Cpp Tricks'n'Quirks #2 [detecting'N'debugging ctors] fix (wide. 80)
RKr (edit. 3)

Gdy zostanie stworzona taka klasa jak poniżej, mająca wiele konstruktorów,
czasem trudno jest odgadnąć, który konstruktor będzie użyty w danej sytuacji.
Jeśli nie znamy wszystkich reguł wyboru konstruktora możemy wpaść w pułapkę,
będziemy oczekiwać inwokacji pewnego konstruktora,
jednocześnie nie rozumiejąc dlaczego wołany jest inny.

```
class ExClass
{
  int num;
  public: ExClass() : num(0) { }
  public: ExClass(int n) : num(n) { }
  public: ExClass(const ExClass& copyable) : num(copyable.num) { }
  public: ExClass(ExClass&& movable) : num(movable.num) { movable.num = 0; }
};

int main()
{
///// cases 2,4,6,8 don't invoke assignments!
/*1*/ ExClass a;
/*2*/ ExClass aa = ExClass();
/*3*/ ExClass b(5);
/*4*/ ExClass bb = ExClass(5);
/*5*/ ExClass c(b);
/*6*/ ExClass cc = ExClass(b);
/*7*/ ExClass d( std::move(b) );
/*8*/ ExClass dd = ExClass( std::move(b) );

    return 0;
}
```

Można by w takim kodzie jak powyżej dodawać komunikaty poprzez printf/cout,
ale wtedy kłopotliwe jest ich aktywowanie/deaktywowanie, wklejanie/wycinanie,
aby nie wprowadzały bałaganu do finalnego programu i standardowego wyjścia.

Możemy jednak zaimplementować zestaw funkcji realizujących takie zadanie,
w taki sposób, aby działanie wszystkich tych funkcji, we wszystkich klasach,
w całym programie, było aktywowane lub deaktywowane jednym przełącznikiem.

```
#if 1 // switch between 0 and 1 ...
  #define ENABLE_CTOR_DBG_PRINTs
#endif // ... to respectively disable or enable CTOR DETECTION PRINTs

void dbgCtorEmpty() {
  #ifdef ENABLE_CTOR_DBG_PRINTs
    cout << "dbg[ctor] empty" << endl;
  #endif
}

void dbgCtorParam() {
  #ifdef ENABLE_CTOR_DBG_PRINTs
    cout << "dbg[ctor] param" << endl;
  #endif
}

void dbgCtorCopy() {
  #ifdef ENABLE_CTOR_DBG_PRINTs
    cout << "dbg[ctor] copy" << endl;
  #endif
}

void dbgCtorMove() {
  #ifdef ENABLE_CTOR_DBG_PRINTs
    cout << "dbg[ctor] move" << endl;
  #endif
}
```

```
class ExClass
{
  int num;
  public: ExClass() : num(0)
            { dbgCtorEmpty(); }
  public: ExClass(int n) : num(n)
            { dbgCtorParam(); }
  public: ExClass(const ExClass& copyable) : num(copyable.num)
            { dbgCtorCopy(); }
  public: ExClass(ExClass&& movable) : num(movable.num)
            { dbgCtorMove(); movable.num = 0; }
};

int main()
{                                           // expected dbg-messages in stdout:
///// cases 2,4,6,8 don't invoke assignments //
/*1*/ ExClass a;                            // dbg[ctor] empty
/*2*/ ExClass aa = ExClass();               // dbg[ctor] empty
/*3*/ ExClass b(5);                         // dbg[ctor] param
/*4*/ ExClass bb = ExClass(5);              // dbg[ctor] param
/*5*/ ExClass c(b);                         // dbg[ctor] copy
/*6*/ ExClass cc = ExClass(b);              // dbg[ctor] copy
/*7*/ ExClass d( std::move(b) );            // dbg[ctor] move
/*8*/ ExClass dd = ExClass( std::move(b) );  // dbg[ctor] move

    return 0;
}
```

Tę samą sztuczkę można wykorzystać do detekcji wszelkich operatorów,
np. przypisania kopiującego i przenoszącego, poniżej różne warianty ich sygnatur.

```
  public: void operator=([const] ExClass& copyable) { ... }
lub
  public: ExClass& operator=([const] ExClass& copyable) { ... }

  public: void operator=([const] ExClass&& movable) { ... }
lub
  public: ExClass& operator=([const] ExClass&& movable) { ... }
```

Operator posiadający **RetType** zdefiniowany jako `void`,
uniemożliwia łańcuchowe złożenia operatora, tj. "x = y = z" zgłosi błąd.
Modyfikatory `const` w typach parametrów wejściowych są opcjonalne,
powinny zgadzać się z intencjami w naszej implementacji,
jednak typowym scenariuszem jest copy-op z `const` oraz move-op bez `const`.
Dzieje się tak dlatego, że na ogół nie chcemy modyfikować obiektów copyable,
natomiast chcemy modyfikować (czyścić/markować) obiekty movable.