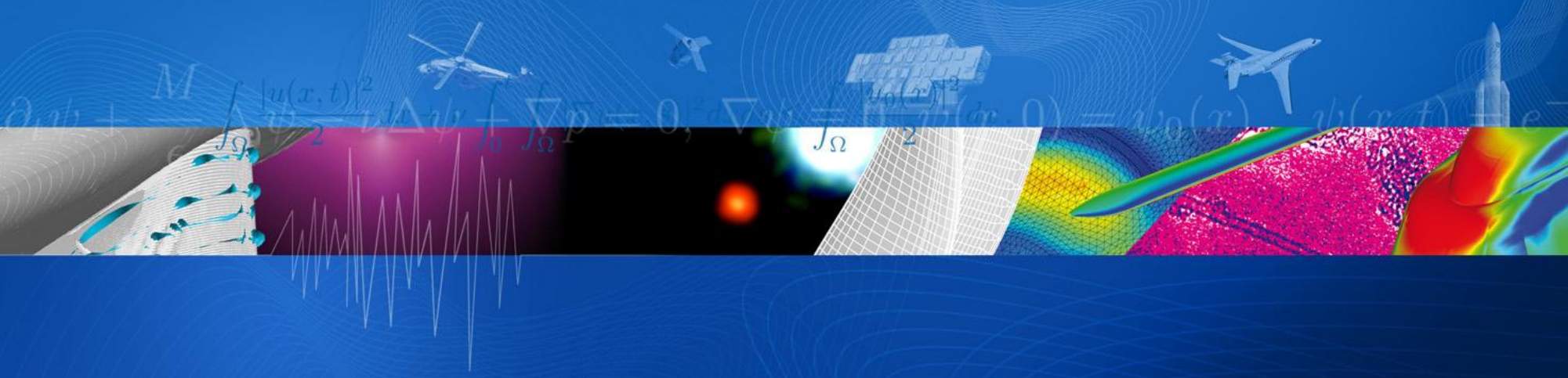


Introduction à la programmation C++

La mémoire

Nicolas Audebert

Mercredi 19 octobre 2016

ONERA
THE FRENCH AEROSPACE LAB

retour sur innovation

Plan de la séance

Déroulé d'un programme C++

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

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	main	main.cpp	19

On rentre progressivement dans les fonctions.

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	main	main.cpp	20

On rentre progressivement dans les fonctions.

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	div	main.cpp	12
1	main	main.cpp	20

On rentre progressivement dans les fonctions.
Chaque fois que l'on rentre dans une fonction on gagne un niveau.

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	div	main.cpp	13
1	main	main.cpp	20

On rentre progressivement dans les fonctions.
Chaque fois que l'on rentre dans une fonction on gagne un niveau.

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	div	main.cpp	14
1	main	main.cpp	20

On rentre progressivement dans les fonctions.
Chaque fois que l'on rentre dans une fonction on gagne un niveau.

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.

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.

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.

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	div	main.cpp	15
1	main	main.cpp	20

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.

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	div	main.cpp	16
1	main	main.cpp	20

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.

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	main	main.cpp	21

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**.

Niveau

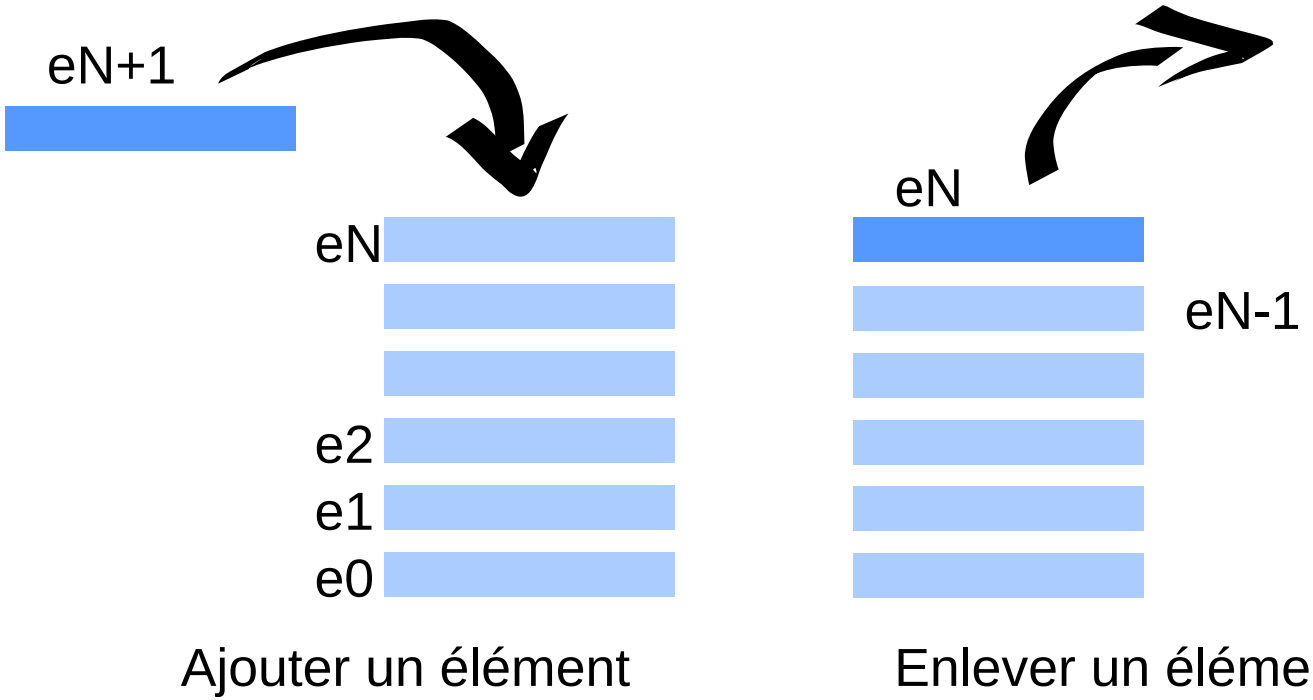
À chaque niveau correspond un contexte d'exécution, avec ses propres variables. Le niveau n n'a **pas** accès aux variables des niveaux $n-1$ et $n+1$, sauf à :

- les passer par référence,
- les déclarer comme globales.

Pile

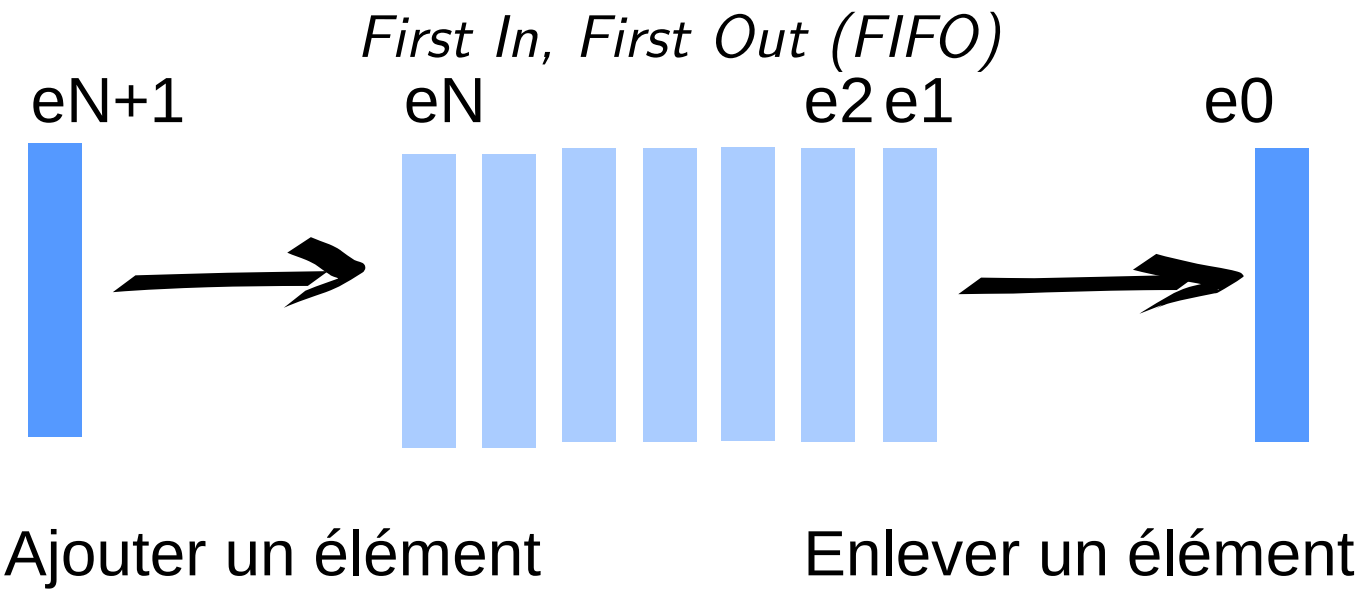
Une pile est structure de données telle que les dernières données ajoutées seront les premières à être retirées (comme une pile d'assiettes).

Last In, First Out (LIFO)



File

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

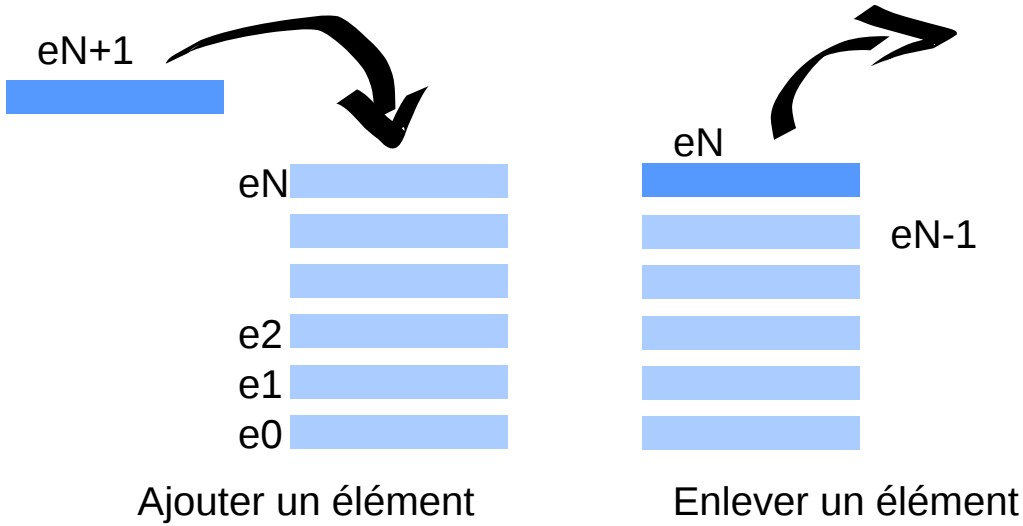


Pile des fonctions

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

Les variables locales

Constat

Les variables locales n'existent que dans les fonctions où elles ont été créées (*i.e.* dans leur niveau).

Pourquoi ?

Les variables locales sont en fait créées dans la pile.
Un élément de la pile correspond à la fonction + les variables locales (arguments compris).

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 }
```

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

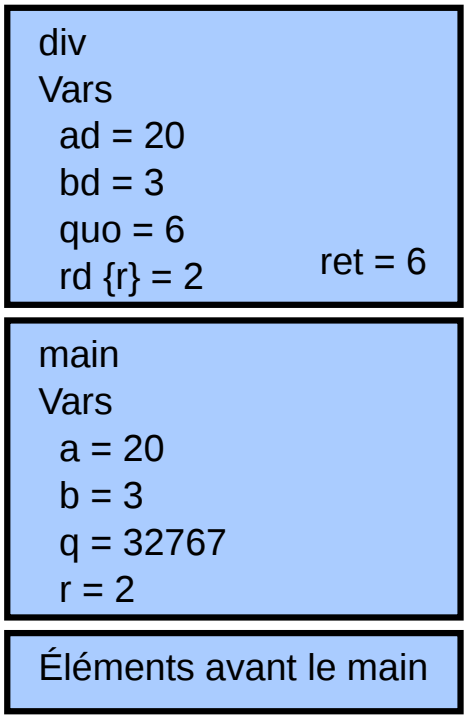
div
Vars
ad = 20
bd = 3
quo = 6
rd {r} = 2

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 }
```

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

Éléments avant le main

Plan de la séance

Fonctions récurrentes

Une fonction récurrente est une fonction qui s'appelle elle-même.
C'est le même principe qu'une suite récurrente :

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

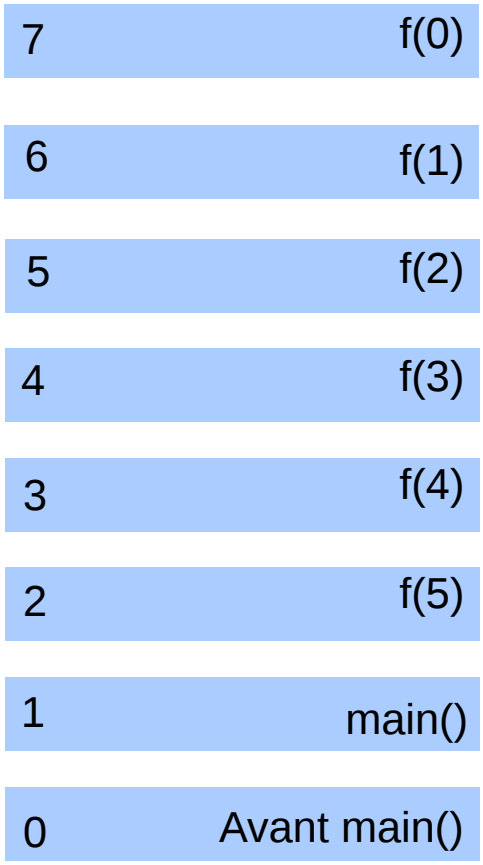
Fonctions récurrentes

Définir des fonctions récurrentes est autorisé en C++, grâce à la pile des fonctions.

C++

```
void f(int x){
    cout << x - 1 << endl;
    if(x > 0){
        f(x - 1);
    }
}

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



Pile des appels

Exemple

La fonction factorielle :

$$fact : n \rightarrow \begin{cases} n * fact(n-1) & \text{si } n > 0 \\ 1 & \text{sinon} \end{cases}$$

C++

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

Recurrent vs séquentiel

Il est toujours possible d'écrire une fonction récurrente sous une forme séquentielle (sans récurrence).
On n'utilise pas la pile des appels, celle-ci devient en quelque sorte explicite dans le code.

C++

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

C++

```
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 », si elle rapide à exécuter.
- ...et qu'il y a beaucoup d'appels à la fonction.

Plan de la séance

Les limites des tableaux statiques

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 : le tas.
C'est la RAM de l'ordinateur qui n'est pas affecté à la pile du programme.

Créer des tableaux dans le tas

C++

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

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

delete[] tab; // desalloue la memoire occuppee dans le tas
```

- La taille n'est pas forcément constante
- Tableaux à taille variable
- **Il ne faut pas oublier le `delete [] nom`**

Tableaux de taille variable et fonctions

Ils s'utilisent comme les tableaux classiques.

C++

```
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;
}
```

Copier un tableau

Comme précédemment, pour copier un tableau, on procède terme à terme.

C++

```
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

C++

```
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 cree deux tableaux (c'est le meme avec deux noms)
tab2[0] = 10;
cout << tab1[0] << endl; // affiche tab1
```

ATTENTION

Ne jamais tenter le diable, ne jamais faire d'égalité directe.

Redimensionner un tableau

On peut redimensionner un tableau, il faut juste ne pas oublier de désallouer avant de réallouer.

C++

```
bool* tab = new bool[10];
...
delete[] tab;
tab = new bool[5000];
...
delete[] tab;
```

Redimensionner un tableau 2

Si on veut garder les éléments, il faut passer par un tableau auxiliaire.

C++

```
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];

// liberation de la memoire
delete[] tab;

// allocation du nouveau tableau
tab = new bool[taille2];

// recopie des elements
for(int i=0; i<taille1; i++) tab[i] = tab_temporaire[i];
delete[] tab_temporaire;
...
delete[] tab;
```

Allocation dynamique : erreurs classiques

C++

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

// oublier de desallouer
int n = 10000;
for(int i=0; i<10; i++){
    bool* tab2 = new bool[n];
    // fuite de memoire
}
```

C++

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

// ce dire que c'est trop
// compliquer et que
// les tableaux statiques
// sont plus simples

int tab4[10000000];
//tableau trop gros
```

Plan de la séance

Des tableaux dans des structures

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

ATTENTION

Surtout pas de tableaux statiques.

C++

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

Structures et allocation dynamique

C++

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

void init(Vect& v);

void cree(Vect& v, int taille);

void detruit(Vect& v);

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

void copie(Vect& v, Vect orig);

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

C++

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

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

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

void remplit(Vect& v, double val){
    for(int i=0; i<v.taille; i++){
        v.t[i] = val;
    }
}
```

Structures et allocation dynamique 2

C++

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

void init(Vect& v);

void cree(Vect& v, int taille);

void detruit(Vect& v);

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

void copie(Vect& v, Vect orig);

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

C++

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

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

Structures et allocation dynamique 2

C++

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

void init(Vect& v);

void cree(Vect& v, int taille);

void detruit(Vect& v);

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

void copie(Vect& v, Vect orig);

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

C++

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

int main(){
    cout << "Debut main" << endl;

    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);
    cout << "Fin main" << endl;
    return 0;
}
```

Plan de la séance

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.

C++

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

Dans ce programme, `assert` permet de vérifier que la variable `n` est strictement positive, avant de créer un tableau.

Attention

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

Plan de la séance

Travail

TP

- Finir le TP Gravitation
- À rendre au plus tard le 02/11.

Exercice individuel

Un robot qui parcourt une grille de façon autonome.

- Utilisation de fonctions récursives
- À rendre au plus tard le 09/11

COMMENTER INDENTER COMPILER