VYLEPŠOVANIE KONCEPTU TRIEDY

Typy tried

- **class** - definuje premenné a metódy (funkcie). Ak nie je špecifikovaná inak, viditeľnosť členov je ***private***.
- **struct** - definuje premenné a metódy (funkcie). Ak nie je špecifikovaná inak, viditeľnosť členov je ***public.***
- **union** - definuje premenné a metódy (funkcie). Ak nie je špecifikovaná inak, viditeľnosť členov je ***public*.** Všetky premenné zdieľajú tú istú adresu. Veľkosť unionu je veľkosť jeho najväčšej premennej. Nemôže byť odvodený z inej triedy/unionu.

Atribúty viditeľnosti

- **public** – každý má prístup k objektom s *public* viditeľnosťou.
- **protected** – len vlastná trieda a jej potomkovia v hierarchii dedenia (odvodenia) majú prístup k objektom s *protected* viditeľnosťou.
- **private** – len trieda definujúca objekt má prístup k objektom s *private* viditeľnosťou

Atribúty viditeľnosti, príklad

```
class A {
public:
   int a;
   int b;
   void fun() {}
protected:
   int c;
   void gun() {}
private:
   int d;
   void hun() {}
};
```

Viditeľnosť zdedených premenných a metód

- Špecifikuje sa pri definícií triedy pred udaním nadradenej triedy
- Ohraničuje viditeľnosť zdedených metód a premenných. Akási (“tranzitívna”) viditeľnosť.
- Týka sa len premenných a metód majúce viditeľnosťou public a protected v nadradenej triede. Ich viditeľnosť sa prenáša do odvodenej triedy nasledovným spôsobom:

1. **public**: public->public, protected->protected
2. **protected**: public->protected, protected->protected
3. **private**: public->private, protected->private

Viditeľnosť zdedených premenných a metód

```
class A {
public:
   int a;
   void fun() {}
protected:
   int b;
   void gun() {}
private:
   int c;
   void hun() {}
};
```

```
class B : public A {
};

class C: protected A {
};

class D : private A {
};
```

Friend trieda, alebo metóda

- Vo vnútri triedy môžete deklarovať, že akási iná trieda, alebo metóda bude mať prístup k private a protected členom.

```
class A {
  private: int a;
  friend void printaa(A *x);
  friend class B;
};

void printaa(A *x) {printf("%d", x->a);}

class B {
  void f(A *x) {printf("%d", x->a);}
};
```

Žiadna chyba

Statické premenné triedy

```
class A {
public:
  int i;
  static int j;
};

int A::j = 7; // definition!

int main() {
  A a, b;
  a.i = 0; a.j = 1;
  b.i = 2; b.j = 3;
  printf("a.i == %d\n", a.i);
  printf("a.j == %d\n", a.j);
  printf("b.i == %d\n", b.i);
  printf("b.j == %d\n", b.j);
}
```

```
Vypíše:

a.i == 0
a.j == 3
b.i == 2
b.j == 3
```

Statické funkcie triedy

```
class A {
public:
  int i;
  static int j;
  static void fun() { j = 0; }
  static void gun();
};

int A::j = 7;
void A::gun() { j = 2;}

int main() {
  A a, b;
  a.i = 0; a.j = 1;
  b.fun(); b.gun();
  printf("a.i == %d\n", a.i);
  printf("a.j == %d\n", a.j);
}
```

```
Vypíše:

a.i == 0
a.j == 2
```

Scope (obzor, pôsobnosť, ...)

- Symboly definované v "scope" sú v jeho rámci viditeľné bez prefixu, ale mimo nie sú viditeľné vôbec.
- { ... } v kóde vytvárajú "scope".
- Trieda v C++ vytvára "scope" (na rozdiel od C).

Scope (obzor, pôsobnosť, ...)

Nasledovný kód je platný v C ale nie v C++ !

```
struct A {
  struct B {
     int b1, b2;
  } x;
  int a1, a2;
};

int main() {
  struct A a;
  struct B b;
  b = a.x;
  b.b1 = 0;
}
```

Post-definície a scope

```
int i;

class A {
  int i, j;
  void fun();
};

int j;

void A::fun() {
  i = 0;       // A.i, nie globalne i
  j = 0;       // A.j, nie globalne j
}
```

Vnorené triedy (nested classes)

```
class A {
   int a;
   class B {
      int b;
      void g();
   } b;
   class C *c;
};

A::B::g() {}

class A::C {
   int x;
};
```

Virtuálne funkcie a konštruktory

- Volanie virtuálnych funkcií v konštruktore nie je virtuálne! Nepoužije sa tabuľka virtuálnych funkcií.

```
class A {
  public:
  virtual void f() {printf("A::f()");}
  A() {f();}
};

class B : public A {
  public:
  virtual void f() {printf("B::f()");}
  B() { }
};

int main() { B b; }
```

Vypíše:
A::f()

Virtuálne funkcie a konštruktory

```
class A {
public:
  virtual void f() {printf("A::f()\n");}
  void g() {f();}
       A() {f();}  // always calls A::f()
};

class B : public A {
public:
  virtual void f() {printf("B::f()\n");}
  B() { g(); A(); }
};

int main() { B b; }
```

Vypíše:

```
A::f()
B::f()
A::f()
```

Nevirtuálne volania virtuálnych metód

- OOP jazyky zvyčajne umožňujú volanie rovnakej metódy z nadradenej triedy. Napríklad v Jave je to volanie cez kľúčové slovo super.xxx(...). A C++ ?

```
class A {
public:  virtual void f() { ... }
};

class B : public A {
public:
  virtual void f() { ...
    super::f();
    A::f();
  }
};
```

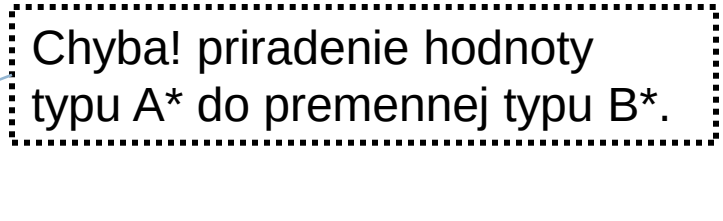
Nevirtuálne volania virtuálnych metód

- V C++ je možné explicitne uviesť triedu, ktorej metóda sa vyvolá. Funguje to aj pre odstránenie nejednoznačnosti pri viacnásobnom dedení.

```
class A {public:  virtual void f() { ... } };
class B {public:  virtual void f() { ... } };

class C : public A, public B {
};

int main() {
   C c;
   c.f();          // chyba A::f alebo B::f
   c.A::f();       // O.K. A::f
}
```

Zmena návratového typu predefinovaných metód, motivácia

```
class A {
  virtual A *clone() { ... }
};

class B : public A {
  virtual A *clone() { ... }
};

int main() {
  B *pb1, *pb2;
  pb1 = pb2->clone();
}
```



Zmena návratového typu predefinovaných metód

- Zmeniť návratový typ možno konzistentným spôsobom v rámci hierarchie

```
class A {
  virtual A *clone() { ... }
};

class B : public A {
  virtual B *clone() { ... }
};

int main() {
  B *pb1, *pb2;
  pb1 = pb2->clone();
}
```

Konštantné premenné, konštanty

```
const double pi = 3.14159;

const int velkost_tuto = 1000;

int tuto[velkost_tuto];

const char *p;
char *const q;


int main() {
   p = "toto";
   q = p;
   p = q;
}
```

V C++ možno konštantnú premennú použiť v konštantných výrazoch

Pointer na nemenný char

Nemenný pointer na char

Chyba

Metóda nemeniaca svoj objekt (const metóda)

```
class A {
   int a;
public:
   void getA() const {
       return a;
   }
   void getComplexCalculationResult() const {
       ...
       ... (a++) ...
       ...
   }
};
```

Chyba! metóda deklarovaná ako const nesmie meniť svoj objekt

Const a mutable

```
class A {
public:
   int x;
   mutable int y;
};

const A a;

void main() {
   a.x = 0;
   a.y = 1;
}
```

Chyba! Pokus o priradenie hodnoty do konštantného objektu

Žiadna chyba! Premenná `y` je deklarovaná mutable. T.j. možno ju vždy meniť (aj keď je súčasť konštantného objektu).

Smerník na metódu, motivácia

```
void map(void (*fun)(int x)) {
  fun(77);
}

void g(int x) { printf("%d", x);}

int main() {
   map( & g );
}
```

Program v C, ktorý vypíše: 77

Tento program funguje rovnako aj v C++. Ako to ale urobiť, ak potrebujem získať pointer na metódu (z nejakej triedy) a potom ju cez tento pointer vyvolať?

Smerník na metódu

```
class A {
public:
  void g(int x) { printf("%d", x); }
};

void map(void (A::*fun)(int x)) {
  A   a, *pa;
  (a.*fun)(77);
  (pa->*fun)(88);
}

int main() {
   map(& A::g);
}
```

Vypíše:
7788

Default parameter

```
int fun(int i, int j=10) {
  return(i * j);
}

int main() {
  int a, b;
  a = fun(2,3);
  b = fun(2);
}
```

a == 6

b == 20

Base and members initializers syntax

```
class X : public A, public B {
  int xx;
  X(int a, int b) : A(a), B(b), xx(0) {}
};
```