```
  while (stream >> t) {
    if (occ.find(t) == occ.end()) {
      occ[t] = 0;
    }
    ++occ[t];
  }
  map<string,int>::iterator iter;
  for (iter = occ.begin(); iter != occ.end(); ++iter) {
    cout << (*iter).first << " = " << (*iter).second <<  endl;
  }
```

1.8 améliorations C++0X et C++11

utiliser g++ 4.7 et l'option `--std=c++11` pour compiler.

1.8.1 inférence de type automatique

Aussi appelée inférence automatique, elle consiste pour le compilateur à déterminer le type des variables sans que l'utilisateur ait à le spécifier explicitement :

```
vector<int> v;
vector<int>::iterator i = v.begin();
```

On pourra alors remplacer par :

```
auto i = v.begin();
```

Le mot clé auto demande donc au compilateur d'inférer le type de la variable lors de sa déclaration.

Une autre mot-clé decltype permet d'inférer le type d'une nouvelle variable à partir d'une variable existante :

```
map<int, int>::iterator start;
decltype(start) end;
```

1.8.2 boucles

On peut dorénavant comme en Java ou PHP simplifier l'écriture d'une itération sur un container :

```
vector<int> v;
vector<int>::iterator i;
for (i = v.begin(); i != v.end(); ++i) {
  ...
}
```

sera remplacé par

```
for (int i : v) {
...
}
```

de la même manière :

```
int data[] = { 1, 2, 3};
for (int& x : data) {
  x = x * 2; // double each value
}
```

1.8.3 nouvelles fonctionnalités liées aux constructeurs

enchaînement de constructeurs

Ou appel d'un constructeur par un autre constructeur, ce que l'on ne pouvait pas faire auparavant.

```
class A {
protected:
  int value_i;
  string value_s;
public:
  A(int a, string s) : value_i(a), value_s(s) { }
  A() :: A(0, "") {}
};
```

initialisation uniforme

```
A a{20, "toto"};
A tab[2]= { {1, "a"}, {2, "b"} };

A f() {
  return {1, "a"};
}

vector<int> data({1, 2, 3});

class Array {
 vector<int> values;
public:
  Array(std::initializer_list<int> data) {
    for (int v : data) {
      values.push_back(v);
    }
  }
}
```

1.8.4 mot clé constrexpr

Il permet de définir une expression constante notamment s'il s'agit d'une fonction.

```
constexpr int getSize() { return 200; }
int a[getSize()];
```

on peut également l'utiliser pour les types liés aux fonctions :

```
typedef int (*FuncType)(int a, double b);
FuncType ptrFunc = &myFunction;
// sera remplace par
using FuncType = int (*)(int, double);
FuncType ptrFunc = &myFunction;
int a[getSize()];
```

1.8.5 alias de type

Permet de définir un alias de type grâce au mot clé `using` :

```
using MyType int;
// equivalent a
typedef int MyType;
```

1.8.6 pointeur null constant

En C un pointeur nul est défini par la valeur 0 ou la constante `NULL`.

```
void f(char *s) { .. }
void f(int n) {...}

f(NULL); // doit appeler f(int)
f((char *)NULL) // appelle f(char *)

f(nullptr) // appelle f(char *)
```

1.8.7 opérateur de conversion explicite

```
class A {
private:
  int value;
public:
  A(int n) : value(n) { }
  operator int() const { return value; }
}

A a1(1111);
int a2 = a1; // realise l'appel du constructeur
int a2 = int(a1); // appelle operateur
```

avec explicit :

```
class A {
private:
  int value;
public:
```

```
  A(int n) : value(n) { }
  explicit operator int() const { return value; }
}

A a1(1111);
int a2 = a1; // erreur
int a2 = static_cast<int>(a1);
```

1.8.8 littéraux définis par l'utilisateur

permet de définir des suffixes utilisés pour convertir des quantités :

```
std::complex<double> operator"" _i(double d) {
  return complex<double>(0, d);
}
complex<double> a = 1_i;
```

1.8.9 fonction lambda

Ce sont des fonctions anonymes (sans nom explicite) en ligne (inline).

```
[capture_block](parameters) mutable exception -> return_type { body }
```

- **capture_block** : indique comment les variables externes à la fonction sont utilisées dans le corps de la fonction
- **parameters** : paramètres de la fonction (arguments)
- **mutable** (optionnel) : les variables externes à la fonction sont par défaut copiée et apparaissent comme constantes. Avec le mot clé mutable, elle sont modifiables mais toujours copiées (dupliquées).

Voici quelques exemples :

```
// definition mais non appelee
[] { cout << "hello" << endl; };
// definie et appelee
[] { cout << "hello" << endl; } ();

auto dbl = [](const int n){ return 2*n; };
int a = dbl(1);
int b = dbl(2);
```

En ce qui concerne la capture, on dispose de deux possibilités qui peuvent ensuite être combinées et énumérées sur les variables :

- `[=]` : capture toutes les variables par valeur
- `[&]` : capture toutes les variables par référence

Par exemple :

- `[&x]` : capture x par référence
- `[=, &x]` : capture toutes les variables par valeur, sauf x par référence

```
vector<int> v(10);
int index=0;
generate(v.begin(), v.end(), [&index]() {return index++;});
for_each(v.begin(), v.end(), [](int i){ cout << i << " "; });
// au lieu de
copy(v.begin(), v.end(), ostream_iterator<int>(cout," "));
```

1.8.10 expressions régulières

pas encore au point.

```
regex pattern("i[sp]+i");
match result;
if (regex_match("Mississippi", result, pattern)) {
  for (int i=0; i<result.size(); ++i) {
    cout << "found " << result[i] << endl;
  }
}
```

1.8.11 codage des caractères

There are three Unicode encodings that C++11 will support : UTF-8, UTF-16, and UTF-32. In addition to the previously noted changes to the definition of char, C++11 adds two new character types : `char16_t` and `char32_t`. These are designed to store UTF-16 and UTF-32 respectively. The following shows how to create string literals for each of these encodings :

```
u8"I'm a UTF-8 string \u2018."
u"This is a UTF-16 string."
U"This is a UTF-32 string."
```

It is also sometimes useful to avoid escaping strings manually, particularly for using literals of XML files, scripting languages, or regular expressions. C++11 provides a raw string literal :

```
R"(The String Data \ Stuff " )"
R"delimiter(The String Data \ Stuff " )delimiter"
u8R"XXX(I'm a "raw UTF-8" string.)XXX"
```