# Programování v c++

Tomáš Turek

*Přednáška 1*

---

## Uvod C/C++

- prvetivejsi asembler
- po prelozeni primo na hw (neni treba run time support)
- kod dela to co ma (mohou nastavat problemy)
- preference hodnotovych typu (jinak explicitne ukazat referencni typy)
- prevazne neni dynamicka alokace, ale objekty vznikaji jinak (treba primo jako promenne)
  - daji se objekty i stehovat (move)
- neni garbage collector
  - pokud se vyrobi dynamicky objekt, tak se musim starat aby i zanikl
  - C++ 11 jsou chytre pointery

*Přednáška 2*

---

### Include

- hlavickovy soubor (vetsinou jen hlavicky funkci) (a.h / a.hpp)
- include probiha textovym pridavanim
- prakticky jen interface fce a pak v jinem souboru je implementace dane fce
- dochazi mene k rekompilaci
- deklarace trid musi byt v hlavickovem souboru (pak se ale musi casto menit, kvuli treba pridani promenne)
- genericke sablony z kterych pak prekladac vyrabi presne instance (stejne rychle, jak natvrdo udelane predem)
  - vzdy se vyrobi nova binarni implementace pro dany typ
  - byva vetsinou take v hlavickovych souborech (aby k nemu mel pristup)
  - prekladac preklada pouze pokud se vyuziva nejaka implementace

### Kompilace

- uz objektove moduly jsou udelane primo na dane CPU a OS

- z prekladacu vypadne meziprodukt (zkontrolovane pravidla)
- linker pak preklada a optimalizuje (proto aby se neprekladala stejna implementace genericke sablony vicekrat)
- linker / prekladac si uchovava databazy vsech fci a jejich prelozenych stejnych fci
  - proto aby se zmenilo jen neco malo v kodu, tak at se nepreklada vse znovu

### Knihovny

- staticke
  - stara se o ne linker
  - binarni a je treba i hlavickovy soubor
- dynamicke
  - primo az do executable

### Program

- hlavni vstup je funkce main
- `char *` je retezec (na konci retezce je `\0`)
- `char **` je vicero retezcu
- existuje i standardtni pole, ale vetsinou se pouziva kontainerr vector
- namespace std
- pro pouziti `::` tak predchozi neni vyraz, jinak pokud je to vyraz ma typ a taky hodnotu, tak se normalne pouziva `.`

```cpp
#include <iostream>

int main(int argc, char** argv) {
    std::cout //normalni output (diky iostream)
        << "Hello World"
        << std::endl;
    return 0;
}
```

- hlavicka syntax
  - `#ifndef a_hpp_` aby prekladac podruhe nebude zabyvat (if not defined)
  - `#define a_hpp_` definice
  - `#endif` konec definice
- C++20 ma moduly
  - `export` interface
  - `module` definice modulu
  - `import` importovani modulu
  - jeste neni zcela vyresene a standardizovane (microsoft zatim udelal jak by se to dalo pouzivat)

  - cilem je usetrit kompilacni cas (jakoby ty hlavicky a.ixx budou v binarni forme a ne jako zdrojak)
- `const t_arg & arg` konstantni reference
  - pro rychlost, jinak by se kopiroval obsah (zalezi na velkikosti predavane hodnoty)
- existuje konstruktor `new`
  - dynamicka alokace (!neni GC!)
  - vznikne odkaz na dany objekt a na konec se musi smazat pomoci `delete`
  - nedoporuceny postup, protoze se musi smazat na spravne misto (jinak muzu sahat do spatne pameti a dostanu se do problemu), takze NEPOUZIVAT
- take existuje `std::shared-ptr<T> x = std::make-shared<T>(...);`
  - pamatuje se taky kolik existuje odkazu (citac)
  - known as: *chytry pointer*
  - 'nahrazka' za garbage collecting, ale je pomalejsi nez GC
- pokud vim, ze bude jen jeden ukazatel (vlastnik), tak lze pouzit `std::unigue-ptr<T> x = std::make-unigue<T>(...);`
  - nelze kopirat, ale da se posunout pomoci `move`
- lokalni promenna `type name(parameters)`
  - prima lokalni alokace, vetsinou staci misto dynamicke alokace

*Přednáška 3*

---

### Trida

```
class C{
    void function f4();
};
```

- pristup ke tride je pomoci `C::f4();`

### Deklarace funkci

- inline
  - pokud je v cpp souboru, tak neprekroci hranici souboru
- implicitne inline
- non inline
- static
  - staticka funkce, ktera se nevaze na dany objekt
- virtual
  - da se podedit a prepsat
- abstraktni trida
  - podobne jako virtual `virtual void f6() = 0;`

### Deklarace promenne

- extern (v hpp)
  - globalni promenna (nejlepe nepouzivat zbytecne)
- inline
  - defiinice promenne (ne jen deklarace) muze byt i deklarace
- staticka promenna uvnitr tridy je prakticky to stejne jako globalni promena
- `cnost`
  - pouziva se u odkazu (nema pravo modifikovat)
- `constexpr`
  - prekladac umi pracovat primo s hodnotou

### Kontejner

- `std::vector<std::string>`
- da se pres cyklus for

```
using t_args = std::vector<std::string>;
t_args p = //..
for (auto && x : p){
    // x is current item
    std::cout << x;
}
```

- reprezentace v pameti
  - uvnitr vectoru jsou tri ukazatel
    - prvni ukazuje na prvni blok
    - druhy ukazuje na prvni prazdny (rozsirovani)
    - posledni ukazuje za konec
  - pokud se jenda o vektor stringu, tak jsou objekty vlastne uplne stejne jak samotnuy vektor
- da se to predstavovat jako opravdovy kontejner, ktery obsahuje pak dane prvky

### Hodnoty a reference

- co kdyz se vytvari objekt a pak se priradi/okopiruje do nove promenne (to je jine u kazdych jazyku)
  - v C++ se vse predava jako hodnota a nedela se refernec (pokud se explicitne nerekne), reference se dela `*x`
- immutable types
  - typy, ktere se nedaji modifikovat po jejim vzniknuti (treba byva string)
  - pokud je immutable, tak pokud se modifikuje, tak se vytvari novy objekt s tou novou hodnotou
- v C++ se da taky prepsat funkce `=`, takze vlastne se nemusi jednat o predavani hodnot
- proto aby byl objekt ulozeny jako ukazatel, tak lze pouzit hloupy a chytry pointer

  - hloupy po sobe sam neuklidi
  - chytre pointery bez inicializace budou inicializovany na nulovy pointer
    * pristup k objektu pokud je ulozen jako ukazatel je pomoci `x -> health();`
    * zatimco u hodnotovych typu je to `x.health();`
    * takze se i takhle da poznat co to je za promennou
  - hloupe pointery se obcas pouzivaji jako postranne pointery
    * za predpokladu, ze maji kratsi zivotnost
    * jednodussi protoze se neresi alokace a realokace
    * musi byt prvni chytry pointer a mit delsi zivotnost
- pak taky lze mit hodnotu jako referennci
  - to funguje podobne jako pointer ale chova se jako objekt
    * takze se k nemu pristupuje `x.health();`
  - pri vznike se musi rict kam ukazuje
  - ten objekt neni vlastnen promennou jako referenci
  - pri `=` se predava obsah (pokud to neni inicializace)

*Přednáška 4*

---

- lze predefinovat operator `=`
  - `class::operator=`

### Refernece

- `const T &`
  - pri pouziti predavani parametru, aby se nekopirovalo
  - aby nesel zmenit obsah (nelze modifikovat)
- `T &`
  - modifikovatelna L- reference
  - prirazeni musi byt L - value
    * opakovatelne pristupny
- `T &&`
  - R - value reference
  - vyraz, ktery pokud se zavola znovu, tak vytvori novy objekt
- `auto && x = ...`
  - forwarding reference
  - muze byt R i L - value

### Argumenty funkci

- Jak predat promenou:
  - pokud je treba menit `T &`
  - pokud je levne kopirovani parametru `T`
  - jinak pokud nepodporuje kopirovani `T &&`
  - jinak `const T &`

---

- vstupni argumenty jako odkaz je nutne pouzit `const`

### Metody trid

```
class my_string{
    public:
        my_string concat(conts my_string & b) const;
        //const na konci je vlastne const my_string* this
}
```

### Navratove hodnoty

- pokud se **zpristupnuje obejkt** v nejake strukture
  - pokud dovolime modifikaci `T &`
  - jinak `const T &`
  - po vraceni objektu musi nejakou dobu prezit
  - kdy nepouzivat reference (dalsi slide)
    - anonymni promenna
      - promenna zanikne drive nez se pouzije
      - muze se take alokovat pro jine data a pak je v pameti uplne neco jineho
      - tady prekladac vyhodi varovani (plus i u druhe moznosti)
    - lokalni promenna
      - technicky stejny jako prvni priklad
    - globalni promenna
      - technicky vzato pouzitelne
      - ale je sdilena pro nekolik volani funkce
    - dynamicka alokace
      - vlastne se vraci pointer
      - dochazi k memory leaku, protoze uz nikdo nesmaze objekt
    - chytry ukazatel
      - pred ukonceni funkce se chytry pointer vynuluje a tedy nic nepreda
    - bohuzel to v jednoduchych pripadech muze fungovat, protoze se ty data jeste nemusi smazat
    - pouzivani referenci
      - vetsinou by melo byt, ze pokud vracim `const` tak dostavam i `const`, poppripade naopak (modifikovatelne)
      - ostani ‘sisate’ moznosti jsou nelogicke (i chybne)
- v ostatnich pripadech predavame hondotu `T`
  - klasicky na konci `return x;`
  - prekladac je donucen pouzit copy/move-elision
    - proto aby bylo predavani rychlejsi

* nepouzivat v retunr `std::move`

### Logicka konstantnost

- ne vzdy kdyz se neco da udelat (treba predavat bud cont data a nebo modifikovatelnou referenci), tak je lepsi to udelat tak, jak to dava smysl

### Pozice const

```
const T * x; // Neda se zmenit obsah, ale modifikace pozice.
T const * x; // Stejny jako prvni.
T * const x; // Meda se modifikovat pozice kam ukazuje, ale data se daji zmenit.
const T * const x; // Nic se nedda menit.
/* Const na konci limituje prakticky jen nas a tak moc nedava smysl. */
```

### Ne vzdy se pouziva logicka konstantnost

```
void f( const shared_ptr<T> & p){
    *p = 7; // Nelzee modifikovat shared pointer, ale lze zmenit data na ktere ukazuje.
}
```

### Overload resolution

- pretizeni
- lisi se poctem, typama a popr. R a L value a ne v jakem je to kontextu

```
int x,y;
double p = /* double */ x/y; // Na intu je vzdy deleni celociselne. Je treba aspon jednu pr
```

- copy and write pred kopirovani datove struktury si nejdrive jen ukazuje na stejny objekt dokud se jeden nezmeni, to se tesne pred tim prekopiruje

### Vraceni promennych jako hodnot

- pokud se vraci lokalni promenna, tak je povinne *copy-elision*, protoze pred tim se udela misto pro navratovou hodnotu a tam se prida pak ta hodnota
  - ta se nazyva jako anonymni promenna
  - pokud se prirazuje hned pri inicializaci, tak to lze hned ulozit do dane promenne
- pokud se jedna o anonymni objekt, tak pak dochazi k *move-elision*, jinak je to obdobne, ale akorat se nekopiruje, ale presouva

*Přednáška 6*

---

- prekladac pokud uvidi `auto` tak neresi jestli je to reference nebo ne (v podstate i `const`)
- prekladac bez copy-elision pred C++11
  - kdyz se hned inicializuje promenna tak se nahradi konstruktor

  - pokud se ale prirazuje pozdeji, tak prekladac vytvori promennou do kterre uklada mezivysledek a pak zkopiruje do dane promenne
- v C++11
  - dve nove funkce move construkctor a move assignement
  - ty se volaji pokud objekt z ktereho se berou data je mozne okrast (preberu ukazatel na ten blok, ten predchozi pointer je treba vynulovat)
- v C++17
  - pridani copy-elision
  - kdyz je prikaz `return` s lokalni promennou, tak se lokalni promenna zrusi a vsechny reference se nahradi vzdalenym pametovym mistem kam se pak vraci
  - `T x = f(...)` tady x inicializuje funkce f a hend ji sklada

Operace `MOVE`

- vyvola se pokud se zavola `std::move()` a nebo pokud je hodnota r-value
- specificke funkce
  - copy-constructor
    - kdyz se inicializuje objekt kopirovanim
    - `T(const T &)`
  - move-constructor
    - kdyz se inicializuje objekt pomoci `move`
    - `T(T &&)`
  - copy-assignement
    - prirazeni nove hodnoty do stareho objektu kopirovanim
  - move-assignement
    - prirazeni nove hodnoty do stareho objekut pomoci `move`

*Přednáška 7*

---

**The Rule of Five**

- pokud je treba napsat destruktor tridy, tak je nutno napsat i vsechny 4 instrukce copy a move assignemnet a constructor (doporuceni)
- radeji se vyhnout ukazatelum `*` a pouzivat typy, ktere se dokazaji o sebe postarat
- pokud se zmeni jedna ze ctyr metod, tak prekladac vynecha implementaci vsech ctyr
  - pro zruseni se da pouzit `= delete`
  - pokud se chteji zachovat tak `= default`

**Abstraktni tridy**

- je lepsi vzdy napsat `virtual ~C(){}` (virtualni destruktor)

  - aby pokud mazu potomka na ktereho se divam jako na predak, tak smazu celeho otomka
- neni mozne napsat `std::vector<AbstarctClass>` protoze se udela prostor pro predky a pokud se tam budu pokouset dat potomky, tak bud to neprojde prekladacem a nebo se tam da pouze ta cast, ktera je spolecna s predkem
  - misto toho se da `std::vector<std::unique_ptr<AbstractClass>>`

## Dynamicka alokace

- dynamicka alokace ma smysl jen v par pripadech, jinak se pokusit tomu vyhnout
  - uziti dedicnosti
  - komplikovana zivotnost objektu
- preference chytrych pointeru (drzet za ocas)

```
#include <memory>
void f(){
    std::unique_ptr<T> p = std::make_unique<T>();
    std::unique_ptr<T> q = std::move(p); // p is nullptr
}

void g(){
    std::shared_ptr<T> p = std::make_shared<T>();
    std::shared_ptr<T> q = p; // pointer copied, it is shared with p and q
}
```

*Přednáška 8*

---

### Ukazatele a reference

- pokud nepotrebujeme prevest vlastnictvi a jen ukazovat, tak je lepsi pouzit hloupe pointery
  - je nutne aby mel chytry pointer delsi zivotnost
- jeste existuje `std::weak<_ptr<T>`, ktery se pouziva hodne malo

## Ukladani hodnot vedle sebe

- pole: `std::array<T,n>a;` kde ‘std::size_t n = c;
- pomoci struktury
- take se da tuple `std::tuple< T1, T2, T3> a;` pak se dostaneme k polozce pres `std::get<1>(a);`
- pro promenlive velikosti je dobre pouzit vektory
  - pokud jsou stejne, tak staci `std::vector< T> a(n);`
  - pokud jsou jine, tak pouzit ukazatele `std::vectr< std::unique_ptr< Tbase>> a;`

- v pameti musi dojit k zarovnani a tudiz se neda spolehnout na tom kde jsou prvky‘

*Přednáška 9*

---

- vekto ma jak `insert()` tak i `emplace()` kdy v insertu se vklada uz vytvoreny objekt a emplace se teprve vytvori uz ve vektoru
- pokud se napise move konstruktor, tak pokud se nakonec napise `noexcept` tak se budou presouvat a nekopirovat prvky pri pridavani prvku do vektoru (pokud je treba zvysit pocet prvku)

## Kontejnery

- genericke datove struktury (seznamy, spojaky, stromy nebo hesovaci tabulky)
- obsahuji primo ty data (resi alokaci a dealokaci)
  - vsechny objekty musi mit stejny typ
  - nove objekty se vytvari in place
- pridavani a odebirani je pomoci funkce kontejneru
- prochazi pomoci iteratoru

### Sekvencni kontejnery

- `array< T, N>`
  - seznam fixovane veliksoti (nelze odebirat a pridavat)
- `vector< T>`
  - dynamicky seznam, pridava a odebira se z konce
  - `stack< T>`
    - obdobne, ale z vrchu
  - `priority_queue< T>`
    - implementace heapu
- `basic_string< T>`
  - podobne vektoru, akorat se da predelat na `const char *`
  - `string` = `basic_string< char>`
  - `u32string` = `basic_string< char32_t>`
- `deque< T>` (double ended queue)
  - rychle pridavani a odebirani na obou koncich
  - `queue< T>` (FIFO)
- `forward_list< T>`
  - spojak
- `list< T>`
  - oboustranny spojak

### Asociativni kontejnery

- objelty se pridavaji na danou pozici

- mnoziny
- mapy
  - par klic a typ
- multi-
  - vice objektu se stejnym klicem lze vlozit

**Setridene**

- `set< T>`
  - hleda se pomoci `T`
- `multiset< T>`
- `map< K, T>`
  - hleda se pomoci `K`
- `multimap< K, T>`

**Hesovaci**

- `unordered_set< T>`
- `unordered_multiset< T>`
- `unordered_map< K, T>`
- `unordered_multimap< K, T>`
- pokud neni nadefinovane `<` tak je vhodne ho dodefinovat (ne vzdy to dava smysl)
- specialni struktura `functor`
  - ma operaci `()` a lze ji prakticky pouzivat jako funkci
  - vraci `bool` promennou
- hesovaci kontejnery potrebuji dva functory
  - hesovaci funkce
    - dobre vracet `std:;size_t`
  - rovnostni porovnani
    - vraci `bool` poku vrati `true` tak se rovnaji
  - pokud nejsou definovane, tak se pokousi pouzivat genericke funktory

**Iteratory**

- kazdy kontejner ma dva typy
- `iterator`
  - vraci referneci, ktera se da modifikovat
- `const_iterator`
  - vraci const referenci
- iterator muze ukazovat bud na objekt v kontejneru anebo na imaginarni objekt za koncem kontejneru
- inicializace pomoci `auto i = begin(c);`

- pokud je treba porovnat jestli je v ramci kontejneru tak `i != end(c);`
  - take existuji primo metody na kontejnerech `c.begin();` a `c.end();`
  - take jeste je `cbegin(c);` a `cend(c);` ktere vraci konstantni iterator
- asociativni kontejnery se take da hledat pomoci `c.find(k);`
  - to vraci take iterator, pokud nenajde tak vraci end iterator (musi se kontrolovat)
- v iterovani kontejnerem je pro podminky pouzivat rovnost/nerovnost, protoze ve vsech kontejnerech jsou definovane porovnavani a ne vzdy jsou mensi/vetsi
  - pak se inkrementuje pres `++` s tim ze se preferuje prefix `++i`
    - prefix vraci referenci a postfix vetsinou hodnotou (takze je pomalejsi)
- dekrementaci a odeciteni nebo i porovnavani(`<` a `>`) iteratoru lze jen nekdy pokud tyto metody jsou definovane
- pokud je treba projit kontejner pozpatku tak nepouzivat dekrementaci, radeji pouzit `for(auto i=c.rbegin(); i!=rend(); ++i)` a to jsou reverzni iteratory

*Přednáška 10*

---

**Plneni kontejneru**

- lze vsechny pozice stejnym prvkem
  - `std::vector< std::string> c1(10, "dummy");`
- nebo kopirovanim z jineho vektoru
  - `std::vector < std::string> c2()c1.begin + 2, c1.end() -2 );`
- expandovnai kontejneru
  - pomoci `insert`
    - vlozi se jiz vytvoreny objekt
  - pomoci `emplace`
    - primo se vytvari novy objekt
- mazani objektu
  - pomoci `erase`
    - smaze jeden prvek a nebo cely intreval
  - pomoci `clear`
    - smaze cely kontejner

## Algoritmy

- sada generickych funkci pro praci s kontejnery
- neprebira cely kontejner ale ukazatele (takze lze jen na interaval)
- vetsinou vraci iteratory
- `copy_if`
  - vraci pointer do kam se kopirovali prvky

- `remove_if`
  - v prvni casti jsou spravne prvky a po iteratoru se musi smazat
- existuji i falesne iteratory
  - `std::back_inserter`
  - `std::ostream_iterator`
- ruzne algoritmy potrebuji jine typy iteratoru
  - `input`, `output`, `forward`, `bidirectional`, `randomacces`
  - v C++20 jsou concepts, ktere reprezentuji tyto typy
- v C++20 je dvojice iteratoru nahrazena jednim `range`
  - kazdy kontejner uz je `range`, je to i vlastnik dat
  - pak je i `view range`, potom budou jen odkazy na data
  - `all_view(k)`, `iota_view(10, 20)`
  - prevzate z unix pipe `range | filter_view(pred)`

### Funktory

- nekdy algoritmy potrebuji i definovany funktor
- bud pomoci globalni funkce, nebo tridy s operaci ()
- take od C++11 existuje lambda funkce
  - rychlejsi nez globalni funkce, protoze je tam uz rovnou vygenerovane co se deje a ne odkaz na funkci

*Přednáška 11*

---

- predavani parametru do funkci
- bud lze pres funktor, anebo lambda funkci (lambda je pak technicky stejna jako funktor)
  - lambda: `std::for_each(c.begin(), c.end(), [value](double & x){x += value;});`
  - funktor: `std::for_each(c.begin(), c.end(), my_functor(value));`
- funktor modifikujici svuj 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 d = std::for_each(c.begin(), c.end(), my_functor());
    return f.s;
}
```

- nebo se da implementovat pomoci lambdy (technicky se deje neco jineho)

```cpp
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 funkce

- `[capture](params)mutable -> rettype{body}`
- naratovy typ lze bud explicitne rict, nebo automaticky odvodit a jinak to je void
- v C++14 se da lambda deklarovat pomoci typu `auto`, pak se z toho vlastne stava sablonou
  - `[](auto x, auto&& y){}`
- v C++20 se da explicitne rict, ze se jedna o sablonu
  - `[] <typename T, ttpename U>(T x, U && y) {}>`

### Capture

- zpusob zpristupneni vnejsich entit
  - lokalni promenne
  - this
- explicitni capture
  - `[a, &b, c, this](){ return a+c+b[c]+m; }`
  - entity se predavaji primo nebo odkazem
  - `this` umoznuje pristup k polozkam objektu

### Tridy

- `class` a `struct` jsou prakticky stejne. jen class jsou privatni a struct verejne
- tridy se deli na tri stupne konstrukce class
  1. neinstanciovana trida
  2. trida nesouci data
  3. trida s dedicnosti

*Přednáška 12*

---

### Namespace

- oproti class muze byt otevirana a dodefinovana vicekrat

```cpp
namespace x{
    class N{};
    const int c = 0;
};
```

- pokud pri volani funkce z namespacu je jako parametr neco z namespacu, tak neni treba explicitne psat, ze je funkce z daneho namespacu a prekladac ho tam pak bude hledat (**argument-dependet-lookup**)
  - treba `std::cout << std::endl;` se operace `<<` hleda v `std`
- `using namespace x` nebo `using x::t` lze pouzivat veci z namespacu bez explicitniho volani `x::`
  - nepsat v havickovem souboru globalne, hodi se pouzivat treba ve funkci

### Konverze typu

- konverzni konstruktor

```
class T{
    T( U x);
};
```

- definice konverze z `U` do `T`
- lze vynutit aby neslo delat konverzi pomoci `explicit`
- nebo take lze udelat konverzni operace

```
class T {
    operator U() const;
};
```

- konverze z `U` do `T`
  - vraci `U` hodnotou
- lze take provest cast
  - z cecka je `(T)e`
  - obdobna je `T(e)`, ktere vypada jako funkce
- existuji i slozitejsi moznosti (podle sily a nebezpecnosti)
  - `const_cast<T>(e)` - narusovani `const`
  - `static_cast<T>(e)` - bezne konverze
  - `reinterpret_cast<T>(e)` - low-level cunarny
- novy run-timovy testovani
  - `dynamic_cast<T>(e)`

### Dedicnost

```
class Base {};
class Derived : public Base {};
```

- je treba psat virtualni destructor `virtual ~Base () noexpect{}`
- klicove slovo `final` uz uzavre delani dalsich potomku
- pouzitim `override` se testuje jestli virtualni funkce predak existuje
- preddevsim proc se to vyuziva je pri pouzivanim ukazatelu a nebo referenci
- **slicing** je kopie jen casti objektu a tedy pokud vytvorim predka z potomka, tak je to jen podcast kde je predek
- lze udelat i vicenasobnou dedicnost, ale to muze vest k problemum

### Abstraktni trida

- po pouziti jakekoliv funkce jako `virtual void function() = 0;`
- dodava typovou informaci

  *Kosoctverec je vzdy problemovy a obvzlast v programovani.*

### Virtualni dedicnost

```
class S{};
class R : public virtual S{};
class W : public virtual S{};
class M : public virtual W, public virtual R{};
```

- aby se dalo provadet vicenasobnou dedicnost bez problemu
- mam pak vice typovych informaci
- vlastne tam jsou dve tridy vedle sebe a pamatuje se offset kde jsou nasledne tridy
- trida `M` se pa kda pouzit jako `W`, nebo `R` a staci jen jeden objekt (takze vlastne interface)

### Pouziti dedicnosti

- **IS-A** hierarchie
  - napr. Zivocich - savec - pes - jezevcik
- **interface-implementace**
  - co ten objekt ma umet
  - spojovani dohromady je jako mnozinove sjednoceni
- byva i kombinace obou dvou, aby implementace sly drzet pomoci jednoho typu objektu

*Přednáška 13*

---

### Typ `std::variant`

- kontejner obsahujici jeden z ruznych typu (fixovane)
- hodi se pouzivat pokud jsou podobne velke objekty

```
using VT = std::variant< T0, T1, T2>;

T0 v0 = /*...*/;
VT a = v0;
VT b(std::in_place_type<T1>, /*...*/);
VT c(std::in_place_index<2>, /*...*/);
c.emplace<T1>(/*...*/); // Also calls T2::~T2.

void action(VT & vo){
    switch(vo.index()){
```

```
        case 0:
            /*...*/
        case 1:
            /*...*/
        case 2:
            /*...*/
    }
}
```

- v bloku jsou vlastne vsechny data pro mozne typy a taky misto na index
- vzdy se pouzije maximalni prostor a lze pouzit jen pro danou velikosti poctu typu(ne velke)
- take lze pouzit polymorfni funktor

```
// Or make it to be a template.
struct VisitorA {
    void operator()(T0 & x) {/*...*/}
    void operator()(T1 & x) {/*...*/}
    void operator()(T2 & x) {/*...*/}
};

void action(VT & vo){
    VisitorA va;
    std::visit(va, vo);
}
```

- pokud se udela jako sablona potom potrebuji typy spolecny interface

## Vyjimky

- pokud se nedeje to co se ma
- jak opustit funkci jinudy nez normalni cestou
- **throw** statement a **try-catch** blok
  - pokud volam funkci v `try` bloku, tak musi mit uvnitr `throw` a to pak vyhodi do `catch` bloku
- ve vyjimce je parametr trida
- taky existuje univerzalni catch blok `catch(...){}`
- hodnota exception se nekde uchova
- **Stack-unwinding**
  - vyskakuje se s vyjimkou dokud se nenajde catch blok pro danou vyjimku
  - behem skoku se smaze spousta lokalnich promennych (volaji se destruktory)
  - potom se muze objevit dalsi vyjimka
- take existuji jiny zpusoby (pokud je vyuzivane vice vlaken)

```
std::exception_ptr
std::current_exception()
```

`std::rethrow_exception(p)`

- je dobre vyjimky pouzivat jen obcas (je to run-timovy a taky pomaly)
- pokud funkce nemuze vyhodit vyjimku, tak lze napsat `noexcept`
- existuje standardni trida vyjimek `<stdexcept>` a ma funkci `what()`
  - pak to jsou `std::exception` (treba `std::logic_error`)
- lze si napsat i svoje tridy vyjimek, ale je dobre vychazet ze standardnich
- **hard errors** jako pristup do spatne pameti, deleni nulou
  - to se prerusi v procesoru a OS na to reaguje
- destruktory nemuhou skoncit vyjimkou (defaultne jsou `noexcept`)
- kompilatory samy osetruji jisty vyjimky

### Rady pro exception

- globalni catch blok

```
#includ <iostream>

int main (int argc, char ** argv){
    try{
        // Code.
    } catch (...){
        std::cout << "Unknown exception caught" << std::endl;
        return -1;
    }
    return 0;
```

- **Exception neutrality** nesmi se zatajovat vyjimky i kdyz nevim jaka to je
  - taky neni nutne zabit cely program (napr. server)