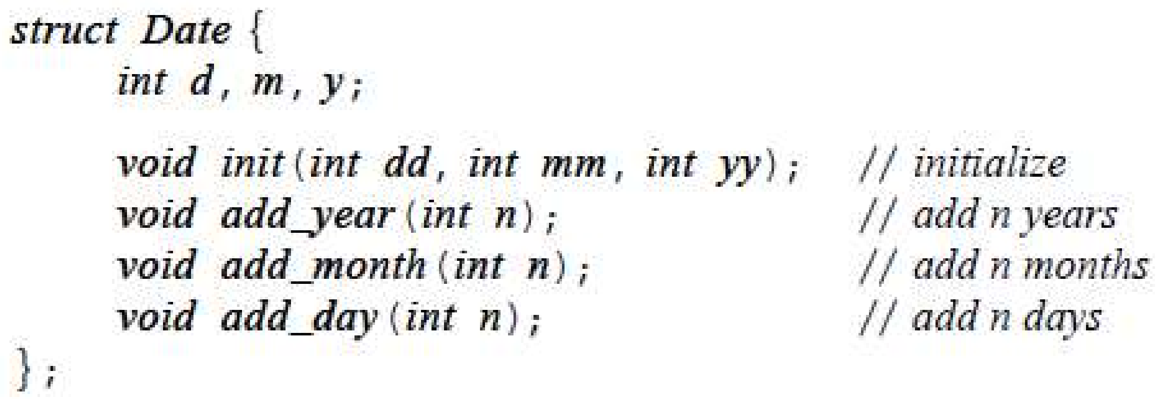
Classes

Dr. Hacer Yalım Keleş

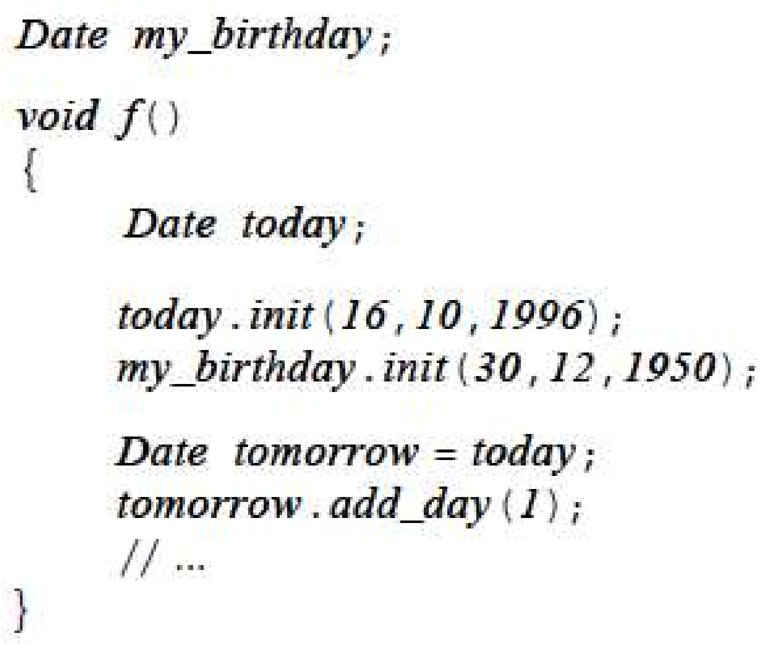
Reference Book: “The C++ Programming Language”, Bjarne Stroustrup.

Structure



```
struct Date {
      int d, m, y;

      void init(int dd, int mm, int yy);    // initialize
      void add_year(int n);                 // add n years
      void add_month(int n);                // add n months
      void add_day(int n);                  // add n days
};
```

Reference Book: “The C++ Programming Language”, Bjarne Stroustrup.



```
Date my_birthday;

void f()
{
        Date today;

        today.init(16, 10, 1996);
        my_birthday.init(30, 12, 1950);

        Date tomorrow = today;
        tomorrow.add_day(1);
        // ...
}
```

Reference Book: “The C++ Programming Language”, Bjarne Stroustrup.

```
void Date::init(int dd, int mm, int yy)
{
        d = dd;
        m = mm;
        y = yy;
}
```

when *Date::init()* is invoked for *today*, *m=mm* assigns to *today.m*.
when *Date::init()* is invoked for *my_birthday*, *m=mm* assigns to *my_birthday.m*.

Classes

```
class Date {
    int d, m, y;
public:
    void init(int dd, int mm, int yy);     // initialize

    void add_year(int n);                  // add n years
    void add_month(int n);                 // add n months
    void add_day(int n);                   // add n days
};
```

The ***public*** label separates the class body into two parts. The names in the first, *private*, part can be used only by member functions. The second, *public*, part constitutes the public interface to objects of the class. A ***struct*** is simply a ***class*** whose members are public by default functions can be defined and used exactly as before.

Reference Book: “The C++ Programming Language”, Bjarne Stroustrup.

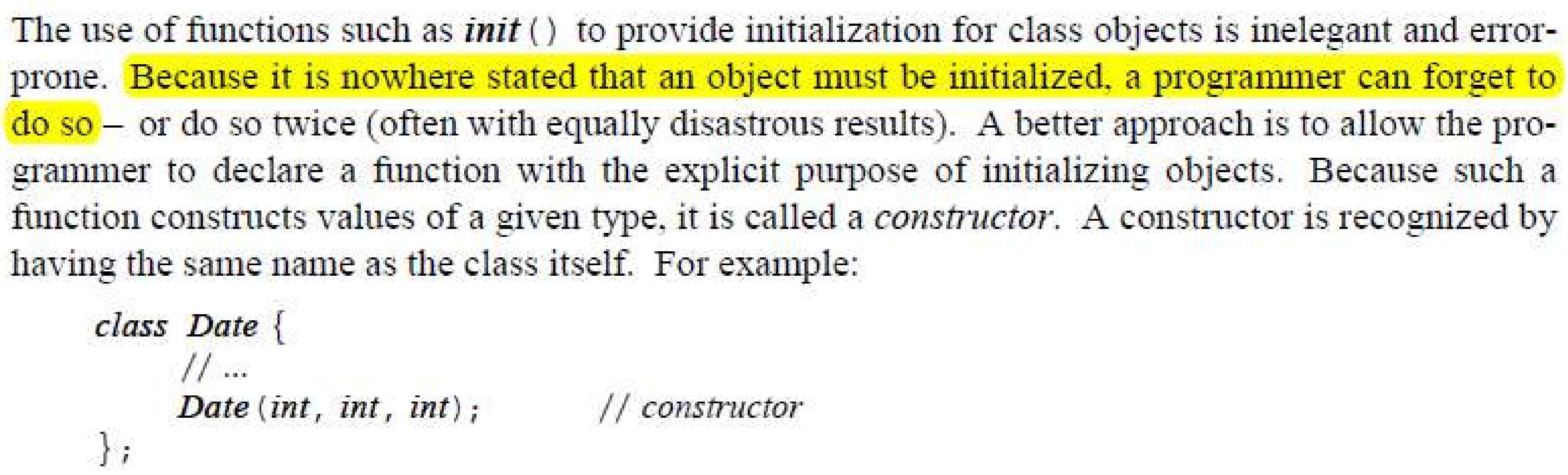
```
class Date {
      int d, m, y;
public:
      void init(int dd, int mm, int yy);     // initialize

      void add_year(int n);                  // add n years
      void add_month(int n);                 // add n months
      void add_day(int n);                   // add n days
};
```

```
inline void Date::add_year(int n)
{
      y += n;
}
```

```
void timewarp(Date& d)
{
      d.y -= 200;      // error: Date::y is private
}
```

Reference Book: “The C++ Programming Language”, Bjarne Stroustrup.

Constructors



The use of functions such as *init*() to provide initialization for class objects is inelegant and error-prone. Because it is nowhere stated that an object must be initialized, a programmer can forget to do so – or do so twice (often with equally disastrous results). A better approach is to allow the programmer to declare a function with the explicit purpose of initializing objects. Because such a function constructs values of a given type, it is called a *constructor*. A constructor is recognized by having the same name as the class itself. For example:

```
class Date {
        // ...
        Date(int, int, int);          // constructor
};
```

Reference Book: “The C++ Programming Language”, Bjarne Stroustrup.

It is often nice to provide several ways of initializing a class object. This can be done by providing several constructors. For example:

```
class Date {
      int d, m, y;
public:
      // ...
      Date(int, int, int);          // day, month, year
      Date(int, int);               // day, month, today's year
      Date(int);                    // day, today's month and year
      Date();                       // default Date: today
      Date(const char*);            // date in string representation
};
```

Reference Book: “The C++ Programming Language”, Bjarne Stroustrup.

Static members

A variable that is part of a class, yet is not part of an object of that class, is called a ***static*** member. There is exactly one copy of a ***static*** member instead of one copy per object, as for ordinary non-***static*** members. Similarly, a function that needs access to members of a class, yet doesn't need to be invoked for a particular object, is called a ***static*** member function.

```
class Date {
      int d, m, y;
      static Date default_date;
public:
      Date(int dd =0, int mm =0, int yy =0);
      // ...
      static void set_default(int, int, int);
};
```

```
Date::Date(int dd, int mm, int yy)
{
      d = dd ? dd : default_date.d;
      m = mm ? mm : default_date.m;
      y = yy ? yy : default_date.y;

      // check that the Date is valid
}
```

Reference Book: "The C++ Programming Language", Bjarne Stroustrup.

We can change the default date when appropriate. A static member can be referred to like any other member. In addition, a static member can be referred to without mentioning an object. Instead, its name is qualified by the name of its class. For example:

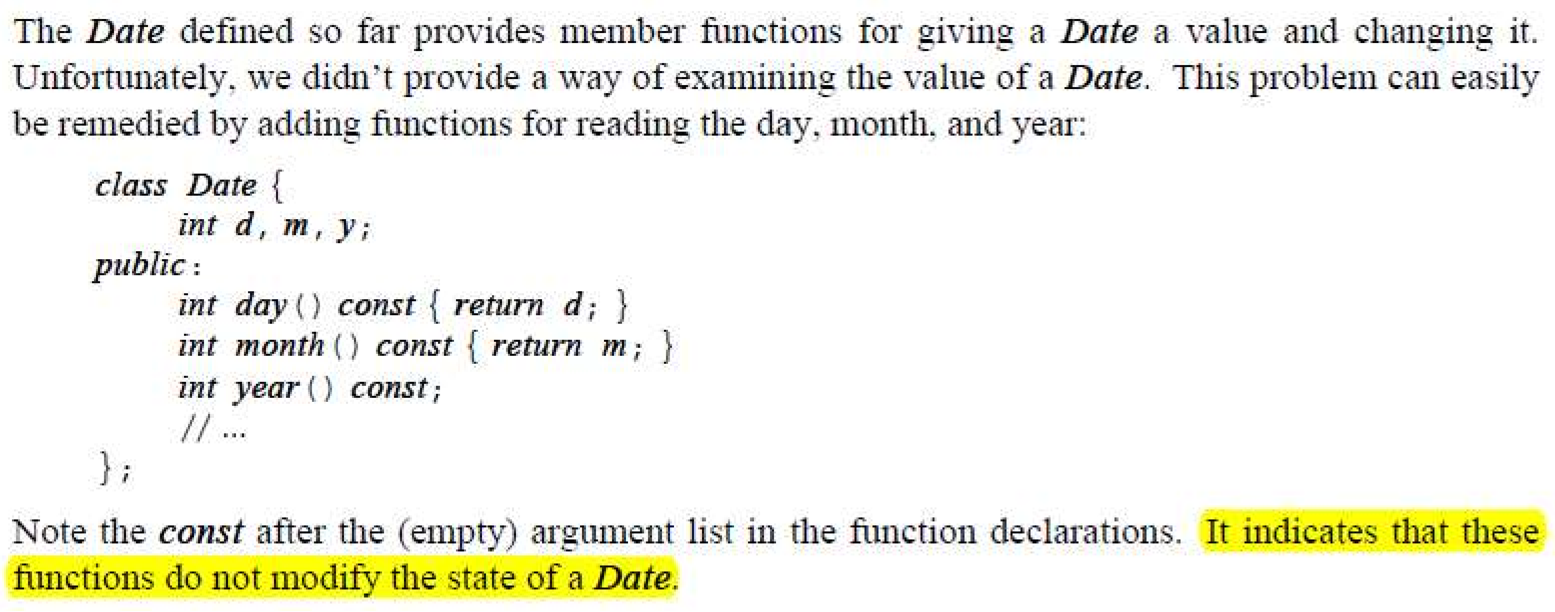
```
void f()
{
      Date::set_default(4,5,1945);
}
```

Static members – both function and data members – must be defined somewhere. For example:

```
Date Date::default_date(16,12,1770);

void Date::set_default(int d, int m, int y)
{
        Date::default_date = Date(d,m,y);
}
```

Now the default value is Beethoven's birth date – until someone decides otherwise.

Constant member functions



The *Date* defined so far provides member functions for giving a *Date* a value and changing it. Unfortunately, we didn't provide a way of examining the value of a *Date*. This problem can easily be remedied by adding functions for reading the day, month, and year:

```
class Date {
      int d, m, y;
public:
      int day() const { return d; }
      int month() const { return m; }
      int year() const;
      // ...
};
```

Note the *const* after the (empty) argument list in the function declarations. It indicates that these functions do not modify the state of a *Date*.

Reference Book: "The C++ Programming Language", Bjarne Stroustrup.

Naturally, the compiler will catch accidental attempts to violate this promise. For example:

```
inline int Date::year() const
{
     return y++;     // error: attempt to change member value in const function
}
```

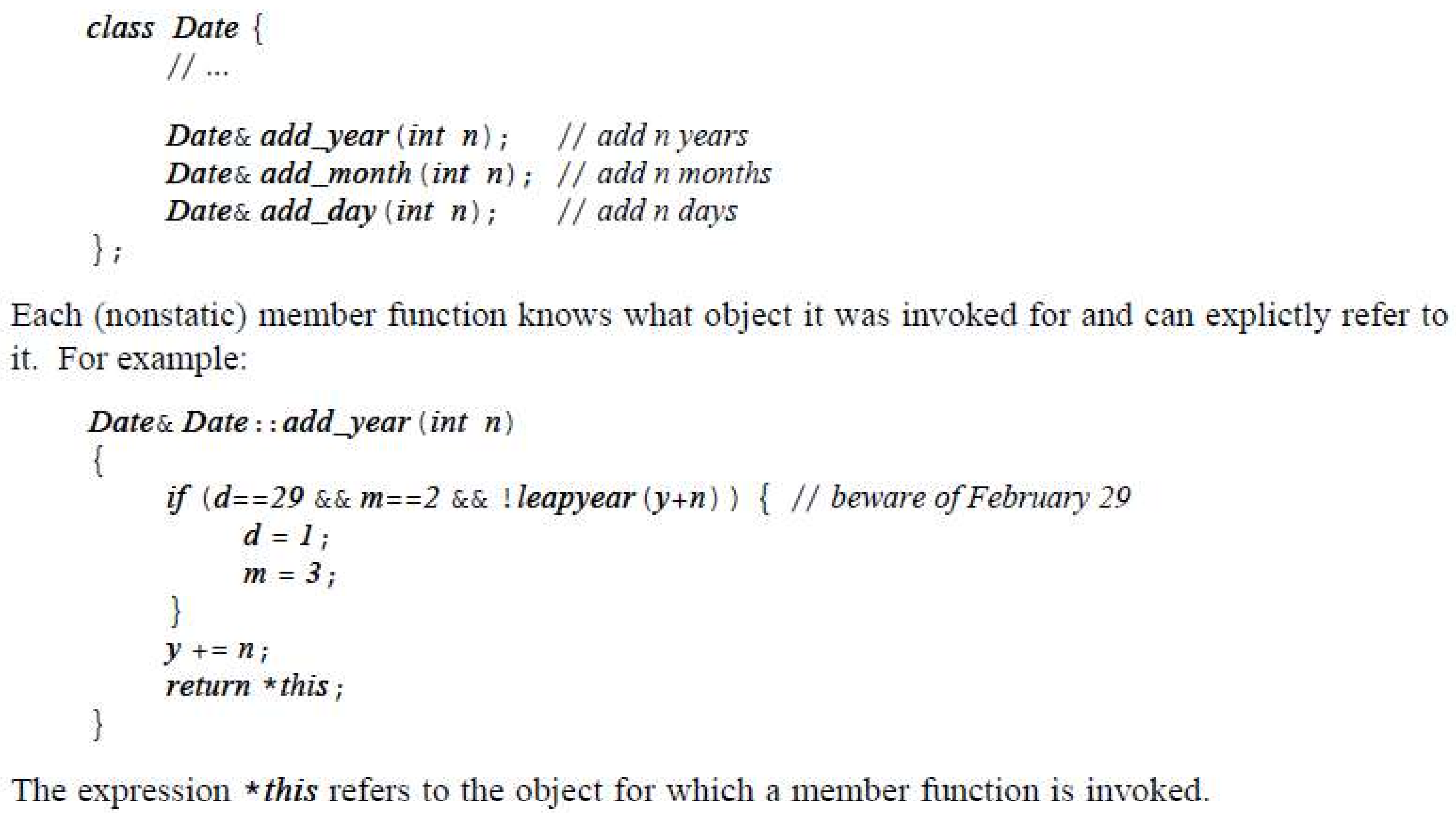
When a ***const*** member function is defined outside its class, the ***const*** suffix is required:

```
inline int Date::year() const          // correct
{
     return y;
}

inline int Date::year()     // error: const missing in member function type
{
     return y;
}
```

A ***const*** member function can be invoked for both ***const*** and non-***const*** objects, whereas a non-***const*** member function can be invoked only for non-***const*** objects. For example:

```
void f(Date& d, const Date& cd)
{
        int i = d.year();         // ok
        d.add_year(1);            // ok

        int j = cd.year();        // ok
        cd.add_year(1);           // error: cannot change value of const cd
}
```

Self Reference



```
class Date {
        // ...

        Date& add_year(int n);      // add n years
        Date& add_month(int n);     // add n months
        Date& add_day(int n);       // add n days
};
```

Each (nonstatic) member function knows what object it was invoked for and can explictly refer to it. For example:

```
Date& Date::add_year(int n)
{
        if (d==29 && m==2 && !leapyear(y+n)) {  // beware of February 29
                d = 1;
                m = 3;
        }
        y += n;
        return *this;
}
```

The expression **this* refers to the object for which a member function is invoked.

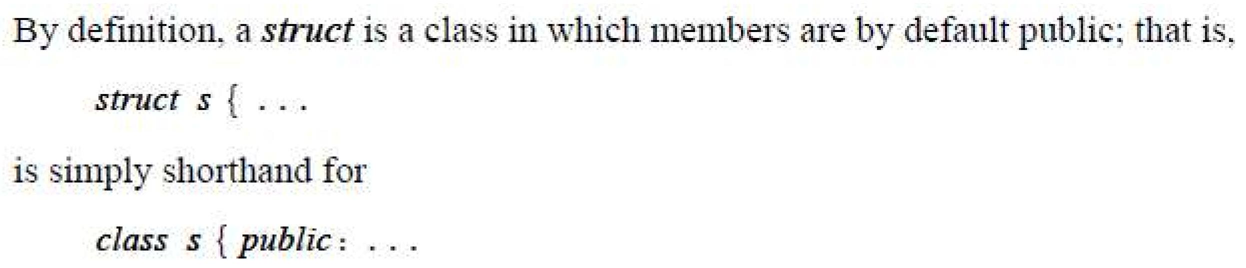
Reference Book: “The C++ Programming Language”, Bjarne Stroustrup.

Most uses of ***this*** are implicit. In particular, every reference to a nonstatic member from within a class relies on an implicit use of ***this*** to get the member of the appropriate object. For example, the ***add_year*** function could equivalently, but tediously, have been defined like this:

```
Date& Date::add_year(int n)
{
        if (this->d==29 && this->m==2 && !leapyear(this->y+n)) {
                this->d = 1;
                this->m = 3;
        }
        this->y += n;
        return *this;
}
```

Reference Book: “The C++ Programming Language”, Bjarne Stroustrup.

Structures and Classes



By definition, a ***struct*** is a class in which members are by default public; that is,

struct s { . . .

is simply shorthand for

class s { *public* : . . .

Reference Book: “The C++ Programming Language”, Bjarne Stroustrup.

The access specifier ***private*:** can be used to say that the members following are private, just as ***public*:** says that the members following are public. Except for the different names, the following declarations are equivalent:

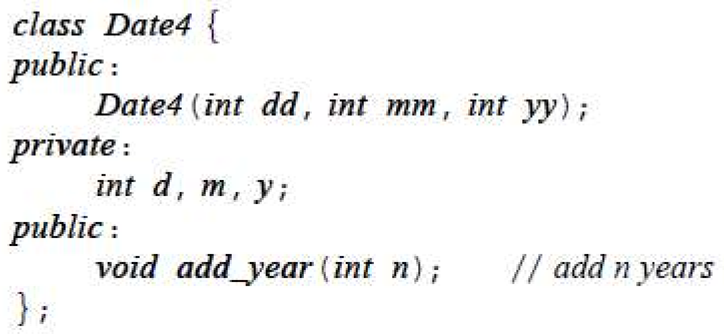
```
class Date1 {
        int d, m, y;
public:
        Date1(int dd, int mm, int yy);

        void add_year(int n);        // add n years
};

struct Date2 {
private:
        int d, m, y;
public:
        Date2(int dd, int mm, int yy);

        void add_year(int n);        // add n years
};
```

Which style you use depends on circumstances and taste. I usually prefer to use ***struct*** for classes that have all data public. I think of such classes as ''not quite proper types, just data structures.''

Reference Book: "The C++ Programming Language", Bjarne Stroustrup.

It is not a requirement to declare data first in a class. In fact, it often makes sense to place data members last to emphasize the functions providing the public user interface. For example:

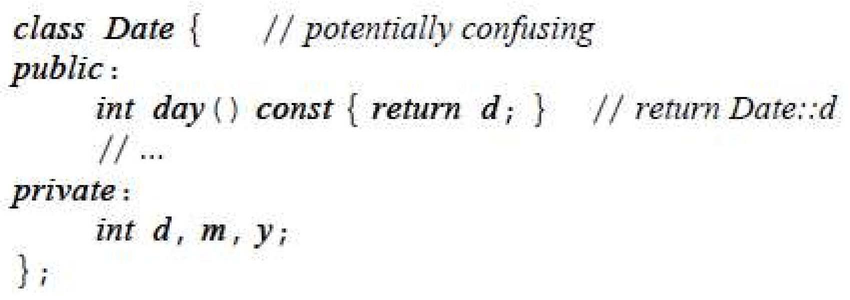
```
class Date3 {
public:
    Date3(int dd, int mm, int yy);
    void add_year(int n);    // add n years
private:
    int d, m, y;
};
```



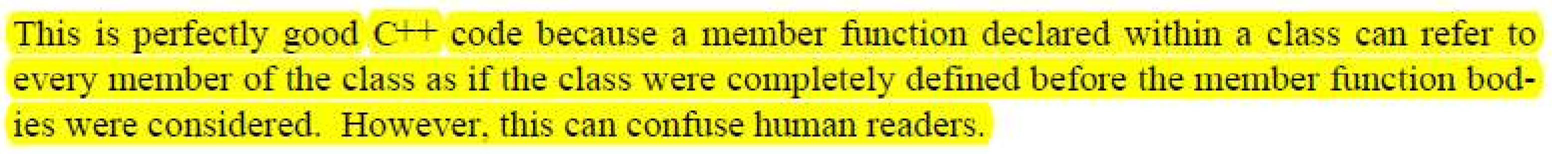
```
class Date4 {
public:
      Date4(int dd, int mm, int yy);
private:
      int d, m, y;
public:
      void add_year(int n);       // add n years
};
```

Reference Book: “The C++ Programming Language”, Bjarne Stroustrup.

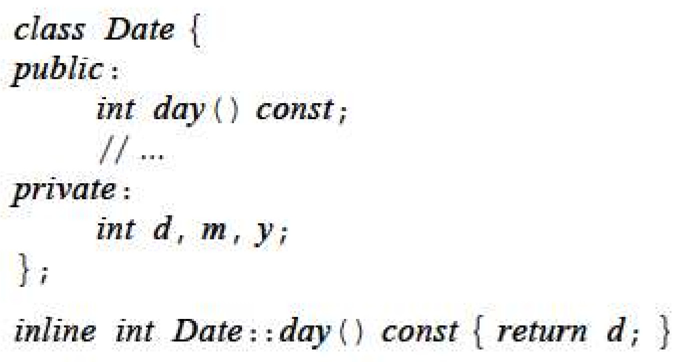
In-Class function definitions



```
class Date {      // potentially confusing
public:
        int day() const { return d; }    // return Date::d
        // ...
private:
        int d, m, y;
};
```

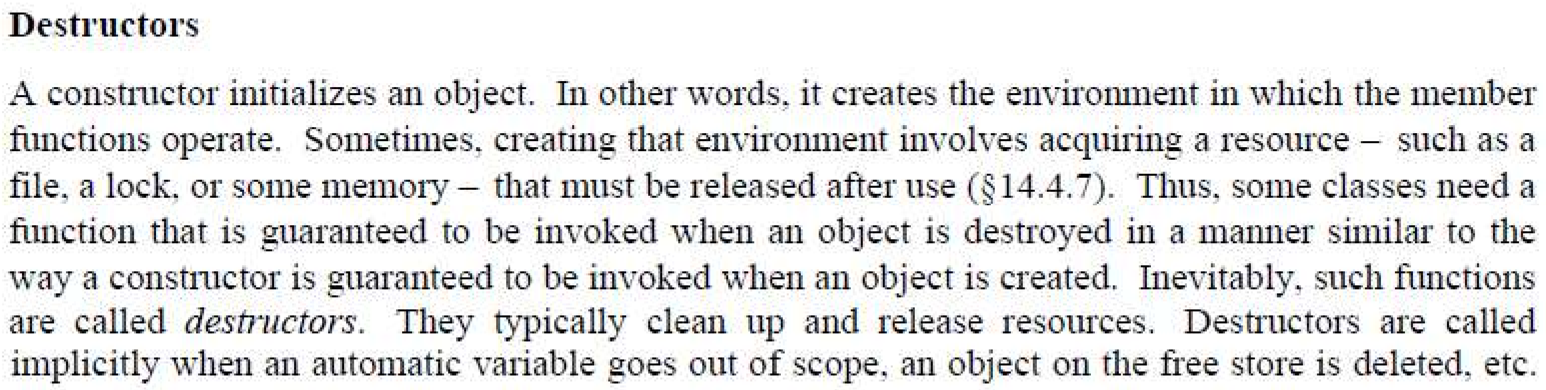


This is perfectly good C++ code because a member function declared within a class can refer to every member of the class as if the class were completely defined before the member function bodies were considered. However, this can confuse human readers.



```
class Date {
public:
        int day() const;
        // ...
private:
        int d, m, y;
};

inline int Date::day() const { return d; }
```

Reference Book: "The C++ Programming Language", Bjarne Stroustrup.



Destructors

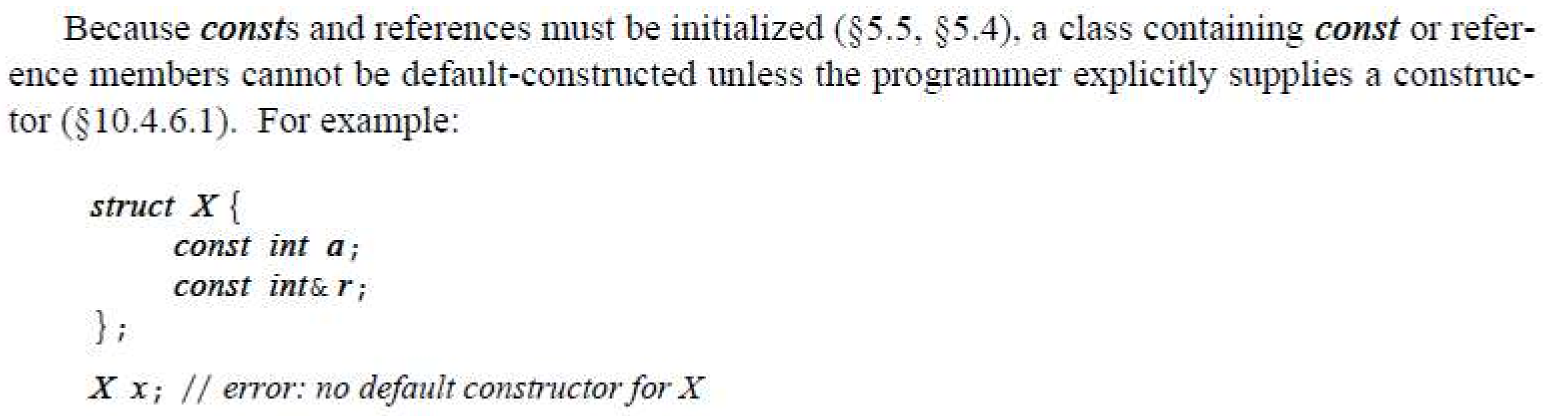
A constructor initializes an object. In other words, it creates the environment in which the member functions operate. Sometimes, creating that environment involves acquiring a resource – such as a file, a lock, or some memory – that must be released after use (§14.4.7). Thus, some classes need a function that is guaranteed to be invoked when an object is destroyed in a manner similar to the way a constructor is guaranteed to be invoked when an object is created. Inevitably, such functions are called *destructors*. They typically clean up and release resources. Destructors are called implicitly when an automatic variable goes out of scope, an object on the free store is deleted, etc.

Reference Book: “The C++ Programming Language”, Bjarne Stroustrup.

The most common use of a destructor is to release memory acquired in a constructor. Consider a simple table of elements of some type ***Name***. The constructor for ***Table*** must allocate memory to hold the elements. When the table is somehow deleted, we must ensure that this memory is reclaimed for further use elsewhere. We do this by providing a special function to complement the constructor:

```
class Name {
    const char* s;
    // ...
};

class Table {
    Name* p;
    size_t sz;
public:
    Table(size_t s = 15) { p = new Name[sz = s]; }// constructor

    ~Table() { delete[] p; }                       // destructor

    Name* lookup(const char *);
    bool insert(Name*);
};
```

Default Constructors



Because *const*s and references must be initialized (§5.5, §5.4), a class containing *const* or reference members cannot be default-constructed unless the programmer explicitly supplies a constructor (§10.4.6.1). For example:

```
struct X {
      const int a;
      const int& r;
};

X x; // error: no default constructor for X
```

Reference Book: "The C++ Programming Language", Bjarne Stroustrup.

Construction and Destruction

Consider the different ways an object can be created and how it gets destroyed afterwards. An object can be created as:

A named automatic object, which is created each time its declaration is encountered in the execution of the program and destroyed each time the program exits the block in which it occurs

A free-store object, which is created using the ***new*** operator and destroyed using the ***delete*** operator

A nonstatic member object, which is created as a member of another class object and created and destroyed when the object of which it is a member is created and destroyed

An array element, which is created and destroyed when the array of which it is an element is created and destroyed

A local static object, which is created the first time its declaration is encountered in the execution of the program and destroyed once at the termination of the program

A global, namespace, or class static object, which is created once ‘‘at the start of the program’’ and destroyed once at the termination of the program

A temporary object, which is created as part of the evaluation of an expression and destroyed at the end of the full expression in which it occurs

Reference Book: “The C++ Programming Language”, Bjarne Stroustrup.

Local Variables

The constructor for a local variable is executed each time the thread of control passes through the declaration of the local variable. The destructor for a local variable is executed each time the local variable's block is exited. Destructors for local variables are executed in reverse order of their construction. For example:

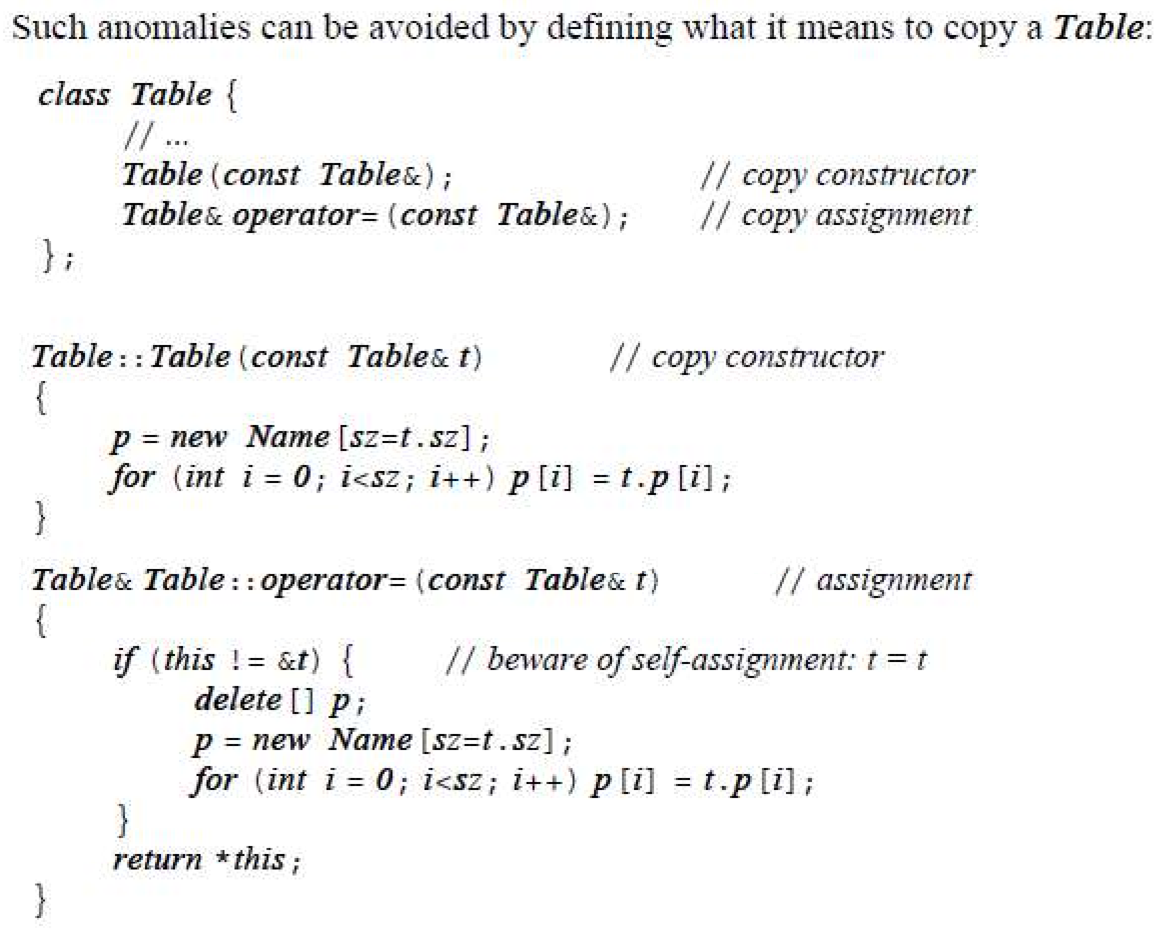
```
void f(int i)
{
    Table aa;
    Table bb;
    if (i>0) {
        Table cc;
        // ...
    }
    Table dd;
    // ...
}
```

Copying Objects

If ***t1*** and *t2* are objects of a class ***Table***, *t2*=***t1*** by default means a memberwise copy of ***t1*** into *t2* (§10.2.5). Having assignment interpreted this way can cause a surprising (and usually undesired) effect when used on objects of a class with pointer members. Memberwise copy is usually the wrong semantics for copying objects containing resources managed by a constructor/destructor pair. For example:

```
void h()
{
        Table t1;
        Table t2 = t1;   // copy initialization: trouble
        Table t3;

        t3 = t2;         // copy assignment: trouble
}
```

Here, the ***Table*** default constructor is called twice: once each for ***t1*** and ***t3***. It is not called for *t2* because that variable was initialized by copying. However, the ***Table*** destructor is called three times: once each for ***t1***, *t2*, and ***t3***! The default interpretation of assignment is memberwise copy, so ***t1***, *t2*, and ***t3*** will, at the end of ***h***(), each contain a pointer to the array of names allocated on the free store when ***t1*** was created.



Such anomalies can be avoided by defining what it means to copy a ***Table***:

```
class Table {
        // ...
        Table(const Table&);                    // copy constructor
        Table& operator=(const Table&);         // copy assignment
};

Table::Table(const Table& t)            // copy constructor
{
        p = new Name[sz=t.sz];
        for (int i = 0; i<sz; i++) p[i] = t.p[i];
}

Table& Table::operator=(const Table& t)         // assignment
{
        if (this != &t) {       // beware of self-assignment: t = t
                delete[] p;
                p = new Name[sz=t.sz];
                for (int i = 0; i<sz; i++) p[i] = t.p[i];
        }
        return *this;
}
```

Reference Book: “The C++ Programming Language”, Bjarne Stroustrup.