Jazyk C++: imperativny, staticky typovany a objektovo orientovany programovaci jazyk

* Umoznuje genericke programovanie (sablony)
* Vysoka rychlost kodu

Implementacia OOP v C++

* Pristup je zalozeny na triedach (class)
* Trieda (class) je datovy typ, je rozsirenim datoveho typu struct z jazyka C
* Okrem atributov (dat) moze obsahovat metody (funkcie)
* Objekt je konkretna instancia triedy
* Metoda realizuje princip posielania sprav

Pristupove prava:

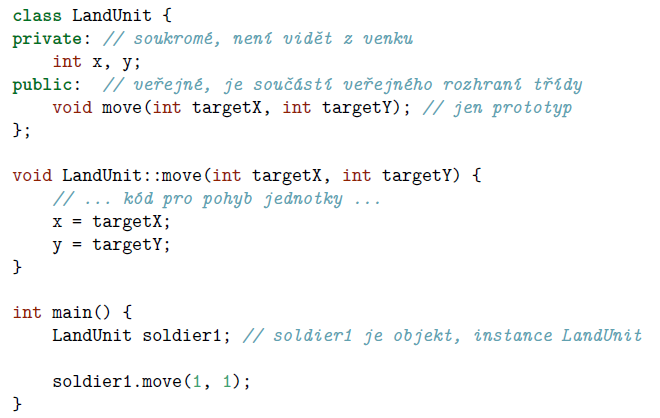
* public: takto oznacene casti su viditelne od vsadial
* protected: takto oznacene casti su viditelne len pre potomkov triedy
* private: takto oznacene casti su viditelne len pre triedu samotnu

Implicitne pristupove prava vo vnutri:

* class: private
* struct: public

Metody:

* v skutocnosti maju jeden skryty implicitny parameter **this** = ukazatel na konkretny objekt

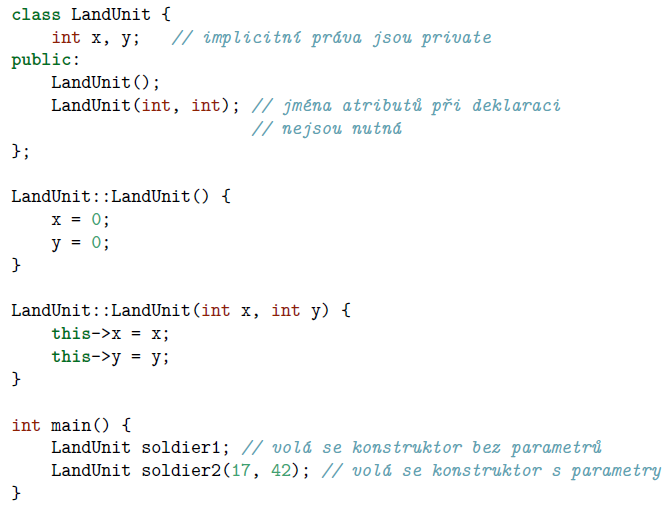


Typ metody move je ako keby void LandUnit::move(LandUnit\* this, int, int);

Volanie metody je ako keby soldier1.move(&soldier1, 1, 1);

Konstruktor:

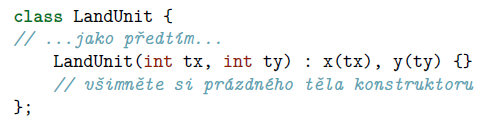
* vola sa automaticky pri vytvoreni objektu
* pouziva sa hlavne pre inicializaciu atributov objektu ale moze mat aj ine funkcie (otvorenie  
  suboru, spojenie so serverom … )
* moze mat argumenty a moze byt pretazeny (viac konstruktorov s roznymi argumentami)
* nema navratovu hodnotu (ani void)
* ma rovnake meno ako trieda samotna
* ked nedefinujeme konstruktor, vytvori sa defaultny konstruktor



Defaultny konstruktor:

* vola defaultne bezparametricke konstruktory atributov, pre primitivne typy to znamena, ze zostanu neinicializovane

Inicializacna sekcia konstruktoru:

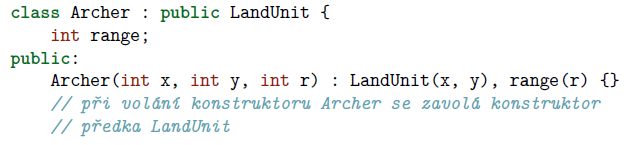
* inicializacia atributov
* lepsi sposob ako priradenim, nemusi sa vytvarat lokalna kopia
* 

Destruktor:

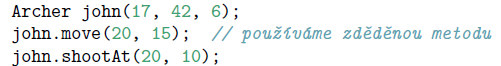
* vola sa pri zaniku objektu
* vola sa automaticky na staticky alokovane lokalne premenne pri opustani bloku
* vola sa automaticky pri zavolani delete pre dynamicky alokovane premenne
* nema ziadne parametry
* neda sa pretazovat
* nevracia ziadnu hodnotu (ani void)
* ma rovnaky nazov ako trieda so znakom ~ pred tymto nazvom
* ak od triedy hodlame dedit, jej destruktor by mal byt **public a virtualny** alebo **protected a nevirtualny**
* napr: ~NazovTriedy() {}

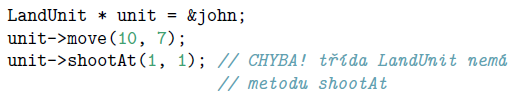
Dedicnost:

* trieda moze dedit od jednej alebo viacerych tried s roznymi pristupovymi pravami
* obmedzuje duplicity v kode



* ukazatelovi na predka mozme predat potomka





Pristupove prava pri dedeni:

* **public:** zdedene polozky dedia prava od predka
* **protected:** verejne zdedene polozky sa menia na protected
* **private:** vsetky zdedene polozky sa menia na private
* **virtual:** da sa kombinovat s niektorym z predchodzich prav

Klucove slovo **virtual:**

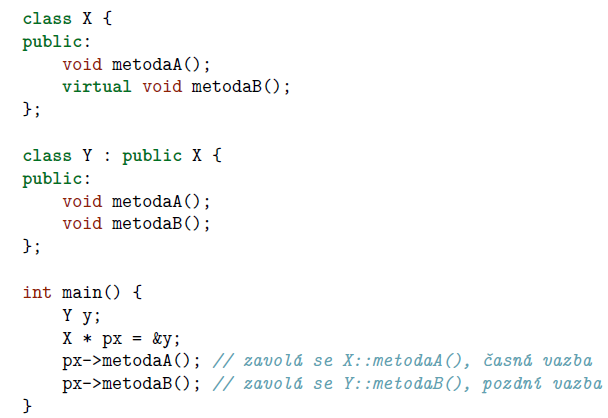
* urcuje pozdni vazbu (neskoru vazbu)

Vcasna vazba:

* prekladac v dobe prekladu presne vie kedy sa ktora metoda zavola

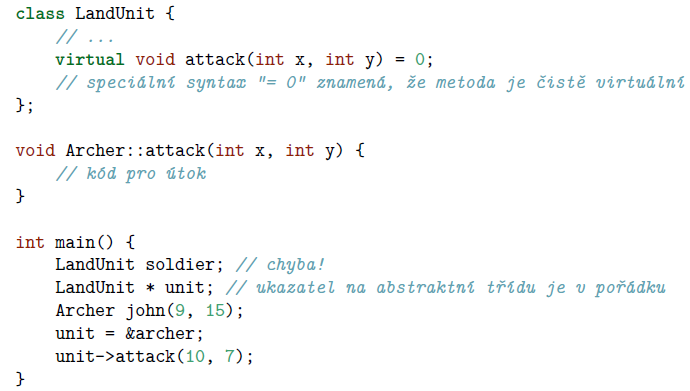
Pozdni vazba:

* o tom, ktora metoda sa zavola sa rozhoduje az za behu programu
* je pomalsia ako vcasna vazba
* urcuje sa klucovym slovom virtual pred metodou



Abstraktna trieda:

* trieda, ktora neimplementuje nejaku metodu (tj. ma aspon jednu cisto virtualnu metodu)
* nieje mozne vytvarat instancie abstraktnej triedy

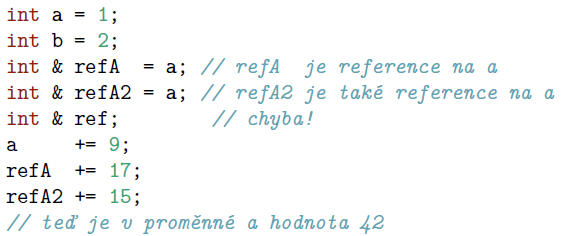


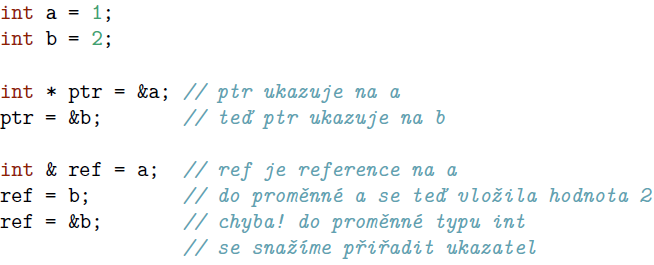
NULL vs nullptr v C++11:

* NULL: definovane ako cislo 0 (prevzate z C)
* nullptr: specialny objekt typu ukazatel

Datovy typ reference:

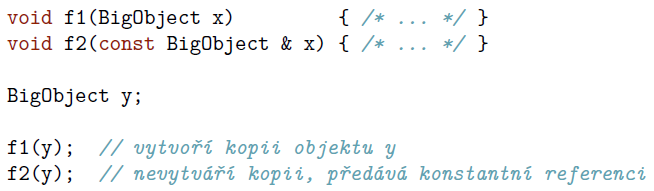
* alternativne meno pre object/premennu (alias)
* znaci sa symbolom **&**
* nemoze byt neinicializovana (ukazatel moze byt neinicializovany)
* vzdy ukazuje na skutocny objekt/premennu (ukazatel moze byt nullptr)
* neda sa “presmerovat” (ukazatel mozeme menit => aby ukazoval na nieco ine ako doteraz)





Predavanie parametrov referenciou:

* vyzera rovnako ako volanie hodnotou premennej
* zmena argumentu vo vnutri funkcie sa prejavi aj vonku
* nevytvara sa lokalna kopia



l-hodnota (l-value):

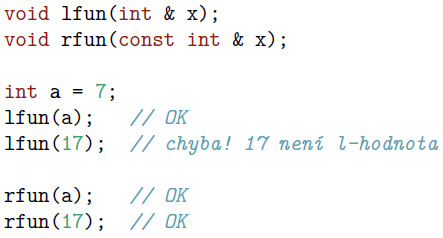
* nieco, co moze stat na lavej strane priradenia, tj. ma to adresu, napr: premenna

r-hodnota (r-value):

* nieco, co moze stat na pravej strane priradenia, napr: ciselna hodnota

Konstantna vs. Nekonstantna referencia ako formalny parameter:

* funkcii, ktora berie ako parameter konstantnu referenciu mozme dat aj l aj r hodnoty
* funkcii, ktora berie ako parameter nekonstantnu referenciu mozme dat len l-hodnoty



Referencia ako atribut triedy:

* neda sa inicializovat az v tele konstruktoru
* musi sa inicializovat inicializacnou sekciou konstruktoru

Pretazovanie (Overloading):

* moznost vytvorit niekolko funkcii/metod s rovnakym nazvom ale roznou implementaciou
* musia sa lisit typom parametrov alebo poctom parametrov
* nestaci, ak sa lisia len navratovym typom
* pri metodach staci, ked sa lisia v klucovom slove const
* o tom, ktora funkcia/metoda sa zavola sa rozhoduje pocas prekladu na zaklade typov skutocnych parametrov
* v skutocnosti to funguje tak, ze prekladac meni mena funkcii, pridava k nim typy parametrov, tzv. name mangling
* napr: gcc kompilator

meno funkcie **void fun (int, char)** sa interne zmeni na **\_Z3funic**

meno funkcie **void fun(int &)** sa interne zmeni na **\_Z3funRi**

Predefinovanie (Overriding):

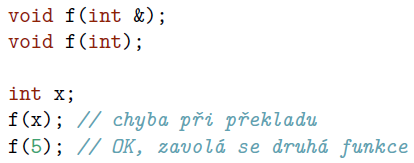
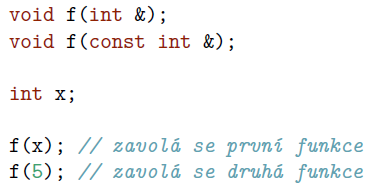
* nahradenie definicie virtualnej metody u predka definiciou u potomka

Skryvanie (Hiding):

* pokial je v potomkovi deklarovana metoda s rovnakym menom a bud nieje virtualna alebo sa lisi parametrami alebo klucovym slovom const

Pretazovanie a referencia:

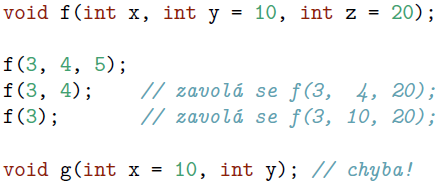
* pokial mame metodu pretazenu na konstantu a nekonstantnu referenciu, prednost ma nekonstantna



* mat funkciu pretazenu pre referenciu a pre zakladny typ nieje vhodne, vznika
* nejednoznacnost

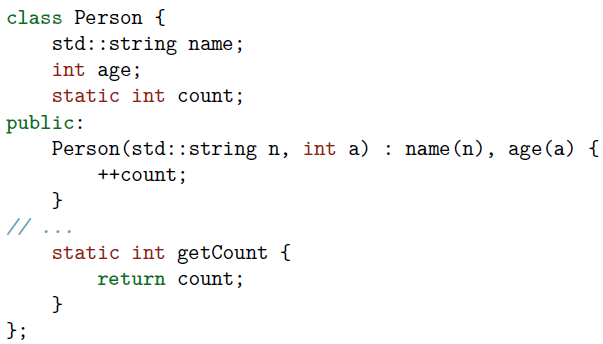
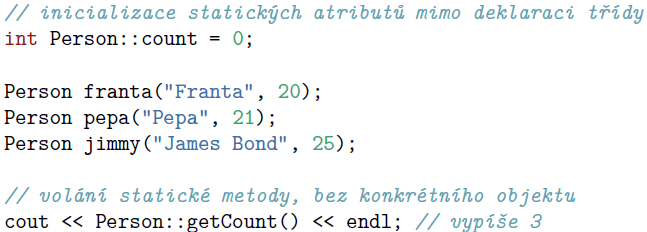
Implicitne parametry:

* funkcie a metody (aj konstruktor) moze mat parametry s implicitnymi hodnotami
* musia byt umiestnene najviac vpravo v zozname formalnych parametrov
* pri oddeleni deklaracie a definicie sa pisu len do deklaracie



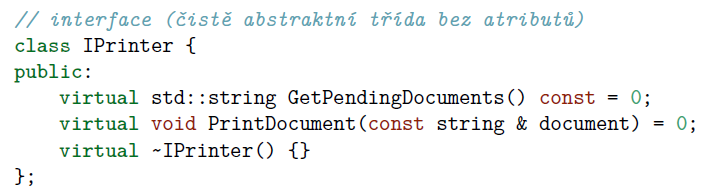
Staticke atributy a metody:

* uvadzaju sa klucovym slovom **static**
* patria triede samotnej, nepatria ziadnemu objektu
* objekty k nim mozu pristupovat

****

Rozhranie (Interface):

* cisto abstraktna trieda bez atributov
* chceme zachytit vsetko co musi trieda vediet, aby sa mohla vydavat za prislusnika danej skupiny
* napr: interface tlaciarne, viz. obrazok



* mozme pisat kod vyuzivajuci rozhranie tlaciarne bez toho, aby sme vedeli s akou konkretnou tlaciarnou pracujeme

override:

* umoznuje explicitne povedat, ze sa chystame danu metodu predefinovat
* prekladac dozrie aby sedel typ metody a aby bola metoda virtualna

final:

* pri metode: dana virtualna metoda uz nesmie byt v potomkoch predefinovana
* pri triede: od danej triedy sa nesmie dedit

Cisto abstraktna trieda:

* vsetky metody su cisto virtualne, s pripadnou vynimkou destruktoru

Kompozicia:

* alternativa k viacnasobnej dedicnosti
* vztah HAS-A (Laptop has a CPU) oproti dedicnosti ktora je vztah IS-A (Student is a Person)
* trieda moze mat ako atribut dalsiu triedu (hodnotou, ukazatelom, referenciou)
* trieda tym obsahuje vsetky vlastnosti tych tried, ktore ma sama ako atributy ale nieje ich potomkom, nemoze ich zastupit

Staticka alokacia na zasobniku:

* deklaracia vo vnutri bloku (lokalne premenne)
* automaticke uvolnenie pamati na konci bloku
* je to rychle (zasobnik je v cache)
* zasobnik ma obmedzenu velkost
* vhodne pre kratkodobe a mensie objekty

Dynamicka alokacia na halde:

* explicitna ziadost o kus pamati
* explicitny pokyn k uvolneniu pamati
* pre dlhodobe vacsie objekty

Dynamicka alokacia v C++:

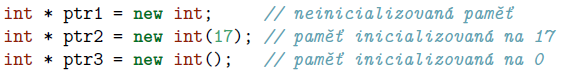
* operatory: **new**, **delete**, **delete[]**
* dva zakladne typy alokacie:

1. alokacia jedneho objektu / jednej hodnoty primitivneho datoveho typu

2. alokacia pola objektov / pola hodnot primitivneho datoveho typu (potrebne miesto je automaticky spocitane)

- objekt nieje mozne nechat neinicializovany

* pri alokacii poli sa vzdy pouzije bezparametricky konstruktor, pre hodnoty primitivnych datovych typov to znamena ze zostanu neinicializovane
* operator **delete** sa pouziva pre dealokaciu samostatneho objektu
* operator **delete[]** sa pouziva pre dealokaciu pola objektov

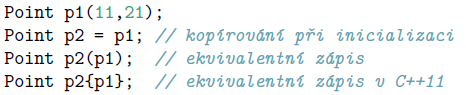




Kopirovaci konstruktor:

* vola sa automaticky pri inicializacii z ineho objektu
* ked ziadny nedefinujeme, vytvori sa implicitny





Plytka kopia:

* kopiruje len hodnoty atributov
* ak mame ako atribut ukazatel, skopiruje sa len jeho hodnota, nie to, na co ukazuje
* casto nastava problem, ze chceme dealokovat uz dealokovanu pamat => CHYBA

Hlboka kopia:

* kopiruje samotne data, na ktore ukazatelovy atribut ukazuje

Priradzovaci **operator=**:

* pouziva sa pri kopirovani mimo inicializacie
* problem so sebapriradenim
* copy and swap idiom



Rule of three:

* ak trieda definuje apson jednu z nasledujucich veci, chceme definovat vsetky tri

1. Destruktor

2. Kopirovaci konstruktor

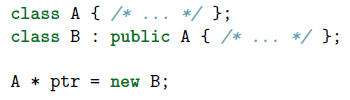
3. Priradzovaci operator=

static\_cast:

* pretypovanie, ktore je mozne previest staticky (pocas kompilacie)
* zmena celociselnej hodnoty na enum
* zmena ukazatela na void \* a naopak
* nemusi byt vzdy bezpecne, pretypovanie hodnoty 3 na enum, ktory ma len dve hodnoty

dynamic\_cast:

* umoznuje bezpecne pretypovat ukazatel predka na ukazatel potomka
* je to mozne len ak je objektova hierarchia virtualna (tj. trieda predka ma niektoru metodu virtualnu)
* rozhoduje sa az za behu programu



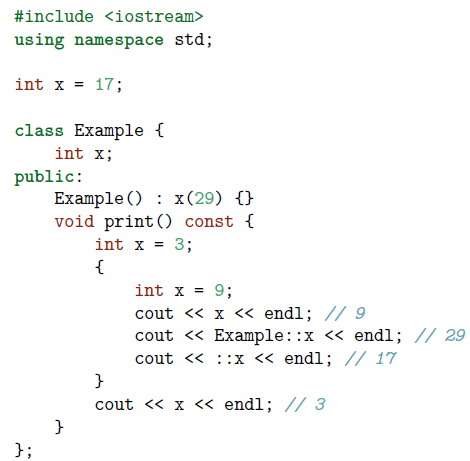


Implicitne menne priestory:

* oblasti platnosti identifikatorov
* vznikaju automaticky
* lokalne menne priestory: funkcie/metody, cykly, bloky
* globalny menny priestor
* v jednom mennom priestore nemozu byt dve entity s rovnakym menom

Pristup k mennym priestorom:

* pomocou operatoru **::**
* globalny menny priestor: **::x**
* menny priestor triedy: **ClassName::x**
* k lokalnym mennym priestorom nemame pristup, niesu pomenovane



Explicitne menne priestory:

* je mozne vytvorit vlastne pomenovane menne priestory
* syntax: **namespace nazov { … }**
* je mozne ich aj vnorovat

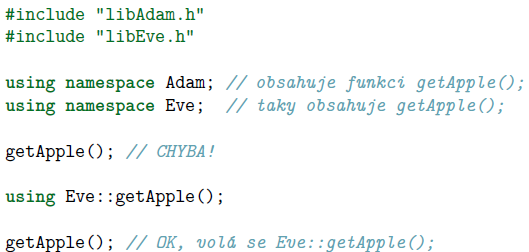
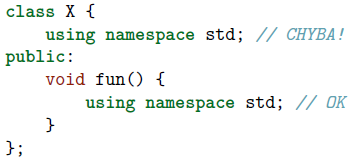
Spristupnenie menneho priestoru:

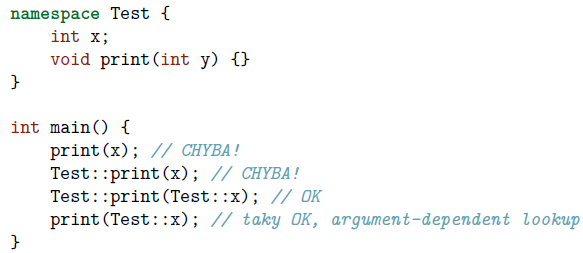
* syntax: **using namespace nazov;**
* vlozi obsah celeho menneho priestoru do aktualneho menneho priestoru
* zvysuje riziko konfliktov
* **nikdy nepouzivat v hlavickovych suboroch a vzdy az po vsetkych #include**

Spristupnenie entity menneho priestoru:

* syntax: **using nazov\_priestoru::nazov\_entity;**
* vlozi len odkazovanu entitu
* ma prednost pred using namespace nazov;

using a using namespace nieje mozne pouzivat vo vnutri triedy ale je mozne pouzivat vo funkciach/metodach a v blokoch



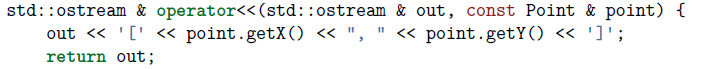


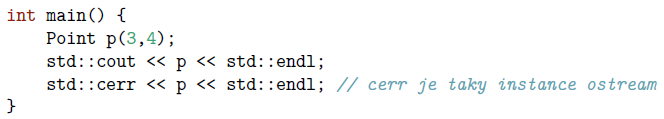
Vstupne/vystupne prudy:

* rozhranie pre vstupne prudy: istream
* rozhranie pre vystupne prudy: ostream
* prudy sa nesmu kopirovat, len presuvat (move)
* **cin:** standardny vstup – instancia triedy **istream**
* **cout:** standardny vystup – instancia triedy **ostream**
* **cerr:** standardny chybovy vystup – instancia triedy **ostream**
* **clog:** standardny logovaci vystup – instancia triedy **ostream**

Operator vystupu <<:

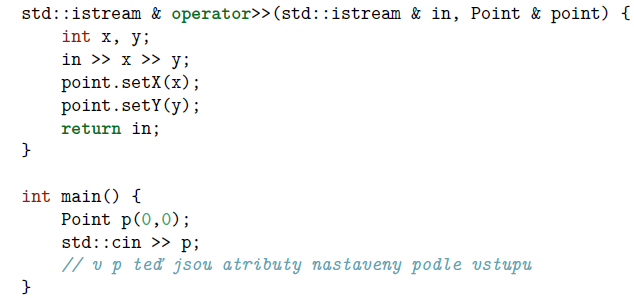
* zapis do vystupneho prudu
* v skutocnosti pretazeny operator bitoveho posunu
* pretazenie << pre vlastne triedy:
* **ostream & operator<< (ostream & out, const Object & object) { … }**
* je to funkcia, nie metoda (nemame ako pridat metodu do triedy ostream)
* prvy parameter: nekonstantna referencia na vystupny prud, vystupom do prudu sa meni jeho vnutorny stav
* druhy parameter: konstanta referencia na objekt, ktory chceme vypisat
* operator vracia referenciu na vystupny prud co umoznuje retazenie vystupu



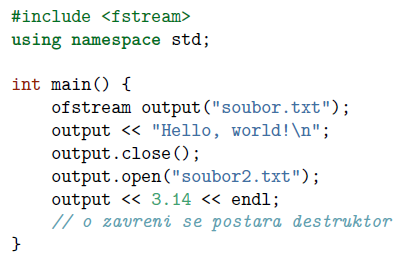


Operator vstupu >>:

* citanie zo vstupneho prudu
* pretazenie >> pre vlastne triedy:
* **istream & (istream & in, Object & object) { … }**
* prvy parameter: nekonstantna referencia na vstupny prud, citanim zo vstupneho prudu sa meni jeho vnutorny stav
* druhy parameter: nekonstantna referencia na objekt, do ktoreho chceme zapisovat, tj. budeme ho menit, preto nekonstantna referencia



Suborove prudy:

* hlavickovy subor <fstream>
* triedy: ifstream, ofstream, fstream
* konstruktor s menom suboru, metody open() a close()

Typ otvorenia:

* ios::in – vstup, implicitny druhy parameter konstruktoru pre ifstream
* ios::out – vystup, implicitny druhy parameter konstruktoru ofstream
* ios::in | ios::out – vstup/vystup, implicitny parameter konstruktoru fstream

Sposob otvorenia:

* ios::binary – binarne data, implicitne su textove
* ios::app – append, vystup na koniec suboru
* ios::trunc – vymaze subor

Trieda istream:

* metoda get(); vracia jeden znak (viac znakov pomocou get(buffer, length) do konca riadku)
* metoda getline(buffer, length); podobne ako get(); ale zahodi znak konca riadku
* metoda read(buffer, count); blokove citanie daneho poctu bytov (hlavne binarne subory)
* metoda gcount(); vrati pocet znakov nacitanych pri poslednom vstupe
* metoda peek(); nahlad na dalsi znak bez jeho precitania
* metoda ignore(count); zahodi count znakov zo vstupu
* atd …

Trieda ostream:

* metoda put(); vypise jeden znak
* metoda write(output, count); protiklad read();

Pozicia a posun v subore:

* tellg(), tellp(): ziskanie pozicie ukazatela
* seekg(), seekp(): nastavenie pozicie ukazatela, parametre: off, way

off: offset – o kolko sa posunut od pozicie way

way: ios::beg – zaciatok suboru, ios::cur – aktualna pozicia, ios::end – koniec suboru

* pociatocna pozicia v subore zavisi na mode otvorenia suboru

Vyrovnavacie buffery:

* data poslane do prudu nemusia byt automaticky zapisane do ciela
* vyrovnavacia pamat typu streambuf pre kazdy prud
* prenos z vyrovnavacej pamate do ciela:

1. pri uzavreni suboru (close(), destruktor)

2. pri zaplneni buffer

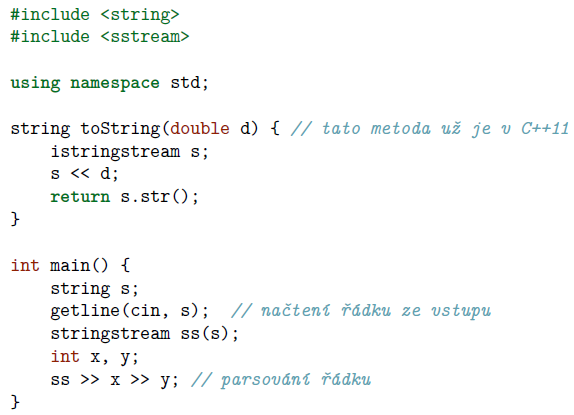
3. explicitne pomocou manipulatorov: endl, flush, sync, unitbuf

Manipulatory:

* specialne objekty, ktore je mozne predavat operatorom << a >>
* definovane v <iostream> a <iomanip>
* flush: vyprazdni buffer
* endl: zapise koniec riadku a vyprazdni buffer
* dec, hex, oct: zmeni sposob reprezentacie cisel
* setw, setfill, setprecision: meni format vstupu a vystupu
* left, right: meni zarovnanie

Retazcove prudy:

* hlavickovy subor <sstream>
* pre ukladanie dat pouziva pamat namiesto suboru
* triedy: istringstream, ostringstream, stringstream
* konstruktor moze brat retazec
* metoda str() nastavi obsah prudu, bez parametru vrati aktualny obsah prudu ako string



Kontajnery:

* usporiadanie datovej struktury je casto nezavisle na konkretnom type dat (pole cisel a pole ukazatelov maju velmi podobny sposob ulozenia v pamati a maju aj podobne operacie)
* genericke kontajnery: funguju nezavisle na tom, co sa do nich uklada

Typy kontajnerov:

Sekvencne:

* array: pole, suvisly blok pamati, pevna velkost, kontrola medzi
* vector: dynamicke pole, suvisly blok pamati, moze sa za behu zvacsovat
* deque: obojsmerna fronta, rychle vkladanie aj odoberanie na oboch koncoch
* forward\_list, list: jednostranne, obojstranne zretazeny zoznam

Asociativne (rychle vyhladavanie):

* set: usporiadana mnozina
* map: asociativne pole, slovnik – mnozina prvkov v tvare (kluc, hodnota) uspor. podla kluca
* multiset, multimap: map umoznujuci opakovanie sa klucov

Adaptery (rozne rozhrania pre sekvencne kontajnery):

* stack (zasobnik)
* queue (fronta)
* priority\_queue (prioritna fronta)

Dvojice, trojice … n-tice:

* nepatria medzi kontajnery, maju ine operacie
* pair: drzi dva objekty rovnakych alebo roznych typov, napr: pair<int, char> p1;
* tuple: drzi fixny pocet objektov rovnakych alebo roznych typov

Retazce string:

* niesu to genericke kontajnery ale funguje pre nich vela veci ako pre kontajnery

Vector:

* dynamicke pole, ktore sa vie zvacsovat
* suvisly blok pamati
* rychlost pristupu k prvkom ako bezne pole
* pri zaplneni vyhradenej pamati sa alokuje pamat s vacsou kapacitou a toto pole sa tam prekopiruje, vacsinou sa alokuje nove pole 2\* vacsie
* operacie: push\_back(); pop\_back(); resize(); reserve(); size(); capacity(); …

Iteratory:

* inteligentne ukazatele
* rozne druhy iteratorov – podla typu kontajnerov
* typicky podporuju sekvencne prechadzanie kontajneru aspon v jednom smere
* rozne metody kontajnerov beru alebo vracaju prave iteratory
* begin() – vrati iterator na zaciatok kontajneru
* end() – vrati iterator za koniec kontajneru

Iterator na vector:

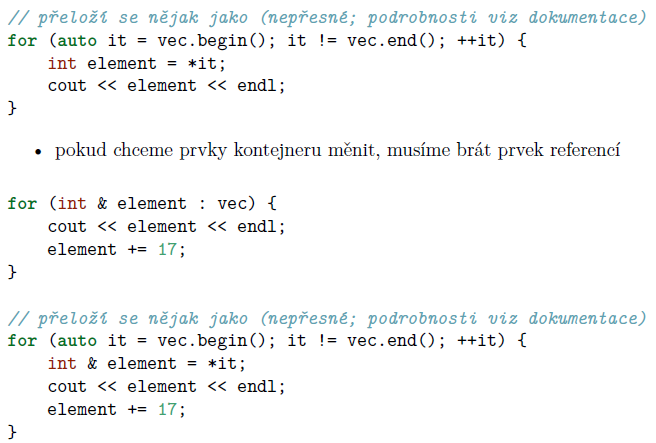
* vector<typ>::iterator nazov, napr: vector<int>::iterator it;
* ako ukazatele do pola, dereferencuje sa rovnako ako ukazatel
* v pripade zvacsenia kapacity vektoru iteratory prestavaju byt platne

Typovy specifikator **auto**:

* dostupny od C++11 – kompilator vydedukuje typ sam z inicializacie
* pr: vector<int> pole; auto it = pole.begin();

Nova forma cyklu for od C++11:

* pr: vector<int> pole; for (int element : pole) { cout << element << endl; }



Set:

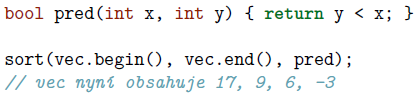
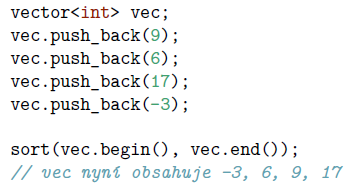
* usporiadana mnozina
* prvky sa nemozu opakovat
* iteratory su vzdy konstantne
* operacie: size(); find(); erase(); insert(); …
* operacia insert vracia pair (dvojicu), prvy prvok je iterator na vlozeny alebo uz existujuci prvok a druhy prvok je hodnota boolean ci prebehlo vlozenie prvku do mnoziny

Map:

* mapovanie klucov na hodnoty
* usporiadany podla klucov
* kluce sa nemozu opakovat
* porovnavanie a usporiadanie sa pozera len na kluc
* kluc sa nesmie modifikovat, hodnota sa moze
* operacie: find(); erase(); size(); …

Algoritmy:

sort:

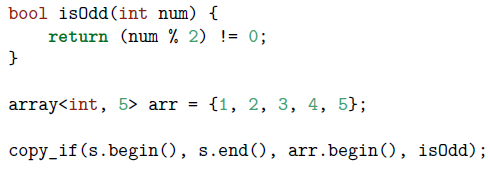


copy:

* berie tri argumenty: prve dva su iteratory urcujuce zdrojovy rozsah, treti argument je iterator na miesto, kam sa ma kopirovat
* je nutne zabezpecit aby v cielovom kontajneri bolo dostatok miesta

copy\_if:

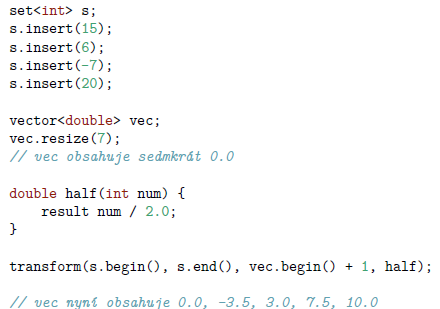
* oproti algoritmu copy ma argument naviac, predikat – kopiruju sa len objekty splnajuce tento predikat



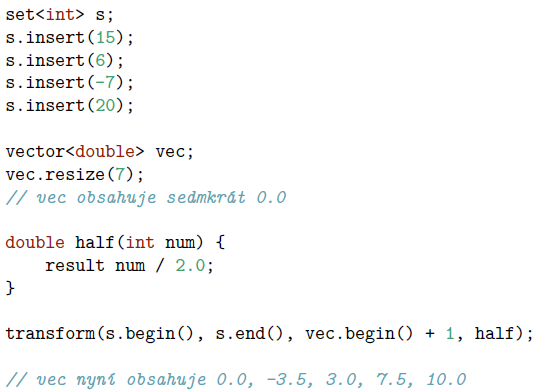


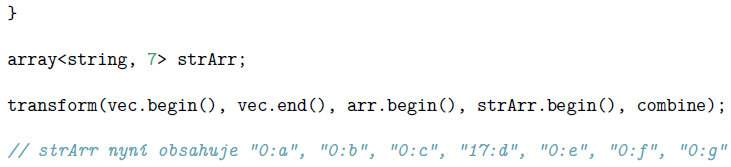
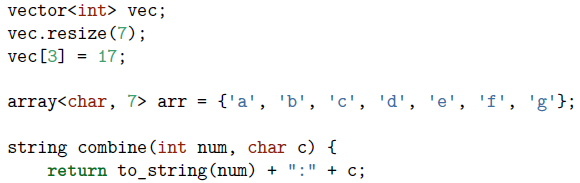
transform:

* algoritmus kopirovania s modifikaciou
* argumenty: 2 iteratory urcujuce zdrojovy rozsah, jeden iterator na ciel, unarna operacia, ktora sa aplikuje na zdrojove prvky, takto modifikovane sa ukladaju do ciela



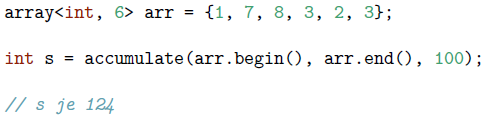
transform (druha varianta):

* argumenty: 2 iteratory urcujuce zdrojovy rozsah, jeden iterator na zaciatok druheho rozsahu, iterator na ciel, binarnu funkciu
* predpoklada sa ze oba zdrojove rozsahy su rovnake, preto sa neudava iterator na koniec druheho rozsahu



accumulate:

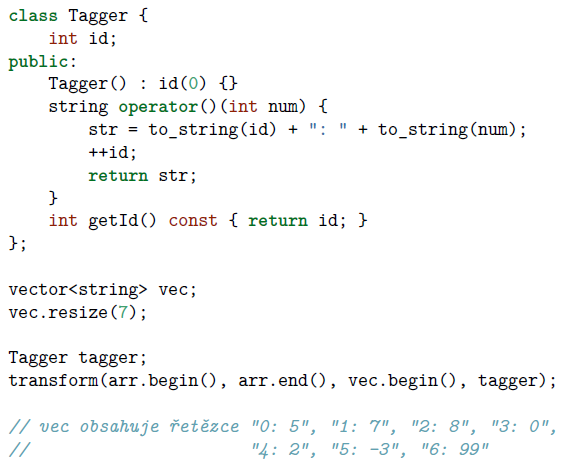
* v zakladnej verzii zrata vsetky objekty v danom rozsahu pomocou operatoru +
* v druhej verzii umoznuje dodat vlastnu funkciu/funkcny objekt miesto operacie +
* obe verzie okrem iteratorov rozsahu beru aj pociatocnu hodnotu
* akumulacia prebieha zlava



Funkcny objekt:

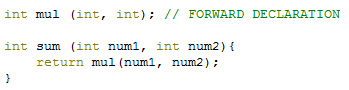
* objekt triedy, ktory ma definovany operator funkcneho volania ()



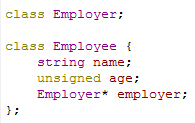
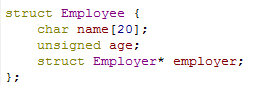


Forward Declaration

- deklarácia identifikátoru bez jeho definície, pred použitím tohto identifikátoru  
- kompilátor pri preklade potrebuje vedieť isté veci ale nie úplne všetky  
napr: potrebuje vedieť veľkosť pre alokáciu pamäte, typ pre typovú kontrolu alebo signatúru funkcie  
ale nepotrebuje vedieť hodnotu, ktorú premenná alebo konštanta drží ani definíciu funkcie

  
*Funkciu mul môžme takto použiť vo funkcii sum aj napriek   
tomu, že nevieme ako vyzerá definícia mul*

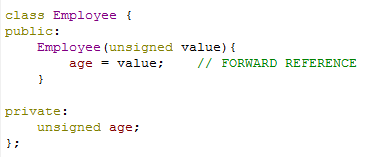
- doprednú deklaráciu môžme použiť aj v prípade tried a štruktúr, vtedy ale môžme použiť len ukazateľ na forward declared triedu alebo štruktúru, nie samotný objekt, pretože sú to užívatelsky definované typy a bez definície nieje možné zistiť veľkost inštancie takejto triedy/štruktúry, pričom ukatateľ na akýkoľvek typ je vždy rovnakej veľkosti



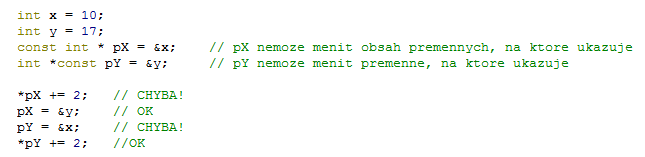
*V jazyku C je tu možné vynechať aj doprednú deklaráciu a to aj v príklade funkcie mul vyššie.*

Forward Reference

- skutočné využitie identifikátoru ešte pred jeho deklaráciou  
- v C++ všeobecne zakázané, ale povolené pre atribúty triedy  
- kompilátor si musí pamätať definíciu funkcie, v ktorej sa používa dopredná referencia, až kým nenájde deklaráciu identifikátora



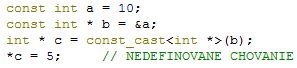
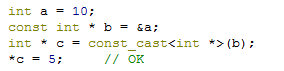
**const** a ukazatele v C/C++



* int & x (v C++) je int \*const x
* a konštantná referencia const int & x je const int \*const x

const\_cast

- odstránenie alebo pridanie špecifikátoru **const** k ukazatelu alebo referencii  
- pri odstraňovaní „konštantnosti“ ukazatela alebo referencie by sme si mali byť istý, že  
 objekt(premenná), na ktorý ukazujú nieje konštantný(tj. môže sa meniť)

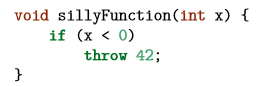
 

Vynimky:

* objekty, reprezentujuce vynimocne stavy za behu programu
* vyhodenie(throw) vynimky signalizuje chybovy stav
* je mozne ich zachytavat a reagovat na ne
* vyhodenie vynimky za pomoci prikazu **throw**, vyhodit(throw) sa moze lubovolna hodnota,  
  pouzivaju sa k tomu ale specialne urcene objekty triedy std::exception

Zachytavanie vynimiek:

* pouzivaju sa **try{ … }** – **catch( … ) { … }** bloky
* blok try obsahuje kod, v ktorom sa ma zachytavat vynimka
* blok catch obsahuje kod, ktory sa ma vykonat pri zachyteni danej vynimky
* chytanie vynimky hodnotou sa nedoporucuje, prebieha vtedy kopirovanie vynimky a hrozi  
  dalsie vyhodenie vynimky
* doporuceny sposob chytania vynimiek je chytanie referenciou
* v bloku catch mozme znovu vyhodit vynimku pomocou throw, alebo vyhodit inu vynimku
* **catch(…)** = catch-all, zachyti vynimky vsetkych typov
* **throw by value, catch by reference**





Stack-unwinding:

* proces odvijania zasobnika
* pri vyhodeni vynimky sa hlada blok try postupnym odvijanim zasobnika(hlada po urovniach),  
  ked ho najde, hlada v jeho catch blokoch, ktory zachytava vynimku daneho typu
* ak sa vynimka nikde nezachyti, zavola sa std::terminate a program skonci

Hierarchia vynimiek:

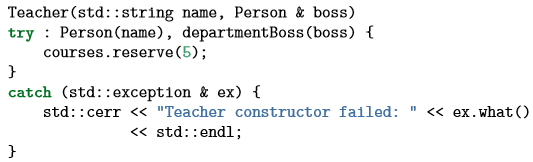
* zakladna trieda je std::exception, obsahuje virtualnu metodu what(), ktora vracia popis  
  vynimky typu const char\*
* je mozne vytvarat vlastne vynimky, doporucuje sa dedit od std::logic\_error alebo  
  std::invalid\_argument

Dedicnost pri chytani vynimiek:

* catch bloky sa prehladavaju v poradi v akom sa nachadzaju v programe, tj. zhora dole
* tzn. catch bloky treba pisat od konkretnych k obecnym, v smere zhora dole
* kompilator na to upozornuje

Vynimky a konstruktory:

* konstruktor nic nevracia, takze vynimka je casto jedinou moznostou reakcie na necakanu  
  behovu chybu vzniknutu v konstruktore
* ked je vyhodena vynimka z konstruktoru, zavolaju sa destruktory vsetkych veci, ktorych  
  inicializacia uz uspesne prebehla ako predkovia a atributy
* pre moznost zachytenia vynimky vzniknutej v inicializacnej sekcii konstruktoru ma C++  
  specialnu syntax



Vynimky a destruktory:

* z destruktora nieje vhodne (takmer) nikdy vyhadzovat vynimky

noexcept:

* prikaz, ktory specifikuje, ze dana funkcia/metoda nevyhodi ziadnu vynimku
* ak taka funkcia/metoda aj tak vyhodi vynimku, vynimka sa nezachytava a rovno sa vola  
  std::terminate
* destruktory su implicitne noexcept
* noexcept sa da pouzit aj ako operator na zistenie moznosti vyhodenia vynimky vyrazu





Vynimky a kniznica iostream:

* objekty kniznice iostream nevyhadzuju vynimky ale nastavuju priznaky
* objektom z iostream mozme povedat aby vyhadzovali vynimky

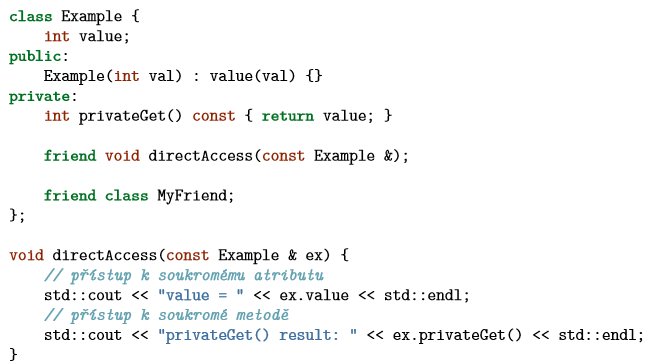
RAII(**R**esource **A**cquisition **I**s **I**nitialization):

* idiom uvolnovania prostriedkov (pamat, otvoreny subor, …)
* ine mozne riesenie: uvolnovat prostriedky aj v bloku try aj v kazdom catch
* v inych jazykoch pomocou bloku finally()
* pokial chceme spravovat prostriedky => Rule of Three, treba zakazat kopirovanie alebo  
  definovat explicitny kopirovaci konstruktor/operator=
* pokial prostriedky nechceme spravovat => Rule of Zero, trieda ma len implicitny destruktor  
  a kopirovanie

inline:

* napoveda pre prekladac, ze telo danej funkcie ma vlozit na miesto jej volania
* v dnesnej dobe prekladace vedia lepsie posudit ako programator ktore funkcie inlinovat
* metody definovane v deklaracii triedy su automaticky inline

friend:

* specifikuje spriatelenu funkciu/metodu/triedu
* sposob ako obist pristupove prava (private/protected)
* specifikator friend pri funkcii/triede udava, ze dana funkcia/trieda moze pristupovat ku  
  vsetkym atributom a metodam danej triedy, ktora to deklaruje
* pri pretazeni funkcie, je treba udelit povolenie kazdej osobitne
* trieda sama si urcuje, kto je jej priatelom(friend)
* priatelstvo nieje dedicne ani tranzitivne, je exkluzivne
* spriatelena funkcia/trieda ziskava ale ciastocny pristup k potomkovi triedy, ktora  
  priatelstvo deklarovala, ale pristup ma len k castiam zdedenym od tejto triedy

Pretazovanie operatorov:

1. Operator ako funkcia  
2. Operator ako metoda

1. Operator ako funkcia

* casto pouzivane s deklaraciou friend
* pocet parametrov podla arity operatoru
* pouzitie operatoru a + b sa prelozi na volanie operator+(a,b)

2. Operator ako metoda

* lavy operand je \*this
* pocet parametrov o jeden menej ako pri implementacii funkciou
* pouzitie operatoru a + b sa prelozi na volanie a.operator+(b)

Operatory, ktore sa nedaju pretazovat: ., ::, ?, :  
Operatory, ktore sa musia implementovat ako metody: =, [], ->, ()

Obmedzenia pretazovania operatorov:

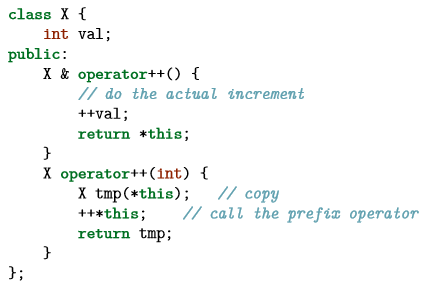
* nedaju sa vytvarat nove operatory
* nemoze sa menit pocet parametrov, okrem operatoru ()
* aspon jeden z parametrov musi byt uzivatelsky definovaneho typu
* neda sa menit priorita a asociativita operatorov

Symetria pretazenia operatorov:

* pokial pretazime operator tak, ze bude brat ako parametre rozne typy, treba pamatat na   
  vsetky mozne poradia zadania argumentov

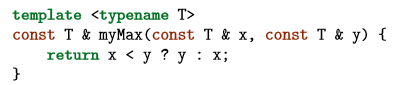


Operator (prefix)++ vs (postfix)++:

* prefix ++ vracia referenciu
* postfix ++ vracia hodnotu a vytvara kopiu, pre odlisenie berie nepouzivany int parameter

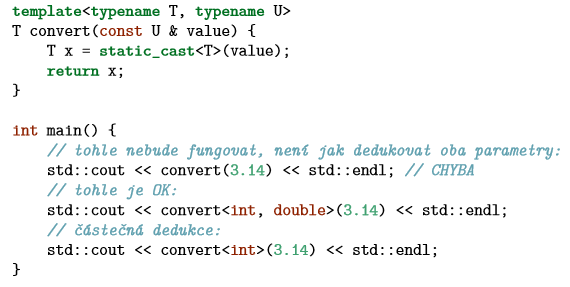
Sablony:

* sposob ako pisat kod aplikovatelny na rozne typy bez duplikacie kodu
* skutocne pouzite typy sa do sablony specifikuju v dobe prekladu(kompilacie)
* syntax: template <zoznam typovych parametrov>
* typove parametre mozu mat implicitne hodnoty
* napr: template <typename T, int size = 100, typename U = bool>
* za deklaraciou sablony nasleduje definicia triedy/funkcie, ktora je sablonou parametrizovana
* sablonove funkcie sa mozu pretazovat
* funkcia moze byt pretazena sablonovou aj nesablonovou verziou, nesablonova ma prednost



Automaticka dedukcia sablonoveho typu funkcie:

* volanie len za pomoci mena funkcie: umoznuje pretazenie nesablonovou verziou
* funkcia a prazdny zoznam sablonovych parametrov <>: neumoznuje pretazenie  
  nesablonovou verziou
* ciastocna automaticka dedukcia: vynechanie niektorych parametrov



Sablony tried:

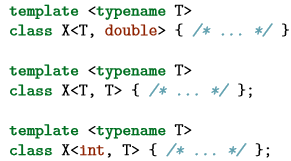
* pri sablonach tried neexistuje automaticka dedukcia sablonoveho typu
* vo vnutri sablonovych tried mozu byt sablonove metody

Specializacia sablony:

* implementacia triedy pre konkretne sablonove parametre
* triedy nevieme pretazovat ako funkcie, preto existuje specializacia

Ciastocna specializacia sablony:

* ciastocne uvedenie sablonovych parametrov



Nesablonove typove aliasy:

* funguju ako typedef ale s inou syntaxou
* pr: using Number = int, Number je odteraz typovym aliasom pre integer

Netypove sablony:

* parametry sablon niesu len typy
* mozu byt celociselne hodnoty, sablony, …
* musia to byt konstatne hodnoty zname v dobe prekladu

Instanciacia:

* samotna sablona nedeklaruje ziadnu triedu ani funkciu
* sablonove funkcie a triedy vznikaju az pri instanciacii(pri konkretnom pouziti sablony)
* instancia sablony sa vytvori len jedna, ta, ktora sa skutocne pouzije

