GLO-1901 Introduction à la programmation avec Python

Introduction au langage C/C++

Marc Parizeau

Plan

- Bref historique
- Langage C/C++
  - ✓ typage des données
  - ✓ énoncés de procédure
  - ✓ appels de fonction
  - ✓ classes
- Processus de développement
  - ✓ fichiers d'en-tête
  - ✓ compilation
  - ✓ édition des liens

Historique

- **Origines**
  - ✓ FORTRAN (54)
  - ✓ LISP (58)
  - ✓ COBOL (59)
  - ✓ APL (62)
  - ✓ Simula (62)
  - ✓ BASIC (64)
- **Grands paradigms**
  - ✓ Pascal (70)
  - ✓ Forth (70)
  - ✓ C (73)
  - ✓ Smalltalk (72)
  - ✓ Prolog (72)
- **Consolida-tion**
  - ✓ C++ (80)
  - ✓ Ada (83)
  - ✓ Eiffel (85)
  - ✓ Erlang (86)
  - ✓ Perl (87)
  - ✓ Tcl (88)
- **Age d'Internet**
  - ✓ Python (91)
  - ✓ Ruby (93)
  - ✓ Java (95)
  - ✓ PHP (95)
  - ✓ XML (97)
  - ✓ Visual Basic
  - ✓ *etc.*

Langage *C* (1973)

Ken Thompson & Dennis Ritchie devant un pdp-11 (1972)

- Inventé par Dennis Ritchie
- Basé sur le langage B
- A permis de réécrire Unix pour pouvoir le porter facilement sur plusieurs plateformes
- Standard ANSI en 1986

Langage compilé *vs* interprété

- Langage *C*
  - ✓ langage compilé
  - ✓ les types des variables sont fixés au moment de la compilation (ne peuvent pas changer durant l'exécution)
  - ✓ langage procédural, assez proche du langage natif de la machine
- Langage *Python*
  - ✓ langage interprété
  - ✓ les types des variables sont dynamiques (peuvent changer durant l'exécution)
  - ✓ langage orienté objet, de beaucoup plus haut niveau

Typage statique

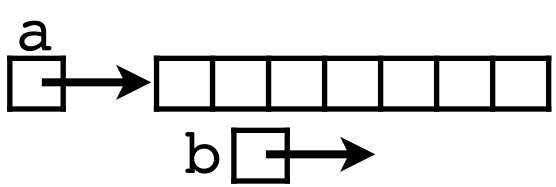
- Une variable doit toujours être déclarée avant d'être utilisée
  - ✓ la déclaration spécifie le type de la variable
  - ✓ y compris les arguments de fonction
  - ✓ et ce qui est retourné par une fonction
- Par exemple:

```
int a;
float b;
double fct(double, double);
```

Types de base

- Nombres
  - ✓ ***int*** = entier
  - ✓ ***long*** = entier long
  - ✓ ***float*** = virgule flottante simple précision
  - ✓ ***double*** = virgule flottante double précision
- Caractères
  - ✓ ***char*** = caractère codé en ASCII
- Tableau = pointeur = adresse
  - ✓ par exemple:

```
int a[7]; /* tableau de 7 entiers */
float *b; /* pointeur sur un float */
```



Nombres

- Sur les ordinateurs modernes, un ***int*** correspond habituellement à un bloc de 32 bits en mémoire
  - ✓ permet d'encoder des nombres entiers dans l'intervalle -2^{31} à $+2^{31}-1$
  - ✓ par exemple:

```
int a = 17;
```

- Et un ***long*** à un bloc de 64 bits
- De même, un ***float*** correspond en général à un bloc de 32 bits, mais avec un encodage différent de celui des ***int***
  - ✓ par exemple:

```
float x = 3.1416;
```

- Et un ***double*** à un bloc de 64 bits

Caractères

- Un char correspond habituellement à un bloc de 8 bits dont la valeur entière varie de -128 à +127
  - ✓ le code ASCII associe un caractère à chacune des valeurs entieres de 0 à 127
  - ✓ par exemple, le «A» vaut 65
- La syntaxe pour spécifier un caractère est:

```
char c = 'A';
```

Tableau

- Un tableau est un bloc de mémoire suffisamment grand pour contenir un nombre fixe de valeurs d'un certain type
  - ✓ par exemple:

    ```
    int tab[10];
    ```

  - ✓ cet énoncé alloue en mémoire un bloc suffisamment grand pour contenir 10 entiers
  - ✓ la variable *tab* est un pointeur qui contient l'adresse du bloc en mémoire
  - ✓ les variables pointeurs font la force du langage *C*, mais engendrent aussi beaucoup de *complexité*
  - ✓ contrairement au *C*, le Python ne permet pas de manipuler les pointeurs

- Pour accéder à un élément d'un tableau, on utilise simplement l'opérateur crochet:

```
int tab[10];          // création du tableau
tab[0] = 1;
tab[1] = 17;
int x = 39;
tab[2] = x;
int i = 3;
tab[i] = x*x;
tab[i+1] = tab[i] + 25;
```

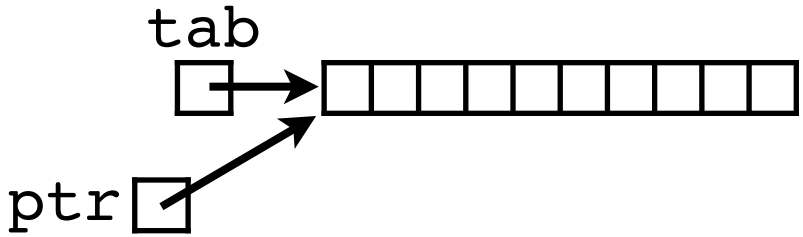
- Attention: il n'y a aucun vérification de la validité des indices d'un tableau en *C*

```
i = 250;
tab[i] = 97;      // erreur!
```

✓ cet énoncé risque fort de corrompre la mémoire!

- En mettant une étoile devant la déclaration d'une variable, on spécifie que cette variable est un pointeur
  - ✓ par exemple:

```
int *ptr = tab;
```



  - ✓ dans ce cas, seule la variable pointeur est créée; aucun espace n'est alloué pour contenir le ou les éléments du tableau qui seront manipulés via ce pointeur
- Une chaîne de caractères, n'est rien d'autre qu'un tableau de caractères

```
char *bonjour = “Bonjour le monde!”;
```

Noms de variables (identifieurs)

- Similaire à Python
- Débute par une lettre et contient des lettres, des chiffres ou des «underscore» (rien d'autre)

  `i``, ` **`x`** `et` **`mon_identifieur`** `sont ok`
  `12x` `et` **`mon-identifieur`** `sont invalides`

- Certains identifieurs sont des mots réservés du langage:

  `int``, ` **`if`**`, ` **`while`**`, ` **`double`**`, `*etc.*

Énoncés de procédure

- Tout comme en Python, on peut définir des énoncés:
  - ✓ séquentiels (*expressions* et appel de *fonction*)
  - ✓ conditionnels (`if` / `else if` / `else` ou `switch` / `case`)
  - ✓ répétitifs (`while` ou `for`)
- La syntaxe est cependant légèrement différente

Énoncés séquentiels

- On forme des énoncés séquentiels comme en Python sauf que
  - ✓ un énoncé se termine toujours par un **point-virgule**
  - ✓ l'indentation ne compte pas, sauf pour la lisibilité du code
- Par exemple:

```
int a = 2;
double x = 4.5, y;
y = a*x;
```

Expression

- La syntaxe des expressions en *C* est similaire à celle du Python
  - ✓ on notera cependant que certains opérateurs diffèrent
  - ✓ par exemple, il n'y a pas d'opérateur d'exponentiation en *C* (d'équivalent à l'opérateur ** en Python)
  - ✓ une opération arithmétique avec deux entiers donne toujours un résultat entier, y compris la division (contrairement à Python)
  - ✓ par contre, une opération impliquant un entier et un nombre à virgule flottante donnera toujours un nombre à virgule flottante (comme en Python)

Bloc d'énoncés

- Un bloc d'énoncés est toujours délimité par une paire d'accolades («{» et «}»)
- Toutes les variables définies dans un bloc sont **locales** à ce bloc
- On peut créer une hiérarchie de blocs:

```
{
  int a = 1;
  {
     int b = 2;
     int c = a+b;
  }
  int b = 12;
  c = b-a;  // erreur, la variable c n'existe plus!
}
```

- L'indentation du code est ignorée par le compilateur, mais améliore sa lisibilité

Énoncés conditionnels

- Il y a deux types d'énoncés conditionnels:
  - ✓ ***if*** / ***else if*** / ***else***
  - ✓ ***switch*** / ***case*** / ***default***
- Le premier est similaire à celui du Python
- Le second n'existe pas en Python
  - ✓ on peut cependant réaliser quelque chose d'équivalent en utilisant un dictionnaire

if / else if / else

- La syntaxe du ***if*** est la suivante:

```
if(expression1) {
    <bloc d'énoncés 1>
} else if(expression2) {
    <bloc d'énoncés 2>
} ... {
    ...
} else {
    <bloc d'énoncés n>
}
```

✓ si l'expression 1 est vrai, seul le bloc 1 sera exécuté; sinon si l'expression 2 est vrai, seul le bloc 2 sera exécuté; autrement, si aucune expression n'est vraie, alors seul le bloc *n* sera exécuté

- Par exemple:

```
int valeur;

/* obtenir la valeur à traiter */

if(valeur == 1) {
   /* traiter le cas de la valeur 1 */
} else if(valeur == 2) {
   /* traiter le cas de la valeur 2 */
} else {
   /* traiter tous les autres cas */
}
```

Énoncés répétitifs

- Il y a deux types d'énoncés répétitifs:
  - ✓ la boucle **`for`**
  - ✓ la boucle **`while`**
- Le `for` est différent de celui du Python
- Par contre, le **`while`** est très similaire
- Aucun de ces énoncés ne possède la clause `else` du Python

Boucle *for*

- La syntaxe du *for* est la suivante:

```
for(<init>; <condition>; <iter>) {
  /* bloc d'énoncés */
  if(<condition>) break;
}
```

  - ✓ <*init*> est un énoncé quelconque qui sera exécuté avant la première itération de la boucle pour initialiser celle-ci
  - ✓ <*condition*> est une expression booléene qui, à chaque itération, détermine si l'on continue ou pas l'exécution de la boucle
  - ✓ <*iter*> est un énoncé quelconque qui sera exécutée à la fin de chaque itération (juste avant de tester la condition)
  - ✓ les énoncés *break* et *continue* sont disponibles en *C* comme en Python

- Par exemple, pour calculer la somme des *n* premier entiers positifs:

```
int somme=0, i;
for(i = 1; i <= n; i += 1) {
    somme += i;
}
```

- L'équivalent en Python:

```
somme = 0
for i in range(1,n+1):
    somme += i
```

Boucle *while*

- La boucle ***while*** est similaire à celle du Python, sauf qu'elle ne supporte pas l'énoncé ***else***
- Sa syntaxe est la suivante:

```
/* initialisation de la boucle */
while(<condition1>) {
   /* traiter l'itération courante de
      la boucle */
   if(<condition2>) break;
   /* agir sur une variable pour éventuellement
      modifier la valeur de <condition1> ou
      <condition2> */
}
```

- Par exemple, pour reproduire l'exemple précédent qui consiste à calculer la somme des *n* premiers entiers positifs:

```
int somme = 0;
int i = 1;
while(i <= n) {
    somme += i;
    i += 1;          // affecte la condition
}
```

Définition d'une fonction

- La syntaxe est la suivante:

```
<type> nom_de_fonction(<type> arg1, <type> arg2, ...)
{
  <bloc d'énoncés>
  return <valeur>;
}
```

- Par exemple, pour calculer la somme des *n* premiers entiers:

```
int somme(int n)
{
  int rep = 0, i;
  for(i=1; i<=n; i+=1) {
     rep += i;
  }
  return rep;
}
```

Déclaration d'une fonction

- Avant de pouvoir appeler une fonction, il faut soit l'avoir défini, soit l'avoir déclaré
- La définition spécifie à la fois l'interface de la fonction et son implantation
- La déclaration ne spécifie que l'interface
- Par exemple, l'énoncé suivant se contente de déclarer la fonction *somme*:

```
int somme(int);
```

✓ celle-ci reçoit un seul argument, un entier, et retourne son résultat également sous la forme d'un entier

Passage d'argument

- Il y a deux façons de passer un argument à une fonction:
  - ✓ par recopie
  - ✓ par adresse
- Le passage par recopie signifie que la fonction reçoit une copie de l'argument
  - ✓ la fonction appelée ne pourra donc pas modifier la valeur dans la fonction appelante
- Le passage par adresse, signifie que la fonction reçoit l'adresse de l'argument
  - ✓ la fonction appelée pourra utiliser ce pointeur non seulement pour lire la valeur de l'argument dans la fonction appelante, mais elle pourra également modifier cette valeur

Mode par recopie

- Par exemple:

```
int fct(int a) {
   /* la variable a est locale à la fonction */
   /* elle contient une copie de la valeur   */
   /* de l'argument au moment de l'appel     */
}

int b = 25, x;
x = fct(b); // la fonction ne peut pas
            // modifier la variable b
```

Mode par adresse

```
int fct(int *a) {
  /* la variable a est locale à la fonction */
  /* mais c'est une variable pointeur;      */
  /* elle contient l'adresse d'une variable */
  /* entière                                */
  *a = 37; // cet énoncé modifie la variable b
}

int b = 25, x;
x = fct(&b); // la fonction peut modifier b
```

- L'opérateur & devant une variable permet de récupérer l'adresse en mémoire de celle-ci
- L'opérateur * devant une variable pointeur, signifie que l'on veut accéder non pas à la valeur de la variable, qui est une adresse, mais bien à la valeur qui se trouve en mémoire à cette adresse

Fonction *main*

- La fonction nommée *`main`* est toujours la première fonction appelée
  - ✓ lorsqu'on exécute un programme, le système d'exploitation commence par le charger en mémoire, puis il appelle la fonction *`main`*
  - ✓ cette fonction est donc obligatoire
- Elle reçoit deux arguments:

```
int main(int argc, char **argv);
```

  - ✓ le premier contient le nombre de d'arguments entrés sur la ligne de commande par l'utilisateur
  - ✓ le second est un tableau de chaînes de caractères contenant chacun de ces arguments

Fichiers d'en-tête

- En Python, il y a le mécanisme d'importation de module
- En *C*, il y a un mécanisme d'inclusion de fichiers d'en-tête
  - ✓ Ces fichiers contiennent essentiellement des déclarations de fonction, et parfois de variables
- Par exemple:

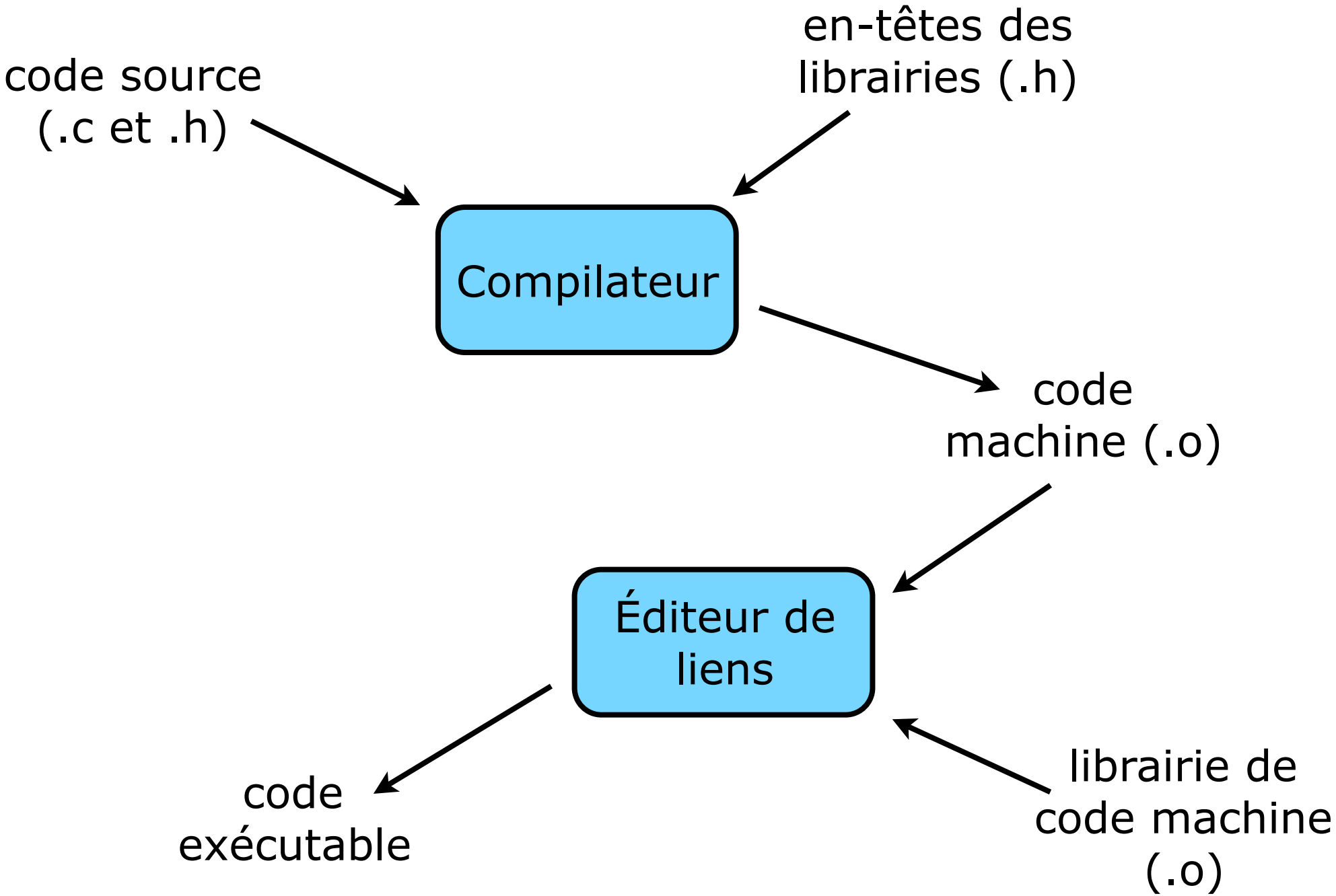
```c
#include <stdio.h>

int main()
{
  printf("Bonjour le monde!");
  return 0;
}
```

  - ✓ *printf* est une fonction de la librairie standard du *C*
  - ✓ elle est déclarée dans le fichier standard *stdio.h*

Processus de création d'un exécutable

- Compilation:
  - ✓ déclarations dans fichier avec extension `.h`
  - ✓ définitions dans fichier avec extension `.c`
  - ✓ un programme peut comporter de nombreux fichiers de code source
  - ✓ chaque fichier est compilé séparément
  - ✓ la compilation produit un fichier avec l'extension `.o`, pour chaque fichier `.c`
- Édition des liens:
  - ✓ cette phase regroupe tous les fichiers .o en un seul
  - ✓ elle effectue également les liens avec les librairies de fonctions requises

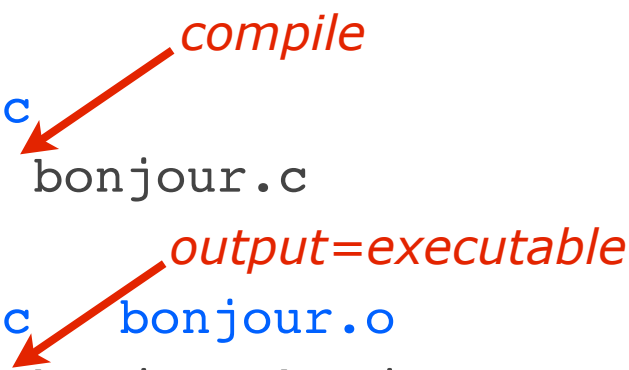

code source
(.c et .h)
en-têtes des
librairies (.h)
Compilateur
code
machine (.o)
Éditeur de
liens
code
exécutable
librairie de
code machine
(.o)

- Fichier *bonjour.c*:

```
#include <stdio.h>

int main()
{
  printf("Bonjour le monde!");
  return 0;
}
```

```
> ls
bonjour.c
> gcc -c bonjour.c
> ls
bonjour.c    bonjour.o
> gcc -o bonjour bonjour.o
> ls -l
-rwxr-xr-x  1 parizeau  staff  8712 30 Nov 14:07 bonjour
-rw-r--r--  1 parizeau  staff    82 30 Nov 14:04 bonjour.c
-rw-r--r--  1 parizeau  staff   736 30 Nov 14:06 bonjour.o
>
```



Langage Cython

- Le Cython est un langage qui ressemble au Python, mais qui permet de faire aisément le lien avec le langage *C*
  - ✓ le Cython ajoute au Python des déclarations pour les types de données du *C*
  - ✓ le Cython permet de compiler un *sous-ensemble* du langage Python en langage *C*
  - ✓ le Cython permet de faire appel à des librairies écrites dans le langage *C*
  - ✓ **les programmes Cython continuent à bénéficier de l'environnement dynamiques du Python**
  - ✓ les fichiers Cython portent l'extension `.pyx`

Langage *C++*

- Version orientée objet du langage *C*
- Permet de définir des classes
  - ✓ encapsulation
  - ✓ héritage
  - ✓ polymorphisme
  - ✓ *etc.*
- Demeure un langage compilé
- Demeure un langage statiquement typé

Définition d'une classe

C++

```
class Toto {
 public:
   Toto() {
     v1 = 1;
   }
   int square(int x) {
     return (x+v1)*(x+v1);
   }
 protected:
   int v1;
 private:
   static double v2;
};

double Toto::v2 = 3.1416;
```

Python

```
class Toto:
    v2 = 3.1416
    def __init__(self):
        self.v1 = 1
    def square(self, x):
        return (x+self.v1)**2
```

- ✓ en Python, c'est plus court!
- ✓ les membres d'une classe sont toujours publics

Héritage

C++

```
class A {...};
class B: public A {...};
class C: protected B {...};
class D: private C {...};
```

- ✓ avec l'héritage *public*, tout ce qui est public reste *public*
- ✓ avec l'héritage *protected*, tout ce qui est *public* devient *protected*
- ✓ avec l'héritage *private*, tout ce qui est *public* ou *protected* devient *private*

Python

```
class A: pass
class B(A): pass
class C(B): pass
class D(C): pass
```

- ✓ En Python, tout est toujours *public*

Membres

- Tous les membres d'une classe sont par défaut des membres des instances de cette classe
  - ✓ les données sont distinctes pour chaque objet de la classe
  - ✓ les méthodes doivent être appelées dans le contexte d'un objet de la classe
- Les membres déclarés `static` font cependant exception à la règle
  - ✓ les données `static` sont partagées par toutes les instances de la classe
  - ✓ les fonctions `static` sont appelées sans faire référence à un objet de la classe

C++

```
class Toto {
 public:
   Toto() {
     v1 = 1;
   }
   int square(int x) {
     return (x+v1)*(x+v1);
   }
 protected:
   int v1;
 private:
   static double v2;
   static double fct(int i) {...}
};

double Toto::v2 = 3.1416;
```

variable d'instance

variable de classe

Python

```
class Toto:
    v2 = 3.1416
    def __init__(self):
        self.v1 = 1
    def square(self, x):
        return (x+self.v1)**2
    def fct(): pass
```

variable de classe

variable d'instance

la fonction ne reçoit pas `self`*!*

Sur-définition des opérateurs

C++

```
class Toto {
 public:
   Toto() {...}
   Toto operator+(Toto x) {...}
```

- ✓ en C++, le self est implicite (sauf pour les fonctions *static*)
- ✓ au lieu d'être *self*, c'est *this* qui est un pointeur sur l'objet pour lequel une méthode de la classe est appelée

Python

```
class Toto:
    def __init__(self): pass
    def __add__(self, x): pass
```

- ✓ on peut définir sensiblement les mêmes opérateurs en Python et en C++

Méthodes virtuelles

- Permet une certaine forme de polymorphisme
- Python est plus général
  - ✓ tout y est polymorphique!
  - ✓ on peut tout redéfinir lors de l'exécution
- En C++, il faut spécifier les méthodes que l'on veut polymorphiques
  - ✓ on utilise le mot réservé *`virtual`*

Exceptions

C++

```
try {
  <bloc d'énoncés>
  throw MonException();
}
catch(MonException err) {
  <bloc d'énoncés>
}
catch(Exception err) {
  <bloc d'énoncés>
}
catch(...) {
  <bloc d'énoncés>
}
```

✓ il n'y a pas d'équivalent au *finally*

Python

```
try:
    <bloc d'énoncés>
    raise MonException()
except MonException as err:
    <bloc d'énoncés>
except Exception as err:
    <bloc d'énoncés>
else:
    <bloc d'énoncés>
finally:
    <bloc d'énoncés>
```

✓ le *raise* correspond au *throw*

✓ le *else* est similaire au *catch(...)*

Passage d'argument

- Par recopie (idem C)
- Par adresse (idem C)

```
void incremente(int *x) { *x += 1; }

int a = 10;
incremente(&a);
```

- Par référence (similaire Python)

```
void increment(int &x) { x += 1; }

int a = 10;
incremente(a);
```

Autres détails

- Entrées / sorties
- Librairie standard
  - ✓ fonctions de manipulation de chaînes de caractères
  - ✓ etc.
- Templates
  - ✓ mécanismes sophistiqués de macro-définitions
- Standard template library
  - ✓ structures de données et algorithmes standards
  - ✓ listes, dictionnaire et autres...
- *etc.*

Conclusion

- Tout développer en Python
- Problème de performance?
  - ✓ isoler le problème
  - ✓ optimiser avec Cython
  - ✓ réécrire en C ou C++ seulement les fonctions problématiques
- Avantages du Python
  - ✓ concision (1/10 à 1/5 des lignes en C/C++)
  - ✓ typage dynamique / langage interprété
  - ✓ programmation orientée objet
  - ✓ programmation fonctionnelle
  - ✓ programmation scientifique

- Défauts du python
  - ✓ langage interprété / lenteur relative d'exécution
- Zen du Python: import this
  - ✓ **Beautiful** is better than ugly.
  - ✓ **Explicit** is better than implicit.
  - ✓ **Simple** is better than complex.
  - ✓ Complex is better than complicated.
  - ✓ **Flat** is better than nested.
  - ✓ **Sparse** is better than dense.
  - ✓ **Readability** counts.
  - ✓ Special cases aren't special enough to break the rules.
  - ✓ Although practicality beats purity.
  - ✓ Errors should never pass silently.
  - ✓ Unless explicitly silenced.

- ✓ In the face of ambiguity, refuse the **temptation** to guess.
- ✓ There should be one-- and preferably only one --obvious way to do it.
- ✓ Although that way may not be obvious at first unless you're Dutch.
- ✓ **Now** is better than never.
- ✓ Although never is often better than *right* now.
- ✓ If the implementation is hard to explain, it's a bad idea.
- ✓ If the implementation is easy to explain, it may be a good idea.
- ✓ **Namespaces** are one honking great idea -- let's do more of those!

- Pour de plus amples détails sur le C++:
  - ✓ http://www.cplusplus.com/doc/tutorial/
  - ✓ http://www.learncpp.com/