Atelier C++
Partie 1

Henri Louvin

Objectifs

Notions abordées durant ces cours

- Pointeurs
- Pointeurs intelligents
- **Vecteurs et itérateurs**
- Surcharge de fonction
- Surcharge d’opérateurs
- **Polymorphisme**

Mise en route

Avant d'aller plus loin...

- Positionnement des bananes

```
for (i=0; i<size; i++)
{
    cout << i << endl;
}
```

vs.

```
for (i=0; i<size; i++) {
    cout << i << endl;
}
```

vs.

```
for (i=0; i<size; i++)
    cout << i << endl;
```

- Nommage des fichiers :
  - Sources : .c .C .c++ .cxx .cpp .cc
  - Headers : .h .H .h++ .hxx .hpp .hh

- Utilisation d'un *Integrated Development Environment* (IDE)?

Mise en route

Commandes terminal indispensables

Lister fichiers

```
$ ls
```

Change directory

```
$ cd
```

***Copy/Move*/Renommer ficher**

```
$ cp/mv $file $destination
```

Créer fichier

```
$ touch $file
```

***Remove* fichier**

```
$ rm $file
```

Make directory

```
$ mkdir $directory
```

***Remove* dossier**

```
$ rm -r $directory
```

Éditer un fichier

```
$ gedit/mousepad/??? file
```

Exécuter un fichier

```
$ ./file
```

Exemple de code

Créez un ficher **test.cc** contenant le code suivant

```
#include <iostream>

int main()
{
    std::cout<<"Hodor."<<std::endl;
    return 0;
}
```

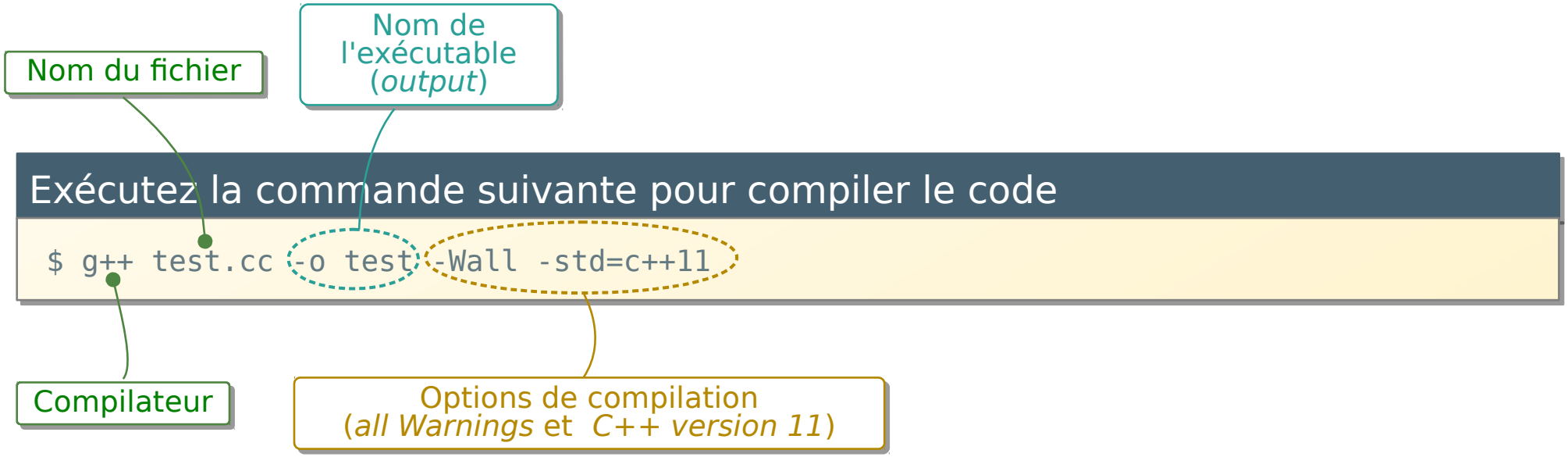
- Namespaces et opérateurs de résolution de portée

```
using namespace std;
```

- Avantage de `cout << endl;` : l'opérateur d'input est *type-safe,* ce qui n'est pas le cas de `printf();`

Compiler et exécuter le code en terminal

- Une fois dans le dossier contenant **test.cc**



Exécutez la commande suivante pour exécuter le code

```
$ ./test
```

DE LA RECHERCHE À L'INDUSTRIE

www.cea.fr

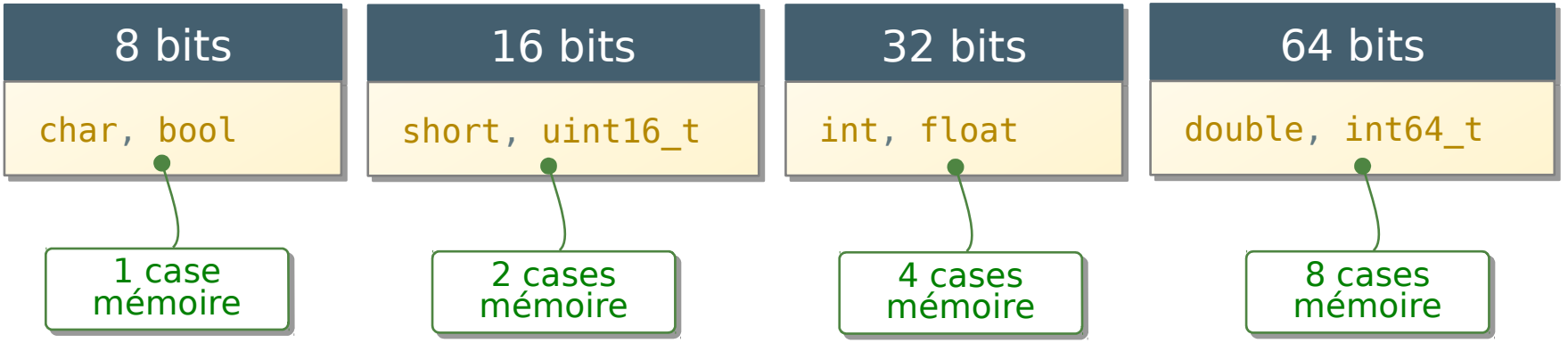
Partie 1/5
Pointeurs

Les variable C++ :

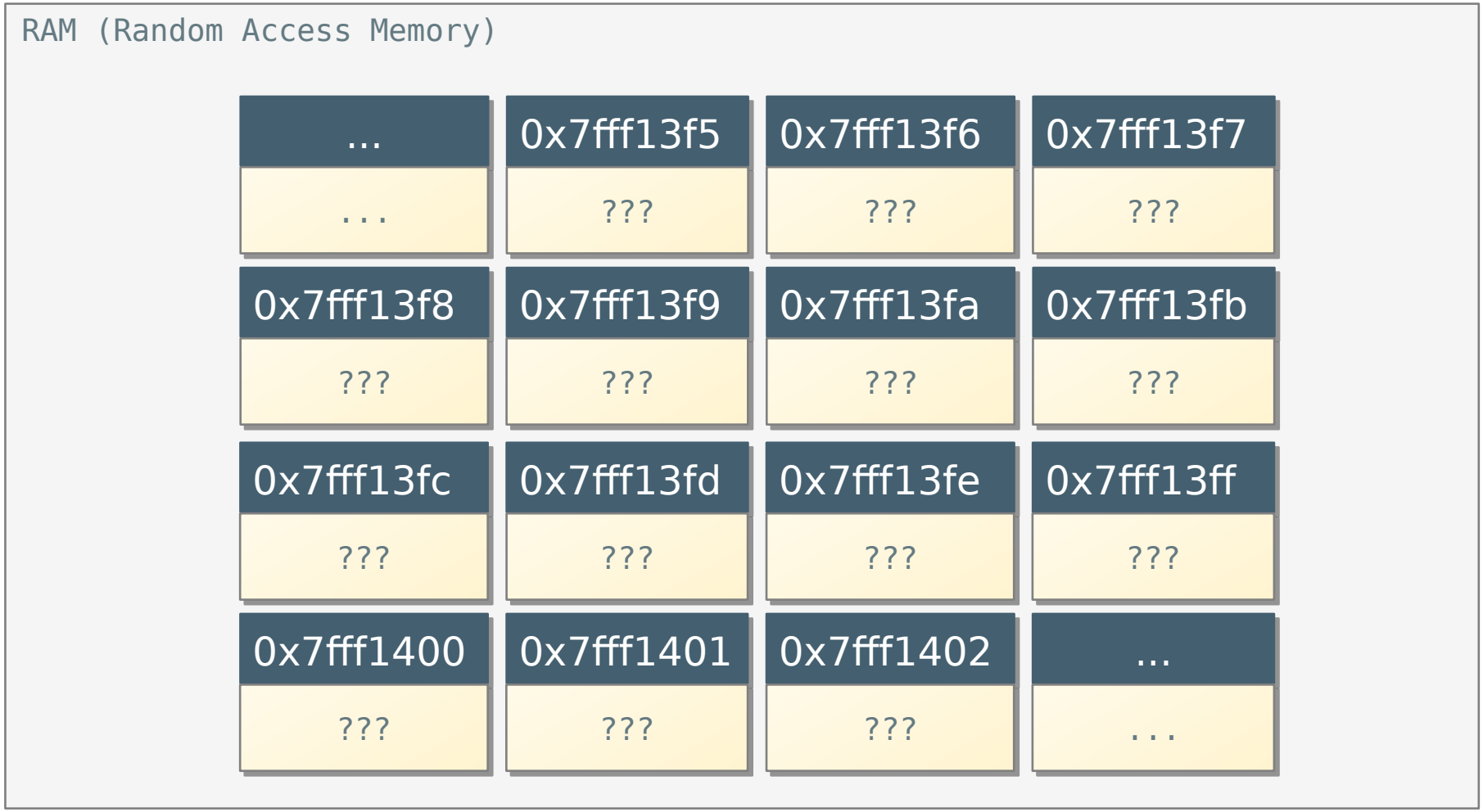
- Les variables déclarées dans le code sont stockées en RAM
- Elles sont stockées sous forme **binaire** : 110001010101
- L'unité de stockage d'un emplacement mémoire un **byte** ("baïte") :
- Depuis les années 70 les ordinateurs sont standardisées :

1 **byte** = 8 **bits** (1 octet)

- La quantité de cases mémoires occupées par une variable dépend de son type

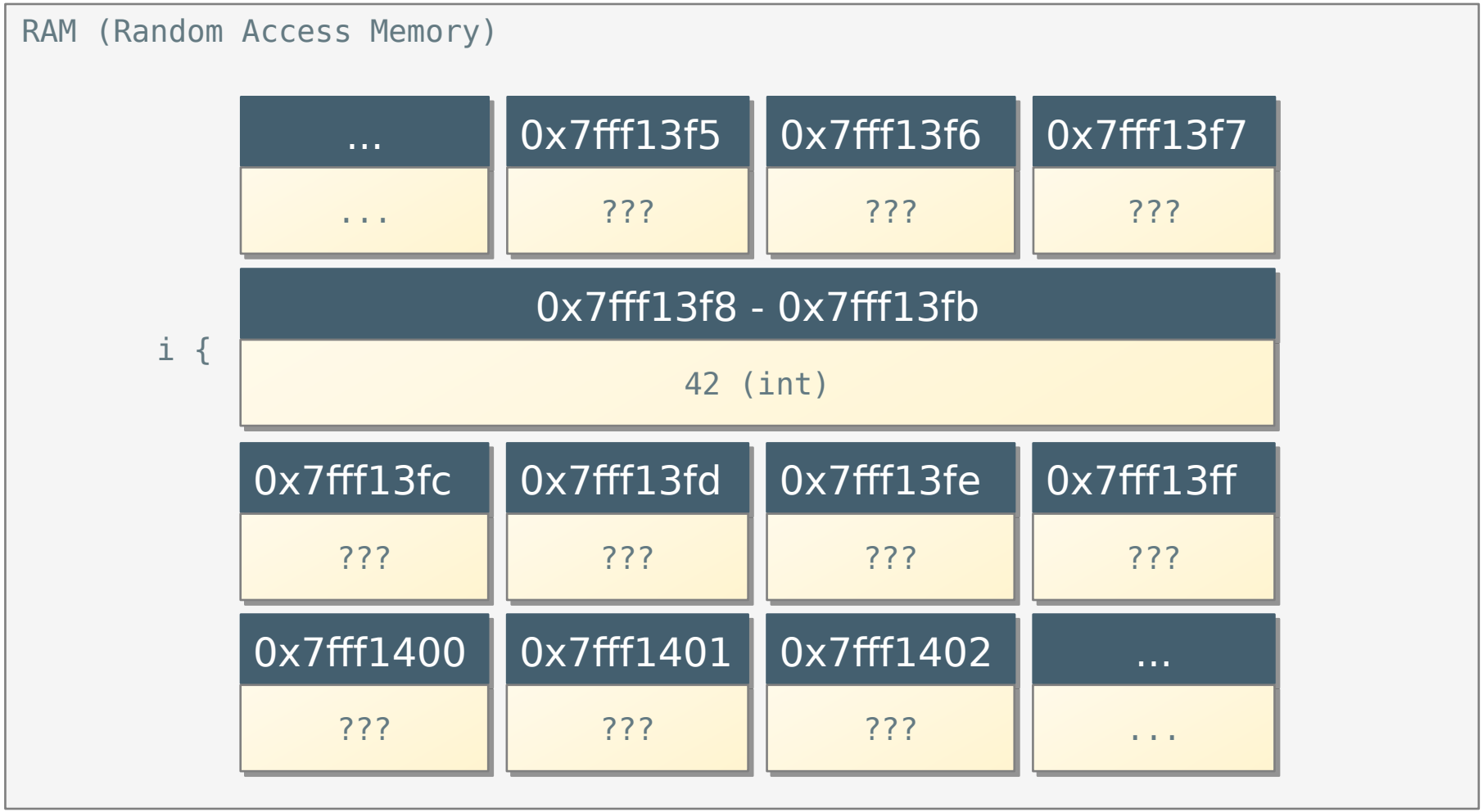


Walk down memory lane



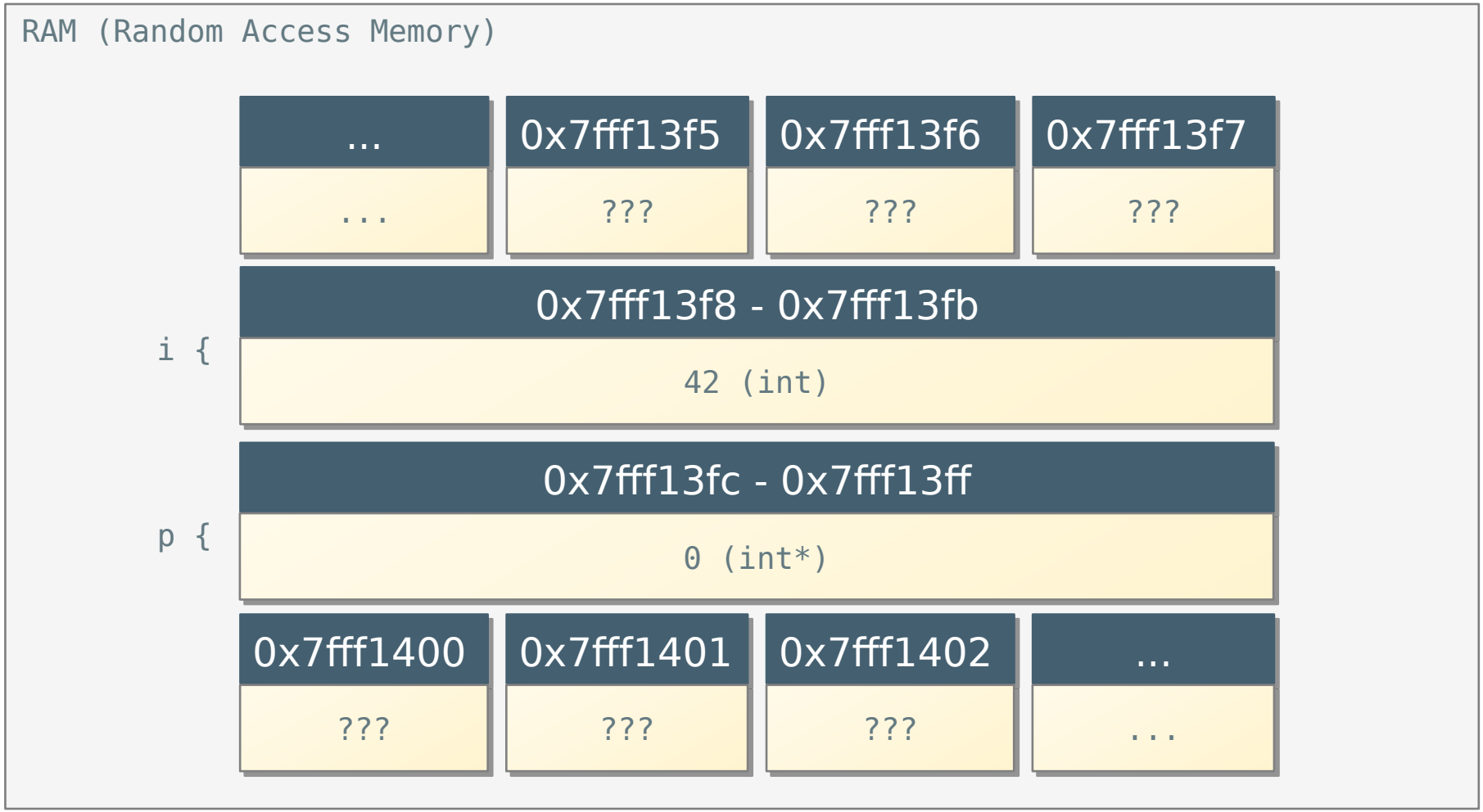
What happens if...

```
int i(42);
```



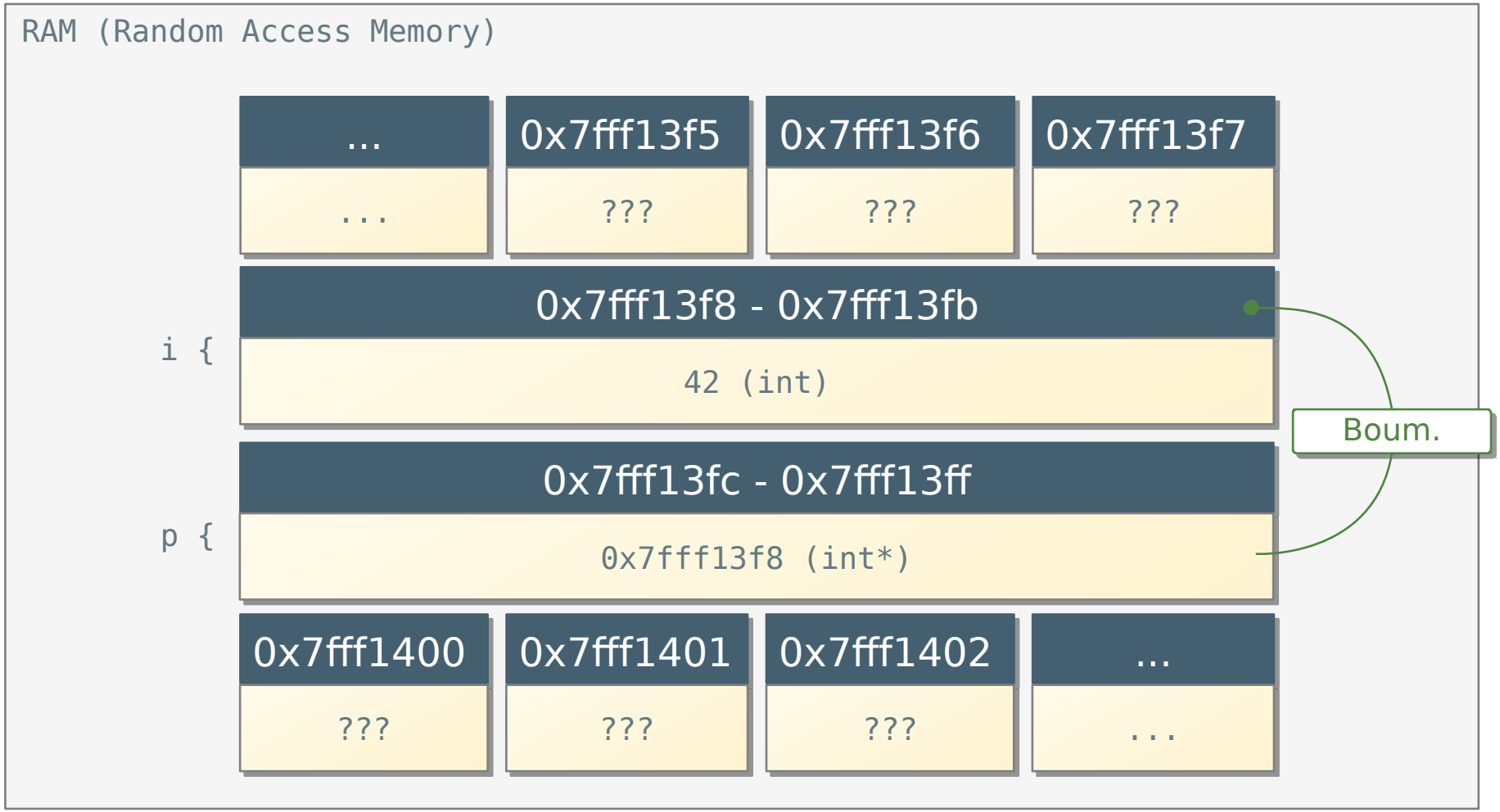
What happens if...

```
int *p(0);
```



What happens if...

```
p = &i;
```



Les variable C++ :

- Les syntaxes sont indépendantes du type !

Undefined behaviour !

type	valeur	adresse	valeur pointée
`int`	`i`	`&i`	`*i`
`int*`	`p`	`&p`	`*p`

- Testez un peu tout ça :

```
int i(42);
int *p(0);
p = &i;
```

`i == 42` `p == &i` `&i == 0x` `&p == 0x` `*i == ??` `*p == i`

DE LA RECHERCHE À L'INDUSTRIE

www.cea.fr

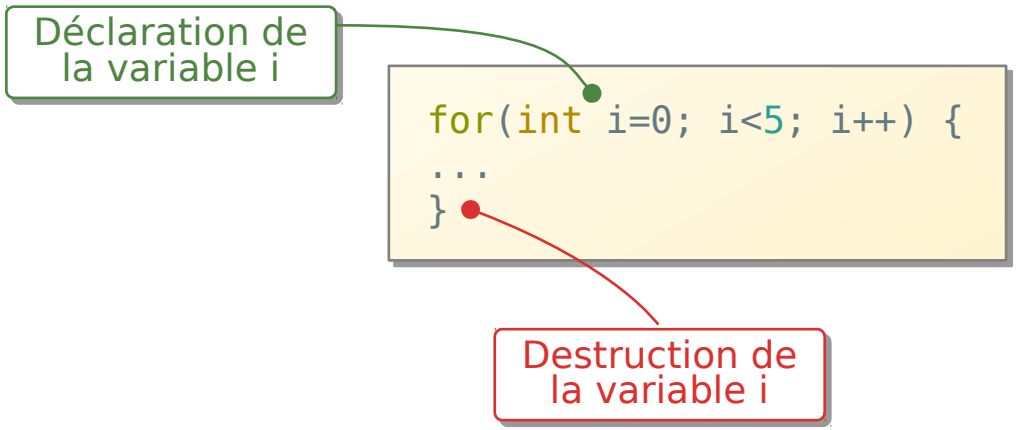
Partie 2/5

Pointeurs intelligents

Variables automatiques

Durée de vie d'une variable C++

- Les variables C++ sont dites "automatiques"
- Elles peuvent être déclarées **n'importe où** dans une fonction et pas nécessairement au début
- Les variables sont **locales** à leur bloc de déclaration.
- Elles sont **automatiquement détruite** à la sortie du bloc dans lequel elles ont été déclarées.
- Elles peuvent même être déclarées dans les instructions d'une boucle! Auquel cas elles sont détruites à la sortie de la boucle.



Variables automatiques

Gestion 'manuelle' de l'allocation

- Elle est basée sur l'utilisation de pointeurs
- La mémoire est allouée dynamiquement via l'opérateur `new`
- Le pointeur lui-même est géré automatiquement, et donc **automatiquement détruit** à la sortie du bloc dans lequel il a été déclaré.

```
int main()
{
    int *var = 0;
    var = new int;

    // ...

    delete var;
    var = 0;

    return 0;
}
```

Fuites mémoires

Hors du monde des bisounours

- La gestion manuelle de la mémoire amène d'énorme risques de **fuites mémoire**

```
int main()
{
    int *var = 0;
    var = new int;

    // ...

    delete var;
    var = 0;

    return 0;
}
```

Que se passe-t-il en cas de sortie inopinée du bloc? (exception)

Oubli de **delete**

Risque de de manipulation de cases mémoires au contenu aléatoire

Fuites mémoires

Hors du monde des bisounours

```
int main()
{
    int *var = 0;
    try {

        var = new int;

        // ...
    }
    catch(int i) {
        // take actions
        delete var;
        var = 0;
    }
    catch(...) {
        // take actions
        delete var;
        var = 0;
    }
    delete var;
    var = 0;

    return 0;
}
```

Attraper toutes les exceptions possibles dans un try-catch et libérer la mémoire dans tout les cas

Sans oublier la sortie 'naturelle' de bloc!

Smart pointers

C++ 11 et les pointeurs intelligents

- Les pointeurs "nus" sont encapsulés dans des **classes** qui gèrent elles-même les ressources
- Plus fiable, plus facilement maintenable, plus compréhensible, plus sécurisé et plus simple
- En pratique:
  - Le pointeur devient un objet **propriétaire** de la ressource
  - La libération de la mémoire est gérée par le smart pointer
  - La zone mémoire gérée par le smart pointer est allouée avec l'opérateur **new**

```
unique_ptr<int> ptr(new int(5)) ;
cout << *ptr << endl;
```

Smart pointers

C++ 11 et les pointeurs intelligents

```
unique_ptr<int> ptr = make_unique<int>(5);
```

- Un unique_ptr est **propriétaire unique** du 'vrai' pointeur
- L'objet géré est détruit automatiquement dès que le unique_ptr libère l'objet (destruction ou changement de valeur)

```
shared_ptr<int> ptr1 = make_shared<int>(5);
shared_ptr<int> ptr2(ptr1) ;
cout << *ptr1 << " " << *ptr2<< endl;
```

- Un shared_ptr est **co-propriétaire** du 'vrai' pointeur
- L'objet géré est détruit automatiquement lorsque tous les shared_ptr partageant la propriété le libèrent

Smart pointers

C++ 11 et les pointeurs intelligents: Mefiat!

Quelles sont les valeurs pointées par a et b suite aux commandes suivantes?

```
int *a = new int(5) ;
if(*a>0)
   unique_ptr<int> b(a) ;
```

- Il faut rajouter `#include <memory>` en en-tête

Smart pointers

C++ 11 et les pointeurs intelligents: Mefiat!

Quelles sont les valeurs pointées par a et b suite aux commandes suivantes?

```
int *a = new int(5) ;
if(*a>0)
   unique_ptr<int> b(a) ;
```

- Il faut rajouter `#include <memory>` en en-tête
- Explications:
  - Les unique_ptr détruisent les objets gérés dès la sortie de bloc
  - La destruction se fait **sans tenir compte** d'autres pointeurs éventuels
  - C'est la responsabilité du programmeur de s'assurer que chaque unique_ptr est effectivement unique

Smart pointers

C++ 11 et les pointeurs intelligents: Mefiat!

Modifiez les déclarations de pointeurs pour que *a = 5 en fin de programme

```
int *a = new int(5) ;
if(*a>0)
   unique_ptr<int> b(a) ;
```

- Attention aux pièges:
  - Les unique_ptr détruisent les objets gérés **sans tenir compte** d'autres pointeurs éventuels
  - Les shared_ptr **doivent** être déclarés en tant que copies de shared_ptr pour garantir le partage de propriété

DE LA RECHERCHE À L'INDUSTRIE

www.cea.fr

Partie 2/5

Vecteurs et itérateurs

Vecteurs et itérateurs

Tableaux

- Principe
  - Un tableau est une succession de plusieurs valeurs contenue dans une seule variable

```
int tab[5] = {1, 2, 3, 4, 5};
tab[0] = 1;
tab[1] = 2;
tab[2] = 3;
tab[3] = 4;
tab[4] = 5;
```

- En mémoire
  - Le nom d'un tableau est un **pointeur** vers son premier élément
  - Les éléments d'un tableau sont stockés en mémoire dans des zones contigüe

Vecteurs et itérateurs

Tableaux

- Tableaux statiques
  - La taille du tableau est donnée à la création de l'objet et ne peut pas être changée

```
int tab[5];
```

- Tableaux dynamiques
  - La taille du tableau est définie indépendamment de l'instanciation de l'objet
  - La destruction du tableau est **sous la responsabilité du développeur**

```
int *tab;
tab = new int[5];
…
delete[] tab;
```

Vecteurs et itérateurs

Vecteurs

- Syntaxe(s) et superpouvoirs

```
vector<type> name(size);
```

```
vector<int> vec;
vector<int> vec(5);
vector<int> vec(5,0);
```

Vecteur vide

Vecteur de taille 5

[0,0,0,0,0]

`vec.push_back(1);` → Ajoute un élément (**1**) à la fin du vecteur **vec**

`vec.pop_back();` → Supprime le dernier élément du vecteur

`vec.size();` → Renvoie la taille du vecteur

Itérateurs

Testez le résultat du code suivant

```
#include <iostream>

using namespace std;

int main()
{
    int tab[5] = {1,2,3,4,5};
    for(int *it=tab; it!=tab+5; ++it) {
        cout << *it << endl;
    }
    return 0;
}
```

Itérateurs: explications

```
for(int *it=tab; it!=tab+5; it++)
```

- Le nom d'un tableau (statique ou new) est un **pointeur** vers son premier élément
- Les éléments d'un tableau sont stockés en mémoire dans des zones contigües
- L'opération '+' incrémente le pointeur → on avance d'une case en mémoire

```
*(tab+3)
```
<=>
```
tab[3]
```

- Pourquoi '!=' et pas '<' ?

Vecteurs et itérateurs

Itérateurs: pour les vecteurs!

- Les vecteurs possèdent des itérateurs internes dépendant du type de vecteur:

```
vector<int>::iterator it;
```

- Un vecteur possède une méthode renvoyant un itérateurs pointant sur son premier élément:

```
vec.begin();
```

- Et également une méthode pour obtenir un itérateur pointant sur le premier élément hors du vecteur:

```
vec.end();
```

Vecteurs et itérateurs

Exercice: vecteurs & itérateurs

- Écrire un code effectuant les tâches suivantes:
  - Lecture de chiffres entrés par clavier durant l'exécution
  - Enregistrement de ces chiffres dans un **vecteur**
  - Affichage de la liste des sommes de chaque élément avec l'élément le suivant

- Contraintes:
  - Utilisez des **itérateurs** pour vos boucles
  - Pensez à rajouter `#include <vector>` en en-tête

- Niveau 2:
  - Affichez plutôt les sommes du premier et du dernier élément, du second et de l'avant-dernier, du troisième et de l'antépénultième, etc.

Exercice: vecteurs & itérateurs

Exemple de solution – partie lecture

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    int entries, entry;
    vector<int> vec;

    cout<<"Size of vector: ";
    cin>>entries;

    cout<<"Content: "<<endl;
    while(vec.size()<entries) {
        cin>>entry;
        vec.push_back(entry);
    }

    return 0;
}
```

Exercice: vecteurs & itérateurs

Exemple de solution – boucle "niveau 1"

```
for(auto it=vec.begin(); it!=vec.end(); ++it) {
    if(it<vec.end()-1)
        cout<<*it+*(it+1)<<" - ";
    else
        cout<<*it<<endl;
}
```

Exercice: vecteurs & itérateurs

Exemple de solution – boucle "niveau 1"

```
for(auto it=vec.begin(); it!=vec.end(); ++it) {
    if(it<vec.end()-1)
        cout<<*it+*(it+1)<<" - ";
    else
        cout<<*it<<endl;
}
```

Exemple de solution – boucle "niveau 2"

```
for(auto it1=vec.begin(), it2=vec.end(); it1<=it2; ++it1, --it2) {
    if(it1+1 < it2)
        cout<<*it1+*it2<<" - ";
    else
        cout<<*it1+*it2<<endl;
}
```