CPP Programming: Session 5

Anupama Chandrasekhar (Aug 30 2018)

Constructor, Initialization, Copy

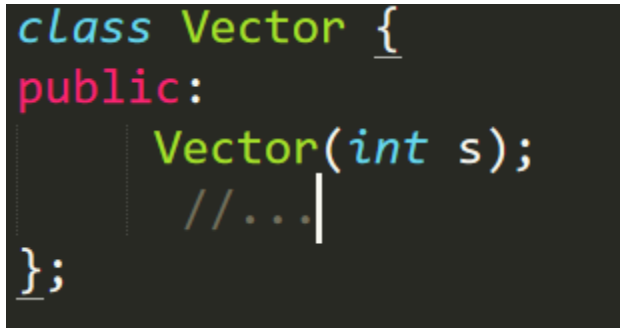
```
string ident(string arg)          // string passed by value (copied into arg)
{
    return arg;                   // return string (move the value of arg out of ident() to a caller)
}

int main ()
{
    string s1 {"Happy"};          // initialize string (construct in s1).
    s1 = ident(s1);               // copy s1 into ident()
                                  // move the result of ident(s1) into s1;
                                  // s1's value is "Happy".

    string s2 {"Joy"};            // initialize string (construct in s2)
    s1 = s2;                      // copy the value of s2 into s1
                                  // both s1 and s2 have the value "Joy".
}
```

Constructors and Destructors

- An object is considered an object of its type after the constructor completes and it remains that type until the destructor starts executing.
- We can specify how an object of a **`class`** is initialized by defining a constructor and to ensure cleanup we can define a destructor at the point of destruction of an object (when it goes out of scope)
- See **`1_cons_des.cpp`**

Constructor Basics

- A member with the same name as its **`class`** is called a constructor



- A constructor declaration specifies an argument list (like a function) but has no return type.

Constructor Basics

- The name of a class cannot be used for an ordinary member function, data member, member type (**`enum/enum class`**). See **`2_cats.cpp`**

```
struct S {

    S();                    // fine
    void S(int);            // error: no type can be specified for a constructor
    int S;                  // error: the class name must denote a constructor
    enum S { foo, bar };    // error: the class name must denote a constructor
};
```

Constructor Basics

- Often a constructor's job is to establish a **class** invariant. In the example below the invariant can be stated as "elem points to an array of **sz** doubles"

```
class Vector {
public:
    Vector(int s);
     //...

private:
    double* elem;   // elem points to an array of sz doubles
    int sz;         // sz is non-negative
};
```

Constructor Best Practices

- Constructor tries to establish an invariant, if it fails it throws an exception.
- The constructor must also ensure that if no object is created that no resources are leaked.
- Resource: memory, locks, file handles and thread handles

```
Vector::Vector(int s)
{
    if (s<0) throw Bad_size{s};
    sz = s;
    elem = new double[s];
}
```

Destructors

- A constructor creates an environment in which member functions operate, creating this environment sometimes involves acquiring a resource like a file, lock or some piece of memory. This must be released after use.
- Classes therefore need a function that is guaranteed to be invoked when the object is destroyed the same way as the constructor is invoked when an object is created. This is the destructor. Denoted by ~<classname>
- Destructor does not take an argument and a class can have only one destructor.
- Destructors are called implicitly when an automatic variable goes out of scope, an object on the free store is deleted etc.
- Should we ever call a destructor explicitly?

Destructor : Example

```
class Vector {
public:
    Vector(int s) :elem{new double[s]}, sz{s} { }; // constructor: acquire memory
    ~Vector() { delete[] elem; }                   // destructor: release memory
     //...
private:
    double* elem;     // elem points to an array of sz doubles
    int sz;           // sz is non-negative
};
```

Destructor : Usage Example

- Here, the **Vector v1** is destroyed upon exit from **f()**.
- Also, the **Vector** created on the free store by **f()**using **`new`** is destroyed by the call of **`delete`**.
- In both cases, **`vector`**'s destructor is invoked to free (deallocate) the memory allocated by the constructor.
- If **`new`** failed to acquire enough memory, a **`std::bad_alloc`** exception is thrown by new and the exception handling mechanism invokes the appropriate destructors so that all the memory that has been acquired is **`free`**d.

```
Vector* f(int s)
{
        Vector v1(s);
         //...
        return new Vector(s+s);
}

void g(int ss)
{
        Vector* p = f(ss);
         //...
        delete p;
}
```

Summarizing the order of operations

- A constructor builds an object bottom up
  - The constructor invokes its base constructors
  - Then it invokes all the member constructors
  - Finally it executes its own body
- A destructor tears down an object in the reverse order
  - It destroys its own body
  - Then invokes its member destructors
  - Finally, it invokes its base class destructors
- Constructors execute member constructors in declaration order (not the order of initializers)

RAII: Resource Acquisition Is Initialization

```
void use_file(const char* fn)  // naive code
{
    FILE* f = fopen(fn,"r");
    //... use f ...
    fclose(f);
}
```

RAII : Bad Solution

```
void use_file(const char* fn)    // clumsy code
{
    FILE* f = fopen(fn,"r");
    try {
        //... use f ...
    }
    catch (...) {          // catch every possible exception
        fclose(f);
        throw;
    }
    fclose(f);
}
```

RAII: Good Solution

```
class File_ptr {
     FILE* p;
public:
     File_ptr(const char* n, const char* a)    // open file n
          : p{fopen(n,a)}
     {
          if (p==nullptr) throw runtime_error{"File_ptr: Can't open file"};
     }

     File_ptr(const string& n, const char* a)  // open file n
          :File_ptr{n.c_str(),a}
     { }

     File_ptr(FILE* pp)                // assume ownership of pp
          :p{pp}
     {
          if (p==nullptr) throw runtime_error("File_ptr: nullptr"};
     }

     ~File_ptr() { fclose(p); }

     operator FILE*() { return p; }
};

void use_file(const char* fn)
{
     File_ptr f(fn,"r");
      //... use f ...
}
```

- The destructor will be called whether the function exits properly or an exception is thrown.
- Managing resources using local objects is called RAII. It relies on the properties of constructors and destructors and their interactions with exception handling.

Object Initialization

- Initialization without constructors

```
int a {1};

char* p {nullptr};
```

- We can initialize objects of a **`class`** for which we have not defined a constructor using
  - Memberwise initialization
  - Copy initialization
  - Default initialization

Object Initialization

```
struct Work {
      string author;
      string name;
      int year;
};

Work s9 { "Beethoven",
          "Symphony No. 9 in D minor, Op. 125; Choral",
          1824
        };                              // memberwise initialization

Work currently_playing { s9 };          // copy initialization
Work none {};                           // default initialization
```

- The three members of **currently_playing** are copies of those of **s9**.
- The default initialization of using **{}** is defined as initialization of each member by **{}**. So, **none** is initialized to **{{},{},{}}**, which is **{"","",0}**
- Where no constructor requiring arguments is declared, it is also possible to leave out the initializer completely.

Object Initialization

```
Work alpha;

void f()
{
    Work beta;
     //...
}
```

- For statically allocated objects the rules are just like you used {}. so the value of **alpha** is **{"","",0}**
- But for local variables the values of the built in types is left uninitialized, so the value of **beta** is **{"","",unknown}.**
- **Reason: Performance, just like the array case with built-ins**

Object Initialization

```
struct Buf {
        int count;
        char buf[16*1024];
};

Buf buf0;           // statically allocated, so initialized by default

void f()
{

        Buf buf1;                   // leave elements uninitialized
        Buf buf2 {};                // I really want to zero out those elements

        int* p1 = new int;          //*p1 is uninitialized
        int* p2 = new int{};        //*p2 == 0
        int* p3 = new int{7};       //*p3 == 7
       //...
}
```

Debug this!

```
class Checked_pointer {  // control access to T* member
private:
    int* p;
public:
     int& operator*();    // check for nullptr and return value
};

Checked_pointer p {new int{7}};   // error: can't access p.p
```

Solution

- If a class has a private non-static data member, it needs a constructor.

Initialization using Constructors

- Where memberwise copy is not sufficient or desirable, a constructor can be defined. In particular the constructor is used to establish a class invariant.
- If a constructor is declared for a **`class`**, some constructor will be used for every object. It is an error to create an object without a proper initializer as required by the constructors

Initializing using constructors

```
struct X{
    int x;
    X(int a){ x = a;}
};
X x0;           // error: no initializer
X x1 {};        // error: empty initializer
X x2 {2};       // OK
X x3 {"two"};   // error: wrong initializer type
X x4 {1,2};     // error: wrong number of initializers
X x5 {x2};      // OK: a copy constructor is implicitly defined
```

Initializing Objects

```
struct S1 {
        int a,b;                                                    // no constructor
};

struct S2 {
        int a,b;
        S2(int aa = 0, int b = 0) : a(aa), b(bb) {}          // constructor
};

S1 x11(1,2);      // error: no constructor
S1 x12 {1,2};     // OK: memberwise initialization
S1 x13(1);        // error: no constructor
S1 x14 {1};       // OK: x14.b becomes 0
S2 x21(1,2);      // OK: use constructor
S2 x22 {1,2};     // OK: use constructor
S2 x23(1);        // OK: use constructor and one default argument
S2 x24 {1};       // OK: use constructor and one default argument
```

Default Constructors

- Default constructors are invoked without an argument.
- Built-in types are considered to have default and copy constructors, but these are not invoked for uninitialized non-static variables.

Default Constructors (from cppreference)

Syntax

`ClassName ( ) ;`	(1)	
`ClassName :: ClassName ( )` *body*	(2)	
`ClassName() = delete ;`	(3)	(since C++11)
`ClassName() = default ;`	(4)	(since C++11)
`ClassName :: ClassName ( ) = default ;`	(5)	(since C++11)

Explanation

1) Declaration of a default constructor inside of class definition.
2) Definition of the constructor outside of class definition (the class must contain a declaration (1)). See constructors and member initializer lists for details on the constructor *body*
3) Deleted default constructor: if it is selected by overload resolution, the program fails to compile.
4) Defaulted default constructor: the compiler will define the implicit default constructor even if other constructors are present.
5) Defaulted default constructor outside of class definition (the class must contain a declaration (1)). Such

Delegating Constructors

- If you want two constructors to do the same action, you can repeat yourself or define “an **init()**function” to perform the common action. Both “solutions” are common (because older versions of C++ didn’t offer anything better).

```
class X {
        int a;
        validate(int x) { if (0<x && x<=max) a=x; else throw Bad_X(x); }
public:
        X(int x) { validate(x); }
        X() { validate(42); }
        X(string s) { int x = to<int>(s); validate(x); }
};
```

Delegating Constructors

- Better way available in C++ 11

```
class X {
        int a;
public:
        X(int x) { if (0<x && x<=max) a=x; else throw Bad_X(x); }
        X() :X{42} { }
        X(string s) :X{to<int>(s)} { }
};
```

In Class Initializers

```cpp
// Preffered
class A {
public:
    int a {7};
    int b = 77;
};

class A {
public:
    int a;
    int b;
    A() : a{7}, b{77} {}
};
```

Static Member and Initializing them

- A **`static`** class member is statically allocated rather than part of each object of the class. Generally, the **`static`** member declaration acts as a declaration for a definition outside the **`class`**.

```
class Node {
      //...
     static int node_count;           // declaration
};

int Node::node_count = 0;             // definition
```

Static Members contd.

- For a few cases you can initialize static members inside a **`class`**

```
class Curious {
public:
    static const int c1 = 7;          // OK
    static int c2 = 11;               // error: not const
    const int c3 = 13;                // OK, but not static
};
```

- The main use of member constants is to provide symbolic constants

```
template<typename T, int N>
class Fixed {  // fixed-size array
public:
    static constexpr int max = N;
private:
    T a[max];
};
```

Copy Constructors

- If we need to move a to b we have two options:
  - Copy : x = y, at the end of this operation both x and y have the same value
  - Move: Leaves x with the value of y and y is some undefined state.
- **Copy** is defined by two operations
  - Copy constructor : X (const X&)
    - This is needed to support, say pass by value
    - If you do not provide one the compiler provides a default memberwise copy
    - See 3_line.cpp
  - Copy Assignment: X& operator=(const X&)
    - This is needed to support a = b.
    - See 4_copyassignment.cpp

Deep Copy versus Shallow copy

- Sometimes member-wise copy is not sufficient, say when some members are raw pointers and you want to copy what the pointer points to and not the pointer value (risk of dangling pointers too)
- In this case we need to define a copy constructor that will perform a deep copy.

Putting it all together

- RAII with smart pointers

Parking Lot Problem

- What classes can we have?

Backup :: Uniform Initialization

- For this reason, **{}** initialization is sometimes referred to as *universal* initialization: the notation can be used everywhere. In addition, **{}** initialization is *uniform*: wherever you initialize an object of type **X**with a value **v** using the **{v}** notation, the same value of type **X** (**X{v}**) is created.
- The **=** and **()** notations for initialization are not universal
- Note that the **{}**-initializer notation does not allow narrowing. That is another reason to prefer the **{}** style over **()** or **=**.

Backup:Some Definitions

- Move and Copy: The difference between move and copy is that after a copy, two objects have the same value but after a move the original object is not required to have the same value. Moves can be used when the source object will not be used again.
- Functions used in the example:
  - A constructor to initialize a string with a string literal
  - A copy constructor copying a string