Langage C / C++

ECP - P2006 - AlgoProg - Amphi5

Au menu aujourd'hui

- Utilisation des stream
- Conteneurs
- Itérateurs

Copie ligne à ligne (1)

```
#include <fstream>

std::ifstream entree("monfich.txt") ;
std::ofstream copie("macopie.txt") ;
if (! entree ||! copie)
  throw std::ios_base::failure("Accès
 impossible") ;
```

Copie ligne à ligne (2)

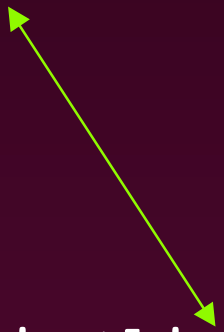
```
#include <string>

std::string ligne ;
std::getline(entree, ligne) ;
while (entree) {
  copie << ligne << "\n" ;
  std::getline(entree, ligne) ;
  }
```

Schéma itératif

```
// Sélectionner le premier élément
while (! fini) {
   // Traiter l'élément courant
   // Passer à l'élément suivant
   }
```

Décodage d'une chaîne

12h45



```
#include <sstream>

std::istringstream flig(ligne) ;
int heure, min ;
char sep ;
flig >> heure >> sep >> min ;
if (! flig)
  // traiter l'erreur
```

Stream standards

`cin`	entrée standard
`cout`	sortie standard (bufferisée)
`cerr`	sortie d'erreur (non bufferisée)
`clog`	sortie d'erreur (bufferisée)

Version `wchar_t` : `wcin`, `wcout`, `wcerr`, `wclog`

Manipulateurs : `flush`, `endl`

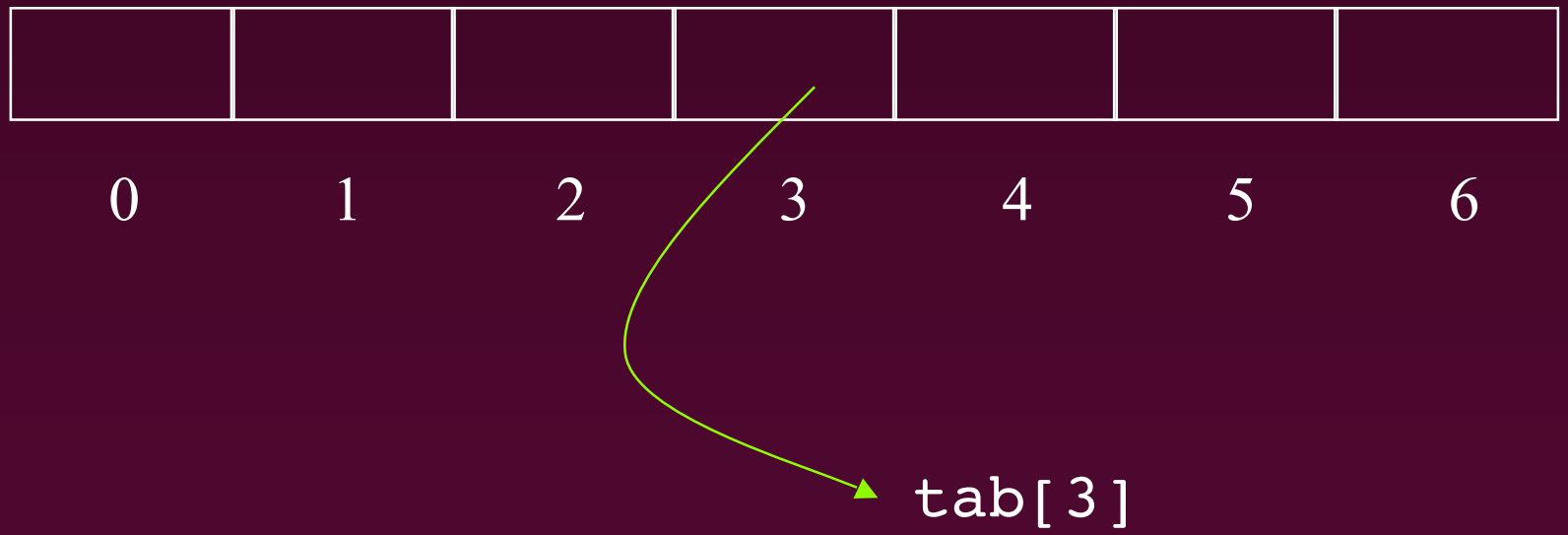
Notion de conteneur

- Objet pouvant contenir d'autres objets
- Noyau du langage C/C++ : tableaux
- Librairie standard :

vector	stack	map
list	queue	multimap
deque	priority_queue	set
		multiset

Notion de tableau

```
int tab[7] ;
```



Tableaux élémentaires

```
int tab[taille] ;
```

- Taille fixée à la création
- Seule opération possible, accès à un élément

```
x = tab[i] ; tab[j] = y ;
```

- Les indices sont entiers et commencent à ZÉRO
- Un tableau ne connaît pas sa taille

Tableaux évolués

```
std::vector<int> tab(taille) ;
```

- La taille peut être modifiée à tout moment

  ```
  tab.resize(nvle_taille)
  ```

- Accès à un élément identique

  ```
  x = tab[i] ; tab[j] = y ;
  ```

- Les indices sont entiers et commencent à ZÉRO
- Le tableau connait sa taille : `tab.size()`

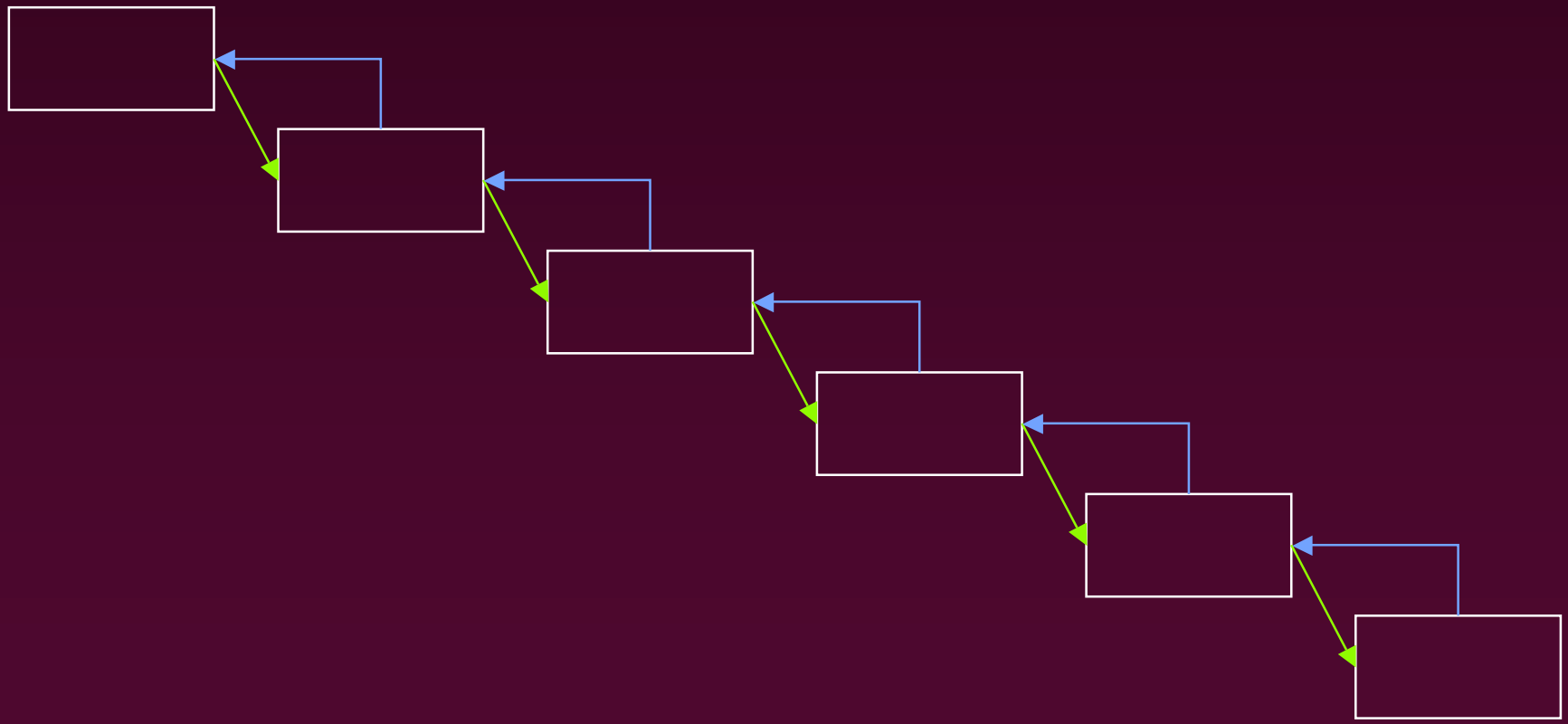
Tableaux évolués : les plus

- L'affectation est possible
- Les opérateurs standards de comparaison sont disponibles (à condition que le type des éléments les supporte)
- Accès protégé à un élément

  `tab.at(i) <=> tab[i]`
- Ajout d'un élément en fin

  `tab.push_back(x)`

Notion de liste



vector vs list

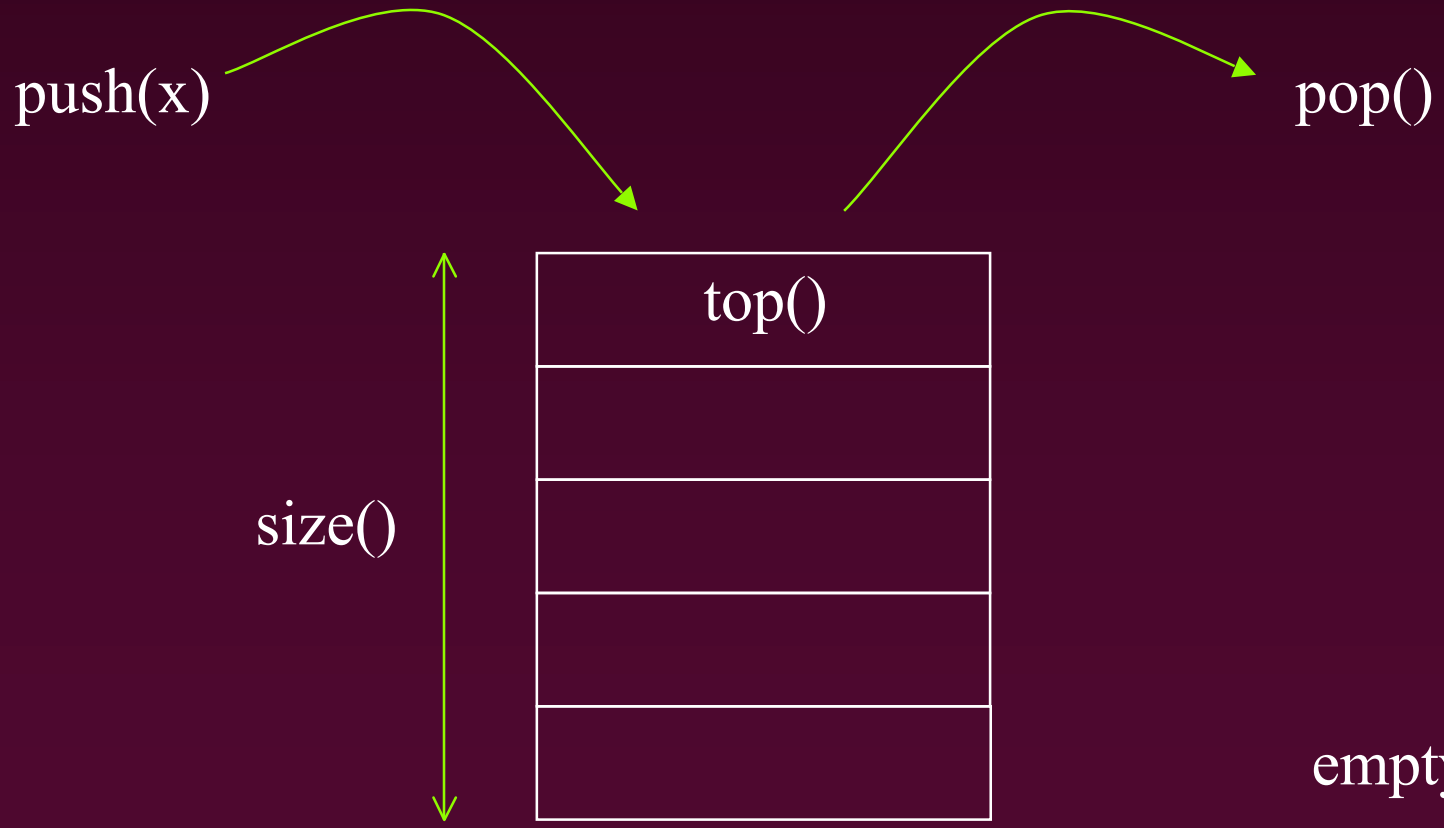
- Accès direct
- Ajout avec décalage
- Suppression avec décalage

Optimisé pour l'accès direct aux éléments

- Accès séquentiel
- Ajout direct
- Suppression directe n'importe où

Optimisé pour ajout et suppression d'éléments

Notion de pile



Utilisation d'une pile

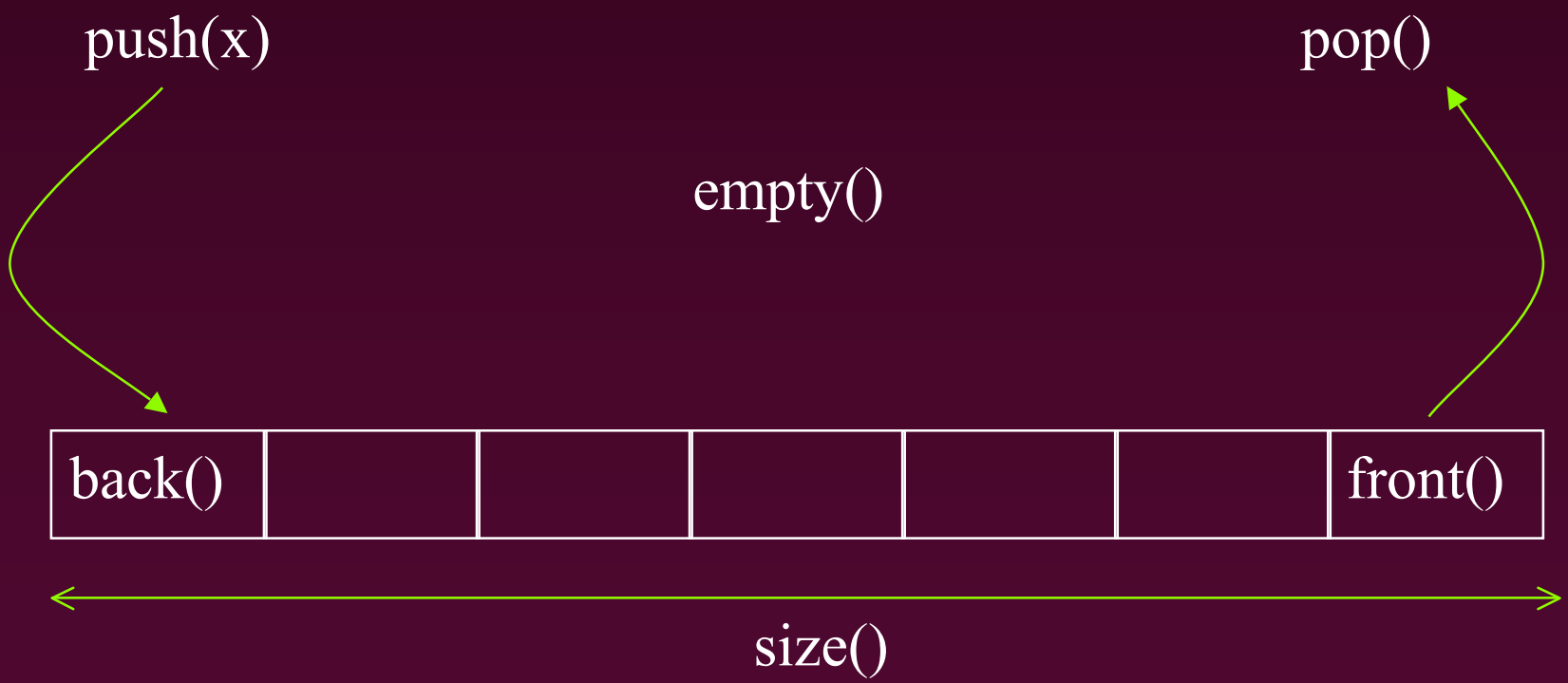
```
#include <stack>

std::stack<int> pile ;
pile.push(12) ; pile.push(45) ;
if (pile.size() != 2) // bizarre…
if (pile.top() != 45) // zarbi…
pile.pop() ;
if (pile.top() != 12) // planté…
```

Débordement d'une pile

```
std::stack<int> pile ;
pile.push(12) ;
pile.pop() ; pile.pop() ;
```

- Pas de contrôle d'underflow en standard
- Conséquences imprévisibles

Notion de file



Utilisation d'une file

```
#include <queue>

std::queue<int> file ;
file.push(12) ; file.push(45) ;
if (file.size() != 2) // bizarre…
if (file.front() != 12) // zarbi…
if (file.back() != 45) // hum…
file.pop() ;
if (file.front() != 45) // planté…
```

Débordement d'une file

```
std::queue<int> file ;
file.push(12) ;
file.pop() ; file.pop() ;
```

- Pas de contrôle d'underflow en standard
- Conséquences imprévisibles

Notion de tableau associatif

- Tableau dont les indices ne sont pas limités à des entiers (positifs), mais sont des objets quelconques
- L'indice prend le nom de clef
- Parfois appelé dictionnaire
- Très utile dans les programmes effectuant du traitement symbolique ou du traitement de texte

Utilisation d'un dictionnaire

```
#include <map>

std::map<std::string, int> dico ;
dico["douze"] = 12 ;
dico["treize"] = 13 ;
if (dico.find("onze") != dico.end())
    // surprise…
```

Ordre des éléments

- Le type utilisé pour les clefs doit disposer d'une fonction de comparaison
- Il est possible d'indiquer la fonction de comparaison à utiliser (au lieu de <) au moment de la création du dictionnaire
- On peut parcourir un dictionnaire dans l'ordre de ses clefs

Récapitulation des conteneurs

- Noyau du langage C/C++ : tableaux
- Librairie standard :

vector	stack	map
list	queue	multimap
deque	priority_queue	set
		multiset

Notion d'itérateur

Pour parcourir les éléments d'un conteneur, il est nécessaire de :

- pouvoir désigner un élément particulier
- passer à l'élément suivant
- connaître le premier élément
- savoir quand on est arrivé à la fin

Parcours d'un tableau élémentaire

```
int tab[10] ;

for (int i = 0 ; i != 10 ; i++)
  std::cout << tab[i] << "\n" ;
```

Parcours d'un tableau évolué

```
std::vector<int> tab(10) ;

for (int i = 0 ;
     i != tab.size() ;
     i++)
  std::cout << tab[i] << "\n" ;
```

Utilisation d'un itérateur

```
std::vector<int> tab(10) ;
typedef std::vector<int>::iterator PV ;

for (PV i = tab.begin() ;
      i != tab.end() ;
      i++)
  std::cout << *i << "\n" ;
```

Utilisation d'un itérateur

- Accès à l'élément

  ```
  *i
  ```

- Passage à l'élément suivant

  ```
  i++
  ```

- Désigner le premier élément

  ```
  i = tab.begin()
  ```

- Est-on à la fin ?

  ```
  i == tab.end()
  ```

Cas particulier des map

- L'itérateur doit permettre l'accès aussi bien à la clef qu'à l'élément lui-même

```
typedef std::map::<string,
  int>::iterator MI ;
for (MI i = dico.begin() ;
      i != dico.end() ;
      i++)
   std::cout << (*i).first << " : "
             << i->second << "\n" ;
```

Emploi d'itérateur

```
sort(tab.begin(), tab.end()) ;
. . .
i = std::find(tab.begin(), tab.end(),
                  123) ;
if (i == tab.end())
  // pas trouvé
else
 *i = 127 ;
```