8. Šablony funkcí, tříd. Vlákna v C++

B2B99PPC – Praktické programování v C/C++

Stanislav Vítek

Katedra radioelektroniky
Fakulta elektrotechnická
České vysoké učení v Praze

Přehled témat

- Část 1 – Další aspekty programování v C++

  Rozložení

  Přetypování

- Část 2 – Šablony funkcí a tříd

  Generické funkce

  Generické třídy

Část I

Další aspekty programování v C++

Kopírování objektů

- Často potřebuje kopírovat objekty podobně jako triviální typy

```
1 int b = a;
```

- I tato zdánlivě jednoduchá operace má svá úskalí

```
1 Trida* instance = new Trida;
2 Trida* jinaInstance = instance;
```

- Je zřejmé, že `jinaInstance` není kopie – oba odkazy ukazují na jedno paměťové místo

Už víme: pokud chceme docílit opravdové kopie, musí být pro danou třídu implementován **kopírovací konstruktor**

I. Další aspekty programování v C++

Rozložení

Přetypování

Rozložení

- Termín **rozložení** odkazuje na to, jak jsou atributy a metody třídy uspořádány v paměti
- V některých případech, jako jsou např. virtuální funkce, může kompilátor rozložit instanci do paměti na více než jedno souvislé místo
- Z hlediska optimalizace je to výhodné, ovšem ztěžuje to přenos objektů (tj. bloků paměti) mezi aplikacemi, kopírování rychlými nízkoúrovňovými funkcemi (`memcopy`) serializaci, atd.
- Od `C++14` lze rozdělit datové typy na
  - triviální
  - standardní rozložení
  - POD – Pure Old Data
- Standardní knihovna má šablony funkcí pro určení kategorie typu
  - `is_trivial<T>`
  - `is_standard_layout<T>`
  - `is_pod<T>`

Triviální typy

- Instance triviálních typů zabírají v paměti souvislý blok, který je možné je kopírovat do pole typu `char` nebo `unsigned char` a z nich zpět do proměnné
- Triviální datový typ se vyznačuje tím, že má
  - triviální konstruktor
  - triviální kopírovací konstruktor
  - triviální operátor přiřazení
  - triviální destruktor

O kopírovacím konstruktoru se dozvíme v další části přednášky.

- Triviální v kontextu konstruktor/destruktor/operátor znamená, že
  (a) není definován uživatelem a
  (b) třída
  - nemá žádné virtuální funkce
  - není potomkem bázové třídy s netriviálním konstruktorem/operátorem/destruktorem
  - žádné atributy (metody) nejsou datového typu s netriviálním konstruktorem/operátorem/destruktorem

Triviální typy – příklad

```
1  struct A {
2    int m;
3  };
5  struct B {
6    B() {}
7  };
9  struct C {
10   C(int a, int b) {}
11   C() = default;
12 };
14 //..
15 std::cout << std::is_trivial<A>::value << '\n';
16 std::cout << std::is_trivial<B>::value << '\n';
17 std::cout << std::is_trivial<C>::value << '\n';
```

lec08/01-layouts.cpp

Klíčové slovo `default`

- Explicitní požadavek na vytvoření defaultního konstruktoru kompilátorem
- Vhodné pokud ve třídě existuje už jiný konstruktor → defaultní není automaticky vytvořen
- Zjevně redundantní, ale pokud je defaultní konstruktor definován uživatelem, není splněna podmínka triviálního datového typu

Klíčové slovo lze spojit i s destruktorem a kopírovacím konstruktorem, vhodné pro zpřehlednění kódu

```
1  struct A {
2    // parametrizovany
3    // konstruktor
4    A(int x) {}
5    // instrukce pro vytvoreni
6    // def.  konstruktoru
7    A() = default;
8  };
```

```
1  struct B {
2    // chyba, neni konstruktor
3    int func() = default;
4    // chyba, neni def.  konst.
5    B(int, int) = default;
6    // chyba, konstruktor
7    // s def.  argumentem
8    B(int = 0) = default;
9  };
```

Standardní rozložení

- Datový typ se standardním rozložením splňuje následující vlastnosti
  - všechny nestatické datové členy (atributy a metody) mají totožnou viditelnost
  - všechny nestatické atributy a bázové třídy mají také standardní rozložení
  - nemá žádné virtuální funkce ani virtuálně nedědí bázovou třídu
  - není odvozena z bázové třídy s nestatickými datovými členy

Příklad

```
1  struct Base1 {
2    int i;
3  };
4  // Derived1 nemá
5  // std.  rozložení
6  struct Derived1 : Base1 {
7    int x;
8  };
```

```
1  struct Base2 {
2    void Foo() {}
3  };
4  // Derived2 má
5  // std.  rozložení
6  struct Derived2 : Base2 {
7    int x;
8  };
```

lec08/02-standard-derived.cpp

I. Další aspekty programování v C++

Rozložení

Přetypování

Explicitní konverze

- V C++ lze stejně jako v C použít explicitní a implicitní přetypování
- K dispozici je ještě alternativní formát zápisu – **konverzní konstruktor**
  - má právě jeden parametr

    Každý konstruktor s jedním parametrem je považován za konverzní, lze zakázat pomocí explicit
  - pokud se při vytváření instance objeví přiřazení jiného datového typu, překladač se pokusí najít odpovídající konstruktor

```
1  double a = 3.1415;
3  // implicitní konverze
4  int i = a;
6  // explicitní konverze známá z jazyka C
7  int j = (int)a;
9  // konverzní konstruktor - funkcionální přetypování
10 int k = int(a);
```

Konverzní konstruktor – příklad

```
1  struct Double {
2    Double (int a) { std::cout << "int\n"; }
3    // zákaz implicitní konverze
4    explicit Double (long a) { std::cout << "long\n"; }
5  };
6  // --
7  Double a = 10L;
8  // 10L je long, ale implicitni konverze long -> Double je zakazana
9  // kompilator proto hleda jiny vhodny konstruktor
10 // Je treba ji provest explicitne
11 Double b = Double(10L);
```

lec08/04-converse-constructor.cpp

Implicitní konverze

```
1  struct A {};
3  struct B {
4    // konverze z A (konstructor)
5    B (const A & x) {}
6    // konverze z A (přiřazení)
7    B & operator= (const A & x) {return *this;}
8    // konverze do A (type-cast operátor)
9    operator A() {return A();}
10 };
11 // --
12 A foo;
13 B bar = foo;     // volá se konstruktor
14 bar = foo;       // operátor přiřazení
15 foo = bar;       // volá se type-cast operátor
```

lec08/05-implicit-cast.cpp

Přetypování – `dynamic_cast`

- Pracuje s ukazateli nebo referencemi na třídy (nebo `code *`)
  - **pointer upcast** – konverze z ukazatele na derivovanou třídu na ukazazatel na bázovou třídu
  - **pointer downcast** – konverze z ukazatele na bázovou třídu na ukazatel na derivovanou třídu, pouze pokud je odkazovaný objekt kompletním objektem cílového typu

Příklad

```
1  Base * pba = new Derived;
2  Base * pbb = new Base;
3  Derived * pd;
5  pd = dynamic_cast<Derived*>(pba);
6  if (pd==0) std::cout << "Nulovy ukazatel v prvnim pripade.\n";
8  pd = dynamic_cast<Derived*>(pbb);
9  if (pd==0) std::cout << "Nulovy ukazatel v druhem pripade.\n";
```

lec08/06-dynamic-cast.cpp

Přetypování – static_cast a reinterpret_cast

static_cast

- Pracuje s ukazateli na třídy v relaci
- Na rozdíl od dynamic_cast může odkazovat na nekompletní objekt

```
1  class Base {};
2  class Derived: public Base {};
4  Base * a = new Base;
5  Derived * b = static_cast<Derived*>(a);
```

reinterpret_cast

- Může provádět konverze mezi ukazateli na libovolné datové typy

```
1  class A {};
2  class B {};
4  A * a = new A;
5  B * b = reinterpret_cast<B*>(a);
```

Část II

Šablony funkcí a tříd

II. Šablony funkcí a tříd

Generické funkce

Generické třídy

Generické funkce

- Generická funkce (šablona) je parametrizovaná deklarace a definice funkce
- Generický parametr (parametr šablony) je datový typ, který je kompilátorem substituován, když vzniká nová instance generické funkce
- Generické funkce mohou parametrizovat datové typy svých parametrů nebo návratové hodnoty

Příklad

```
1  // generická funkce
2  T f (T x, T y);
4  // instance generické funkce
5  int f (int x, int y);
6  double f (double x, double y);
7  char f (char x, char y);
```

Generické funkce

```
1  template <class T> // T je generický parametr
2  T max (T x, T y ) {
3    return x > y ? x : y;
4  }
6  template <typename T> // alternativní syntaxe
7  T max ( T x, T y ) {
8    return x > y ? x : y;
9  }
```

- Funkce je generickou funkcí – reálná funkce (instance generické funkce) je odvozena, když je generická funkce použita (tj. volána).
- V cílovém programu může existovat více instancí generické funkce – jedna instance pro každý typ dat.

Generické funkce – instance 1/3

- Instance jsou vytvořeny, když je generická funkce použita:

```
1  template < class T >
2  T max ( T x, T y ) { return x > y ? x : y; }
3  // jméno max - konflikt s std::max -> plně kvalifikované ::max
4  int i = 10, j = 20;
5  unsigned u = 40;
6  char c = 'a';
7  cout << ::max ( i, 10 );  // int max ( int, int )
8  cout << ::max ( i, j );   // int max ( int, int )
9  cout << ::max ( c, 'b' ); // char max ( char, char )
10 cout << ::max ( i, c );   // chyba - nejednoznačné
11 cout << ::max ( i, u );   // chyba - nejednoznačné
```

Při vytváření instance generické funkce nejsou používány standardní typové konverze.

Generické funkce – instance 2/3

- Když skutečné parametry funkce nedávají jednoznačný výběr datového typu její šablony, musí parametr šablony napsat programátor explicitně:

```
1  template <class T>
2  T max ( T x, T y ) {
3    return x > y ? x : y;
4  }
6  int i = 10, j = 20;
7  unsigned u = 40;
8  char c = 'a';
10 cout << ::max<char> (i, c); // char max (char, char)
11 cout << ::max<int> (i, u);  // int max (int, int)
```

Generické funkce – instance 3/3

- Jsou-li typy specifikovány explicitně, může šablona parametrizovat dokonce datový typ návratové hodnoty:

```
1  template <class T>
2  T max ( T x, T y ) {
3    return x > y ? x : y;
4  }
6  int i = 10;
7  char c = 'a';
9  cout << ::max<char> (c, 'b') << endl; // zobrazeno b
10 cout << ::max<int> (c, 'b') << endl; // zobrazeno 98
11 cout << ::max<int,int> (i, c) << endl; // zobrazeno 97
```

Generické funkce – přetížení 1/2

```
1  template < class T > T max ( T x, T y ) {
2    return x > y ? x : y;
3  }
5  template <class T> T max ( T x, T y, T z ) {
6    return x > y ? ( x > z ? x : z ) : ( y > z ? y : z );
7  }
9  int main () {
10   int a = 10, b = 20, c = 30;
11   std::cout << ::max (a, b) << std::endl; // 20
12   std::cout << ::max (a, b, c) << std::endl; // 30
13   return 0;
14 }
```

Generické funkce – přetížení 2/2

- Generická funkce a obyčejná funkce mohou být přetíženy.
- Obyčejná funkce má přednost.

```
1  template <class T> T max (T x, T y) {
2    return x > y ? x : y;
3  }
4  const char * max (const char * x, const char *y) {
5    return strcmp (x, y) > 0 ? x : y;
6  }
8  int main () {
9    std::const char *a = "Hello", *b = "Hi";
10   std::cout << ::max (a, b) << std::endl; // Hi
11   std::cout << ::max ((void*)a, (void*)b) << std::endl;
12   return 0;
13 }
```

Generické funkce – explicitní instance 1/2

- Instance generické funkce může být vytvořena explicitně.
- To může pomoci při hledání chyb v generické funkci:
  - kompilátor má jen omezené šance najít chybu v generické funkci. Kompilátor nezná datové typy, když čte a rozebírá zdrojový text generické funkce, proto nemůže validovat parametry,
  - generická funkce musí být kompilována znovu a znovu pro každý generický parametr,
  - kompilátor může najít zbývající chyby až při vytvoení instancí.

```
1  template < class T > T max ( T x, T y ) {
2    return x > y ? x : y;
3  }
4  // explicitní vytvoření instance generické funkce:
5  template int max (int x, int y);
6  // potlačení generické funkce pro určitý datový typ:
7  const char * max (const char * x, const char * y);
8  // Ve skutečnosti kompilátor hledá "obyčejnou" funkci.
9  // Není-li funkce implementována, nastane chyba.
```

Generické funkce – explicitní instance 2/2

```
1  template < class T > T max ( T x, T y ) {
2    return x > y ? x : y;
3  }
4  struct S {
5    int a;
6  };
8  int main ( ) {
9    S x = {1}, y = {2}, z;
10   z = ::max (x, y); // zde je chyba
11   cout << z.a << endl;
12   return 0;
13 }
```

Jak opravit chybu?

Přidat přetížený operátor > pro datový typ S.

Příklad – zobrazení pole

```
1  // Zobrazit pole, n - počet prvků
2  // rowLen - formátování (počet prvků na řádku)
3  template <class T>
4  void printArray (T *arr, int n, int rowLen = 10) {
5    int i;
6    for (i = 0; i < n; i++) {
7      if (i % rowLen != 0)
8        cout << ' ';
9      cout << arr[i];
10     if (i % rowLen == rowLen - 1)
11       cout << endl;
12   }
13   if (i % rowLen != 0 )
14     cout << endl;
15 }
```

Příklad – řazení pole

```
1  // Seřadit pole, n - počet prvků
2  template <class T>
3  void sortArray (T *arr, int n) {
4    for (int i = 0; i < n - 1; i++) {
5      int min = i;
6      for (int j = i + 1; j < n; j++)
7        if (arr[j] < arr[min])
8          min = j;
9      T tmp = arr[i];
10     arr[i] = arr[min];
11     arr[min] = tmp;
12   }
13 }
```

II. Šablony funkcí a tříd

Generické funkce

Generické třídy

Generické třídy

- Generická třída je parametrizovaná implementace třídy.
- Kompilátor odvozuje instanci generické třídy nahrazením generických parametrů skutečnými hodnotami.
- Generické třídy jsou obvykle parametrizovány typovým(i) parametrem(y).

Příklad – čítač 1/2

```
1  template <class T>
2  class Counter {
3    T value, init;
4  public:
5    Counter (T in_init): init(in_init) {reset ();}
6    void increment () {value++;}
7    void reset () {value = init;}
8    T get () {return value;}
9  };
10 // --
11 Counter<int> a(0);
12 Counter<char> b('A');
13 std::cout << a.get() << std::endl;
14 b.increment();
```

Příklad – čítač 2/2

- Pokud jsou metody implementovány vně deklarace generické třídy, musí každá metoda začínat deklarací šablony template.

```
1  template <class T>
2  class Counter {
3    T value, init;
4  public:
5    Counter (T in_init);
6    void increment () {value++;}
7  // --
8  };
10 template <class T>
11 Counter<T>::Counter (T in_init) {value = in_init; reset();}
13 template <class T>
14 void Counter<T>::increment () {value++;}
```

Příklad – dynamické pole 1/6

- Pole bude obsahovat prvky jakéhokoli typu (generický parametr).
- Implementace bude korektně implementovat kopírující konstruktor a operátor =.

```
1  template <class T>
2  class Array {
3    T * a_data;
4    int a_size;
5  public:
6    Array (int size = 10);
7    ~Array ();
8    Array (const Array<T> & src);
9    int size() const {return a_size;}
10   Array<T> & operator = (const Array<T> & src);
11   T & operator [] (int idx);
12   const T & operator [] (int idx) const;
13 };
```

Příklad – dynamické pole


```
1  template <class T>
2  Array<T>::Array (int size): a_size(size) {
3    a_data = new T[a_size];
4  }
6  template <class T>
7  Array<T>::~Array () {
8    delete [] a_data;
9  }
11 template <class T>
12 Array<T>::Array (const Array<T> & src ) {
13   a_size = src.a_size;
14   a_data = new T[a_size];
15   for (int i = 0; i < a_size; i++) a_data[i] = src.a_data[i];
16 }
```

Příklad – dynamické pole


```
1  template <class T>
2  Array<T> & Array<T>::operator = (const Array<T> & src) {
3    if (this != &src) {
4      delete [] a_data;
5      a_size = src.a_size;
6      a_data = new T[a_size];
7      for (int i = 0; i < a_size; i++) a_data[i] = src.a_data[i];
8    }
9    return *this;
10 }
12 template <class T>
13 T & Array<T>::operator [] (int idx) {
14   if (idx < 0 || idx >= a_size) throw "Spatny index";
15   return a_data[idx];
16 }
```

Příklad – dynamické pole


```
1  template <class T>
2  const T & Array<T>::operator [] (int idx) const {
3    if (idx < 0 || idx >= a_size)
4      throw "Spatny index";
5    return a_data[idx];
6  }
8  template <class T>
9  ostream & operator<< (ostream & o, const Array<T> & x) {
10   for (int i = 0; i < x.size(); i++)
11     o << x[i] << ' ';
12   return o;
13 }
```

Příklad – dynamické pole


```
1  Array<int> a (5);
2  for (int i = 0; i < a.size (); i++) a[i] = i;
3  cout << "array a: " << a << endl; // [0 1 2 3 4]
5  Array<int> b = a;
6  b[1] = 10;
7  cout << "array a: " << a << endl; // [0 1 2 3 4]
8  cout << "array b: " << b << endl; // [0 10 2 3 4]
10 Array<double> c (5);
11 c[1] = 20;
12 cout << "array c: " << c << endl; // [?  20 ?  ?  ?]
14 Array<double> d = c;
15 d[2] = 30;
16 cout << "array d: " << d << endl; // [?  20 30 ?  ?]
```

Příklad – dynamické pole

- Mohou mít prvky pole generický datový typ?

```
int main() {
  Array<Array<int>> a (5);
  for (int i = 0; i < a.size(); i ++ )
    for (int j = 0; j < a[i].size(); j++)
      a[i][j] = i + j;
  cout << "array a: " << a << endl;
  Array<Array<int>> b = a;
  b[1][2] = -10;
  cout << "array a: " << a << endl;
  cout << "array b: " << b << endl;
  return 0;
}
```

- Ano, pro inicializaci polí řádků je volán implicitní konstruktor, výsledkem je matice 5x10

Část III

Paralelní programování

III. Paralelní programování

Vlákna – Threads

Koncepty vícevláknových aplikací

Synchronizace vláken

Vlákna v C++ (STL)

Vlákna

- V rámci předmětu jsme se už seznámili s procesy
- Vlákno je soubor instrukcí spouštěných nezávisle na hlavním procesu
  - Malý program zaměřený na specifickou část většího úkolu
- Vlákno běží uvnitř procesu
  - Sdílí tu samou paměť a paměťový prostor
- Vlákno má vlastní prostředí
  - Vlastní oblast pro proměnné
  - Vlastní identifikátor a prostor pro synchronizační primitiva
  - Vlastní Program counter (PC)/Instruction pointer (IP)
  - Vlastní oblast paměti pro lokální proměnné – Stack
- Vlákno může běžet v uživatelském prostoru nebo OS
  - Vlákna v uživatelském prostoru nepotřebují podporu OS, ale nemohou běžet současně
  - Vlákna běžící v OS mohou být plánována v rámci soutěže se všemi ostatními vlákny v systému, mohou běžet současně, ale vytváření vláken stojí čas

Kdy využívat vlákna

- Úloha obsahuje několik nezávislých částí
- Část úlohy může být blokována po určitý čas
- Úloha obsahuje výpočetně náročnou část (kde je potřeba stále interagovat s uživatelem)
- Aplikace musí rychle reagovat na asynchronní události
- Úloha obsahuje části s nižší a vyšší prioritou než zbytek aplikace
- Výpočetně náročná část může být urychlena paralelním zpracováním dat při využití více výpočetních jader

Typické aplikace využívající vláken

- Servery: obsluhují více uživatelů současně, využívají sdílené zdroje (databáze) a provádějí mnoho I/O operací
- Výpočetní aplikace: úspora času při využití více procesorů
- Aplikace reálného času: využití specifik plánovače – vícevláknová aplikace bude efektivnější než komplexní asynchronní program

Použití vláken

Efektivní využití výpočetních zdrojů

- Čekání na periferie → vlákno je blokované a řízení je předáno jiným vláknům
- Na systémech s více procesory/jádry je možné použít paralelní algoritmy

Řešení asynchronních situací

- Např. blokované vstupně/výstupní operace
- Procesor může řešit něco jiného (např. jedno vlákno řeší I/O operace z kamery a další vypočítává změny v obraze)

Vstupně výstupní operace

- Komunikace s periferiemi obsahuje hodně čekání – uživatelský vstup

Interakce s GUI

- Požadujeme okamžitou reakci na vstup uživatele – jinak se zdá, že systém zamrzl
- Uživatel v každém okamžiku generuje spoustu událostí, které ovlivňují aplikaci
- Výpočetně náročné operace by neměly ovlivnit interaktivnost aplikace – scrolování fotek

Vlákno a proces

Proces

- Výpočetní tok
- Vlastní paměťový prostor
- Součást OS
- IPC pomocí volání OS služeb
- CPU přidělovaný OS
- Vytvořit proces trvá

Vlákno (v procesu)

- Výpočetní tok
- Běží ve stejném paměťovém prostoru jako proces
- Uživatelská součást nebo součást OS
- Synchronizace pomocí exkluzivního přístupu k proměnným
- CPU je přidělován v rámci času dedikovaného mateřskému procesu
- Je rychlejší vytvořit vlákno než proces

III. Paralelní programování

Vlákna – Threads

Koncepty vícevláknových aplikací

Synchronizace vláken

Vlákna v C++ (STL)

Boss/Worker

- Hlavní vlákno přijímá požadavky z vnějšku – zpracovává požadavky v cyklu
  - Přijme požadavek
  - Vytvoří/vybere pracovní vlákno, kterému přiřadí zpracování požadavku
  - Čeká na další požadavek
- Výsledek operace/požadavku je řízen
  - Buď přímo pracovním vláknem
  - Nebo hlavním vláknem, přičemž se použije nějaký synchronizační mechanismus

```
switch(getRequest()) {
  case taskX :
    create_thread(taskX);
    break;
  case taskY:
    create_thread(taskY);
    break;
}
```

```
taskX() {
  // solve the task
}
taskY() {
  // solve the task
}
```

Peer

- Neobsahuje řídicí vlákno
- První vlákno (procesu) vytvoří další vlákna a následně:
  - Se přepne do módu pracovního vlákna
  - Uspí samo sebe a čeká na ostatní vlákna
- Každé vlákno je odpovědné za svůj vstup a výstup

```
// 1st thread
create_thread(task1);
create_thread(task2);
.
.
start all threads;
wait to all threads;
```

```
task1() {
  wait to be exectued
  solve the task
}
task2() {
  wait to be exectued
  solve the task
}
```

Pipeline

- Dlouhý vstupní tok dat (stream) obsahují sekvence pro zpracování
  - Každý vstup musí být zpracován všemi částmi z pipeline
- V jednom časovém okamžiku jsou rozdílná vstupní data zpracována nezávislou částí

```
create_thread(stage1);
...
create_thread(stageN);
wait // for all pipeline
stage1() {
  while(input) {
    get next program input;
    process input;
    pass result to next the
    stage;
  }
}
```

```
stageN() {
  while(input) {
    get next input from
    thread;
    process input;
    pass result to output;
  }
}
```

Producent/Consumer

- Předávání dat mezi vlákny je realizováno pomocí bufferu (nebo ukazatelů)
- **Producer** – vlákno předávající data dalšímu vláknu
- **Consumer** – vlákno přijímající data z jiného vlákna
- Přístup do bufferu musí být synchronizovaný (exkluzivní přístup)

III. Paralelní programování

Vlákna – Threads

Koncepty vícevláknových aplikací

Synchronizace vláken

Vlákna v C++ (STL)

Synchronizační mechanismus

- Vlákna používají obdobné mechanismy jako procesy
  - Protože vlákna sdílí paměť procesu, tak hlavní komunikační prostředek je paměť a globální proměnné
  - Kritický je přístup do stejné části paměti – je potřeba zajistit exkluzivní přístup do kritické sekce
- Základní synchronizační primitiva
  - Mutex/zámek – exkluzivní přístup do kritické sekce
  - Podmíněná proměnná (Conditional Variable) – sdílená proměnná umožňující synchronizaci vláken – spící vlákno může být probuzeno signálem z jiného vlákna

Základní synchronizační primitiva

- **Mutex** – sdílená proměnná, která je přístupná z ostatních vláken
- Poskytuje operace
  - Uzamknutí (lock) – mutex získalo vlákno, které mutex uzamklo
    - jestliže jiné vlákno požádá o stejný mutex (uzamčený), tak toto vlákno je systémem uspáno/blokováno a čeká na uvolnění mutexu
  - Odemčení (unlock) – lze provést na mutexu, který byl dříve zamčen
    - při odemčení se kontroluje, jestli na daný mutex nečeká jiné vlákno
    - pokud ano, tak se jedno z čekajících vláken vybere a dovolí se mu pokračovat v činnosti
- **Podmíněná proměnná** – umožňuje signalizovat z jednoho vlákna do druhého
- Poskytuje operace
  - Wait – proměnná je modifikovaná
  - Timed – čeká na signál z jiného vlákna
  - Signalizuje dalšímu vláknu změnu
  - Signalizuje všem čekajícím vláknům
    - Všechna vlákna jsou probuzena, ale protože přístup k podmíněné proměnné je chráněn mutexem, tak může mutex získat jen jedno vlákno

Příklad použití mutexu

- Zajištění exkluzivního přístupu k podmíněné proměnné z různých vláken

```
Mutex mtx; // shared variable for both threads
CondVariable cond; // shared condition variable
// Thread 1
Lock(mtx);
// Before code, wait for Thread 2
CondWait(cond, mtx); // wait for cond
... // Critical section
UnLock(mtx);
// Thread 2
Lock(mtx);
... // Critical section
CondSignal(cond, mtx); // signal on cond
UnLock(mtx);
```

Požadavky na funkce

- V případě paralelního zpracování, funkce jsou:
  - **Reentrant** – v jednom okamžiku je možné funkci spustit několikrát
  - **Thread-safe** – funkce může být volána různými vlákny ve stejný okamžik
- Jak to zajistit:
  - Reentrant funkce nepoužívá statická data a globální data
  - Thread-safe funkce striktně přistupuje ke globálním datům pomocí synchronizačních mechanismů

Problémy

- **Deadlock** – vlákno čeká na mutex, který je zamknutý jiným vláknem, přičemž jiné vlákno čeká na mutex zamčený prvním vláknem
- **Race condition** – přístup více vláken ke sdílené proměnné, kde minimálně jedno vlákno nepoužívá synchronizační prostředky

III. Paralelní programování

std::thread

- Třída pro práci s vlákny (`<thread>`)
  - Spuštění funkce ve vláknu – `std::thread(function, args...)`
  - Čekání na vlákno – `std::thread::join()`
  - Odpojení vlákna – `std::thread::detach()`

```
1  std::thread t(&thread_function); // startuje vlakno t
2  std::cout << "main thread\n";
3  t.join();  // hlavni vlakno ceka, az se t dokonci
4  /*
5    vlakno t je mozne pustit jako samostatny proces (demon)
6    t.detach() nelze kombinovat s join()
7    vlakno nic nevypise, protoze hlavni program mezitim skonci
8  */
9  return 0;
```

lec08/09-thread-join.cpp

std::mutex

- Třída pro práci s mutexy (`<mutex>`)
  - Zamknutí mutexu – `std::mutex::lock()`
  - Pokus o odemknutí – `std::mutex::try_unlock()`
  - Odemknutí mutexu – `std::mutex::unlock()`

```
1  std::mutex mu;
3  void shared_cout (std::string msg, int id) {
4    mu.lock();
5    std::cout << msg << ":" << id << std::endl;
6    mu.unlock();
7  }
9  std::thread t(&thread;_function);
10 for (int i = 100; i > 0; i--)
11   shared_cout("main thread", i);
12 t.join();
```

lec08/10-thread-mutex.cpp

Mutex a RAII

- Obecné ovládání mutexu (RAII princip)
  - `std::lock_guard (Lockable &m);`
  - `std:: lock_guard();`
  - `std::unique_lock();`

```
1  std::mutex mu;
3  void addToList(int max, int interval) {
4    std::lock_guard guard (mu);
5    for (int i = 0; i < max; i++) {
6      if((i % interval) == 0) myList.push_back(i);
7    }
8  }
10 std::thread t1(addToList, 100, 1);
11 std::thread t2(addToList, 100, 10);
```

lec08/11-thread-mutex-guard.cpp