 is not const
  }

  void m2(int x) {
     dm = x;
  }
};
```

Method Overloading

- Overloaded methods
  - Identically named methods with distinct signatures can be declared within a single class (cf., function overloading)

```
class C {
 public:
   void set(const string& n)   { name = n; }
   void set(const char* n)     { name = n; }

   const string& get() const   { return name; }     // error
   const char*   get() const   { return name.c_str(); }
 private:
   string name;
};
```

```
C c1;

string s1 = "Who's Afraid of
             Virginia Wolf?";
```

```
c1.set(s1);  // string

c1.set("What, me worry?");
                // C-string
```

Pointers to Object

- Pointers to object are frequently used in C++ codes in the following context
  - When using dynamically allocated objects by **new** and **new[]**
    - E.g.)
```
Person* boxer = new Person;
Person* boxers = new Person[10];
```
  - When using call or return by reference

- Access to an object's members through a pointer
  - Using the class indirection operator "->" instead of the member selector operator "."
    - Cf., to access object members through references, the member selector operator "." is used

Using Object Pointers

```
class C {
 public:
   void m() {  ...  }
};

void f(C*);
C* g();

void f(C* p) {
   p.m();
   // error: p is a pointer

   p->m();
}
```

```
// a factory function
C* g() {
   C* c = new C;
   // creates a C object

   return c;
}

void main() {
   C c1;
   C* c2;

   f(&c1);
   c2 = g();
   c2->m();
}
```

Pointer Constant `this`

- The pointer constant **`this`** can be used inside a method to access the object containing the method itself

```
class C {
 private:
   int x;
     ...
 public:
   m() { x = 0; }
     ...
}
```

=

```
class C {
 private:
   int x;
     ...
 public:
   m() { this->x = 0; }
     ...
}
```

Usage of `this` (1)

- To avoid name conflicts
  - Sometimes, programmers prefer to give parameters the same name as the data member to be accessed
  - In this case, **`this`** can be useful to avoid a name conflict

```
Private:
  string id;
        ...
public:
  void setID(string& id)
  {
     id = id;
          // name conflict
  }
        ...
```

```
void setID(string& id_val)
{
   id = id_val;
}
```

```
void setID(string& id)
{
   this->id = id;
}
```

Usage of `this` (2)

- To test if an object given as an argument of a method is the object to which the method belongs
  - In the following example, the **`f1.copy`** method tests if a **`File`** object **`f1`** itself is given as the destination of the copy

```
void File::copy(File& dest)
{
   // copy src to dest file
   ...
}

void main()
{
   File f1;

   f1.copy(f1);
   // can't copy a File to
   //   itself
}
```

```
void File::copy(File& dest)
{
   // check if the src and
   //   the dest objects are
   //   the same?
   if (this == &dest) ...;

   // if different,
   //   copy the src file to
   //   the dest file
   ...
}
```

Usage of `this` (3)

- Notices on using **`this`**
  - "**`this`**" is a constant, thus, it can not be changed
  - "**`this`**" is available only within non-**`static`** methods
    → implies objects must be created prior to being referred with **`this`**

```
class C {
 private:
   static int count;

 public:
   void m(const C* obj) {
      this = obj;          // error: "this" is a constant
        ...
   }

   static void s() {
      this->count = 0;     // error: s() is a static method
   }                       //      (* refer to class member)

};
```

Constructors and Destructors

- Commonly, methods of a class are invoked manually using function call operator "( )"

- Some special methods are not required to be invoked manually
  - They are automatically invoked by the compiler
  - Those methods are called class ***constructor*** and ***destructor***
    - ***Constructors*** are automatically invoked when objects are created
      - For **initialization** process: to allocate memories or set initial values of data members
    - ***Destructors*** are automatically invoked when objects are removed
      - For **finalization** process: to delete allocated memories or clear existing values of data members

Constructors

- A constructor is a method whose name is **the same as the class name** and has **no return type**
  - Can be overloaded: the most suitable constructor is invoked automatically every time an object of the class is created
  - Types of constructors
    - Default constructors (without parameters)
      - If no constructors are provided, the compiler generated one automatically
    - Parameterized constructors (copy constructors, convert constructors)

```
class Person {
public:
   Person();                          // default constructor
   void Person();                     // error: no return type required
   Person(const string& n);           // parameterized constructor
   Person(const char* n);             // parameterized constructor
   ...
   void setName(const string& n);
private:
   string name;
};
```

Defining Constructors

- Constructors can be defined inline or outside class declarations

```
class Person {
public:
   Person() { name = "Unknown"; }          // defined inline
   Person(const string& n);
   Person(const char* n);

   void setName(const string& n);
   void setName(const char* n);
   const string& getName() const;
private:
   string name;
};

Person::Person(const string& n) {          // defined outside
   name = n;
}

Person::Person(const char* n) {            // defined outside
   name = n;
}
```

Invocation of Constructors

- The most appropriate constructor is invoked automatically whenever an object is created
  - Programmers do not need to invoke appropriate constructor to initialize the object
  - This feature would make classes robust

```
#include "Person.h"

void main() {
   Person anonymous;                  // invokes the default
                                      //   constructor: Person()

   Person jc("J. Coltrane");          // invokes a parameterized
                                      //   constructor:
                                      //      Person(const char* n)
   ...
}
```

Arrays of Objects and the Default Constructor

- The default constructor is invoked for each **C** object within the array of **C** objects

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

unsigned count = 0;

class C {
public:
   C() { cout << "Creating C" << ++count << '\n'; }
};

C ar[1000];    // array of 1,000 C objects
```

```
Creating C1
Creating C2
...
Creating C999
Creating C1000
```

Restricting Object Creation with Constructors

- Restriction by making selected constructors **`private`**
  - E.g., C++ programmers can ensure that the objects are properly initialized with some specific parameters by making the default constructor **`private`**

```
class Emp {
public:
   Emp(unsigned id) { id_no = id; }

private:
   Emp();          // emphasize to prevent an Emp obj
                   //       from being created w/o a given id

   unsigned id_no;         // employee's ID number
   ...
};

int main() {
   Emp elvis;     // error: Emp() is private
   Emp cher(12345);
   ...
}
```

Detailed classification of constructors

- Class constructors
  - Default constructors
  - Parameterized constructors
    - ***Copy constructors***
    - ***Convert constructors***

Copy Constructors

- Create new objects by **copying** another object

```
// Declaration of copy constructors
Person(const Person& p);  // must have a reference parameter
                          //   of the same class

Person(const Person& p, bool married = false);
    // when there are more than one parameter, every
    //   parameter beyond the first must have a default value

// Usage of copy constructors
Person orig("Dawn Upshaw");       // create a Person object
Person clone(orig);               // clone it by CC
```

❖ *If a programmer does not provide any copy constructor, the compiler automatically provides one*

Defining a Copy Constructor

- Copy constructor is defined either by
  - a programmer, or
  - the compiler (if not defined by the programmer)

- Compiler-supplied copy constructor
  - Simply copies each value of original object's data members to its own data members
  - Note that there could be the danger of using a compiler-supplied copy constructor if the class have pointer data members
    - It simply **copies** the pointer data member of the original object to that of the target even though the programmer wants the target to have a **separate storage** for its pointer data member

Example of Using Compiler-Supplied Copy Constructor

```
#include <iostream>
#include <string>
using namespace std;
class Namelist {
public:
   Namelist() { size = 0; p = 0; }
   Namelist( const string [ ], int );
   void set( const string&, int );
   void set( const char*, int );
   void dump() const;
private:
   int size;
   string* p;
};
Namelist::Namelist( const string s[ ], int si ) {
   p = new string[ size = si ];
   for ( int i = 0; i < size; i++ )
      p[ i ] = s[ i ];
}
void Namelist::set( const string& s, int i ) {
   p[ i ] = s;
}
void Namelist::set( const char* s, int i ) {
   p[ i ] = s;
}
void Namelist::dump() const {
   for ( int i = 0; i < size; i++ )
      cout << p[ i ] << '\n';
}
int main() {
   string list[ ] = { "Lab", "Husky", "Collie" };
   Namelist d1( list, 3 );
   d1.dump(); // Lab, Husky, Collie
   Namelist d2( d1 );
   d2.dump(); // Lab, Husky, Collie
   d2.set( "Great Dane", 1 );
   d2.dump(); // Lab, Great Dane, Collie
   d1.dump(); // ***** Caution: Lab, Great Dane, Collie
   return 0;
}
```

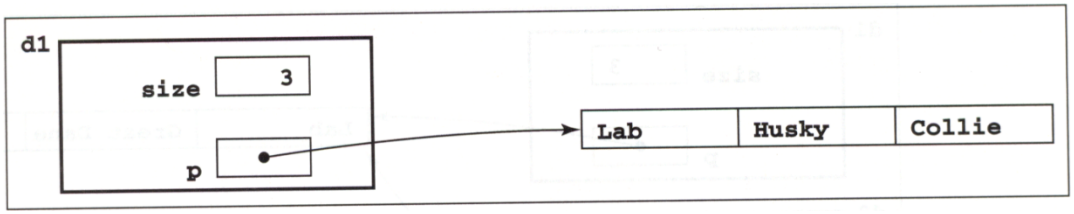
No copy constructor defined

Pointer data member

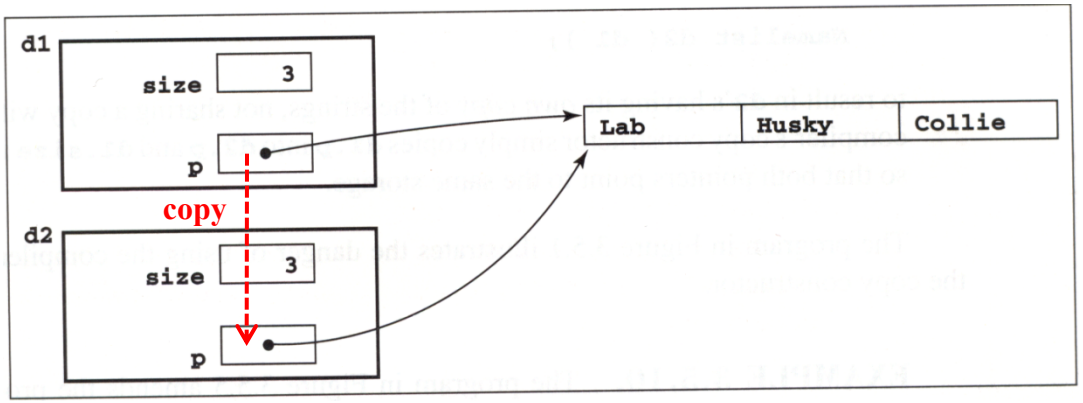
Invoking compiler-supplied copy constructor: simply copies pointer d1.p to d2.p

Example of Using Compiler-Supplied Copy Constructor (Cont.)

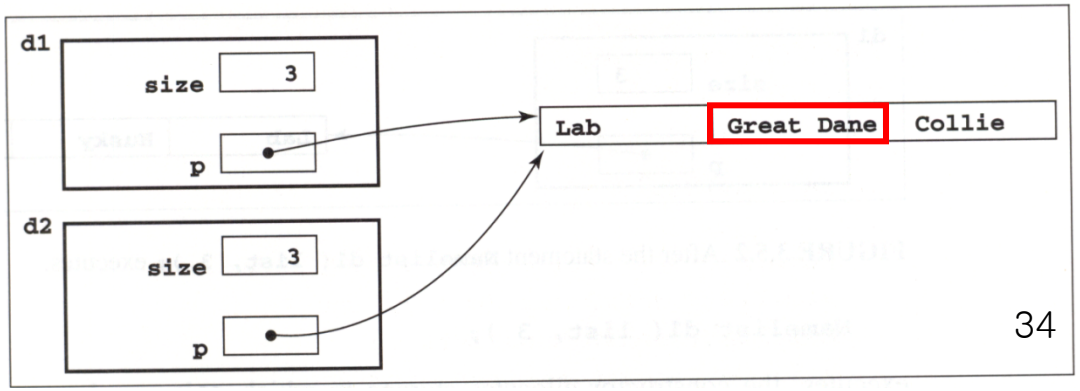
```
string list[] = {"Lab","Husky","Collie"};

Namelist d1(list, 3);
d1.dump(); → Lab
             Husky
             Collie
```




```
Namelist d2(d1);
d2.dump(); → Lab
             Husky
             Collie
```




```
d2.set("Great Dane", 1);
d2.dump(); → Lab
             Great Dane
             Collie
d1.dump(); → Lab
             Great Dane
             Collie
```



Amending the Previous Example by a Programmer-Written Copy Constructor

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

class Namelist {
public:
   Namelist() { size = 0; p = 0; }
   Namelist( const string [ ], int );
   Namelist( const Namelist& );
   void set( const string&, int );
   void set( const char*, int );
   void dump() const;
private:
   int size;
   string* p;
   void copyIntoP( const Namelist& );
};

Namelist::Namelist( const string s[ ], int si ) {
   p = new string[ size = si ];
   for ( int i = 0; i < size; i++ )
      p[ i ] = s[ i ];
}
Namelist::Namelist( const Namelist& d ) {
   p = 0;
   copyIntoP( d );
}
void Namelist::copyIntoP( const Namelist& d ) {
   delete[ ] p;
   if ( d.p != 0 ) {
      p = new string[ size = d.size ];
      for ( int i = 0; i < size; i++ )
         p[ i ] = d.p[ i ];
   }
   else {
      p = 0;
      size = 0;
   }
}
void Namelist::set( const string& s, int i ) {
   p[ i ] = s;
}
void Namelist::set( const char* s, int i ) {
   p[ i ] = s;
}
```

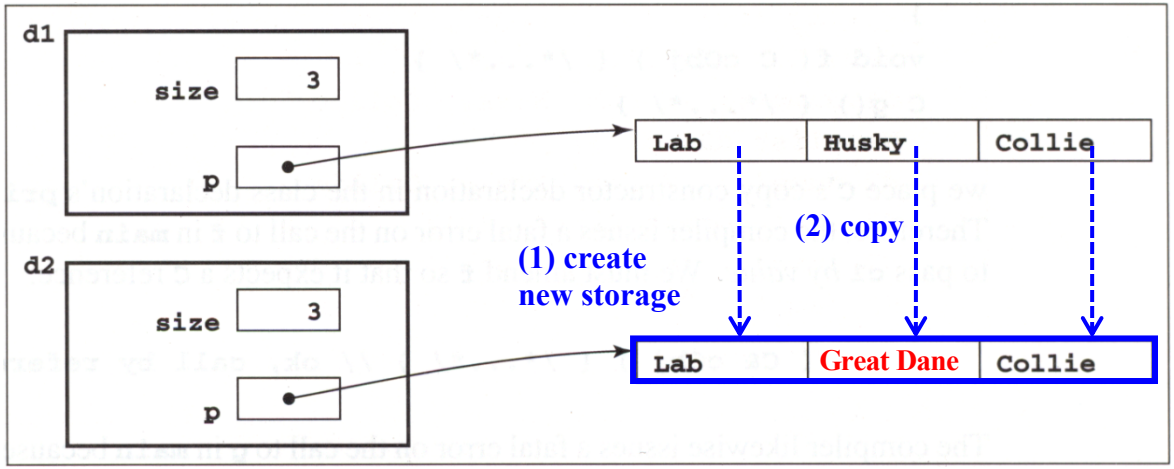
Programmer-written copy constructor

```
void Namelist::dump() const {
   for ( int i = 0; i < size; i++ )
      cout << p[ i ] << '\n';
}
int main() {
   string list[ ] = { "Lab", "Husky", "Collie" };
   Namelist d1( list, 3 );
   d1.dump(); // Lab, Husky, Collie
   Namelist d2( d1 );
   d2.dump(); // Lab, Husky, Collie
   d2.set( "Great Dane", 1 );
   d2.dump(); // Lab, Great Dane, Collie
   d1.dump(); // Lab, Husky, Collie
   return 0;
}
```

Amending the Previous Example by a Programmer-Written Copy Constructor (Cont.)

- The programmer's version of copy constructor
  - Ensures that the new object and the original object have **separate copies** of the same data member
  - I.e., it does not simply copy **`d1.p`** to **`d2.p`**, but **creates separate storage** for **`d2.p`** first and then copies each element (**`string`**) in the storage to which **`d1.p`** points

```
Namelist d2(d1);

d2.set("Great Dane", 1);
d2.dump(); → Lab
             Great Dane
             Collie

d1.dump(); → Lab
             Husky
             Collie
```



Disabling Passing and Returning by Value for Class Objects

- Recall that, in C++, objects should be passed or returned by reference
  - To remove the inefficiency of copying objects
  - Normally, it is the programmer's responsibility

- Sometimes, a class designer may wish to prevent programmers from copying objects
  - E.g., authors of C++ windows classes typically disable copying of windows objects which are generally very large

- Such object copy prevention can be achieved by declaring the copy constructor **`private`**
  - When being passed or returned by value, objects are copied by invocation of the copy constructor

Example of Disabling Object Copy

```
class C {
 public:
   C();

 private:
   C(C&); // declared private
};

// C object is passed by value
void f1(C cObj) {
   // trying to copy actual param.
   //    by calling C(C&)
   ...
}

// Returns a C object by value
C g1() {   ...   }
```

```
// OK: call by reference
//     - no need to call C(C&)
void f2(C& cObj) {   ...   }

// OK: return by reference
C& g2() {   ...   }

int main() {
   C c1, c2;

   f1(c1);     // error: C(C&) is
               //        private

   c2 = g1(); // error: C(C&) is
               //        private
   ...
}
```

Convert Constructors

- One-argument constructor of class ***C*** used to **convert** a non-***C*** type argument to a ***C*** object
  - So far, we have seen some convert constructors already

```
class Person {
public:
   Person() { name = "Unknown"; }
   Person(const string& n);   // convert constructor:
                              //   string → Person object
   Person(const char* n);     // convert constructor:
                              //   char* → Person object
   ...
private:
   string name;
};

void main() {
   Person soprano("Dawn Upshaw");
   ...
}
```

Implicit Type Conversion

- A convert constructor can be used as an alternative way to overload functions with class parameters by supporting ***implicit type conversion***
  - The compiler invokes the **most appropriate convert constructor** with the given type of argument for a function so that the generated class object is available as the function's expected argument

```
class Person {
public:
  Person(const string& n)
  { name = n; }
  ...
};

void f(Person& s) { /* ... */ }

void main() {
  Person p("foo");
  f(p);   // OK: p is a Person obj.

  string b = "bar";
  f(b);   // OK: even though no
}         //  f(string&) defined
```

```
Person _pobj_(b);
  // string b is converted
  //   to a Person object _pobj_
  //   by the compiler
f(_pobj_);
```

Disabling Implicit Type Conversion

- Implicit type conversion is convenient for programmers, but may lead to unforeseen errors
  - Usually, they are subtle and hard to detect
  - Thus, a programmer can disable implicit type conversions anytime by declaring convert constructors with keyword **explicit**

```
class Person {
public:
   explicit Person(const string& n) { name = n; }
   ...
};

void f(Person& s) { /* ... */ }

void main() {
   Person p("foo");
   f(p);

   string b = "bar";
   f(b);   // a compile-time error
           // : no implicit type conversion for string available
}
```

Constructor Initializers

- Data members of a class can be initialized in
  - the body of the constructor using the assignment operator "=", or
  - the initialization section of the constructor using the ***constructor initializers***

```
class C {
 private:
  int x;
  const int c;

 public:
  C() {
     x = -1;
     c = 0; // error: c is const
  }
};
```

```
class C {
 private:
  int x;    // initialized first
  const int c;

 public:
  C() : c(0), x(-1) {
     /* empty */
  }
};
```

❖ Note that
  - Operator "=" can not be applied to **const** members, but the initializer can
  - Initialization occurs in the order in which the members are declared

Constructors and the Operator `new` and `new[]`

- For dynamic allocations of class objects
  - Use of **`new`** and **`new[]`** ensures the invocation of the appropriate constructor
  - Whereas use of **`malloc`** or **`calloc`** does **NOT** (not recommended)

```
class Emp {
public:
    Emp() { /* ... */ }
    Emp(const char* name) { /* ... */ }
    ...
};

int main() {
    Emp* elvis = new Emp();                  // default constructor
    Emp* cher = new Emp("Cher");             // convert constructor
    Emp* lotsOfEmps = new Emp[1000];    // default 1,000 times

    Emp* foo = malloc(sizeof(Emp));          // no constructor invoked
    Emp* foos = calloc(10, sizeof(Emp));     // no constructor invoked
    ...
}
```

Destructors

- A method automatically invoked whenever objects are destroyed
  - For example,
    - when local objects go out of scope, or
    - dynamically allocated objects are deleted
  - Destructor prototype of a class ***C***

```
~C();
```

    - No return type and no parameters
    - Only one destructor per a class

Destructor Example

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

class C {
public:
 C() {
  name = "anonymous";
  cout << name << " constructing.\n";
 }

 C(const char* n) {
  name = n;
  cout << name << " constructing.\n";
 }

 ~C() {   // destructor
  cout << name << " destructing.\n";
 }

private:
  string name;
};
```

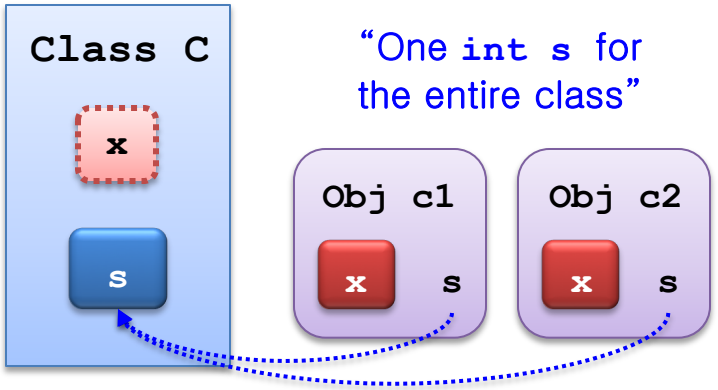
```
int main() {
  C c0("bar");

  {
     C c1;
     C c2("foo");
  }  // c1 and c2 are destroyed

  C* ptr = new C();
  delete ptr;  // ptr is destroyed

  return 0;    // c0 is destroyed
}
```

```
bar constructing.
anonymous constructing.
foo constructing.
foo destructing.
anonymous destructing.
anonymous constructing.
anonymous destructing.
bar destructing.
```

Class Members

- So far, we have seen data members and methods associated with individual objects
  - I.e., when we create two objects, each has its own data members and methods (called *object members* or *instance members*)
- C++ supports members associated with the class itself rather than its objects
  - They are called ***class members*** as opposed to object members
  - The keyword **`static`** is used to create class members

```
class C {
   int x;  // object data member

   static int s;
           // class data member
   ...
};

C c1, c2;
```



Using Class Data Members

- A class (**`static`**) data member declared inside the class must be defined or initialized within the global scope

```
class Task {
 private:
   static unsigned n;  // declares count of Task objects
   ...
 public:
   Task(const string& ID) {
      ...
      n++;     // increases the count whenever a Task is created
   }

   ~Task() {
      ...
      n--;     // decreases the count whenever a Task is destroyed
   }
};

unsigned Task::n = 0;  // need to initialize the object count n
                       //    within the global scope
```

`static` Data Members and the `sizeof` a Class

- **`static`** data members do **NOT** affect the **`sizeof`** a class or an object of this class

```
class C {
  unsigned long dm1; // 8 bytes
  double dm2;        // 8 bytes
};

C c1;
```

`sizeof(C) = sizeof(c1) = 16`

```
class C {
  unsigned long dm1;  // 8 bytes
  double dm2;         // 8 bytes

  static long dm3;    // X
  static double dm4;  // X
};

C c1;
```

`sizeof(C) = sizeof(c1) = 16`

Class Methods

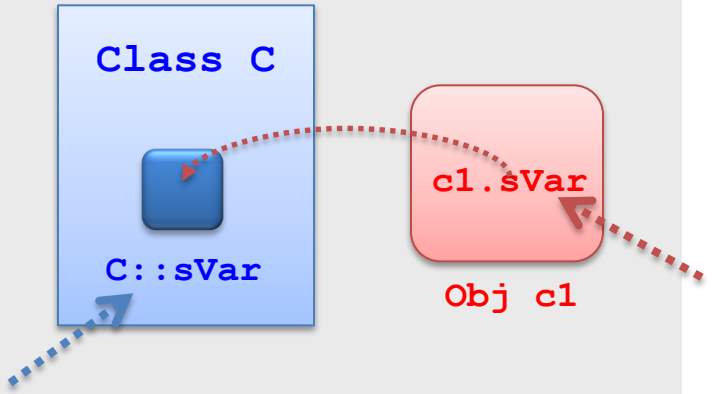
- In addition to class data members, a class may have class methods (i.e., **`static`** methods)
  - **`static`** methods can access **ONLY** other **`static`** members (**`static`** data and **`static`** methods) and **`static`** functions
    - ❖ Cf., non-**`static`** methods and functions can access both **`static`** and non-**`static`** members

```
class Task {
  private:
    static unsigned n;
    time_t st;
    ...
  public:
    void setST() {   ...   }

    static unsigned getN() {
       setST();                    // error: setST() is not static
       st = time(0);               // error: st is not static
       ...
       return n;                   // OK: n is static
    }
    ...
};
```

Accessing `static` Members from Outside a Class

- There are two ways to access the **`static`** members of class **`C`** from outside of **`C`**
  - Through **`C`** objects (using member selector op. ".")
  - Directly through class **`C`** (using scope resolution op. "::")
    - Preferred way to access **`static`** members, because they are directly associated with the class (no need to create an object before access)

```
class C {
 public:
   static int sVar;
   static void sMeth();
};

void main() {
   C c1;

   unsigned x = c1.sVar; // accesses sVar through an object
   unsigned y = C::sVar; // directly (recommended)

   c1.sMeth();    // invokes method sMeth through an object
   C::sMeth();    // directly through the class (recommended)
}
```



`static` Variables Defined inside Methods

- Local variables defined in a method can also be **`static`**
  - In this case, the variable has one underlying storage cell shared by all objects when the method is invoked (**initialized once**)
  - Unlike class members, the **`static`** variable has **block scope**
    - I.e., accessible only inside the method body

```
class C {
  public:
    void m();
  private:
    int x;
};

void C::m() {
  static int s = 0;
       // 1 copy of s for all objects
       //   (initialized once)
  cout << ++s << '\n';
}
```

```
void main() {
   C c1, c2, c3; // 3 objs share s

   c1.m();          // ++s (s == 1)
   c2.m();          // ++s (s == 2)
   c3.m();          // ++s (s == 3)
}
```