EXERCICES DE C++

Introduction

Ce document est une compilation des exercices que j'ai conçus depuis que je donne des cours de C++. Soit des interros écrites, soit des exemples de cours, soit des examens ou des exercices de TD.

Commentaires

exercice 1.1

Le programme suivant est-il valide en C++ ? en C ?

```
int truc = 0;  //  la variable truc est initialisée  /* a la ligne suivante
                   elle va etre incrementee */
truc++;
```

exercice 1.2

Le programme suivant est-il valide en C++ ? en C ?

```
int truc = 0;  /*  la variable truc est initialisée  // a la ligne suivante
                   elle va etre incrementee */
truc++;
```

Passage de paramètres par référence

exercice 2.1

Soit le programme principal suivant:

```
main() {
  int a=1; int b=-2; int c=0;
  cout << "avant appel de ajouter" << endl;
  cout << "a = " << a << endl;
  cout << "b = " << b << endl;
  cout << "c = " << c << endl;
  ajouter(a,b,c);
  cout << "apres appel de ajouter" << endl;
  cout << "a = " << a << endl;
  cout << "b = " << b << endl;
  cout << "c = " << c << endl;
}
```

Donner la sortie du programme dans les 3 cas suivants:

```
void ajouter (int a, int b, int c) { c = a+b; }
void ajouter (int a, int b, int * c) { *c = a+b; }
void ajouter (int a, int b, int & c) { c = a+b; }
```

Comment s'appelle chacun de ces 3 passages de paramètres ?

Exploitation de fonctions C dans des fichiers .h et .c

exercice 3.1

Un programme comprend les 5 fonctions suivantes:

```
void creer_tables() { ... }
void creer_chaises() { ... }
void detruire_tables() { ... }
void detruire_chaises() { ... }
main () {
        creer_tables();
        creer_chaises();
        detruire_tables();
        detruire_chaises();
}
```

Afin de programmer de façon modulaire, ranger les déclarations et définitions de ces fonctions dans les fichiers suivants: principal.cpp, table.h, table.cpp, chaise.h, chaise.cpp
Pour chacun de ces fichiers,
on indiquera les directives #include #ifndef et #define nécessaires,
on indiquera les déclarations et/ou définitions des fonctions du fichier,
on ne précisera pas le corps (ou contenu) des fonctions.

exercice 3.2

Un programme comprend les 5 fonctions suivantes:

```
void assembler_bois() { ... }
void desassembler_bois() { ... }
void creer_tables() { assembler_bois(); }
void detruire_tables() { desassembler_bois(); }
main () {
        creer_tables();
        detruire_tables();
}
```

Afin de programmer de façon modulaire, ranger les déclarations et définitions de ces fonctions dans les fichiers suivants: principal.cpp, table.h, table.cpp, bois.h, bois.cpp
Pour chacun de ces fichiers,
on indiquera les directives #include #ifndef et #define nécessaires,
on indiquera les déclarations et/ou définitions des fonctions du fichier,
on ne précisera pas le corps (ou contenu) des fonctions.

Pointeurs et références

exercice 4.1

Soit le programme C++ suivant:

```
#include <iostream.h>
main() {
        int x[3] = { 3, 6, 11 }; int y = 4; int &z = y;
        cout << "1 " << x[0] << x[1] << y << z << endl;

        int * p = &x; int * p = x; *p = 5;
        cout << "2 " << x[0] << x[1] << y << z << endl;

        int * q = &y; int * q = y; *q = 7;
        cout << "3 " << x[0] << x[1] << y << z << endl;

        q = new int; if (q!=NULL) *q = 2; delete q;
        cout << "4 " << x[0] << x[1] << y << z << endl;

        *(++p) = 8;
        cout << "5 " << x[0] << x[1] << y << z << endl;

        z = *(x+2)-2;
        cout << "6 " << x[0] << x[1] << y << z << endl;
}
```

a) Quel est le nom du rapport entrenu par les variables z et y ? par les variables p et x ?
b) Trouver et supprimer les 2 instructions incorrectes.
c) Donner la sortie du programme.

Constructeurs et destructeurs

Exercice CSTR-DSTR

a) Donner la sortie du programme C++ suivant et commenter très brièvement.

```
#include <iostream.h>
class Truc { public :
        Truc() { cout << « ++ normal » << endl ; }
        ~Truc() { cout << « -- normal » << endl ; }
} ;
void main() {
        Truc x;
}
```

a') Même question avec.

```
void main() {
        Truc * x = new Truc();
}
```

a'') Même question avec.

```
void main() {
        Truc * x = new Truc();
        delete x ;
}
```

Exercice RECOPIE-CLONE

a) Donner la sortie du programme C++ suivant et commenter en précisant notamment le nombre d'objets créés et détruits et la nature (plantage ou pas) de la terminaison de l'exécution.

```
#include <iostream.h>

class Machin { public :
      int bidule ;
      Machin(int b) {
            cout << « ++ Machin normal » << endl ;
            bidule = b ;
      }
      ~Machin() { cout << « -- Machin normal » << endl ; }
} ;

class Truc { public :
      Machin * machin ;
      Truc(Machin * m) {
            cout << « ++ Truc normal » << endl ;
            machin = m ;
      }
      ~Truc() {
            cout << « -- Truc normal » << endl ;
            if (machin != NULL) delete machin ;
            machin = NULL ;
      }
} ;

void main() {
      Machin m(1) ;
      Truc x(&m);
      Truc y(x);
}
```

a') On rajoute le constructeur suivant dans la classe Truc.

```
      Truc(Truc & t) {
            cout << « ++ Truc recopie » << endl ;
            machin = new Machin(t.machin->bidule) ;
      }
```

Donner la sortie du programme C++ et commenter en précisant le nombre d'objets créés et détruits et la nature (plantage ou pas) de la terminaison de l'exécution.

a'') On rajoute le constructeur suivant dans la classe Machin.

```
      Machin(Machin & m) {
            cout << « ++ Machin recopie » << endl ;
            bidule = m.bidule ;
      }
```

Ré-écrire le constructeur de la classe `Truc` en utilisant le constructeur de la classe `Machin`. Donner la sortie du programme et commenter en précisant le nombre d'objets créés et détruits et la nature (plantage ou pas) de la terminaison de l'exécution.

a^{ter}) Comment modifier le main() pour améliorer le programme ? Commenter brièvement.

exercice 5.1

Soit le programme C++ suivant:

```
#include <iostream.h>
class Cle  { public :
        int a;
        int b;
        Cle (int x, int y) { a = x; b = y; cout << "++ cle " << this << " " << a << " " << b << endl; }
        ~Cle() { cout << "-- cle " << this << " " << a << " " << b << endl; }
};
class Coffre { public :
        Cle * cle;
        Coffre(Cle * c) { cle = c;
                cout << "++ coffre CLE " << this << " " << cle->a << " " << cle->b << endl; }
        Coffre(int n) { cle = new Cle(100+n, 1000+n);
                cout << "++ coffre ENTIER " << this << " " << cle->a << " " << cle->b << endl; }
        Coffre(Coffre & c) { cle = new Cle(10+ c.cle->a, 10+ c.cle->b);
                cout << "++ coffre COFFRE " << this << " " << cle->a << " " << cle->b << endl; }
        ~Coffre() { delete cle; cout << "-- coffre " << this << endl; }
};
main () {
        Cle * cle1 = new Cle(1, 2);
        Coffre c1 = cle1;
        Coffre c2 = 3;
        Coffre c3 = c1;
}
```

Donner la sortie du programme. On indiquera si la fin de l'éxécution est normale ou pas.

exercice 5.2

Soit le programme C++ suivant:

```
#include <iostream.h>
class Cle  { public :
        int a;
        int b;
        Cle (int x, int y) { a = x; b = y; cout << "++ cle " << this << " " << a << " " << b << endl; }
        ~Cle() { cout << "-- cle " << this << " " << a << " " << b << endl; }
};
class Coffre { public :
        Cle * cle;
        Coffre(Cle * c) { cle = new Cle(10+c->a, 10+c->b);
                cout << "++ coffre CLE " << this << " " << cle->a << " " << cle->b << endl; }
        Coffre(int n) { cle = new Cle(100+n, 1000+n);
                cout << "++ coffre ENTIER " << this << " " << cle->a << " " << cle->b << endl; }
        Coffre(Coffre & c) { cle = c.cle;
                cout << "++ coffre COFFRE " << this << " " << cle->a << " " << cle->b << endl; }
        ~Coffre() { delete cle; cout << "-- coffre " << this << endl; }
};
main () {
        Cle * cle1 = new Cle(4, 5);
        Coffre c1 = cle1;
        Coffre c2 = 6;
        Coffre c3 = c1;
}
```

Donner la sortie du programme. On indiquera si la fin de l'exécution est normale ou pas.

Exercice STATIC

a) Le programme C++ suivant n'est pas compilable par g++. Pourquoi ? Rajouter les deux lignes nécessaires pour qu'il le soit.

```
#include <iostream.h>
class R { public:
  static R * notreR;
  static int j;
  int i;
  R * monR;

  R(int a) { i = a; monR = NULL; j++; }

  static void p(R * r) {
    j = 0;
    notreR = new R(0);
    m(notreR);
  }
  static void q(R * r) {
    cout << "R::notreR = "; n(notreR);
    cout << "R::j = " << j << endl;
  }
  static void m(R * r) {
      cout << "R, i = " << r->i << endl;
  }
  static void n(R * r) {
    m(r);
    cout << "monR = ";
    if (r->monR!=NULL) r->monR->m(r->monR);
    else cout << "NULL" << endl;
  }
};

class S { public:
  static void main() {
    R::p(NULL);
    R * r1 = new R(1);
    R::notreR->monR = r1;
    R * r2 = new R(2);
    cout << "r2 = "; R::n(r2);
    r1->monR = r2;
    cout << "r1 = "; R::n(r1);
    R::q(r1);
  }
};

void main() {
  S::main();
}
```

b) Maintenant que g++ compile ce programme, quelle est sa sortie ?

c) Le mot-clé 'static' est correctement placé en ce qui concerne les attributs mais pas pour les méthodes. Reprogrammer ce programme pour que sa sortie soit identique, que les mot-clé 'static' soient partout correctement placés, et enfin que les paramètres inutiles soient enlevés.

d) Donner le pour et le contre de la phrase : « le constructeur est une méthode d'instance ».

Exercice MOINS-MOINS

a) g++ ne compile pas le programme C++ suivant :

```
class Poule  { public :
      Oeuf * monOeuf ;
      void maMethode() { ;  }
} ;
class Oeuf  { public :
      Poule * maPoule ;
      void maMethode() { ; }
} ;
void main() {
      Poule * p = new Poule(); Oeuf * o = new Oeuf();
      p->monOeuf = o; o->maPoule = p;
      p->maMethode(); o->maMethode();
}
```

Pourquoi ? Rajouter l'instruction convenable pour qu'il soit compilable par g++.

b) Le programme étant modifié convenablement, on remplace le corps de la méthode `void maMethode()` de la classe Poule par `monOeuf->maMethode() ;` A nouveau, g++ ne compile pas le programme.
Pourquoi ? Modifier le programme pour qu'il soit compilable par g++.

c) Le programme étant modifié convenablement, on rajoute le test `if(monOeuf !=NULL)` devant l'appel de `monOeuf->maMethode() ;` A nouveau, g++ ne compile pas le programme.
Pourquoi ? Rajouter une ligne convenable pour qu'il soit compilable par g++.

d) g++ ne compile pas le programme C++ suivant :

```
class Poule  { public :
      Oeuf monOeuf ;
      void maMethode() { ;  }
} ;
class Oeuf  { public :
      Poule maPoule ;
      void maMethode() { ; }
} ;
void main() {// idem
}
```

Pourquoi ? Y a-t-il une solution ?

Héritage

Exercice 6.1

On a les déclarations et définitions suivantes:

```
#include <iostream.h>
class Bubu { public:
        int x;
        Bubu(int);
        ~Bubu();
};
class Toto { public:
        Bubu * pbu;
        int y;
        Toto(int, int);
        ~Toto();
};
class Titi: public Toto { public:
        int z;
        Titi(int, int, int);
        ~Titi();
};
Titi::Titi(int a, int b, int c) : Toto(a, a+b) {
        cout << "++ Titi debut" << endl;
        z = c;
        cout << "++ Titi z = " << z << endl;
        cout << "++ Titi fin" << endl;
}
Titi::~Titi() : {
        cout << "-- Titi debut z = " << z << endl;
        cout << "-- Titi fin" << endl;
}
Toto::Toto(int a, int b) {
        cout << "++ Toto debut" << endl;
        y = b;
        cout << "++ Toto y = " << y << endl;
        pbu = new Bubu(a);
        cout << "++ Toto fin" << endl;
}
Toto::~Toto() {
        cout << "-- Toto debut y = " << y << endl;
        delete pbu;
        cout << "-- Toto fin" << endl;
}
Bubu::Bubu(int a) {
        cout << "++ Bubu debut" << endl;
        x = a;
        cout << "++ Bubu x = " << x << endl;
        cout << "++ Bubu fin" << endl;
}
Bubu::~Bubu(int a) {
        cout << "-- Bubu debut x = " << x << endl;
        cout << "-- Bubu fin" << endl;
}
main() {
        Toto to(1, 2);
        Titi ti(3, 4, 5);
}
```

Question a

Donner la sortie du programme.

Question b

On ajoute la fonction membre imprimer() dans la classe Toto et on modifie le main() :

```
class Toto { public: ... void imprimer(); ... };
Toto::imprimer() {
        cout << "Toto::imprimer: " << pbu->x << " " << y << endl;
}
main () {
        Toto to(6, 7);
        Titi ti(8, 9, 0);
        to.imprimer();          // instruction A
        ti.imprimer();          // instruction B
}
```

Donner la sortie des instructions A et B.

Question c

On ajoute la fonction membre imprimer() dans la classe Titi :

```
class Titi: public Toto { public:  ... void imprimer(); ... };
Titi::imprimer() {
        cout << "Titi::imprimer: " << pbu->x << " " << y << " " << z << endl;
}
```

Donner la sortie des instructions A et B de la question précédente.

Question d

On modifie le programme principal :

```
main () {
        Toto to(44, 55);
        Titi ti(11, 22, 33);
        Toto * pto;
        Titi * pti;
        pto = &to;
        pti = &ti;
        pto->imprimer(); // instruction A
        pti->imprimer();  // instruction B
        pto = pti;
        pto->imprimer(); // instruction C
}
```

Donner la sortie des instructions A, B et C.

Question e

On ajoute le mot-clé virtual devant la fonction membre imprimer de la classe Toto :

```
class Toto { public:  ... virtual void imprimer(); ... };
```

Donner la sortie des instructions A, B, C de la question précédente.

Exercice 7 (abcde)

On a les déclarations et définitions suivantes:

```
#include <iostream.h>
class A { public :
        A * monA ;
        virtual void maMethode();
        virtual void saMethode();
} ;
class C;
class B : public A { public :
        C * monC ;
        void saMethode();
} ;
class C : public A { public :
        B * monB ;
        void maMethode();
        void saMethode();
} ;
class E;
class D : public B { public :
        E * monE;
        void maMethode();
        void saMethode();
} ;
class E : public B { public :
        D * monD ;
        void maMethode();
} ;

void A::maMethode() { cout << " A ";  }
void A::saMethode() { monA->maMethode();  }
void B::saMethode() { monC->maMethode() ;  }
void C::maMethode() { cout << " C ";  }
void C::saMethode() { monB->maMethode() ;  }
void D::maMethode() { cout << " D "; }
void D::saMethode() { monE->maMethode() ;  }
void E::maMethode() { cout << " E " ;  }

void main(int argc, char * argv[]) {
        A * a = new A(); B * b = new B(); C * c = new C(); D * d = new D(); E * e = new E();
        a->monA = b; b->monA = c ; c->monA = d; d->monA = e; e->monA = a;
        b->monC = c; c->monB = b; d->monC = c; e->monC = c; d->monE = e; e->monD = d;
        a->maMethode();  b->maMethode();  c->maMethode();  d->maMethode();  e->maMethode();
        a->saMethode();  b->saMethode();  c->saMethode();  d->saMethode();  e->saMethode();
}
```

Question « UML »

Dessiner le diagramme de généralisation, un diagramme classe-association et de diagramme d'instances correspondant.

Question « sortie »

Donner la sortie du programme.

Exercice 8 (ballon)

Donner la sortie du programme pour chacun des cas a) b) c) d) e) f).

Dans les 6 cas, a) b) c) d) e) f), le programme principal est le même :

```
#include <iostream.h>
// les classes sont déclarees ici...
main() {
      Objet o;
      Porcelaine p;
      Ballon b;
      BallonRugby bR;
      BallonFoot bF;
      o.rebondir();
      p.rebondir();
      b.rebondir();
      bR.rebondir();
      bF.rebondir();
      Objet * po = &o; po->rebondir();
      po = &p; po->rebondir();
      po = &b; po->rebondir();
      po = &bR; po->rebondir();
      po = &bF; po->rebondir();
}
```

a) Les classes sont les suivantes :

```
class Objet { public: ... void rebondir() { cout << "?" << endl; } };
class Porcelaine: public Objet{ public: ... }; // rebondir() pas definie
class Ballon: public Objet { public: ... };    // rebondir() pas definie
class BallonRugby: public Ballon{ public: ...}; //rebondir() pas definie
class BallonFoot: public Ballon{public: ... }; // rebondir() pas definie
```

b) Les classes Objet, BallonRugby, BallonFoot sont identiques à **a)**.

```
class Porcelaine: public Objet { public: ...
      void rebondir() { cout << ":-(" << endl; } };
class Ballon: public Objet { public: ...
      void rebondir() { cout << ":-)" << endl; } };
```

c) Les classes Objet, Porcelaine et Ballon sont identiques à **b)**.

```
class BallonRugby : public Ballon { public: ...
      void rebondir() { cout << "()" << endl; }  };
class BallonFoot : public Ballon { public: ...
      void rebondir() { cout << "O" << endl; }  };
```

d) Les classes Objet, Porcelaine et Ballon sont identiques à **c).**

```
class BallonRugby: public Ballon {public:...
      virtual void rebondir() { ... } };
class BallonFoot: public Ballon {public:...
      virtual void rebondir() { ... } };
```

e) Les classes Objet, BallonRugby, BallonFoot sont identiques à **c).**

```
class Porcelaine : public Objet { public: ...
      virtual void rebondir() { ... } };
class Ballon : public Objet { public: ...
      virtual void rebondir() { ... } };
```

f) Les classes Porcelaine, Ballon, BallonRugby et BallonFoot sont identiques à **c).**

```
class Objet { public: ...
      virtual void rebondir() { ... } };
```

Problème (Constructeurs, destructeurs, héritage)

Exercice 1 : Appel des constructeurs et destructeurs

On a les déclarations et définitions suivantes:

```
#include <iostream.h>

class Objet { public:
        static int nombre_objets = 0;
        int numero ;
        Objet();
        ~Objet();
};

class Porte : public Objet { public:
        Porte();
        ~Porte();
};

class Batiment : public Objet { public:
        Porte * porte;
        int surface;
        Batiment(int);
        ~Batiment();
};

class Tour : public Objet { public:
        int hauteur;
        Muraille * muraille;
        Tour(int, Muraille *);
        ~Tour();
};

class Donjon : public Tour { public:
        Chateau * chateau;
        Donjon(Chateau *, int);
        ~Donjon();
};

class Muraille : public Objet { public:
        int longueur;
        Tour * tour1;
        Tour * tour2;
        Chateau * chateau;
        Muraille(int, int, int, Chateau *);
        ~Muraille();
};

class Chateau : public Batiment { public:
        Donjon * donjon;
        Muraille * muraille;
        Chateau(int, int, int, int, int);
        ~Chateau();
};

int Objet::nombre_objets = 0 ;

Objet::Objet() {
        cout << "++ Objet debut" << endl;
        nombre_objets++;
        numero = nombre_objets ;
        cout << "++ Objet fin" << nombre_objets << endl;
}
Objet::~Objet() {
        cout << "-- Objet debut " << endl;
        nombre_objets--;
        cout << "-- Objet fin " << endl;
}

Porte::Porte() {
        cout << "++ Porte debut" << endl;
```

```
		cout << "++ Porte fin" << endl;
}
Porte::~Porte() {
		cout << "-- Porte debut" << endl;
		cout << "-- Porte fin" << endl;
}

Batiment::Batiment(int s) {
		cout << "++ Batiment debut" << endl;
		porte = new Porte();
		surface = s;
		cout << "++ Batiment fin" << surface << endl;
}
Batiment::~Batiment() {
		cout << "-- Batiment debut" << endl;
		delete porte;
		cout << "-- Batiment fin" << endl;
}

Tour::Tour(int h, Muraille * m) {
		cout << "++ Tour debut" << endl;
		hauteur = h;
		muraille = m;
		cout << "++ Tour fin " << hauteur << endl;
}
Tour::~Tour() {
		cout << "-- Tour debut" << endl;
		cout << "-- Tour fin" << endl;
}

Donjon::Donjon(Chateau * c, int h) : Tour(h, NULL) {
		cout << "++ Donjon debut" << endl;
		chateau = c;
		cout << "++ Donjon fin" << endl;
}
Donjon::~Donjon() {
		cout << "-- Donjon debut" << endl;
		cout << "-- Donjon fin" << endl;
}

Muraille::Muraille(int l, int h1, int h2, Chateau * c) {
		cout << "++ Muraille debut" << endl;
		longueur = l;
		tour1 = new Tour(h1, this);
		tour2 = new Tour(h2, this);
		chateau = c;
		cout << "++ Muraille fin " << longueur << endl;
}
Muraille::~Muraille() {
		cout << "-- Muraille debut" << endl;
		delete tour1;
		delete tour2;
		cout << "-- Muraille fin" << endl;
}

Chateau::Chateau(int l, int h1, int h2, int hd, int s) : Batiment(s) {
		cout << "++ Chateau debut" << endl;
		muraille = new Muraille(l, h1, h2, this);
		donjon = new Donjon(this, hd);
		cout << "++ Chateau fin" << endl;
}
Chateau::~Chateau() {
		cout << "-- Chateau debut" << endl;
		delete muraille;
		delete donjon;
		cout << "-- Chateau fin" << endl;
}


main() {
		Chateau chateau(400, 40, 60, 80, 10000);
}
```

1) Dessiner le graphe d'héritage des classes.

2) Donner la sortie du programme.

Exercice 2 : Membres d'instance et membres de classe

En C++, comment précise-t-on le "type" :
"d'instance"
ou bien:
"de classe"
d'un membre ou d'une fonction membre ?

Pour chaque (fonction) membre des classes suivantes, nous avons volontairement omis le mot-clé static quand il était nécessaire. Malgré tout, en vous aidant du nom du membre, indiquer si le membre est "d'instance" ou bien "de classe".

```
class Objet {
        int nombre_objets;
        Liste * liste_objets;
        int numero;
        ...
};
class Chateau : public Batiment { public:
        Muraille * muraille;
        Liste * liste_tours;
        Liste * liste_chateaux;
        void imprimer_la_muraille();
        void imprimer_les_tours();
        void imprimer_les_chateaux();
        void imprimer();
        ...
};
```

Exercice 3 : Structure de la mémoire

Dessiner la structure de la mémoire après l'exécution de l'instruction `Chateau chateau(400, 40, 60, 80, 10000);` et avant l'accolade `}` en utilisant le formalisme utilisé en cours.

Exercice 4 : Constructeurs par recopie

Soit le programme principal suivant :

```
main() {
        Chateau a(400, 40, 60, 80, 10000);
        Chateau b(a);
}
```

1) Pour que le programme principal suivant marche, écrire un constructeur par recopie pour la classe Chateau qui n'utilise que les constructeurs de Muraille et de Donjon définis dans l'énoncé de l'exercice 1. Avantages ? Inconvénients ?

2) Pour réutiliser les classes facilement, écrire les constructeurs par recopie des classes Chateau, Batiment, Muraille, Tour, Donjon, Porte, Objet.

Exercice 5 : Fonctions virtuelles

1) On ajoute la fonction membre defendre() dans la classe Batiment :

```
class Batiment { public: ... void defendre(); ... };
Batiment::defendre() {
        cout << "Porte fermée, mais sans plus, le BATIMENT se defend" << endl;
}
```

Le programme principal est :

```
main () {
        Chateau chateau(400, 40, 60, 80, 10000);
        chateau.defendre();
}
```

Donner la sortie de l'instruction chateau.defendre();

2) On définit les fonctions membres defendre() dans les classes suivantes :

```
class Chateau { public: ... void defendre(); ... };
Chateau::defendre() {
        cout << "Fort de ses tours, de son donjon et de sa muraille, le CHATEAU se defend" << endl;
        donjon->defendre();
        muraille->tour1->defendre();
        muraille->tour2->defendre();
        muraille->defendre();
}
class Tour { public: ... void defendre(); ... };
Tour::defendre() {
        cout << "Du haut de ses " << hauteur << " metres, la TOUR se defend" << endl;
}
class Muraille { public: ... void defendre(); ... };
Muraille::defendre() {
        cout << "Du long de ses " << longueur << " metres, la MURAILLE se defend" << endl;
}
```

Donner la sortie de l'instruction chateau.defendre();

3) On modifie la déclaration de la fonction membre defendre() de la classe Batiment :

```
class Batiment { public: ... virtual void defendre(); ... };
```

Donner la sortie de l'instruction chateau.defendre();

4) On suppose que la fonction defendre() est redéfinie dans la classe Donjon :

```
class Donjon { public: ... void defendre(); ... };
Donjon::defendre() {
        cout << "Du haut de ses " << hauteur << " metres, le DONJON se defend" << endl;
}
```

Donner la sortie de l'instruction chateau.defendre();

5) On suppose que la définition de Chateau::defendre() est maintenant:

```
Chateau::defendre() {
        cout << "Fort de ses tours, le CHATEAU se defend" << endl;
        Tour * t;
        t = donjon;
        t->defendre();
        t = muraille->tour1;
        t->defendre();
        t = muraille->tour2;
        t->defendre();
}
```

Donner la sortie de l'instruction chateau.defendre();

6) On suppose que la déclaration de defendre() de la classe Tour est maintenant:

```
class Tour { public: ... virtual void defendre(); ... };
```

Donner la sortie de l'instruction chateau.defendre();