Γλώσσα Προγραμματισμού C++

Δαμιανός Κυπριάδης

Τι είναι η C++

Είναι C (διαδικαστικός προγραμματισμός)

Υποστηρίζει αντικειμενοστραφή προγραμματισμό (object oriented programming)

- ➢Ενθυλάκωση (encapsulation)
- ➢Επαναχρησιμοποίηση (reusability)
- ➢Πολυμορφισμό (polymorphism)

Τελεστές

<table>
<tr><td>Αριθμητικοί Τελεστές</td><td>+, -, *, /, %</td></tr>
<tr><td>Bitwise</td><td>|, &, ^, >>, <<, ~</td></tr>
<tr><td>Λογικοί</td><td>||, &&,!</td></tr>
<tr><td>Σχεσιακοί Τελεστές</td><td>==, !=, <=,>=, < ,>,</td></tr>
<tr><td>Τελεστής Ανάθεσης</td><td>=</td></tr>
<tr><td>Τελεστής Σύγκρισης</td><td>? :</td></tr>
<tr><td>Τελεστής Επίλυσης Εμβέλειας</td><td>::</td></tr>
</table>

Ενσωματωμένοι Βασικοί Τύποι

boolean

(signed/unsigned) char

(signed/unsigned) short int

(signed/unsigned) int

(signed/unsigned) long int

float

double

long double

Εντολές

Επιλογής

if (expression) statement

if (expression) statement **else** statement

switch (expression) statement

Επανάληψης

while (expression)   statement

do statement **while** (expression)

for (initial condition; expression; expression) statement

Μεταφοράς Ελέγχου

break;

continue;

return;

case expression : statement

default: statement

Δηλώσεις (declaration)

Οι δηλώσεις εισάγουν ονόματα (μεταβλητών, κλάσεων, συναρτήσεων) σε ένα πρόγραμμα.

Συνδέουν ένα όνομα με ένα είδος οντότητας

Επιτρέπονται πολλαπλές εμφανίσεις της ίδιας δήλωσης

```
extern int x;
int func1(int, char);
double abs(double);
```

Ορισμοί (definition)

Σε ένα ορισμό δημιουργείται μια οντότητα

Δεσμεύεται χώρος στη μνήμη

Ένας ορισμός είναι ταυτόχρονα και δήλωση

Ένας μόνο ορισμός είναι επιτρεπτός

```
int x;

double y=5.68;

double abs(double x){
   if(x>=0)
        return x;
   else
        return -x;
}
```

Συναρτήσεις

τύπος_επιστροφής όνομα_συνάρτησης (πλήθος παράμετρων) {
σώμα συνάρτησης }

```
int x;

double y=5.68;

double abs(double x){
   if(x>=0)
        return x;
   else
        return -x;
}
```

Απαιτείται δήλωση της συνάρτηση για να μπορέσει να κληθεί

Μία συνάρτηση μπορεί να οριστεί μόνο μία φορά σε ένα πρόγραμμα

Υπερφόρτωση Ονομάτων Συναρτήσεων

Overloading

Γίνεται χρήση του ίδιου ονόματος συνάρτησης για διαφορετικές συναρτήσεις

Διάκριση των συναρτήσεων βάσει των παραμέτρων

Δεν μπορεί να γίνει διάκριση βάσει της επιστρεφόμενης τιμής

```
int myfunction(int,int);
int myfunction(double,double);
void myfunction(char);
```

Υπερφόρτωση Ονομάτων Συναρτήσεων (συν.)

```
#include<iostream>

using namespace std;

int division(int a,int b) {return a/b;}
double division(double a,double b) {return a/b;}
double division(double a,int b) {return a/b;}
double division(double a) {return a/3;}

int main(){

int n1=10,n2=4;
double d1=10,d2=4;

cout <<division(n1,n2) <<endl;

cout <<division(d1,d2) <<endl;

cout <<division(d1,n2) <<endl;

cout <<division(d1) <<endl;

return 0;
}
```

Default Τιμές Ορισμάτων

```
void myFunction(int a,int b=0){
   cout <<a*b <<endl;
}
```

Κλήσεις της συνάρτησης:

```
   myFunction(5,6);
   myFunction(5);
```

Δεν επιτρέπεται η χρήση default τιμών σε ενδιάμεσα ορίσματα

```
void myFunction(int a=0,int b); //Λάθος
void myFunction(int a=0,int b=0);//Σωστό
```

Default Τιμές Ορισμάτων

```
void myFunction(int a,int b=0);
```

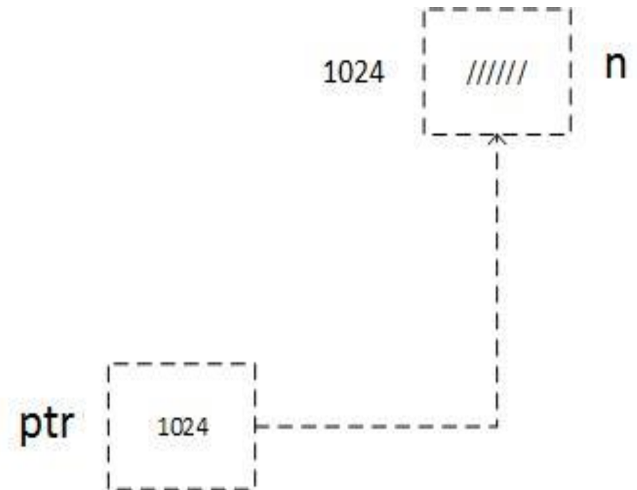
Ισοδυναμεί με:

```
void myFunction(int a,int b);
void myFunction(int a);
```

Δείκτες (pointers)

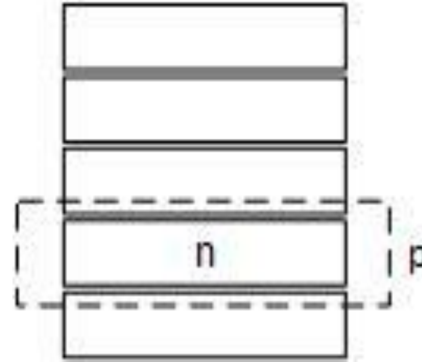
Θέση στη μνήμη που περιέχει τη διεύθυνση κάποιας άλλης θέσης

```
int n=10;
int *p;  //ορισμός δείκτη προς int

p=&n;  // ανάθεση της διεύθυνσης του n στον p

cout <<*p <<endl;  //Εκτύπωση της τιμής της n

*p=20;  //n=20

int m=30;

p=&m  //O δείκτης p δείχνει στη διεύθυνση της μεταβλητής m
```



Αναφορές (references)

```
#include <iostream>

using namespace std;


int main(){

int n=10;

int &p=n;

cout <<"Reference. Value:" <<p <<" Address:" <<&p <<endl;
cout <<"Variable.  Value:" <<n <<" Address:" <<&n <<endl;


return 0;
}
```



Έξοδος προγράμματος:

Reference. Value:10 Address:0x5fcad4

Variable. Value:10 Address:0x5fcad4

Πέρασμα Παραμέτρων σε Συναρτήσεις

Με χρήση δεικτών

```
#include <iostream>
using namespace std;
void absValue(int* p){
    if(*p<0)
          *p= (*p) * (-1);
}
int main(){
int n=6;
int *p;
p=&n;

absValue(p);

cout <<"*p=" <<*p <<endl;

return 0;
}
```

Με αναφορές

```
#include <iostream>
using namespace std;
void absValue(int& p){
    if(p<0)
          p=p*(-1);
}
int main(){

int n=-6;

absValue(n);

cout <<"n=" <<n <<endl;

return 0;
}
```

Διαφορές Δεικτών - Αναφορών

Το αντικείμενο στο οποίο “δείχνει” ένας δείκτης μπορεί να αλλάξει

Η αναφορά συμπεριφέρεται ως ένας σταθερός δείκτης

```
int n=10;
int m=20;
int *p;

p=&n;
cout <<*p <<endl; //prints 10

p=&m;
cout <<*p <<endl; //prints 20

int &rf=n;
cout <<rf <<endl; //prints 10

rf=m; //n=m
cout <<rf <<endl; //prints 20
cout <<n <<endl; //???
```

Σταθερές

const

Η λέξη const μετατρέπει μια μεταβλητή σε σταθερά

```
const int n=10;
```

Το περιεχόμενο της μεταβλητής δε μπορεί να αλλάξει

```
const int z=30;
z=3; //Λάθος
```

Επειδή δεν επιτρέπεται μεταβολή της τιμής της μεταβλητής, πρέπει να αρχικοποιείται

```
const int m; //Λάθος
```

Σταθερές και Δείκτες

Δήλωση σταθεράς για το αντικείμενο στο οποίο δείχνει ο δείκτης

```
const int n=10;
const int *ptr=&n;
```

Δήλωση σταθεράς για το δείκτη

```
int m=3;
int *const ptr=&m;
```

Δήλωση σταθεράς και για τα δύο

```
const int x=4;
const int* const ptr=&x;
```

Σταθερές και Αναφορές

Αν μια αναφορά δηλωθεί const η τιμή της μεταβλητής στην οποία “δείχνει” δεν μπορεί να αλλάξει μέσω αυτής

```
int z=4;
const int &rf=z;
//z++;
//rf++;
const int x=6;
const int &rf2=x;
//int &rf3=x;
```

Καθολικές Μεταβλητές

Ορίζονται εκτός συναρτήσεων

Δηλώνονται με χρήση της λέξης **extern**

file1.cpp

```
#include <iostream>
using namespace std;
extern int exVar;
int main(){
        exVar=45;
        int exVar=32;
        cout <<"Local extVar" <<exVar <<endl;
        cout <<"External exVar" <<::exVar <<endl;
}
```

file2.cpp

```
int exVar;
```

Στατικές Μεταβλητές

Δηλώνονται με χρήση της λέξης **static**

```
static int x=4;
```

Μια μεταβλητή static μπορεί να χρησιμοποιηθεί μόνο εντός του αρχείου που έχει ορισθεί

file1.cpp

```
#include <iostream>

using namespace std;

extern int statVar;

int main(){

//statVar=45;

}
```

file2.cpp

```
static int statVar;
```

Στατικές Μεταβλητές σε Συναρτήσεις

Δηλώνονται με χρήση της λέξης **static** στο σώμα της συνάρτησης

Πρέπει να αρχικοποιούνται

Διατηρούν την τιμή τους κατά τις κλήσεις της συνάρτησης

```
#include <iostream>

using namespace std;

void myFunction(){

        static int count=0;

        cout <<"Inside function count:" <<count++ <<endl;
}

int main(){

        for(int i=0; i<4; i++)
                myFunction();
}
```

Χώροι Μνήμης

Στατικός χώρος

Ορίζεται κατά την αρχή εκτέλεσης ενός προγράμματος. Διατηρείται καθ' όλη τη διάρκεια εκτέλεσης (π.χ. καθολικές μεταβλητές)

Μεταβαλλόμενος (stack) χώρος

Δεσμεύεται χώρος για αποθήκευση μεταβλητών όταν ο έλεγχος εκτέλεσης του προγράμματος φτάσει στον ορισμό τους

Χώρος δυναμικής δέσμευσης (ελεύθερος χώρος, heap)

Χώρος μνήμης ο οποίος δεσμεύεται κατά την εκτέλεση του προγράμματος κάθε φορά που απαιτείται δημιουργία ενός αντικειμένου

Δυναμική Δέσμευση Μνήμης

Με χρήση του τελεστή **new**

Ο τελεστής new επιστρέφει ένα δείκτη σε αντικείμενο του τύπου που προσδιορίζεται

```
int* ptr= new int;
int* ptr=new int(10); (αρχικοποίηση κατά τη
                        δημιουργία)
```

Δημιουργία arrays αντικειμένων

```
int* n=new int[5];
char* c=new char[10];
```

Αποδέσμευση μνήμης

Η αποδέσμευση του χώρου μνήμης που έχει αποδοθεί δυναμικά, γίνεται με τον τελεστή **delete**

```
int* n=new int; (δέσμευση)
delete n; (αποδέσμευση)

double* d=new double[32];
delete[] d;
```

Χώροι ονομάτων

Namespaces

Ορίζει εμβέλεια (scope)

Ομαδοποιούν δηλώσεις (declaration) μεταβλητών/συναρτήσεων

```
namespace mySpace{
  int n=6;
  double PI=3.14;
}
```

Χώροι ονομάτων (συν.)

Προσπέλαση των μεταβλητών που είναι δηλωμένες εντός ενός χώρου ονομάτων γίνεται με χρήση του τελεστή επίλυσης εμβέλειας **::**

```
namespace mySpace{
	int n=6;
	double PI=3.14;
}

cout <<mySpace::n <<endl;
```

Η λέξη **using**:

```
using namespace mySpace;
double p=PI;
```

Με χρήση της λέξης **using** μπορεί να γίνει προσπέλαση του περιεχομένου ενός χώρου ονομάτων χωρίς τη χρήση του ονόματος του χώρου.

Η επίδραση του using είναι μέχρι το τέλος της τρέχουσας εμβέλειας

Πρότυπα

Templates

Επιτρέπουν τον ορισμό ενός συνόλου σχετιζόμενων συναρτήσεων

```
template <class T>
void function(T t1, T t2) { //// }
```

T: παράμετρος τύπων

Κατά τη μεταγλώττιση ανάλογα με τους τύπους παράγεται ο κατάλληλος κώδικας

Παράμετροι τύπων μπορούν να υπάρξουν:

- Στα ορίσματα της συνάρτησης
- Στον τύπο που επιστρέφει η συνάρτηση
- Στο σώμα της συνάρτησης

Templates

```
#include <iostream>

using namespace std;

template<typename T>
T division(T& t1,T& t2){
   T res;
   res=t1/t2;
   return res;
}

int main(){

int n1=5,n2=3;

double d1=5.0,d2=3.0;

cout <<"Calling division
   with ints:"
   <<division(n1,n2) <<endl;

cout <<"Calling division
   with doubles:"
   <<division(d1,d2) <<endl;

return 0;
}
```

Κλάση

Τύπος ορισμένος από το χρήστη

Γενική μορφή

```
class <class name>{
   private:
        /////
   public:
        ////
};
```

Επιτρέπουν ελεγχόμενη πρόσβαση στα αντικείμενα της κλάσης μέσω κατάλληλων **συναρτήσεων μελών** (member functions)

Η δημιουργία αντικειμένων γίνεται μέσω κατάλληλων συναρτήσεων που ονομάζονται **συναρτήσεις κατασκευής** (constructors)

Για τη καταστροφή των αντικειμένων χρησιμοποιούνται οι **συναρτήσεις καταστροφής** (destructors)

Constructors

Δεσμεύουν χώρο στη μνήμη για το αντικείμενο

Αρχικοποιεί τα αντικείμενα

Έχει το ίδιο όνομα με αυτό της κλάσης

```
class X{
   public:
        X();
};
```

Αν δεν οριστεί κάποιος constructor από το χρήστη καλείται ένας default constructor από το μεταγλωττιστή

Μπορούν να οριστούν πολλαπλοί κατασκευαστές (overloading)

```
X::X(){ //// };

X::X(int d,int z){ //// };
```

Destructors

Αποδεσμεύουν το χώρο στη μνήμη που έχει ανατεθεί στο αντικείμενο

Μορφή destructor για μια κλάση X: **~X()**

Όταν ένα αντικείμενο βγαίνει από την εμβέλεια, τότε καλείται ο destructor της κλάσης για το αντικείμενο αυτό

Δομή Κλάσεων

Δομικά Στοιχεία κλάσης (building blocks)

Συναρτήσεις Κατασκευής (constructors)

Συναρτήσεις Καταστροφής (destructors)

Συναρτήσεις Ανάθεσης Τιμών

Συναρτήσεις Ανάκτησης Τιμών

Παράδειγμα Κλάσης

```
class Date{
     private:
              int day;
              int month;
              int year;
     public:
              Date();
              Date(int d,int m,int y);
              ~Date();
              int getDay();
              int getMonth();
              int getYear();
              void setDay(int d);
              void setMonth(int m);
              void setYear(int y);
};
Date::Date(){
     day=0;
     month=0;
     year=0;
     cout <<"Constructing Date with no arguments" <<endl;
}
Date::Date(int d,int m,int y){
     day=d;
     month=m;
     year=y;
     cout <<"Constructing Date with arguments" <<endl;
}
Date::~Date(){ cout <<"Deleting Date" <<endl; }
int Date::getDay(){          return day; }
int Date::getMonth(){ return month; }
int Date::getYear(){ return year; }

void Date::setDay(int d){ day=d; }
void Date::setMonth(int m){ month=m; }
void Date::setYear(int y){ year=y; }
```

Δείκτης this

Αυτοαναφορά

Είναι ένας δείκτης προς το ίδιο το αντικείμενο

Κάθε συνάρτηση μέλος μιας κλάσης “γνωρίζει” για ποιο αντικείμενο έχει κληθεί και μπορεί να αναφέρεται σε αυτό

Δεν μπορεί να ανατεθεί τιμή στο this

Κλάση Date

Η κλάση Date με χρήση του δείκτη **this**

```
class Date{
	private:
		int day;
		int month;
		int year;
	public:
		Date();
		Date(int d,int m,int y);
		~Date();
		int getDay();
		int getMonth();
		int getYear();
		void setDay(int d);
		void setMonth(int m);
		void setYear(int y);
};
Date::Date(){
	this->day=0;
	this->month=0;
	this->year=0;
	cout <<"Constructing Date with no arguments" <<endl;
}
Date::Date(int d,int m,int y){
	this->day=d;
	this->month=m;
	this->year=y;
	cout <<"Constructing Date with arguments" <<endl;
}
Date::~Date(){
	cout <<"Deleting Date" <<endl;
}
int Date::getDay(){ return this->day; }
int Date::getMonth(){ return this->month; }
int Date::getYear(){ return this->year; }
void Date::setDay(int d){ this->day=d; }
void Date::setMonth(int m){ this->month=m; }
void Date::setYear(int y){ this->year=y; }
```

Συνάρτηση Κατασκευής Αντιγράφων

Copy Constructors

Γενική Μορφή:

```
class X{
   private:
        ////
   public:
        X(const X& y1){ //// }
}
```

Αρχικοποίηση αντικειμένων με αντιγραφή

Παράδειγμα για την κλάση Date

```
Date(const Date& d){
   this->day=d.day;
   this->month=d.month;
   this->year=d.year;
}
```

Ανάθεση Στιγμιοτύπων

Υλοποιείται με υπερφόρτωση του τελεστή **=**

```
Date& Date::operator=(const Date& d){

  if(this!=&d){
     this->day=d.day;
     this->month=d.month;
     this->year=d.year;
  }

  return *this;
}
```

Ανάθεση - Αντιγραφή

Ανάθεση τιμών γίνεται σε ήδη δεσμευμένη μνήμη. Το αντικείμενο έχει ήδη δημιουργηθεί. Γίνεται ανάθεση νέων τιμών στα μέλη του αντικειμένου

Κατά την αντιγραφή στιγμιοτύπων δεσμεύεται μνήμη για το νέο αντικείμενο και ανατίθενται τιμές στα πεδία μέλη του

Κατά τον ορισμό ενός στιγμιοτύπου μιας κλάσης η χρήση του τελεστή ανάθεσης <u>**δεν**</u> αποτελεί έκφραση ανάθεσης. **Καλείται ο copy constructor**

Static Μέλη Κλάσεων

```
class X{
  static int i;
  public:
      ///////
};
```

Ένα static μέλος κλάσης ανήκει στην κλάση και όχι σε κάθε στιγμιότυπο (αντικείμενο) της κλάσης που δημιουργείται

Δημιουργείται μια μοναδική θέση στη μνήμη για κάθε static μέλος

Ο ορισμός γίνεται έξω από την κλάση

```
int X::i=1;
```

Static Μέθοδοι Κλάσεων

```
class X{
  static int i;
  public:
     //////
     static void foo();
};
```

Static μέθοδοι μπορούν να προσπελάσουν μόνο static μέλη της κλάσης

Μπορούν να καλέσουν μόνο static μεθόδους της κλάσης

Initializer list

Τρόπος αρχικοποίησης αντικειμένων

Η συνάρτηση κατασκευής της κλάσης Date

```
Date::Date(int d,int m,int y){
   this->day=d;
   this->month=m;
   this->year=y;
   cout <<"Constructing Date with arguments" <<endl;
}
```

Η ίδια συνάρτηση κατασκευής με χρήση λίστας αρχικοποίησης

```
Date::Date(int d,int m,int y):day(d),month(m),year(y){
   cout <<"Constructing Date with arguments" <<endl;
}
```

Const Συναρτήσεις Μέλη

Με χρήση της λέξης const κατά τη δήλωση της συνάρτησης

Οι const συναρτήσεις δεν μπορούν να τροποποιήσουν πεδία του αντικειμένου

```
class Date{

   private:
        //////
   public:
        //////
        int getDay() const;
};


int Date::getDay() const {   return this->day; }
```

Const Μέλη Κλάσεων

Δηλώνονται με χρήση της λέξης const

Δεν μπορεί να μεταβληθεί το περιεχόμενό τους

Για το λόγο αυτό λαμβάνουν τιμή κατά τη δημιουργία του αντικειμένου

Επαναχρησιμοποίηση

Reusability

Επαναχρησιμοποίηση

Μέσω Σύνθεσης (composition)

has a

Μέσω Κληρονομικότητα (inheritance)

is a

Σύνθεση

Μια κλάση έχει μέλη αντικείμενα άλλης κλάσης

Η σειρά κατασκευής των αντικειμένων: πρώτα κατασκευάζονται τα αντικείμενα που είναι μέλη μια κλάσης και μετά τα αντικείμενα της εξωτερικής κλάσης

Η καταστροφή των αντικειμένων γίνεται με αντίστροφη σειρά

Παράδειγμα Σύνθεσης

```
class Address{
	private:
		char* street;
		int number;
	public:
		Address(char* ,int );
		~Address();
		char* getStreet();
		int getNumber();

};


Address::Address(char* st,int n){
	this->street=new char[strlen(st)+1];
	strcpy(this->street,st);
	this->number=n;
	cout <<"Constructing Address" <<endl;
}

Address::~Address(){
	delete[] this->street;
	cout <<"Deleting Address" <<endl;
}

int Address::getNumber(){
	return this->number;
}

char* Address::getStreet(){
	return this->street;
}
```

```
class Person{
	private:
		char* name;
		int age;
		Address address;
	public:
		Person(char* , int, char* ,int);
		~Person();
		int getAge();
		char* getName();
};


Person::Person(char* nm,int a,char* st,int
	nu):address(st,nu){
	cout <<"Constructing Person" <<endl;
	this->name=new char[strlen(nm)+1];
	strcpy(this->name,nm);
	this->age=a;
}

Person::~Person(){
	delete[] this->name;
	cout <<"Deleting Person" <<endl;
}

char* Person::getName(){
	return this->name;
}

int Person::getAge(){
	return this->age;
}
```

Κληρονομικότητα

Δημιουργία νέων κλάσεων από ήδη υπάρχουσες κλάσεις

Οι παραγόμενες κλάσεις κληρονομούν τα χαρακτηριστικά της αρχικής κλάσης

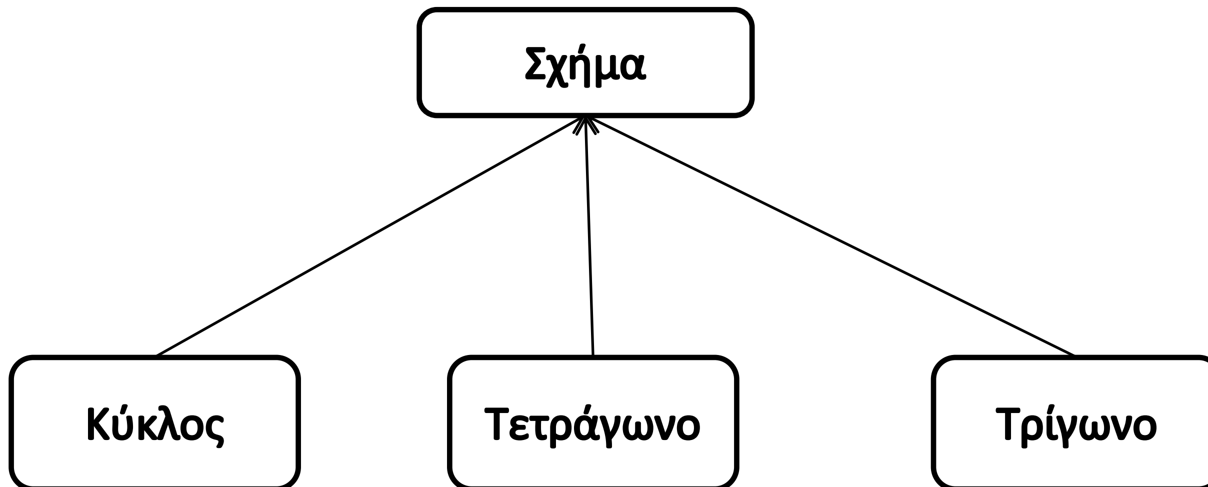
Μπορούν να επαναορίσουν τα χαρακτηριστικά της αρχικής κλάσης

Μπορούν να προστεθούν μέλη (συναρτήσεις - δεδομένα)

Κληρονομικότητα

Η κλάση *Σχήμα* είναι η βασική κλάση ή υπερκλάση

Οι κλάσεις *Κύκλος*, *Τετράγωνο*, *Τρίγωνο* είναι οι παραγόμενες κλάσεις ή υποκλάσεις



Κληρονομικότητα

```
class X{

};


class Y:public X{

};
```

X: βασική κλάση/υπερκλάση

Y: παραγόμενη κλάση/υποκλάση. Αποτελεί εξειδίκευση της κλάσης X

Κληρονομικότητα

Σειρά **δημιουργίας** αντικειμένων σε ιεραρχία κλάσεων

- Πρώτα δημιουργείται η βασική κλάση και έπειτα η παραγόμενη

Σειρά **καταστροφής** αντικειμένων

- Πρώτα καλείται η συνάρτηση καταστροφής (destructor) της παραγόμενης κλάσης και έπειτα της βασικής κλάσης

Protected Κληρονομικότητα

```
class X{ };

class Y:proteced X{ };
```

Public Y = Public Y

Protected Y = Protected Y + public X

Private Y = Private Y

Private Κληρονομικότητα

```
class X{ };

class Y:private X{ };
```

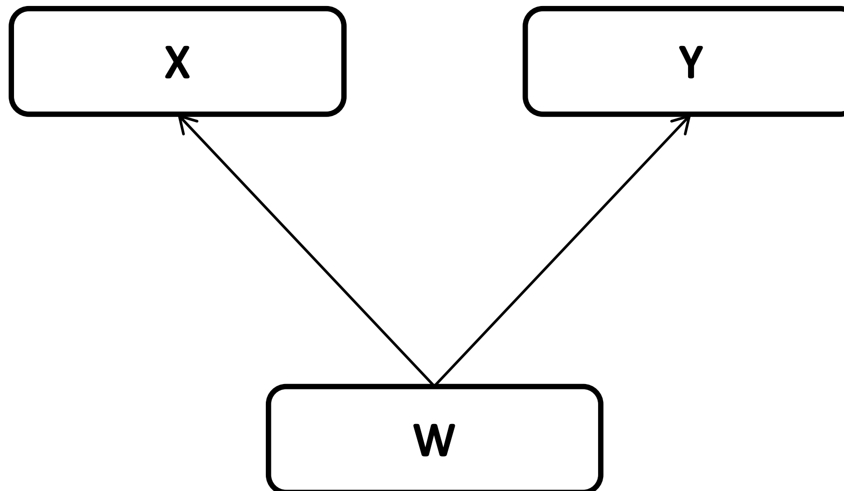
Public Y = Public Y

Protected Y = Protected Y

Private Y = Private Y + protected X + public X

Πολλαπλή Κληρονομικότητα

Μια κλάση μπορεί να κληρονομεί παραπάνω από μια κλάσεις



Ιεραρχία κλάσεων και Δείκτες

```
class A{ };
class B: public A{ };
```

Δεν επιτρέπεται η ανάθεση στιγμιοτύπου της βασικής κλάσης σε στιγμιότυπο παραγόμενης κλάσης

```
B* a=new A(); //Error
```

Επιτρέπεται όμως ανάθεση στιγμιοτύπου της παραγόμενης κλάσης σε στιγμιότυπο της βασικής κλάσης

```
A* b=new B();
```

Ιεραρχία κλάσεων και Δείκτες

```
class A{
   void print(){cout <<"This is A class" <<endl;}
};

class B: public A{
   void print(){cout <<"This is B class" <<endl;}
};
```

Η χρήση δεικτών στη βασική κλάση συνεπάγεται κλήση των μεθόδων της βασικής κλάσης

```
A* b=new B();
b->print(); //Output: This is A class
```

Πολυμορφισμός

Με χρήση της λέξης virtual μπορεί να κληθεί η κατάλληλη μέθοδος της ιεραρχίας των κλάσεων

```
class A{
    virtual void print(){cout <<"This is A class" <<endl;}
};

class B: public A{
    void print(){cout <<"This is B class" <<endl;}
};

A* b=new B();
b->print(); //Output: This is B class
```

Δεν γίνεται ορισμός virtual συναρτήσεων κατασκευής

Μπορούν να οριστούν όμως virtual συναρτήσεις καταστροφής

Από τη στιγμή που μια συνάρτηση ορίζεται virtual παραμένει virtual για όλη την ιεραρχία κλάσεων από εκεί και κάτω

Πολυμορφισμός
Πίνακας virtual συναρτήσεων

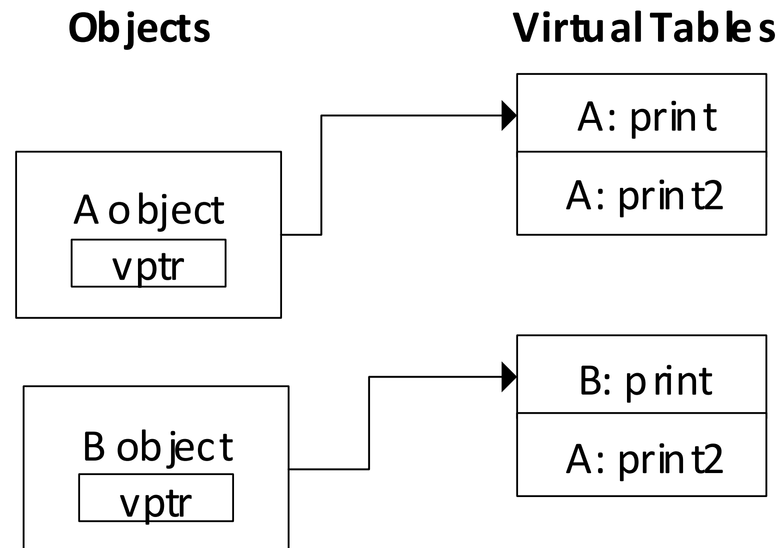
Για κάθε κλάση που περιέχει virtual συναρτήσεις δημιουργείται ένας πίνακας (VTABLE).

Ο μεταγλωττιστής τοποθετεί τη διεύθυνση κάθε virtual συνάρτησης στον πίνακα αυτό.

Σε κάθε κλάση τοποθετείται ένας δείκτης (vpointer) που δείχνει στον πίνακα αυτό.

Πολυμορφισμός
Πίνακας virtual συναρτήσεων

```
class A{
   virtual void print(){/////}
   virtual void print2(){/////}
   };

class B: public A{
   void print(){/////}
};
```



Αφηρημένες κλάσεις

Αν μια κλάση περιέχει pure virtual μεθόδους ονομάζεται αφηρημένη κλάση

Pure virtual μέθοδος:

```
virtual void print() =0;
```

Δεν μπορούν να οριστούν αντικείμενα μια αφηρημένης κλάσης

Μια παραγόμενη κλάση που δεν ορίζει μια αφηρημένη κλάση παραμένει και αυτή αφηρημένη κλάση

Φιλικές Συναρτήσεις - Κλάσεις

Δηλώνονται με τη χρήση της λέξης **friend**

Μια κλάση μπορεί να δηλώσει μια συνάρτηση ως φιλική, δίνοντας τη δυνατότητα πρόσβασης στο ιδιωτικό κομμάτι της κλάσης.

Αντίστοιχα μπορούν να δηλωθούν φιλικές κλάσεις

Η δήλωση μπορεί να γίνει οπουδήποτε μέσα στην κλάση (ιδιωτικό/δημόσιο μέρος)

Η ιδιότητα αυτή δεν κληρονομείται

Δεν έχει μεταβατική ή συμμετρική ιδιότητα

Φιλικές Συναρτήσεις - Κλάσεις

```
class X{

        private:
                int i;
                //friend class Y;
        public:
                X(int ii=0){i=ii;}

        friend void printI(const X&);
        friend class Y;
};


class Y{
        private:
                int value;
        public:
                Y(const X& x){value=x.i;}

};

void printI(const X& x){
        cout <<"X:" <<x.i <<endl;
}
```

Δείκτες σε Συναρτήσεις

```
void (*fptr) ()
```

Δείκτης σε συνάρτηση που δεν παίρνει όρισμα και δεν επιστρέφει κάποια τιμή

```
void (*fptr) (int,int)
```

Δείκτης σε συνάρτηση που παίρνει δύο ακεραίους ως ορίσματα και δεν επιστρέφει κάποια τιμή

```
int *(*fptr) (int,int)
```

Δείκτης σε συνάρτηση που παίρνει δύο ακεραίους ως ορίσματα και επιστρέφει ακέραιο

Operator Overloading

Σύνταξη

operator @

όπου @ συμβολίζει το είδος του τελεστή (operator)

Το πλήθος των ορισμάτων καθορίζεται από:

- Το αν είναι unary ή binary τελεστής
- Αν είναι συνάρτηση μέλους ή όχι

Πρόβλημα Ιστογράμματος

Έστω το σύνολο $S = \{1, \dots, n\}$, n διακριτών κλειδιών.

Έστω $keys = [3, 5,\ n-1,\ n, 2\ , 1, \dots]$ πίνακας εμφανίσεων των κλειδιών

Έξοδος: Μια λίστα των διακριτών κλειδιών και της συχνότητας εμφάνισής τους

Ιστόγραμμα με Array

```
void main(void){

int n,r;

cout <<"Enter number of elements
and range" <<endl;

cin >>n >>r;

int *h;

h=new int[r+1];

for(int i=0; i<=r; i++)
        h[i]=0;
```

```
for(int i=0; i<=n; i++){

int key;
cout <<"Enter element " <<i
<<endl;
cin >>key;

h[key]++
}

cout <<"Distinct elements and
frequencies are" <<endl;

for(int i=0; i<=n; i++){
    if(h[i])
      cout <<i <<" " <<h[i] <<endl;
}
}
```

Ιστόγραμμα με ΔΔΑ

```
class eType{

friend void main(void);
friend void Add1(eType& );
friend ostream& operator <<(ostream&
,eType);

public:

operator int() const {return key;}

private:
int key;
int count;

};

ostream& operator<<(ostream& out, eType
x){
     out <<x.key <<" " <<x.count <<" "
;
     return out;
}

void Add1(eType& e){
     e.count++;
}
```

```
void main(void){

BSTree<eType,int> T;

int n;

cout <<"Enter number of elements"
<<endl;

cin >>n;


for(int i=1; i<=n; i++){

     eType e;

     cout <<"Enter element" <<i <<endl;

     cin >>e.key;

     e.count=1;

     T.InsertVisit(e,Add1);
     }

}
```

Πρόβλημα Συσκευασίας Κιβωτίων

Ορισμός: Έστω $\mathrm{S} = (S_1, S_2, S_3, \dots)$ σύνολο κιβωτίων ίδιας χωρητικότητας V, και A σύνολο n αντικειμένων όγκου a_i το καθένα, όπου $i = 1, \dots, n$.

Ζητείται να βρεθεί ο ελάχιστος αριθμός κιβωτίων που χρειάζεται για να συσκευαστούν όλα τα n αντικείμενα

Πρόβλημα Συσκευασίας Κιβωτίων με χρήση ΔΔΑ (1/2)

```
template<class E,class K>
bool BSTree<E,K>::FindGE(const K& k, K& Kout) const{
        BinaryTreeNode<E> *p=root;
        BinaryTreeNode<E> *s=0;

        while(p){
                if(k<=p->data){
                        s=p;
                        p=p->LeftChild;
                }
                else
                        p=p->RightChild;

        }
        if(!s)
                return false;

        Kout=s->data;
        return true;
}
```

Πρόβλημα Συσκευασίας Κιβωτίων με χρήση ΔΔΑ (2/2)

```
class BinNode{
friend void BestFitPack(int *,int,int);
friend ostream* operator<<(ostream&,
BinNode );

public:
operator int() const {return avail;}

private:
int ID,avail;
};
```

```
void BestFitPack(int s[],int n, int c){

int b=0;
BSTree<BinNode,int> T;

for(int i=1; i<=n; i++){
   int k;
   BinNode e;
   if(T.FindGE(s[i],k))
         T.Delete(k,e);
   else{
         e=*(new BinNode);
         e.ID=++b;
         e.avail=c;
   }

   e.avail-=s[i];
   if(e.avail)
         T.Insert(e);
   }

}
```