Programming 2

Object Oriented Programming

Daniel POP

Unit 3

Agenda

1. Constructors
    1. Definition. Examples
    2. Special constructors
    3. explicit constructor

2. Destructor

Reference

- **DEFINITION [Reference] A reference is an alternative (alias) name for an object.**

- Syntax: X& ref = object;   // declaration of a lvalue reference to type X
- Syntax: X&& ref = object; // declaration of a rvalue reference to type X

- Each reference MUST BE initialized to a valid object!
- **References are not objects**; they do not necessarily occupy storage, although the compiler may allocate storage if it is necessary to implement the desired semantics (e.g. a non-static data member of reference type usually increases the size of the class by the amount necessary to store a memory address).
- Because references are not objects, **there are no arrays of references, no pointers to references, and no references to references**
- Usages
  - Operator overloading
  - Passing arguments by reference in function calls
- Common implementation of references is as a constant pointer that is transparently dereferenced on each usage.
- When a function's return type is lvalue reference, the function call expression becomes an lvalue expression
- Rvalue reference can be used to extend the lifetimes of temporary objects (see example)

- More about reference [Stroustrup, 1997 – Section 5.5]

Reference

```
void f() {
     int i = 1;
     int& r = i; // r and i reference the same memory location
     int& r2; // ERROR: initializer is missing
     int x = r; // x = 1
     r = 2; // i = 2 as well
     int* p = &r; // p points to i
}
```

```
void increment(int& i) {
     i++;
}

void f() {
     int a = 5;
     increment(a);
     // => a=6
}
```

```
char& char_number(std::string& s, std::size_t n) {
    return s.at(n); // string::at() returns a reference to char
}

int main() {
    std::string str = "Test";
    char_number(str, 1) = 'a'; // the function call is lvalue, can be assigned to
    std::cout << str << '\n';  // outputs ‘Tast’
}
```

```
int main() {
    std::string s1 = "Test";
    // std::string& r1 = s1 + s1;    // error: can't bind to lvalue
    const std::string& r2 = s1 + s1; // okay: lvalue reference to const extends lifetime
    //  r2 += "Test";                // error: can't modify through reference to const

    std::string&& r3 = s1 + s1;      // okay: rvalue reference extends lifetime
    r3 += "Test";                    // okay: can modify through reference to non-const
    std::cout << r3 << '\n';
}
```

Constructors (I)

- Rationale: to prevent errors caused by non-initialized objects

- Example: Date objects, a Stack with dangling pointer to its top element etc.

- **DEFINITION [Constructor] A non-static member function of a class whose role is to initialize the class instances (objects).**

- Object creation process:
  1. Memory allocation
  2. Find an appropriate constructor
  3. Call the constructor to initialize the object's state **after** the data members have been previously constructed/initialized by calling their constructors

- Q: Which member functions are the constructors?
- A: The constructor name = class name

- Characteristics of constructors:
  - The name = Class name
  - They have NO return value (Remark: void **is** a return value! => don't use it)
  - No pointers can be assigned to constructors (PMF pmf = &Date::Date; is illegal)

- Except for the above constraints, constructors are handled as any other member function of a class (for example, they may have 0, 1 or more arguments, can call other member functions etc.)

Constructors (II)

```
class Date {
     int _day, _month, _year;
     void init(int day, int month, int year);

public:
     // Constructors
     Date(int day, int month, int year);
     Date(int day, int month);
     Date(int day);
};

void f() {
     // Old-style
     Date d0 = Date(6, 10, 2003); // correct
     Date d1(1, 1); // correct
     Date d2(5);   // correct
     // ERROR: No appropriate constructor is found
     Date d3;
     Date* pd = new Date(7, 10, 2003); // correct

     // C++ 11 style
     // Use brackets {} instead of ()
     // Preferred over old-style for clarity
     Date d0 = Date{6, 10, 2003};
     Date d1{1, 1};
     Date d2{5};
     Date* pd = new Date{7, 10, 2003};
}
```

```
class Date {
     int _day, _month, _year;
     void init(int day, int month,
               int year);

public:
     // Constructors!
     Date(int day = 0, int month=0,
          int year=0);
};

void f()
{
     // CORRECT: Constructor found
     Date d3;
}
```

```
#include "Date.h"

Date::Date(int day, int month, int year) {
     _day = day ? day : today.day;
     _month = month ? month : today.month;
     _year = year ? year : today.year;
}
```

Constructors (III). Default Constructor

- Constructor without arguments.

- Prototype: **X();**

- If a class has no constructors then the compiler generates a default inline constructor for that class that implicitly calls the default constructors for class' members (of class type) and base classes
- X() = delete; inhibiting the automatic generation of a default constructor by the compiler
- X() = default; explicitly forcing the automatic generation of a default constructor by the compiler

```
class Date {
 public:
     // default constructor
     Date(int day=0, int month=0, int year=0);
};

class String {
     public:
          String(); // default constructor
};

class Student {
     Date birthday;
     String name;
     // automaticlly generated default constructor that
     // calls Date and String defaults constructors
};
```

*Remark: If a class has const or reference members then the default is **not** generated automatically because consts and references must be initialized.*

```
class Abc {
 public:
     // no default constructor generated
     Abc(int day);
     // unless we force its generation
     Abc() = default;
};

class Test {
     const int a;
     int& r;
     // no default constructor generated
};
```

Constructors (IV). Copy Constructor

- Constructor with one argument of type lvalue reference to its own class.

- Prototype: **X(const X&);**

- const – indicates that the source object is not modifiable
- It is called:
  - Declaration of an object like X obj = obj_source;
  - Passing an object as an argument to a function call func(X); **WHY?**
  - When a temporary object is created during expression evaluation
- If it is not defined for one class, then the compiler automatically generates a non-explicit, inline one that bitwise copies the content of argument object.
- X(const X&) = delete; inhibiting the automatic generation of a copy constructor by the compiler

- **To 'deep' copy complex objects, the copy-constructor is mandatory! (Examples in lab)**

Constructors (V). Copy Constructor

```
class Date {
public:
     Date(int day=0, int month=0, int year=0); // default constructor
     Date(const Date& ref); // user-defined copy constructor

    void setDay(int day);
};
```

```
void g(Date d) {
     d.setDay(15);
}

void f() {
     Date d{7, 3, 2007}; // user-defined constructor
     Date d1; // default constructor
     Date d2 = d; // copy constructor
     d2 = d; // assignment operator

     g(d); // copy constructor is called
}
```

Constructors (VI). Move Constructor (C++11)

- Constructor with one argument of type rvalue reference to its own class.

- Prototype: **X(const X&&);**

- The move constructor is called when an object is initialized (by direct-initialization or copy-initialization) from a rvalue of the same type, including:
  - Initialization: T a = std::move(b);
  - Argument passing: f(std::move(a));
  - Function return: return a; inside a function X f();
- Move constructors typically "steal" the resources held by the argument (e.g. pointers to dynamically-allocated objects, file descriptors, TCP sockets, I/O streams, threads, etc), rather than make copies of them, and leave the argument in some valid but otherwise indeterminate state.
- For example, moving from a std::string or from a std::vector may result in the argument being left empty. However, this behavior should not be relied upon.
- If it is not defined for one class, then the compiler automatically generates a non-explicit, inline one that bytewise copies the content of argument object (same as copy constructor).
- X(const X&&) = delete; inhibiting the automatic generation of a move constructor by the compiler

Constructors (VII). Move Constructor

```
#include <string>
#include <iostream>
#include <iomanip>
#include <utility>

struct A {
    std::string s;
    A() : s("test") {
         std::cout << “default ctor\n";
    }
    A(const A& o) : s(o.s) {
         std::cout << “copy ctor\n";
    }
    A(A&& o) noexcept : s(std::move(o.s)) {
         std::cout << “move ctor\n";
    }
};

A f(A a) {
    return a; // will call move constructor
}

int main() {
    A a1 = f(A());        // default-ctor followed by move-ctor from rvalue temporary
    A a2 = std::move(a1); // move-construct from rvalue
}
```

Constructors (VIII). Type Conversion Constructors

- Convert from one data type to another data type, user-defined class (built-in type, or user-defined-type => class).

- Prototype: **X(Datatype);**

- Called in
  - declarations like X x = value; where value’s type is Datatype (1)
  - function calls to cast the actual argument to required type (2)

```
class Double {
	double val;
public:
	Double(double v) {
		value = v;
	}
} ;

vod f() {
	// three equivalent declarations
	Double obiect = 2.5;
	Double obiect = Double{2.5};
	Double obiect{2.5};
}
```

```
class Date {
public:
	// . . . Declarations . . .
	// type-conversion constructor
	Date(const char* str);
	void setDay(int d);
};

void g(Date d) {
	d.setDay(7);
}

void f() {
	Date d = “6/3/2007”; // (1)
	g(d);
	g(“This is tricky ;-)”); // (2)
}
```

Constructors (IX). explicit Constructors

- A constructor declared with the keyword **explicit** can only be used for initialization and explicit conversions, and not used in implicit conversions.

- Explicit (direct) initialization; Date d1{15}, d2(12);
- Implicit initialization: Date d3 = 15;
- Implicit initialization also occurs when passing arguments to function calls

- As good practice, you should declare all constructors that can be called with a single argument using *`explicit`* keyword

```
class Double {
     double val;
public:
     explicit Double(double v) {
          value = v;
     }
} ;

vod f() {
     // three equivalent declarations
     Double object = 2.5; // ERROR: implicit
     Double object = Double{2.5}; // OK explicit
     Double object{2.5}; // OK: explicit
}
```

```
class Date {
public:
     explicit Date(const char* str);
     void setDay(int d);
};

void g(Date d) {
     d.setDay(7);
}

void f() {
     Date t{“6/3/2007”};  // OK or ERR?
     Date d = “6/3/2007”; // OK or ERR?
     g(d);                // OK or ERR?
     g(“This is tricky”); // OK or ERR?
}
```

Destructor (I)

- Rationale: to prevent errors caused by un-released objects (unreleased resources).

- Example: Stack without freeing the memory used, File without closing the handle etc.

- **DEFINITION [Destructor] A member function of a class whose role is to release the class instances (objects) from memory.**

- Object destruction process:
  1. Call the destructor function
  2. Call destructors of data member
  3. Free the memory

- Q: Which member function is the destructor?
- A: The destructor name = ~class_name (i.e. ~X)

- Characteristics of destructor:
  - Prototype: **X::~X();**
  - They have NO return value (Remark: void is a return value) and they take no arguments.
  - No pointers can be assigned to destructor (PMF pmf = &Date::~Date; is illegal)

- Except for the above constraints, destructor is handled as any other member function of a class.

Destructor (II)

- **A class can have at most one destructor.**
- The compiler generates an empty default destructor.

```
class String {
     char* str;

public:
     // Constructors
     String(char* psz=NULL);
      ~String();
};

void f() {
     String s1;
     String s2=“We like C++;)”;
     String s3=s2;
     String* s4 = new String(“Do we?”);

     delete s4; // destructor is called
} // destructors for s3, s2, s1 are called in this order
```

```
String::String(char* psz) {
     if(psz==NULL)
          str=NULL;
    else {
          str = new char [strlen(psz)+1];
          strcpy(str, psz);
    }
}

String::~String() {
     if(str!=NULL)
          delete [] str;
}
```

Try this &
Explain the behaviour!

Constructors & Destructor

- Both, constructor and destructor can be explicitly called.
  NOT RECOMMENDED TO DO THIS!

```
void f() {
     String s1;
     s1.String::String(“Explicit call”);
     s1.~String();
}
```

RECOMMENDATION: If a class has a pointer member, then it (most likely) needs:

- a copy constructor,
- a destructor and
- an assignment operator (see operator overloading).

Further Reading

1. [Stroustrup, 1997] Bjarne Stroustrup – The C++ Programming Language 3rd Edition, Addison Wesley, 1997 [Chapter 10]

2. [Stroustrup, 2013] Bjarne Stroustrup – The C++ Programming Language 4th Edition, Addison Wesley, 2013 [Chapter 16]