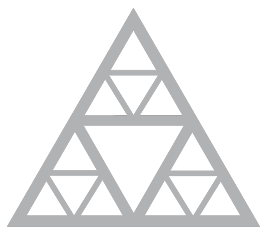
Introduction à la programmation en C++

Gestion de la mémoire

Nicolas Audebert

Vendredi 20 octobre 2017



Avant toute chose

Rendus de TP et des exercices

Les rendus se font sur **Educnet**.

1. Le code rendu **doit compiler**.
2. Le code rendu doit **être propre** (indentation, noms de variables clairs).
3. Le code rendu doit **être commenté** (réponses aux questions, fonctionnement du code).
4. Rassembler le code dans une seule archive (`.zip`, `.rar`, `.tar.gz`, etc.).

Plan de la séance - Gestion de la mémoire

Multiples fichiers sources

On peut désormais organiser un projet en plusieurs fichiers sources (`.cpp`) et fichiers d'en-tête (`.h`).

```cpp
// fichier1.cpp

// Signature de autre_fonction
void autre(int arg);

int ma_fonction(int var){
  ...
  autre(var); // OK
  ...
}
```

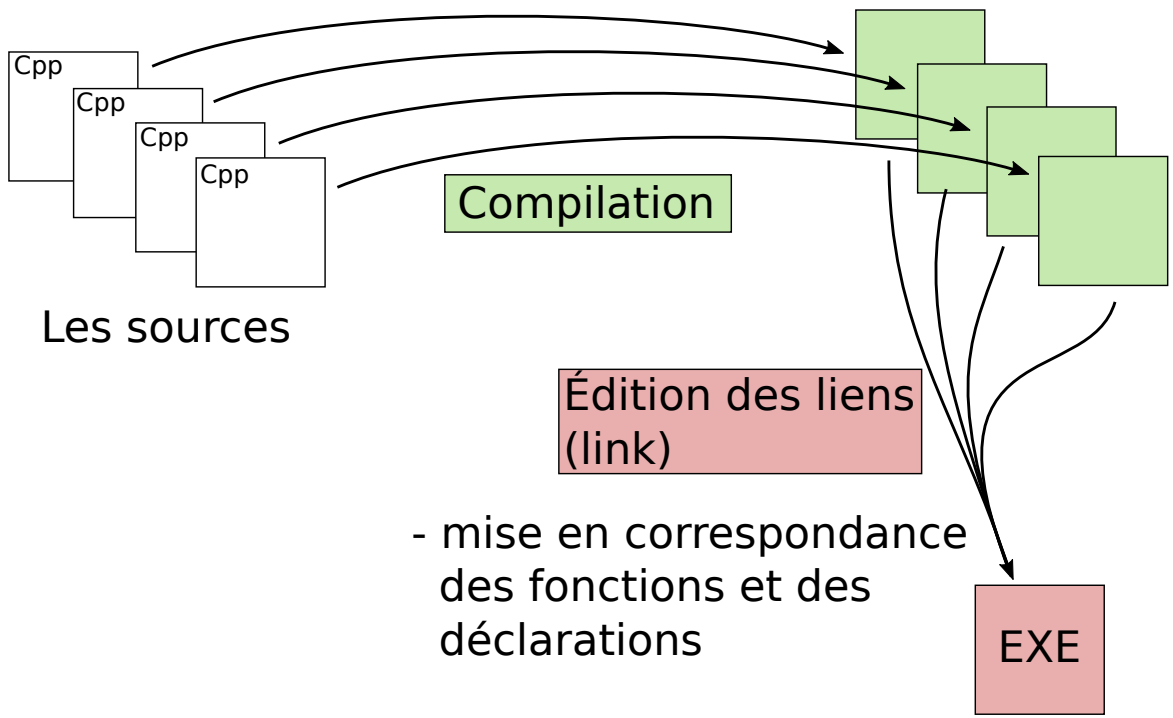
```cpp
// fichier2.cpp

void autre(int arg){
    ...
}
```

Mécanisme de compilation

La production d'un exécutable à partir du code source C++ se réalise en deux étapes :

1. La compilation (transforme le code en fichiers objets)
2. L'édition des liens (transformes les fichiers objets en exécutable)



Surcharge des opérateurs

```
// opérateur * pour deux vecteurs
double operator*(Point vA, Point vB){
    return vA.x*vB.x + vA.y*vB.y;
}

// opérateur * vecteur et réel
Point operator*(Vect vA, double alpha){
    Point v = {alpha*v.x, alpha*v.y};
    return v;
}

Point v1 = {1, 2}, v2 = {5, 5};

// produit scalaire
double s = v1*v2;

// multiplication par un réel
double m = 5.5;
Point v3 = v1 * m;
```

Plan de la séance - Gestion de la mémoire

Déroulé d'un programme C++

Un programme C++, une fois compilé, s'exécute toujours de la même façon :

Fonctionnement d'un programme C++

1. Initialisation des variables globales
2. Entrée dans la fonction `main()`
3. Exécution d'instructions et de diverses fonctions
4. Sortie de la fonction `main()`
5. Fin du programme

Exemple sur un cas réel - appel d'une fonction

Level	Function	File	Line
0	main	main.cpp	19

```
1  #include <iostream>
2
3  using namespace std;
4
5  bool verif(int av, int bv, int qv, int rv){
6      if(qv<0 || rv>=bv || av != bv*qv+rv)
7          return false;
8      return true;
9  }
10
11 int div(int ad, int bd, int& rd){
12     int quo = ad/bd;
13     rd = ad - bd*quo;
14     cout << verif(ad,bd,quo,rd)<< endl;
15     return quo;
16 }
17
18 int main(){
19     int a=20, b=3, r;
20     int q = div(a,b,r);
21     return 0;
22 }
```

On rentre progressivement dans les fonctions.

Exemple sur un cas réel - appel d'une fonction

Level	Function	File	Line
0	main	main.cpp	20

```
1  #include <iostream>
2
3  using namespace std;
4
5  bool verif(int av, int bv, int qv, int rv){
6      if(qv<0 || rv>=bv || av != bv*qv+rv)
7          return false;
8      return true;
9  }
10
11 int div(int ad, int bd, int& rd){
12     int quo = ad/bd;
13     rd = ad - bd*quo;
14     cout << verif(ad,bd,quo,rd)<< endl;
15     return quo;
16 }
17
18 int main(){
19     int a=20, b=3, r;
20     int q = div(a,b,r);
21     return 0;
22 }
```

On rentre progressivement dans les fonctions.

Exemple sur un cas réel - appel d'une fonction

Level	Function	File	Line
0	div	main.cpp	12
1	main	main.cpp	20

```
1  #include <iostream>
2
3  using namespace std;
4
5  bool verif(int av, int bv, int qv, int rv){
6      if(qv<0 || rv>=bv || av != bv*qv+rv)
7          return false;
8      return true;
9  }
10
11 int div(int ad, int bd, int& rd){
12     int quo = ad/bd;
13     rd = ad - bd*quo;
14     cout << verif(ad,bd,quo,rd)<< endl;
15     return quo;
16 }
17
18 int main(){
19     int a=20, b=3, r;
20     int q = div(a,b,r);
21     return 0;
22 }
```

On rentre progressivement dans les fonctions.

Chaque fois que l'on rentre dans une fonction on gagne un niveau.

Exemple sur un cas réel - appel d'une fonction

Level	Function	File	Line
0	div	main.cpp	13
1	main	main.cpp	20

```
1  #include <iostream>
2
3  using namespace std;
4
5  bool verif(int av, int bv, int qv, int rv){
6      if(qv<0 || rv>=bv || av != bv*qv+rv)
7          return false;
8      return true;
9  }
10
11 int div(int ad, int bd, int& rd){
12     int quo = ad/bd;
13     rd = ad - bd*quo;
14     cout << verif(ad,bd,quo,rd)<< endl;
15     return quo;
16 }
17
18 int main(){
19     int a=20, b=3, r;
20     int q = div(a,b,r);
21     return 0;
22 }
```

On rentre progressivement dans les fonctions.

Chaque fois que l'on rentre dans une fonction on gagne un niveau.

Exemple sur un cas réel - appel d'une fonction

Level	Function	File	Line
0	div	main.cpp	14
1	main	main.cpp	20

```
1  #include <iostream>
2
3  using namespace std;
4
5  bool verif(int av, int bv, int qv, int rv){
6      if(qv<0 || rv>=bv || av != bv*qv+rv)
7          return false;
8      return true;
9  }
10
11 int div(int ad, int bd, int& rd){
12     int quo = ad/bd;
13     rd = ad - bd*quo;
14     cout << verif(ad,bd,quo,rd)<< endl;
15     return quo;
16 }
17
18 int main(){
19     int a=20, b=3, r;
20     int q = div(a,b,r);
21     return 0;
22 }
```

On rentre progressivement dans les fonctions.

Chaque fois que l'on rentre dans une fonction on gagne un niveau.

Exemple sur un cas réel - appel d'une fonction

```
1  #include <iostream>
2
3  using namespace std;
4
5  bool verif(int av, int bv, int qv, int rv){
6      if(qv<0 || rv>=bv || av != bv*qv+rv)
7          return false;
8      return true;
9  }
10
11 int div(int ad, int bd, int& rd){
12     int quo = ad/bd;
13     rd = ad - bd*quo;
14     cout << verif(ad,bd,quo,rd)<< endl;
15     return quo;
16 }
17
18 int main(){
19     int a=20, b=3, r;
20     int q = div(a,b,r);
21     return 0;
22 }
```

Level	Function	File	Line
0	verif	main.cpp	6
1	div	main.cpp	14
2	main	main.cpp	20

On rentre progressivement dans les fonctions.

Chaque fois que l'on rentre dans une fonction on gagne un niveau.

Exemple sur un cas réel - appel d'une fonction

```
1  #include <iostream>
2
3  using namespace std;
4
5  bool verif(int av, int bv, int qv, int rv){
6      if(qv<0 || rv>=bv || av != bv*qv+rv)
7          return false;
8      return true;
9  }
10
11 int div(int ad, int bd, int& rd){
12     int quo = ad/bd;
13     rd = ad - bd*quo;
14     cout << verif(ad,bd,quo,rd)<< endl;
15     return quo;
16 }
17
18 int main(){
19     int a=20, b=3, r;
20     int q = div(a,b,r);
21     return 0;
22 }
```

Level	Function	File	Line
0	verif	main.cpp	8
1	div	main.cpp	14
2	main	main.cpp	20

On rentre progressivement dans les fonctions.

Chaque fois que l'on rentre dans une fonction on gagne un niveau.

Exemple sur un cas réel - appel d'une fonction

```
1  #include <iostream>
2
3  using namespace std;
4
5  bool verif(int av, int bv, int qv, int rv){
6      if(qv<0 || rv>=bv || av != bv*qv+rv)
7          return false;
8      return true;
9  }
10
11 int div(int ad, int bd, int& rd){
12     int quo = ad/bd;
13     rd = ad - bd*quo;
14     cout << verif(ad,bd,quo,rd)<< endl;
15     return quo;
16 }
17
18 int main(){
19     int a=20, b=3, r;
20     int q = div(a,b,r);
21     return 0;
22 }
```

Level	Function	File	Line
0	verif	main.cpp	9
1	div	main.cpp	14
2	main	main.cpp	20

On rentre progressivement dans les fonctions.

Chaque fois que l'on rentre dans une fonction on gagne un niveau.

Exemple sur un cas réel - appel d'une fonction

Level	Function	File	Line
0	div	main.cpp	15
1	main	main.cpp	20

```
1  #include <iostream>
2
3  using namespace std;
4
5  bool verif(int av, int bv, int qv, int rv){
6      if(qv<0 || rv>=bv || av != bv*qv+rv)
7          return false;
8      return true;
9  }
10
11 int div(int ad, int bd, int& rd){
12     int quo = ad/bd;
13     rd = ad - bd*quo;
14     cout << verif(ad,bd,quo,rd)<< endl;
15     return quo;
16 }
17
18 int main(){
19     int a=20, b=3, r;
20     int q = div(a,b,r);
21     return 0;
22 }
```

On rentre progressivement dans les fonctions.

Chaque fois que l'on rentre dans une fonction on gagne un niveau.

Chaque fois que l'on sort d'une fonction on perd un niveau.

Exemple sur un cas réel - appel d'une fonction

Level	Function	File	Line
0	div	main.cpp	16
1	main	main.cpp	20

```
1  #include <iostream>
2
3  using namespace std;
4
5  bool verif(int av, int bv, int qv, int rv){
6      if(qv<0 || rv>=bv || av != bv*qv+rv)
7          return false;
8      return true;
9  }
10
11 int div(int ad, int bd, int& rd){
12     int quo = ad/bd;
13     rd = ad - bd*quo;
14     cout << verif(ad,bd,quo,rd)<< endl;
15     return quo;
16 }
17
18 int main(){
19     int a=20, b=3, r;
20     int q = div(a,b,r);
21     return 0;
22 }
```

On rentre progressivement dans les fonctions.

Chaque fois que l'on rentre dans une fonction on gagne un niveau.

Chaque fois que l'on sort d'une fonction on perd un niveau.

Exemple sur un cas réel - appel d'une fonction

Level	Function	File	Line
0	main	main.cpp	21

```
1  #include <iostream>
2
3  using namespace std;
4
5  bool verif(int av, int bv, int qv, int rv){
6      if(qv<0 || rv>=bv || av != bv*qv+rv)
7          return false;
8      return true;
9  }
10
11 int div(int ad, int bd, int& rd){
12     int quo = ad/bd;
13     rd = ad - bd*quo;
14     cout << verif(ad,bd,quo,rd)<< endl;
15     return quo;
16 }
17
18 int main(){
19     int a=20, b=3, r;
20     int q = div(a,b,r);
21     return 0;
22 }
```

On rentre progressivement dans les fonctions.

Chaque fois que l'on rentre dans une fonction on gagne un niveau.

Chaque fois que l'on sort d'une fonction on perd un niveau.

C'est une structure de **pile**.

Contextes d'exécutions et portée des variables

À chaque niveau de profondeur (un étage de la pile d'appels) correspond un **contexte d'exécution** qui contient ses propres variables. Le niveau n n'a **pas** accès aux variables des niveaux $n-1$ et $n+1$, à moins :

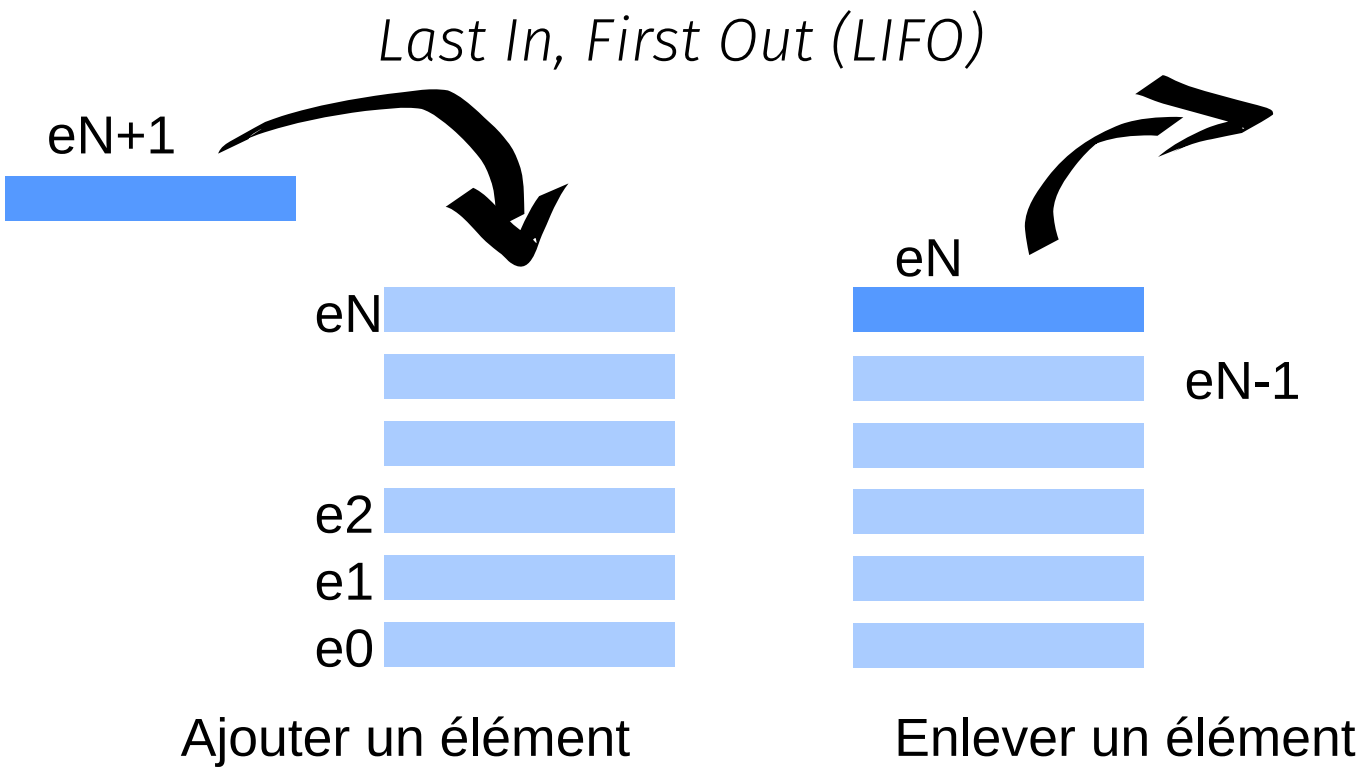
- d'avoir passé ces variables par référence,
- d'avoir déclaré ces variables comme globales.

Le passage par référence permet de donner au niveau $n+1$ accès à des variables du niveau n.

Les variables globales sont accessibles depuis tous les contextes d'exécution.

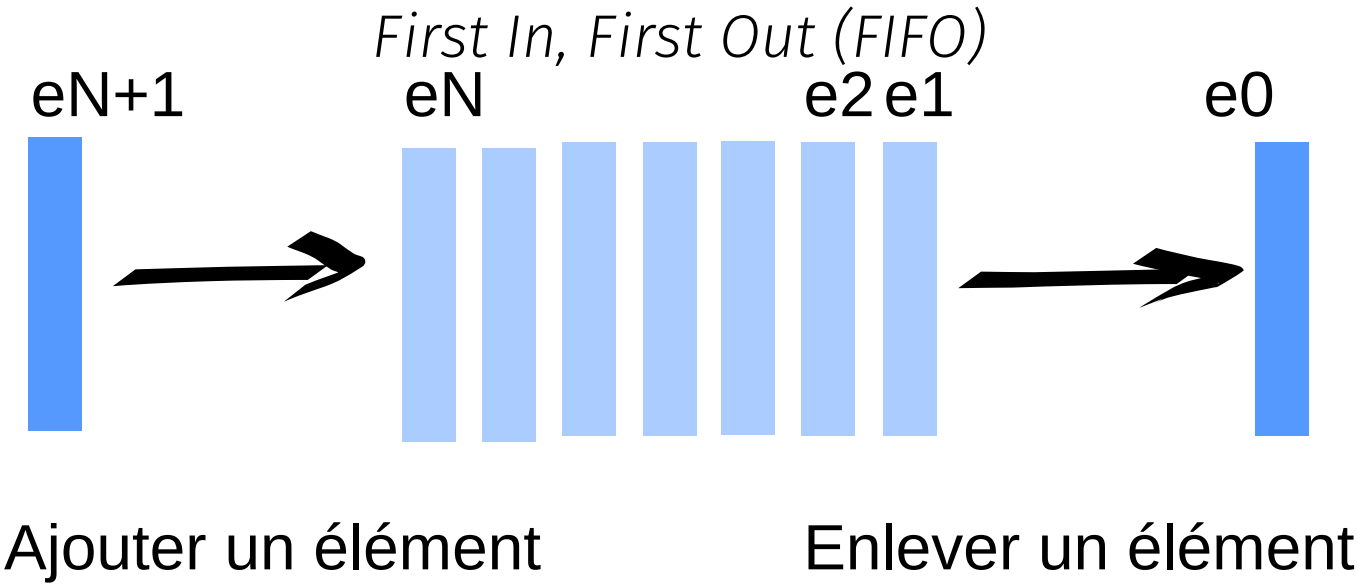
Pile - Dernier arrivé, premier servi

Une pile est structure de données telle que **les dernières données ajoutées seront les premières à être retirées** (comme une pile d'assiettes).



File - Premier arrivé, premier servi

Une file est une structure de données telle que **les premières données ajoutées seront les premières à être retirées** (comme une file d'attente).

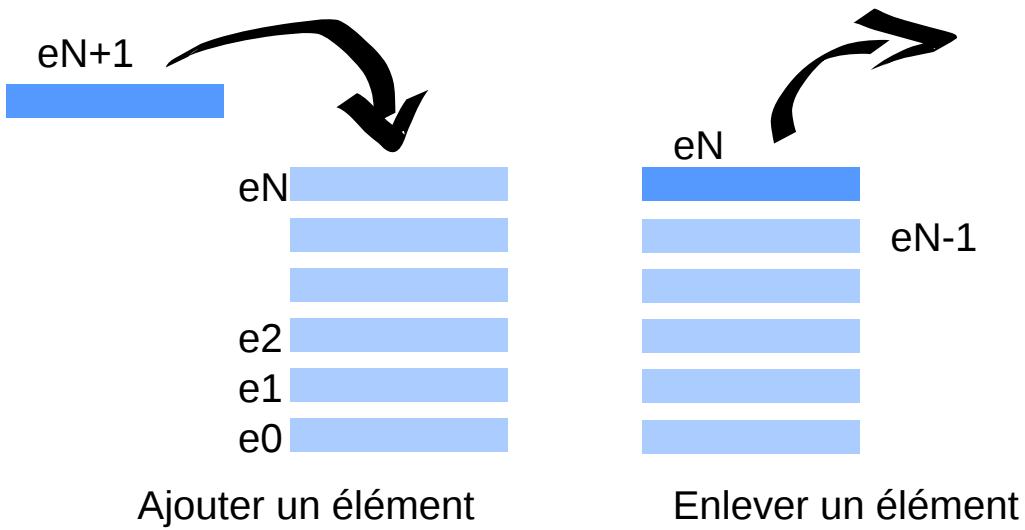


Pile des fonctions

Les appels aux fonctions sont gérés à l'aide d'une pile.

- **Entrer dans une fonction** : ajouter un élément à la pile
- **Sortir d'une fonction** : enlever un élément à la pile

La pile des fonctions permet de garder en mémoire l'ordre d'appel des fonctions.



Plan de la séance - Gestion de la mémoire

Les variables locales

Contenu d'un contexte d'exécution

Un contexte d'exécution (un élément de la pile d'appel) contient :

- La fonction qui est appelée,
- Les variables locales, **y compris les arguments de la fonction**.

Attention

Les variables locales sont créées et accessibles uniquement dans le contexte d'exécution dans lequel elles ont été déclarées.

Appel d'une fonction

```
1  #include <iostream>
2
3  using namespace std;
4
5  bool verif(int av, int bv, int qv, int rv){
6      if(qv<0 || rv>=bv || av != bv*qv+rv)
7          return false;
8      return true;
9  }
10
11 int div(int ad, int bd, int& rd){
12     int quo = ad/bd;
13     rd = ad - bd*quo;
14     cout << verif(ad,bd,quo,rd)<< endl;
15     return quo;
16 }
17
18 int main(){
19     int a=20, b=3, r;
20     int q = div(a,b,r);
21     return 0;
22 }
```

main
Vars
a = 0
b = -7936
q = 32767
r = 4196864

Éléments avant le main

Appel d'une fonction

```
1  #include <iostream>
2
3  using namespace std;
4
5  bool verif(int av, int bv, int qv, int rv){
6      if(qv<0 || rv>=bv || av != bv*qv+rv)
7          return false;
8      return true;
9  }
10
11 int div(int ad, int bd, int& rd){
12     int quo = ad/bd;
13     rd = ad - bd*quo;
14     cout << verif(ad,bd,quo,rd)<< endl;
15     return quo;
16 }
17
18 int main(){
19     int a=20, b=3, r;
20     int q = div(a,b,r);
21     return 0;
22 }
```

main
Vars
a = 20
b = 3
q = 32767
r = 4196864

Éléments avant le main

Appel d'une fonction

```
1  #include <iostream>
2
3  using namespace std;
4
5  bool verif(int av, int bv, int qv, int rv){
6      if(qv<0 || rv>=bv || av != bv*qv+rv)
7          return false;
8      return true;
9  }
10
11 int div(int ad, int bd, int& rd){
12     int quo = ad/bd;
13     rd = ad - bd*quo;
14     cout << verif(ad,bd,quo,rd)<< endl;
15     return quo;
16 }
17
18 int main(){
19     int a=20, b=3, r;
20     int q = div(a,b,r);
21     return 0;
22 }
```

div
Vars
  ad = 20
  bd = 3
  quo = 0
  rd {r} = 4196864

main
Vars
  a = 20
  b = 3
  q = 32767
  r = 4196864

Éléments avant le main

Appel d'une fonction

```
1  #include <iostream>
2
3  using namespace std;
4
5  bool verif(int av, int bv, int qv, int rv){
6      if(qv<0 || rv>=bv || av != bv*qv+rv)
7          return false;
8      return true;
9  }
10
11 int div(int ad, int bd, int& rd){
12     int quo = ad/bd;
13     rd = ad - bd*quo;
14     cout << verif(ad,bd,quo,rd)<< endl;
15     return quo;
16 }
17
18 int main(){
19     int a=20, b=3, r;
20     int q = div(a,b,r);
21     return 0;
22 }
```

div
Vars
  ad = 20
  bd = 3
  quo = 6
  rd {r} = 4196864

main
Vars
  a = 20
  b = 3
  q = 32767
  r = 4196864

Éléments avant le main

Appel d'une fonction

```
1  #include <iostream>
2
3  using namespace std;
4
5  bool verif(int av, int bv, int qv, int rv){
6      if(qv<0 || rv>=bv || av != bv*qv+rv)
7          return false;
8      return true;
9  }
10
11 int div(int ad, int bd, int& rd){
12     int quo = ad/bd;
13     rd = ad - bd*quo;
14     cout << verif(ad,bd,quo,rd)<< endl;
15     return quo;
16 }
17
18 int main(){
19     int a=20, b=3, r;
20     int q = div(a,b,r);
21     return 0;
22 }
```

div
Vars
  ad = 20
  bd = 3
  quo = 6
  rd {r} = 2

main
Vars
  a = 20
  b = 3
  q = 32767
  r = 2

Éléments avant le main

Appel d'une fonction

```
1  #include <iostream>
2
3  using namespace std;
4
5  bool verif(int av, int bv, int qv, int rv){
6      if(qv<0 || rv>=bv || av != bv*qv+rv)
7          return false;
8      return true;
9  }
10
11 int div(int ad, int bd, int& rd){
12     int quo = ad/bd;
13     rd = ad - bd*quo;
14     cout << verif(ad,bd,quo,rd)<< endl;
15     return quo;
16 }
17
18 int main(){
19     int a=20, b=3, r;
20     int q = div(a,b,r);
21     return 0;
22 }
```

verif
Vars
  ad = 20
  bd = 3
  quo = 6
  rv = 2

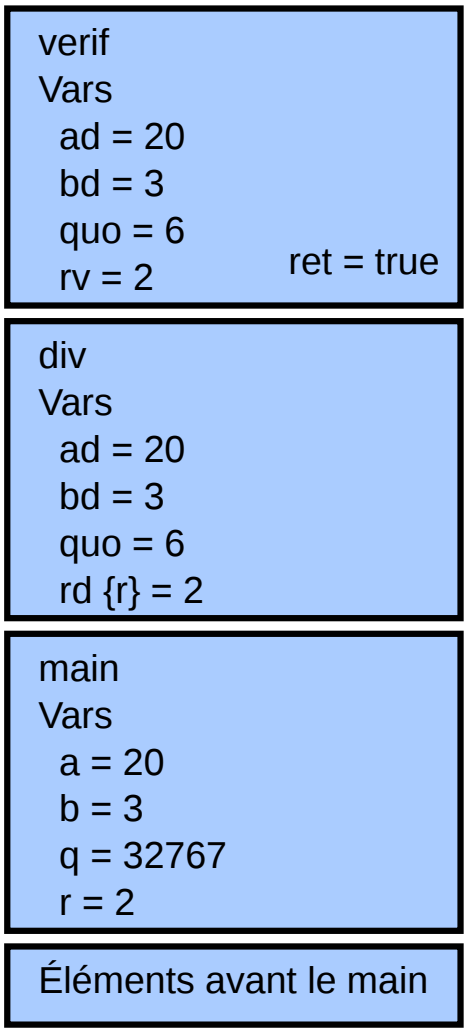
div
Vars
  ad = 20
  bd = 3
  quo = 6
  rd {r} = 2

main
Vars
  a = 20
  b = 3
  q = 32767
  r = 2

Éléments avant le main

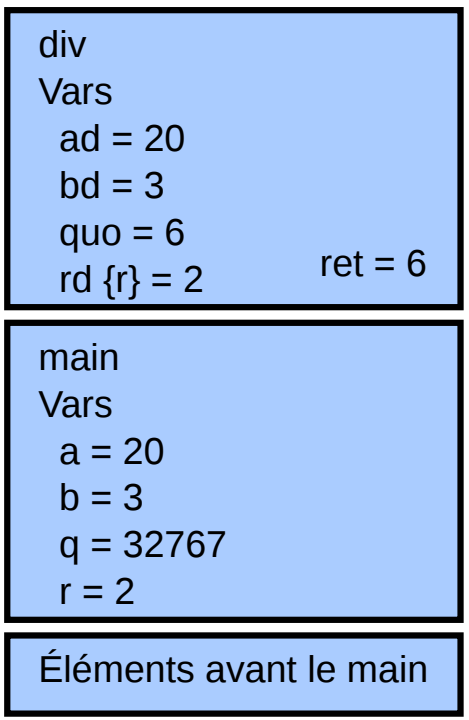
Appel d'une fonction

```
1  #include <iostream>
2
3  using namespace std;
4
5  bool verif(int av, int bv, int qv, int rv){
6      if(qv<0 || rv>=bv || av != bv*qv+rv)
7          return false;
8      return true;
9  }
10
11 int div(int ad, int bd, int& rd){
12     int quo = ad/bd;
13     rd = ad - bd*quo;
14     cout << verif(ad,bd,quo,rd)<< endl;
15     return quo;
16 }
17
18 int main(){
19     int a=20, b=3, r;
20     int q = div(a,b,r);
21     return 0;
22 }
```



Appel d'une fonction

```
1  #include <iostream>
2
3  using namespace std;
4
5  bool verif(int av, int bv, int qv, int rv){
6      if(qv<0 || rv>=bv || av != bv*qv+rv)
7          return false;
8      return true;
9  }
10
11 int div(int ad, int bd, int& rd){
12     int quo = ad/bd;
13     rd = ad - bd*quo;
14     cout << verif(ad,bd,quo,rd)<< endl;
15     return quo;
16 }
17
18 int main(){
19     int a=20, b=3, r;
20     int q = div(a,b,r);
21     return 0;
22 }
```



Appel d'une fonction

```
1  #include <iostream>
2
3  using namespace std;
4
5  bool verif(int av, int bv, int qv, int rv){
6      if(qv<0 || rv>=bv || av != bv*qv+rv)
7          return false;
8      return true;
9  }
10
11 int div(int ad, int bd, int& rd){
12     int quo = ad/bd;
13     rd = ad - bd*quo;
14     cout << verif(ad,bd,quo,rd)<< endl;
15     return quo;
16 }
17
18 int main(){
19     int a=20, b=3, r;
20     int q = div(a,b,r);
21     return 0;
22 }
```

main
Vars
a = 20
b = 3
q = div_ret = 6
r = 2

Éléments avant le main

Plan de la séance - Gestion de la mémoire

Définition des fonctions récurrents

Une **fonction récurrente** est une fonction qui s'appelle elle-même.

C'est le même principe qu'une suite récurrente :

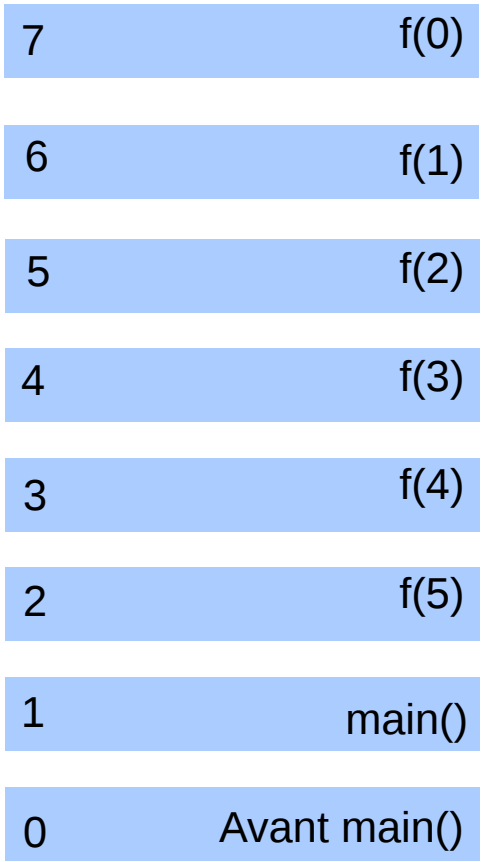
$$u : u_{n+1} = f(u_n)$$

Il est possible de définir des fonctions récurrentes en C++ grâce à la pile des appels.

Fonctions récurrentes

```
void f(int x){
    cout << x - 1 << endl;
    if(x > 0){
    // f s'appelle elle-même
        f(x - 1);
    }
}

int main(){
    f(5);
}
```



Pile des appels

Exemple - Calcul de $n!$

Expression mathématique sous forme récursive

$$fact(n) = \begin{cases} n \times fact(n-1)\ , & \text{si } n > 0 \\ 0\ , & \text{sinon.} \end{cases}$$

Implémentation récurrente en C++

```
int fact(int n){
    if(n <= 0){
      // Cas de base
        return 1;
    } else {
      // Formule de récurrence
        return n * fact(n - 1);
    }
}
```

Implémentation récurrente ou séquentielle ?

Il est **toujours possible** d'écrire une fonction récurrente sous une forme séquentielle (sans récurrence) en explicitant la pile d'appels dans le code.

Exemple

```java
// Implémentation récurrente
int fact(int n){
    if(n <= 0){
        return 1;
    } else {
        return n*fact(n - 1);
    }
}
```

```java
// Implémentation
 ↪  séquentielle
int fact(int n){
    int res = 1;
    for(int i=1; i<n+1; i++){
        res *= i;
    }
    return res;
}
```

Attention

Terminaison

Il faut s'assurer qu'il y a **terminaison** de la fonction récurrente dans tous les cas (*i.e.* éviter les boucles infinies !).

Taille de la pile

La taille de la pile est limitée. S'il y a trop d'appels de fonctions, la pile atteint sa taille maximale et stoppe le programme.

Vitesse

Appeler une fonction est « coûteux », une version séquentielle sera en général plus rapide...

- ...si la fonction est « petite »(rapide à exécuter),
- ...et qu'il y a beaucoup d'appels à cette fonction.

Plan de la séance - Gestion de la mémoire

Les limites des tableaux statiques

Remplir la pile

La taille de la pile est limitée, par exemple :

```
int tab[4000000];
```

remplit la pile, et fait planter le programme.

Le tas

Il existe une autre zone mémoire, accessible dynamiquement à la demande du programme : **le tas**.

Créer des tableaux dans le tas

```
...
int taille = 1e7; // taille pas forcément constante

int* tab = new int[taille]; // réserve la place dans le tas
...
// utilisation du tableau comme un tableau classique
...

delete[] tab; // désalloue la mémoire occupée dans le tas
```

- La taille du tableau n'a pas à être connu au moment de la compilation,
- Le tableau peut changer de taille au cours du programme,
- Il ne faut pas oublier le **`delete [] nom`**.

Manipulation des tableaux dynamiques

Ils s'utilisent comme les tableaux classiques.

```
void remplir_tableau(int tab[], int taille){
    for(int i = 0; i < taille, i++)
      tab[i] = rand() %10; }

int somme(int tab[], int taille){
    int s=0;
    for(int i = 0; i < taille; i++)
        s+=tab[i];
    return s; }

int main(){
    int t1[20];
    remplir_tableau(t1, 20);
    cout << somme(t1,20) << endl;
    int taille2 = 1000;
    int* t2= new int[taille2];
    remplir_tableau(t2, taille2);
    cout << somme(t2,taille2) << endl;
    delete[] t2; }
```

Manipulation des tableaux dynamiques

Comme précédemment, pour effectuer des opérations sur un tableau, on procède terme à terme.

```
int taille = 10000;
int* tab1, tab2;
tab1 = new int[taille];
...
tab2 = new int[taille];
for(int i=0; i<taille; i++){
    tab2[i] = tab1[i];
}
...
delete[] tab1;
delete[] tab2;
```

Copier un tableau : erreur

```
int taille = 10000;
int* tab1 = new int[taille];
for(int i=0; i<taille; i++) tab1[i]=1;
...
int* tab2;
tab2 = tab1; // ATTENTION ne produit pas d'erreur de
 ↪  compilation
             // on verra pourquoi au chapitre 8

// mais on n'a pas créé deux tableaux (c'est le même avec deux
 ↪  noms)
tab2[0] = 10;
cout << tab1[0] << endl; // affiche tab1
```

Attention

Ne jamais faire d'affectation d'un tableau dans un autre.

Redimensionner un tableau

Il est possible de redimensionner un tableau. Dans ce cas, il est nécessaire de le désallouer avant de réallouer.

```
bool* tab = new bool[10];
// tab est un tableau de taille 10
...
delete[] tab;
tab = new bool[5000];
// tab est désormais un tableau de taille 5000
...
delete[] tab;
```

Redimensionner un tableau

Si l'on souhaite conserver les éléments du tableau, il faut passer par un tableau auxiliaire.

```
int taille1 =10, taille2=500;
bool* tab = new bool[taille1];
bool* tab_temporaire = new bool[taille1];

// Copie
for(int i=0; i<taille1; i++) tab_temporaire[i] = tab[i];

// Libération de la mémoire
delete[] tab;
// Allocation du nouveau tableau
tab = new bool[taille2];

// Recopie des éléments
for(int i=0; i<taille1; i++) tab[i] = tab_temporaire[i];
delete[] tab_temporaire;
delete[] tab;
```

Erreurs classiques

```
// Oublier d'allouer
int m = 100;
double* tab; // !!
for(int i=0; i<m; i++){
    tab[i] = 0;
}
delete[] tab;

// Oublier de désallouer
int n = 10000;
for(int i=0; i<10; i++){
    bool* tab2 = new bool[n];
    // Fuite de mémoire !!
}
```

```
// Désallouer deux fois
int tab3 = new int[1000];
...
delete[] tab3;
...
delete[] tab3; // !!

// Se dire que c'est trop
// compliqué et que les
// tableaux statiques
// sont plus simples

int tab4[10000000]; // !!
```

Plan de la séance - Gestion de la mémoire

Des tableaux dans des structures

Il est possible d'utiliser des tableaux **dynamiques** dans les structures.

Attention !

Ne surtout pas utiliser des tableaux statiques.

```
struct Vect{
    int taille; // taille du tableau
    double* t;  // le tableau, ne pas allouer
                // dans la déclaration de la structure
};
```

Structures et allocation dynamique

```
// Vect.h
struct Vect{
    int n; // taille
    double* t; // tableau
};

void init(Vect& v);

void cree(Vect& v, int n);

void detruit(Vect& v);

void remplit(Vect& v, double
 ↪  val);

void copie(Vect& v, Vect o);

Vect operator+(Vect v1, Vect
 ↪  v2);
```

```
// Vect.cpp
#include "Vect.h"
void init(Vect& v){
    v.n = 0;
}

void cree(Vect& v, int n){
    assert(n > 0);
    v.n = n;
    v.t = new double[v.n];
}

void detruit(Vect& v){
    if(v.taille > 0){
        v.taille = 0;
        delete[] v.t;
    }
}

void remplit(Vect& v, double
 ↪  val){
```

Structures et allocation dynamique

```
// Vect.h
struct Vect{
    int n; // taille
    double* t; // tableau
};

void init(Vect& v);

void cree(Vect& v, int n);

void detruit(Vect& v);

void remplit(Vect& v, double
 ↪  val);

void copie(Vect& v, Vect o);

Vect operator+(Vect v1, Vect
 ↪  v2);
```

```
// Vect.cpp
#include "Vect.h"
void copie(Vect& v, Vect o){
    detruit(v);
    cree(v, o.taille);
    for(int i=0;i<v.n;i++)
        v.t[i] = o.t[i];
}

Vect operator+(Vect v1, Vect
 ↪  v2){
    assert(v1.n == v2.n);
    Vect v;
    cree(v, v1.n);
    for(int i=0;i<v.n; i++)
        v.t[i] =
         ↪  v1.t[i]+v2.t[i];
    return v;
}
```

Structures et allocation dynamique

```
// Vect.h
struct Vect{
    int n; // taille
    double* t; // tableau
};

void init(Vect& v);

void cree(Vect& v, int n);

void detruit(Vect& v);

void remplit(Vect& v, double
 ↪  val);

void copie(Vect& v, Vect o);

Vect operator+(Vect v1, Vect
 ↪  v2);
```

```
// main.cpp
#include "Vect.h"

int main(){
    Vect v1,v2;
    init(v1);
    init(v2);

    cree(v1, 10);
    remplit(v1, 5.6);

    copie(v2, v2);

    Vect v3 = v1 + v2;

    detruit(v1);
    detruit(v2);
    detruit(v3);
    return 0;
}
```

Plan de la séance - Gestion de la mémoire

Compilation

C++ est un langage **compilé**.
L'étape de compilation peut subir différents types d'optimisation pour rendre le programme plus sûr, plus rapide ou plus facile à débugger.

Release vs *Debug*

Release est un mode optimisé, pour rendre l'exécution plus efficace. Il n'est plus possible de suivre l'exécution du programme pas à pas.

- Ne pas essayer de débugger en mode *Release*
- Rester en mode *Debug* le plus longtemps possible (pour être sûr que le programme fonctionne correctement) avant de passer en Release.

Comment choisir son mode ?

Avec CMake, préciser la variable :

```
-CMAKE_BUILD_TYPE Debug,Release
```

Les assertions

Les assertions sont des tests qui ne sont exécutées **qu'en mode *Debug***. Elles permettent de tester des valeurs à certains endroits du programme pour faciliter le débuggage.

```
#include<cassert>
...
int n;
cin >> n;
assert(n > 0);
int* tab = new int[n];
...
delete[] tab;
```

Attention

En mode *Release*, le test n'est pas effectué.

Plan de la séance - Gestion de la mémoire

Travail

TP

- Finir le TP Gravitation
- À rendre au plus tard le 08/11.

Exercice individuel

Calculer l'histogramme des intensités dans une image.

- Utilisation des tableaux dynamiques
- À rendre au plus tard le 25/10

COMMENTER INDENTER COMPILER