# Programming Language Concepts

## Object Oriented Prog: Objects

Onur Tolga Şehitoğlu

Bilgisayar Mühendisliği

CC BY SA

# Outline

# Object Oriented Programming

- Abstraction
- Encapsulation
- Hiding
- Inheritance

# Encapsulation/Scope

- Objects consist of:
  - attributes (member variables)
  - methods (member functions)

  encapsulated in a package scope
  - attributes: state of objects
  - methods: behaviour of objects
  - alternative terminology: messages
    call a method $\equiv$ send message to an object
  - A class is the family for similar objects.
  - An object is an instance of a class.

| **Person** |
|---|
| name<br>surname<br>no |
| + getname()<br>+ setno() |

```
class Person {
    char name[40], surname[40];
    int no;
public:
    const char * getname() { return name;}
    void setno(int);
} obj ;

void Person::setno(int a) {
    no=a;
}
```

- C++ allows definitions inside the class or outside by scope operator ‘::’
- Environment is recursive collateral.
- `obj.getname();` calls the method in the context of object `obj`.
- `this` keyword denotes pointer to current object in member functions. ( `self()` in some other languages)

# Hiding

- Interface vs detail. Details are hidden, only interface members are exported outside.
- C++ uses `private:`, `protected:`, and `public:` labels to mark hiding.
- only members following a `public:` label are visible outside (the object for example). Member functions can access all members regardless of their labels.
- `obj.setno(4)` is legal, `obj.no` is not.
- Hiding depends on scope and it is lexical. In C++ pointer conversions can violate hiding.
- By convention all member variables should be private, some member functions can be private, only some of member functions are public.
- `protected` keyword is useful with inheritance.

# Abstraction

- An object is an abstraction over the programming entity defined by the model in the design.
- Model: customer, bank, registration, course, advisor, mail, chatroom,...
- Class should provide:
  - Transparent behaviour for the objects, access via interface functions.
  - Data integrity. Objects should be valid through their lifetimes.
- Data integrity at the beginning of lifetime provided by constructors (+destructors in C++)

# Constructors

- Special member functions called when lifetime of the object starts just after storage of members are ready
- Automatically called. No explicit calls.
- no return value, name is same with the class
- can be overloaded

```
class Person {
    char *name[40], *surname[40];
    int no;
public:
    Person(const char *n, const char *s) {
        strcpy(name,n) ; strcpy(surname,s) ; no=0;
    }
    Person() { name[0]=0; surname[0]=0; no=0;}
} obj ;
```

- Constructors can be overloaded

| Definition | Constructor |
|---|---|
| `Person a ;` | `Person()` |
| `Person a("ali","veli");` | `Person(const char *, const char *)` |
| `Number a=3;` | `Number(int)` |
| `Number a(3);` | `Number(int)` |
| `Number b=a;` | `Number(Number &a)` |
| `Number a[2]={0,1}` | `Number(int)` |

- If no constructor implemented, empty constructor (do nothing) assumed
- If at least one constructor exists, variables should match at least one of them, no empty constructor assumed
- Constructors are called by the language when lifetime started:
  1. start of program for global objects
  2. entrance to function for local objects
  3. when heap objects are created (with new)

# Heap Objects

- `new` and `delete` operators instead of `malloc()` and `free()`. Why?
- ```
  Person *p=new Person("ali","veli");
  delete p;
  ```
- Array allocation/deallocation:
  ```
  Person *p=new Person[100];
  delete [] p;
  ```
  ```

# Destructors

- When storage (members) of an object allocated dynamically
- Lifetime is over : garbage
- We need calls to collect heap variables within the object
- Java solution: garbage collector does the job. We need nothing
- C++: destructors: member functions called when lifetime is over.
- A class only have one destructor with exact type and name: `~ClassName()`. Called:
  1. end of program for global objects
  2. return from function for local objects
  3. when heap objects are deallocated (with `delete`)

# Constructor Calls

- Calling a constructor as a member function is not allowed:
  `Person p ; p.Person("john");` is a compiler error
- Constructor definitions can call eachother with special syntax:
  `Person():Person("john␣doe") { ... }` (C++11 only)
- Explicit constructor calls create a temporary object:
  `Person p; p = Person("john");` is equivalent to:
  `Person p; { Person tmp("john"); p = tmp; }`
  note that lifetime of `tmp` is over at the end of line
- In definitions, no intermediate object created:
  `Person p = Person("john");`
- A constructor also works as a type convertion operator:
  `p = (Person) "john";` is equivalent to `p = Person("john")`, creates a temporary object
- Also type conversion works implicitly. C++ calls constructors when type conversion is required:
  `p = "john";` is equivalent to the call above.

# const Keyword

- C++ does strict type checking on constant restriction on `const`
- `const char *p` VS `char *const q`
  1. `p[3]='a';` ×
  2. `q[3]='a';` √
  3. `p++;` √
  4. `q++;` ×
- `const char * const p`
- `f(const ClassName &a)` makes the parameter object constant during the function scope
- `const ClassName &f()` makes the returned object reference constant in expression containing the function call
- What's beside assignment? constant member functions

# Constant Member Functions

- `void f(const Rational &a) { ...; a.clear(3);...;a.out();}`

  `void Rational::clear() { a=b=0;}`

  What is wrong above?
- `void Rational::out() const {...; a=b=0; }`

  `const` keyword preceding the function body makes member function a constant function.
- Constant functions cannot update member variables, only can inspect them

  `a=b=0` in `out()` is invalid above
- If an object is constant, only constant member functions can be called.

  `a.clear(3);` is invalid above
- Type system of C++ prohibits those → Syntax error.

# Copy Constructor

- Destructor does not solve all problems with objects with heap members:
  - Semantics of assignment
  - Semantics of parameter passing
  - Semantics of return value
  - Initialization
- Default behaviour of C++ is copy member values byte by byte.
- Java assigns/passes by reference. No copying.
- C++ Solution: implement your own semantic by Copy constructor and overloading assignment operator.
- Assignment operator destroys an existing object and replaces with the data from new one, copy constructor copies data into an empty object.

# Copy Constructor

- Type is: `ClassName(const ClassName &)`
- Called when:
  - Object passed by value: `void add(ClassName a) {...}`
  - Object initialized by object: `ClassName a,b=a;`
  - Object returned as a value `ClassName getVal() {...}`
- Last one is a little tricky.
- Default behaviour exists even if it other constructors exist.

```
class List {
        struct Node { int x; Node *next} *head;
public: List() { head=NULL;}
        List(cons List &); // Copy constructor
        ~List();
};
void passbyvalue(List a) {          Copy Constructor
...
}
List returnasvalue(List &a) {
    List b = a;  <-- Copy Constructor, explicit
...
    return a;
}
...
passbyvalue(c);
...
d=returnasvalue(c)  Copy Constructor
...
```

- Pass by value of objects are constructed by the copy constructor
- Return an object as a value creates a temporary object in place of return and uses it:

  `d=returnasvalue(c);` $\equiv$ `{List tmp=returnasvalue(c); d=tmp; }`
- Temporary objects are created at such expressions and deallocated at the end of the line (at ';'), destructors are called regularly.
- Explicit call to a constructor also creates such a temporary object.

  ```
  g=Person("ali","veli");
  ```
- C++ compilers avoid copy constructor calls when possible, called copy elision.

  ```
  List f() { List t;...; return t;} ... ; d=f(); ...
  ```

  If possible, compiler binds local object and returned temporary object same storage → No constructor call.

# Pass by reference

- `Person &a` denotes a reference type and implements pass by reference. No new object is created for parameter.
- Can be used in declaration to create an alias:
  `Person & q = p;` (subject to lifetime of `p`)
  `const Person &t = Person("john");` (temporary read only object with a longer lifetime)
- Returning reference type is also possible:
  `int & Person::get() { ...}`
  `p.get()++;`
- `const` references follow the semantics of `const`.
- Temp. objects, r-values cannot be passed by non-const references:
  `void print(Person &p) { ...}`
  `print( (Person) "marry");` is an error.
- r-values can be passed by constant references:
  `void print(const Person &p) { ...}`
  `print( (Person) "marry");` is ok.

# Effiency of Parameter Passing

- Pass by reference is efficient but modification of actual parameter is not always desired
- `const` references avoid modification of actual parameters and may get r-values however paremeter object cannot be modified.
- Copy constructor and pass by value solves modifiable parameter object and r-value problem.
- Copying is expensive and r-values, temporary objects have a very short lifetime.
- One solution is stealing the resources of an object instead of copying.
- An r-value has a short lifetime, so stealing its resources would not harm the integrity.
- C++11 defined rvalue references, move constructor and assignment move operator to solve this problem.

# Move Constructor

- C++11 introduced the following:
  1. rvalue reference: `Person &&p`
  2. move constructor: `Person(Person &&p)`
  3. assignment move: `operator=(Person &&p)`
  4. `T &&std::move(T &)`, a reference converter.
- rvalue references are created for temporary objects and by making explicit `std::move()` calls.
- move constructor and all functions getting rvalue references (including assignment) gets resources directly from parameter object and leaves the paremeter in a valid but nullified state.
- move constructor does not allocate and copy values. Just get references and pointers. Therefore they are more efficient.
- After call, parameter will not contain its previous values but a minimum valid state.

- move constructor is bound to return as a value cases (copy cons. called if it does not exist)
- Similarly passing temporary objects to assignment or rvalue parameters will be more efficient.
- Programmers may convert lvalue references to rvalue references explicitly by calling `std::move()` if the objec does not need its content afterwards.
- Move constructor is subject to copy elision. Compiler avoids calling it if possible.
- Destructor is called for the moved rvalue when its lifetime is over. So it should be left in a state without dangling references, double free problems etc.

```
class LList {
    struct Node { int a; Node *next; } *head;
public:
    LList() { head = nullptr;}
    LList(const LList &l) { // copy constructor
        Node **prev = &head;
        for (Node *p = l.head; p != nullptr; p = p->next) {
            Node *q = new Node;
            q->a = p-> a; (*prev) = q; prev = &(q->next);
        }
        *prev = nullptr;
    }
    LList(LList &&l) {  // move constructor
        head = l.head; l.head = nullptr; // steal and nullify
    }
    ~LList() {  /* a decent llist destructor here */}
};
LList series(int n) { // assume no copy elision!
    LList t;
    for (int i = n; i >= 0 ; i--)
        t.push(i);
    return t;
}
...
series(10).out(); // picks MOVE (or none if copy elision)
```

```
LList LList::operator=(LList &&l) {   // move assignment
    for (Node *p = head; p != nullptr;) {
        Node *q = p;           // deallocate current list
        p = p->next; delete q;
    }
    head = l.head; l.head = nullptr; // steal and nullify
}

...
// first move constructor (if no elision) then assignment
// above is called. But it is the same linked list
// which is local to series() is assigned to c
c = series(10);
```

# Operator Overloading

- Not an essential feature of object oriented programming but improves readability in some cases.
- Especially usefull in implementing selector abstraction, algebra based applications.
- Do not use it when the operator is not intuitive for the context (class and the operation).
- C++ allows overloading of existing operators with same arity and precedence and only if at least one class type involves in the operator
- Operator can be implemented as a member function (first parameter is the class) or as an external function (which has at least one parameter being a class)

- All C++ operators except '`.`' , '`?:`', '`::`', '`.*`' and '`->*`'
- For unary operators:
  1. `void ClassName::operator++();`
  2. `void operator++(ClassName &a);`
- For binary operators:
  1. `void ClassName::operator&&(int a);`
  2. `void operator&&(int a,ClassName &b);`
- First versions are member functions, can exist private members. Only operand in unary case, LHS in binary case is the current object
- Second versions are outside of the definition. You need `friend` declaration if they need to access private members.

```
Rational & Rational::operator+(Rational &b) {...}
Rational & Rational::operator+(int n) {...}
Rational & Rational::operator<(Rational &b) {...}
Rational & Rational::operator!() {...}
Rational & Rational::operator++() {...}
Rational & Rational::operator++(int nouse) {...}
Rational & Rational::operator double() {...}

void Hash::operator=(Hash &a) {...}
double Hash::operator[](int a) {...}
double Hash::operator[](const char a[]) {...}
Hash & Hash::operator()(const char a[]) {...}

double Pointer::operator*() {...}
void * Pointer::new(size_t size) {...}
void * Pointer::delete(void *p, size_t size) {...}

Rational a,b,c; Hash h,j; Pointer p,*q;
a+b;        a+3;      if (a<b) ... ;      !a;
++a;        a++;      x=(double)a;

h=j;        x=h[3];   x=h["ali"];         i=h("a-b");

x=*p;       q=new Pointer;           delete q;
```

```
int operator+(int a, Rational &b) {...}
Rational & operator++(Rational &b) {...}
ostream  & operator<<(ostream &os, Rational &a) {...}
istream  & operator>>(istream &os, Rational &a) {...}

void operator+=(Hash &a,Rational b) {...}

Rational a,b; Hash h,j;

i=i+a;
++a;
cout << a; cout << 3 << a << b ;
cin >> b;
h+=a;
```

# Friends

- When an external function or class needs to access private members, friend declaration is used to grant access.

```
class Rational {
    friend class Hash;
    friend ostream & operator<<(ostream &,const Rational &);
    int a,b;
public: ...
};
class Hash {
    ...
   void operator+=(Rational &a) { .. a.a; .. a.b; ...}
};
ostream & operator<<(ostream &os,const Rational &a) {
    os << a.a << "//" << a.b << '\n';
    return os;
}
```

# Implementation of Objects

class Person

| | |
|---|---|
| `char name[40]` | 40*sizeof(char) |
| `int id` | sizeof(int) |
| ~~`char * getname()`~~ | ~~sizeof(char *(*)())~~ |
| ~~`void print()`~~ | ~~sizeof(void (*)())~~ |

- What is size of object? Size of member variables + sizeof member function pointers?
- No! Each object does not have to store the function information.
  Its storage is same with the structure without any member functions.
- Function membership handled by the type system:
  `Person::getname()` instead of `getname()`

- How functions get object context (which object they refer to?)?
- `Person::getname(Person *this)` instead of no parameters
- `Person a; a.getname();`
  converted to `Person::getname(&a);` internally
- All member references inside member function are converted to:

```
char *getname() {.. id=5; ... ; strlen(name);...} →
char *Person::getname(Person *this) {
.. this->id=5; ... ; strlen(this->name);...}
```