Chapters 17-18

Classes: A Deeper Look, Parts I & II

C How to Program, 6/e, 7/e

prepared by SENEM KUMOVA METİN
modified by UFUK ÇELİKKAN and ILKER KORKMAZ

The textbook's contents are also used

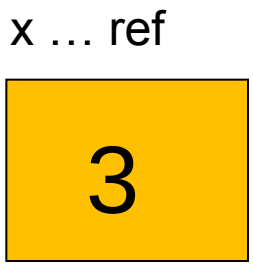
References and Reference Parameters

- Two ways to pass arguments to functions in many programming languages are pass-by-value and pass-by-reference.
- When an argument is passed by value, a copy of the argument's value is made and passed (on the function call stack) to the called function. Changes to the copy do not affect the original variable's value in the caller.
- With pass-by-reference, the caller gives the called function the ability to access the caller's data directly, and to modify that data if the called function chooses to do so.

What is a Reference

- Reference is signaled by **&** and provides an alternative name for storage

```
int x;
int & ref = x;
x = 3;
ref = 3;
```

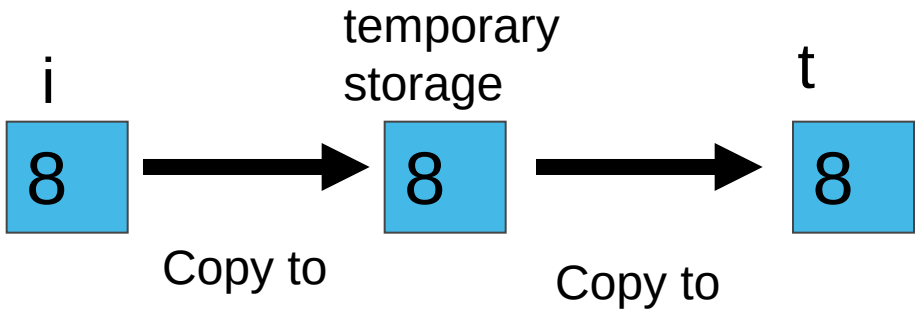


Call by Reference : SWAP example

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

void swap (int &, int &);

void main(){
  int i=7 ; int j=-3;
  swap (i,j);
  cout << i <<endl;
  cout << j << endl;
}

void swap (int & a, int & b) {
  int t = a;
  a = b;
  b = t;
}
```

RETURN BY VALUE

```
int val1() {
  int i=8;
      …
  return i;
}
main(){
  int t;
  t = val1();
      …
}
```



Returning References

- Returning references from functions can be dangerous.
- When returning a reference to a variable declared in the called function, the variable should be declared `static` within that function.
- Otherwise, the reference refers to an automatic variable that is discarded when the function terminates; such a variable is "undefined," and the program's behavior is unpredictable.
- References to undefined variables are called dangling references.

RETURN BY REFERENCE

```
int & val1() {
  int i=8;
  return i;
}
main(){
  int t;
  t = val1();
      …
}
```

NO TEMPORARY STORAGE
IS USED !!!
t IS NOW i

DANGER: **`val1`** returns a reference to a value that is going to go out of scope when the function returns. The caller receives a reference to garbage. Fortunately, your compiler will give you an error if you try to do this.

```
D:\home\celikk...        In function 'int& val1()':
D:\home\celikk... 13     warning: reference to local variable 'i' returned
```

References : Aliases

- References can also be used as aliases for other variables. Once a reference is declared as an alias for a variable, all operations "performed" on the alias (i.e., the reference) are actually performed on the original variable.
- The alias is simply another name for the original variable.
- For example, the code

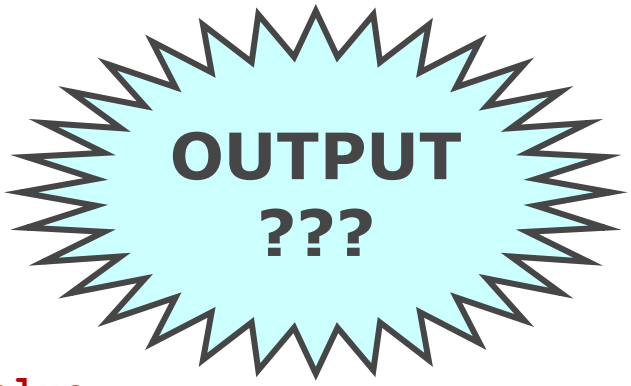
```
int count = 1; // declare integer variable count
int &cRef = count; // create cRef as an alias for count
cRef++; // increment count (using its alias cRef)
```

increments variable `count` by using its alias `cRef`.

Passing and Returning Objects by Value

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

Person  func1() {
  Person p;
  p.setAge(4);
  return p;  // Returning object
}

unsigned func2( Person y) { // Call by value
  y.setAge(3);
  return y.getAge();
 }
```

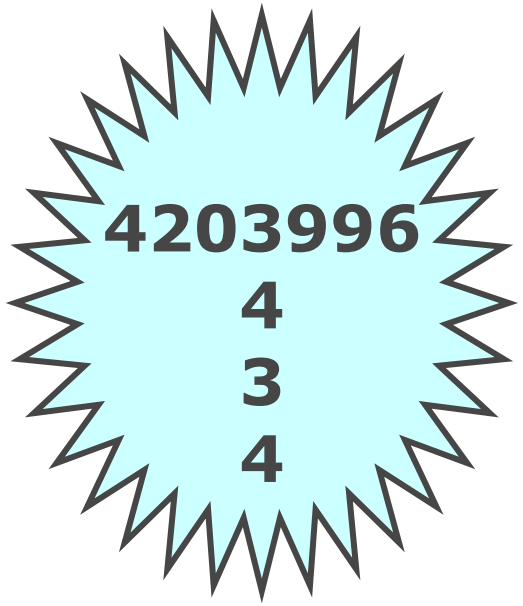
```
main() {
  Person x;
  cout << x.getAge() << endl;
  x = func1();
  // cout << func1().getAge(); ?
  cout << x.getAge() << endl;
  cout << func2(x) << endl;
  cout << x.getAge() << endl;
}
```



Passing and Returning Objects by Value

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

Person  func1() {
  Person p;
  p.setAge(4);
  return p;  // Returning object
}

unsigned func2( Person y) { // Call by value
  y.setAge(3);
  return y.getAge();
 }
```

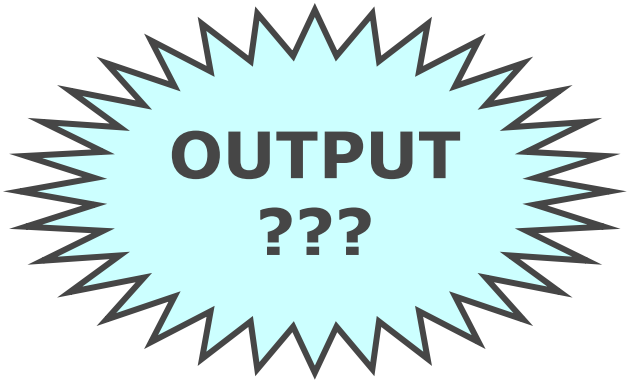
```
main() {
  Person x;
  cout << x.getAge() << endl;
  x = func1();
  // cout << func1().getAge(); ?
  cout << x.getAge() << endl;
  cout << func2(x) << endl;
  cout << x.getAge() << endl;
}
```



Passing and Returning Objects by Reference

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

Person  & func3(){
  Person p;
  p.setAge(4);
  return p;
}

unsigned func4( Person & y){ // Call by reference
  y.setAge(3);
  return y.getAge();
}
```

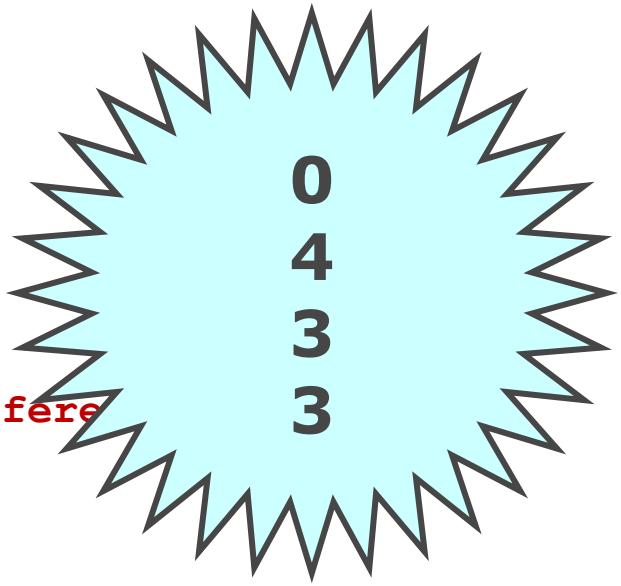
```
main() {
  Person x;
  cout << x.getAge() << endl;
  x = func3();
  cout << x.getAge() << endl;
  cout << func4(x) << endl;
  cout << x.getAge() << endl;
}
```



Passing and Returning Objects by Reference

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

Person  & func3(){
  Person p;
  p.setAge(4);
  return p;
}

unsigned func4( Person & y){ // Call by refere
  y.setAge(3);
  return y.getAge();
}
```

```
main() {
  Person x;
  cout << x.getAge() << endl;
  x = func3();
  cout << x.getAge() << endl;
  cout << func4(x) << endl;
  cout << x.getAge() << endl;
}
```

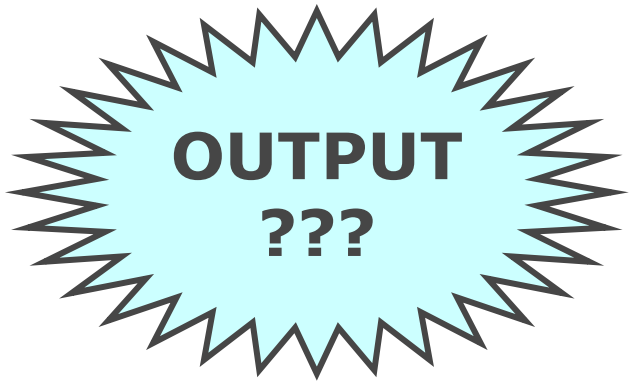


Pointer to Objects

Accessing to an object's members through a pointer requires class indirection operator "->"

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

void func(Person * ptr){
  ptr->setAge(5);
  cout << ptr->getAge() << endl;
}
```

```
void main() {
  Person x;
  x.setAge(4);
  cout << x.getAge()<< endl;

  func(&x);
  cout << x.getAge() << endl;
}
```



const methods

`const` objects and `const` member functions
prevent modifications of objects and enforce the principle of least privilege.

```
class Time {
public:
       void setTime( int h, int m) {hour = h; minute = m;}; // OK.
       void printUniversal()  const {hour = 12; }; //??????
       void printStandard() const;

private:
       int hour;
       int minute;

};
```

The keyword `const` in methods printUniversal and printStandard shows that unlike method SetTime, these methods can not change the value of any `Time` data member.

const keyword in input and output parameter

```
class Time {
public:
   void setTime( const int &m, const int &h ) { minute=m; hour=h;};
   const int & getHour() { return hour;};
private:
   int hour;
   int minute;
};

main(){
   int a = 16, b = 15;
   const int &x = 0;    // Intialization is OK !
   Time obj;

   // setTime() cannot change  the value of a or b of main()
   obj.setTime(a, b) ;
   // main() cannot change the return value of getHour()
   x = obj.getHour();
}
```

```
D:\home\celikk...        In function 'int main()':
D:\home\celikk... 157    error: assignment of read-only reference 'x'
```

Initializing Objects with Constructors

- A special method that initializes class members.
- Same name as the class.
- No return type (not even void).
- Member variables can be initialized by the constructor or set afterwards
- Normally, constructors are declared `public`.

Initializing Objects with Constructors (cont.)

- C++ requires a constructor call for each object that is created, which helps ensure that each object is initialized before it's used in a program.
- The constructor call occurs implicitly when the object is created.
- If a class does not explicitly include a constructor, the compiler provides a default constructor—that is, a constructor with no parameters .

Initializing Objects with Constructors (cont.)

- A class gets a default constructor in one of two ways:
  - The compiler implicitly creates a default constructor in a class that does not define a constructor.
  - You explicitly define a constructor that takes no arguments.
- If you define a constructor with arguments, C++ will not implicitly create a default constructor for that class.

Constructors : inline definition

```
class Person {
  public :
       // Constructor
       Person() { age = 0; name =“Unknown”; }

       void setAge (unsigned n) { age = n };
       unsigned getAge() const { return age };
       void getName() const { cout <<name<<endl; }

  private:
       unsigned age;
       string name;
  };

void main()
{
  Person p;
  cout << p.getAge()<<endl;
  cout << p.getName()<<endl;
}
```

Constructors

```
class Person {
  public :
       Person();       // Constructor
       void setAge (unsigned n) { age =n };
       unsigned getAge() const { return age };
       void getName() const { cout << name << endl; }
  private:
       unsigned age;
       string name;
};

Person ::Person() {
  age = 0;
  name = "Unknown";
}

void main()
{
  Person p;
  cout <<p.getAge()<<endl;
  cout <<p.getName()<<endl;
}
```

Constructors : Overloading

```
class Person {
public :

   Person() { age = 0; name = "Unknown"; }    // First Constructor
   Person(string n) { name = n; }     // Second Constructor

   void setAge (unsigned n) { age = n };
   unsigned getAge() const { return age };
   void getName() const { cout <<name<<endl; }

private:
   unsigned age;
   string name;
};


void main()
{
   Person q;              // USING FIRST CONSTRUCTOR
   cout << q.getAge() << endl;
   cout << q.getName()<< endl

   Person p("John"); // USING SECOND CONSTRUCTOR
   cout << p.getAge() << endl;
   cout << p.getName()<< endl;
}
```

Destructors

- A constructor is automatically invoked whenever an object is created
- A destructor is automatically invoked whenever an object is destroyed
- A destructor takes no arguments and no return type, there can be only one destructor per class

```
class C
{
  public :
    C()    { … };        // constructor
    ~C()   { … };        // destructor
  …
}
```

Destructor: Example

```
class C {
public :
  C(){name="anonymous";}
  C(const char * n) { name=n;}
  ~C() { cout <<"destructing" <<name<<"\n";}
private:
  string name;
};

int main(){
  C c0("John");
  {   // entering scope....
      C c1;
      C c2;
  } // exiting scope. destructors for c1 and c2 are called

  C * ptr = new C();
  delete ptr; // destructor for ptr object is called

  return 0; // destructor for c0 is called
}
```

CONSTRUCTORS

```
class Person {
public :
   Person() {name =“Unknown”; };
   Person(cons string & n) ;
   Person(cons char * n);
   void getName() const { cout << name; }
private:
   string name;
};

Person :: Person(const string & n) {
   name = n;
   cout <<“Creating objects with string!!\n”;
}

Person :: Person(const char * n) {
   name = n;
   cout <<“Creating objects with char *!!\n”;
}

void main() {
   Person p;
   p.getName();
   Person p1(“NOT IMPORTANT”);  // char *   string s1(“VERY IMPORTANT”)
                                            Person p2(s1);
}
```

CONSTRUCTORS : Restricting Object Creation

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

void main() {
   Emp orhan;                    //IS IT POSSIBLE??
   Emp ferdi(111222333);         //IS IT POSSIBLE??
}
```

If a class explicitly declares any constructor, the compiler does not provide a public default constructor.

If a class declares a Non-public default constructor, compiler does not provide a public default constructor.

CONSTRUCTORS : Restricting Object Creation

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

void main() {
   Emp orhan;                    //IS IT POSSIBLE?? NO
   Emp ferdi(111222333);         //IS IT POSSIBLE?? YES
}
```

```
D:\home\celikk...          In function 'int main()':
D:\home\celikk... 68       error: 'Emp::Emp()' is private
D:\home\celikk... 169      error: within this context
```

Default Memberwise Assignment

- The assignment operator (=) can be used to assign an object to another object of the same type.
- By default, such assignment is performed by memberwise assignment
  - Each data member of the object on the right of the assignment operator is assigned individually to the same data member in the object on the left of the assignment operator.
- [*Caution:* Memberwise assignment can cause serious problems when used with a class whose data members contain pointers to dynamically allocated memory]

Default Memberwise Assignment (cont.)

- Objects may be passed as function arguments and may be returned from functions.
- Such passing and returning is performed using pass-by-value by default—a copy of the object is passed or returned.
  - C++ creates a new object and uses a copy constructor to copy the original object's values into the new object.
- For each class, the compiler provides a default copy constructor that copies each member of the original object into the corresponding member of the new object.
  - Copy constructors can cause serious problems when used with a class whose data members contain pointers to dynamically allocated memory.

CONSTRUCTORS : Copy Constructor

- A copy constructor creates a new object as a copy to another object
- Copy constructor for Person class → **`Person(Person &);`**

```
class Person {
public :
     Person(const string & , const unsigned ) ;
     string name;
     unsigned age;
};
Person :: Person(const string & n, const unsigned a) {
     cout << " Calling constructor with "<< n << "," << a << endl;
     name=n;
     age=a;
}
main(){
     string s1("Bob");
     Person p1(s1,15);
     Person p2(p1); // which constructor works?????
     cout << p2.name;
}
```

CONSTRUCTORS : Copy Constructor

Copy constructor may have more than one parameter but all parameters beyond the first must have default values.

```
class Person {
public :
    Person(const string & , const unsigned ) { name=n; age=a;}
    Person(Person & x, const unsigned );
    string name;
    unsigned age;
};

Person :: Person(Person & x, const unsigned y = 0 ) {
    cout << "Calling Copy Constructor" << y ;
    name=x.name; age=y;
}
main(){
    string s1("Bob");
    Person p1(s1,15);
    Person p2(p1); // which constructor works?????
}
```

Constructor: Convert

Used to convert a non-C type such as an int or string to a C object

```
class Person {
  public :
      Person() {name =“Unknown”; }
      Person(const string & n)   { name = n;}
      Person(const char * n)     { name = n; } // called
private:
      string name;
};

void main() {
  Person p1(“John”);
      // converts a string constant to a Person object
}
```

CONSTRUCTORS : Disabling Passing and Returning by Value for Class Objects

```
class C {
  public :    C();
  private:    C(C &); // Copy Constructor
};

void f(C);    // call by value
C g() {
        C obj;
        return obj; // return by value
}

main() {
        C c1, c2;
        f(c1);          // NOT POSSIBLE
        c2=g();         // NOT POSSIBLE
}
```

If the copy constructor is private top-level functions and methods in other classes cannot pass or return class objects by value because this requires a call to copy constructor!!!

CONSTRUCTOR INITIALIZERS

```
class
C {
public :
     C() { x=0; y=0;} // cannot change the value of const y
private:
     int x;
     const int y;     // const data member
};

// WE HAVE TO USE CONSTRUCTOR INITIALIZER to INITIALIZE A
// const DATA MEMBER!!!

class C {
public :
     C() : y(0) { x=0; } // or C() : y(0), x(0){ }

private:
       int x;
       const int y; // const data member
  };
```

CONSTRUCTORS : new and new [] operators

```
class Emp {
public :
        Emp() {cout << "calling default constructor";};
        Emp(const char * name) {n = name;}
        string n;
};
int main()
{
  int *x = new int;
                                  // default constructor initializes
  Emp *orhan = new Emp();
                                  // second constructor initializes
  Emp *ferdi = new Emp("Ferdi");
                                  // default constructor initializes
  Emp *lots_of_people = new Emp[1000];
                                  // NO constructor initializes
  Emp *foo = (Employee *) malloc(sizeof(Emp));
}
```

The order of the *ctor and dtor calls*

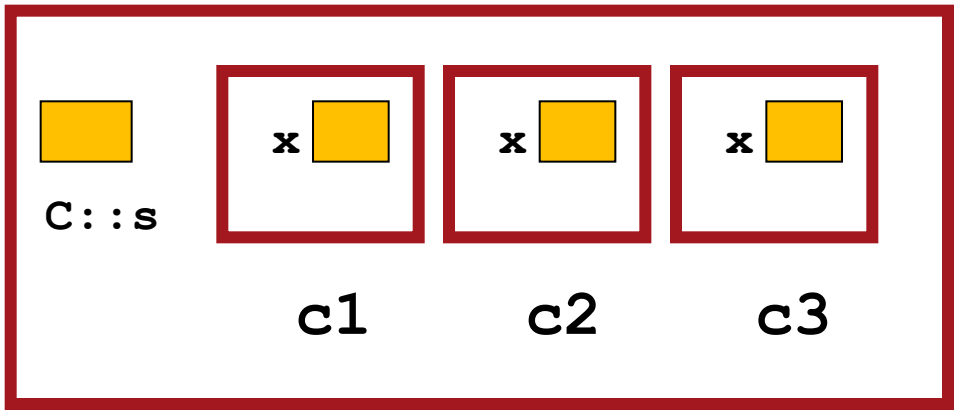
- Destructor calls are made in the reverse order of constructor calls for the corresponding objects

- Scope or storage class differences of the objects can alter the order

static Class Members

- In certain cases, only one copy of a variable should be shared by all objects of a class.
- A static data member is used for these and other reasons. `static` keyword is used.
- Such a variable represents "class-wide" information.

```
class C {
  int x;
  static int s;
};

C c1, c2, c3;
```



Static members : Data members

- A static member does not effect the size of a class or an object of this class type.
- A static data member may be declared inside a class declaration but must be defined outside.
- A class's `static` members exist even when no objects of that class exist.

```
class Task {
public:
    …
private:
  static unsigned n; // declaration
  static const int id = 0;  // const is required if
                                initialized
};
unsigned Task::n=0; // definition
```

`static` Class Members (cont.)

- Although they may seem like global variables, a class's `static` data members have class scope.
- `static` members can be declared `public`, `private` or `protected`.
- A fundamental-type `static` data member is initialized by default to `0`.
- If you want a different initial value, a `static` data member can be initialized *once*.
- A `static const` data member of `int` or `enum` type can be initialized in its declaration in the class definition.
- All other `static` data members must be defined *at global namespace scope and can be initialized only in those definitions.*
- If a `static` data member is an object of a class that provides a default constructor, the `static` data member need not be initialized because its default constructor will be called.

Static members : Methods

A static method can access only other static members

```
class Task{
public:
  static unsigned getN() {return n;}
  static int getK() const {return k;} //NOT POSSIBLE!!!
private:
      static unsigned n;
      int k;
};
```

Public static members : Accessing

To access a `public static` class member when no objects of the class exist, prefix the class name and the binary scope resolution operator (`::`) to the name of the data member.

Public static data members : Accessing

```
class Task {
public:
      static unsigned getN() const {return n;}
      static unsigned n;
};
unsigned Task::n=5;

int main() {
  Task c1, c2;

  c1.getN();         // access through an object. c1 exists.
  Task::getN();      // access through class (direct access)

  unsigned x = c1.n; // access through an object
  unsigned y = c2.n; // access through an object

  unsigned z= Task::n; // access through class(direct access)
}
```

Private static data members : Accessing

To access a **private** or **protected** static class member when no objects of the class exist, provide a public static member function and call the function by prefixing its name with the class name and binary scope resolution operator.

```
class Task {
public:
      static unsigned getN() const {return n;}
private:
      static unsigned n;
};
unsigned Task::n=5;

int main() {
  unsigned z= Task::n; // ERROR !!! n is private
                       // access through class (direct access
  unsigned z = Task::getN();
}
```

Static vars defined inside methods

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

void C::m() {
  static int s = 0; // one copy for all objects!!
  cout << ++s << '\n'
}

int main(){
  C c1,c2,c3;

  c1.m(); // 1
  c2.m(); // 2
  c3.m(); // 3
  return 0;
}
```

sizeof an object

- People new to object-oriented programming often suppose that objects must be quite large because they contain data members and member functions.
- Logically, this is true—you may think of objects as containing data and functions (and our discussion has certainly encouraged this view); physically, however, this is not true.

sizeof an object

Performance Tip 17.2

Objects contain only data, so objects are much smaller than if they also contained member functions. Applying operator `sizeof` *to a class name or to an object of that class will report only the size of the class's data members. The compiler creates one copy (only) of the member functions separate from all objects of the class. All objects of the class share this one copy. Each object, of course, needs its own copy of the class's data, because the data can vary among the objects. The function code is nonmodifiable and, hence, can be shared among all objects of one class.*

The Pointer Constant *this*

`this` pointer

- an implicit argument to each of a class's non-`static` member functions.
- allows those member functions to access the correct object's data members and other non-`static` member functions.

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

IS SAME WITH

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

The Pointer Constant *this*

```
class Person {
public :
    Person( string & name) {
            this->name = name; // name = name does NOT
  work
            name="Mary";        // name is input parameter
    }
    string getName(){ return name;}
  private:
    string name;              // = this->name
};
void main() {
    string n("Joe");
    Person p(n);
    cout << n << "  " << p.getName();
}
```

Utility functions

- Member functions with *private* access specifier

Composition

- The use of *"has-a relationship"*
- Classes can have objects of other classes as data members

```
class DifferentClass{
   // ...
};
class ExampleClass{
   DifferentClass exampleDataMember;
};
```

`friend` Functions and `friend` Classes

- A friend function of a class is defined outside that class's scope, yet has the right to access the non-`public` (and `public`) members of the class.
- Standalone functions, entire classes or member functions of other classes may be declared to be friends of another class.
- Using `friend` functions can enhance performance.
- Friendship is granted, not taken.
- The friendship relation is neither symmetric nor transitive.

```
1  // Fig. 18.15: fig18_15.cpp
2  // Friends can access private members of a class.
3  #include <iostream>
4  using namespace std;
5
6  // Count class definition
7  class Count
8  {
9     friend void setX( Count &, int ); // friend declaration
10 public:
11    // constructor
12    Count()
13       : x( 0 ) // initialize x to 0
14    {
15       // empty body
16    } // end constructor Count
17
18    // output x
19    void print() const
20    {
21       cout << x << endl;
22    } // end function print
```

Fig. 18.15 | Friends can access `private` members of a class. (Part 1 of 3.)

```
23  private:
24     int x; // data member
25  }; // end class Count
26
27  // function setX can modify private data of Count
28  // because setX is declared as a friend of Count (line 9)
29  void setX( Count &c, int val )
30  {
31     c.x = val; // allowed because setX is a friend of Count
32  } // end function setX
33
34  int main()
35  {
36     Count counter; // create Count object
37
38     cout << "counter.x after instantiation: ";
39     counter.print();
40
41     setX( counter, 8 ); // set x using a friend function
42     cout << "counter.x after call to setX friend function: ";
43     counter.print();
44  } // end main
```

Fig. 18.15 | Friends can access `private` members of a class. (Part 2 of 3.)

```
counter.x after instantiation: 0
counter.x after call to setX friend function: 8
```

Fig. 18.15 | `Friend`s can access `private` members of a class. (Part 3 of 3.)

HW & LAB

- HWs are not assessed. Nevertheless, LAB works may be similar to HWs. Therefore, you are advised to do HWs.

- LABs are not graded. However, midterm questions may be similar to LABs. So, you are advised to attend in LAB classes.

TO DO @ HOME:

- Dissect the example codes provided in Chapters 17 and 18.