Le préprocesseur

```
#include <fichier>
#include “fichier” // Dans le répertoire courant


#define SYMBOLE
#define SYMBOLE valeur


#ifdef SYMBOLE
(...)
#endif


#ifndef SYMBOLE
(...)
#endif
```

Modularité

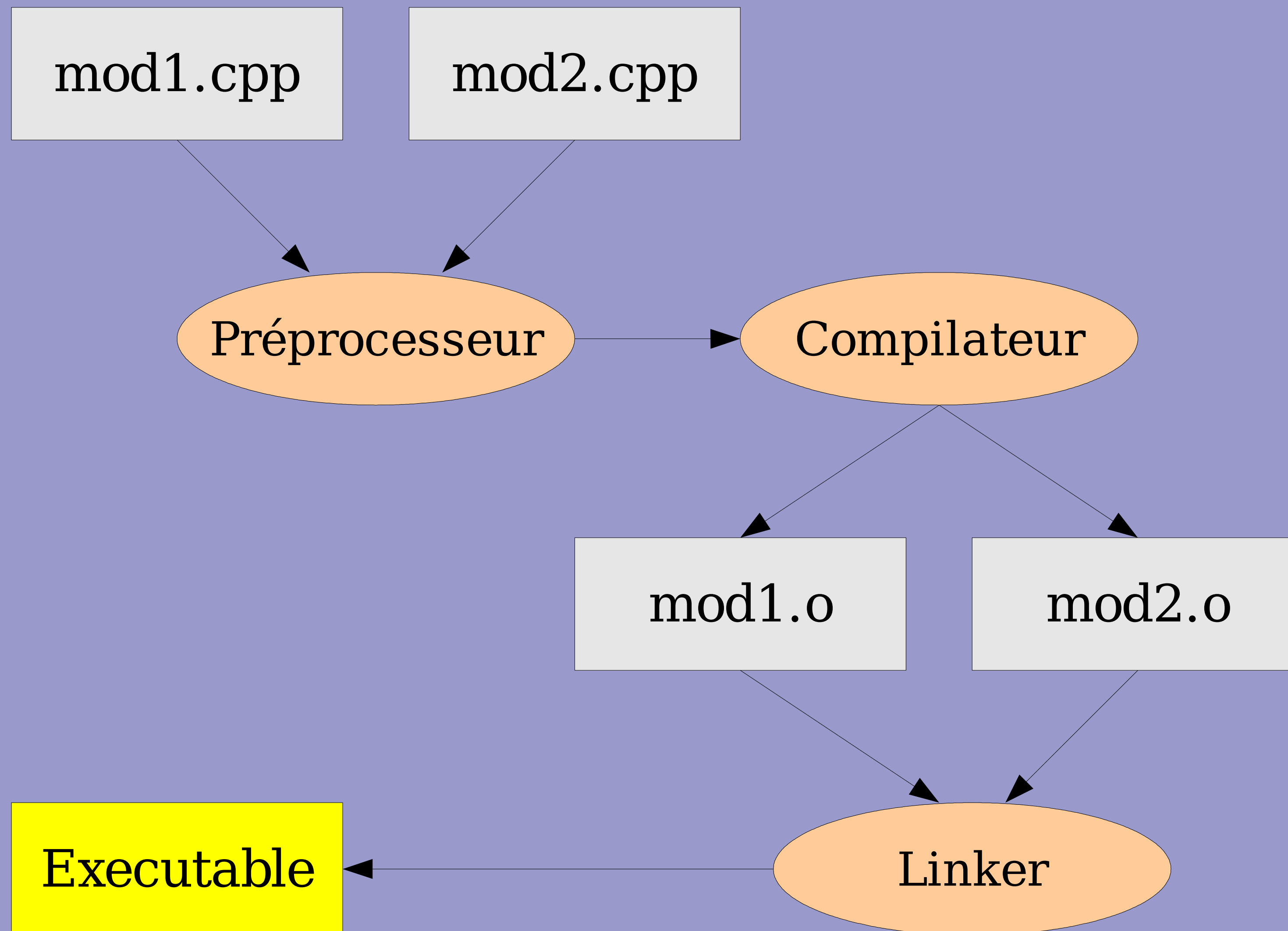
- Un logiciel complet (et complexe) ne peut et ne dois pas etre implémenté dans le meme fichier source.
- On va chercher à isoler des modules et les séparer en des fichiers distincts.
- L'objectif est de découpler au maximum les différents constituants d'un 'module'.
- On cherche à promouvoir la réutilisation de code et la séparation interface / implémentation.

Un module

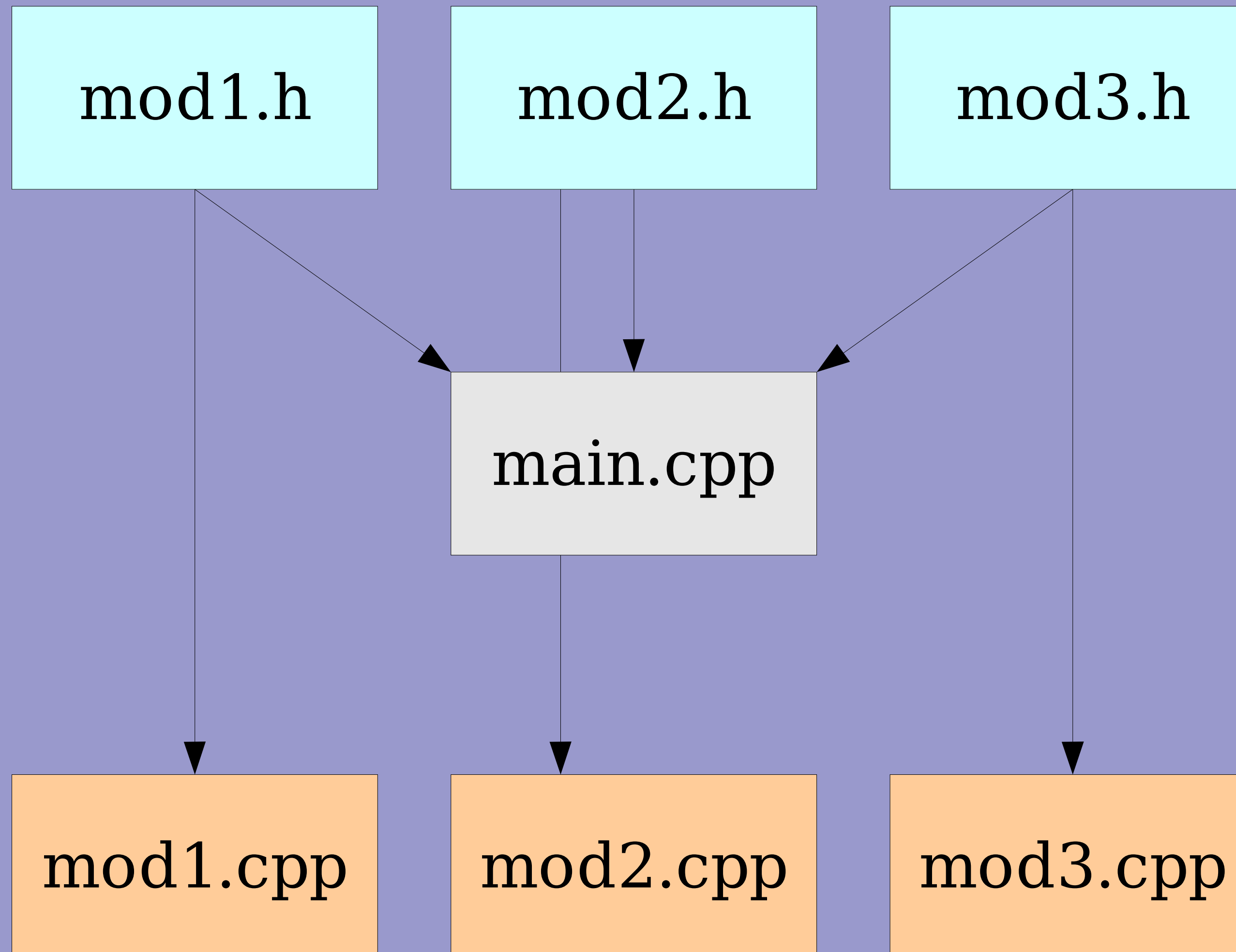
```
// module.h

#ifndef MODULE_H
#define MODULE_H

struct plop
{
   int x;
   int y;
}

void foo();
void bar();

#endif
```

```
// module.cpp

#include “module.h”

void foo()
{
   // ...
}

void bar()
{
   // ...
}
```

Processus de compilation



Compilation séparée



Utilisation de GCC

- Compilateur sous licence libre.
- Respect du standard C++.
- 'g++' : frontal au préprocesseur au compilateur et au linker.
- Options dans le cadre de ce cours :
  - -Wall : tous les warnings
  - -g : ajout de symboles de debug
  - -O, -O0, -O1, -02, -Os : optimiser
  - -o <nom_executable>

Utilisation de GCC (2)

```
// En bloc
g++ -o app -Wall -g main.cpp mod1.cpp mod2.cpp

-> produit app




// Séparé
g++ -Wall -g main.cpp
g++ -Wall -g mod1.cpp
g++ -Wall -g mod2.cpp
g++ -o app main.o mod1.o mod2.o

-> produit main.o mod1.o mod2.o et app
```

Détour hors-programme

- La programmation objet se fait à l'aide de classes.
- Une classe regroupe données et fonctions, appellées alors données membres et méthodes.
- Les données membres et les fonctions possèdent une visibilité (public, protected, private) qui régissent ce qui est invocable à l'extérieur de l'instance.

Exemple de classe

Personne
- nom : string - age : unsigned int
+ getAge() : unsigned int + getName() : string

```
cout << p.getAge() << p.getName() << endl;
```

Retour sur 'struct'

- Une structure défini en C++ une classe.
- On peut donc définir des fonctions dans les structures.
- Une structure est une classe ou la visibilité est toujours publique.

```
struct Personne {
   string nom;
   void afficher() { cout << nom << endl; }
};

Personne p = {“Toto”};
p.afficher();
```

Les opérateurs

- Un opérateur est une fonction. On va donc pouvoir faire de la surcharge d'opérateurs.
- Deux types d'opérateurs: unaires et binaires.
- Exceptions: '::'; '.', '*'.
- Par défaut, les opérateurs des types de base sont définis.

Les opérateurs (2)

```
// Opérateur unaire
<type> operator<signe>(const <type> &p)
{
   (...)
   return res;
}


// Opérateur binaire
<type> operator<signe>(const <type> &p1,
                       const <type> &p2)
{
   (...)
   return res;
}
```

Exemple sur les opérateurs

```
struct Point
{
   int x;
   int y;
};

Point operator+(const Point &p1, const Point &p2)
{
   Point p;

   p.x = p1.x + p2.x;
   p.y = p1.y + p2.y;

   return p;
}

// Usage: p1 = p2 + p3;
```

Exemple sur les opérateurs (2)

```
bool operator!=(const Point &p1, const Point &p2)
{
   return (p1.x != p2.x) && (p1.y != p2.y);
}

bool operator==(const Point &p1, const Point &p2)
{
   return !(p1 != p2);
}

// Usage
Point p1 = {0, 0};
Point p2 = {1, 1};
Point p3 = p1 + p2;
if (p2 == p3)
{
   (...)
}
```

Opérateur d'affectation

- Sur l'exemple précédent, il y aura un problème: comment affecter une instance de 'Point' ?
- Par défaut: copie brute en mémoire.
- Il faut définir l'opérateur '=' pour 'Point' (question de propreté).
- A déclarer dans la structure.
- L'opérateur '=' doit renvoyer une référence.

Opérateur d'affectation (2)

```
struct Point
{
  int x;
  int y;

  Point& operator=(const Point &p)
  {
    if (&p != this)
    {
      x = p.x;
      y = p.y;
    }
    return *this;
  }

};
```

Appels d'opérateurs surchargés

```
Point p1;
Point p2;
Point p3;

p1 = p2 + p3;

      <=>

p1.operator=(operator+(p2, p3));
```

Les flots

- Abstraction des opérations d'entrée sortie:
  - vers un fichier
  - depuis un fichier
  - vers les flux standards (cout, cerr, cin).
- On utilise des instances d'un objet d'entrée/sortie et les opérateurs de flot '<<' et '>>'.
- Nous allons nous limiter à des fonctions basiques sur les E/S.

Les flots (2)

- Il existe 2 classes de base:
  - istream (#include <istream>)
  - ostream (#include <ostream>).
- Pour les fichiers: #include <fstream>:
  - ifstream
  - ofstream.
- On les utilise comme cout, cerr et cin.
- A vrai dire, cout, cerr et cin sont des instances de sous-classes spéciales de istream et ostream.

Les opérateurs de flots

- Les opérateurs '<<' et '>>' peuvent etre redéfinis pour des types définis par l'utilisateur. Ils sont piégeux.

```
ostream& operator<<(ostream &os, const Point &p)
{
   os << p.x << p.y;
   return os;
}

istream& operator>>(istream &is, Point &p)
{
   is >> p.x >> p.y;
   return is;
}
```

Méthodes des flots

- Pour tous les types de flots, 'close()' provoque la fermeture de celui-ci.
- Pour les flux de sortie, il existe 'flush()' qui provoque l'envoi des données mises en tampon.
- Pour les flux d'entrée, 'eof()' indique si l'on est parvenu à la fin du flux.

Manipulateurs de flots

- #include <iomanip>, #include <ios>
- Ils s'insèrent dans un flot.
- Quelques-un:
  - skipws / noskipws
  - uppercase / nouppercase
  - oct, hex, fixed, scientific
  - left / right
  - ... endl .

Exemple sur les flots

```
#include <iostream>
#include <fstream>
#include <iomanip>

int main()
{
   ofstream f(“toto.txt”);
   f << uppercase << “hello world” << endl;
   f.close();
   return 0;
}
```

Les exceptions

- Méthode plus élégante pour gérer les erreurs que le if / then / else.
- Idée: déclarer un type d'exception pour chaque type d'erreur que l'on veut gérer.
- Une exception est lancée en cas d'erreur et se propage jusqu'à etre interceptée.
- Si elle n'est jamais interceptée, alors le programme s'arrete sur un échec.

Les exceptions (2)

- Du code succeptible de lancer une exception est placé dans un bloc 'try'.
- Ensuite, on place des blocs 'catch'. La forme générale est donc:

  try / catch / ... / catch .

- Chaque bloc 'catch' se destine à un type d'exception particulier:
  - catch (TypeException &err) { }
  - catch (...) { }.

Exemple sur les exceptions

```
// Attention, cet exemple est totalement stupide ...

#include <iostream>
using namespace std;

struct CMal { }; // On peut aussi y mettre des choses !

int main()
{
    try
    {
        int i = 1;
        if (i != 1) throw CMal();
        (...)
    }
    catch (CMal &err)
    {
        cerr << “C'est _mal_ !” << endl;
    }
    return 0;
}
```

Les exceptions (3)

- Les exceptions sont un moyen efficace de gérer les erreurs.
- Cependant, il ne faut pas non plus en abuser !
- En effet, le lancement d'une exception implique l'allocation d'une instance de l'exception.
- Autre facteur: les sauts induits par la propagation de l'exception.

Exercices :-)

- Reprendre la structure Personne.
- Lui ajouter une méthode d'affichage, 'afficher()'.
- Redéfinir les opérateurs suivants:
  - '<<' [penser à l'opération inverse ...]
  - '=', '==', '!=', '>>'.
- Faire une fonction qui enregistre un tableau de personnes dans un fichier dont le nom et la taille sont passés en paramètres.
- Faire une fonction qui lit un fichier de personnes et affiche une à une celles-ci à l'écran. [eof() est votre ami]

Exercies (suite)

- Une exception standard du C++ est levée quand le fichier que l'on veut lire n'existe pas. Interceptez cet erreur (catch(...)) et demandez à l'utilisateur de saisir à nouveau le nom du fichier dans ce cas.
- Proposez une structure pour stocker un vecteur de taille variable. Proposez les opérateurs suivants, en tenant compte de la variabilité de la taille du vecteur:
  - '+', '-' (lancer une exception si les tailles sont incompatibles)
  - '==', '!=', '<<', '>>'.