Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií
Ústav automatizace a měřící techniky

Semestrální projekt

Tvorba úloh pro výuku předmětu: Praktické programování v C/C++

Petr Vrzal
B3AMT
2004/2005

Vedoucí práce: Ing. Petr Petyovský
Vedoucí ústavu: Doc. Ing. Pavel Jura CSc.

Termín zadání práce: 27.09. 2004
Termín odevzdání práce: 23.12. 2004

Zadání semestrálního projektu:

Cílem je navrhnout a vytvořit vhodné úlohy pro cvičení předmětu spolu s podpůrnými softwarovými prostředky pro detekci a odstranění typických chyb studentů v zadaných úlohách.

1. Seznamte se s dostupnou literaturou a obsahem předmětu: Praktické programování v C/C++.
2. Navrhněte a vytvořte demonstrační úlohy pro tento předmět.

Obsah:

1. Pole v jazyce C/C++

1.1 Statické jednorozměrné pole

1.1.1 Zadání:

Vytvořte jednorozměrné statické pole o velikosti 1x10, které bude obsahovat hodnoty celočíselného datového typu *int*. První položka pole bude obsahovat hodnotu 0, přičemž každá další položka bude mít hodnotu o 1 vyšší než předcházející. Poslední položka bude mít tedy hodnotu 9. Vypište toto pole na obrazovku.

1.1.2 Rozbor úlohy

Statické jednorozměrné pole se v C/C++ dá jednoduše deklarovat podobně jako proměnná určitého datového typu (např. *char, int, short, float, …*). Za název proměnné se pouze přidají hranaté závorky, do kterých uvedeme rozměr tohoto pole. Například pro pole 1x5 s názvem pole, které bude obsahovat datové typy *char*, by to vypadalo následovně:

char pole[5]

Hodnota uvedená v hranatých závorkách musí být celočíselná konstanta (např. 10, hodnota *const*, konstantní výraz typu *8*sizeof(int), …*). Naplnění a výpis tohoto pole lze efektivně udělat pomocí cyklu *for*.

1.2 Statické dvourozměrné pole

1.2.1 Zadání

Vytvořte jednorozměrné statické pole z předchozího příkladu 1.1 a dvourozměrné statické pole o rozměrech 2x10, do jehož prvního řádku překopírujete jednorozměrné pole, do jeho druhého řádku překopírujete jednorozměrné pole tak, že na nejnižším indexu dvourozměrného pole bude uložena hodnota nejvyššího indexu jednorozměrného pole a na nejvyšším indexu dvourozměrného pole bude uložena hodnota nejnižšího indexu jednorozměrného pole → jednorozměrné pole bude ve druhém řádku dvourozměrného uloženo obráceně. Tedy v prvním řádku budou hodnoty vzestupně od 0 do 9, ve druhém řádku budou hodnoty sestupně od 9 do 0.

1.2.2 Rozbor úlohy

Dvourozměrné pole si můžeme představit jako tabulku, která má jak řádky tak i sloupce dat. C/C++ neposkytuje zvláštní typ dvourozměrného pole, místo toho vytváříme pole, jehož každým prvkem je samo pole. Chceme-li vytvořit pole o rozměrech 2x3 s názvem pole, které bude obsahovat proměnné typu *char*, můžeme použít deklaraci:

char pole[2][3];

První hranatá závorka reprezentuje počet řádků, druhá počet sloupců. Indexy tohoto pole jsou pak rozmístěny následovně:

0,0	0,1	0,2
1,0	1,1	1,2

Chceme-li potom přistupovat k jednotlivým položkám, musíme naadresovat jak řádek tak i sloupec. K adresaci dvourozměrného pole je dobré použít dvou vnořených cyklů *for*, jeden pro naadresování řádku, druhý pro naadresování sloupce. Budou-li se řídící proměnné cyklů jmenovat *i* a *j*, lze potom pro adresaci psát:

int x = pole[i][j];

Z tohoto příkladu je zřejmé, že se do proměnné *x* uloží hodnota pole na adrese *i* a *j*, které musí obsahovat hodnoty *int*.

1.3 Dynamické jednorozměrné pole

1.3.1 Zadání

Vytvořte jednorozměrné dynamické pole o rozměru 1x5, které bude obsahovat vzestupně čísla datového typu *int* vzestupně od 0 do 4. K vytvoření použijte operátor *new*. Výsledek vypište na obrazovku. Inkrementujte ukazatel na pole a vypište pole znovu. Pozor, pole se nyní bude jevit jako pole 1x4 ! Uvolňete alokovanou pamět pomocí operátoru *delete*.

1.3.2 Rozbor úlohy

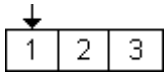
Jednorozměrné dynamické pole je jednoduché v C++ vytvořit. Zápis na alokaci jednorozměrného dynamického pole o 10-ti prvcích, které bude obsahovat hodnoty typu *int*, by vypadal asi takto:

*int * pole = new int [10];*

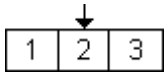
Příkaz vlastně vytvoří ukazatel s názvem *pole*, který ukazuje na první prvek o 10-ti hodnotách datového typu *int*. Je to vlastně pomyslný prst ukazující na tento prvek. Předpokládejme, že datový typ *int* zabírá čtyři bajty. Posunutím tohoto pomyslného prst o 4 bajty správným směrem, můžeme ukazovat na další prvek pole. Společně existuje 10 prvků → to je oblast, přes kterou můžeme náš prst posouvat. Příkaz *new* nám tedy poskytuje veškerou informaci potřebnou k identifikaci každého prvku bloku.
Přistoupit k prvnímu prvku pole není problém, protože **pole* ukazuje přímo na něj. Výhodou použití ukazatele je v tom, že k zápisu **pole* je ekvivalntní zápis *pole[0]*. Potom nám tedy zápis *pole[1]* vrací hodnotu druhého prvku pole, atd… . Aplikujeme-li příkaz *pole+=1*, zvýšíme ukazatel počátku na původní druhou položku pole, tedy druhá položka pole se stane první a počet indexů pole zbývajících do konce se sníží o 1. Pro použití příkazu *delete* k odalokování pole musí ukazatel ukazovat na původní adresu.

Pro ilustraci mějme pole s rozměrem 1x3, které bude obsahovat proměnné typu *int* a na které bude ukazovat ukazatel typu *int* s názvem *array*. Šipka u buňky znázorňuje adresu, na kterou ukazatel *array* právě ukazuje.

Vytvoříme pole příkazem *int *array = new int [3];* a naplníme je hodnotami 1, 2, 3. ukazatel *array* nám ukazuje na počátek tohoto pole.



Po použití příkazu *x=array[0];* nebo *x=*array;* bude proměnná *x*, která musí být *int*, obsahovat hodnotu 1. Příkaz *x=array[1];* nám vrátí do *x* hodnotu 2. Použitím příkazu *array+=1;* nebo *array=array+1;* zvýšíme hodnotu ukazatele o 1 a posuneme pomyslný prst doprava.



Potom nám *x=array[0];* vrátí do *x* hodnotu 2 a *x=array[1];* hodnotu 3. Poté uvedeme ukazatel do původního stavu příkazem *array-=1;* nebo *array=array-1;* a aplikujeme operátor *delete [] array*, který pole odalokuje.

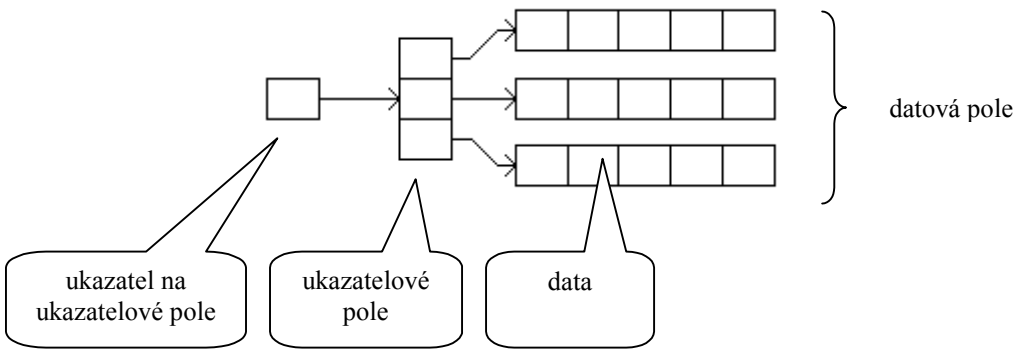
1.4 Dynamické dvourozměrné pole 1

1.4.1 – Zadání

Vytvořte čtvercové (pravoúhlé) dynamické dvourozměrné pole 5x4, které bude obsahovat hodnoty datového typu *int*. Naplňte toto pole libovolnou hodnotou *int* a vypište jej na obrazovku. Pole poté odalokujte. K alokaci pole použijte příkaz *malloc()*. Šířku a délku pole definujte pomocí makra *#define*.

1.4.2 Rozbor úlohy

Vytvoření dvourozměrného dynamického pole není zase tak jednoduché. Nejprve musíme vytvořit ukazatel na jednorozměrné ukazatelové pole. Počet položek ukazatelového pole je shodný jako počet řádků zadaného pole (jako rozměr *y*). Každá položka ukazatelového pole obsahuje ukazatel na příslušný řádek jednorozměrného datového pole, ve kterém jsou uloženy jednotlivá data. Je zcela evidentní, že počet datových polí je shodný jako počet položek ukazatelového pole. Každé datové pole má tedy rozměr *x*. Vše vysvětluje obrázek, kde je znázorněno pole 5x3:



Tento typ má velkou výhodu, že není vytvářen při překladu, ale při běhu programu. To znamená, že rozměry takovéhoto pole můžeme za běhu programu měnit. Také velikosti datových polí mohou být různé.

Nyní si předvedeme postup alokace pole, které je znázorněno na obrázku.

Vytvoříme si ukazatel na ukazatelové pole s názvem *array*. To nám zajistí příkaz:

```
int **array;
```

Nyní vytvoříme ukazatelové pole a nastavíme ukazatel *array*, aby na něj ukazoval.

```
array = (int**)malloc(3*sizeof(int));
```

Výraz *(int**)* vyjadřuje přetypování na typ ukazatel na ukazatel na *int*. Funkce *malloc* vyžaduje jako parametr velikost alokované paměti. V tomto případě potřebujeme vyhradit v paměti trojnásobnou velikost typu *int*, protože alokujeme paměť pro ukazatelové pole, které má tři položky. Poté již alokujeme jednotlivá pole pomocí příkazu:

```
array[0] = (int*)malloc(5*sizeof(int));
```

Zde vyjadřuje výraz *(int*)* přetypování na datový typ ukazatel na *int*, protože se již nacházíme v ukazatelovém poli. Naalokujeme tedy datové pole a nastavíme na něj příslušný ukazatel v ukazatelovém poli. Nyní potřebujeme vyhradit v paměti pětinásobnou velikost typu *int*, protože datové pole obsahuje pět položek. Obdobně bychom postupovali i s dalšími datovými poli. Jedná-li se o čtvercové dynamické dvourozměrné pole, můžeme třeba pro alokování datových polí použít cyklus *for*.

Přístup k jednotlivým položkám dynamického pole je obdobný jako přístup k položkám ve statickém poli:

```
int i = array[2][3]; //do promenne i ulozi hodnotu 3. radku a 4. sloupce
```

Odalokování provedeme pomocí příkazu *free()*, kde parametr je název naalokovaného místa. Nejprve odalokujeme jednotlivá datová pole, poté odalokujeme i ukazatelové pole.

1.5 Dynamické dvourozměrné pole 2

1.5.1 Zadání

Vytvořte dynamické dvourozměrné pole s proměnným počtem řádků. Počet řádků nadefinujte pomocí makra *#define*. Všechny řádky budou obsahovat hodnoty typu *int*. První řádek bude mít jednu položku. Druhý řádek bude mít dvě položky, třetí tři položky a každý následující řádek bude mít o jednu položku více. Každá položka bude obsahovat hodnotu 0. Demonstrujte tímto, že dynamické pole nemusí mít stejný rozměr datových řádků. Vypište toto pole na obrazovku a odalokujte jej.

1.5.2 Rozbor úlohy

Nejprve vytvoříme ukazatel na ukazatelové pole a ukazatelové pole jako v předešlé úloze. Ukazatelové pole bude mít rozměr jako hodnota definovaná makrem *#define* v úvodu programu (počet řádků). Poté pomocí cyklu *for* adresujeme jednotlivé prvky ukazatelového pole. Nyní naalokujeme datové pole. Počet naalokovaných prvků datového pole je o jedna vyšší než číslo (index) řádku ukazatelového pole, na kterém alokujeme (indexy ukazatelového pole začínají od 0). Naplnění a výpis pole provedeme pomocí dvou vnořených cyklů *for*, přičemž první *for* adresuje řádek a druhý *for*, jehož maximální hodnota je o jedna vyšší než řídící proměnná prvního *for* cyklu (indexy ukazatelového pole začínají od 0), adresuje sloupec. Druhý *for* je závislý na prvním. Dealokace tohoto pole a přístup k jednotlivým položkám jsou totožné s předchozím příkladem.

2. Třída *string* v jazyce C++

2.1 Konstruktory třídy *string*

Pozn.: K použití třídy *string* je zapotřebí zavést knihovnu *string* pomocí příkazu *#include*.
K použití příkazu *cin* a *cout* je zapotřebí zavést knihovnu *iostream* pomocí příkazu *#include*.

2.1.1 Zadání

Vytvořte šest objektů třídy *string* pomocí šesti různých konstruktorů, které tato třída podporuje. Budou to objekty a konstruktory obsažené v následující tabulce

č.	Jméno objektu	Typ konstruktoru	Řetězec
1	a	*string (const char *s)*	Hello world
2	b	*string (size_type n, char c)*	xxxxx
3	c	*string (const string *str)*	Ahoj…..
4	d	*string()*	
5	e	*string (const char *s, size_type n)*	Ahoj.
6	f	*template <class iter>; string (begin iter, end iter)*	…..

Jméno třídy udává, jak se výsledný objekt bude jmenovat, typ konstruktoru udává, jaký typ konstruktoru máme použít a řetězec nám říká, jaký výsledný řetězec bude objekt obsahovat. K výpisu jednotlivých řetězců na obrazovku použijte přetížený operátor << a příkaz *cout* z knihovny *iostream*.

2.1.2 Rozbor úlohy

Vytvoření objektu pomocí.prvního konstruktoru *string (const char *s)* je velice jednoduché, lze to udělat pomocí:

string nazev_objektu(“retezec”);

Kde *nazev_objetu* je název výsledného objetu a *retezec* je řetězec, který má objekt obsahovat. Tento konstruktor vlastně inicializuje objekt pomocí NBTS (zkratka označující řetězec zakončený bajtem s nulovou hodnotou, tedy tradiční řetězec jazyka C zakončený nulovou hodnotou), na který ukazuje *s*.
Druhý konstruktor *string (size_type n, char c)* inicializuje objekt typu *string* s délkou *n*, kde každý prvek tohoto řetězce bude inicializován na hodnotu *c*. Například vytvoříme objekt *string* s názvem *retezec*, který bude mít 10 znaků. Každý znak tohoto objektu bude mít hodnotu „*a*“. Uděláme to pomocí:

string retezec(10,’a’);

Třetí konstruktor inicializuje objekt *string* pomocí řetězce. Vytvoření takovéhoto objetu provedeme pomocí:

string nazev(retezec);

Kde parametr *retezec* je již existující řetězec a *nazev* je právě vytvořený nový objekt.
Čtvrtý konstruktor vytvoří implicitní objekt této třídy s nulovou velikostí. Syntaxe takovéhoto vytvoření je:

string nazev;

Pátý konstruktor *string (const char *s, size_type n)* inicializuje objekt *string* pomocí NBTS, na který ukazuje ukazatel *s*, má délku *n* znaků. Počet znaků může být delší než délka řetězce NBTS. Pro ilustraci, máme řetězec *char retezec[7] = “abcdef”* a chceme z něj vytvořit pomocí pátého konstruktoru nový objekt. Provedeme to následovně:

string objekt(retezec, 3);

Výsledný objekt bude obsahovat řetězec „*abc*“.
Šestý konstruktor *string (begin iter, end iter)* inicializuje objekt typu *string* z intervalu <*begin, end*>, přičemž *begin* a *end* se chovají jako ukazatele a určují místo v paměti. Do dané oblasti patří pouze *begin*, *end* tam nepatří. Pro příklad mějme řetězec *char retezec[7] = "abcdef"* a chceme z něj vytvořit objekt *string* pomocí šestého konstruktoru. Provedeme:

string nazev(retezec +2, retezec +4);

Výsledkem bude objekt s názvem *nazev* a s hodnotou „*cd*“.
Výpis na obrazovku provedeme pomocí příkazu:

cout << nazev << endl;

Kde *nazev* je jméno objektu a *endl* zajišťuje odřádkování. Operátor << je přetížený.

2.2 Přetěžování operátorů ve třídě *string*

2.2.1 Zadání

Vytvořte tři objekty třídy *string*. První objekt se jménem *a* bude vytvořen konstruktorem typu *string (const char *s)* a bude obsahovat hodnotu „*Hello*“. Druhý se jménem *b* bude vytvořen konstruktorem typu *string (size_type n, char c)* a bude obsahovat jednu mezeru. Třetí objekt se bude jmenovat *c* a vytvoříte jej konstruktorem *string (const char *s)*, bude obsahovat slovo „*world"*. Vytvořte prázdný objekt *d*. Přetížením operátorů =, +=, << a použitím objektů *a*, *b*, *c* zajistěte, aby *d* obsahovalo text „*Hello world*“. Objekt *d* vypište na obrazovku.

2.2.2 Rozbor úlohy

Knihovna *string* obsahuje, jak je vidět z předchozího příkladu, hodně konstruktorů. Obsahuje ale také mnoho přetížených operátorů pro přiřazování, spojování (sčítání) a porovnávání (komparaci) řetězců.
V tomto příkladě použijeme přetížení těchto operátorů:

- = přiřazení
- += součet řetězců
- << s *cout* slouží pro výpis na obrazovku

Ukažme si příklad na přetěžování operátorů: Mějme tři řetězce:

string a("aaa");
string b("bbb");
string c("ccc");

Přetížíme-li nyní operátor = provedením příkazu:

a = b;

přepíšeme hodnotu objektu *a* hodnotou objektu *b*. Objekt *a* bude tedy obsahovat hodnotu „*bbb*“. Nyní použijeme přetížený operátor += ke sloučení dvou řetězců:

b+=c;

Objekt *b* obsahoval hodnotu „*bbb*“, když jej sloučíme s objektem *c*, dostaneme v objektu *b* hodnotu „*bbbccc*“.
Pro výpis na obrazovku použijeme přetíženého operátoru << s příkazem *cout*.

cout << b << endl;

Kde *endl* nám zajistí odřádkování na další řádek.

2.3 Porovnávání řetězců a velikosti řetězců

2.3.1 Zadání

Vytvořte tři objekty třídy *string* se jmény *a*, *b*, *c*. Objekty *a*, *b* budou mít stejnou hodnotu „*aaa*“. Objekt *c* bude obsahovat hodnotu „*abc*“. Porovnejte v programu, jestli má objekt *a* menší hodnotu než objekt *c*, jestli je *a* totožný s *b*, jestli je *a* jiný než *b*. Potom zjistěte, jestli objekt *a* má stejnou velikost v paměti jako *b*.

2.3.2 Rozbor úlohy

Opět využijeme přetěžování operátorů. Ve třídě *string* jsou přetížené také operátory <, >, ==, *!*=, můžeme je tedy použít. Tyto operátory (<, >, ==, *!*=) slouží k porovnání dvou řetězců (menší, větší, shodný, různý) a můžeme je použít v podmínkách *if* . Každý z těchto relačních operátorů je přetížen třemi způsoby, takže objekt třídy *string* můžeme porovnat s jiným objektem téže třídy, s řetězcem jazyka C a řetězec jazyka C můžeme porovnat s objektem třídy *string*. Za menší se považuje ten objekt, který se v posloupnosti řazení počítače vyskytuje dříve.
Vše vysvětlí příklad. Máme dva objekty třídy *string* s názvy *x*, *y*. Objekt *x* obsahuje hodnotu „*i*“, objekt *y* obsahuje hodnotu „*j*“. Vytvoříme podmínku:

if(x == y) { cout << “Ahoj”; }

Příkaz *cout* uvnitř podmínky se neprovede, protože podmínka není splněna a slovo „Ahoj“ se tedy nevypíše. Velikost řetězce vrací členská funkce *length()* nebo *size()*. Obě dělají totéž, ale funkce *length()* pochází z dřívějších verzí třídy *string*, kdežto funkce *size()* byla přidána kvůli kompatibilitě se standardní knihovnou šablon. Pro ilustraci inicializujeme objekt *string x(“i”);* a objekt *string y(“j”);* a poté porovnáme velikost obou objektů pomocí podmínky:

if (x.length() == y.length()) { cout << “Ahoj”; }

Nyní se příkaz *cout* uvnitř podmínky provede, protože je podmínka splněna a slovo „Ahoj“ se vypíše.

2.4 Vyhledávání podřetězce v řetězci

2.4.1 Zadání

Vytvořte řetězec s názvem „*retezec*“, který bude obsahovat text „Prakticke programovani v C/C++“. Vyzkoušejte členské funkce třídy *string* uvedené v tabulce.

č.	Zápis funkce	Parametr
1	retezec.find(const char *s, size_type pos = 0)	“programovani”, 0
2	retezec.rfind(const char *s, size_type pos = 0)	‘a’
3	retezec.find_first_of(const char *s, size_type pos = 0)	‘ef’
4	retezec.find_last_of(const char *s, size_type pos = 0)	‘ef’
5	retezec.find_first_not_of(const char *s, size_type pos = 0)	‘ef’
6	retezec.find_last_not_of(const char *s, size_type pos = 0)	‘ef’

Sloupec parametr nám udává, s jakým parametrem budeme volat funkci na příslušném řádku. Vypište indexy, které našly tyto členské funkce, na obrazovku.

2.4.2 Rozbor úlohy

Všechny funkce v tabulce mají ještě další čtyři varianty, jak je volat. My se však omezíme pouze na jednu variantu pro každou funkci, na tu, která je pro každou z nich uvedena v tabulce.

První parametr každé funkce uvedené v tabulce obsahuje vždy hledaný znak nebo podřetězec. Druhý parametr funkce uvedené v tabulce udává index, od kterého se začne hledat a je pro všechny funkce v tabulce stejný.

První funkce najde výskyt a vrátí index prvního podřetězce nebo znaku ve volaném řetězci.
Druhá funkce vrací index posledního znaku nebo podřetězce.

Třetí funkce najde ve volaném řetězci první výskyt libovolného ze znaků v prvním parametru.
Čtvrtá funkce najde ve volaném řetězci poslední výskyt libovolného ze znaků v prvním parametru.
Pátá funkce najde ve volaném řetězci první výskyt libovolného ze znaků, které nejsou v prvním parametru.
Šestá funkce najde ve volaném řetězci poslední výskyt libovolného ze znaků, které nejsou v prvním parametru.

Mějme například řetězec s názvem „*retezec*“, který obsahuje text „Ahoj“. Zadáme příkaz:

int x = retezec.find_first_of('A', 0);

Do proměnné *x* se nám uloží index prvního velkého *A* v řetězci, tedy hodnota 0.

3. Vlastní třída String v jazyce C++

3.1 Vytvoření třídy

3.1.1 Zadání

Definujte vlastní třídu s názvem *String* (s velkým S na začátku). V sekci *public* bude třída obsahovat proměnnou *txt*, která bude typu ukazatel na *char* a proměnnou *delka*, která bude typu *int*. Vytvořte objekt *a*, který bude typu *String*. Proměnnou *délka* změňte na hodnotu 10.

3.1.2 Rozbor úlohy

Třída se definuje klíčovým slovem class, za které se do složených závorek uvede její tělo, obsahující vlastní data a metody. Tyto data a metody se definují jako normální proměnné, či funkce v C/C++. Existují tři klíčová slova, která mají zásadní vliv na to, kdy se daná data, či metoda mohou použít, či která funkce je může použít, či s nimi manipulovat. Jsou to:

- *private* – datové členy jsou přístupné pouze zevnitř dané třídy, jsou tedy lokální
- *public* – členy jsou přístupné i vně třídy
- *protected* – viz. *private*, rozdíl se projevuje při dědění

Hned na začátku jakékoli definované třídy je implicitně sekce *private*, změní ji až klíčové slovo *public* nebo *protected*. Klíčové slovo *private* se nemusí tedy na začátku jakékoli třídy zadávat. Implicitní nastavení třídy vypadá tedy následovně:

```
class pokus
{
private:
//
// data a metody
//
}
```

Přístup k jednotlivým datům se provádí pomocí tečky za názvem třídy a za tečkou uvedeme název příslušné proměnné, se kterou chceme manipulovat. Mějme třídu

```
class pokus
{
public:
int a;
char b;
}
```

Potom příkazem *pokus.b=10;* přiřadíme do proměnné *b* hodnotu 10.

3.2 Vytvoření konstruktorů a destruktoru

3.2.1 Zadání

Ke třídě z předchozího příkladu přidejte do sekce *public* proměnné *Aktivnich*, *Poradi*, *Index*, které budou typu *int*. Proměnné *Aktivnich* a *Poradi* budou statické datové členy. Do této sekce nadefinujte implicitní konstruktor, konverzní konstruktor pro načtení typů *float*, kopykonstruktor pro načtení řetězce *char* a kopykonstruktor pro zkopírovaní existující třídy *String* do třídy *String*, která právě vzniká. V programu vytvořte několik objektů této třídy tak, abyste vyzkoušeli všechny vytvořené konstruktory. Všechny konstruktory budou dále přiřazovat do proměnné Index pořadí posledního objektu třídy String, poté inkrementují proměnné *Poradi* a *Aktivnich*. Vytvořte, doplňte a nadefinujte destruktor třídy *String*, který dekrementuje proměnnou *Aktivnich* a správně odalokuje paměť proměnné *txt* pomocí operátoru *delete* a do proměnné *delka* uloží hodnotu 0. Proměnná *Aktivnich* tedy určuje skutečný počet objektů, které jsou v programu v daný okamžik přítomny. Kdežto proměnná *Poradi* určuje pořadí vytvoření objektu *String* od začátku běhu programu, i když už nějaké objekty byly zrušeny.

3.2.2 Rozbor úlohy

Statický datový člen třídy je výhodné použít v případě, je-li nutné mít v rámci celé třídy společnou proměnnou. Deklaruje se jako normální proměnná, akorát před typ proměnné uvedeme klíčové slovo *static*. Příklad:

static int promenna;

Konstruktor má vždy stejný název jako jméno třídy a nikdy nemá návratovou hodnotu. Konstruktorů může mít daná třída několik a v tom případě platí pravidla pro přetěžování funkcí. Základní typy konstruktorů:

- Implicitní konstruktor – bez parametrů
- Kopykonstruktor – parametrem je odkaz na objekt stejné třídy
- Konverzní konstruktor – má jeden parametr, slouží k převodu jednoho datového typu na druhý

V našem případě bude implicitní konstruktor obsahovat pouze rutiny obsluhující proměnné *Aktivnich*, *Poradi* a *Index*. Dále naplní proměnnou *txt* hodnotou *NULL* a proměnnou *delka* hodnotou 0.
Konverzní konstruktor pro načítání typů *double* bude také obsahovat rutiny obsluhující proměnné *Aktivnich*, *Poradi* a *Index*. Dále však musí obsahovat příkazy, které zajistí konverzi formátu *double* na řetězec. Struktura takovéhoto konstruktoru by tedy mohla být:

1) Přiřazení proměnné *Poradi* do proměnné *Index*, inkrementace proměnných *Poradi* a *Aktivnich*.
2) Vytvoření a deklarace dostatečně dlouhého pomocného řetězce. Například *char pomoc[50]*.
3) Zapsání parametru typu double do pomocného řetězce pomocí funkce:
   *sprintf(char *, const char *, ...);*, kde první parametr funkce je název pomocného řetězce, druhý parametr je formát zapisovaných dat a třetí parametr je zapisovaný typ *double* (parametr v hlavičce konstruktoru)
4) Do proměnné délka uložíme délku řetězce, pomocí funkce *strlen(…)*.
5) Vytvoření nové proměnné *txt* pomocí operátoru *new* (nezapomenout na přetypování na *char**), která bude obsahovat o jednu položku více, než je uloženo v proměnné délka (kvůli zakončovacímu znaku).
6) Zkopírujeme pomocný řetězec do proměnné *txt*, například pomocí *for* cyklu.
7) Na poslední pozici proměnné *txt*, která je volná, uložíme zakončovací znak *'/0'*.

Postup při vytváření kopykonstruktoru *String(const char *t)*, který kopíruje řetězec z parametru do proměnné *txt*, by mohl mít šablonu:

1) Přiřazení proměnné *Poradi* do proměnné *Index*, inkrementace proměnných *Poradi* a *Aktivních*.
2) Do proměnné délka uložíme délku řetězce, pomocí funkce *strlen(…)*.
3) Test, jestli není řetězec nulový. Jestli není, tak vytvoříme novou proměnnou *txt* pomocí operátoru *new* (nezapomenout na přetypování na *char**), která bude obsahovat o jednu položku více, než je uloženo v proměnné délka (kvůli zakončovacímu znaku). Řetězec pomocí *for* cyklu překopírujeme z řetězce v parametru. Jestli je řetězec nulový, uložíme do *txt* hodnotu *NULL* a do proměnné *Delka* hodnotu 0.

Postup při vytváření kopykonstruktoru *String(const String &b),* který kopíruje řetězec *txt* jiné třídy *String* do nově vznikající třídy *String* a její proměnné *txt*, je následující:

1) Přiřazení proměnné *Poradi* do proměnné *Index*, inkrementace proměnných *Poradi* a *Aktivních*.
2) Do proměnné délka uložíme délku řetězce, pomocí *Delka=b.Delka;*, protože délku řetězce známe díky tomu že ji máme uloženou v objektu třídy *String* ze kterého kopírujeme.
3) Test, jestli není řetězec nulový. Jestli není, tak vytvoříme novou proměnnou *txt* pomocí operátoru *new* (nezapomenout na přetypování na *char**), která bude obsahovat o jednu položku více, než je uloženo v proměnné délka (kvůli zakončovacímu znaku). Řetězec pomocí *for* cyklu překopírujeme z *b* a jeho proměnné *txt*. Jestli je řetězec nulový, uložíme do *txt* hodnotu *NULL* a do proměnné *Delka* hodnotu 0.

Destruktor je vždy pouze jeden a nemá žádné parametry ani návratovou hodnotu. Je vždy volán při destrukci nějakého objektu dané třídy. V našem případě dekrementuje proměnnou *Aktivnich*. Pokud obsahuje nějaké hodnoty, smaže řetězec (pomocí operátoru *delete*) na který ukazuje *txt*. Do ukazatele *txt* uloží *NULL* a do proměnné délka přiřadí 0.

2. Vzorové příklady:

Statické jednorozměrné pole

```
#include <stdio.h>

void main(void)
{
	int i;
	int pole[10];                                   //nadeklarování pole
	for(i=0; i<10; i++)                             //cyklus for, kterým plníme pole
	{
		pole[i]=i;
		printf("%i  ",pole[i]);                 //zároveň vypisujeme
	}
}
```

Statické dvourozměrné pole

```
#include <stdio.h>

void main(void)
{
	int i,j;
	int pole[10];
	int array[2][10];
	for(i=0;i<10;i++)
	{
		pole[i]=i;
    }
	for(i=0;i<2;i++)
	{
		for(j=0;j<10;j++)
		{
			if(i<1)
			{
				array[i][j]=pole[j];
				printf("%i  ",array[0][j]);
			}
			else
			{
				array[i][j]=9-pole[j];
				printf("%i  ",array[1][j]);
			}
		}
		printf("\n");
	}
}
```

Dynamické jednorozměrné pole

```
#include <stdio.h>                                      //vložení hlavičkových souborů

void main(void)
{
	int i;
	int *pole=new int[5];                           //vytvoření pole
	for(i=0;i<5;i++)                                //naplnění a výpispole
	{
		pole[i]=i;
		printf("%i  ",pole[i]);
	}
	pole+=1;                                        //zvýšení ukazatele
	printf("\n");                                   //odřádkování
	for(i=0;i<4;i++)                                //výpis "nového" pole
	{
		printf("%i  ",pole[i]);
	}
	pole-=1;                                        //vrácení ukazatele
	delete [] pole;                                 //odalokování pole
}
```

Dynamické dvourozměrné pole 1

```
#include <stdio.h>                                  //přidání hlavičkových souborů
#include <malloc.h>

#define x 5                                         //definice rozměrů pole
#define y 4

void main(void)
{
        int i,j;
        int **pole;                                 //vytvoření ukazatele na pole ukazatelů
        pole = (int**)malloc(y*sizeof(int));        //vytvoření pole ukazatelů
        for(i=0;i<y;i++)                            //vytvoření jednotlivých polí s daty
        {
                pole[i] = (int*)malloc(x*sizeof(int));
        }
        for(i=0;i<y;i++)                            //cykly pro naplnění a výpis polí
        {
                for(j=0;j<x;j++)
                {
                        pole[i][j]=1;               //naplnění
                        printf("%i",(pole[i][j]));  //výpis
                }
                printf("\n");                       //odřádkování po naplnění řádku
        }
        for(i=0;i<y;i++)                            //dealokace polí s daty
        {
                free(pole[i]);
        }
        free(pole);
}
```

Dynamické dvourozměrné pole 2

```
#include <stdio.h>                                  //přidání hlavičkových souborů
#include <malloc.h>

#define n 5                                         //definice počtu řádků

void main(void)
{
        int i,j;
        int **pole;                                 //ukazatel na na pole ukazatelů
        pole=(int**)malloc(n*sizeof(int));          //alokace n-rozměrného pole ukazatelů
        for(i=0;i<n;i++)                            //alokace n datových polí s i+1 položkami
        {
                pole[i]=(int*)malloc((i+1)*sizeof(int));
        }
        for(i=0;i<n;i++)                            //naplnění a výpis pole
        {
                for(j=0;j<(i+1);j++)                //plní a vypisuje řádek až do jeho konce
                {
                        pole[i][j]=0;
                        printf("%i",pole[i][j]);
                }
                printf("\n");                       //odřádkování za koncem řádku
        }
        for(i=0;i<n;i++)                            //uvolnění paměti
        {
                free(pole[i]);
        }
        free(pole);
}
```

Konstruktory třídy *string*

```
#include <iostream>                                 //Přidání hlavičkových souborů
#include <string>

using namespace std;

void main(void)
```

```
{
	string a("Hello world");			//první konstruktor
	cout << a << endl;				//výpis na obrazovku a přetížení <<
	string b(5,'x');				//druhý konstruktor
	cout << b << endl;				//výpis na obrazovku a přetížení <<
	string c(a);					//třetí konstruktor
	cout << c << endl;				//výpis na obrazovku a přetížení <<
	string d;					//čtvrtý konstruktor
	cout << d << endl;				//výpis na obrazovku a přetížení <<
	char retezec[10] = "Ahoj.....";			//vytvoření demonstračního řetězce
	string e(retezec,5);				//pátý konstruktor
	cout << e << endl;				//výpis na obrazovku a přetížení <<
	string f(retezec +5,retezec +10);		//šestý konstruktor
	cout << f << endl;				//výpis na obrazovku a přetížení <<
}
```

Přetěžování operátorů ve třídě *string*

```
#include <iostream>					//definování knihoven
#include <string>

using namespace std;

void main(void)
{
	string a("Hello");				//vytvoření 1. řetězce
	string b(1,' ');				//vytvoření 2. řetězce
	string c("world");				//vytvoření 3. řetězce
	string d;					//vytvoření 4. řetězce
	d = a;						//přetížení operátoru =
	d += b;						//přetížení operátoru +=
	d += c;
	cout << d << endl;				//přetížení operátoru <<
}
```

Porovnávání řetězců a velikostí řetězců

```
#include <iostream>					//přidání hlavičkových souborů
#include <string>

using namespace std;

void main(void)
{
	string a("aaa");				//vytvoření řetězců
	string b("aaa");
	string c("abc");
	if(a<c)						//porovnání jestli je a mensi nez c
	{
		cout << "a < c" << endl;
	}
	if(a==b)					//porovnání jestli jsou a i b stejné
	{
		cout << "a == b" << endl;
	if(a!=b)					//porovnání jestli jsou a i b různé
	{
		cout << "a != b" << endl;
	}
	if(a.length()==b.length())			//porovnání jestli jsou a i b stejně velké
	{
		cout << "a i b jsou stejne velke" << endl;
	}
}
```

2.4 Vyhledávání podřetězce v řetězci

```
#include <iostream>
#include <string>

using namespace std;

void main(void)
{
	string retezec("Prakticke programovani v C/C++");
	int index = retezec.find("programovani",0);
```

```
        cout << "Hledany index podretezce: " << index <<endl;
        index = retezec.rfind('a');
        cout << "Index posledniho vyskytu znaku 'a': " << index << endl;
        index = retezec.find_first_of("ef");
        cout << "Index prvniho vyskytu znaku 'e' nebo 'f': " << index << endl;
        index = retezec.find_last_of("ef");
        cout << "Index posledniho vyskytu znaku 'e' nebo 'f': " << index << endl;
        index = retezec.find_first_not_of("ef");
        cout << "Index prvniho znaku, ktery neni 'e' nebo 'f': " << index << endl;
        index = retezec.find_last_not_of("ef");
        cout << "Index posledniho znaku, ktery neni 'e' nebo 'f': " << index << endl;
}
```