MATURITNÍ OTÁZKY 2016

PROGRAMOVÁNÍ

1. Tvorba programu – algoritmizace a programování

A. Postup při řešení problému

1. Analýza problému
2. Stanovení podmínek, za kterých má program fungovat (druh počítače, vstupní data)
3. Sestavení algoritmu – vytvoření postupu řešení – algoritmizace
4. Sestavení programu – přepis algoritmu do zvoleného programovacího jazyka
5. Testování (odladění) programu

B. Definice a vlastnosti algoritmu

Je to postup řešení problému. Je to postup, jehož realizací získáme ze zadaných vstupních údajů po *konečném počtu činností, v konečném čase správný výsledek.* Jedná se o teoretický princip řešení, algoritmus se může objevit ale i v jakémkoli jiném vědeckém odvětví

Vlastnosti správného algoritmu:

Elementárnost – algoritmus se skládá z konečného počtu jednoduchých kroků, které jsou pro realizátora srozumitelné

Determinovanost – v každé fázi zpracování musí být určen další postup. Každý krok algoritmu musí být jednoznačně a přesně definován. Vždy musí být naprosto přesné, co a jak se má provést, jak má provádění algoritmu pokračovat.

Konečnost – činnost algoritmu skončí v reálném čase a v konečném počtu kroků. Tento počet kroků může být libovolně velký, ale pro každý jednotlivý vstup musí být konečný.

Rezultativnost – algoritmus dává pro stejné vstupní údaje vždy stejné výsledky

Hromadnost – algoritmus musí být použitelný pro všechny úlohy stejného typu. Algoritmus neřeší konkrétní problém, ale obecnou třídu obdobných problémů.

Efektivnost – algoritmus se uskuteční v co nejkratším čase a při nejmenším počtu činností.

C. Možnosti zápisu algoritmu

Slovní vyjádření – používá se pro skupinu lidí, která nemá programátorské vzdělání. Výhodou je, že se lze domluvit i s laikem. Nevýhodou je malá přehlednost.

Grafické vyjádření – jedná z nejdokonalejších forem zápisu algoritmu. Jsou to symbolické, algoritmické jazyky, které se používají pro názorné zobrazení algoritmu. Jsou přehledné. Výborně se hodí k dokumentačním účelům. Nevýhodou může být, že jim laik nemusí rozumět. Jedná se o značky (ovály, obdélníky, kosočtverce,…).

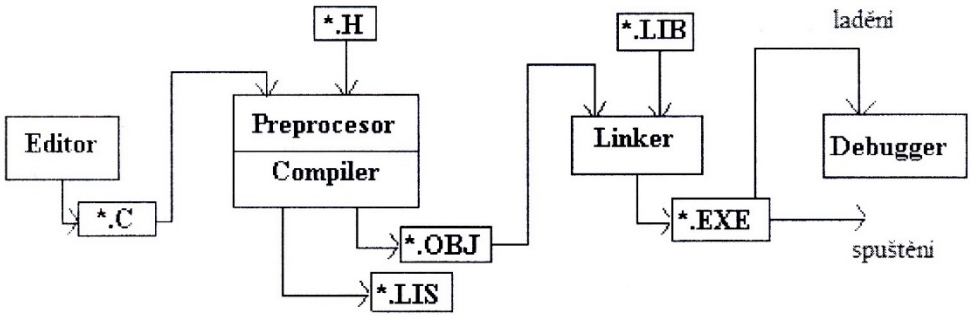
Počítačový program – je to algoritmus zapsaný v jazyce, kterému počítač rozumí a umí z něho vytvořit strojový kód. Tomuto vyjádření rozumí pouze programátor, který

umí konkrétní programovací jazyk. Je napsaný v jazyce, se kterým si počítač poradí, pokud je vybaven příslušným překladačem. Překladač je program, který umí přeložit program napsaný v programovacím jazyce, do strojového kódu.

Tabulka 1 – značky vývojového diagramu

Značka		Příklad	
Koncová značka		Start	Konec
Značka přiřazení		P=P+1	y=sin(60)
Vstup nebo výstup dat		Načti A	Tiskni A
Značka rozhodování		A>0 (-, +)	
Značka podprogramu		Záměna	
Spojka		1	

D. Zpracování programu v jazyce C



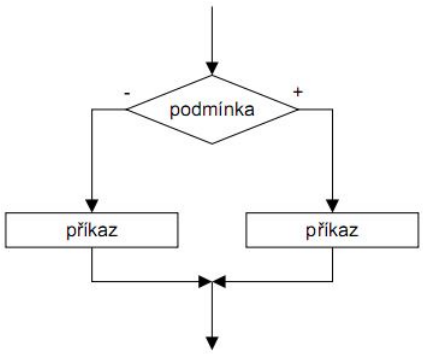
Zpracování programu probíhá v několika fázích. Nejprve je potřeba v editoru napsat zdrojový soubor s příponou *.c. Dále přichází na řadu preprocesor, který bývá součástí překladače, a který dále zpracovává zdrojový text a vkládá hlavičkové soubory, rozvíjí makra, atd. V další fázi compiler (překladač) provádí překlad upraveného zdrojového textu do relativního kódu a vytvoří soubor s příponou *.obj. V této fázi nejsou ještě známy adresy proměnných a funkcí. Následuje sestavování programu pomocí linkeru, který zajišťuje nahrazení relativních adres adresami absolutními a provede odkazy na příslušné knihovní funkce. Výsledkem této činnosti je již spustitelný soubor s příponou *.exe (případně com). Debugger je ladící program, který běží zároveň se spuštěným programem a hledá chyby, které nastanou při běhu programu. Je

také určený pro krokování programu a hledání chyb. Umožňuje sledovat běh programu řádek po řádku a sledovat při tom hodnoty proměnných, registrů nebo dokonce vybraných míst v paměti.
Během kompilace programu je potřeba nejprve soubor uložit. Některá vývojová prostředí totiž obsahují kompilátory jazyka C i C++ (což jsou do značné míry rozdílné programovací jazyky)

2. Řídící struktury – větvení

Větvení obsahuje obvykle tři části. První částí je otázka, na kterou existuje kladná nebo záporná odpověď. Druhou částí je krok. Který se provede v případě kladné odpovědi na otázku. Třetí částí je krok, který se provede v případě záporné odpovědi na otázku. První část větvení (otázka) je povinná, zbylé dvě části jsou nepovinné. Pokud však současně chybí druhý i třetí krok, ztrácí větvení smysl. Rozlišujeme několik typů větvení – alternativy:

A) Úplné



V úplném větvení jsou zařazeny kroky pro kladnou i zápornou odpověď:

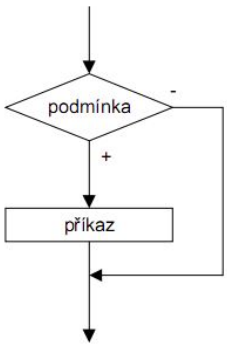
if (podmínka) Příkaz_1;

else Příkaz_2;

Za klíčovým slovem if následuje podmínka povinně uzavřená do závorek. Je-li podmínka pravdivá (nenulová), potom se provede Příkaz_1, je-li podmínka nepravdivá (nulová) provede se Příkaz_2.

Patří zde také ternární operátor – *podmínka ? příkaz_ano : příkaz_ne*

B) Neúplné



V neúplné alternativě jsou zařazeny kroky pouze pro kladnou odpověď:

If (podmínka) Příkaz_1;

Za klíčovým slovem if následuje podmínka povinně uzavřená do závorek. Je-li podmínka pravdivá (nenulová), potom se provede Příkaz_1, je-li podmínka nepravdivá (nulová) pokračuje se v programu dále.

C) Vnořené

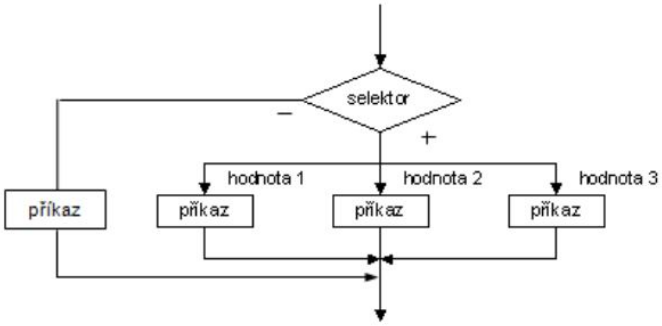
Vnořené větvení znamená, že krok pro kladnou nebo pro zápornou odpověď (nebo pro obě odpovědi) je tvořen opět větvením, které se vnoří – vloží do původního větvení:

If (podmínka_1) Příkaz_1;

else if (podmínka_2) Příkaz_2; else Příkaz_3;

Za klíčovým slovem if následuje podmínka povinně uzavřená do závorek. Je-li podmínka_1 pravdivá (nenulová), potom se provede Příkaz_1, je-li podmínka_1 nepravdivá (nulová) následuje opětovné větvení na základě podmínky_2. Je-li podmínka_3 pravdivá (nenulová), potom se provede Příkaz_2, je-li podmínka_2 nepravdivá (nulová) provede se Příkaz_3.

D) Několikanásobné



V případě vyhodnocování složitých výrazů, které způsobují větvení algoritmu do mnoha různých částí, není příliš praktické používat podmínky, protože je to nepřehledné. Daleko lepší je použít tzv. přepínač, který pohodlně rozdělí algoritmus na libovolný počet různých odvětví. Hovoříme pak o vícenásobné alternativě – vícenásobném větvení. Toto větvení slouží k rozdělení běhu programu do několika směrů na základě celočíselné hodnoty sektoru. Jako selektor musí být ordinální datový typ (char, int). Syntaxe vypadá následovně:

```
switch (selector)
{
case hod_1: Příkaz_P1; break;
case hod_2: { příkaz_Q1; příkaz_Q2;  break;  }
case hod_3: { příkaz_R1; příkaz_R2; příkaz_R3; break;  }
// sekvenci příkazů uzavřeme mezi složené závorky begin – end {}
default:  příkaz_D1;  break;  //na posledním řádku nemusí být break
}
```

Příkaz break způsobí ukončení provádění příkazu switch. Za jeho absence by se provedly i následující příkazy až po break. Příkaz default není povinný. Provede se v případě, pokud podmínce nevyhovuje žádná z nabízených hodnot. Pokud není větev default součástí příkazu switch a nenalezne shodu ani v jedné z dostupných větví case, rozhodování automaticky končí, čímž dochází k opuštění celého rozhodovacího příkazu

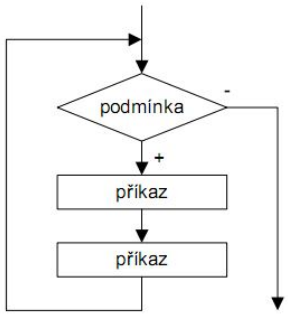
3. Řídící struktury – cykly

A. Typy a charakteristika cyklů

Cykly použijeme vždy tam, kde nastane potřeba některé činnosti zopakovat. To, zda se opakování provede či nikoliv, závisí na vyhodnocení určité podmínky. Iterace označuje jedno opakování cyklu. Během jedné iterace cyklu jsou postupně zpracovány všechny příkazy z těla cyklu. Cyklus přijímá rozhodnutí o tom, zda bude provedena další iterace cyklu, či nikoliv. Toto rozhodnutí většinou vyplývá z vyhodnocení podmínky. Rozlišujeme dva typy cyklů – iterace:

a) Cyklus s podmínkou na začátku
b) Cyklus s podmínkou na konci

ad 1) Cyklus s podmínkou na začátku - while



Pomocí cyklů realizujeme opakování určitého příkazu v závislosti na hodnotě podmínky. Tento cyklus testuje podmínku před samotným provedením těla. Z toho vyplývá, že cyklus nemusí proběhnout ani jednou. Cyklus se používá v případě, pokud nevíme, kolikrát v programu proběhne. Ukončovací podmínka tedy závisí na nějaké akci (příkazů) uvnitř těla cyklu. Obecně tedy konstrukce while vypadá:

while (podmínka) // Hlavička cyklu
Příkaz; // Tělo cyklu

Výraz tvořící podmínku musí platit pro pokračování cyklu. Bude-li podmínka nepravdivá, činnost cyklu se končí a běh programu pokračuje zpracováním nejbližšího příkazu za cyklem. Test na pravdivost výrazu se provádí ještě před započetím průchodu cyklem – může nastat situace, ve které výraz neplatí již před průchodem a tělo cyklu se nikdy nevykoná. Příkaz může být i složený. Konstrukce cyklu while se složeným tělem cyklu vypadá:

while (podmínka) // Hlavička cyklu
{ Příkaz_1; //Složené tělo cyklu

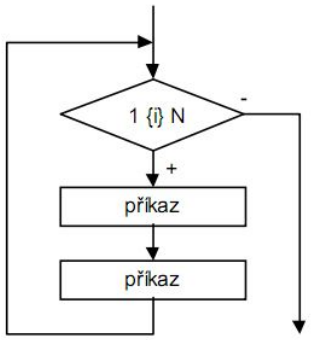
Příkaz_2;

...

Příkaz_n;

}

ad 2) Zvláštní forma cyklu s podmínkou na začátku – for



Cyklus s podmínkou na začátku se může zapsat také pomocí klíčového slova for. Používá se, pokud počet opakování známe předem. Jeho obecný zápis vypadá takto:

for (start_výraz; podmíněný výraz; iterační výraz) Příkaz;

Start_výraz je vyhodnocen pouze jedenkrát, ještě před vyhodnocením podmíněného výrazu. Slouží k inicializaci proměnných. Není povinný a může být i vypuštěn. Podmíněný výraz řídí opakování cyklu a je vyhodnocen ještě před vykonáním příkazu. Při neplatnosti podmínky nemusí být příkaz nikdy vykonán. I tento výraz může být vynechán. Cyklus s chybějícím podmíněným_výrazem bude nekonečný. Iterační výraz je vyhodnocován po každém průchodu cyklem. Může být vynechán. Při vynechání všech tří výrazů získáme nekonečný cyklus.

for (; ;) { //Tělo cyklu }

Můžeme se tomu vyhnout nadefinováním proměnných v tělu / venku cyklu a přidání podmínky

```
i=0;
for (  ;      ;  )
{
  if (i==10) break;
  i++;
}
```

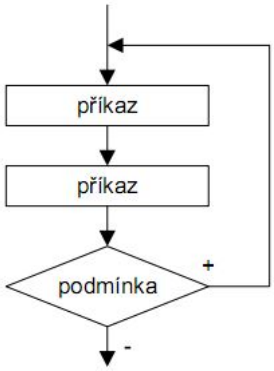
Cyklus je řízen řídící proměnnou. Jejím identifikátorem je nejčastěji i, j, k. Hodnoty řídící proměnné jsou omezeny počáteční start_výraz a koncovou podmíněný_výraz hodnotou cyklu.

for (i=1 ; i<=N ; i++) { // Tělo cyklu Prikazy;}

Pokud je hodnota řídící proměnné menší nebo rovna koncové hodnotě, pak se vykoná tělo cyklu, provede se návrat na začátek cyklu, zvýší se hodnota řídící proměnné o 1 (před každým

výpisem se podmínka ověřuje). Cyklus končí tehdy, když řídící proměnná mí hodnotu vyšší než je hodnota koncová.

b) Cyklus s podmínkou na konci – do while



Jinými slovy, je to cyklus s podmínkou na konci. Používá se v případech, kdy víme, že cyklus musí být proveden alespoň jednou, přičemž nemusíme znát kolik opakování přesně bude. Pokud program dospěje k cyklu, provede bez okolků všechny příkazy v jeho těle a potom narazí na podmínku. Pokud je tato podmínka splněna, program se vrací na začátek těla cyklu a ten se opakuje. V opačném případě je cyklus ukončen.

```
do {
  tělo cyklu
} while (výraz);
```

Na rozdíl od předchozího cyklu, tento provede o jedno vypsání více, protože ještě než dojde k testování podmínky, tělo cyklu je jednou provedeno. Jinak není v podstatě na tomto cyklu jiný významný rozdíl.

B. Příkazy break, continue

V cyklu for se **break** používá většinou v rámci nějaké podmínky. Když na něj program narazí, cyklus okamžitě skončí, i kdyby podmínka cyklu ještě platila. Můžeme jej použít ve všech typech cyklů. Příkaz **continue** může být použit také v jakémkoli druhu cyklu a pokud je proveden, přeruší právě probíhanou smyčku cyklu. Udělá skok na konec cyklu a vynutí jeho opakování, cyklus se teda neukončuje.

BREAK

```
int a = 0;
while (true)
{
```

```
  System.Console.WriteLine(a);
  a++;
  if (a == 5)
  break;
}
```

CONTINUE

```
int y = 0;
for (int x=1; x<101; x++)
{
  if ((x % 7) == 0)
  continue;
  y++;
}
```

4. Datové typy a proměnné v jazyce C a C++

Datový typ určuje pro proměnnou:

1. Množinu přípustných hodnot
2. Množinu operací
3. Velikost paměti

Jazyk C je striktně typový jazyk, kde musíme vždy každé proměnné určit nejen její název, ale i typ. Pokud použijeme například celočíselný datový typ int, pak mlžem prohlásit, že se hodnotami tohoto typu smí provádět všechny základní aritmetické operace.

1. Rozdělení datových typů v jazyce C a C++

2. Deklarace jednotlivých datových typů a charakteristika jejich vlastností

1) a) Jednoduché

Nemají vnitřní strukturu. Složitější typy jsou ve skutečnosti složeny z těchto základních.

Tyto typy jsou ordinální – hodnoty tohoto typu tvoří lineárně uspořádanou množinu, kde pro každý prvek je přesně definovaný předchůdce i následovník (z posledního prvku dojde k tzv. přetečení na první)

- **Celočíselné**
  - Short (16b), int (16b/32b dle typu CPU), long (32b), char (8b), bool (true=1, false=0)
  - V jazyce C existuje několik celočíselných typů lišícím se rozsahem hodnot, které v nich lze uchovat. Jednotlivé rozsahy se dají zjistit aplikací operátorů sizeof a jsou určeny překladači. Výsledek je v bajtech. Nejmenším typem, který je možno reprezentovat celá čísla je typ char, který zabere v paměti jeden byte a používá se často k ukládání znaků. Ty jsou pak v proměnných uloženy jako ASCII kódy znaků a lze s nimi i nadále pracovat jako s čísly.
- **Reálné**
  - Float (32b), double long (64b), double (80b)
  - Reálné čísla jsou v jazyce C reprezentovány třemi typy. Nejúspornější z nich, co se paměti týče je typ float. Poté existuje ještě typ double. Pro větší přenosnost pak ještě slouží long double.
- **Typy bez hodnoty**
  - Void
  - Nemá definovaný rozsah. Případy použití:
    - Návratová hodnota funkce – funkce nemá návratovou hodnotu *void main()*
    - Typ formálního parametru funkce – funkce nemá žádné formální parametry *int main (void)*

- **Výčtový**
  - Enum (8b / 16b / 32b)
  - Slouží k definici seznamu symbolických konstant, které jsou většinou vzájemně závislé
  - Čísluje se implicitně od nuly, ale lze je explicitně inicializovat
  - Definice s typedef je nejvíce používaná

```
typedef enum {
  MODRA, CERVENA, ZLUTA, ZELENA
} BARVY;
BARVY x = MODRA;
typedef enum {
  MODRA = 5, CERVENA = 8, ZLUTA, ZELENA = 15
} BARVY; // ZLUTA bude mít číslo 9
```

1) b) Strukturované

Datový typ obsahuje jeden nebo více prvků. Říkáme, že je homogenní, jsou-li prvky stejného typu – pole, řetězec. Jsou-li prvky různých datových typů, říkáme jim heterogenní – struktura, unie

- **Pole a řetězec**
  - Pole je kolekce proměnných stejného typu, které jsou označovány společným jménem. Pole v C jsou výhradně jednorozměrné.
  - Řetězec je jednorozměrné pole znaků ukončené speciálním znakem ve funkci zarážky = ukončovací nula – ´\0´

  Typ_promenne jmeno [cislo]; **deklarace pole**

  char s [SIZE]; **- deklarace řetězce**

- **Struktura**
  - Struktura vám umožní spojit několik datových typů do jednoho a pracovat s nimi jako s celkem.
  - Je možné přistupovat k prvkům a vytvářet strukturu v jiné struktuře. Struktury jsou velmi silným nástrojem C pro zpracování dat.

```
struct jmeno {
  dat_typ jmeno_polozky;
  dat_typ jmeno_polozky;
  ...
} promenne;
```

- **Unie**
  - Syntaxe typu union je stejná jako syntaxe struktury, až na to, že místo klíčového slova struct se použije klíčové slovo union. Význam jednotlivých položek je stejný. Jeho použití s konstrukcí typedef také.
  - Rozdíl tu však je. A zásadní. Zatímco struktura si vytvoří paměťové místo pro všechny položky, typ union zabírá v paměti jen tolik místa, kolik jeho největší položka. Z toho také vyplývá, že lze používat v jeden okamžik jen jednu položku.
  - Šetří paměť

```
union jmeno {
 dat_typ jmeno_polozky;
 dat_typ jmeno_polozky;
 ...
} promenne;
```

- **Soubor**
  - FILE
  - Soubor stdio.h obsahuje deklaraci typedef ve tvaru

```
Typedef struct { ... } FILE;
```

  - Proměnná fp je ukazatel na FILE a kompilátor zná detaily struktury tohoto typu

```
FILE *fp;
```

1) c) Zvláštní datový typ

- **Ukazatel (pointer)**
  - Pointer, česky ukazatel, je proměnná do které se ukládají výhradně adresy proměnných, funkcí, či obecněji paměťových bloků. Mohou být spjaty pouze s daným typem proměnné. Mluvíme pak o pointeru (ukazateli) na daný typ (pointer na int, pointer na double…)
  - Pointer na daný typ definujeme podobně jako obyčejnou proměnnou. Jediný rozdíl je v přidání hvězdičky před název proměnné

```
int *pi;
```

2) Definování nových datových typů pomocí typedef

Příkaz typedef je standartní příkaz C, díky kterému můžeme vytvářet aliasy jiných datových typů pro zjednodušené použití

typedef int nahrada;
nahrada X;

typedef struct {
int a, b, c;} nahrada;
nahrada X;

Tato konstrukce pouze definuje nový datový typ s názvem nahrada, ale tento typ má zcela stejné vlastnosti jako typ int. Dále deklarována proměnná X je datového typu náhrada, který je synonymem typu int.

5. Proměnné v jazyce C a C++

A. Charakteristika a deklarace lokálních a globálních proměnných

- Proměnné jsou paměťová místa přístupná prostřednictvím identifikátoru. Hodnotu proměnných můžeme během výpočtu měnit. Tím se proměnné zásadně odlišují od konstant, které mají po celou dobu chodu programu hodnotu neměnnou - konstantní. Proměnné deklarujeme uvedením datového typu, jež je následován identifikátorem, nebo seznamem identifikátorů, navzájem oddělených čárkami. Deklarace končí středníkem. Datový typ určuje její velikost v paměti, množinu hodnot a množinu operací.
- Proměnné deklarované mimo všechny funkce (tzn. i mimo main) jsou globální. Můžeme je použít kdekoliv v programu a jsou implicitně nulové.
- Proměnné deklarované ve funkcích nebo uvnitř bloku se nazývají lokální proměnné. Můžeme je použít v dané funkci či bloku a jsou nenulové. *Lokální proměnná vždy přebije proměnou globální!* Pokud ji programátor pouze deklaruje a neinicializuje, pak její obsah může být jakýkoliv. Bloky příkazů jsou ohraničeny složenými závorkami { a }.
- V jazyce C musí být všechny lokální proměnné deklarovány (nebo definovány) na začátku těla funkce, k níž patří. Toto omezení je platné pouze pro jazyk C, ovšem nikoliv pro další jazyky, jako jsou C++ nebo C#.

```
int x; // globální proměnná
int main() { int a; // lokální proměnná – uvnitř funkce main()
      {int b; // lokální proměnná. Její působení omezeno na blok.
      } // zde již proměnná b nelze použít
}
```

B. Typy charakteristiky a význam paměťových tříd v jazyce C

- Paměťová třída určuje, kde bude proměnná v paměti uložena, jakou bude mít viditelnost a jakou životnost. Paměťové třídy se zapisují v následujícím formátu:

```
název_pam_třídy datový_typ identifikátor_proměnné;
extern int a;
```

Existují tyto paměťové třídy: **auto**, **register**, **extern** a **static**

1) **AUTO**
   - Paměťová třída je implicitně používána pro všechny lokální proměnné. Jedná se o pozůstatek ze starších verzí.
2) **REGISTER**
   - Používá se pro lokální proměnné. Je alokována do registru v CPU, není-li v něm místo, alokuje se do zásobníku. Nelze použít operátor adresy & = nedá se načíst. Jelikož třída zvyšuje rychlost přístupu, používá se pro často používané proměnné (např. řídící proměnné cyklu).
3) **EXTERN**
   - Implicitní paměťová třída pro globální proměnné. Má využití při odděleném překladu. Je-li třeba, aby dva či více modulů sdílelo tutéž proměnnou v jednom souboru, bude definována proměnná bez slova extern a ve všech ostatních zapíšeme klíčové slovo extern.
4) **STATIC**
   - Jako globální se používá při odděleném překladu. Globální proměnné paměťové třídy static jsou viditelné pouze v modulu, ve kterém jsou definovány.

6. Zdrojové a hlavičkové soubory v jazyce C a C++

A. Stavba programu v jazyce C a C++

- Platí že v programu můžeme pracovat jen s tím, co překladač už zná. V C/C++ máme k dispozici tzv. hlavičkové soubory. Ty obsahují všechny informace, které překladač nezbytně potřebuje,aby mohl přeložit volání knihovní funkce, použití knihovního objektu apod.
- Standardní hlavičkové soubory v C++ jsou bez přípony, standardní hlavičkové soubory v jazyce C mají příponu .h. Pro uživatelem definované hlavičkové soubory se zpravidla používá přípona .h.
- Informace o základních vstupních a výstupních proudech jazyka C++ jsou v hlavičkovém souboru iostream (ve starších překladačích se jmenoval iostream.h). Aby je měl překladač k dispozici, musíme tento soubor vložit do svého programu. K tomu slouží direktiva #include, za kterou v lomených závorkách < a > následuje jméno souboru. Umožňuje vyčlenit některé části zdrojového kódu programu do zvláštních souborů, které lze opakovaně používat.
- Standardní knihovna jazyka C a standardní knihovna jazyka C++ tradičně deklarují své funkce v hlavičkových souborech.

Struktura programu:

,,hlavicka.h”

Hlavička

Příkazy preprocesoru (knihovny atd.) : #include<stdio.h>

Konstanty: #define N 10

Deklarace globálních proměnných: int a,b;

Definice globálních proměnných: int x=50;

Deklarace funkcí: int secti(int a, int b);

,,main.c”

Hlavní funkce: void main (void)

Hlavní program: {

Definice lokáních proměnných: int a;

Tělo programu

Konec hlavního programu: }

,,fce.c”

Definice funkce: int secti(int a, int b)

B. Stavba vlastního hlavičkového souboru

- Hlavičkové soubory obsahují makra, hlavičky funkcí, definice nových datových typů a globální proměnné. To vše musí být uvnitř podmíněného překladu, který zamezí vícenásobnému vložení obsahu hlavičkového souboru.

Podmíněný překlad:

Slouží k tomu, aby se hlavičkový soubor nevkládal vícekrát.

```
#ifndef SOUBOR_A
#define SOUBOR_A
...hlavičky
#endif
```

C. Obvyklá stavba souborů při tvorbě tříd

- Když tvořím třídu, deklaraci píšu do souboru s příponou .h, do .cpp píšu definici (metody).

7. Projekty

A. Projekty v programovacím jazyce C

- Projekt je celek složený z několika modulů obsahujících zdrojový kód.
- Projekty se zavádí, protože nám zjednoduší práci tím, že program rozdělíme do více modulů a tím ho zpřehledníme. Zároveň jsou také výhodné, protože provedeme-li změnu jen v jednom modulu, nemusíme kompilovat celý program, jak bychom museli udělat při změně jediného zdrojového souboru, ale stačí zkompilovat pouze daný modul. Tento fakt se nazývá oddělený překlad.

B. Práce s více moduly

- V programu, kde chceme používat proměnné nebo funkce jiného modulu, jednoduše includujeme hlavičkový soubor tohoto modulu.
- Tím do programu vložíme všechny informace, které jsou potřebné pro úspěšný překlad.
- Nakonec se přeložené soubory všech použitých modulů spojí linkerem do jednoho spustitelného souboru.
- Většina dnešních překladačů již ale přímo podporuje práci s odděleně překládanými moduly (formou tzv. „projektů"), a tak se uživatel o linkování souborů většinou vůbec nemusí starat a stačí mu pouze přidat soubory modulů do projektu.

Rozdíl mezi vkládáním souborů a odděleným překladem:

a) Vkládání souborů

obrázek

- Všechny soubory se vloží do hlavního souboru a překládají se společně
- Musí se zbytečně překládat i soubory, které nebyly změněny

b) Oddělený překlad = práce s moduly

obrázek

- Výhodné při rozsáhlejších programech a možnost práce více programátorů

- Po překladu jsou pak vzniklé soubory (moduly) spojeny do jednoho programu tzv. *linkerem*
- Při dalších překladech jsou již kompilovány pouze ty soubory, ve kterých skutečně došlo ke změně, což celý proces značně urychluje

C. Spojování a překlad programu

- Oddělený překlad znamená, že se každý soubor přeloží zvlášť, vznikne více .obj souborů a ty se spojí do jednoho programu pomocí linkeru.
- Výhodou je, že stačí překládat pouze soubor, který se změnil a při práci v týmu lze použít jen .obj soubor, čímž se zabrání např. nechtěné modifikaci cizích kódů.

8. Struktura, union, výčet v jazyce C

A. Struktura

- Je heterogenní datový typ (je složen z libovolného počtu prvků různých datových typů)
- Struktura vám umožní spojit několik datových typů do jednoho a pracovat s nimi jako s celkem.
- Je možné přistupovat k prvkům a vytvářet strukturu v jiné struktuře. Struktury jsou velmi silným nástrojem C pro zpracování dat.

Definice

1. Struktura není pojmenována, nedá se využít dále v programu, lze používat pouze definované proměnné.

```
struct {
      int vyska;
      float vaha;
} pavel, honza, karel;
```

2. Struktura je pojmenována, dá se využít dále v programu

```
struct miry {
      int vyska;
      float vaha;
} pavel, honza, karel;
```

3. Předchozí způsob, kdy je oddělena definice struktury od definic proměnných. Lze vícekrát definovat proměnné

```
struct miry {
      int vyska;
      float vaha;
};
struct miry pavel;
struct miry honza, karel;
```

4. Definice struktury jako nový datový typ pomocí příkazu *typedef*, k definici proměnných není potřeba klíčové slovo *struct*

```
typedef struct {
      int vyska
      float vaha;
} MIRY;
MIRY pavel, honza, karel;
```

5. Definice pomocí *typedef*, kdy je struktura pojmenována, používá se, když struktura odkazuje na sebe sama

```
typedef struct polozka {
      int hodnota;
      struct polozka *dalsi;
} POLOZKA
```

Použití

- Přístup k prvkům pomocí tečkové notace

```
Pavel.vyska = 186;
```

- Pole struktur

```
MIRY lide[100]; //definice
Lide[50].vyska = 176; //přístup
```

- Je možné pracovat s celou strukturou najednou (honza = pavel – kopie celé struktury najednou)
- Pole ve struktuře

```
typedef struct {
      int pole[10];
} STR_POLE;
STR_POLE x;
x.pole[1] = 5;
```

- Pointer na strukturu
  - Při práci se strukturou ve funkci
  - Při práci se strukturami v dynamické paměti

```
typedef struct {
      char jmeno[30];
      int rocnik;
} STUDENT;
STUDENT s, *p_s;
```

- Přístup k dynamické struktuře

```
s.rocnik = 3; //pomocí jména struktury
(*p_s).rocnik = 4; //pomocí pointeru;
p_s->rocnik = 4; //pomocí pointer jednodušeji
```

- Inicializace struktury

```
MIRY a = {190, 84.3};
```

B. Union

- Syntaxe typu union je stejná jako struktury až na to, že místo klíčového slova struct se použije klíčové slovo union. Význam jednotlivých položek je stejný. Jeho použití s konstrukcí typedef také.
- Rozdíl tu však je. A zásadní. Zatímco struktura si vytvoří paměťové místo pro všechny položky, typ union zabírá v paměti jen tolik místa, kolik má největší položka. Z toho také vyplývá, že lze používat v jeden okamžik jen jednu položku.
- Šetří paměť

```
typedef union {
      char c;
      int I;
      float f;
} TEST;
```

- Union neposkytuje informaci o prvku, který do něj byl naposledy uložen

C. Enum

- Slouží k definici seznamu symbolických konstant, které jsou většinou vzájemně závislé

```
typedef enum {
      MODRA, CERVENA, ZELENA, ZLUTA
} BARVY;
BARVY x = MODRA;
```

- Čísluje se implicitně od nuly, lze explicitně inicializovat (vždy by se měli inicializovat všechny prvky)

```
typedef enum {
      MODRA = 5, CERVENA = 8, ZELENA, ZLUTA = 15
} BARVY; //ZELENA bude mít číslo 9
```

9. Operátory v jazyce C a C++

Operand je hodnota, se kterou počítáme (konstanta, proměnná, výsledek funkce). Operátor určuje operaci, kterou hodláme provést s operandem nebo operandy.

A. Rozdělení operátorů

1. Podle funkčnosti operátorů – aritmetické, logické, relační, přiřazovací, bitové, operátory bitového posuvu, přístupové, oddělovací
2. Podle počtu operandů – unární, binární, ternární

Arita operátoru udává počet jeho operandů, čili počet výrazů, se kterými se daná operace provádí. V jazyku C máme k dispozici operátory unární (tj. s aritou 1), binární (tj. s aritou 2) a jeden ternární operátor (tj. s aritou 3).

Unární operátory

- +a, -a Unární plus a mínus (určuje znaménko čísla)
- !a Logická negace (výsledek TRUE nebo FALSE)
- *a Dereference (získání hodnoty objektu dle adresy = pointer)
- &a Reference (získání adresy objektu)
- ~ Bitová negace (~00000101 je 11111010)
- sizeof() Operátor pro získání délky objektu nebo typu
- (typ) Přetypování
- a++,a—Inkrementace a dekrementace
  - ++vyraz – Inkrementace před použitím
    - Výraz je nejprve zvětšen o 1 a pak je nová hodnota vrácena jako hodnota
  - vyraz++ - Inkrementace po použití
    - je vrácena původní hodnota výrazu a pak je výraz zvětšen o 1

```
int i = 6, j;
j = ++i;        // j = 7, i = 7
j = i++;        // j = 7, i = 8
```

Binární operátory

- **<,> ; <= ; >= ; == ; != ; + ; - ; * ;** / - u dělení rozhoduje typ operandů (je-li alespoň jeden z nich typu float bude dělení reálné)
- **%** - zjištění zbytku po celočíselném dělení (modulo)
- **&&** - logické AND
- == - rovnost
- = - přiřazení
- **!=** - nerovnost

Ternární operátor

Nejčastěji využívány u podmíněného operátoru, který se nejčastěji používá pro zkrácení zápisu vyhodnocené podmínky.

(podmínka) ? pravda : nepravda

B. Priorita operátorů

Priorita stanovuje, která část výrazu se vyhodnotí dříve a která později. Pro každý typ operátoru je přesně určeno, v jakém pořadí se budou vyhodnocovat jeho operandy. Prioritu vyhodnocování udává tabulka preferencí. Řada operátorů má stejnou prioritu vyhodnocování. Pokud se na jedné úrovni výrazu vyskytuje více operátorů téže priority, jsou příslušné operace vyhodnocovány podle jejich asociativity zleva doprava nebo zprava doleva.

Z důvodů vyhnutí se chybám v prioritě vyhodnocování je lepší vždy používat závorky!

10. Přetěžování operátorů

A. Pravidla pro přetěžování operátorů

- Nelze přetížit operátory :: , ? , : , . , .* , sizeof
- Nelze definovat nové operátory (např. vzít znak @ a definovat jej jako operátor)
- Nelze měnit obecné charakteristiky operátorů – počet operandů, prioritu
- Nelze měnit definici operátorů pro standartní datové typy

B. Přetěžování operátorů bez tříd

1. Ve strukturách

Příklad (sčítaní komplexních čísel):

```
typedef struct
{
    int re,im;
}KOMPLEX;

KOMPLEX operator+(KOMPLEX a, KOMPLEX b)
{
z.re,z.im)
    KOMPLEX pom;
    pom.re = a.re + b.re;
    pom.im = a.im + b.im;
    return pom;
}

z = operator+(x,y);
```

```
int main()
{
        KOMPLEX x,y,z;
        x.re=2; x.im=3;
        y.re=4; y.im=5;
                z=x+y;
        printf("%d+%d          i",

        return 0;
}
```

C. Přetěžování operátorů pro třídy

- Při přetěžování operátorů pro třídy musí operátorská funkce pracovat s atributy, které jsou většinou privátní
- Pro přetěžování operátorů ve třídách píšeme operátorská funkce jako metody třídy
- Pokud operátorská funkce není metodou třídy, musí se definovat ve třídě jako friend
    - Mohou pak pracovat s privátními atributy, ale musí být jako metody volány (tzn. Pro daný objekt)

Přátelé (friends)

- Obcházejí pravidla přístupových práv k členským datům třídy
- Mají plná přístupová práva ke všem členům dané třídy, i když jimi nejsou
- Mohou být jednotlivé funkce, metody jiných tříd nebo u celé třídy
- Označují se slovem friend
- Nemohou pracovat s ukazatelem this

1. Přetěžování unárních operátorů (1 operand)

MIMO TŘÍDU(FRIEND):

Definice: typ **operator@**(operand)

Volání: **operator@(operand)** nebo zkráceně **@operand**

```
class KOMPLEX{
        private:
                int re,im;
        public:
                KOMPLEX(int r,int i){re=r; im=i}
                void vypis() { if (im<0) cout << re << im << "i" << endl;
                friend bool operator! (KOMPLEX a);
};

bool operator! (KOMPLEX a){
        if(a.re==0 && a.im==0)
                return true;
        else return false;
}

int main()
{
        KOMPLEX prvni(0,0);
        if(!prvni)
        {
                cout << "Komplexni cislo";
```

```
            prvni.vypis();
            cout << " je nulove" << endl;
        }
}
```

JAKO METODA UVNITŘ TŘÍDY:

Definice: typ **operator@()**

Volání: **operator@()** nebo zkráceně **@operand**

```
class KOMPLEX{
        private:
            int re,im;
        public:
            KOMPLEX(int r,int i) {re=r; im=i;}
            void vypis () {if (im<0) cout << re << im << "i" << endl;
            KOMPLEX operator-() {return KOMPLEX(-re,-im)};
};

int main()
{
        KOMPLEX prvni(2,3), druhe(1,1);
        prvni.vypis();
        druhe=-prvni; // NEBO druhe=prvni.operator-();
        druhe.vypis();
}
```

2. Přetěžování binárních operátorů (2 operandy)

MIMO TŘÍDU(FRIEND):

Definice: typ **operator@(op1, op2)**

Volání: **operator@**(operand1, operand2) nebo zkráceně **operand1@operand2**

```
class KOMPLEX
{
        private:
```

```
        int re,im;
    public:
        KOMPLEX(int r,int i) {re=r; im=i;}
        friend KOMPLEX operator+ (KOMPLEX a, KOMPLEX b);
};
KOMPLEX operator+ (KOMPLEX a, KOMPLEX b){return
KOMPLEX(a.re+b.re,a.im+b.im); }


int main()
{
    KOMPLEX prvni(2,3), druhe(1,1), treti(0,0);
    treti=prvni+druhe; // nebo treti=operator+(prvni,druhe);
}
```

JAKO METODA UVNITŘ TŘÍDY:

Definice: typ **operator@(op2)**

Volání: **operand1.operator@(operand2)** nebo zkráceně **operand1@operand2**

```
class KOMPLEX
{
    private:
        int re,im;
    public:
        KOMPLEX(int r,int i) {re=r; im=i;}
        KOMPLEX operator+(KOMPLEX b) {return KOMPLEX(re+b.re,
im+b.im); }
        friend KOMPLEX operator- (KOMPLEX a, KOMPLEX b);
};

KOMPLEX operator-(KOMPLEX a, KOMPLEX b)
{ return KOMPLEX(a.re-b.re, a.im-b.im); }
```

```
int main()
{
        KOMPLEX prvni(2,3), druhe(1,1), treti(0,0);
        treti=prvni+druhe;      // NEBO treti=prvni.operator+(druhe)
        treti=první-druhe;      // NEBO operator-(prvni,druhe);
}
```

T …… Náš datový typ nebo třída
t1,t2 …… jsou proměnné nebo objekty
i ……. je celé číslo

Příklady deklarací přetížených operátorů metodami

t1 + t2	T operator+(const T &t2) const;
i + t1	**nelze**, první operand musí být typu T, řešíme pomocí funkce friend
t1 + i	T operator+(const int i) const;
t1 += t2	T& operator+=(const T &t2);
t1 += i	T& operator+=(const int i);
-t1	T operator- () const;
++t1	T& operator++(); //preinkrementace
t1++	T& operator++(int i); //postinkrementace
t1<=t2	bool operator<=(const T &t2) const;
t1!=i	bool operator!=(const int i) const;
t1 = t2	T& operator=(const T &t2); //přiřazení
výstup	**nelze**, první operand musí být typu ostream, ne T nutno řešit pomocí friend funkce
vstup	**nelze**, první operand musí být typu istream, ne T nutno řešit pomocí friend funkce

Příklady deklarací přetížení operátorů friend funkcemi

t1 + t2	T operator+ (const T &t1, const T &t2);
i + t1	T operator+ (const int i, const T &t1);
t1 + i	T operator+ (const T &t1, const int i);
t1 += t2	T& operator+= (T &t1, const T &t2);
t1 += i	T& operator+= (T &t1, const int i);
-t1	T operator- (const T &t1);
++t1	T& operator++ (T &t1); //preinkrementace
t1++	T& operator++ (T &t1, int i); //postinkrementace
t1 <= t2	bool operator<=(const T &t1, const T &t2);
t1!=i	bool operator!=(const T &t1, const int i);
výstup	ostream& operator<< (ostream& os, const T& t1);
vstup	istream& operator>> (istream& is, T& t1);

11. Funkce v jazyce C a C++

12. Činnost preprocesoru v jazyce C

A. Charakteristika činnosti preprocesoru

- Preprocesor zpracovává hlavičkové soubory, rozvíjí makra, nepropouští komentáře a umožňuje provádět podmíněný překlad zdrojového textu. Preprocesor předchází překladač. Preprocesor zpracovává vstupní text jako text, provádí v něm textové změny a jeho výstupem je opět text. Preprocesor přijímá následující direktiva:

#define	#elif	#else	#endif
#error	#if	#ifdef	#ifndef
#include	#line	#pragma	#undef

Direktiva preprocesoru **není** příkaz jazyka C. **Neukončujeme** ji proto středníkem. Direktiva preprocesoru **musí** být vždy uvozena znakem **#**. Každá direktiva musí být na samostatném řádku zdrojového kódu. Může jí předcházet libovolný počet bílých znaků a musí být ukončena přechodem na nový řádek. Na jednom řádku nemohou být dvě direktivy nebo zároveň direktiva a příkaz.

B. Tvorba maker, srovnání makra a funkce

V programech se využívají velice často a to proto, že zbavují programy tzv. *Magických čísel*. Tzn. Konstant, které se bez vysvětlení objevují v programu. Většinou jsou konstanty použity na začátku programu a jejich rozumné použití zvyšuje čitelnost zdrojového kódu. Oproti funkcím mají také výhodu šetření paměti.

Makra bez parametrů:

- Velká písmena
- Pro symbolické konstanty (PI)

```
#define PI 3.14159  //definice
#undef PI      //rušení definice
#define AHOJ printf(„AHOJ“); //i toto lze
```

Makra s parametry

- Malá písmena
- Může mít více parametrů

```
#define mocnina (x) ((x)*(x))
//volání
```

```
A = mocnina (x+2);
```

C. Podmíněný překlad

Slouží k tomu, aby se hlavičkový soubor nevkládal vícekrát.

```
#ifndef SOUBOR_A
#define SOUBOR_A
#include <header.h>
#endif
```

D. Hlavičkové soubory

Hlavičkové soubory jsou v jazyce C určeny pro všechny definice, které ve výsledném přeloženém programu nezabírají žádné místo. Jedná se obvykle o definice maker, symbolů, vkládání dalších hlavičkových souborů a podobně. Obsahují všechny informace, které překladač nezbytně potřebuje, aby mohl přeložit volání knihovny funkce, použití knihovního objektu apod. Standardní hlavičkové soubory v C++ jsou bez přípony .h, standardní hlavičkové soubory v jazyce C mají příponu .h. Standardní knihovny jazyk C a standardní knihovna jazyka C++ tradičně deklarují své funkce v hlavičkových souborech.

```
#include <header.h> //vložení hlavičkového souboru primárně v C
```

Vlastní hlavičkový soubor

Vlastní knihovna musí být ve stejném adresáři jako program. Využívá se zde podmíněný překlad, aby se nevkládal vícekrát.

13. Pointery v jazyce C a C++

A. Deklarace, charakteristika a využití pointerů

- Pointer je statická proměnná, jejíž obsahem je adresa v paměťovém prostoru
- Je nutná alokace paměti nebo přiřazení adresy již existující paměti
- Nulový pointer *NULL* – pro označení, že pointer nikam neukazuje
- Adresu pointeru v příkazu printf vypíšeme pomocí formátovacího znaku %p
- Operátor reference **&** - slouží k získání adresy proměnné
- Operátor dereference ***** - slouží k přístupu k obsahu, který je uložený na adrese, kde ukazuje pointer (zapsání nebo přečtení hodnoty)

```
Int *p, číslo = 4, x;
*p = 3; //nelze, protože pointer má uloženou náhodnou adresu
P = &číslo; //p obsahuje adresu proměnné číslo
X = *p; //přečtení hodnoty, x = 4
*p = 140; //zápis hodnoty, číslo se teď rovná 140, ale x je stále 4
```

B. Dynamické přidělování paměti v C a C++

Dynamické přidělování paměti znamená přidělování paměti až za chodu programu. Tímto způsobem lze např. řešit problémy, kdy hodláme v rámci našeho programu použít pole, jehož velikost má být určena až při provádění programu a nikoli již za překladu, Dynamickou paměť přidělujeme z **haldy**. Velikost hlady je v principu daná dostupnou (v daném okamžiku nevyužitou) částí paměti RAM. Velikost **haldy** může být ovlivněna i nastavením kompilátoru.

Alokace paměti

JAZYK C

```
Int *p;
P = (int*)malloc(sizeof(int)*pocet_prvku);
*p = 50;
Free (p);
P= NULL; //zakotvení
```

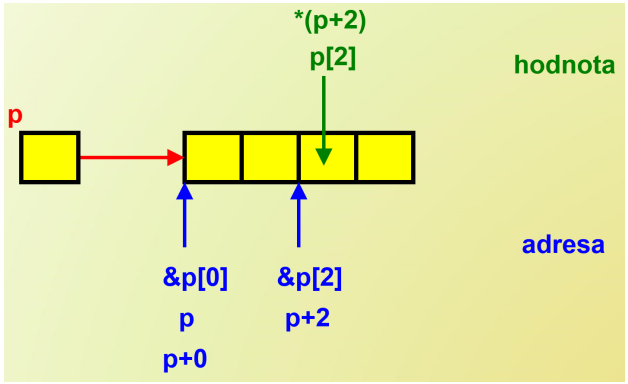
JAZYK C++

```
Int *p;
P = new int[pocet_prvku];
*p = 50;
Delete [] p;
P = NULL; //zakotvení
```

Funkce malloc() je funkce, která alokuje paměťový prostor v **haldě**. Jejím parametrem je počet bytů potřebný pro alokaci. Ten jsme zjistili operátorem pro zjištění velikosti datového typu sizeof(). Jelikož funkce malloc() vrací ukazatel na datový typ void (generický poiter na libovolný datový typ) je nutné tento pointer přetypovat pomocí operátoru přetypování (datový_typ*). **Nesmíme zapomenout odalokovat paměť.**

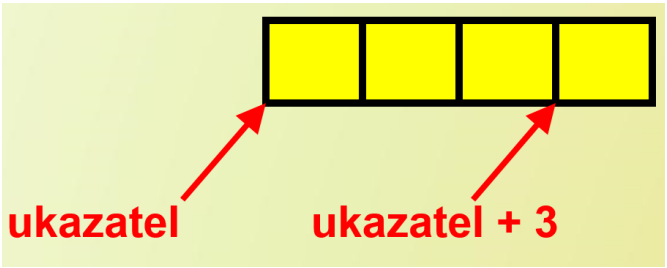
C. Pointerová aritmetika

Využití: Pointerovou aritmetiku používáme k přístupu jednotlivých prvkům pole



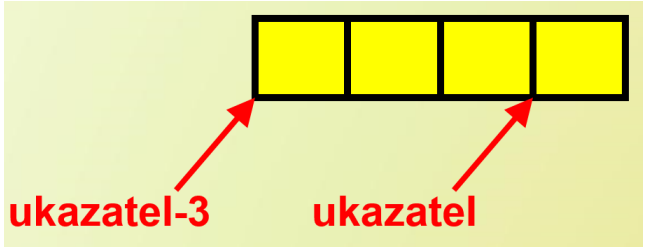
Součet (rozdíl) pointeru a celého čísla

Výsledek bude adresa n-tého prvku za (před) pointerem



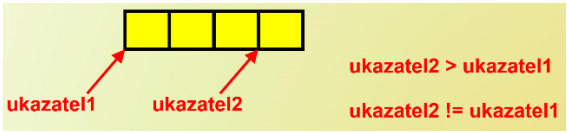
Rozdíl dvou pointerů:

Má smysl pouze tehdy, pokud ukazují na stejný celek paměti. Tento rozdíl vrací počet položek mezi nimi.



Porovnání dvou pointerů:

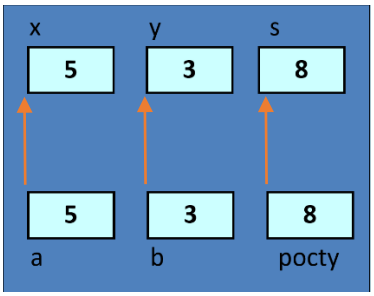
Má smysl pouze tehdy, pokud ukazují na stejný celek paměti a jsou stejného typu. Tento rozdíl vrací počet položek mezi nimi.

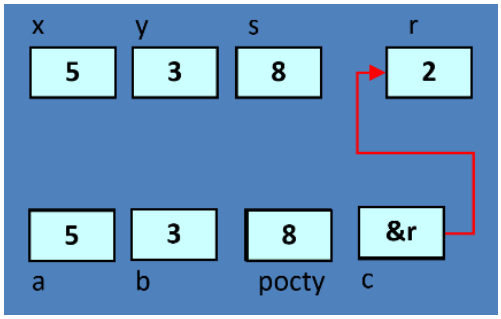


D. Pointery a funkce

```
Int pocty(int a, int b);
Int main()
{
     Int x = 5, y = 3, s;
     S = pocty(x,y); //skutečné parametry
     Printf („Součet je %d“,s);
     Return 0;
}
Int pocty(int a, int b) //formální parametry
{
     Return(a+b);
}
```

Skutečné parametry předají hodnotu formálním parametrům. Po skončení funkce končí platnost lokálních proměnných funkce pocty – jsou odalokovány. Pomocí return můžeme vracet jen jednu hodnotu! **NELZE:** return(a+b, a-b);





Pointery využíváme ve funkcích, pokud chceme, aby funkce vracela více než jednu hodnotu.

```
Int pocty(int a, int b, int *c){
     *c = a-b;
     Return(a+b);
}
Int main(){
     Int x = 5, y = 3, s, r;
     S = pocty(x,y,&r);
     Printf(„Součet je %d“, s);
     Printf(„Rozdíl je %d, r);
     Return 0;}
```

14. Jednorozměrné pole v jazyce C

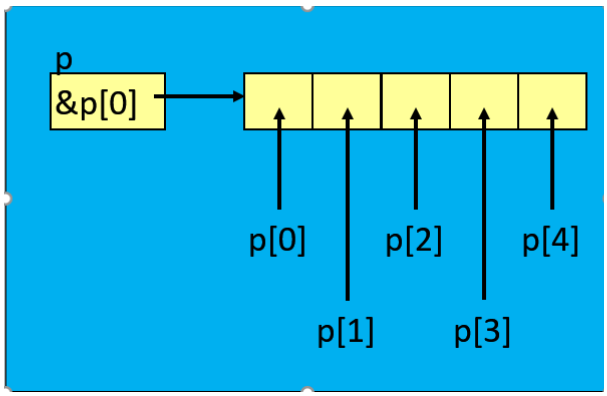
- Pole je uspořádaná množina hodnot stejného datového typu, tj. homogenní strukturovaný datový typ
- Prvky pole jsou indexovány od nuly
- Těsný vztah s pointery (umožňuje kouzla)

A. Deklarace a charakteristika statického pole

- V době překladu musí být znám počet prvků pole

Deklarace

```
Int p[5];
```



Vytvoří pětiprvkové pole celých čísel a pointer p, který ukazuje na toto pole (obsahuje adresu prvního prvku pole). Vše se nachází na zásobníku. Pointer p je statický a po celý běh programu neměnný!

K jednotlivým prvkům pole přistupujeme pomocí indexů. Jazyk C vždy indexuje od nuly a nekontroluje meze pole.

`Int pole[4] = {8,2,4,3};` - Pole se může při deklaraci inicializovat – naplnit.

`Int pole [] = {10,20,30};` - Překladač vytvoří tříprvkové pole podle počtu hodnot v závorce.

```
Int a;
```

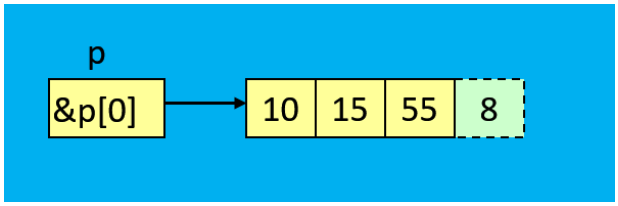
`Float pole[a];` - Nelze provést, protože v době překladu neznáme hodnotu proměnné a nevíme, kolik prvků je potřeba alokovat.

Plnění pole

```
Int p[3];
P[0]=10;
Scanf(„%d“,&p[1]); //15
Srand(time(NULL));
```

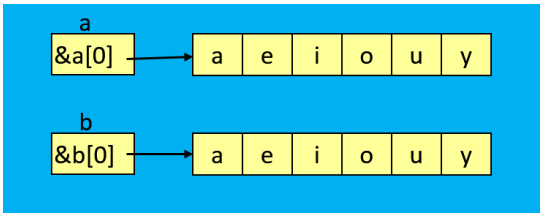
```
P[2]=rand()100; // třeba se vygeneruje 55
P[3]=8;
```

Zapíše hodnotu na nevyalokované místo v paměti – C nekontroluje meze pole



Kopírování prvků pole

```
char a[ ]={'a', 'e', 'i', 'o', 'u', 'y'}, b[6];
for (i=0;i<6;i++)
   b[i]=a[i];
```



Výpis pole

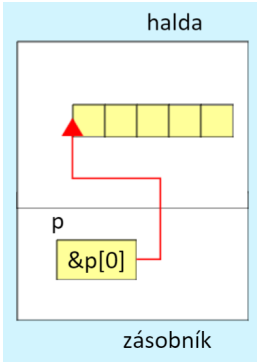
```
Float a[5]={0,2,4,6};
For (i=0;i<5;i++)
Printf(„ %f",a[i]);
```

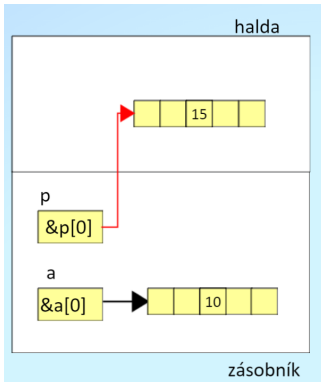
B. Deklarace a charakteristika dynamického pole

- Při deklaraci musíme znát jen datový typ prvků pole
- Nemusíme znát počet prvků
- Prvky musíme alokovat a odalokovat sami

```
Int *p;
```

Vytvoří pointer na celé číslo a je jen na nás, jestli bude ukazovat na jedno číslo nebo na celé pole. Práce se statistickým a dynamickým polem je stejná. Rozdíl je v alokaci a odalokaci pole a jeho umístění v paměti.






```
Int a[5], *p;
P=(int*)malloc(sizeof(int)*5);
…
Free(p);
P=NULL;
```

C. Třídící algoritmus

- Používají se pro rychlejší vyhledávání a přehlednost

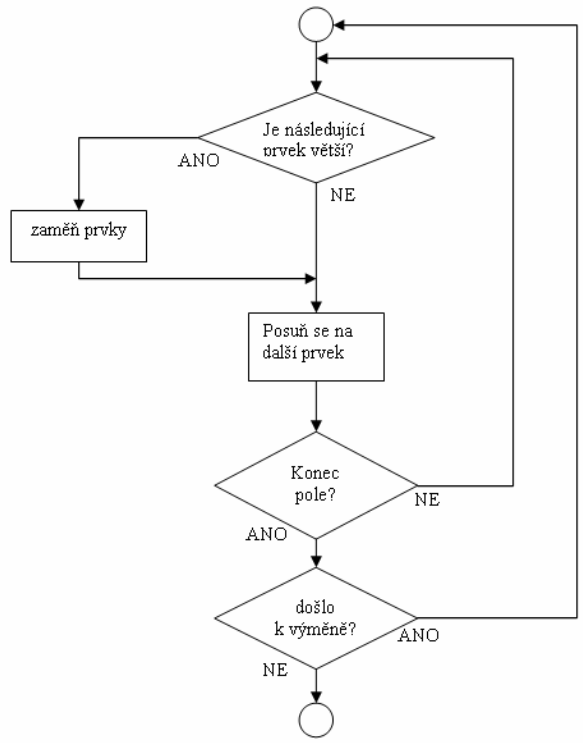
Základní druhy

- Řazení výběrem (vybere se jedna položka a umístí se na konec)
- Řazení vkládáním (vybraná položka se umístí na konkrétní místo)
- Řazení záměnou (dvě položky se prohodí)
- Řazení slučováním (rozdělení na více částí, každá část se třídí samostatně a nakonec se částí sloučí tak, ať je i výsledek seřazen)

Přehled běžných univerzálních algoritmů řazení

Název		**Metoda řazení**
Anglicky	**Česky**	
Bubble sort	Bublinkové řazení	Záměna
Heapsort	Řazení haldou	Halda, záměna
Insert sort	Řazení vkládáním	Vkládání
Merge sort	Řazení slučováním	Slučování
Quicksort	Rychle řazení	Záměna
Selection sort	Řazení výběrem	Výběr
Hell sort	Shellovo řazení	vkládání

Bubble sort




```
#define MAX 5
Inz i,j,a,pole[MAX]={8,5,9,3,1};
For (j=0;j<MAX-1; j++)
{
     For (i=0;i<MAX-1;i++)
     {
          If (pole[i]>pole[i+1])
          {
               A=pole[i];
               Pole[i]=pole[i+1];
               Pole[i+1]=a
          }
     }
}
```

D. Pole jako parametr funkce

Vytváříme-li funkci pro zpracovávání pole (např. pro zjištění maximální hodnoty nebo pro uspořádání pole), posíláme do funkce toho pole jako její parametr. Je-li pole parametrem funkce, předává se do funkce **jen adresa pole – adresa prvního prvku pole, ne celé pole**. Jde

tedy o parametr volaný odkazem. To znamená, že jakákoliv změna způsobená v poli ve funkci, v poli zůstane i po skončení této funkce.

Obecný zápis hlavičky funkce, jejíž parametr je statické pole takovýto:
Dat_typ id_fce(dat_typ pole[], int počet,…)

Hlavička funkce, jejíž parametr je dynamické pole se zapisuje obdobně:
Dat_typ id_fce(dat_typ *pole, int počet,…)

Z těchto zápisů je zřejmé, že je nutné zavést jak parametr funkce taky celočíselnou proměnou počet, která uchovává informaci počtu prvků pole a slouží tak pro kontrolu mezí.

Volání funkce v hlavní části programu se provádí takto:

Id_fce(pole,počet,…);

15. Vícerozměrné pole v jazyce C

16. Znaky a řetězce v jazyce C a C++

17. Soubory v jazyce C a C++

18. Hlavní rysy a rozdíly jazyků C a C++

A. Historie a vývoj jazyků C a C++

Historie jazyka C

- Vznik 1972
- Je univerzální programovací jazyk pro všechny operační systémy a procesory
- Má úsporné vyjadřování, programy jsou lehce přenositelné
- Vychází z něj další programovací jazyky (C++, Java, PHP apod.)
- Vyvinuli jej Ken Thompson a Dennis Ritchie pro potřeby OS UNIX
- Vyvinul se z nižších programovacích jazyků Frotran, Assembler, CPL, BCPL, B.
- Duální povaha:
  - Jako strukturovaný jazyk vysoké úrovně – lze v něm vytvořit vlastní datové typy a uživatelské aplikace
  - Jako Assembler nízké úrovně – lze v něm tvořit OS, kompilátory apod.

Historie jazyka C

- Vznik 1983
- Je objektově orientovaný programovací jazyk
- Byl vyvinut v Bellových laboratořích AT&T počátkem 80. let rozšířením jazyka C, zachována kompatibilita s jazykem C, ne však striktně.
- Základním krokem od C k C++ bylo zavedení tříd, což otevřelo cestu do světa OOP.

B. Strukturované programování, OOP

Strukturované programování

Program je chápán jako jedna funkce obsahující vstupní parametry mající jediný výstup. Tato funkce pak může být dále rozložitelná na podfunkce. Popisuje program pomocí posloupnosti příkazů a určuje přesný postup – algoritmus – jak danou úlohu řešit.

Objektově orientované programování

Kombinuje strukturované programování s výkonnými novými koncepty – objekty, které dovolí program organizovat mnohem efektivněji. Každý objekt je samostatný a nezávislý, obsahuje svoje instrukce a data. Umožňuje dekompozici problému na základní prvky a také práci s rozsáhlejšími programy.

C. Rozdíly jazyků C a C++

1. Názvy základních souborů

   Zdrojové soubory v jazyku C++ mají příponu cpp. U hlavičkových souborů se většinou požívá přípona .h stejně jako v jazyku C.

2. Nový datový typ bool

   V jazyku C++ byl zaveden nový datový typ bool. Jedná se o logický datový typ, který může nabývat hodnot true a false. Ve skutečnosti se za těmito klíčovými slovy schovávají celočíselné hodnoty 1 a 0.

3. Rozlišovací operátor

   Unární operátor :: (čtyřtečka) umožňuje pracovat s globálními jmény, když jsou zastíněná lokálními deklaracemi.

```
int k=5;
void main(){
      int k=3;
      printf("%d %d", ::k, k); //vypíše: 5 3
}
```

::k je odkaz na globální proměnnou

4. Deklarace proměnných

   Zatímco v jazyku C je nutné deklarace uvádět vždy na začátku bloku, v C++ je můžeme umístit, kde zrovna potřebujeme. Jejich platnost pak končí s koncem bloku, ve kterém jsou deklarovány. Pokud tedy za tímto blokem deklarujeme proměnnou se stejným názvem, jedná se již o novou proměnnou a její obsah už s obsahem minulé proměnné nemá nic společného.

5. Implicitní hodnoty parametrů funkcí

   Zatímco v jazyku C jsme museli při volání funkce vždy uvádět hodnoty všech parametrů, v jazyku C++ můžeme parametrům přiřadit implicitní (výchozí, přednastavené) hodnoty.

6. Předávání parametrů funkcí referencí

   Používá se zde operátor & jako označení reference. Referenci lze chápat jako „alias“, to je náhradní jméno proměnné.

```
void zamen(float &a, float &b);
int a;
int &b = a; // b je jiný název pro a
```

7. Dynamická alokace paměti

   V jazyku C jsme dynamicky alokovali místo v paměti pomocí funkce malloc a odalokovali pomocí free. Tyto funkce existují v jazyku C++ nadále. Byla však zavedena nová možnost alokace pomoci operátorů new a delete. Nesmíme tyto možnosti míchat!

```
int *p = new int[10];
delete []p;
p=NULL;
```

19. Základy OOP

A. Charakteristika pojmu objekt a třída

OOP vychází z předpokladu, že každý program je simulací reálného či virtuálního světa a každý svět je tvořen objekty.

Třída

- Zobecňuje a popisuje vlastnosti množiny podobných objektů.
- Je v podstatě synonymem pro datový typ.

Objekt

- Entita, která má svou identitu (lze jej odlišit od jiného objektu).
- Zjednodušený pohled na skutečný předmět, osobu či pojem.
- Má své vlastnosti (atributy, členská data) a své metody (funkce).
- Liší se od ostatních objektů svou jedinečností, ale sdílí s nimi společné vlastnosti a podobný způsob chování.

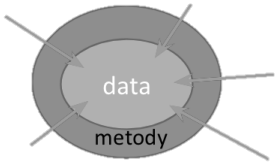
Příklad

- Třída auto popisuje základní vlastnosti a chování všech aut (obecná šablona).
- Objekt auto Škoda Fabia s SPZ 1T2 3456 je instancí třídy auto (konkrétní výskyt).

B. Vlastnosti OOP – zapouzdření, dědičnost a polymorfismus

Zapouzdření

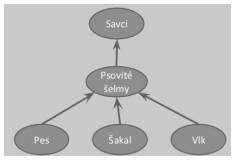
- Spojuje data s metodami obsaženými v objektu.
- Umožňuje skrýt atributy a metody objektu před zbytkem programu.



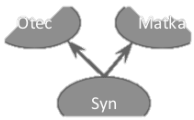
Dědičnost

- Jde o možnost odvozovat nové třídy, které dědí data a metody z jedné, nebo více tříd.
- Určuje konkretizaci tříd potomků, v odvozených třídách je možno přidávat, nebo předefinovat nová data a metody.
- Vytváříme stromovou hierarchii tříd, třídu v nejvyšší úrovni označujeme jako kořenovou třídu

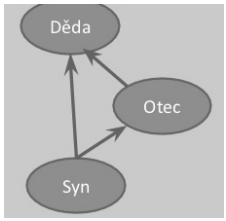
- Dělí se na:
    - Jednoduchá – 1 předek



    - Vícenásobná – více předků



    - Opakována – více cest



Polymorfismus

- vícetvarost
- objekty (instance různých tříd) na stejný podnět – na vyvolání stejně nazvané metody, například tisk, reagují různě.

C. Tvorba třídy na základě dědičnosti

```
class Osoba{
     private: int vek;
     public:
          Osoba(){vek=20;}
          Osoba(int  v){vek=v}  //  (int  vek){this-
>vek=vek}
          Osoba(vypis){cout<<"Vek:"<<vek;}
};
class Student:pulbic Osoba{
     private: float prumer;
     public:
          Student(){prumer=2.1;}
          Student(float  p,  int  v):Osoba{v}{prumer
p;}
          void tisk(){
               vypis();
               cout << "Prumer: " << prumer;
          }
};
```

20. Třídy a objekty

A. Deklarace třídy

- Deklarace je velice podobná deklaraci struktury, na rozdíl od struktur obsahují kromě členských dat také členské funkce – metody, které umožňují s členskými daty pracovat.
- Při deklaraci třídy můžeme definovat přístupová práva k členským datům a metodám

B. Přístupová práva

```
class jmeno [:předchůdci třídy]{
     private:
     //data přístupná jen v rámci třídy – soukromá
     protected:
     //data přístupná jen v rámci třídy a jejich potomků - chráněná
     public:
     //data přístupná odkudkoli – tvoří rozhraní třídy - veřejná
}
```

- **Private** – soukromá data, přístupná pouze v rámci třídy
- **Protected** – chráněná, přístupná v rámci třídy a jejich potomků
- **Public** – veřejná, přístupná odkudkoli, tvoří rozhraní třídy

C. Členská data a členské metody

Třída se liší od stuktury hlavně v tom, že kromě datových členů v sobě také obsahuje funkce, které umí s datovými členy pracovat. Tyto funkce nazýváme metody třídy. Platí, že všechny objekty (instance) dané třídy sdílejí jednu sadu těchto funkcí.

Pouze členská data má každá instance svá. Stejně jako u členských dat, tak i u metod určujeme jeijch viditelnost, tedy zda jsou public, private či protectred. Třídu nejčastěji deklarujeme v jejich samostatném hlavičkovém souboru. Metody třídy deklarujeme uvnitř deklarace této třídy definice pak u krátkých funkcí můžeme uvést hned v deklaraci třídy místo jejich deklarací anebo je častěji uvádíme v samostatném souboru s příponou *.cpp* s tím, že před název funkce uvedeme název třídy následovaný dvěma dvojtečkami.

D. Konstruktory

- Speciální členská funkce třídy, která vytvoří instanci třídy a případně ji inicializuje.
- Má stejný název jako třída a nedefinuje se u nich návratový typ.

- Může jich ve třídě být více, podle toho jak chceme vznikající objekt inicializovat.
- Není-li definovaný žádný konstruktor, volá se tzv. **implicitní konstruktor**, ten nijak neovlivňuje členská data, pouze provádí různé skryté inicializace.

```
class Zlomek{
     private:
          int citatel, jmenovatel;
     public:
          Zlomek(int c, int j);
          Zlomek(){citatel = 1; jmenovatel = 1;}
          void vypis();
}
Zlomek::Zlomek(int c, int j){citatel = c; jmenovatel = j;}
```

Kopírovací konstruktor

- používá se při kopírování objektu do nově vytvořeného objektu.
- Má jediný parametr – odkaz na objekt stejné třídy, který je navíc uveden jako konstantní, protože konstruktor nebude předávaný objekt měnit.

```
Zlomek(const Zlomek &zdroj){
     citatel = zdroj.citatel;
     jmenovatel = zdroj.jmenovatek;
}
int main(){
Zlomek prvni, druhy(5,3), treti(druhy);
//prvni = 1,1
//druhy = 5,3
//treti = 5,3
}
```

Destruktor

- Opak konstruktoru, volá se v době, kdy končí platnost objektu.
- Stejný název jako třída, ale před název se dává znak ~ (tilda), nic nevrací a nemá ani žádné parametry.
- Je obecně potřebný hlavně tam, kde se pracuje s dynamickou pamětí.

```
~komplex(){}
```

21. VCL aplikace v Turbo C++

A. Popis jednotlivých částí prostředí Turbo C++ a práce s nimi

Firma Borland vytvořila vlastní knihovnu objektů – VCL – Visual Component Library a do jednotlivých tříd zabalila popisy práce. Jedná se o framework určený pro vývoj aplikací pro MS Windows. Programátor pouze nastaví některá členská data příslušného objektu, kteými blíže určí co má objekt dělat a naplnění všech struktur a volání funkcí Win32 API už provede objekt sám.

B. Tvorba VCL aplikace v Turbo C++, soubory tvořící VCL aplikaci

New Prooject -> VCL Form Application

Základní soubory:

- **Project1.bsdroj**
  - Projektový soubor, zde je popsáno ze kterých souborů se projekt skládá a jak se má přeložit a zlinkovat.
  - Spravuje ho IDE, nezasahujeme do něj
- **Project1.cpp**
  - Hlavní soubor aplikace
  - Obsahuje funkci WinMain
  - Tento soubor obyčejně neupravujeme – pracuje s ním IDE
- **Project1.res**
  - Soubor se zdroji (resources) projektu
  - Obsahuje například ikonku
- **Unit1.dfm**
  - Soubor se zápisem grafického rozhraní
  - Obsahuje instrukce jak vykreslit formulář
- **Unit1.h**
  - Hlavičkový soubor formuláře
  - Obsahuje deklarace tříd
- **Unit1.cpp**
  - Zdrojový soubor formuláře
- **Debug_Build**
  - Adresář obsahující obj soubory, předkompilované hlavičky tds a exe
  - Ve verzi potřebné pro debbugger

- **Relase_Build**
    - Adresář, stejný jako v Debug_Build
    - Připravené pro uživatele
- **__history**
    - adresář se záložními verzemi souborů

C. Vlastnosti a práce s komponentami knihovny VCL – properties, metods, events

Prakticky u každé použité komponenty z knihovny VCL vždy sledujeme jaké má základní vlastnosti (properties), metody a reakce na základní události (events). Events jsou vlastně také metody, ale specializované na ošetření událostí. Názvy těchto metod začínají On (např. OnClick). Můžeme jim dát samozřejmě i jiné názvy, ale pro snadné odlišení je vhodné ono On na začátku ponechat.

Nejdříve se vždy soustředíme na vlastnosti a pak postupujeme k vlastnostem předků.

Nejčastější properties:

Owner určuje komponentu, která je zodpovědná za zrušení naší komponenty. Nejčastěji to je samotný formulář. Pak zde uvádíme pointer this.
Parent určuje komponentu, na které naše komponenta leží. Pokud se nachází přímo v klientské oblasti formuláře, píšeme zde opět this. Pokud naše komponenta lež např. na panelu Panel1, pak zde píšeme Panel1.
Name je název instance komponenty, u formuláře název třídy. IDE vám výchozí název navrhne samo. Pro snadnou čitelnost programu je ale vhodné ho změnit na takový, který by hned vypovídal komu patří.
Caption je nadpis naší komponenty (pokud ho má).
Left, Top jsou souřadnice levého horního rohu v pixelech. Měřeno vůči komponentě, která je
Width, Height vnější šířka a výška
ClientWidth, ClientHeight šířka a výška vnitřní, pracovní oblasti
Font určuje vlastnosti v komponentě použitého písma (znaková sada, font, barva písma, velikost, styl atd.).

Size – určuje velikost písma v pixelech

Height – velikost i s místem nad a pod písmenem a je tedy větší než Size. Zadává se jako záporné.

Anchors - vůči čemu jsou zakotveny souřadnice komponenty.

Color - většinou nastavení barvy pozadí

Tag - číslo typu int, do kterého si můžeme něco poznačit. Je jen pro nás. IDE ho nemění.

Align - jak se má naše komponenta umístit v okně Parent

Visible - true znamená, že je naše komponenta viditelná

Enabled - je-li true, můžeme komponentu ovládat myší a klávesnicí.

Nejčastější events (události):

OnClick - kliknuli jsme myší na komponentu

OnMouseDown, **OnMouseUp** - stiskli nebo pustili jsme levé tlačítko myši na naší komponentě

OnEnter - komponenta dostala focus

OnExit - komponenta ztratila focus

OnClose - formulář je destruktorem rušen

OnShow - formulář je zrovna zobrazen

OnPaint - formulář je překreslen

OnResize - formuláři byly změněny rozměry

Metody komponenty:

Metody jsou specifické pro každou komponentu. Každá komponenta však obsahuje konstruktor a destruktor. Tyto volá IDE při zakládání komponenty při spouštění programu nebo je voláme my, pokud komponentu vytváříme dynamicky sami. V obou případech jsou používány operátory new a delete. Z toho také plyne, že na komponentu se vždy odvoláváme pomocí pointeru (pomocí špky).

22. Základní třídy a komponenty

A. Komponenty TForm, TPanel

TForm

Třída formuláře

Události

- **OnActivate** – vznikne, když se okno stane aktivním (dostane focus)
- **OnCreate** – okno bylo založeno konstruktorem
- **OnClose** – při zavření
- **OnShow** – okno je zobrazeno
- **OnHide** – okno je skryto
- **OnPaint** – okno je překreslováno
- **OnResize** – okno změnilo rozměry

Metody

- **Show**() – otevření dalšího formuláře, lze pracovat současně se všemi
- **ShowModal**() – otevření jako dialogové, lze pracovat pouze s jedním

TPanel

Kontejner pro vložení dalších

B. Komponenty TLabel, TButton, TEdit

TLabel

Slouží pro popisek

Vlastnosti

- **Caption** – text
- **Aligment** – zarovníní textu
- **WordWrap** – je-li true, pak delší texty mohou být zalamovány

TButton

Vlastnosti

- **ModalResult** – určuje jaká hodnota se má vrátit, pokud tlačítkem zavíráme modální okno

Metody

- **Click** – kliknutí
- **SetFocus** – Tlačítko získá focus

TEdit

Pro zadávání textu

Vlastnosti

- **Aligment** – zarovnání textu
- **CharCase** – velikost písma
- **ReadOnly**– pouze na čtení
- **Enabled** – je-li false, nelze vložit
- **Text** - text

Metody

- **Clear** – Vymazání
- **SetFocus** –získá focus
- **Focused** – vybrán

Události

- **OnChange** – Text byl změněn
- **OnEnter** – při focusu
- **OnExit** – při opuštění
- **OnKeyUp** – Byla povolena klávesa
- **OnKeyDown** – stisknutá klávesa

C. Komponenty TCheckBox, TRadioButton, TRadioGroup

TCheckBox

True/False. Můžeme se dotazovat na vlastnost Checked, která určuje zda-li je zaškrnut.

Vlastnosti

- **Aligment** – zarovnání textu k okénku
- **Checked** – zda-li je zaškrnutý
- **Enabled** – je-li false, nelze měnit

Události

- **OnClick** – Po kliknutí, nebo po stisknutí mezery na klívesnici

TRadioButton

Navzájem vylučující volby. Může být zaškrnutý pouze jeden najednou. Prováží se automaticky. Mezi panely se neovlivňují.

Vlastnosti

- **Aligment** – zarovnání textu k okénku
- **Checked** – zdali je zamáčknutý. Může být jen jeden

Události

- **OnClick** – Po kliknutí, nebo po najetí klávesnicí pomocí šipek

TRadioButton

Umožňuje rychle do formuláře nebo panelu umístit skupinu provázaných tlačítek typu RadioButton. Na jednotlivá tlačítka se snadno neodstaneme. Celé se to chová jako jeden prvek.

Vlastnosti

- **Columns** – počet sloupců, do kterých budou tlačítkla rozmístěna. Máme-li pět tlačítek a dáme **Columns** do 5, budou vedle sebe vodorovně.
- **ItemIndex** – které tlačítko bude při zahájení programu vybráno. Čísluje se od **0** a hodnota **-1** znamená že nebude vybráno žádné.

Items – seznam názvů jednotlivých tlačítek. Co řádek to jedno tlačítko.

23. Třídy a komponenty pro práci s textem, seznamy, časy

A. Třída AnsiString, Tstrings

AnsiString

Třída pro uložení textového řetězce.

Metody

- **AnsiCompare** – porovnává svůj text se zadaným a zohledňuje velikost písma
- **AnsiCompareIC** – porovnává svůj text se zadaným a ignoruje velikost písmen
- **Insert** – vloží do vlastního textu zadaný text na zadanou pozici
- **Lenght** – vrací délku textu
- **Printf** – naformátování textu do objektu pomocí pravidel platících pro printf
- **SubString** – vrací část vlastního textu
- **UpperCase** – převede text na velká písmena
- **LowerCase** – převede text na malá písmena
- **C_str** – funkce vracející adresu začátku stringu s textem
- **Trim** – odříznutí zbytečných mezer na začátku a konci
- **Operator[]** – vrací písmeno, jehož index zadáme, indexuje se od 1
- **Přežítené operátory**

Konverzní funkce

- **IntToStr, FloatToStr, FloatToStrF**
- **StrToInt, StrToFloat** – jestliže při převodu dojde k chybě, je generována vyjímka

TStrings

Předek všech tříd obsahujících seznamy stringů.

Vlastnosti

- **Int Count** – počet položek v seznamu
- **AnsiString Strings[int Index]** – pole jednotlivých položek seznamu
- **AnsiString Text** – celý text uložený v seznamu (celý text najednou). Jednotlivé řetězce jsou odděleny znaky \r\n.

B. Komponenty TListBox, TComboBox, TMemo, TStringGrid

TListBox

Komponenta obsahující seznam položek. Seznam lze rolovat pomocí scrollbaru.

Vlastnosti

- **Int Columns** – počet sloupců
- **Int ItemIndex** – pořadové číslo vybrané položky
- **TStrings* Items** – Pointer na instanci třídy TStrings obsahující jednotlivé řetězce
- **Bool MultiSelect** – je-li true, tak lze vybrat více položek najednou
- **Int SelCount** – počet vybraných položek při MultiSelect=true

- **Bool Selected[]** – pole, má-li daná položka hodnotu true, je vybrána
- **Bool Sorted** – je-li true, tak jsou položky setříděny podle abecedy
- **Int TabWidth** – šířka sloupců, pokud je nastaveno Columns>1
- **Int TopIndex** – pořadové číslo položky, která je zobrazena na vrcholu

Metody

- **Clear()** – vymaže všechny položky z ListBoxu

TComboBox

Spojení komponent Edit a ListBox. Přejímá od nich většinu vlastností, metod a události.

Vlastnosti

- **Int DropDownCout** – určuje, kolik položek bude zobrazovat ListBox
- **Bool DroppedDown** – ture, znamená, že list box bude zobrazován
- **TComboBoxStyle Style** – definuje 5 stylů combo boxu

TMemo

Komponenta sloužící pro zobrazení a editaci víceřádkového textu

Vlastnosti

- **TScrollStyle ScrollBars** – určuje kolik scrollbarů se bude zobrazovat
- **Bool WantReturns** – false znamená zákaz vkládání znaku ENTER do textu
- **Bool Tabs** – false zakazuje vkládání TAB do textu
- **TCaretPos CaretPos** – určení polohy caretu (kurzoru v textu)
- **TStrings* Lines** – pointer na seznam řádků
- **Bool Modified** – true znamená, že text v okně byl nějak změněn
- **Bool ReadOnly** – true znamená, že do okna nelze psát
- **TMemoSelection Selection** – vrací strukturu údajů o rozsahu selektovaného textu
- **Int SelLenght** – určení délky selektovaného textu
- **Int SelStart** – pozice prvního selektovaného znaku v textu
- **AnsiString SelText** – vracínebo mění text v selektované oblasti
- **TCaption Text** – celý text uložený v TMemo. Řádky jsou odděleny znaky \r\n
- **Bool WordWrap** – true znamená, že řádky budou zalamovány tak, aby se vešly do okna. Nesmí být ovšem nastaveno ScrollBars na ssHorizontal nebo ssBoth.

Metody

- **Append(AnsiString Text)** – připojení textu Text na konec
- **Clear()** – vymazání celého textu
- **ClearSelection()** – vymazání vybraného textu
- **CopyToclipboard()** – zkopírovaní vybraného textu do schránky
- **CutToClipboard()** – vyříznutí vybraného textu do schránky
- **PasteFromClipboard()** – vložení textu ze schránky
- **Insert()** – vložení textu

- **SelectAll()** – označení celého textu
- **UnSelect()** – zrušení označení vybraného textu

Události

- **OnChange** – událost vznikne, když je v textu něco změněno

TStringGrid

Komponenta umožňující pracovat s celou řadou AnsiStringů uspořádaných do formy dvourozměrné tabulky. Tabulka může mít několik pevných prvních sloupců a řádků pro např. nadpisy. Tyto zůstavají pevné, při scrollovaní se nepřesouvají a umožňují uživateli mít stále přehled o tom, na jaká data zrovna kouká. Buňky, které nejsou obsaženy v pevných řádcích a sloupcích, mohou být po správném nastavení editovatelné nebo ne.

Vlastnosti

- **TScrollStyle ScrollBars** – určuje, kolik scrollbarů bude v komponentě
- **TGridSelection** – určuje obdelník buněk vybrané oblasti.
- **Long Col** – číslo sloupce vybrané buňky (čísluje se od 0)
- **Long Row** – číslo řádku vybrané buňky (čísluje se od 0)
- **long ColCount** – počet sloupců mřížky
- **long RowCount** – počet řádků mřížky
- **int ColWidths[]** – pole se šírkami jednotlivých sloupců
- **int RowHeights[]** – pole s výškami jednotlivých řádků
- **int DefaultColWidth** – standardní šířka sloupců
- **int DefaultRowHeight** – standardní výška řádků
- **long LeftCol** – první momentálně viditelný sloupec
- **long TopRow** – první momentálně viditelný řádek
- **int VisibleColCont** – počet momentálně viditelných sloupců
- **int VisibleRowCount** – počet momentálně viditelných řádků
- **int FixedCol** – počet pevných sloupců
- **int FixedRow** – počet pevných řádků
- **intGridLineWidth** – šírka dělících čar mezi buňkami v pixelech
- **TGridOptions**
    - Množina hodnot (Set) určujících chování mřížky.
    - Důležitá je např. hodnota goEditing, která je-li v množině obsažena, znamená že mřížku lze editovat
- **AString Cells[col][row]** – pole řetězců mřížky.
- **TStrings* Cols[col]** – seznam obsahující řetězce obsažené v daném sloupci
- **TStrings* Rows[col]** – seznam obsahující řetězce obsažené v daném řádku.

Metody

- **TRect CellRect (int ACol, int ARow)** – vrátí obdélník buňky o souřadnicích ACol a ARow v pixelech celé obrazovky. Dáme-li parametry nulové, dostaneme obdélník

levého horního rohu viditelné oblasti mřížky. Jestliže adresovaná buňka není momentálně viditelná, dostaneme prázdný obdélník. Tato funkce se může použít tehdy, když se rozhodneme překreslovat obsahu buněk sami (DefaultDrawing=false)

- **Void MouseToCell(int X, int Y, int &ACol, int &ARow)** – Přepočítá souřadnice v rámci celé obrazovky na číslo sloupce a řádku. Používá se v událostech od myši
- **TGridCoord MouseCoord(int X, int Y)** – přepočítá souřadnice v pixelech celé obrazovky na souřadnice buňky v mřížce v číslech řádku a sloupce. Používáme v obsluze události od myši, abychom si určili buňku, nad kterou se zrovna nachází kurzor.

C. Třída TDateTime

Pro uložení data a času. Základem této třídy je hodnota deklarovaná takto: double Val;

V tomto údaji uchován jak datum, tak i čas. Celočíselná část hodnoty Val představuje počet dnů od 30.12.1899. Desetinná část představuje čas ve dni.

Třída obsahuje mnoho přetíženýách operátorů a členských funkcí pro práci s těmito údaji.

Metody

- **Static TDateTime CurrentDate** – vrací dnešní datum
- **Static TDateTime CurrentDateTime** – vrací momentální datum a čas
- **Static TDateTime CurrentTime** – vrací momentální čas
- **AnsiString DateString** – převede uložené datum na řetězec
- **AnsiString TimeString** – převede čas na řetězec
- **Int DayOfWeek** – vrátí pořadové číslo dne v týdnu. Neděle má číslo 1
- **Void DecodeDate()** – dekóduje datum na rok, měsíc a den
- **Void DecodeTime()** dekóduje čas na hodinu, minuta, sekunda a milisekunda
- **Int FileDate()** – dekóduje obsah instance na hodnotu s údajem času souboru používaného v DOSu.
- **Static FileDateToDateTime()** – převede čas souboru z DOSu do TDateTime
- **AnsiString FormatString()** – určuje způsob formátování řetězce data a času

Díky přetížení různých operátorů lze údaje typu TDateTime různě sčítat, odečítat, přičítat k nim celá čísla a podobně

D. Komponenta TTimer

Pro vykonávání nějaké činnosti v pravidelných intervalech

Vlastnosti

- **Bool Enabled** – true znamená, že je běh časovače povolen
- **Unsigned int Interval** – čas v milisekundách, po jehož vypršení dojde k události OnTimer

TApplication->ProcessMessages();

V systému Windows je celý program řízen na základě obsluh událostí. Někdy se může stát, že některá obsluha bude trvat hodně dlouho. V tom případě, než dojde k ukončení této obsluhy, přestane celý program reagovat.

Aby k takovému absolutnímu „zamrznutí“ programu nedocházelo, má třída TApplication která je základní třídou našeho programu, metodu ProcessMessages. Zavoláním této metody způsobíme to, že se chod obsluhy události zastaví a provedou se obsluhy všech požadavků, které mezitím vznikly. Pak se program vrátí zpět do přerušené obsluhy a pokračuje v jejím provádění.

24. Menu, dialogové komponenty a výjimky v C++

A. Komponenty TMainMenu, TPopUpMenu

TMainMenu

Komponenta spojující v sobě horní pruh ve formuláři a roletové menu. Každá položka menu je tvořena jako instance třídy TMenuItem. Jednotlivé položky můžeme vytvářet v editovacím okně, které se objeví po poklepání na ikonu komponenty.

Vlastnosti

- **TMenuItem* Items** – pointer na instanci jednotlivých položek menu
- **TBitmap* Bitmap** – bitmapa kreslená jako podklad menu
- **TColor Color** – barva menu
- **TCustomImageList* Images** – seznam obrázku, které lze použí jako pozadí pro jednotlivé položky menu

TPopUpMenu

Komponenta reprezentuje kontextové menu, které se zobrazí, když na komponentě stiskneme pravé tlačítko myši. Každá komponenta formuláře může mít své popup menu jehož název se komponentě dá do vlastnosti PopupMenu. Další používání je podobné práci s hlavním menu.

Vlastnosti

- **TPopupAignment Alignment** – určuje zarovnání popup menu vůči kurzoru myši.
- **Bool AutoPopup** – false zakazuje vynořování popup menu pomocí pravého tlačítka myši
- **TMenuItem* Items** – pointer na instanci jednotlivých položek menu

Události

- **OnPopup** – vznikne když je popup menu vyvoláno

TMenuItem

Komponenta reprezentující položku TMainMenu a TPopupMenu.

Vlastnosti

- **Bool checked** – je-li true, přidá před položku zaškrtnutí
- **Bool Default** – je-li true, má položka menu titulek tučným písmem a je jako vybrána
- **Byte GroupIndex** – určuje, které položky jsou spolu spojeny, pokud se použije RAdioItem=true
- **Bool RadioItem** – true znamená, že položka patří do skupiny položek spojených spolu jako radio button. Do které skupoiny patří určujeme pomocí GroupIndex
- **TShortCut ShortCut** – určuje klávesovou zkratku, kterou lze použít pro vyvoální této položky menu. Nastavíme-li např. ShortCut=Ctrl+S, rovná se stisk klávesy Ctrl+S kliknutí myši na této položce.

Metody

- **Add(TMenuItem* Item)** – můžeme dynamicky přidat na konec menu novou položku
- **Delete(int Index)** – zrušení položky dynamicky
- **Insert(int Index, TMenuItem* Item)** – vložení položky na polohu Index

B. Komponenty TOpenDialog, TSaveDialog

Dialog, který se používá pro volbu souboru, který chceme otevřít pro čtení nebo zápis. Po správném ukončení dialogu můžeme z komponenty vyčíst název souboru i s celou cestou. Před otevřením dialogu je vhodné nastavit některé vlastnosti a tím uživateli zpříjemnit práci.

Vlastnosti

- **AnsiString DefaultExt** – výchozí přípona souboru (uvádíme pouze příponu skládající se ze tří písmen a bez úvodní tečky)
- **AnsiString FileName** – při zápisu určuje výchozí název souboru, po ukončení dialogu obsahuje název vybraného souboru i s cestou
- **TStrings *Files** – obsahuje seznam souborů, které jsme vybrali. Můžeme vybrat i více souborů. Musíme to ovšem povolit v Options prvkem ofAllowMultiSelect.
- **AnsiString InitialDir** – název výchozího adresáře
- **AnsiString Title** – titulek dialogu
- **AnsiString Filter** – obsahuje povolené přípony souborů včetně jejich popisů.
- **Int FilterIndex** – pořadí řetězce z Filter, který je použit jako výchozí. Čísluje se od 1.
- **TOpenOption Option** – množia (Set), která určuje vzhled a chování dialogu.

C. Komponenty TFontDialog, TColorDialog

TFontDialog

Dialog určení pro volbu fontu a jeho vlastnosti.

TColorDialog

Dialog pro určení barvy

D. Použití a deklarace vyjímky

Ošetřením výjimky zaručujeme, že aplikace reaguje na chyby předvídaným způsobem, dokáže podle možnosti zajistit zotavení ze vzniklé chyby a případně uzavřít rozpracovanou činnost bez ztráty dat.

Pro práci s výjimkami slouží tato klíčová slova: **try, catch** a **throw.**

Všechny operace, které jsou jistým způsobem nebezpečné a jejichž provádění by se nemuselo podařit, provádíme v pokusném bloku – označený příkazem **try**, za kterým následuje jedna nebo více obsluh výjimek (handlerů), které jsou označené příkazem **catch**.

Dále lze ještě použít klíčové slovo **_finally**. Pokud potřebuje zajistit, aby se některá část programu (např. zavření souboru) provedla bezpodmínečně vždy, pak vytvoříme nadřazený blok **try** zakončený slovem **_finally**.

```
int main()
{
     int a;
     cout << "Zadej cislo 1" << endl;
     try {
          cin >> a;
          if(a<1)throw 10;
          if(a>1)throw 20;
          cout << "Povedlo se";

     } catch (int i) {
          if(i==10)cout<<"Zadal jsi mensi cislo"<<endl;
          if(i==20)cout<<"Zadal jsi vetsi cislo"<<endl;
     }
     return 0;
}
class ExceptionNula{};
class ExceptionTEXT{
     public: char text[];
     EJinaVyjimka(char *rezez) {strcpy(text,retez);}};
float deleni(float a, float b){
     if(b==0)throw ExceptionNULA();
     if(a==b)throw ExceptionTEXT("Cisla jsou stejna");
     if(a<b)throw 100;
     if(b<0)throw 3.6;
     return a/b;
}
```

```
int main(){
     float z, citatel=200, delitel=52;
     try {z=deleni(citatel, delitel);}
     catch (ExceptionNULA &x){cout << "Deleni nulou";
cin.get(); return 0;}
     catch (ExceptionTEXT &x) {cout << x.text; cin.get();
return 0;}
     catch(int &x) {cout<<"Chyba
cislo:"<<x<<endl<<"citatel<jmenovatel."; return 0;}
     catch(...) {cout << "Jina nespecifikovana vyjimka" <<
endl; cin.get(); return 0;}
     cout << "Vse je OK, zadal si spravna cisla a vysledej
je:"<<z<<endl;
}
```

25. Grafické komponenty, komponenty pro tisk

A. Třídy TCanvas, TBrush, TPen, TBitmap

Práce s grafikou pod Win32 API představuje poměrně náročnou práci, při které je nutná také řádná dávka opatrnosti. Pro práci s grafikou si Windows drží řadu různým datových struktur (prostředky), které jsou společné pro celá Windows. Prostor pro tyto struktury není ovšem neomezený. Proto je třeba si pro nějakou svou činnost v rámci grafického systému vždy z těchto prostředků vypůjčit to, co zrovna potřebujeme (např. strukturu pro popis štětce nebo pera) a až když už tyto prostředky nepotřebujeme, tak je opět uvolnit, aby je mohly Windows nabídnout opět jinému programu. Jestliže si nedáme pozor, může se stát, že tyto grafické prostředky Windows vyčerpáme a znemožníme práci celých Windows.

TCanvas

Pro zjednodušení práce s grafikou byla v rámci knihovny VCL zavedena třída **TCanvas** (plátno), která za nás provede potřebné zabírání a opětné uvolňování prostředků a zjednoduší nám volání různým funkcí Win32 API, Třída obsahuje poměrně málo datových položek, ale tyto položky většinou představuji ukazatele na instance jiných tříd.

Vlastnosti

- **Font** – Určuje font použitý při psaní textu na obrázku. Pomocí objektu TFont můžeme určit písmo, barvu, velikost a styl písma

- **Brush** – Určuje barvu a vzor, které TCanvas používá pro vyplňování grafických tvarů a pozadí. Pomocí objektu TBrush určujeme barvu, vzor nebo bitmapu, které budou použity při vyplňování mezer na plátně.
- **Pen** – Určuje druh pera použitého pro kreslení čar a tvarů. Pomocí objektu TPen určujeme barvu, styl, šířku a druh pera.
- **PenPos** – Určuje aktuální pozici pera na plátně.
- **Handle** – Čtyřbytový handle, který můžeme při volání funkcí Win32 API pro kreslení do našeho Canvasu.

Metody

- **DrawPoint** – Vykreslí bod
- **DrawPoints** – Vykreslí řadu bodů
- **MoveTo** – Změna aktuální pozice ve výkresu na bod X,Y
- **LineTo** – Kreslení čáry na plátno z aktuální pozice na místo určené pomoci X,Y a nastavení pozice pera na X,Y.
- **Polyline** – kreslí aktuálním perem řadu úseček, jejich body jsou uvedeny v poli Points
- **Polygon** – kreslí aktuálním perem řadu úseček, jejich body jsou uvedeny v poli Points, poslední bod spojí s prvním a takto vzniklou lochu vyplní aktuálním štětcem.
- **FrameREct** – Nakreslí aktivním perem obrys obdélníka
- **Recdtangle** – Vykreslí pomocí aktuálního pera na plátno zadaný obdelník a vyplní ho pomoci aktuálního štětce.
- **RoudRect** – Vykreslení obdélníku se zaokrouhlenými rohy
- **FillRect** – Vyplní zadaný obdelník na plátně pomoci aktuálního štětce.
- **DrawFocusRect** – Kreslí obdélník označující objekt na který se máme zaměřit
- **CopyRect** – Kopie části obrazu z jednoho plátna na jiné plátno
- **Ellipse** – Kreslí na plátno elipsu definovanou ohraničujícím obdélníkem
- **Arc** – Kreslí na obrázek oblouk tvořený obvodem elipsy ohraničené zadaným obdélníkem.
- **Chord** – Kreslí uzavřenou části elipsy zadanou obdélníkem a utnutou zadanou úsečkou
- **Pie** – Kreslí na plátno část koláče určeného elipsou.
- **TextOut** – Vypíše na zadanou pozici aktuálním fontem zadaný text. Funkce nevymaže pozadí pod textem. K jeho vymazání musíme nejdříve použít funkci FillRect
- **TextRect** – Vypíše do zadaného obdélníka na zadané souřadnice aktuálním fontem zadaný text
- **TextWidth** – Vrátí v pixelech šířku zadaného textu, kdyby se vypsal aktuálním fontem
- **TextHeight** – Vrátí v pixelech výšku zadaného textu, kdy se vypsal aktuálním fontem.
- **Draw** – Vykreslí grafický objekt určený parametrem Graphic na plátně v místě daném souřadnicemi X,Y

- **StretchDraw** – Nakreslí grafický objekt do canvasu do zadaného obdélníka. Případně provede smrštění nebo roztažení obrazu tak, aby vyplnil přesně zadaný obdelník

Překreslování

Okna se musí často překreslovat kvůli např. překrývání oken. Je generována událost OnPaint. Navíc je v datové položce Canvas->ClipRect specifikován obdélník, který je nutno překreslit. Takže často není nutné překreslit celé okno, ale pouze jeho výřez.

V knihovně VCL jsou definovány funkce Invalidate a Update. Nelze zde určit oblast, takže se vždy jako invalidní označí celá plocha a navíc vždy dochází k jejímu vymazání.

Ve VCL existují ještě dvě takřka shodné funkce REpaint a Refresh. Obě dělají totéž, tedy volají Invalidate() a pak Update(), takže zajistí okamžité překreslení celé plochy okna.

TBrush

Vlastnosti

- **TColor Color** – určuje barvu štětce
- **TBitmap *Bitmapa** – určuje bitovou mapu, která se používá jako tvar štětce
- **BRUSH Handle** – handle na objekt brush (lze použít ve funkcích Win32 API)
- **TBrushStyle Style** – určuje styl štětce (bsSolid, bsClear, bsHorizontal, bsVertical, …)

TPen

Vlastnosti

- **TColor Color** – určuje barvu pera
- **TPenStyle Style** – určuje styl pera (psSolid, psClear, psDash, psDot, …)
- **TPenMode Mode** – určuje, jak bude pero kreslit na Canvas
- **Int Width** – určuje šířku pera v pixelech

TBitmap

Určuje operace nad bitovou mapou. Třída TBitmap je definována ve windows.hpp pro operace pod Win32 API a ve graphics.hpp pro bitmapupod VCL. Vždy musíme tedy použít rozlišovač viditelnosti.

*Graphics::TBitmap *Bitmapa*

TBitmap můžeme používat jako úschovu bitových map v paměti pro jejich další použití

Vlastnosti

- **Int Height** – výška bitmapy
- **Int Width** – šířka bitmapy
- **Bool Transparent** – zapíná transparentní mód
- **TColor TransparentColor** – určuje která barva je průhledná. Používáme např. při kreslení kulatých rohů, kdy kulatý roh vyplníme právě uvedenou barvou.

- **TransparentMode** – je-li tmAuto, bere se automaticky jako transparentní barva, barva levého spodního rohu. Je-li tmFixed, pak transparentní barvu musíme určit sami.

Metody

- **LoadFromFile** – natažení obrázku ze souboru s příponou BMP.
- **SaveToFile** – uložení do souboru

B. Komponenty TImage, TPaintBox

TImage

TImage používáme k zobrazování obrázků. Pro zadání aktuální bitové mapy použijeme objekt TPicture, který vlastní TImage. Vlastnosti a metody TPicture lze použít pro takové věci, jako je načítání obrázku ze souboru, vymazání obrazu v TImage a k přenosu obrázku do dalších komponent. TImage má několik vlastnosti určujících, jak se má obraz zobrazit v rámci hranic TImage objektu.

TPaintBox

TPaintBox představuje plátno, které mohou využívat další komonenty pro vykreslování obrazu. Např. TPanel nemá přístupný Canvas. Proti si vypomůžeme tím, že do panelu vložíme TPaintBox a kreslíme vlastně do jeho Canvasu.

C. Třída TPrinter

Při vytváření tiskových sestav a dokumentů použijeme znalosti, které jsme získali při pokusech s grafikou. V knihovně VCL existuje třída TPrinter, která slouží pro ovládání tiskárny. Na instanci této třídy, která je pro nás založena hned při spuštění programu, se můžeme dotázat pomocí funkce Printer(). Třída TPrinter obsahuje vlastnost Canvas, takže práce při tiscích se vlastně převádí do kreslení do tohoto Canvasu.

TPrinter

Vlastnosti

- **Canvas** – malířské plátno, do kterého se kreslí dokument
- **PageHeight**, **PageWidth** – rozměry stránky v pixelech
- **Title** – Titulek dokumentu při tisku, který se zobrazuje v tiskové frontě
- **PageNumber** – pořadové číslo právě tisknuté stránky
- **Orientation** – orientace dokumentu (poPortrait, poLandscape)
- **Copies** – počet kopií, které se mají vytisknout
- **Fonts** – seznam názvů fontů, kter jsou pro vybranou tiskárnu dostupné
- **Printers** – seznam názvů tiskáren instalovaných ve Windows
- **PrinterIndex** – index vybrané tiskárny (standardní tiskárna má index 1)
- **Printing** – indikuje, zda právě probíhá tisk (true) nebo už byl dokončen
- **Aborted** – jestliže byl přerušen tisk pomocí metody Abort() má hodnotu true
- **Handle** – handle na objekt tiskárny

Metody

- **Void BeginDoc()** – založí tiskovou úlohu
- **Void Abort()** – zrušení tisku, zbývající část dokumentu se vyjme z tiskové fronty
- **Void NewPage()** – přechod na novou stránku, PageNumber se zvýší o 1, pozice pera se nastaví na souřadnice 0,0
- **Void EndDoc()** – ukončí dokument, začne se tisknout

D. Komponenty TPrinterSetupDialog, TPrintDialog

TPrinterSetupDialog

Dialog pro výběr aktivní tiskárny a parametrů tisku. Tyto parametry se nastavují přímo do systému, proto nemá metoda Execute návratovou hodnotu.

TPrintDialog

Zobrazuje dialogové okno pro nastavení parametrů tisku. Pracuje se s tímto dialogem stejně jako s jinými standardními dialogy.

Programování tisku

Při programování tiskové sestavy začínáme vždy zavoláním funkce **BeginDoc**

Následně si pomocí vlastnosti **PageWidth** a **PageHeight** zjistíme rozměry stránky v pixelech. Tyto rozměry mohou jít např. u laserových tiskáren do několika tisíc. Proto nemá smysl udávat např. Šířky pera přímo v pixelech, ale pixely šířky zjisti jako zlomek např. šířky stránky (např. 1/100)

Podobné určujeme i např. okraje tisku. Také je bereme jako zlomek celkových rozměrů stránky a zapamatujeme si je ve vhodných proměnných.

Tiskneme-li text na více řádků, pak si založíme ukazovátko, které bude ukazovat na horní okraj následujícího řádku. Toto ukazovátko pak při přechodu na další řádek zvyšujeme o velikost písma.

Po každém řádku zkontrolujeme, zda už ukazovátko polohy příštího řádku neukazuje za spodní okraj, který jsme si zvolili. Pokud ano, pak pomocí funkce **NewPage** odstránkujeme a ukazovátko řádku nastavíme na horní okraj následující stránky.

Na stránce nemusíme postupovat při tisku důsledně shora dolů, ale můžeme se např. i vracet. Lze např. vytisknout text na celé stránce a pak kolem něho nakreslit rámeček nebo dokonce přes text něco nakreslit.

Celá navržená tisková sestava se začne tisknout teprve tehdy, až zavoláme funkci **EndDoc**