Programming Language Concepts

Object Oriented Prog: Objects

Onur Tolga Şehitoğlu

Bilgisayar Mühendisliği

Outline

Object Oriented Programming

- Abstraction
- Encapsulation
- Hiding
- Inheritance

Encapsulation/Scope

Person
name surname no
+ getname() + setno()

- 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.

```
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`)

- 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) {
...
}
List returnasvalue(List &a) {
    List b = a;
...
    return a;
}
...
passbyvalue(c);
...
d=returnasvalue(c)
...
```

Copy Constructor

Copy Constructor, explicit

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);` ≡ `{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++ optimizer avoids copy constructor calls when possible.

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

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.

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);...}
```