

Programming Language Concepts

Encapsulation

Onur Tolga Şehitoğlu

Bilgisayar Mühendisliği



Outline

- 1 Encapsulation
- 2 Packages
- 3 Hiding
- 4 Abstract Data Types
- 5 Class and Object
 - Object
 - Class
- 6 Closure

Encapsulation

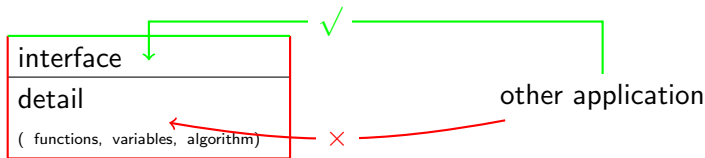
Managing the complexity → Re-usable code and abstraction.

Example:

| | |
|---------------|--|
| 50 lines | no abstraction is essential, all in main() |
| 500 lines | function/procedure abstraction sufficient |
| 5,000 lines | function groups forming modules, modules are combined to form the application |
| 500,000 lines | heavy abstraction and modularization, all parts designed for reuse (libraries, components etc) |

Modularization and Encapsulation

- Building an independent and self complete set of function and variable declarations ([Packaging](#))
- Restricting access to this set only via a set of interface function and variables. ([Hiding and Encapsulation](#))



Advantages of Encapsulation

- High volume details reduced to interface definitions ([Ease of development/maintenance](#))
- Many different applications use the same module via the same interface ([Code re-usability](#))
- Lego like development of code with building blocks ([Ease of development/maintenance](#))
- Even details change, applications do not change (as long as interface is kept same) ([Ease of development/maintenance](#))
- Module can be used in following projects ([Code re-usability](#))

- A group of declarations put into a single body.
- C has indirect way of packaging per source file. Python defines modules per source file.
- C++

```
namespace Trig {  
    const double pi=3.14159265358979;  
    double sin(double x) { ... }  
    double cos(double x) { ... }  
    double tan(double x) { ... }  
    double atan(double x) { ... }  
    ...  
};
```

- `Trig::sin(Trig::pi/2+x)+Trig::cos(x)`
- C++: (`::`) Scope operator.
- Identifier overlap is avoided. `List::insert(...)` and `Tree::insert(...)` no name collisions.

Hiding

- A group of functions and variables hidden inside. The others are interface. Abstraction inside of a package:

```
double taylorseries(double);
double sin(double x);
double pi=3.14159265358979;
double randomseed;
double cos(double x);
double errorcorrect(double x);
```

```
{-- only sin, pi and cos are accessible --}
module Trig(sin,pi,cos) where
  taylorseries x = ...
  sin x = ...
  pi=3.14159265358979
  randomseed= ...
  cos x = ...
  errorcorrect x = ...
```

Abstract data types

- Internals of the datatype is hidden and only interface functions provide the access.
- Example: rational numbers: $3/4$, $2/5$, $19/3$

```
data Rational = Rat (Integer,Integer)
x = Rat (3,4)
add (Rat(a,b)) (Rat(c,d)) = Rat (a*d+b*c,b*d)
```

 - 1 Invalid value? `Rat (3,0)`
 - 2 Multiple representations of the same value?
`Rat (2,4) = Rat (1,2) = Rat(3,6)`
- Solution: avoid arbitrary values by the user.

Main purpose of abstract data types is to use them transparently (as if they were built-in) without losing **data integrity**.

```
module Rational(Rational, rat, add, subtract, multiply, divide) where
  data Rational = Rat (Integer, Integer)
  rat (x,y) = simplify (Rat(x,y))
  add (Rat(a,b)) (Rat(c,d)) = rat (a*d+b*c, b*d)
  subtract (Rat(a,b)) (Rat(c,d)) = rat (a*d-b*c, b*d)
  multiply (Rat(a,b)) (Rat(c,d)) = rat (a*c, b*d)
  divide (Rat(a,b)) (Rat(c,d)) = rat (a*d, b*c)
  gcd x y = if (x==0) then y
             else if (y==0) then x
             else if (x<y) then gcd x (y-x)
             else gcd y (x-y)
  simplify (Rat(x,y)) = if y==0 then error "invalid value"
                        else let a=gcd x y
                              in Rat(div x a, div y a)
```

Initial value? We need **constructor** function/values. (remember we don't have the data definition)

rat (x,y) instead of Rat (x,y)

Object

- Packages containing hidden variables and access is restricted to interface functions.
- Variables with state
- Data integrity and abstraction provided by the interface functions.
- Entities in software can be modelled in terms of functions (server, customer record, document content, etc). Object oriented design.
- Example (invalid syntax! imaginary C++)

```
namespace Counter {  
private:    int counter=0;  
public:    int get() { return counter;}  
public:    void increment() { counter++; }  
};  
Counter::get()           Counter::increment()
```

Class

- The set of same typed objects form a **class**
- An object is an **instance** of the class that it belongs to (a counter type instead of a single counter)
- Classes have similar purposes to abstract data types
- Some languages allows both objects and classes
- C++ class declaration (valid syntax):

```
class Counter {  
private:    int counter;  
public:    Counter() { counter=0; }  
           int get() { return counter;}  
           void increment() { counter++; }  
} men, vehicles;  
men.increment(); vehicles.increment();  
men.get(); vehicles.get();
```

Abstract data type

interface (constructor, functions)

detail (**data type definition**, auxiliary functions)

Object

interface (constructor, functions)

detail (**variables**, auxiliary functions)

Purpose

- preserving data integrity,
- abstraction,
- re-usable codes.

Closure

- **Closure** is an abstraction method using the saved environment state in a scope.
- When a function returns a local object or function as its result and language keeps the environment state along with the returned value, it becomes a **closure**

```
def newid():
    c = 0 # this is the hidden variable in the environment
    def incget():
        nonlocal c #python 3, binds c above
        c += 1
        return c
    return incget

>>> a = newid()
>>> b = newid()
>>> a()
1
>>> b()
1
>>> b()
2
```

- Local variables of closures stay alive after call, as long as returned value is alive.
- **closures** can be used for generating new functions as in higher order functions:

```
def mult(a):
    def nested(b):
        return a*b
    return nested    # a different behaviour for each a value
twice = mult(2)
tentimes = mult(10)
a=twice(4)+tentimes(50)
```

- Also can be used for prototyping objects. Javascript example:

```
function counter() {
    var c = 0    // this is jailed in local environment, hidden
    var newObj = {} // create a new empty object
    newObj.incr = function () { c++; }
    newObj.get = function () { return c; }
    return newObj
}
a = counter()
b = counter()
a.incr()
a.get()
b.get()
```

C++ 2011 Closures

- C++ 2011 implements closures in lambda expressions by adding a set of captured variables within `[]`. This copy or get reference of auto variables in the environment in an object.
- However C++ closures do not extend lifetime of captured variables. After exit, the behaviour is undetermined.
- `[a,&b] (int x) { return a+x+b; }` captures `a` and `b` from current environment, `a` is by copy, `b` by reference.

```
std::function<int(int)> multiply(int a) {
    return [&] (int b) { return a*b; }; // capture by value
};
std::function<int()> cid() {
    int c = 0;
    return [=] () mutable { return ++c; }; // capture by copy
};
int main() {
    std::function<int(int)> twice = multiply(2);
    std::function<int(int)> three = multiply(3);

    cout << twice(12) << '\n' << three(34) << endl;

    auto c1 = cid();
    auto c2 = cid();

    cout << c1() << '\n' << c2() << endl;
    c1(); c1(); c1();
    cout << c1() << '\n' << c2() << endl;
    return 0;
}
```

Further Reusability

- Class relations. Extending one class definition to create more specific class definitions.
- Classes containing other classes
- Classes derived from other classes: [inheritance](#)
- Abstract classes and [interfaces](#)
- Polymorphism
- [Design patterns](#): standard object oriented designs applicable to a family of similar software problems. Not included in this course.