

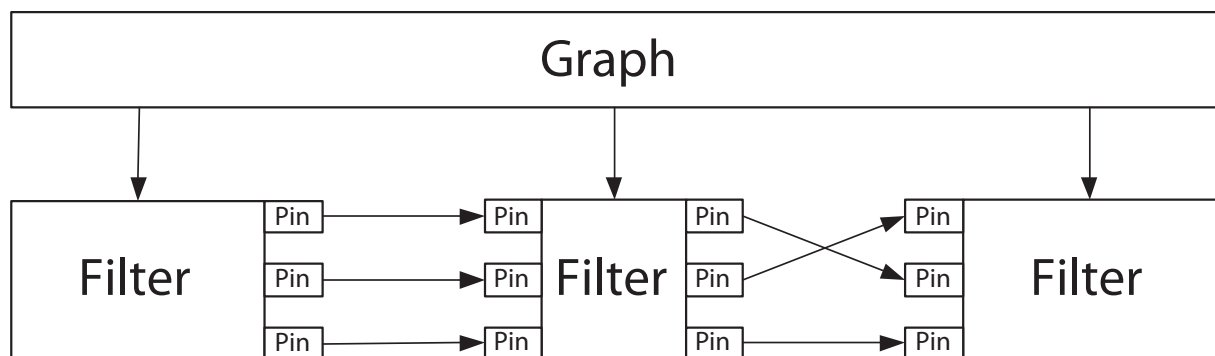
Architektúra FILTERGRAPH

Obsah

1. Architektúra	2
2. Trieda TPin a komunikácia	2
3. Trieda TFilter	5
4. Trieda TGraph	7
5. Knižnica filtrov	9
6. Základné dátové typy	10
7. Návrátové a chybové kódy	12
8. Iné použité triedy	12
8.1 Trieda TSerializer	12
8.2 Trieda CBitStream	13
9. Príklad odvodenej triedy: Trieda TCropFilter	15
cropresource.h (definícia zdrojov):	15
cropecfg.h (definícia konfiguračnej štruktúry):	15
crop.h (definícia triedy filtra):	16
crop.cpp (implementácia triedy filtra):	17
cropdll.h (definícia knižnice):	26
cropdll.cpp (implementácia knižnice):	26

1. Architektúra

Architektúra Filtergraph je implementáciou dizajnového vzoru „graf filtrov“. Umožňuje riadenie toku dát jednoduchou zmenou grafu alebo zmenou nastavení filtrov. Tento návrh nepodporuje spracovanie tokov dát (streaming), pracuje len so statickými dátami. Základom architektúry sú tri základné triedy: TGraph, TFilter a TPin. Trieda TPin je základné komunikačné rozhranie poskytujúce funkcie spájania pinov a predávania dát medzi nimi. Trieda TFilter je virtuálna trieda, ktorá poskytuje priestor pre implementáciu vlastných algoritmov. Objekty triedy TFilter komunikujú medzi sebou pomocou objektov triedy TPin. TFilter riadi základné operácie s pinmi ako pridávanie a odoberanie pinov. Každá odvodená trieda má priestor na implementáciu hlavného algoritmu, konfiguračných funkcií ako napr. konfiguračný dialóg, inicializačnej a ukončovacej funkcie. Trieda TGraph riadi spúšťanie jednotlivých filtrov, zabezpečuje beh a ukončenie spracovania, monitoruje priebeh spracovania.



Obrázok 1: Architektúra Filtergraph.

Header: filtergraph.h

Lib: filtergraph.lib

2. Trieda TPin a komunikácia

Trieda TPin je komunikačné rozhranie poskytujúce funkcie na predávanie dát a kontrolu stavu prístupnosti dát. Objekt triedy TPin je prijímacie zariadenie, ktoré žiada o dáta z pripojeného pinu (pin si pamätá len to, od koho má prijímať dáta). Každý pin môže byť pripojený len na jeden pin, ale na neho môže byť pripojených viacero pinov. Pin je charakterizovaný typom (neznámy, vstupný, výstupný, konfiguračný, iný, ...), a môže byť pomenovaný. Dáta medzi pinmi sú prenášané v štruktúre TBuffer. Táto štruktúra zaobaluje pamäťový buffer. Obsahuje informácie o type, veľkosti a umiestnení dát v pamäti. Štruktúra TBuffer je definovaná takto:

```
typedef struct {
    int      size;           // veľkosť dát
    int      type;          // typ dát
    void*    data;          // ukazovateľ na dáta v pamäti
} TBuffer;
```

K dispozícii sú základné globálne funkcie na prácu s bufframi:

```
void ClearBuffer(TBuffer *buffer);
```

- vyčistí *buffer*, nastaví hodnoty v štruktúre na základné (prázdne)

*void FreeBuffer(TBuffer *buffer);*

- vyčistí *buffer* a uvoľní alokovanú pamäť

*TBuffer MakeBuffer(int type, int size, void *data);*

- vytvorí štruktúru *TBuffer* so vstupných parametrov *type*, *size* a *data*, vráti skonštruovaný *buffer*

TBuffer CopyBuffer(TBuffer src);

- alokuje potrebné pamäťové miesto a skopíruje obsah buffra *src* do nového buffra, ktorý vráti

Trieda *TPin* je definovaná nasledovne:

```
class TPin {
private:
    int      Type;           // typ pinu
    TPin*    Target;         // cieľový pin, od ktorého si tento pin žiada dáta
    TBuffer  Buffer;         // buffer s poskytovanými alebo prijatými dátami
    bool     Connected;     // indikátor pripojenia
    bool     DataReady;     // indikátor pripravenosti dát
    char*    Name;          // meno pinu
public:
    TPin();                 // základný konštruktor
    TPin(char* name, int type); // konštruktor, nastaví meno a typ pinu
    ~TPin();                // deštruktor

    int connect(TPin *pin); // pripojí sa na cieľový pin
    int disconnect();       // odpojí sa od cieľového pinu
    int receiveData();      // vyžiada si dáta od cieľového pinu a dá ich do buffra
    int setData(TBuffer buffer); // nastaví poskytované dáta na buffer a zapne indikátor
    int freeData();        // uvoľní buffer a vypne indikátor pripravených dát
    int clearData();       // vyčistí buffer a vypne indikátor pripravených dát
    int setDataReady();    // zapne indikátor pripravených dát
    int setDataNotReady(); // vypne indikátor pripravených dát
    int clearBuffer();     // vyčistí buffer
    int freeBuffer();      // uvoľní buffer

    int      getType();     // vráti typ pinu
    TPin*    getTarget();   // vráti cieľový pin
    TBuffer  getBuffer();   // vráti buffer
    char*    getName();     // vráti meno
    bool     isConnected(); // vráti stav indikátoru pripojenia
    bool     isDataReady(); // vráti stav indikátoru pripravených dát

    int setType(int type); // nastaví typ pinu
    int setTarget(TPin *pin); // nastaví cieľový pin
    int setBuffer(TBuffer buffer); // nastaví buffer
    int setName(char* name); // nastaví meno pinu
};
```

Všetky metódy, ktoré majú návratový typ *int* a nie sú „getre“ (metódy s názvom začínajúcim na *get* alebo *is*), vracajú návratové alebo chybové kódy, ktoré budú neskôr popísané.

Základné operácie s pinmi:

Konštrukcia:

```
TPin *pin1 = new TPin(); // vytvorenie pinu bez nastavení
TPin *pin2 = new TPin(„Output“, PT_OUTPUT); // vytvorenie pinu s menom a typom
```

Pripojenie:

```
int rc; // návratový kód
rc = pin1->connect(pin2); // pin1 sa pripojí na pin2, pin2 o tom vôbec nevie
```

Poskytnutie dát:

```
int i = 200; // hodnota, ktorú chceme poskytnúť
TBuffer buffer = MakeBuffer(DT_INT, sizeof(int), &i); // vytvoríme buffer
pin2->setData(buffer); // sprístupníme dáta, odteraz si ich môže ktokoľvek prečítať
```

Prevzatie dát:

```
int i; // hodnota, ktorú chceme prevziať
int rc; // návratový kód
TBuffer buffer; // prijímací buffer
rc = pin1->receiveData(); // prijatie dát
if (rc == RC_OK) { // ak prijímanie prebehlo v poriadku,
    buffer = pin1->getBuffer(); // zoberieme prijatý buffer,
    if (buffer.data) { // skontrolujeme ho,
        i = (int*) buffer.data; // a vyberieme si jeho obsah
    }
}
```

Dáta sa medzi pinmi len sprístupňujú, **nie sú** kopírované z dôvodu šetrenia pamäťou. Poskytnutím prístupu môžu byť dáta iným filtrom samozrejme aj modifikované.

Zneprístupnenie dát:

```
pin2->setDataNotReady(); // zneprístupní dáta, ale ponechá si nezmenený buffer
alebo
pin2->clearData(); // zneprístupní dáta a vyčistí buffer bez uvoľnenia pamäte
alebo
pin2->freeData(); // zneprístupní dáta a uvoľní alokovanú pamäť
```

Odpojenie:

```
pin1->disconnect(); // odpojí pin od cieľového pinu
```

Deštrukcia:

```
delete pin1; // zavolá deštruktor
delete pin2; // zavolá deštruktor
```

Základné typy pinov (Pin Type):

PT_UNKNOWN	0	// neznámy, nedefinovaný typ pinu
PT_INPUT	1	// vstupný pin
PT_OUTPUT	2	// výstupný pin
PT_CONFIGIN	3	// vstupný konfiguračný pin
PT_CONFIGOUT	4	// výstupný konfiguračný pin

Užívateľ nie je limitovaný na tieto typy pinov, môže si podľa potreby zadať vlastné typy pinov (v rozsahu typu *int*).

Header: filtergraph.h

Lib: filtergraph.lib

3. Trieda TFilter

Trieda TFilter poskytuje priestor na implementáciu vlastných algoritmov formou odvodených tried. Každá odvodená trieda by mala byť identifikovaná vlastným unikátnym číslom – Magic number. Informácie o sebe poskytuje v štruktúre TFilterInfo:

```
typedef struct {  
    char*      FilterName;           // meno triedy filtrov  
    UINT       MajorVersion;         // významnejšia časť verzie  
    UINT       MinorVersion;         // menej významná časť verzie  
    int        Magic;                // Magic number  
} TFilterInfo;
```

Verzia triedy je kódovaná ako *MajorVersion.MinorVersion*.

Trieda TFilter je definovaná nasledovne:

```
class TFilter {  
protected:  
    TPin**     Pins; // pole viditeľných zaregistrovaných pinov  
    int        cPins; // počet zaregistrovaných pinov  
  
    int        State; // stav filtra  
    TGraph*    ParentGraph; // ukazovateľ na rodičovský graf  
    ProgressCallback ProgressFunc; // callback funkcia monitorovania priebehu  
    TFilterInfo FilterInfo; // informácie o triede  
  
    int addPin(TPin *pin); // zaregistrovanie viditeľného pinu  
    int removePin(int idx); // odregistrovanie viditeľného pinu  
  
    int setFilterInfo(TFilterInfo filterInfo); // nastavenie informácií o triede  
    void callProgress(float pos); // zavolanie callback funkcie priebehu  
public:  
    TFilter(); // základný konštruktor  
    TFilter(TGraph *parent); // konštruktor s parametrom rodičovského grafu  
    ~TFilter(); // deštruktor (uvolní všetky zaregistrované piny)  
  
    int getPinCount(); // vráti počet zaregistrovaných pinov  
    TPin* getPin(int idx); // vráti i-ty pin  
    int getPinIndex(TPin *pin); // vráti index zaregistrovaného pinu  
  
    int getState(); // vráti stav filtra  
    TFilterInfo getFilterInfo(); // vráti informácie o triede  
    ProgressCallback getProgressCallback(); // vráti ukazovateľ na callback funkciu
```

```

int setParent(TGraph* graph);           // nastaví rodičovský graf
int setProgressCallback(ProgressCallback progressFunc); // nastaví callback funkciu

virtual int setConfigData(TBuffer config,int type) = 0; // nastavenie konfigurácie
virtual TBuffer getConfigData(int type) = 0;           // vráti konfiguráciu
virtual int initialize() = 0;                          // inicializácia
virtual int run() = 0;                                 // beh hlavného algoritmu
virtual int reset() = 0;                              // reset stavu
virtual int stop() = 0;                               // stop behu
virtual int finalize() = 0;                           // finalizácia zdrojov
virtual int showConfigDialog() = 0;                  // konfiguračný dialóg
};

```

Všetky funkcie s návratovým typom *int* okrem „getrov“ vracajú návratové a chybové kódy definované neskôr.

Odvozená trieda musí definovať virtuálne metódy predpísané v triede *TFilter*. Sú to: inicializácia (*initialize*), ukončenie (*finalize*), spustenie hlavného algoritmu (*run*), ukončenie počas behu (*stop*), reset do základného stavu (*reset*), nastavenie konfigurácie (*setConfigData*), vrátenie konfigurácie (*getConfigData*) a konfiguračný dialóg (*showConfigDialog*). Na monitorovanie práce filtra je použitý funkcionálny typ, ktorý slúži ako *callback* funkcia,

```
typedef void (*ProgressCallback)(float); ,
```

ktorý má vstup jedno reálne číslo v rozsahu $<0.0, 1.0>$, kde *0.0* znamená začiatok práce a *1.0* znamená koniec práce, hodnoty medzi *0.0* a *1.0* znamenajú pomer hotovej práce k celkovej potrebnej práci (odhady). Filter počas svojho životného cyklu prechádza rôznymi stavmi, ktoré hovoria o jeho aktuálnom stave a možnostiach.

Filter môže byť v týchto stavoch (*Filter State*):

<i>FS_NONE</i>	0	// nedefinovaný stav
<i>FS_READY</i>	1	// pripravený stav
<i>FS_RUNNING</i>	2	// stav behu
<i>FS_STOPPED</i>	3	// stav po stopnutí
<i>FS_DONE</i>	4	// stav po dokončení práce
<i>FS_ERROR</i>	5	// chybový stav.

Prechod medzi týmito stavmi musí zabezpečiť každá trieda osobitne podľa vnútorných potrieb. Na začiatku po konštrukcii má filter stav *FS_NONE* (ak konštrukcia prebehla v poriadku) alebo *FS_ERROR* (ak pri konštruovaní nastali chyby, ktoré bránia filtru v korektnej funkčnosti, ak je chyba trvalá mal by sa tento stav zachovávať aj pri pokuse o volanie iných funkcií filtra). Teraz by mal byť filter inicializovaný volaním *initialize* a prejsť do stavu *FS_READY*. Funkcia *initialize* poskytuje priestor pre inicializáciu, ktorá nemohla byť urobená v čase konštrukcie. Vracia návratový kód *RC_OK*, ak prebehla v poriadku, inak vráti chybový kód. V stave *FS_READY* je filter pripravený na beh. Volaním funkcie *run* sa spustí hlavný algoritmus. Filter by si mal skontrolovať všetko potrebné na prácu, načítať dáta s pinov a začať prácu. Ak môže bežať, prejde do stavu *FS_RUNNING* a začne výpočet. Ak nemôže pracovať kvôli chýbajúcim dátam na pinoch vráti chybový návratový kód *RC_PIN_DATANOTREADY*, ak nemôže pracovať kvôli niečomu inému (nepripojené piny, chybné dáta alebo vnútorné chyby) vráti iný chybový kód (napr. všeobecný kód *RC_ERROR*),

inak v prípade úspešného ukončenia práce prejde do stavu *FS_DONE* a vráti návratový kód *RC_OK*. Filter môže počas behu hlásiť kompletnosť práce volaním funkcie *callProgress*. V prípade potreby ukončiť prácu počas behu, zavoláme funkciu *stop*, na ktorú by mal filter zareagovať zastavením behu, prejdením do stavu *FS_STOPPED* a vrátiť návratový kód *RC_STOPPED*. Po stopnutí vrátime filter do stavu *FS_READY* volaním funkcie *reset*. Pred deštruovaním filtra zavoláme funkciu *finalize*, ktorá zabezpečí uvoľnenie prostriedkov, ktoré nemôžu byť uvoľnené v deštruktore, a filter prejde do stavu *FS_NONE*. Nakoniec, keď už filter nepotrebujeme, môžeme ho deštruovať.

Filter je možné konfigurovať volaním konfiguračných funkcií prípadne cez konfiguračné piny, ak takéto filter poskytuje. Funkcia *setConfigData* vloží konfiguráciu do filtra. Konfigurácia je posielaná v štruktúre *TBuffer*, v ktorej je buď zabalená konfiguračná štruktúra špecifická pre každú triedu, alebo serializovaná konfigurácia (typ sa zadáva ako parameter do funkcie). Rozdiel medzi týmito dvoma spôsobmi konfigurovania je v tom, že štruktúra môže obsahovať aj ukazovatele do pamäti, ktoré nie je vhodné napr. ukladať na disk. Serializovaná konfigurácia obsahuje všetky potrebné konfiguračné dáta v zhustenej podobe v jednom pamäťovom buffri (a dá sa ľahko uložiť aj na disk). Takisto je možné získať konfiguráciu volaním funkcie *getConfigData*. Tu si tiež môžeme vyžiadať buď štruktúru alebo serializovanú konfiguráciu. Obidva typy sú vrátené v štruktúre *TBuffer*, kde typ dát je nastavený na *DT_CONFIG*. Predávania konfigurácie cez piny funguje rovnako, s tým rozdielom, že konfiguráciu si musí filter vyžiadať sám z konfiguračného pinu, a mal by ju dostať v serializovanej podobe (resp. takisto poskytnúť v serializovanej podobe). Pre interakciu s užívateľom je doplnená aj funkcia *showConfigDialog*, ktorá by mala vyvolať konfiguračný dialóg s možnosťou nastavenia parametrov filtra.

Typy konfigurácií (*Config Data Type*):

```
CDT_STRUCTURE 0    // konfigurácia v štruktúre
CDT_SERIALIZED 1   // serializovaná konfigurácia
```

Pridávanie a registrácia pinov je možná cez funkciu *addFilter*. Filter môže mať ľubovoľne veľa pinov, ak chceme aby boli piny viditeľné pre ostatných, musia byť zaregistrované. Odregistrovanie pinov je možné cez funkciu *removePin*. Všetky zaregistrované piny sú uvoľnené pri deštrukcii filtra.

Životný cyklus filtra:

```
Create->Initialize->[SetConfigData]->Run->Reset->[SetConfigData]->Run->Reset-> ... ->
Finalize->Destroy.
```

Header: filtergraph.h

Lib: filtergraph.lib

4. Trieda TGraph

Trieda *TGraph* poskytuje funkcie na kontrolovanie a monitorovanie behu grafu.

Trieda *TGraph* je definovaná nasledovne:

```
class TGraph {
private:
```

```

TFilter**  Filters;      // pole zaregistrovaných filtrov
int        cFilters;    // počet zaregistrovaných filtrov

char*      Name;        // meno grafu
int        State;       // stav grafu
bool       StopFlag;    // indikátor zastavenie
int        Progress;    // počet úspešne vykonaných filtrov
ProgressCallback ProgressFunc; // callback funkcia priebehu pre graf
int        RunningFilter; // index aktuálne bežiacého filtra

void callProgress(float pos); // volanie callback funkcie grafu
public:
    TGraph();                // základný konštruktor
    ~TGraph();               // deštruktor

    int initialize();        // inicializácia všetkých zaregistrovaných filtrov
    int run();               // spustenie grafu
    int reset();             // reset všetkých zaregistrovaných filtrov
    int stop();              // stop behu grafu
    int finalize();          // finalizácia všetkých zaregistrovaných filtrov

    int getFilterCount();    // vráti počet zaregistrovaných filtrov
    TFilter* getFilter(int idx); // vráti i-ty filter
    int getFilterIndex(TFilter* filter); // vráti index objektu filter
    int getRunningFilter();  // vráti index aktuálne bežiacého filtra

    int addFilter(TFilter* filter); // zaregistruje filter
    int removeFilter(int idx);      // odregistruje filter

    int connectPins(TPin* src, TPin* dest); // pripojí pin dest na pin src (spojenie src->dest)
    int disconnectPin(TPin* pin);          // odpojí pin od nastaveného cieľa

    char* getName();           // vráti meno filtra
    int getProgress();         // vráti počet úspešne dokončených filtrov
    int getState();            // vráti stav grafu

    int setName(char* name);   // nastaví meno grafu
    int setFilterProgressFunc(ProgressCallback pfnCallback); // nastaví callback pre filtre
    int setGraphProgressFunc(ProgressCallback pfnCallback); // nastaví callback pre graf
};

```

Všetky funkcie, vracajúce typ *int* okrem „getrov“, vracajú návratové a chybové kódy definované neskôr.

Graf môže byť v týchto stavoch (*Graph State*):

```

GS_NONE      0    // nedefinovaný stav
GS_READY     1    // stav pripravený na beh
GS_RUNNING   2    // stav behu
GS_STOPPED   3    // stav po stopnutí
GS_DONE      4    // stav úspešného vykonania grafu

```


Filtre musia byť najprv zaregistrované v grafe aby ich bolo možné neskôr spúšťať. Registrácia filtra sa realizuje volaním funkcie *addFilter*, ktorá pridá filter do vnútorného zoznamu. Takisto je možné filter odregistrovať volaním funkcie *remove Filter* (pred tým je potrebné zistiť index filtra volaním funkcie *getFilterIndex*). Graf poskytuje aj funkcie spájania pinov, *connectPins* a *disconnectPin*, ktoré je možné obísť priamim volaním funkcií pinov.

Graf prechádza týmito stavmi. Po konštrukcii má stav *GS_NONE*, ak konštrukcia prebehla v poriadku, alebo *GS_ERROR*, ak sa vyskytli neopraviteľné chyby pri konštrukcii. Po konštrukcii sa do grafu môžu registrovať filtre a spojiť vybrané piny. Po zaregistrovaní všetkých filtrov môžeme graf inicializovať volaním funkcie *initialize*, ktorá inicializuje všetky ešte neinicializované filtre. Funkcia *initialize* vráti *RC_OK* a graf prejde do stavu *GS_READY*, ak sa všetky filtre podarilo inicializovať, inak vráti *RC_ERROR* a graf prejde do stavu *GS_ERROR*. V stave *GS_READY* môžeme graf spustiť funkciou *run*. Ak graf môže bežať, prejde do stavu *GS_RUNNING*, inak prejde do stavu *GS_ERROR* a vráti *RC_ERROR*. Odporúčame vytvoriť si osobitné vlákno a v ňom spustiť graf, kvôli lepšiemu manipulovaniu v prípade nepredvídaných okolností (možnosť zabiť vlákno). Počas behu, graf volá callback funkciu nastavenú cez volanie *setGraphProgressFunc* (má rovnakú formu ako callback pre filtre) po každom úspešne vykonanom filtri. V prípade, že graf dostane návratový kód *RC_ERROR* od filtra, celý výpočet sa zastaví a graf prejde do stavu *GS_ERROR*. Ak dostane iný chybový kód, pokúsi sa vykonať filter inokedy. Ak sa podarí vykonať všetky filtre úspešne, graf prejde do stavu *GS_DONE* a vráti návratový kód *RC_OK*. Počas behu je tiež možné zavolať funkciu *stop*, ktorá zapne indikátor zastavenia, počká na zastavenie aktuálne bežiacieho filtra a ďalej nepokračuje vo výpočte, graf prejde do stavu *GS_STOPPED* a vráti kód *RC_STOPPED*. Graf sa vráti do stavu *GS_READY* volaním funkcie *reset*. Pred deštrukciou grafu je možné zavolať funkciu *finalize*, ktorá finalizuje všetky zaregistrované filtre. Pri deštrukcii grafu nie sú deštruované žiadne filtre.

Životný cyklus grafu:

Create->AddFilters->Initialize->[SetConfigData on Filters]->Run->Reset->[SetConfigData on Filters]->Run->Reset-> ... -> Finalize->Destroy.

Header: filtergraph.h

Lib: filtergraph.lib

5. Knižnica filtrov

Ak je potreba, tak triedy filtrov balíme do dynamických knižníc (DLLs) kvôli prenositeľnosti a možnosti širšieho využitia v iných programoch. Jedna knižnica môže obsahovať implementáciu viacerých rôznych tried filtrov s rôznymi identifikačnými číslami. Knižnica poskytuje informácie o sebe v štruktúre *TFilterLibraryInfo* definovanej takto:

```
typedef struct {
    char*      LibraryName;      // meno knižnice
    UINT       MajorVersion;     // významnejšia časť verzie
    UINT       MinorVersion;     // menej významná časť verzie
    int        FilterCount;      // počet tried filtrov v knižnici
} TFilterLibraryInfo;
```

Pre používanie knižnice nám stačí zadať funkčné prototypy v programe:

```
typedef TFilterLibraryInfo (*GetFilterLibraryInfoFunc)(void);    // informácie o knižnici
typedef TFilterInfo (*GetFilterInfoFunc)(int);                  // informácie o triede filtrov
typedef TFilter* (*CreateFilterFunc)(int);                      // vytváranie filtrov
typedef void (*DestroyFilterFunc)(TFilter*,int);                // uvoľňovanie filtrov
```

Header: filterdlltypes.h

Knižnicu načítame systémovou funkciou *LoadLibrary*, funkcie získame cez systémovú funkciu *GetProcAddress*, a nakoniec knižnicu uvoľníme systémovou funkciou *FreeLibrary*.

Knižnica tried filtrov musí exportovať tieto funkcie s danými názvami:

```
TFilterLibraryInfo GetFilterLibraryInfo(void);
- vráti informácie o knižnici
TFilterInfo GetFilterInfo(int idx);
- vráti informácie o konkrétnej triede filtrov s indexom i
TFilter* CreateFilter(int idx);
- vytvorí objekt filtra konkrétnej triedy filtrov s indexom i a vráti ukazovateľ naň
void DestroyFilter(TFilter *filter,int idx);
- uvoľní konkrétny objekt filtra danej triedy filtrov s indexom i
```

Usporiadanie a indexy tried filtrov závisia od konkrétnej knižnice.

Header: filterdll.h

6. Základné dátové typy

Pre podporu spracovania obrazovej informácie sme zadefinovali základné dátové typy použité pri výmene informácií medzi filtermi. Konštanty dátových typov sa používajú na identifikáciu dát v štruktúre *TBuffer* (v člene *type*). Používateľ si môže dedefinovať vlastnú sadu dátových typov, ktoré budú vhodnejšie na spracovanie konkrétnej problematiky.

Základné dátové elementy (*Data Element Type*):

<i>DET_BYTE</i>	<i>BYTE</i>	// dátový element bajt
<i>DET_FLOAT</i>	<i>double</i>	// dátový element reálne číslo
<i>DET_STRING</i>	<i>char*</i>	// dátový element reťazec

Používanie dátových elementov nie je záväzné, sú definované len kvôli kontrole veľkosti a presnosti.

Základné konštanty dátových typov (*Data Type*):

<i>DT_UNKNOWN</i>	0	// neznámy, všeobecný typ (<i>void*</i>)
<i>DT_CONFIG</i>	1	// konfigurácia (štruktúra závisí od triedy filtrov)
<i>DT_PLANE</i>	2	// obrazová rovina, štruktúra <i>TPlane</i>
<i>DT_IMAGE3</i>	3	// obrazová rovina trojíc, štruktúra <i>TImage3</i>
<i>DT_IMAGE4</i>	4	// obrazová rovina štvoríc, štruktúra <i>TImage4</i>
<i>DT_STRING</i>	5	// reťazec, typ <i>DET_STRING</i>
<i>DT_INT</i>	6	// celočíselný typ, typ <i>int</i>
<i>DT_FLOAT</i>	7	// desatinný typ, typ <i>DET_FLOAT</i>

```
DT_INTARRAY      8      // pole celých čísel, štruktúra TIntArray
DT_FLOATARRAY    9      // pole reálnych čísel, štruktúra TFloatArray
```

Použité štruktúry ku konštantám dátových typov:

Obrazová rovina:

```
typedef struct {
    int          cx;      // rozlíšenie v x-ovom smere
    int          cy;      // rozlíšenie v y-ovom smere
    DET_FLOAT    min;     // minimum dynamického rozsahu (väčšinou 0.0)
    DET_FLOAT    max;     // maximum dynamického rozsahu (väčšinou 1.0 alebo 255.0)
    int          size;    // počet bajtov alokovaných pre dáta
    void*        data;    // ukazovateľ na dáta (1 pixel je reálne číslo typu DET_FLOAT)
} TPlane;
```

Obrazová rovina trojíc:

```
typedef struct {
    int          cx;      // rozlíšenie v x-ovom smere
    int          cy;      // rozlíšenie v y-ovom smere
    DET_FLOAT    min;     // minimum dynamického rozsahu (väčšinou 0.0)
    DET_FLOAT    max;     // maximum dynamického rozsahu (väčšinou 1.0 alebo 255.0)
    int          size;    // počet bajtov alokovaných pre dáta
    void*        data;    // ukazovateľ na dáta (1 pixel sú 3 reálne čísla DET_FLOAT)
} TImage3;
```

Obrazová rovina štvoríc:

```
typedef struct {
    int          cx;      // rozlíšenie v x-ovom smere
    int          cy;      // rozlíšenie v y-ovom smere
    DET_FLOAT    min;     // minimum dynamického rozsahu (väčšinou 0.0)
    DET_FLOAT    max;     // maximum dynamického rozsahu (väčšinou 1.0 alebo 255.0)
    int          size;    // počet bajtov alokovaných pre dáta
    void*        data;    // ukazovateľ na dáta (1 pixel sú 4 reálne čísla DET_FLOAT)
} TImage4;
```

Pole celých čísel:

```
typedef struct {
    int          count;    // počet prvkov poľa
    int*         data;     // ukazovateľ na prvý prvok poľa (ostatné nasledujú za ním)
} TIntArray;
```

Pole reálnych čísel:

```
typedef struct {
    int          count;    // počet prvkov poľa
    DET_FLOAT*   data;     // ukazovateľ na prvý prvok poľa
} TFloatArray;
```

Header: datatypes.h

7. Návrátové a chybové kódy

Architektúra Filtergraph definuje základné návratové a chybové kódy, použité ako návratové hodnoty z funkcií, pre ktoré je vhodné vedieť stav ukončenia operácie. Ako typ návratových kódov sme zvolili typ *int*.

Základné návratové kódy (*Return Code*):

<i>RC_OK</i>	0	// bezchybné ukončenie operácie
<i>RC_UNKNOWNERROR</i>	1	// neznáma alebo nedefinovaná chyba
<i>RC_NOTENOUGHMEMORY</i>	2	// nedostatok pamäte
<i>RC_OUTOFBOUNDS</i>	3	// mimo hraníc (napr. parameter)
<i>RC_STOPPED</i>	4	// operácia ukončená pred dokončením
<i>RC_ERROR</i>	5	// vážna chyba, bližšie nepopísaná
<i>RC_UNKNOWNCONFIGTYPE</i>	6	// neznámy typ konfigurácie

Základné návratové kódy pre piny (*Return Code for Pin*):

<i>RC_PIN_DATANOTREADY</i>	20	// dáta na pine nie sú pripravené
<i>RC_PIN_NOTCONNECTED</i>	21	// pin nie je pripojený
<i>RC_PIN_CONNECTINGTONULL</i>	22	// pokus o pripojenie pinu na neplatný ukazovateľ

Header: filtergraph.h

8. Iné použité triedy

8.1 Trieda TSerializer

Trieda TSerializer sa používa na serializovanie dát do jedného pamäťového buffra. Dokáže zapisovať dáta v zhustenej podobe a taktiež ich dokáže aj čítať.

Trieda TSerializer je definovaná nasledovne:

```
class TSerializer {
private:
    void*      Data;          // pamäťový buffer obsahujúci dáta
    int        Size;          // veľkosť pamäťového bufferu
    void*      ReadPtr;       // čítací ukazovateľ
    int        ReadPos;       // čítacia pozícia od začiatku bufferu
    void*      WritePtr;      // zapisovací ukazovateľ
    int        WritePos;      // zapisovacia pozícia od začiatku bufferu
public:
    TSerializer();           // základný konštruktor
    ~TSerializer();         // deštruktor

    int reset();            // reset stavu, uvoľnenie pamäti a vynulovanie pozícií

    int setData(TBuffer buffer); // nastaví dáta, na ktorých bude pracovať
    TBuffer getData();          // vráti kópiu serializovaných dát

    int serialize(void *lp, int sz); // serializuje sz bajtov z adresy lp do buffra
```

```

int serializeStr(char* str);           // serializuje reťazec z adresy str, uloží si aj jeho dĺžku

int deserialize(void *dest, int sz); // deserializuje sz bajtov z buffra na adresu dest
int deserializeStr(char **dest);     // deserializuje reťazec z buffra na adresu dest

int getReadPos();                     // vráti čítaciu pozíciu
int getWritePos();                   // vráti zapisovaciu pozíciu

int setReadPos(int newpos);          // nastaví čítaciu pozíciu
int setWritePos(int newpos);         // nastaví zapisovaciu pozíciu
};

```

Základné operácie s triedou TSerializer:

Serializovanie:

```

TSerializer *ser = new TSerializer; // nový serializér
char* str = „String“;               // reťazec
int i = 200;                         // hodnota
int a[50];                          // pole hodnôt
TBuffer buffer;                     // buffer na výsledné dáta

ser->serialize(&i,sizeof(int));       // serializujeme celočíselnú hodnotu
ser->serializeStr(str);               // serializujeme reťazec
ser->serialize(&a,50*sizeof(int));    // serializujeme pole hodnôt ako dátový buffer
buffer = ser->getData();              // vyberieme serializované dáta zo serializéru
delete ser;                          // uvoľníme alokované zdroje

```

Deserializovanie:

```

TSerializer *ser = new TSerializer; // nový serializér
char *str;                          // ukazovateľ na reťazec
int i;                              // hodnota
int a[50];                          // pole hodnôt

ser->setData(buffer);                 // nastavíme dáta, na ktorých budeme pracovať
ser->deserialize(&i,sizeof(int));     // deserializujeme celočíselnú hodnotu
ser->deserializeStr(&str);            // deserializujeme reťazec
ser->deserialize(&a,50*sizeof(int));  // deserializujeme pole hodnôt ako dátový buffer

delete ser;                          // uvoľníme zdroje
FreeBuffer(buffer);                  // uvoľníme buffer

```

Header: filtergraph.h

Lib: filtergraph.lib

8.2 Trieda CBitStream

Trieda CBitStream slúži na zhustený zápis dát po bitoch. Dokáže čítať dáta z pamäťového buffra po bitoch a takisto ich dokáže zapisovať po jednotlivých bitoch, alebo bitových sekvenciách.

Trieda CBitStream je definovaná nasledovne:

```
#define BSITEM BYTE      // najmenšia jednotka, na ktorej trieda pracuje
class CBitStream {
private:
    int      iSize;      // veľkosť jednotky v bajtoch
    int      ibSize;     // veľkosť jednotky v bitoch
    BSITEM   *mask;      // bitové masky, na oddelovanie bitov

    int      Size;       // počet alokovaných jednotiek
    int      Count;      // počet použitých jednotiek
    int      Align;      // alokačné zarovnanie
    BSITEM   *data;      // ukazovateľ na dátový buffer
    BOOL     Attached;    // indikátor, či sú dáta pripojené alebo nie
    BOOL     FreeOnDelete; // indikátor, či sa dáta majú uvoľniť pri deštrukcii

    int      rpos;       // čítacia pozícia od začiatku buffra v jednotkách
    int      rb;         // čítacia pozícia v bitoch od začiatku aktuálnej čítacej jednotky
    int      wpos;       // zapisovacia pozícia od začiatku buffra v jednotkách
    int      wb;         // zapisovacia pozícia v bitoch od začiatku aktuálnej zapisovacej jednotky

    int      resize();   // alokuje viac pamäte pre dáta, zarovnáva na násobok Align
public:
    CBitStream();        // konštruktor
    CBitStream(int Alignment, BOOL bFreeOnDelete); // konštruktor s parametrami
    ~CBitStream();       // deštruktor

    int setData(void *ndata, int size); // nastaví dáta, na ktorých sa bude pracovať
    void* getData(int *size);           // vráti dáta z bufferu

    int read(void *dest, int bitcount); // načíta bitcount bitov na ukazovateľ dest
    int nmread(void *dest, int bitcount); // čítanie bez pohybu čítacích pozícií

    int write(void *src, int bitcount); // zapíše bitcount bitov na z ukazovateľa src
    int nmwrite(void *src, int bitcount); // zápis bez pohybu zapisovacích pozícií

    int setRPos(int bitcount); // nastaví čítaciu pozíciu v bitoch od začiatku bufferu
    int getRPos();             // vráti čítaciu pozíciu v bitoch od začiatku bufferu
    int setWPos(int bitcount); // nastaví zapisovaciu pozíciu v bitoch od začiatku buffru
    int getWPos();             // vráti zapisovaciu pozíciu v bitoch od začiatku buffru
};
```

Základné operácie s triedou CBitStream:

Serializovanie:

```
CBitStream *bs = new CBitStream(32,FALSE); // nový CBitStream, zarovnanie na 32
int i = 200; // hodnota
TBuffer buffer; // dátový buffer

bs->write(&i,4); // serializujeme 4 bity z čísla i
bs->write(&i,5); // serializujeme 5 bitov z čísla i
buffer.type = DT_UNKNOWN; // nastavíme typ dát v buffri na neznámy
```

```
buffer.data = bs->getData(&buffer.size); // preberieme zhustené dáta z bitstreamu
delete bs; // uvoľníme zdroje
```

Deserializovanie:

```
CBitStream *bs = new CBitStream(); // nový CBitStream
int i1,i2; // hodnoty

bs->setData(buffer.data,buffer.size); // nastavíme dáta, nad ktorými budeme pracovať
bs->read(&i1,4); // deserializujeme 4 bity do premennej i1
bs->read(&i2,5); // deserializujeme 5 bitov do premennej i2
delete bs; // uvoľníme zdroje
FreeBuffer(buffer); // uvoľníme buffer
```

Header: bitstream.h

Lib: bitstream.lib

9. Príklad odvodenej triedy: Trieda TCropFilter

Uvedieme príklad filtra, ktorý realizuje orezanie obrazu.

Projekt sa skladá z nasledovných súborov:

<i>cropresource.h</i>	// súbor definujúci externé zdroje (dialóg, ikony, ...)
<i>cropcfg.h</i>	// popis konfiguračnej štruktúry
<i>crop.h</i>	// popis triedy filtra
<i>crop.cpp</i>	// implementácia filtra
<i>cropdll.h</i>	// popis knižnice filtra
<i>cropdll.cpp</i>	// implementácia knižnice filtra

cropresource.h (definícia zdrojov):

Tento súbor je potrebné vytvoriť len, keď chceme do projektu zaradiť aj externé zdroje (*resources*) ako formulár dialógu, ikony, obrázky a iné. Tento súbor by sa mal automaticky vytvoriť s vložením zdrojového skriptu (*resource script*) do projektu v prostredí Microsoft Visual C++ 6.0, a je automaticky generovaný a aktualizovaný týmto prostredím.

```
#define IDD_DIALOG1 101
#define IDC_TLABSRADIO 1000
#define IDC_TLRELRADIO 1001
#define IDC_X1EDIT 1002
#define IDC_Y1EDIT 1003
#define IDC_X2EDIT 1004
#define IDC_Y2EDIT 1005
#define IDC_BRABSBORRADIO 1006
#define IDC_BRRELBORRADIO 1007
```

cropcfg.h (definícia konfiguračnej štruktúry):

Ak chceme triedu filtra používať len v medziach triedy TFilter a nechceme volať neštandardné (pridané) metódy odvodenej triedy, stačí nám použiť len tento súbor spolu s dll knižnicou (namiesto použitia definície celej odvodenej triedy zo súboru *crop.h*).

```
typedef struct {
    int TLMode; // mód ľavého, horného rohu orezávacieho obdĺžnika
```

```

    int      BRMode;      // mód pravého, dolného rohu orezávacieho obdĺžnika
    double   Posx1;       // ľavá súradnica orezávacieho obdĺžnika
    double   Posy1;       // horná súradnica orezávacieho obdĺžnika
    double   Posx2;       // pravá súradnica orezávacieho obdĺžnika
    double   Posy2;       // dolná súradnica orezávacieho obdĺžnika
} TCropConfig;

// Position modes (módy pozícií rohov)
#define PM_ABSTOBORDER 0 // hodnoty Posx a Posy znamenajú pozíciu
v pixeloch od okraja
#define PM_RELTOBORDER 1 // hodnoty Posx a Posy znamenajú relatívnu
pozíciu od okraja (presná pozícia sa vypočíta z rozmerov vstupného obrazu)

crop.h (definícia triedy filtra):
#include "filtergraph.h" // definícia základnej triedy TFilter a triedy TPin
#include "datatypes.h"   // základné dátové typy
#include "cropcfg.h"     // konfiguračná štruktúra

TFilterInfo getCropFilterInfo(); // globálna funkcia vracajúca informácie o triede

class TCropFilter : public TFilter { //definícia triedy
private:
    HINSTANCE hInstance; // systémový identifikátor inštancie knižnice (kvôli zdrojom)

    int  TLMode;          // interne uložená konfigurácia
    int  BRMode;          // ...
    double Posx1;         // ...
    double Posy1;         // ...
    double Posx2;         // ...
    double Posy2;         // ...

    TPlane* InPlane;      // ukazovateľ na vstupnú rovinu
    TPlane OutPlane;      // výstupná rovina

    TPin* CfgInPin;       // vstupný konfiguračný pin
    TPin* CfgOutPin;      // výstupný konfiguračný pin
    TPin* InPin;          // vstupný pin
    TPin* OutPin;         // výstupný pin

    bool StopFlag;        // indikátor zastavenia

    int setConfigFromPin(TPin *pin); // nastavenie konfigurácie z pinu
    int putConfigOnPin(TPin *pin);   // sprístupnenie konfigurácie na pine
    int freeConfigOnPin(TPin *pin);  // uvoľnenie konfigurácie na pine
    int clearOutputPins();           // vyčistenie výstupných pinov
    int freeOutputData();            // uvoľnenie výstupných dát
public:
    TCropFilter();                  // základný konštruktor
    TCropFilter(HINSTANCE hInst);  // konštruktor s identifikátorom knižnice
    ~TCropFilter();                // deštruktor

```



```

int setConfigData(TBuffer config,int type);      // nastaví konfiguráciu
TBuffer getConfigData(int type);                // vráti konfiguráciu
int initialize();      // inicializácia objektu
int run();             // beh algoritmu
int reset();          // reset stavu
int stop();           // zastavenie behu algoritmu
int finalize();       // finalizácia objektu
int showConfigDialog(); // vyvolanie konfiguračného dialógu
};

```

crop.cpp (implementácia triedy filtra):

```

#include <windows.h>      // definície systému
#include <stdio.h>        // definície štandardných funkcií
#include "crop.h"         // definícia triedy
#include "cropresource.h" // definícia zdrojov

#define CROPFILTER_NAME "Crop"           // meno triedy
#define CROPFILTER_MAJORVERSION 1       // hlavné číslo verzie
#define CROPFILTER_MINORVERSION 0       // vedľajšie číslo verzie
#define CROPFILTER_MAGIC 110            // identifikačné číslo triedy

TFilterInfo CropFilterInfo = {           // vyplnenie informačnej štruktúry
    CROPFILTER_NAME,
    CROPFILTER_MAJORVERSION,
    CROPFILTER_MINORVERSION,
    CROPFILTER_MAGIC};

TFilterInfo getCropFilterInfo() {         // funkcia vracajúca informácie o triede
    return CropFilterInfo;
}

TCropConfig *pCfg = NULL;                // globálna konfigurácia (pre použitie dialógu)

TCropFilter::TCropFilter() {              // základný konštruktor
    hInstance = NULL;                     // inicializácia premenných

    TLMode = PM_RELTOBORDER;
    BRMode = PM_RELTOBORDER;
    Posx1 = 0.0;
    Posy1 = 0.0;
    Posx2 = 1.0;
    Posy2 = 1.0;

    InPlane = NULL;

    OutPlane.cx = 0;
    OutPlane.cy = 0;
    OutPlane.min = 0.0;
    OutPlane.max = 0.0;
}

```

```

    OutPlane.size = 0;
    OutPlane.data = NULL;

    StopFlag = false;
    FilterInfo = CropFilterInfo;

    CfgInPin = new TPin("Cfg in", PT_CONFIGIN);          // vytvorenie pinov
    CfgOutPin = new TPin("Cfg out", PT_CONFIGOUT);
    InPin = new TPin("In", PT_INPUT);
    OutPin = new TPin("Out", PT_OUTPUT);

    addPin(CfgInPin);          // zaregistrovanie pinov
    addPin(CfgOutPin);
    addPin(InPin);
    addPin(OutPin);

    if ((CfgInPin == NULL)|| (CfgOutPin == NULL)|| (InPin == NULL)|| (OutPin == NULL))
    {
        State = FS_ERROR;    // kontrola pinov
    }
}

TCropFilter::TCropFilter(HINSTANCE hInst) {    // konštruktor so systémovou identifikáciou
    hInstance = hInst;                        // inicializácia premenných

    TLMode = PM_RELTOBORDER;
    BRMode = PM_RELTOBORDER;
    Posx1 = 0.0;
    Posy1 = 0.0;
    Posx2 = 1.0;
    Posy2 = 1.0;

    InPlane = NULL;

    OutPlane.cx = 0;
    OutPlane.cy = 0;
    OutPlane.min = 0.0;
    OutPlane.max = 0.0;
    OutPlane.size = 0;
    OutPlane.data = NULL;

    StopFlag = false;
    FilterInfo = CropFilterInfo;

    CfgInPin = new TPin("Cfg in", PT_CONFIGIN);          // vytvorenie pinov
    CfgOutPin = new TPin("Cfg out", PT_CONFIGOUT);
    InPin = new TPin("In", PT_INPUT);
    OutPin = new TPin("Out", PT_OUTPUT);

    addPin(CfgInPin);          // zaregistrovanie pinov

```

```

    addPin(CfgOutPin);
    addPin(InPin);
    addPin(OutPin);

    if ((CfgInPin == NULL)|| (CfgOutPin == NULL)|| (InPin == NULL)|| (OutPin == NULL))
    {
        State = FS_ERROR;    // kontrola pinov
    }
}

TCropFilter::~TCropFilter() {    // deštruktor
    freeOutputData();           // uvoľnenie výstupných dát
    freeConfigOnPin(CfgOutPin); // uvoľnenie konfigurácie na pine

    CfgInPin = NULL;           // zaregistrované piny sú automaticky uvoľnené triedou TFilter
    CfgOutPin = NULL;
    InPin = NULL;
    OutPin = NULL;
}

int TCropFilter::initialize() {    // inicializácia (tu nie je potrebné robiť nič špeciálne)
    State = FS_READY;
    return RC_OK;
}

int TCropFilter::finalize() {    // finalizácia (ani tu nie je potrebné robiť nič špeciálne)
    State = FS_NONE;
    return RC_OK;
}

int TCropFilter::run() {    // hlavný algoritmus
    int res,cnt;
    int tlx,tly,brx,bry;
    int x,y;
    int width,height;
    TBuffer buffer;

    if (StopFlag) {    // kontrola zastavenia
        State = FS_STOPPED;
        return RC_STOPPED;
    }
    else
        StopFlag = false;

    if (State == FS_ERROR) return RC_ERROR;    // kontrola chyby
    State = FS_RUNNING;                        // ináč môžeme bežať
    callProgress(0.0);                         // informujeme o priebehu (začiatok)

    if (OutPlane.data != NULL) {                // vyčistenie výstupov po predošlom behu
        free(OutPlane.data);
    }
}

```

```

    OutPlane.data = NULL;
}

res = setConfigFromPin(CfgInPin);           // pokus o načítanie konfigurácie z pinu
if (res == RC_PIN_DATANOTREADY) return res;

res = InPin->receiveData();                 // žiadosť o dáta zo zdrojového pinu

if (res != RC_OK) {                         // nepodarilo sa získať dáta
    if (res == RC_PIN_NOTCONNECTED) {      // pin nie je pripojený, koniec behu
        State = FS_STOPPED;
        return RC_ERROR;
    } else return res;                     // inak vrátime návratový kód
} else {                                    // podarilo sa získať dáta, beh pokračuje
    buffer = InPin->getBuffer();             // vyberieme buffer z pinu
    if (buffer.data == NULL) {              // skontrolujeme ho
        State = FS_STOPPED;
        return RC_ERROR;
    }
    InPlane = (TPlane*) buffer.data;        // vyberieme obrazovú rovinu z buffra
    if (InPlane->data == NULL) {             // skontrolujeme ju
        State = FS_STOPPED;
        return RC_ERROR;
    }
}
}

switch(TLMode) {                            // vstup je pripravený, vypočítame pozície
case PM_ABSTOBORDER:
    tlx = (int)Posx1;
    tly = (int)Posy1;
    break;
case PM_RELTOBORDER:
    tlx = (int)(InPlane->cx*Posx1);
    tly = (int)(InPlane->cy*Posy1);
    break;
}

switch(BRMode) {                            // vypočítame aj druhú pozíciu podľa nastavení
case PM_ABSTOBORDER:
    brx = (int)Posx2;
    bry = (int)Posy2;
    break;
case PM_RELTOBORDER:
    brx = (int)((InPlane->cx-1)*Posx2);
    bry = (int)((InPlane->cy-1)*Posy2);
    break;
}

if (tlx > brx) {                             // skontrolujeme pozície
    x = tlx;
    tlx = brx;
}

```

```

    brx = x;
}

if (tly > bry) {
    y = tly;
    tly = bry;
    bry = y;
}

width = brx - tlx + 1; // vypočítame šírku a výšku výstupu
height = bry - tly + 1;
if ((tlx < 0)|| (tlx >= InPlane->cx)||
    (tly < 0)|| (tly >= InPlane->cy)||
    (brx < 0)|| (brx >= InPlane->cx)||
    (bry < 0)|| (bry >= InPlane->cy)) { // skontrolujeme vypočítané pozície
    State = FS_STOPPED;
    return RC_ERROR;
}

cnt = sizeof(DET_FLOAT)*width*height; // vypočítame veľkosť pamäte pre výstup
OutPlane.cx = width; // inicializujeme výstupnú rovinu
OutPlane.cy = height;
OutPlane.min = InPlane->min;
OutPlane.max = InPlane->max;
OutPlane.size = cnt;
OutPlane.data = malloc(cnt); // alokujeme pamäť pre výstup
if (OutPlane.data == NULL) { // skontrolujeme alokáciu
    State = FS_STOPPED;
    return RC_ERROR;
}

// začneme orezávanie
DET_FLOAT *rptr, *wptr; // čítací a zapisovací ukazovateľ
wptr = (DET_FLOAT*) OutPlane.data; // zapisovací ukazovateľ na začiatok výstupu
// hlavný cyklus
for (y=0; y < height; y++) {
    if (StopFlag) { // kontrola indikátora zastavenie
        free(OutPlane.data); // ak treba zastaviť, treba uvoľniť nedokončené zdroje
        OutPlane.data = NULL;
        State = FS_STOPPED;
        return RC_STOPPED;
    }
    rptr = (DET_FLOAT*)InPlane->data + InPlane->cx*(y + tly) + tlx; // čítací ukazovateľ
    for(x=0; x < width; x++) {
        *wptr = *rptr;
        wptr++;
        rptr++;
    }
    callProgress((float)y/(float)height); // po každom spracovanom riadku sa ohlásime
}

```

```

    callProgress(1.0); // práca dokončená, ohlásime aj dokončenie práce

    ClearBuffer(&buffer); // pošleme výstupné dáta na výstupný pin
    buffer = MakeBuffer(DT_PLANE, sizeof(TPlane), &OutPlane);
    OutPin->setData(buffer);

    freeConfigOnPin(CfgOutPin); // aktualizujeme výstupný konfiguračný pin
    putConfigOnPin(CfgOutPin);

    State = FS_DONE; // zmeníme stav
    return RC_OK; // práca je dokončená, návrat
}

int TCropFilter::reset() { // obnovenie stavu
    clearOutputPins(); // vyčistíme výstupné piny
    freeOutputData(); // uvoľníme výstupné dáta
    State = FS_READY; // aktualizujeme stav
    StopFlag = false; // vynulujeme indikátor zastavenia
    return RC_OK; // vrátime úspešné obnovenie stavu filtra
}

int TCropFilter::stop() { // zastavenie práce
    StopFlag = true; // zapneme indikátor zastavenia
    State = FS_STOPPED; // zmeníme stav, pre istotu ☺
    return RC_OK; // návrat
}

BOOL CALLBACK CF_DialogProc(HWND hwndDlg, UINT uMsg, WPARAM wParam,
LPARAM lParam) { // Window procedúra dialogového okna (zobrazí a načíta konfiguráciu)
    char* str;
    switch(uMsg) {
    case WM_INITDIALOG:
        if (pCfg != NULL) {
            switch(pCfg->TLMode) {
            case PM_ABSTOBORDER:
                CheckRadioButton(hwndDlg, IDC_TLABSRADIO, IDC_TLRELRADIO,
IDC_TLABSRADIO);
                break;
            case PM_RELTOBORDER:
                CheckRadioButton(hwndDlg, IDC_TLABSRADIO, IDC_TLRELRADIO,
IDC_TLRELRADIO);
                break;
            }
            switch(pCfg->BRMode) {
            case PM_ABSTOBORDER:
                CheckRadioButton(hwndDlg, IDC_BRABSBORRADIO, IDC_BRRELBORRADIO,
IDC_BRABSBORRADIO);
                break;
            case PM_RELTOBORDER:

```

```

        CheckRadioButton(hwndDlg, IDC_BRABSBORRADIO, IDC_BRRELBORRADIO,
        IDC_BRRELBORRADIO);
        break;
    }
    str = (char*) malloc(256);
    if (str != NULL) {
        sprintf(str, "%.2f", pCfg->Posx1);
        SetDlgItemText(hwndDlg, IDC_X1EDIT, str);
        sprintf(str, "%.2f", pCfg->Posx2);
        SetDlgItemText(hwndDlg, IDC_X2EDIT, str);
        sprintf(str, "%.2f", pCfg->Posy1);
        SetDlgItemText(hwndDlg, IDC_Y1EDIT, str);
        sprintf(str, "%.2f", pCfg->Posy2);
        SetDlgItemText(hwndDlg, IDC_Y2EDIT, str);
        free(str);
    }
}
return true;
case WM_SHOWWINDOW:
    return true;
case WM_COMMAND:
    WORD wID = LOWORD(wParam);
    switch (wID) {
        case IDCANCEL:
            EndDialog(hwndDlg, IDCANCEL);
            return true;
        case IDOK:
            if (pCfg != NULL) {
                if (IsDlgButtonChecked(hwndDlg, IDC_TLABSRADIO)) pCfg->TLMode =
PM_ABSTOBORDER;
                if (IsDlgButtonChecked(hwndDlg, IDC_TLRELRADIO)) pCfg->TLMode =
PM_RELTOBORDER;
                if (IsDlgButtonChecked(hwndDlg, IDC_BRABSBORRADIO)) pCfg->BRMode =
PM_ABSTOBORDER;
                if (IsDlgButtonChecked(hwndDlg, IDC_BRRELBORRADIO)) pCfg->BRMode =
PM_RELTOBORDER;
                str = (char*) malloc(256);
                if (str != NULL) {
                    GetDlgItemText(hwndDlg, IDC_X1EDIT, str, 256);
                    pCfg->Posx1 = atof(str);
                    GetDlgItemText(hwndDlg, IDC_X2EDIT, str, 256);
                    pCfg->Posx2 = atof(str);
                    GetDlgItemText(hwndDlg, IDC_Y1EDIT, str, 256);
                    pCfg->Posy1 = atof(str);
                    GetDlgItemText(hwndDlg, IDC_Y2EDIT, str, 256);
                    pCfg->Posy2 = atof(str);
                }
            }
        }
    EndDialog(hwndDlg, IDOK);
    return true;

```

```

    }
    return true;
case WM_CLOSE:
    EndDialog(hwndDlg, IDCANCEL);
    return true;
}
return false;
}

int TCropFilter::showConfigDialog() { // vyvolanie konfiguračného dialógu
    if (hInstance == NULL) return RC_ERROR;
    int res;
    pCfg = (TCropConfig*) getConfigData(CDT_STRUCTURE).data; // konfigurácia
    res = DialogBox(hInstance, "CropConfig", NULL, CF_DialogProc); // vyvolanie dialógu
    if (res == IDOK) { // bolo stlačené tlačidlo OK
        if (pCfg != NULL) { // skontrolujeme konfiguráciu a zavoláme konfiguračnú funkciu
            setConfigData(MakeBuffer(DT_CONFIG, sizeof(TCropConfig), pCfg), CDT_STRUCTURE);
        }
    }
    free(pCfg); // uvoľníme konfiguráciu
    pCfg = NULL;
    return RC_OK;
}

int TCropFilter::setConfigData(TBuffer config, int type) { // nastavenie konfigurácie
    switch(type) { // podľa typu konfigurácie
    case CDT_STRUCTURE: // konfigurácia v štruktúre
        TCropConfig *pcfg;
        pcfg = (TCropConfig*) config.data;
        if (pcfg != NULL) { // skontrolujeme a načítame parametre
            TLMode = pcfg->TLMode;
            BRMode = pcfg->BRMode;
            Posx1 = pcfg->Posx1;
            Posx2 = pcfg->Posx2;
            Posy1 = pcfg->Posy1;
            Posy2 = pcfg->Posy2;
        }
        else return RC_ERROR;
        break;
    case CDT_SERIALIZED: // serializovaná konfigurácia
        if (config.data != NULL) { // skontrolujeme a deserializujeme parametre
            TSerializer *ser = new TSerializer();
            ser->setData(config);
            ser->deserialize(&TLMode, sizeof(int));
            ser->deserialize(&BRMode, sizeof(int));
            ser->deserialize(&Posx1, sizeof(double));
            ser->deserialize(&Posx2, sizeof(double));
            ser->deserialize(&Posy1, sizeof(double));
            ser->deserialize(&Posy2, sizeof(double));
        }
    }
}

```



```

        delete ser;
    }
    else return RC_ERROR;
    break;
}
return RC_OK;
}

TBuffer TCropFilter::getConfigData(int type) {    // vráti konfiguráciu
    TBuffer buffer;
    ClearBuffer(&buffer);                    // vyčistenie buffera
    switch(type) {                            // podľa typu
    case CDT_STRUCTURE:                       // zostavíme konfiguračnú štruktúru
        TCropConfig *pcfg;
        pcfg = (TCropConfig*) malloc(sizeof(TCropConfig));
        if (pcfg == NULL) return buffer;
        pcfg->TLMode = TLMode;
        pcfg->BRMode = BRMode;
        pcfg->Posx1 = Posx1;
        pcfg->Posx2 = Posx2;
        pcfg->Posy1 = Posy1;
        pcfg->Posy2 = Posy2;
        buffer = MakeBuffer(DT_CONFIG, sizeof(TCropConfig), pcfg);
        break;
    case CDT_SERIALIZED:                     // zostavíme serializovanú konfiguráciu
        TSerializer *ser = new TSerializer();
        if (ser == NULL) return buffer;
        ser->serialize(&TLMode, sizeof(int));
        ser->serialize(&BRMode, sizeof(int));
        ser->serialize(&Posx1, sizeof(double));
        ser->serialize(&Posx2, sizeof(double));
        ser->serialize(&Posy1, sizeof(double));
        ser->serialize(&Posy2, sizeof(double));
        buffer = ser->getData();
        buffer.type = DT_CONFIG;
        delete ser;
        break;
    }
    return buffer;                            // vrátime zostavený buffer
}

int TCropFilter::setConfigFromPin(TPin *pin) {    // nastaví konfiguráciu z pinu
    int res;
    res = pin->receiveData();                 // vyberie dáta z pinu
    if (res != RC_OK) return RC_ERROR;
    TBuffer buffer;
    buffer = pin->getBuffer();
    res = setConfigData(buffer, CDT_SERIALIZED);    // ak OK, tak konfiguruje
    if (res != RC_OK) return RC_ERROR;
    return RC_OK;
}

```

```

}

int TCropFilter::freeConfigOnPin(TPin* pin) {    // uvoľnenie konfigurácie na pine
    if (pin == NULL) return RC_OK;
    pin->freeData();
    return RC_OK;
}

int TCropFilter::putConfigOnPin(TPin *pin) {    // sprístupnenie konfigurácia na pine
    TBuffer buffer;
    buffer = getConfigData(CDT_SERIALIZED);
    pin->setData(buffer);
    return RC_OK;
}

int TCropFilter::clearOutputPins() {            // vyčistenie výstupných pinov
    freeConfigOnPin(CfgOutPin);
    if (OutPin != NULL) OutPin->clearData();
    return RC_OK;
}

int TCropFilter::freeOutputData() {            // uvoľnenie výstupných dát
    if (OutPlane.data != NULL) {
        free(OutPlane.data);
        OutPlane.data = NULL;
    }
    return RC_OK;
}

```

cropdll.h (definícia knižnice):

```

#include "crop.h"                // definícia triedy
#define INDEX_CROP      0       // zoznam indexov tried v knižnici

#define LIB_FILTER_COUNT    1    // počet tried v knižnici
#define LIB_NAME            "Crop Library" // meno knižnice
#define LIB_MAJORV          1    // hlavná verzia knižnice
#define LIB_MINORV         0     // vedľajšia verzia knižnice

TFilterLibraryInfo CropFilterLibraryInfo = {    // informačná štruktúra o knižnici
    LIB_NAME,
    LIB_MAJORV,
    LIB_MINORV,
    LIB_FILTER_COUNT};

```

cropdll.cpp (implementácia knižnice):

```

#include <windows.h>              // systémové definície
#include "filterdll.h"           // definície typov a exportovaných funkcií
#include "cropdll.h"             // definícia knižnice

HINSTANCE hInstance;            // systémový identifikátor inštancie knižnice

```

```

BOOL APIENTRY DllMain(HINSTANCE hModule, DWORD ul_reason_for_call,
LPVOID lpReserved) { // hlavná funkcia knižnice volaná pri jej načítaní
switch (ul_reason_for_call) {
case DLL_PROCESS_ATTACH:
    hInstance = hModule; // uloženie systémového identifikátora
    break;
case DLL_THREAD_ATTACH:
case DLL_THREAD_DETACH:
case DLL_PROCESS_DETACH:
break;
}
return TRUE;
}

EXPORT TFilterLibraryInfo GetFilterLibraryInfo(void) { // vráti informácie o knižnici
return CropFilterLibraryInfo;
}

EXPORT TFilterInfo GetFilterInfo(int idx) { // vráti informácie o triede filtrov
switch(idx) { // podľa indexu
case INDEX_CROP:
return getCropFilterInfo();
}

TFilterInfo finfo; // ak index nemá žiadna trieda, vráti sa prázdna štruktúra
finfo.FilterName = "";
finfo.Magic = 0;
finfo.MajorVersion = 0;
finfo.MinorVersion = 0;
return finfo;
}

EXPORT TFilter* CreateFilter(int idx) { // vytvorí objekt triedy s indexom idx
TFilter* filter;
filter = NULL;
switch(idx) { // podľa indexu vytvorí objekt
case INDEX_CROP:
filter = new TCropFilter(hInstance); // systémový identifikátor kvôli zdrojom
break;
}
return filter; // vráti vytvorený objekt
}

EXPORT void DestroyFilter(TFilter *filter,int idx) { // uvoľní objekt triedy
if (filter != NULL) {
if (idx == -1)
delete filter; // ak všeobecný index, tak použije všeobecný deštruktor
else
switch(idx) { // inak podľa indexu triedy zavolá daný deštruktor

```

```
        case INDEX_CROP:
            delete (TCropFilter*)filter;
        }
    }
}
```