

3. Imperativní programování, programovací jazyk C, abstraktní datové typy a spojové struktury

vše o C: <https://drive.google.com/drive/folders/0B33G3DM4Z57yWHVGVGdwNVJQTWc?usp=sharing>
Faiglový hľadajte otázky a ďalšie materiály k učeniu: <https://drive.google.com/drive/u/0/folders/0B33G3DM4Z57yck1za1BOT2Z0UIE>

Imperativní programování

Imperativní programování je jedno z programovacích paradigm, neboli způsobů, jak jsou v programovacím jazyku formulována řešení problémů. Imperativní programování popisuje výpočet pomocí posloupnosti příkazů a určuje přesný postup (algoritmus), jak danou úlohu řešit. Program je sadou proměnných, jež v závislosti na vyhodnocení podmínek mění pomocí příkazů svůj stav. Základní metodou imperativního programování je procedurální programování, tyto termíny bývají proto často zaměňovány.

Procedurální programování

Je to poddruh imperativního programování ale tyto názvy se často zaměňují. V procedurálním programování se program skládá z vícero procedur. Je to modernější přístup (vesměs veškeré současné programy jsou procedurální), protože umožňuje přehlednější a čitelnější kód. Druhým podtypem imperativního programování je objektově orientované programování.

Pro procedurální programování jsou typické `for`, `while` a `if` pro imperativní pak spíše `goto`.

Procedural	Object-oriented
procedure	method
record	object
module	class
procedure call	message

Programovací jazyk C

C je nízkoúrovňový, komplikovaný. Je dostatečně mocný na většinu systémového programování (ovládače a jádro OS), přičemž zbytek lze dořešit tzv. inline assemblerem, tedy metodou zápisu assembleru přímo do kódu. Zdrojový kód C je přitom mnohem čitelnější než assembler, je jednodušší ho zapamatovat a navíc je daleko snáze přenositelný na jiné procesory a počítačové architektury. Proto jsou často operační systémy, překladače, knihovny a interprety vysokoúrovňových jazyků implementovány právě v C.

Jazyk C byl vyvinut v Bell Labs mezi lety 1972 a 1973 D. Ritchiem.

Ukládání dat je v C řešeno třemi základními způsoby: statickou alokací paměti (při překladu), automatickou alokací paměti na zásobníku a dynamickou alokací na haldě (heap) pomocí knihovních funkcí. Jazyk disponuje jen minimální abstrakcí nad alokací: s pamětí se pracuje přes ukazatel, který drží

odkaz na paměťový prostor daného typu proměnné, ale je na něm možné provádět aritmetické operace (tyto operace ale neoperují s ukazateli přímo na úrovni jednotlivých bajtů, nýbrž přihlíží k velikosti datového typu, na který ukazují – existují ale také ukazatele typu `void *`, které mohou odkazovat na jakýkoliv typ dat uložený v paměti.).

C - Obecná charakteristika

C je nízkoúrovňový, komplilovaný, relativně minimalistický programovací jazyk.

- Univerzální programovací jazyk nižší až střední úrovně
 - Strukturovaný (funkce + data)
 - Zdrojový kód přenositelný (portable), nutno ctít podmínky přenositelnosti, překladač ne (je závislý na platformě)
 - Rychlý, efektivní, kompaktní kód, mnohdy nepřehledný
- Pružný a výkonný, ale nestabilní
- Podpora
 - konstrukcí jazyka vysoké úrovně (funkce, datové struktury)
 - operací blízkých assembleru (ukazatele, bitové operace,...)
 - Slabá typová kontrola
 - Málo odolný programátorovým chybám
- Dává velkou volnost programátorovi v zápisu programu
 - Výhoda: dobrý programátor vytvoří efektivní, rychlý a kompaktní kód
 - Nevýhoda: špatný nebo unavený programátor, pak nepřehledný program náchylný k chybám
- Nutná vlastní správa paměti
- silná vazba na HW, využití jeho možností (inline assemblér), špatná nebo žádná podpora národních zvyklostí
- Použití:
 - operační systémy
 - řídicí systémy
 - grafika
 - databáze
 - programování ovladačů grafických, zvukových a dalších karet
 - programování vestavěných - embedded systémů
 - číslicové zpracování signálů (DSP)
- Program začíná funkcí `main()`, je základní funkcí programu, každý program musí obsahovat právě jednu tuto funkci
- `char, short, int, long, float, double`
- programový přepínač `switch()`, `case`, `default`, `break`
- C je komplilovaný jazyk
 - Zdrojový kód je nezávislý (portable) na platformě (málo závislý)
 - Spustitelný kód je závislý na platformě
- Vlastní správa paměti
- Definování konstant a makr
- Vytváření strukturovaných proměnných (mohou být i statické)

```
1 int main (int argc, char** argv) {  
2     printf("I-copy-and-paste-all-the-time Policy \n");  
3     return (0);  
4 }  
5 // delší ukázka vseho možného, je syntakticky správné, některé veci přidány jen  
// pro ukazku možnosti...  
6 // cyklus for, stdin, osetrení vstupu, konstanty...
```

```

7 #include <stdio.h> /* hlavičkový soubor, přípona h */
8 #include <stdlib.h>
9 #include "konstanty.h" // uživatelské soubory se vkládají takto
10 #define NASOBITEL 5 /* symbolicka konstanta */
11 #include <math.h> // obsahuje funkce jako sqrt()...
12 // Zpracovani posloupnosti
13 // argc počet parametrů na příkazovém řádku
14 // *argv[] pole ukazatelů na příkazy příkazového řádku
15 int main(int argc, char** argv) {
16     int i, suma, dalsi;
17     const double PI = 3.14; // unused, jen ukazka -> v math.h není PI
18     printf("Zadejte 5 cisel \n");
19     suma = 0;
20     for(i = 1; i <= 5; i++){
21         scanf("%d", &dalsi);
22         // osetreni vstupu
23         if(dalsi < 1){
24             printf("\n n = %d není prirozene cislo \n\n", n);
25             exit(EXIT_FAILURE);
26         }
27         suma = suma + dalsi;
28     }
29     printf("suma = %d \n\n", suma);
30     return (EXIT_SUCCESS);
31 }
```

Co C nemá

- Automatickou správu paměti
 - Garbage collector
- Velikost proměnných nezávislou na platformě
- Způsob uložení proměnných nezávislý na OS (Little-endian, Big endian,...)

Co je v C navíc

- Preprocessor
 - Vkládání hlavičkových souborů (header file) do zdrojového kódu
 - Podmíněný překlad
 - Makra
 - **#pragma** – doplňující příkazy závislé na platformě
- Linker – spojování přeložených modulů a knihoven do spustitelného kódu
- **Ukazatele (pointer) jako prostředek nepřímého adresování proměnných**
- struct a bitová pole (strukturowané proměnné z různých prvků)
- union – překrytí proměnných různého typu (sdílení společné paměti)
- typedef – zavedení nových typů pomocí již známých typů
- sizeof – určení velikosti proměnné (i strukturovaného typu)

- Jiné názvy i parametry funkcí ze standardních knihoven (práce se soubory, znaky, řetězci, matematické funkce,)
- příkaz goto navesti;

Program C obsahuje

- Příkazy preprocesoru (Preprocessor commands)
- Definice typů (type definitions)
- Prototypy funkcí (function prototypes) kde je uvedena deklarace:
 - Jména funkce
 - Vstupních parametrů
 - Návratové hodnoty funkce
- Proměnné (variables)
- Funkce (functions) (procedura v C je funkce bez návratové hodnoty nebo void)
- Komentáře: //, /* */ , nesmí být vnořené

Uvnitř funkce nelze definovat lokální funkce (definice funkci nesmí být vnořené).

C komplikace

- 1. Preprocesor:**
 - a. Čte zdrojový kód v C
 - b. Odstraní komentáře (nahradí každý komentář jednou mezerou, pozor: některé překladače nepodporují // komentáře, pouze /* */)
 - c. Upraví zdrojový text podle direktiv preprocesoru (řádky začínající #)
 - i. Vloží do textu obsah jiného souboru `#include ...`
 - ii. Odebere text vymezený direktivami podmíněného překladu
 - iii. Expanduje makra
- 2. Překladač C:**
 - a. Čte výstup z preprocesoru
 - b. Kontroluje syntaktickou správnost textu
 - c. Hlásí chyby a varování
 - d. Generuje text v assembleru (když nejsou chyby)
- 3. Assembler:**
 - a. Čte výstup z překladače C
 - b. Generuje relokovatelný object kód (kód s nevyřešenými odkazy mezi moduly)
 - c. Přeloží případné moduly zapsané přímo v assembleru (mix programovacích jazyků)
- 4. Linker (spojovací program):**
 - a. Čte object kód všech zúčastněných modulů programu
 - b. Připojí knihovní object moduly (přeložené dříve nebo dodané)
 - c. Vyřeší odkazy mezi moduly
 - d. Generuje spustitelný kód (zjednodušené)

Celočíselné typy

- Rozsahy celočíselných typů v C nejsou dány normou, ale implementací (pro 16ti a 64 bitové prostředí jsou jiné než je uvedeno v následující tabulce - ta je pro 32 bitové prostředí)
- `limits.h`, `float.h`
- Norma pouze garantuje
 - short <= int <= long
 - unsigned short <= unsigned <= unsigned long
- Celočíselné literály (zápisy čísel):
 - dekadický 123 456789
 - hexadecimální 0x12 0xFFFF (začíná 0x nebo 0X)

- oktalový 0123 0567 (začíná 0)
- unsigned 123456U (přípona U nebo u)
- long 123456L (přípona L nebo l)
- unsigned long 123456UL (přípona UL nebo ul)
- Není-li uvedena přípona, jde o literál typu int
- C - racionální čísla (neceločíselné datové typy):
 - Velikost reálných čísel určená implementací
 - Většina překladačů se řídí standardem IEEE-754-1985
- C - typ void: void značí prázdnou hodnotu nebo proměnnou bez typu (jen ukazatelé)
 - void funkce1 (...) - fukce bez návratové hodnoty (procedura)
 - int funkce2 (void) - funkce bez vstupních parametrů
 - void *ptr; - ukazatel bez určeného typu (viz dále)
- **#define** konstanta
 - **#define** CERVENA 0 /* Zde muze byt komentar */
 - je to makro bez parametrů, každé **#define** musí být na samostatné řádku)
 - Preprocessor provede textovou nahradu všech výskytů slova CERVENA znakem 0
 - může být i vnořená: **#define** MAX_2 MAX_1+30

Funkce

C je modulární jazyk = funkce je jeho základním stavebním blokem.

- Každý program v C obsahuje minimálně funkci main()

```
int main(int argc, char** argv) { ... }
```

- Běh programu začíná na začátku funkce **main()**
- Definice funkce obsahuje hlavicku funkce a její tělo
- C používá prototypu funkce k deklaraci informací nutných pro překladač, aby mohl správně přeložit volání funkcí i v případě, že definice funkce je umístěna dále v kódu modulu nebo je jiném modulu
- int max(int a,int b);
 - Deklarace se skládá pouze z hlavičky funkce, (odpovídá interface v Javě)
 - Parametry se do funkce předávají hodnotou (call by value), parametrem může být i ukazatel (pointer). Ten pak dovolí předávat parametry i odkazem.
 - C nepovoluje funkce vnořené do jiných funkcí (lokální funkce ve funkci)
 - Jména funkcí jsou implicitně extern, a mohou se exportovat do ostatních modulů (samostatně překládaných souborů)
 - Specifikátor **static** před jménem funkce omezí viditelnost jejího jména pouze na daný modul (lokální funkce modulu)
 - Formální parametry funkce jsou lokální proměnné inicializované skutečnými parametry při volání funkce
 - C dovoluje rekurzi, lokální proměnné jsou pro každé jednotlivé volání zakládány znovu (v zásobníku). Kód funkce v C reentrantní (reentrant = Reentrantní provádění bloků znamená, že je možné provádět několikanásobně volaný blok paralelně.).
 - Funkce nemusí mít žádné vstupní parametry, zapisuje se funkceX(void)
 - Funkce nemusí vracet žádnou funkční hodnotu, pak je návratový typ void (je to procedura)
 - Pokud v definici parametrů funkce je klíčové slovo **const** - tento parametr (předaný odkazem, např. pole) nelze uvnitř funkce měnit
 - př. **int fce (const char *src) { ... }**, z pole lze pouze číst, jeho prvky nelze uvnitř funkce měnit

Stdin

Načtení hodnoty ze stdin: **scanf("%d", &x);**

- do funkce scanf vstupuje jako parametr adresa (resp. reference) paměťového místa, kam se má

načtená hodnota uložit

- jde o předání parametru odkazem, jde tedy o parametr, který může být využit pro vstup i výstup hodnoty
- funkce v C může takto „vracet“ více hodnot

Bezpečné načtení double v C

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int nextDouble(double *cislo){
4     // === Bezpecne pro libovolny zadany pocet znaku ===
5     // Navratova hodnota:
6     // TRUE - zadano realne cislo
7     // FALSE - neplatny vstup
8     enum boolean {FALSE,TRUE}; // v ANSI C99 uz existuje true a false, zde vycsty
9     const int BUF_SIZE = 80;
10    char vstup[BUF_SIZE],smeti[BUF_SIZE];
11    // fgets prekte az sizeof(vstup)-1 znaku ze stdin (standardni vstup) a pushne
12    // je do vstup
13    fgets(vstup,sizeof(vstup),stdin);
14    // sscanf je jako scanf, ale cteno z bufferu
15    if(sscanf(vstup,"%lf[^\\n]",cislo,smeti) != 1)
16        return(FALSE); // Input error
17    return(TRUE);
18 }
```

Dynamická správa paměti - Alokace paměti

Pole lze alokovat normálně staticky, musí se ale předem zadat jeho velikost.

Způsob dynamické alokace:

```
(int*)malloc(count*sizeof(int)) //zde je typ promenné int*, ne int !
```

Komplexnější příklad použití:

```
1 int* ctiPole1 (int *delka, int max_delka){
2     // Navratovou hodnotou funkce je ukazatel na prideleno pole
3     int i, *p;
4     printf(" Zadejte pocet cisel = ");
5     // funkce nextInt je temer totozna, jako funkce nextDouble predstavena vyse
6     if (!nextInt(delka)) {
7         printf("\n Chyba - Zadany udaj není cele cislo\n\n");
8         exit(EXIT_FAILURE);
9     }
10    if(*delka < 1 || *delka > max_delka){
11        printf("\n Chyba - pocet cisel = <1,%d> \n\n",MAX_DELKA);
12        exit(EXIT_FAILURE);
13    }
14    // Alokace pameti (prideleni pameti z "heapu")
```

```

15 if((p=(int*)malloc(*delka)*sizeof(int))) == NULL){
16     printf("\n Chyba - není dostatek volné paměti \n\n");
17     exit(EXIT_FAILURE);
18 }
19 printf("\n Zadejte celá čísla (kazde ukončit ENTER)\n\n");
20 for (i = 0; i < *delka; i++) {
21     if (!nextInt(p+i)) {
22         printf("\n Chyba - Zadaný údaj není cele číslo\n\n");
23         exit(EXIT_FAILURE);
24     }
25 }
26 return(p); // p - ukazatel na pridělené a naplněné pole
27 }

```

Dealokace probíhá pomocí funkce `free`, předává se jí ukazatel na začátek alokované paměti:

```
void free(void *ptr)
```

Operátory

Výraz může být operandem, výraz má typ a hodnotu (x void hodnotu nemá). Priority operátorů:

- Ve výrazu: funkce1() + funkce2(), není definováno, která funkce se provede jako první.
- Máme (2 příkazy): int a = 3; a+=a++ + ++a * a++;
 - v JAVE: 31
 - v C: 26, 31, ... není definováno pořadí, programátor není upozorněn, že jde o nejednoznačný zápis
- Zkrácené vyhodnocování logických operátorů
- Operandy musí být stejněho aritmetického typu, nebo oba struct nebo union stejněho typu, nebo oba pointery stejněho typu (pravý může být NULL)

Ukazatelé - Pointery

- Motivace
 - Předávání parametru odkazem
 - Práce s poli, řízení průchodem polem
 - Pointer na funkci
 - Využívání pole funkcí
 - Vytváření seznamových struktur
 - Hashování
- Ukazatel v C je přímo implementován pamětí procesoru a je možné přímo adresovat, včetně požadavku na registry
 - Mocný nástroj pro implementaci strojově orientovaných aplikací
- Ukazatel (pointer) je proměnná jejíž hodnotou je „ukazatel“ najinou proměnnou (analogie nepřímé adresy ve strojovém kódu či v assembleru)
- Ukazatel má též typ proměnné na kterou může ukazovat
 - ukazatel na char, int,..
 - „ukazatel na pole“
 - ukazatel na funkci
 - ukazatel na ukazatel
 - atd.
- Ukazatel může být též bez typu (void), pak může obsahovat adresu libovolné proměnné. Její velikost pak nelze z vlastností ukazatele určit

- Neplatná adresa, ale definovaná v ukazateli má hodnotu konstanty NULL (kterémukoliv pointeru lze přiřadit hodnotu NULL)
- C za běhu programu nekontroluje zda adresa v ukazateli je platná
- Specialitou C je pointer na funkci!
 - ukazatel umožní, aby funkce byla parametrem funkce
- Pomocí ukazatele lze předávat parametry funkci odkazem (call by reference) – základní využití
- Adresa proměnné se zjistí adresovým operátorem & (ampersand), tzv. referenční operátor
- K proměnné na kterou ukazatel ukazuje se přistoupí operátorem nepřímé adresy * (hvězdička)

Ukázka pro lepší pochopení

```

1 int x=30; // promenná typu int, &x - adresa promenné x
2 int *px; // *px promenná typu int, px promenná typu pointer na int
3 px=&x; // do promenné typu pointer na int se uloží adresa promenné x
4 printf(" %d " "%d \n", x, px); // 30 2280564
5 printf(" %d " "%d \n", &x, *px); // 2280564 30
6 printf(" %d " "%d \n", *(&x), &(*px)); // 30 2280564

```

Další ukázka

```

1 int x;
2 int *px;
3 int **ppx;
4 x=1;
5 px=NULL;
6 ppx=NULL;
7 px=&x;
8 ppx=&px;
9 **ppx=6;
10 *px=10;
11 x = 55;
12 // *ppx = 20 // CHYBA, konverze int na int* nelze
13 // px = 20 // CHYBA, konverze int na int* nelze
14 printf(" %d " "%d " "%d" " \n", x, *px, **ppx); // 55 55 55
15 printf(" %d %d %d " , &x, px, ppx, *ppx); // 2293527 2293527 2293527 (pozn. p
ointa: vsechny 4 hodnoty stejne, cislo bude mit kazdej jiny, kdyz si to zkusite)

```

Operace s pointery

- Povolené aritmetické operace s ukazateli:
 - -pointer + integer
 - pointer - integer
 - pointer1 - pointer2 (musí být stejného typu)
- Povolené operandy relace:
 - dva ukazatele (pointers) shodného typu nebo jeden z nich NULL nebo typu void
- Jednoduché přiřazení - povolené operandy
 - dva operandy typu - pointer (stejného typu) nebo pravý operand=NULL nebo jeden pointer typu void
- Aritmetické operace jsou užitečné když ukazatel ukazuje na pole

Pole

- Pole je množina prvku (proměnných) stejného typu
- Index prvního prvku je vždy roven 0
- Pole se funkcím předává odkazem: void funkcePole (int p[])
- Prvky pole mohou být promenné libovolného typu (i strukturované)
- Definice pole urcuje:
 - jméno pole
 - typ prvku pole
 - počet prvků pole
- Prvky pole je možné inicializovat
- Počet prvku statického pole musí být znám v době prekladu
- Prvky pole zabírají v paměti souvislou oblast!
- Velikost pole (byte) = počet prvku pole * sizeof (prvek pole)
- C - nemá promennou typu String, nahrazuje se jednorozmerným polem z prvku typu char. Poslední prvek takového pole je vždy '\0' (null char)
 - lze také zapsat char *c; c = "ja jsem string"; // c je ukazatel na první prvek pole

Hlavičkové soubory

- Důvody:
 - Deklarace funkčního prototypy před použitím
 - Prostředek pro zpřehlednění struktury programu
 - Ukrytí definice funkce, možnost vytváření knihoven
 - Předání souboru .h + .obj
 - Vlastní definice funkce v souborech s relativním kódem .obj
- Obsahují
 - Hlavičky funkcí (funkční prototypy)
 - deklarace funkce
 - Deklarace globálních proměnných
 - Definice datových typů
 - Definice symbolických konstant
 - Definice make
- „obdoba interface“
- Příklad soubor xxx.h:

```
1 /* podmineny preklad proti opakovánemu vkládání „include“ */
2 #ifndef XXX // čti: if not defined = proti duplicitě = podmíněný překlad
3 #define XXX
4 /* definice symb. konstant využívaných i v jiných modulech */
5 #define CHYBA -1.0
6 /* definice maker s parametry */
7 #define je_velke(c) ((c) >= 'A' && (c) <= 'Z')
8 /* definice globálních typů */
9 typedef struct{
10     int vyska;
11     int vaha;
12 } MIRY;
13 /* deklarace globálních proměnných modulu xxx.c */
14 extern MIRY m; // v jiném modulu bude definice MIRY m;
```

```

15 /* uplne funkci prototypy globalnich funkci modulu xxx.c */
16 extern double vstup_dat(void);
17 extern void vystup_dat(double obsah);
18 #endif
19 - Příklad soubor xxx.c (includuje xxx.h)
20 #include <stdio.h> /* standardni vklad*/
21 #include „xxx.h“ /* natazeni konstant,prototypu funkci a globalnich typu vlastni
ho modulu */
22 /* deklarace globalnich promennych */
23 extern int z; /*ktere nebyly definovány v hlavičkovém soubor */
24 /* definice globalnich promennych */
25 int y; /* které nejsou definovány v hlavičkovém soubor */
26 /* lokalni definice symbolickych konstant a maker */
27 #define kontrola(x) ( ((x) >= 0.0) ? (x) : CHYBA_DAT )
28 /* lokalni definice novych typu */
29 typedef struct{} OSOBA;
30 /* definice statickyh globalnich promennych */
31 static MIRY m;
32 /* uplne funkci prototypy lokalnich funkci */
33 int nextDouble(double *cislo);
34 /* funkce main() */
35 int main(int argc, char** argv){}
36 /* definice globalnich funkci - to, ze je glob., bylo definovano v xxx.h */
37 double vstup_dat(void){ ...return ();}
38 /*funkční prototypy v.h souboru*/
39 void vystup_dat(double obsah){ ... }
40 /* definice lokalnich funkci */
41 int nextDouble(double *cislo){... }

```

Typedef

Umožňuje vytvářet nové datové typy:

```

1 typedef double *PF;
2 typedef int CELE;
3 PF x,y;
4 CELE i,j;

```

Struktury

- Struktura je konečná množina prvků (proměnných), které nemusí být stejného typu
 - Obdoba třídy bez metod v Javě
 - Record v jiných jazycích
- Skladba struktury je definovaná uživatelem jako nový typ sestavený z již definovaných typů
- K prvkům struktury se přistupuje tečkovou notací
- K prvkům struktury je možné přistupovat i pomocí ukazatele na strukturu operátorem ->

- Struktury mohou být vnořené (jak se to řeší v Javě? odpověď: Patrně kompozicí.)
- Pro struktury stejného typu je definována operace přiřazení struct1=struct2
 - (pro proměnné typu pole přímé přiřazení není definováno, jen po prvcích) – co z toho vyplývá???
- Struktury (jako celek) nelze porovnávat relačním operátorem ==
- co z toho vyplývá???
- Struktura může být do funkce předávána hodnotou i odkazem
- Struktura může být návratovou hodnotou funkce

```

1 typedef struct { // <==== Pomoci Typedef
2     char jmeno[20]; // Prvky struktury, pole
3     char adresa[50]; // - " - pole
4     int telefon; // - " -
5 } Tid, *Tidp;
6 Tid sk1, skAvt[20]; // struktura, pole struktur
7 Tidp pid; // ukazatel na strukturu
8 sk1.jmeno="Jan Novak"; // teckova notace
9 skAvt[0].jmeno="Jan Novak"; // prvek pole
10 pid=&sk1; // do pid adresa struktury
11 pid->jmeno="Jan Novak"; // odkaz pomoci ->
12 (*pid).jmeno="Jan Novak"; // odkaz pomoci *

```

Union

- Union je množina prvků (proměnných), které nemusí být stejného typu
- Prvky unionu sdílejí společně stejná paměťová místa (překrývají se) •
- Velikost unionu je dána velikostí největšího z jeho prvků
- Skladba unionu je definovaná uživatelem jako nový typ sestavený z již definovaných typů
- K prvkům unionu se přistupuje tečkovou notací
- př:

```

1 union Tnum{ // <==== Tnum=jmeno sablony (tag)
2     long n;
3     double x;
4 };
5 union Tnum nx; // nx - promenna typu union
6 nx.n=123456789L; // do n hodnota long
7 nx.x=2.1456; // do x hodnota double (prekryva n)

```

Podmíněný překlad

```

1 #if VERSE_CITACE == 1
2 do {
3 ...
4 } while(TRUE);
5 #endif

```

Definice vs. deklarace

1. Deklarace určuje interpretaci a vlastnosti identifikátoru(ů)

2. Definice je deklarace včetně přidělení paměti (memory allocation) proměnným, konstantám nebo funkcím

Standardní knihovny

Vlastní jazyk C neobsahuje žádné prostředky pro vstup a výstup dat, složitější matematické operace, práci s řetězci, třídění, blokové přesuny dat v paměti, práci s datem a časem, komunikaci s operačním systémem, správu paměti pro dynamické přidělování, vyhodnocení běhových chyb (run-time errors) apod.. Tyto a další funkce jsou však obsaženy ve standardních knihovnách (ANSI C Library) dodávaných s překladači jazyka C. Uživatel dostává k dispozici přeložený kód knihoven (který se připojuje – linkuje k uživatelskému kódu) a hlavičkové soubory (headers) s prototypy funkcí, novými typy, makry a konstantami. Hlavičkové soubory (obdoba interface v Javě) se připojují k uživatelskému kódu direktivou preprocesoru `#include <...>`. Je zvykem, že hlavičkové soubory mají rozšíření *.h, např. stdio.h.

Příklad:

- Vstup a výstup (formátovaný i neformátovaný) - `stdin.h`
- Rozsahy čísel jednotlivých typů - `limits.h`
- Matematické funkce - `stdlib.h, math.h`
- Zpracování běhových chyb (run-time errors) - `errno.h, assert.h`
- Klasifikace znaků (typ char) - `ctype.h`
- Práce s řetězci (string handling) - `string.h`
- Internacionalizace (adaptace pro různé jazykové mutace) - `locale.h`
- Vyhledávání a třídění - `stdlib.h`
- Blokové přenosy dat v paměti - `string.h`
- Správa paměti (Dynamic Memory Management) - `stdlib.h`
- Datum a čas - `time.h`
- Komunikace s operačním systémem - `stdlib.h, signal.h`
- Nelokální skok (lokální je součástí jazyka, viz goto) - `setjmp.h`

Abstraktní datové struktury

Abstraktní datový typ vs. Abstraktní datová struktura

- u datového typu se zabývame predevsim jeho chováním a matematickými vlastnostmi
- u datové struktury nás zajímá konkrétní reprezentace a implementace
- datové typy jsou spíše z pohledu uživatele, abstraktní
- datové struktury jsou konkrétní a dležite pro toho, kdo implementuje

Pojmy

- Lineární spojová struktura (spojovalý seznam) - každý prvek má nanejvýš jednoho následníka
- Nelineární spojová struktura (strom) - každý prvek může mít více následníků
- Binární strom - každý prvek (uzel) má nanejvýš dva následníky

(Abstraktní) datový typ

ADT jsou nezávislé na vlastní implementaci

Počet složek:

- neměnný = statický datový typ, počet položek je konstantní, pole, řetězec, třída
- proměnný = dynamický datový typ, počet složek je proměnný, mezi operace patří vložení, odebrání určitého prvku

Typ položek, dat:

- homogenní = všechny položky stejného typu
- nehomogenní = různého typu

Existence bezprostředního následníka

- lineární = existuje [např. pole, fronta, seznam,...]
- nelineární = neexistuje [strom, tabulka,...]

Základní ADT jsou například:

- asociativní pole
- zásobník
- seznam
- fronta
- množina
- textový řetězec
- strom
- halda neboli prioritní fronta
- hashovací tabulka

Spojové seznamy

Lineární seznam (také lineární spojový seznam) je dynamická datová struktura, vzdáleně podobná poli (umožňuje uchovat velké množství hodnot ale jiným způsobem), obsahující jednu a více datových položek (struktur) stejného typu, které jsou navzájem lineárně provázány vzájemnými odkazy pomocí ukazatelů nebo referencí. Aby byl seznam lineární, nesmí existovat cykly ve vzájemných odkazech.

Lineární seznamy mohou existovat jednosměrné a obousměrné. V jednosměrném seznamu odkazuje každá položka na položku následující a v obousměrném seznamu odkazuje položka na následující i předcházející položky. Zavádí se také ukazatel nebo reference na aktuální (vybraný) prvek seznamu. Na konci (a začátku) seznamu musí být definována zarážka označující konec seznamu. Pokud vytvoříme cyklus tak, že konec seznamu navážeme na jeho počátek, jedná se o kruhový seznam.

Základním prvkem spojových struktur je dvojice:

- reference - odkaz(y) na další prvek spojové struktury, definuje operátor new – znázorňujeme šipkou
- hodnota – informace libovolného typu

Nejjednodušší typ spojové struktury – jednosměrné spojové seznamy.

Spojové struktury jsou mocným implementačním prostředkem pro ADT, viz další přednáška o ADT a předmět Algoritmizace:

1. – Fronty
2. – Zásobníky
3. – Stromy
4. – Grafy

Stromy

V informatice je strom široce využívanou datovou strukturou, která představuje stromovou strukturu s propojenými uzly.

Pojmy:

1. „Cesta“ k nějakému uzlu je definována jako posloupnost všech uzlů od kořene k uzlu.
2. „Délka cesty“ je rovna počtu hran, které cesta obsahuje, tedy počtu uzlů posloupnosti – 1 (minus jedna).
3. „Hloubka uzlu“ je definována jako délka cesty od kořene k uzlu. Prvky se stejnou hloubkou jsou na „téže úrovni“.
4. „Výška stromu“ je rovna hodnotě maximální hloubky uzlu, se označuje též za „hloubku stromu“.
5. „Šírkou stromu“ je počet uzlů na stejně úrovni.
6. Strom má „nejmenší výšku“ právě tehdy, když na všech úrovních (s možnou výjimkou té poslední) má tato struktura plný počet uzlů. Úroveň všech listů je stejná nebo se liší maximálně o 1.

Uspořádanost

„Uspořádaný“ nebo také „seřazený strom“ je takový strom, ve kterém jsou všichni přímí potomci každého

uzlu seřazeni. Tudíž, pokud uzel má n dětí, lze určit prvního přímého potomka, druhého přímého potomka, až n-tého přímého potomka. U „neuspořádaného stromu“ se jedná o strom v čistě strukturálním smyslu. To znamená, že pro daný uzel nejsou uspořádáni potomci.

Vyvážený strom

„Vyvážený strom“ je takový strom, který má uzly rovnoměrně rozložené, tedy má nejmenší výšku. Ideální situace je taková, kdy má strom v každé hladině, kromě poslední, maximální počet uzlů, a v poslední hladině má uzly co nejvíce vlevo.

Procházení stromu (podrobněji pravděpodobně v ALG)

1. **Do šířky** - Procházením „stromu do šířky“ (anglicky „level-order“) se rozumí procházení stromem po vrstvách úrovní (tzn. po hladinách).
2. **Do hloubky** - Procházení začíná v kořeni stromu a postupuje se vždy na potomky daného vrcholu. Procházení končí, když v žádné větvi (tj. v žádném podstromu) již není následník. Podle pořadí, ve kterém se prochází uzly uspořádaného stromu, se rozlišují tři základní metody:
 - a. PREORDER
 - i. proved' akci
 - ii. projdi levý podstrom
 - iii. projdi pravý podstrom
 - b. INORDER
 - i. projdi levý podstrom
 - ii. proved' akci
 - iii. projdi pravý podstrom
 - c. POSTORDER
 - i. projdi levý podstrom
 - ii. projdi pravý podstrom
 - iii. proved' akci

Binární stromy

Každý prvek (uzel) má nanejvýš dva následníky (obecně lze prvkům stromu říkat také node, z ang., čteno [nouda]). Binární strom obsahuje uzly, které mají nejvíce dva syny.

Důležité pojmy:

1. kořen stromu - nejvyšší prvek, nemá rodiče
2. levý podstrom - strom levých potomků
3. pravý podstrom - pravých
4. vnitřní uzel - prvek, který má potomky (často se sem nepočítá kořen, někdy ano)
5. list - prvek, který již nemá potomky

U binárního stromu rozlišujeme další pojmy:

- Plný binární strom - všechny jeho listy jsou ve stejně hloubce.
- Úplný binární strom - každý vnitřní uzel má dva syny.
- Vyvážený binární strom - hloubka listů se od sebe liší maximálně o jedna.

Trochu matiky

- h – hloubka stromu,
- n – počet uzlů
- n_0 – počet listů
- n_2 – počet vnitřních uzlů

Úplný binární strom:

minimální počet uzlů: $n = 2h + 1$

maximální počet uzlů: $n = 2 \cdot (h + 1) - 1$

počet listů: $\lceil n/2 \rceil$

PREVIOUS

Základní pojmy

Informatika

Informatika:

- zabývá se zpracováním informací nejen na počítačích
- studuje výpočetní a informační procesy z hlediska hardware i software – „technická informatika“
- je součástí teorie informací, věda spojující aplikovanou matematiku a elektrotechniku za účelem kvantitativního vyjádření informace

Software

V informatice sada všech počítačových programů v počítači. Software zahrnuje:

- operační systém (zajišťuje běh programů)
- aplikační software (pracuje s ním uživatel)
- další (knihovny, middleware, BIOS, firmware apod.)

Knihovna

Knihovna je v informatice označení pro soubor funkcí a procedur (v objektovém programování též objektů, datových typů a zdrojů), který může být sdílen více počítačovými programy. Knihovna usnadňuje programátorovi tvorbu zdrojového kódu tím, že umožňuje použít již vytvořený kód i v jiných programech. Knihovna navenek poskytuje své služby pomocí API (aplikační rozhraní), což jsou názvy funkcí (včetně popisu jejich činnosti), předávané parametry a návratové hodnoty. Knihovny lze rozdělit podle vazby na program, který je používá, na statické a dynamické.

Statické knihovny

Statická knihovna tvoří s přeloženým programem kompaktní celek. V závěrečné fázi překladu programu ze zdrojového kódu do strojového kódu jsou volané funkce (a další jimi kaskádově volané funkce) připojovány linkerem (odtud označení linkování) do výsledného spustitelného souboru. V horším případě je k programu připojena celá knihovna bez ohledu na to, jaké části program skutečně využívá. Výsledný spustitelný soubor tak v sobě obsahuje všechny části, které jsou nutné pro jeho běh. Staticky slinkovaný spustitelný soubor proto typicky při spuštění nepotřebuje žádné další soubory, takže je ho možné překopírovat na jiný systém a jednoduše spustit (tj. bez instalace, tak, jak je známá například pro běžné programy v Microsoft Windows). Př. .lib, .a.

Dynamické knihovny

Dynamické knihovny nejsou (na rozdíl od statických knihoven) k výslednému spustitelnému souboru přidávány. V závěrečné fázi překladu programu ze zdrojového kódu do strojového kódu jsou odkazy na volané knihovní funkce pomocí linkeru (odtud označení linkování) zapsány do speciální tabulky symbolů, která je připojena k výslednému spustitelnému souboru. Pro chod programu (tj. spuštění výsledného spustitelného souboru) je pak nutné mít k dispozici též příslušné dynamické knihovny. Pokud dynamická knihovna využívá ke své činnosti jiné dynamické knihovny, vzniká řetězec závislostí a všechny potřebné knihovny musí být při spuštění programu přítomny. Př. .dll, .o, .so.

1.3 Hardware

Zahrnuje všechny fyzické součásti počítače

- čistě elektronická zařízení (procesor, paměť, display)
- elektromechanické díly (klávesnice, tiskárna, diskety, disky, jednotky CD-ROM, páskové jednotky, reproduktory) pro vstup, výstup a ukládání dat.
- Počítač se skládá z procesoru, operační paměti a vstupně-výstupních zařízení.

1.4 Algoritmus

Postup při řešení určité třídy úloh, který je tvořen seznamem jednoznačně definovaných příkazů a zaručuje, že pro každou přípustnou kombinaci vstupních dat se po provedení konečného počtu kroků

dospěje k požadovaným výsledkům.

- **hromadnost** - měnitelná vstupní data
- **determinovanost** - každý krok je jednoznačně definován
- **konečnost a resultativnost** - pro přípustná vstupní data se po provedení konečného počtu kroků dojde k požadovaným výsledkům

1.5 Program

Program je předpis (zápis algoritmu) pro provedení určitých akcí počítačem zapsaný v programovacím jazyku.

1.6 Programovací jazyk

Programovací jazyk je prostředek pro zápis algoritmů, jež mohou být provedeny na počítači.

1. strojově orientované

- **strojový jazyk** = jazyk fyzického procesoru
- **asembler** = jazyk symbolických adres

2. vyšší jazyky

- **imperativní** (příkazové, procedurální)
- **neimperativní** (např. funkcionální)

1.6.1 Imperativní programování

Hlavní rysy imperativních jazyků (např. C, C++, Java, Pascal, Basic, ...)

- zpracovávané údaje mají formu datových objektů různých typů, které jsou v programu reprezentovány pomocí proměnných resp. konstant
- program obsahuje deklarace a příkazy
- deklarace definují význam jmen (identifikátorů)
- příkazy předepisují akce s datovými objekty nebo způsob řízení výpočtu

1.7 Syntaxe

Souhrn pravidel udávajících přípustné tvary dílčích konstrukcí a celého programu.

1.8 Sémantika

Udává význam jednotlivých konstrukcí.

1.9 Interpret

Interpret je v informatice speciální počítačový program, který umožňuje **přímo vykonávat (interpretovat)** zápis jiného programu v jeho zdrojovém kódu ve zvoleném programovacím jazyce. Program proto není nutné převádět do strojového kódu cílového procesoru, jako je tomu v případě překladače. Interpret tak umožňuje programování kódu, který je snadno přenositelný mezi různými počítačovými platformami.

1. provádějí přímo zdrojový kód (unixový shell, COMMAND.COM nebo interpret jazyka BASIC)
2. přeloží zdrojový kód do efektivnějšího mezikódu, který následně spustí (Perl, Python nebo MATLAB)
3. přímo spustí předem vytvořený překompilovaný mezikód, který je produktem části interpretu (UCSD Pascal a Java - zdrojové kódy jsou komplikovány předem, uloženy ve strojově nezávislém tvaru, který je po spuštění linkován a interpretován nebo komplikován v případě použití JIT).

1.9.1 Vnitřní forma

Při interpretaci se zdrojový program přeloží do vnitřní formy. Ta není strojovým jazykem fyzického procesoru, ale je jazykem virtuálního počítače. Provedení programu ve vnitřní formě na konkrétním počítači zajistí interpret.

1.10 Kompilátor = překladač

Překladač (též kompilátor) je v nejčastějším smyslu slova softwarový nástroj používaný programátory pro

vývoj softwaru. Kompilátor slouží pro překlad algoritmů zapsaných ve vyšším programovacím jazyce do jazyka strojového, či spíše do strojového kódu. Z širšího obecného hlediska je kompilátor stroj, respektive program, provádějící překlad z nějakého vstupního jazyka do jazyka výstupního.

Části programovacích jazyků

Proměnná

V imperativním programování je proměnná „úložiště“ informace (tedy vyhrazené místo v paměti - v některých jazycích se ovšem v průběhu výpočtu může místo, kde je proměnná uložena, měnit). Proměnná nebo (v beztypových jazycích) její hodnota má typ.

Proměnná je datový objekt, který je označen jménem a je v něm uložena hodnota nějakého typu, která se může měnit.

Přiřazení

Pro přiřazení hodnoty proměnné slouží příkazovací příkaz. Má tvar <proměnná> = <výraz>;

V jazyku Java zavedeme proměnnou deklarací int promenna = 0;

Přiřazovací příkaz: promenna = 37;

Konstantě nelze přiřadit hodnotu: MAX = 32 -> CHYBA!!

Jazyk JAVA má silnou typovou kontrolu, následující nelze: boolean b; b = 1; -> CHYBA!!

2.1.1.1 Přiřazovací operátory

- <proměnná> = <proměnná> <OP> <výraz> e.g. x = x +1
- <proměnná> <OP>= <výraz> x += 1

2.1.2 Přidělení paměti proměnným

Přidělením paměti proměnné rozumíme určení adresy umístění proměnné v paměti počítače.

2.1.3 Typy proměnných

- lokální proměnné funkci - pamět přidělena při volání funkce, po jejím skončení uvolněna, pamět na zásobníku
- statické proměnné tříd - pamět přidělena při zavedení kódu třídy do paměti (JVM), až do konce programu

Primitivní datové typy

Datový typ (zkráceně jen typ) specifikuje:

- **množinu hodnot** - pro int: -2147483648 .. 2147483647
- **množinu operací**, které lze s hodnotami daného typu provádět (pro int je to například: relační operace ==, !=, >, >=, <, <=, jejichž výsledkem je hodnota typu boolean)

V jazyce JAVA je třeba deklarovat, s jakým typem dat budeme pracovat. Překladač tuto deklaraci hlídá. V důsledku reprezentace dat v počítači vznikají nepřesnosti (1.00 + 2.00 se nemusí rovnat 3.00). Čísla jsou approximována. Čím větší exponent (tzn. čím dále od nuly), tím je větší mezera mezi approximacemi. Kolem nuly je čísel nejvíce. Reprezentace dat v počítači (int, double apod.) bude podrobněji rozebrána pravděpodobně v nějaké otázce z APO.

JAVA - Primitivní datové typy

1. **byte**: Datový typ byte obsahuje 8bitové dvakrát doplněné celé číslo se znaménkem. Jeho minimální hodnota je -128 a maximální 127. Datový typ byte může být použit pro ušetření paměti ve velkých polích, kde se může ve speciálních případech hodit každý bit. Taktéž může být náhradou za typ int, kde mohou jeho limity pomoci objasnit váš kód; takto je hodnota limitována více než údajem v dokumentaci.
2. **short**: Datový typ short je 16bitové dvakrát doplněné celé číslo. Jeho minimální hodnota je -32 768 a maximální 32 767. Stejně jako u typu byte jsou zde stejné případy použití: typ short můžete použít k ušetření paměti při vytváření velkých polí u náročných aplikací.
3. **int**: Datový typ int je 32bitové dvakrát doplněné celé číslo. Jeho minimální hodnota je -2 147 483 648

a maximální $2\ 147\ 483\ 647$ (2^{31}). Pro celočíselné hodnoty je tento typ tou základní volbou, pokud se nedostanete do výše popsaných situací. Tento datový typ bude dostatečně velký pro většinu čísel ve vaší aplikaci, pokud však budete potřebovat širší rozsah hodnot, použijte místo něj typ long.

4. **long:** Datový typ long je 64bitové dvakrát doplněné celé číslo. Jeho minimální hodnota je $-9\ 223\ 372\ 036\ 854\ 775\ 808$ a maximální $9\ 223\ 372\ 036\ 854\ 775\ 807$ (2^{63}). Používejte tento datový typ, pokud vám nebude stačit rozsah hodnot poskytovaný datovým typem int.
5. **float:** Datový typ float je 32bitové IEEE 754 číslo s pohyblivou desetinou čárkou. Stejně jako u typu byte a short používejte typ float (místo double), pokud potřebujete ušetřit paměť ve velkých polích. Tento datový typ byste nikdy neměli používat pro přesné hodnoty, jako jsou finanční částky. Pro tento případ budete muset použít třídu java.math.BigDecimal.
6. **double:** Datový typ double je 64bitové IEEE 754 číslo s pohyblivou desetinou čárkou. Pro desetinná čísla je tento datový typ tou základní volbou. Stejně jako typ float se double nehodí pro přesné hodnoty, jako jsou finanční částky.
7. **boolean:** Datový typ boolean má povoleny pouze 2 hodnoty: true a false. Používejte tento datový typ, abyste specifikovali, zda je podmínka pravdivá (true) nebo nepravdivá (false). Tento datový typ obsahuje 1 bit informací, ale jeho velikost není přesně definována.
8. **char:** Datový typ char je jednoduchý 16bitový Unicode znak (proto jsou odstraněny problémy s různými jazyky). Jeho minimální hodnou je u0000 (nebo 0) a maximální uffff (nebo 65535). Konverze znaků probíhá v JAVA většinou automaticky podle nastavení jazyka OS.

byte +-* / byte ... int

short +-* / short ... int

int +-* / int ... int (pozor na přetečení)

long +-* / long ... long

Typové konverze

Typová konverze je operace, která hodnotu nějakého typu převede na hodnotu jiného typu.

- implicitní - např. int na double (když se očekává hodnota double, ale je tam INT, dojde k implicitní konverzi) - implicitní konverze je bezpečná
- explicitní - programátor musí explicitně označit, je potenciálně nebezpečná (může dojít ke ztrátě informace) - eg. double -> int, double d = 1E30; int i = (int)d; // i je 2147483647 asi 2E10

Výrazy

Výraz se skládá z operandů a operátorů. **Operandem** může být konstanta, proměnná, volání metody nebo opět výraz. **Operátory** udávají, co se má provést s jednotlivými hodnotami operandů.

Pozor na asociativitu. Odčítání je asociativní zleva, mocnění zprava.

Příkazovací příkaz může být výraze: $y = x = x + 6$, tj. $y = (x = (x + 6))$;

Operace `&&` a `||` se vyhodnocují zkráceným způsobem, druhý operand se nevyhodnocuje, když lze výsledek určit již z prvního operandu.

Výstup

Pro výpis dat na obrazovku se v Javě používá příkaz `System.out.println(parametr);` = výpis na standardní výstup. Metoda je přetížená, jako její parametr lze zadat všechny primitivní datové typy i String.

Formátovaný výstup

Př. `System.out.printf("Cislo Pi = %6.3f %n ", Math.PI);`

Specifikace formátu `%[$indexParametru][modifikátor][šířka].[přesnost]konverze`

- konverze - povinný parametr
- typ celé číslo d,o,x - dekadicky, oktalově a hexadecimálně
- typ double f je desetinný zápis; e,E vědecký s exponentem
- šířka - počet sázených míst, zarovnání vpravo
- .přesnost - počet desetinných míst
- modifikátor - v závislosti na typu konverze určuje další vlastnosti, například pro konverzi f (typ

double) -> symbol + určuje, že má být vždy sázeno znaménko, -> symbol - určuje zarovnání vlevo, -> symbol 0 doplnění čísla zleva nulami.

Vstup

Pro vstup dat zadaných na klávesnici poslouží třída Scanner. Je třeba vytvořit objekt třídy Scanner a napojit jej na standardní vstupní proud: Scanner sc = new Scanner(System.in);

- sc.nextInt() : přečte celé číslo z řádku zadávaného klávesnicí (řádek je zakončen klávesou Enter, číslo je zakončeno mezerou nebo Enter) a vrátí je jako funkční hodnotu typu int
- sc.nextDouble() : přečte číslo z řádku zadávaného klávesnicí a vrátí je jako funkční hodnotu typu double, jako oddělovač použijte . nebo čárku v závislosti na definovaném jazyku OS. Lze změnit pomocí před vytvořením Scanneru Locale.setDefault(Locale.ENGLISH);
- sc.nextLine() : přečte zbytek řádku zadávaného klávesnicí a vrátí je jako funkční hodnotu typu String

Řídicí struktury

Řídicí struktura je programová konstrukce, která se skládá z dílčích příkazů a předepisuje pro ně způsob provedení. Tři druhy řídicích struktur:

1. posloupnost, předepisující postupné provedení dílčích příkazů
2. větvení, předepisující provedení dílčích příkazů v závislosti na splnění určité podmínky
3. cyklus, předepisující opakované provedení dílčích příkazů v závislosti na splnění určité podmínky

Podmínka

Příkaz if (podmíněný příkaz) umožňuje větvení na základě podmínky.

- if (podmínka) příkaz1 else příkaz2
- if (podmínka) příkaz1

Cyklus

Základní příkaz cyklu, který má tvar: while (podmínka) příkaz (blok příkazů).

Příkaz cyklu do se od příkazu while liší v tom, že podmínka se testuje až za tělem cyklu. Tvar příkazu: do příkaz while (podmínka);

Poznámka k sémantice: příkaz do provede tělo cyklu alespoň jednou, nelze jej tedy použít v případě, kdy lze očekávat ani jedno provedení těla cyklu

Cyklus For: je často řízen proměnnou, pro kterou je stanoveno:

- jaká je počáteční hodnota
- jaká je koncová hodnota
- jak změnit hodnotu proměnné po každém provedení těla cyklu

2.6.2.1 Konečnost cyklů

Vstupní podmínu konečnosti cyklu lze určit témař ke každému cyklu (někdy je to velmi obtížné, až nespočetné). Splnění vstupní podmínky konečnosti cyklu musí zajistit příkazy předcházející příkazu cyklu.

Switch

Příkaz switch (přepínač) umožňuje větvení do více větví na základě různých hodnot výrazu (nejčastěji typu int nebo char).

Sémantika (zjednodušeně):

- vypočte se hodnota výrazu a pak se provedou ty příkazy, které jsou označeny konstantou označující stejnou hodnotu
- není-li žádná větev označena hodnotou výrazu, provedou se příkazy def

Funkce

Funkce v programování je část programu, kterou je možné opakováně volat z různých míst kódu. Funkce může mít argumenty (též parametry) – údaje, které ji jsou předávány při volání – a návratovou hodnotu, kterou naopak vrací.

Deklaraci funkce tvoří hlavička funkce a tělo funkce

- Hlavička funkce v jazyku Java má tvar static typ jméno(specifikace parametrů) kde
- typ je typ výsledku funkce (funkční hodnoty)
- jméno je identifikátor funkce
- specifikací parametrů se deklarují parametry funkce, každá deklarace má tvar typ_parametru jméno_parametru (a oddělují se čárkou)
- specifikace parametrů je prázdná, jde-li o funkci bez parametrů
- Tělo funkce je složený příkaz nebo blok, který se provede při volání funkce
- Tělo funkce musí dynamicky končit příkazem return x; kde x je výraz, jehož hodnota je výsledkem volání funkce

Parametry

Parametry funkce slouží pro předání vstupních dat algoritmu, který je funkcí realizován. Parametr může být výraz.

Procedura

Funkce, jejíž typ výsledku je void, nevrací žádnou hodnotu.

Rozklad problému na podproblémy

Postupný návrh programu rozkladem problému na podproblémy:

- zadaný problém rozložíme na podproblémy
- pro řešení podproblémů zavedeme abstraktní příkazy
- s pomocí abstraktních příkazů sestavíme hrubé řešení
- abstraktní příkazy realizujeme pomocí procedur (void)

Rekurze

V imperativním programování rekurze představuje opakování vnořené volání stejné funkce (podprogramu).

Rekurzivní algoritmus

Rekurzivní algoritmus předepisuje výpočet „shora dolů“ v závislosti na velikosti (složitosti) vstupních dat:

- pro nejmenší (nejjednodušší) data je výpočet předepsán přímo
- pro obecná data je výpočet předepsán s využitím téhož algoritmu pro menší (jednodušší) data

Výhoda: jednoduchost a přehlednost (to je diskutabilní).

Nevýhoda: Nevýhodou může být časová náročnost způsobená např. zbytečným opakováním výpočtu.

Rekurzivní funkce

Rekurzivní funkce (procedury) jsou přímou realizací rekurzivních algoritmů. Použití: faktoriál, fibonacciho posloupnost (začíná 1,1,2,3,5,8...).

Iterace

Iterace v programování znamená opakování volání funkce v počítačovém programu. Zvláštní formou iterace je rekurze. Pro naše účely lze chápout iteraci jako opakování provádění bloku příkazu, typicky pomocí cyklu.