IP of the Max-Planck-Institute for Physics of Complex Systems (mpipks),
Nöthnitzer Straße 38, 01187 Dresden

# MPI PKS C++ Lecture

Bennet Becker

2019-09-09

MPI PKS

# Object Orientation

# Object Orientation

***Object-oriented programming (OOP)***

# Object Orientation

***Object-oriented programming (OOP)*** *is a programming paradigm*

# Object Orientation

***Object-oriented programming (OOP)*** *is a programming paradigm based on the concept of "objects", which may contain data*

# Object Orientation

***Object-oriented programming (OOP)*** *is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes;*

# Object Orientation

***Object-oriented programming (OOP)*** *is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code,*

# Object Orientation

***Object-oriented programming (OOP)*** *is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods.*

# Object Orientation

***Object-oriented programming (OOP)*** *is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods. A feature of objects is that an object's procedures can access and often modify the data fields of the object with which they are associated (objects have a notion of "this" or "self").*

# Object Orientation

***Object-oriented programming (OOP)*** *is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods. A feature of objects is that an object's procedures can access and often modify the data fields of the object with which they are associated (objects have a notion of "this" or "self"). In OOP, computer programs are designed by making them out of objects that interact with one another.*

# Object Orientation

***Object-oriented programming (OOP)*** *is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods. A feature of objects is that an object's procedures can access and often modify the data fields of the object with which they are associated (objects have a notion of "this" or "self"). In OOP, computer programs are designed by making them out of objects that interact with one another. There is significant diversity of OOP languages, but the most popular ones are class-based, meaning that objects are instances of classes, which typically also determine their type.*

# Classes

# Classes i

- Classes are an expanded concept of data structures
- like data structures, they can contain data members, but they can also contain functions as members
- An object is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.
- Classes are defined using either keyword class or keyword struct, with the following syntax:

# Classes ii

```
class class_name {
        access_specifier_1:
                member1;
        access_specifier_2:
                member2;
        ...
} object_names;
```

- Where `class_name` is a valid identifier for the class, `object_names` is an optional list of names for objects of this class.
- same format as plain data structures, except that they can also include functions and have these new things called access specifiers

# Access Modifiers

- 3 visibility scopes:
  - `private`, only accessible by the object it self
  - `protected`, only accessible by the object and it's children
  - `public`, publicly accessible by everyone via the object instance
- applied to both methods and attributes
- By default, all members of a class declared with the class keyword have private access for all its members. Therefore, any member that is declared before any other access specifier has private access automatically

# Example i

```
class Rectangle {
                int width, height;
        public:
                void set_values (int,int);
                int area (void);
} rect;
```

# Example ii

```
// classes example
#include <iostream>
using namespace std;

class Rectangle {
                int width, height;
        public:
                void set_values (int,int);
                int area() {return width*height;}
};

void Rectangle::set_values (int x, int y) {
        width = x;
        height = y;
}

int main () {
        Rectangle rect;
        rect.set_values (3,4);
        cout << "area: " << rect.area();
        return 0;
}
```

# Example iii

# Constructors i

- What would happen in the previous example if we called the member function `area` before having called `set_values` ?

# Constructors i

- What would happen in the previous example if we called the member function `area` before having called `set_values` ?
- An undetermined result, since the members width and height had never been assigned a value.

# Constructors ii

- to avoid that, a class can include a special function called its constructor, which is automatically called whenever a new object of this class is created, allowing the class to initialize member variables or allocate storage.
- This constructor function is declared just like a regular member function, but with a name that matches the class name and **without any return type; not even void**.

# Constructors iii

```
// example: class constructor
#include <iostream>
using namespace std;

class Rectangle {
                int width, height;
        public:
                Rectangle (int,int);
                int area () {return (width*height);}
};

Rectangle::Rectangle (int a, int b) {
        width = a;
        height = b;
}

int main () {
        Rectangle rect (3,4);
        Rectangle rectb (5,6);
        cout << "rect area: " << rect.area() << endl;
        cout << "rectb area: " << rectb.area() << endl;
        return 0;
}
```

# Overloading Constructors i

- Like any other function, a constructor can also be overloaded with different versions taking different parameters: with a different number of parameters and/or parameters of different types. The compiler will automatically call the one whose parameters match the arguments

# Overloading Constructors ii

```
// overloading class constructors
#include <iostream>
using namespace std;

class Rectangle {
                int width, height;
        public:
                Rectangle ();
                Rectangle (int,int);
                int area (void) {return (width*height);}
};

Rectangle::Rectangle () {
        width = 5;
        height = 5;
}

Rectangle::Rectangle (int a, int b) {
        width = a;
        height = b;
}

int main () {
        Rectangle rect (3,4);
        Rectangle rectb;
        cout << "rect area: " << rect.area() << endl;
        cout << "rectb area: " << rectb.area() << endl;
        return 0;
}
```

# Overloading Constructors iii

- But this example also introduces a special kind constructor: the default constructor.
- default constructor is the constructor that takes no parameters, and it is special because it is called when an object is declared but is not initialized with any arguments.
- But:

```
Rectangle rectb;   // ok, default constructor called
Rectangle rectc(); // oops, default constructor NOT called
```

# Member initialization in constructors i

- When a constructor is used to initialize other members, these other members can be initialized directly, without resorting to statements in its body.
- This is done by inserting, before the constructor's body, a colon (:) and a list of initializations for class members

```
Rectangle::Rectangle (int x, int y) : width{x}, height{y} { }
```

- For members of fundamental types, it makes no difference, because they are not initialized by default
- but for member objects (those whose type is a class), if they are not initialized after the colon, they are default-constructed.

# Member initialization in constructors ii

```
// member initialization
#include <iostream>
using namespace std;

class Circle {
                double radius;
        public:
                Circle(double r) : radius(r) { }
                double area() {return radius*radius*3.14159265;}
};

class Cylinder {
                Circle base;
                double height;
        public:
                Cylinder(double r, double h) : base (r), height(h) {}
                double volume() {return base.area() * height;}
};

int main () {
        Cylinder foo (10,20);

        cout << "foo's volume: " << foo.volume() << '\n';
        return 0;
}
```

# Class members

## Association

- classes can have relationships to each other
- most common "has-a" relationship
  - Aggregation. Objects can exist without each other
  - Composition. Objects can not exist without each other

# Pointers to Classes i

- Objects can also be pointed to by pointers: Once declared, a class becomes a valid type, so it can be used as the type pointed to by a pointer `Rectangle * prect;`
- Similarly as with plain data structures, the members of an object can be accessed directly from a pointer by using the arrow operator (->).

# Pointers to Classes ii

```
// pointer to classes example
#include <iostream>
using namespace std;

class Rectangle {
                int width, height;
        public:
                Rectangle(int x, int y) : width(x), height(y) {}
                int area(void) { return width * height; }
};


int main() {
        Rectangle obj (3, 4);
        Rectangle * foo, * bar, * baz;
        foo = &obj;
        bar = new Rectangle (5, 6);
        baz = new Rectangle[2] { {2,5}, {3,6} };
        cout << "obj's area: " << obj.area() << '\n';
        cout << "*foo's area: " << foo->area() << '\n';
        cout << "*bar's area: " << bar->area() << '\n';
        cout << "baz[0]'s area:" << baz[0].area() << '\n';
        cout << "baz[1]'s area:" << baz[1].area() << '\n';
        delete bar;
        delete[] baz;
        return 0;
}
```

# Operator with Objects

- *x pointed to by x
- &x address of x
- x.y member y of object x
- x->y member y of object pointed to by x
- (*x).y member y of object pointed to by x (equivalent to the previous one)
- x[0] first object pointed to by x
- x[1] second object pointed to by x
- x[n] (n+1)th object pointed to by x

# Classes defined with struct and union

- Classes can be defined not only with keyword class, but also with keywords struct and union.
- The keyword struct, generally used to declare plain data structures, can also be used to declare classes that have member functions, with the same syntax as with keyword class.
- only difference : members of classes declared with the keyword struct have public access by default, while members of classes declared with the keyword class have private access by default
- the concept of unions is different from that of classes declared with struct and class - unions only store one data member at a time - but nevertheless they are also classes and can thus also hold member functions. The default access in union classes is public.

# Operator Overloading

# Overloading Operators

- Classes, essentially, define new types to be used in C++ code
- types in C++ not only interact with code by means of constructions and assignments
- They also interact by means of operators
- meaning of Operators with fundamental types is obvious. But what about objects of classes and structs

```
struct myclass {
        string product;
        float price;
} a, b, c;
a = b + c;
```

# Overloadable Operators

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

# Overloading Operators

- Operators are overloaded by means of operator functions
- regular functions with special names: their name begins by the operator keyword followed by the operator sign that is overloaded.

```
type operator sign (parameters) { /*... body ...*/ }
```

# Example

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

class CVector {
        public:
                int x,y;
                CVector () {};
                CVector (int a,int b) : x(a), y(b) {}
                CVector operator + (const CVector&);
};

CVector CVector::operator+ (const CVector& param) {
        CVector temp;
        temp.x = x + param.x;
        temp.y = y + param.y;
        return temp;
}

int main () {
        CVector foo (3,1);
        CVector bar (1,2);
        CVector result;
        result = foo + bar;
        cout << result.x << ',' << result.y << '\n';
        return 0;
}
```

# Example

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

class Rectangle {
                int width, height;
        public:
                Rectangle(int x, int y) : width(x), height(y) {}
                Rectangle(const Rectangle& other) : width(other.width), height(other.height){};
                int area(void) { return width * height; }
                Rectangle operator+ (const Rectangle&);
};

Rectangle Rectangle::operator+ (const Rectangle& other){
        Rectangle a(*this);
        a.width += other.width;
        a.height += other.height;
        return a;
}

int main() {
        Rectangle a (3, 4);
        Rectangle b (5, 6);
        auto c = a + b;

        cout << "a's area: " << a.area() << '\n';
        cout << "b's area: " << b.area() << '\n';
        cout << "c's area: " << c.area() << '\n';

        return 0;
}
```

# Operator Overloading

- operator overloads are just regular functions which can have any behavior
- there is actually no requirement that the operation performed by that overload bears a relation to the mathematical or usual meaning of the operator, although it is strongly recommended
- parameters depend on operator. for binary operators like + e.g. the right hand side of the operation is required
  - `@a` `A::operator@()`
  - `a@` `A::operator@(int)`
  - `a@b` `A::operator@(B)`
  - `a(b,c...)` `A::operator()(B,C...)`
  - `a->b` `A::operator->()`
  - `(TYPE) a` `A::operator TYPE()`

# Operator Overloading outside the class

- usually non-modifying operators can also be overloaded outside the class

```
CVector operator+ (const CVector& lhs, const CVector&
↪  rhs) {
        CVector temp;
        temp.x = lhs.x + rhs.x;
        temp.y = lhs.y + rhs.y;
        return temp;
}
```

# this

# Keyword `this`

- represents a pointer to the object whose member function is being executed
- used within a class's member function to refer to the object itself

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

class Dummy {
        public:
                bool isitme (Dummy& param){
                        return (&param == this);
                }
};

int main () {
        Dummy a;
        Dummy* b = &a;
        if ( b->isitme(a) )
                cout << "yes, &a is b\n";
        return 0;
}
```

# static and const members

# static members i

- static data member of a class is also known as a "class variable"
- only one common variable for all the objects of that same class, sharing the same value
- or in other words: a static member can exsist without an object

# static members ii

```
// static members in classes
#include <iostream>
using namespace std;

class Dummy {
        public:
        static int n;
        Dummy () { n++; };
};

int Dummy::n=0;

int main () {
        Dummy a;
        Dummy b[5];
        cout << a.n << '\n';
        Dummy * c = new Dummy;
        cout << Dummy::n << '\n';
        delete c;
        return 0;
}
```

# static members iii

- static members have the same properties as non-member variables but they enjoy class scope
- For that reason, and to avoid them to be declared several times, they cannot be initialized directly in the class, but need to be initialized somewhere outside it.
- Classes can also have static member functions.
- represent the same: members of a class, common to all object of that class, acting exactly as non-member functions but being accessed like members
- Because they are like non-member functions, they cannot access non-static members of the class (neither member variables nor member functions). They neither can use the keyword this.

# const members i

- const Objects `const MyClass myobject;` are restricted to access data members read-only

```
int main() {
        const MyClass foo(10);
        // foo.x = 20;            // not valid: x cannot
        ↪  be modified
        cout << foo.x << '\n';  // ok: data member x can
        ↪  be read
        return 0;
}
```

# const members ii

- member functions of a const object can only be called if they are themselves specified as const members

```
class MyClass {
        public:
                int x;
                MyClass(int val) : x(val) {}
                int get() {return x;}
};
```

# const members iii

- To specify that a member is a const member, the const keyword shall follow the function prototype, after the closing parenthesis for its parameters:
  ```
  int get() const {return x;}
  ```
- Member functions specified to be const cannot modify non-static data members nor call other non-const member functions. In essence, const members shall not modify the state of an object.

# uncommon edge-case?

- Most functions taking classes as parameters actually take them by const reference, and thus, these functions can only access their const members

```
// const objects
#include <iostream>
using namespace std;

class MyClass {
          int x;
        public:
          MyClass(int val) : x(val) {}
          const int& get() const {return x;}
};

void print (const MyClass& arg) {
        cout << arg.get() << '\n';
}

int main() {
        MyClass foo (10);
        print(foo);

        return 0;
}
```

# Class Templates

# Class Templates i

- Just like with function, classes can be templated too

```
template <class T>
class mypair {
        T values [2];
        public:
        mypair (T first, T second)
        {
                values[0]=first; values[1]=second;
        }
};
```

# Class Templates ii

```
// class templates
#include <iostream>
using namespace std;

template <class T>
class mypair {
        T a, b;
        public:
        mypair (T first, T second)
        {a=first; b=second;}
        T getmax ();
};

template <class T>
T mypair<T>::getmax ()
{
        T retval;
        retval = a>b? a : b;
        return retval;
}

int main () {
        mypair <int> myobject (100, 75);
        cout << myobject.getmax();
        return 0;
}
```

# Special Members

# Default Constructor

- as already seen: constructor called when objects of a class are declared, but are not initialized with any arguments
- If a class definition has no constructors, the compiler assumes the class to have an implicitly defined default constructor
- But as soon as a class has some constructor taking any number of parameters explicitly declared, the compiler no longer provides an implicit default constructor

# Destructor i

- opposite functionality of constructors
- responsible for the necessary cleanup needed by a class when its lifetime ends
  - free pointers
  - delete objects
  - ...

# Destructor ii

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

class Example4 {
        string* ptr;
        public:
        // constructors:
        Example4() : ptr(new string) {}
        Example4 (const string& str) : ptr(new string(str)) {}
        // destructor:
        ~Example4 () {delete ptr;}
        // access content:
        const string& content() const {return *ptr;}
};

int main () {
        Example4 foo;
        Example4 bar ("Example");

        cout << "bar's content: " << bar.content() << '\n';
        return 0;
}
```

# Destructor iii

# Copy Constructor i

- When an object is passed a named object of its own type as argument, its copy constructor is invoked in order to construct a copy.

```
MyClass::MyClass (const MyClass&);
```

- If a class has no custom copy nor move constructors (or assignments) defined, an implicit copy constructor is provided

# Copy Constructor ii

```
// copy constructor: deep copy
#include <iostream>
#include <string>
using namespace std;

class Example5 {
        string* ptr;
        public:
        Example5 (const string& str) : ptr(new string(str)) {}
        ~Example5 () {delete ptr;}
        // copy constructor:
        Example5 (const Example5& x) : ptr(new string(x.content())) {}
        // access content:
        const string& content() const {return *ptr;}
};

int main () {
        Example5 foo ("Example");
        Example5 bar = foo;

        cout << "bar's content: " << bar.content() << '\n';
        return 0;
}
```

# Copy Constructor iii

# Copy assignment

content...