# Algorithmique et Programmation
# Niveau 2

Frédéric Goualard

Laboratoire des Sciences du Numérique de Nantes, UMR CNRS 6004
Office #112-11

# Présentation du cours

- Lire le syllabus sur madoc
- Fascicules
- Format :
  - 3 cours magistraux seulement
    - Présentation synthétique des éléments du cours
  - 6 séances de travaux dirigés
    - Deux contrôles continus sur table
      (CC1, 30 minutes, sur 5 points et CC2, 1h, sur 15 points)
    - Trois quiz sur madoc (Q1, Q2, Q3)
    - **Faire les exercices à l'avance** (liste minimale sur le planning)
  - 8 séances de travaux pratiques
    - Un projet (P1) sur 20 points
- Évaluation :

$$\text{Note CC} = \frac{60(CC1 + CC2) + 40P1}{100}$$

- Langage C++ *sans POO* (standard C++11/C++17) *utilisé comme du C*

# Plan du cours

- Le langage C++ et son environnement
  - C++ et la mémoire
  - Pointeurs
  - Fonctions (passage de paramètres, pointeurs, mémoire)
  - Types C++ (tableaux, `union`, `enum`, `struct`)
- Les types abstraits, implémentations et algorithmes associés
  - Piles, files
  - Arbres, graphes
  - Tables de hachage
  - Ensembles et multi-ensembles
- Algorithmes
  - Performances des algorithmes (notation $\mathcal{O}$ — « grand O »)
  - Programmation dynamique
  - Algorithmes gloutons
  - Algorithmes *branch-and-bound*

# Planning prévisionnel

## X2BI040

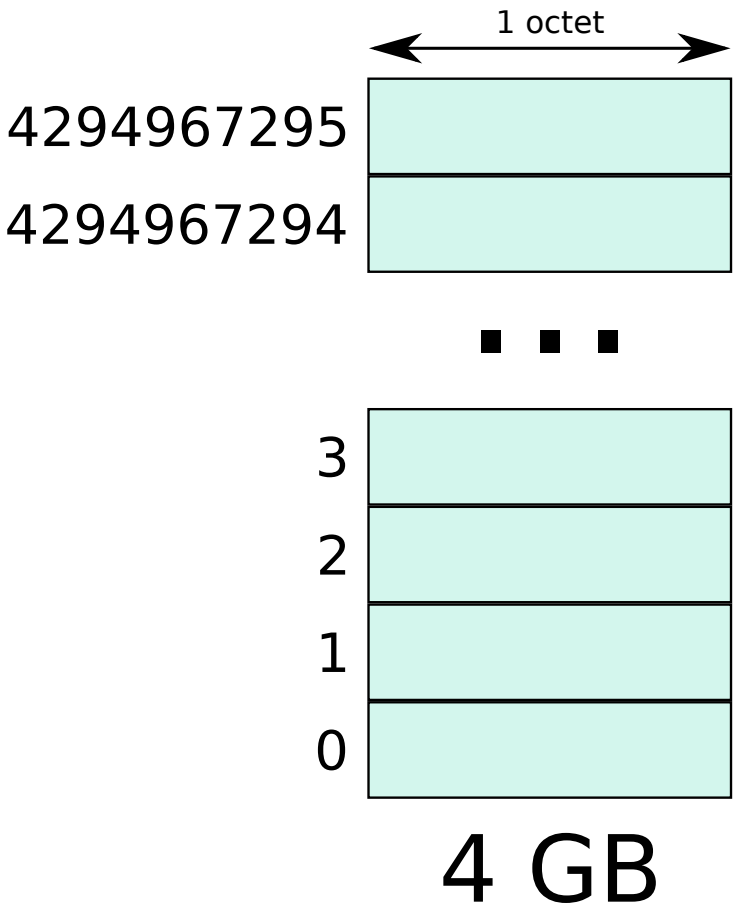
2022/2023

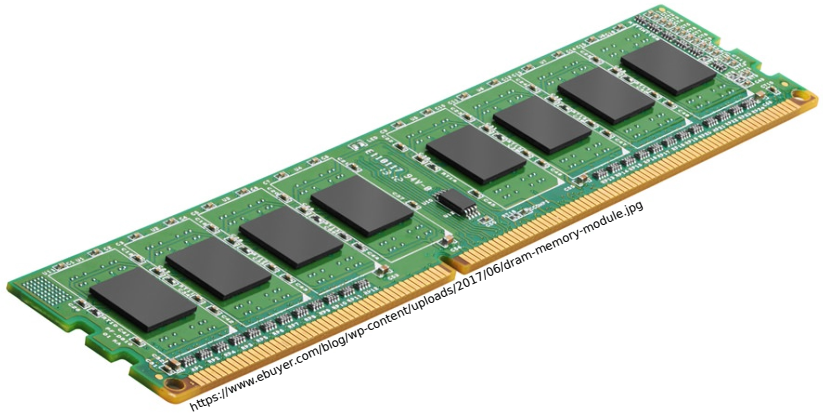
| v. 2, 2023-01-02<br>3 CMs, 6 TDs, 8 TPs | **Exercices de TDs à préparer (au minimum)** | **Quiz**<br>(date limite de complétion) | | | |
|---|---|---|---|---|---|
| (1) 02/01–06/01 | | | CM 1<br>Présentation + Langage C++ | TP 1<br>Langage C++ | TD 1<br>Langage C++ |
| (2) 09/01–13/01 | 1.4, 1.9 | | CM 2<br>Types abstraits | TD 2<br>Langage C++ | TP 2<br>Langage C++ |
| (3) 16/01–20/01 | 2.1, 2.2 | Langage C++ & environnement | TD 3<br>Types abstraits | CM 3<br>Algorithmes | TP 3<br>Types abstraits |
| (4) 23/01–27/01 | 2,4 | | CC 1<br>Langage C++ & environnement<br>TD 4<br>Types abstraits | | |
| (5) 30/01–03/02 | | Types abstraits | TD 5<br>Algorithmes | TP 4<br>Types abstraits | |
| (6) 06/02–10/02 | 3,1, 3.2 | Algorithmes | CC 2<br>Types abstraits + algorithmes | TP 5<br>Types abstraits | |
| (7) 13/02–17/02 | | | TP 6<br>Projet | | |
| (8) 20/02–24/02 | VACANCES | VACANCES | VACANCES | VACANCES | VACANCES |
| (9) 27/02–03/03 | | | TP 7<br>Projet | TP 8<br>Projet | |

Susceptible de modifications (dernière version sur madoc)

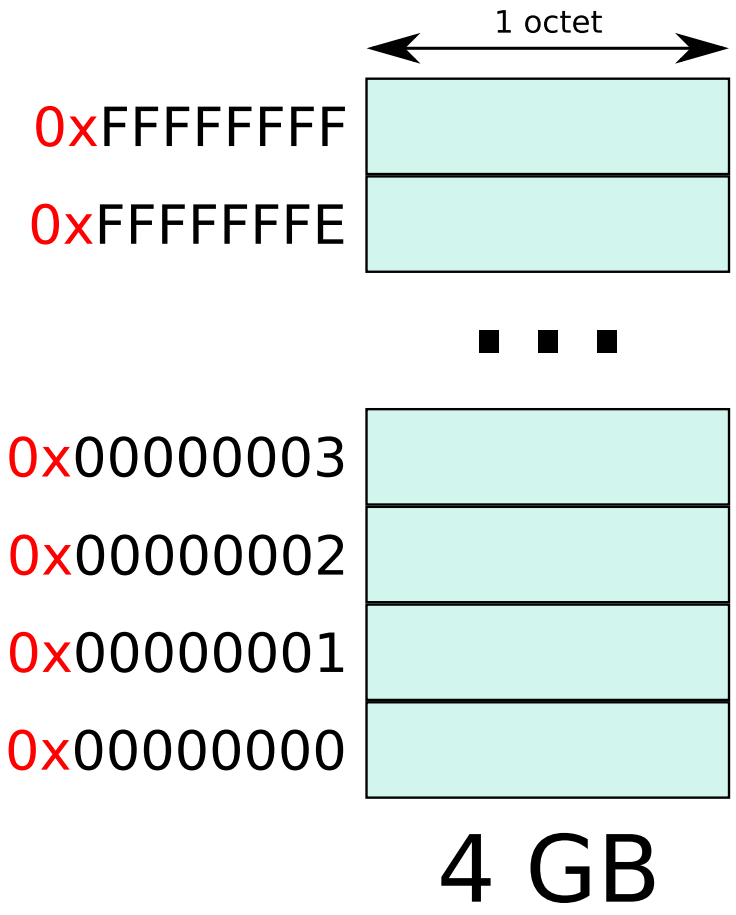
# Le langage C++ et son environnement

# La mémoire


1 octet
4294967295
4294967294
3
2
1
0
4 GB


https://www.ebuyer.com/blog/wp-content/uploads/2017/06/dram-memory-module.jpg

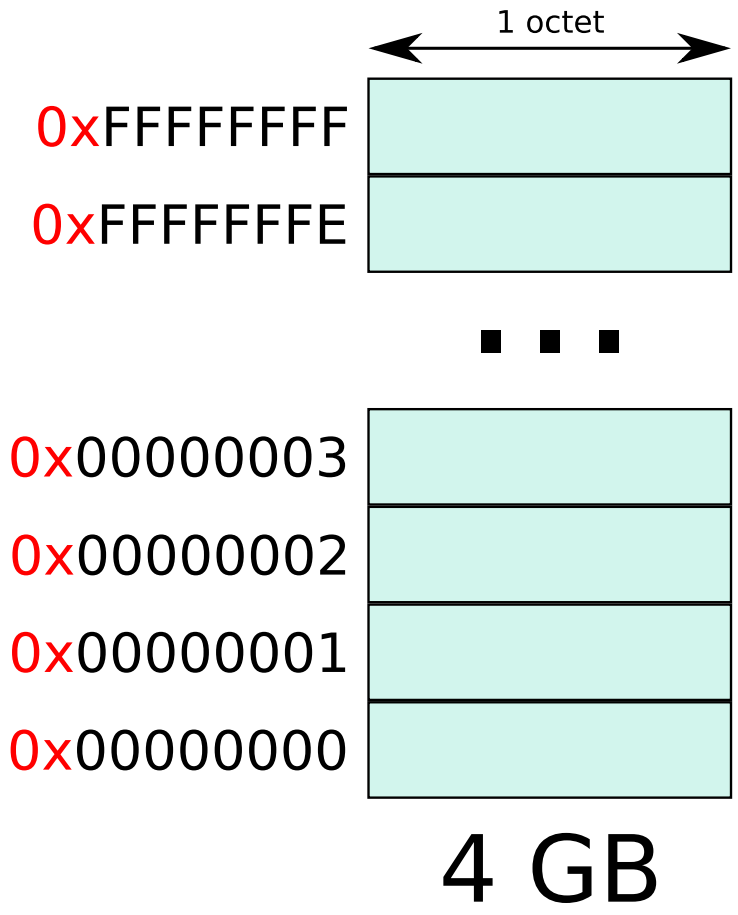
# La mémoire




## Notation hexadécimale des adresses

$$
\begin{aligned}
4294967295_{10} =& 4 \times 10^9 + 2 \times 10^8 + \cdots \\
&+ 9 \times 10^1 + 5 \times 10^0 \\
=& 15 \times 16^7 + 15 \times 16^6 \cdots \\
&+ 15 \times 16^1 + 15 \times 16^0 \\
=& F \times 16^7 + F \times 16^6 \cdots \\
&+ F \times 16^1 + F \times 16^0 \\
=& FFFFFFFF_{16}
\end{aligned}
$$

# La mémoire



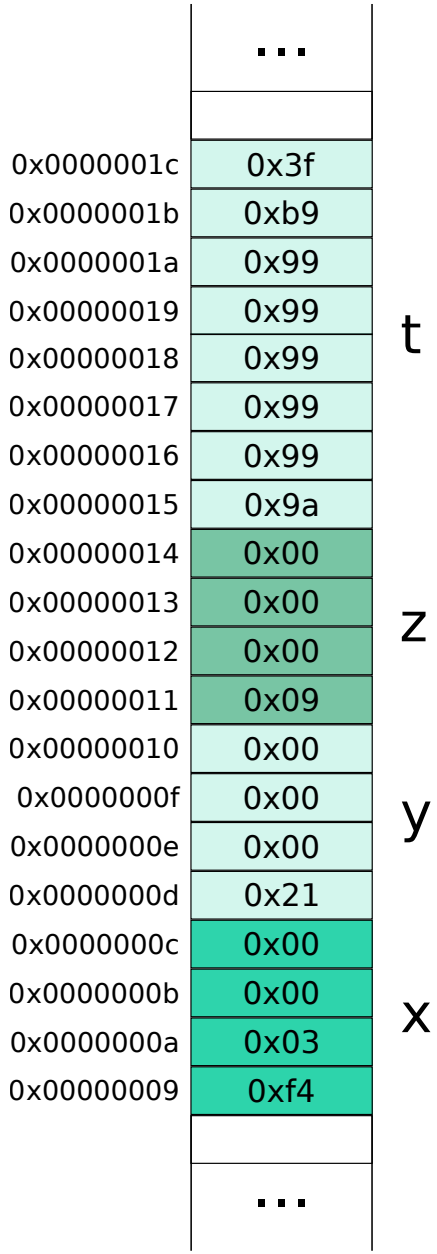

## Notation hexadécimale des adresses

$$
\begin{aligned}
4294967295_{10} =& 4 \times 10^9 + 2 \times 10^8 + \cdots \\
&+ 9 \times 10^1 + 5 \times 10^0 \\
=& 15 \times 16^7 + 15 \times 16^6 \cdots \\
&+ 15 \times 16^1 + 15 \times 16^0 \\
=& F \times 16^7 + F \times 16^6 \cdots \\
&+ F \times 16^1 + F \times 16^0 \\
=& FFFFFFFF_{16}
\end{aligned}
$$

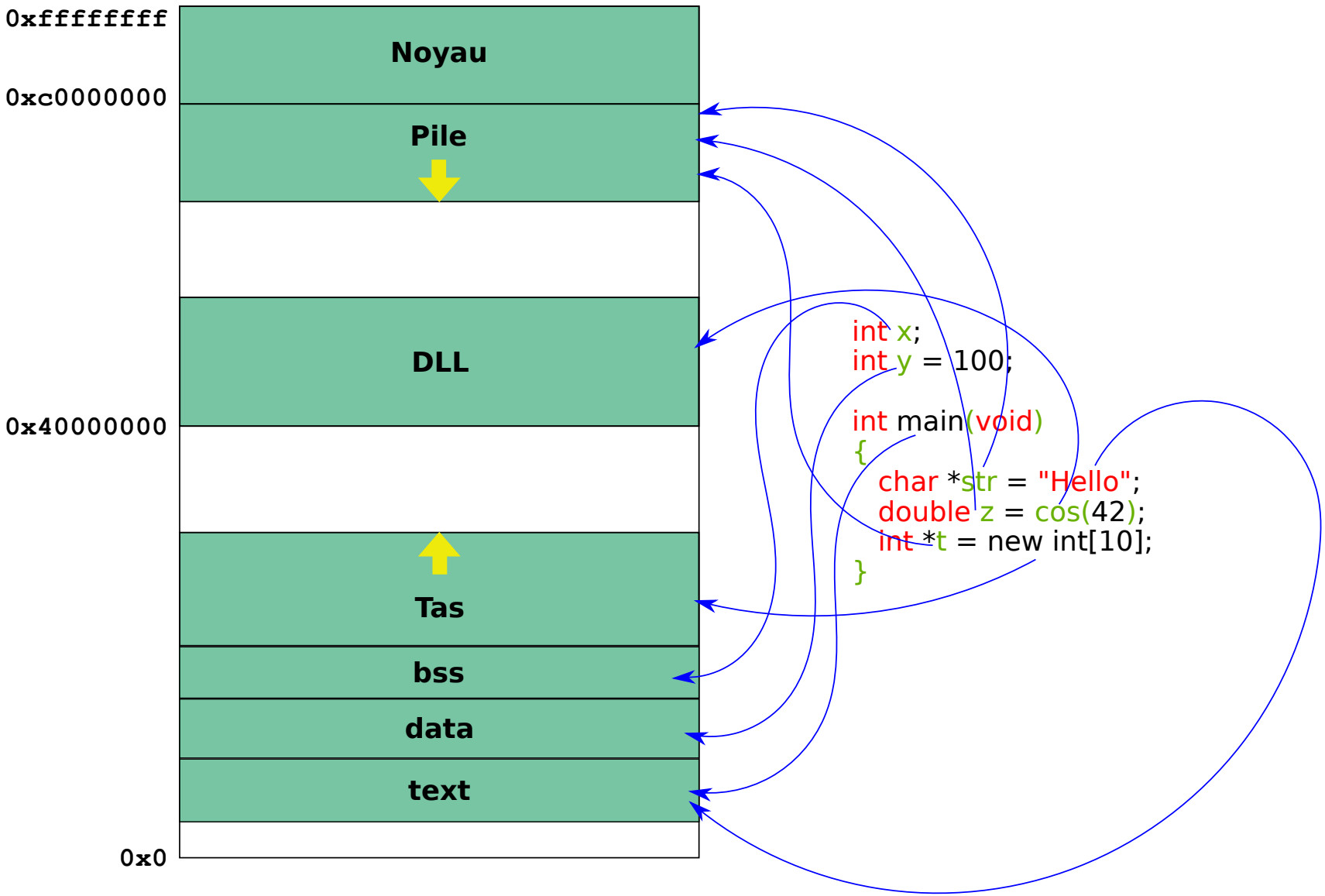
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |

# Représentation en mémoire

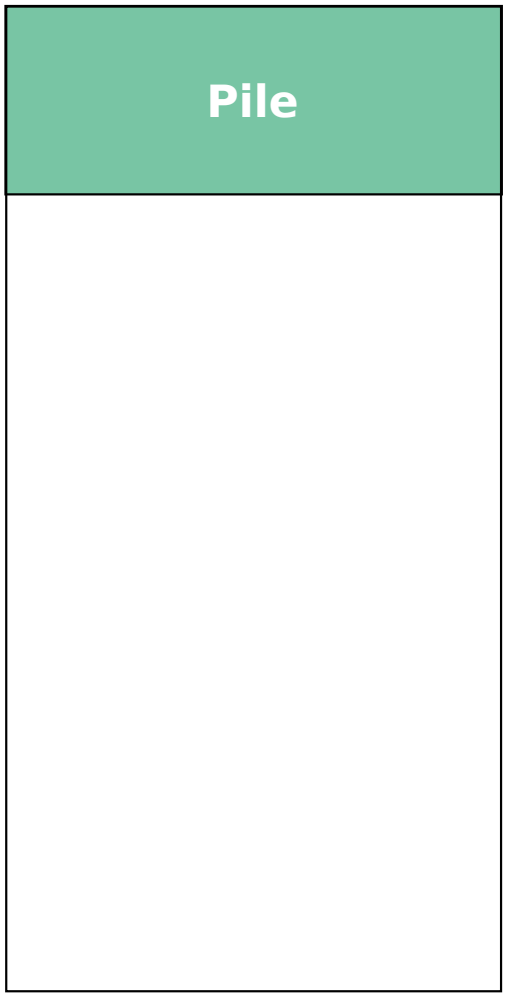
```c
// Programme sur une machine 32 bits
int x = 1012; // x == 0x3f4
int y = 0x21; // y == 2*16^1 + 1*16^0 = 33
int *z = &x;
double t = 0.1;

int main(void)
{
  // [...]
}
```



# Chargement d'un programme en mémoire

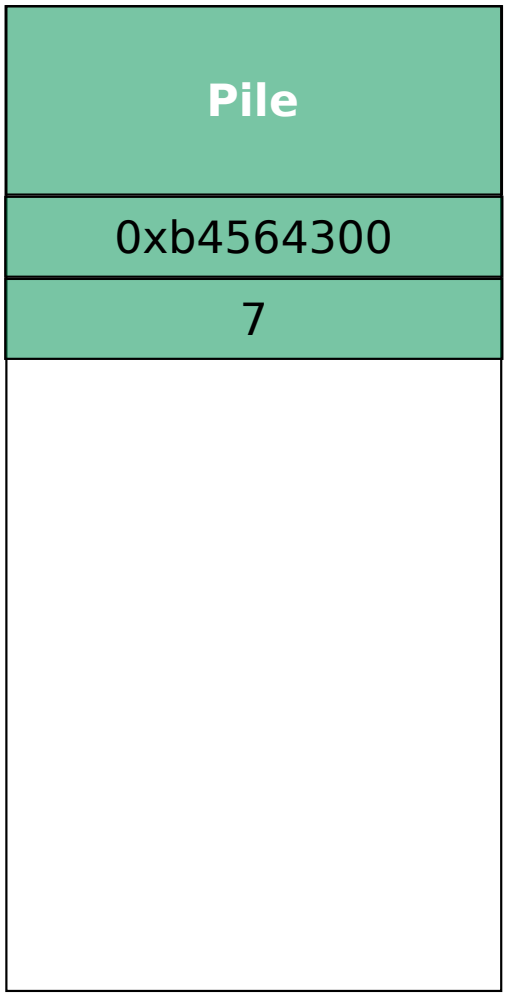


# Appel de fonction (1)



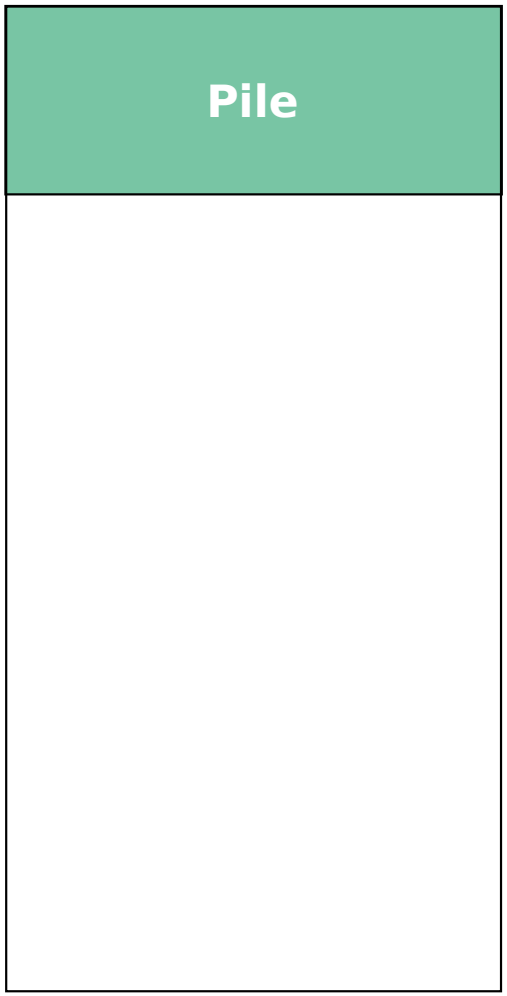

```
void f(int *a, int b)
{
  *a = b;
}

int main(void)
{
  int x = 3;
  f(&x, 7);
}
```

# Appel de fonction (1)




```
void f(int *a, int b)
{
  *a = b;
}

int main(void)
{
  int x = 3;
  f(&x, 7);
}
```

# Appel de fonction (1)




```
void f(int *a, int b)
{
  *a = b;
}

int main(void)
{
  int x = 3;
  f(&x, 7);
}
```

# Appel de fonction (2)

```
#include <iostream>

using namespace std;

int *f(int a)
{
  int b = a*a;
  return &b;
}

int main(void)
{
  int *x = f(6);

  int z1 = *x+2;
  cout << z1 << endl;
  int z2 = *x*2;
  cout << z2 << endl;
}
```

Résultat ?

# Appel de fonction (2)

```
#include <iostream>

using namespace std;

int *f(int a)
{
  int b = a*a;
  return &b;
}

int main(void)
{
  int *x = f(6);

  int z1 = *x+2;
  cout << z1 << endl;
  int z2 = *x*2;
  cout << z2 << endl;
}
```

Résultat ?

- Depuis g++ 5.0.0 : *Warning* + mise à 0 de l'adresse retournée
- Crash lors du calcul `*x+2`

# Appel de fonction (2)

```
#include <iostream>
using namespace std;

void f(int a, int **c)
{
  int b = a*a;
  *c=&b;
}


int main(void)
{
  int *x;
  f(6,&x);

  int y1 = *x+2;
  cout << y1 << "\n";
  int y2 = *x+2;
  cout << y2 << "\n";

}
```

Résultat ?

# Appel de fonction (2)

```
#include <iostream>
using namespace std;

void f(int a, int **c)
{
  int b = a*a;
  *c=&b;
}


int main(void)
{
  int *x;
  f(6,&x);

  int y1 = *x+2;
  cout << y1 << "\n";
  int y2 = *x+2;
  cout << y2 << "\n";

}
```

Résultat ?

- 38, puis 2
- Valeur locale de b écrasée sur la pile

# Passage de paramètres

- Passage par valeur. On met une copie du paramètre sur la pile

```
#include <iostream>
void add(int a, int b)
{
   a += b;
}
int main(void)
{
  int a = 12;
  std::cout << a << "\n"; // 12
  add(a,45);
  std::cout << a << "\n"; // 12
}
```

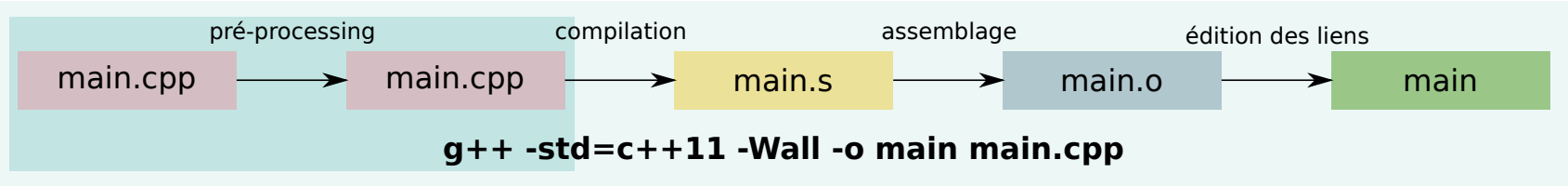
- Passage par adresse. On met une copie de l'*adresse* du paramètre sur la pile

```
#include <iostream>
void add(int *a, int b)
{
   *a += b;
}
int main(void)
{
  int a = 12;
  std::cout << a << "\n"; // 12
  add(&a,45);
  std::cout << a << "\n"; // 57
}
```

- Passage par référence. On met une copie de l'*adresse* du paramètre sur la pile

```
#include <iostream>
void add(int& a, int b)
{
   a += b;
}
int main(void)
{
  int a = 12;
  std::cout << a << "\n"; // 12
  add(a,45);
  std::cout << a << "\n"; // 57
}
```

# Macros et pré-processing



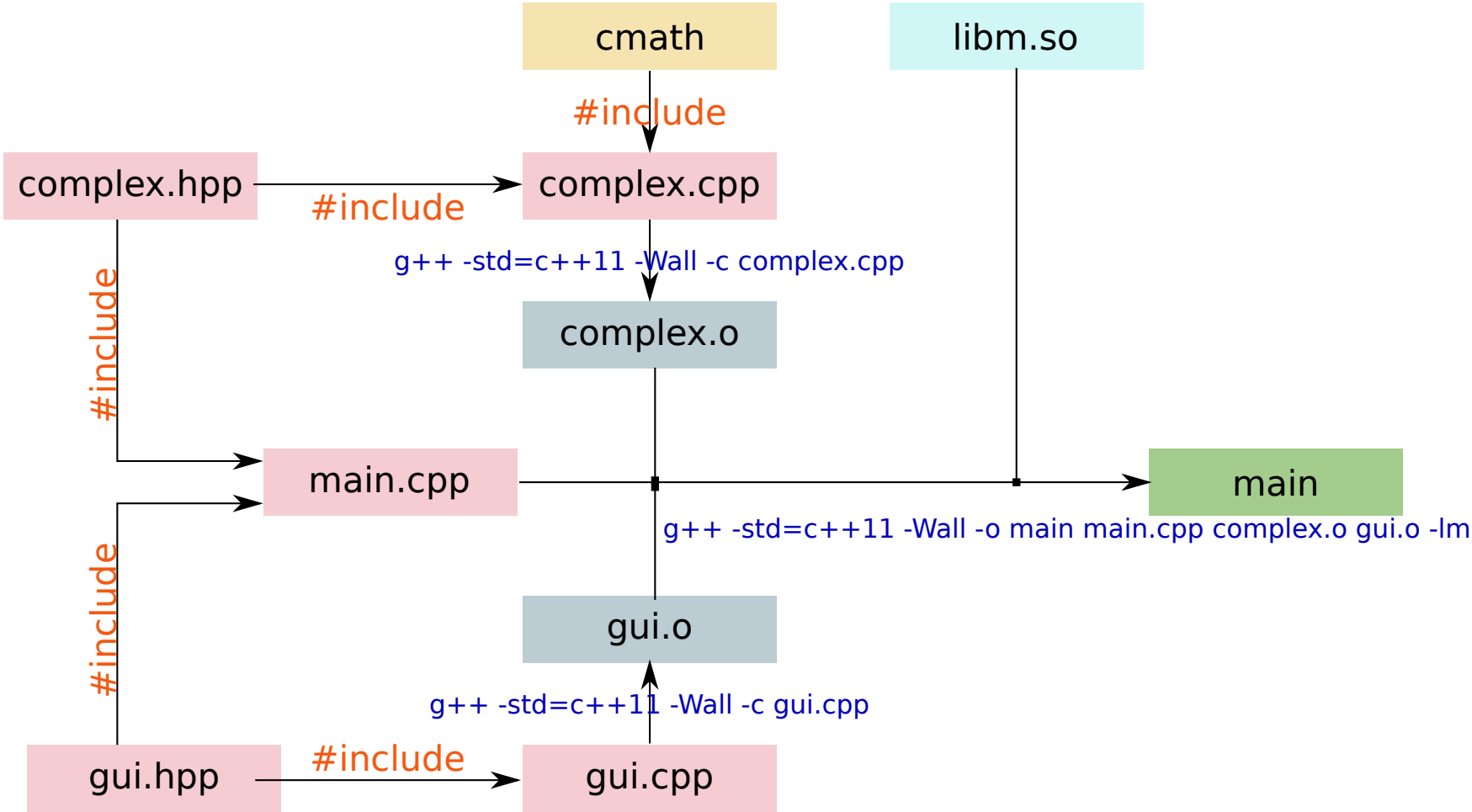

- Remplacement textuel dans l'étape de pré-processing

```
#include <iostream>
 #define SIZE 9
#ifndef NDEBUG
#  define DBG_MSG(str) do { \
          std::cerr << "DEBUG MESG: " << str << "\n";         \
  } while (0)
#else
#  define DBG_MSG(str)
#endif

int main(void)
{
  int T[SIZE];
  DBG_MSG("the program was there.");
}
```

---

```
// [Contenu de iostream]
int main(void)
{
  int T[9];
  do { std::cerr <<  "DEBUG MESG: " << "the program was there." << "\n"; } while (0);
}
```

# Compilation séparée (1)



# Compilation séparée (2)

- *Unité de compilation* : un fichier « **.cpp** »
- Chaque unité de compilation contient un ensemble cohérent de fonctions
- Fournir un en-tête « **.hpp** » pour chaque unité listant les déclarations des fonctions et des types à exporter
- Chaque unité est compilée séparément en un fichier objet
- Plusieurs fichiers objets peuvent être rassemblés en une bibliothèque (statique ou dynamique)
- *Édition des liens* : collecte de tous les fichiers objets et des bibliothèques pour générer un exécutable

# Les tableaux / chaînes de caractères

```cpp
#include <iostream>
using namespace std;
const uint32_t SZ = 5;
int T1[]{ 2, 3, 4, 5, 6 };
double T2[SZ]{ 0.5, -6.5, 3.2, 0.0, -9.1 };
char T3[]{"Hello"};
const char *T4{"Hello"};
char T5[]{'H','e','l','l','o','\0'};
int main(void)
{
  double *T6 = new double[SZ*SZ];
  //  T6[3][2] = 0.0; // Non !
  T6[3*SZ+2] = 0.0; // Oui
  *(T6+3*SZ+2) = 0.0; // Oui
  delete[] T6;
  for (uint32_t i = 0; i < SZ; ++i) {
        cout << T2[i] << " ";
  }
  cout << "\n";
}
```

```cpp
#include <iostream>
#include <vector>
#include <array>
#include <string>
using namespace std;

const uint32_t SZ = 5;
vector<int> T1{2,3,4,5};
vector<double> T2{ 0.5, -6.5, 3.2, 0.0, -9.1 };
array<double, SZ> T2b{ 0.5, -6.5, 3.2, 0.0, -9.1 };
string T3{"Hello"};

int main(void)
{
  array<array<double, SZ>, SZ> T6;
  T6[3][2] = 0.0;
  for (auto v : T2b) {
        std::cout << v << " ";
  }
  std::cout << "\n";
}
```

- Pas de *range checking* (attention aux *buffers overflow/overrun*)
- Temps d'accès en $\mathcal{O}(1)$
- Éléments contigus en mémoire $\Rightarrow$ exploitation des caches possible
- `array` : connaît sa taille (`.size()`) ; *value semantic*
- Utiliser « `array` » si tableau de taille fixe connue à la compilation
- Utiliser « `vector` » si tableau de taille variable
- Éviter les tableaux POD si possible

# Les enums

```cpp
#include <iostream>
int main(void)
{
  enum ville_t {
    londres,       // 0
    paris,         // 1
    bruxelles = 5, // 5
    amsterdam      // 6
  };
  enum class pays_t {
    france, // 0
    espagne, // 1
    belgique // 2
  };

  ville_t x = londres;
  ville_t y = amsterdam;
  if (x+y >= 6) {
    std::cout << "ok!\n";
  }
  std::cout << int(pays_t::france) << "\n";
}
```

- `enum` :
  - Déclaration de constantes globales de type `int`
- `enum class`
  - Déclarations de constantes dans l'espace de nom du type
  - Pas de valeur `int` sans cast explicite

# Les unions

```c
int main(void)
{
  union divers_t {
          int i;
          double d;
          char c;
  };
  divers_t x {.i = 5};
  x.d = 12.3;
}
```

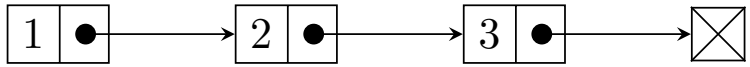
- Tous les champs partagent l'espace mémoire (`sizeof(divers_t)` $\geqslant$ taille du plus grand champ)
- Stocker à part quel est le champ actuellement utilisé
- Deux utilisations :
  - Réduire l'espace utilisé
  - réinterpréter une valeur dans un nouveau contexte

# Les structs

```cpp
#include <cstdint>
int main(void)
{
  struct personne_t {
         const char *nom;
         const char *prenom;
         uint8_t age;
  };
  using personnep_t = personne_t*; //  typedef personne_t* personne_t_p;
  personne_t ichabod = {
    "Ichabod",
    "Crane",
    35
  };
  personnep_t icha_p = &ichabod;
}
```

- Chaque champ a son propre espace mémoire
- Utile pour stocker des valeurs en relation
- `sizeof(personne_t)` $\geqslant$ `sizeof(char*)*2+sizeof(uint8_t)`

# Les listes chaînées




```cpp
#include <iostream>
using namespace std;
struct node_t {
  int v;
  node_t *next;
};

int main(void)
{
  node_t *head {nullptr};
  for (int i = 3; i >= 1; --i) {
      node_t *current {new node_t};
      current->v = i;
      current->next = head;
      head = current;
  }
  node_t *tmp {head};
  while (tmp != nullptr) {
      cout << tmp->v << " ";
      tmp = tmp->next;
  }
  cout << '\n';

  // Destruction de la liste
  while (head != nullptr) {
      node_t *next {head->next};
      delete head;
      head = next;
  }
}
```

- Construction indispensable pour la gestion de structures dynamiques (piles, files, ...)
- Temps d'accès en $\mathcal{O}(n)$
- Réorganisation des éléments peu coûteuse
- Éléments non contigus en mémoire $\Rightarrow$ impact sur les performances
- Disponible dans la bibliothèque standard STL (« `list` »)

# Fin du cours