Funktory

STL – Funktory

- Příklad - funkce for_each

```
template<class InputIterator, class Function>
Function for_each( InputIterator first, InputIterator last, Function f)
{
    for (; first != last; ++first)
        f( * first);
    return f;
}
```

  - f je cokoliv, co lze zavolat syntaxí f(x)
    - globální funkce (ukazatel na funkci), nebo
    - **funktor**, tj. třída obsahující operator()
  - Funkce f (případně metoda f.operator()) je zavolána na každý prvek v zadaném intervalu
    - prvek se předává jako * iterator, což může být odkazem
    - funkce f tedy může modifikovat prvky seznamu

STL – Algoritmy

- Jednoduché použití funkce for_each

```
void my_function( double & x)
{
    x += 1;
}
void increment( std::list< double> & c)
{
    std::for_each( c.begin(), c.end(), my_function);
}
```

- [C++11] Lambda
  - Výraz generující funktor (s unikátním anonymním typem)
    - Efekt je v tomto případě tentýž jako u pojmenované funkce
    - Překladačem ale realizováno jinak (většinou rychleji)

```
void increment( std::list< double> & c)
{
    for_each( c.begin(), c.end(), []( double & x){ x += 1;});
}
```

- Předávání parametrů vyžaduje **funktor**

```
class my_functor {
public:
    double v;
    void operator()( double & x) const { x += v; }
    my_functor( double p) : v( p) {}
};
void add( std::list< double> & c, double value)
{
    std::for_each( c.begin(), c.end(), my_functor( value));
}
```

- Lambda s přístupem k lokální proměnné
  - V tomto případě technicky identické s ručně vytvořeným funktorem

```
void add( std::list< double> & c, double value)
{
    std::for_each( c.begin(), c.end(), [value]( double & x){ x += value;});
}
```

- Funktory modifikující svůj obsah

```
class my_functor {
public:
    double s;
    void operator()( const double & x) { s += x; }
    my_functor() : s( 0.0) {}
};
double sum( const std::list< double> & c)
{
    my_functor f = std::for_each( c.begin(), c.end(), my_functor());
    return f.s;
}
```

- Lambda s referencí na lokální proměnnou
  - Jiný princip než modifikující se funktor

```
double sum( const std::list< double> & c)
{   double s = 0.0;
    for_each( c.begin(), c.end(), [& s]( const double & x){ s += x;});
    return s;
}
```

Lambda

Lambda výrazy

- Lambda výraz [C++11]
- [capture](params) mutable -> rettype { body }
  - Deklaruje třídu ve tvaru

```
class ftor {
public:
  ftor( TList ... plist) : vlist( plist) ... { }
  rettype operator()( params ) const { body }
private:
  TList ... vlist;
};
```

    - vlist je určen proměnnými použitými v body
    - TList je určen jejich typy a upraven podle capture
    - operator() je const pokud není uvedeno mutable
  - Lambda výraz je nahrazen vytvořením objektu

```
ftor( vlist ...)
```

Lambda výrazy – návratový typ a typ funkce

- Návratový typ operátoru
  - Explicitně definovaný návratový typ

```
[]() -> int { /*…*/ }
```

  - Automaticky odvozený
    - Všechny příkazy return uvnitř lambdy musejí vracet tentýž typ

```
[]() { /*...*/ return /*...*/; }
```

  - Jinak void

```
[]() { /*...*/ }
```

- Tato syntaxe návratových hodnot je možná i u pojmenovaných funkcí [C++14]
  - Explicitně definovaný návratový typ s alternativní syntaxí
    - Užitečné, pokud typ závisí na argumentech funkce (pomocí decltype apod.)
    - Nebo pokud je návratový typ deklarován uvnitř třídy takto definované metody

```
auto f() -> int;    // int f();
```

  - Automaticky odvozený
    - Pouze u definice funkce, nikoliv u deklarace
    - Použitelné obvykle pouze pro krátké inline funkce

```
auto f() { /*...*/ return /*...*/; }
```

    - Také ve variantě vracení odkazem (forwarding reference)

```
auto && g() { /*...*/ return /*...*/; }
```

Lambda expressions – generic parameter types

- Generická lambda [C++14]
  - Parametry lambdy mohou být deklarovány pomocí **auto**

```
[](auto x, auto && y) { /*...*/ }
```

  - Výsledný operátor je pak šablona funkce

```
class ftor { public:
  template< typename T, typename U>
  void operator()(T x, U && y) const
  { /*...*/ }
};
```

  - Explicitně generické lambdy [C++20]
    - Konečně je v C++ konstrukce zahrnující všechny 4 druhy závorek!

```
[]< typename T, typename U>(T x, U && y){ /*...*/ }
```

- Pojmenované funkce také mohou mít parametry s **auto** [C++20]

```
void f(auto x, auto && y)
{ /*...*/ }
```

  - Zkratka za

```
template< typename T, typename U>
void f(T x, U && y)
{ /*...*/ }
```

Lambda výrazy – capture

- Capture
  - Způsob zpřístupnění vnějších entit
    - lokální proměnné
    - this
  - Určuje typy datových položek a konstruktoru funktoru
  - Explicitní capture
    - Programátor vyjmenuje všechny vnější entity v *capture*

```
[a,&b,c,this](){ return a+c+b[c]+m; }
```

    - Entity označené & předány odkazem, ostatní hodnotou
    - **this** umožňuje přístup k položkám objektu (stejně jako v metodě obsahující lambda výraz)
    - Capture s výrazem deklaruje novou proměnnou viditelnou v těle lambda výrazu

```
[a=std::move(a),&y=b[c],z=m](){ return a+y+z; }
```

    - [a,&b] je zkratka za [a=a,&b=b]
  - Implicitní capture (nedoporučuje se)
    - Překladač sám určí vnější entity, *capture* určuje způsob předání

```
[=]
[=,&b,&y=b[c]]
```

    - předání hodnotou, vyjmenované výjimky odkazem

```
[&]
[&,a,c]
```

    - předání odkazem, vyjmenované výjimky hodnotou

Lambda výrazy

- Každá lambda vytváří nový (anonymní) typ
  - Funkce akceptující lambdy musejí být šablony
    - Lambdy obvykle předávány hodnotou, někdy univerzální referencí (F &&)

```
template< typename F>
void g( F f)
{
  f(/*...*/);
}
```

  - Funkčnost lambdy je předána v typu (F), ne v hodnotě (f)
    - Vyřešeno při kompilaci – rychlost volání lambdy je stejná jako u běžné funkce
  - Za běhu se předávají pouze hodnoty okolních proměnných sbalené v objektu f

- Lambdu je možné uložit v lokální proměnné
  - Slouží jako vnořená funkce
    - Se schopností přistupovat k okolním lokálním proměnným

```
void f(std::ostream & out)
{
    auto print = [& out](auto && x){ out << x << std::endl; };

   for (auto && a : k) { print(a); }
   // same as
   std::for_each( k.begin(), k.end(), print);
}
```

Lambda výrazy

- Vracení lambdy z funkce je obtížné a nebezpečné

```
auto h(int p) { return [p](int & x){ x += p}; }
```

- Použití lambdy mimo funkci je obvykle příliš komplikované
  - např. pro definici porovnání pro asociativní kontejner

```
using M = std::map<K,T,decltype([](const K&a, const K&b){ return /*...*/; })>;
```

- Různé lambdy není možné přímo míchat dohromady
  - Ikdyž mají lambdy stejné argumenty, jejich typy se vždy liší

```
cond ? [](int & x){ ++x; } : [](int & x){ --x; } // ERROR

k[0] = [](int & x){ ++x; }; k[1] = [](int & x){ --x; };// ERROR
```

Lambda výrazy

- std::function
  - std::function může obsahovat libovolnou lambdu s danou signaturou
    - typy parametrů a návratové hodnoty se specifikují dohromady jako typ funkce

```
using fnc_t = std::function<void(int&)>;
std::vector<fnc_t> k(2);
```

    - každou lambdu lze implicitně konvertovat na std::function odpovídající signatury

```
k[0] = [](int & x){ ++x; };
k[1] = [](int & x){ --x; };
```

  - std::function nahrazuje kompilační mechanismus běhovým
    - std::function je prostorově i časově dražší než lambda
      - Zdrojový functor/lambda je okopírován/přesunut do dynamicky alokovaného objektu
      - std::function je (sdíleným) vlastníkem tohoto objektu
    - std::function se používá, jen když nelze použít přímo funktor/lambdu
      - Kontejnery obsahující různé funktory/lambdy
      - Nelokální proměnné a datové položky obsahující lambdy
    - std::function může prodloužit život lambdy
      - Obvykle nelze použít pro lambdy s referenčními capture
  - std::function se sama chová jako funktor
    - Může být použito jako argument algoritmů apod.
      - Bude pomalejší než přímo použitá lambda/funktor