

C++

and Object Oriented Programming (OOP)

New in C++

General	OOP
<ul style="list-style-type: none">• input / output• Strings, booleans• function calls (by reference)• Templates• STL (standard template library)<ul style="list-style-type: none">• Containers (vectors, maps etc.)• Algorithms (sort, find etc.)• memory allocation (malloc vs new) <p>Threads (Parallel programming)</p> <p>Exceptions (Error catching)</p>	<p>Classes and Objects</p> <ul style="list-style-type: none">• Encapsulation• Inheritance• Polymorphism <p>RAII (Memory leaks)</p> <p>Iterators</p> <p>Smart pointers</p>

Console I/O

cin >>
cout <<
setw
setprecision
setbase

```
1  #include <iostream>
2  #include <iomanip>
3
4  using namespace std;
5
6  int main(int argc, char** argv) {
7      // OUTPUT
8      cout<<setw(10)<<"ten"<<setfill('-')<<setw(10)<<"four"<<setw(6)<<"four"<<"stop";
9      cout<<"hello"<<endl;
10     cout<<left<<setw(10)<<42<<endl;
11     int pr = cout.precision();
12     cout << setprecision(3) << 2.71828 << endl;
13     cout << 3.141596<<endl;
14     cout << 0.00042123 << " " << 0.0000042123 << endl;
15     cout <<setbase(16)<<42<<" "<<123456<<endl;
16     cout << 10+20 << endl;
17     cout << 18.45268 << "\n" << setprecision(pr) << 18.45268 << endl;
18     cout<< pr << endl;
19     cout << setprecision(3) << 1.0/3.0 << endl;
20     cout << setprecision(20) << 1.0/3.0 << endl;
21     cout << setprecision(20) << 1.0/10.0 << endl;
22     cout << 42 << endl;
23     // INPUT
24     int x, y, z;
25     cout <<"Enter 3 integers: \n";
26     cin >> x >> y >> z;
27     cout << "Mean: " << (x+y+z)/3.0 << endl;
28     return 0;
29 }
30
```

File I/O

Input from file
f >>

Output to file
f <<

```
1  #include <iostream>
2  #include <iomanip>
3  #include <fstream>
4
5
6  using namespace std;
7
8  int main() {
9      ofstream myfile("out.txt");
10
11     myfile << "hello" << 42 << endl;
12
13     for (int i=0; i<10; i++){
14         myfile << setw(12-i) << i << endl;
15     }
16
17     ifstream f("test.txt");
18     int value = 5;
19     f >> value;
20     cout << value << endl;
21     while(f >> value){
22         cout << value << endl;
23     }
24
25     return 0;
26 }
27
```

New types: String, Boolean

```
1  #include <iostream>
2  #include <string>
3
4  using namespace std;
5
6  int main(int argc, char** argv) {
7      string a("Hello");
8      string b = "World";
9      string c = a + b;
10     cout << c << endl;
11     cout << a << " " << b << "\t" << 42 << endl;
12     char d[] = "old C style.";
13     c = d;
14     string e(d);
15     cout << d << e << endl;
16
17     bool k = 3 < 1;
18     cout << k << (13 > 2) << true << false << 2 + true << endl;
19     if(k) cout << "Correct";
20     else cout << "Wrong";
21
22     cout << endl;
23     a = "test"; b = "test";
24     char f[] = "old C style.";
25     cout << (d==f) << endl;
26     cout << (a==b) << endl;
27     return 0;
28 }
```

Swap algorithm

- `temp = a`
- `a = b`
- `b = temp`

Hands on: Write a function `[void exchange(int,int)]` that accepts two integers and swaps their values. In `main()` declare `int x=5, y =3`. Call `exchange(x,y)` and then `cout x` and `y`, to check that `x=3` and `y=5`.

[Don't use `swap(x, y)` - `swap` is the name of a function of the standard library]

References

A reference is an alias to a variable.

```
int a = 5;
```

```
int &x = a;
```

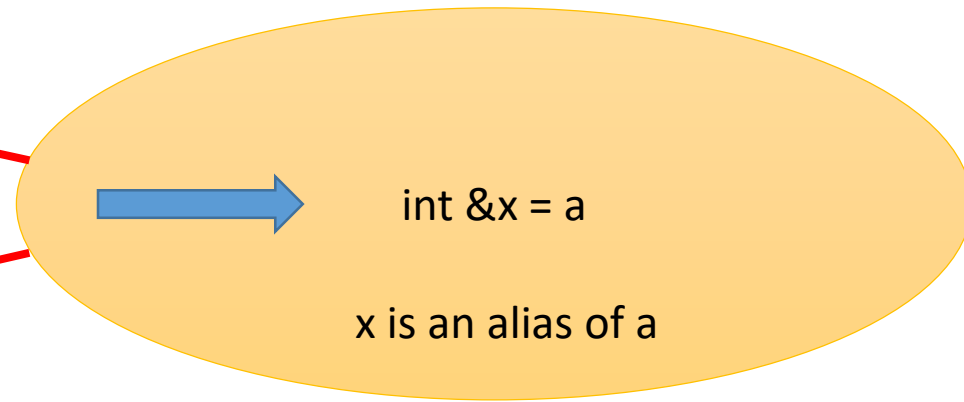
x is an alias (pseudonym) of a.

Used in functions when we want to:

- a) return multiple values
- b) make changes to the original variables
- c) send large arrays / objects

Call by reference

```
void somefunction(int &x){  
    x = 42;  
}  
  
int main(){  
    int a = 23;  
    somefunction(a);  
    // a is now 42  
}
```



Any change in x inside the function, will reflect to variable a in main.

Swapping

```
1  #include <iostream>
2
3  using namespace std;
4
5  void exchange(int& a, int& b){
6      int temp = a;
7      a = b;
8      b = temp;
9  }
10
11 int main(int argc, char** argv) {
12     int x = 5, y = 3;
13     cout << "x= " << x << " y= " << y << endl;
14     exchange(x,y);
15     cout << "x= " << x << " y= " << y << endl;
16     return 0;
17 }
```

But...

What if we want to swap **double's**?
long int's? **unsigned int's**?

Templates for the rescue!!

Template function

```
1  #include <iostream>
2
3  using namespace std;
4
5  template<typename T> void exchange(T& a, T& b){
6      T temp = a;
7      a = b;
8      b = temp;
9  }
10
11 int main(int argc, char** argv) {
12     double x = 65.2, y = 73.6;
13     cout << "x= " << x << " y= " << y << endl;
14     exchange(x,y);
15     cout << "x= " << x << " y= " << y << endl;
16     return 0;
17 }
```

T can be any type: int,
double, long int, unsigned
int, char, unsigned char, etc.
It can even be a custom
type defined by us (a class)

STL (Standard Template Library)

Containers	Algoriithms	Other stuff
<ul style="list-style-type: none">• Vector• Map• Set• Deque• List etc	<ul style="list-style-type: none">• Sort• Find• binary_search• Count• Shuffle etc	<ul style="list-style-type: none">• Iterators• Functors

Vector (1)

- Replaces C arrays
- Is actually an array, with many extras
 - Dynamically allocated
 - Resizable
 - Knows its size
 - Can be send to and returned by a function
 - We can use algorithms like `sort()`, `shuffle()` etc.
 - Can be safe (if we let it!)


Vector (2)

Declaration / initialization examples:

`vector<int> a; // zero sized vector`

`vector<int> a(n); // n elements`

`vector<int> a(n, v) // n elements, all initialized to v`



Can actually be
any type

Usage:

```
a[0] = 42;  
int x = a[1];  
just like an array!
```

Exercise

In `main()` create a vector of 100 random integers in the range `[1..1000]`

Write a function that sorts an array of ints (bubble sort, merge sort etc.) and call it from `main()`, sending the vector by reference.

Write a function that prints the elements of the (sorted) vector to a file.

Classes

C

```
struct Point {  
    int x;  
    int y;  
}
```

C++

```
class Point {  
    int x;  
    int y;  
public:  
    int getDistance();  
    void setX(int);  
    void setY(int);  
}
```

Procedural vs OO Programming

- Procedural

Think of what needs to be done, the procedures we have to follow and then decide about the data structures

- OOP

Decide what your “Actors” or objects are, describe their state and how this state changes.

Παραδειγμα: Εφαρμογή για Τράπεζα

- Procedural

Άνοιγμα λογαριασμού, Κατάθεση, Ανάληψη, Ερώτημα Υπολοίπου, Χορήγηση δανείου, επιβολή επιτοκίου, υπολογισμός τόκων κλπ.

- OOP

Objects: Πελάτες (Άτομα – Επιχειρήσεις), Λογαριασμοί (καταθέσεως – όψεως), Χαρτοφυλάκια μετοχών, Υπάλληλοι κλπ.

Classes - Encapsulation

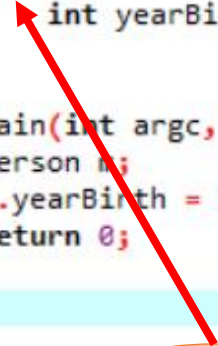
```
1  #include <iostream>
2
3  using namespace std;
4
5  class Person{
6      string name;
7      string contactPhone;
8      string contactAddress;
9      int yearBirth;
10 };
11
12 int main(int argc, char** argv) {
13     Person m;
14     m.yearBirth = 1990;
15     return 0;
16 }
17
```

Privacy Violation!!

Classes - Encapsulation


Two solutions:

```
1  #include <iostream>
2
3  using namespace std;
4
5  class Person{
6      string name;
7      string contactPhone;
8      string contactAddress;
9      public:
10         int yearBirth;
11 };
12
13 int main(int argc, char** argv) {
14     Person m;
15     m.yearBirth = 1990;
16     return 0;
17 }
18
```



BAD !!!

```
1  #include <iostream>
2
3  using namespace std;
4
5  class Person{
6      string name;
7      string contactPhone;
8      string contactAddress;
9      int yearBirth;
10     public:
11         void setYearBirth(int y){
12             yearBirth = y;
13         }
14 };
15
16 int main(int argc, char** argv) {
17     Person m;
18     //m.yearBirth = 1990;
19     m.setYearBirth(1990);
20     return 0;
21 }
22
```



GOOD !!!

Class structure

```
1  #include <iostream>
2
3  using namespace std;
4
5  class Person{
6
7      private:
8          string name;
9          string contactPhone;
10         string contactAddress;
11         int yearBirth;
12
13     public:
14         void setYearBirth(int y);
15         int getYearBirth();
16         void setName(string ph);
17         string getName();
18         void setContactPhone(string ph);
19         string getContactPhone();
20         void setContactAddress(string ad);
21         string getContactAddress();
22     };
23
24  int main(int argc, char** argv) {
25      Person m;
26      //m.yearBirth = 1990;
27      m.setYearBirth(1990);
28      return 0;
29  }
```

state

interface

main.cpp

```
1  #include <iostream>
2
3  using namespace std;
4  #include "Person.h"
5
6  int main(int argc, char** argv) {
7      Person m;
8      //m.yearBirth = 1990; /* ILLEGAL */
9      m.setYearBirth(1990);
10     return 0;
11 }
12
```

Person.h

```
1  class Person{
2
3      private:
4          string name;
5          string contactPhone;
6          string contactAddress;
7          int yearBirth;
8      public:
9          void setYearBirth(int y);
10         int getYearBirth();
11         void setName(string ph);
12         string getName();
13         void setContactPhone(string ph);
14         string getContactPhone();
15         void setContactAddress(string ad);
16         string getContactAddress();
17     };
18
19     void Person::setYearBirth(int y){
20         yearBirth = y;
21     }
22     int Person::getYearBirth(){
23         return yearBirth;
24     }
25
```

Constructor - Κατασκευαστής

- Special method (function) that gets called **automatically**, every time an object is created
- Always has the **same name** as the class and NO return type. It doesn't return anything, but we don't specify a "void" return type.
- Used for initializing the variables of the object (state) and possibly set up any resources needed (e.g. allocate memory, open a filestream, establish a connection etc.)
- If we don't specify one in our code a "default" one is created by the compiler, with no arguments, which does NOTHING.

Constructors

```
5 |  
6 |  
7 |  
8 |  
9 |  
10 |  
11 |  
12 |  
13 |  
14 |  
15 |  
16 |  
17 |  
18 |  
19 |  
20 |  
21 |  
22 |  
23 |  
24 |  
25 |  
26 |
```

```
    string contactPhone;  
    string contactAddress;  
    int yearBirth;  
public:  
    Person(){           // default constructor  
    }  
    Person(string n, string p, string a, int y){ // constructor  
        name = n;  
        contactPhone = p;  
        contactAddress = a;  
        yearBirth = y;  
    }  
    void setYearBirth(int y);  
    int getYearBirth();  
    void setName(string ph);  
    string getName();  
    void setContactPhone(string ph);  
    string getContactPhone();  
    void setContactAddress(string ad);  
    string getContactAddress();  
};
```

Object creation

```
1  #include <iostream>
2
3  using namespace std;
4  #include "Person.h"
5
6  int main(int argc, char** argv) {
7      Person m;
8      //m.yearBirth = 1990; /* ILLEGAL */
9      m.setYearBirth(1990);
10     Person john("John Smith", "+306998424242", "10, 3rd September, Thessaloniki", 1992);
11     return 0;
12 }
```


Hands on

- Write a Person class, with private fields 'age' and 'name'. The code must be written in a separate file e.g. "person.h"
- The interface should provide functions (getters – setters) for the private fields and two constructor (the default and one for initializing all fields). Also, a void printPerson() function, which prints: "Name: xxx, Age: yyy" at the console.
- In main() create 2 person objects. One using the default constructor (set the fields immediately after creating the object) and one using the other const'r.

Array of Objects

- We can create an **array** of objects:

```
Person ps[10];
```

```
ή
```

```
vector<Person> ps(10);
```

The objects are first created and then inserted (copied) into the array.

No default constructor

- If we define a constructor, the compiler does not create the default one. So we cannot declare (create) an object simply like this:

Person someone; // **ILLEGAL**

- We can also not create an array of objects, as we did in the previous page

SOLUTIONS

1. Create each object individually and push_back() it into the vector
2. Create an array of Person **pointers**

Static and Dynamic allocation of memory

C

Static:

```
int x;  
int y[1000];
```

Dynamic:

```
int* i = (int*)malloc(sizeof(int));  
int* a = (int*)malloc(1000*sizeof(int));
```

```
free(i);  
free(a);
```

C++

Static:

```
int x;  
int y[1000];
```

Dynamic:

```
int* i = new int;  
int* a = new int[1000];
```

```
delete i;  
delete[] a;
```

Creation and lifetime of objects

- Static (creation in stack memory):

```
Person a;
```

```
Person a("mike", 10);
```

- Dynamic (creation in heap)

```
Person* a;
```

```
a = new Person();
```

```
Person* a = new Person();
```

```
Person* a = new Person("mike",10);
```



delete a;

Creation and lifetime of objects

- Static (creation in stack memory):

The object lives until it is out of scope

- Dynamic (creation in heap)

The object lives until it is explicitly destroyed by **delete**.

Even if the pointer itself is destroyed by going out of scope

Destructors

- In every class there is one (and only one) destructor. If we don't provide one, a default destructor (doing nothing) is created by the compiler.
- The destructor is called automatically whenever an object is destroyed. We cannot call it explicitly. It is used for the “cleanup” of whatever resources the object has acquired, when it was created.
- It has the same name as the Class (with a preceding ~), no parameters and no return type:


```
~Person(){  
    // do stuff here...  
}
```

```
class Test{
    int x;
    int y;
public:
    Test(){
        cout << "Test created \n";
    }
    ~Test(){
        cout << "Bye bye world :( \n";
    }
};
```

Try with dynamic creation of Test object

```
int main() {
    Test a;
    cin.get();
    Test b[3];
    cin.get();

    {
        Test c;
        cin.get();
    }
    cin.get();
    Test c = a;
    cin.get();
}
```



Memory leaks

```
int main(){
    if(something){
        int* a = new int[10000];
    }
    // pointer a is destroyed here
    // the block of memory is not FREEd
    // a[13] illegal, "a" doesn't exist, we can't access the array of ints
    // although it is still there, in memory
    ...
}
```

Memory leaks

```
void fnc(){
    int *x = new int[10];
    // do something
}
int main(){
    for(int i=0;i<10000;i++){
        fnc();
    }
    // do something
}
```

```
int main(){
    int* x;
    for(int i=0;i<10000;i++){
        x = new int[10];
    }
    // do something
    delete[] x;
}
```

Memory leaks

```
FILE* fp;  
fp = fopen("filename.txt");  
// do stuff  
fclose(fp);
```

If we don't `fclose()`, the file stays in memory, even if `fp` is destroyed!

```
ofstream fp("filename.txt");
```

No opening or closing needed!!

RAII: Resource Acquisition Is Initialization

```
class ofstream{
    string fn;
    FILE* fp;
public:
    ofstream(string name){
        fn = name;
        fp=fopen(fn);
    }
    ~ofstream(){
        fclose(fp);
    }
};
```

```
class vector{
    int *a;
public:
    vector(int nr){
        a = new int[n];
    }
    ~vector(){
        delete[] a;
    }
};
```

Operator overloading

```
class Vector3d{
    public:
        int x, y, z;
        Vector3d(int a, int b, int c){
            x=a; y=b; z=c;
        }
};

Vector3d operator+(Vector3d v, Vector3d w){
    Vector3d result(v.x+w.x, v.y+w.y, v.z+w.z);
    return result;
}
```

Operator overloading

```
ostream& operator<<(ostream& o, Vector3d& v){  
    o << "(" << v.x << ", " << v.y << ", " << v.z << ")";  
    return o;  
}
```

```
int main(int argc, char** argv) {  
    Vector3d a(1,3,5);  
    Vector3d b(2,2,8);  
    cout << a << b << endl;  
    Vector3d z = a + b;  
    cout << z << endl;  
    return 0;  
}
```

Άσκηση

Φτιάξτε μια κλάση `Complex` με `private` μεταβλητές: `double real`, `double imag`. Να έχει κατασκευαστή που θα δέχεται 2 ορίσματα (`double` αριθμούς) για να αρχικοποιεί το πραγματικό και το φανταστικό μέρος του μιγαδικού.

Κάντε `overload` τους τελεστές `+`, `-`, `*` ώστε να εκτελούν σωστά τις πράξεις μεταξύ δύο `objects` τύπου `Complex`.

Κάντε επίσης `overload` τον τελεστή `<<` ώστε να τυπώνεται στην οθόνη: `(re, im)` όταν εκτελείται η εντολή: `cout << z;`

Άσκηση - συνέχεια

Στη `main()` κατασκευάστε 2 μιγαδικούς αριθμούς. Τυπώστε τους στην οθόνη εκτελώντας την εντολή: `cout << z1 << z2`

Κάντε πράξεις με τους 2 αριθμούς και τυπώστε το αποτέλεσμα τους στην κονσόλα.

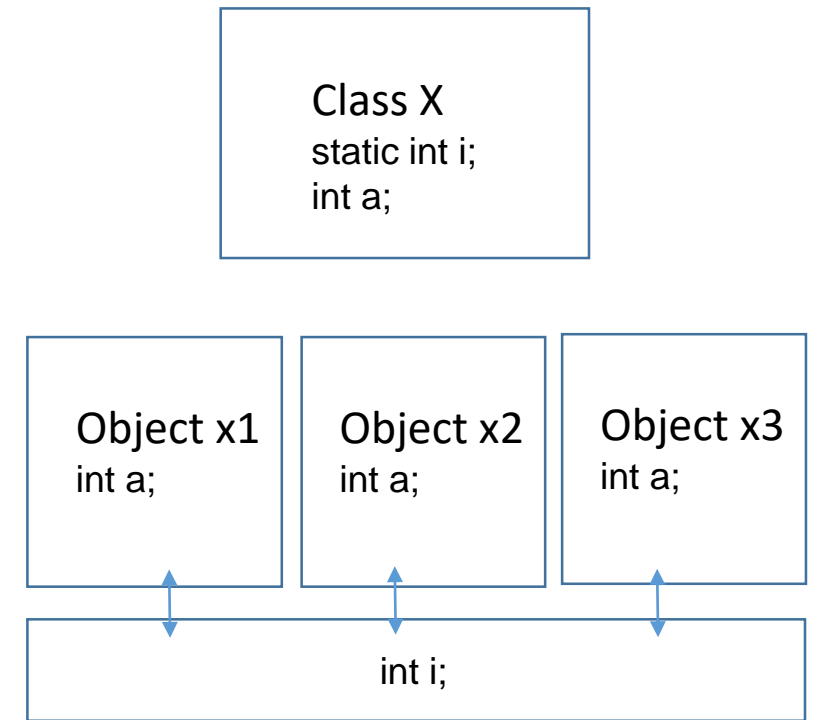
Δοκιμάστε αν μπορεί να γίνει η πράξη `z + 5` (δηλαδή `Complex + int`). Πώς πρέπει να κάνουμε `overload` τον `operator +`, ώστε να δουλέψει σωστά; Μπορεί να γίνει και η πρόσθεση `5 + z`;

static μεταβλητές κλάσης

```
class X {  
    public:  
        static int i;  
        int a;  
};  
int X::i = 0; // definition outside class declaration
```

A typical use of static members is for recording data **common to all objects** of a class.

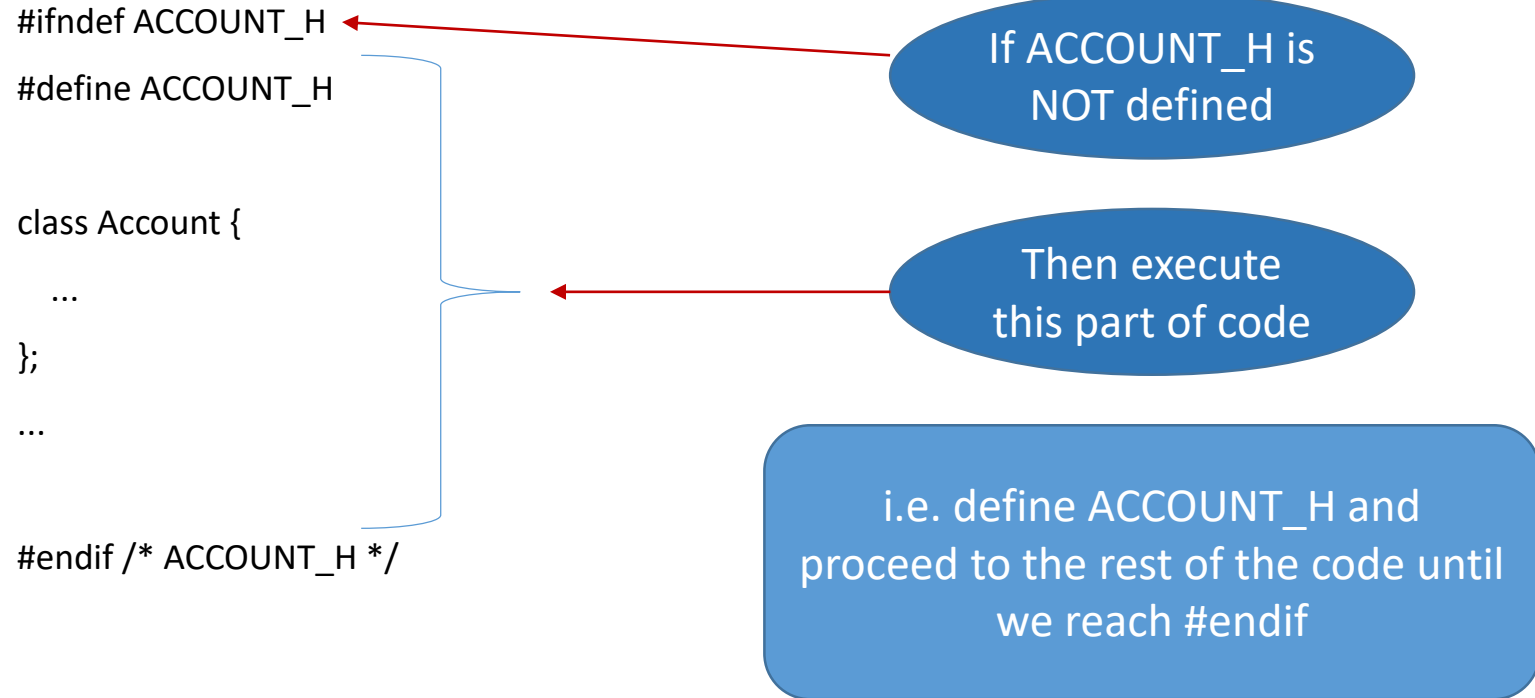
You access a static member by using the :: (scope resolution) operator. In the above example, you can refer to the static member i of class type X as X::i even if no object of type X is ever declared.



Each object of class X has its own copy of int a (x1.a, x2.a, x3.a) but there is only one instance of int i (X::i)

C++ Preprocessor Directives

Header Guards



This way we can `#include` "Account.h" multiple times in a file, but only the first time it will actually be included

Άσκηση

Στην κλάση SavingsAccount φτιάξτε μια συνάρτηση calcCost() που θα υπολογίζει τα λειτουργικά έξοδα του λογαριασμού ως εξής: Αν το balance είναι μικρότερο από μια τιμή minBalance, να αφαιρεί ένα ποσό ίσο με cost. Αν το balance ήταν μικρότερο του cost, τότε να μηδενίζεται (να μην γίνεται αρνητικό). Οι τιμές των minBalance και cost να δηλωθούν ως static, ώστε να είναι κοινές σε όλα τα αντικείμενα SavingsAccount. Ορίστε μια νέα μεταβλητή numberOfTransactions (αριθμός συναλλαγών), που θα αυξάνεται κατά 1, κάθε φορά που γίνεται deposit ή withdraw. Στον κατασκευαστή να αρχικοποιείται σε μηδέν.

Άσκηση (συνέχεια)

Δημιουργείστε μια κλάση Bank, η οποία θα έχει ως private μεταβλητή ένα vector που τα στοιχεία του θα είναι τύπου SavingsAccount. Να έχει μια public συνάρτηση insertAccount() η οποία θα δέχεται ως όρισμα ένα SavingsAccount και θα το εισάγει στο vector. Φτιάξτε επίσης μια συνάρτηση findAccount() η οποία θα δέχεται ως όρισμα ένα string με το IBAN ενός λογαριασμού και θα διατρέχει το vector ελέγχοντας αν κάποιος από τα SavingsAccount έχει αυτό το IBAN. Αν το βρει, θα επιστρέφει το index του vector. Αν δεν υπάρχει, να επιστρέφει -1.

Άσκηση (συνέχεια)

Σε όλα τα αρχεία *.h που έχετε, να εφαρμόσετε header guards (#ifndef, #define, #endif)

Δημιουργείστε μια νέα κλάση CheckingAccount η οποία θα είναι ίδια με την SavingsAccount, εκτός από τον ορισμό της calcCost. Αν οι συναλλαγές είναι πάνω από 2, το κόστος θα είναι το γινόμενο costPerTransaction *(numberOfTransactions - 2). Η μεταβλητή costPerTransaction να δηλωθεί ως static, ώστε να είναι κοινή σε όλους τους checkingAccount λογαριασμούς.

Inheritance

Person

```
name : string
yearBirth : int
AFM : string
contactPhone : string
contactAddress : string
```

```
Person(string,string)
void setName(string)
string getName()
void setYearBirth(int)
int getYearBirth ()
```

...

We need to describe:

Legal Person: Company etc – No Birth Date

Natural Person: Human – No Date of establishment

NaturalPerson

```
name : string
yearBirth : int
AFM : string
contactPhone : string
contactAddress : string
```

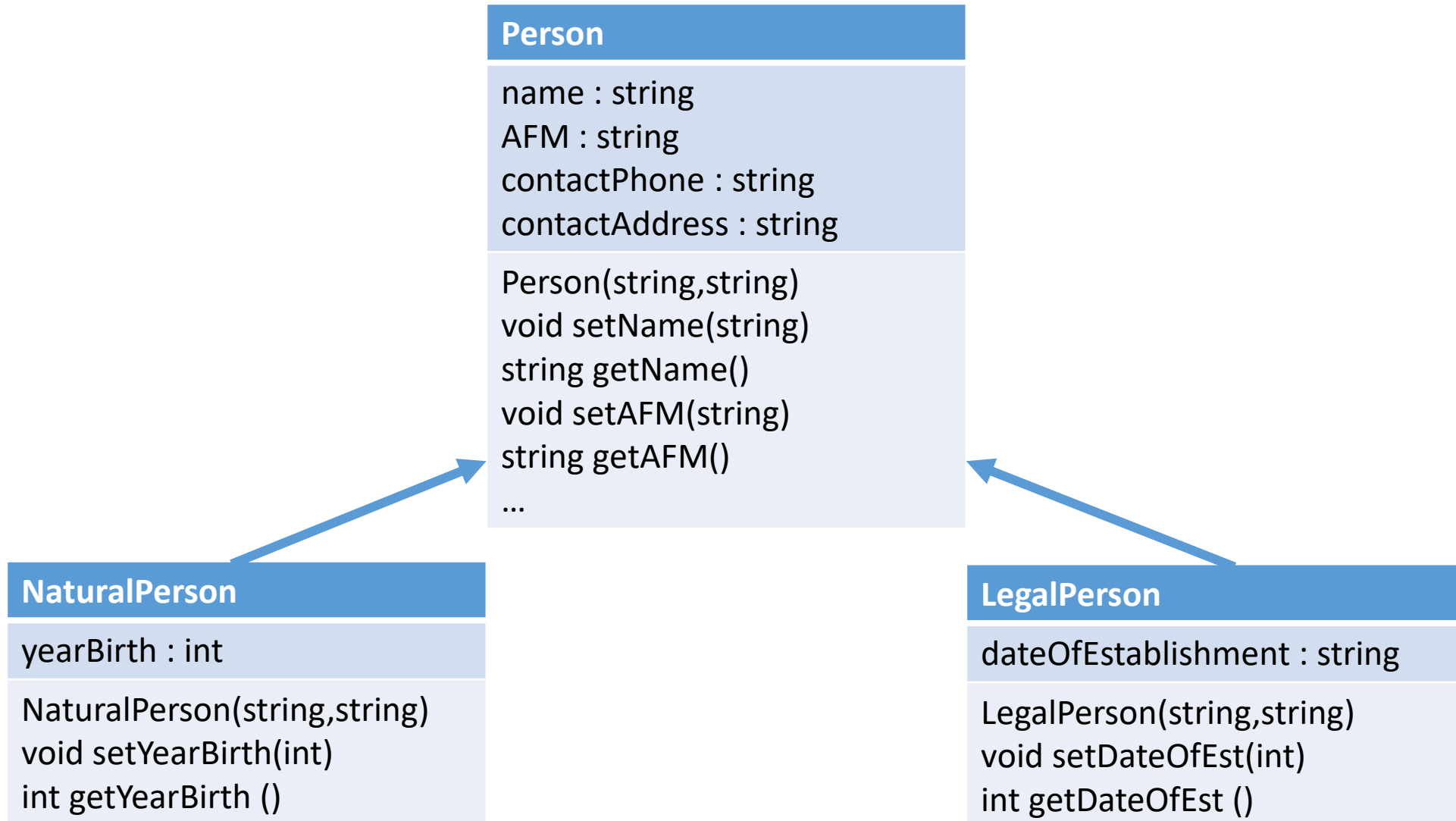
```
NaturalPerson(string,string)
void setName(string)
string getName()
void setYearBirth(int)
int getYearBirth ()
```

LegalPerson

```
name : string
dateOfEstablishment : string
AFM : string
contactPhone : string
contactAddress : string
```

```
LegalPerson(string,string)
void setName(string)
string getName()
void setDateOfEst(int)
int getDateOfEst ()
```

Inheritance – Class Diagram



Inheritance

SavingsAccount

owner : string
iban : string
balance : double
contactPhone : string
minBalance : static double

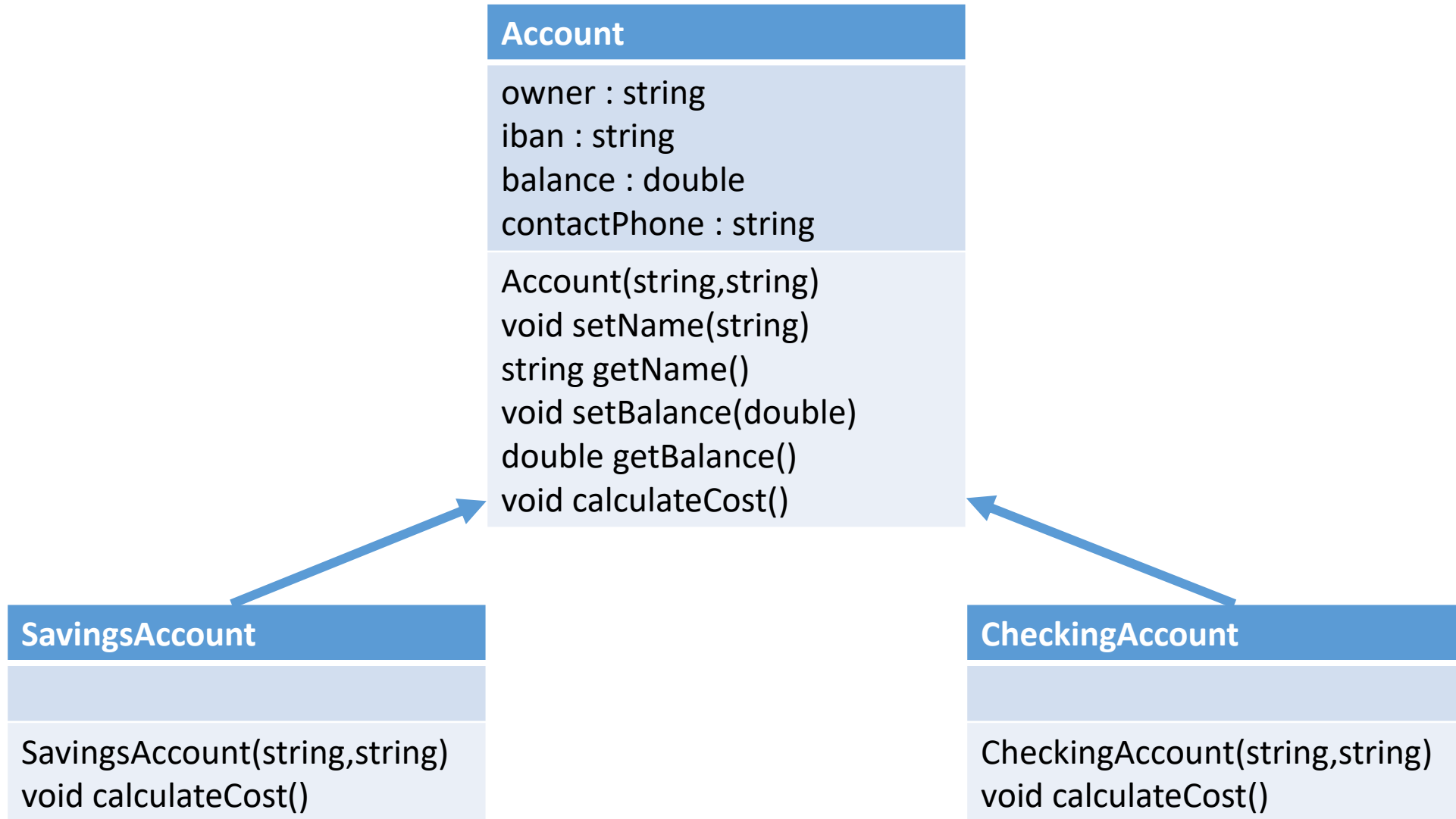
SavingsAccount(string,string)
void setName(string)
string getName()
void setBalance(double)
double getBalance()
void calculateCost()

CheckingAccount

owner : string
iban : string
balance : double
contactPhone : string
cost : static double

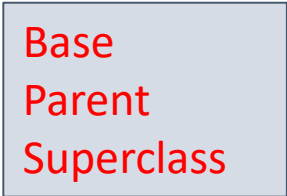
CheckingAccount(string,string)
void setName(string)
string getName()
void setBalance(double)
double getBalance()
void calculateCost()

Inheritance – Class Diagram



Inheritance - Syntax

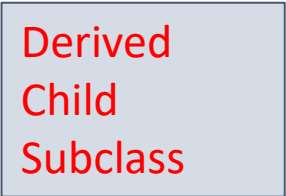

```
class Account
{
    public:
        Account(std::string Owner0, std::string IBAN0);
        std::string getOwner();
        std::string getIBAN();
        void Show();
        virtual void calculateCost();
    protected:
        std::string Owner;
        std::string IBAN;
};
```



```
class CheckingAccount : public Account
{
    public:
        static double CostPerTransaction;
        CheckingAccount();
        void calculateCost();

    private:

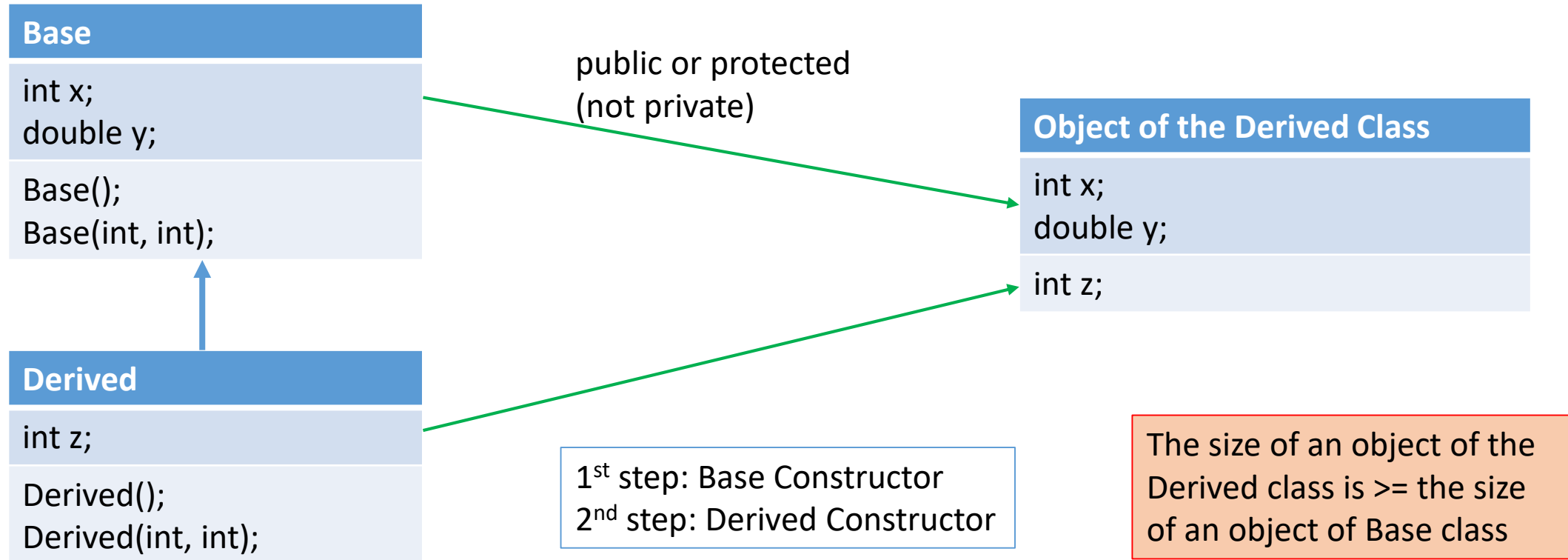
};
```



Access modifiers

- **public**
visible to everyone
- **protected**
visible only to sub-classes
- **private**
visible only to class methods

Construction of Derived object



Initialization list

```
class Account
{
    public:
        Account(std::string Owner, std::string IBAN);
        std::string getOwner();
        std::string getIBAN();
        void Show();
        virtual void CalcCost();
    private:
        std::string Owner;
        std::string IBAN;
};
```

Derived Class Contructor:

```
SavingsAccount(string own, string ib) : Account(own,ib)
{
    //
}
```

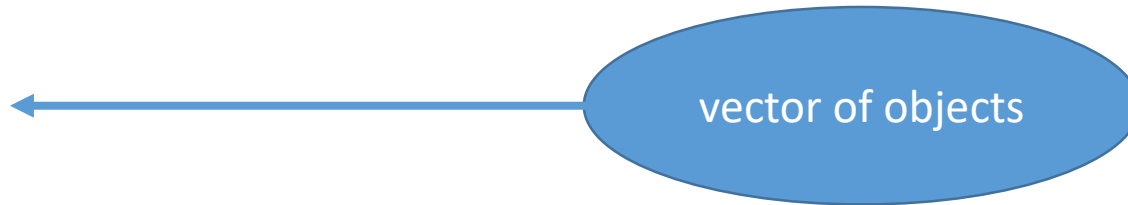
Pointers to objects

- It is not safe (or even possible) to fill a Base array, with objects of Derived class, because they probably won't fit

```
vector<Base> v;
```

```
Derived d;
```

```
v.push_back(d)
```

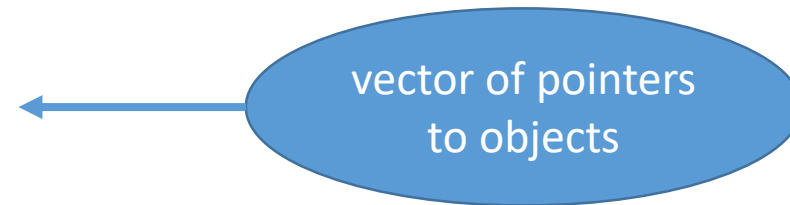


- Solution: Make it an array of **object pointers**

```
vector<Base*> v;
```

```
Base* d = new Derived();
```

```
v.push_back(d);
```



Object creation

We have seen 3 ways of creating an object:

Statically

```
SavingsAccount s(Constr. Parameters here);
```

Dynamically

```
SavingsAccount* sp = new SavingsAccount(Constr. Parameters here);
```

Dynamically – Polymorphism

```
Account* ap = new SavingsAccount(Constr. Parameters here);
```

s : Object of type SavingsAccount
sp: Pointer of type SavingsAccount
ap: Pointer of type SavingsAccount,
pointing to a SavingsAccount object

Polymorphism

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Useful for:

Having the same interface (i.e. same function name) while executing different code, e.g. `calculateCost()` executes different code, depending on the type of object.

Keeping different types of objects (or pointers) to the same container, e.g. `vector<Account*> v`, holds pointers to both Savings and Checking accounts.

Polymorphism example

```
class Base{
public:
    virtual void someFunction();
};

void Base::someFunction(){
    cout << "Hi from Base\n";
}
```

virtual
keyword

```
class Derived1 : public Base{
public:
    void someFunction();
};

void Derived1::someFunction(){
    cout << "Hi from 1\n";
}
```

inheritance
colon operator

```
class Derived2 : public Base{
public:
    void someFunction();
};

void Derived2::someFunction(){
    cout << "Hi from 2\n";
}
```

```
int main(){
    Base* b = new Base();
    Base* b1 = new Derived1();
    Base* b2 = new Derived2();
    b->someFunction();
    b1->someFunction();
    b2->someFunction();
}
```

Result:

```
Hi from Base
Hi from 1
Hi from 2
-----
```

polymorphism

with the "virtual" keyword, C++ calls
the function of the actual object

Without the "virtual" keyword:

```
Hi from Base
Hi from Base
Hi from Base
-----
```

since b, b1 and b2 are
Base* pointers, the
Base function is called