

Langage C++

CPP023



Copyright © 2012 – Éric Robert

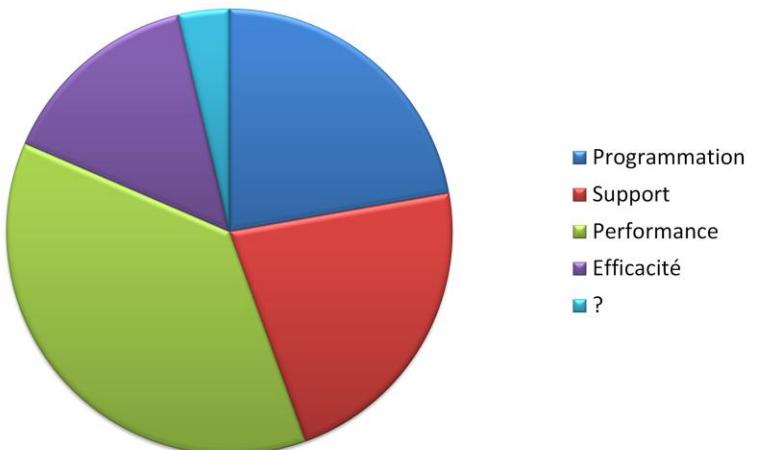


Présentations

- Éric Robert
 - Spécialités & expériences
 - C++, temps-réel, performance, architecture
 - NUBO
 - Services de consultation, coaching, formation...
 - Contacts :
 - www.nubo.ca (1.855.I.GO.NUBO)
 - eric.robert@nubo.ca
- Et vous ?

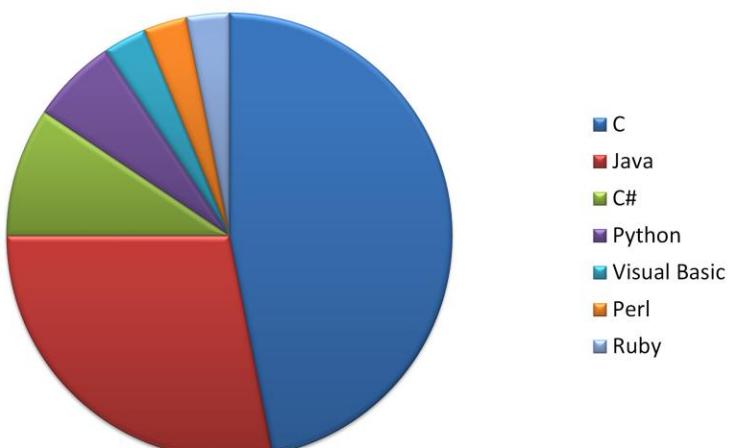


Motivation



 **nubo**
ENCADREMENT LOGICIEL

Expérience



 **nubo**
ENCADREMENT LOGICIEL

Logistique

- CRIM
 - Salle de bain, cellulaire, lien sans fil, poste de travail, coin détente, café-jus, vestiaire
- Horaire (2 jours, 12 heures)
 - 9h00 à 16h30
 - Pause du matin et de l'après-midi (~15 minutes)
 - Lunch de 12h00 à 13h30



Objectif

- Développer et comprendre des applications simples en C++
- Bref :
 - Avoir une connaissance générale du C++
 - Utiliser les outils de développement
 - Appliquer ces principes dans le développement d'une application simple en C++

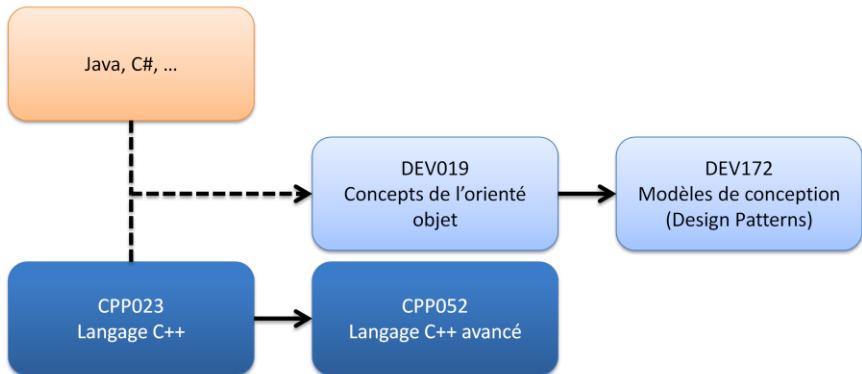


Méthodologie

- Formation axée sur la pratique
 - Exemples de code
 - Environnement de développement i.e. *IDE*
 - Projet C++
 - 10 itérations



Cheminement



Introduction

- Utilisations du langage C++
 - Performance
 - Domaines d'applications : Applicatif, système, embarqué, pilotes de périphériques, serveurs haute performance, multimédia, jeux vidéo...
 - Code existant i.e. *legacy code*
- Règles de design du C++
 - Extensions compatibles avec le C
 - Efficace et portable
 - Principe sans charge i.e. *zero-overhead principle*



C++

- Points forts
 - Langage compilé i.e. natif
 - Simplifie et favorise l'utilisation d'abstractions
 - Design orienté objet (OO)
 - Découplage
 - Efficace et maintenable
 - Typage statique i.e. *statically typed*
 - Templates
 - Extensible



C++

- Points faibles
 - Librairie standard
 - Librairie d'exécution i.e. *runtime*
 - STL
 - Complexité
 - Permet de faire de mauvais choix
 - Permet les abus
 - Évolue lentement
 - Compilateurs, extensions et compatibilité
 - C++98 à C++11



GIT

- Système distribué de contrôle des versions
- Quelques commandes :
 - \$ git log --oneline master
 - \$ git checkout 4ff326b
 - \$ git checkout master
 - \$ git add
 - \$ git commit
 - \$ git status



Exercice Dirigé

Hello World

d0d8ffa - 4ff326b

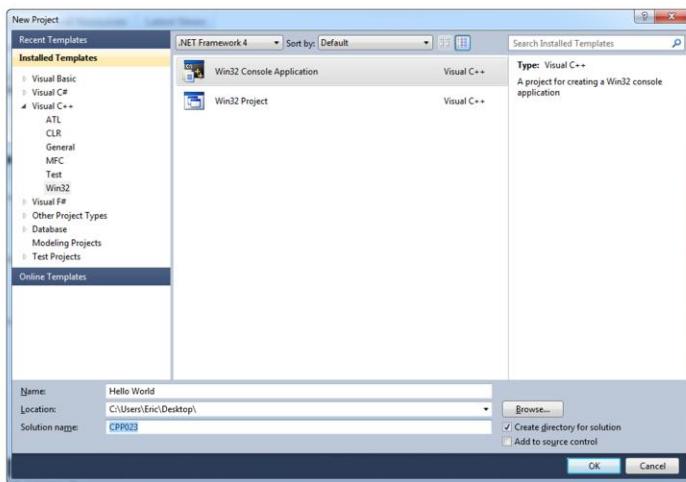


Microsoft Visual Studio 2010

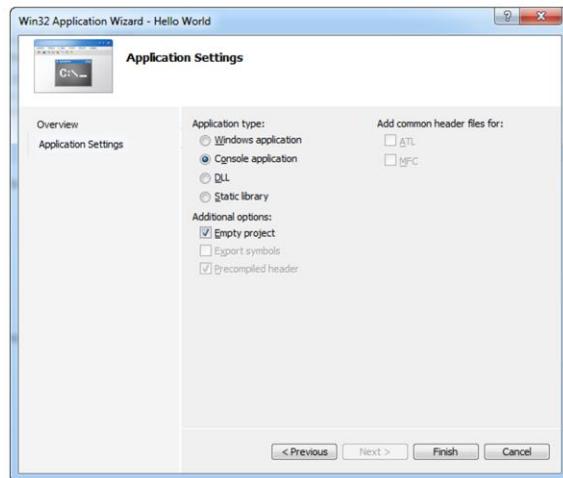


 **nubo**
ENCADREMENT LOGICIEL

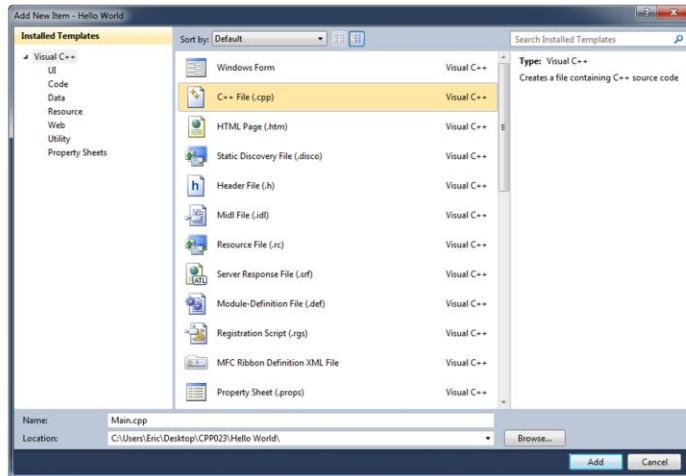
Projet



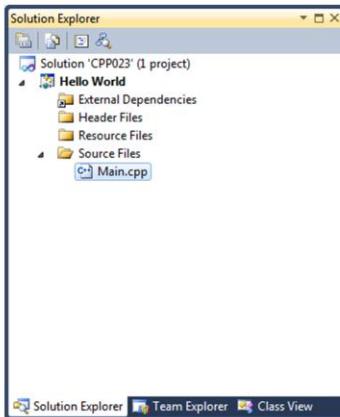
Configuration



CPP



Solution



Hello, World

```
#include <stdio.h>

int main() {
    printf("hello, world\n");
}
```



Hello, World

```
#include <iostream>

int main() {
    std::cout << "hello, world" << std::endl;
}
```



Variables

- Une variable est une instance d'un type
e.g. int i;
- Une instance possède une adresse mémoire
e.g. &i
- Un type détermine l'interprétation d'un espace mémoire donné
e.g. sizeof(int);
sizeof(i);



Entiers

```
int a; // indéfinie
int b = 0;
int c = a; // indéfinie
int d = -1;
int e(d);
int f = b * d + e * 2;
signed int g = f;
signed h = f;
unsigned int i = 0;
unsigned j = i - 1;
int k, l, m = 1; // indéfinie pour k, l
```

Name	Value	Type
a	-858993460	int
b	0	int
c	-858993460	int
d	-1	int
e	-1	int
f	-2	int
g	-2	int
h	-2	int
i	0	unsigned int
j	4294967295	unsigned int
k	-858993460	int
l	-858993460	int
m	1	int



- a. Définition d'entier
- b. Définition avec initialisation
- c. Warning
- d. Le type int est signé
- e. Initialisation avec la notation de constructeur
- f. Initialisation avec une expression plus complexe (voir la précédence d'opérateurs)
- g. Utilisation explicite de signed – idem à d
- h. Utilisation implicite de int – idem à g
- i. Utilisation explicite de unsigned qui encode les entiers sans signes i.e. positifs
- j. Utilisation implicite de int
- k. Définitions multiples sur la même ligne

Entiers

```
int main() {  
    int i; // valeur indéfinie  
    i = 0;  
}
```

F10

Name	Value	Type
i	-858993460	int
&i	0x003efafc	int *

Name	Value	Type
i	0	int
&i	0x003efafc	int *



Commandes utiles pour le débuggeur :

- F9 : Breakpoint
- F10 : Step
- F11 : Step Into
- Shift-F11 : Step Out

Entiers

```
short a = 256;
signed short int b = a * a;
int c = a + b; // short ≤ int
unsigned d = a;
long e = d * d; // int ≤ long
short f = e; // oups!
int g = f * e; // oups?
unsigned long h = -1;
long long i = e;
unsigned long long int j = -1;
```

Name	Value	Type
a	256	short
b	0	short
c	256	int
d	256	unsigned int
e	65536	long
f	0	short
g	0	int
h	4294967295	unsigned long
i	65536	_int64
j	18446744073709551615	unsigned _int64



- a. Définition d'entier court
- b. Utilisation explicite de signed et de int
- c. Conversion implicite
- d. Utilisation implicite de int avec conversion implicite
- e. Conversion implicite
- f. Warning
- g. Warning ?
- h. Représentation maximale sur 32 bits
- i. Support 64 bits (C++11)
- j. Représentation maximale sur 64 bits

Entiers

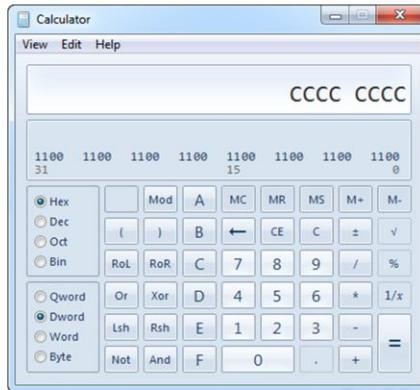
```
int a = sizeof(short);
int b = sizeof(int);
int c = sizeof(unsigned int);
int d = sizeof(signed int);
int e = sizeof(long);
int f = sizeof(long long);
int g = 1 << (a * 8);
int h = 1 << (a * 8 - 1);
short i = 32767;
short j = 32768;
```

Name	Value	Type
a	2	int
b	4	int
c	4	int
d	4	int
e	4	int
f	8	int
g	65536	int
h	32768	int
i	32767	short
j	-32768	short



- a. 2 bytes
- b. 4 bytes
- c. 4 bytes
- d. 4 bytes
- e. 4 bytes
- f. 8 bytes
- g. Quantité de nombres représentables sur 2 bytes
- h. Quantité de nombres représentables sur 2 bytes signé
- i. Valeur positive maximale sur 2 bytes
- j. Valeur suivante en complément 2 i.e. négative maximale sur 2 bytes

-858993460 ?



Quelques motifs mémoire typiques sous Windows :

- 0xbaadf00d après HeapAlloc
- 0xffffeeee après HeapFree
- 0xabababab gardes autour de l'allocation par HeapAlloc
- 0xcdcdcdcd après malloc
- 0xfdfffffd gardes autour de l'allocation par malloc
- 0xffffffff après free

Entiers

- Un entier peut déborder i.e. *overflow*
 - Échec silencieux i.e. *wrap around*
0 ... 65535 ... 0
0 ... 32767 -32768 ... 0
- Sa taille est définie par le compilateur
 - Garanties : short \leq int \leq long \leq long long
 - Garanties en bits :
short (16), int (16), long (32), long long (64)



Réels

```
float a = 3.1415f;  
float b = 1 / a;  
float c = a * b - 1.0f;  
float d = a + 2 * c;  
double e = 355.0 / 113.0;  
double f = e - a;  
double g = f / 0;  
double h = g / g;  
double i = 1e-10;  
double j = 5.0 + (1 / 5);
```

Name	Value	Type
a	3.1415000	float
b	0.31831926	float
c	-4.2550113e-008	float
d	3.1415000	float
e	3.141592903539825	double
f	9.2924168679786590e-005	double
g	1.#INF00000000000000	double
h	-1.#IND00000000000000	double
i	1.00000000000000e-010	double
j	5.00000000000000	double



- a. Précision simple
- b. Conversion implicite d'entiers en précision simple
- c. Attention à la précision i.e. = 0 ?
- d. Attention à la précision i.e. = a ?
- e. Précision double
- f. Conversion implicite
- g. Division par zéro sans exceptions
- h. Calcul invalide
- i. Représentation scientifique
- j. Règles de conversion implicite d'entiers

Réels

- Les réels sont une approximation
 - Norme IEEE 754
 - Signe + Exposant + Mantisse
 - NaN, INF, ...
 - Précision (\sim) en nombre de chiffres significatifs
 - Calcul de l'erreur par analyse numérique
- Sujet en soi :
[What Every Computer Scientist Should Know About Floating Point Arithmetic](#) par David Goldberg



Mémoire

- Adressage
 - Par byte (1 octet = 8 bits)
 - Modèles de mémoire
 - Linéaire i.e. *flat memory model*
 - Paginée
 - Segmentée
 - Espace d'addressage i.e. *address space*
 - Physique
 - Virtuel e.g. par processus



Mémoire

- Types d'allocations
 - Statique
 - Alloué avant même l'exécution
 - Initialisé à 0 par défaut
 - Dynamique
 - Alloué par le programmeur (new, malloc)
 - Et à libérer (delete, free)
 - Pile i.e. *stack*
 - Alloué par le compilateur



Mémoire

00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
10	11	12	13	14	15	16	17	18	19	1a	1b	1c	1d	1e	1f
20	21	22	23	24	25	26	27	28	29	2a	2b	2c	2d	2e	2f
30	31	32	33	34	35	36	37	38	39	3a	3b	3c	3d	3e	3f
40	41	42	43	44	45	46	47	48	49	4a	4b	4c	4d	4e	4f
50	51	52	53	54	55	56	57	58	59	5a	5b	5c	5d	5e	5f
60	61	62	63	64	65	66	67	68	69	6a	6b	6c	6d	6e	6f
70	71	72	73	74	75	76	77	78	79	7a	7b	7c	7d	7e	7f
80	81	82	83	84	85	86	87	88	89	8a	8b	8c	8d	8e	8f
90	91	92	93	94	95	96	97	98	99	9a	9b	9c	9d	9e	9f
a0	a1	a2	a3	a4	a5	a6	a7	a8	a9	aa	ab	ac	ad	ae	af
b0	b1	b2	b3	b4	b5	b6	b7	b8	b9	ba	bb	bc	bd	be	bf



1. Les (quelques) premières pages sont réservées et inaccessibles en lecture/écriture (rouge)
2. Le code exécutable (orange)
3. Les données statiques initialisées en lecture seule (jaune)
4. Les données statiques (vert)
5. La pile (bleu)
6. Tout le reste : disponible pour l'allocation dynamique

Pointeurs

```
int a = 0;
int * b = &a;
int * c = 0;
int d = *b;
int e = b[0];
int f[4]; // indéfinie
int g[4] = { 0 };
int h[] = { 0, 1, 2, 3 };
int * i = h;
int * j = &h[3];
```

Name	Value	Type
a	0	int
b	0x0034fe50	int *
c	0	int
d	0x00000000	int *
e	0	int
f	0x0034fe08	int [4]
g	0x0034fdf0	int [4]
[0]	0	int
[1]	0	int
[2]	0	int
[3]	0	int
h	0x0034fdd8	int [4]
[0]	0	int
[1]	1	int
[2]	2	int
[3]	3	int
i	0x0034fdd8	int *
j	0	int
[0]	0x0034fde4	int *
[1]	3	int



- a. Allocation sur le stack
- b. Pointeur sur a i.e. adresse de a
- c. Pointeur nul
- d. Déréférence de pointeur i.e. ce qui se trouve à l'adresse contenue dans b
- e. Notation indexée
- f. Allocation sur le stack d'un tableau de 4 int i.e. 16 bytes
- g. Allocation sur le stack d'un tableau de 4 int et initialisation à 0 des 4 entiers
- h. Allocation sur le stack d'un tableau implicitement de 4 int initialisés à 0, 1, 2 et 3
- i. Pointeur sur le tableau i.e. sur l'élément = 0
- j. Pointeur sur le 4^e élément du tableau i.e. sur l'élément = 3 (index 3)

Pointeurs

```
int a[] = { 0, 1, 2, 3 };
int b = sizeof(a);
int c = a[8]; // oups!
int * d = a + 4;
int e = *d; // oups!
int f = d[-1];
int g = d - a;
int * h = d - 4;
int ** i = &h;
int j = (*i)[3];
```

Name	Value	Type
a	0x0038fc1c	int [4]
[0]	0	int
[1]	1	int
[2]	2	int
[3]	3	int
b	16	int
c	1	int
d	0x0038fc2c	int *
*	-858993460	int
e	-858993460	int
f	3	int
g	4	int
h	0x0038fc1c	int *
*	0	int
i	0x0038fbcb	int **
*	0x0038fc1c	int *
*	0	int
j	3	int



- a. Allocation sur le stack d'un tableau implicitement de 4 int initialisés à 0, 1, 2 et 3
- b. Taille du tableau
- c. Accès hors du tableau en notation indexée
- d. Arithmétique de pointeur positionné 1 élément après la fin i.e. $a + 4 == \&a[4]$
- e. Accès hors du tableau par déréférence
- f. Accès au dernier élément i.e. sur l'élément = 3
- g. Calcul de distance
- h. Arithmétique de pointeur positionné au début i.e. sur l'élément = 0
- i. Pointeur double i.e. pointeur sur une variable de type pointeur d'entier
- j. Déréférence du pointeur de pointeur puis indexation

Caractères

```
char a = 0;
char b = '0';
char c = 'A' + 32;
char * d = &c;
char * e = "hello";
char f = e[0];
char g = e[5];
char h[256]; // indéfinie
char i[] = "world";
int j = sizeof(i);
```

Name	Value	Type
a	0	char
b	48 '0'	char
c	97 'a'	char
d	0x0043fe97 "a" [16]	char *
e	97 'a'	char
f	0x010757a8 "hello" [16]	char *
g	104 'h'	char
h	0	char
i	0x0043fd5c "world" [16]	char [256]
j	0x0043fd4d "world" [6]	char [6]
[0]	119 'w'	char
[1]	111 'o'	char
[2]	114 'r'	char
[3]	108 'l'	char
[4]	100 'd'	char
[5]	0	char
j	6	int



- a. Entier de 1 byte (caractère)
- b. Utilisation de la notation ANSI pour spécifier la valeur de l'entier – caractères spéciaux e.g. \" , \' , \t , \n , \r , \b , \\
- c. Conversion en minuscule
- d. Pointeur sur un caractère – convention C pour une chaîne de caractères i.e. *string*
- e. Pointeur sur le premier caractère d'une chaîne de caractères statique
- f. Pointe sur le caractère h
- g. Pointe sur le caractère 0 (null terminated string)
- h. Allocation d'un buffer sur le stack
- i. Allocation et remplissage d'un tableau implicitement de la même taille que le littéral
- j. Taille incluant le 0 implicite

Table ASCII

	0	1	2	3	4	5	6	7
0	NUL	DLE	space	Ø	@	P	`	p
1	SOH	DC1	XON	!	1	A	Q	a
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	\$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(8	H	X	h	x
9	HT	EM)	9	I	Y	i	y
A	LF	SUB	=	:	J	Z	j	z
B	VT	ESC	+	;	K	[k	{
C	FF	FS	.	<	L	\	l	
D	CR	GS	-	=	M]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	del



Notes :

1. Le code ASCII est défini de 0 à 127
2. Le code ASCII étendu est défini de 128 à 255
3. Nombres : 0x30 + N
4. Alphabet : 0x41/0x61 + i
5. Le code ANSI lui est identique au code ASCII de 0 à 127 mais diffère sur la partie étendue

Pointeurs

- Bête noire ?
 - Fuites de mémoire i.e. *memory leak*
 - Pointeur fou i.e. *wild pointer, dangling pointer*
- Arithmétique de pointeurs
 - Déplacement avant/arrière selon la taille du type
 - Calcul de distance
- Convention pour les chaînes de caractères
 - Pointeur de caractère
 - Terminée par 0



Types

Type	Taille*	Notes
char	1	-1, 0, 1, 'c'
short int	2	(short) 2
int	4	4
long int	4	4L
long long int	8	8LL (C++11)
bool	1	true, false
float	4	0.1f
double	8	0.1
long double	8	0.1L
wchar_t	2	-1, 0, 1, L'c'
T *	4	&x



* Exemples de tailles déterminées sur un système (architecture, compilateur et options) en particulier

Opérateurs

- Arithmétique

$a + b, a - b, a * b, a / b, a \% b$

$+a, -a$

$++a, a++, --a, a--$

- Binaire

$a \& b, a | b, a ^ b$

$a << b, a >> b$

$\sim a$



Opérateurs

- Assignation

$a = b$

$a += b, a *= b, a /= b, a \%= b$

$a &= b, a |= b, a ^= b$

$a <<= b, a >>= b$

- Pointeurs

$\&a, *a, a[b]$



Projet formatif

- Jeu d'échecs
 - 1 seul mode de jeu : AI versus AI



 **nubo**
ENCADREMENT LOGICIEL

Jeu d'échecs (1)

- But :
 - Afficher l'échiquier initial
 - Utilisez
 - 1 caractère par case
 - MAJUSCULES (blanc), minuscules (noir)
 - La notation
 - P (pion), R (tour), N (chevalier), B (fou), Q (reine), K (roi)



Échiquier

	a	b	c	d	e	f	g	h	
8	r	n	b	q	k	b	n	r	8
7	p	p	p	p	p	p	p	p	7
6									6
5									5
4									4
3									3
2	P	P	P	P	P	P	P	P	2
1	R	N	B	Q	K	B	N	R	1
	a	b	c	d	e	f	g	h	

Noir

Blanc



Exercice Pratique

Chess-01

4ff326b - 44af90c





Obfuscated C Mandelbrot and Julia-set Generator
Copyright 2000-2001 by Stijn Wolters

The International Obfuscated C Code Contest

<http://www.ioccc.org>

Fonctions

- Outil de structuration du code
 - Réutilisation du code
 - Division logique, pas toujours physique
- Anatomie :
 - Prototype
 - Type de valeur renvoyée, identificateur, paramètres
 - Corps
 - Instructions, contexte i.e. *scope*



Fonctions

```
// fonction globale
int square(int i) {
    return i * i;
}

// fonction représentant le point d'entrée du programme
int main() {
    int a = square(16);
    int b = square(square(a));
    square(a);
    int d = square(b) + 2 * square(a);
    int e = ++square(16); // oups!
    int * f = &square(a); // oups!
}
```



- a. Invocation
- b. Invocation chaînée
- c. Code mort
- d. Invocation avec précédence
- e. Invalide – la valeur de retour est une valeur temporaire (l-value required)
- f. Invalide – valeur temporaire sans adresse

Fonctions

```
void display() {
    std::cout << "Copyright (c)" << std::endl;
}

int divide(int i) {
    return i / 2;
}

int divide(int i, int j) {
    return i / j;
}

int main() {
    display();
    return divide(4, divide(4));
}
```



Notes :

1. Le mot clé void indique qu'il n'y a pas de type de retour
2. Les 2 fonctions divide sont indépendantes puisqu'elles n'ont pas la même liste de paramètres
3. La valeur retournée de la fonction main est transférée au OS

Fonctions

```
int divide(int i, int j = 2) {
    return i / j;
}

template<typename T>
T divide(T i, T j = 2) {
    return i / j;
}

int main() {
    int a = divide(100, 5);
    int b = divide(100);
    float c = divide(1.0f, 5.0f);
    float d = divide(1.0f, 5); // attention!
    double e = divide(1.0);
    double f = divide<float>(0);
}
```



- a. Invocation sans template (int) est préférée
- b. Invocation sans template (int) avec valeur par défaut
- c. Invocation implicite T=float
- d. Invocation sans template (int) préférée
- e. Invocation implicite T=double et paramètre par défaut
- f. Invocation avec T explicite et conversions implicites (int-float et float-double)

Fonctions

- Paramètres par défaut
 - Réservé aux N derniers paramètres
 - Valeurs constantes littérales
- Surcharge i.e. *overload*
 - Basée sur la signature
 - Ignore le type de la valeur retournée
 - Ignore la valeur des paramètres par défaut
- Template
 - Instanciation au point d'invocation



Contexte

- Structure hiérarchique
 - Global
 - Local ou imbriqués i.e. *nested*
- Défini la recherche de nom i.e. *name lookup*
 - Local vers global
- Gère la pile
 - Allocation & désallocation



Contexte

```
// variables globales
int a;
int b = 1;

int foo(int a) { // nouveau 'a' qui cache la variable globale
    b += a;
    return b;
}

int main() {
    ++a;
    int a = b; // nouveau 'a' qui cache la variable globale
    int b = foo(a); // nouveau 'b' qui cache la variable globale
    ::b += b;
}
```



Notes :

1. Les variables globales sont allouées statiquement i.e. initialisées à 0 par défaut
2. Les paramètres sont inclus dans le contexte de la fonction
3. Les variables peuvent être définies n'importe où dans le contexte
4. L'opérateur :: est l'opérateur de résolution de contexte et permet ici de référer au contexte global

Flux d'exécution

- Mots clés
 - if, else, while, do...while, for, switch
 - return, break, continue
- Opérateurs
 - a ? b : c
 - Opérateurs court-circuit i.e. *short-circuit*
 - a && b, a || b
- Pointeur de fonction
- Exception



Conditions

```
int strcmp(char * a, char * b) {
    if(!a) {
        return b ? -1 : 0;
    }

    if(!b) {
        return 1;
    }

    while(*a == *b && *a) {
        ++a;
        ++b;
    }

    return *a - *b; // comparaison lexicographique
}
```



Notes :

1. La fonction strcmp permet de trier les chaînes de caractères en utilisant la valeur de retour i.e. plus petit est négatif, 0 égal et plus grand est positif
2. L'instruction if exécute l'instruction suivante si la condition est vraie
3. Le else est optionnel et est exécuté si la condition est fausse
4. Utilisez des accolades {} pour substituer l'instruction pour un bloc d'instructions
5. L'instruction while exécute l'instruction suivante tant que la condition est vraie

Logique

- Logique booléenne
 - bool
- Condition est la valeur de l'expression
 - Faux : 0
 - Vrai : !0
- Opérateurs
 - <, <=, ==, !=, >, >=
 - !, &&, ||



Itération

```
int strncmp(char * a, char * b, int n) {
    if(!a) {
        return b ? -1 : 0;
    }

    if(!b) {
        return 1;
    }

    for(int i = 1; i < n; ++i) {
        if(*a != *b || !*a)
            break;
        ++a;
        ++b;
    }

    return *a - *b; // comparaison lexicographique
}
```



Notes :

1. L'instruction `for` définit un contexte imbriqué
2. La syntaxe est définie par 3 expressions optionnelles :
 - a. Initialisation
 - b. Condition
 - c. Incrémentation
3. L'instruction `break` permet de sortir de la boucle

Itération

```
int strncmp(char * a, char * b, int n) {
    if(!a) {
        return b ? -1 : 0;
    }

    if(!b) {
        return 1;
    }

    for(int i = 1; i < n && *a == *b && *a; ++i, ++a, ++b)
        ;

    return *a - *b; // comparaison lexicographique
}
```



Notes :

1. L'opérateur virgule (,) permet de chaîner les opérations
2. Notez le point virgule pour exécuter une instruction nulle à chaque itération

Sélection

```
#include <iostream>

int main() {
    for(;;) {
        char c = std::cin.get();
        switch(c) {
            case '?':
                std::cout << "Usage: Q to quit!" << std::endl;
                break;
            case 'Q':
            case 'q':
                return 0;
            default:
                std::cout << "Use '?' for help" << std::endl;
        }
    }
}
```



Notes :

1. Notez le `for(;;)` pour exécuter une boucle infinie
2. Le `switch` permet de sauter conditionnellement à la valeur d'un entier sur une partie de code
3. Le mot clé `default` est optionnel et permet un comportement de replis
4. Le `break` est utilisé pour terminer l'exécution d'un cas
5. Dans cette situation, le `break` ne sort pas de la boucle `for` mais bien du `switch`

Jeu d'échecs (2)

- But :
 - Faire une variable globale pour l'échiquier
 - 64 caractères
 - Coder les fonctions
 - print
 - movePiece(int x, int y, int i, int j)
 - de (x, y) vers (i, j)
 - une coordonnée valide doit être entre 0 et 7 inclusivement



Test

```
int main() {
    print();
    movePiece(1, 0, 2, 2); // bouge le chevalier blanc
    print();
}
```



Exercice Pratique

Chess-02

44af90c - c35ba67



Structure

- Définit l'organisation d'un espace mémoire
 - Type composite
 - Membres
 - Un décalage par rapport à la structure i.e. *offset*
 - Un type
 - Un identificateur unique



Structure

```
struct Data
{
    int x;
    int y[4];
};

int main() {
    int a = sizeof(Data);
    Data b;
    Data c = { 0 };
    Data d = { 0, { 1, 2, 3, 4 } };
    int e = d.x;
    Data * f = &d;
    int g = f->x;
    int h = &d.y[0] - &d.x;
}
```

Name	Value	Type
a	20	int
b	{x=0x858993460 y=0x0034fc8c}	Data
c	{x=0 y=0x0034fc70 }	Data
x	0	int
y	0x0034fc70	int [4]
y [0]	0	int
y [1]	0	int
y [2]	0	int
y [3]	0	int
d	{x=0 y=0x0034fc54 }	Data
x	0	int
y	0x0034fc54	int [4]
y [0]	1	int
y [1]	2	int
y [2]	3	int
y [3]	4	int
e	0	int
f	0x0034fc50 {x=0 y=0x0034fc51}	Data *
g	0	int
h	1	int



- a. Notez que `sizeof(Data) = 4 + 4 * 4 = 20 bytes`
- b. Allocation sur la pile
- c. Allocation sur la pile et initialisation à 0
- d. Allocation sur la pile et initialisation explicite
- e. Accès à un membre
- f. Pointeur sur la structure allouée sur la pile
- g. Accès à un membre via pointeur
- h. Calcul de l'offset de y à l'aide de l'adresse des membres

Constructeur

```
struct Data
{
    int x;
    int y[4];

    // constructeur par défaut
    Data() : x(0) {
        for(int i = 0; i != 4; ++i)
            y[i] = i;
    }
};

int main() {
    Data a;
    int b = sizeof(a);
    Data c(a);
}
```



- a. Allocation sur la pile
- b. Notez que l'ajout d'une fonction dans la structure ne change pas sa taille
- c. Construction par copie

Constructeur

```
struct Color
{
    char red, green, blue;

    Color(char k = 0) : red(k), green(k), blue(k) {}

    Color(char r, char g, char b) : red(r), green(g), blue(b) {}

};

int main() {
    Color a;
    Color b(255);
    Color c(0x41, 0x69, 0xE1); // royalblue
}
```



- a. Constructeur par défaut
- b. Constructeur avec 1 paramètre
- c. Constructeur avec plusieurs paramètres

Méthode

```
struct Color
{
    /* ... */

    unsigned int getRGB32() {
        return (red << 16) + (green << 8) + blue;
    }

    unsigned int getRGB32(Color * c) {
        return (c->red << 16) + (c->green << 8) + c->blue;
    }

    int main() {
        Color a;
        int b = a.getRGB32();
        int c = getRGB32(&a);
    }
}
```



- a. Allocation sur la pile et construction par défaut
- b. Fonction membre
- c. Fonction globale i.e. OO style C

Méthode

- Notation simplifiée
 - Représentation plus compacte
 - Association explicite entre données et fonctions
 - Possède un paramètre implicite constant
 - this

```
// pseudo code
unsigned int getRGB32(Color * const this) {
    return (this->red << 16) + (this->green << 8) + this->blue;
}
```



Encapsulation

- Principe de la programmation orientée objet
- Abstraction du fonctionnement interne
 - Accès restreint aux données
 - Interface fonctionnelle
- Implémentation par la gestion des accès



Accès

- Public
 - Accessible par tous
- Private
 - Accessible depuis les autres membres de la structure
- Protected
 - Accessible depuis les autres membres de la structure et des structures dérivées



Classe

- Une classe est une structure avec un accès privé par défaut
- Une structure est une classe avec un accès public par défaut



Classe

```
class Color
{
    char red, green, blue;

public:
    Color(char r = 0, char g = 0, char b = 0)
        : red(r), green(g), blue(b) {}

    unsigned int getRGB32() {
        return (red << 16) + (green << 8) + blue;
    }
};
```



Notes :

1. Les membres red, green et blue sont définis comme privés i.e. accès par défaut puisque c'est une classe
2. Les fonctions sont accessibles publiquement
3. Les fonctions sont les 2 seuls endroits où les membres red, green, blue peuvent être utilisés

Classe

```
class Color
{
public:
    Color(char r = 0, char g = 0, char b = 0)
        : red(r), green(g), blue(b) {}

    unsigned int getRGB32() {
        return (red << 16) + (green << 8) + blue;
    }

private:
    char red, green, blue;
};
```



Notes :

1. On place souvent la partie privée en second puisqu'il s'agit de détails d'implémentation
2. Ainsi, l'interface public (utilisable) se trouve en tout premier

Classe

```
class Color
{
public:
    Color(char r = 0, char g = 0, char b = 0);
    unsigned int getRGB32();

private:
    char red, green, blue;
};

Color::Color(char r, char g, char b) : red(r), green(g), blue(b) {

unsigned int Color::getRGB32() {
    return (red << 16) + (green << 8) + blue;
}
```



Notes :

1. On peut séparer l'implémentation des fonctions de la déclaration de la classe
2. Concept de prototypes de fonctions

Jeu d'échecs (3)

- But :
 - Coder la classe Board
 - Restructurer les fonctions en fonctions membres
 - Éliminer la variable globale
 - Constructeur
 - initialiser le tableau



Test

```
int main() {
    // allocation de l'échiquier
    Board a;

    a.print();
    a.movePiece(1, 0, 2, 2); // bouge le chevalier blanc
    a.print();

    // duplique l'échiquier
    Board b(a);
    b.movePiece(1, 7, 2, 5); // bouge le chevalier noir
    b.print();
}
```



Exercice Pratique

Chess-03

c35ba67 - d317fde



Constante

- Indication qu'une variable est immuable
- Déclaration à titre indicatif seulement
 - Indique l'intention logique
 - N'indique pas une possibilité d'optimisation
- Nouveau type
 - Conversion de non const à const est implicite
 - L'inverse doit être fait explicitement
 - Transtypage i.e. *cast*



Constante

```
const int a = 16;
int const b = 32;
const int * c = &a;
int const * d = c;
int const * const e = d;
const int * const f = e;
int const * const * g = &f;
const int * const * h = g;
int const * const * const i = h;
int * const j = &a; // oups!
```



- a. Entier constant
- b. Idem
- c. Pointeur sur un entier constant
- d. Idem
- e. Pointeur constant sur un entier constant
- f. Idem
- g. Pointeur sur un pointeur constant sur un entier constant
- h. Idem
- i. Pointeur constant sur un pointeur constant sur un entier constant
- j. Invalide i.e. demande une conversion explicite

Constante

```
struct A
{
    int x;
};

int main() {
    const A a = { 32 };
    const A * b = &a;
    int const * c = &a.x;
    int const * d = &b->x;
    *d = 0; // oups!
}
```



- a. Déclaration d'une structure constante
- b. Pointeur à une structure constante
- c. Pointeur à un entier constant – le membre d'une structure constante est constant
- d. Idem

Constante

```
struct A
{
    void print() const {
        std::cout << x << std::endl; // this->x is 'const int'
    }

    int square() const {
        return x * x; // sans effets de bords
    }

    int x;
};

int main() {
    A a = { 32 };
    a.print();
}
```



Notes :

1. L'ajout du mot clé const après la liste de paramètres indique la définition d'une fonction membre constante
2. Une fonction membre constante définie le pointeur this comme pointant à une structure constante
3. Il est possible d'appeler une fonction const même si la variable n'est pas const
4. Mais pas l'inverse

Constante

```
struct A
{
    void print() const {
        std::cout << x << " const" << std::endl;
    }

    void print() {
        std::cout << x << " not const" << std::endl;
    }

    int x;
};

int main() {
    A a = { 16 };
    const A * b = &a;
    a.print();
    b->print();
}
```



Notes :

1. Le const est inclus dans la signature de la fonction
2. Permet donc la surcharge

Référence

- Système d'alias
 - Similaire aux liens symboliques
- Évite (partiellement) l'utilisation de pointeurs
 - Pointeur implicite
- Plus sécuritaire
 - Ne permet pas de valeur nulle/indéfinie
 - Ne permet pas de référence à une référence
 - Ne permet pas d'être modifiée



Référence

```
int a = 32;
int & b = a;
const int & c = b;
int const & d = b;
int const & const e = b; // anachronisme
int * f = &a;
int *& g = f;
int * const & h = f;
int i = *f;
int const & j = *h;
```



- a. Entier
- b. Référence à un entier
- c. Référence à un entier constant
- d. Idem
- e. Anachronisme i.e. une référence est toujours constante
- f. Pointeur sur un entier
- g. Référence à un pointeur sur un entier
- h. Référence à un pointeur constant sur un entier
- i. Déréférence du pointeur
- j. Idem

Référence

```
struct A {  
    A(int & _x) : x(_x) { // initialisation requise  
    }  
  
    int & x;  
};  
  
void foo(A a); // oups! collision avec foo(A & a)  
void foo(A * a);  
void foo(A & a);  
void foo(A const * a);  
void foo(A const & a);  
  
int main() {  
    int a = sizeof(A);  
    A b(a);  
    foo(b);  
    foo(&b);  
}
```



Notes :

1. Un membre défini comme une référence doit être initialisé à la construction
 2. Notez qu'on préfixe souvent les paramètres de fonction par un `_` afin d'éviter les confusions
 3. Les identificateurs qui débutent par `_` puis une lettre majuscule sont réservés
 4. Les identificateurs qui débutent par plus d'un `_` sont réservés
 - a. Copie de structure
 - b. Copie de pointeur à une structure
 - c. Référence à une structure
 - d. Copie de pointeur à une structure constante
 - e. Référence à une structure constante
- a. Taille de la structure A i.e. `sizeof(int *)`
 - b. Constructeur
 - c. Appel par copie ou par référence
 - d. Appel par pointeur

Paramètres de fonction

- Par copie
 - invoque le constructeur copie lors de l'appel
 - est efficace pour les types natifs
- Par pointeur
- Par référence
 - syntaxe similaire à la copie i.e. `foo(a)` vs. `foo(&a)`
 - `T const & item`
 - `T & item`
 - performance identique à celle du pointeur



Valeur de retour

```
struct A {
    A(int i) : x(i) {}
    int x;
};

A f1() {
    return A(1); // retour par copie
}

A const & f2() {
    return A(2); // warning: returning address of local variable
}

int main() {
    A a = f1();
    A b = f2();
}
```



Notes sur la séquence d'exécution :

a)

1. Constructeur d'une temporaire locale
2. Copie vers la temporaire de retour
3. Destructeur de la temporaire locale
4. Copie de la temporaire de retour vers a
5. Destruction de la temporaire de retour

b)

1. Constructeur d'une temporaire locale
2. Destructeur de la temporaire locale
3. Retour de la référence sur la temporaire locale déjà détruite
4. Comportement indéterminé lors de la copie vers b

Copie

- Mécanismes de copies
 - Constructeur par copie
 - Assignation i.e. redéfinition de l'opérateur égal (=)
- Génération
 - Copie des membres
 - Assignation et constructeur par copie
 - Copie des types natifs
 - Types de base, pointeurs, références et tableaux



Constructeur copie

```
class String
{
public:
    String() : text(0) {}

    String(String const & other) {
        if(text) {
            free(text); // libère la mémoire si nécessaire
        }

        text = strdup(other.text); // copie
    }

private:
    char * text;
};
```



Assignment

```
String & operator=(String const & other) {
    if(this != &other) {
        if(text) {
            free(text); // libère la mémoire si nécessaire
        }

        text = strdup(other.text); // copie
    }

    return *this;
}
```



Notes :

1. L'opérateur = doit normalement être blindé contre l'auto assignation i.e. *self assignment*
2. Notez le retour de l'objet

Static

- Devant une variable globale
 - Limite la visibilité au fichier courant
- Devant une variable membre
 - Déclare une allocation statique (à implémenter)
- Devant une variable locale
 - Indique une allocation statique
- Devant une fonction membre
 - Limite l'accès aux variables membres statiques



Static

```
static int a;
static int b = 1;

struct A {
    A() : c(d) {
        ++d;
    }

    int c;
    static int d;
    static int e;
};

// définition des variables statiques
int A::d;
int A::e = sizeof(A);
```



- a. Variable globale statique initialisée à 0
- b. Variable globale statique initialisée
- c. Variable membre
- d. Variable membre statique initialisée à 0
- e. Variable membre statique initialisée à la taille de la structure

Static

```
struct A
{
    A() {
        a = b;
        static int c = ++b; // ce code est exécuté une seule fois
        ++c;
    }

    static void foo() {
        ++b; // ne peut accéder 'a' sans pointeur 'this'
    }

    int a;
    static int b;
};

int A::b;
```



Notes :

1. La variable statique c est initialisé au premier appel à A::A()
2. Le compilateur insère donc une autre variable statique interne pour éviter d'exécuter ce code la seconde fois
3. Une fonction statique ne possède pas de pointeur this implicite

Static

```
struct A
{
    static const int N = 32; // constante statique entière initialisée
    static const A nul;

    A(const int k = 0) {
        for(int i = 0; i != N; ++i) {
            a[i] = k;
        }
    }

    int a[N];
};

const A A::nul;
```



Notes :

1. Il est possible d'initialiser les constantes statiques entières
2. Si on déclare une variable membre constante, elle doit être définie dans le contexte global

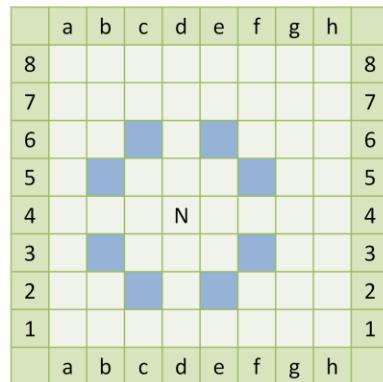
Jeu d'échecs (4)

- But :
 - Utilisez une constante plutôt que 8
 - Restructurez la fonction globale en membre
 - Board::isValid
 - Coder les fonctions
 - Board::getPieceAt(i, j)
 - Board::printPossibleMoves(c)
 - Knight::printPossibleMoves(board, i, j)
 - Knight::printMove(board, x, y, i, j)



Chevalier

- Règles
 - $i \pm 1, j \pm 2$
 - $i \pm 2, j \pm 1$
- Cible
 - Vide
 - Pièce adverse



 **nubo**
ENCADREMENT LOGICIEL

Test

```
int main() {
    // jeu initial
    Board board;
    board.print();

    // ouvertures
    board.printPossibleMoves('n');

    // bouge le chevalier noir
    board.movePiece(1, 7, 2, 5);
    board.print();

    // et encore...
    board.printPossibleMoves('n');
}
```



Exercice Pratique

Chess-04

d317fde - 7920379



```
// détermine les mouvements possibles selon le type de pièce
char c = getPieceAt(i, j);
switch(c) {
    case 'n': case 'N':
        Knight::printPossibleMoves(*this, i, j);
        break;
    case 'r': case 'R':
        Castle::printPossibleMoves(*this, i, j);
        break;
    case 'b': case 'B':
        Bishop::printPossibleMoves(*this, i, j);
        break;
    case 'k': case 'K':
        King::printPossibleMoves(*this, i, j);
        break;
    case 'q': case 'Q':
        Queen::printPossibleMoves(*this, i, j);
        break;
    case 'p': case 'P':
        Pawn::printPossibleMoves(*this, i, j);
}
```

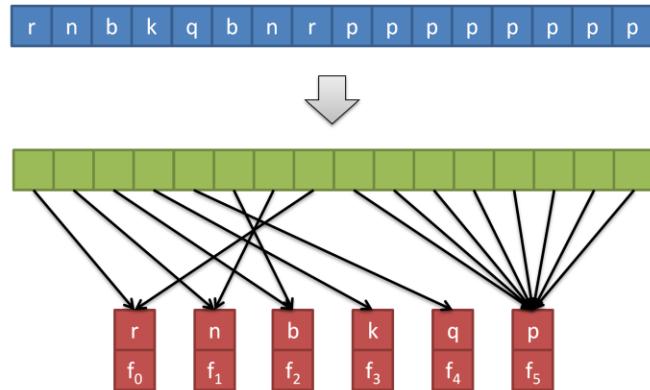


Problématique

- Ici, on effectue une recherche (linéaire)
 - Clé : le caractère
 - Problème d'efficacité
 - Problème d'évolutivité i.e. *scalability*
- Maintenance
 - Ajout d'une nouvelle pièce ?
 - Ajout d'une dépendance de code
 - Problème d'extensibilité



Solution



 **nubo**
ENCADREMENT LOGICIEL

Notes :

1. On a présentement un tableau de caractères qui sera restructuré comme :
2. Un tableau de pointeurs sur des objets contenant
 1. Le caractère à afficher
 2. Un pointeur de fonction spécifique à ce type de pièce
3. On aura donc 1 seul objet par type de pièce

Allocation

- Réserve un espace mémoire
 - Allocateur
 - Standard : malloc/free
 - Autres :
 - Doug Lea : dlmalloc/dlfree
 - Microsoft Windows : HeapAlloc/HeapFree
 - Google : tcmalloc/tcfree
- On doit invoquer le constructeur en C++
 - new/delete



Allocation

```
#include <stdlib.h>

int main() {
    // style C++
    int * a = new int;
    int * b = new int[1024];

    // style C avec stdlib.h
    int * c = (int *) malloc(1024 * sizeof(int));

    // libération
    delete a;
    delete[] b;
    free(c);

    a = b = c = 0;
}
```



- a. Allocation d'un entier
- b. Allocation d'un tableau d'entiers
- c. Idem

Notes :

1. Notez la correspondance entre new/delete et new[]/delete[]
2. Ne pas mixer les allocateurs !
3. Une bonne pratique consiste à remettre le pointeur à 0 (null) après la libération

Allocation

```
#include <stdlib.h>

struct A {
    int x;
    A(int i = 0) : x(i) {}
};

int main() {
    A * a = new A;
    A * b = new A[32];
    A * c = new A[1024];
    A * d = (A *) malloc(sizeof(A)); // oups!
    // ...
    delete a;
    delete b;
    delete[] c;
    free(d);
    a = b = c = 0;
}
```



- a. Allocation d'un objet et initialisation avec le constructeur par défaut
- b. Allocation d'un objet et initialisation avec le constructeur
- c. Allocation d'un tableau d'objets et initialisation avec le constructeur par défaut sur chaque élément
- d. Allocation de la mémoire sans initialisation

Allocation

```
template<typename T>
void safeDelete(T *& pointer) {
    delete pointer;
    pointer = 0;
}

template<typename T>
void safeDeleteItems(T *& pointer) {
    delete[] pointer;
    pointer = 0;
}

int main() {
    /* ... */
    safeDelete(a);
    safeDelete(b);
    safeDeleteItems(c);
    /* ... */
}
```



Dérivation

- Mécanisme de réutilisation de code
 - La classe dérivée hérite de la classe de base
 - Accès à tout ce qui est public ou protected
 - Relation est-un i.e. *is-a*
- Hiérarchie de classes
 - Classe de base
 - Aussi : classe parent i.e. *superclass*
 - Classe dérivée
 - Aussi : classe enfant/fille i.e. *child class, subclass*



Dérivation

```
struct A {  
    int x;  
};  
  
struct B : public A {  
    int y;  
};  
  
int main() {  
    A a;  
    B b;  
    int c = sizeof(a);  
    int d = sizeof(b);  
    int * e = &b.x;  
    int * f = &b.y;  
}
```

Name	Value	Type
a	{x=-858993460 }	A
x	-858993460	int
b	{y=-858993450 }	B
A	{x=-858993460 }	A
x	-858993460	int
y	-858993460	int
c	4	int
d	8	int
e	0x003bfcc0	int *
f	0x003bfc08	int *
	-858993460	int



Notes :

1. On spécifie la classe de base et son accès après le :
2. Il est d'ailleurs possible de restreindre l'accès à cette classe de base en utilisant les mots clés limitant l'accès i.e. private, protected, public

Dérivation

```
struct A {
    int x;
};

struct B {
    A a; // ajout de A au début (offset = 0) de la structure
    int y;
};

int main() {
    A a;
    B b;
    int c = sizeof(a);
    int d = sizeof(b);
    int * e = &b.a.x; // l'accès doit être explicite
    int * f = &b.y;
}
```



Notes :

1. On peut obtenir sensiblement le même résultat sans héritage en spécifiant explicitement A comme membre de B
2. On utilise normalement ce mécanisme en C pour simuler l'héritage

Dérivation

```
struct A {
    int x;
    A(int i) : x(i) {}
};

struct B : public A {
    int y;
    B(int i = 0, int j = 0) : A(i), y(j) {}
};

int main() {
    A a(0);
    B b;
    A * c = &a;
    A * d = &b;
    B * e = &b;
}
```



- a. Constructeur de A (initialisation de x, A::A)
- b. Constructeur de B (initialisation de x, A::A, initialisation de y, B::B)
- c. Pointeur sur le type A
- d. Conversion implicite d'un pointeur sur le type B vers un pointeur sur le type A
- e. Pointeur sur le type B

Dérivation

- Permet une conversion implicite
 - De la classe dérivée vers la classe de base
 - L'inverse doit être explicite
 - Via le transtypage i.e. *cast*
 - Opération à risque
 - Souvent une indication d'un mauvais design



Fonctions virtuelles

```
struct A {
    virtual void foo() {
        std::cout << "A" << std::endl;
    }
};

struct B : public A {
    void foo() {
        std::cout << "B" << std::endl;
    }
};

struct C : public A {
    void foo() {
        std::cout << "C" << std::endl;
    }
};
```



Notes :

1. A::foo est identifiée comme virtuelle
2. Inutile de refaire l'identification dans les classes dérivées i.e. virtual est implicite

Fonctions virtuelles

```
void stub(A * a) {
    a->foo(); // ici, a peut pointer sur un objet de type A, B ou C
}

int main() {
    A a;
    B b;
    C c;
    a.foo();
    b.foo();
    stub(&a);
    stub(&b);
    stub(&c);
    A * p = new A;
    A * q = new B;
    p->foo();
    q->foo();
}
```



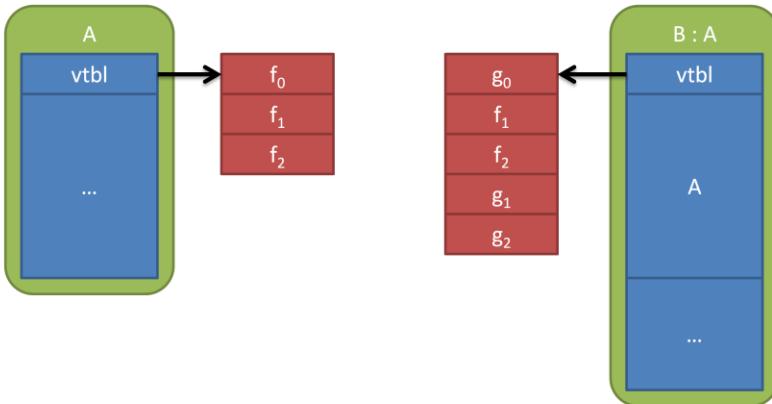
- a. Instance de A
- b. Instance de B
- c. Invocation directe de la fonction A::foo
- d. Invocation directe de la fonction B::foo
- e. Invocation de la fonction virtuelle via le pointeur sur le type A i.e. A::foo
- f. Invocation de la fonction virtuelle via le pointeur sur le type A i.e. B::foo
- g. Invocation de la fonction virtuelle via le pointeur sur le type A i.e. C::foo
- h. Allocation d'une instance de A
- i. Allocation d'une instance de B
- j. Invocation de la fonction virtuelle via le pointeur sur le type A i.e. A::foo
- k. Invocation de la fonction virtuelle via le pointeur sur le type A i.e. B::foo

Polymorphisme

- Fonctions membres (non statiques) seulement
 - Redéfinition i.e. *override*
- Publie une interface commune
 - Classe de base
- Comportement déterminé à l'exécution
 - i.e. *late binding, dynamic binding*



Internes



Notes :

1. L'ajout d'une seule fonction marquée comme virtuelle augmente la taille de la structure pour y stocker le pointeur de table virtuelle
2. Ce pointeur est initialisé dans le constructeur
3. Attention à l'utilisation de fonctions virtuelles lors de la construction

Jeu d'échecs (5)

- But :
 - Coder la classe de base Piece
 - Définir printPossibleMoves comme virtuelle
 - Coder la classe Knight qui dérive de Piece
 - Redéfinir printPossibleMoves
 - Restructurez la classe Board
 - Utilisez un tableau de pointeur de Piece
 - Ne supporter que le Knight
 - Enlever le paramètre à Board::printPossibleMoves



Notes :

1. Board aura besoin de Piece et Piece aura besoin de Board
2. Commencez le fichier avec :

```
class Board;  
class Piece;
```

Exercice Pratique

Chess-05

7920379 - c4f01de



Dépendances

- Déclaration
 - Enregistre l'identificateur dans le système
 - Prototype de fonction
 - Déclaration anticipée de classes/structures
- Définition
 - Crée la ressource
 - Corps de la fonction
 - Corps de la classe/structure
 - Variable statique



Dépendances

```
class A; // déclaration anticipée

void foo(A * a) {
    // ici, la classe A n'est pas définie
}

struct B {
    A * a; // idem
    B();
};

struct A { // définition de A
    int x;
};

B::B() : a(new A) { // instantiation maintenant possible
}
```



Structure physique

- Division du code en fichiers
 - Partage et gestion des sources i.e. *source control*
 - Dépendances physiques
 - Influence le temps de compilation
- Unités de compilation
 - C (*.c) et C++ (*.cpp)
- Convention pour les fichiers d'entête
 - C (*.h) et C++ (*.hpp)



Structure physique

- Structure populaire :
 - 1 fichier hpp (ou h) et cpp par classe
 - Board.hpp
 - Board.cpp
- H ou HPP
 - Contient la définition de la classe
- CPP
 - Contient l'implémentation de la classe
 - Définition des membres



Préprocesseur

- Inclusion de fichiers
 - Dossiers de recherche spécifiés à la compilation
 - Remplace la ligne avec le contenu du fichier

```
#include <stdlib.h>
#include <iostream>
#include "Board.hpp"
```
- Gardes d'inclusion
 - Évitent les définitions multiples



Notes :

1. Un fichier spécifié avec <> est cherché dans les répertoires systèmes
2. Un fichier spécifié avec "" est cherché dans le répertoire courant puis dans les répertoires additionnels définis lors de l'invocation du compilateur e.g. via une option de compilation ou une variable d'environnement

HPP

```
#ifndef BOARD_INCLUDED
#define BOARD_INCLUDED

// déclaration anticipée
class Piece;

class Board {
public:
/* ... */

void set(int i, int j, Piece * item);
void set(int i, int j, Piece * white, Piece * black);

private:
Piece * squares[N * N];
};

#endif
```



Notes :

1. Toujours préférer les déclarations anticipées aux inclusions lorsque c'est possible
2. Pour y arriver, limiter les fonctions dans les fichiers d'entête

HPP

```
#ifndef KNIGHT__INCLUDED
#define KNIGHT__INCLUDED

// dépendances
#include "Piece.hpp"

class Knight : public Piece
{
public:
    Knight(char c);
    void printPossibleMoves(Board const & board, int i, int j);

private:
    void printMove(Board const & board, int x, int y, int i, int j);
};

#endif
```



Notes :

1. On inclus normalement les dépendances
2. Sinon, on force l'utilisateur de notre classe à connaître les dépendances

CPP

```
#include "Board.hpp"
#include "Piece.hpp"
#include "Knight.hpp"

Board::Board() {
    /* ... */

    set(0, 0, new Piece('R'), new Piece('r'));
    set(1, 0, new Knight('N'), new Knight('n'));
    set(2, 0, new Piece('B'), new Piece('b'));
    set(3, 0, new Piece('Q'));
    set(4, 0, new Piece('K'));
    set(3, 7, new Piece('q'));
    set(4, 7, new Piece('k'));

    /* ... */
}
```



Notes :

1. Sans gardes d'inclusion, la classe Piece serait définie 2 fois
2. Notez donc que l'inclusion de Piece.hpp est superflue mais courante en pratique

Jeu d'échecs (6)

- But :
 - Réorganiser la structure physique en fichiers
 - Main.cpp
 - Coder les classes
 - Pawn, Castle, Bishop, Queen, King
 - Coder les fonctions
 - King::printPossibleMoves
 - Pawn::printPossibleMoves



Notes :

1. Sans la promotion
2. Sans la prise en passant
3. Sans le roque
4. Sans considérer les situations d'échecs

Pion

- Direction
 - $dy = +/- 1$
- Avancée
 - $j + dy$
 - $j + 2 * dy$ (1^{er} coup)
- Attaque
 - $i +/- 1, j + dy$

	a	b	c	d	e	f	g	h	
8									8
7									7
6									6
5									5
4							P		4
3									3
2				P					2
1									1
	a	b	c	d	e	f	g	h	



Roi

- Règles
 - $i \pm 1, j \pm 1$
- Cible
 - Vide
 - Pièce adverse

	A	b	c	d	e	f	g	h	
8									8
7									7
6									6
5									5
4				K					4
3									3
2									2
1									1
	a	b	c	d	e	f	g	h	



Exercice Pratique

Chess-06

c4f01de - 70bb818



```

void Pawn::printPossibleMoves(Board const & board, int i, int j) {
    Piece * self = board.getPieceAt(i, j);

    // on détermine la direction du mouvement selon la couleur
    int d = self->isWhite() ? 1 : -1;
    int y = j + d;

    // on teste les diagonales
    printMove(board, i, j, i - 1, y);
    printMove(board, i, j, i + 1, y);

    // et devant
    if(!board.getPieceAt(i, y)) {
        printMove(board, i, j, i, y);

        // et encore devant si la pièce se trouve sur la position initiale
        int w = self->isWhite() ? 1 : 6;
        if(j == w) {
            y += d;
            if(!board.getPieceAt(i, y)) {
                printMove(board, i, j, i, y);
            }
        }
    }
}

```



Dérivation multiple

- Permet plusieurs classes de base
- Fonctionnalité complexe
 - Instances multiples d'une classe de base
 - Problème du diamant i.e. *deadly diamond of death*
 - Code d'ajustement i.e. *adjustor thunk*
 - Conversion
 - Fonctions virtuelles
- À utiliser avec précautions...



Dérivation multiple

```
struct A {
    int x;
};

struct B {
    int y;
};

struct C : public A, public B {
    int z;
};

int main() {
    C a;
    int * b = &a.x;
    int * c = &a.y;
    int * d = &a.z;
}
```



Dérivation multiple

```
struct A {  
    int x;  
};  
  
struct B : public A {  
    int y;  
};  
  
struct C : public A {  
    int z;  
};  
  
struct D : public B, public C {  
    int w;  
};
```



Notes :

1. Exemple de dérivation multiple avec un doublon hiérarchique
2. La taille de D est donc de 5*sizeof(int)
3. L'accès à la variable membre x depuis la classe D est ambiguë

Dérivation multiple

```
struct A {
    virtual void foo() { std::cout << "A" << std::endl; }
};

struct B : public A {
    virtual void foo() { std::cout << "B" << std::endl; }
};

struct C : public A {
    virtual void foo() { std::cout << "C" << std::endl; }
};

struct D : public B, public C {
    virtual void foo() { std::cout << "D" << std::endl; }
};
```



Notes :

1. Ici, `sizeof(D) == 8`
2. `sizeof(A) == 4`
3. `sizeof(B) == sizeof(C) == 4`
4. Donc, il y a maintenant plus d'un pointeur de vtable

Dérivation multiple

```
struct A {
    virtual void foo() { std::cout << "A" << std::endl; }
};

struct B : public virtual A {
    virtual void foo() { std::cout << "B" << std::endl; }
};

struct C : public virtual A {
    virtual void foo() { std::cout << "C" << std::endl; }
};

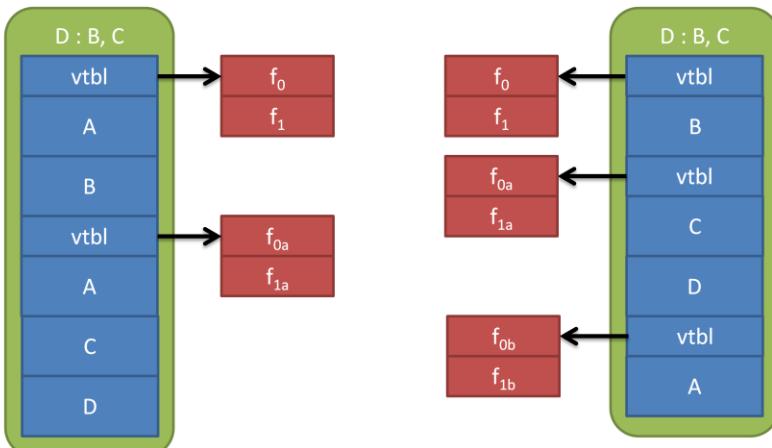
struct D : public B, public C {
    virtual void foo() { std::cout << "D" << std::endl; }
};
```



Notes :

1. On peut éviter la duplication de la structure A en spécifiant la dérivation virtuelle
2. Ici, la taille grimpe i.e. `sizeof(D) == 12`
3. `sizeof(A) == 4`
4. `sizeof(B) == sizeof(C) == 8`

Internes



nubo
ENCADREMENT LOGICIEL

Notes :

1. Dérivation multiple
2. Dérivation multiple virtuelle (structure en diamant : complexité accrue)

Fonction virtuelles pures

```
struct A {
    virtual void foo() = 0;
};

struct B {
    virtual void boo() = 0;
};

struct C : public A, public B {
    void foo() { std::cout << "C" << std::endl; }
    void boo() { std::cout << "C" << std::endl; }
};

int main() {
    C * a = new C;
}
```



Notes :

1. Une fonction virtuelle pure (= 0) déclare une entrée dans la vtable sans la définir
2. Toutes les entrées dans la vtable doivent être définies pour permettre l'instanciation
3. Ici, impossible d'instancier A ou B

Interface

- Une classe d'interface possède :
 - Aucune variable membre
 - Uniquement des fonctions virtuelles pures
- Héritage multiple ?
 - Cas beaucoup plus courant
 - Moins complexe
 - Sans la problématique du diamant
 - Similaire à JAVA, C#, Objective-C, ...



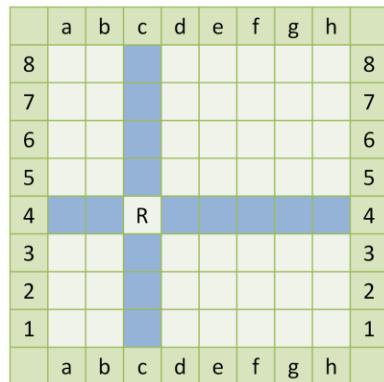
Jeu d'échecs (7)

- But :
 - Coder la classe Mover
 - Coder la fonction
 - Mover::slide
 - Modifier les classes Castle, Bishop et Queen



Tour

- Règles
 - i +/- N
 - j +/- N
- Cible
 - Tracé vide
 - Pièce adverse



 **nubo**
ENCADREMENT LOGICIEL

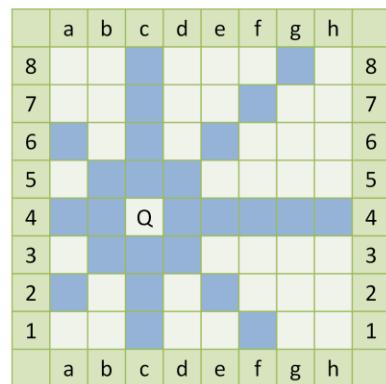
Fou

- Règles
 - $i +/- N, j +/- N$
- Cible
 - Tracé vide
 - Pièce adverse

	a	b	c	d	e	f	g	h	
8									8
7									7
6									6
5									5
4			B						4
3									3
2									2
1									1
	a	b	c	d	e	f	g	h	

Reine

- Règles
 - Tour + Fou



 **nubo**
ENCADREMENT LOGICIEL

Exercice Pratique

Chess-07

70bb818 - ad1a106

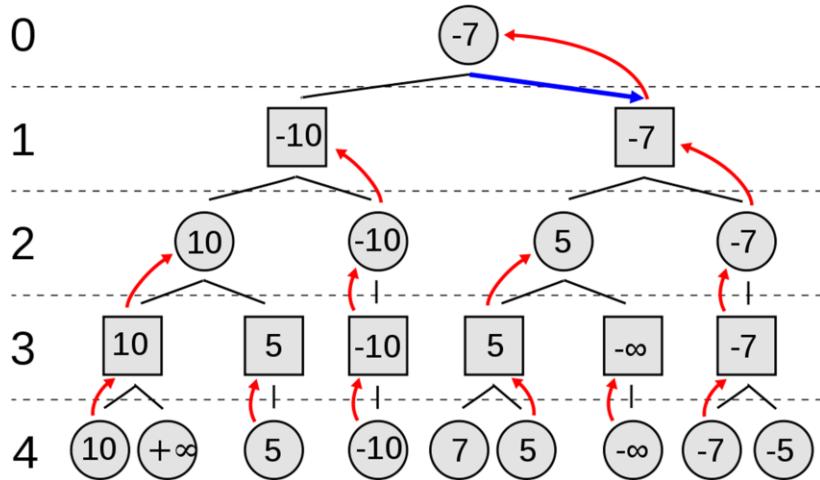


Minimax

- Algorithme décisionnel simple
 - Teste toutes les possibilités
 - Assigne un score à chaque possibilité
 - Choisi le score maximal
- Version récursive
 - Teste les possibilités de l'adversaire
 - Assume que l'adversaire va maximiser son score



Minimax



 **nubo**
ENCADREMENT LOGICIEL

Jeu d'échecs (8)

- But :
 - Coder la classe Explorer
 - Explorer::add(board, x, y, i, j)
 - Restructurer
 - Explorer au lieu d'imprimer
 - Board::explorePossibleMoves(opponent, explorer)
 - Piece::explorePossibleMoves(explorer, board, i, j)
 - Calculer le score d'une possibilité
 - Piece::getCost avec p=1, n=b=3, r=5, q=9, k=100



Test

```
#include "Board.hpp"
#include "Explorer.hpp"

int main() {
    // jeu initial
    Board board;
    board.print();

    // explore l'arbre de possibilités
    Explorer root(2);
    board.exploreMoveTree(board.getPieceAt(4, 7), root);
}
```



Exercice Pratique

Chess-08

ad1a106 - 4d1df4b



STL

- Standard Template Library
 - Partie intégrante de la librairie standard C++
- Contient
 - Algorithmes et foncteurs
 - e.g. std::sort, std::for_each, std::find, std::copy...
 - Conteneurs & itérateurs
 - e.g. std::string, std::vector, std::map, std::list...



std::vector<T>

- Tableau de taille dynamique
 - Éléments contigus (T)
 - Accès direct i.e. *random access*
 - Gestion de la mémoire lors d'ajouts/retraits
 - À la toute fin
 - Complexité constante amortie i.e. *amortized constant time*
 - Ailleurs
 - Linéaire
 - Spécialisation pour le type bool



std::vector<T>

- [i], at(n), back(), front()
- size()
- empty()
- insert(...), erase(...), push_back(), pop_back()
- begin(), end()
- clear()
- reserve(n)
- swap()



std::vector<T>

```
#include <vector>

int main() {
    std::vector<int> items;

    for(int i = 0; i != 4; ++i) {
        items.push_back(i);
    }

    items.pop_back();

    for(int i = 0; i != items.size(); ++i) {
        std::cout << &items[i] << ' ' << items[i] << std::endl;
    }

    // 00854C90 0
    // 00854C94 1
    // 00854C98 2
}
```



std::vector<T>

```
#include <vector>

typedef std::vector<int> Type;

void print(Type const & items) { // paramètre par référence
    for(int i = 0; i != items.size(); ++i) {
        std::cout << items[i] << std::endl;
    }
}

int main() {
    Type values;
    for(int i = 0; i != 4; ++i) {
        values.push_back(i);
    }

    print(values);
}
```



Notes :

1. On définit souvent un type de façon à réduire la lourdeur de la syntaxe
2. Attention au paramètre : une copie est si vite arrivée

std::vector<T>

```
#include <vector>
#include <string>

template<typename T>
void print(std::vector<T> const & items) {
    for(int i = 0; i != items.size(); ++i) {
        std::cout << items[i] << std::endl;
    }
}

int main() {
    std::vector<std::string> items;
    items.push_back("toto");
    items.push_back("titi");
    items.push_back("tata");
    print(items);
}
```



std::vector<T>

```
#include <algorithm>

int main() {
    std::vector<std::string> items;
    items.push_back("toto");
    items.push_back("titi");
    items.push_back("tata");
    items.push_back("tutu");

    // copy
    std::vector<std::string> sorted(items);

    // sort
    std::sort(sorted.begin(), sorted.end());
    print(sorted);
}
```



std::vector<T>

```
class Color { /* ... */ };

bool operator<(Color const & a, Color const & b) {
    return a.getRGB32() < b.getRGB32();
}

int main() {
    std::vector<Color> items;
    items.push_back(Color(128, 0, 0));
    items.push_back(Color(0, 128, 0));
    items.push_back(Color(0, 0, 128));
    items.push_back(Color(256, 0, 0));
    items.push_back(Color(0, 256, 0));
    items.push_back(Color(0, 0, 256));

    // sort
    std::sort(items.begin(), items.end());
}
```



Jeu d'échecs (9)

- But :
 - Coder la classe Move
 - Pour std::vector<Move>
 - Coder la classe Tree
 - Dérive de la classe Explorer (add est virtuelle)
 - Conserver toutes les possibilités ayant le même score
 - Tree::play(board) en choisi une au hasard



Test

```
#include "Board.hpp"
#include "Tree.hpp"

int main() {
    Board board;

    Tree white(2);
    board.exploreMoveTree(board.getPieceAt(4, 7), white);
    white.play(board);
    board.print();

    Tree black(2);
    board.exploreMoveTree(board.getPieceAt(4, 0), black);
    black.play(board);
    board.print();
}
```



Exercice Pratique

Chess-09

4d1df4b - 8acc1c9



Destructeur

- Permet de libérer les ressources allouées
- Appel à la fin de vie
 - Sortie du contexte de définition
 - Libération de la mémoire
 - Destruction de l'objet qui le contient
- Hiérarchie de classe
 - Appel faits dans l'ordre inverse de la construction
 - Classe enfant avant ses parents



Destructeur virtuel

```
struct A {
    ~A() {
        std::cout << "a deleted" << std::endl;
    }
};

struct B : public A {
    ~B() {
        std::cout << "b deleted" << std::endl;
    }
}

std::vector<int> items;
};

int main() {
    A * a = new A;
    A * b = new B;
    delete a;
    delete b; // invoque uniquement le destructeur de A
}
```



Destructeur virtuel

```
struct A {
    virtual ~A() {
        std::cout << "a deleted" << std::endl;
    }
};

struct B : public A {
    ~B() {
        std::cout << "b deleted" << std::endl;
    }

    std::vector<int> items;
};

// a deleted
// b deleted
```



Notes :

1. On ajoute le mot clé `virtual` pour ajouter le destructeur dans la vtable
2. Il faut faire particulièrement attention au destructeur virtuel si les objets de classes dérivées sont détruits sans conserver le type : le destructeur de la classe dérivée ne sera pas appelé et le destructeur de ses membres non plus

Destructeur

- Gestion des ressources
 - *Resource Acquisition Is Initialization - RAII*
- Variable sur la pile
 - Réserve la ressource au constructeur
 - Libère la ressource au destructeur
 - Automatique
 - Garantie d'appel lors d'exception i.e. *exception-safe*



RAII

```
#include <windows.h>

class Mutex
{
public:
    Mutex() {
        handle = CreateMutex(NULL, FALSE, NULL);
    }

    ~Mutex() {
        if(handle) {
            CloseHandle(handle);
            handle = NULL;
        }
    }

private:
    HANDLE handle;
};
```



RAII

- Gestion de la mémoire i.e. *smart pointers*
 - std::auto_ptr *
 - Libère l'objet alloué à la destruction i.e. *delete*
 - Prend possession du pointeur
 - std::shared_ptr
 - Permet le partage par comptage i.e. *reference counted*
 - Libère l'objet alloué lorsqu'il n'y a plus de référence
 - Attention aux références circulaires
 - ...



1. Avec C++11, la classe auto_ptr est obsolète et remplacée par unique_ptr
2. La classe shared_ptr est apparue dans Boost et est maintenant standard avec C++11

RAII

```
struct A {
    A(char const * string) : text(string) {
    }

    ~A() {
        std::cout << text << std::endl;
    }

    std::string text;
};

int main() {
    typedef std::vector<std::unique_ptr<A>> Type;
    Type items;
    items.push_back(Type::value_type(new A("a")));
    items.push_back(Type::value_type(new A("b")));
}
```



Notes :

1. Notez la définition de type pour simplifier la syntaxe
2. La classe std::vector<T> définie le type value_type comme étant le type T

Jeu d'échecs (10)

- But :
 - Coder la classe Game
 - Dérive de Board
 - Gère l'allocation des objets de type Piece
 - Implémente Game::play
 - Boucle de jeu
 - Alterne les joueurs
 - Teste la fin ?



Test

```
#include "Game.hpp"

int main() {
    // création du jeu d'échecs
    Game game;
    game.print();
    game.play();
}
```



Exercice Pratique

Chess-10

8acc1c9 - 2d19d23



Avant de quitter

- Remplir la feuille d'évaluation
- Remise des attestations
- Vous avez des questions après la formation ?

Courriel :

eric.robert@nubo.ca

1.855.I.GO.NUBO#1784



Avant de quitter

- Remplir la feuille d'évaluation
- Remise des attestations
- Vous avez des questions après la formation ?

Courriel :

eric.robert@nubo.ca

1.855.I.GO.NUBO#1784



Références

- Bjarne Stroustrup
 - [The C++ Programming Language](#)
 - [Programming: Principles and Practice Using C++](#)
- Scott Meyers
 - [Effective C++](#)
 - [Effective STL](#)
- Herb Sutter & Andrei Alexandrescu
 - [C++ Coding Standards](#)
- Andrew Koenig & Barbara Moo
 - [Accelerated C++](#)



Sites

- <http://www.parashift.com/c++-faq-lite/>
- <http://cplusplus.com/>
- <http://cppandbeyond.com/>

