Programmation C++

2ème année Informatique

J. Allali
julien.allali@labri.fr

ENSEIRB

Prog. C++

Plan

Plan

Plan

Plan

Historique (wikipedia)

- 198x: Bjarne Stroustrup (AT&T Bell): “C with classes”:
  - Classes
  - fonctions virtuelles
  - surcharge d’opérateurs
  - héritage multiple
  - ...

Historique (wikipedia)

- 198x: Bjarne Stroustrup (AT&T Bell): “C with classes”:
  - Classes
  - fonctions virtuelles
  - surcharge d’opérateurs
  - héritage multiple
  - ...

Normes:

- 1998: Normalisation du C++ par l’ISO (International Organization for Standardization) et ANSI ISO/CEI 14882:1998

Historique (wikipedia)

- 198x: Bjarne Stroustrup (AT&T Bell): “C with classes”:
  - Classes
  - fonctions virtuelles
  - surcharge d’opérateurs
  - héritage multiple
  - ...

Normes:

- 1998: Normalisation du C++ par l’ISO (International Organization for Standardization) et ANSI ISO/CEI 14882:1998
- 2003: Dernière version 14882:2003

C (++)

Le C++ est un language impératif, *orienté objets*:

C (++)

Le C++ est un language impératif, *orienté objets*:

L'ajout de fonctionnalités permettant la mise en oeuvre de concepts objets dans le language C:

- l'objet: attributs(données internes) + méthodes (comportements), encapsulation.
- Typage des objets.
- Polymorphisme: un objet peut avoir plus d'un type.
- redéfinition.
- classe: description et génération des objets.

C (++)

Le C++ est un language impératif, *orienté objets*:

L'ajout de fonctionnalités permettant la mise en oeuvre de concepts objets dans le language C:

- l'objet: attributs(données internes) + méthodes (comportements), encapsulation.
- Typage des objets.
- Polymorphisme: un objet peut avoir plus d'un type.
- redéfinition.
- classe: description et génération des objets.

Il permet aussi la *programmation générique*: écriture de fonctions (et d'objets) indépendantes du type de ces arguments. C'est l'idée du "*code à trous*".

Incompatibilité en le C et le C++

Source C qui ne compile (invalidité syntaxique) pas en C++:

- Si le source contient un des nouveaux mots clés du C++

- Le C autorise la conversion implicite de void * en n'importe quel autre type de pointeur:

exemple

```
int *i=malloc(4);
```

Régles typographique:

Quelques régles typographiques:

- **mots clés du language**
- les lexèmes de la grammaire
- *Les commentaires*
- **attention**
- important
- attributs et methodes

Plan

Allocation automatique

L'allocation automatique se fait dans la pile.

Allocation automatique

L'allocation automatique se fait dans la pile.

Exemple

```
int i;
```

La variable i est allouée dans la pile automatiquement.

Allocation automatique

L'allocation automatique se fait dans la pile.

Exemple

```
int i;
```

La variable i est allouée dans la pile automatiquement.

Ainsi, &i correspond à une adresse dans la pile à laquelle sizeof(int) octets sont réservés.

Allocation automatique

L'allocation automatique se fait dans la pile.

Exemple

```
int i;
```

La variable i est allouée dans la pile automatiquement.

Ainsi, &i correspond à une adresse dans la pile à laquelle sizeof(int) octets sont réservés.

À la sortie du bloc dans lequel est déclaré i, il y a dépilement et donc l'adresse &i correspond à une zone mémoire qui n'est plus réservée.

Allocation dynamique

L'allocation dynamique en C++ se fait à l'aide de l'opérateur **new**.

On distingue deux types d'allocation dynamique: l'allocation d'un objet ou l'allocation d'un tableau d'objets.

Exemple

new int
new int [10]

Dans le premier cas, on réserve sizeof(int) octets dans le tas, l'opérateur **new** retourne l'adresse de début de cette zone.

Dans le premier cas, on réserve sizeof(int)*10 octets dans le tas, l'opérateur **new** retourne l'adresse de début de cette zone.

Ces zones mémoires sont réservées tant que l'on n'a pas indiqué explicitement que l'on souhaitait les libérer avec l'opérateur **delete**.

Allocation dynamique

L'allocation dynamique en C++ se fait à l'aide de l'opérateur **new**.

On distingue deux types d'allocation dynamique: l'allocation d'un objet ou l'allocation d'un tableau d'objets.

Exemple

new int
new int [10]

Dans le premier cas, on réserve sizeof(int) octets dans le tas, l'opérateur **new** retourne l'adresse de début de cette zone.

Dans le premier cas, on réserve sizeof(int)*10 octets dans le tas, l'opérateur **new** retourne l'adresse de début de cette zone.

Ces zones mémoires sont réservées tant que l'on n'a pas indiqué explicitement que l'on souhaitait les libérer avec l'opérateur **delete**.

Allocation dynamique

L'allocation dynamique en C++ se fait à l'aide de l'opérateur **new**.

On distingue deux types d'allocation dynamique: l'allocation d'un objet ou l'allocation d'un tableau d'objets.

Exemple

new int
new int [10]

Dans le premier cas, on réserve sizeof(int) octets dans le tas, l'opérateur **new** retourne l'adresse de début de cette zone.

Dans le premier cas, on réserve sizeof(int)*10 octets dans le tas, l'opérateur **new** retourne l'adresse de début de cette zone.

Ces zones mémoires sont réservées tant que l'on n'a pas indiqué explicitement que l'on souhaitait les libérer avec l'opérateur **delete**.

Allocation dynamique

L'allocation dynamique en C++ se fait à l'aide de l'opérateur **new**.

On distingue deux types d'allocation dynamique: l'allocation d'un objet ou l'allocation d'un tableau d'objets.

Exemple

new int
new int [10]

Dans le premier cas, on réserve sizeof(int) octets dans le tas, l'opérateur **new** retourne l'adresse de début de cette zone.

Dans le premier cas, on réserve sizeof(int)*10 octets dans le tas, l'opérateur **new** retourne l'adresse de début de cette zone.

Ces zones mémoires sont réservées tant que l'on n'a pas indiqué explicitement que l'on souhaitait les libérer avec l'opérateur **delete**.

Allocation dynamique

L'allocation dynamique en C++ se fait à l'aide de l'opérateur **new**.

On distingue deux types d'allocation dynamique: l'allocation d'un objet ou l'allocation d'un tableau d'objets.

Exemple

new int
new int [10]

Dans le premier cas, on réserve sizeof(int) octets dans le tas, l'opérateur **new** retourne l'adresse de début de cette zone.

Dans le premier cas, on réserve sizeof(int)*10 octets dans le tas, l'opérateur **new** retourne l'adresse de début de cette zone.

Ces zones mémoires sont réservées tant que l'on n'a pas indiqué explicitement que l'on souhaitait les libérer avec l'opérateur **delete**.

Allocation dynamique: libération

Il faut libérer la mémoire allouée avec **new** en utilisant **delete**.

Il faut libérer la mémoire allouée avec **new type[]** en utilisant **delete[]**.

exemple

```
delete (new int);
delete [] (new int [10]);
```

Une fois l'opérateur **delete** appelé, la mémoire qui était réservée cet adresse ne l'est plus.

Allocation dynamique: libération

Il faut libérer la mémoire allouée avec **new** en utilisant **delete**.

Il faut libérer la mémoire allouée avec **new type[]** en utilisant **delete[]**.

exemple

```
delete (new int);
delete [] (new int [10]);
```

Une fois l'opérateur **delete** appelé, la mémoire qui était réservée cet adresse ne l'est plus.

Allocation dynamique: libération

Il faut libérer la mémoire allouée avec **new** en utilisant **delete**.

Il faut libérer la mémoire allouée avec **new type[]** en utilisant **delete[]**.

exemple

```
delete (new int);
delete [] (new int [10]);
```

Une fois l'opérateur **delete** appelé, la mémoire qui était réservée cet adresse ne l'est plus.

Allocation dynamique: libération

Il faut libérer la mémoire allouée avec **new** en utilisant **delete**.

Il faut libérer la mémoire allouée avec **new type[]** en utilisant **delete[]**.

exemple

```
delete (new int);
delete [] (new int [10]);
```

Une fois l'opérateur **delete** appelé, la mémoire qui était réservée cet adresse ne l'est plus.

Allocation: exemple complet

exemple complet

```
int *i = new int;
int *t = new int[10];
delete i;
delete [] t;
```

Allocation: exemple complet

exemple complet

```
int *i = new int;
int *t = new int[10];
delete i;
delete [] t;
```

Question: combien y-a-t-il d'allocations effectuées dans cet exemple?

Allocation: exemple complet

exemple complet

```
int *i = new int;
int *t = new int[10];
delete i;
delete [] t;
```

Question: combien y-a-t-il d'allocations effectuées dans cet exemple?

Réponse: 4, 2 automatiques et 2 dynamiques.

Allocation: exemple complet

exemple complet

```
int *i = new int;
int *t = new int[10];
delete i;
delete [] t;
```

Question: combien y-a-t-il d'allocations effectuées dans cet exemple?

Réponse: 4, 2 automatiques et 2 dynamiques.

En effet, il y a deux allocations automatiques de sizeof(int *) pour les variables i et t

Allocation: exemple complet

exemple complet

```
int *i = new int;
int *t = new int[10];
delete i;
delete [] t;
```

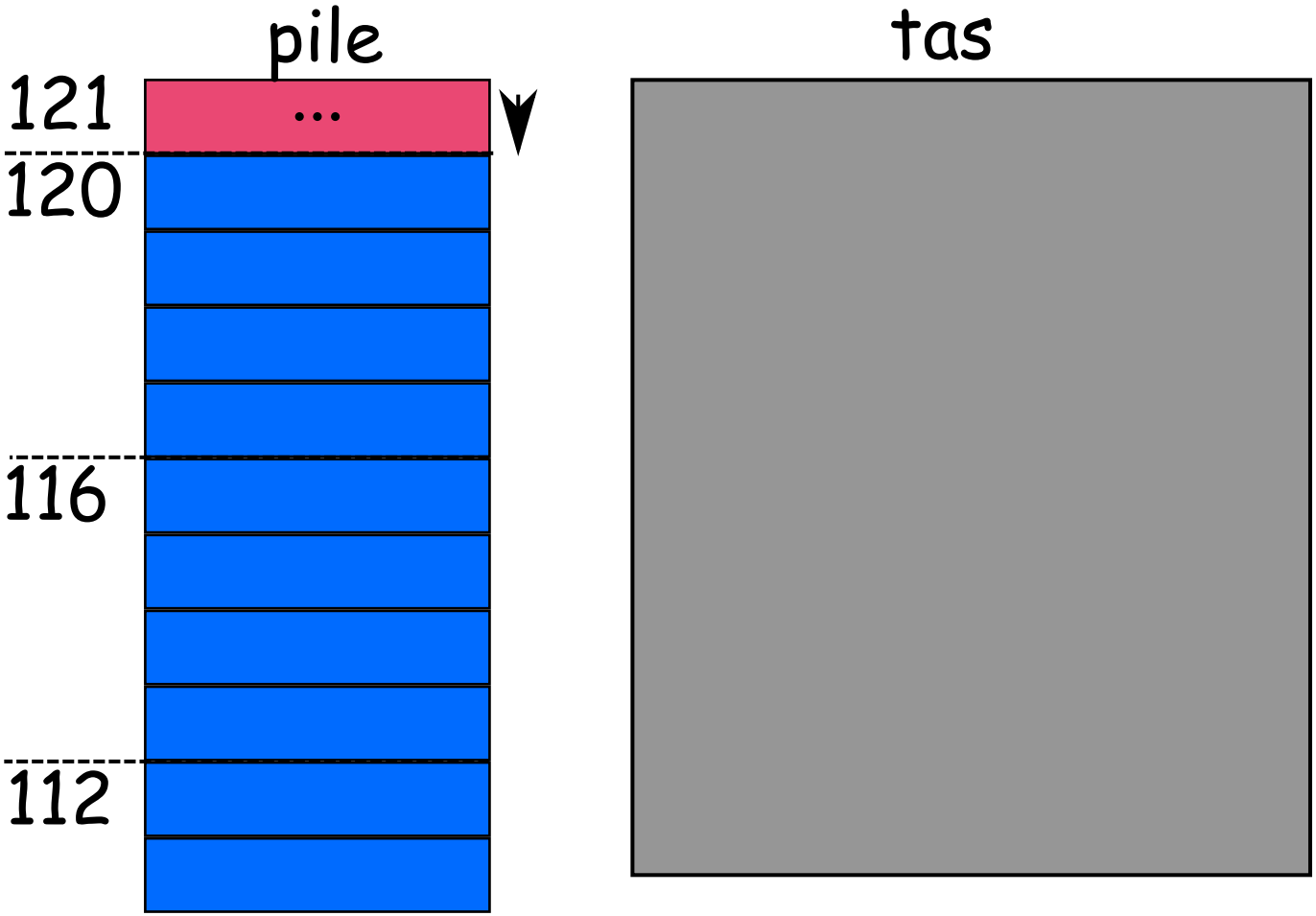
Question: combien y-a-t-il d'allocations effectuées dans cet exemple?

Réponse: 4, 2 automatiques et 2 dynamiques.

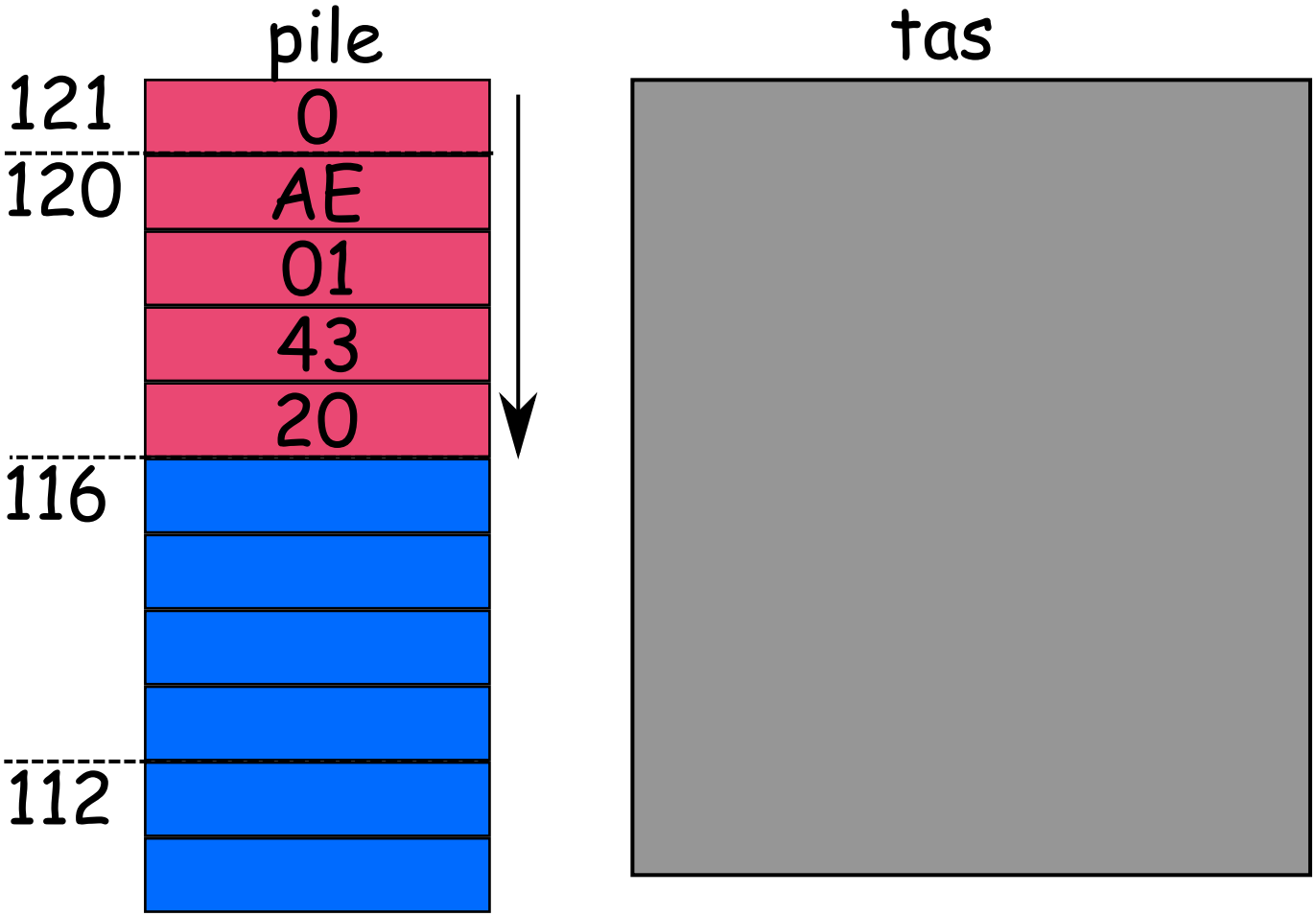
En effet, il y a deux allocations automatiques de sizeof(int *) pour les variables i et t

et deux allocations dynamiques effectuées avec **new** et dont les adresses de début de zones mémoires sont conservées dans i et t.

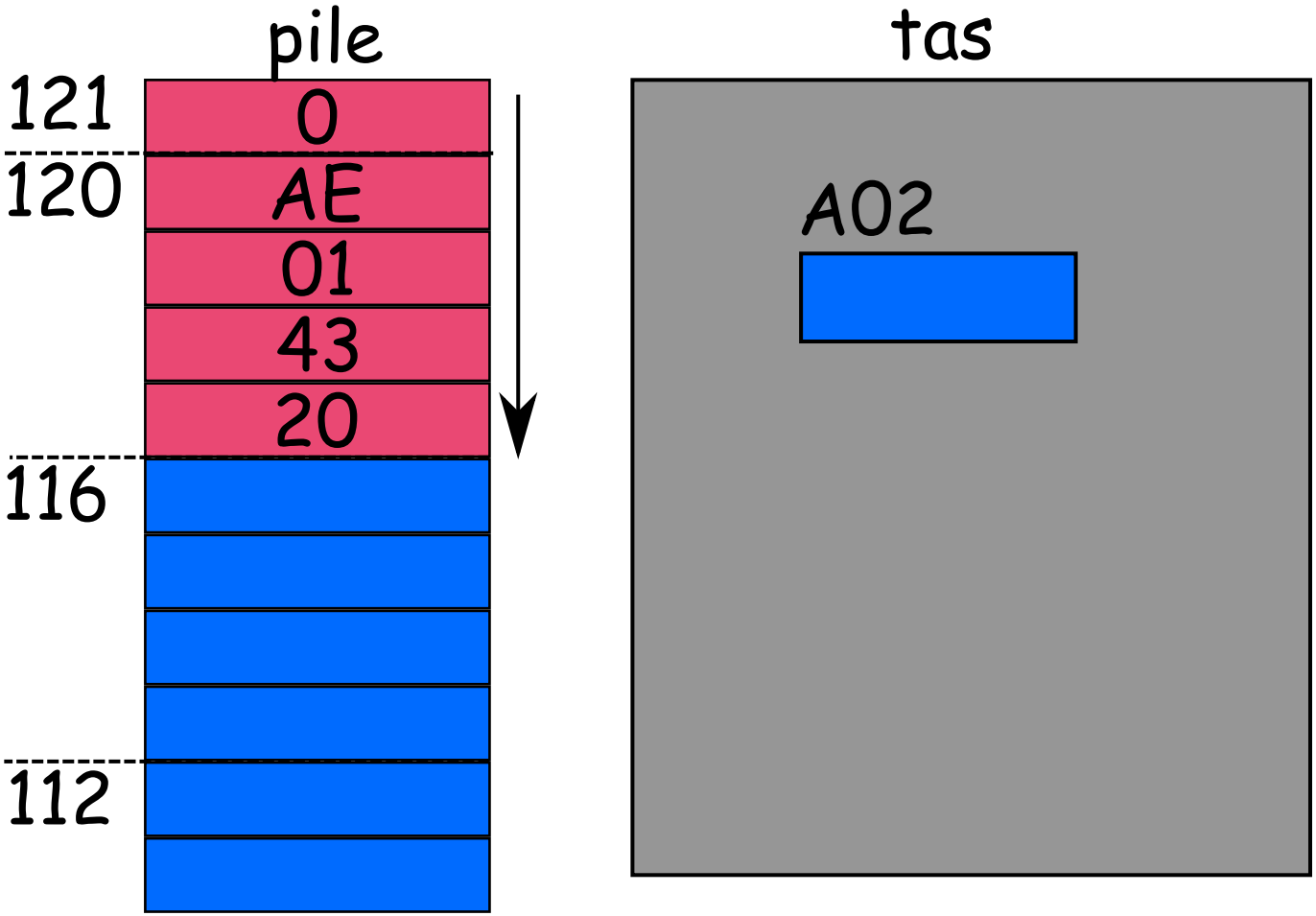
Allocation: exemple complet



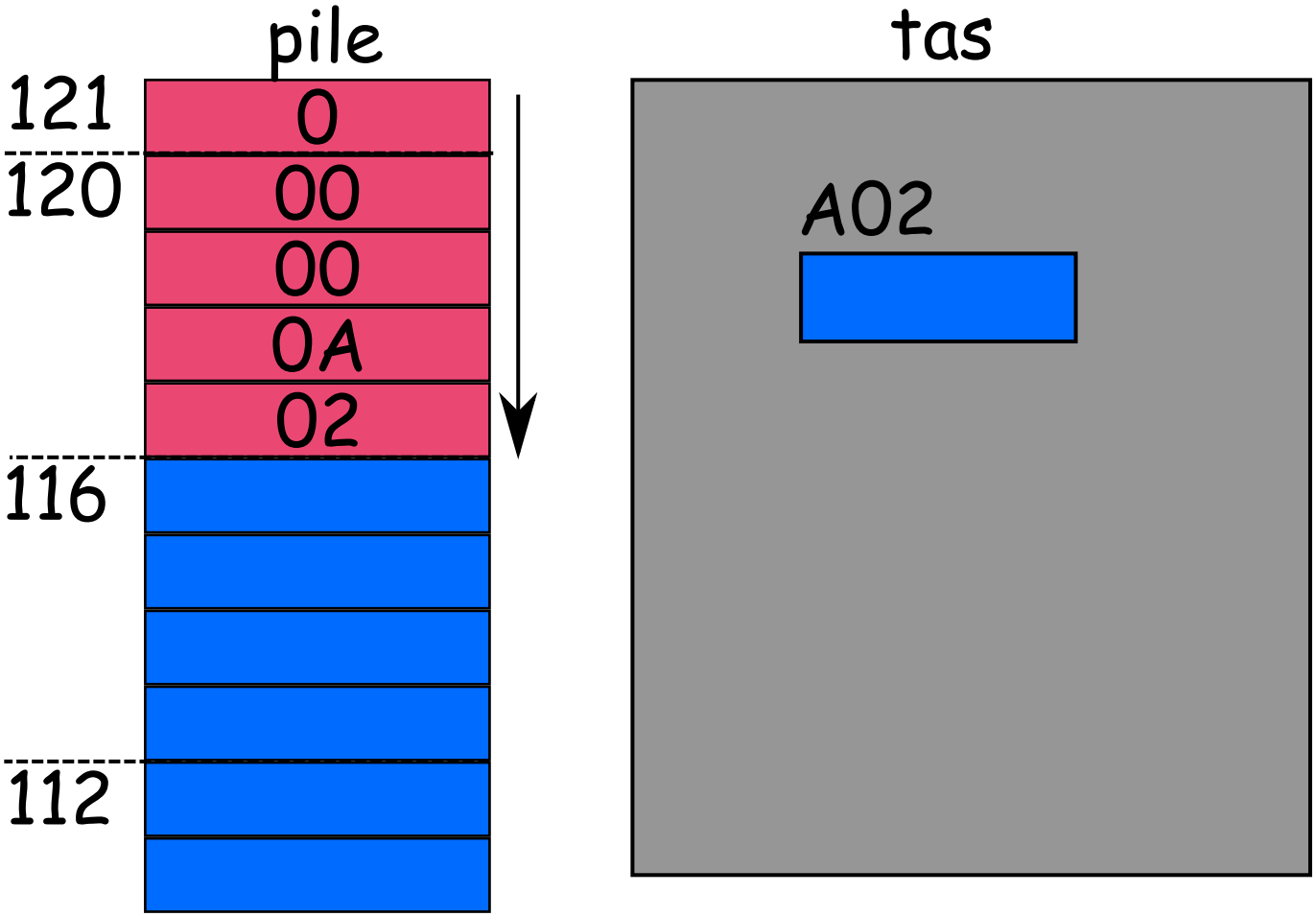
Allocation: exemple complet



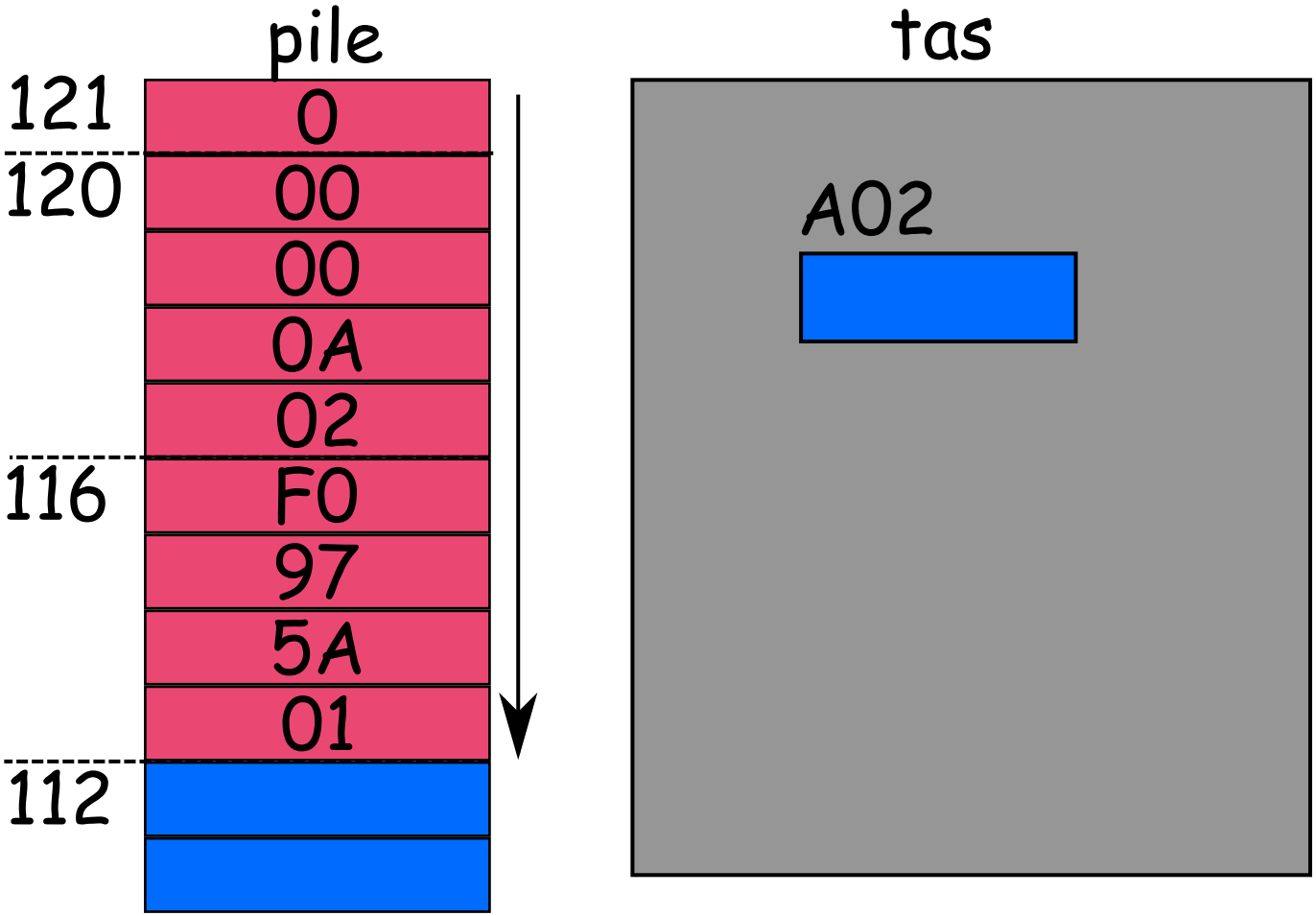
Allocation: exemple complet



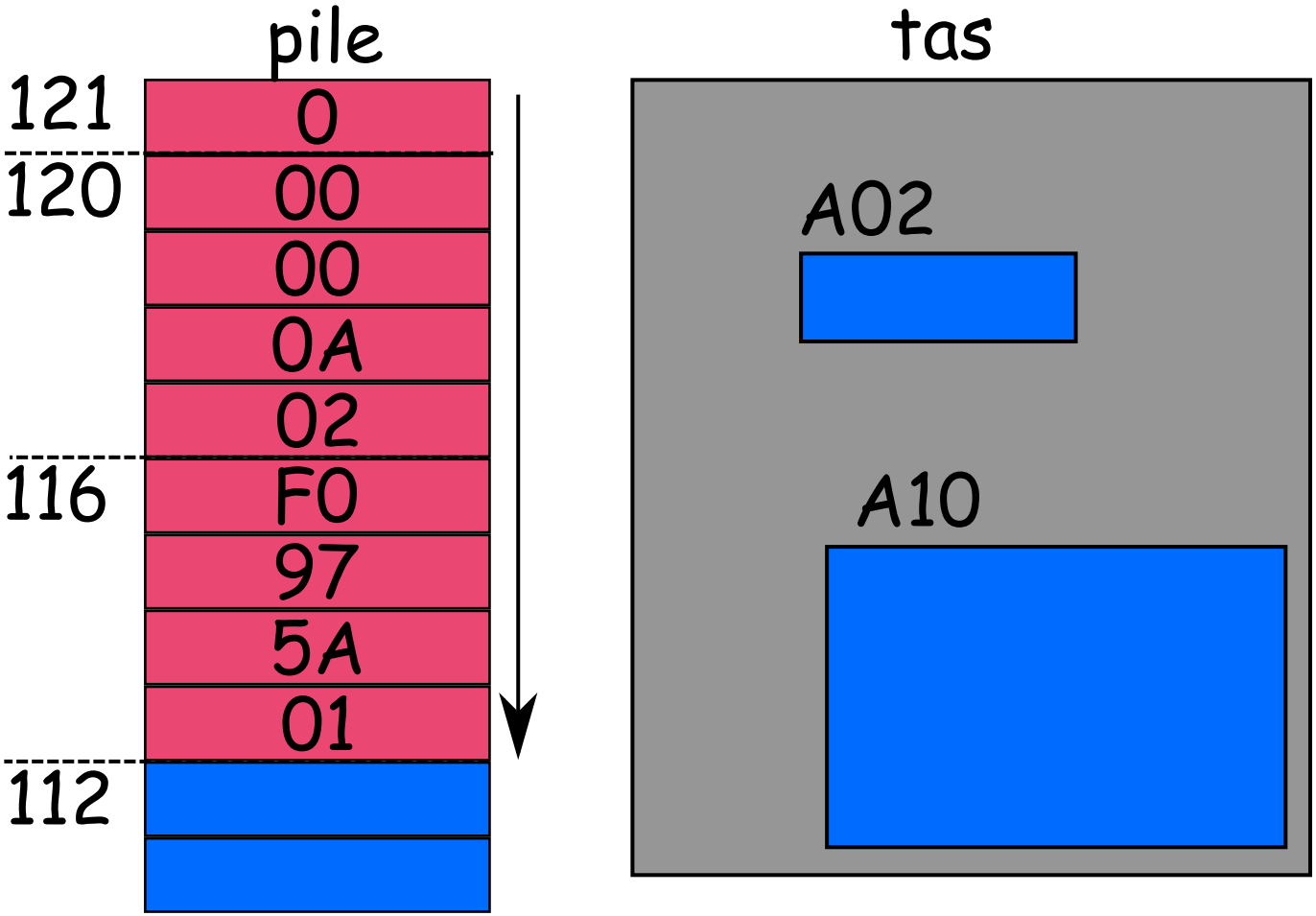
Allocation: exemple complet



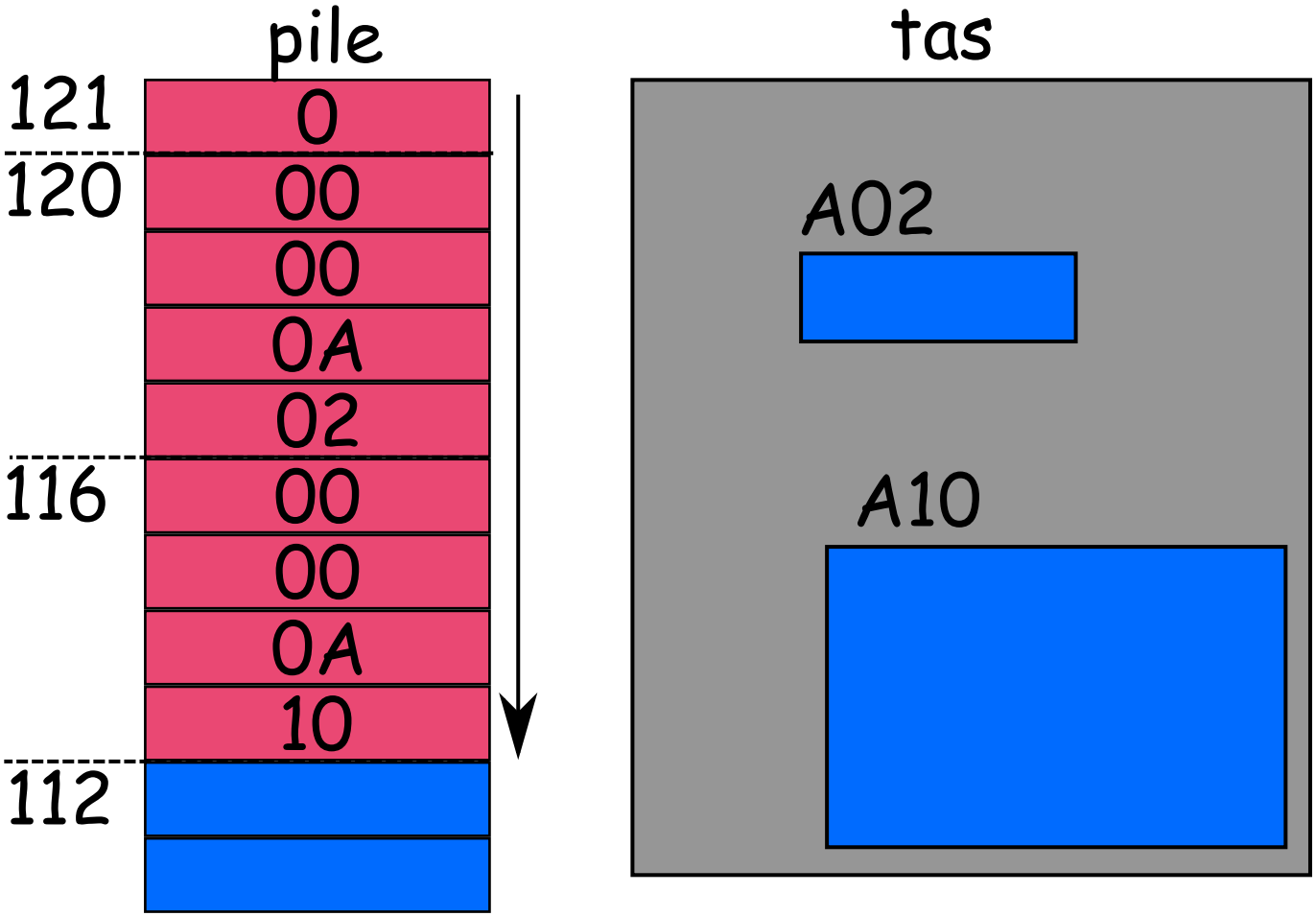
Allocation: exemple complet



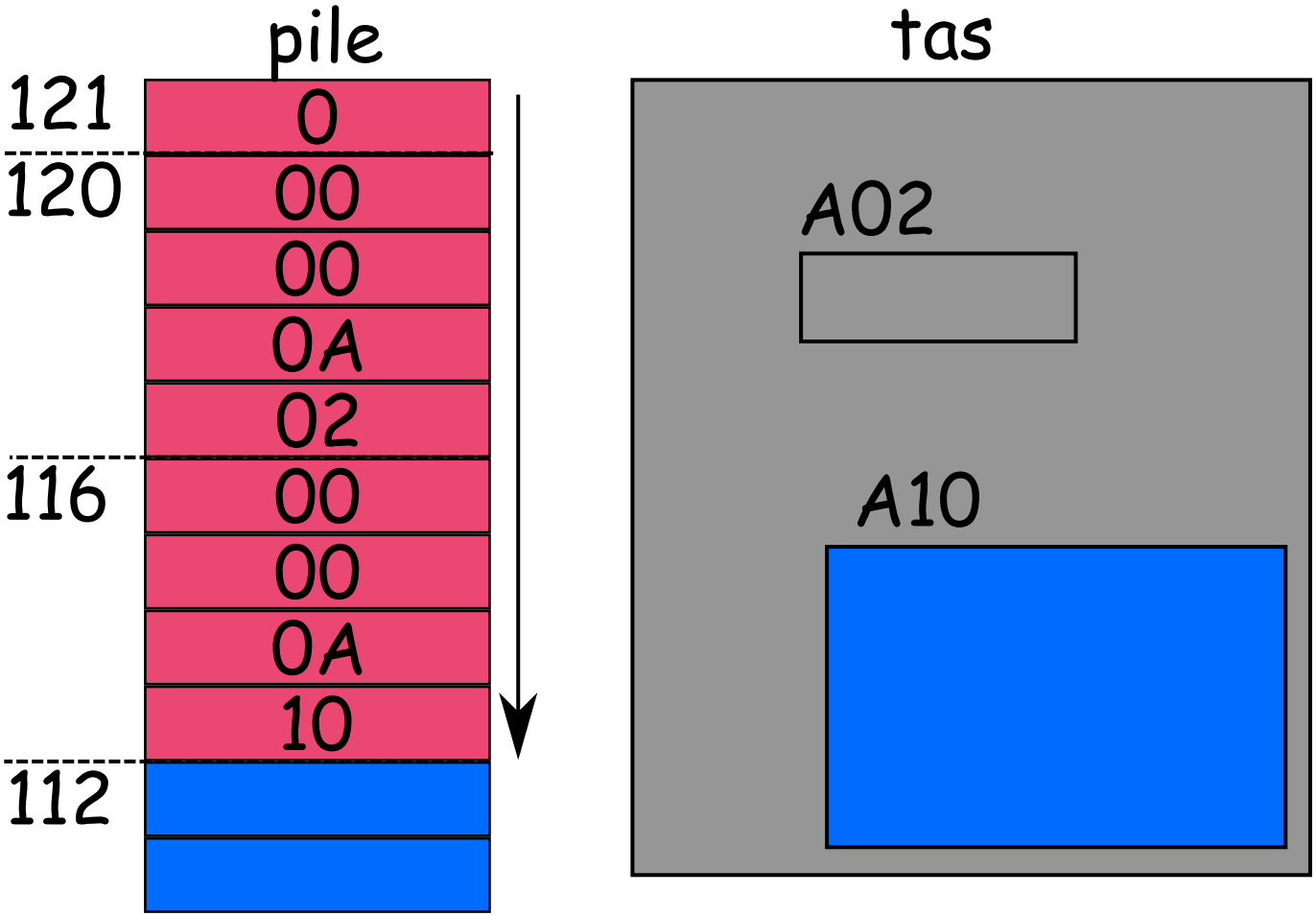
Allocation: exemple complet



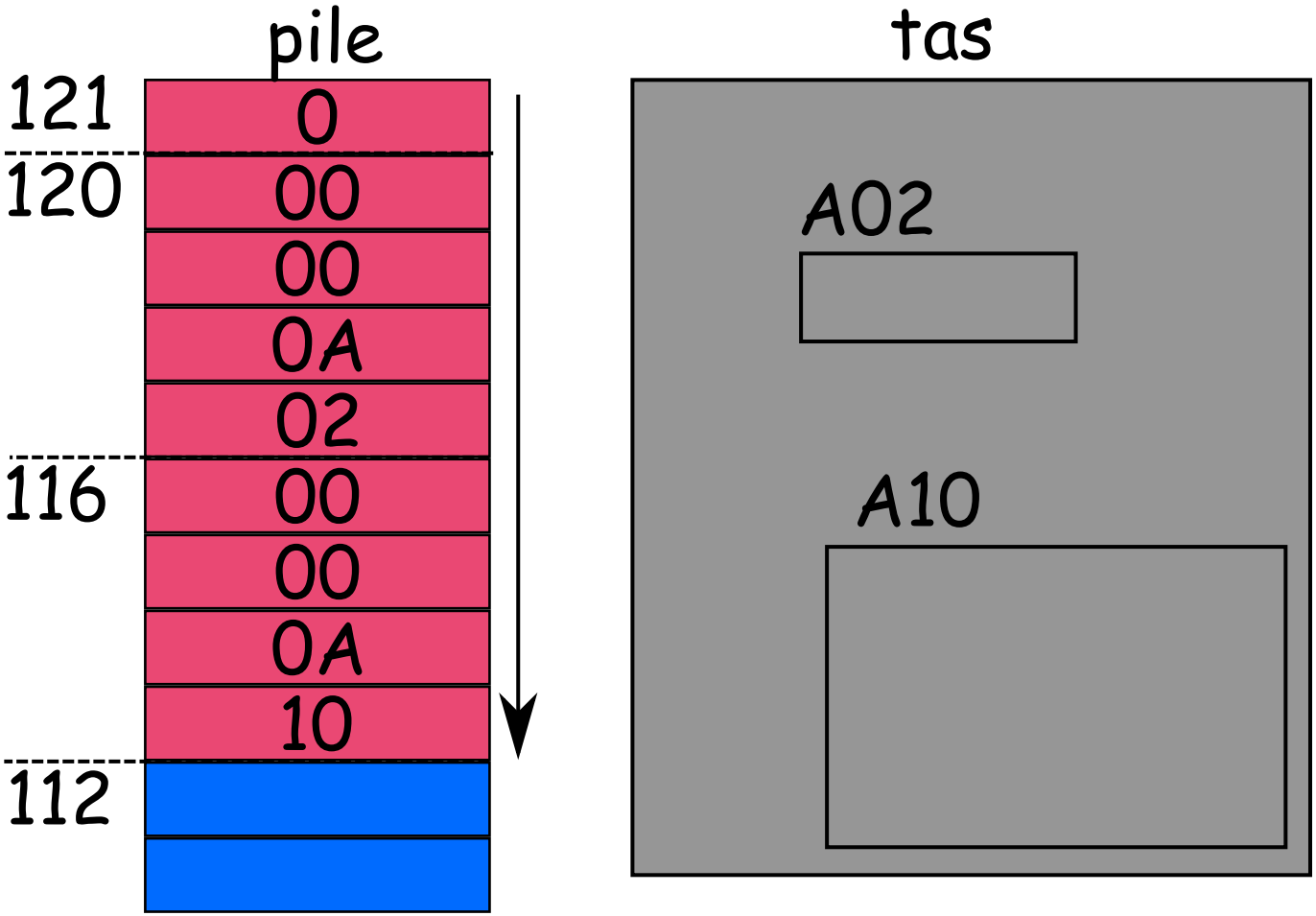
Allocation: exemple complet



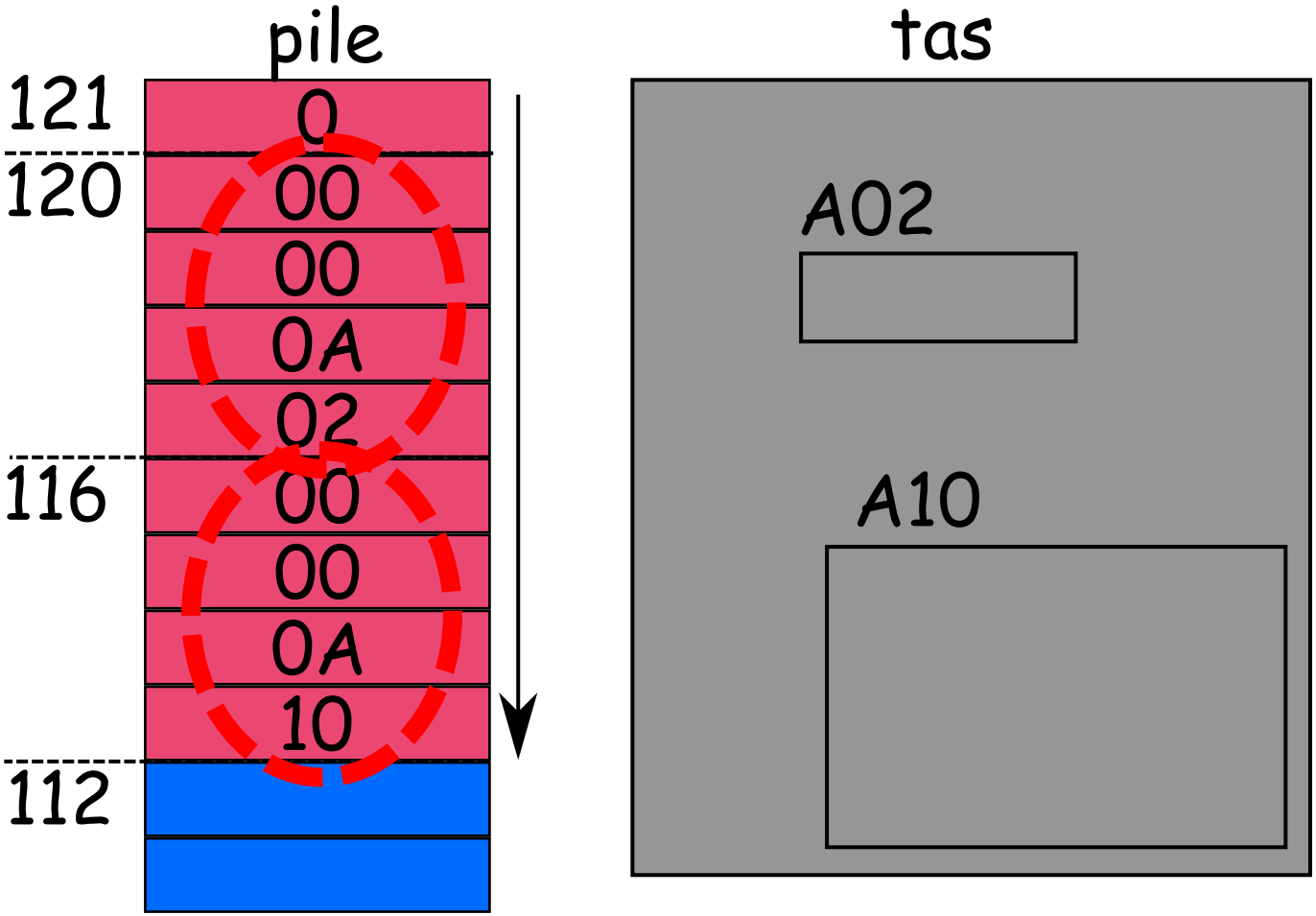
Allocation: exemple complet



Allocation: exemple complet



Allocation: exemple complet



Allocation: exemple complet

exemple complet

```
int *i = new int;
int *t = new int[10];
delete i;
i=NULL;
delete [] t;
t=NULL;
```

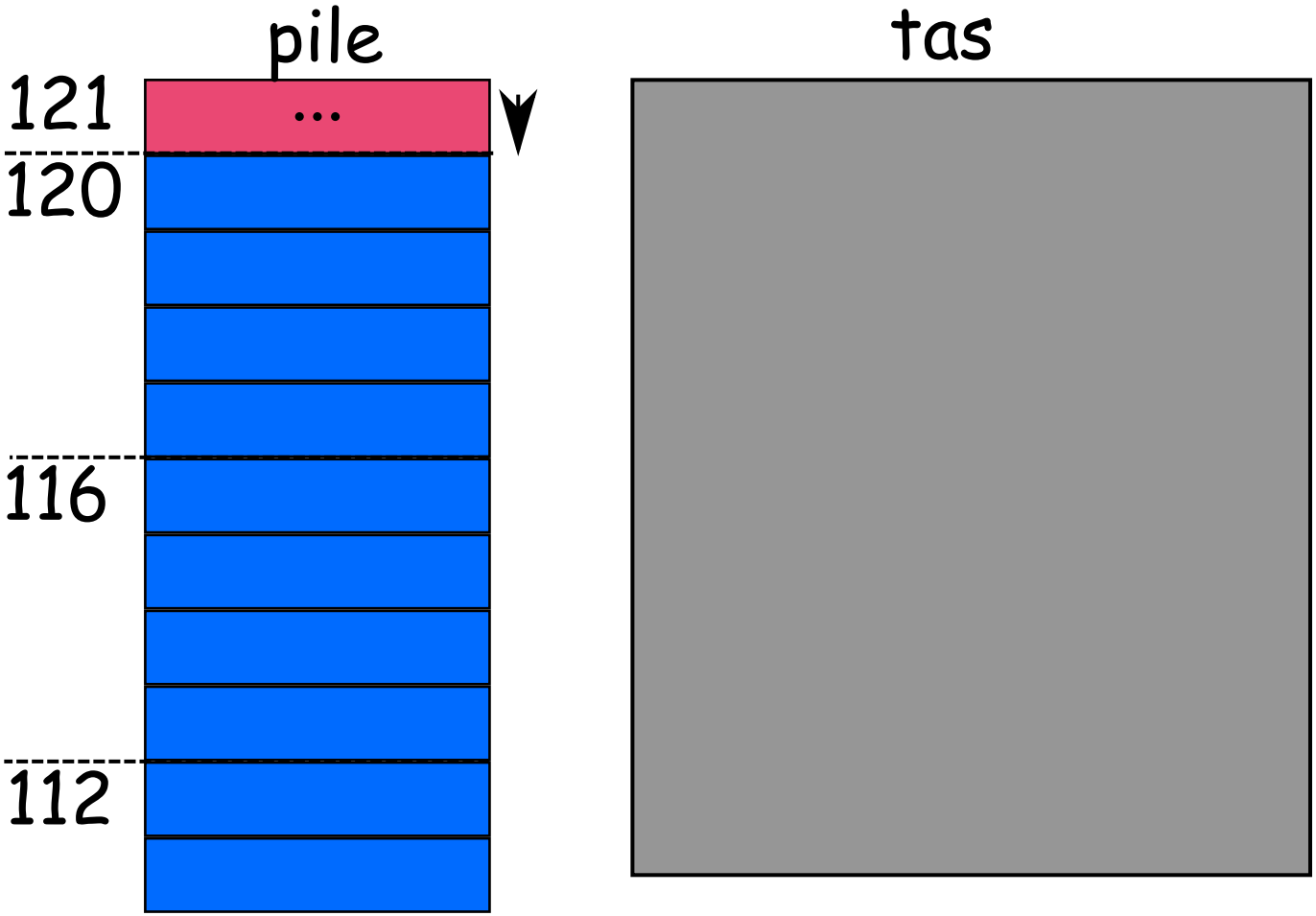
Allocation: exemple complet

exemple complet

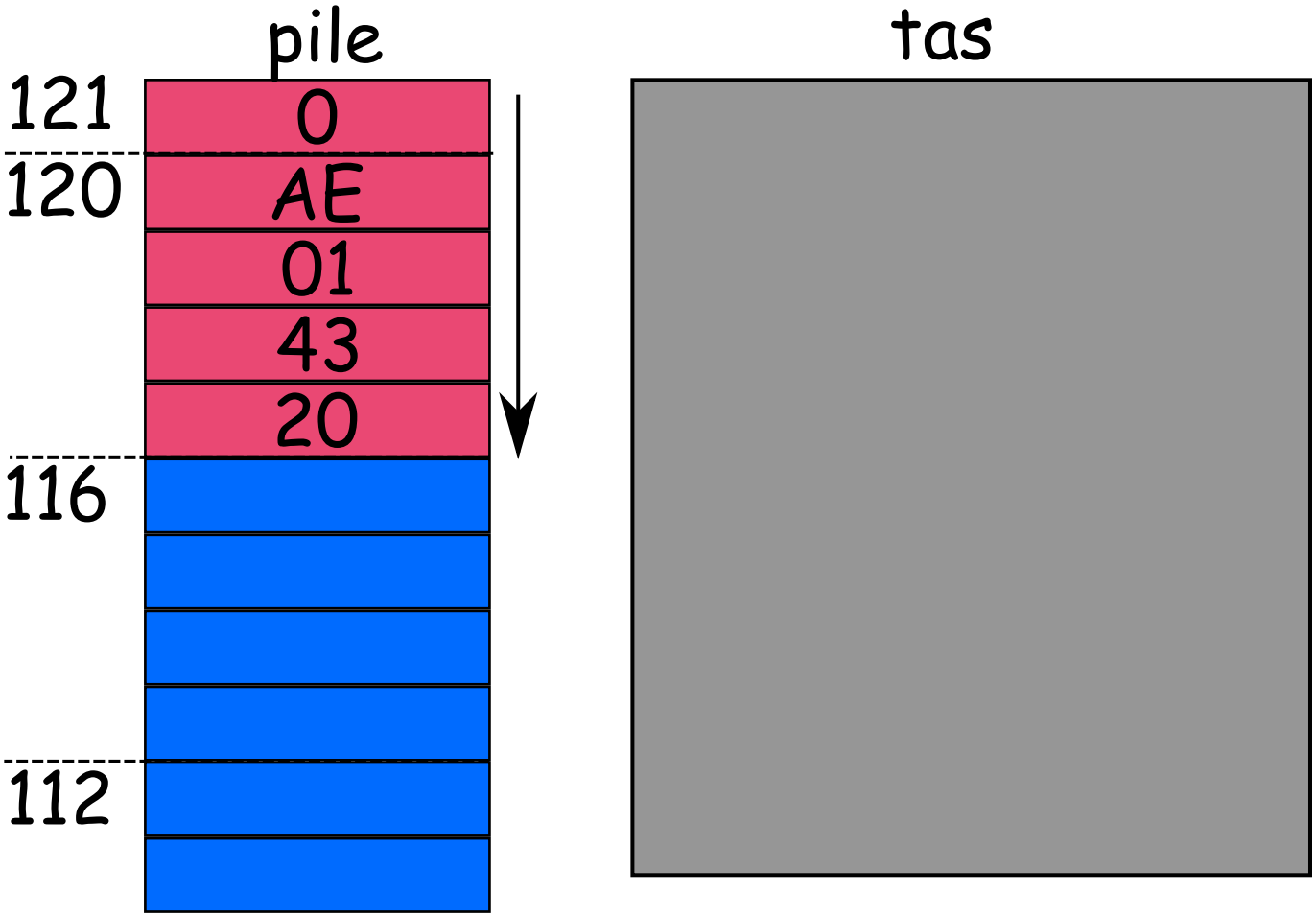
```
int *i = new int;
int *t = new int[10];
delete i;
i=NULL;
delete [] t;
t=NULL;
```

On réinitialise toujours un pointeur après avoir libéré la mémoire à l'adresse contenue dans ce pointeur.

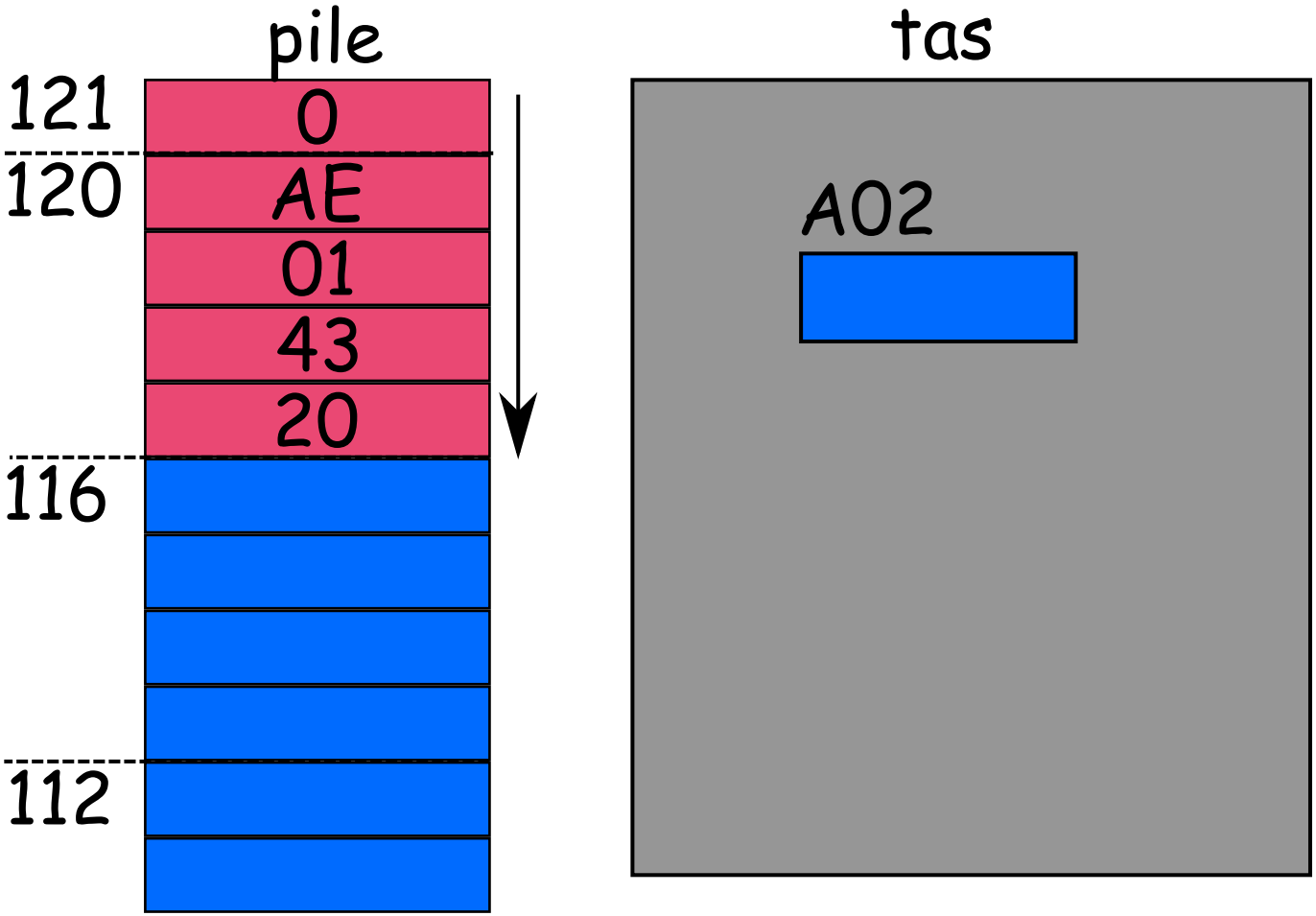
Allocation: exemple complet



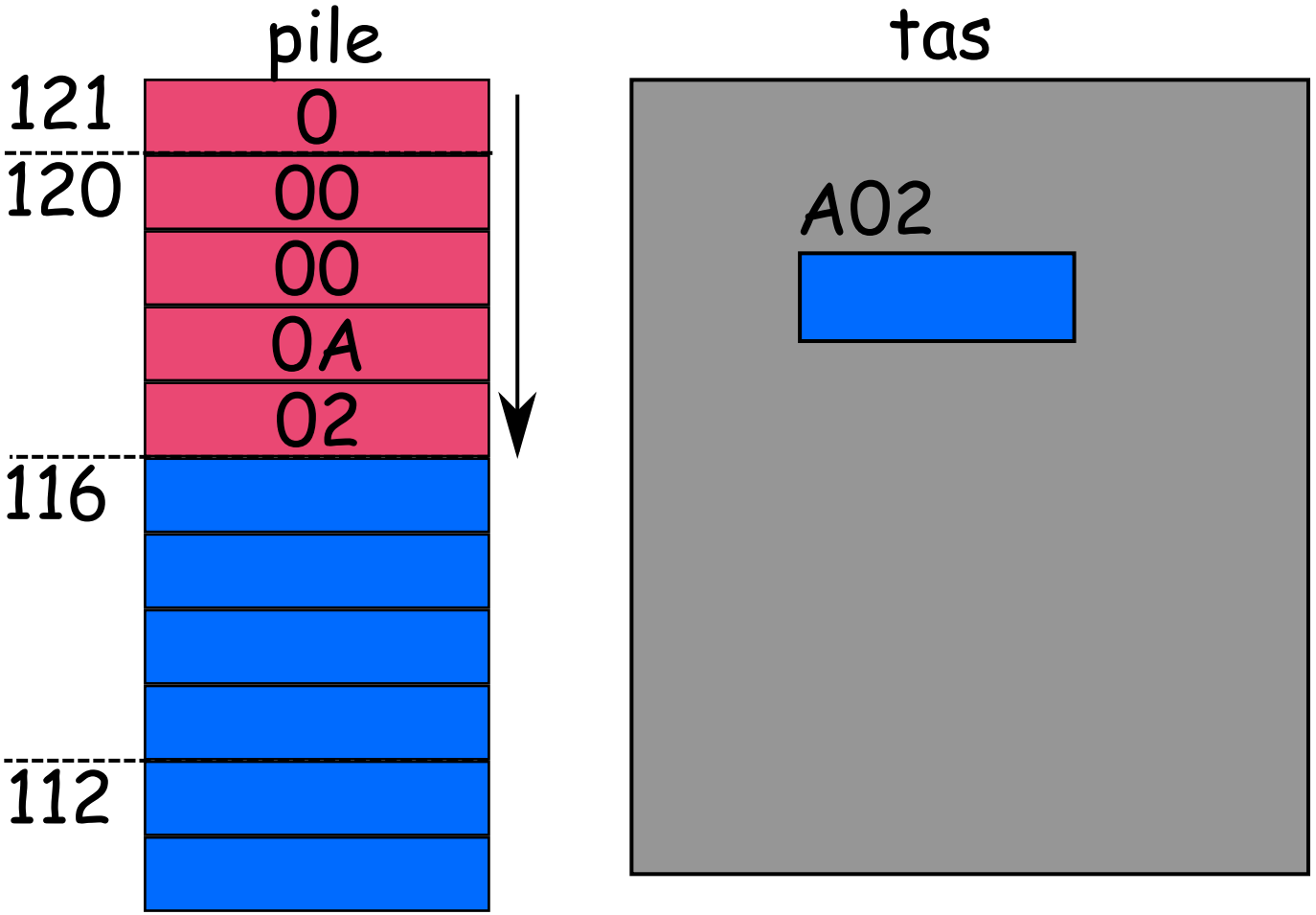
Allocation: exemple complet



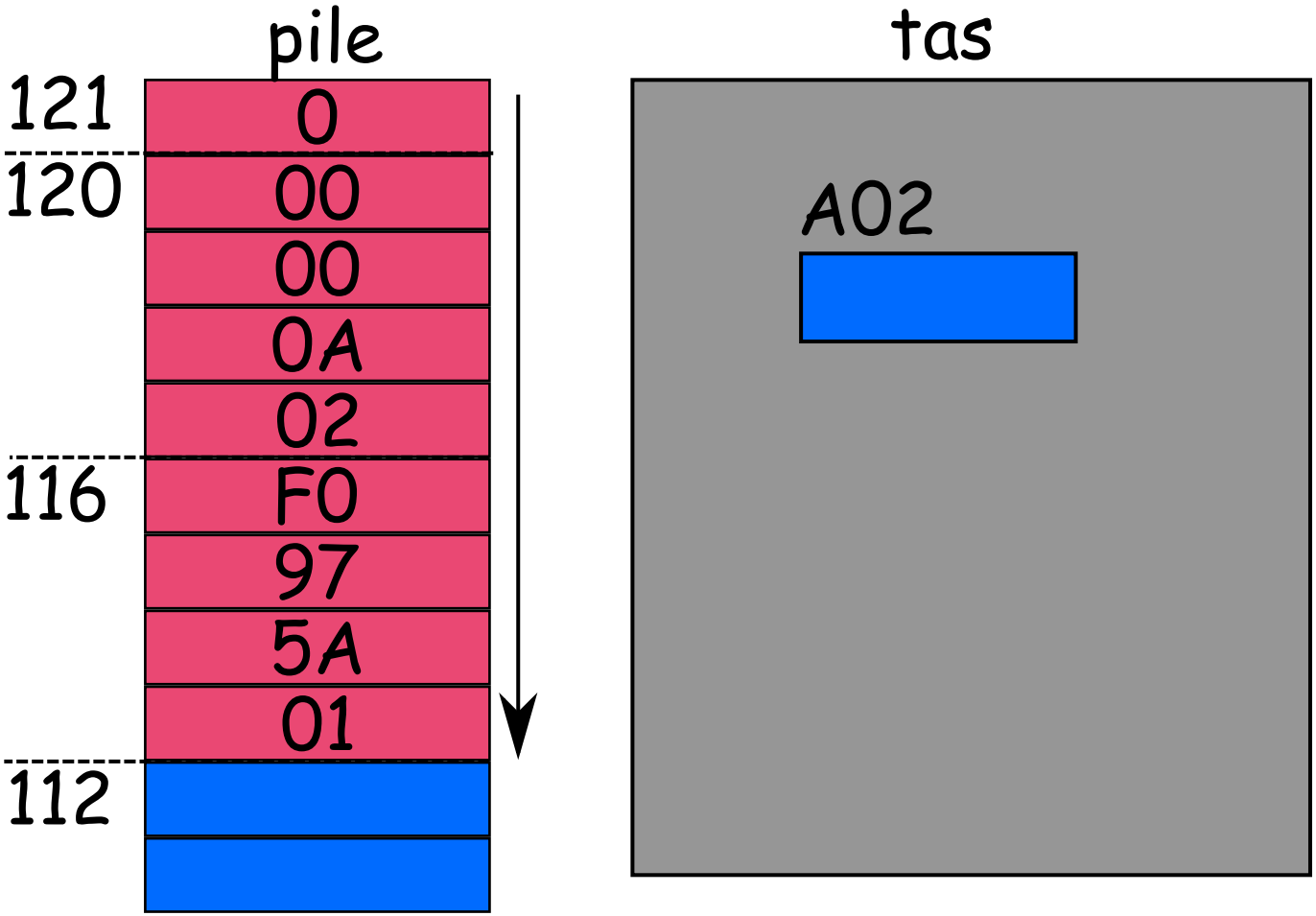
Allocation: exemple complet



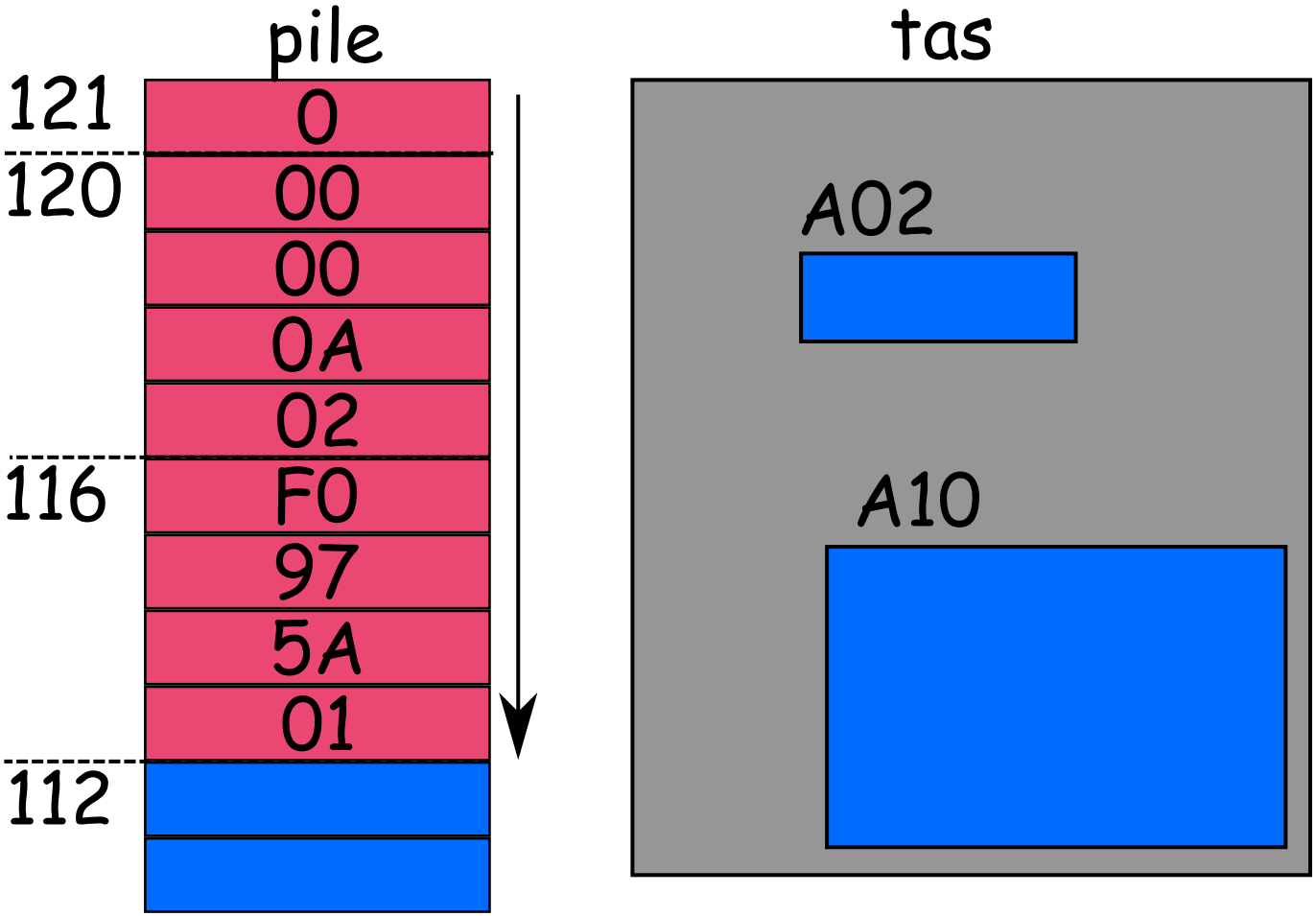
Allocation: exemple complet



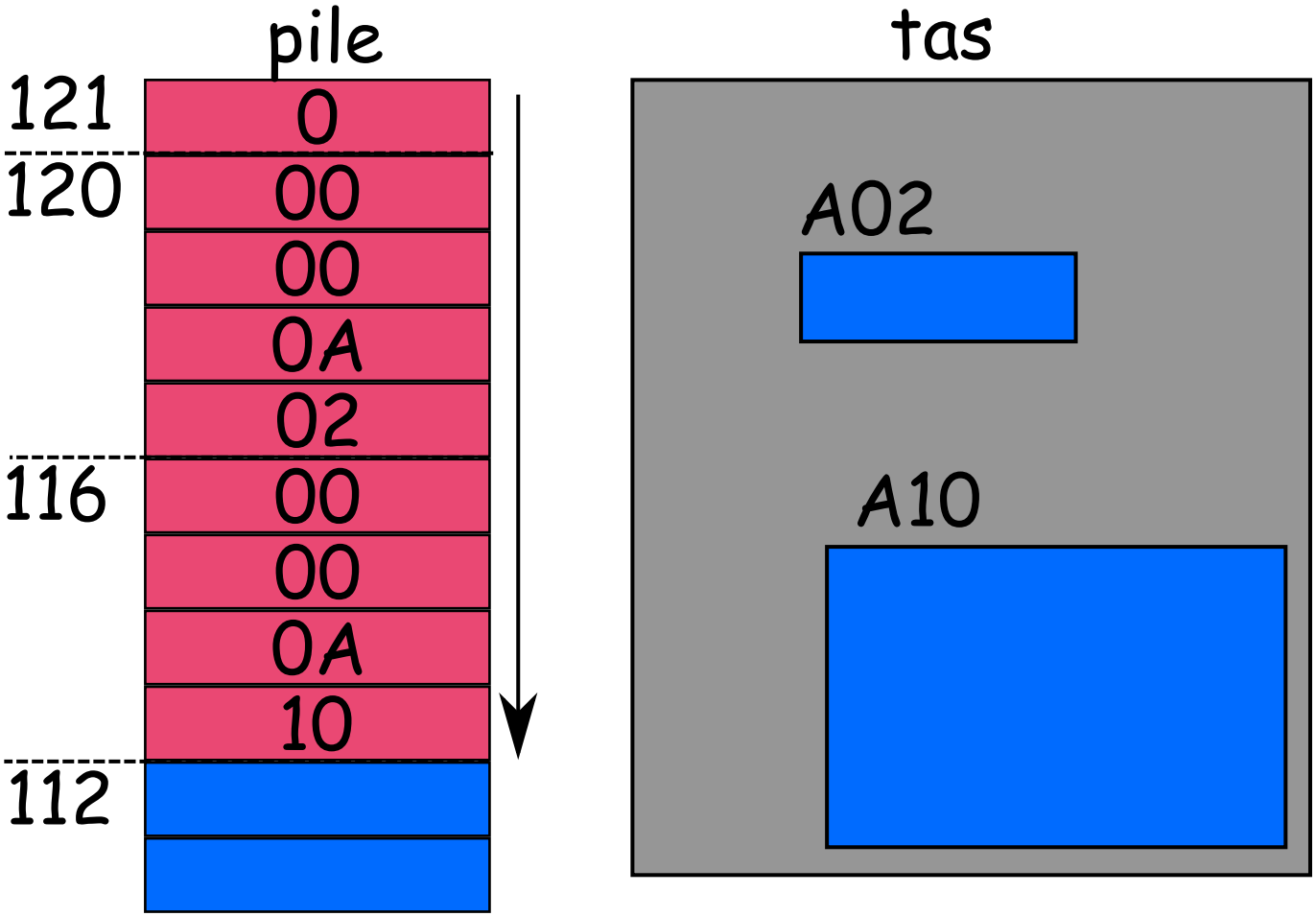
Allocation: exemple complet



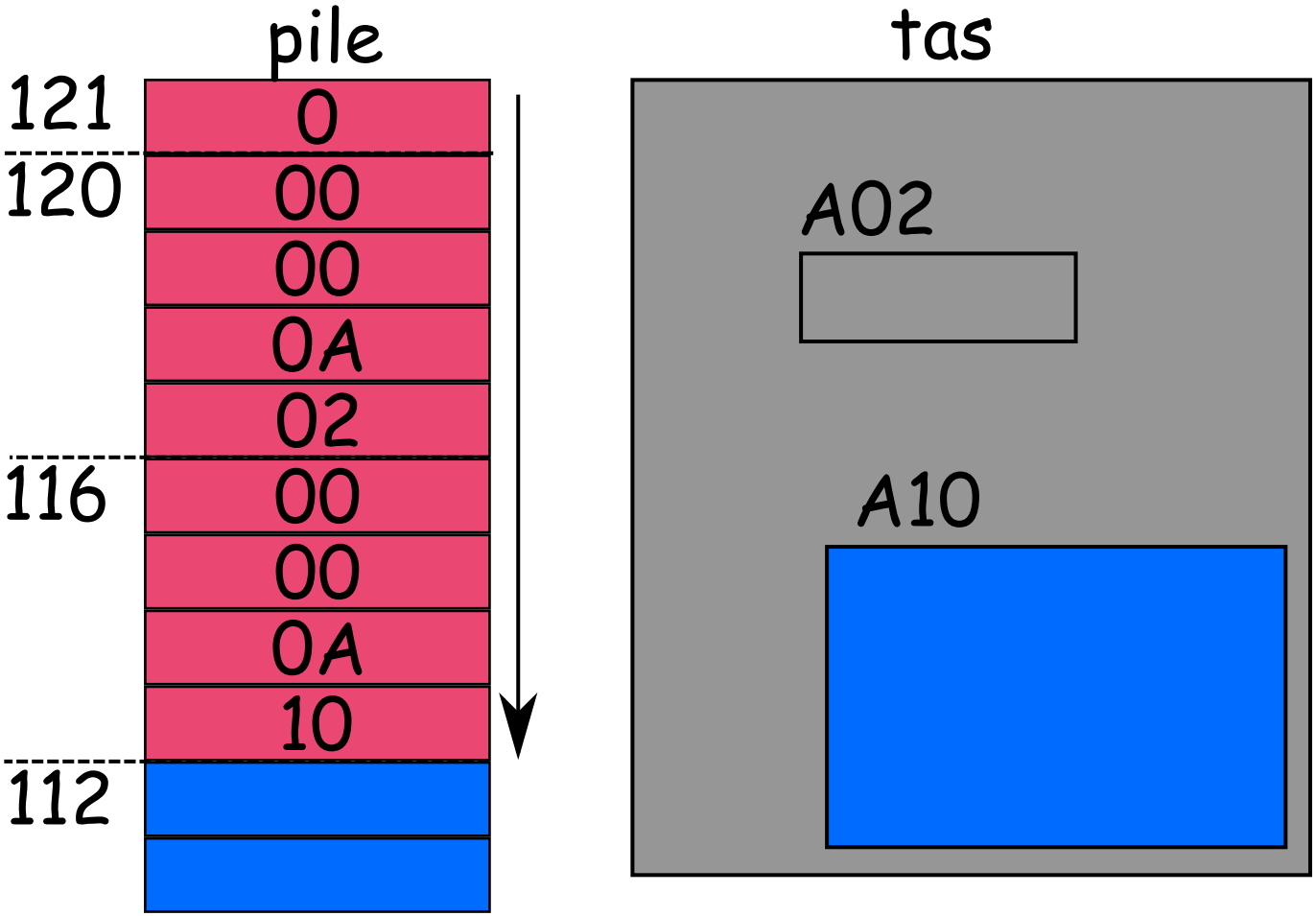
Allocation: exemple complet



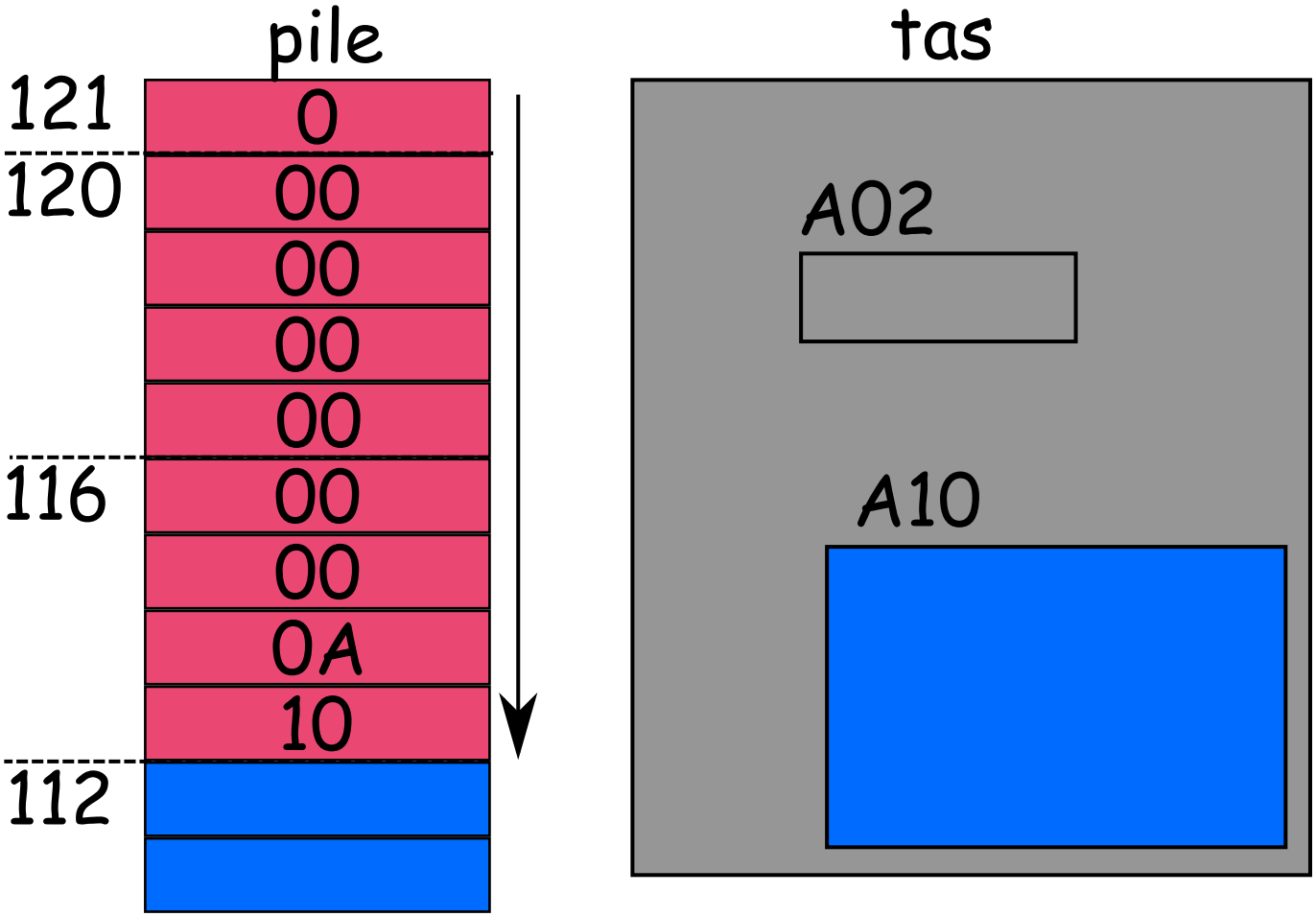
Allocation: exemple complet



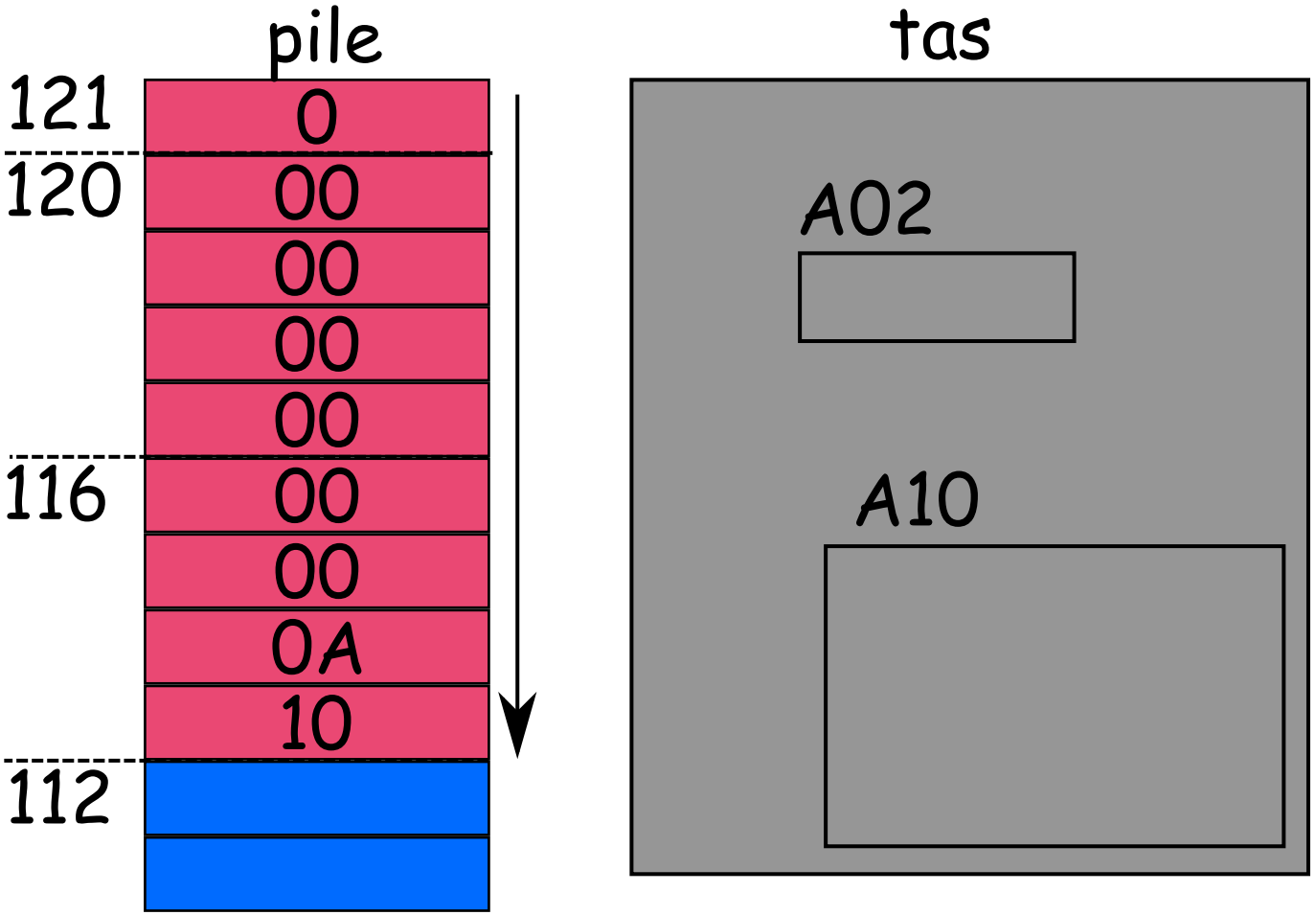
Allocation: exemple complet



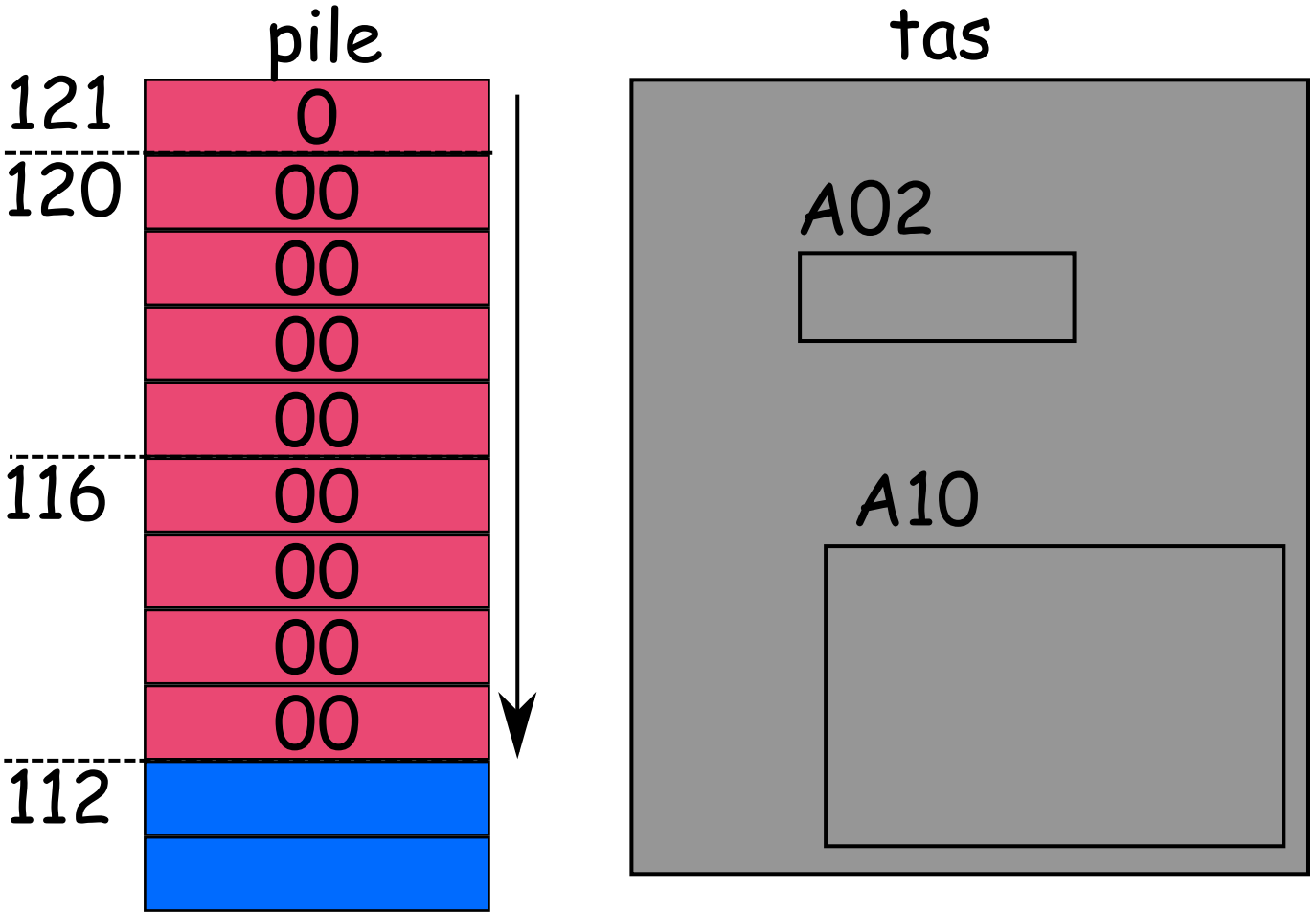
Allocation: exemple complet



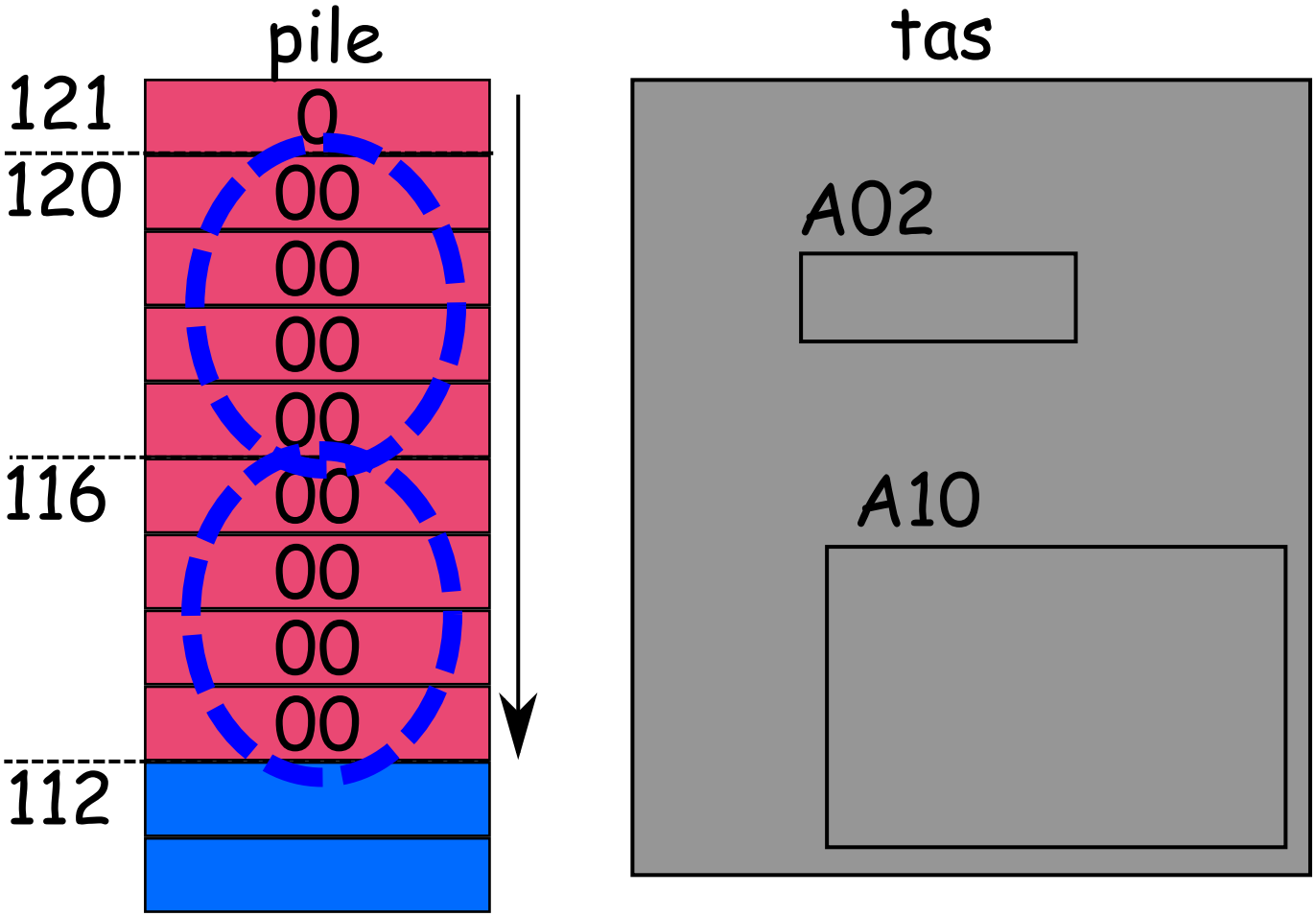
Allocation: exemple complet



Allocation: exemple complet



Allocation: exemple complet



Plan

Plan

Les classes: déclaration

Une classe consiste en un regroupement de méthodes et d'attributs.

Exemple

```
class NomClasse {
attributs
...
méthodes
...
};
```

L'ordre des déclarations ne compte pas.

La classe est une description des données internes et comportements qu'aura une instance générée par cette classe.

Les classes: instanciation

L'instanciation (ou réification), c'est à dire la création d'un objet instance (ressource) à partir de l'objet classe (description/générateur) peut se faire de façon dynamique:

allocation dynamique

```
A *a=new A();
```

ou automatique:

allocation automatique

```
A a;
```

Dans les deux cas, une ressource (adresse mémoire) est associée à l'instance. Dans l'allocation dynamique, cette ressource est située dans le tas, dans le cas automatique elle est située dans la pile.

Plan

Visibilité: définition

On contrôle l'accès aux méthodes et attributs pour une sous-classe ou un objet exterieur avec `public,protected et private:`

	la classe	sous-classe	exterieur
public	oui	oui	oui
protected	oui	oui	non
private	oui	non	non

Remarque: Les membres protected et private ne peuvent être accédées qu'à partir d'une fonction membre de l'instance

Visibilité: déclaration

La visibilité par défaut dans une classe est **private**. On modifie la visibilité de la façon suivante

Source

```
class NomClasse {
attributs et methodes privés
public:
attributs et methodes publiques
protected:
attributs et methodes protégés
...
};
```

On peut à tout moment changer la visibilité, celle-ci sera appliquée à toutes les déclaractions suivantes jusqu'au prochain changement de visibilité.

Les structures: déclaration

Les structures se déclarent comme en C:

Exemple

```
struct Nom {
...
attributs et méthodes publiques
...
};
```

En C++, les structures sont des classes dont la visibilité par défaut est **public**.

Par conséquent, elle peuvent contenir des attributs et des méthodes.

Deux nom de type sont associés à la structure: struct Nom et Nom

on voit que les structures C sont alors un cas particulier des structures C++

Les structures: instanciation

L'instanciation des structures se fait comme pour les classes.

allocation dynamique:

```
struct A { ... };
A *a=new A();
struct A *a=new A();
A *a=new struct A();
struct A *a=new struct A();
```

allocation automatique:

```
struct A { ... };
A a;
struct A a;
```

Friend: définition

Le mot clé **friend** permet, dans une class A, de donner à une fonction ou une autre classe les même droits qu'une méthode de A.

	la classe	friend	sous-classe	exterieur
public	oui	oui	oui	oui
protected	oui	oui	oui	non
private	oui	oui	non	non

une fonction f amie d'une classe A pourra accéder aux attributs privés de A ainsi qu'aux attributs protected d'une des classes parents de A

Friend: déclaration

La déclaration se fait dans la classe en indiquant soit le prototype de la fonction amie soit le nom de la classe amie:

Classe amie

```
class NomClasse {
...friend class NomClasseAmie ;
...}
```

Fonction amie

```
class NomClasse {
...friend void fonctionAmie(int,char,NomClasse) ;
...}
```

la visibilité courante n'importe pas pour déclarer une classe ou fonction amie.

Plan

Attributs et méthodes: définition

Une classe est un regroupement d'attributs, propriétés interne qu'aura une instance de la classe, et de méthodes, comportements qu'aura une instance.

Ces attributs sont déclarés à l'intérieur de la classe.

L'implémentation des méthodes peut se faire au moment de la déclaration ou à l'extérieur de la classe. On l'écrira dans la classe pour:

- les classes template
- les classes à usage locale au fichier

Sinon on écrit un fichier entête "NomClasse.hpp" et un fichier source "NomClasse.cpp".

Exemple de classe: 1

Un seul fichier

```
class Test {
        int i; // attribut privé
    public:
        void print(){ /* méthode publique */
            printf("Test\n");
        }
};
```

Exemple de classe: 1

Un seul fichier

```
class Test {
        int i; // attribut privé
    public:
        void print(){ /* méthode publique */
            printf("Test\n");
        }
};

Test t;
t.print();
```

Exemple de classe: 1

Un seul fichier

```
class Test {
        int i; // attribut privé
    public:
        void print(){ /* méthode publique */
            printf("Test\n");
        }
};

Test *t = new Test;
t->print();
```

Exemple de classe: 1

Un seul fichier

```
class Test {
        int i; // attribut privé
    public:
        void print(){ /* méthode publique */
            printf("Test\n");
        }
};

Test *t = new Test;
t->print();
```

La compilation se fait avec la commande:

```
g++ -Wall Test.cpp -o test
```

Exemple de classe: 2

Deux fichiers

Test.hpp:

```
class Test {
        int i; // attribut privé
    public:
        void print(); /* méthode publique */
};
```

Exemple de classe: 2

Deux fichiers

Test.hpp:

```
class Test {
        int i; // attribut privé
    public:
        void print(); /* méthode publique */
};
```

Test.cpp:

```
#include"Test.hpp"
voidTest::print(){
    printf("Test\n");
}
```

Exemple de classe: 2

Deux fichiers

Test.hpp:

```
class Test {
        int i; // attribut privé
    public:
        void print(); /* méthode publique */
};
```

Test.cpp:

```
#include"Test.hpp"
voidTest::print(){
    printf("Test\n");
}
```

Exemple.cpp:

```
Test t;
t.print();
```

Exemple de classe: 2

Deux fichiers

Test.hpp:

```
class Test {
        int i; // attribut privé
    public:
        void print(); /* méthode publique */
};
```

Test.cpp:

```
#include"Test.hpp"
voidTest::print(){
    printf("Test\n");
}
```

Exemple.cpp:

```
Test *t = new Test;
t->print();
```

Exemple de classe: 2

Deux fichiers

Test.hpp:

```
class Test {
        int i; // attribut privé
    public:
        void print(); /* méthode publique */
};
```

Test.cpp:

```
#include"Test.hpp"
voidTest::print(){
    printf("Test\n");
}
```

Exemple.cpp:

```
Test *t = new Test;
t->print();
```

La compilation se fait en deux temps:

```
g++ -Wall -c Test.cpp -o Test.o
g++ -Wall -c Exemple.cpp -o Exemple.o
g++ Test.o Exemple.o -o Exemple
```

Accès aux attributs

L'accès aux champs d'une classe se fait à l'aide de . ou -> selon que l'on dispose d'une instance de classe ou d'un pointeur sur une instance de classe.

Accès aux attributs

L'accès aux champs d'une classe se fait à l'aide de . ou -> selon que l'on dispose d'une instance de classe ou d'un pointeur sur une instance de classe.

Tout membre de classe à mon nom NomClasse::membre. Lorsqu'il n'y a pas d'ambiguité, NomClasse:: peut être omis.

Accès aux attributs

L'accès aux champs d'une classe se fait à l'aide de . ou -> selon que l'on dispose d'une instance de classe ou d'un pointeur sur une instance de classe. Tout membre de classe à mon nom NomClasse::membre. Lorsqu'il n'y a pas d'ambiguité, NomClasse:: peut être omis.

Cas automatique

```
ClasseA a;
a.i = 1 ; //i est un attribut public de ClasseA
a.ClasseA::i = 1 ; // idem
a.methode(); // appel de méthode
a.ClasseA::methode(); // idem
```

Dans ce cas l'instance se trouve dans la pile.

Accès aux attributs

L'accès aux champs d'une classe se fait à l'aide de . ou -> selon que l'on dispose d'une instance de classe ou d'un pointeur sur une instance de classe.

Tout membre de classe à mon nom NomClasse::membre. Lorsqu'il n'y a pas d'ambiguité, NomClasse:: peut être omis.

Cas dynamique

```
ClasseA *a = new ClasseA();
a->i = 1 ; //i est un attribut public de ClasseA
a->ClasseA::i = 1 ; // idem
a->methode(); // appel de méthode
a->ClasseA::methode(); // idem
```

Dans ce cas l'instance est alloué dynamiquement dans le tas.

Plan

Le mot clé **this**

Le mot clé **this** permet, lors d'un appel de méthode, de faire référence à l'instance source.

Le mot clé **this**

Le mot clé **this** permet, lors d'un appel de méthode, de faire référence à l'instance source.
Le type de la variable **this** est NomDeClasse const *: c'est un pointeur. (*Nous verrons plus tard le sens de* **const**.)

Le mot clé **this**

Le mot clé **this** permet, lors d'un appel de méthode, de faire référence à l'instance source.
Le type de la variable **this** est NomDeClasse const *: c'est un pointeur. (*Nous verrons plus tard le sens de* **const**.)

Exemple

```
class A {
        int attributPrive;
    public:
        void setAttribut(int valeur) {
            this->attributPrive = valeur ;
        }
};
```

Le mot clé **this**

Le mot clé **this** permet, lors d'un appel de méthode, de faire référence à l'instance source.
Le type de la variable **this** est NomDeClasse const *: c'est un pointeur. (*Nous verrons plus tard le sens de* **const**.)

Exemple

```
class A {
        int attributPrive;
    public:
        void setAttribut(int valeur) {
                attributPrive = valeur ;
        }
};
```

Lorsqu'il n'y a pas de variable locale à la méthode de même nom que l'attribut, on peut omettre **this**.

Opérateur de portée

L'opérateur de portée :: peut-être utilisé pour indiquer précisement la variable que l'on souhaite manipuler.

Opérateur de portée

L'opérateur de portée :: peut-être utilisé pour indiquer précisément la variable que l'on souhaite manipuler.

Exemple:

```
int variable; // variable globale, beurk!
class A {
        char variable;  // attribut privé de la classe
    public:
        void printAllVariable(double variable) {  // variable locale à la méthode
                variable;  // fait référence à la variable locale
                A::variable;  // fait référence à l'attribut d'instance
                this->A::variable;  // fait référence à l'attribut d'instance
                this->variable;  // fait référence à l'attribut d'instance
                ::variable;  // fait référence à la variable globale
        }
};
```

Plan

Les espaces de nom

Les espaces de nom permettent de regrouper un ensemble d'éléments (classes, variables globales, fonctions, ...).

Les espaces de nom

Les espaces de nom permettent de regrouper un ensemble d'éléments (classes, variables globales, fonctions, ...). Les espaces de nom sont utilisé pour structurer le code et pour éviter les problèmes de collisions

Les espaces de nom

Les espaces de nom permettent de regrouper un ensemble d'éléments (classes, variables globales, fonctions, ...). Les espaces de nom sont utilisé pour structurer le code et pour éviter les problèmes de collisions Pour déclarer un espace de nom on utilise le mot clé **namespace**:

Les espaces de nom

Les espaces de nom permettent de regrouper un ensemble d'éléments (classes, variables globales, fonctions, ...). Les espaces de nom sont utilisé pour structurer le code et pour éviter les problèmes de collisions Pour déclarer un espace de nom on utilise le mot clé **namespace**:

Espace de nom:

```
namespace Nom { // début de l'espace de nom
        class A {
        ...
        };
        int i;
        int fonction();
} // fin de l'espace de nom
```

Espace de noms

Lors de l'écriture du code des fonctions ou des méthodes des classes dans le fichier .cpp on peut soit ré-ouvrir l'espace de nom, soit indiquer explicitement le nom de l'objet avec l'opérateur de porté.

Espace de noms

Lors de l'écriture du code des fonctions ou des méthodes des classes dans le fichier .cpp on peut soit ré-ouvrir l'espace de nom, soit indiquer explicitement le nom de l'objet avec l'opérateur de porté.

A.hpp

```
namespace tec {
    class A {
    public:
        void m();
    };
}
```

Espace de noms

Lors de l'écriture du code des fonctions ou des méthodes des classes dans le fichier .cpp on peut soit ré-ouvrir l'espace de nom, soit indiquer explicitement le nom de l'objet avec l'opérateur de porté.

A.hpp

```
namespace tec {
    class A {
    public:
        void m();
    };
}
```

A.cpp

```
#include "A.hpp"
namespace tec {
    void A::m(){
    ...
    }
}
```

Espace de noms

Lors de l'écriture du code des fonctions ou des méthodes des classes dans le fichier .cpp on peut soit ré-ouvrir l'espace de nom, soit indiquer explicitement le nom de l'objet avec l'opérateur de porté.

A.hpp

```
namespace tec {
    class A {
    public:
        void m();
    };
}
```

A.cpp

```
#include "A.hpp"
void tec::A::m(){
...
}
```

Espace de noms

Lors de l'écriture du code des fonctions ou des méthodes des classes dans le fichier .cpp on peut soit ré-ouvrir l'espace de nom, soit indiquer explicitement le nom de l'objet avec l'opérateur de porté.

A.hpp

```
namespace tec {
    class A {
    public:
        void m();
    };
}
```

A.cpp

```
#include "A.hpp"
void tec::A::m(){
...
}
```

Il n'y pas pas de limite sur le nombre d'espace de nom différents pouvant être défini dans un fichier ni sur le niveau d'imbrication de ces espaces. Cependant la taille des noms de fonctions est limité et les espaces de nom font partis du nom de la fonction.

Espace de noms

Lors de l'écriture du code des fonctions ou des méthodes des classes dans le fichier .cpp on peut soit ré-ouvrir l'espace de nom, soit indiquer explicitement le nom de l'objet avec l'opérateur de porté.

A.hpp

```
namespace tec {
    class A {
    public:
        void m();
    };
}
```

A.cpp

```
#include "A.hpp"
void tec::A::m(){
...
}
```

D'une certaine manière, une classe peut-être vue comme un espace de nom particulier.

Espace de noms: **using**

Le mot clé **using** permet de rendre accessible tout ou partie d'un espace de nom dans une fonction, un espace de nom ou tout le code:

```
namespace spatial {
        class Navette {};
}
namespace missions {
        Navette n; //Erreur!
}
Navette n; //Erreur!
void starWars(){
    Navette n; //Erreur!
}
```

Espace de noms: **using**

Le mot clé **using** permet de rendre accessible tout ou partie d'un espace de nom dans une fonction, un espace de nom ou tout le code:

```
namespace spatial {
        class Navette {};
}
namespace missions {
        spatial::Navette n; //ok
}
spatial::Navette n; //ok
void starWars(){
    spatial::Navette n; //ok!
}
```

Utilisation de l'opérateur de porté

Espace de noms: **using**

Le mot clé **using** permet de rendre accessible tout ou partie d'un espace de nom dans une fonction, un espace de nom ou tout le code:

```
namespace spatial {
        class Navette {};
}
using spatial::Navette;
namespace missions {
        Navette n; //ok
}
Navette n; //ok
void starWars(){
    Navette n; //ok!
}
```

using sur un objet

Espace de noms: **using**

Le mot clé **using** permet de rendre accessible tout ou partie d'un espace de nom dans une fonction, un espace de nom ou tout le code:

```
namespace spatial {
        class Navette {};
}
namespace missions {
        Navette n; //Erreur!
}
using spatial::Navette;
Navette n; //ok
void starWars(){
    Navette n; //ok!
}
```

Le **using** n'est actif que sur les instructions qui suivent...

Espace de noms: **using**

Le mot clé **using** permet de rendre accessible tout ou partie d'un espace de nom dans une fonction, un espace de nom ou tout le code:

```
namespace spatial {
        class Navette {};
}
namespace missions {
        using spatial::Navette;
        Navette n; //ok
}
Navette n; //Erreur!
void starWars(){
    Navette n; //Erreur!
}
```

Le **using** dans un espace de nom est local à cet espace

Espace de noms: **using**

Le mot clé **using** permet de rendre accessible tout ou partie d'un espace de nom dans une fonction, un espace de nom ou tout le code:

```
namespace spatial {
        class Navette {};
}
namespace missions {
        Navette n; //Erreur!
}
Navette n; //Erreur!
void starWars(){
    using spatial::Navette;
    Navette n; //ok
}
```

Le **using** dans une fonction est local à cette fonction

Espace de noms: **using**

Le mot clé **using** permet de rendre accessible tout ou partie d'un espace de nom dans une fonction, un espace de nom ou tout le code:

```
namespace spatial {
        class Navette {};
}
namespace missions {
        Navette n; //Erreur!
}
Navette n; //Erreur!
void starWars(){
    using namespace spatial;
    Navette n; //ok
}
```

Le **using** peut se faire sur l'ensemble des objets d'un espace

Using

Le **using** crée des synonymes locaux entre les espaces de nom:

Fichier exemple.hpp:

```
namespace math{
  const double pi=3.14;
}

namespace cercle{
  using math::pi;
  double surf(double);
}
```

Fichier exemple.cpp:

```
#include''exemple.hpp''

double cercle::surf(double d)
{
  return pi*pi*d;
}
```

Using

Le **using** crée des synonymes locaux entre les espaces de nom:
Fichier exemple.hpp:

```
namespace math{
  const double pi=3.14;
}

namespace cercle{
  using math::pi;
  double surf(double);
}
```

Fichier exemple.cpp:

```
#include''exemple.hpp''

double cercle::surf(double d)
{
  return pi*pi*d;
}
```

- Permet de remplacer facilement une référence externe par une autre.

Using

Le **using** crée des synonymes locaux entre les espaces de nom:
Fichier exemple.hpp:

```
namespace math{
  const double pi=3.14;
}

namespace cercle{
  using math::pi;
  double surf(double);
}
```

Fichier exemple.cpp:

```
#include''exemple.hpp''

double cercle::surf(double d)
{
  return pi*pi*d;
}
```

- Permet de remplacer facilement une référence externe par une autre.
- Permet aussi l'introduction de modularité sans modification profonde du code.