# Introduction to Standard C++

## Lecture 02: A Primer on Classes

Massimiliano Culpo[1]

[1]CINECA - SuperComputing Applications and Innovation Department

07.04.2014

# Table of contents - Language rules

# Table of contents - Language rules

# What is a class in the first place?

A class is a new type. The following snippet:

```
struct X { int a; };
struct Y { int a; };
X a1;
Y a2;
int a3;
```

declares three variables of three different types. This implies that:

```
a1 = a2; // error: Y assigned to X
a1 = a3; // error: int assigned to X

int f(X); // overload for type X
int f(Y); // overload for type Y
```

# Class names (§9.1)

```
struct A {
  int a; // the name A is seen here...
};
// ... and from here on
struct B {
  A a;
};
struct C; // forward declaration of C
```

1. A class-name is inserted into the scope immediately after it is seen
2. The class name is also inserted in the scope of the class itself
3. A declaration consisting solely of the class-key identifier is either:
    - a redeclaration of the name in the current scope
    - a forward declaration of the identifier as a class name

# Members of a class (§9.2)

The member-specification in a class definition declares the full set of members of the class:

```
struct A {
  // 1. Nested types
  typedef float value_type;
  struct B { double b };
  // 2. Nested enumerations
  enum { RED, BLACK };
  // 3. Member functions
  int compute(float input);
  // 4. Member data
  float a;
};
```

No other member can be added elsewhere

# Data members (§9.2)

A class is a complete type at the closing };  of the class specifier:

```
struct A {
  float a;
  // error: type A is incomplete
  A  b;
  // ok: pointer to A is a complete type
  A* c;
};
```

1. It follows that a class **A**:
   - shall not contain a non-static member of class **A**
   - can contain a pointer to an object of class **A**
2. Non-static data members shall not have incomplete types

# Member functions (§9.3)

Functions declared within a class definition are member functions:

```
struct X {
  // 1. Implicit inline member function
  void f(int) {}
  void g(int);
  void h(int);
};
// 2. Explicit inline member function
inline void X::g(int t) { }
// 3. Non inline member function (X.cpp)
void X::h(int t) { }
```

1. There shall be at most one definition of a non-inline member function
2. If the definition of a member function is outside its class definition, the member function name shall be qualified

# Non-static member functions (§9.3.1)

A member function may be called using the class member access syntax:

```
struct tnode {
  // Member data
  tnode * left;
  tnode * right;
  // Member functions
  void set(tnode* l, tnode* r);
};

void f(tnode& n1, tnode& n2) {
  // Member access syntax
  n1.set(&n2,0);
  n2.set(0   ,0);
}
```

for an object of its class type

# Non-static member functions (§9.3.1)

A member function may be called using the function call syntax:

```
struct tnode {
  // Member data
  tnode * left;
  tnode * right;
  // Member functions
  void set     (tnode* l, tnode* r);
  void execute(tnode* l, tnode* r);
};

void tnode::execute(tnode* l, tnode* r) {
  set(l,r);
  // Do something else
}
```

from within the body of a member function

# Non-static member functions (§9.3.1)

A non-static member function may be declared **const** and/or **volatile**:

```
struct X {
  // volatile member function
  void f() volatile;
  // const member function
  void g() const;
  // const volatile member function
  void h() const volatile;
};
```

1. The cv-qualifiers affect the type of:
   - the object calling the member function (through the **this** pointer)
   - the member function itself

# The **this** pointer (§9.3.2)

The keyword **this**:

1. is defined in the body of non-static member functions
2. returns the address of the object for which the function is called
3. has type **X**$*$ in a member function of a class **X**
4. has type **const X**$*$ in a **const** member function
5. has type **volatile X**$*$ in a **volatile** member function
6. has type **const volatile X**$*$ in a **const volatile** member function

A cv-qualified member function can be called on an object:

1. if it is as cv-qualified as the member function
2. if it is less cv-qualified than the member function

# Static member functions (§9.4.1)

A data or function member of a class may be declared **static**:

```
struct process {
  static void reschedule();
};
process& g();
void f() {
  process::reschedule(); // qualified-id
  g().reschedule(); // class member access
}
```

1. A static member of a class may be referred to:
   - using a qualified-id expression
   - using the class member access syntax
2. A static member function does not have a **this** pointer

# Static member data (§9.4.2)

```
class process {
  static process* run_chain;
  static process* running;
};

// Definition of static data-members
process* process::running = get_main();
process* process::run_chain = running;
```

1. A static data member is not part of the sub-objects of a class
2. One copy of this member is shared by all the objects of the class
3. The declaration of a static data member is not a definition

# Access specifiers (§11)

Each member of a class has one access specification:

```
class A { // class is private by default
public:
  int   i; // public access
  float f; // public access
private:
  double d;// private access
};
```

**private** : the name can be used only by members and friends

**protected** : the name can be used only by members and friends of the class in which it is declared, by classes derived from that class, and by their friends

**public** : the name can be used anywhere without access restriction

# Access specifiers (§11)

Any number of access specifiers is allowed:

```
struct S {
  int a; // S::a is public by default
protected:
  int b; // S::b is protected
private:
  int c; // S::c is private
public:
  int d; // S::d is public
};
```

Members of a class defined with the keyword:

1. **class** are **private** by default
2. **struct** are **public** by default

# The **friend** keyword (§11.3)

A class specifies its friends, if any, by way of friend declarations:

```
class X {
  int a;
  friend void friend_set(X*, int);
public:
  void member_set(int);
};
void friend_set(X* p, int i) { p->a = i; }
void X::member_set(int i) { a = i; }
void f() {
  X obj;
  friend_set(&obj,10);
  obj.member_set(10);
}
```

A friend of a class is a function or class that is given permission to use the private and protected member names from the class.

# The **friend** keyword (§11.3)

```
class M {
  // definition of global f,
  // a friend of M, not the
  // definition of a member function
  friend void f() { }
};
```

1. A function can be defined in a friend declaration of a class iff:
   - the class is a non-local class
   - the function name is unqualified
   - the function has namespace scope
2. Friendship is neither inherited nor transitive
3. Friend declarations do not depend on access specification

# Q: Are all these accesses to members well-formed?

```
class A {
  typedef int I;
  I f();
  friend I g(I);
  static I x;
};

A::I A::f() { return 0; }
A::I g(A::I p = A::x);
A::I g(A::I p) { return 0; }
A::I A::x = 0;
```

# Table of contents - Language rules

# Special member functions (§12)

```
struct A { }; // implicit A::operator=
A a, b;
/* ... */
b = a; // well formed
b.operator=(a); // well formed as well
```

1. The following functions are considered special member functions:
   - default constructor
   - copy constructor and copy assignment operator
   - move constructor and move assignment operator
   - destructor
2. The implementation will implicitly declare these member functions for some class types
3. Programs may explicitly or implicitly refer to special member functions

# Constructors (§12.1)

A special syntax is used to declare or define constructors:

```
struct S {
  S(); // declares the constructor
};

S::S() { // defines the constructor
 }
```

1. A constructor is used to initialize objects of its class type
2. A constructor shall not be virtual or static
3. A default constructor is a constructor that takes no arguments
4. If there is no user-declared constructor for class **S**, a default constructor is implicitly declared as an inline public member

# Data-members inizialization (§12.6)

```
class X {
  int a;  int b;
  int i;  int j;
  const int& r;

  X(int i) : // Initializers
  b(i), i(i), j(this->i), r(a)
  { }
};
```

1. Initializers for non-static data members can be specified by a list of constructor initializer
2. If a given non-static data member is not designated by a member initializer, then it is default constructed.

# Copy and assignment (§12.8)

A class object can be copied in two ways:

```
struct X {
  X(int);
  X(const X&, int = 1);
};

X a(1); // calls X(int);
X b(a, 0); // calls X(const X&, int);
X c = b; // calls X(const X&, int);
c = a;
```

1. by initialization, using a copy constructor operator
2. by assignment, using a copy assignment operator

# Copy constructor (§12.8)

1. A constructor for class **X** is a copy constructor if:
   - its first parameter is of type **X**& (or any cv-qualified variant)
   - there are no other parameters
   - all other parameters have default arguments
2. If the class definition does not explicitly declare a copy constructor:
   - a copy constructor is implicitly declared as defaulted
   - ...unless the class has a user-declared copy assignment or destructor
3. The implicitly-defined copy constructor performs a memberwise copy
4. Non-static data members are initialized in the order of declarations

# Assignment operator (§12.8)

A user-declared copy assignment operator:

```
struct X {
  X();
  X& operator=(const X&);
};

const X cx;
X x;
void f() {
  x = cx;
}
```

1. is a non-static member function of **X**
2. has exactly one parameter of type **X** or (cv-qualified) **X&**

# Assignment operator (§12.8)

1. If a class does not explicitly declare a copy assignment operator and:
   - there is no user-declared move constructor
   - there is no user-declared move assignment operator

   a copy assignment operator is implicitly declared as defaulted
2. Such implicit declaration is deprecated if the class has:
   - a user-declared copy constructor
   - a user-declared destructor

# Object inizialization (§12.6)

```
struct complex {
  complex(); // Default constructor
  complex(double);
  complex(double,double);
};
complex sqrt(complex,complex);
// complex(double)
complex a(1);
// complex(double,double)
complex c = complex(1,2);
// sqrt(complex,complex)+copy
complex d = sqrt(a,c);
// complex()
complex e;
```

# Q: What is the difference, if any, between the following?

```
SomeType t = u;
SomeType t(u);
SomeType t();
SomeType t;
```

It may be useful to review the following issues:

1. Default constructor
2. Copy constructor
3. Assignments
4. Declarations

before answering GotW #01.

# Destructor (§12.4)

A special syntax is also used to declare the destructor:

```
struct B {
  ~B(); // Destructor declaration
};

B::~B() {} // Destructor definition
```

1. A destructor is used to destroy objects of its class type
2. A destructor takes no parameters
3. No return type can be specified for it (not even void)
4. A destructor shall not be static

# Destructor (§12.4)

1. If a class has no user-declared destructor, a destructor is implicitly declared as defaulted
2. An implicit destructor is an inline public member of its class
3. Destructors are invoked implicitly for constructed objects:
   - with static storage duration at program termination
   - with automatic storage duration at block exit
   - if they are temporary, when the their lifetime ends
   - allocated by a new-expression, through use of a delete-expression

# Table of contents - Language rules

# Type conversions (§12.3)

Type conversions are specified by constructors and conversion functions:

```
struct X { operator int(); };
struct Y { operator X(); };
// At most one UD conversion per value
Y a;
int b = a; // error
int c = X(a) // ok
```

1. These user-defined conversions and are used for:
   - implicit type conversions
   - initialization
   - explicit type conversions
2. At most one user-defined conversion is implicitly applied

# Type conversion by constructor (§12.3.1)

A constructor declared without the function-specifier **explicit** :

```
struct X {
  X(int);
  X(const char*, int =0);
  explicit X(float);
};
void f(X arg) {
  X a = 1; // a = X(1)
  X b = "Jessie"; // b = X("Jessie",0)
  a = 2; // a = X(2)
  f(3); // f(X(3))
  X c = 2.0f; // error: explicit constructor
}
```

specifies a conversion from the types of its parameters

# Type conversion by constructor (§12.3.1)

An explicit constructor:

```
struct Z {
  explicit Z();
  explicit Z(int);
};

Z a;
Z a1 = 1; // error: no implicit conversion
Z a3 = Z(1);
Z a2(1);
Z* p = new Z(1);
```

1. Constructs objects where the direct-initialization syntax is used
2. Constructs objects where casts are explicitly used

# Conversion functions (§12.3.2)

If a conversion function is explicit:

```
class Y { };
struct Z {
  operator int();
  explicit operator Y() const;
};
void h(Z z) {
  Y y1(z);  // direct-initialization
  int a = y1; // conversion
  Y y2 = z; // error: copy-initialization
  Y y3 = (Y)z; //  cast notation
}
```

it is only considered as a user-defined conversion for direct-initialization

# Table of contents - Language rules

# Overloaded operators (§13.5)

A class may overload the following operators:

```
new   delete   new[]   delete[]
+     -     *     /     %     ^     &     |     []
!     =     <     >     +=    -=    *=    /=    %=
^=    &=    |=    <<    >>    >>=   <<=   ==    !=
<=    >=    &&    ||    ++    --    ,     ->*   ->
()    []
```

Operator functions are usually not called directly; instead they are invoked to evaluate the operators they implement:

```
complex z = a.operator+(b);
z = a + b;
```

# Overloaded operators (§13.5)

1. An operator function shall either be:
   - a non-static member function
   - a non-member function
2. It is not possible to change:
   - the precedence
   - the grouping
   - the number of operands

   of operators defined by the standard
3. An operator function, in general, cannot have default arguments
4. Operator functions cannot have more or fewer parameters than the number required for the corresponding operator by the standard

# Example: overloaded operators (§13.5)

```
struct X {
  X& operator++();     // prefix ++a
  X operator++(int); // postfix a++
};
struct Y { };

Y& operator++(Y&); // prefix ++b
Y  operator++(Y&, int); // postfix b++
void f(X a, Y b) {
  ++a; // a.operator++();
  a++; // a.operator++(0);
  ++b; // operator++(b);
  b++; // operator++(b, 0);
}
```

# Table of contents - Best practices

# Table of contents - Best practices

# Know what is implicit and what is not

If you don't declare them, compilers will declare their own versions of the default and copy constructor, copy assignment operator and destructor.

Thus, writing:

```
class A {};
```

is essentially the same as writing:

```
class A {
  A() {}
  A(const A & rhs) {}
  A & operator=(const A & rhs){}
  ~A(){}
};
```

# Disallow what you don't want

If you don't want a class to support a particular kind of functionality, you simply don't declare the function that would provide it.

This doesn't work for the copy constructor and assignment operator.

One possible solution is to declare the copy constructor and copy assignment operator as **private**:

```
class Uncopyable {
public:
  /* ... */
private:
  Uncopyable(const Uncopyable&);
  Uncopyable& operator=(const Uncopyable&);
};
```

The functions of course should not be defined, to disallow the possibility of member and friend functions calling them.

# Return value of the assignment operators

Assignment can be chained together:

```
x = y = z = 15;
```

Another interesting point is that assignment is right-associative:

```
// The previous statement is parsed like:
x = (y = (z = 15));
```

The way this behavior is implemented is that assignment returns a reference to its left-hand argument:

```
class A {
public:
  A& operator=(const A& rhs) {
    /* ... */
    return *this; } };
```

# Assignment to self

Code that operates on references or pointers to multiple objects of the same type, needs to consider that the objects might be the same.

Consider for instance the following situation:

```
class Resource { /* ... */ };

class ResourceHandler {
public:
  /* ... */
private:
  /* ... */
  Resource *pres;
};
```

where a resource handler manages a pointer to an heap-allocated object.

# Assignment to self

The most common pitfall in this case is to release a resource before you are done using it:

```
ResourceHandler&
ResourceHandler::operator=
(const ResourceHandler& rhs) {
  // Stop using current resource
  delete pres;
  // Deep copy of rhs resource
  pres = new Resource(*rhs.pres);
  // Return a reference to this
  return *this; }
```

Though this implementation looks reasonable at a first glance, a problem occurs if *this and rhs are the same object.

# Assignment to self

The traditional way to prevent this error is to check for assignment to self via an identity test:

```
ResourceHandler&
ResourceHandler::operator=
(const ResourceHandler& rhs) {
  // Handle self-assignment
  if (this == &rhs) return *this;
  // Stop using current resource
  delete pres;
  // Deep copy of rhs resource
  pres = new Resource(*rhs.pres);
  // Return a reference to this
  return *this; }
```

# Table of contents - Best practices

# RAII (Resource Allocation Is Initialization)

Consider the following code snippet:

```
class Resource{ /* ... */ };

void dummy_function() {
  Resource * pRes = new Resource;
  /* ... */
  delete pRes;
}
```

This looks fine, but there are several ways the function could fail to delete the **Resource** object. To avoid this inconvenience we need to put that resource inside an object devised to release it during destruction:

```
void dummy_function() {
  ResourceHandler pRes(new Resource);
  /* ... */
}
```

# RAII (Resource Allocation Is Initialization)

```
class ResourceHandler {
  Resource * pointer_

public:

  ResourceHandler(Resource * pointer)
  : pointer_(pointer) {};

  ~ResourceHandler() { delete pointer_; }
  /* ... */
}
```

RAII - When an object is used to manage a resource:

1. the constructor acquires immediately the resource
2. the destructor ensures the release of the resource

# The Rule of Three

Consider the following class that manages a resource:

```
class Person {
    char* name_;
    int   age_;
public:
    // the constructor acquires a resource:
    // dynamic memory obtained via new[]
    Person(const char* the_name, int the_age) {
        name_ = new char[strlen(the_name) + 1];
        strcpy(name, the_name);
        age_ = the_age;
    }
    // the destructor releases this resource
    ~Person() {
        delete[] name;
    }
};
```

# The Rule of Three

A class written in this way may have several unpleasant effects:

```
int main() {
  Person a("Giulio Cesare",62);
  Person b(a); // change in a <---> change in b
  { Person c("Napoleone Bonaparte",21);
    a = c //  dangling pointer + memory leak
  } }
```

Since memberwise copying does not behave correctly, we must define the copy constructor and the copy assignment operator explicitly:

```
// 1. Copy constructor
Person(const person& that) {
  name = new char[strlen(that.name) + 1];
  strcpy(name, that.name);
  age = that.age;
}
```

# The Rule of Three

```
// 2. Copy assignment operator
Person& operator=(const Person& that) {
 if ( this != &that ) {
   char* local_name = new char[strlen(that.name)+1];
   // If the above statement throws, the object
   // is still in the same state as before.
   strcpy(local_name, that.name);
   delete[] name_;
   name_ = local_name;
   age_  = that.age;
 } return *this; }
```

Rule of Three - If you need to explicitly declare either:

- the destructor
- the copy constructor
- the copy assignment operator

you probably need to explicitly declare all three of them.

# The Copy and Swap Idiom

Let's dwell a little more on the copy assignment operator:

```
Person& operator=(const Person& that) {
 if ( this != &that ) { // (1)
   char* local_name = new char[strlen(that.name)+1];
   strcpy(local_name, that.name); // (2)
   delete[] name_;
   name_ = local_name; // (2)
   age_  = that.age; // (2)
 }
 return *this;
}
```

This implementation suffers from at least 2 problems:

1. the self-assignment test is rarely needed, but always evaluated
2. part of the code is duplicated from the copy constructor

# The Copy and Swap Idiom

The copy-and-swap idiom is an elegant solution to write a copy assignment operator for classes that manage resources:

```
class Person {
    char* name_;
    int   age_;
public:
    /* ... */
   friend void swap(Person& first, Person& second) {
     using std::swap;
     swap(first.name_, second.name_);
     swap(first.age_ , second.age_ );
  }
  Person& operator=(Person other) {
    swap(*this, other);
    return *this;
  }
};
```

# Table of contents - Best practices

# Know when to return an object

Consider the following class:

```
class Rational {
public:
  Rational(int numerator = 0, int denominator = 1);
  /* ... */
private:
  int n,d;
};
```

Apparently, two different signatures may be used to implement **operator***:

```
Rational  operator*(const Rational& lhs,
                    const Rational& rhs);
Rational& operator*(const Rational& lhs,
                    const Rational& rhs);
```

# Know when to return an object

As there is no reason to expect that an object of type **Rational** exists prior to the call to **operator***, the only right way to return a new object from within a function is:

```
Rational operator*(const Rational& lhs,
                   const Rational& rhs) {
  return Rational(lhs.n*rhs.n,lhs.d*rhs.d);
}
```

Never return:

- a pointer or reference to a local stack object
- a reference to a heap-allocated object
- a pointer or reference to a local static object

if there is a chance that more than one such object will be needed

# The one slide summary of the lecture

## Classes basics

1. A class is a user-defined type that aggregates structure and behavior
2. Access specifiers may be used to enforce encapsulation
3. Constructor, destructor and assignment are special member functions
4. A class may overload operators or define custom conversions

## Best practices

1. Special member functions must be disallowed if you don't want them
2. Resource management is done following well-established idioms
3. Care must be taken to decide when to return by value or by reference

# Table of contents - Appendices

MEYERS, S.
*Effective C++: 55 Specific Ways to Improve Your Programs and Designs*, 3rd ed.
Addison-Wesley Professional, 2005.

SUTTER, H.
*Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*.
Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

SUTTER, H., AND ALEXANDRESCU, A.
*C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*, 1 ed.
Addison-Wesley Professional, Nov. 2004.