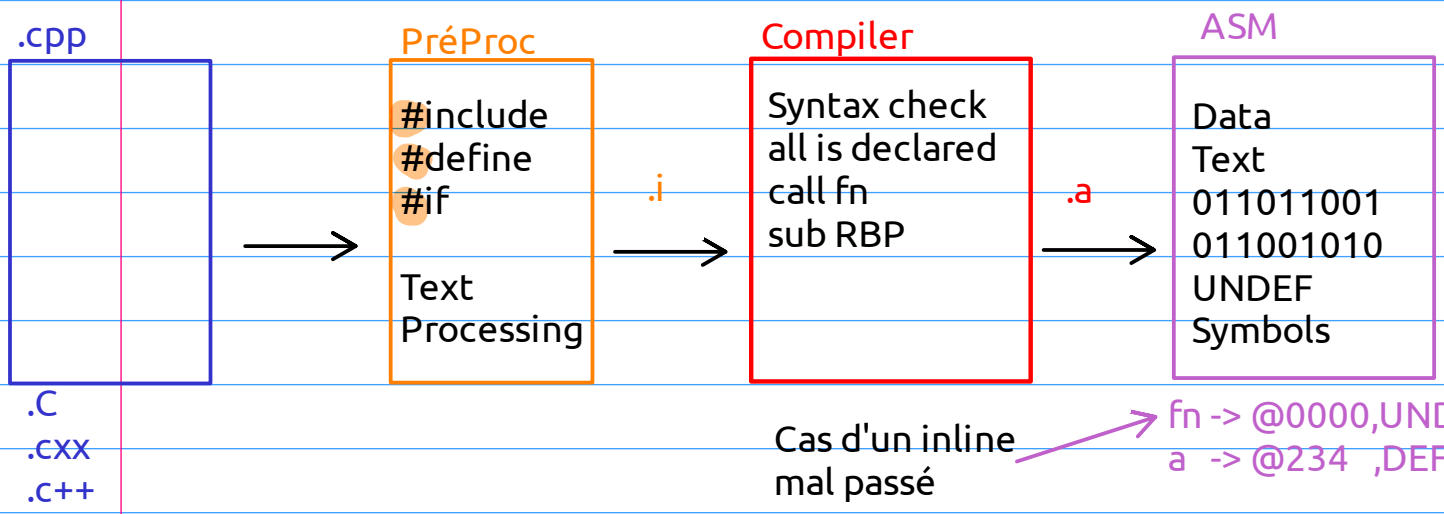
Compilations steps :

TU : Tanslations Unit



.o

Linker

Relocation
Appeler avec tout
les .o du project

Include Guards

```
#ifndef DOG
#define DOG

#endif
```

```
#pragma once
```

Quand on include plusieur fois un même .hpp (fichier qui contiens les déclaration et non les définitions de fonction), il est important d'utiliser un Include Guards pour évité que le compilateur essaie de redéfinir des fonctions déjà définie.

Cette include guards est cool mais pas safe si on utilise des lien symbolique dans nos projet donc pas legacy.

ODR (One Decarations Rule) & inline

Une seule définition de variable, fonction, type de classe, type d'énumération, concept ou modèle est autorisée dans une unité de traduction (TU) (certaines d'entre elles peuvent avoir plusieurs déclarations, mais une seule définition est autorisée).

```
inline int fn() { return 1; }
```

inline ce trouve dans les header, il permet de déclaré et définir une fonciton plusieur fois. Utile si la fonction en question est triviale,ainsi le compilateur peux la remplacer directement par du code et évité un call.

ATTENTION : toutes les définitions de la fonction doivent être identique !

Lier une Lib

Ici on veut lier animal à dog,
donc on include le header de animal
au header de dog.

animal.hpp > ...

```cpp
#pragma once
#include <fmt/format.h>

class animal {
  int age;

public:
  virtual void eat() = 0;

  int getAge() { return age; }

  void drink() { fmt::print("animal drinking\n"); }

  virtual void shout() { fmt::print("animal shouting\n"); }
};
```

dog.hpp > DOG

```cpp
#ifndef DOG
#define DOG
#include "animal.hpp"
#include <fmt/format.h>

class dog : public animal {
public:
  void eat() override;
  void shout() override;
  void drink() { fmt::print("dog drinking\n"); }
};
#endif
```

Puis on lie le header de dog au corp.

dog.cpp > shout()

```cpp
#include "dog.hpp"
#include <fmt/format.h>

void dog::eat() { fmt::print("dog eating\n"); }

void dog::shout() { fmt::print("dog shouting\n"); }
```

Attention la ça se passe bien
parce que tout les fichiers
sont au même endroit.
Dans le cas contraires il font mettre
le chemin relatif.

Basic on pointeur/reference

Valeur simple

Value : 1
Type : int
Size : size(int)
@ : 0001

Pointeur

Value : @1 = 0001
Type : intpointeur
Size : size(intpointeur)
@ : 0821

tp -> v
(*tp).v

Les références sont plus un alias d'une variable qu'autre chose. On peut modifier le contenue pointer par une référence; mais on ne peux pas modifier le contenue de la référence.

```cpp
int t = 0;

int *tp = &t;
*tp = 2;

int &tr = t;
tr = 2;
```

Valeur de base

Pointeur, remarque que l'on doit faire le déréférencement

Référence, on n'as pas besoin de faire de déréférencement. Et on ne peux pas modifier le fait que tr pointe vers t.

Unary operator

&var : prend l'@
*var : déréférence

Binary operator

var & var : binary and
var * var : multiplication

Declarations

Type& name : declaration de référence
Type* name : declaration de pointeur

---

readelf

Commande qui prent de lire des fichier elf,(Executable and Linkable Format).)
ELF est un format de fichier binaire utilisé pour l'enregistrement de code compilé (objets, exécutables, bibliothèques de fonctions).

Héritage de classe

Sert pour définir le comportement d'un object proche d'un autre sans réécrire du code. Ou lier un object avec un autre permetant le polymorphisme.

Lorsque qu'une classe membre hérité qu'une classe mère, la classe membre gagne les attributs de la classe mère ainsi que ces fonction membre.

```
class animal {
  int age;

public:
  virtual void eat() = 0;

  int getAge() { return age; }

  void drink() { fmt::print("animal drinking\n"); }

  virtual void shout() { fmt::print("animal shouting\n"); }
};
```

```
class dog : public animal {
public:
  void eat() override;
  void shout() override;
  void drink() { fmt::print("dog drinking\n"); }
};
```

Si on déclare un object du type DOG, alors il gagne comme attribut age et les fonctions membre getAge, Drink, eat et shout. On peut cependant voir que certain fontion membre semble être redéclaré avec le mot clef override pour dog et virtual pour animal.

Virtual

Virtual, ce mot est pas choisi au hazard. En faite chaque classe on une variable membre caché appeler la virtual table. Cette virtual table permet simplement de redéfinir les fonctions noté virtual dans les classes membres le souhaitant.

V pure

ATTENTION : si une fonction membre est purument virtual, alors il est obligatoire de rédifinir sont comportement puisqu'elle n'en à pas, la classe qui contient au moins une fonction membre pure virtual devient abtract et n'est pas instanciable.

```
virtual void eat() = 0;
```

Exemple de fonction pure virtual, rendant animal comme une classe abtraite.

```
vtable for dog:
        .quad   0
        .quad   typeinfo for dog
        .quad   dog::eat()
        .quad   dog::shout()
vtable for animal:
        .quad   0
        .quad   typeinfo for animal
        .quad   __cxa_pure_virtual
        .quad   animal::shout()
```

Vtable de dog

Vtable de animal

pure virtual

Virtual call

En gros, c'est un appel résolue par le typage dynamique d'un object. Les appels à des fonction virtuelle.

Le polymorphisme dynamique

```
animal &a = dog1;
a.eat();
a.drink();
a.shout();
```

```
dog eating
animal drinking
dog shouting
```

Direct

On vas recherche des fonction virtual dans dog et les fonction membre dans animal.

ATTENTION : au destructor (dtor) dans le cas d'un polymrphisme. Example

```
class Parent
{
public:
    Widget m_data1;
}

class Child : public Parent
{
public:
    Widget m_data2;
}

int main()
{
    Parent *ptr = new Child;
    delete ptr; // <-- clears-up Parent but not Child
}
```

Pour éviter ça, il faut mettre le destructeur en virtual lui aussi.