Plus de C++

ISIMA - ZZ2 - 2009

Christophe Duhamel
Andréa Duhamel

ISIMA Forme Normale de Coplien

- 4 méthodes
  - constructeur par défaut `A::A ();`
  - constructeur par recopie `A::A (const A &);`
  - opérateur d'affectation `A & A::operator= (const A &);`
  - destructeur `A::~A ();`
- Comportement par défaut : gestion bit à bit
- Adapté pour des classes simples
- Inapproprié lorsque
  - gestion de mémoire dynamique, pointeurs
  - gestion de fichier, socket, buffer I/O

2

ISIMA Création d'objets (Rappel)

- Trois classes d'allocation (comme en C)
  - automatique : variable locale sur la pile
  - statique : variable globale, variable locale statique
  - dynamique : variable allouée sur le tas
    - allocation avec **`new`** + constructeur
    - destruction avec **`delete`**
- Gestion mémoire
  - statique et automatique : système
  - dynamique : construction / destruction à la main

3

ISIMA Éléments nécessaires (1/2)

- Constructeur par défaut
  - construit un objet par défaut
  - syntaxe : `A::A();`
  - ex : initialisation dans un tableau

```
Point * nuage;
nuage = new Point[taille];
```

- Constructeur par recopie
  - construit un objet par clonage
  - syntaxe : `A::A (const A &);`
  - ex : paramètre passé par valeur

référence obligatoire !
sinon crée objet A tempo
→ boucle infinie

4

ISIMA Éléments nécessaires (2/2)

- Opérateur d'affectation
  - recopie un objet dans un autre
  - syntaxe : `A &A::operator= (const A &);`
  - copie hors initialisation, gestion plus complexe

retour par référence
pour un chaînage éventuel

- Destructeur : libère les ressources de l'objet
  - libère les ressources de l'objet
  - syntaxe : `A::~A ();`

5

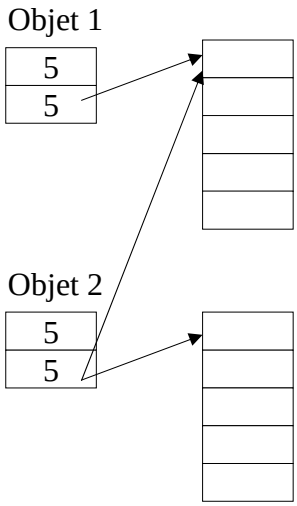
ISIMA Constructeur par recopie

- Objet transmis par valeur en argument à une fonction/méthode
- Objet renvoyé par valeur comme résultat d'une fonction/méthode
- Objet initialisé lors de sa déclaration avec un autre objet de même type

6

ISIMA Pourquoi c'est necessaire?

```
class A
{
  protected :
    int taille;
    int * tableau;
  public :
    A(int n) : taille(n)
    {
      tableau = new int[taille];
    }
    exemple(A a){…}
};
```

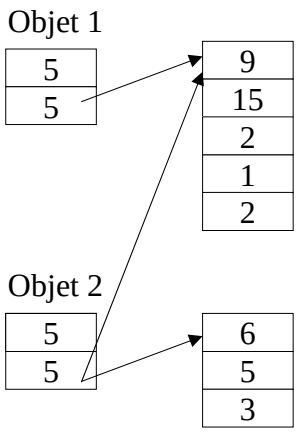



```
A objet1(5);
exemple(objet1);
```

7

ISIMA Affectation : problème voisin

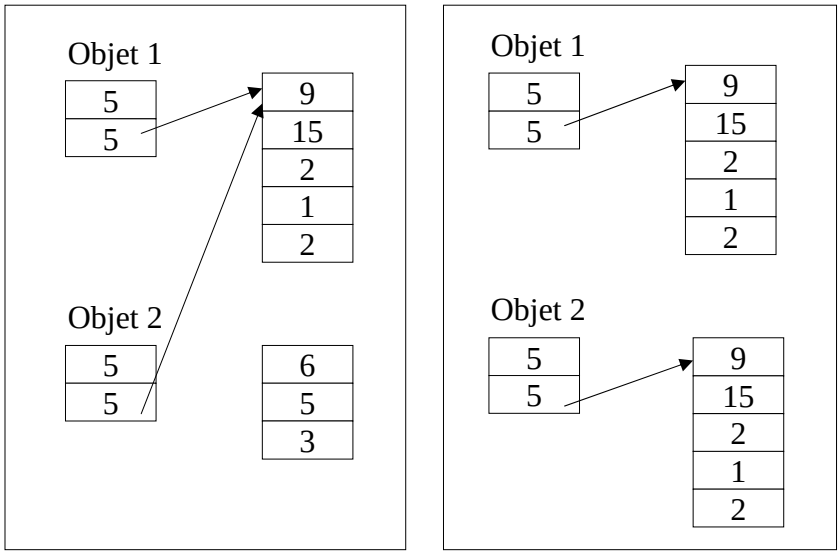
```
class A
{
  protected :
    int taille;
    int * tableau;
  public :
    A(int n) : taille(n)
    {
      tableau = new int[taille];
    }
    exemple(A a){…}
};
```




```
A objet1(5), objet2(3);
objet2 = objet1;
```

8

ISIMA Affectation : problème voisin



9

ISIMA Une classe **`Chaîne`**

- Problème : gestion dynamique de la longueur
- Solution
  - allocation dynamique d'un tableau de caractères
  - un attribut taille et un attribut tableau
- Gestion complexe de mémoire
  - à la mort de l'objet, effectuer la désallocation
  - lors de la copie d'une chaîne dans une autre
    - ne pas recopier le pointeur sur le tableau (copie physique/ fausse copie)
    - copier le tableau (copie logique/duplication)

10

ISIMA Déclaration de **Chaîne**

```
#ifndef __CHAINE_H__
#define __CHAINE_H__

class Chaine
{
  private:
    int   taille;
    char *tableau;

  public:
    Chaine(int t=0);
    Chaine(const char *str);
    Chaine(const Chaine &);
    Chaine & operator=(const Chaine &);
    ~Chaine();
    int getTaille(void) const;
    const char * getStr();
};

#endif
```

11

ISIMA Définition de **Chaîne** (1/2)

```
#include "chaine.h"

Chaine::Chaine (int t) : taille(t ? t+1 : 0)
{
  tableau = (taille ? new char[taille] : 0);
}

Chaine::Chaine (const char * str)
{
  if (str)
  {
    taille = strlen(str) + 1;
    tableau = new char[taille];
    strcpy(tableau,str);
  }
  else
  {
    taille = 0;
  }
}

Chaine::~Chaine ()
{
   if (taille) delete [] tableau;
}
```

Constructeur avec paramètre par défaut

constructeur sur chaîne

destructeur

12

ISIMA Définition de **Chaîne** (2/2)

```
Chaine::Chaine (const Chaine & c)
{
  taille = c.taille;
  if (taille)
  {
    tableau = new char[taille];
    strcpy(tableau, c.tableau);
  }
}
Chaine & Chaine::operator= (const Chaine & c)
{
  if (this != &c)
  {
    if (tableau)
      delete [] tableau;
    taille = c.taille;
    if (taille)
    {
      tableau = new char[taille];
      strcpy(tableau, c.tableau);
    }
  }
  return *this;
}
```

constructeur par recopie

opérateur d'affectation

important !

factoriser

pour le chaînage des opérations : `c3 = c2 = c1;`

13

ISIMA Initialisation ou affectation ?

- Principe

  une initialisation se fait uniquement dans une définition

- exercice

```
T t1;
T t2(params);
T t3(t1);
T t4 = t1;
T t5();
t1 = t4;
```

```
T::T();
T::T(lst_params);
T::T(const T &);
T::T(const T &);
T t5(void);
T & T::operator=(const T &);
```

14

ISIMA Passage d'objets par référence (rappel)

- Lors d'un passage par valeur
  - création d'un objet temporaire
  - initialisation par le constructeur par recopie
- Toujours passer les objets par référence
  - évite une création d'objet inutile
  - préserve le polymorphisme
  - passer en référence constante si objet non modifiable
- Lors du renvoi d'un objet
  - passer par référence si on renvoie l'objet courant (**`this`**)
  - sinon, toujours par copie (car destruction de l'objet local)

15

ISIMA La surcharge d'opérateurs

- La langage C réalise déjà la surdéfinition de certains opérateurs, par exemple :
  - (a + b) le symbole "+" effectue une opération qui dépends du type de a et b…
  - (*) peut désigner une multiplication ou une indirection (a = *ptr).
- Surdéfinir des opérateurs existant (unaire ou binaire) pour qu'ils portent sur au moins un objet.
- Exemple : soit a et b du type nombres complexes, on va pouvoir donner signification à des expressions, telles que (a+b), (a-b), (a*b), (a,/b),…

16

ISIMA La surcharge d'opérateurs

- Forme de polymorphisme faible
  - appliquée aux opérateurs du langage
  - adapter les opérateurs à de nouveaux types
  - maintenir une syntaxe « simple » pour l'utilisateur
- Deux types d'opérateurs
  - méthode : `b + c` → `b.operator+(c)`
  - fonction : `b + c` → `+(b,c)`
- Principe

définir en méthode si `this` a un rôle prépondérant

17

ISIMA La surcharge d'opérateurs

- Opérateur α externe : fonction
  - syntaxe : `T operator`α` (const T &, const T &);`
  - utilisation : `t1` α `t1` ↔ `operator`α` (t1,t2)`
  - surtout opérateurs dyadiques (2 paramètres)
  - aucun n'est prépondérant
- Opérateur α interne : méthode
  - syntaxe : `T T::operator`α` (const T &);`
  - utilisation : `t1` α `t2` ↔ `t1.operator`α`(t2)`
  - opérateurs monadiques et dyadiques
  - **`this`** est prépondérant

18

ISIMA La surcharge d'opérateurs

- Opérateurs surchargeables
  - en méthode : **=**, **+=**, etc, **()**, *****, **[]**, **++** et **--**, **->**, **<<** et **>>** (décalage), **new** et **delete**
  - en fonction : **<<** et **>>** (flux), **+-/***
- Opérateurs à éviter
  - **,**, **!**, **||** et **&&**
- Opérateurs interdits
  - **.**, **::** et **?:** (ternaire)

19

ISIMA La surcharge d'opérateurs : exemple 1

```
class Point
{
  private:
    int absc;
    int ordo;
    static int nb_points;

  …
  public :

  …
    Point & operator+ (const Point & tmp)
    {
       absc += tmp.absc;
       ordo += tmp.ordo;
       return *this;
    }
};
```

c

20

ISIMA La surcharge d'opérateurs : exemple 2

```
class Point
{
  private:
    int absc;
    int ordo;
    static int nb_points;

  …
  public :

  …

};
int operator* (const Point & p1,
               const Point & p2)
{
  return (p1.getAbs() * p2.getAbs() +
          p1.getOrd() * p2.getOrd());
}
```

21

ISIMA La surcharge d'opérateurs

- Conclusion
  - opérateurs monadiques et affectations en interne (méthode)
  - opérateurs dyadiques en externe (fonctions)
- Attention
  - en interne, transtypage sur le premier paramètre interdit
  - en externe, transtypage possible sur tous les paramètres
  - en externe, pas d'accès direct aux attributs des objets
  - passer par les accesseurs ou déclarer **`friend`**

22

ISIMA Utilisation de `friend` (1/2)

- Autoriser l'accès pour une entité externe
  - dans la déclaration de la classe
  - classe ou fonction
  - syntaxe :

```
class A
{
    …
    friend class B;
    friend void f(int, double);
    …
};
```

  - ne se substitue pas à la déclaration du prototype
  - accès complet à la classe
  - pas transitif, ni hérité

23

ISIMA Utilisation de `friend` (2/2)

- Avantages
  - efficacité du code, accès direct
  - notion (très limitée) de package
- Inconvénients
  - violation du principe d'encapsulation
  - pas de restriction ni de contrôle sur les accès
- Garde-fous
  - déclaration dans la classe cible
  - pas de transitivité ni d'héritage

24

ISIMA Concaténation pour **Chaîne**

- Utiliser l'opérateur **+**
  - syntaxe : `ch3 = ch1 + ch2;`
  - **this** n'est pas prépondérant → opérateur externe

version **friend**

```
Chaine operator+ (const Chaine & ch1, const Chaine & ch2)
{
  Chaine tmp(ch1.taille + ch2.taille - 1);

  strcpy(tmp.tableau, ch1.tableau);
  strcat(tmp.tableau, ch2.tableau);

  return tmp; // retour par copie nécessaire
}
```

→ efficace mais doit connaître (et respecter) l'implémentation

25

ISIMA Accès pour **Chaîne** (1/3)

```
…
Chaine ch("bonjour");            //appel à quel constructeur?


//Accès aux éléments du tableau
cout << ch.tableau[3] << endl;   //pas possible
cout << ch.getElement(3) << endl;//possible, mais inconvénient
cout << ch[3] << endl;           //pas possible maintenant, …
```

26

ISIMA Accès direct pour **Chaîne** (2/3)

- Utiliser l'opérateur **[]**
  - deux utilisations possibles
    - en lecture : `c = ch[0];`
    - en écriture : `ch[0] = c;`
  - deux versions
    - non constante, renvoie une référence sur un élément modifiable
    - syntaxe : **`char & Chaine::operator[] (int i);`**
    - lecture / écriture pour les non-constants
    - constante, renvoie une référence constante sur un élément
    - syntaxe : **`const char & Chaine::operator[] (int i) const;`**
    - lecture seule sur les constants

27

ISIMA Accès direct pour **Chaîne** (3/3)

```
char & Chaine::operator[] (int i)
{
  // test de securite
  if ((i < 0) || (i >= taille))
  {
    cerr << "acces hors limites" << endl;
    exit (EXIT_FAILURE);
  }

  return tableau[i];
}
```

```
const char & Chaine::operator[] (int i) const
{
  // test de securite
  if ((i < 0) || (i >= taille))
  {
    cerr << "acces hors limites" << endl;
    exit (EXIT_FAILURE);
  }

  return tableau[i];
}
```

le code est identique !

28

ISIMA Rappel sur les flots d'entrées et sorties

```
#include <iostream>

…

cout << " Informe la valeur de n" << endl;
cin  >> n;
```

Désigne le flot de sortie

Opérateur d'injection

– Les opérateurs d'injections servent à:
  - assurer le transfert de informations vers, par exemple: un périphérique, un fichier, …
  - assurer l'éventuel formatage de l'information

29

ISIMA Rappel sur les flots d'entrées et sorties

- Les flots prédéfinis
  – **cout**
  – **cin**
  – **cerr et clog** : flots de sortie connectés à la sortie standard d'erreur, mais **cerr** n'a pas de tampon intermédiaire, alors que **clog** utilise un tampon intermédiaire.
- La lecture sur le clavier et l'écriture sur l'écran
- Connexion d'un flot à un fichier

30

ISIMA Rappel sur les flots d'entrées et sorties

- Connexion d'un flot de sortie à un fichier

```
#include <iostream>
#include <fstream>
…
ofstream sortie("nome_fichier", <mode d'ouverture>);
sortie << … << … << …;
```

- Connexion d'un flot d'entrée à un fichier

```
#include <iostream>
#include <fstream>
…
ifstream entree("nome_fichier", <mode d'ouverture>);
entree >> … >> … >> …;
```

- Exemple de modes d'ouverture d'un fichier : ios::in (lecture), ios::out (écriture), ios::app (écriture en fin de fichier), …

31

ISIMA La surcharge des opérateurs de flux

- Opérateur d'insertion dans le flux de sortie : **<<**
  - objet principal : flux en sortie (**ostream**)
  - opérateur externe
  - syntaxe : **ostream & operator<< (ostream &, const A &);**

version non **friend**

```
ostream & operator<< (ostream & o, const Chaine & ch)
{
  o << ch.getStr();

  return o;
}
```

  - chemin atypique pour **ostream** (retour du flux modifié)
  - retour par référence pour le chaînage

32

ISIMA La surcharge des opérateurs de flux

- Opérateur d'extraction sur le flux d'entrée: **>>**
  - objet principal : flux d'entrée (**istream**)
  - opérateur externe
  - syntaxe : **istream & operator>> (istream &, A &);**

version non **friend**

```
istream & operator>> (istream & i, Chaine & ch)
{
  char str[64]; // pourrait faire mieux !
  i >> str;
  ch = str;     // Conversion de char * en Chaine, puis
                // l'appelle du opérateur =
  return i;
}
```

  - mêmes remarques, opérateur plus complexe

33

ISIMA La surcharge des opérateurs de flux

- Schéma pour la surdéfinition de << et >>

```
ostream & operator<< (ostream & sortie, type_classe objet)
{
   /* envoi sur le flot de sortie des membres objet en utilisant les
      possibilités classiques de << pour le type de base*/
    // instructions dans la forme: sortie << …;
    …
    return sortie;
}
```

```
istream & operator>> (istream & entree, type_classe & objet)
{
  /* lecture des informations correspondant aux différents membres en
     utilisant les possibilités classiques de >> pour le type de base*/
    // instructions dans la forme: entree >> …;
    …
    return entree;
}
```

34

ISIMA La surcharge de << et >> : exemple 1

- Exemple pour un rationnel

```
class Rationnel
{
 private:
   int num;
   int den;

 public:

  Rationnel::Rationnel(int n=0, int d=1): num(n),den(d)
  {…}
  void reduce(void);
  friend ostream & operator<< (ostream &, const Rationnel &);
  friend istream & operator>> (istream &, Rationnel &);

};
```

35

ISIMA La surcharge de << et >> : exemple 1

- Exemple pour un rationnel

```
ostream & operator<< (ostream & o, const Rationnel & r)
{
  o << r.num;
  if (r.den > 1)
    o << '/' << r.den;
  return o;
}

istream & operator>> (istream & i, Rationnel & r)
{
  char c;
  i >> r.num;
  i.get(c);
  switch (c)
  {
    case '/': i >> r.den;
              if (r.den == 0) throw Rationnel::DenominateurNul();
              if (r.den < 0) { r.den = -r.den; r.num = -r.num; }
              r.reduce();
              break;
    default : r.den = 1; i.putback(c);
  }
  return i;
}
```

exception !

36

ISIMA La surcharge de << et >> : exemple 2

- Supposons qu'une valeur de type point se présente toujours en lecture qu'en écriture sous la forme : <entier,entier>
- Il peut exister éventuellement des espaces blancs supplémentaires

```
class Point
{
  private:
    int absc;
    int ordo;
    static int nb_points;

  …
  public :

  …
    friend ostream & operator<< (ostream &, point);
    friend istream & operator>> (istream &, point &);
};
```

37

ISIMA La surcharge de << et >> : exemple 2

```
ostream & operator << (ostream & sortie, Point p)
{
  sortie << "<" << p.x << "," << p.y << ">";
  return sortie;
};
```

Exercice: fournir la surcharge du opérateur >> pour la classe point (à la fin de ce cours).

38

ISIMA Les opérateurs ++ et - -

- Opérateurs d'incrémentation et de décrémentation
  - deux modes d'utilisation
    - préfixé : `++a`, `--a`
    - postfixé : `a++`, `a--`
  - exemple: ++ incrémente d'une unité les deux coordonnées d'un point

```
Point a(2,5), c;
c = a++; //quelles sont les valeurs de absc et ordo?
         //a:(3,6) et c:(2,5)

c = ++a; //b:(4,7) et c:(4,7)
```

39

ISIMA La surcharge des opérateurs ++ et - -

- Opérateurs d'incrémentation de décrémentation
  - deux modes d'utilisation
    - préfixé : `++a`, `--a`
    - postfixé : `a++`, `a--`
  - opérateur interne
  - comment différencier les deux modes : signature !
    - préfixé : **`T & T::operator++ ();`**
    - postfixé : **`T T::operator++ (int);`**

      param. muet
    - idem pour **`--`**
    - postfixé utilise un objet temporaire → toujours préférer préfixé

40

ISIMA Les exceptions

- Mécanisme de gestion dynamique des erreurs
- Issus de ADA

- Avantages
  - gestion rigoureuse, intégrée dans le langage
  - ne peut être ignoré (sinon fin du programme)

- Inconvénients
  - syntaxe lourde
  - impact sur les performances (gestion dynamique)

41

ISIMA Les approches en C

- Valeur de retour de la fonction (statut)
  - indique si l'exécution s'est bien passée ou non
  - exemple

```
int racine (float f, float * r)
{
  if (f < 0.0) return 1;
  *r = sqrt(f);
  return 0;
}
```

  - avantages
    - simple à mettre en œuvre
    - peut indiquer plusieurs types d'erreurs
  - inconvénients
    - bloque le retour de la fonction
    - pas d'obligation de vérification
    - traitement des cas par un **`switch`**

42

ISIMA Mécanismes classiques

- Fonction renvoyant un statut
  - avantages
    - facile à mettre en œuvre
    - permet un diagnostic avancé
  - Inconvénients
    - type de retour monopolisé par le statut
    - rien n'oblige l'utilisateur à vérifier le statut
    - **`switch`** sur le type de retour
- Variante : variable globale positionnée
  - type de retour non monopolisé
  - conflits possibles sur la variable
  - énormément de valeurs possibles

ISIMA Mécanismes classiques

```
int fonction(parametres formels)
{
  if (ConditionErreur1)
    return CONSTANTE_ERREUR1;
  if (ConditionErreur2)
    return CONSTANTE_ERREUR2;
  …
  return CONSTANTE_SUCCES;
}

…

switch (fonction(parametres effectifs))
{
  case CONSTANTE_ERREUR1: … break;
  case CONSTANTE_ERREUR2: … break;
  case SUCCES: … break;
}
```

ISIMA Mécanismes classiques

- Message d'erreur et terminaison
  - avantages
    - possibilité de terminer proprement avec la mise en place de fonctions de terminaison (`atexit`)
  - inconvénients
    - termine toujours le programme
    - impossible d'adapter le fonctionnement du programme

```
if (condition erreur)
{
  cerr << "Message d'erreur" << endl;
  exit (1);
}
```

45

ISIMA Exceptions

- Lancement d'exceptions
  - dans fonctions ou méthodes
  - en principe, tout type de données peut être support
  - habituellement, classes spécialisées
- Traitement des exceptions
  - blocs de codes surveillés
  - gestionnaires d'exceptions : code spécialisé
- **Une exception non traitée entraîne l'arrêt du programme**

46

ISIMA Exemple

```
class Chaine
{
  public:
    class ExceptionBornes {};
  …
    char & operator[] (int index)
    {
      if ((index < 0) || (index > taille_))
      {
        throw ExceptionsBornes();
      }
      return tableau[index];
    }
};
```

lancement de l'exception c'est un **objet**

```
try
{
  c=chaine[3];
}
catch (Chaine::ExceptionBornes & e)
{
  // traitement
}
```

bloc d'instructions surveillées

type d'exception traitée

traitement des exceptions

47

ISIMA Le type des exceptions

- Éviter d'encombrer l'espace de nommage
  - utiliser des classes imbriquées
  - créer une hiérarchie d'exceptions
- Librairie standard du C++

```
class exception
{
  public:
    exception () throw();
    exception (const exception& rhs) throw();
    exception & operator= (const exception& rhs) throw();
    virtual ~exception() throw();
    virtual const char * what() const throw();
};
```

- Bon comportement : dériver de `exception`

```
#include <exception>
```

48

ISIMA Exemple de hiérarchies d'exceptions

```
class Vecteur
{
  public:
    class ExVecteur : public exception
    {
      public:
        const char * what() const throw()
        {
          return "Exception de vecteur";
        }
    };
    class ExVecteurBornes : public ExVecteur
    {
      // …
    };
    class ExVecteurBorneInf : public ExptVecteurBornes …
    class ExVecteurBorneSup : public ExptVecteurBornes …
    class ExVecteurAllocation : public ExptVecteur
    {
      // …
    };
};
```

49

ISIMA Traitement des hiérarchies d'exception

- Toujours traiter les exceptions les plus spécialisées d'abord
- Utiliser un objet passé par référence pour éviter une recopie
- Prévoir un traitement pour **`exception`**
- Gestionnaire spécialisé **`catch`**`(…)` qui ramasse « tout ce qui traîne »

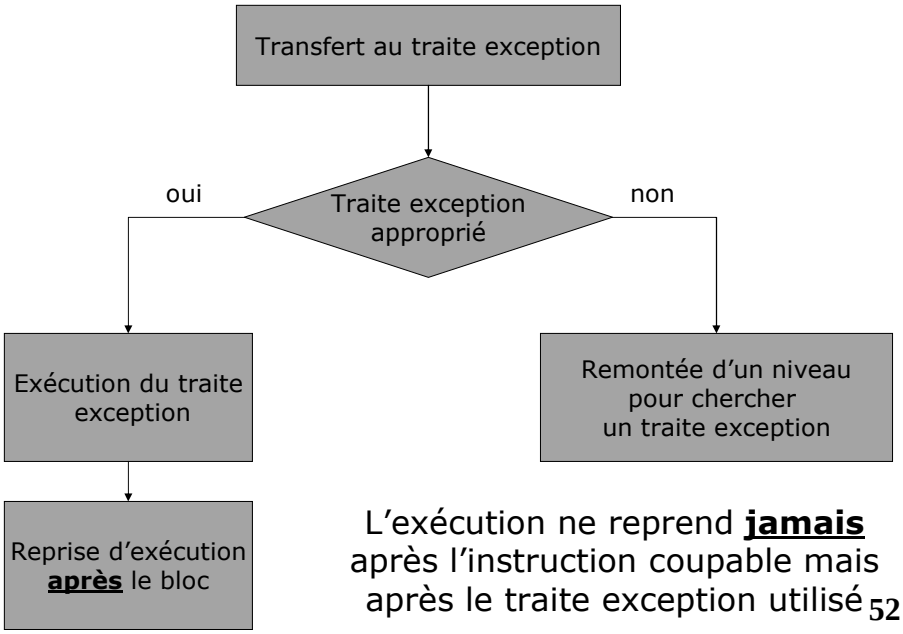
50

ISIMA Exemple de traitement des hiérarchies

```
try
{
 // code susceptible de lancer une exception
}
catch (Vecteur::ExptVecteurAllocation & e)
{
  // traitement exception spécialisée ExptVecteurAllocation
}
catch (Vecteur::ExptVecteurBorneInf & e)
{
  // Traitement exception spécialisée ExptVecteurBorneInf
}
catch (Vecteur::ExptVecteur & e)
{
  // Traitement exception plus générale ExptVecteur
}
catch (exception & e)
{
  // Traitement sommet hiérarchie
}
catch (...)
{ /* Fourre tout */ }
```

51

ISIMA Mécanisme de gestion des exceptions



L'exécution ne reprend **jamais** après l'instruction coupable mais après le traite exception utilisé

52

ISIMA Relancer une exception

- Pourquoi ?
  - traite exception incapable d'assurer le retour à la normale
- Conséquences
  - traitement d'une même exception à plusieurs niveaux
  - terminaison éventuelle du programme
- Syntaxe
  - simplissime

```
throw;
```

53

ISIMA La procédure `terminate`

- Met fin au programme lorsqu'une exception n'a pas été traitée
- Ne ferme pas les fichiers ni ne libère les ressources
  - ⇨prévoir les libérations de ressources dans chaque classe ...
  - ⇨ou prévoir une alternative à l'aide de la fonction **`set_terminate`**
  - ⇨prototype :

```
void fRemplacement(void);
```

54

ISIMA Spécificateurs d'exceptions

- But
  - définir l'ensemble d'exceptions que l'utilisateur peut s'attendre à voir lever par une méthode ou une fonction
- Difficulté
  - difficile à définir car le spécificateur doit prévoir tous les sous appels !
- Si une exception non prévue est levée :
  - appel de la procédure **`unexpected`**
  - par défaut celle-ci appelle **`terminate`**
  - possible de redéfinir ce comportement (**`set_unexpected`** similaire à **`set_terminate`**)

55

ISIMA Syntaxe des spécificateurs d'exceptions

- Syntaxe :

```
Typeretour ident(params) throw (liste,);
```

- Exemple :

```
char & Chaine::operator[](int index) throw (Vecteur::ExptVecteurBornes);
```

- Remarques :
  - spécification de `ExptVecteurBornes`
  - inclut les sous classes
  - les méthodes propres des classes dérivées de `exception` ne sont pas sensées pouvoir lever une exception

56

ISIMA Exercices (récapitulatif)

1. Pourquoi les opérateurs de flux ne peuvent pas être définis dans une méthode de classe ?
2. Proposez une version **friend** pour les méthodes des slides 32 et 33.
3. Proposez la surcharge du opérateur >> pour la classe point.

57

ISIMA Support de la généricité

- Implémentation de fonctions et classes paramétrées par des types ou des constantes.
- Mécanisme orthogonal au paradigme objet.
- **Puissant** car il suffit d'écrire une seule fois la définition d'une fonction pour que le compilateur puisse l'adapter à n'importe quel type.
- **Restrictif** car toutes les fonctions ainsi fabriquées par le compilateur doivent correspondre à la même définition.

58

ISIMA Fonctions paramétrées

- Exemple : maximum de 2 nombres
- 1ère solution : fonctions dédiées

```
int max (int a, int b)
{
  return ((a > b) ? a : b);
}
double max (double a, double b)
{
  return ((a > b) ? a : b);
}
```

- Avantage : vérification de types
- Inconvénients
  - duplication de code (taille + risque d'erreur)
  - taille de l'exécutable

59

ISIMA Fonctions paramétrées

- 2ème solution : macro

```
#define max (a, b) (((a) > (b)) ? (a) : (b))
```

- Avantages
  - code réutilisé
  - performances
- Inconvénients
  - pas de spécification de types
  - syntaxe absconse
  - effets de bords (ex: `i = max(++i,10)` ?)
  - taille de l'exécutable

60

ISIMA Fonctions paramétrées

- 3ème (et bonne) solution : fonction paramétrée

```
template <typename T>
const T & max (const T & a, const T & b)
{
  return ((a > b) ? a : b);
}
```

  - T est utilisé comme s'il était une classe (cf. arguments)

- Avantages
  - réutilisabilité, vérification de types
  - possibilité d'inlining pour les performances
- Inconvénients (?)
  - pas de conversion de types automatique

61

ISIMA Fonctions paramétrées

- Exemple

```
int      i, j;
double   k, z;
Vecteur  a, b;

cout << max(i,j) << endl;
cout << max(k,z) << endl;
cout << max(i,k) << endl;

cout << max<int>(i,k) << endl;
…
cout << max(a,b) << endl;
```

`int max(int,int)`

`double max(double,double)`

erreur : les 2 arguments doivent être de même type, pas de conversion automatique

correct : les 2 arguments sont considérés comme des `int` et les conversions s'appliquent

correct pour les types Vecteur doit fournir >

```
//comparaison sur le module des vecteurs
bool operator>(const Vecteur & a, const Vecteur & b) {
  return (a.x*a.x + a.y*a.y > b.x*b.x +b.y*b.y);
}
```

62

ISIMA Fonctions paramétrées

```
int n;
unsigned int m;
const int c1, c2;
int t[10];
int * p;
char c;

max(n,c);          //erreur
max(n,m);          //erreur
max(n,c1);         //dépend du compilateur
max(c1,c2);        //OK
max(t,p);          //erreur
max<int> (c,n);    //OK car on force le type au moment
                   //de l'appel du patron
```

63

ISIMA Fonctions paramétrées

- Exemple : char *

```
char * c1="monsieur";
char * c2="bonjour";

cout << max(c1,c2) << endl;
```

```
char * operator>(char * c1, char * c2)
{
  if (strcmp(c1, c2) > 0) return c1;
  else  return c2;
}
```

On ne peut surcharger > pour un type « élémentaire »

64

ISIMA Spécialisation de fonctions paramétrées

```
template <typename T>
const T & max (const T & a, const T & b)
{
  return ((a > b) ? a : b);
}

template <>
char * max<char *>(char * c1, char * c2)
{
  if (strcmp(c1, c2) > 0) return c1;
  return c2;
}
```

65

ISIMA Surdéfinition de patrons

```
//patron numéro I
template <typename T>
const T & max (const T & a, const T & b)
{
  return ((a > b) ? a : b);
}

//patron numéro II
template <typename T>
const T & max (const T & a, const T & b, const T & c)
{
  return max (max(a,b),c);
}
```

```
main()
{
   int    n=12, p=15, q=2;
   float  x=3.5, y=4.25, z=0.25;
   cout << max(n,p);      //Patron I : int max(int, int)
   cout << max(n,p,q);    //Patron II : int max(int, int, int)
   cout << max(x,y,z);    //Patron II : float max (float, float, float)
}
```

66

ISIMA Fonctions paramétrées

- Avec plusieurs paramètres de type

```
template <typename T, typename U>
T somme (T x, U y, T z)
{
  return (x+y+z);
}
```

```
main()
{
   int    n=1, p=2, q=3;
   float  x=2.5, y=5.0;
   cout << somme(n,x,p);                  //OK affiche la valeur 5 (int)
   cout << somme(x,n,y);                  //OK affiche la valeur 8.5 (float)
   cout << somme(n,p,q);                  //OK affiche la valeur 6 (int)
   cout << somme(n,p,x);                  //Erreur : pas de correspondance
   cout << somme<int,float>(n,p,x);       //OK force l'utilisation de int pout T et float pour U
   cout << somme<float>(n,p,x);           //OK force l'utilisation de float pour T
}
```

67

ISIMA Classes parametrées

- Exemple 1 : patron pour la classe point

```
#ifndef __POINT_HXX__
#define __POINT_HXX__

template <typename T>
class Point
{
  protected:
    T absc;
    T ordo;
  public:
    Point (const T & x, const T & y)
    {
        absc = x; ordo = y;
     }
    void affiche () const;
};

template <typename T>
void Point<T>::affiche() const
{
     cout << absc << "\t" << ordo << endl;
}
#endif
```

```
//Instanciation
Point <int>   a;
Point <float> b;
```

68

ISIMA Classes paramétrées

- Exemple 2: patron pour une classe Pile
  - TAD : fonctionne de la même manière quelque soit le type de données utilisé
  - simplification : tableau de taille fixe
  - 4 opérations
    - `void push (const T &);` empiler un élément
    - `void pop (void);` dépiler un élément
    - `const T & top (void) const;` accéder au premier élément
    - `bool empty (void) const;` test de vacuité

69

ISIMA Classes paramétrées

- Exemple 2 : une patron pile

```
#ifndef __PILE_HXX__
#define __PILE_HXX__

template <typename T>
class Pile
{
  protected:
    int first;
    T   tab[256];

  public:
    Pile () : first(0) {}
    const T & top (void) const { return tab[first-1]; }
    bool empty (void) const { return (first == 0); }
    void push (const T & elt) { tab[first++] = elt; }
    void pop (void) { --first; }
};

#endif
```

70

ISIMA Organisation du code source (1/4)

- Spécificité du code générique
  - le code générique n'est pas compilable
  - seulement compilable sur l'appel instancié

  → une classe paramétrée n'est jamais compilée telle quelle

  - le code doit être intégralement placé dans le **`.hxx`**
  - y compris les méthodes déportées et les fonctions
  - ainsi, le code final est entièrement généré à l'appel

  - une génération de code par instanciation différente
  - une compilation par instanciation différente

71

ISIMA Organisation du code source (2/4)

fichier A.hxx

```
#ifndef __A_HXX__
#define __A_HXX__

template <typename T>
class A
{
  // méthodes et attributs
};

// fonctions

#include "A.cxx"

#endif
```

fichier A.cxx

```
// initialisations statiques
…

// méthodes déportées
…

// fonctions déportées
…
```

72

ISIMA Organisation du code source (3/4)

- Application à la pile

```
#ifndef __PILE_HXX__
#define __PILE_HXX__

template <typename T>
class Pile
{
  protected:
    int first;
    T   tab[256];

  public:
    Pile ();
    const T & top (void) const;
    bool empty (void) const;
    void push (const T & elt);
    void pop (void);
};

#include "pile.cxx"

#endif
```

```
template <typename T>
Pile<T>::Pile () : first(0)
{}

template <typename T>
const T & Pile<T>::top (void) const
{ return tab[first-1]; }

template <typename T>
bool Pile<T>::empty (void) const
{ return (first == 0); }

template <typename T>
void Pile<T>::push (const T & elt)
{ tab[first++] = elt; }

template <typename T>
void Pile<T>::pop (void)
{ --first; }
```

ISIMA Organisation du code source (4/4)

- Utilisation de la pile

```
#include <iostream>
#include "pile.hxx"

using std::cout; using std::endl;

int main (int, char **)
{
  typedef Pile<int> PileInt;

  PileInt p1;

  p1.push(10);
  p1.push(20);

  while (p1.empty() == false)
  {
    cout << p1.top() << endl;
    p1.pop();
  }
  return 0;
}
```

instanciation

autre exemple

```
typedef Pile<Point> PilePoint;

PilePoint p2;
Point pt(5,27);

p2.push(pt);
p2.push(Point(1,1));
```

74

ISIMA Des constantes en paramètre générique

- Une pile dont la taille est définie génériquement

```
#ifndef __PILE_HXX__
#define __PILE_HXX__

template <typename T, const int TAILLE = 256>
class Pile
{
  protected:
    int first;
    T   tab[TAILLE];
  public:
    Pile ();
    const T & top (void) const;
    bool empty (void) const;
    void push (const T & elt);
    void pop (void);
};

#include "pile.cxx"

#endif
```

La taille est figée à l'instanciation, pas besoin de FNC !

instanciations

```
typedef Pile<int>     PileInt256;
typedef Pile<int,10>  PileInt10;

PileInt256 p1;
PileInt10  p2;

p1 = p2; // interdit, types ≠
```

75

ISIMA Avantages et inconvénients

- Généricité + FNC
  - avantages
    - réallocation dynamique
    - taille de l'exécutable
  - inconvénient
    - lenteur de **`new`** / **`delete`**
- Généricité + taille constante
  - avantages
    - rapidité car mémoire allouée dans l'exécutable
  - inconvénient
    - taille de l'exécutable
    - pas de réallocation dynamique

76

ISIMA Exemple de généricité pratique

- Génération par extension des comparateurs
  - si on a **<** et **==**, on peut faire toutes les comparaisons
  - exemple

```
template <typename T>
bool operator<= (const T & a , const T & b)
{
  return !(b < a);
}

template <typename T>
bool operator>= (const T & a, const T & b)
{
  return !(a < b)
}
```

  - bémol : attention à la qualité du code produit (nb appels)

77