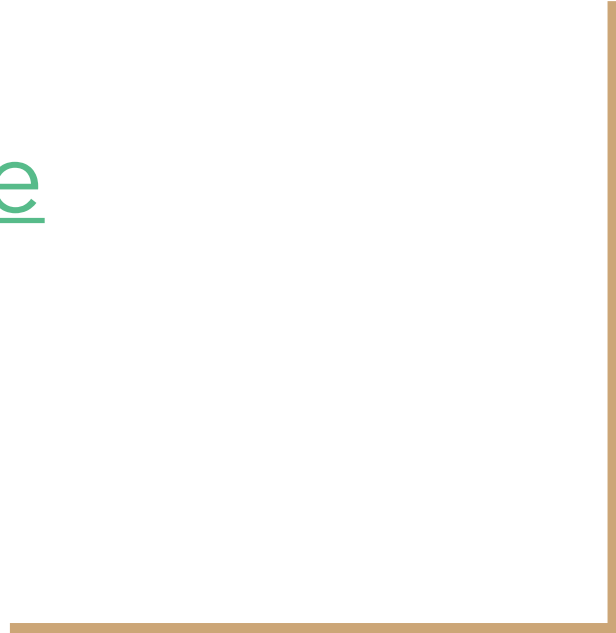


C++ : Fondements

UE15
Informatique appliquée

Frédéric Pluquet
pluquetf@helha.be



2018-2019

Entrée / sortie console

- `int i = 0, j = 0;`
- `cin >> i >> j;`
- `cout << i << j;`

Integers : types différents

Types allowed for integer literals		
suffix	decimal bases	hexadecimal or octal bases
no suffix	<code>int</code> <code>long int</code> <code>long long int</code> (since C++11)	<code>int</code> <code>unsigned int</code> <code>long int</code> <code>unsigned long int</code> <code>long long int</code> (since C++11) <code>unsigned long long int</code> (since C++11)
u or U	<code>unsigned int</code> <code>unsigned long int</code> <code>unsigned long long int</code> (since C++11)	<code>unsigned int</code> <code>unsigned long int</code> <code>unsigned long long int</code> (since C++11)
l or L	<code>long int</code> <code>unsigned long int</code> (until C++11) <code>long long int</code> (since C++11)	<code>long int</code> <code>unsigned long int</code> <code>long long int</code> (since C++11) <code>unsigned long long int</code> (since C++11)
both l/L and u/U	<code>unsigned long int</code> <code>unsigned long long int</code> (since C++11)	<code>unsigned long int</code> (since C++11) <code>unsigned long long int</code> (since C++11)
ll or LL	<code>long long int</code> (since C++11)	<code>long long int</code> (since C++11) <code>unsigned long long int</code> (since C++11)
both ll/LL and u/U	<code>unsigned long long int</code> (since C++11)	<code>unsigned long long int</code> (since C++11)

Integers : Tailles

Type specifier	Equivalent type	Width in bits by data model									
		C++ standard	LP32	ILP32	LLP64	LP64					
<code>short</code>	<code>short int</code>	at least 16	16	16	16	16					
<code>short int</code>											
<code>signed short</code>											
<code>signed short int</code>											
<code>unsigned short</code>	<code>unsigned short int</code>	at least 16	16	32	32	32					
<code>unsigned short int</code>											
<code>int</code>	<code>int</code>						at least 16	16	32	32	32
<code>signed</code>											
<code>signed int</code>											
<code>unsigned</code>											
<code>unsigned int</code>	<code>unsigned int</code>	at least 32	32	32	32	64					
<code>long</code>	<code>long int</code>										
<code>long int</code>											
<code>signed long</code>											
<code>signed long int</code>											
<code>unsigned long</code>	<code>unsigned long int</code>	at least 64	64	64	64	64					
<code>unsigned long int</code>											
<code>long long</code>	<code>long long int</code> (C++11)										
<code>long long int</code>											
<code>signed long long</code>											
<code>signed long long int</code>											
<code>unsigned long long</code>	<code>unsigned long long int</code> (C++11)	at least 64	64	64	64	64					
<code>unsigned long long int</code>											

Domaines

Type	Size in bits	Format	Value range	
			Approximate	Exact
character	8	signed (one's complement)	-127 to 127	
		signed (two's complement)	-128 to 127	
		unsigned	0 to 255	
	16	unsigned	0 to 65535	
	32	unsigned	0 to 1114111 (0x10ffff)	
integer	16	signed (one's complement)	$\pm 3.27 \cdot 10^4$	-32767 to 32767
		signed (two's complement)		-32768 to 32767
		unsigned	0 to $6.55 \cdot 10^4$	0 to 65535
	32	signed (one's complement)	$\pm 2.14 \cdot 10^9$	-2,147,483,647 to 2,147,483,647
		signed (two's complement)		-2,147,483,648 to 2,147,483,647
		unsigned	0 to $4.29 \cdot 10^9$	0 to 4,294,967,295
	64	signed (one's complement)	$\pm 9.22 \cdot 10^{18}$	-9,223,372,036,854,775,807 to 9,223,372,036,854,775,807
		signed (two's complement)		-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
		unsigned	0 to $1.84 \cdot 10^{19}$	0 to 18,446,744,073,709,551,615
floating point	32	IEEE-754 🔗	$\pm 3.4 \cdot 10^{\pm 38}$ (~7 digits)	<ul style="list-style-type: none"> min subnormal: $\pm 1.401,298,4 \cdot 10^{-45}$ min normal: $\pm 1.175,494,3 \cdot 10^{-38}$ max: $\pm 3.402,823,4 \cdot 10^{38}$
	64	IEEE-754 🔗	$\pm 1.7 \cdot 10^{\pm 308}$ (~15 digits)	<ul style="list-style-type: none"> min subnormal: $\pm 4.940,656,458,412 \cdot 10^{-324}$ min normal: $\pm 2.225,073,858,507,201,4 \cdot 10^{-308}$ max: $\pm 1.797,693,134,862,315,7 \cdot 10^{308}$

Opérateurs arithmétiques

- Les opérateurs arithmétiques binaires (+, -, * et /) et les opérateurs relationnels ne sont définis que pour des opérandes d'un même type parmi :
 - int, long int (et leurs variantes non signées)
 - float, double et long double

Opérateurs arithmétiques

- Mais on peut constituer des expressions mixtes (opérandes de types différents) ou contenant des opérandes d'autres types (bool, char et short), grâce à l'existence de deux sortes de conversions implicites :
- les conversions d'ajustement de type, selon l'une des hiérarchies :
 - `int -> long -> float -> double -> long double`
 - `unsigned int -> unsigned long -> float -> double -> long double`
- les promotions numériques, à savoir des conversions systématiques de char (avec ou sans attribut de signe), bool et short en int.

Opérateurs logiques

Opérande 1	Opérateur	Opérande 2	Résultat
0	&&	0	faux
0	&&	non nul	faux
non nul	&&	0	faux
non nul	&&	non nul	vrai
0		0	faux
0		non nul	vrai
non nul		0	vrai
non nul		non nul	vrai
	!	0	vrai
	!	non nul	faux

Les deux opérateurs && et || sont « à court-circuit » : le second opérande n'est évalué que si la connaissance de sa valeur est indispensable.

Opérateur de cast

```
int n = 3, p = 5;
```

```
cout << (double) n / p;
```

```
cout << static_cast<double> (n/p);
```

Priorité des opérateurs

Priorité	Opérateur	Description	Associativité
1	::	Résolution de portée	Gauche à droite
2	++ --	Incrémentation et décrémentation suffixe/postfixe	
	()	Appel de fonction	
	[]	Accès dans un tableau	
	. ->	Sélection membre par référence Sélection membre par pointeur	
3	++ --	Incrementation et décrementation préfixe	Droite à gauche
	+ -	Plus et moins unaires	
	! ~	NON logique et NON binaire	
	(type)	Transtypage	
	*	Indirection (déréféréce)	
	&	Adresse	
	sizeof	Taille	
	new, new[]	Allocation dynamique de la mémoire	
	delete, delete[]	Libération dynamique de la mémoire	
4	.* ->*	Pointeur vers un membre	Gauche à droite
5	* / %	Multiplication, division et reste	
6	+ -	Addition et soustraction	
7	<< >>	Décalage binaire à gauche et à droite	
8	< <=	Respectivement pour les opérateurs de comparaison < et ≤	
	> >=	Respectivement pour les opérateurs de comparaison > et ≥	

Priorité des opérateurs

9	<code>==</code> <code>!=</code>	Respectivement pour les comparaisons <code>=</code> et <code>≠</code>	
10	<code>&</code>	ET binaire	
11	<code>^</code>	XOR binaire (ou exclusif)	
12	<code> </code>	OU binaire (ou inclusif)	
13	<code>&&</code>	ET logique	
14	<code> </code>	OU logique	
15	<code>?:</code>	opérateur conditionnel ternaire	Droite à gauche
	<code>=</code>	Affectation directe (fourni par défaut pour les classes C++)	
	<code>+=</code> <code>-=</code>	Affectation par somme ou différence	
	<code>*=</code> <code>/=</code> <code>%=</code>	Affectation par produit, division ou reste	
	<code><<=</code> <code>>>=</code>	Affectation par décalage binaire à gauche ou à droite	
	<code>&=</code> <code>^=</code> <code> =</code>	Affectation par ET, XOR ou OU binaire	
16	<code>throw</code>	opérateur Throw (pour les exceptions)	
17	<code>,</code>	Virgule	Gauche à droite

Exercice 1

- Soient les déclarations :

```
char c = '\x01' ;  
short int p = 10 ;
```

- Quels sont le type et la valeur de chacune des expressions suivantes :

`p + 3`

`c + 1`

`p + c`

`3 * p + 5 * c`

Exercice 2

- Soient les déclarations :

```
char c = '\x05' ;
```

```
int n = 5 ;
```

```
long p = 1000 ;
```

```
float x = 1.25 ;
```

```
double z = 5.5 ;
```

- Quels sont le type et la valeur de chacune des expressions suivantes :

```
n + c + p
```

```
2 * x + c
```

```
(char) n + c
```

```
(float) z + n / 2
```

Exercise 3

```
#include <iostream>
using namespace std ;
main()
{
    int n=10, p=5, q=10, r ;

    r = n == (p = q) ;
    cout << "A : n = " << n << " p = " << p << " q = " << q
         << " r = " << r << "\n" ;

    n = p = q = 5 ;
    n += p += q ;
    cout << "B : n = " << n << " p = " << p << " q = " << q << "\n" ;

    q = n < p ? n++ : p++ ;
    cout << "C : n = " << n << " p = " << p << " q = " << q << "\n" ;

    q = n > p ? n++ : p++ ;
    cout << "D : n = " << n << " p = " << p << " q = " << q << "\n" ;
}
```


Exercice 4

```
#include <iostream>
using namespace std ;
main()
{  int n=0 ;
   do
   {  if (n%2==0) { cout << n << " est pair\n" ;
                  n += 3 ;
                  continue ;
                }
     if (n%3==0) { cout << n << " est multiple de 3\n" ;
                  n += 5 ;
                }
     if (n%5==0) { cout << n << " est multiple de 5\n" ;
                  break ;
                }
     n += 1 ;
   }
   while (1) ;
}
```

Exercise 5

```
#include <iostream>
using namespace std ;
main()
{   int i, n ;

    for (i=0, n=0 ; i<5 ; i++) n++ ;
    cout << "A : i = " << i << " n = " << n << "\n" ;

    for (i=0, n=0 ; i<5 ; i++, n++) {}
    cout << "B : i = " << i << " n = " << n << "\n" ;

    for (i=0, n=50 ; n>10 ; i++, n-=i ) {}
    cout << "C : i = " << i << " n = " << n << "\n" ;

    for (i=0, n=0 ; i<3 ; i++, n+=i,
        cout << "D : i = " << i << " n = " << n << "\n" ) ;
    cout << "E : i = " << i << " n = " << n << "\n" ;

}
```

Exercice 6

- Écrire un programme qui calcule les racines carrées de nombres fournis en entrée (cin). Il s'arrêtera lorsqu'on lui fournira la valeur 0. Il refuserait les valeurs négatives. Son exécution se présentera ainsi :
donnez un nombre positif : 2
sa racine carrée est : 1.414214e+00
donnez un nombre positif : -1
svp positif
donnez un nombre positif : 5
sa racine carrée est : 2.236068e+00
donnez un nombre positif : 0
- Rappelons que la fonction `sqrt` fournit la racine carrée (`double`) de la valeur (`double`) qu'on lui donne en argument.

Tableaux

- Créer un tableau de 10 float :
 - `float t [10];`
 - Accès : `t[0], t[1], ...`
- Créer un tableau de 5x3 float (deux dimensions) :
 - `float t [5][3];`
 - Accès : `t[0][0], t[0][1], ...`
- Idem pour 3, 4, 5, ..., n dimensions

Tableaux : initialisation

```
int t1[5] = { 10, 20, 5, 0, 3 } ;
```

```
int t2 [5] = { 10, 20 } ;
```

```
int tab [3] [4] = { { 1, 2, 3, 4 },  
                    { 5, 6, 7, 8 },  
                    { 9, 10, 11, 12 } } ;
```

```
int tab [3] [4] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 } ;
```

Les pointeurs, les opérateurs & et *

```
int * adi ;  
// adi contiendra des adresses d'entiers de type int  
  
float * adf ;  
// adf contiendra des adresses de flottants de type float  
  
int n ;  
.....  
adi = &n ; // adi contient l'adresse de n  
  
int p = *adi ; // place dans p la valeur de n  
*adi = 40 ;  
// place la valeur 40 à l'adresse contenue dans adi,  
// donc ici dans n
```


Opérations sur les pointeurs

- Les pointeurs peuvent être manipulés pour avancer / reculer dans la mémoire, en suivant leur type.

```
int n = 5;  
int * adi = &n;  
adi++; // va avancer de 4 octets
```

```
char c = 5;  
char * adc = &c;  
adc--; // va reculer d'un octet
```

Tableaux et pointeurs

- Un nom de tableau n'est qu'une constante pointeur.

```
int t[10];  
cout << t; // affiche l'adresse contenue dans le pointeur  
t+1; // équivalent à &t[1]  
*(t+i); // équivalent à t[i]
```

```
int t2[3][4];  
t; // équivalent à &t[0][0] ou t[0]  
t+2; // équivalent à &t[2][0] ou t[2]
```

Gestion dynamique de la mémoire

```
int n = 5; // crée automatique une place pour un entier
           // lors de l'ouverture du bloc qui le contient
```

```
int *adn = new int; // crée dynamiquement une place
                   // pour un entier
```

```
*adn = 5;          // utilisation
delete adn;        // libération de la place allouée
adn = nullptr;     // bonne pratique
```

Gestion dynamique de la mémoire

```
int t[5]; // crée automatiquement 5 places contiguës  
         // (lors de l'ouverture du bloc qui le contient)
```

```
int *adt = new int[5]; // crée dynamiquement une place  
                      // pour un entier
```

```
*adt = 5;           // utilisation (première position)  
*(adt+2) = 10;       // utilisation (troisième position)  
delete[] adt;        // libération des 5 places allouées  
adt = nullptr;       // bonne pratique
```

Chaîne de caractères

```
char ch[20] = "Bonjour";
```

```
char ch[20] = { 'B', 'o', 'n', 'j', 'o', 'u', 'r', '\0' };
```

```
// les deux instructions sont équivalentes
```

```
// Que veut donc dire main(int argc, char * argv[]) ?
```

Exercise 7

```
#include <stdio.h>

#include <iostream>
using namespace std ;

main()
{
    int t [3] ;
    int i, j ;
    int * adt ;

    for (i=0, j=0 ; i<3 ; i++) t[i] = j++ + i ;

    for (i=0 ; i<3 ; i++) cout << t[i] << " " ;
    cout << "\n" ;

    for (i=0 ; i<3 ; i++) cout << *(t+i) << " " ;
    printf ("\n") ;

    for (adt = t ; adt < t+3 ; adt++) cout << *adt << " " ;
    cout << "\n" ;

    for (adt = t+2 ; adt>=t ; adt--) cout << *adt << " " ;
    cout << "\n" ;
}
```


Exercice 8

- Ecrire, de deux façons différentes, un programme qui lit 10 nombres entiers dans un tableau avant d'en rechercher le plus grand et le plus petit :
 - en utilisant uniquement le « formalisme tableau »
 - en utilisant le « formalisme pointeur » , à chaque fois que cela est possible.

Exercise 9

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

main()
{   int t[4] = {10, 20, 30, 40} ;
    int * ad [4] ;
    int i ;
    for (i=0 ; i<4 ; i++) ad[i] = t+i ;
    for (i=0 ; i<4 ; i++) cout << * ad[i] << " " ;
    cout << "\n" ;
    cout <<  * (ad[1] + 1) << " " << * ad[1] + 1 << "\n" ;
}
```

Des questions ?