Langage C++

Sylvain GROSDEMOUGE

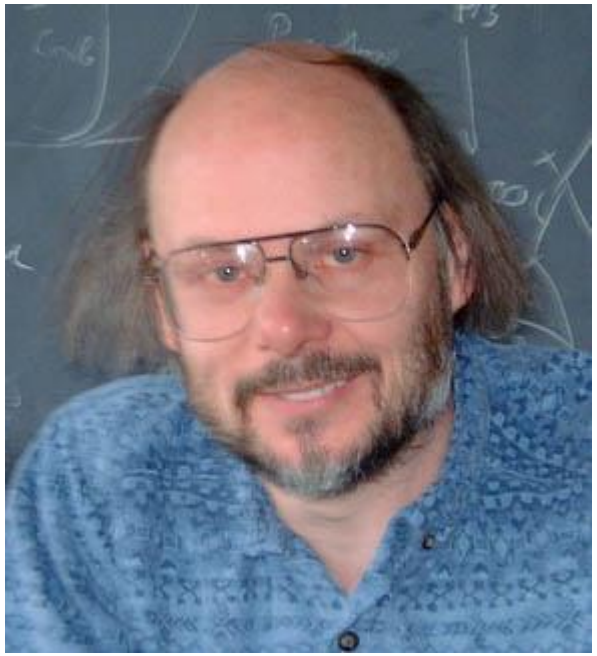
Historique

Développé par Bjarne Stroustrup au cours des années 1980, alors qu'il travaillait dans le laboratoire de recherche Bell d'AT&T.

Amélioration du langage C :

- notion d'objets (classes)
- héritage
- fonctions virtuelles
- surcharge des opérateurs
- exceptions
- templates

...



Références

The C++ Programming Language - Bjarne Stroustrup
Addison-Wesley
1986 (1st edition)
1991 (2nd edition)
1997 (3rd edition)
2000 (special edition)

Effective C++ - Scott Meyers
Addison-Wesley
1992 (1st edition)
1998 (2nd edition)
2005 (3rd edition).

Langage – Formalisme

Déclarations : dans des fichiers **.h** (ou .hh)
Implémentation : dans des fichiers **.cpp** (ou .cc)
Inlines : dans des fichiers **.inl**

Exemple :

Add.h

```
int Add(int a, int b);
```

Add.cpp

```
#include « Addition.h »

int Add (int a, int b)
{
        return a+b;
}
```

Langage – Processus de compilation

2 phases avant l'obtention d'un exécutable

- Préprocesseur / Compilateur

  Compilation des .cpp en des .o :
  *g++ -c *.cpp*

- Linker

  Liaison des .o en un exécutable :
  *g++ *.o*

Langage – Types de base

TYPE	TAILLE (en octets)	VALEUR MIN	VALEUR MAX
bool	1	false	true
char	1	-128	127
unsigned char	1	0	255
short	2	-32768	32767
unsigned short	2	0	65535
int	4	-2147483648	2147483647
unsigned int	4	0	4294967295
long	8	$-2^{e}64$	$2^{e}64 - 1$
unsigned long	8	0	$2^{e}64$
float	4	$+/- 3.4*10^{e}-38$	$+/- 3.4*10^{e}38$
double	8	$+/- 1.7*10^{e}-308$	$+/- 1.7*10^{e}308$

Langage – Commentaires

Deux façon de commenter :

```
// Commentaire sur une seule ligne
```

Tout ce qui se trouve sur la même ligne après // est ignoré par le préprocesseur

```
/* Commentaire
     sur plusieurs lignes */
```

Tout ce qui se trouve entre /* et */ est ignoré par le préprocesseur.

Langage – Mots clés

Mots clés = mots qui ne peuvent et ne doivent être en aucun cas utilisés autrement que pour ce à quoi ils sont destinés. Ils ont une signification particulière pour le compilateur.

Exemples :

Mots clés	Utilisation
+, -, /, *, =, ., <, <=, ==, >=, > &&, ||, &, |, !, != , ()(parenthèses), [] (Crochets)… (Triples Points), , (Virgule)	Opérateurs
&, *	Opérateur de référencement/déréferencement (référence/pointeur)
#	Préfixe de directives préprocesseur
0x (Zéro-x)	Préfixe de nombre hexadécimal
0 (Chiffre zéro)	Préfixe de nombre octal
{} (Accolades)	Délimitation de portée
:: (double deux points)	Opérateur de déréférencement de portée

Langage – Mots clés

Mots clés	Utilisation
asm	Déclarateur de code assembleur !
auto	Déclarateur de variable à désallocation automatique "Pile"(stack).
break	Instruction de branchement dans une boucle ou un traitement de cas.
bool	Type de donnée logique dit "booléen". Prend la valeur vrai (true) ou faux (false).
case	Déclarateur de cas dans une instruction switch.
catch	Récupérateur d'erreur.
char	Type de donnée entier dit "caractère". En programmation structurée ce type est déconseillé à l'utilisation mais il permet de rendre certains services.
class	Déclarateur de définition de classe.
const	Déclarateur de constantes.
continue	Instruction de branchement dans une boucle imbriquée.
default	Déclarateur de cas par défaut dans une instruction switch.
delete	Désallocateur de mémoire dynamique "Tas"(heap).
do	Déclarateur de boucle. Ne peut être utilisé qu'en association avec while.
double	Type de donnée nombre flottant à double précision.

Langage – Mots clés

Mots clés	Utilisation
else	Déclarateur de cas par défaut dans une instruction if.
enum	Type de donnée énuméré.
extern	Déclarateur d'une variable déclarée dans un autre fichier.
explicit	Interdit les constructeurs pour casts implicites.
false	Valeur logique (Faux)
float	Type de donnée nombre flottant à simple précision.
for	Déclarateur de boucle paramétrée.
friend	Déclarateur de classe ou de fonction ayant accès aux données privées.
goto	Instruction de branchement, en développement structuré son utilisation est interdite car elle rend la compréhension du code plus difficile.
if	Déclarateur de traitement conditionnel.
inline	Déclarateur de MACRO
int	Type de donnée entier. En programmation structurée ce type est déconseillé à l'utilisation mais il permet de rendre certains services.
long	Modificateur de longueur de type.
main	Méthode point d'entrée du programme.

Langage – Mots clés

Mots clés	Utilisation
mutable	Rend une partie d'un objet constant modifiable.
new	Allocateur de mémoire dynamique "Tas"(heap).
operator	Déclarateur de surcharge d'opérateur.
private	Déclarateur de membre privé.
protected	Déclarateur de membre protégé.
public	Déclarateur de membre public.
register	Déclarateur de variable registre.
return	Instruction de branchement.
short	Modificateur de longueur de type.
signed	Modificateur d'interprétation de signe de type entier.
sizeof	Opérateur spécial permettant de renvoyer la taille d'une variable stockée en pile(stack).
static	Déclarateur de variable statique dite "de classe". En programmation structurée, il est déconseillé à l'utilisation mais permet de rendre certains services.
struct	Déclarateur de structure. En programmation structurée il est déconseillé à l'utilisation il est préférable d'utiliser des classes à la place.
switch	Déclarateur de traitement conditionnel cardinal.

Langage – Mots clés

Mots clés	Utilisation
template	Declarateur de paramétrage.
this	Pointeur spécial désignant l'instance en cours de l'objet. En programmation structurée il est systématiquement et presque obligatoirement utilisé car il améliore la lisibilité.
throw	Déclencheur d'exceptions.
try	Déclarateur de section à déclenchement d'exceptions.
true	Valeur logique (Vrai)
typedef	Déclarateur de type.
unsigned	Modificateur d'interprétation de signe de type entier.
union	Déclarateur d'union. En programmation structurée, il est très fortement déconseillé à l'utilisation il n'est utilisé que dans des cas très rares et pour des applications très spécifiques.
virtual	Déclarateur de méthode virtuelle.
void	Indicateur d'une absence de type (là où un type serait attendu).
volatile	Déclarateur de membre critique nécessitant un traitement d'actualisation particulier notamment lors de l'utilisation de threads.
while	Déclarateur de boucle conditionnelle.
include, define, ifdef, ifndef, pragma, error	Directives préprocesseur
malloc, realloc, calloc, free	Opérateurs C pour l'allocation / la désallocation de mémoire dynamique

Langage – Directives préprocesseur

#include : inclusion d'un fichier dans un autre

Exemple :

```
#include <stdio.h>
#include <math.h>

#include "point.h"
#include "triangle.h"
#include "rectangle.h"
```

…

Langage – Directives préprocesseur

#if / #ifdef / #ifndef : Conditionnement de la compilation à la valeur qui suit l'instruction

```
#define _USE_STL 0

#if _USE_STL
#      include <vector>
#      include <map>
#else
#      include <myvector>
#      include <mymap>
#endif // _USE_STL
```

Langage – Directives préprocesseur

#error : affichage d'une erreur sur la ligne de commande

```
#ifndef CURRENT_PLATFORM
#      error "CURRENT_PLATFORM has to be defined !"
#endif // CURRENT_PLATFORM
```

→ Arrêt du processur de pré-compilation

Langage – Directives préprocesseur

#pragma : affichage d'un message lors de la pré-compilation

```
#pragma message("fixme: circular dependency when including
application.h")
#include "application.h"
```

Langage – Premier programme (C)

Addition.h :

```
int Addition(int a, int b);
```

Addition.cpp :

```
#include "Addition.h"
int Addition(int a, int b)
{
	return(a+b);
}
```

main.cpp :

```
#include <stdio.h>
#include "Addition.h"

void main(void)
{
	int a = 2;
	int b = 4;
	printf("%d + %d = %d\n", a, b, Addition(a,
b));
	return(0);
}
```

Langage – Variables + Constantes

Exemple :

```
bool b = false;

int a = 1;

float f = 1.2f;

double very_small = 0.000000012354534;

const float fPi = 3.141592f;
```

Langage – Namespaces

Permet d'isoler une partie du code source et d'éviter certains conflits.

Exemple :

```
#include <stdio.h>

namespace n1
{
        void func()
        {
                printf("Inside first_space\n");
        }
}

namespace n2
{
        void func()
        {
                printf("Inside second_space\n");
        }
}

int main()
{
        n1::func(); // Appel de la fonction définie dans 'n1'
        n2::func(); // Appel de la fonction définie dans 'n2'

        return 0;
}
```

Langage – Méthodes

Définition d'une méthode :

```
void          method_name          (void);
type_name     method_name          (arg_1, arg_2, …, arg_n);
```

Langage – Surcharge

Polymorphisme :

```
void method(int arg_1);
void method(int arg_1, int arg_2);
```

→

```
method(1);
method(1, 2);
```

Langage – Références

Référence : &

```
int a = 2;                  // Définition d'une variable a
                            // et affectation de la valeur 2

printf(«%d\n», a);          // Affiche «2»

int & aref = a;             // Définition d'une référence à la variable a

printf(«%d\n», aref);       // Affiche «2»

aref = 3;                   // Affectation de la valeur 3 à la
                            // variable référencée par aref

printf(«%d\n», a);          // Affiche « 3 »
printf(«%d\n», aref);       // Affiche « 3 »
```

Langage – Pointeurs

Pointeur : *

```
int a = 2;                  // Définition d'une variable a
                            // et affectation de la valeur 2

printf(«%d\n», a);          // Affiche «2»

int * ptra = &a;            // Définition d'un pointeur sur la variable a

printf(«%d\n», *ptra);      // Affiche «2»

*ptra = 3;                  // Affectation de la valeur 3 à la
                            // variable référencée par aref

printf(«%d\n», a);          // Affiche « 3 »
printf(«%d\n», *ptra);      // Affiche « 3 »
```

Mémoire – Tas / Pile (Heap / Stack)

Allocations sur la pile (ou 'Stack') : Utilisées pour stocker les données temporaires au scope courant .

Allocations sur le tas (ou 'Heap') : Utilisées pour stocker les données persistantes.

Allocations sur la pile (Stack)

Allocation d'un élément :

```
{
        // Allocation sur la pile d'un entier
        int i = 2;
}
// Dès la sortie du scope les allocations temporaires sur la pile sont détruites
```

Allocation d'un tableau de plusieurs éléments :

```
{
        // Allocation sur la pile de 4 éléments
        int aInt[4] = {0, 1, 2, 3};
}
// Dès la sortie du scope les allocations temporaires sur la pile sont détruites
```

Allocations sur le tas (Heap)

Allocation d'un élément :

```
{
        // Allocation sur le tas (Heap) à l'aide de la méthode new
        int * pInteger = new int(2);
}
// Les allocations dynamiques persistent une fois sorti du scope.

// Elles doivent être détruites à l'aide de la méthode delete
delete pInteger;
```

Pour un tableau de plusieurs éléments :

```
{
        int * pInteger = new int[2];
}

delete [] pInteger;
```

Notions d'objets – Classes (déclaration / instanciation)

Déclarations :

```
class CCarWheel
{
        …
};

class CCarBody
{
        …
};

class CCar
{
public:

        CCarBody        m_body;
        CCarWheel       m_aWheel[4];
}
```

Instanciation :

```
CCar myCar;
```

Notions d'objets – Classes (instanciation / destruction)

Instanciation sur la pile :

```
{
        // Instanciation de myCar sur la pile
        CCar myCar;

        // …
}
// myCar est détruit
```

Instanciation dynamique :

```
// Instanciation dynamique d'une instance de Ccar et récupération du pointeur vers cette instance
CCar * pCar = new CCar();

// L'objet pointé par pCar est détruit
delete pCar;
```

Notions d'objets – Classes (scopes)

```
class CObject
{
friend class CObjectFriend;

public:

        // Accessible depuis la classe, ses classes dérivées,
        // CObjectFriend et l'extérieur

protected:

        // Accessible depuis la classe, ses classes dérivées,
        // CObjectFriend

private:

        // Accessible depuis la classe uniquement

};
```

Notions d'objets – Classes (méthodes)

```
class CPoint
{
public:
                CPoint  (float x, float y)
                : m_x(x)
                , m_y(y)
                {

                }

                ~CPoint (void)
                {

                }

        void    SetX    (float x) { m_x = x; }
        void    SetY    (float y) { m_y = y; }

        float   GetX    (void) const { return m_x; }
        float   GetY    (void) const { return m_y; }

protected:

private:

        float m_x, m_y;

};
```

Notions d'objets – Héritage

```
class CAnimal
{
        // …
};

class CAnimalCat : public CAnimal
{
        // …
};

class CAnimalDog : public CAnimal
{
        // …
};
```

Notions d'objets – Héritage multiple

```
class CAnimal
{
        // …
};

class CAIEntity
{
        // …
};

class CAnimalCat : public CAnimal, public CAIEntity
{
         // …
};

class CAnimalDog : public CAnimal , public CAIEntity
{
         // …
};
```

Notions d'objets – Héritage (méthodes virtuelles)

```
class CAnimal
{
	virtual void MakeNoise(void) { }
};

class CAnimalDog : public CAnimal
{
	virtual void MakeNoise(void) { printf(«Waf\n»); }
};

class CAnimalCat : public CAnimal
{
	virtual void MakeNoise(void) { printf(«Mahooow\n»); }
};

CAnimalDog dog;
CAnimalCat cat;
Dog.MakeNoise();
Cat.MakeNoise();
```

→

```
Waf
Mahooow
```

Notions d'objets – Héritage (méthodes virtuelles pures)

```
class CAnimal
{
        virtual void MakeNoise(void) = 0;
};

class CAnimalDog : public CAnimal
{
//      virtual void MakeNoise(void) { printf(«Waf\n»); }
        → ERREUR DE COMPILATION
};

class CAnimalCat : public CAnimal
{
        virtual void MakeNoise(void) { printf(«Mahooow\n»); }
};
```

Une **méthode virtuelle pure**, déclarée dans la classe de base avec '=0' ***doit*** **être implémentée dans toutes les classes dérivées**.

Notions d'objets – Surcharge d'opérateurs (à l'intérieur de la classe)

```
class CVector2
{
public:

        CVector2 operator * (const CVector2 & v2)
        {
                CVector2 vRet;

                vRet.m_x = m_x * v2.m_x;
                vRet.m_y = m_y * v2.m_y;

                return vRet;
        }

        float   m_x, m_y;
}
```

Notions d'objets – Surcharge d'opérateurs (à l'extérieur de la classe)

```
class CVector2
{
public:

        float   m_x, m_y;
}

CVector2 operator * (const CVector2 & v1, const CVector2 & v2)
{
        CVector2 vRet;

        vRet.m_x = v1.m_x * v2.m_x;
        vRet.m_y = v1.m_y * v2.m_y;

        return vRet;
}
```

Exceptions

```
try
{
    throw EXCEPTION;
}
catch (EXCEPTION)
{
    printf(«Exception occured !\n»);
}
```

EXCEPTION peut-être de n'importe quelle nature :

```
throw 18;                    // code d'erreur n° 18
throw «Fatal error !»;       // message d'erreur
throw CError(e_error_18)     // Instance de la classe CError avec comme
                             // code d'erreur 18 (la classe CError
                             // permettant de transcrire ce code dans
                             // une string localisée par exemple)
```

Exceptions

Pour quoi faire ?

Pour annuler l'exécution à tout moment, et sortir avec un code d'erreur … Y compris d'un constructeur

Dans les faits : Alourdit le code généré (jusqu'à 30% de perte mesurées en temps CPU) – à éviter dans les applications temps-réel.

Fonctions template

```
template <class T>
inline T myMin(const T & a, const T & b)
{
	return((a < b) ? a : b);
}

template <class T>
inline T myMax(const T & a, const T & b)
{
	return((b < a) ? a : b);
}

template <class T>
inline T myMin(const T & a, const T & b, const T & c)
{
	return((myMin(a,b) < c) ? myMin(a,b) : c);
}
```

Classes template

```
template <class T>
class TColor
{
public:

        …

        inline void    SetR    (T r)           { m_r = r; }
        inline T       GetR    (void) const    { return(m_r); }

        …

private:

        T m_r;
        T m_g;
        T m_b;

};
```