TEMA 4

POLIMORFISME

Temporalització: 4 Sessions (6 hores)

Versió 0.9

Tema 4. Polimorfisme
Objectius bàsics

- Comprendre el concepte de polimorfisme
- Conéixer i saber utilitzar els diferents tipus de polimorfisme.
- Comprendre el concepte d'enllaçat estàtic i dinàmic en els llenguatges OO.
- Comprendre la relació entre polimorfisme i herència en els llenguatges fortament tipats.
- Apreciar la manera en què el polimorfisme fa que els sistemes siguen extensibles i mantenibles.

Índex

1. Motivació

- Objectiu de la POO
  - Aproximar-se al mode de resoldre problemes en el món real.

- El polimorfisme és el mode en què els llenguatges OO implementen el concepte de **polisèmia** del món real:
  - Un únic nom per a molts significats, segons el context.

1. Conceptes previs: Signatura

- Signatura de tipus d'un mètode:
  - Descripció dels tipus dels seus arguments, el seu ordre i el tipus retornat pel mètode.
    - Notació: ***<arguments> → <tipus retornat>***
    - Omitim el nomb del mètode, el de la classe a la qual pertany (el tipus del receptor)
  - Exemples

    `double power (double base, int exp)`
    - ***double*int → double***

    `bool Casella::setPeça(Peça& p)`
    - ***Peça → bool***

1. Conceptes previs: Àmbit

- Àmbit d'un nomb:
  - Porció del programa en la qual un nomb pot ser utilitzat d'una determinada manera.
  - Exemple:

    `double power (double base, int exp)`
    - La variable *`base`* només pot ser utilitzada dintre del mètode `power`
  - Àmbits actius: pot haver diversos simultániament

```
class A {
 int x,y;
 public:
  void f() {
   // Àmbits actius:
   // GLOBAL
   // CLASSE (atribs. de classe i d'instància)
   // MÈTODE (argumento, var. locals)
   if (...) {
    string s;
    // LOCAL (var. locals)
   }
  }
```

1. Conceptes previs: Temps d'enllaç

- Moment en el què s'identifica el fragment de codi a executar associat a un missatge (cridada a mètode) o l'objecte concret associat a una variable.
  - **Temps de Compilació: Enllaç estàtic (early binding)**
    - *EFICIENCIA*
  - **Temps de Execució: Enllaç dinàmic (late binding)**:
    - *FLEXIBILITAT*
  - Aplicable a**:**
    - **OBJECTES:**
      - **Amb enllaç estàtic** el tipus d'objecte que conté una variable se determina en temps de compilació.
      - Amb **enllaç dinàmic** el tipus d'objecte al qual fa referència una variable no està predefinit, pel que el sistema gestionarà la variable en funció de la naturalessa real de l'objecte que referèncie en cada moment.
        - Llenguatges com Smalltalk sempre utilitzen enllaç dinàmico amb variables
        - C++ només permet enllaç dinàmic amb variables quan estos són punters, i només dintre de jerarquies d'herència.
    - **OPERACIONS:**
      - **Amb enllaç estàtic** l'elecció de què mètode serà l'encarregat de respondre a un missatge es realitza en temps de compilació, en funció del tipus que tenia l'objecte destinatari de la cridada en temps de compilació.
      - Amb **enllaç dinàmic** l'elecció de què mètode serà l'encarregat de respondre a un missatge es realitza en temps d'execució, en funció del tipus corresponent a l'objecte que referència la variable mitjançant la que s'invoca al mètode a l'instant actual.

1. Conceptes previs: Temps d'enllaç

- **El tipus de llenguatge utilitzat (fortament o dèbilment tipat) determina el seu suport a l'enllaç dinàmic:**
  - **Llenguatge fortament tipat**
    - **Llenguatges Procedimentals**: **no suporten enllaç dinàmic**: el *tipus* de tota expresió (identificador o fragment de codi) es coneix en temps de compilació.
    - **LOO**: només suporten **enllaç dinàmic dintre de la *jerarquia de tipus*** a la qual pertany tota expresió (identificador o fragment de codi) establida en temps de compilació.
  - **Llenguatge dèbilment tipat**
    - **Si suporten enllaç dinàmic**: l'enllaç entre variables i tipus (sense restriccions) es fa en temps d'execució (variables es compilen sense assignar-les tipus concret).

2. Polimorfisme
Definició

- **Capacitat d'una entitat de referenciar distints elements en distints instants de temps.**
  - El polimorfisme ens permet programar de manera general en lloc de programar de manera específica.
  - Hi ha quatre tècniques, cadascuna permet una forma distinta de **reutilització de software**, que facilita al mateix temps el desenvolupament ràpid, la confiança i la facilitat d'ús i manteniment.
    - Sobrecàrrega
    - Sobreescriptura
    - Variables polimòrfiques
    - Genericitat

Tipus de polimorfisme

- **Sobrecàrrega (*Overloading*, Polimorfisme ad-hoc)**: només un nomb de mètode i moltes implementacions distintes.
  - Les funcions sobrecarregades normalment es distingueixen en temps de compilació per tindre distints paràmetres d'entrada i/o eixida.
- **Sobreescriptura (*Overriding*, Polimorfisme d'inclusió)**: Tipus especial de sobrecàrrega que ocorre dins de relacions d'herència.
  - En este cas la signatura és la mateixa (refinament o reemplaçament del mètode del pare) però els mètodes estan en dues classes distintes relacionades mitjançant herència.
- **Variables polimòrfiques** (Polimorfisme d'assignació): variable que es declara com d'un tipus però que referència en realitat un valor d'un tipus distint.
  - Quan una variable polimòrfica s'utilitza com argument, la funció resultant es diu que exhibeix un **polimorfisme pur.**
- **Genericitat (plantilles o *templates*):** forma de crear ferramentes de propòsit general (classes, mètodes) i especialitzar-les per a situacions específiques.

Tipus de polimorfisme

- **Sobrecàrrega**
  - Factura::**imprimir**()
  - Factura::**imprimir**(int numCopias)
  - LlistaCompra::**imprimir**()
- **Sobreescriptura**
  - Cuenta::**abonarInteres**()
  - CompteJove::**abonarInteres**()
- **Variables polimòrfiques**
  - **Compte** *pc=new **CompteJove**();
- **Genericitat**
  - Llista<Client>
  - Llista<Artícul>
  - Llista<Alumne>
  - Llista<Habitacio>

3. Sobrecàrrega (*Overloading*, polimorfisme *ad-hoc*)

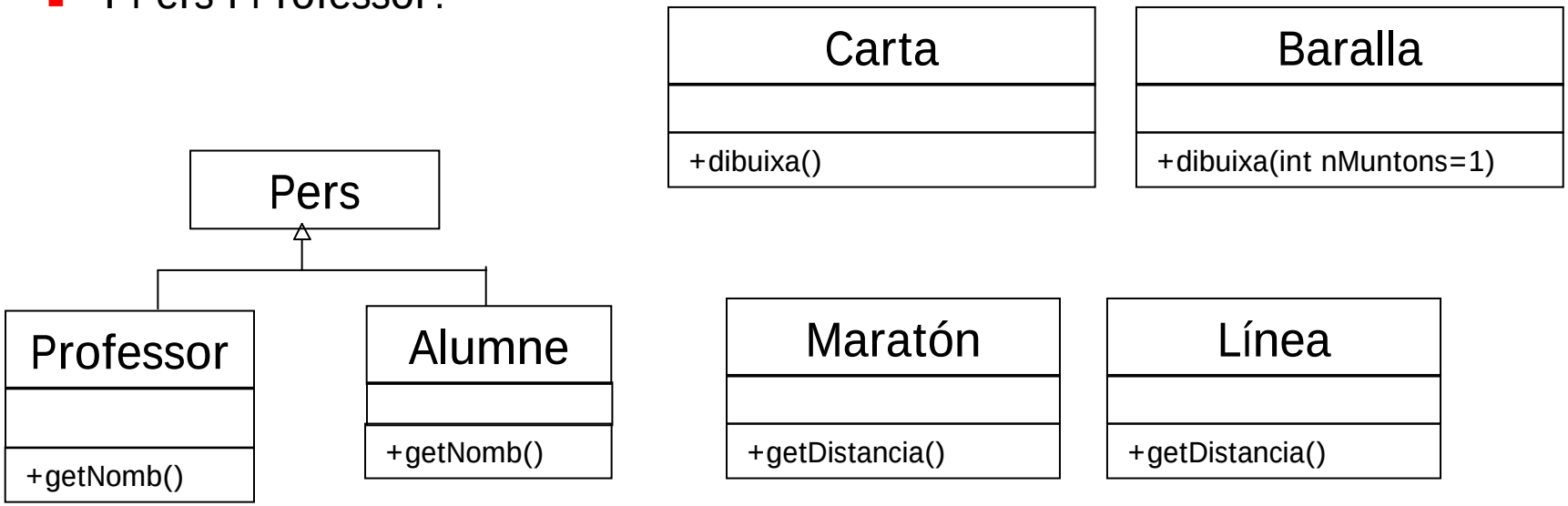
- Un mateix nomb de missatge associat a vàries implementacions
- La sobrecàrrega es realitza en **temps de compilació** (enllaç estàtic) en funció de la signatura completa del missatge.

- Dos tipus de sobrecàrrega:
  - **Basada en àmbit**: Mètodes amb **diferents àmbits de definició**, independentment de les seues signaturas de tipus. Permés en tots els llenguatges OO.
    - Un mateix mètode pot ser utilitzat en dues o més classes.
    - P. ex. Sobrecàrrega d'operadors com funcions membre.

  - **Basada en signatura**:Mètodes amb **diferents signaturas de tipus en el mateix àmbit de definició.** No permés en tots els llenguatges OO.
    - P. ex. Cualsevol conjunt de funcions no membre (en l'àmbit de definició global) que comparteixen nomb.
    - Dos o més mètodes en la mateixa classe poden tindre el mateix nomb sempre que tinguen distinta signatura de tipus.

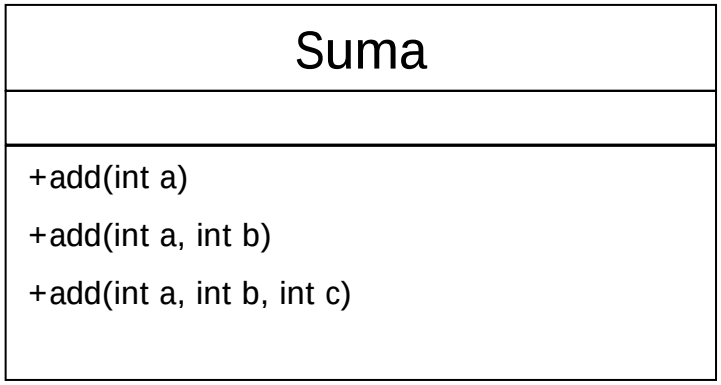
Sobrecàrrega basada en àmbit

- Distints àmbits impliquen que el mateix nomb de mètode pot aparéixer en ells sense ambigüetat ni pèrdua de precisió.
- La sobrecàrrega per àmbit no requereix que les funcions associades amb un nomb sobrecarregat tinguen cap similitud semàntica, ni la mateixa signatura de tipus.
- Exemples
  - Són Professor i Alumne a àmbits distints?
  - I Pers i Professor?



Sobrecàrrega basada en signatures de tipus

- Mètodes en el mateix àmbit poden compartir el mateix nomb sempre que diferisquen en nombre, ordre i, en llengutges amb tipat estàtic, el tipus dels arguments que requereixen.
  - Este estil ocorreix en molts llenguatges funcionals, en alguns imperatius (e.x. ADA) i en llenguatges OO com C++, C#, Java, Delphi Pascal o CLOS
  - C++ permet esta sobrecàrrega de manera implícita sempre que la selecció del mètode requerit per l'usuari puga establir-se de manera no ambigua en temps de compilació.
    - Açò implica que la signatura no pot distingir-se només pel tipus de tornada



Sobrecàrrega basada en signatures de tipus

- Esta manera de sobrecàrrega en tems de compilació pot donar lloc a resultats inesperats:

```
class Pare{…};
class Fill: public Pare{…};
void Test (Pare *p) {cout << “pare”;};
void Test (Fill *h) {cout << “fill”;}; //ppi sust.
int main(){
   Pare *obj;
   int tipus;cin>>tipus;
   if (tipus==1) obj=new Pare();
   else obj=new Fill(); //ppi de sustitución
   Test (obj); //a qui invoque?
}
```

Sobrecàrrega basada en signatures de tipus

- **No tots els LOO permeten la sobrecàrrega:**
  - Permeten sobrecàrrega de mètodes i operadors: C++
  - Permeten sobrecàrrega de mètodes però no d'operadors: Java, Python, Perl
  - Permeten sobrecàrrega d'operadors però no de mètodes: Eiffel

Sobrecàrrega d'operadors en C++

- Dintre de la sobrecàrrega basada en signatures de tipus, té especial relevància la **sobrecàrrega d'operadors**

- **Ús**: Utilitzar operadors tradicionals amb tipus definits per l'usuari.
  - S'ha d'anar amb cura amb el seu ús, perquè pot dificultar la comprensió del codi si s'utilitzen en un sentit distint a l'habitual.

- Forma de sobrecarregar un operador @:

  <tipus retornat> operator@(<args>)

- Per utilitzar un operador amb un objecte de tipus definit per usuari, este ha de ser sobrecarregat.
  - Excepcions: operador d'assignació (=) i l'operador de direcció (&)
    - L'operador direcció (&) per defecte retorna la direcció de l'objecte.
    - L'operador assignació per defecte assigna camp a camp.

Sobrecàrrega d'operadors en C++

- En la sobrecàrrega d'operadors no es poden canviar
  - *Precedència* (què operador s'avalua abans)
  - *Associativitat* a=b=c ⇗ a=(b=c)
  - *Aritat* (operadors binaris per a que actuen com unaris o viceversa)

- No es pot crear un nou operador

- No es poden sobrecarregar operadors per a tipus predefinits.

- Es poden sobrecarregar tots els operadors definits en C++ com unaris i binaris excepte "**.**", "**.***", "**::**", **sizeof**.

- "**?:**" es l'únic operador ternari definit en C++, i no es pot sobrecarregar

Sobrecàrrega d'operadors en C++

- La sobrecàrrega d'operadors es pot realitzar mitjançant funcions membre o no membre de la classe.
  - Com a funció membre: l'operant de l'esquerra (en un operador binari) ha de ser un objecte (o referència a un objecte) de la classe:

```
Complex Complex::operator+(const Complex&)
…
Complex c1(1,2), c2(2,-1);
c1+c2; // c1.operator+(c2);
c1+c2+c3; // c1.operator+(c2).operator+(c3)
```

Sobrecàrrega d'operadors en C++

- Sobrecàrrega d'operador com funció no membre:

1. Quan l'operant de l'esquerra no és membre de la classe

```
ostream& operator<<(ostream&, const Complex&);
istream& operator>>(istream&, Complex&);
…
Complex c;
cout << c; // operator<<(cout,c)
cin >> c; //  operator>>(cin,c)
```

2. Per claredat

```
Complex operator+(const Complex&, const Complex&);
```

Sobrecàrrega d'operadors en C++

- En C++, els operadors "=", "[]", "->", "()", "new" y "delete", només poden ser sobrecarregats quan es defineixen com **membros d'una classe.**
- Si un operador té formats unaris i binaris (+,&), ambdós formats poden ser sobrecarregats.

Sobrecàrrega d'operadors en C++
Operadors unaris

- Sobrecàrrega d'operadors unaris:
  - Els operadors unaris poden ser sobrecarregats sense arguments o amb un argument
  - Usualment són implementats com funcions membre.

    `<tipus>` **`operator<operador>`**`();`
  - Exemple:

```
class Temps{
 public:
  Temps(int, int);
  void mostrar();
  Temps operator++();
 private:
   int hora;
   int minut;
};
```

```
Temps
Temps::operator++()
// PREINCREMENT
{
  minut++;
 if (minut >= 60)
  {
    minut = minut - 60;
    hora++;
  }
 return *this;
}
```

Sobrecàrrega d'operadors en C++
Operadors unaris

```
class Temps{
 public:
  Temps(int, int);
  void mostrar();
   Temps operator++();//pre
   Temps operator++(int);//post
//l'argument int distingeix una
//versió de l'altra
 private:
   int hora;
   int minuto;
};
```

- Quina seria l'eixida d'este programa?
- Sobrecàrrega del mateix mode l'operador -- (predecrement i postdecrement)

```
Temps
Temps::operator++(int x)
{
  Temps aux(*this);
  minut++;
  if (minut >= 60){
     minut = minut - 60;
     hora++;
   }
  return aux;
}
void main(){
    Temps t1(17,9);
    t1.mostrar();//??
    (t1++).mostrar();//??
    t1.mostrar();//??
    (++t1).mostrar();//??
    t1.mostrar();//??
}
```

Sobrecàrrega d'operadors en C++

Exercici: operadors unaris

```
class TPunt {
public:
   TPunt (int, int);
private :
     int x;
     int y;
};
```

- Es desitja que l'operador ! indique *true* si x i y són diferents i *false* en cas contrari.

Sobrecàrrega d'operadors en C++
Operadors binaris

- Funcions no membre: **dos arguments (ambdós operants)**
  - `<tipus_retornat> operator<operador>(<tipus>,<tipus>) ;`
    - Poden ser funció amiga
- Funcions membre: **només un argument (operant de la dreta)**
  - `<tipus_retornat> operator<operador>(<tipus>);`

```
class Temps{
 public:
  Temps(int h=0, int m=0)
    {hora=h;  minut=m;};
  void mostrar(){
    cout <<hora
         <<":"<<minut<<endl;
  };
  …
 private:
   int hora;   int minut;
};
```

OBJECTIU:

```
int main(){
   Temps t1,t2,tsuma;
   tsuma=t1+t2;
   tsuma.mostrar();
}
```

Sobrecàrrega d'operadors en C++
Operadors binaris

```
Temps
Temps::operator+(const Temps &t)
{
  Temps aux;

  aux.minut = minut + t.minut;
  aux.hora = hora + t.hora;
  if (aux.minut >= 60)
   {
     aux.minut = aux.minut - 60;
     aux.hora++;
   }
  return aux;
}
```

```
int
main()
{
  Temps T1(12,24);
  Temps T2(4,45);

  T1 = T1 + T2;
//T1=T1.operator+(T2)
  T1.mostrar();
}
```

Sobrecàrrega d'operadors en C++
Exemple: classe Cadena

```
class Cadena{
 public:
  Cadena();
  Cadena(char *);
  ~Cadena();
  void mostrar() const;
 private:
  char *cadena;
  int longitud;
};

Cadena::Cadena(){
  cadena = NULL;
  longitud=0;
};
```

```
Cadena::Cadena(char *cad){
  cadena = new char[strlen(cad)+1];
  strcpy(cadena,cad);
  longitud = strlen(cad);
}
Cadena::~Cadena(){
 if(cadena!=NULL)
  delete[] cadena;
 longitud = 0;
}
void Cadena::mostrar() const{
  cout << "La cadena  '" << cadena;
  cout << "té long " << longitud;
}

// OBJETIVO:
int main(){
   Cadena C1("Primera"), C2("Segona"), C3;
   C3 = C1+C2;
 //C3.operator=(C1.operator+(C2));
   return (0);
}
```

Sobrecàrrega d'operadors en C++
Exemple: classe Cadena

```
Cadena
Cadena::operator+(const Cadena &c){
  Cadena aux;

  aux.cadena = new char[strlen(c.cadena)+strlen(cadena)+1];
  strcpy(aux.cadena,cadena);
  strcat(aux.cadena,c.cadena);
  aux.longitud = strlen(cadena)+strlen(c.cadena);
  return aux;
}

int main(){
   Cadena C1("Primera part"), C2("Segona Part"), C3;
   C3 = C1+C2; //C3.operator=(C1.operator+(C2));
   return (0);
}
```

Funcionaria bé esta suma sense definir res més, així com ocorria en Temps?

Sobrecàrrega d'operadors en C++
Exemple: classe Cadena i operador =

- **Solució: SOBRECARREGAR L'OPERADOR D'ASSIGNACIÓ**

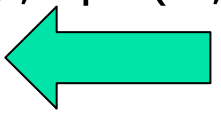
```
Cadena& Cadena::operator=(const Cadena &c){
 if(this != &c){
   if (cadena!=NULL) delete[] cadena;
   if (c.cadena!=NULL){
     cadena = new char[strlen(c.cadena)+1];
     strcpy(cadena,c.cadena);
     longitud = c.longitud;
   }
   else{
     cadena = NULL;
     longitud = 0;
   }
 }//end (this != &c)
 return (*this);
}
```

Sobrecàrrega d'operadors en C++
Exercici: operadors binaris

Donat el següent codi:

```
class TPunt {
   TPunt (int x=0, int y=0){
      this->x=x;
      this->y=y;
   };
private :
      int x,y;
};
```

```
int main(){
 TPunt p1(1,2), p2(3,4);
 p1+=p2;
 return(0);
}
```



Es demana:

- Declara (.h) i defineix (.cpp) la sobrecàrrega de l'operador += per a la classe TPunt.
- Fes el mateix suposant que no podem afegir funcions membre a la classe

Sobrecàrrega d'operadors en C++
Operadors d'entrada i eixida

- La possibilitat de sobrecàrrega dels operadors << (eixida) i >> (entrada) resol el problema tradicional de llibreries d'I/O pensades per a treballar exclusivament amb tipus predefinits (built-in)
- L'abstracció 'flux d'eixida' (ostream) representa una entitat que accepta una seqüència de caràcters. Aqueixa entitat pot després convertir-se en un fitxer, una finestra o una xarxa. L'abstracció 'flux d'entrada' (istream) representa una entitat que envia una seqüència de caràcters. Eixa entitat pot després convertir-se en un teclat, un fitxer o una xarxa.
- La llibreria estàndard proporciona una colecció de definicions sobrecarregades per a l'operador << y >>
  - `ostream& operator << (ostream& dest, int source);`
  - `ostream& operator << (ostream& dest, char source);`
  - `ostream& operator << (ostream& dest, char * source);`

  …

  oper esq oper dreta
  - Exemple

```
cout << "El resultat es " << x << '\n';  // (cout,char*)
        cout               << x << '\n';  // (cout,int)
                 cout      << '\n';       // (cout,char)
```

Sobrecàrrega d'operadors en C++
Operadors d'entrada i eixida

- L'extensió d'esta llibreria per a altres tipus de dades segueix el mateix patró:
  - `ostream & operator << (ostream& o, const <nouTipus>& t)`
  - `istream& operator >> (istream & i, <nouTipus>&t)`
  - Per a que esta funció tinga accés a la part privada/protegida de t, hem de definir-la com friend de la classe <nouTipus>.
    - Amb compte! En qualsevol cas ha de tractar-se d'una funció **no membre**, ja que l'operador de l'esquerra no és un objecte de la classe.

```
class Temps{
 friend ostream &operator<<(ostream &,const Temps &);
 friend istream &operator>>(istream &,Temps &);
 public:
  …
 private:
  int hora;
  int minut;
};
```

Sobrecàrrega d'operadors en C++
Exemple: operadors I/O en classe Temps

```
#include <iostream>
using namespace std;
…
ostream& operator<<(ostream &s, const Temps &t){
  s << t.hora << ":" << t.minut << endl; // 14:59
  return s;
}

istream& operator>>(istream &e,Temps &t){
  e >> t.hora >> t.minut; // 14 59
  return e;
}

void main(){
   Temps T1,T2(17,36),T3;
   cin >> T1; //operator>>(istream&, Temps&)
    // operator>>(cin,T1);
   T3 = T1 + T2;
   cout << T3<<endl; //operator<<(ostream, const Temps&)
    // operator<<(cout,T3);
}
```

Sobrecàrrega d'operadors en C++
Exemple: operadors I/O en classe TPunt

```
class TPunt {
public:
   TPunt (int, int);
private :
     int x;
     int y;
};

// OBJECTIU
int main(){
 TPunt p1(1,2), p2(3,4);
 cout << p1 << "y" << p2 << endl;
  //La cridada cout<<p1 s'interpreta com operator<<(cout, p1);
 return (0);
}
```

Sobrecàrrega d'operadors en C++
Exemple: operadors I/O en classe TPunt

```
// .h
class TPunt {
      friend ostream&
      operator<<(ostream&, const TPunt&);
      public:
        ...
      private :
        int x,y
};

// .cpp
ostream& operator<<(ostream &o, const TPunt &p) {
      o << "(" << p.x << "-" << p.y << ")"; // (3-2)
      return o;
}
```

Sobrecàrrega d'operadors en C++
Exemple: operadors I/O en classe TPunt

```
class TPunt {
public:
   TPunt (int, int);
private :
      int x;
      int y;
};

// OBJECTIU 2
int main(){
 TPunt p1(1,2), p2(3,4);
 //La cridada cin>>p1 s'interpreta com operator>>(cin, p1);
 cin >> p1;
 cin >> p2;
 cout << p1 << "y" << p2 << endl;
}
```

Sobrecàrrega d'operadors en C++
Exemple: operadors I/O en classe TPunt

```
istream&
operator>>(istream &i,  TPunt &p){
     // Llig en format (x-y)
     i.ignore(); // Salta el (
     i >> p.x;
     i.ignore(); // salta el -
     i >> p.y;
     i.ignore(); // salta el )
     return i;
}
```

Sobrecárrega d'operadors en C++
Exemple: operador [] en classe Vector

- Signatura operador claudàtor ([])
  - **<tipo>& operator[] (int index)**
- Exemple:

```
class Vector
{
  public:
    Vector( int ne=1) {
      num = ne;
      pV = new int[ num];
    };
    ~Vector(){ delete [] pV; pV=NULL;};
    int& operator[](int i) {return pV[ i];};

  private:
    int *pV; // punter al 1 element
    int num; // num de elements
};
```

Sobrecàrrega d'operadors en C++
Exemple: operador [] en classe Vector

```
int main(){
    int num,i;
    cout << "Número:"; cin>> num;
    Vector v1(num);
    for (i= 0; i< num; i++)
       v1[i] = rand();
    for (i= 0; i< num; i++) {
       cout << "v1["<< i<<"]="<<v1[i] << endl;
    }
    return (0);
}
```

Sobrecàrrega d'operadors en C++
Exercici: Classe Vector

- Definiu (.h) i implementeu (.cpp) vostra pròpia classe Vector de sencers amb les següents característiques:

Vector
...
+ Vector(int capacitat) +~Vector +Vector (const Vector &v) +operator=(const Vector&v): Vector& +operator==(const Vector &v): bool <<const>> +operator!=(const Vector&v): bool <<const>> +operator[](const int indice) const int& << const>> //rvalue: ha de controlar index fora rang <<friend>> operator>>(Vector &v) <<friend>> operator<< (const Vector &v)

Sobrecàrrega d'operadors en C++
Exercici: Classe Vector

- Modifica el que siga necessari en la declaració i/o definició de l'operador assignació per a que controle els següents casos:
  - (a=a) : ha de deixar "a" intacte
  - (a=b)=c (ha de ser erroni).
- Si encara no ho has fet, redefineix el constructor de còpia en base a la definició de l'operador =
- Modifica el codi associat a l'operador [] per a que puga actuar de lvalue en expresions del tipus
  - Vector v;v[3]=2;

Sobrecàrrega d'operadors en C++
Exemple: classe TTablaAsoc

Objectiu:

Implementeu una TablaAssociativa que continga parells clau-valor (string-int), i permeteu l'accés mitjançant la clau al valor corresponent

E.g. int v=mitabla["clave"]

- Exemple: Ús d'una taula Associativa per a guardar el nombre de vegades que introduïsc cada paraula

```
int main(){
    char paraula[256];
    TTaulaAssoc t(512);

    for (register int i=0;i<10;i++){
        cin>>paraula;
        t[paraula]++;
    }
    cout<<t<<endl;
    return (0);
}
```

```
hola
hola
hola
hola
hola
hola
hola
hola
hola
adios
hola:9
adios:1
```

Sobrecàrrega d'operadors en C++
Exemple: classe TTaulaAssoc

```
struct Tupla{
        char * nomb; //per defecte visibilitat pública
        int valor;
    };

class TTaulaAssoc {
 public:
   TTaulaAssoc(int i);
   int& operator[] (const char* c);
   friend ostream& operator<<(ostream& o,TTaulaAssoc& t);
   ~TTaulaAssoc();
 private:
     Tupla * pt;
     int maxLong;
     int primerLliure;
     TTaulaAssoc (const TTaulaAssoc &t);
     TTaulaAssoc& operator= (const TTaulaAssoc &t);
};
```

Sobrecàrrega d'operadors en C++
Exemple: classe TTaulaAssoc

```
TTaulaAssoc::TTaulaAssoc(int i){
    maxLong= (i<16)?16:i;
    primerLliure=0;
    pt= new Tupla[maxLong];
};

TTaulaAssoc::~TTaulaAssoc(){
  Tupla *ptaux;
    for (ptaux=pt;ptaux-pt<primerLliure;ptaux++){
      delete ptaux->nomb;
    }
    delete [] pt; pt=NULL;
};

ostream& operator<< (ostream& o,TTaulaAssoc& t){
    for (register int i=0;i<t.primerLliure;i++)
        o<<t.pt[i].nomb<<":"<<t.pt[i].valor<<endl;
}
```

Sobrecàrrega d'operadors en C++
Exemple: classe TTaulaAssoc

```
int& TTaulaAssoc::operator[] (const char* c){
//retorna una referència al valor, creant abans la clau si és necessari
    register Tupla* ptaux;
    if (primerLliure>0){
        for (ptaux=&pt[primerLliure-1];pt<=ptaux;ptaux--)
           if (strcmp(c,ptaux->nomb)==0)
              return ptaux->valor;
    }
    //no es troba la paraula a la taula
    if (primerLliure==maxLong){
        Tupla* ptaux2=new Tupla[maxLong*2];
        for (register int i=0;i<maxLong;i++) ptaux2[i]=pt[i];
        delete[] pt; pt=ptaux2; maxLong*=2;
    }

    //en este punt segur que hi ha espai pel nou string
    ptaux=&pt[primerLliure++];
    ptaux->nomb=new char(strlen(c)+1);
    strcpy(ptaux->nomb,c);
    ptaux->valor=0;
    return ptaux->valor;
};
```

Alternatives a sobrecàrrega: Funcions amb nombre variable d'arguments

- **Les funcions amb nombre variable d'arguments *(poliàdiques)*** es troben en distints llenguatges
  - P. ex. printf de C y C++
- Si el nombre màxim és conegut, en C++ i Delphy Pascal podem acudir a la definició de valors per defecte per a paràmetres opcionals
  - P. ex. int sum (int e1, int e2, int e3=0, int e4=0);
    - Respondria a sum (1,2), sum (1,2,3) y sum(1,2,3,4);
- Si el nombre màxim no és conegut, en C/C++ hem d'acudir a la llibreria stdarg:

```
#include <stdarg.h>
int sum (int numEle, ...){ //... indica núm variable d'args
    va_list ap; //tipus predefinits en stdarg.h
    int resp=0;
    va_start(ap, numEle); //inicialitza llista amb el num arguments
    while (numEle>0){
        resp += va_arg(ap,int);
        numEle--;
    }
    va_end(ap);
    return (resp);
}
```

Alternatives a sobrecàrrega:
Coerció i Conversió

- En algunes ocasions la sobrecàrrega pot sustituir-se per una operació semànticament diferent: **La COERCIÓ**
  - Un valor d'un tipus es converteix DE MANERA IMPLÍCITA en un valor d'altre tipus distint
    - P.ex. Coerció implícita entre reals i sencers: la seua adició pot ser interpretada al menys de tres formes distintes:
      - Quatre funcions distintes: integer+integer, integer+real, real+integer, real+real: només sobrecàrrega
      - Dues funcions distintes: integer+integer, real+real (si algun és real, l'altre es coerciona a real): coerció+sobrecàrrega
      - Una sola funció: real+real: només hi ha coerció i no sobrecàrrega
  - El principi de sustitució en els LOO introdueix a més una forma de coerció que no es troba en els llenguatges convencionals

Alternatives a sobrecàrrega:
Coerció i Conversió

- Quan el canvi en tipus es sol·licitat de manera explícita pel programador parlem de **CONVERSIÓ**
  - L'operador utilitzat per a sol·licitat esta conversió s'anomena CAST
    - Exemple:
      - `float x; int i;`
      - `x= i + x; // COERCIO`
      - `x= ((double) i) + x; //CONVERSIO`
    - Un cast pot canviar la representació interna de la variable (convertir un sencer en un real, p.ex.) o canviar el tipus sense canviar la representació (p.ex., convertir un punter a fill en un punter a pare).
    - Dintre d'una classe C podem definir el *cast*:
      - D'un tipus extern al tipus definit per la classe: implementació de constructor amb només un paràmetre del tipus des del qual volem convertir.
      - Del tipus definit per la classe a altre tipus distint en C++: implementació d'un operador de conversió.

Alternatives a sobrecàrrega:
Coerció i Conversió

- E.x. suposem la següent classe:

```
class Fraccio{
  public: Fraccio(int n,int d):num(n),den(d){};
       int numerador(){return num;};
       int denominador(){return den;};
       Fraccio operator*(Fraccio &dreta){…};
  private: int num, den;
};
```

- Per a poder realitzar la conversió:

```
Fracció f=(Fraccio) 3;
//pasar de sencer a fracció
```

Seria necessari afegir a la classe fracció el constructor:

```
Fracció (int i) {num=i;den=1;};
```

```
Esto al mateix temps permetria operacions com
   Fracció f(2,3),f2; //dos terços
   f2=f*3; // donaria 6/3
```

Alternatives a sobrecàrrega:
Coerció i Conversió

- Si a més desitgem realitzar la conversió:

```
double d=f*3.14;
```

  Seria necessari afegir a la classe fracció l'operador:

```
operator double() { //SENSE TIPUS RETORN
  return (numerador()/(double)denominador());
};
```

- Òbviament, quan la coerción (conversió implícita), la conversió (explícita) i la substitució de valors es combinen en una sola sentència, l'algoritme que ha d'utilitzar el compilador per a resoldre una funció sobrecarregada pot ser molt complex.

Polimorfisme en jerarquies d'herència

Sobrecàrrega en jerarquies d'herència

- Mètodes amb el mateix nomb, la mateixa signatura de tipus i enllaç estàtic:
  - **Shadowing (refinament/reemplaçament)**: les signatures de tipus són les mateixes en classes pare i filles, però el mètode a invocar es decideix en temps de compilació
    - En C++ implica que el mètode no es va declarar com virtual en classe pare.
- Mètodes amb el mateix nomb i distinta signatura de tipus i enllaç estàtic:
  - **Redefinició**: classe filla defineix un mètode amb el mateix nomb que el pare però amb **distinta signatura de tipus**.
    - Model MERGE: SOBRECÀRREGA
    - Model JERÀRQUIC: NO HI HA SOBRECÀRREGA

Polimorfisme en jerarquies d'herència

Sobrecàrrega en jerarquies d'herència

- Dues formes de resoldre la **redefinició** en LOO:
  - Model ***merge*** (Java):
    - Els diferents significats que es troben en tots els àmbits actualment actius s'uneixen per a formar una sola col·lecció de mètodes.
  - Model **jeràrquic** (C++):
    - Quan es troba un àmbit en el qual el nomb està definit, la coincidència mès propera en eixe àmbit serà la seleccionada per a resoldre la cridada.

```
class Pare{
  public: void exemple(int a){cout<<“Pare”;};
};
class Filla: public Pare{
  public: void exemple (int a, int b){cout<<“Filla”;};
};
int main(){
  Filla h;
  h.exemple(3); // OK en Java
                  //però ERROR DE COMPILACIÓ EN C++
  return (0);
}
```

Polimorfisme en jerarquies d'herència
Sobreescriptura (Overriding, polimorfisme *d'inclusió*)

- Diem que un mètode en una classe derivada sobreescriu un métode en la classe base si els dos mètodes tenen el mateix nomb, la mateixa signatura de tipus i enllaç dinàmic.
  - Un mateix nom de missatge associat a diverses implementacions (com la sobrecàrrega) PERÒ:
    - en classes relacionades mitjançant jerarquies d'herència
    - La signatura de tipus ha de ser EXACTAMENT la mateixa
    - Els mètodes sobreescrits poden suposar un reemplaçament del comportament (mètode independent del del pare, igual que la sobrecàrrega) o un **refinament** (mètode no independent)
    - La resolució del mètode a invocar es produeix en **temps d'execució** (enllaç dinàmic o *late-binding*) en funció del tipus dinàmic del receptor del missatge.
  - El tipus de qualsevol altre argument passat juntament amb el missatge sobreescrit generalment no juga cap paper en el mecanisme de selecció del mètode sobreescrit.
- La sobreescriptura és important quan es combina amb el ppi. de substitució.
  - Una variable del tipus del pare pot, pel principi de substitució, ser declarada com de tipus Pare però contindre en realitat un valor del tipus fill. Quan un missatge que es correspon amb un mètode sobreescrit es passa a aquesta variable, el mètode que serà executat és el proporcionat pel fill, NO pel pare.

Polimorfisme en jerarquies d'herència
Sobreescriptura

- En alguns llenguatges (Java, Smalltalk) la simple existència d'un mètode amb el mateix nomb i signatura de tipus en classe base i derivada indica sobreescriptura.
- En altres llenguatges (Object Pascal), la classe derivada ha d'indicar que sobreescriu un mètode.
- Altres llenguatges (C#, Delphi Pascal) exigeixen que tant la classe base com la derivada l'indiquen.
- En C++ és la classe base la que ha d'indicar explícitament que un mètode pot ser sobreescrit (encara que eixa marca no obliga a que ho siga)
  - Paraula reservada **`virtual`**.

```
class Pare{
    public: virtual int exemple(int a){cout<<"pare";};
};
class Filla : public Pare{
    public: int exemple (int a){cout<<"filla"};
};
```

Polimorfisme en jerarquies d'herència
Sobreescriptura i mètodes diferits (virtuals purs o abstractes)

- En els llenguatges fortament tipats, si volem sobreescriure un mètode en les classes filles per a poder invocar- lo mitjançant una variable polimòrfica, és necessari que eixe mètode esté defini en la classe pare. No obstant això, és possible que eixa classe pare no puga definir cap comportament per mancar de la informació necessària.
- Solució: mètodes diferits o abstractes
  - C++: virtual pur

```
class Forma{
  public: virtual void dibuixar()=0;
  //mec. abstracció: associa el concepte dibuixar() amb la classe
  //Forma i no amb cadascuna de les filles
 };
  class Circulo: public Forma{
        public: void dibuixar(){...}
  };
```

- Els mètodes virtuals purs (de sobreescriptura necessària, i que implementen l'especialització) sovint conviuen amb mètodes no sobreescribibles, base del reús de codi.

- **És important distingir entre Sobreescriptura, Shadowing i Redefinició (vistos en Sobrecàrrega)**

  - **Sobreescriptura**: la signatura de tipus per al missatge és la mateixa en classe base i derivada, però el mètode s'enllaça amb la cridada en temps d'execució (en C++ el mètode es declara com `virtual` en la classe pare).

  - **Shadowing**: la signatura de tipus per al missatge és la mateixa en classe base i derivada, però el mètode s'enllaça en temps de compilació (en funció del tipus de la variable receptora).

  - **Redefinició**: La classe derivada defineix un mètode amb el mateix nomb que en la classe base i amb **distinta signatura de tipus**.

Variables Polimòrfiques (Polimorfisme d'assignació)

- Una variable polimòrfica és aquella que pot referenciar més d'un tipus d'objecte
  - Pot mantindre valors de distints tipuss en distints moments d'execució del programa.
- En un llenguatge dèbilment tipat totes les variables són potencialment polimòrfiques
- En un llenguatge fortament tipat la variable polimòrfica és la materialització del principi de substitució.

Variables Polimòrfiques. Tipus

- **Variables polimòrfiques simples**
  - `Vaixell *b[10];` //array de punters a Vaixell que en realitat
    //conté punters a les distintes classes derivades

- **Variable polimòrfica com receptor de missatge**
  - **`this`**: en cada classe representa un objecte d'un tipus distint.
  - Especialment potent quan es combina amb sobreescriptura.

```
class Vaixell{
    virtual int getNumPeces()=0;
    void colocaPeça(Peça &p, Coordenada);
    bool colocaVaixell(Vaixell &b){
      Coordenada caleat;
      Direccio daleat;
      for (int x=0;x<this->getNumPeces();x++){
       // this apunta a objectes de classe derivada
       colocaPeça(b[x],caleat);
       caleat.incrementa(daleat);
      }
    } ...
}
```

Variables Polimòrfiques. Tipus

- **Downcasting** (polimorfisme invers):
  - Procés de ‘desfer’ el ppi. de substitució. Pot donar problemes.
  - En C++:

```
child *c=dynamic_cast<Child *>(aParentptr);
if (c!=0) {…
//nul si no és legal, no nul si OK
```

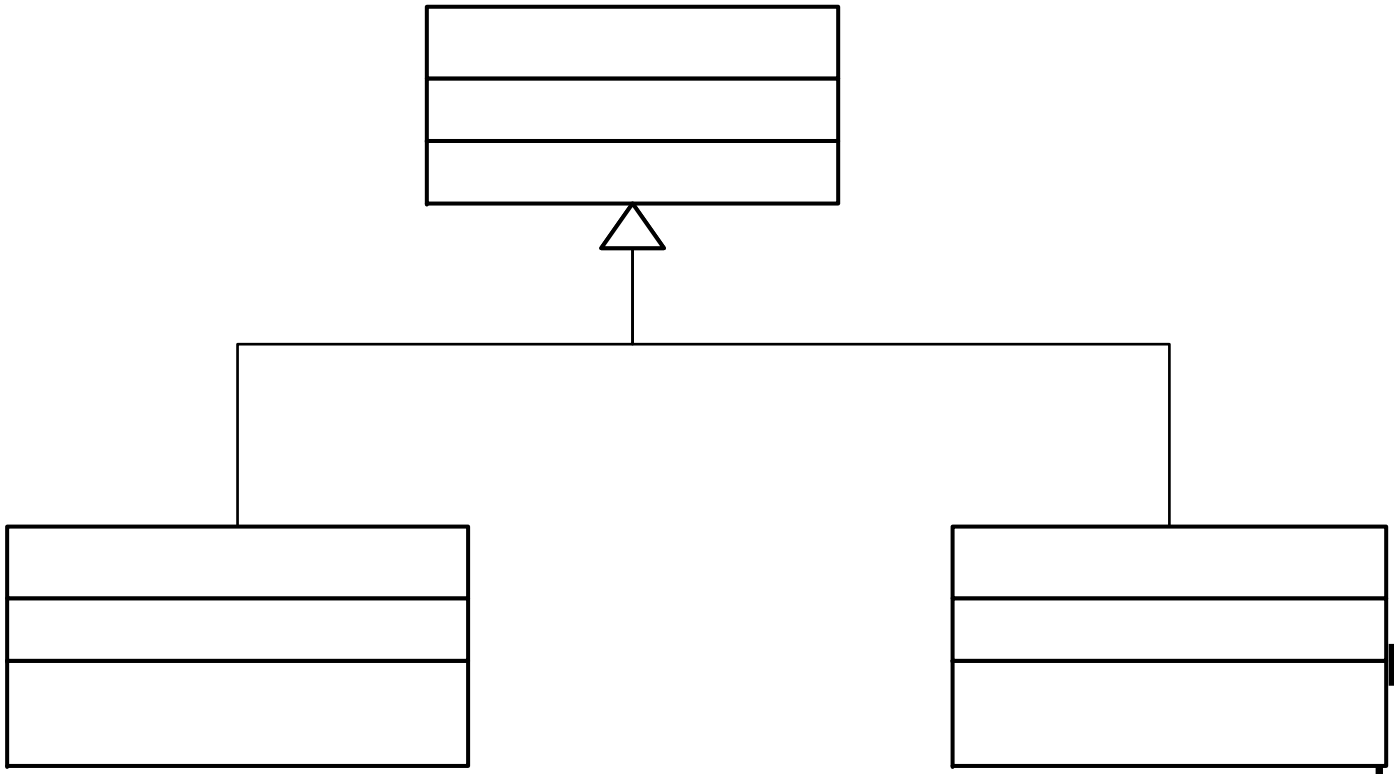
- Variable polimòrfica utilitzada com argument d’un mètode: **polimorfisme pur** o mètode polimòrfic.
  - Només un mètode pot ser utilitzat amb un nombre potencialment ilimitat de tipus distints d’argument.

Variables Polimòrfiques. Polimorfisme pur

- Exemple de polimorfisme pur

```
class Aplicacio(){
    public:
      int getNumVaixellsAplicacio(){
        return numSub+numAcor+numDest+numPort;}
    bool crearColocarVaixells(){
      b=new Barco*[numVaixells];
      for (int x=0;x<numSub;x++)
        b[i]=new Submari();
      for (int x=numSub;x<numSub+numAcor;x++)
        b[i]=new Cuirassat();
      …
      for (int x=0;x<getNumVaixellsAplicacio();x++)
         t.colocaVaixell(*b[i]); //bool Tauler::colocarVaixell(Vaixell &b)
          //espera un objecte de tipus vaixell, i rep objectes de tipus
          //Cuirassat, Submari, etc. POLIMORFISME PUR
    }
    private:
      Tauler t;
      Vaixell **b;
      int numSub,numCuir,numDest,numPort;
}
```

Exemple: Ús de polimorfisme i jerarquia de tipus



Exemple: Ús de polimorfisme i jerarquia de tipus

```
class Persona {
  public:
     Persona(string n)   {nomb=n;};
     ~Persona();
     string getDades() {return (nomb);};
    ...
  private:
     string nomb;
};
```

Exemple: Ús de polimorfisme i jerarquia de tipus

```
class Empleat: public Persona {
  public:
        Empleat(string n,string e): Persona(n){empresa=e;};
        ~Empleat(){};
        string getDades(){
         return Persona::getDades()+“treballa en " + empresa);
        };
        ...
  private:
        string empresa;
};
class Estudiant: public Persona {
  public:
        Estudiant(string n,string c): Persona (n){carrera=c;};
        ~Estudiant(){};
        string getDades(){
         return(Persona::getDades() + " estudia " + carrera);
        };
        ...
  private:
        string carrera;
};
```

Refinament

Refinament

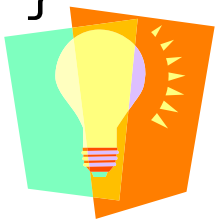
Exemple: Ús de polimorfisme i jerarquia de tipus

```
int main(){

  Empleat empleatDesc(“Carles”,”Bugaderia”);
  Persona pers(“Joan”);
  empleatDesc=pers;
  cout<<empleatDesc.getDades()<<endl;

}
```

Quina eixida donarà este programa?

Exemple: Ús de polimorfisme i jerarquia de tipus

```
int main(){
  int i;
  Empleat *empleatDesc=NULL;
  Estudiant *estudiantDesc= NULL;
  Persona *pers=new Persona(“Josep”);
  empleatDesc=pers;
  estudiantDesc=pers;
  cout<<empleatDesc->getDades()<<endl;
  cout<<estudiantDesc->getDades()<<endl;
  estudiantDesc=NULL;
  empleatDesc=NULL;
  delete (pers);
  pers=NULL;
}
```



Quina eixida donarà este programa?
Es produeix un enllaçat estàtic o dinàmic?

Exemple: Ús de polimorfisme i jerarquia de tipus

```
int main(){
  Empleat un("Carles", “bugaderia");
  Estudiant dos("Joan", "empresarials");
  Persona desc("desconeguda");
  desc=un; //li assigne empleat
  cout << desc.getDades() << endl;
  return (0);
}
```

Quina eixida donarà este programa?
Es produeix un enllaçat estàtic o dinàmic?

Exemple: Ús de polimorfisme i jerarquia de tipus

```
int main(){
  Empleat *emp= new Empleat("Carles", “bugaderia");
  Estudiant *est= new Estudiant("Joan",
  "empresarials");
  Persona *pers;
  pers = emp;
  cout<<emp->getDades()<<endl;
  cout<<est->getDades()<<endl;
  cout << pers->getDades() << endl;
  //Ací s’hauria de lliberar memòria: com?
}
```

Quina eixida donarà este programa?
Es produeix un enllaçat estàtic o dinàmic?

Exemple: Ús de polimorfisme i jerarquia de tipus

```
class Persona {
  public:
     Persona(string n)   {nomb=n;};
     ~Persona();
     virtual string getDades() {
       return (nomb);
  };
   ...
  private:
     string nomb;
};
```

Pot ser resolt dinàmicament

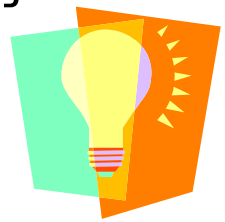
Exemple: Ús de polimorfisme i jerarquia de tipus

```
int main(){
   Empleat un("Carles", “bugaderia");
   Estudiant dos("Joan", "empresarials");
   Persona desc("desconeguda");
   desc=un; //un empleat és una persona
   cout << desc.getDades() << endl;
}
```

Quina eixida donarà este programa?
Es produeix un enllaçat estàtic o dinàmic?

Exemple: Ús de polimorfisme i jerarquia de tipus

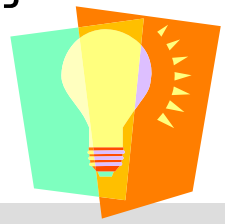
```
int main(){
  Empleat *un= new Empleat("Carles", “bugaderia");
  Estudiant *dos= new Estudiant("Joan",
  "empresarials");
  Persona *desc;
  desc = un;
  cout << uno->getDades()<<endl;
  cout << dos->getDades()<<endl;
  cout << desc->getDades() << endl;
  desc=NULL;
   delete(un); un=NULL;
   delete(dos); dos=NULL;

}
```



Quina eixida donarà este programa?
Es produeix un enllaçat estàtic o dinàmic?

Exemple: Ús de polimorfisme i jerarquia de tipus

```
int main1(){
    Empleat *un= new Empleat("Carles", “bugaderia");
    Persona *desc = un;
    cout << desc->getEmpresa() << endl;
    desc=NULL;  delete(un); un=NULL;
    return (0);
}

int main2(){
    Persona *desc = new Persona("Carles");
    Empleat *empPtr=(Empleat*) desc;
    cout<<empPtr->getEmpresa()<<endl;
    empPtr=NULL;  delete(desc); desc=NULL;
    return (0);
}
```



Quina eixida donarà main1? I main2?
Es produeix un enllaçat estàtic o dinàmic?

Polimorfisme

Conseqüències per a la definició de destructors en jerarquies

```
int main(){
  Empleat *un= new Empleat("Carles", “bugaderia");
  Estudiant *dos= new Estudiant("Joan",
  "empresarials");
  Persona *desc;
  desc = un;
  cout<<un->getDades()<<endl;
  cout<<dos->getDades()<<endl;
  cout << desc->getDades() << endl;
  delete desc; desc=NULL;
  delete dos; dos=NULL;
  un=NULL;
}
```

Alliberarà correctament la memòria este main? Què faries per a evitar l’error sense canviar el codi del main?

Polimorfisme

Implementació interna en C++

- Les funcions virtuals són una mica menys eficients que les funcions normals.
  - Cada classe amb funcions virtuals disposa d'un vector de punters cridat v_table. Cada punter correspon a una funció virtual, i apunta a la seua implementació més convenient (la de la pròpia classe o, en cas de no existir, la de l'ancestre més proper que la tinga definida)
  - Cada objecte de la classe té un punter ocult a eixa v_table.

Polimorfisme

Avantatges

- El polimorfisme fa possible que un usuari puga afegir noves classes a una jerarquia sense modificar o recompilar el codi original.
  - Estableixes classe base
  - Defineixes noves variables i funcions
  - Ensambles amb el codi objecte que tenies
  - Els mètodes de la classe base poden ser reutilitzats amb variables i paràmetres de la classe derivada.
- Permet programar a nivell de classe base utilitzant objectes de classes derivades (possiblement no definides encara): Tècnica base de les llibreries/frameworks

Repàs
Ús de *virtual* en C++

- Resolució de conflictes en herència múltiple
- Especificació de possibilitat d'enllaçat dinàmic
  - Necessari (funcions virtuals pures)
  - Possible (funcions virtuals on existeix refinament/reemplaçament en les classes filles)

Genericitat
Motivació

- La genericitat és altre tipus de polimorfisme
- Per a ilustrar la idea de la genericitat es proposa un exemple:
  - Suposeu que volem implementar una funció *màxim,* on els paràmetres poden ser de distint tipus (int, double, float)

    Què faríeu?

Genericitat
Motivació

- Solució:

```
double maxim(double a, double b){
    if (a > b)
        return a;
    else
        return b;
};
int main (void){
    int y,z;
    float b,c;
    double t,u;
    double s= maxim(t,u);
    double a= maxim((double)b,(double)c);
    double x= maxim((double)y,(double)z);
}
```

Genericitat

Motivació

- Ara suposeu que creem una classe TCompte, i també volem poder comparar-les amb la funció maxim().

Genericitat

Motivació

1º Sobrecarregar l'operador > per a TCompte
2º Sobrecarregar la funció máxim per a TCompte

```
TCompte maxim(TCompte a, TCompte b){
   if (a>b)
      return a;
   else
      return b;
}
void main (void){
    double s,t,u;
    TCompte T1,T2,T3;
    s= maximo(t,u);
    T1= maxim(T2,T3);
}
```

Conclusió: Tenim dues funcions màxim definides, una per a double i altra per a TCompte, però el codi és el mateix. L'única diferència són els paràmetres de la funció i el valor retornat per aquesta.

Genericitat
Motivació

- I si no sabem a priori els tipus que altres van a crear per a ser comparats?
  - Necessitaríem una funció genèrica que ens servira per a qualsevol tipus sense necessitat de tenir la funció duplicada.

Genericitat
DEFINICIÓ

- *Propietat que permet definir una classe o una funció sense tindre que especificar el tipous de tots o alguns dels seus membres o arguments.*

- La seua utilitat principal és la d'agrupar variables de les quals el tipus base no està predeterminat (p. ex., llistes, cues, piles etc. de objectes genèrics).
- És l'usuari el que indica el tipus de la variable quan crea un objecte d'eixa classe.
- En C++ esta característica va aparéixer a finals dels 80.

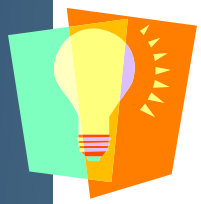
Genericitat en C++:
Templates

- Utilitzen la paraula clau ***template*** (plantilla)
- Dos tipus de plantilles:
  - **Plantilles de funcions**: són útils per a implementar funcions que acepten arguments de tipus arbitrari.
  - **Plantilles de classe**: la seua utilitat principal consisteix en agrupar variables el tipus de les quals no està predeterminat (classes contenidores)

Un argument genèric

```
template <class T>
T maxim(T a, T b){
    if (a > b)
        return a;
    else
        return b;
};
```

```
int main (void){

TCompte a,b,c;
int x,y,z;
z= maxim(x,y);  //OK
c= maxim(a,y);  //INCORRECTE
c= maxim(a,b);  //?
}
```



Què hauria de fer per a, aplicant la funció *màxim*, poder obtindre el compte amb més saldo?

Més d'un argument genèric

```
template <class T1, class T2>
T1 sumar (T1 a, T2 b){
    return (a+b);
};
```

```
int main (void){

   int eun=1;
   int edos=2;
   char cun='a';
   char cdos='d';

   cout<<sumar(eun,cun);
   cout<<sumar(cun,eun);

   TCompte c;
   cout<<sumar(eun,c);

}
```

Genericitat
Sobrecàrrega de funcions genèriques

- La sobrecàrrega de les funcions genèriques és igual a la de les funcions normals.

```
template <class T>
T min(T a, T b){...}      //OK

template <class T>
T min(int a, T b){...}    //OK

template <class T>
T min(int a, int b){...}  //OK, pero inútil

// pero...

int main() {
  int a,b,c;
  c=min(a,b) //?
}
```

Genericitat

Classes genèriques en C++

- Utilitzen la paraula clau *template*
- Dos tipus de plantilles:
  - **Plantilles de funcions**: són útils per a implementar funcions que accepten arguments de tipus arbitrari.
  - **Plantilles de classe**: la seua utilitat principal consisteix en agrupar variables el tipus de les quals no està predeterminat

Genericitat

Exemple Classe Genèrica

- A continuació es planteja un exemple d'una classe genèrica *vector,* este vector va a contindre elements de tipus genèric, no es coneixen a priori.

Genericitat

Exemple Classe Genèrica

```
template <class T>
class vector {
    private:
        T *v;
        int tam;
    public:
        vector(int);
        T& operator[](int);
};
void main (void){
    vector <int> vi(10);
    vector <float> vf(20);
    vector <double> vd(50);
}
```

Genericitat

Exemple Mètodes d'una Classe Genèrica

```
template <class T>
vector<T>::vector(int size) {
    tam=size;
    v=new T[size];
};

template <class T>
T& vector<T>::operator[](int i) {
    return(v[i]);
};
```

Genericitat

Exemple d'ús d'una Classe Genèrica

- Creació de l'objecte:

```
vector<double> d(30);

vector<char> *p=new vector<char>(50);
```

Genericitat

Exemple: Pila d'Objectes Genèrica

- El següent exemple és una pila que contindrà objectes de qualsevol tipus, per a això l'anem a definir com una classe genèrica o plantilla. En aquesta pila podrem introduir nous elements i imprimir el contingut de la mateixa.

Genericitat

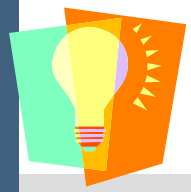
Exemple: Pila d'Objectes Genèrica

```
template <class T>
class Pila{
    public:
        Pila(int nelem=10);
        void apilar (T);
        void imprimir();
        ~Pila();

    private:
        int nelements;
        T* info;
        int cim;
        static const int limit=30;
};
```

Genericitat

Exemple: Pila d'Objectes Genèrica

```
template <class T>
Pila<T>::Pila(int nelem){
    if (nelem<=limit){
        nelements=nelem;
    }
    else {
        nelements=limit;
     }
    info=new T[nelements];
    cim=0;
};
```



Com implementaríeu el mètode *apilar*?

Genericitat

Exemple: Pila d'Objectes Genèrica

```
template <class T>
void Pila<T>::apilar(T elem){
    if (cim<nelements)
        info[cim++]=elem;
}

template <class T>
void Pila<T>::imprimir(){
    for (int i=0;i<cima;i++)
       cout<<info[i]<<endl;
}

template <class T>
Pila<T>::~Pila(){
  delete [] info; info=NULL;
}
```

Genericitat

Exemple: Pila d'Objectes Genèrica

```
#include "TPila.h"
#include "TCuenta.h"


int main(){

    Pila <TCompte> pComptes(6);
    TCompte c1("Cristina",20000,5);
    Tcompte c2("Antoni",10000,3);
    pComptes.apilar(c1);
    pComptes.apilar(c2);
    pComptes.imprimir();
```

```
Pila <char> pchar(8);
    pchar.apilar('a');
    pchar.apilar('b');
    pchar.imprimir();
} //end main
```

De manera anàloga, plantegeu una llista d'objectes genèrica.

Genericitat

Herència en classes genèriques

- Es poden derivar classes genèriques d'altres classes genèriques:

Classe base genèrica:

```
template <class T>
class Pila{
    public:
        void posar(T a):
    private:
        T* buffer;
        int cap;
};
```

Classe derivada genèrica:

```
template <class T>
class doblePila: public Pila<T>
{
    public:
        void posar2(T a);
};
```

Genericitat

Herència en classes genèriques

- Es pot derivar una classe no genèrica d'una genèrica

Classe base genèrica:

```
template <class T>
class Pila{
    public:
        void posar(T a):
    private:
        T* buffer;
        int cap;
};
```

Genericitat

Herència en classes genèriques

Classe derivada no genèrica:

```
class munto: public Pila<int>
{
    public:
        void posar2(int a);
};
```

- En C++, no existeix relació alguna entre dues classes generades des de la mateixa classe genèrica.

Genericitat

Constants en plantilles

- Els arguments d'una plantilla no estan restringits a ser classes definides per l'usuari, també poden ser tipus de dades existents.
- Els valors d'aquests arguments es converteixen en constants en temps de compilació per a una instanciació particular del template.
- També es poden usar valors per defecte per a aquests arguments.

Genericitat

Exemple Constants en plantilles

```
template <class T, int size=100>
class Array{
    public:
    T& operator[](int index) ;
    int length() const { return size; }
    private:
   T array[size];
};
int main(){
    Array <Compte,30> t;
 int i=10;
  Array <Compte,i> t2; // ERROR, no constant.
}
```

Polimorfisme
Resum

- Amb les funcions virtual i el polimorfisme és possible dissenyar i implementar sistemes que siguen més fàcilment extensibles. És possible escriure programes perquè processen objectes de tipus que tal vegada no existisquen quan el programa està sota desenvolupament.
- La programació polimórfica amb funcions virtual pot eliminar la necessitat de lògica de switch. El programador pot utilitzar el mecanisme de funció virtual per a realitzar automàticament la lògica equivalent, evitant d'aquesta forma els tipus d'errors associats amb la lògica de switch. El codi client que pren decisions sobre tipus d'objectes i representacions és senyal d'un pobre disseny de classes.
- Les classes derivades poden proporcionar la seua pròpia implementació d'una funció virtual de classe base, en cas de ser necessari, però si no ho fan s'utilitza la implementació de la classe base.
- Si s'invoca una funció virtual fent referència a un objecte específic per nom i utilitzant l'operador punt de selecció de membres, la referència es resol en temps de compilació (a açò se li crida enllaç estàtic), i la funció virtual que es diu és aquella definida per a (o heretada per) la classe d'aqueix objecte particular.

Polimorfisme
Resum

- Hi ha moltes situacions en les quals és útil definir classes per a les quals el programador mai pretén instanciar cap objecte. A tals classes se'ls crida classes abstractes. Degut al fet que només s'utilitzen com classes base, normalment les esmentarem com classes base abstractes. No és possible instanciar en un programa cap objecte d'una classe abstracta.
- Les classes de les quals es poden instanciar objectes són cridades classes concretes.
- Una classe es fa abstracta mitjançant la declaració que una o més de les seues funcions virtual siguen pures. Una funció virtual pura és aquella que té un inicializador de = 0 en la seua declaració.
- Si una classe es deriva d'una classe que té funcions virtual pures sense proporcionar una definició per a aqueixa funció virtual pura en la classe derivada, aquesta funció segueix sent pura en la classe derivada. Per consegüent, la classe derivada també és una classe abstracta (i no pot tenir cap objecte).
- C++ permet el polimorfismo —l'habilitat que objectes de diferents classes relacionats per herència responguen en forma diferent a la mateixa cridada de funció membre.

Polimorfisme
Resum

- El polimorfisme s'implementa per mitjà de funcions virtual.
- Quan es fa una petició, per mitjà d'un apuntador o referència a classe base, perquè s'utilitze una funció virtual, C++ tria la funció sobreposada correcta en la classe derivada adequada que està associada amb l'objecte.
- Mitjançant l'ús de funcions virtual i el polimorfismo una cridada de funció membre pot causar diferents accions, depenent del tipus de l'objecte que rep la cridada.
- Encara que no podem instanciar objectes de classes base abstractes, sí podem declarar apuntadors cap a classes base abstractes. Eixos apuntadors poden utilitzar-se per a permetre manejos polimòrfics d'objectes de classes derivades quan eixos objectes s'instancien a partir de classes concretes.
- Regularment s'agreguen nous tipus de classes als sistemes. Les noves classes són emmotlades mitjançant l'enllaç dinàmic (també cridat enllaç tardà). No és necessari saber el tipus d'un objecte en temps de compilació perquè es puga compilar una cridada de funció virtual. En temps d'execució la cridada de funció virtual es fa coincidir amb la funció membre de l'objecte que la rep.

Polimorfisme
Resum

- L'enllaç dinàmic permet que els ISV distribuïsquen software sense revelar els seus secrets propis. Les distribucions de programari poden consistir solament d'arxius d'encapçalat i arxius objecte. No és necessari revelar gens del codi font. Els desenvolupadors de software després poden utilitzar l'herència per a derivar noves classes a partir de les quals proporcionen els ISV. El software que funciona amb les classes que proporcionen els ISV continuarà funcionant amb les classes derivades i utilitzarà (per mitjà d'enllaç dinàmic) les funcions virtual sobreposades que es proporcionen en aqueixes classes.
- L'enllaç dinàmic requereix que en temps d'execució la cridada a una funció membre virtual siga dirigida cap a la versió de la funció virtual adequada per a eixa classe. Una taula de funcions virtuals, cridada vtable, està implementada com un arranjament que conté apuntadors de funció. Cada classe que conté funcions virtual té una vtable. Per a cada funció virtual de la classe, la vtable té una entrada que conté un apuntador a funció cap a la versió de la funció virtual que cal utilitzar per a un objecte d'aqueixa classe. La funció virtual que cal utilitzar per a una classe particular podria ser la funció definida en aqueixa classe, o una funció heretada directa o indirectament d'una classe base que està en un nivell més elevat en la jerarquia.

Polimorfisme
Resum

- Quan una classe base proporciona una funció membre virtual, les classes derivades poden sobreposar a la funció virtual però no estan obligades a fer-lo. Per tant, una classe derivada pot utilitzar la versió de la classe base de la funció membre virtual, i açò estaria indicat en la vtable.
- Cada objecte d'una classe que té funcions virtual conté un apuntador cap a la vtable d'eixa classe. L'apuntador a funció adequat en la vtable s'obté i desreferència per a completar la cridada en temps d'execució. Aquesta recerca en la vtable i la desreferenciació de l'apuntador requereixen una sobrecàrrega ínfima del temps d'execució, que en general és menor que el millor codi client possible.
- Declare com virtual al destructor de la classe base si la classe conté funcions virtual. Açò fa que tots els destructors de les classes derivades siguen virtual encara que no tinguen el mateix nom que el destructor de la classe base. Si un objecte de la jerarquia és destruït explícitament per mitjà de l'aplicació de l'operador delete a l'apuntador de classe base a un objecte de classe derivada, s'invoca el destructor de la classe adequada.
- Qualsevol classe que tinga un o més apuntadors 0 en el seu vtable és una classe abstracta. Les classes que no tinguen un apuntador a vtable que siga 0 (menge Point, Circle i Cylinder) són classes concretes.

Polimorfisme
Bibliografia

- Timothy Budd. **An Introduction to Object Oriented Programming**. 3rd Edition. Addison-Wesley, 2002