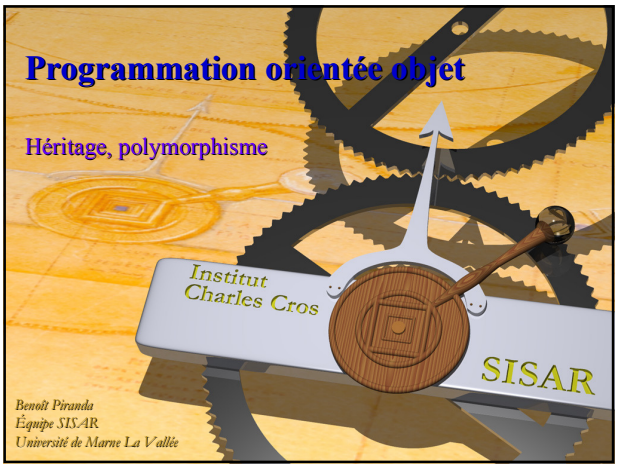
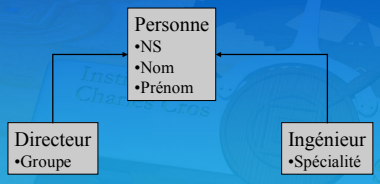


Principe

- Objet
  - Instance d'une classe
  - Regroupe des données et des traitements
- Notion d'héritage
  - Un objet peut contenir toutes ses primitives
  - Un objet peut être un sous ensemble d'un objet plus général
    - Relation « est un »
      - ✓ Un chien est un animal
      - ✓ Un ouvrier est un employé
    - Partager ses primitives
      - ✓ Attributs
      - ✓ Méthodes

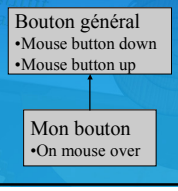
Intérêts de l'héritage

- Aspect statique
  - Sur des classes que l'on réalise, mettre en commun
    - Des données
    - Des traitements



Intérêts de l'héritage

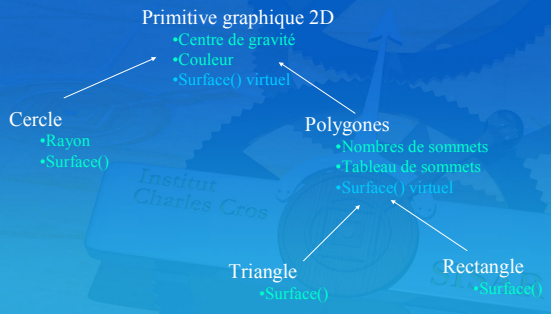
- Aspect statique 2
  - Compléter une classe existante par des données ou traitements particuliers
    - Profiter de la définition de l'objet général
    - L'ajout de classes dérivées se fait sans modification de la classe de base
    - Très courant dans les interfaces graphiques



Héritage

- Un objet hérite de son père
  - Ses attributs (communs à tous les fils)
  - Ses méthodes (générales à tous les fils)
- Un objet admet des primitives particulières
  - Ses attributs spécifiques
  - Ses méthodes propres
- Un objet surcharge
  - Les méthodes virtuelles du père

Exemple d'héritage



Droit et héritage

- **Dérivation publique**
  - Notion de « est un »
  - La classe fille a accès aux éléments du père
    - Public
    - Protected
- **Dérivation privée**
  - Méthodes de la classe de bases utilisées par la classe dérivée

Définition d'un héritage

- **En C++**
  - Déclaration de la classe fille
    - class fille : public mere
  - Constructeur appelle le constructeur de la classe mère
    - fille::fille(…) : mere(…)

Définition de l'héritage en C++

```
class geometrique
{ protected :
   double x,y;
   geometrique() { this->x = 0.0; this->y = 0.0; }
public :
   geometrique(double x,double y)
};

class cercle : public geometrique
{ private :
   double rayon;
   public :
   cercle(double x,double y,double r) : geometrique(x,y)
   { rayon = r;
   }
};
```

Exemple d'héritage

```
class Personne
{
protected :
  int id;
  string nom,prenom;
  int jour,mois,annee;
public :
  Personne(int id);

  void setNom(string nom);
  void setPrenom(string prenom);
  void setDDN(int jour,int mois,int annee);

  int getId() { return id; };
  string getNom() { return nom; };
  string getPrenom() { return prenom; };
  void getDDN(int &jour,int &mois,int &annee;
};

class Client:public Personne
{
private :
  double achat;
public :
  Client(int id):Personne(id) { this->achat = 0.; };

  void setAchat(double a) { this->achat = a; };
  double getAchat() { return this->achat; };
};
```

Méthodes virtuelles

- **Méthodes pouvant être surchargées par la classe fille**
  - Si surcharge
    - La méthode de la classe fille
  - Sinon
    - La méthode la plus basse définie dans l'héritage au-dessus de la classe
- **Les méthodes virtuelles admettent un traitement par défaut**
  - Appelé si non surchargé dans une classe fille

Exemple de méthode virtuelle

```
class geometrique
{ protected :
       double x,y;
       geometrique() { this->x = 0.0; this->y = 0.0; }
public :
       geometrique(double x,double y)
       { this->x = x; this->y = y; }
       virtual string qui() { return "geometrique"; };
};

class cercle : public geometrique
{ private :
   double rayon;
   public :
   cercle(double x,double y,double r) : geometrique(x,y)
   { rayon = r;
   }
   string qui() { return "cercle"; };
};
```

Méthodes virtuelles pures

- Méthodes définies uniquement au niveau des feuilles de la structure arborescente de l'héritage
  - Nœuds interne
    - Prototype permettant la généricité
  - Méthodes devant être obligatoirement surchargées
  - Exemple
    - La méthode surface ne doit pas être appelée avec l'objet géométrique
    - Syntaxe
      - ✓ virtual type methode(…)=0;
- Pas de corps de méthode

Exemple de méthode virtuelle pure

```
class geometrique
{ protected :
        double x,y;
        geometrique() { this->x = 0.0; this->y = 0.0; }
public :
        geometrique(double x,double y)
        { this->x = x; this->y = y; }
        virtual double surface()=0;
};

class cercle : public geometrique
{ private :
  double rayon;
  public :
  cercle(double x,double y,double r) : geometrique(x,y)
  { rayon = r;
  }
  double surface() { return 3.14159265*rayon*rayon; };
};
```

Classe virtuelle

- Une classe virtuelle est une classe qui ne contient que des méthodes virtuelles pures
  - Permet de regrouper une famille de classes
  - Classes héritant de cette classe virtuelle

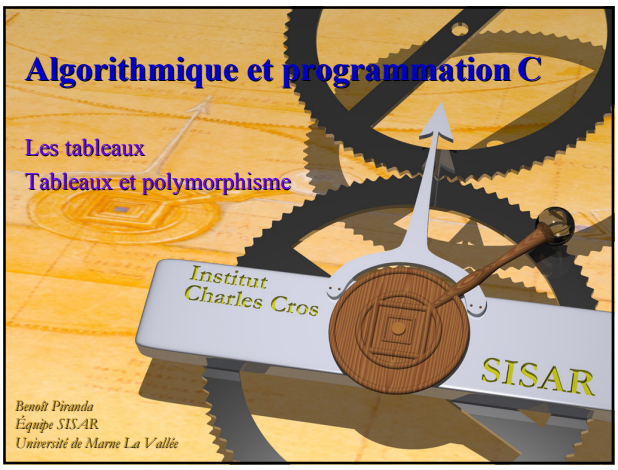
Polymorphisme

- Principe
  - Un objet d'une sous-classe est en même temps un objet de la classe de base.
  - Un objet du type du père peut être alloué comme un objet fils.
- Les méthodes
  - Accessibles : les méthodes virtuelles du père.
  - Effectivement exécutées : la méthode la plus proche du fils dans l'arborescence de l'héritage

Exemple de polymorphisme

```
geometrique *g1,*g2;

g1 = new cercle(0.0,0.0,1.0);
cout << g1->surface() << endl;
delete g1;

g2 = new rectangle(0.0,0.0,1.0,1.0);
cout << g2->surface() << endl;
delete g2;
```

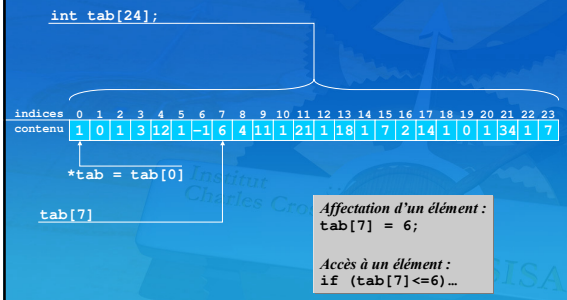
```
3.14159
1
```



Les tableaux

- **Principe**
  - Regrouper plusieurs variables de même type
    - Chaque élément est directement accessible en indiquant son numéro d'ordre
    - Un tableau est un bloc mémoire contigu
    - Sa taille est fixe (définie lors de sa création)
- **En C/C++ :**
  - Un tableau de *n* éléments est indicé de *0* à *n-1*
  - Un tableau est un pointeur sur une zone mémoire de *n* * la taille d'un élément

Les tableaux



Affectation d'un élément :
`tab[7] = 6;`

Accès à un élément :
`if (tab[7]<=6)…`

Allocation statique d'un tableau

- **Définition : on précise le nombre d'éléments**
  - Aspect statique, le nombre d'élément est connu à la compilation

```
double vecteur[3];
:
int i;
for (i=0; i<3; i++)
{ vecteur[i] = 0.0;
}
:
```

Initialisation d'un tableau

- **Possibilité d'initialisation à la déclaration**
  - Liste de valeurs
    - De même type que le tableau
    - Nombre inférieur ou égale à la taille
    - Détermination automatique de la taille du tableau

```
int v1[]={1,2,3,4};
char v2[2]={'A','B','C'};
int v3[5]={1,2,3};
```

génère une erreur
complète par des 0

Allocation dynamique d'un tableau

- **Définition : réservation d'un pointeur**
  - Réservation d'une zone mémoire pouvant contenir *n* éléments, lorsque *n* est connu

```
double *tab;
int n;

cin >> n;

tab = new double[n];
```

- **Désallocation**
  - libération de la mémoire associée au pointeur

```
delete [] tab;
```

Parcours d'un tableau

Exemple : compter le nombre de valeurs 1 dans un tableau

```
#define NB_ELEMENTS     10
int i,n,tab[NB_ELEMENTS]={0,1,2,1,4,2,1,1,8,1};

n=0;
for (i=0; i<NB_ELEMENTS; i++)
{ if (tab[i]==1) n++;
}
// n contient le nombre de 1 dans tab
```

Définir un tableau d'objets statiques

- Idée :
  - Mémoriser une liste de personnes dans un tableau
  - On connaît le nombre maximum de personnes au moment de la compilation
- Première méthode :
  - On réserve l'ensemble de la mémoire dès le début
  - On modifie le contenu des cellules du tableau au cours de la saisie
  - Inconvénient : mémoire excessive utilisée.

```
#define NB_PERSONNES    50
Personne tab[NB_PERSONNES];
```

Définir un tableau d'objets dynamiques

- Idée :
  - Mémoriser une liste de personnes dans un tableau
  - On connaît le nombre maximum de personnes au moment de la compilation
  - Mémoire optimum : on veut les mémoriser les personnes au fur et à mesure de la saisie
- 1 : allocation d'un tableau de pointeur
  - Aucune mémoire n'est réservée pour les personnes
  - On a juste un « tableau d'accès à des personnes »

```
#define NB_PERSONNES    50
Personne *tab[NB_PERSONNES];
```

Définir un tableau d'objets

- 2 : Ajout d'une personne
  - Allocation de la mémoire pour cette personne
  - Faire pointer une case du tableau pointe sur cette nouvelle personne
  - Il faut une variable pour mémoriser le nombre de personnes effectivement enregistrées dans le tableau

```
Personne *nouv;

nouv = new Personne(id,nom,prenom);

tab[0] = nouv;
```

Algorithme, insertion placée

- Insérer un élément à sa place dans un tableau trié
  - Rechercher la position d'insertion
  - Décaler la fin du tableau
  - Insérer le nouvel élément
  - Augmenter le nombre de personnes effectivement mémorisées

Insérer un élément à une place donnée

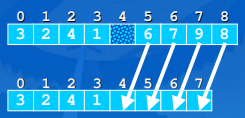
- Insérer un élément *v* à la position *indice*
  - Modifier le contenu du tableau pour tous les éléments d'indice supérieur
  - Placer la nouvelle valeur *v* à sa place
  - Incrémenter le nombre d'éléments

Programme C++

```
bool Ensemble::ajouter(Personne &P)
{ int i=0,j;

  if (nbrePersonnes==nbrePersonnesMax) return false;

  // recherche de l'indice de la premiere personne d'id supérieur
  while (i<nbrePersonnes && P.getId()>tabPersonnes[i].getId())
  { i++;
  }

  // décaler la fin du tableau
  for (j=nbrePersonnes; j>i; j--)
  { tabPersonnes[j] = tabPersonnes[j-1];
  }

  // inserer la personne
  tabPersonnes[i]=P;

  nbrePersonnes++;
  return true;
}
```

Supprimer un élément

- Supprimer l'élément d'indice *indice*
  - Modifier le contenu du tableau pour tous les éléments d'indice supérieur
  - Modifier le nombre d'éléments



Exemple de suppression

```
bool Ensemble::supprimer(int v)
{ int i=0;
  // recherche de la position de v
  while (i<nbrePersonnes && tabPersonnes[i].getId()!=v)
  { i++;
  }
  if (i==nbrePersonnes) return false; // pas trouvé
  // suppression
  delete tabPersonnes[i];

// décalage des éléments pointés
  i=indice;
  while (i<nbrePersonnes-1)
  { tabPersonnes[i]=tabPersonnes[i+1];
  }
  nbrePersonnes--;
}
```

Polymorphisme dans les tableaux

- Définir des classes filles à la classe personne
  - Client (achats)
  - Collaborateur (domaine)
  - Chaque élément de la liste peut être un client ou un collaborateur

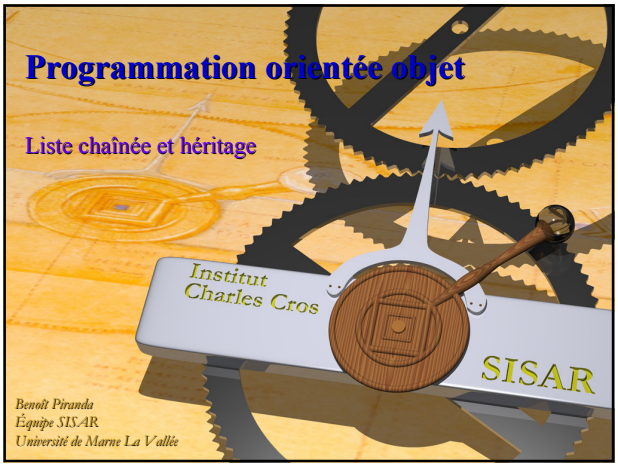
```
class Client:public Personne
{
private :
  double achat;
public :
  Client():Personne() { this->achat = 0.; };
  Client(int id):Personne(id) { this->achat = 0.; };
  Client(int id,string nom,string prenom):Personne(id,nom,prenom)
   { this->achat=0.; };
  Client(int id,int jour,int mois,int
   annee):Personne(id,jour,mois,annee) { this->achat=0.; };

  void setAchat(double a) { this->achat = a; };
  double getAchat() { return this->achat; };
};
```

Polymorphisme dans les tableaux

```
Client P1(1,"Duchemin","Robert"),P2(2,"Dupond","Paul");
Collaborateur P3(3,"Dupuis","Pierre");
Personne *P;
P1.setNom("Durand");
P1.setPrenom("Pierre");
P1.setDDN(3,10,2001);

Ensemble E(15);

P = new Client(5,"Dumoulin","Etienne");

E.ajouter(P2);
E.ajouter(4,"Dubois","Paul",11,4,1968);
E.ajouter(P1);
E.ajouter(P3);
E.ajouter(*P);
E.afficher();
```

Programmation orientée objet

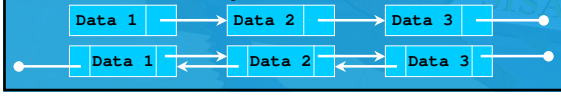
Liste chaînée et héritage



Benoît Piranda
Équipe SISAR
Université de Marne La Vallée

Notion de liste chaînée

- Structure de données
  - Dynamique
    - On ne connaît pas à priori le nombre d'éléments
  - Chaînage mémoire des données
    - Données (attributs de classe)
    - Lien vers les données suivantes (pointeur)
      - ✓ Simple
      - ✓ Multiple
  - Notion d'ordre
    - Lien vers le suivant
    - Lien vers le précédent



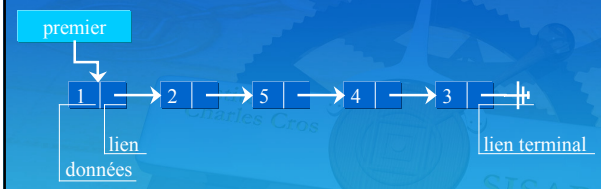
Structure de chaînage

- **Données de la liste**
  - **Pour chaque élément (structure C)**
    - Données stockés dans l'élément
    - Pointeurs
      - ✓ liens vers les éléments liés
      - ✓ NULL si extrémité de la liste
  - **Pour accéder à la liste**
    - Variables extérieures à la liste
    - Pointeur sur certains éléments clés
      - ✓ Premier
      - ✓ Dernier
      - ✓ Régulier (table de hachage)

Structure de liste simplement chaînée

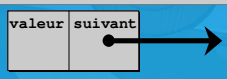
- **Liste simplement chaînée d'éléments**
  - **Pour chaque élément**
    - Données de l'élément
    - Un pointeur (lien) vers l'élément suivant
  - **Pour accéder à la liste**
    - Une variable extérieure à la liste
      - ✓ Pointeur sur le premier élément
- **Avantage / Inconvénient**
  - **Structure très simple**
    - Peu de données ajoutées pour le chaînage
  - **Parcours unidirectionnels**
    - Séquentiel

Exemple de liste simplement chaînée



Structure de liste simplement chaînée

```
class element
{ public :
  int valeur;
  element *suivant;
  element(int v) {valeur=v; };
};

class Liste
{ element* premier;
public :
  Liste() { premier=NULL; };
}
```



Méthode d'ajout d'un élément

```
void Liste::ajouterDebut(element *e)
{ e->suivant = premier;
  premier = e;
}
```

Structure de liste simplement chaînée

```
void main(void)
{ Liste L;

  L.ajouterDebut(new element(3));
  L.ajouterDebut(new element(4));
  L.ajouterDebut(new element(5));
  L.ajouterDebut(new element(2));
  L.ajouterDebut(new element(1));
}
```

Comparaison liste / tableau

- Besoin identique
  - Ensemble de données homogènes
  - Lorsqu'on ne connaît pas la quantité par avance
- Avantages
  - Optimise la taille des structures de données
  - Manipulations plus rapides / tableau
    - Ajout d'un élément
    - Suppression d'un élément
    - Déplacement d'un élément
- Inconvénients
  - Certaines manipulations plus compliquées / tableau
    - Pas d'accès directe à un élément

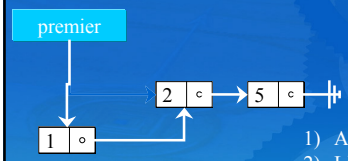
Opérations simples sur les listes

- Avantage du non-ordonnancement des données en mémoire
  - Insertion d'un élément
  - Insertion d'une sous-liste
  - Suppression d'un élément

Algorithme d'ajout d'un élément

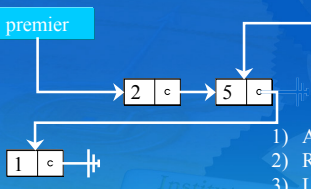
- Principe
  - Allocation d'un nouvel élément
  - Lien entre l'élément précédent et l'élément créé
  - Maintien de la cohérence de la liste
    - Lien entre le nouvel élément et « l'ancien suivant du précédent »
    - Affectation du pointeur de début de liste (si nécessaire)
- 3 cas d'ajout d'élément dans une liste
  - ajout en tête de liste
  - ajout en queue de liste
  - ajout à l'intérieur de la liste

Ajout en tête de liste



1) Allocation d'un nouvel élément
2) Lien avec le début de liste
3) Modification du premier élément

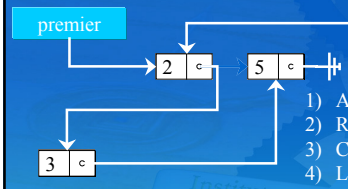
Ajout en queue de liste



1) Allocation d'un nouvel élément
2) Recherche du dernier élément
3) Lien vers le nouvel élément

```
void Liste::ajouterQueue(element *e)
{ element *ptr = premier;
  if (ptr==NULL) return;

②  while (ptr->suivant!=NULL) ptr=ptr->suivant;
③  ptr->suivant = e;
}
```

Algorithme d'ajout d'un élément placé



1) Allocation d'un nouvel élément
2) Recherche de l'élément précédent
3) Chaînage du nouvel élément
4) Lien vers le nouvel élément

```
void Liste::ajouterPlace(element *e)
{ element *ptr = premier;

  if (ptr==NULL) return;
②  while (ptr->suivant!=NULL && ptr->suivant->valeur<e->v)
     ptr=ptr->suivant;
③  e->suivant = ptr->suivant;
④  ptr->suivant = e;
}
```

Ajout d'un élément placé

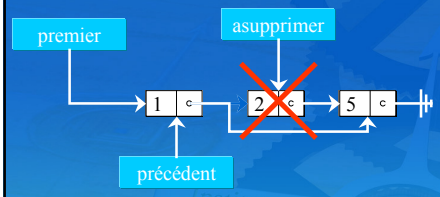
- **Question :**
  - Que ce passe-t-il si *v* est la valeur la plus grande de la liste ?

```
void Liste::ajouterPlace(element *e)
{ element *ptr = premier;

  if (ptr==NULL) return;
  while (ptr->suivant!=NULL && ptr->suivant->valeur<e->v)
    ptr=ptr->suivant;
  e->suivant = ptr->suivant;
  ptr->suivant = e;
}
```

Algorithme de suppression d'un élément

- **Principe**
  - Libérer la mémoire occupée par un élément
  - Maintien de la cohérence de liste
    - Lien entre le précédent et le suivant de l'élément supprimé
    - Affectation du pointeur de début de liste (si nécessaire)
- **3 cas de suppression d'élément dans une liste**
  - Suppression en tête de liste
  - Suppression en queue de liste
  - Suppression à l'intérieur de la liste

Algorithme de suppression d'un élément



1) Recherche du précédent l'élément à supprimer
2) Garder un pointeur sur l'élément à supprimer
3) Chaînage de la liste sans l'élément à supprimer
4) Désallocation

Algorithme de suppression d'un élément

```
void Liste::supprimer (int v)
{ element *ptr,*asupprimer;
  if (premier->valeur==v)
  { asupprimer = premier;
    premier = premier->suivant;
    delete asupprimer;
  } else
  { ptr = premier;
    while (ptr->suivant!=NULL && ptr->suivant->valeur!=v)
      ptr=ptr->suivant;
    if (ptr->suivant!=NULL)
    { asupprimer = ptr->suivant;
      ptr->suivant = ptr->suivant->suivant;
      delete asupprimer;
    }
  }
}
```

Algorithmes de parcours

- **Lecture des données**
  - Parcours complet de la liste
- **Algorithme itératif**
  - À chaque élément (élément courant), on applique un traitement
  - On passe à l'élément suivant en modifiant le pointeur indiquant l'élément courant
- **Algorithme récursif**
  - Parcourir une liste c'est :
    - Si la liste est vide ne rien faire
    - Traiter l'élément courant puis parcourir la sous-liste suivante

Algorithmes de parcours

- **Avantage de la méthode itérative**
  - Économique en ressources
    - Un pointeur : l'élément courant

```
ptr = premier;
tant que (ptr!=NULL) faire
  traiter l'élément pointé par ptr
  ptr = ptr->suivant; // passage à l'élément suivant
fait
```

```
ptr = premier;
while (ptr!=NULL)
{ // traiter l'élément pointé par ptr
  ptr = ptr->suivant; // passage à l'élément suivant
}
```

Algorithme de parcours

- **Méthode récursive**
  - **Parcourir une liste c'est :**
    - Si la liste est vide ne rien faire
    - Traiter l'élément courant
    - Parcourir la sous-liste suivante

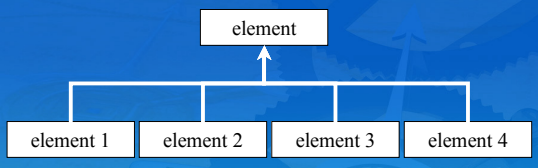
```
procedure parcours(element *ptr) faire
 si (ptr!=NULL) alors
  traiter l'élément pointé par ptr
  parcours(ptr->suivant); // parcours de la liste suivante
 fin si
fait
```

```
void parcours(element *ptr)
{ if (ptr!=NULL)
  { // traiter l'élément pointé par ptr
    parcours(ptr->suivant);
  }
}
```

Liste chaînée générique

- **Utilisation du polymorphisme**
  - **Liste chaînée d'objets de la classe mère**
    - Allocation dynamique des éléments
    - Choix du type des objets parmi les fils
  - **Permet d'obtenir une liste chaînée d'objets de types différents**

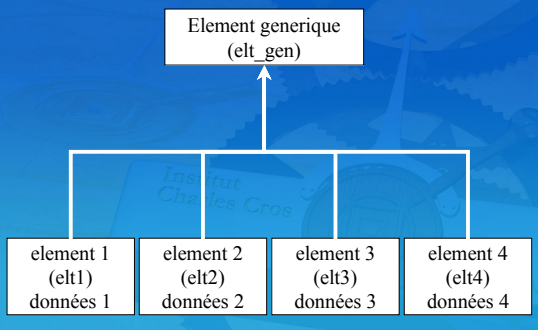
Liste chaînée générique



Structure de liste simplement chaînée

- **Liste simplement chaînée d'éléments différents**
  - **Élément :**
    - Pointeur sur un élément de la classe mère
    - Liens
  - **Données**
    - Au niveau des classes filles
  - **Pour accéder à la liste**
    - Une classe contenant les liens sur certains éléments clés
- **Avantage / Inconvénient**
  - **Ajout d'un type de donnée ne modifie pas la gestion du chaînage**

Liste chaînée générique



Liste chaînée générique

```
class elt_gen
{
protected :
  elt_gen() {};
  ~elt_gen() {};
  virtual void afficher()=0;
};
```

```
class elt4 : public elt_gen
{ char t[50];
protected :
  elt4(char *t)
  { strncpy(this->t,t,50); };
  ~elt4() {};
  void afficher() { cout << t; };
};
```

```
class elt1 : public elt_gen
{ int i;
protected :
  elt1(int i) { this->i=i; };
  ~elt1() {};
  void afficher() { cout << i; };
};
```

```
class elt2 : public elt_gen
{ double d;
protected :
  elt2(double d) { this->d=d; };
  ~elt2() {};
  void afficher() { cout << d; };
};
```

```
class elt3 : public elt_gen
{ char c;
protected :
  elt3(char c) { this->c=c; };
  ~elt3() {};
  void afficher() { cout << c; };
};
```

Structure de liste d'éléments différents

```
class element
{ elt_gen *ptr;
  element *suivant;
public :
  element(elt_gen *e,element *s=NULL)
  { ptr=e;
    suivant = s;
  }
  ~element()
  { delete ptr;
  };

  void affichage()
  { ptr->afficher();
    cout << endl;
    if (suivant) suivant->affichage();
  }
};
```

```
class liste
{ element *premier;
public :
    liste() { premier=NULL; };
    ~liste();
    void ajouterTete(elt_gen*);
    void affichage() { premier->affichage(); };
};

void liste::ajouterTete(elt_gen *e)
{ premier = new element(e,premier);
}

void main()
{ liste L;
  L.ajouterTete(new elt1(2));
  L.ajouterTete(new elt2(3.0));
  L.ajouterTete(new elt3('a'));
  L.ajouterTete(new elt4("coucou"));
  L.affichage();
}
```

```
coucou
a
3.0
2
```