16. Algoritmizace: vlastnosti algoritmu, způsoby zápisu algoritmu, časová a paměťová složitost

Algoritmus = jednoznačný postup řešení problému splňující následující kritéria

- ~~**Elementárnost** řešení je vyjádřeno v jednoznačných realizovatelných krocích~~ (pouze v imperativním paradigmatu! Algoritmus nemusí být nutně posloupnost kroků, v objektovém paradigmatu např. popis všech událostí.) Vyplývá z úplnosti a jednoznačnosti
- ~~**Determinovanost** v každém kroku je definováno, kterým krokem pokračovat~~ algoritmy mohou pracovat s náhodnými čísly, nedeterministický stavový automat reprezentující algoritmus lze převést na deterministický
- **Úplnost** algoritmus popisuje všechny stavy, které mohou nastat (*well-defined*)
- **Konečnost** algoritmus ~~skončí po konečném počtu kroků~~ může obsahovat nekonečný cyklus, je ale popsán konečným počtem stavů (*finite sequence of instructions*)
- **Jednoznačnost** (*unambiguous*) symboly či stavy a jejich přechody, kterými je algoritmus vyjádřen, musí být jednoznačně interpretovány

Zcestně je algoritmus popsán na české wikipedii, dobře je algoritmus popsán na anglické wikipedii. Škrtnuté věci jsou špatně pochopené věci na české wikipedii učené teoretiky na českých školách. Kurzívou jsou uvedena anglická kritéria algoritmu.

Příklad

```
Načti x // není popsáno, co znamená "načti", není elementární
Ošetři hodnoty x // není popsáno jak, není jednoznačné
Dokud nevypneme počítač: // potenciálně nekonečné, v pořádku
  Nastav x na náhodné číslo 1-10 // není deterministické, v pořádku
  Zvětši y o 1 // není popsána původní hodnota y, není úplné
  Pokud je y>25, nastav y na 1
  Blikni diodou na pozici [x,y]
```

Způsoby zápisu algoritmu

- **Pseudokód** = popis v přirozeném jazyce, krok na řádek
- **Program** = algoritmus zapsaný v programovacím jazyce (obvykle skriptem)
- **Stavový diagram**

Časová složitost = počet kroků nutných k vykonání algoritmu v závislosti na velikosti vstupu **n**

Příklad: nalezení pozice prvku v poli

```
Vstup: pole p, číslo x
Nastav i na 0
Dokud je i menší než délka p:
  Pokud je p[i]=x vrať i
Vrať -1 // nenalezeno
```

Nejlepší případ: 3 kroky (i=0, test i<délka p, vrácení hodnoty)
Nejhorší případ: 2n+1 kroků (při nalezení na konci pole 2x testování: i<p, p[i]=x, vrácení -1)
Průměrný případ: n+1 kroků (nalezení uprostřed pole)

Třída časové složitosti = vynechání konstant v časové složitosti (nemění příliš řádově výsledný počet kroků, dobré k odhadu):

- Konstantní (počet kroků nezávisí na délce vstupu)
- Lineární (n) cyklus
- n*log(n) (logaritmická = cesta stromovou strukturou od kořene k listu)
- Kvadratická (n*n), obecně polynomiální (n^k) vnořené cykly
- NP (non-polynomial) horší než polynomiální (např. exponenciální k^n, prakticky neřešitelná pro n větší než 100)

Příklad exponenciální časové složitosti

```
funkce NP(n)
  pokud je n≤1, vrať 1
  vrať NP(n-1)+NP(n-1)
```

Je to rekurzivní funkce: počet zanoření je lineárně závislý na `n`, při prvním zanoření se volá funkce 2x, při druhém kroku 2x2 = 4, při třetím 8, obecně při ntém 2^n. Při velkém `n` je to prakticky exaktně neřešitelný problém, který však můžeme zapsat ekvivalentně jako:

```
funkce P(n)
  pokud je n≤1, vrať 1
  vrať 2x P(n-1)
```

Zde je počet zanoření `n`, třída složitosti je lineární, funkce vrátí výsledek velmi rychle i pro velmi velká `n`, stejný, jako by vrátila funkce `NP`.

Asymptotická časová složitost (většinou určována pro průměrný případ, značí se velkým Omikron – O): nejmenší třída časové složitosti, pro kterou od určitého **n** výše platí, že O(n) je větší než počet kroků algoritmu v nejhorším případě. Například O(n*log(n)) znamená, že algoritmus skončí řádově zhruba stejně jako např. při 230n*log(17n) + 560. Např. složitost třídy 3*n*n+2, má od určitého **n** řádově mnohem více kroků než předchozí případ. Máme-li tedy velký vstup, na konstantách obvykle nezáleží a roli hraje pouze funkce n.

Paměťová časová složitost je totéž, ale sledovaná veličina je spotřebovaná paměť místo počtu diskrétních kroků.

17. HTML: srovnání s XML, hlavička a tělo dokumentu, blokový a řádkový element, sémantické tagy, formuláře

HTML – hypertext markup language, jazyk původně určený pro psaní textu s odkazy
XML – eXtensible markup language, jazyk pro formátování dat
XHTML – HTML kompatibilní s XML (např. zákaz nepárových tagů), nepoužívá se
HTML 5 – pravidla pro automatické doplňování tagů, důraz na sémantické tagy

Formátovací tagy převážně zastaralé (krom b, i), definují vzhled (přesunuto do CSS)
Sémantické tagy určují význam v dokumentu (např. main, article, span)

Tělo dokumentu definuje obsah (uvnitř tagu body), **hlavička dokumentu** informace (znaková sada, autor, kešování, styly ap., uvnitř tagu head), následuje před tělem

Tělo i hlavička je uzavřeno v tagu html. HTML5 automaticky doplňuje html, head i body tak, aby hlavička byla před tělem. Použitý standard (deklarace dokumentu) se musí nacházet úplně na začátku a pro HTML 5 vypadá jako

```
<!DOCTYPE html>
```

Řádkový element je součástí textu, nemá rozměry, okraje ani pozicování
Blokový element je mimo text, má rozměry, okraje a pozicování

18. CSS: selektory, pseudotřídy, výběr dle atributu, priorita pravidel, boxing model

CSS (*cascading style sheet*) jsou formátovací pravidla pro HTML dokument umožňující týž kaskádový zápis, jako je kaskádová struktura HTML stromu, např.:

```
div b { color: green; } /* pro všechny tagy b uvnitř div */
b { color: red; } /* pro všechny tagy b */
div b i { color: #222; } /* pro vschny tagy i uvnitř b, který je uvnitř div
```

V případě kolidujících pravidel pozdější přepíše dřívější, ale specifičtější má přednost před méně specifickým. Uvedený příklad proto `<div><b>test</b></div>` zobrazí zeleně.

Pravidla se zapisují ve formátu (zapamatujte si tyto termíny): `selektor { vlastnost: hodnota; vlastnost: hodnota; }`. Krom kaskádového selektoru se používají nejčastěji následující:

- `div > b` pro tag b, který je přímým potomkem div. neaplikuje se tedy např. na značky `<div><i><b>test</b></i></div>`.
- `div ~ b` pro tag b, který je následuje na stejné úrovni po div
- `div + b` pro tag b, který následuje bezprostředně po div
- `.test` pro libovolný tag, který obsahuje třídu (atribut class) `test`
- `.test1.test2` pro libovolný tag obsahující třídy `test1` a `test2`
- `#test` pro libovolný tag s atributem id rovným `test`
- `[atribut=hodnota]` pro libovolný tag, jehož atribut je roven uvedené hodnotě

Pseudotřída je selektor, který se chová jako automaticky doplněná třída, zapisuje se dvojtečkou, např.

- `a:hover` elementy `a`, nad kterými je umístěn kurzor
- `input:not([disabled])` všechna políčka input, která nemají nastaven atribut disabled
- `li:nth-child(2n+1)` liché položky seznamu

Pseudoelement je automaticky doplněný element před (`::before`) či za (`::after`) nějaký prvek. Následující kód přidá české apostrofy kolem tagu i:

```
i::before { content: "„"; }
i::after { content: "“"; }
```

Boxing model je způsob interpretace rozměrů. Rozměry mají pouze blokové elementy.

- `span { width: 200px; }` nevalidní CSS, span je řádkový element, nemůže mít rozměry
- `span { display: block; width: 200px;}` validní CSS, ale nevalidní HTML: není definováno, jak se má zobrazit např. `<b>hello<span>world</span></b>`, když HTML definuje, že `b` může obsahovat pouze řádkové elementy.

Ve výchozím stavu je do rozměru elementu započítán pouze obsah, tj bez vnitřního okraje (padding) a ohraničení (border). Tj. např. `<body><div style="border: 1px solid black; width: 100%"></div></body>` je div široký 100% šířky body plus 2 pixely pro ohraničení zleva a zprava, zobrazí se tedy posuvník. Při nastavení `box-sizing: border-box` se rozměr aplikuje na celý element (vč. padding a border), tj. pokud tuto vlastnost do uvedeného příkladu přidáme, posuvník se nezobrazí.

19. Javascript: funkce, objekty, prototypy, HTML DOM, kontext a rozsah platnosti proměnné, datové typy

Javascript je skriptovací jazyk, což je interpretovaný jazyk, který určuje typy proměnných automaticky a automaticky je převádí. Na logickou hodnotu `false` jsou převedeny `0`, `""`, `[]`, `{}`, `null`, `undefined`, `NaN`. (operátor porovnávání s `NaN` vrací vždy false bez ohledu na hodnotu druhého argumentu) Tyto hodnoty jsou označovány jako *falsy values*.

Funkce je pojmenovaný kód s parametry vracející hodnotu příkazem return, který ukončí funkci v místě volání. Neobsahuje-li funkce tento příkaz, je vrácená hodnota rovna konstantě `undefined`.

Funkce jsou v javascriptu často používány jako parametry. Takovým funkcím se říká *callback*. Callback funkce jsou vyžadovány pro obsluhu událostí (*event handlers*), ve svém parametru dostávají objekt události, který obsahuje informace o pozice myši, stisknuté klávese a další, podle typu události.

Objekty jsou složené proměnné, které mají svůj *kontext* (ukazatel `this`). Každá proměnná má svůj *rozsah platnosti* (*scope*) ve kterém je viditelná. Objekt v javascriptu vznikne např. aplikováním operátoru `new` na funkci.

```
function test() { alert(this); }

var x = test(); // [Object window, x je undefined]
var x = new test(); // [Object object, x obsahuje objekt]
```

V prvním případě je kontext objekt, ve kterém je funkce volána (při testování v prohlížeči window. V druhém případě se vytvoří nový kontext, který je funkcí vrácen (pokud neobsahuje return). Funkce, na které aplikujeme `new`, nazýváme *konstruktory*.

Objekt můžeme také vytvořit pomocí literálu `{}`, tentýž objekt jako výše uvedený lze také vytvořit takto:

```
var x = {
  test: function() { var a; alert(this); }
}
```

Pokud javascript voláme v prohlížeči na stránce obsahující HTML, prohlížeč automaticky vytvoří objektovou strukturu této stránky popsanou jako DOM (*Document Object Model*) a úpravou vlastností těchto objektů můžeme manipulovat s obsahem stránky. Tato objektová struktura je uložena v objektu `document`. Následující kód přidá do stránky funkčnost, která kliknutím na libovolný element zobrazí hlášku s jeho `id` atributem:

```
Array.from(document.querySelectorAll("*")).forEach(function(e) {
  alert(e.target.id);
});
```

Javascript je objektový jazyk, který ale (vyjma posledního rozšíření) neobsahuje třídy, pouze objekty. Každý objekt obsahuje speciální ukazatel `prototype` obsahující kontext jeho konstruktoru.

```
function test() { }
var x = new test();

// přidá funkci objektu x
x.a = function() { alert("a"); }

// přidá funkci všem objektům s kontextem test
test.prototype.b = function() { alert("b"); }

var y = new test(); // obsahuje funkci b, x obsahuje funkce a,b
y.b();
```

20. PHP: proměnné, funkce, třídy a objekty, zpracování formulářů, session

PHP je skriptovací jazyk (definici viz v předchozí otázce) na serveru. Narozdíl od javascriptu není prototypální, ale obsahuje třídy. Třída je datový typ, ze kterého odvozujeme objekty. Funkcím objektu zde říkáme metody. Krom objektových metod máme též metody *statické*, které jsou volány na třídě, ne na objektu. Nemají tedy vlastní kontext a chovají se jako funkce ve jmenném prostoru třídy. Do jednou vytvořených objektů již nelze přidávat nové metody (existující třídy lze rozšířit děděním), což lze obejít *magickými metodami* (terminus technicus) `__call` a `__callStatic`, které jsou volány v případě, že nebyla nalezena metoda v objektu resp. třídě. Název volané metody je jejich prvním parametrem, druhý parametr je pole obsahující parametry původní metody:

```
class Test {
  static function __callStatic($method, $args) {
    echo "volal jsi metodu $method s parametry ".implode(",", $args);
  }
}

Test::hello("a", 3.14);
```

PHP proměnné se deklarují znakem `$`, jejich rozsah platnosti je blok kódu s místem jejich deklarace. Lze převzít proměnnou z globálního prostoru operátorem `global`.

```
$answer = 42;

function test() {
  global $answer; // převezmi proměnnou $answer z globálního kontextu
  echo $answer;
}
```

Krom toho PHP definuje tzv. *superglobální proměnné* (terminus technicus), které mají vždy globální rozsah platnosti (jsou tedy viditelné odevšad). Tyto jsou:

- `$GLOBALS` pole obsahující globální proměnné
- `$_GET`, `$_POST` proměnné poslané HTTP metodou GET resp. POST
- `$_FILES` proměnné obsahující data souborů poslaných metodou POST
- `$_COOKIE` cookie proměnné (musí být nastaveny před HTML výstupem)
- `$_SESSION` proměnné relace

Proměnné relace mají rozsah platnosti na klientovi, který volal PHP skript. Server vytvoří unikátní identifikátor, který uloží jako cookie proměnnou na klientovi do souborů v domovském adresáři uživatele. Pod tímto identifikátorem též vytvoří asociativní pole. Při HTTP požadavku si pak přečte hodnotu cookie na klientovi a zpřístupní mu pole pod tímto identifikátorem.

Bezpečnostní problém je, že ke cookies se dostane též javascript běžící na klientovi. Pokud si útočník zajistil přístup k dokumentům uživatele (např. SSH nebo pokud uživatel na chvilku odběhl od počítače), může si hodnotu session ID přečíst a nastavit si ji na vlastním stroji. Pokud je pak klient na serveru přihlášen do privilegované sekce a je to uloženo v session, útočník se tam pak dostane také ze svého stroje. Počítejte tedy, že pokud se někde přihlašujete bez TLS, pokud si někdo dá tu práci, pak jsou všechna vaše data veřejná. TLS lze obejít na napadených routerech, čímž jsou nechvalně známy vlády Číny, USA, Ruska, Íránu, Turecka a pravděpodobně všech, kde mají vlády přístup k superpočítačům. Obranou je VPN (např. NordVPN)

Zpracování html formuláře:

```
// formulář u klienta
<form method="POST">
  <input type="password" name="pass">
  <input type="submit"
</form>
```

```
// kód na serveru
<?
  session_start(); // uloží cookie PSPSESSID na klientovi

  // aby nebylo heslo uloženo ve zdrojovém kódu
  if(md5($_POST["pass"])=="2a1ecd4376fa1b0decf6248351d2462b") {
    $_SESSION["admin"] = true;
  }

  // při každém načtení stránky až do odhlášení
  if($ SESSION["admin"]) {
    echo "Tajná data"; // veřejná bez TLS, bez VPN veřejná vládám velmocí
  }
?>
```

21. C: ukazatelé, statická, dynamická a automatická paměť, datové typy

C je kompilovaný nízkoúrovňový jazyk. Každá proměnná má svou adresu (místo, kde je uložena) a hodnotu (obsah paměti na dané adrese). Datový typ určuje, kolik bytů proměnná zabírá a jak se její obsah interpretuje. Základní datové typy jsou:

- `char` (1B) pořadové číslo znaku ASCII
- `int` (alespoň 2B, dle šířky datové sběrnice) znaménkové celé číslo pro aritmetické operace s ALU
- `float` (4B) plovoucí desetinná čárka dle standardu IEEE 754
- `double` (8B) plovoucí desetinná čárka dle standardu IEEE 754
- `size_t` (alespoň 2B, dle šířky paměťové sběrnice) bezznaménkové celé číslo rozsahem dostatečné pro libovolnou adresaci na dané architektuře

Všechny základní datové typy jsou hodnotové, proměnná obsahuje hodnotu dle popisu výše. Tímto způsobem ale nemůžeme uložit text. Ten se ukládá do souvislé paměti typu char ukončené hodnotou 0 (neplést si se znakem '0', který má hodnotu 48). Textová proměnná je tak reprezentovaná **ukazatelem** na první znak. Ze základních datových typů můžeme udělat ukazatel na typ přidáním * (např. `char*`). Ukazatel neobsahuje hodnotu, ale adresu. Můžeme též zjistit adresu hodnotových proměnných a hodnotu ukazatelů dle následující tabulky:

typ	deklarace	hodnota	adresa
hodnotový	`char x;`	`x`	`&x`
ukazatel	`char* y;` či `char *y;`	`*y`	`y`

Pozor na dvojí význam `*y` (deklarace vs. získání hodnoty).

Pole se chovají jako konstantní ukazatel, tj. ukazatel, jehož **adresu** nelze změnit. Nezaměňujte s ukazatelem na konstantní paměť, tj. jehož **hodnotu** nelze změnit.

- `const char * x` je ukazatel na konstantní paměť
- `char * const y` či `char y[80]` jsou konstantní ukazatele

Pozor na různé významy definice řetězců:

```
char* x = "hello"; // chyba, "hello" je konstanta, char* je ukazatel na
nekonstantní paměť
const char* y = "hello";
char z[] = "hello"; // není nutné uvádět velikost
y[0] = 'H'; // chyba, paměť je konstantní
z[0] = 'H'; // v pořádku, pole je konstantní ukazatel na nekonst. paměť
```

statická paměť jsou globální proměnné a proměnné uvozené slovem `static`. Jsou vynulovány na začátku programu a do konce programu nemohou být uvolněny.

automatická paměť (též zásobníková) je alokovaná (a nevynulovaná) při začátku bloku a uvolněná (automaticky) při jeho skončení. Jsou to parametry funkce a lokální proměnné.

dynamická paměť je alokovaná pomocí `new` či `malloc` a uvolněna pomocí `delete` či `free`. Pokud na tuto paměť ukazuje zásobníková proměnná, která je uvolněna před uvolněním dynamické paměti, je tato paměť nepřístupná po zbytek programu (únik paměti, *memory leak*). Příklad

```
void test() {
  int* data = new int[20]; // zásobníková proměnná data ukazuje na
                           // dynamickou paměť
}
test(); // proměná data automaticky uvolněna
        // na dynamickou paměť nic neukazuje a nemůže být uvolněna
```

22. C++: konstruktor (výchozí, implicitní, obecný), výchozí parametry, dynamické objekty, virtuální metody, druhy dědičnosti

Konstruktor v C++ je metoda bez jména, která vrací objekt třídy, v níž je definovaná. Pokud konstruktor nenapíšeme, kompilátor doplní **implicitní** konstruktor, který zavolá výchozí konstruktor rodičovských tříd. **Výchozí** (*default*) konstruktor je takový, který může být volán bez parametrů (tj. nemá parametry nebo jsou všechny parametry výchozí). Pokud nedodáme vlastní, kompilátor vytvoří též **implicitní kopírovací konstruktor**, který očekává jako parametr referenci téže třídy a provede mělkou kopii všech jejích členů. Aby toho nebylo málo, existue též **implicitní volání konstruktoru**. Vše objasní příklad:

```
struct Test() {
  int i;
  Test(int i, bool b=true) : i(i) { } //konstruktor s výchozím parametrem b
  // kompilátor nedoplní implicitní výchozí konstruktor
  // kompilátor doplní implicitní kopírovací konstruktor
};

Test t1; // volání chybějícího výchozího konstruktoru, chyba
Test t2 = Test(2); // explicitní volání konstruktoru
Test t3(3); // totéž, zkrácený zápis
Test t4 = 4; // implicitní volání konstruktoru
Test t5(t4); // volání kopírovacího konstruktoru

struct Test1 : Test {
  Test1(int i=0) : Test(i) { } // výchozí konstruktor
};
```

```
Test1 u; // v pořádku, volání výchozího konstruktoru
Test1 u1 = Test1(); // totéž
Test1 u2(); // zákeřnost: deklarace funkce u2 vracející objekt Test1
Test1* u3 = new Test1(); // dynamický objekt
```

Virtuální metoda je taková, která umožňuje **dynamicky vytvořenému** potomkovi přepsat definici funkce:

```
struct Computer {
  void meaning() { printf("The answer is %d.", getAnswer()); }
  virtual int getAnswer() { return 0; }
};

struct Earth : Computer {
  int getAnswer() { return 42; }
};

int main() {
  Computer c = Earth();
  c.meaning(); // The answer is 0.

  Computer* c1 = new Earth();
  c1->meaning(); // The answer is 42.
  // pokud by getAnswer nebyla virtuální, odpověď by byla 0
}
```

U staticky vytvořeného objektu musí být kód metod znám v době kompilace. U dynamicky vytvořeného objektu se kód metod určí až za běhu. To je základ **polymorfismu**.

Dědičnost znamená specializaci objektu, tj. přidání vlastností. Dle principu Liškovové lze potomka staticky převést na rodiče. **Neveřejná dědičnost** (specialita C++) není dědičností v tomto smyslu, znamená omezení práv přístupu členů rodiče. Vystihoje vazbu *je vytvořen pomocí*.

```
struct Father {
  int i;
  Father() : i(42) { }
};
struct Son : private Father {
  int getI() { return i; }
};

int main() {
  Son bob = Son();
  printf("%d", bob.getI()); // 42

  Father joe = Son(); // chyba, Father je nedostupný rodič
}
```

23. C#: automatická správa paměti, výstupní parametr, proměnná, množství parametrů, properties, projekt, vizuální programování, .NET

Přečtěte si odstavec o dynamické paměti z otázky 21.

U jazyků s automatickou (*managed*) správou paměti. Nedochází k únikům paměti a není třeba dynamickou paměť uvolňovat. Program si udržuje počet referencí na dynamickou

paměť a neuvolňuje ji, dokud to operační systém neuzná za vhodné (dle optimalizace správy prostředků). Poté spustí nástroj zvaný *garbage collector* (*gc*), který projde lokální tabulku deskriptorů a paměť s počtem referencí 0 označí jako volnou k dalšímu použití. Výhodou tohoto přístupu je, že programátor vůbec neřeší uvolňování paměti: volá pouze `new` a nezná `delete`. Nevýhodou je, že uvolňování nemá pod kontrolou (OS může spustit gc v nevhodnou dobu, kdy se program chystá provést výpočetně náročnou operaci).

výstupní parametr je v C# označen klíčovým slovem `ref` nebo `out`. Je to ekvivalent předání pomocí reference v jazyce C++. U `ref` parametrů musí být hodnota před voláním inicializovaná. Používáme, má-li mít funkce více výstupů:

```
void Test(out int result1, out int result2) {
  result1 = 10;
  result2 = 20;
}

int a, b;
Test(a, b);
Console.WriteLine("{0} {1}", a, b); // 10 20
```

V C může mít funkce libovolné množství parametrů: makra knihovny <cstdarg> umožňují pracovat přímo se zásobníkem funkce. C# je vysokoúrovňový jazyk a ekvivalentní funkčnosti dosahuje zabalením parametrů do pole. Takto zabalený parametr musí být poslední a je označen slovem `params`.

```
int Sum(params int[] data) {
  int result = 0;
  for(int i=0; i<data.Length; ++i) result+= data[i];
  return result;
}

int x = Sum(5, 8, 3, 1, 6); // x = 23
```

properties (vlastnosti) umožňují zavolat kód při čtení a zápisu proměnné. To umožňuje implementovat veřejné datové položky třídy při zachování zásady **zapouzdření**. V C# se implementuje pomocí bloků get (volá se při čtení) a set (při zápisu):

```
private int  percent;
public int percent {
  set { _percent = value; }
  get { return _percent<100 ? _percent : 100; }
};

percent = 80;
percent+= 30; Console.WriteLine(percent); // 100
percent-= 20; Console.WriteLine(percent); // 90
```

Vlastnosti se kompilují jinak než obyčejné datové položky, veřejné členy třídy je tedy vhodné psát jako **automaticky implementované vlastnosti**:

```
public int myProperty { get; set; };
```

Podrobnosti zde. O vizuálním programování viz otázku 26.

24. Podmínky a cykly: while, for v různých jazycích, rozsah platnosti (scope), iterátory; if, konverze na logický typ, operátory

Podmínky jsou jazykové konstrukty pro větvení programu: `if` a konverze na logický typ viz *falsy values* v otázce č. 19, vše ostatní se převede na true. `switch` dělí program na více větví: každou část je nutno ukončit příkazem `break`, jinak propadává níže. Nepovinná část `default` se použije, pokud nedošlo na jinou:

```
switch(c) {
  case 'a': case 'e': case 'i': case 'o': case 'u': case 'y':
    puts("samohlaska");
  break;
  case 'r': case 'l':
    printf("obojetna "); // intentional fall-through
  default:
    puts("souhlaska");
}
```

Cykly mohou obsahovat příkazy `break` (ukonči cyklus) a `continue` (pokračuj další iterací), což jsou v podstatě formy `goto`. Většina jazyků obsahuje příkaz `for(init; test; step)`, kde část `init` se provede na začátku, `test` je podmínka, která při splnění provádí tělo, `step` je příkaz provedený na konci těla. Typicky `for(int i=0; i<10; ++i)`, ale v C často také pomocí ukazatelů na průchod řetězce `for(const char* c=str; *c!=0; ++c)`.

Iterátor je objekt obsahující pole a ekvivalenty metod `init` a `test` pro procházení onoho pole, případně ekvivalent funkce `foreach` procházející všechny prvky onoho pole:

c++
```
vector<int> data = {4, 6, 1};
for(auto i=data.begin(); i!=data.end(); ++i) cout << *i;
for(auto& i: data) cout << i; // varianta foreach
```

php
```
$data = [4, 6, 1];
for(reset($data); current($data)!==false; next($data)) echo current($data);
foreach($data as $i) echo $i;
```

javascript
```
var data = [4, 6, 1];
for(let i=0; i<data.length; ++i) console.log(data[i]);
data.foreach( x => console.log(x) );
```

Rozsah platnosti viz též otázka 19. Některé jazyky (PHP) mají rozsah platnosti funkční blok, jiné (C++) jakýkoliv blok. Javascript má rozsah platnosti funkční blok pro proměnné deklarované `var` a jakýkoliv blok pro proměnné deklarované `let`.

Operátory dle *arity* lze dělit na unární, binární, ternární (jeden, dva, tři argumenty), dle způsobu zápisu prefixové např. +(1,2), infixové (např. 1+2), vzácněji postfixové: (1,2)+ (např. negace je zvykem psát prefixově, součet infixově, faktoriál postfixově), dle funkce pak aritmetické, logické, bitové (dle jazyka i další druhy, více zde.

25. OOP: rozdíly oproti procedurálnímu paradigmatu, návrh objektů, zapouzdření, polymorfismus

Paradigma je přístup k řešení problému. Nejstarší programovací paradigma, **procedurální**, je pohled na program jako posloupnost příkazů, která se vykoná metodou shora dolů (až na cykly). To odpovídá např. PHP programu pro vygenerování HTML stránky, který má jasně definovaný výstup. Jsou ale i jiné programy (např. interaktivní chat), které jsou potenciálně nekonečné a jejichž výstup nelze odvodit z počátečního vstupu – místo toho program neustále reaguje na události. Zde je vhodné **objektové** paradigma, podle kterého je program popis interakcí mezi objekty, přičemž objekt je do bloku uzavřená množina proměnných a operací s nimi. Programování pak není tvorba posloupnosti příkazů jako v procedurálním paradigmatu, ale rozdělení programu na objekty a popis jejich interakcí (událostí). Není zde jedna posloupnost, ale řada krátkých posloupností, které se přepínají. Jsou i jiná paradigmata, např. **funkcionální**, kde je na program nahlíženo jako na zpřesňování funkce (pojem příkaz zde ani neexistuje; obtížně zapsatelné pro člověka, ale dobře optimalizovatelné pro kompilátor, jazyky Haskell, Lisp), dále se používá **logické** paradigma, které na program nahlíží jako na úplný popis pravidel pozměňující vstupní data (např. SQL, Prolog).

Jsou jazyky ryze objektové (C#, Java), které neumožňují zápis globální funkce – vše je objekt. Pak jsou jazyky multiparadigmatické, které umožňují více paradigmat (C++ procedurální a objektové, C# objektové a logické – rozšíření Linq). Pak jsou jazyky, které syntakticky podporují objektové paradigma (např. PHP podporuje tvorbu objektů pomocí tříd), ale *sémanticky* je program procedurální (objekt v PHP slouží jako jmenný prostor či složená proměnná, nepopisuje ale události).

Objektové paradigma lze dále rozdělit podle způsobu tvorby objektů: založené na třídách (*class-based*), kde třída je šablona ro tvorbu objektu; a prototypální, kde objekty vznikají odvozením z jiného objektu. Prototypální paradigma viz otázku 19, založené na třídách otázku 22, kde též viz zapouzdření a polymorfismus. Je důležité následovat nativní paradigma daného jazyka: pokusy o implementaci jiného (např. v javascriptu pokus o funkcionální paradigma v jQuery či pokus o class-based paradigma v Sencha) vede k neefektivnímu kódu, který obtížně spolupracuje s klientským kódem (jQuery např. syntakticky nemůže implementovat *live collections*).

Prvním krokem v psaní programu je volba vhodného paradigmatu: jakým způsobem chceme popsat problém. Pokud např. potřebujeme zapsat síťovou komunikaci mezi dvěma servery, můžeme tak rychle a přehledně zapsat pomocí **vizuálního programování** v Node-RED (viz otázka 26). Tutéž úlohu bychom mohli vyřešit v C++, tam bychom ale většinu času strávili hledáním vhodných knihoven a popisu převodu dat mezi protokoly, tedy ne původním problémem: popisem komunikace.

26. Vizuální programování: GUI (popis prvků jako objektů), API, IDE, událostmi řízená aplikace

Vizuální programování je modelování aplikaci pomocí grafických nástrojů a diagramů, které na pozadí generuje odpovídající zdrojový kód. Některé jazyky jsou ryze vizuální (Node-RED, Scratch), kde programátor vůbec nepíše zdrojový kód a program „kreslí“ pomocí grafických nástrojů.

Jiné jazyky vizuální programování používají pouze pro návrh vzhledu aplikace, který umožňuje zasadit jej do kódu pomocí **frameworku**. Příkladem je Unity framework, kde vizuálně pracujeme se *Scénou* a programujeme interakce, .NET framework, kde vizuálně navrhujeme formulář a programujeme události (podobně funguje GTK framework) či Nette Framework pro tvorbu webových stránek.

Framework je definovaná struktura kódu, která předepisuje, kam co zapsat. To je omezení, které je občas na překážku, na oplátku ale framework udělá řadu věcí automaticky. Různé typy programů mají společné věci (hry obsahují scénu a hráčem ovládané objekty, webové stránky obsahují formuláře, GUI aplikace jsou v okně). Pokud tvoříme jednu typovou aplikaci za druhou, framework ušetří spoustu času. Pokud tvoříme unikátní aplikaci, je dobré se frameworkem pouze inspirovat a udělat vlastní návrh.

GUI (*Graphical User Interface*) je častou součástí frameworků. Grafické prvky jsou často provázány s operačním systémem a často je k jejich vykreslování zapotřebí práce s grafickou kartou. Pokud ale program má řešit zpracování uživatelských dat a GUI slouží k tomu, aby je mohl pohodlně zadat, je tvorba vlastního GUI odbočení od cílů programu a je vhodné použít framework s GUI.

API (*Application Programming Interface*) je sada knihoven, které slouží jako nástroje komunikace s frameworkem, který využíváme. Windows má pro tvorbu aplikací WinAPI, webové aplikace často definují REST API.

IDE (*Integrated Development Environment*) je vývojové prostředí, které integruje více nástrojů. Nejčastěji framework, GUI, API, ladící (*debug*) nástroje a zvýrazňovač syntaxe. IDE často předgeneruje mnoho kódu, programátor pak píše pouze **události**: kód, který je spuštěn v okamžiku, kdy se program dostane do požadovaného stavu (kliknutí na tlačítko, dokončení nahrávání souboru, přesun okna a podobně).

27. Kompilovaný a interpretovaný program: popis, přenositelnost, vykonávání programu, příklady jazyků; kompilační proces, skriptovací jazyk

Kompilovaný program je takový, kde se zdrojový kód převádí na spustitelný **objektový** kód (nezaměňujte s objektovým programováním). Spustitelný na dané *architektuře* (HW+OS). Objektový kód se liší od strojového v tom, že nepoužívá absolutní adresaci a privilegované instrukce. Místo toho je vsazen do kontejneru, který poskytuje OS informace o HW prostředcích, které program požaduje. Pokud kompilujeme program na architekturu bez OS

(typicky mikrokontrolery), zdrojový kód se převádí na strojový kód. OS naopak může poskytnout službu sdílení zkompilovaného kódu v dynamických knihovnách, díky nimž mohou programy kód sdílet. OS také přiděluje prostředky (zejména paměť a procesor) tak, aby více programů běželo současně co nejefektivněji.

Interpretovaný program se nepřekládá, zdrojový kód krok po kroku přímo vykonává *interpret*, zkompilovaný program. Javascript je vykonáván prohlížečem, PHP je vykonáváno HTTP serverem. Interpret je třeba zkompilovat pro každou architekturu, kde má interpretovaný program běžet. Díky tomu je zdrojový kód nezávislý na HW, má ale menší možnosti správy prostředků a je o poznání pomalejší než zkompilovaný program.

Skriptovací jazyk je interpretovaný jazyk, který řeší typy proměnných automaticky a automaticky mezi nimi převádí. To umožňuje programátorovi soustředit se na popis vysokoúrovňových věcí, ale také to přináší problémy: `0` je kupříkladu při převedení na logický typ (např. v podmínce `if`) převedena na hodnotu `false`, ale `"0"` (jako řetězec) na hodnotu `true`, jelikož je to neprázdný řetězec.

Mezistupněm jsou programy pro virtuální stroje. Příkladem jsou Java programy, které se kompilují pro *Java Virtual Machine* (JVM), aplikaci, která zpřístupňuje HW prostředky jednotným rozhraním. Díky tomu tentýž zkompilovaný kód může běžet na různých operačních systémech, pro které byl JVM sestaven. Výstupem této kompilace je **bytový** kód, což je objektový kód pro virtuální stroj (řada literatury obojí nazývá objektovým kódem).

Přenositelný je z podstaty každý interpretovaný kód. U kompilovaných programů jsou přenositelné ty, které používají pouze standardní knihovny. Teoreticky je tento standard popsán jako POSIX, prakticky ale Windows není POSIXový systém a některé (není jich tolik) standardy nedodržuje. Příkladem přenositelného kódu je ten, který komunikuje s konzolí na PC jako se standardním výstupem (stdout). Tentýž zdrojový kód lze zkompilovat pro libovolný POSIX systém. Příkladem nepřenositelného kódu je ten, který s konzolí komunikuje jako s hardwarem (tj. přesune se na souřadnice na obrazovce, píše barevné znaky) to POSIX nepopisuje a každý systém to řeší po svém.

28. Modelování a životní cyklus aplikace: fáze životního cyklu, UML diagram užití, tříd, waterfall vs. agilní metodika

Životní cyklus aplikace se skládá minimálně z těchto kroků:

- **analýza** co se vyplatí udělat a jaké prostředky k tomu použít
- **návrh** kdo, kdy a co bude dělat
- **implementace**
- **testování** a nasazení (beta verze, pilotní provoz)

Podcenění toho přístupu se nazývá **extinct by instinct**. Motivací je idea, že skutečná práce je jen ve fázi implementace, a proto ostatní fáze přeskočíme. V průběhu vývoje se pak obvykle

ukáže, že je třeba vše přepsat a udělat znovu či že se vytvořilo něco, co ve skutečnosti není třeba. Přecenění tohoto přístupu se nazývá **analysis paralysis**: analýza se provádi příliš do hloubky a vyjadřuje se k ní příliš mnoho lidí, nakonec se vše zamotá a zkomplikuje natolik, že se neudělá vůbec nic.

Existují dvě základní metodiky tvorby aplikace:

Rigidní nepočítá se změnou v průběhu vývoje a s objevem nečekaných překážek. Fáze vývoje se označí za milníky a jakmile se některá fáze dokončí už se nevrací zpět (případně probíhají dvě fáze současně). Tato metodika je vhodná, pokud při vývoji není třeba dělat nic nového, pokud je zadání přesné a detailní.

Agilní počítá se změnou v průběhu vývoje, počítá, že po dokončení životního cyklu bude třeba další životní cyklus, který odstraní nedostatky. Vývoj pak probíhá v iteracích, přičemž každá iterace se navrhne tak, aby výstupem byla funkční aplikace, která umí něco navíc a aby žádná iterace netrvala příliš dlouho (obvykle max 14 dní). Nultá iterace má minimální funkčnost, často jen grafický návrh. Tato metodika je vhodná, pokud je zadání vágní a směr vývoje nejistý.

Pro analýzu se používají standardizované diagramy UML (*Unified Modelling Language*). Některé znich jsou velmi praktické, jiné téměř zbytečné. Nejčastěji se používají:

UML diagram užití (use-case): systém je ohraničen obdélníkem, vně něhož stojí *aktéři*: obvykle lidé provádějící charakteristické *aktivity*, které jsou zakresleny v podobě elips do obdélníku systému. Mezi aktivitami mohou být vazby, ale důležitým výstupem analýzy je kdo a co bude se systémem dělat (systém je zde množina aplikací přizpůsobená konkrétním aktérům). V návrhové části modelování aplikace se pak programují aplikace splňující tyto aktivity a pokud možno nic jiného (obchodní logika).

UML diagram tříd (class diagram): se používá v návrhové části (obchodní logika) a implementační části (implementační logika, více detailů) pro popis komunikace mezi objekty. Třída se zapisuje do obdélníku vertikálně rozděleného na 3 části: nahoře je název třídy, uprostřed seznam vlastností a dole seznam metod. Privátní členy se uvozují `-`, chráněné `#`, veřejné `+`. Vazby mezi objekty jsou dědičnost (prázdná šipka), agregace (prázdný diamant), kompozice (plný diamant). **Agregace** znamená, že objekt uchovává kolekci jiných objektů, které jsou na něm nezávislé (např. studenti navštěvující kurz: po skončení kurzu nevymažeme studenty ze systému, mohou navštěvovat další kurzy). **Kompozice** znamená také kolekci objektů, ale jejich kontejner je vytváří při svém vzniku a maže při svém zániku (např. formulář obsahující ovládací prvky: po odeslání formuláře můžeme objekt z paměti smazat a s ním i všechny v něm obsažená políčka a tlačítka).

UML stavový diagram (state machine): ilustruje stavy, jimiž objekt může procházet, používá se v testovací fázi. Skládá se z počátečního, případně koncového stavu (plný kruh) a přechodových šipek. Odvozený diagram používá např. Unity framework pro tvorbu animací.

Vývojový diagram (flowchart) není součástí UML, ale používá se pro popis procesů (na české wikipedii je opět špatně, např. šipky vedoucí doprostřed jiné šipky).

29. Vyjímky, ladění, druhy a ošetření chyb: krokování programu v rámci IDE, zápis výjimky, chybový stav, chybová událost

Při běhu programu mohou nastat nežádoucí situace: nevyplněné políčko, nedoručený soubor, hodnota mimo rozsah, výpadek serveru a podobně. Existují standardní způsoby, jak se s nimi vypořádat:

- **chybový stav** (též příznak, *error flag*) používáme tehdy, když obvykle nevadí, ale občas je třeba ho opravit. Například nevyplněné políčko ve formuláři, kde většina políček je nepovinných a jen některá povinná. C++ iostream používá chybový stav pokud je vstup v neočekávaném formátu, např.: `int i; cin >> i; if(cin.fail()) puts("na vstupu není číslo");`.
- **chybová událost** používáme také, když chyba obvykle nevadí a kdy její vyřešení zajistí jednoduchá funkce. V javascriptu je to událost `onerror`.
- **vyjímka** (exception) používají knihovní funkce, když nemají dost informací, jak a na jaké úrovni bude chtít klientský kód chybu zpracovat.

Příklad vyjímky

Představme si, že někdo napsal úžasný javascriptový kód na zpracování formulářových dat:

knihovní kód

```
const ERROR_AGE = 1;
const ERROR_NAME = 2;

function getAge(value) {
  if(value>6 && value<80) return value;
  throw 1;
}
function getName(value) {
  if(!value.trim().length) throw 2;
  return value;
}
```

Věk musí být v rozsahu 6 až 80 a jméno nesmí být prázdné. Tvůrce knihovny neví, jak bude klientský kód chtít reagovat, pokud tomu tak nebude, proto vyhodí vyjímku a o víc se nestará.

klientský HTML

```
<input id="age" placeholder="věk">
<input id="user" placeholder="jméno">
<button id="submit">Test</button>
<div id="result"></div>
```

Bez vyjímek bychom museli testovat každý vstup pomocí `if` a kód by tak narostl na dvojnásobek a byl nepřehledný. Pomocí vyjímek můžeme vše uzavřít do bloku `try`, předpokládat, že vše dopadne dobře a pokud ne, ošetřit to na jediném místě `catch`.

klientský javascript

```
submit.onclick = function() {
  try {
    var myAge = getAge(age.value);
    var myName = getName(user.value);
    result.innerHTML = `Uživatel ${myName} starý ${myAge} let.`;
  }
  catch(e) {
    switch(e) {
      case ERROR_AGE: result.innerHTML = "Věk mimo rozsah 6-80"; break;
      case ERROR_NAME: result.innerHTML = "Chybí jméno"; break;
      default: result.innerHTML = e;
    }
  }
}
```

Vývojová prostředí (IDE) nabízejí často v režimu ladění možnost zarážek (*breakpoints*) a krokování. Při vykonávání kódu se program zastaví na dané zarážce a programátor si může ve vývojovém prostředí prohlédnout aktuální hodnoty proměnných. Dále může program *trasovat*. Nástroji IDE:

- **next step** – posunutí vykonávání programu o jeden příkaz dále
- **step into** – posunutí vykonávání programu dovnitř nadcházející funkce
- **step out** – dokončení prováděné funkce, vynoření a pokračování dalším příkazem
- **continue** – pokračování až do další zarážky
- **stop** – ukončení trasování (obvykle při nalezení zdroje chyby)

30. Architektura aplikace: návrhové vzory, vrstvová aplikace, MVC

Tvoříme-li rozsáhlejší aplikaci nebo systém (více spolupracujících aplikací), je vhodné oddělit vzhled, aplikační logiku a data, aby každou tuto část bylo možné použít nezávisle a pro omezení zavlečení chyb do programu (např. úprava vzhledu garantovaně nezpůsobí chybu vkládání dat).

Starší systémy používaly **třívrstvou architekturu** spočívající v oddělení dat (databáze) vzhledu (šablonovací systém) a aplikační logiky (např. PHP). V tomto systému je obvykle vrstva aplikační logiky výrazně rozsáhlejší než zbytek, populárnější se proto stala architektura MVC: Model – View – Controller.

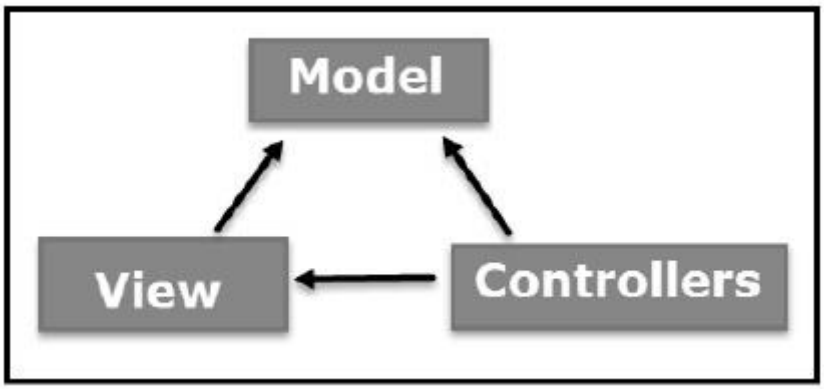
Model je databáze a kód s ní pracující (např. funkce `insertUser(name, email)`). Zbytek aplikace volá pouze funkce modelu a je od databáze odstíněn. Při změně struktury databáze tak stačí provést pouze změnu Modelu.

View je HTML šablona doplněná o klientský javascript. View může volat funkce Modelu (tj. může na něm být závislý), pokud to potřebuje pro aktualizaci svých ovládacích prvků (např. naplnění selectboxu názvy měst v zadaném okrese)

Controller řídí obsahuje aplikační logiku: Pokud uživatel není přihlášen, přesměruje na View s přihlašovací obrazovkou, Pokud uživatel nemá oprávnění, přesměruje na View, kde to

uživatel může nastavit, pokud dokončil nákup, je přesměrován na objednávkový formulář a podobně.

MVC je graficky špatně znázorněný na české i anglické wikipedii. Šipky tam tvoří estetický kruh, ale podstatu zakrývá různými významy šipek. Z pohledu návrhu šipka znamená „závisí na“ a klíčová vlastnost Modelu je, aby nebyl závislý na zbytku aplikace, tj. nesmí z něj vést žádná šipka. Správně je tento obrázek, vše ostatní (zkuste si vygooglit obrázek MVC) je nepochopení této architektury.



Návrhové vzory jsou netriviální návody na zakódování často řešených problémů v objektovém paradigmatu. Jejich popis vychází z dobré praxe většiny zkušených programátorů a usnadňuje komunikaci. Některé z nich mají syntaktickou podporu jazyků, jiné je potřeba zapsat pomocí nástrojů daného jazyka. Pro komunikaci se často používají tyto vzory:

- **async-await** vzor v multivláknové aplikaci, kdy se funkce spustí jako samostatné vlákno (async) a jiná funkce může čekat na dokončení práce tohoto vlákna (await)
- **publisher-subscriber** populární v MQTT protokolu: zdroj událostí (publisher) je posílá jednomu objektu (v MQTT je to broker), který je rozesílá všem objektům, kteří se přihlásili k odběru této události (subsribers)
- **klient-server** je návrh, kdy klient vznáší požadavky a server na ně odpovídá. Rozdíl oproti předchozímu je ten, že server nemůže nikdy iniciativně kontaktovat klienta. Na tomto principu pracuje HTTP.

Původní zmínka o návrhových vzorech pochází od Ericha Gammy z roku 1994, který je dělil na vytvářecí, strukturální, behaviorální a konkurenční. Uveďme si nejjednodušší z každé kategorie, rozvést však můžete libovolný, který je vám bližší (každý programátor má zažitou pouze podmnožinu existujících vzorů, nezažitým vzorům je lépe se vyhnout: jejich špatné použití způsobuje více škody než užitku).

Líná inicializace (vytvářecí) slouží k odložení vytvoření objektu až na dobu jejího prvního použití. Příkladem je navázání spojení s databází, které může být zdlouhavé. Je možné, že

uživatel provede akci, kde spojení s databází nebude potřebovat. Aplikaci proto urychlí, když nevytvoří databázové spojení při spuštění, ale až při prvním databázovém dotazu.

Dekorátor (strukturální) se používá tehdy, když potřebujeme přidat něco k funkčnosti již existující třídy. Využívá se zde dědičnosti a překrytí požadované metody.

Iterátor (behaviorální) viz otázku 24.

Zamykání zdrojů (konkurenční) se používá pro vyřešení situace, kdy souběžně pracující vlákna operují nad stejnými daty a změna pořadí vykonávání vláken může způsobit neplatnost dat. .NET framework za tímto účelem vytvořil příkaz lock, viz též Race hazard.