Bases de Programmation Orientée Objet et évènementielle

Introduction

Spécificités du C++

Benoît Piranda
Équipe SISAR
Université de Marne La Vallée

Niveau d'abstraction

- Langage de haut niveau
  - C, C++, Java, Fortran, Basic, Perl
- Langage de bas niveau
  - Assembleur, langage machine

Différents types de langage

- Langages interprétés
- Langages compilés

C	C++	Fortran	Basic	Perl	Php	Java
Compilateur			Interpréteur			Compilateur
Fichier exécutable						Pseudo-code
						Interpréteur JVM
Langage machine du processeur						

Langages interprétés

- L'analyse et la traduction du langage de haut niveau se fait pendant l'exécution

Code source

```
…
for i=1 to 10
j=2*i+3
next i
…
```

Analyse de l'instruction ⇨ Exécution du traitement

Langages interprétés

- Avantages
  - exécution immédiate du code
  - recherche des erreurs d'exécution
    - état du programme au moment de l'erreur
    - poursuite du programme après correction
- Inconvénients
  - la même analyse est faite à chaque exécution
    - perte de temps
  - impossibilité de mélanger le code avec d'autres langages

Langages compilés

- Traitement du langage de haut niveau en deux étapes :
  - la compilation : traduction en langage machine
  - l'exécution par le système hôte

Code source

```
…
for(i=1;i<=10;
        i++)
j=2*i+3
…
```

Compilation ⇨ Fichier exécutable

Fichier exécutable ⇨ *Interprétation du code machine* ⇨ Exécution du traitement

Langages compilés

- Avantages
  - optimisation du code à priori
    - vitesse d'exécution plus grande
  - possibilité de faire coexister plusieurs modules
    - pouvant éventuellement être produits par différents langages
- Inconvénients
  - temps de compilation
  - ensemble de fichiers
  - correction des erreurs plus complexes

7

Développement

- Utilisation de deux langages orientés objets
  - C++
  - Java
- Domaines d'applications différentes
- Outils complémentaires

8

Historique

- Le langage C
  - 1971, Dennis Ritchie
  - fortement associé au développement d'Unix
  - commercialisé en 1978
  - 1983, le C est normalisé (ANSI)
- Le langage C++
  - 1980 Dr. Bjarne Stroustrup (Bell Laboratories)

9

Le langage C++

- Intérêts
  - Langage compilé très rapide
  - Profite des fonctions du C
    - Langage fonctionnel
    - Variable globales
  - Complète le langage C par la notion d'objet
- Langage intermédiaire
  - Liberté du mode fonctionnel
  - Rigueur du tout objet

10

Réalisation d'un programme C++

- Modules écrits par l'utilisateur
  - Les programmes sources en C++
    - utilisation d'un éditeur de texte
    - fichier de suffixe .cpp .C .c++
  - Le fichier makefile
    - pour les programmes multi-modules
- Compilation
  - compilateur C : CC ou g++
  - Options
    - -g : instructions pour débugger
    - -O2 : optimisation du code
    - -o nom de l'exécutable ou -c fic.c à compiler en .o
    - -I/chemin de fichier include
    - -l bibliothèque
    - -L/chemin de bibliothèque
- Edition de liens (link)
  - produit le fichier exécutable

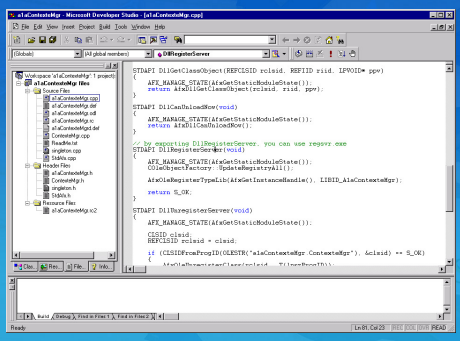
11

Principe d'utilisation

- Editer un fichier en langage C++
  - `Editeur de texte`
    - `vi, emacs, nedit, notepad…`
  - `Interface de programmation`
    - `kdevelop, kdestudio…`
- Compiler un code source
  - Fichier unique (édition de lien inutile)
    `g++ fichier.c -o prog`
  - Plusieurs fichiers (Makefile)
    `make`
- Exécuter le fichier résultant de la compilation
  `./prog`

12

Principe d'utilisation (Windows)



13

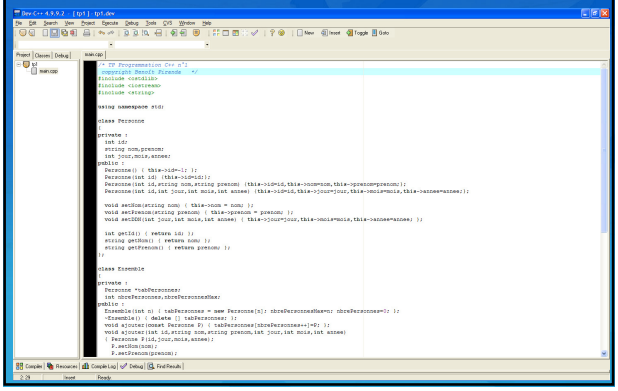
Le premier programme C++

- Inclure les bibliothèques
- Définir une fonction `main` sans argument
  - Existe dans tout programme C++
- Utilisation de l'opérateur `cout` (`endl` produit un passage à la ligne)

```
#include <iostream>

using namespace std;

void main(void) {
  cout << "hello, world"  << endl;
}
```

14

Principe d'utilisation



make et Makefile

- Langage de traitement de tache interdépendantes
- Principe
  - Définition d'une liste de règles sous la forme

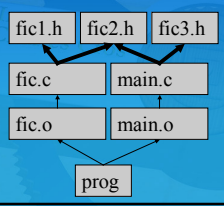
    `but : dep1 dep2`

    ↔ action but
  - Que l'on interprète par
    - but dépend de dep1 et dep2
    - Si but n'existe pas ou bien est antérieur à dep1 ou dep2 alors réaliser le traitement action but
- Remarque
  - But doit être mis à jour par le traitement

16

make et Makefile

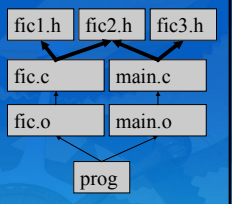
- Utilisation dans un contexte de compilation
  - Il faut compiler un module
    - lorsque il a été modifié
    - ou bien si un des fichiers qu'il inclut a été modifié.
  - Il faut réaliser l'édition de lien
    - si au moins un des modules a été modifié



17

make et makefile

```
prog : fic.o main.o
       gcc -o prog fic.o main.o

fic.o : fic.c fic1.h fic2.h
       gcc -c fic.c

main.o : main.c fic2.h fic3.h
       gcc -c main.c

exec : prog
       ./prog
clean :
       rm fic.o main.o
save :
       tar cvfz fic.tgz *.c *.h
```



compilation
`make ou make prog`
compilation puis exécution
`make exec`
suppression des fichiers de compilation
`make clean`
mémorisation du code dans une archive
`make save`

Réalisation d'un programme C++

- Modules écrits par l'utilisateur
  - Les programmes sources en C++
    - utilisation d'un éditeur de texte
    - fichier de suffixe .cpp .C .c++
  - Le fichier makefile
    - pour les programmes multi-modules
- Compilation
  - compilateur C : CC ou g++
  - Options
    - -g : instructions pour débugger
    - -O2 : optimisation du code
    - -o nom de l'exécutable ou -c fic.c à compiler en .o
    - -I/chemin de fichier include
    - -l bibliothèque
    - -L/chemin de bibliothèque
- Edition de liens (link)
  - produit le fichier exécutable

19

Principe d'utilisation

- Editer un fichier en langage C++
  - `Editeur de texte`
    - `vi, emacs, nedit, notepad…`
  - `Interface de programmation`
    - `kdevelop, kdestudio…`
- Compiler un code source
  - Fichier unique (édition de lien inutile)
    `g++ fichier.c -o prog`
  - Plusieurs fichiers (Makefile)
    `make`
- Exécuter le fichier résultant de la compilation
  `./prog`

20

Historique

- Le langage Java
  - 1990 Bill Joy de Sun
  - 1993 Interpréteur Java pour Netscape
- S'appuie sur
  - la syntaxe C
  - La philosophie de Smalltalk

25

Réalisation d'un programme java

- Modules écrits par l'utilisateur
  - Les programmes sources en Java
    - utilisation d'un éditeur de texte
    - fichier de suffixe .java
- Compilation
  - compilateur java : `javac exemple.java`
  - produit des fichiers *.class* : `exemple.class`
- Interpréteur de pseudo-code
  - Interpréteur : `java exemple`
  - Interprète les fichiers *.class* (on ne fait pas figurer l'extension)

26

Intérêts de Java

- Simple relativement à C/C++
  - Gestion de la mémoire plus dirigée
- Orienté objet
- Indépendant de l'architecture
- Orienté Réseau
  - Outils intégré d'interfaçage sur le réseau Internet
- Robuste
  - Impose beaucoup de vérifications (typage…)
  - Évite des erreurs
- Sécurisé
- Multitâche

27

Limites

- Langage compilé puis interprété
  - Besoin d'une interprétation du pseudocode par un logiciel spécifique
- Assez lent
  - Relativement à des logiciels compilés

28

Intérêts de la programmation de Java

- **Profiter des classes déjà existantes**
  - **Être immédiatement opérationnel avec un minimum de développement**
  - **Application Programming Interfaces (API)**
    - Les API de base
      - ✓ java.lang (types de bases, Thread, ClassLoader, Math, …)
      - ✓ java.util (classes utilitaires : Date, Vector, Stack,…)
      - ✓ java.applet (classe de base pour les applets)
      - ✓ java.awt (interface graphique)
      - ✓ java.net (outils pour le réseau : Socket, InetAddress, URL…)
      - ✓ java.security (signature numérique, cryptographie…)
    - Autres API
      - ✓ Java Management API (administration du réseau sur Internet)
      - ✓ Java Generic Library

29

Domaines d'utilisation de Java

- **Logiciels portables**
  - **Programme multiplateforme**
  - **Type de programme**
    - Logiciels très communicants et pas très exigeants en rapidité
    - StarOffice
- **Applets pour l'Internet**

30

Réalisation d'un programme Java

- **Modules écrits par l'utilisateur**
  - **Les programmes sources en Java**
    - utilisation d'un éditeur de texte
    - fichier de suffixe .java
- **Compilation**
  - **compilateur java : javac**
  - **produit des fichiers .class**
- **Interpréteur de pseudo-code**
  - **java**

31

Premier programme Java : texte

- **Afficher un texte à l'écran en mode texte**
  - **Créer une classe helloWorldTexte**
  - **Définir une méthode `main`**
  - **Appeler la fonction `println` de la bibliothèque `System`**

helloWorldTexte.java

```
public class helloWorldTexte
{ public static void main(String[] args)
  { System.out.println("hello, world!");
  }
}
```

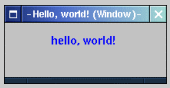
```
[]$ javac helloWorldTexte.java
[]$ java helloWorldTexte
hello, world!
[]$
```

32

Deuxième programme Java : graphique

helloWorldWindow.java

```
import java.awt.*

public class helloWorldWindow extend Frame
{ public helloWorldWindow()
  { super("-hello, world! (Window)-");
    setSize(210,100);
    show();
  }
  public void paint(Graphics g)
  { g.setColor(Color.blue);
    g.drawString("hello, world!",60,50);
  }

  public static void main(String[] args)
  { new helloWorldWindow();
  }
}
```

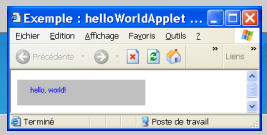


33

Troisième programme Java : applette

helloWorldApplet.java

```
import java.awt.*
import java.applet.*

public class helloWorldApplet extend Applet
{ public void paint(Graphics g)
  { g.setColor(Color.blue);
    g.drawString("hello, world!",60,50);
  }
}
```



helloWorldApplet.html

```
<html>
<head> <title>Exemple : helloWorldApplet </title></head>
<body>
<applet code="helloWorldApplet.class"></applet>
</body>
</html>
```

34

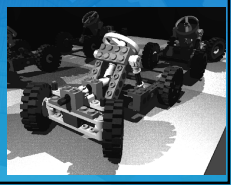
Programmation orientée objet

Spécificités de la programmation objet et du C++

Benoît Piranda
Équipe SISAR
Université de Marne La Vallée

Exemple de programmation fonctionnelle

- Logiciel de synthèse d'image
  - Objets géométriques canoniques
    - Sphère, paramètres : rayon de la sphère
    - Cube, paramètres : cotés du cube
    - Cylindre, paramètres : rayon et hauteur
    - …
  - Lancer de rayons
    - Mémoriser une liste d'objets
    - A chaque objet on associe une fonction de calcul d'intersection entre l'objet et un rayon

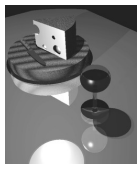


Exemple de programmation fonctionnelle

- Première solution
  - Structure pour chaque type d'objet
  - Fonction pour chaque objet
  - Structure de liste permettant d'accéder à des données de type différent
  - Étude de cas, à chaque traitement, sélectionner la bonne fonction
- Très peu efficace, beaucoup de tests

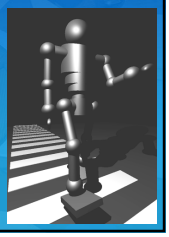
Exemple de programmation fonctionnelle

```
struct data_sphere {
  double rayon;
};

struct data_cylindre {
  double rayon,hauteur;
};

struct element {
  void *objet;
  int type;
  struct element *suivant;
};

void intersection_sphere(struct matrice *mat, struct data_sphere *sph) {…}
void intersection_cylindre(struct matrice *mat, struct data_cylindre *cyl) {…}

void intersection(struct matrice *mat, struct element *elm)
{ while (elm)
  { switch (elm->type)
    { case 0 : intersection(mat,(struct data_sphere*) elm->objet); break;
      case 1 : intersection(mat,(struct data_cylindre*) elm->objet); break;
    }
    elm = elm->suivant;
  }
}
```



Exemple de programmation fonctionnelle

- Deuxième solution
  - Définition d'un tableau de fonctions
    - Indexées par le numéro de type de l'objet
- Évite les tests multiples



Solution : la programmation objet

- Associer les données et les traitements
  - OBJET =
    - Données de l'objet géométrique
    - Traitement des intersections

Différences C / C++, passage à la POO

- Tout code C peut être incorporé dans le code C++
- Choisir des règles de programmation de POO
- Regrouper les données et les traitements associés
  - Éviter (voir supprimer) les pointeurs
    - Passage par référence plutôt que par adresse
    - Pas de alloc / free
  - Éviter (voir supprimer) les fonctions
    - Utilisation des méthodes de classe
    - Utilisation des flux (cin, cout, …)
  - Éviter (voir supprimer) les variables globales

41

Les types fondamentaux de données C++

- Booléens (bool)
  - Deux valeurs possibles : vrai (true) ou faux (false)
  - Faux correspond à la valeur 0, Vrai aux autres valeurs
  - Un pointeur peut être directement converti en booléen (vrai si différent de NULL)

```
bool egalite = (a==b);

if (egalite)
{ cout << "a=b" << endl;
}
```

42

Les types fondamentaux de données C++

- Caractère (char)
  - Peut stocker un caractère unique
    - Constante caractère entre guillemets simples
  - Codage d'un entier sur un octet
    - Utilisation
      - ✓ Codage de données à faibles combinaison de valeur
        - ➢ Image : 3 octets RVB par pixel
        - ➢ Son CDROM : 2 octets par échantillons (1/44100 s)
    - signed char
      - ✓ Compris entre –128 et 127
    - unsigned char
      - ✓ Compris entre 0 et 255

```
char lettre = 'a';
cout << lettre << " " << int(lettre) << endl;
```

43

Les types fondamentaux de données C++

- Entiers (int)
  - Trois tailles :
    - short ou short int : forme courte (2 octets)
    - int : forme normale
    - long ou long int : forme longue (4 octets)
  - Deux interprétations :
    - Préfixe signed ou pas de préfixe
    - Préfixe unsigned : uniquement positifs
  - Combinaisons possibles

```
int a;
unsigned short b = 543;
signed long int c;
```

44

Les types fondamentaux de données C++

- Les constantes entières
  - Codage décimal : nombre entier sans préfixe
  - Codage octal : préfixe 0
  - Codage hexadécimal : préfixe 0x

Décimal	0	2	63	83
Octal	00	02	077	0123
Hexadécimal	0x0	0x2	0x3f	0x53

45

Les types fondamentaux de données C++

- Virgules flottantes
  - Simple précision (float)
  - Double précision (double)
  - Quadruple précision (long double)

46

Mémoire associée aux types

Type	Taille (octets)	Bornes	Bornes (unsigned)
char	1	-128..127	0..255
short int ou short	2	-32 768..32 767	0..65 535
int	2 ou 4	-2 147 483 648.. 2 147 483 647	0..4 294 967 296
long int ou long	4		
float	4	(23-1-8) 3.4e-38 .. 3.4e+38	
double	8	(52-1-11) 1.7e-308 à 1.7e+308	
long double	10 ou 12	(64-1-15) 3.4e-4932 à 1.1e+4932	

47

Mémoire associée aux types

- Quelques traitements sur les types
  - Valeurs limites : include <limits>

```
#include <limits>

cout << numeric_limits<float>::max() << endl;
```

48

Les types fondamentaux de données C++

- void
  - Code indiquant qu'une méthode est sans paramètres ou sans résultat
  - void* utilisé en C doit être évité en C++
    - Pointeur sans type devant ensuite être converti par le programmeur
    - Ex : void *malloc(int)

```
int *ptr;

ptr = (int*)malloc(100*sizeof(int));
```

```
int *ptr;

ptr = new int[100];
```

Les types fondamentaux de données C++

- Énumérations enum *type* { *valeur0*, *valeurs…*}
  - Type associé à un ensemble de valeurs possibles
    - Constantes entières placées dans l'ordre croissant
    - Par défaut : ordre croissant, pas de 1, à partir de 0
  - Codage :
    - Codé sur l'entier de taille minimum pour contenir les valeurs

```
enum cpt { PREMIER, DEUXIEME, DIXIEME };
-> PREMIER=0, DEUXIEME=1, DIXIEME =2
Codé sur 2 bits

enum cpt { PREMIER=1, DEUXIEME, DIXIEME =10 };
-> PREMIER=1, DEUXIEME=2, DIXIEME=10
Codé sur 4 bits
```

Variable et constante (C++)

- Identifiant associé à une valeur fixée
  - Une variable déclarée avec le préfixe `const` : « variable » qui ne pourra pas être modifiée dans le programme.

```
void main(void)
{ const int un=1;   Initialisation de la constante
  int deux;

  un = 2;           Opération interdite
  deux = un+un;
}
```

  - Routine de pré-compilation `#define`
    - Remplace dans le programme toutes les occurrences d'une constante par sa valeurs

```
#define UN   1     Initialisation de la constante
```

51

Variable static

- Identifiant associé à un contexte
  - Une variable déclarée avec le préfixe `static`
  - Premier passage sur la déclaration de la variable
    - Allocation
    - Initialisation
  - Passage suivant
    - Récupération de la variable précédente
    - Adresse mémoire et valeur restituée

```
int incremente(void)
{ static int cpt=0;
  return ++cpt;
}
:
cout << incremente() << endl;
cout << incremente() << endl;
cout << incremente() << endl;
```

52

Expression mathématique (C++)

- Opérations identique à C
  - les 4 opérations mathématiques
    - +, -, *, /
  - Opérations spécifiques sur les entiers
    - / (division entière) et % (modulo)
  - Opérations booléennes (bit à bit)
    - & (et), | (ou) et ! (non)
  - Opérations logiques (tests)
    - && (et), || (ou) et ! (non)
  - Incréments
    - ++, +=, -- , -=
  - Comparaisons
    - <,<=, >=, >, ==, !=
  - Opérateur de flux
    - >>, <<

53

Expression mathématique (C++)

- Tout opérateur peut être surchargé
  - Redéfinis pour des types ou des classes utilisateurs
  - Exemple string : + représente la concaténation

54

Affectation

- Vérification de type
  - Les deux opérandes doivent être de même type
- Transtypage : conversion entre type
  - Consiste à préciser le type entre parenthèse
  - Généralement possible
    - Pour les valeurs numériques
    - Les chaînes de caractères (retourne le code ASCII)
    - Les pointeurs
  - Peut générer une perte d'information
  - Aux risques de l'utilisateur
- Transtypage automatique
  - Si pas de perte d'information

55

Conversion implicite de type

- Transtypage
  - Format C : (type) expression
  - Format C++ : int(expression)

```
int x;
double y=10.5;

x = (int) y; /* x = y provoque un message à la compilation */
             /* x vaut 10 */

y = -10.5;
x = (int) y; /* x vaut -10 */
```

```
int x;

x = (int) 'A'; /* x vaut 65 */
```

Espace de nom et bibliothèques

- Espace de nom
  - Regroupement de déclarations sous un nom
    - Types
    - Énumérations
    - Fonctions
    - Classes
- Utilisation
  - using namespace nom;
  - Exemple : les outils d'entrée / sortie

```
using namespace std;

cout << …
```

57

La bibliothèque standard

- Propose
  - Le type string
  - Des outils de manipulation des entrées sorties standards
  - Des conteneurs (traitements indépendant des données)
    - Gestionnaire de vecteurs
    - Gestionnaire de piles
    - Gestionnaire de files
    - Gestionnaire de listes
    - Gestionnaire de d'ensembles

58

La bibliothèque standard

- Le type string
  - Mémorisation de chaîne de caractères
  - Gestion dynamique de la mémoire
  - Surcharge des opérateurs pour une manipulation naturelle des chaînes
  - Traitements de string.h intégrés

59

La bibliothèque standard

- Exemple d'utilisation du type string

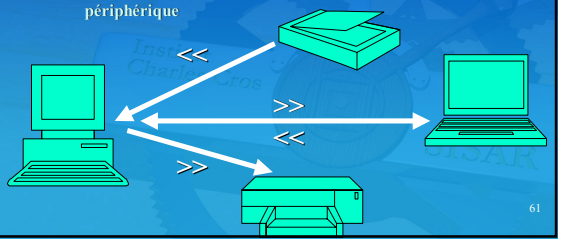
```
#include <string>

string s1="Hello";
string s2="world";

void main()
{ string s3=s1+", "+s2+"!\n";
  cout << s3;
}
```

```
Hello, world!
```

60

La bibliothèque standard

- Des outils de manipulation des entrées sorties standards
  - Deux types
    - Entrée : transférer une information dans la machine
    - Sortie : extraire une information de la machine vers un périphérique



61

Les opérateurs d'entrée sortie

- Opérations d'entrée sortie en C++
  - Les flux de données
  - Opérateurs permettant de faire transiter des données entre un port et une variable
  - Souvent surchargés pour réaliser des échanges entre les ports et des classes
- Deux opérateurs
  - Opérateur d'entrée (>>)
  - Opérateur de sortie (<<)

62

La bibliothèque standard

- Les ports d'entrée sortie standard
  - Sortie standard
    - cout (l'écran)
  - Sortie d'erreur standard
    - cerr (l'écran)
  - Entrée standard
    - cin (le clavier)

63

La bibliothèque standard

- Exemple d'utilisation des ports standards

```
#include <iostream>

void main()
{ float x;
  cout << "entrez une longueur:";
  cin >> x;
  cout << "longueur : " << x << endl;
}
```

```
entrez une longueur:_
```

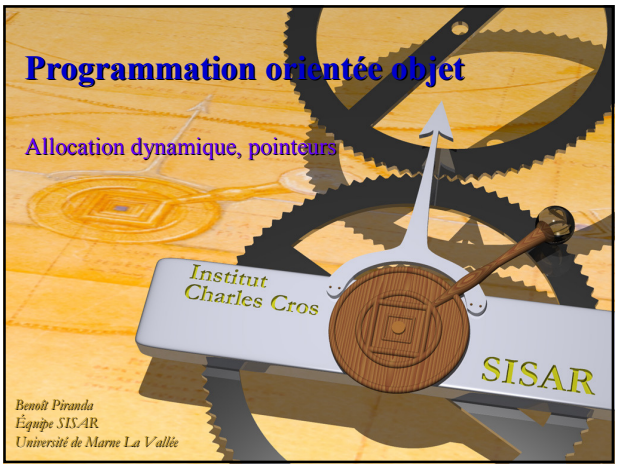
64

Exemple d'entrée-sortie standard

```
#include <iostream.h>

void main()
{ char texte[20];

  cout << "tapez un texte : ";
  cin >> texte;
  cout << "vous avez saisi :" << texte << endl;
  cerr << "pas d'erreur" << endl;
}
```

```
$>./a.out
tapez un texte : coucou
vous avez saisi : coucou
pas d'erreur
$>./a.out 2> toto
tapez un texte : coucou
vous avez saisi : coucou
```

65



Allocation dynamique

- Allocation dynamique = association d'une zone mémoire à un pointeur
  - Réalisé par l'opérateur *new type*
  - Réserve une zone mémoire pour mémoriser un objet de type *type* et retourne son adresse
  - Initialisation éventuelle entre parenthèses
  - Retourne *NULL* ou *null* en cas de problème d'allocation
    - Pas assez de mémoire disponible

67

Exemples d'allocations dynamiques

```
int *ptr;
ptr = new int(0);

etudiant *pe = new etudiant(numen);
```

68

Désallocation

- Désallocation
  - Destruction des objets
  - Libération de la mémoire qu'ils occupent opérateur `delete(ptr)`
  - Libère la zone mémoire pointée par *ptr*
  - `ptr` est inchangé
    - Pointe sur une zone mémoire non disponible !
- Peut produire des erreurs
  - Type : erreur de segmentation (segmentation fault)
  - Cas : désallocation d'une adresse non autorisée

```
delete ptr;
```

69

Allocation de tableaux

- Tableau
  - ensemble de données
    - Contiguës en mémoire
    - Homogènes (de même type)
  - Accès à l'aide d'un pointeur
- Allocation C++
  - Opérateur *new*

```
int *tab = new int[100];

tab[0]=2;
```

70

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};
```

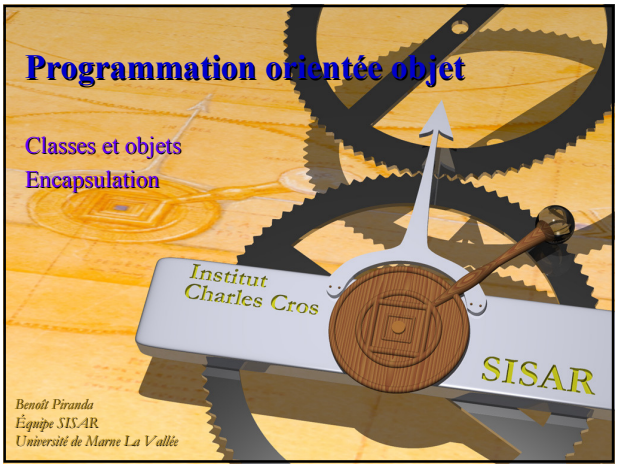
génère une erreur
complète par des 0

71

Désallocation de tableaux

- **Opérateur *delete []***
  - **Libère l'ensemble de la place utilisée par le tableau**
- **Attention**
  - **Opérateur *delete* insuffisant**

72



Encapsulation : expression du besoin

- **En C, pour associer les données et les traitements**
  - **Utilisation des noms de fonctions**
    - Besoin de passer des données en paramètres
    - Accès aux données libres
  - **Regroupement dans un fichier unique**

```
struct element
{ int n;
  struct element *suivant;
};
void add_element(struct element *e,int n)
{ e->n=n; e->suivant=NULL;
}
int existe_element(struct element *e,int n)
{ if (e->n==n) return 1;
  if (e->suivant==NULL) return 0;
  return existe_element(e->suivant,n);
}
void supprime_liste_element(struct element *e)
{ if (e->suivant!=NULL)
    supprime_liste_element(e->suivant);
  free(e);
}
```

```
int main(void)
{ struct element *el;

  init_element(el,0);
  :
  :
  if (existe_element(el,5))…
  :
  supprime_liste_element(el);
}
```

74

L'encapsulation : les objets

- **Outils de base des langages orientés objets**
  - **Les classes :**
    - Type d'objets
    - Un objet est une instance d'une classe
  - **Regroupent plusieurs *membres* :**
    - Les données : attributs
      - ✓ De type simple, pointeur, tableau
      - ✓ Les données complexes (autres que les types de bases)
    - Les traitements : les méthodes
      - ✓ Assimilables à des fonctions
      - ✓ Regroupent les traitements sous un nom
      - ✓ Peuvent admettre des paramètres
      - ✓ Retournent un résultat

75

Les méthodes

- **Regroupent des traitements sous un nom**
  - **Assimilables à des fonctions C**
  - **Réalisent des traitements sur les données de l'objet auquel elles appartiennent**
- **Peuvent retourner un résultat (unique)**
- **Peuvent admettre des paramètres**
  - **Paramètres d'entrée**
    - Exécuter les traitements sur des données différentes
  - **Paramètres de sortie**
    - Permettent de retourner des résultats multiples

76

Déclaration des méthodes

- A l'intérieur de la déclaration de la classe
  - Méthode associée à une classe
- Résultat
  - Unique
  - Variable, pointeur ou référence [NOUVEAU]

```
Type_résultat nom_fonction(type_param1 param1,…)
{ Type_résultat result;
  :
  return result;
}
```

```
Type_référence& nom_fonction(type_param1 param1,…)
{ Type_résultat result;
  :
  return result;
}
```

77

Déclaration des méthodes (C++)

- Déclaration des méthodes en deux étapes
  - Prototype dans la définition de la classe
    - Contient uniquement l'entête de la fonction
    - Déclaration de la liste des méthodes de la classe avec leurs types
  - Entête de méthode
    - Type de données retournées (sauf constructeurs et destructeur)
    - Format Classe::méthode
    - Paramètres de la méthode
      - ✓ Couples *type variables*
      - ✓ éventuellement *void* ou aucun
  - Corps
    - Les instructions proprement dites
    - Exécuté à chaque appel
    - L'instruction *return* permet de renvoyer le résultat (obligatoire pour les fonctions non *void*)
    - Se termine :
      - ✓ À la fin du bloc
      - ✓ Après l'appel de return

78

Exemple de méthodes (C++)

```
class disque
{ int cx,cy,rayon;
public :
  disque(int,int,int);
  double surface(void);
};

disque::disque(int x,int y,int r)
{ cx = x;
  cy = y;
  rayon = r;
}

double disque::surface(void)
{ return M_PI * rayon * rayon;
}
```

79

Valeur retournée (méthode/fonction)

```
double surface(void)
{ return 1.0;
}
```

```
double *surface(void)
{ double *ptr = new double[5];
  return ptr;
}
```

```
double *surface(void)
{ double a=1.0;
  return &a;
}
```

```
double &surface(void)
{ double a=1.0;
  return a;
}
```

Faux car une variable locale n'existe que dans la fonction où elle est déclarée

80

Passage de paramètres

- Paramètres d'entrée
  - Permettent d'initialiser les valeurs de la méthode
  - Ne sont pas modifiés à la sortie de la fonction
  - Peuvent être des constantes
- Paramètres de sorties
  - Transmettent un résultat à la fin de la fonctions
  - Sont forcément des modifiables
- Il existe trois types de passage de paramètres
  - Par valeur (paramètre d'entrée)
  - Par adresse (paramètre d'entrée & sortie)
  - Par référence (paramètre d'entrée & sortie) [NOUVEAU]

81

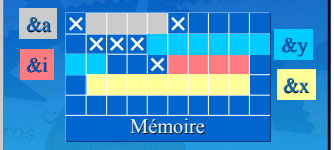
Passage par valeur

- Paramètres passés par valeur
  - Paramètres d'entrée uniquement
  - Les variables peuvent être modifiée à l'intérieur du corps de la fonction mais ne modifient pas le paramètre d'appel
- Principe
  - Copier le contenu du paramètre d'appel dans la variable paramètre de la fonction
  - Utiliser une copie des données
- Syntaxe
  - Déclaration : *type var*
  - Utilisation : *var*
  - Paramètre d'appel : *var* ou *constant*

82

Passage par valeur

```
float fonction(int i)
{ float x;

  x = 3.14*i;
  return x;
}

void main(void)
{ int a=1;
  float y;

  y = fonction(a);
}
```



83

Pièges ?

- Un exemple :

```
int Racine (double x) {
  if (x<0.0)  return 0;
  x = sqrt(x);
  return 1;
}

void main (void) {
  double x; x = 2.0;
  if (Racine(x)==1) printf("%lf\n", x);
}
```

Ne fonctionne pas

84

Pièges ?

- Un autre exemple :

```
void swap (double x, double y) {
  double tmp;
  tmp = x;
  x   = y;
  y   = tmp;
}

void main (void) {
  double x, y; x = 2.0; y = 5.0
  swap(x, y);
  printf("x=%lf, y=%lf\n", x, y);
}
```

Ne fonctionne pas

85

Passage par adresse

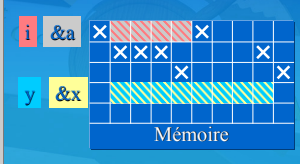
- Principe
  - passer l'adresse de la variable par valeur
  - modifier le contenu de la variable correspondante
- Syntaxe C++
  - Déclaration : *type *ptr*
  - Utilisation : **ptr*
  - Paramètre d'appel : *&var*

86

Passage par adresse

```
float* fonction(int *i)
{ float *x;

  x = (float*) malloc(*i*sizeof(float));
  if (x==NULL) *i=0;
  return x;
}
```

```
void main(void)
{ int a=1;
  float *y;

  y = fonction(&a);
  free(y);
}
```



87

Passage par adresse

- Attention le pointeur n'est pas modifiable…

```
void allouer(char *ptr,int i)
{ ptr = (char*) malloc(i*sizeof(float));
}

void main(void)
{ char *ptr=NULL;

  allouer(ptr,3);
  if (ptr==NULL) printf("NULL");
}
```

Ne fonctionne pas

88

Passage par adresse

…mais son contenu est modifiable.

```
void allouer(char **ptr,int i)
{ *ptr = (char*) malloc(i*sizeof(float));
}

void main(void)
{ char *ptr=NULL;

  allouer(&ptr,3);
  if (ptr==NULL) printf("NULL");
}
```

Peu lisible, propre au C
Doit être évité en C++

Passage par référence

- La variable en paramètre de la méthode représente la variable d'appel pour le corps de la méthode
  - Apport important du C++ sur le C
  - Avantages
    - Éviter l'utilisation des pointeurs (vrais paramètres d'entrée/sortie)
    - Pas de duplication de mémoire (*structures, classes*)
- Syntaxe C++
  - Déclaration : *type &var*
  - Possibilité de définir des paramètres en lecture seule : *const*
  - Utilisation dans la méthode : *var*
  - Paramètre d'appel : *var*

Passage par référence

```
int polynome2::racines(double &k1, double &k2)
{ double delta = b*b-4.0*a*c;

  if (delta<0.0) return 0;
  if (delta==0.0)
  { k1 = -b/(2.0*a);
    return 1;
  }
  delta = sqrt(delta);
  k1 = (-b-delta)/(2.0*a);
  k2 = (-b+delta)/(2.0*a);
  return 2;
}
:
{ polynome2 p(1.0,2.0,1.0);
  double l1,l2;

  int n = p.racines(l1,l2);
}
```


Passage par référence

```
void etudiant::copier(const etudiant &e)
{ this.numen = e.numen;
  …
}

:
{ etudiant e,olde;

  e.copier(olde);
}
```


Valeurs par défaut des paramètres

- Il est possible de préciser une valeur par défaut des paramètres placés à droites dans une déclaration de méthode.
  - Si lors de l'appel de la méthode ces paramètres sont omis, ils sont utilisés avec la valeur par défaut associée.

```
void etudiant::setNote(double note, double coef=1.)
{ this->points += note * coef;
}

:   :

e.setNote(12.,2.); // ajout d'une note coef. 2
e.setNote(11.); // ajout d'une note coef. 1 par defaut
```

Les méthodes

- Méthodes particulières
  - Constructeurs
    - Appelé automatiquement lors de la création d'un objet
    - Ses paramètres correspondent à ceux de l'allocation
    - Ne retourne aucun résultat
    - A pour tâche d'initialiser les attributs
  - Destructeur (unique)
    - Appelé automatiquement lors de la destruction d'un objet
    - A pour tâches de libérer les attributs alloués dynamiquement

Création d'un objet et constructeur

- (C++) la création d'un objet peut être statique ou dynamique
- La création d'un objet exécute son constructeur
  - Constructeur implicite
    - Sans paramètre
  - Constructeurs spécifiques
    - Peut recevoir des paramètres
    - Peuvent être surchargés (plusieurs types de paramètres)
- Opérations à la charge du constructeur
  - Initialiser les références de l'objet
  - Allouer les sous-objets mémorisés dans l'objet

95

Destructeur

- La destruction d'un objet
  - (C++) exécute la méthode : *~nomclasse()*
- Opérations à la charge du destructeur
  - Fermeture de fichiers, de connexion réseau…
  - Libération de la mémoire allouée dynamiquement dans les méthodes de l'objet

96

Exemple

```
class ensemble
{ element *tabElements;
  int nbreElementsMAx;
public :
  ensemble(int n)
  { nbreElementsMax = n;
    tabElements = new element[n];
  };
  ~ensemble()
  { delete [] tabElements;
  };
};
```

97

Accès aux données d'une classe

- Tous les attributs et toutes les méthodes sont soumis au droit d'accès
- Type d'accès
  - À l'intérieur de la classe
  - Depuis le programme appelant
  - Depuis une autre classe
- Trois niveaux de droit d'accès
  - Privé : private
  - Protégé : protected
  - Publique : public

98

Accès aux données d'une classe

- Niveau publique :
  - Accès libre des données et méthodes de l'objet
  - Depuis toute fonction, toute autre classe
  - Obligatoire pour l'un des constructeurs de la classe !

99

Accès aux données d'une classe

- Niveau protégé
  - Accès réservé aux classes filles

100

Accès aux données d'une classe

- **Niveau privé (par défaut)**
  - **Les méthodes de la classe peuvent utiliser les attributs et méthodes privées**
  - **Les fonctions « amies »**
    - Déclarée « friend » dans le corps de la classe

Accès aux données d'une classe

```
#include <iostream>
using namespace std;
class c1{
  int att1;
  int meth1() { return att1; };
public :
  c1(int a):att1(a) {};
   int att2;
   int meth2() { return att1; };
protected :
  int meth3() { return att1; };
friend int fct1(c1);
};
int fct1(c1 obj)
{ return obj.meth1();
}

int main(void)
{ c1 obj1(2);
  cout << obj1.att1 << endl;

  cout << fct1(obj1) << endl;
}
```

Autorisé, car fct1 est amie de c1

erreur à la compilation :
`int c1::att1' is private
La fonction main n'est pas amie de c1

Utilisation des droits : interface

- **L'interface d'une classe**
  - **L'ensemble de ses méthodes publiques :**
    - Permettent de gérer des restrictions d'accès aux données privées.
    - Méthodes classiques d'interfaces
      - ✓ set : affecter une variable privée en vérifiant ses valeurs.
      - ✓ get : transmettre une valeur de variable privée.

```
#include <iostream>
using namespace std;
class c1{
  double rayon,r2;
public :
  void setRayon(double r)
  { if (r>0.) { rayon=r;, r2=r*r; };
  double getRayon(void)
  { return rayon; };
};
```


Surcharge des méthodes

- **Principe**
  - **Définir plusieurs méthodes de même nom avec des types de paramètres différents**
    - Permet d'adapter les traitements aux types de données
    - Évite les noms de fonctions trop compliqués précisant le type des paramètres
    - Possible pour les constructeurs
  - **Les fonctions surchargées peuvent admettre des droits d'accès différents**

Exemple de surcharge de méthodes

```
class materiel
{ int numSerie;
  double prix;
public :
// constructeurs
  materiel() { numSerie = -1; };
// interface
  inline int getNumSerie() { return numSerie; };
  inline double getPrix() { return prix; };
protected :
// constructeurs
  materiel(int numSerie,double prix)
  { this.numSerie = numSerie;
    this.prix = prix;
  };
  inline void set(double prix) { this.prix = prix; };
  inline void set(int numSerie) { this.numSerie=numSerie; };
  inline void afficher() { cout << "materiel ?"; };
};
```


Surcharge des opérateurs

- **Les opérateurs sont gérés comme des méthodes**
  - **Permet de simplifier l'écriture des expressions.**

```
complexe operator + (complexe x, complexe y)
{ complexe resultat;

 resultat.setR = x.getR() + y.getR();
 resultat.setI = x.getI() + y.getI();
  return resultat;
}

: :

complexe a(1.,2.),b(3.,-1.),c;

c = a + b;
```