STU FEI

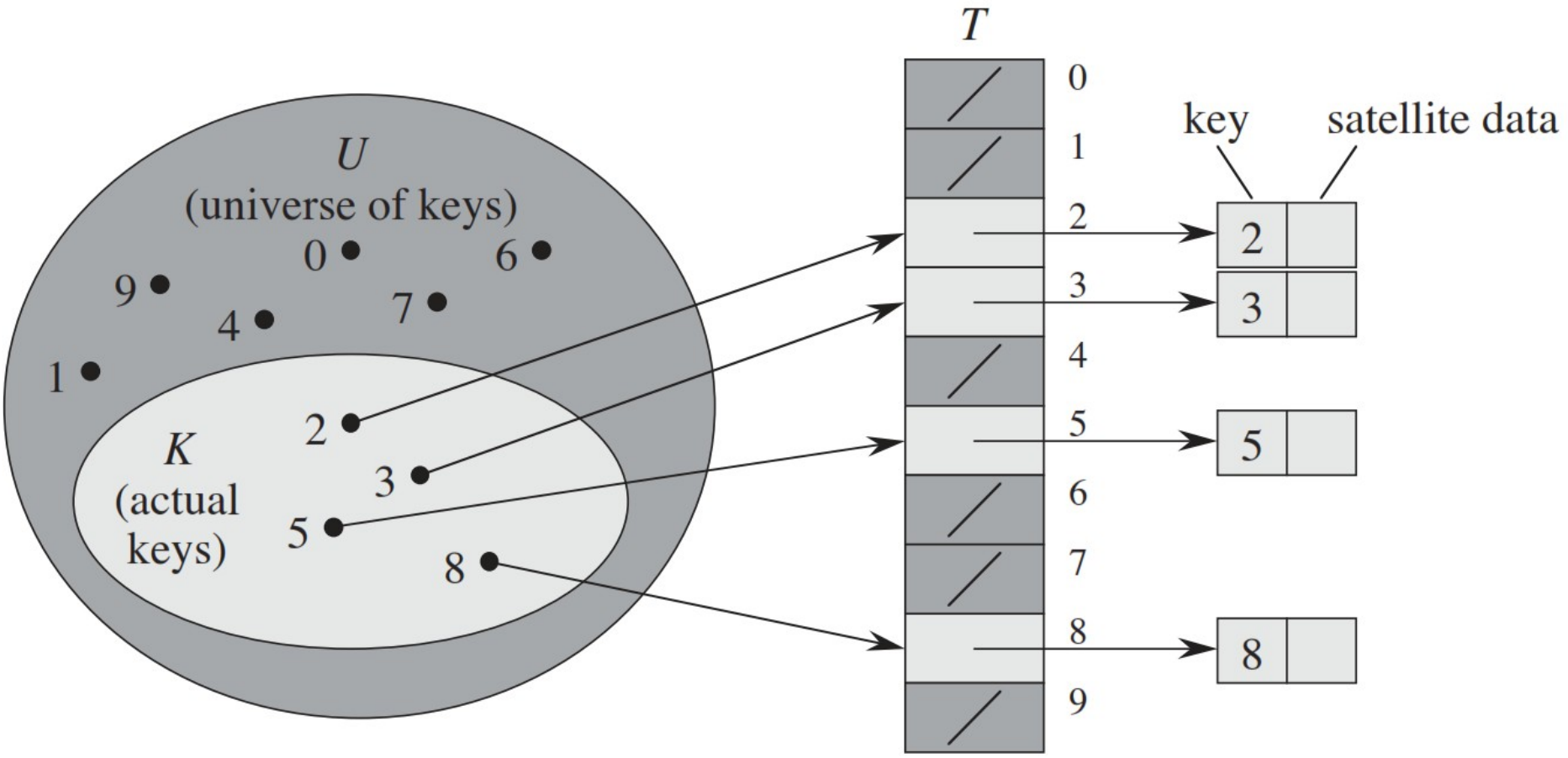
Programovacie techniky

9. hašovanie, unordered kontajner, odvodená trieda, virtuálna metóda, inline, friend, const metóda

Prednáška

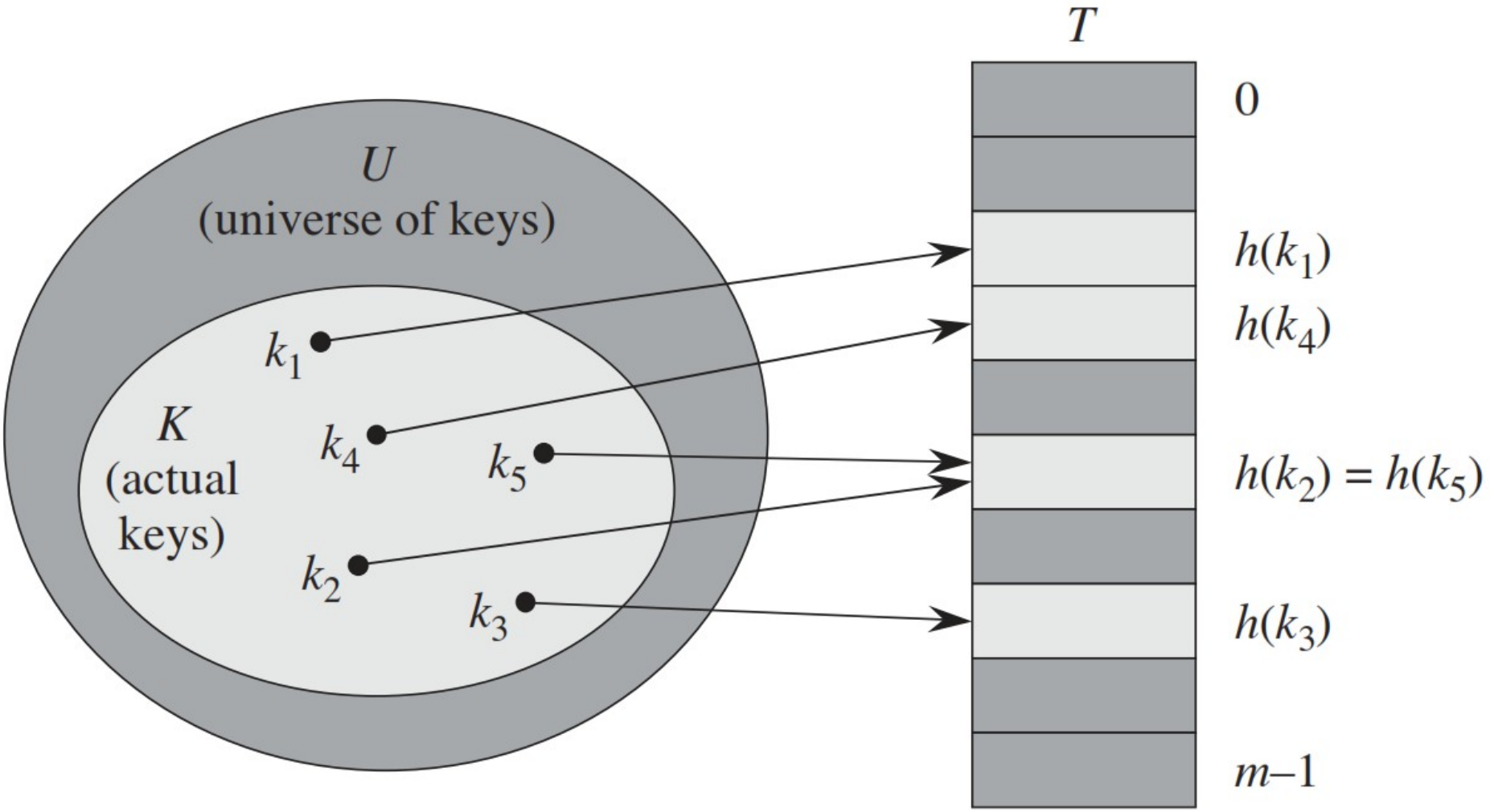

qikkDB
20.11.2019 13:00
Veľké dáta
Rýchlejšie a efektívnejšie.
Prednáška v BC300
Peter Krátky
instarea qikk.ly

Tabuľka s priamym prístupom



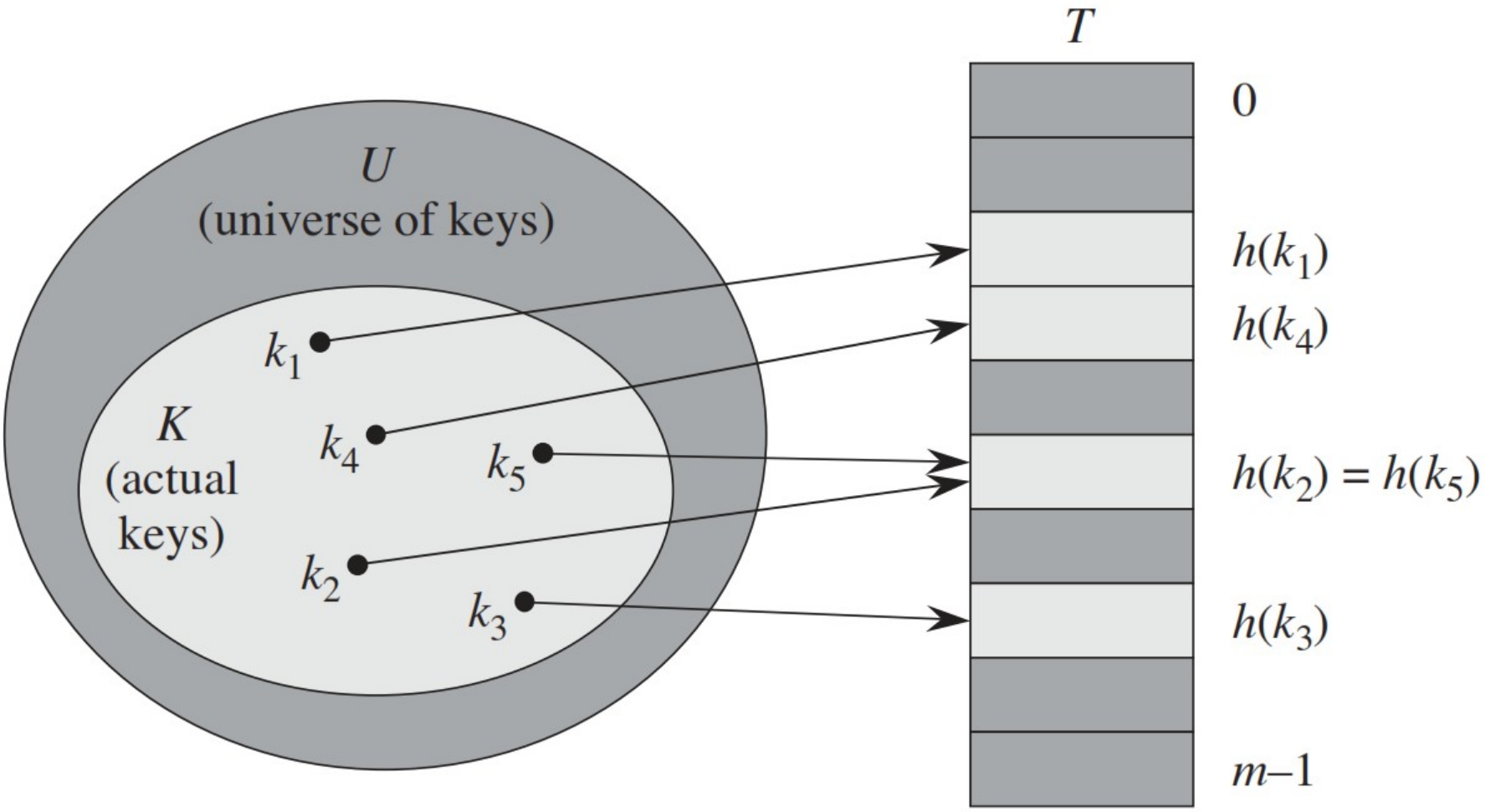
Nevýhoda: pre veľký počet položiek je potrebné veľké pole T

Hašovanie



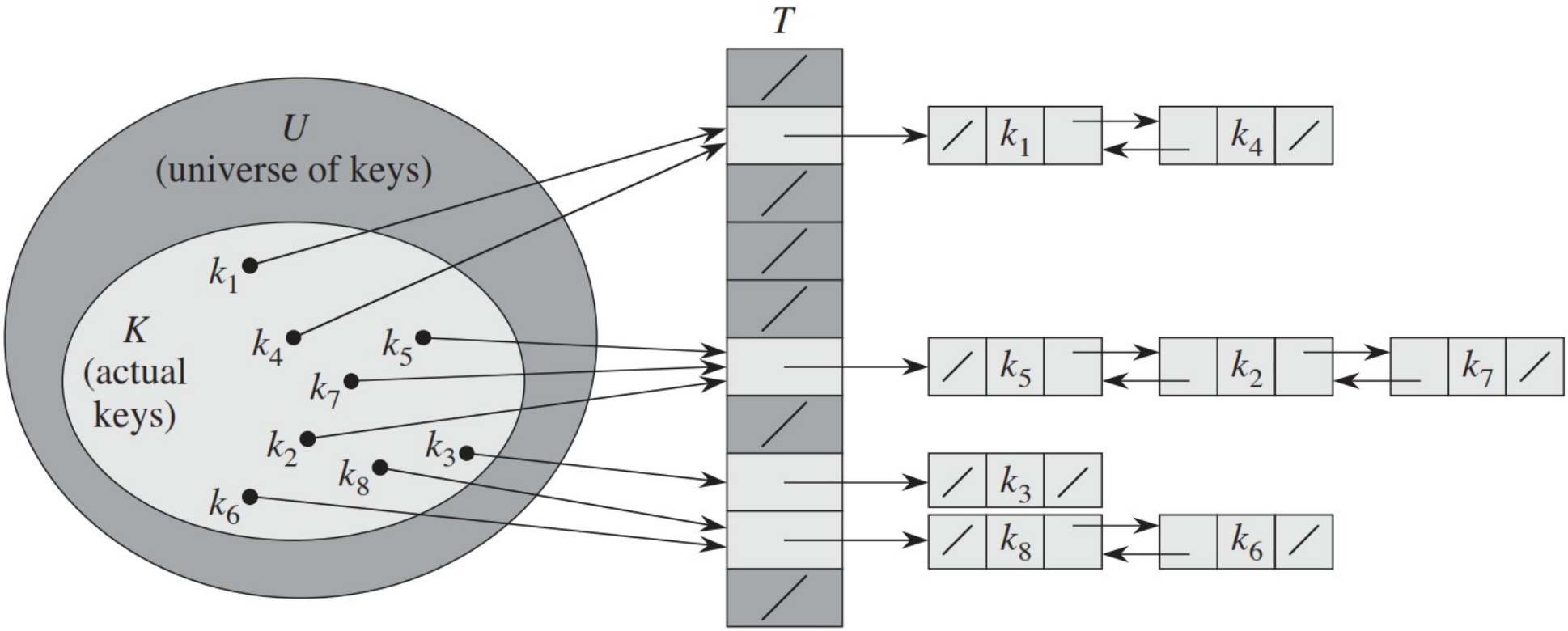
h je hašovacia funkcia, ktorá vloží položku ki na pozíciu h(ki) v poli T, ktoré má veľkosť m

Hašovanie: kolízia



k2 a k5 sú hašované na to isté miesto v poli T (ako minimalizovať kolízie?)

Hašovanie: kolízia



Pri kolízii môžeme použiť zreťazený zoznam (ako obmedziť dĺžku zreťazeného zoznamu?)

Hašovacia funkcia

Definícia:

$$h : U \rightarrow \{0, 1, \ldots, m-1\}$$

kde U je množina kľúčov a m je veľkosť hašovacej tabuľky (pola)

Potrebujeme:

$$\mathrm{E}[n_j] = n/m$$

Očakávaný počet prvkov nj na každej pozícii tabuľky T je rovnaký

Modulo metóda

Definícia:

$$h(k) = k \bmod m$$

Čo zvoliť ako m?

Najhoršia voľba: $m = 2^p$

Lepšia voľba: $m = 2^p - 1$

Ešte lepšia voľba?

Modulo metóda

p = prvočíslo

n = 2000, kde n je počet prvkov
max. 3 prvky na jednu pozíciu
m = n / 3
Prvočíslo p väčšie ako m je napr. 701

Hašovacia funkcia:

$$h(k) = k \bmod 701$$

unordered_map

```
#include <unordered_map>
#include <iostream>

int main() {
  std::unordered_map<std::string, int> m;
  m.insert({"Joe", 1}); m.insert({"Mary", 5});
  m.insert({"Zoe", 6});

  for(auto i : m) {
    std::cout << i.first << " " << i.second << " ";
  }

  std::cout << m.bucket_count(); //5

  return 0; }
```

Zoe 6 Joe 1
Mary 5

Počet košov

Porovnanie

Prvky v kontajneri nie sú zoradené:

std::unordered_set
std::unordered_multiset

std::unordered_map
std::unordered_multimap

Vkladanie, hľadanie a odstraňovanie prvkov v priemere O(1), v najhoršom prípade O(n). Využíva hašovanie.

Prvky v kontajneri sú zoradené:

std::set
std::multiset

std::map
std::multimap

Vkladanie, hľadanie a odstraňovanie prvkov O(log n). Využíva vyvážený binárny strom.

Základná trieda (base class)

```
#include <iostream>
using namespace std;

class Clovek {
private:
  int time = 0;

protected:  //prístupné pre odvodené triedy
   char pohlavie = 'y';
   int vek = 0;

public:
  int getTime() {
    return time;
  }
};
```

Odvodená trieda (derived class)

```
class Student: public Clovek {
   public:
      void setPriemer(double p) {
         priemer = p;
      }

      double getPriemer() {
         return priemer;
      }

   private:
      double priemer = 0.0;
};
```

Odvodená trieda (derived class)

```
int main(){
  Clovek t;
  Student s;

  //getTime() je metóda triedy Clovek
  cout << s.getTime() << endl;

  //getPriemer je metóda triedy Student
  cout << s.getPriemer();

  return 0;
}
```

formát

class derived-class: access-specifier base-class

access-specifier

- public
- private
- protected

class Student: public Clovek
class Student: private Clovek
class Student: protected Clovek

Viacnásobné dedenie:

- class derived-class: access baseA, access baseB....

access specifier

public base class:

- public ostane public
- protected ostane protected
- private nie sú pristupné v odvodenej triede

protected base class:

- public a protected sú protected

private base class:

- public a protected sú private

Odvodená trieda

Nezdedí:

friend: „priateľstvo“ nie je zdedené, pretože „priateľ“ by získal plný prístup aj do odvodenej triedy

Automatické generovanie:

Pre každý objekt (vrátane inštancií odvodených tried) je v prípade potreby (ak neexistuje) implicitne generovaný defaultný konštruktor, copy konštruktor, deštruktor a operator=.

Dedenie:

Členy triedy sú dedené v zmysle použitého špecifikátora prístupu.

Zdedený operator+, generovaný operator=

```
#include <iostream>
using namespace std;

class Token {
public:
  Token operator+(const Token& t) {
    cout << "operator+(const Token&)";

    return Token();
  }
};
```

Zdedený operator+, implicitný operator=

```
class mToken: public Token {
public:
  int size=0;
};

int main () {
  mToken mt0, mt1, mt2;
  mt0 = mt1 + mt2;

  return 0;
}
```

Neskompiluje, zdedený operator+ vráti objekt typu Token a implicitne vytvorený operator= triedy mToken ho nevie priradiť mt0.

no match for ‘operator=’ (operand types are ‘mToken’ and ‘Token’)

Zdedený operator+, implicitný operator=

```
class mToken: public Token {
public:
  int size=0;

class mToken: public Token {
public:
  int size=0;

  mToken operator+(const mToken& t) {
    cout << "operator+(const mToken&)";

    return mToken();
  }
};
```

Skompiluje, pretože operator+ triedy mToken vráti mToken a implicitne vytvorený operator= triedy mToken ho vie priradiť mt0.

Virtuálna metóda

Trieda, ktorá obsahuje čisto virtuálnu metódu (pure virtual method) sa nazýva abstraktná trieda

```
class Clovek {
private:
  int time = 0;

protected:  //prístupné pre odvodené triedy
   char pohlavie = 'y';
   int vek = 0;

public:
  int getTime() {
    return time;
  }
  virtual double getPriemer() = 0; //čisto virtuálna metóda
};
```

virtual metóda musí byť zadefinovaná v odvodenej triede

Virtuálna metóda

```
class Student: public Clovek {
public:
  void setPriemer(double p) {
    priemer = p;
  }

  //double getPriemer() {
  //return priemer;
  //}

private:
  double priemer = 0.0;
};
```

error: cannot declare variable ‘s’ to be of abstract type ‘Student’ … because the following virtual functions are pure within ‘Student’: … virtual double getPriemer() = 0;

Defaultný konštruktor

```
class Token {
private:
  int time = 0;

public:
  Token(int t): time(t) {};
};

class mToken: public Token {
private:
  int size = 0;

public:
  mToken(int s):size(s){};
};
```

Chýba defaultný konštruktor

Defaultný konštruktor

```
int main(){
  mToken m(10);

  return 0;
}
```

error: no matching function for call to ‘Token::Token()’

Základná trieda potrebuje defaultný konštruktor teda konštruktor, ktorý je možné zavolať bez parametrov

Defaultný konštruktor

```
class Token {
private:
  int time = 0;

public:
  Token() {};
  Token(int t): time(t) {};
};

class mToken: public Token {
private:
  int size = 0;

public:
  mToken(int s):size(s){};
};
```

Defaultný konštruktor

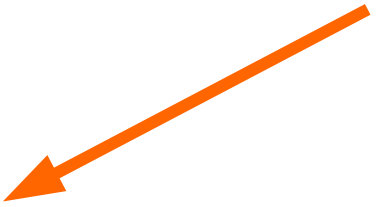
Konštruktor: ktorý použiť

```
class Token {
private:
  int time=0;

public:
  //Token() {};
  Token(int t): time(t) {};
};

class mToken: public Token {
private:
  int size=0;

public:
  mToken(int s, int t): Token(t), size(s) {};
};
```



private

```
class Token {
private:
  int time=0;

public:
  int x = 0;
  Token(int t): time(t) {};
};

class mToken: private Token {
private:
  int size=0;

public:
  mToken(int s, int t): Token(t), size(s) {
    x = 10;
  }; };
```

x je private premenná triedy mToken

private

```
class mToken2: public mToken{
public:
  int getX() {
    return x;
  }
};
```

‘int Token::x’ is inaccessible

protected

```
class mToken: protected Token {
private:
  int size=0;

public:
  mToken(int s, int t):Token(t), size(s) {
    x = 10;
  };
};

class mToken2: public mToken{
public:
  int getX() {
    return x;
  }
};
```

x je protected premenná triedy mToken
Takže to skompiluje :)

inline funkcia

Kompilátor použije skompilovaný kód priamo na mieste volania

```
#include <iostream>
using namespace std;

inline void hello() {
  cout << "hello";
}

int main() {
  hello(); //skompilovaný kód funkcie
  hello(); //skompilovaný kód funkcie ešte raz

  return 0;
}
```

friend

```
#include <iostream>
#include <ostream>

class Token {
  friend std::ostream& operator<<(std::ostream&,
const Token&);
  private:
  int a{1};
};

inline std::ostream& operator<<(std::ostream&
stream, const Token& t0) {
  stream << t0.a;
  return stream;
}
```

friend

```
int main() {
  Token t0;
  std::cout << t0;

  return 0;
}
```

Private/protected premenné/metódy nemôžu byť pristúpené mimo triedu, okrem friend!

const

```
class Token {
  private:
  int a{1};
  public:
  int getA() const {
    return a;
  }
};
```

const metóda nemutuje (nemení hodnoty) premenné triedy

const

```
void foo(const Token& t) {
  int a = t.getA();
}

int main() {
  Token t;
  foo(t);

  return 0;
}
```

getA() musí byť const, pretože t je const Token&. Kompilátor musí vedieť, že getA() nič nezmení.