Programmation Orientée Objet en C++

Emanuel Aldea <emanuel.aldea@u-psud.fr>
http://hebergement.u-psud.fr/emi

M1-E3A

Premier chapitre

Introduction

- Historique
- Justification
- La création d'une application
- Le préprocesseur
- Les variables

Origines et approche C++

Apparition :

- **Extension syntactique** du langage C (un compilateur C++ compile du C)
- Travail entamé en 1979 par Bjarne Stroustrup
- Standardisation : ANSI/ISO '98, '03, '07, '11, '14, '17
- Principale extension : **les classes**
- Différence importante dans l'**architecture** du programme et dans l'**approche** de programmation
- Compréhension du langage :
  - $\Leftarrow$ compréhension de la réalité "physique" des données
  - $\Leftarrow$ compréhension du flot des données
  - $\Leftarrow$ essentiel en robotique ou en informatique industrielle

Plan

- Historique
- Justification
- La création d'une application
- Le préprocesseur
- Les variables

Caractéristiques d'un langage performant

Quelques objectifs majeurs

- **Extensibilité** logicielle :
  - $\Leftarrow$ faculté d'adaptation aux changements de spécification
  - $\Leftarrow$ architecture simple, découplage

Caractéristiques d'un langage performant

Quelques objectifs majeurs

- **Extensibilité** logicielle :
  - $\Leftarrow$ faculté d'adaptation aux changements de spécification
  - $\Leftarrow$ architecture simple, découplage
- **Réutilisabilité** et **compatibilité**
  - $\Leftarrow$ faculté d'être réutilisé en partie pour de nouvelles applications
  - $\Leftarrow$ moins de code, mieux structuré

Caractéristiques d'un langage performant

Quelques objectifs majeurs

- **Extensibilité** logicielle :
  - $\Leftarrow$ faculté d'adaptation aux changements de spécification
  - $\Leftarrow$ architecture simple, découplage
- **Réutilisabilité** et **compatibilité**
  - $\Leftarrow$ faculté d'être réutilisé en partie pour de nouvelles applications
  - $\Leftarrow$ moins de code, mieux structuré
- **Intégrité**
  - $\Leftarrow$ faculté de protéger les composants contre des accès non autorisés

Caractéristiques d'un langage performant

Quelques objectifs majeurs

- **Extensibilité** logicielle :
  - ⇐ faculté d'adaptation aux changements de spécification
  - ⇐ architecture simple, découplage
- **Réutilisabilité** et **compatibilité**
  - ⇐ faculté d'être réutilisé en partie pour de nouvelles applications
  - ⇐ moins de code, mieux structuré
- **Intégrité**
  - ⇐ faculté de protéger les composants contre des accès non autorisés
- **Efficacité** et **portabilité**
  - ⇐ utilisation optimale des ressources matérielles
  - ⇐ adaptation à différents environnements matériels et logiciels

Caractéristiques d'un langage performant

Quelques objectifs majeurs

- **Extensibilité** logicielle :
  - $\Leftarrow$ faculté d'adaptation aux changements de spécification
  - $\Leftarrow$ architecture simple, découplage
- **Réutilisabilité** et **compatibilité**
  - $\Leftarrow$ faculté d'être réutilisé en partie pour de nouvelles applications
  - $\Leftarrow$ moins de code, mieux structuré
- **Intégrité**
  - $\Leftarrow$ faculté de protéger les composants contre des accès non autorisés
- **Efficacité** et **portabilité**
  - $\Leftarrow$ utilisation optimale des ressources matérielles
  - $\Leftarrow$ adaptation à différents environnements matériels et logiciels

Le support POO améliore directement les premiers trois items.

Fonctionnement et alternatives

Règles de base

- Identifier les classes nécessaires

Quelles sont les alternatives courantes ?

Fonctionnement et alternatives

Règles de base

- Identifier les classes nécessaires
- Fournir l'ensemble d'opérations nécessaires pour chaque classe

Quelles sont les alternatives courantes ?

Fonctionnement et alternatives

Règles de base

- Identifier les classes nécessaires
- Fournir l'ensemble d'opérations nécessaires pour chaque classe
- Utiliser des mécanismes POO (i.e. héritage) pour structurer les fonctionnalités communes à de différentes classes

Quelles sont les alternatives courantes ?

Fonctionnement et alternatives

Règles de base

- Identifier les classes nécessaires
- Fournir l'ensemble d'opérations nécessaires pour chaque classe
- Utiliser des mécanismes POO (i.e. héritage) pour structurer les fonctionnalités communes à de différentes classes

Quelles sont les alternatives courantes ?

- Pas de POO : C, efficace et portable mais inadapté pour gérer la complexité d'un logiciel de taille importante (exception notable : le noyau Linux)

Fonctionnement et alternatives

Règles de base

- Identifier les classes nécessaires
- Fournir l'ensemble d'opérations nécessaires pour chaque classe
- Utiliser des mécanismes POO (i.e. héritage) pour structurer les fonctionnalités communes à de différentes classes

Quelles sont les alternatives courantes ?

- Pas de POO : C, efficace et portable mais inadapté pour gérer la complexité d'un logiciel de taille importante (exception notable : le noyau Linux)
- POO : Java, gestion automatique de la mémoire, plus "safe", multiplateforme, mais en général plus lent, plus restrictif

Fonctionnement et alternatives

Règles de base

- Identifier les classes nécessaires
- Fournir l'ensemble d'opérations nécessaires pour chaque classe
- Utiliser des mécanismes POO (i.e. héritage) pour structurer les fonctionnalités communes à de différentes classes

Quelles sont les alternatives courantes ?

- Pas de POO : C, efficace et portable mais inadapté pour gérer la complexité d'un logiciel de taille importante (exception notable : le noyau Linux)
- POO : Java, gestion automatique de la mémoire, plus "safe", multiplateforme, mais en général plus lent, plus restrictif
- autres langages POO (C#, Objective-C, Eiffel, Smalltalk etc.) ou avec des éléments POO (Go, PHP, Python etc.) : en général, plus loin de la couche matérielle que C++ , mais chacun avec ses "features"

Plan

- Historique
- Justification
- La création d'une application
- Le préprocesseur
- Les variables

Le workflow

Les éléments de base

- le fichier source qui contient le **point d'entrée** dans le programme

Le workflow

Les éléments de base

- le fichier source qui contient le **point d'entrée** dans le programme
- d'autres fichiers source éventuellement avec l'**implémentation** de certaines fonctionnalités auxiliaires

Le workflow

Les éléments de base

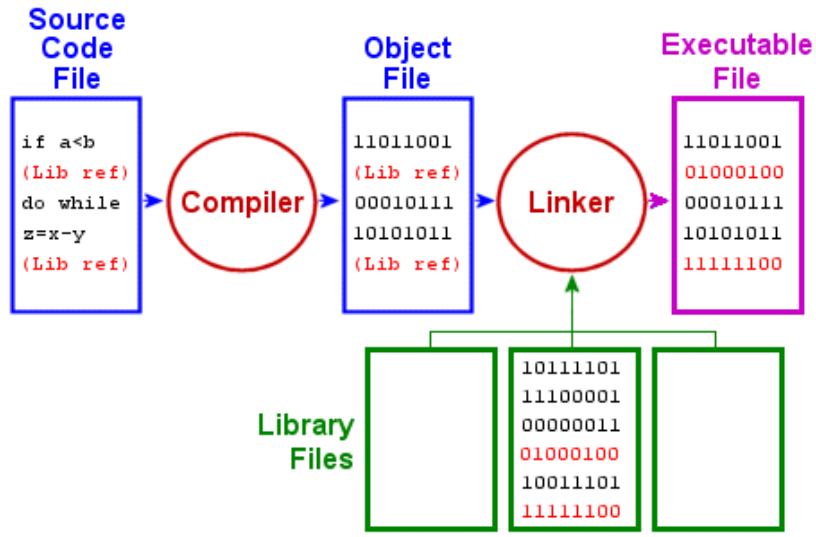
- le fichier source qui contient le **point d'entrée** dans le programme
- d'autres fichiers source éventuellement avec l'**implémentation** de certaines fonctionnalités auxiliaires
- des bibliothèques avec le code **binaire** de certaines fonctionnalités auxiliaires

Le workflow

Les éléments de base

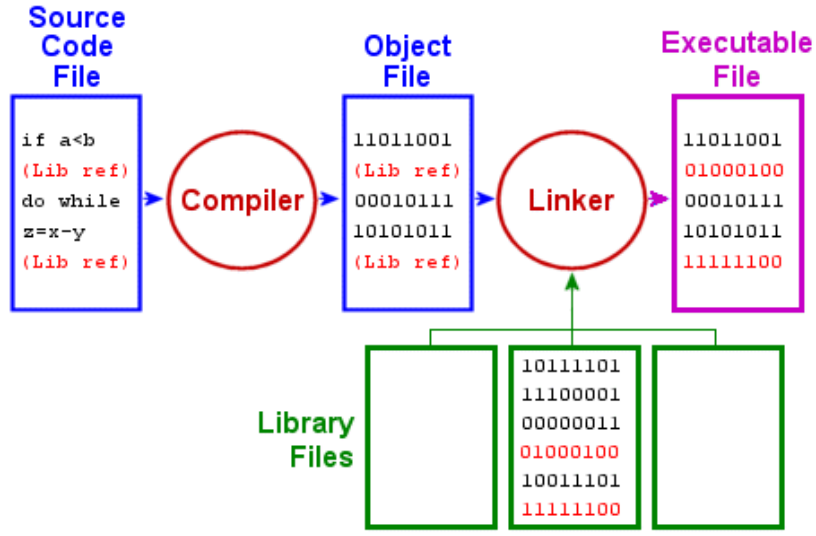
- le fichier source qui contient le **point d'entrée** dans le programme
- d'autres fichiers source éventuellement avec l'**implémentation** de certaines fonctionnalités auxiliaires
- des bibliothèques avec le code **binaire** de certaines fonctionnalités auxiliaires
- des fichiers entête avec la **définition** des fonctionnalités fournies soit par les autres fichiers source, soit par les bibliothèques

Un premier exemple



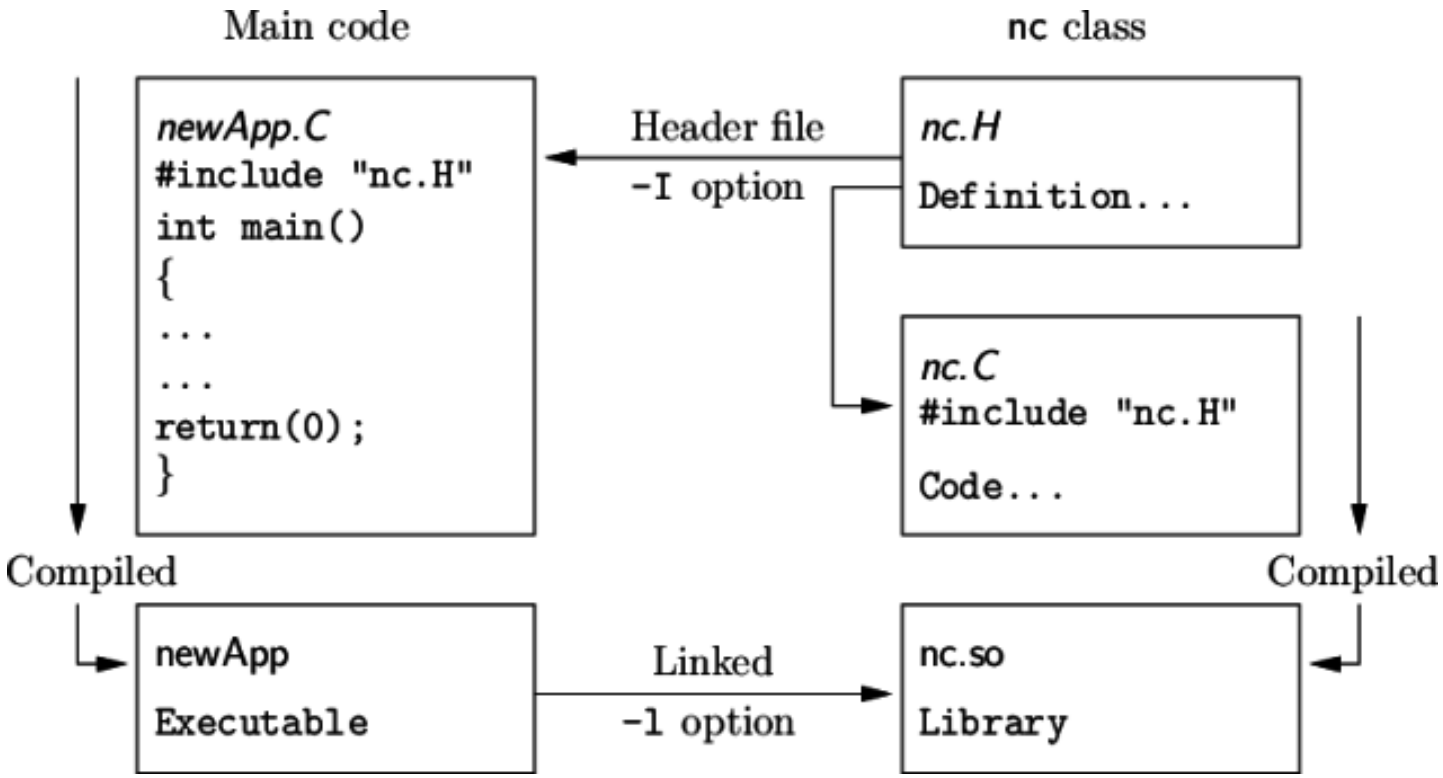
- le fichier source qui contient le **point d'entrée** dans le programme (obligatoire)

Un premier exemple



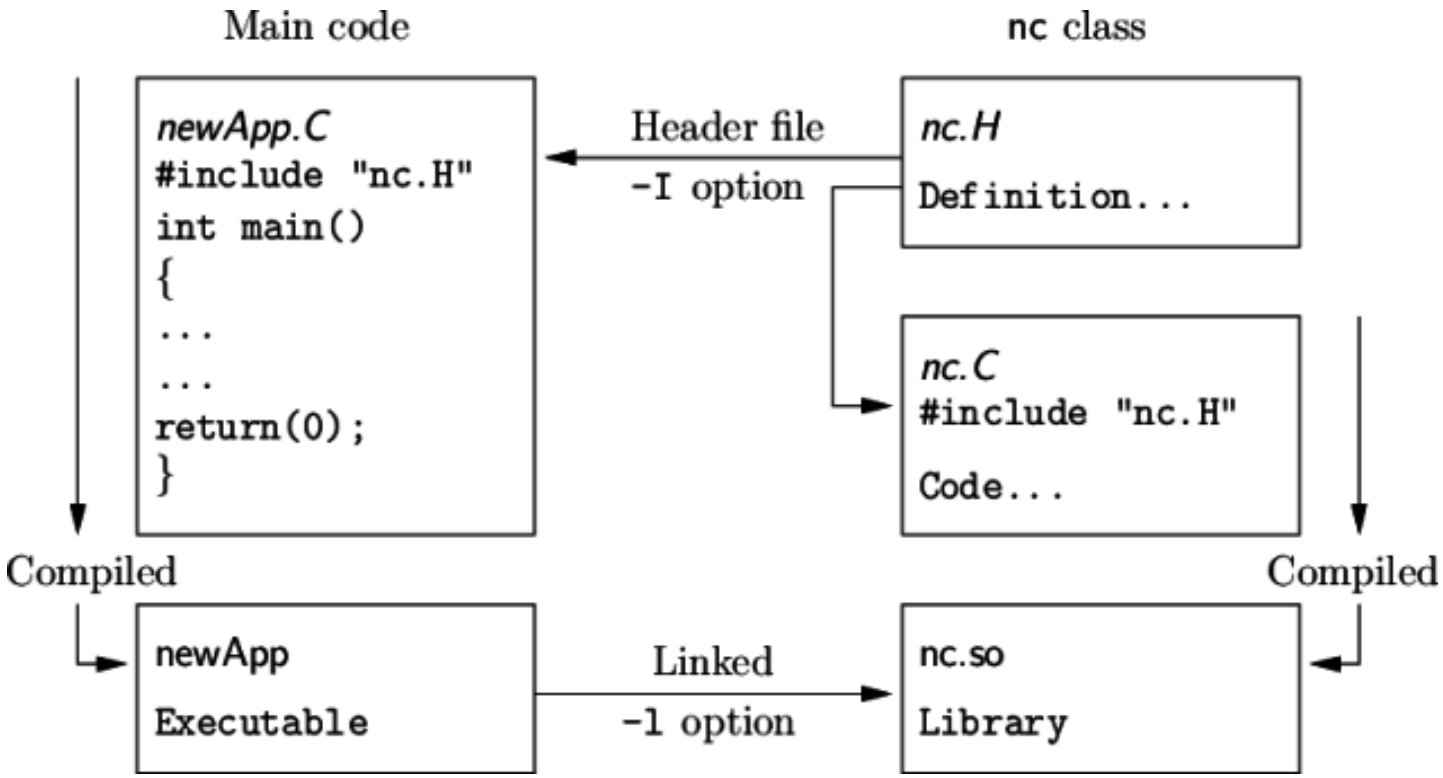
- le fichier source qui contient le **point d'entrée** dans le programme (obligatoire)
- appel à des fonctionnalités fournies par trois bibliothèques, qui existent déjà lors de la construction (l'édition des liens) de notre binaire

Un deuxième exemple



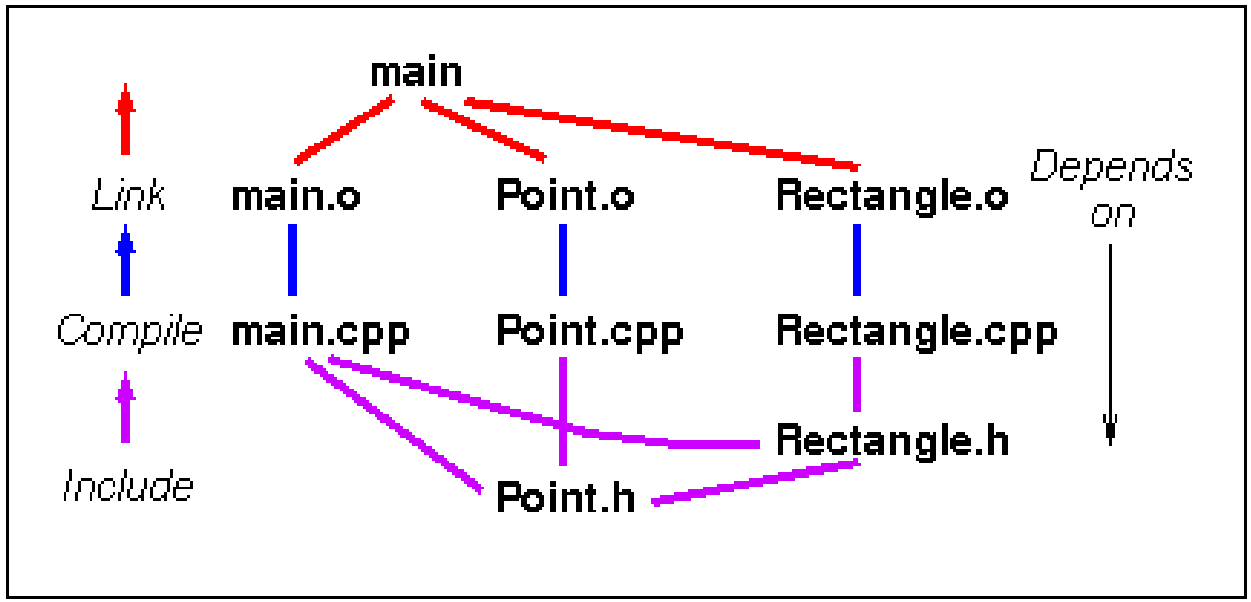
- on voit explicitement comment certaines fonctionnalités de la bibliothèque sont exposées à travers leurs définition par le fichier entête `nc.H`

Un deuxième exemple



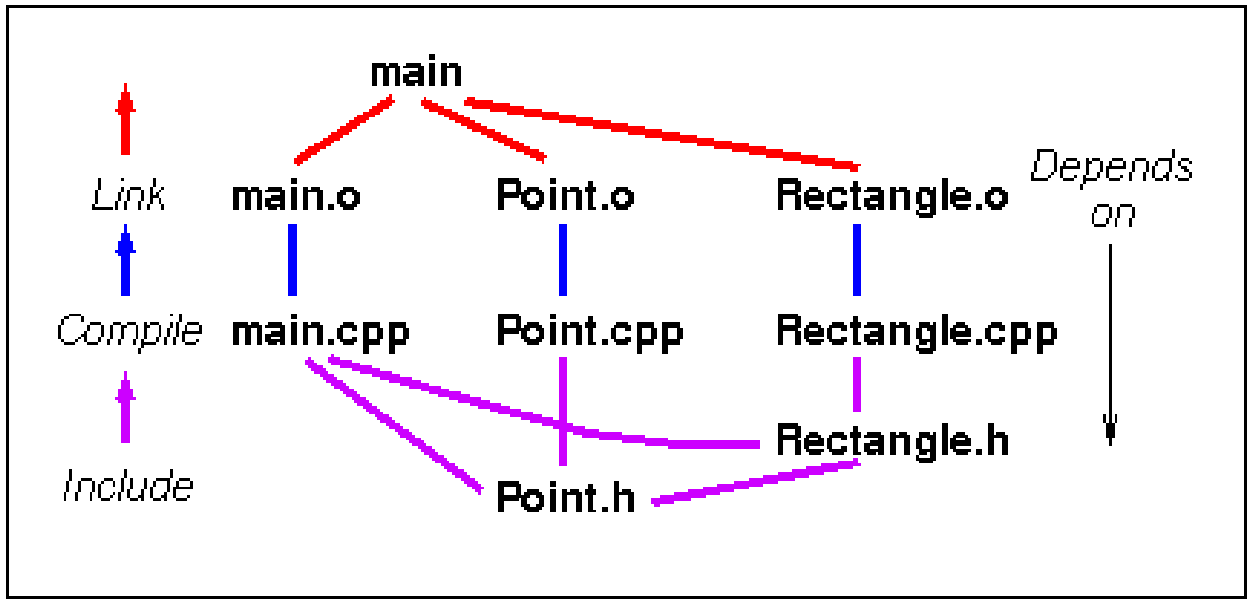
- on voit explicitement comment certaines fonctionnalités de la bibliothèque sont exposées à travers leurs définition par le fichier entête `nc.H`
- on peut les utiliser sans connaître les détails de leur implémentation

Un troisième exemple



- plusieurs fichiers sources sont reliés à travers les fichiers entête correspondants

Un troisième exemple



- plusieurs fichiers sources sont reliés à travers les fichiers entête correspondants
- tous les binaires sont construits lors de la compilation + édition des liens

Une application minimale

```
#include <stdio.h> //un fichier header repertoire standard
#include <iostream> //un fichier header librairie standard
#include "mesfuncs.h" //dans le repertoire courant, sinon erreur

/*
  le compilateur demande la fonction main
  qui est le point d'entree
  dans l'application
*/

int main(){
  printf("Hello world en C!\n");
  std::cout << "Hello world en C++!" << std::endl;
  return 0;
}
```

Pour que l'application puisse être générée, il faut que :

- les fonctionnalités invoquées soient définies dans les fichiers entête spécifiés en haut

Une application minimale

```
#include <stdio.h> //un fichier header repertoire standard
#include <iostream> //un fichier header librairie standard
#include "mesfuncs.h" //dans le repertoire courant, sinon erreur

/*
  le compilateur demande la fonction main
  qui est le point d'entree
  dans l'application
*/

int main(){
  printf("Hello world en C!\n");
  std::cout << "Hello world en C++!" << std::endl;
  return 0;
}
```

Pour que l'application puisse être générée, il faut que :

- les fonctionnalités invoquées soient définies dans les fichiers entête spécifiés en haut
- l'éditeur des liens ait accès aux bibliothèques qui fournissent l'implémentation de ces fonctionnalités

Plan

- Historique
- Justification
- La création d'une application
- Le préprocesseur
- Les variables

Le préprocesseur

Le préprocesseur effectue un travail de modification (translation) des fichiers source **avant** l'invocation du compilateur. Il gère notamment :

- l'inclusion des fichiers entête dans les fichiers source qui les invoquent

Le préprocesseur

Le préprocesseur effectue un travail de modification (translation) des fichiers source **avant** l'invocation du compilateur. Il gère notamment :

- l'inclusion des fichiers entête dans les fichiers source qui les invoquent
- l'expansion des macros

Le préprocesseur

Le préprocesseur effectue un travail de modification (translation) des fichiers source **avant** l'invocation du compilateur. Il gère notamment :

- l'inclusion des fichiers entête dans les fichiers source qui les invoquent
- l'expansion des macros
- la compilation conditionnelle

L'inclusion des entêtes

Danger : inclusion multiple du même fichier entête, ce qui engendre des déclarations répétées de la même fonctionnalité, et donc des erreurs de compilation.
Solution 1 - gardes d'inclusion ; dans le fichier `toto.h` :

```
#ifndef _TOTO_H_
#define _TOTO_H_
... //reste du fichier
#endif  // _TOTO_H_
```

Solution 2 - directive d'inclusion unique ; dans le fichier `toto.h` :

```
#pragma once
... //reste du fichier
```

Les deux méthodes sont efficaces en pratique, et posent des problèmes dans des cas très particuliers.

L'expansion des macros

Un mécanisme très populaire en C, pas si populaire que cela en C++ .

```
#define N 10
```

remplace au moment du preprocessing les occurrences de N par 10. On peut également définir des macros avec des paramètres, mais attention à la priorité des opérations !

```
#define ADD(a,b) a+b
#define MUL(a,b) a*b
...
MUL(ADD(a,b),ADD(c,d))
```

L'expansion de la dernière expression donne :

```
a+b*c+d
```

alors qu'on aurait souhaité obtenir

```
(a+b)*(c+d)
```

Pour cela, il ne faut pas oublier de mettre dans les macros des parenthèses de protection :

```
#define ADD(a,b) ((a)+(b))
#define MUL(a,b) ((a)*(b))
```

En général, on déconseille l'utilisation des macros (pourquoi ?).

La compilation conditionnelle

La modification du code source en fonction des variables qui sont accessible au préprocesseur.

Situation 1 : inclusion des fichiers entête en fonction de l'OS

```
#ifdef __unix__
  /* __unix__ is usually defined by compilers
  targeting Unix systems */
  # include <unistd.h>
#elif defined _WIN32
  /* _Win32 is usually defined by compilers
  targeting 32 or 64 bit Windows systems */
  # include <windows.h>
#endif
```

Situation 2 : exécution optionnelle de certaines partie du code

```
#if VERBOSE >= 2
  print("trace message");
#endif
```

Plan

- Historique
- Justification
- La création d'une application
- Le préprocesseur
- Les variables

Les variables

L'essentiel :

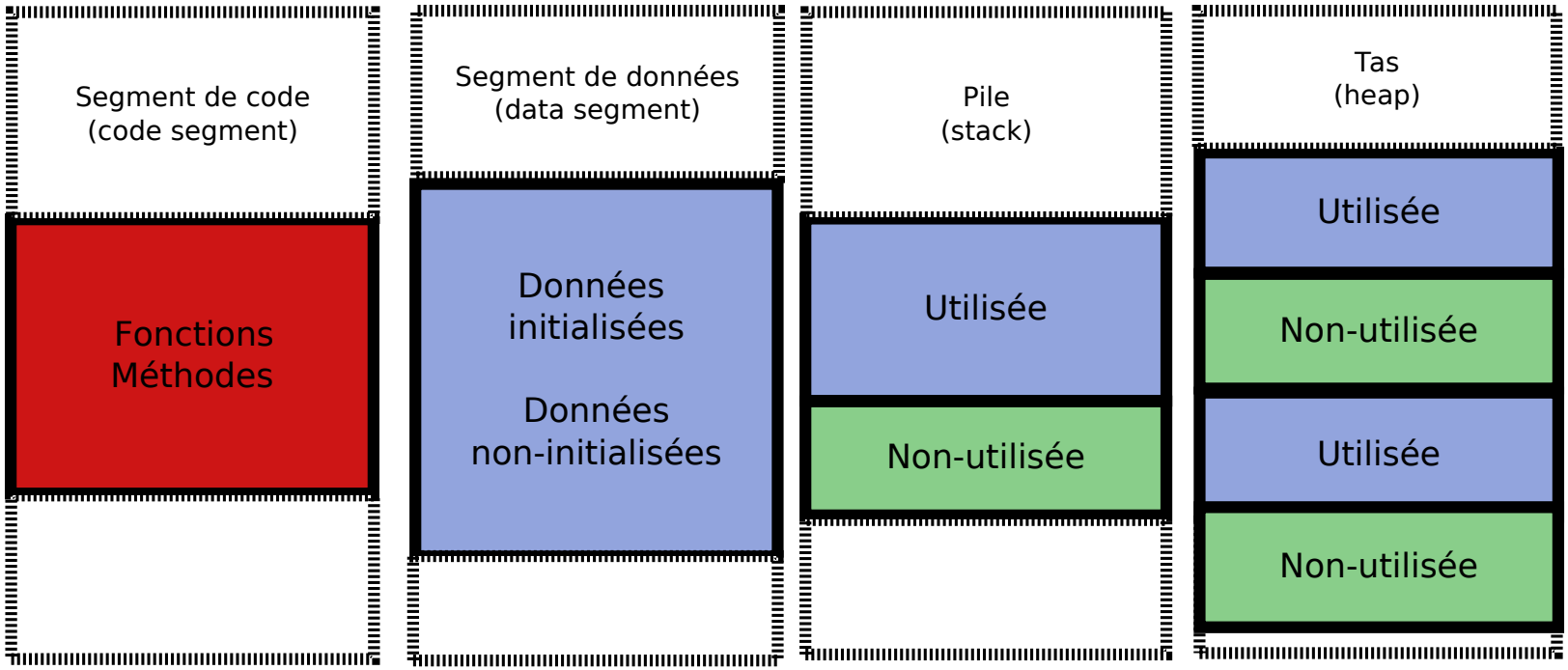
- Nom
  - unique
  - commence par une lettre
  - case-sensitive
- Type
  - fondamental ou construit
- Valeur
  - constante ou variable ?
  - initialisation

Pour bien maîtriser les variables (objets)

Il faut savoir :

- C++ est un **langage typé** : toute variable et toute expression ont un **unique** type
- Faire la distinction entre **type** et **variable**
- Toute variable occupe une **place mémoire** de la taille du type pendant sa vie
- La **durée de vie** d'une variable
- L'**espace mémoire** dans lequel une variable vit
- **Où** et **comment** une variable est accessible
- Si l'évaluation d'une expression mène vers une variable (un conteneur) ou non : **l-value** et **r-value**

Organisation mémoire proposée par C++



Emplacement des données par défaut :

- Segment de données : variables **globales** et **statiques**
- Pile : variables **locales** et **temporaires**, **paramètres** d'appel et de retour des fonctions / méthodes
- Tas : variables dynamiques
- Registres micro-processeur :
  - les variables précédées par les mots clé **register** (interprété comme une suggestion)
  - toute variable locale, temporaire ou paramètre de fonction **inline** que le compilateur considère nécessaire, dès que l'optimisation est demandée (mode Release)

Données : portée, accessibilité (lecture/écriture)

Directe (par le nom) :

- **Globales** : dans le module courant, et dans les autres à partir de la redéclaration avec **extern**
- **Globales statiques** : dans le module courant
- **Locales** : jusqu'à la fin du bloc courant
- **Paramètres** : à l'intérieur de la fonction appelée
- **Dynamiques**, **de retour** ou **temporaires** : jamais (pas de nom !)
- Masquage : **locales** > **paramètres** > **attributs** > **globales**
- Espaces : **instruction** < **bloc** < **fonction/méthode** < **classe** < **namespace** < **module** < **programme**

Données : portée, accessibilité (lecture/écriture)

Indirecte :

- En évaluant une expression qui vaut la variable en question
- Par **adresse** : pointeur ou référence
- Dans une structure/classe/union : par `.` ou `->`
- Dans un tableau : arithmétique des pointeurs `*`, `+`, `-`, `[]`
- accès à un autre espace par l'opérateur de résolution de portée `::`

Le langage C++ ne garantit pas que l'évaluation d'une expression mène vers une location mémoire valide, ceci est la responsabilité du concepteur !

Types de base disponibles

- Types **numériques** de base
  - **taille** en mémoire : 1, 2, 4, 8, 10, 16 octets
  - présence d'un **signe** : signé / non-signé
  - **granularité** :
    - **char** : 256 valeurs sur 1 octet
    - **int** : entier sur 2 ou 4 octets
    - **float** : flottant simple précision, 4 octets
    - **double** : flottant double précision, 8 octets
    - **long double** : flottant précision étendue, 10 octets
    - **bool** : deux valeurs, `true` ou `false` taille

Types de base disponibles

- Types **numériques** de base
  - **taille** en mémoire : 1, 2, 4, 8, 10, 16 octets
  - présence d'un **signe** : signé / non-signé
  - **granularité** :
    - **char** : 256 valeurs sur 1 octet
    - **int** : entier sur 2 ou 4 octets
    - **float** : flottant simple précision, 4 octets
    - **double** : flottant double précision, 8 octets
    - **long double** : flottant précision étendue, 10 octets
    - **bool** : deux valeurs, `true` ou `false` taille
- types d'**adressage** (permettent d'accéder à une variable en mémorisant son adresse)
  - **pointeur** : une adresse, un type et une taille
  - **référence** : une adresse, un type, une variable à "cloner" et une taille

Court rappel pointeurs

Opérations sur les pointeurs :

- valeur pointée : opérateur *
- déclaration du pointeur : `type *nomvar1, *nomvar2;`
- adresse d'une variable : opérateur &
- valeur d'un pointeur nul : 0

```
int n = 10;
int *p = &n;

/* p contient l'adresse memoire de la variable
  n qui est de type entier
*/

*p = 20;

/* on accede au contenu de la case sur laquelle
   p pointe, et on la modifie
*/
```

Références : déclaration

- La référence est un alias pour la variable à laquelle elle est associée

Références : déclaration

- La référence est un alias pour la variable à laquelle elle est associée
- On ne peut jamais déclarer une référence sans l'associer à une variable (par l'attribution à la déclaration)

```
type_t & reference = < var_ou_ref_de_type_t > ;
```

Références : déclaration

- La référence est un alias pour la variable à laquelle elle est associée
- On ne peut jamais déclarer une référence sans l'associer à une variable (par l'attribution à la déclaration)

```
type_t & reference = < var_ou_ref_de_type_t > ;
```

- Exemple :

```
int i=5, j=10;
int & ri=i; int &ri2 = ri;
cout << i   << " " << j    << " "
     << ri  << " " << ri2  << endl;
cout << &i  << " " << &j   << " "
     << &ri << " " << &ri2 << endl;
return 0;
```

Références : déclaration

- La référence est un alias pour la variable à laquelle elle est associée
- On ne peut jamais déclarer une référence sans l'associer à une variable (par l'attribution à la déclaration)

```
type_t & reference = < var_ou_ref_de_type_t > ;
```

- Exemple :

```
int i=5, j=10;
int & ri=i; int &ri2 = ri;
cout << i   << " " << j    << " "
     << ri  << " " << ri2  << endl;
cout << &i  << " " << &j   << " "
     << &ri << " " << &ri2 << endl;
return 0;
```

- On peut utiliser partout une référence à la place de sa variable associée (le type est le même à l'évaluation !)

Références : déclaration

- La référence est un alias pour la variable à laquelle elle est associée
- On ne peut jamais déclarer une référence sans l'associer à une variable (par l'attribution à la déclaration)

```
type_t & reference = < var_ou_ref_de_type_t > ;
```

- Exemple :

```
int i=5, j=10;
int & ri=i; int &ri2 = ri;
cout << i   << " " << j    << " "
     << ri  << " " << ri2  << endl;
cout << &i  << " " << &j   << " "
     << &ri << " " << &ri2 << endl;
return 0;
```

- On peut utiliser partout une référence à la place de sa variable associée (le type est le même à l'évaluation !)
- La référence cache en réalité un pointeur vers la variable associée. La référence d'une variable var vaut toujours *(&var)

Références : déclaration

- La référence est un alias pour la variable à laquelle elle est associée
- On ne peut jamais déclarer une référence sans l'associer à une variable (par l'attribution à la déclaration)

```
type_t & reference = < var_ou_ref_de_type_t > ;
```

- Exemple :

```
int i=5, j=10;
int & ri=i; int &ri2 = ri;
cout << i   << " " << j    << " "
     << ri  << " " << ri2  << endl;
cout << &i  << " " << &j   << " "
     << &ri << " " << &ri2 << endl;
return 0;
```

- On peut utiliser partout une référence à la place de sa variable associée (le type est le même à l'évaluation !)
- La référence cache en réalité un pointeur vers la variable associée. La référence d'une variable var vaut toujours *(&var)
- Q : peut-on avoir l'adresse d'une référence ?

Références : déclaration

- La référence est un alias pour la variable à laquelle elle est associée
- On ne peut jamais déclarer une référence sans l'associer à une variable (par l'attribution à la déclaration)

```
type_t & reference = < var_ou_ref_de_type_t > ;
```

- Exemple :

```
int i=5, j=10;
int & ri=i; int &ri2 = ri;
cout << i   << " " << j    << " "
     << ri  << " " << ri2  << endl;
cout << &i  << " " << &j   << " "
     << &ri << " " << &ri2 << endl;
return 0;
```

- On peut utiliser partout une référence à la place de sa variable associée (le type est le même à l'évaluation !)
- La référence cache en réalité un pointeur vers la variable associée. La référence d'une variable var vaut toujours *(&var)
- Q : peut-on avoir l'adresse d'une référence ?
- R : non. La référence en elle même n'est pas un objet, la référence est son référent.

Références : déclaration

- La référence est un alias pour la variable à laquelle elle est associée
- On ne peut jamais déclarer une référence sans l'associer à une variable (par l'attribution à la déclaration)

```
type_t & reference = < var_ou_ref_de_type_t > ;
```

- Exemple :

```
int i=5, j=10;
int & ri=i; int &ri2 = ri;
cout << i   << " " << j    << " "
     << ri  << " " << ri2  << endl;
cout << &i  << " " << &j   << " "
     << &ri << " " << &ri2 << endl;
return 0;
```

- On peut utiliser partout une référence à la place de sa variable associée (le type est le même à l'évaluation !)
- La référence cache en réalité un pointeur vers la variable associée. La référence d'une variable var vaut toujours *(&var)
- Q : peut-on avoir l'adresse d'une référence ?
- R : non. La référence en elle même n'est pas un objet, la référence est son référent.
- Q : peut-on modifier la variable associée ?

Références : déclaration

- La référence est un alias pour la variable à laquelle elle est associée
- On ne peut jamais déclarer une référence sans l'associer à une variable (par l'attribution à la déclaration)

```
type_t & reference = < var_ou_ref_de_type_t > ;
```

- Exemple :

```
int i=5, j=10;
int & ri=i; int &ri2 = ri;
cout << i   << " " << j    << " "
     << ri  << " " << ri2  << endl;
cout << &i  << " " << &j   << " "
     << &ri << " " << &ri2 << endl;
return 0;
```

- On peut utiliser partout une référence à la place de sa variable associée (le type est le même à l'évaluation !)
- La référence cache en réalité un pointeur vers la variable associée. La référence d'une variable var vaut toujours *(&var)
- Q : peut-on avoir l'adresse d'une référence ?
- R : non. La référence en elle même n'est pas un objet, la référence est son référent.
- Q : peut-on modifier la variable associée ?
- R : non. Cela est spécifié dans le standard ISO.

Références vs. pointeurs : à retenir

Pour quoi les références sont utiles :

- La référence pointe toujours vers un objet, alors que l'utilisation d'un pointeur nécessite une vérification (pas NULL)

Références vs. pointeurs : à retenir

Pour quoi les références sont utiles :

- La référence pointe toujours vers un objet, alors que l'utilisation d'un pointeur nécessite une vérification (pas NULL)
- ... mais les horreurs sont toujours possibles en C++ :

```
int& getLocalVariable()
{
    int x;
    return x;
}
```

Références vs. pointeurs : à retenir

Pour quoi les références sont utiles :

- La référence pointe toujours vers un objet, alors que l'utilisation d'un pointeur nécessite une vérification (pas NULL)
- ... mais les horreurs sont toujours possibles en C++ :

```
int& getLocalVariable()
{
    int x;
    return x;
}
```

- Sans faire rien de spécial, on évite de faire des copies au passage d'objets en paramètres de fonction

Références vs. pointeurs : à retenir

Pour quoi les références sont utiles :

- La référence pointe toujours vers un objet, alors que l'utilisation d'un pointeur nécessite une vérification (pas NULL)
- ... mais les horreurs sont toujours possibles en C++ :

```
int& getLocalVariable()
{
    int x;
    return x;
}
```

- Sans faire rien de spécial, on évite de faire des copies au passage d'objets en paramètres de fonction
- ... et on peut décider si la fonction a le droit de les modifier ou pas :

```
int workWithClass(
  MyClass& a_class_object )
{
  ...
}
```

OK pour modifier l'objet

```
int workWithClass(
   const MyClass& a_class_object )
 {
   ...
 }
```

interdiction de modifier

Références vs. pointeurs : à retenir

Pour quoi les références sont utiles :

- La référence pointe toujours vers un objet, alors que l'utilisation d'un pointeur nécessite une vérification (pas NULL)
- ... mais les horreurs sont toujours possibles en C++ :

```
int& getLocalVariable()
{
    int x;
    return x;
}
```

- Sans faire rien de spécial, on évite de faire des copies au passage d'objets en paramètres de fonction
- ... et on peut décider si la fonction a le droit de les modifier ou pas :

```
int workWithClass(
  MyClass& a_class_object )
{
  ...
}
```

OK pour modifier l'objet

```
int workWithClass(
   const MyClass& a_class_object )
 {
   ...
 }
```

interdiction de modifier

- Attention : références vers objets à allocation dynamique (ambiguïtés)

Exemple simple : références

```
float add1(float x1,float x2){
  return x1+x2;
}

void add2(float x1,float x2, float* res){
  *res=x1+x2;
}

void add3(float x1,float x2, float& res){
  res=x1+x2;
}

int main()
{
  float x1=0.5, x2=1, y;
  y = add1(x1,x2); // version retour de fonction
  add2(x1,x2,&y);  // version pointeur
  add3(x1,x2,y);   // version reference
  return 0;
}
```

Fonctions/méthodes : portée, accessibilité (appel)

Directe :

- Par le nom de la fonction/méthode et ()
- Le nom seul d'une fonction est une adresse dont le type est déterminé par le prototype de la fonction

```
int f(int x){
  return x;
}
int main(){
  int (*fp)(int);
  fp = &f;
  (*fp)(5); //utilisation canonique
  fp(6);    //permis en C aussi, utilisation courante
  return 0;
}
```

Indirecte :

- Expression dont l'évaluation mène à l'adresse d'une fonction, puis appel avec ()
- Pour une méthode il est impératif d'avoir comme expression un objet (ou adresse d'objet) valide, puis l'opérateur d'accès . ou -> et appel avec ()

Données : accessibilité

```
static int i1=30;
double d2=3.14;
float f ( float f1){
  char c='d' ;
  static int st=7;
  add (&i1,( int)c );
  st += 5;
  return f1;
}
int main(){
  f(3);f(4);
  return 0;
}
```

main.cpp

```
extern double d2;



void add ( int* pa,int b){
  long var2=32145;
  for( int i=0;i<10;i++){
    double d2=i;
    d2+=* pa ;
  }
  var2=d2;
}
```

part2.cpp

Données : accessibilité

```
static int i1=30;
double d2=3.14;
float f ( float f1){
  char c='d' ;
  static int st=7;
  add (&i1,( int)c );
  st += 5;
  return f1;
}
int main(){
  f(3);f(4);
  return 0;
}
```

main.cpp

```
extern double d2;



void add ( int* pa,int b){
  long var2=32145;
  for( int i=0;i<10;i++){
    double d2=i;
    d2+=* pa ;
  }
  var2=d2;
}
```

part2.cpp

La variable statique `i1` :

- variable globale, mais visible uniquement dans son module

Données : accessibilité

```
static int i1=30;
double d2=3.14;
float f ( float f1){
  char c='d' ;
  static int st=7;
  add (&i1,( int)c );
  st += 5;
  return f1;
}
int main(){
  f(3);f(4);
  return 0;
}
```

main.cpp

```
extern double d2;



void add ( int* pa,int b){
  long var2=32145;
  for( int i=0;i<10;i++){
    double d2=i;
    d2+=* pa ;
  }
  var2=d2;
}
```

part2.cpp

La variable statique `i1` :

- variable globale, mais visible uniquement dans son module
- accessible par son adresse dans les autres modules

Données : accessibilité

```
static int i1=30;
double d2=3.14;
float f ( float f1){
  char c='d' ;
  static int st=7;
  add (&i1,( int)c );
  st += 5;
  return f1;
}
int main(){
  f(3);f(4);
  return 0;
}
```

main.cpp

```
extern double d2;



void add ( int* pa,int b){
  long var2=32145;
  for( int i=0;i<10;i++){
    double d2=i;
    d2+=* pa ;
  }
  var2=d2;
}
```

part2.cpp

La variable statique `st` :

- variable locale, visible uniquement au niveau du bloc

Données : accessibilité

```
static int i1=30;
double d2=3.14;
float f ( float f1){
  char c='d' ;
  static int st=7;
  add (&i1,( int)c );
  st += 5;
  return f1;
}
int main(){
  f(3);f(4);
  return 0;
}
```

main.cpp

```
extern double d2;



void add ( int* pa,int b){
  long var2=32145;
  for( int i=0;i<10;i++){
    double d2=i;
    d2+=* pa ;
  }
  var2=d2;
}
```

part2.cpp

La variable statique `st` :

- variable locale, visible uniquement au niveau du bloc
- valeur persistante entre les parcours du bloc. Où réside t elle en mémoire ?

Données : accessibilité

```
static int i1=30;
double d2=3.14;
float f ( float f1){
  char c='d' ;
  static int st=7;
  add (&i1,( int)c );
  st += 5;
  return f1;
}
int main(){
  f(3);f(4);
  return 0;
}
```

main.cpp

```
extern double d2;



void add ( int* pa,int b){
  long var2=32145;
  for( int i=0;i<10;i++){
    double d2=i;
    d2+=* pa ;
  }
  var2=d2;
}
```

part2.cpp

La variable statique `st` :

- variable locale, visible uniquement au niveau du bloc
- valeur persistante entre les parcours du bloc. Où réside t elle en mémoire ?
- R : en data segment, soit dans la partie de données initialisées explicitement (i.e. `st`), soit dans la partie de données non-initialisées explicitement

Données : accessibilité

```
static int i1=30;
double d2=3.14;
float f ( float f1){
  char c='d' ;
  static int st=7;
  add (&i1,( int)c );
  st += 5;
  return f1;
}
int main(){
  f(3);f(4);
  return 0;
}
```

main.cpp

```
extern double d2;



void add ( int* pa,int b){
  long var2=32145;
  for( int i=0;i<10;i++){
    double d2=i;
    d2+=* pa ;
  }
  var2=d2;
}
```

part2.cpp

La variable statique `st` :

- variable locale, visible uniquement au niveau du bloc
- valeur persistante entre les parcours du bloc. Où réside t elle en mémoire ?
- R : en data segment, soit dans la partie de données initialisées explicitement (i.e. `st`), soit dans la partie de données non-initialisées explicitement
- initialisation effectuée une seule fois

Données : accessibilité

```
static int i1=30;
double d2=3.14;
float f ( float f1){
  char c='d' ;
  static int st=7;
  add (&i1,( int)c );
  st += 5;
  return f1;
}
int main(){
  f(3);f(4);
  return 0;
}
```

main.cpp

```
extern double d2;



void add ( int* pa,int b){
  long var2=32145;
  for( int i=0;i<10;i++){
    double d2=i;
    d2+=* pa ;
  }
  var2=d2;
}
```

part2.cpp

La variable statique `st` :

- variable locale, visible uniquement au niveau du bloc
- valeur persistante entre les parcours du bloc. Où réside t elle en mémoire ?
- R : en data segment, soit dans la partie de données initialisées explicitement (i.e. `st`), soit dans la partie de données non-initialisées explicitement
- initialisation effectuée une seule fois
- limiter l'utilisation des variables statiques locales

Données : accessibilité

```
static int i1=30;
double d2=3.14;
float f ( float f1){
  char c='d' ;
  static int st=7;
  add (&i1,( int)c );
  st += 5;
  return f1;
}
int main(){
  f(3);f(4);
  return 0;
}
```

main.cpp

```
extern double d2;



void add ( int* pa,int b){
  long var2=32145;
  for( int i=0;i<10;i++){
    double d2=i;
    d2+=* pa ;
  }
  var2=d2;
}
```

part2.cpp

La variable globale `d2` :

- visible au niveau du module

Données : accessibilité

```
static int i1=30;
double d2=3.14;
float f ( float f1){
  char c='d' ;
  static int st=7;
  add (&i1,( int)c );
  st += 5;
  return f1;
}
int main(){
  f(3);f(4);
  return 0;
}
```

main.cpp

```
extern double d2;



void add ( int* pa,int b){
  long var2=32145;
  for( int i=0;i<10;i++){
    double d2=i;
    d2+=* pa ;
  }
  var2=d2;
}
```

part2.cpp

La variable globale `d2` :

- visible au niveau du module
- visible également à l'extérieur du module avec une déclaration extern ...

Données : accessibilité

```
static int i1=30;
double d2=3.14;
float f ( float f1){
  char c='d' ;
  static int st=7;
  add (&i1,( int)c );
  st += 5;
  return f1;
}
int main(){
  f(3);f(4);
  return 0;
}
```

main.cpp

```
extern double d2;




void add ( int* pa,int b){
  long var2=32145;
  for( int i=0;i<10;i++){
    double d2=i;
    d2+=* pa ;
  }
  var2=d2;
}
```

part2.cpp

La variable globale `d2` :

- visible au niveau du module
- visible également à l'extérieur du module avec une déclaration extern ...
- ...mais attention aux masquages

Données : accessibilité

```cpp
static int i1=30;
double d2=3.14;
float f ( float f1){
  char c='d' ;
  static int st=7;
  add (&i1,( int)c );
  st += 5;
  return f1;
}
int main(){
  f(3);f(4);
  return 0;
}
```

main.cpp

```cpp
extern double d2;



void add ( int* pa,int b){
  long var2=32145;
  for( int i=0;i<10;i++){
    double d2=i;
    d2+=* pa ;
  }
  var2=d2;
}
```

part2.cpp

Le paramètre de fonction/méthode `f1` :

- visible uniquement au niveau de la fonction/méthode

Données : accessibilité

```
static int i1=30;
double d2=3.14;
float f ( float f1){
  char c='d' ;
  static int st=7;
  add (&i1,( int)c );
  st += 5;
  return f1;
}
int main(){
  f(3);f(4);
  return 0;
}
```

main.cpp

```
extern double d2;



void add ( int* pa,int b){
  long var2=32145;
  for( int i=0;i<10;i++){
    double d2=i;
    d2+=* pa ;
  }
  var2=d2;
}
```

part2.cpp

Le paramètre de fonction/méthode `f1` :

- visible uniquement au niveau de la fonction/méthode
- durée de vie : destruction à la fin de la fonction/méthode

Données : accessibilité

```
static int i1=30;
double d2=3.14;
float f ( float f1){
  char c='d' ;
  static int st=7;
  add (&i1,( int)c );
  st += 5;
  return f1;
}
int main(){
  f(3);f(4);
  return 0;
}
```

main.cpp

```
extern double d2;



void add ( int* pa,int b){
  long var2=32145;
  for( int i=0;i<10;i++){
    double d2=i;
    d2+=* pa ;
  }
  var2=d2;
}
```

part2.cpp

La variable compteur `i` :

- définie à l'intérieur de la boucle (for, while)

Données : accessibilité

```
static int i1=30;
double d2=3.14;
float f ( float f1){
  char c='d' ;
  static int st=7;
  add (&i1,( int)c );
  st += 5;
  return f1;
}
int main(){
  f(3);f(4);
  return 0;
}
```

main.cpp

```
extern double d2;



void add ( int* pa,int b){
  long var2=32145;
  for( int i=0;i<10;i++){
    double d2=i;
    d2+=* pa ;
  }
  var2=d2;
}
```

part2.cpp

La variable compteur `i` :

- définie à l'intérieur de la boucle (for, while)
- attention : les compilateurs Microsoft considèrent la déclaration d'un compteur comme une déclaration de variable locale, et donc le compteur persistera au delà de la boucle respective

Données : accessibilité

```
static int i1=30;
double d2=3.14;
float f ( float f1){
  char c='d' ;
  static int st=7;
  add (&i1,( int)c );
  st += 5;
  return f1;
}
int main(){
  f(3);f(4);
  return 0;
}
```

main.cpp

```
extern double d2;



void add ( int* pa,int b){
  long var2=32145;
  for( int i=0;i<10;i++){
    double d2=i;
    d2+=* pa ;
  }
  var2=d2;
}
```

part2.cpp

La variable compteur `i` :

- définie à l'intérieur de la boucle (for, while)
- attention : les compilateurs Microsoft considèrent la déclaration d'un compteur comme une déclaration de variable locale, et donc le compteur persistera au delà de la boucle respective
- l'option /Zc :forScope force le comportement standard (contexte restreint à l'intérieur de la boucle)

Accessibilité : espace de noms

- en C++ on peut cloisonner les variables et les fonctions globales à l'aide de l'espace de nom (namespace)
- déclarer dans un espace de noms

```
namespace Espace1
{
  // declarations de type (et de classes)
  // declarations de variables
  // declarations et definitions de fonctions
}
```

- y faire référence

```
Espace1::variable
Espace1::fonction()
```

méthode 1

```
using namespace Espace1;
variable
```

méthode 2

Exemple : espace de noms

```
namespace Anglais
{
  char * color []={"White"," Yellow "," Red ","Blue"};
  void Couleurs ( int i)
  {
    printf (" Color number %d is %s \ n", i,color [i]);
  }
}

namespace Francais
{
  char * color []={"Blanc","Jaune","Rouge","Bleu"};
  void Couleurs ( int i)
  {
    printf ("Couleur numero %d est %s \ n", i,color [i]);
  }
}
```

```
...
EspaceCouleurs::Francais::Couleurs(2);

...
```

accès indirect

```
using namespace EspaceCouleurs ;
using namespace Francais;
...
Couleurs(2);
```

accès direct

Données : durée de vie

- entre la **création** (allocation de la mémoire) et la **destruction** (la libération de la mémoire)

Données : durée de vie

- entre la **création** (allocation de la mémoire) et la **destruction** (la libération de la mémoire)
- **globales** et/ou **statiques** : toute la durée d'exécution du programme

Données : durée de vie

- entre la **création** (allocation de la mémoire) et la **destruction** (la libération de la mémoire)
- **globales** et/ou **statiques** : toute la durée d'exécution du programme
- **locales** : de la déclaration jusqu'à la fin du bloc (sauf **statiques**)

Données : durée de vie

- entre la **création** (allocation de la mémoire) et la **destruction** (la libération de la mémoire)
- **globales** et/ou **statiques** : toute la durée d'exécution du programme
- **locales** : de la déclaration jusqu'à la fin du bloc (sauf **statiques**)
- **temporaires** (sans nom) : jusqu'à la fin de l'instruction courante

Données : durée de vie

- entre la **création** (allocation de la mémoire) et la **destruction** (la libération de la mémoire)
- **globales** et/ou **statiques** : toute la durée d'exécution du programme
- **locales** : de la déclaration jusqu'à la fin du bloc (sauf **statiques**)
- **temporaires** (sans nom) : jusqu'à la fin de l'instruction courante
- **paramètres d'appel et de retour** : pendant la durée de l'appel de la fonction

Données : durée de vie

- entre la **création** (allocation de la mémoire) et la **destruction** (la libération de la mémoire)
- **globales** et/ou **statiques** : toute la durée d'exécution du programme
- **locales** : de la déclaration jusqu'à la fin du bloc (sauf **statiques**)
- **temporaires** (sans nom) : jusqu'à la fin de l'instruction courante
- **paramètres d'appel et de retour** : pendant la durée de l'appel de la fonction
- **dynamiques** : entre création par **new** et destruction par **delete**

Types disponibles par agrégation

- types **homogènes** (vecteurs d'éléments ou tableaux), une matrice 2D étant un vecteur de vecteurs. Caractéristiques :
  - type de l'élément
  - nombre d'éléments (spécifié à la création !)
  - adresse du premier élément

Types disponibles par agrégation

- types **homogènes** (vecteurs d'éléments ou tableaux), une matrice 2D étant un vecteur de vecteurs. Caractéristiques :
  - type de l'élément
  - nombre d'éléments (spécifié à la création !)
  - adresse du premier élément
- types **hybrides** (structures, unions, classes). Caractéristiques :
  - liste de **champs** avec droit d'accès, type et nom
  - pour classes : liste et nom des **méthodes**

Les tableaux à taille fixe

- Déclaration possible comme variable locale, globale ou statique. Pourquoi une taille fixe ?

Les tableaux à taille fixe

- Déclaration possible comme variable locale, globale ou statique. Pourquoi une taille fixe ?
- En tant que paramètre de fonction il n'est pas copié dans la pile ! (règle historique C : les tableaux sont interprétés comme des pointeurs s'ils sont passés en paramètre)

Les tableaux à taille fixe

- Déclaration possible comme variable locale, globale ou statique. Pourquoi une taille fixe ?
- En tant que paramètre de fonction il n'est pas copié dans la pile ! (règle historique C : les tableaux sont interprétés comme des pointeurs s'ils sont passés en paramètre)
- Pour s'amuser, le passage par valeur est possible de manière indirecte :

```
struct A { int arr[2]; };
void func(A a){ // copie profonde
  ...
}
```

Les tableaux à taille fixe

- Déclaration possible comme variable locale, globale ou statique. Pourquoi une taille fixe ?
- En tant que paramètre de fonction il n'est pas copié dans la pile ! (règle historique C : les tableaux sont interprétés comme des pointeurs s'ils sont passés en paramètre)
- Pour s'amuser, le passage par valeur est possible de manière indirecte :

```
struct A { int arr[2]; };
void func(A a){ // copie profonde
  ...
}
```

- L'opérateur sizeof donne sa vraie taille

Les tableaux à taille fixe

- Déclaration possible comme variable locale, globale ou statique. Pourquoi une taille fixe ?
- En tant que paramètre de fonction il n'est pas copié dans la pile ! (règle historique C : les tableaux sont interprétés comme des pointeurs s'ils sont passés en paramètre)
- Pour s'amuser, le passage par valeur est possible de manière indirecte :

```
struct A { int arr[2]; };
void func(A a){ // copie profonde
  ...
}
```

- L'opérateur sizeof donne sa vraie taille
- Il n'y a pas de tableaux 2-D ou N-D

Les tableaux à taille fixe

- Déclaration possible comme variable locale, globale ou statique. Pourquoi une taille fixe ?
- En tant que paramètre de fonction il n'est pas copié dans la pile ! (règle historique C : les tableaux sont interprétés comme des pointeurs s'ils sont passés en paramètre)
- Pour s'amuser, le passage par valeur est possible de manière indirecte :

```
struct A { int arr[2]; };
void func(A a){ // copie profonde
  ...
}
```

- L'opérateur sizeof donne sa vraie taille
- Il n'y a pas de tableaux 2-D ou N-D
- Considérons :

```
int tab[3][5];
```

Les tableaux à taille fixe

- Déclaration possible comme variable locale, globale ou statique. Pourquoi une taille fixe ?
- En tant que paramètre de fonction il n'est pas copié dans la pile ! (règle historique C : les tableaux sont interprétés comme des pointeurs s'ils sont passés en paramètre)
- Pour s'amuser, le passage par valeur est possible de manière indirecte :

```
struct A { int arr[2]; };
void func(A a){ // copie profonde
  ...
}
```

- L'opérateur sizeof donne sa vraie taille
- Il n'y a pas de tableaux 2-D ou N-D
- Considérons :

```
int tab[3][5];
```

- Rappel : une l-value est une expression qui désigne la localisation d'un objet en mémoire qui peut être examinée et réécrite ; une r-value est une expression qui a une valeur

Les tableaux à taille fixe

- Déclaration possible comme variable locale, globale ou statique. Pourquoi une taille fixe ?
- En tant que paramètre de fonction il n'est pas copié dans la pile ! (règle historique C : les tableaux sont interprétés comme des pointeurs s'ils sont passés en paramètre)
- Pour s'amuser, le passage par valeur est possible de manière indirecte :

```
struct A { int arr[2]; };
void func(A a){ // copie profonde
  ...
}
```

- L'opérateur sizeof donne sa vraie taille
- Il n'y a pas de tableaux 2-D ou N-D
- Considérons :

```
int tab[3][5];
```

- Rappel : une l-value est une expression qui désigne la localisation d'un objet en mémoire qui peut être examinée et réécrite ; une r-value est une expression qui a une valeur
- Q : quel est son emplacement et structure ?

Les tableaux à taille fixe

- Déclaration possible comme variable locale, globale ou statique. Pourquoi une taille fixe ?
- En tant que paramètre de fonction il n'est pas copié dans la pile ! (règle historique C : les tableaux sont interprétés comme des pointeurs s'ils sont passés en paramètre)
- Pour s'amuser, le passage par valeur est possible de manière indirecte :

```
struct A { int arr[2]; };
void func(A a){ // copie profonde
  ...
}
```

- L'opérateur sizeof donne sa vraie taille
- Il n'y a pas de tableaux 2-D ou N-D
- Considérons :

```
int tab[3][5];
```

- Rappel : une l-value est une expression qui désigne la localisation d'un objet en mémoire qui peut être examinée et réécrite ; une r-value est une expression qui a une valeur
- Q : quel est son emplacement et structure ?
- Q : type de tab ? l-value/r-value ?

Les tableaux à taille fixe

- Déclaration possible comme variable locale, globale ou statique. Pourquoi une taille fixe ?
- En tant que paramètre de fonction il n'est pas copié dans la pile ! (règle historique C : les tableaux sont interprétés comme des pointeurs s'ils sont passés en paramètre)
- Pour s'amuser, le passage par valeur est possible de manière indirecte :

```
struct A { int arr[2]; };
void func(A a){ // copie profonde
  ...
}
```

- L'opérateur sizeof donne sa vraie taille
- Il n'y a pas de tableaux 2-D ou N-D
- Considérons :

```
int tab[3][5];
```

- Rappel : une l-value est une expression qui désigne la localisation d'un objet en mémoire qui peut être examinée et réécrite ; une r-value est une expression qui a une valeur
- Q : quel est son emplacement et structure ?
- Q : type de tab ? l-value/r-value ?
- R : int[3][5] ; l-value non-modifiable (comme les types incomplets et les types const)

Les tableaux à taille fixe

- Déclaration possible comme variable locale, globale ou statique. Pourquoi une taille fixe ?
- En tant que paramètre de fonction il n'est pas copié dans la pile ! (règle historique C : les tableaux sont interprétés comme des pointeurs s'ils sont passés en paramètre)
- Pour s'amuser, le passage par valeur est possible de manière indirecte :

```
struct A { int arr[2]; };
void func(A a){ // copie profonde
  ...
}
```

- L'opérateur sizeof donne sa vraie taille
- Il n'y a pas de tableaux 2-D ou N-D
- Considérons :

```
int tab[3][5];
```

- Rappel : une l-value est une expression qui désigne la localisation d'un objet en mémoire qui peut être examinée et réécrite ; une r-value est une expression qui a une valeur
- Q : quel est son emplacement et structure ?
- Q : type de tab ? l-value/r-value ?
- R : int[3][5] ; l-value non-modifiable (comme les types incomplets et les types const)
- Q : type de tab[1] ? l-value/r-value ?

Les tableaux à taille fixe

- Déclaration possible comme variable locale, globale ou statique. Pourquoi une taille fixe ?
- En tant que paramètre de fonction il n'est pas copié dans la pile ! (règle historique C : les tableaux sont interprétés comme des pointeurs s'ils sont passés en paramètre)
- Pour s'amuser, le passage par valeur est possible de manière indirecte :

```
struct A { int arr[2]; };
void func(A a){ // copie profonde
  ...
}
```

- L'opérateur sizeof donne sa vraie taille
- Il n'y a pas de tableaux 2-D ou N-D
- Considérons :

```
int tab[3][5];
```

- Rappel : une l-value est une expression qui désigne la localisation d'un objet en mémoire qui peut être examinée et réécrite ; une r-value est une expression qui a une valeur
- Q : quel est son emplacement et structure ?
- Q : type de tab ? l-value/r-value ?
- R : int[3][5] ; l-value non-modifiable (comme les types incomplets et les types const)
- Q : type de tab[1] ? l-value/r-value ?
- R : int[5] ; l-value non-modifiable

Les tableaux à taille fixe

- Déclaration possible comme variable locale, globale ou statique. Pourquoi une taille fixe ?
- En tant que paramètre de fonction il n'est pas copié dans la pile ! (règle historique C : les tableaux sont interprétés comme des pointeurs s'ils sont passés en paramètre)
- Pour s'amuser, le passage par valeur est possible de manière indirecte :

```
struct A { int arr[2]; };
void func(A a){ // copie profonde
  ...
}
```

- L'opérateur sizeof donne sa vraie taille
- Il n'y a pas de tableaux 2-D ou N-D
- Considérons :

```
int tab[3][5];
```

- Rappel : une l-value est une expression qui désigne la localisation d'un objet en mémoire qui peut être examinée et réécrite ; une r-value est une expression qui a une valeur
- Q : quel est son emplacement et structure ?
- Q : type de tab ? l-value/r-value ?
- R : int[3][5] ; l-value non-modifiable (comme les types incomplets et les types const)
- Q : type de tab[1] ? l-value/r-value ?
- R : int[5] ; l-value non-modifiable
- Q : type de (tab+1) ? l-value/r-value ?

Les tableaux à taille fixe

- Déclaration possible comme variable locale, globale ou statique. Pourquoi une taille fixe ?
- En tant que paramètre de fonction il n'est pas copié dans la pile ! (règle historique C : les tableaux sont interprétés comme des pointeurs s'ils sont passés en paramètre)
- Pour s'amuser, le passage par valeur est possible de manière indirecte :

```
struct A { int arr[2]; };
void func(A a){ // copie profonde
  ...
}
```

- L'opérateur sizeof donne sa vraie taille
- Il n'y a pas de tableaux 2-D ou N-D
- Considérons :

```
int tab[3][5];
```

- Rappel : une l-value est une expression qui désigne la localisation d'un objet en mémoire qui peut être examinée et réécrite ; une r-value est une expression qui a une valeur
- Q : quel est son emplacement et structure ?
- Q : type de tab ? l-value/r-value ?
- R : int[3][5] ; l-value non-modifiable (comme les types incomplets et les types const)
- Q : type de tab[1] ? l-value/r-value ?
- R : int[5] ; l-value non-modifiable
- Q : type de (tab+1) ? l-value/r-value ?
- R : int(*)[5] ; r-value

Les tableaux à taille fixe

- Déclaration possible comme variable locale, globale ou statique. Pourquoi une taille fixe ?
- En tant que paramètre de fonction il n'est pas copié dans la pile ! (règle historique C : les tableaux sont interprétés comme des pointeurs s'ils sont passés en paramètre)
- Pour s'amuser, le passage par valeur est possible de manière indirecte :

```
struct A { int arr[2]; };
void func(A a){ // copie profonde
  ...
}
```

- L'opérateur sizeof donne sa vraie taille
- Il n'y a pas de tableaux 2-D ou N-D
- Considérons :

```
int tab[3][5];
```

- Rappel : une l-value est une expression qui désigne la localisation d'un objet en mémoire qui peut être examinée et réécrite ; une r-value est une expression qui a une valeur
- Q : quel est son emplacement et structure ?
- Q : type de tab ? l-value/r-value ?
- R : int[3][5] ; l-value non-modifiable (comme les types incomplets et les types const)
- Q : type de tab[1] ? l-value/r-value ?
- R : int[5] ; l-value non-modifiable
- Q : type de (tab+1) ? l-value/r-value ?
- R : int(*)[5] ; r-value
- Q : type de tab[0][3] ? l-value/r-value ?

Les tableaux à taille fixe

- Déclaration possible comme variable locale, globale ou statique. Pourquoi une taille fixe ?
- En tant que paramètre de fonction il n'est pas copié dans la pile ! (règle historique C : les tableaux sont interprétés comme des pointeurs s'ils sont passés en paramètre)
- Pour s'amuser, le passage par valeur est possible de manière indirecte :

```
struct A { int arr[2]; };
void func(A a){ // copie profonde
  ...
}
```

- L'opérateur sizeof donne sa vraie taille
- Il n'y a pas de tableaux 2-D ou N-D
- Considérons :

```
int tab[3][5];
```

- Rappel : une l-value est une expression qui désigne la localisation d'un objet en mémoire qui peut être examinée et réécrite ; une r-value est une expression qui a une valeur
- Q : quel est son emplacement et structure ?
- Q : type de tab ? l-value/r-value ?
- R : int[3][5] ; l-value non-modifiable (comme les types incomplets et les types const)
- Q : type de tab[1] ? l-value/r-value ?
- R : int[5] ; l-value non-modifiable
- Q : type de (tab+1) ? l-value/r-value ?
- R : int(*)[5] ; r-value
- Q : type de tab[0][3] ? l-value/r-value ?
- R : int ;l-value

Allocation dynamique

- En C++ on utilise les opérateurs new pour la création et delete pour la destruction

Allocation dynamique

- En C++ on utilise les opérateurs new pour la création et delete pour la destruction
- L'opérateur new retourne un pointeur vers une variable du type demandé, qui sera allouée dans le tas :

```
type_t * ptr1 =( type_t *) malloc (sizeof ( type_t )); // version C
type_t * ptr2 =new type_t ;                            // version C++
```

Allocation dynamique

- En C++ on utilise les opérateurs new pour la création et delete pour la destruction
- L'opérateur new retourne un pointeur vers une variable du type demandé, qui sera allouée dans le tas :

```
type_t * ptr1 =( type_t *) malloc (sizeof ( type_t )); // version C
type_t * ptr2 =new type_t ;                             // version C++
```

- L'opérateur new[] retourne un pointeur vers N variables du type demandé ; allocation également dans le tas :

```
type_t * ptr1 =( type_t *) malloc (sizeof ( type_t ) * N); // version C
type_t * ptr2 =new type_t[N] ;                              // version C++
```

Allocation dynamique

- En C++ on utilise les opérateurs new pour la création et delete pour la destruction
- L'opérateur new retourne un pointeur vers une variable du type demandé, qui sera allouée dans le tas :

```
type_t * ptr1 =( type_t *) malloc (sizeof ( type_t )); // version C
type_t * ptr2 =new type_t ;                            // version C++
```

- L'opérateur new[] retourne un pointeur vers N variables du type demandé ; allocation également dans le tas :

```
type_t * ptr1 =( type_t *) malloc (sizeof ( type_t ) * N); // version C
type_t * ptr2 =new type_t[N] ;                             // version C++
```

- L'opérateur delete libère la place occupée par la variable pointée par le pointeur passé en paramètre :

```
free( ptr1 );  // version C
delete ptr2 ;  // version C++
```

Allocation dynamique

- En C++ on utilise les opérateurs **new** pour la création et **delete** pour la destruction
- L'opérateur **new** retourne un pointeur vers une variable du type demandé, qui sera allouée dans le **tas** :

```
type_t * ptr1 =( type_t *) malloc (sizeof ( type_t )); // version C
type_t * ptr2 =new type_t ;                             // version C++
```

- L'opérateur **new[]** retourne un pointeur vers N variables du type demandé ; allocation également dans le **tas** :

```
type_t * ptr1 =( type_t *) malloc (sizeof ( type_t ) * N); // version C
type_t * ptr2 =new type_t[N] ;                              // version C++
```

- L'opérateur **delete** libère la place occupée par la variable pointée par le pointeur passé en paramètre :

```
free( ptr1 );  // version C
delete ptr2 ;  // version C++
```

- L'opérateur **delete[]** libère la place occupée par les N variables pointées par le pointeur passé en paramètre :

```
free( ptr1 );  // version C
delete[] ptr2 ;  // version C++
```

Allocation dynamique

- En C++ on utilise les opérateurs new pour la création et delete pour la destruction
- L'opérateur new retourne un pointeur vers une variable du type demandé, qui sera allouée dans le tas :

```
type_t * ptr1 =( type_t *) malloc (sizeof ( type_t )); // version C
type_t * ptr2 =new type_t ;                             // version C++
```

- L'opérateur new[] retourne un pointeur vers N variables du type demandé ; allocation également dans le tas :

```
type_t * ptr1 =( type_t *) malloc (sizeof ( type_t ) * N); // version C
type_t * ptr2 =new type_t[N] ;                              // version C++
```

- L'opérateur delete libère la place occupée par la variable pointée par le pointeur passé en paramètre :

```
free( ptr1 );  // version C
delete ptr2 ;  // version C++
```

- L'opérateur delete[] libère la place occupée par les N variables pointées par le pointeur passé en paramètre :

```
free( ptr1 );  // version C
delete[] ptr2 ;  // version C++
```

- Le pointeur fourni par new est votre seul lien avec la (les) variable(s) allouée(s) ; ne le modifiez pas car il faut le passer à delete pour libérer la mémoire

Allocation dynamique

- En C++ on utilise les opérateurs new pour la création et delete pour la destruction
- L'opérateur new retourne un pointeur vers une variable du type demandé, qui sera allouée dans le tas :

```
type_t * ptr1 =( type_t *) malloc (sizeof ( type_t )); // version C
type_t * ptr2 =new type_t ;                            // version C++
```

- L'opérateur new[] retourne un pointeur vers N variables du type demandé ; allocation également dans le tas :

```
type_t * ptr1 =( type_t *) malloc (sizeof ( type_t ) * N); // version C
type_t * ptr2 =new type_t[N] ;                             // version C++
```

- L'opérateur delete libère la place occupée par la variable pointée par le pointeur passé en paramètre :

```
free( ptr1 );  // version C
delete ptr2 ;  // version C++
```

- L'opérateur delete[] libère la place occupée par les N variables pointées par le pointeur passé en paramètre :

```
free( ptr1 );  // version C
delete[] ptr2 ;  // version C++
```

- Le pointeur fourni par new est votre seul lien avec la (les) variable(s) allouée(s) ; ne le modifiez pas car il faut le passer à delete pour libérer la mémoire
- C'est à vous de mémoriser le nombre d'éléments alloués, il n'y a pas d'autre moyen pour retrouver l'information.

Allocation dynamique

- En C++ on utilise les opérateurs **new** pour la création et **delete** pour la destruction
- L'opérateur **new** retourne un pointeur vers une variable du type demandé, qui sera allouée dans le **tas** :

```
type_t * ptr1 =( type_t *) malloc (sizeof ( type_t )); // version C
type_t * ptr2 =new type_t ;                             // version C++
```

- L'opérateur **new[]** retourne un pointeur vers N variables du type demandé ; allocation également dans le **tas** :

```
type_t * ptr1 =( type_t *) malloc (sizeof ( type_t ) * N); // version C
type_t * ptr2 =new type_t[N] ;                             // version C++
```

- L'opérateur **delete** libère la place occupée par la variable pointée par le pointeur passé en paramètre :

```
free( ptr1 );   // version C
delete ptr2 ;   // version C++
```

- L'opérateur **delete[]** libère la place occupée par les N variables pointées par le pointeur passé en paramètre :

```
free( ptr1 );   // version C
delete[] ptr2 ;  // version C++
```

- Le pointeur fourni par **new** est votre seul lien avec la (les) variable(s) allouée(s) ; ne le modifiez pas car il faut le passer à **delete** pour libérer la mémoire
- C'est à vous de mémoriser le nombre d'éléments alloués, il n'y a pas d'autre moyen pour retrouver l'information.
- Comment est-ce que **delete[]** connaît la taille du bloc mémoire à libérer ?

Allocation dynamique

- En C++ on utilise les opérateurs **new** pour la création et **delete** pour la destruction
- L'opérateur **new** retourne un pointeur vers une variable du type demandé, qui sera allouée dans le **tas** :

```
type_t * ptr1 =( type_t *) malloc (sizeof ( type_t )); // version C
type_t * ptr2 =new type_t ;                             // version C++
```

- L'opérateur **new[]** retourne un pointeur vers N variables du type demandé ; allocation également dans le **tas** :

```
type_t * ptr1 =( type_t *) malloc (sizeof ( type_t ) * N); // version C
type_t * ptr2 =new type_t[N] ;                              // version C++
```

- L'opérateur **delete** libère la place occupée par la variable pointée par le pointeur passé en paramètre :

```
free( ptr1 );  // version C
delete ptr2 ;  // version C++
```

- L'opérateur **delete[]** libère la place occupée par les N variables pointées par le pointeur passé en paramètre :

```
free( ptr1 );  // version C
delete[] ptr2 ;  // version C++
```

- Le pointeur fourni par **new** est votre seul lien avec la (les) variable(s) allouée(s) ; ne le modifiez pas car il faut le passer à **delete** pour libérer la mémoire
- C'est à vous de mémoriser le nombre d'éléments alloués, il n'y a pas d'autre moyen pour retrouver l'information.
- Comment est-ce que **delete[]** connaît la taille du bloc mémoire à libérer ?
- Le pointeur n'est pas modifié après la libération, il garde toujours l'adresse qui maintenant est invalide ! On conseille de le réinitialiser à zéro.

Allocation dynamique : exemple

- Allocation, utilisation et libération d'un tableau de double :

```
void test1( int taille){
  double * tabd = new double[taille];
  for( int i=0; i<taille; i++)
    tabd [i]=rand()/1000;
  delete [] tabd ;
}
```

Allocation dynamique : exemple

- Allocation, utilisation et libération d'un tableau de double :

```
void test1( int taille){
  double * tabd = new double[taille];
  for( int i=0; i<taille; i++)
    tabd [i]=rand()/1000;
  delete [] tabd ;
}
```

- Testez la fonction et dessiner l'emplacement de chaque variable.

Allocation dynamique : exemple

- Allocation, utilisation et libération d'un tableau de double :

```
void test1( int taille){
  double * tabd = new double[taille];
  for( int i=0; i<taille; i++)
    tabd [i]=rand()/1000;
  delete [] tabd ;
}
```

- Testez la fonction et dessiner l'emplacement de chaque variable.
- Implémentez une fonction qui alloue une matrice 2D d'entiers signés, de taille voulue, et une autre qui la libère :
  - il faut passer par un tableau de pointeurs pour les lignes
  - dessinez l'emplacement des variables en mémoire et leurs relations (architecture de données)
  - testez la matrice dans le programme principal, en comparant son utilisation à une matrice 2D locale (initialisation et affichage)

Pile vs. tas

Caractéristiques déclaration variables pile :

- Allocation, libération de mémoire très rapides

Avantages déclaration variables tas :

Pile vs. tas

Caractéristiques déclaration variables pile :

- Allocation, libération de mémoire très rapides
- Idéale pour les variables de taille réduite à durée de vie limitée

Avantages déclaration variables tas :

Pile vs. tas

Caractéristiques déclaration variables pile :

- Allocation, libération de mémoire très rapides
- Idéale pour les variables de taille réduite à durée de vie limitée
- Espace en mémoire relativement limité, mais utilisation compacte

Avantages déclaration variables tas :

Pile vs. tas

Caractéristiques déclaration variables pile :

- Allocation, libération de mémoire très rapides
- Idéale pour les variables de taille réduite à durée de vie limitée
- Espace en mémoire relativement limité, mais utilisation compacte
- Contraintes imposées par le mécanisme Last In First Out (LIFO) : libération de la mémoire uniquement à partir du sommet

Avantages déclaration variables tas :

Pile vs. tas

Caractéristiques déclaration variables pile :

- Allocation, libération de mémoire très rapides
- Idéale pour les variables de taille réduite à durée de vie limitée
- Espace en mémoire relativement limité, mais utilisation compacte
- Contraintes imposées par le mécanisme Last In First Out (LIFO) : libération de la mémoire uniquement à partir du sommet

Avantages déclaration variables tas :

- Accès à un espace mémoire important

Pile vs. tas

Caractéristiques déclaration variables pile :

- Allocation, libération de mémoire très rapides
- Idéale pour les variables de taille réduite à durée de vie limitée
- Espace en mémoire relativement limité, mais utilisation compacte
- Contraintes imposées par le mécanisme Last In First Out (LIFO) : libération de la mémoire uniquement à partir du sommet

Avantages déclaration variables tas :

- Accès à un espace mémoire important
- Persistance des variables à travers l'exécution

Pile vs. tas

Caractéristiques déclaration variables pile :

- Allocation, libération de mémoire très rapides
- Idéale pour les variables de taille réduite à durée de vie limitée
- Espace en mémoire relativement limité, mais utilisation compacte
- Contraintes imposées par le mécanisme Last In First Out (LIFO) : libération de la mémoire uniquement à partir du sommet

Avantages déclaration variables tas :

- Accès à un espace mémoire important
- Persistance des variables à travers l'exécution
- Possibilité de réallocation des variables

Pile vs. tas

Caractéristiques déclaration variables pile :

- Allocation, libération de mémoire très rapides
- Idéale pour les variables de taille réduite à durée de vie limitée
- Espace en mémoire relativement limité, mais utilisation compacte
- Contraintes imposées par le mécanisme Last In First Out (LIFO) : libération de la mémoire uniquement à partir du sommet

Avantages déclaration variables tas :

- Accès à un espace mémoire important
- Persistance des variables à travers l'exécution
- Possibilité de réallocation des variables
- Fragmentation de la mémoire

Pile vs. tas

Caractéristiques déclaration variables pile :

- Allocation, libération de mémoire très rapides
- Idéale pour les variables de taille réduite à durée de vie limitée
- Espace en mémoire relativement limité, mais utilisation compacte
- Contraintes imposées par le mécanisme Last In First Out (LIFO) : libération de la mémoire uniquement à partir du sommet

Avantages déclaration variables tas :

- Accès à un espace mémoire important
- Persistance des variables à travers l'exécution
- Possibilité de réallocation des variables
- Fragmentation de la mémoire
- Des accès significativement plus lents pour allocation

Fonctions C++

- Chaque fonction en C++ est identifiée uniquement par sa signature : nom et types d'arguments :

```
type_ret Fonct1( type1 p1, type2 p2, type3 p3);
```

Fonctions C++

- Chaque fonction en C++ est identifiée uniquement par sa signature : nom et types d'arguments :

```
type_ret Fonct1( type1 p1, type2 p2, type3 p3);
```

- La signature ainsi obtenue détermine le nom décoré (mangled) de la fonction

Fonctions C++

- Chaque fonction en C++ est identifiée uniquement par sa signature : nom et types d'arguments :

```
type_ret Fonct1( type1 p1, type2 p2, type3 p3);
```

- La signature ainsi obtenue détermine le nom décoré (mangled) de la fonction
- On peut avoir des fonctions C++ avec le même nom mais des signatures différentes

Fonctions C++

- Chaque fonction en C++ est identifiée uniquement par sa signature : nom et types d'arguments :

```
type_ret Fonct1( type1 p1, type2 p2, type3 p3);
```

- La signature ainsi obtenue détermine le nom décoré (mangled) de la fonction
- On peut avoir des fonctions C++ avec le même nom mais des signatures différentes
- On peut avoir des paramètres avec des valeurs par défaut

Fonctions C++ : les paramètres par défaut

- Les paramètres avec des valeurs par défaut sont toujours les derniers :

```
float Mult ( float p1, float p2, float p3 =1.0f , float p4 =1.0f );
```

Fonctions C++ : les paramètres par défaut

- Les paramètres avec des valeurs par défaut sont toujours les derniers :

```
float Mult ( float p1, float p2, float p3 =1.0f , float p4 =1.0f );
```

- Il faut déclarer une seule fois les valeurs par défaut :
  - dans le prototype s'il existe
  - sinon dans la définition

Fonctions C++ : les paramètres par défaut

- Les paramètres avec des valeurs par défaut sont toujours les derniers :

```
float Mult ( float p1, float p2, float p3 =1.0f , float p4 =1.0f );
```

- Il faut déclarer une seule fois les valeurs par défaut :
  - dans le prototype s'il existe
  - sinon dans la définition
- On peut appeler Mult avec 2, 3 ou 4 paramètres, le compilateur rajoute ceux qui manquent.

Fonctions C++ : les paramètres par défaut

- Les paramètres avec des valeurs par défaut sont toujours les derniers :

```
float Mult ( float p1, float p2, float p3 =1.0f , float p4 =1.0f );
```

- Il faut déclarer une seule fois les valeurs par défaut :
  - dans le prototype s'il existe
  - sinon dans la définition
- On peut appeler Mult avec 2, 3 ou 4 paramètres, le compilateur rajoute ceux qui manquent.
- Écrire la fonction, faites afficher les paramètres et testez-là avec 2, 3 et 4 paramètres

Fonctions C++ : protection à la modification

- Parfois on utilise des références ou des pointeurs pour réduire le temps d'appel et la mémoire occupée dans la pile.

Fonctions C++ : protection à la modification

- Parfois on utilise des références ou des pointeurs pour réduire le temps d'appel et la mémoire occupée dans la pile.
- Mais on ne veut pas modifier les valeurs pointées ou référencées. Alors, il faut utiliser le mot **const** :

```
void AfficheTab (double* t1, int taille);          // pas de protection
void AfficheTab (const double* t1, int taille);    // protection

void Add (float & a, float & b, float & res);                // pas de prot.
void Add (const float & a, const float & b, float & res); // protection
```

Fonctions C++ : protection à la modification

- Parfois on utilise des références ou des pointeurs pour réduire le temps d'appel et la mémoire occupée dans la pile.
- Mais on ne veut pas modifier les valeurs pointées ou référencées. Alors, il faut utiliser le mot **const** :

```
void AfficheTab (double* t1, int taille);          // pas de protection
void AfficheTab (const double* t1, int taille);    // protection

void Add (float & a, float & b, float & res);                // pas de prot.
void Add (const float & a, const float & b, float & res); // protection
```

- Le compilateur vérifie que la variable pointée ou référencée n'est pas modifiée par le code

Fonctions C++ : protection à la modification

- Parfois on utilise des références ou des pointeurs pour réduire le temps d'appel et la mémoire occupée dans la pile.
- Mais on ne veut pas modifier les valeurs pointées ou référencées. Alors, il faut utiliser le mot **const** :

```
void AfficheTab (double* t1, int taille);          // pas de protection
void AfficheTab (const double* t1, int taille);    // protection

void Add (float & a, float & b, float & res);                // pas de prot.
void Add (const float & a, const float & b, float & res); // protection
```

- Le compilateur vérifie que la variable pointée ou référencée n'est pas modifiée par le code
- Idem pour les références ou pour les pointeurs de retour

Exemple : protection à la modification

- Concevoir une fonction SearchMin qui recherche l'élément de valeur minimale dans un tableau de float passé en paramètre (avec sa taille) et qui retourne l'adresse de cet élément.

Exemple : protection à la modification

- Concevoir une fonction SearchMin qui recherche l'élément de valeur minimale dans un tableau de float passé en paramètre (avec sa taille) et qui retourne l'adresse de cet élément.
- Testez-la avec un tableau à 5 éléments. Mémoriser l'adresse retournée et afficher dans la fonction main la valeur minimale trouvée.

Exemple : protection à la modification

- Concevoir une fonction SearchMin qui recherche l'élément de valeur minimale dans un tableau de float passé en paramètre (avec sa taille) et qui retourne l'adresse de cet élément.
- Testez-la avec un tableau à 5 éléments. Mémoriser l'adresse retournée et afficher dans la fonction main la valeur minimale trouvée.
- Protéger à la modification les éléments du tableau qui arrive en paramètre de SearchMin . Quel est le problème soulevé par le compilateur ?

Exemple : protection à la modification

- Concevoir une fonction SearchMin qui recherche l'élément de valeur minimale dans un tableau de float passé en paramètre (avec sa taille) et qui retourne l'adresse de cet élément.
- Testez-la avec un tableau à 5 éléments. Mémoriser l'adresse retournée et afficher dans la fonction main la valeur minimale trouvée.
- Protéger à la modification les éléments du tableau qui arrive en paramètre de SearchMin . Quel est le problème soulevé par le compilateur ?
- Comment le résoudre ? Suivre la propagation de la contrainte.

Le qualificatif volatile

- Si une variable risque d'être modifiée par plusieurs régions de code, qui s'exécutent en parallèle (multi-tâche , multi-thread ou interruptions), alors lui donner le qualificatif volatile oblige le compilateur à désactiver toute optimisation liée aux accès

Le qualificatif volatile

- Si une variable risque d'être modifiée par plusieurs régions de code, qui s'exécutent en parallèle (multi-tâche , multi-thread ou interruptions), alors lui donner le qualificatif volatile oblige le compilateur à désactiver toute optimisation liée aux accès
- il va lire chaque fois la variable même s'il l'avait déjà fait une instruction avant

Le qualificatif volatile

- Si une variable risque d'être modifiée par plusieurs régions de code, qui s'exécutent en parallèle (multi-tâche , multi-thread ou interruptions), alors lui donner le qualificatif volatile oblige le compilateur à désactiver toute optimisation liée aux accès
- il va lire chaque fois la variable même s'il l'avait déjà fait une instruction avant
- il va l'écrire aussitôt, sans la garder dans la mémoire cache ou dans des registres

Le qualificatif volatile

- Si une variable risque d'être modifiée par plusieurs régions de code, qui s'exécutent en parallèle (multi-tâche , multi-thread ou interruptions), alors lui donner le qualificatif volatile oblige le compilateur à désactiver toute optimisation liée aux accès
- il va lire chaque fois la variable même s'il l'avait déjà fait une instruction avant
- il va l'écrire aussitôt, sans la garder dans la mémoire cache ou dans des registres
- attention : cela ne rend pas l'accès atomique

Le qualificatif volatile

- Si une variable risque d'être modifiée par plusieurs régions de code, qui s'exécutent en parallèle (multi-tâche , multi-thread ou interruptions), alors lui donner le qualificatif volatile oblige le compilateur à désactiver toute optimisation liée aux accès
- il va lire chaque fois la variable même s'il l'avait déjà fait une instruction avant
- il va l'écrire aussitôt, sans la garder dans la mémoire cache ou dans des registres
- attention : cela ne rend pas l'accès atomique
- attention : la vitesse d'accès diminue dramatiquement (10-100 fois)

Types en C++

- natifs au langage : scalaires (et pointeurs)

Types en C++

- natifs au langage : scalaires (et pointeurs)
- complexes par agrégation homogène (+ **typedef**)

Types en C++

- natifs au langage : scalaires (et pointeurs)
- complexes par agrégation homogène (+ **typedef**)
- complexes par agrégation hybride : union/structure/classe + **typedef**

Types en C++

- natifs au langage : scalaires (et pointeurs)
- complexes par agrégation homogène (+ **typedef**)
- complexes par agrégation hybride : union/structure/classe + **typedef**
- mais le **typedef** n'est pas obligatoire en C++ pour l'agrégation hybride car la syntaxe est plus libre ...

Types en C++

- natifs au langage : scalaires (et pointeurs)
- complexes par agrégation homogène (+ **typedef**)
- complexes par agrégation hybride : union/structure/classe + **typedef**
- mais le **typedef** n'est pas obligatoire en C++ pour l'agrégation hybride car la syntaxe est plus libre ...
- avec **typedef** on ne déclare que des types :

```
typedef union UnionType1
{
// ...
} UnionType2 ;
typedef struct StructType1
{
// ...
} StructType2 ;
typedef class ClassType1
{
// ...
} ClassType2 ;
```

Types en C++

- natifs au langage : scalaires (et pointeurs)
- complexes par agrégation homogène (+ **typedef**)
- complexes par agrégation hybride : union/structure/classe + **typedef**
- mais le **typedef** n'est pas obligatoire en C++ pour l'agrégation hybride car la syntaxe est plus libre ...
- avec **typedef** on ne déclare que des types :

```
typedef union UnionType1
{
// ...
} UnionType2 ;
typedef struct StructType1
{
// ...
} StructType2 ;
typedef class ClassType1
{
// ...
} ClassType2 ;
```

- **type** en C : `struct StructType1` ou `StructType2`

Types en C++

- natifs au langage : scalaires (et pointeurs)
- complexes par agrégation homogène (+ **typedef**)
- complexes par agrégation hybride : union/structure/classe + **typedef**
- mais le **typedef** n'est pas obligatoire en C++ pour l'agrégation hybride car la syntaxe est plus libre ...
- avec **typedef** on ne déclare que des types :

```
typedef union UnionType1
{
// ...
} UnionType2 ;
typedef struct StructType1
{
// ...
} StructType2 ;
typedef class ClassType1
{
// ...
} ClassType2 ;
```

- **type** en C : `struct StructType1` ou `StructType2`
- **type** en C++ : `StructType1` ou `StructType2`

Types en C++

- sans **typedef**, on déclare des types et des variables

```
union UnionType1
{
// ...
} Union1 , Union2 ;
struct StructType1
{
// ...
} Struct1 , Struct2 ;
class ClassType1
{
// ...
} Class1 , Class2
```

Types en C++

- sans **typedef**, on déclare des types et des variables

```
union UnionType1
{
// ...
} Union1 , Union2 ;
struct StructType1
{
// ...
} Struct1 , Struct2 ;
class ClassType1
{
// ...
} Class1 , Class2
```

- en C : **variable** `Struct1` de **type** `struct StructType1` ;

Types en C++

- sans **typedef**, on déclare des types et des variables

```
union UnionType1
{
// ...
} Union1 , Union2 ;
struct StructType1
{
// ...
} Struct1 , Struct2 ;
class ClassType1
{
// ...
} Class1 , Class2
```

- en C : **variable** `Struct1` de **type** `struct StructType1` ;
- en C++ : **variable** `Struct1` de **type** `StructType1` ;

Types en C++

- sans **typedef**, on déclare des types et des variables

```
union UnionType1
{
// ...
} Union1 , Union2 ;
struct StructType1
{
// ...
} Struct1 , Struct2 ;
class ClassType1
{
// ...
} Class1 , Class2
```

- en C : **variable** `Struct1` de **type** `struct StructType1` ;
- en C++ : **variable** `Struct1` de **type** `StructType1` ;
- ...mais les règles de bonne programmation nous imposent de déclarer séparément les types et les variables :

```
class ClassType1
{
// ...
};
...
ClassType1 Class1 , Class2;
```