

# Programmation et généricité en C/C++

# Structures

- Equivalent du 'record' de Pascal.
- Mot-clé 'struct'.
- Accès aux membres avec l'opérateur '.'

```
struct Personne
{
    string nom;
    unsigned char age;
    string adresse;
};
```

```
Personne p;
```

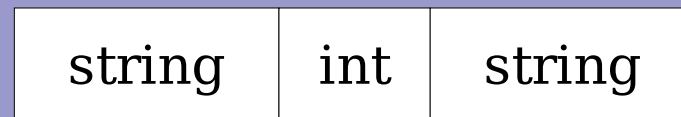
```
p.nom = "Toto";
// ...
```

# Structures (2)

- Il est possible d'initialiser tous les membres à la déclaration.

```
Personne p = { "Toto", 34, "Quelque part" };
```

```
cout << "Nom: " << p.nom << endl  
      << "Age: " << p.age << endl  
      << "Adresse: " << p.adresse << endl;
```



Personne

# Tableaux

- Suite d'éléments de même type placés en mémoire de façon contigue.
- Un tableau possède une taille fixe  $n$ .
- Les éléments sont indicés de 0 à  $n$ .
- **Pas d'affectation entre tableaux !**

```
// Tableau de taille 10
int tableau[10];

for (int i = 0; i < 10; ++i)
{
    tableau[i] = i;
    cout << tableau[i] << endl;
}
```

# Tableaux (2)

- On peut utiliser un entier constant pour définir la taille d'un tableau.
- On peut initialiser un tableau à la déclaration.

```
// Avec une constante
const int taille = 3;
int tab[taille] = {1, 2, 3};

// Taille implicite
double tab2[] = {0.1, 2.3, 3.4, 5.6};

// Tableau de constantes
const int tab3[] = {1, 2, 3, 4}
```

# Références

- Le contenu d'une variable est logé en mémoire à une adresse.
- L'adresse d'une variable est accessible par l'opérateur '&'.
- Il est possible de créer des références.
- Une référence est une variable dont l'adresse est la même que celle d'une autre.
- Les références sont utiles pour le passage de paramètres aux fonctions.

# Exemple sur les références

```
// Déclarations
int x = 12;
int &y = x;           // y est une référence sur x

// Affichage des adresses
cout << "@x = " << &x << endl
     << "@y = " << &y << endl;

// Modification de y et conséquences sur x
cout << x << endl;
y = 20;
cout << x << endl;
```

# Pointeurs

- Un pointeur contient une adresse mémoire.
- Déclaration: 'type\* ptr'.
- Le pointeur nul a pour valeur 0 ou NULL.
- L'opérateur '\*' (déréférencement) permet d'accéder à la valeur de l'adresse pointée.

# Exemple sur les pointeurs

```
// Déclarations
int  x = 12;
int* y = &x;           // y pointe sur l'adresse de x

// Affichage des adresses
cout << "@x = " << &x << endl
     << "y = "   << y << endl;

// Modification de *y et conséquences sur x
cout << x << endl;
*y = 20;
cout << x << endl;
```

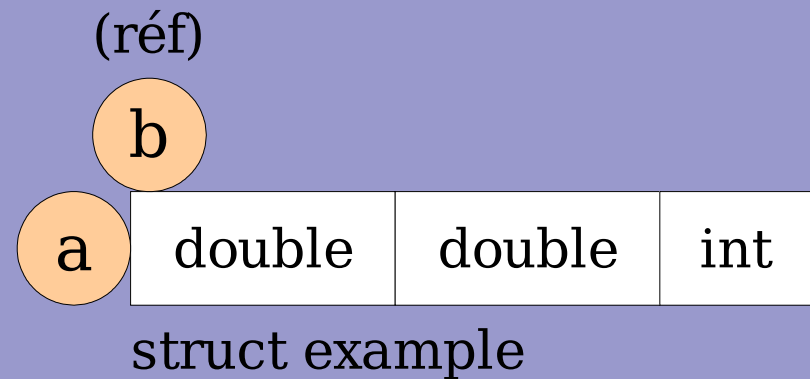
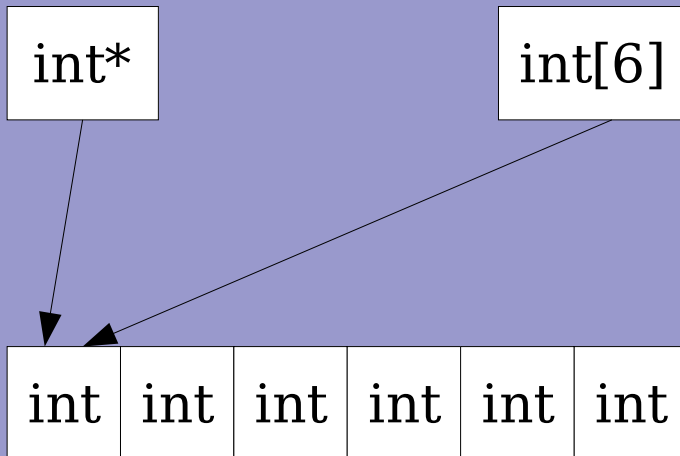
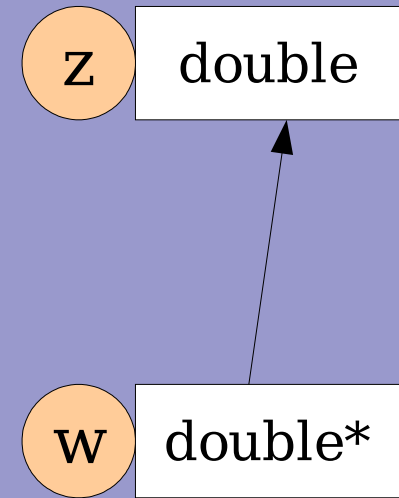
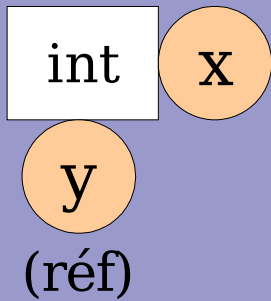
# Retour sur les tableaux

- Les tableaux sont en réalité des pointeurs.
- Le nom d'un tableau est le pointeur sur le premier élément.

```
int tab[3] = {1, 2, 3};
int *ptr = tab;

for (int i = 0; i < 3; ++i)
{
    cout << ptr[i] << endl;
}
```

# Illustration en mémoire



# Allocation mémoire

- Il est possible d'allouer dynamiquement des objets en mémoire.
- L'opérateur 'new' effectue l'allocation et renvoie un pointeur sur la zone mémoire allouée.
- L'opérateur 'delete' permet de désallouer une zone mémoire.
- **1 new = 1 delete.**

# Allocation mémoire (2)

- 'new' permet d'allouer un tableau de taille dynamique.
- 'delete[]' permet de désallouer un tableau dynamique.
- 'new' et 'delete' sont une grande avancée comparé au C.
- En C, il faut allouer la mémoire en tenant compte de la taille occupée par les données. Les 2 opérateurs prennent en charge cette opération.

# Exemples 'new'/'delete'

```
// Allocation d'un objet simple
int *x;
x = new int;
*x = 3;
delete x;
```

```
// Allocation de tableau
int *tab;
tab = new int[30];
for (int i = 0; i < 30; ++i)
{
    tab[i] = i;
}
delete[] tab;
```

# Equivalents C

```
// Allocation d'un objet simple
int *x;
x = (int*)malloc(sizeof(int));
*x = 3;
free(x, sizeof(int));

// Allocation de tableau
int *tab;
tab = (int*)malloc(30 * sizeof(int));
for (int i = 0; i < 30; ++i)
{
    tab[i] = i;
}
free(tab, 30 * sizeof(int));
```

# Pointeurs et structures

- Pour accéder aux membres d'une structure par le biais d'un pointeur, il existe l'opérateur '->'.

```
// Sans ->
Personne *p = new Personne;
(*p).nom = "Toto";
(*p).age = 10;
(*p).adresse = "Quelque part";
```

```
// Avec ->
Personne *p = new Personne;
p->nom = "Toto";
p->age = 10;
p->adresse = "Quelque part";
```

# Arithmétique de pointeurs

- On peut faire 'avancer' ou 'reculer' en mémoire un pointeur d'un nombre donné d'octets avec '+' et '-'.
- Par exemple, on peut itérer sur les éléments d'un tableau avec les pointeurs.
- Avancer de n éléments revient à avancer en mémoire de  $n * |\text{type}|$  éléments.
- '++' et '--' sont disponibles.

# Itération par les pointeurs

- Anticipation sur les itérateurs du C++ ...

```
int[] tab = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
for (int *iter = tab; iter != &tab[10]; ++iter)  
{  
    cout << *iter << endl;  
}
```

// équivalent à :

```
for (int i = 0; i < 10; ++i)  
{  
    cout << tab[i] << endl;  
}
```

# Les fonctions

- Pas de distinction procédure / fonction comme en Pascal.
- L'équivalent d'une procédure est une fonction qui ne renvoie rien (type void).
- On peut déclarer les fonctions avant de les implémenter (déclaration des prototypes). Ceci évite les problèmes liés à "l'ordre" des fonctions.
- 'return' renvoie la valeur de la fonction.

# Exemples de fonctions

```
// Déclaration des prototypes
void coucou();
int carre(int x);
double somme(double x, double y);

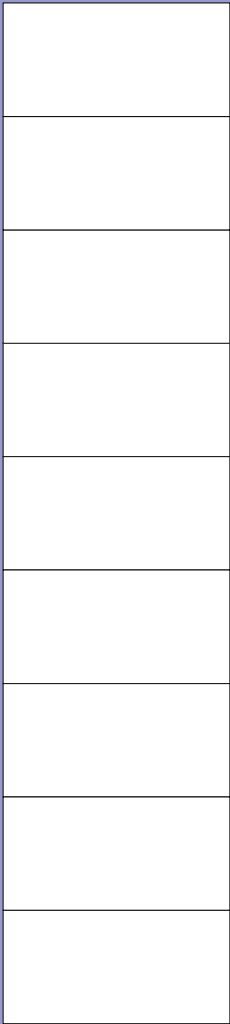
// Implémentations
void coucou()
{
    cout << "coucou" << endl;
}
int carre(int x)
{
    return x * x;
}
double somme(double x, double y)
{
    return x + y;
}
```

# Passage des paramètres

- Le passage des paramètres se fait par copie.
- Sur de gros objets (par exemple une structure de grosse taille), l'appel de la fonction est lourd.
- On peut souhaiter faire modifier une variable par une fonction.
- On peut utiliser des références ou des pointeurs pour pallier à ces problèmes.

# Appel de fonction

Pile



```
void plop()  
{  
    int n2 = carre(5);  
}
```

```
int carre(int n)  
{  
    return n * n;  
}
```

An arrow points from the 'carre(5)' call in the first code block to the 'carre(int n)' function definition in the second code block, indicating the call site.

# Bien utiliser les paramètres

- Passage de paramètres 'classiques' de façon efficace:

```
void toto(const double &x, const double &y) { ... }
```

- Permettre la modification d'un paramètre:

```
// Style C++  
void toto(double &x, double &y) { ... }
```

```
// Style C  
void toto(double* x, double* y) { ... }
```

# Valeurs par défaut

- Il est possible d'attribuer des valeurs par défaut aux paramètres des fonctions.
- Ceci peut permettre d'alléger l'écriture de code.

```
void fonction(const int &p1, const int& p2 = 1);  
  
(...)  
  
fonction(1); // p2 == 1  
  
fonction(1, 2);
```

# Surcharge de fonctions

- En C/C++, plusieurs fonctions peuvent porter un nom identique.
- Il ne faut pas qu'il y ait une possible ambiguïté.
- Exemples d'ambiguïtés:
  - paramètres (meme ordre + meme type)
  - utilisation de valeurs par défauts.

# Surcharges de fonctions (2)

```
// Bien
void foo();
void foo(const int &n);
int  foo(const int &n);

// Pas bien
void bar(const int &x, const int &y);
void bar(const int &y, const int &x);

// Attention !
int plop(const int &x);
int plop(const int &x, const int &y = 2);
(...)
n = plop(1, 2); // Ok
n = plop(3);    // Lequel appeller ???
```