Referenční příručka k jazyku KRKAL C

Jiří Margaritov
j.margaritov@volny.cz

Obsah

1. Stručný úvod do KRKAL C

Krkal C je jazyk, který syntaxí svých příkazů a výrazů vychází z jazyka C. Pro účely skriptovacího systému KRKAL však byl jazyk obohacen o některé nové rysy. V následujících kapitolách se pokusím zdokumentovat veškeré vlastnosti jazyka KRKAL C, nicméně bych rád upozornil, že tento dokument je referenční příručka a nikoliv učebnice. Pokud chcete poradit, jak rychle napsat skript, přečtěte si prosím dokument *Jak psát skripty*.

Rovněž si u čtenáře této příručky dovoluji předpokládat alespoň zběžnou znalost programování v jazyce C, jehož syntaxi zde nebudu zdlouhavě popisovat. Upozorním pouze na rozdíly, kterými se KRKAL C od tradičního C odlišuje. Čtenáři určitě nebude na škodu ani znalost nějakého objektově orientovaného jazyka – např. C++. KRKAL C má objektové rysy, ovšem mezi objektem KRKAL C a objektem C++ jsou značné rozdíly, takže rozhodně nedoporučuji tyto pojmy libovolně zaměňovat.

2. Prvky jazyka převzaté z C a jejich rozšíření

Nejprve proberu ty prvky, které jsme do KRKAL C převzali z tradičního C. Upozorním, pokud něco v KRKAL C chybí nebo naopak doplním to, co je tam navíc. Ovšem nejdůležitější nové rysy KRKAL C jsou popsány až ve třetí kapitole.

2.1. Preprocesor

KRKAL C implementuje pouze některé nejdůležitější direktivy preprocesoru. Lze používat následující direktivy v jejich obvyklém významu:

direktiva	význam
#define *symbol* [*hodnota*]	Definuje *symbol*, případně k němu přiřadí *hodnotu*.
#undef *symbol*	Oddefinuje *symbol.*
#ifdef *symbol*	Podmíněný překlad až do prvního #endif, pokud byl definován *symbol*.
#ifndef *symbol*	Podmíněný překlad až do prvního #endif, pokud nebyl definován *symbol*.
#endif	Ukončuje podmíněný překlad.
#head	Deklaruje hlavičku zdrojového souboru.
#endhead	Ukončuje hlavičku zdrojového souboru.

Poznámka k #define: Přiřazení se děje na úrovni textového řetězce, tj. *hodnota* musí být řetězec znaků nepřerušený whitespace znaky. Uvedení jména symbolu definovaného s hodnotou kdekoliv ve zdrojovém souboru pak má za následek nahrazení každého výskytu jména symbolu řetězcem *hodnota*. Makra s parametry nejsou podporována.

Poznámka k #include: Preprocesor nepodporuje direktivu #include, protože vkládání souboru se řídí v hlavičce. Z pohledu uživatele to znamená jen tolik, že místo ***#include***, na které byl zvyklý v C, je nyní třeba psát ***include*** bez mříže na začátku, a to výhradně do hlavičky souboru.

Hlavička

Hlavičkou nazýváme krátkou část na začátku každého KRKAL C souboru, která je vymezena direktivami preprocesoru #head a #endhead. Hlavička obsahuje některé důležité údaje o souboru

(například jeho verzi) a rovněž seznam souborů, které se mají přeložit před vlastní kompilací tohoto souboru.

Na každé řádce hlavičky může být uvedena některá z následujících direktiv:

direktiva hlavičky	význam
game *jméno hry*	Jméno právě vytvářené hry.
author *jméno autora hry*	Jméno autora – toho, kdo hru programuje.
version *verze hry*	Verze hry.
include *jméno souboru*	Vložení souboru – obdoba direktivy #include.

Poznámka k direktivám game a autor: Tyto direktivy nemají na překlad žádný vliv, pouze se v případě úspěšné kompilace uloží spolu s ostatními údaji do registru pod klíč příslušného skriptu.

Poznámka k direktivě version: Verze je 64 bitová hodnota, zde ovšem zapsaná jako čtveřice čtyřciferných hexadecimálních čísel oddělených podtržítky. Příklad verze: `0001_FFFF_0001_0001.` Skripty se identifikují právě pomocí svojí verze.

Poznámka k direktivě include: Jméno souboru musí být uvedeno i s cestou, pokud se nenachází přímo v adresáři vytvářené hry. Nevyžadují se uvozovky ani špičaté závorky, ale jméno souboru tím pádem nesmí obsahovat mezeru a musí být uvedeno i s příponou, která je standardně „.kc“).

2.2. Příkazy

Narozdíl od preprocesoru jsou v KRKAL C implementovány všechny příkazy jazyka C, s výjimkou příkazu ***goto***. Tedy příkazy ***if, while, do, for, switch, break, continue a return*** lze používat stejně jako v jazyku C.

2.3. Konstanty

Rovněž konstanty lze zapisovat stejně jako v C s jediným nepodstatným omezením, které vyplývají z rozdílné typové sady. U *celočíselných konstant* nelze v KRKAL C specifikovat písmenem na konci konstanty její typ. Nelze tedy například psát:

```
-123L
```

ale pouze

```
-123
```

Celá čísla je možné zapisovat i v osmičkové a šestnáctkové soustavě:

```
020       // == 16
0xFF      // == 256
```

Veškeré celočíselné konstanty se považují buď za typ int, nebo za typ char, podle kontextu v případě přiřazení.

Konstanty s desetinnou tečkou se vždy považují za konstanty typu double a je možné je zapisovat jak běžným způsobem:

```
2.8
```

```
-3.12345
```

tak za pomocí exponentu:

```
1.7e-12
-11E23          // na velikosti písmene E nezáleží
```

Znakové konstanty se uvozují apostrofem a jsou plně podporovány, včetně escape sekvencí. Lze tedy psát např.:

```
'a'
'\n'
'\0'
```

Řetězcové konstanty je třeba zadávat v uvozovkách:

```
"ahoj"
""              // prázdný řetězec
```

2.4. Základní datové typy

KRKAL C sice nemá stejně bohatou nabídku číselných datových typů jako klasické C, zato však disponuje zcela novými datovými typy, se kterými se v C nesetkáme. Datové typy, jejich popis a velikosti znázorňuje následující tabulka:

jméno typu	velikost v B	příklad deklarace	poznámka
char	1	char c;	Char se vždy chápe jako unsigned.
int	4	int i;	Int se vždy chápe jako signed.
double	8	double d;	Double odpovídá typu double z C.
void	-	-	Prázdný datový typ, stejný jako v C.
objptr	4	objptr o;	Pointer na objekt (viz kapitola 3).
name	4	name n;	KSID jméno (viz kapitola 3).
string	1 - 251	string s1; string[38] s2;	KRKAL C podporuje řetězce délky 0 – 250 znaků, vždy jsou zakončené nulou.

2.5. Pole kernelu

Mezi základní datové typy se počítají ještě pole kernelu. Jsou to dynamická pole, která je třeba vždy nejprve vytvořit pomocí operátoru new, pouze v případě globálních proměnných těchto typů se alokace provádí automaticky. Existuje celkem 5 typů polí kernelu. Jedná se o vzájemně nekompatibilní datové typy.

jméno typu	typ položky
chara	char
inta	int
doublea	double
objptra	objptr
namea	name

U všech těchto typů deklarovaná proměnná označuje pointer na pole kernelu, který je třeba nejprve inicializovat operátorem ***new***:

```
inta ip;
ip = new inta;
```

Poté je možné přistupovat k prvkům jako u kteréhokoliv jiného pole:

```
ip[0] = 2;
ip[1] = ip[0] + 17;
```

Výhodou polí kernelu je fakt, že se nemusíme starat o jejich velikost. Můžeme do něj přidávat libovolné množství prvků a pole se bude dynamicky zvětšovat, dokud se nevyčerpá veškerá volná paměť. Navíc lze přes pointer pole volat dvě speciální metody:

```
int count = ip->GetCount();
count += 200;
ip->SetCount(count);
```

jméno metody	návratová hodnota	parametry
GetCount()	int – počet prvků pole	žádné
SetCount(int set)	void	nová velikost pole

Metoda GetCount vrací aktuální počet prvků uložených v poli. Metoda SetCount nastavuje novou velikost pole. Při té příležitosti se může pole fyzicky zvětšit nebo zmenšit.

Poté, co již pole nepotřebujeme je vhodné jej dealokovat operátorem ***delete***.

```
delete ip;
```

2.6. Modifikátory základních typů

U parametrů vracených hodnotou (viz kapitola Metody) a u funkcí, které vracejí typ int, lze příslušný typ ještě modifikovat pro návrat. Modifikátory jsou zároveň klíčová slova jazyka a jejich seznam je uveden v následující tabulce:

modifikátor	význam
ret	vracení hodnotou
retand	vracení hodnotou s operací and
retor	vracení hodnotou s operací or
retadd	vracení hodnotou s operací add

`ret` znamená, že hodnota parametru se po skončení metody zapíše zpět, hovoříme o *vracení hodnotou*. Typy všech metod, které mají návratovou hodnotu se uvažují automaticky s modifikátorem ret.

`retand, retorr` a **`retadd`** mají význam tam, kde existuje více metod stejného jména – ty se potom volají všechny najednou. Při návratu (ať už hodnoty funkce nebo parametru) se navíc použije příslušná operace, tj. všechny návratové hodnoty se buď andují, orují a nebo sečtou. K této problematice se znovu vrátíme v kapitole Metody.

2.7. Null

Krkal C rozlišuje nulu (číslo 0 typu int) a **null**, což je nulový pointer typu void*. Null je klíčové slovo jazyka a označuje konstantu 0 typu void*. Existují i další nully, jejichž seznam uvádí následující tabulka:

klíčové slovo	null pro typ
null	void*
nnull	name
onull	objptr
anullint	inta
anullchar	chara
anulldouble	doublea
anullobjptr	objptra
anullname	namea
anullvoid	voida

2.8. Struktury

Struktury lze v KRKAL C definovat pouze uvnitř objektů, ale mají globální platnost. Struktura smí obsahovat položky libovolných základních datových typů, případně pointerů na ně. Struktura nesmí obsahovat jako položku další strukturu. Strukturu deklarujeme klíčovým slovem ***struct***:

```
struct sAkce
{
     sAkce     *next;
     int       x1, x2, y1, y2;
     objptra   objs1, objs2;
     name      typ1, typ2;
}
```

V tomto příkladu jsme deklarovali struktru sAkce. Jak je vidět, tak jejím členem může být také pointer na svůj vlastní typ. Zatím však máme pouze deklaraci typu struktury. Použití by mohlo vypadat třeba takto:

```
sAkce* a = new sAkce;
a->next = null;
a->x1 = 0;
...
```

2.9. Výrazy a operátory

Krkal C podporuje plnou sadu operátorů jazyka C, se stejnými prioritami a s pořadím vyhodnocování vždy zleva doprava. Výrazy nelze nijak přetypovávat a je tedy třeba spolehnout se na implicitní konverze:

tabulka konverzí
(typ*) > void*
char > int > double
int > char
string > char*

Logické operátory && a | | se vyhodnocují vždy zkráceně.

2.10. Identifikátory

Stejně jako v C musí každý identifikátor začínat písmenem nebo znakem podtržítko a obsahovat smí jen písmena, číslice a znak podtržítko. V KRKAL C existuje ještě další omezení – identifikátor **nesmí začínat** na **_KN** (podtržítko, K, N) a uvnitř **nesmí obsahovat __M_** (podtržítko, podtržítko, M, podtržítko). Toto omezení je nutné kvůli kompilovaným skriptům, které identifikátory skládají právě z těchto skupin znaků a při jejich dalším použití uvnitř identifikátoru by snadno mohlo dojít ke zmatkům.

2.11. Struktura programu KRKAL C

Především: program v KRKAL C se skládá výhradně z:

1) definic KSID jmen,
2) definic závislostí mezi jmény,
3) definic objektů
4) definic globálních proměnných.

Kromě toho se ještě mohou na nejvyšší úrovni nacházet tzv. nečisté operace, ale ty prozatím ponechme stranou, neboť mívají zpravidla spíše destruktivní účinky. Mimo jiné je tedy vidět, že není přípustné definovat funkci s globální platností, lze definovat pouze metody u objektů (viz dále).

3. Speciální syntaxe KRKAL C

Syntaxe běžného C je příliš obecná pro programování skriptů, které mají konkrétní cíl – vytváření objektů, které by se uplatnily ve hře, spolupracovaly s kernelem atd. Proto jsme jí také výrazně obohatili mimo jiné o objektové rysy. Popis veškeré přidané syntaxe je uveden v této kapitole.

3.1. KSID jména

KSID jména jsou globálně platné identifikátory. Rozeznáváme celkem 4 typy jmen:

klíčové slovo	deklaruje
voidname	obecné KSID jméno – může označovat cokoliv
objectname	KSID jméno objektu
methodname	KSID jméno metody
paramname	KSID jméno parametru metody

3.1.1. definování KSID jména

KSID jména se definují na nejvyšší úrovni ve zdrojovém souboru. Definice je velmi snadná – například:

```
objectname bomba, mina, pas, znacka;
paramname  timer, X, Y;
voidname   jih, vychod, sever, zapad;
methodname vybuchni;

methodname zjistiPocet returns int retadd;
```

Začíná se vždy uvedením příslušného klíčového slova, potom následuje seznam definovaných KSID jmen oddělených čárkou a ukončený středníkem. Za definicí methodname ještě může následovat klíčové slovo ***returns***, které uvozuje návratový typ metody – v příkladu je uveden i s modifikátorem ***retadd***. Ovšem stále se jedná jen o definici KSID jména a nikoliv metody. Podmínkou pro zdárné definování jména je, aby již ve stejné verzi toto KSID jméno neexistovalo.

Pomocí operátoru :: můžeme definovat i složená KSID jména, a to do libovolné hloubky. Například:

```
methodname bomba::vybuchni;
```

KSID jménům se přiřadí identifikátory, které je možné spatřit například v kompilovaných skriptech nebo v registru. Identifikátor KSID jména má tento formát:

KSID *jméno_verze* [**__M_***druhá část jména_verze druhé části*[**__M_** *třetí část ...*]]

Například KSID jménu **bomba** by mohl odpovídat identifikátor

```
_KSID_bomba_0001_FFFF_0000_0001
```

a KSID jménu **bomba::vybuchni** zase identifikátor

```
_KSID_bomba_0001_FFFF_0000_0001__M_vybuchni_0001_FFFF_0000_0002
```

Pokud definujeme složené KSID jméno (např. *bomba::vybuchni*) a již je definováno KSID jméno tvořící prefix nově definovaného jména (konkrétně zde *bomba*), pak nové složené KSID jméno převezme prefix a s ním i jeho verzi. Předchozí případ by odpovídal situaci, kdy KSID jméno bomba bylo definováno ve verzi **`0001_FFFF_0000_0001`** a KSID jméno bomba::vybuchni ve verzi **`0001_FFFF_0000_0002.`**

Je důležité porozumět faktu, že se ve všech případech jedná jen o definice KSID jmen a ne jakýchsi globálních proměnných. S takto definovanými jmény se zachází jako s konstantami typu *name*. Hodnotou proměnné typu *name* je vždy buď některé KSID jméno a nebo nula (*nnull*). Je možné je například přiřazovat do proměnných typu name, nebo je porovnávat (viz dále). Také je možné mezi nimi vytvářet závislosti:

3.1.2. definování závislostí mezi KSID jmény

Závislost mezi KSID jmény je třeba chápat jako budování acyklického orientovaného grafu. Nejprve se definují uzly – to jsou KSID jména, tak jak jsme je definovali v předchozím odstavci.

Poté je možné uzly spojovat orientovanými hranami, ovšem tak, aby v grafu nevznikl cyklus. Kernel, který tento graf jmen spravuje nad ním vytvoří tranzitivní obal a dokáže potom odpovídat na otázky typu: Jsou tato dvě jména spojena orientovanou hranou? Což už není nic jiného, než porovnávání KSID jmen.

Závislost mezi již existujícími KSID jmény vytvoříme pomocí klíčového slova ***depend***, resp. ***depends.***

Syntaxe deklarace ***depend*** je následující:

```
depend množina1 >> množina2;        nebo
depend množina1 << množina2;
```

množina1 a *množina2* mají jednu ze čtyř následujících podob:

a) *KSID jméno*
b) { *KSID jméno1*, *KSID jméno2*, ..., *KSID jméno n*}
c) *množina3* >> *množina4*
d) *množina3* << *množina4*;

Složitý zápis syntaxe osvětlíme několika názornými příklady:

```
depend    smer << jih;
```

je totéž jako

```
depend    jih >> smer;
```

a vytváří závislost mezi KSID jmény *smer* a *jih*. KSID jméno *jih* se přidá do množiny *smer*.

Místo jednoho KSID jména lze uvést celý seznam – množinu KSID jmen, a to na obou stranách šipky:

```
depend    smer << {jih, sever, vychod, zapad};
depend    {smer, strana} << {jih, sever, vychod, zapad};
```

Ve druhém případě se směry přidají jak do množiny *smer*, tak do množiny *strana*.

V deklaraci depend se mohou šipky řetězit; například:

```
depend    pojem << {smer, strana} << {vychod, zapad};
```

Tato deklarace přidá KSID jména *vychod* a *zapad* do množin *smer* a *strana* a KSID jména *smer* a *strana* do množiny *pojem*.

Deklarace ***depends*** funguje podobně jako ***depend*** a její syntaxe je následující:

```
depends
{
depend-decl1,
depend-decl2,
...
```

```
depend-decln
}
```

Za klíčovým slovem ***depends*** následuje ve složených závorkách seznam deklarací dle syntaxe ***depend*** oddělených středníky - například:

```
depends
{
smer << {jih, sever};
strana << {vychod, zapad};
{smer, strana} >> pojem;
}
```

3.1.3. známá jména

Některá KSID jména není třeba definovat, protože mají nějaký speciální a předem známý význam. Proto se nazývají *známá jména*. Známá jména začínají znakem zavináč @. Rovněž mezi známými jmény se dají vytvářet závislosti. Plný seznam známých jmen, spolu s jejich popisem, je možné nalézt v referenční příručce kernelu.

3.1.4. porovnávání KSID jmen

Porovnat dvě KSID jména znamená zjistit, zda mezi nimi existuje závislost. Ve zdrojovém kódu lze psát konstrukce typu:

```
name n1, n2;

n1 = bomba::vybuchni;
n2 = smer;

if(n1 < n2)
     ...
```

Porovnání ve finále provede vždy kernel, který spravuje graf KSID jmen. V kompilovaných skriptech by se tento test přeložil jako volání služby kernelu pro exkluzivní porovnání dvou KSID jmen, podobně interpret by na tomto místě narazil na instrukci volání této služby kernelu.

3.2. Objekty

Základní představa o objektu v KRKAL C se příliš neliší od té, kterou máme o objektech v C++.

objekt = atributy + metody

3.2.1. definování objektu

Objekt vzniká definováním objektového jména (definice *objectname*). Takový objekt je prázdný, tj. nemá žádné atributy ani metody. Ty můžeme do objektu přidávat v upřesnění definice objektu – složených závorkách uvozených klíčovým slovem ***object*** a jménem objektu (tzv. *objektové závorce)*:

```
objectname bomba;
object bomba
{
     int  silaVybuchu;
     int  timer;

     void direct SetTimer(int _timer) { timer = _timer; }
}
```

V tomto objektu jsme definovali dva atributy (*silaVybuchu* a *timer*) a jednu metodu *SetTimer*.

3.2.2. postupné definování objektů

Předchozí definice objektu není konečná, pokud bychom chtěli, mohli bychom na jiném místě zdrojového souboru uvést další atributy a metody:

```
object bomba
{
     int odjistena;

     void direct Odjisti() { odjistena = 1; }
}
```

Takto je definice objektu doplněna o další atribut a další metodu. Objektových závorek můžeme ke každému objektu uvést libovolné množství a například tak „vylepšovat“ i objekty, které byly původně definovány v jiných verzích.

3.2.3. vytvoření objektu, pointer na objekt

Objekt lze kdykoliv vytvořit operátorem ***new***. Ten v případě vytváření objektu vrací typ objptr, což je pointer na objekt. Prostřednictvím pointeru na objekt můžeme s objektem pracovat – volat jeho metody, případně jej zrušit. Lze vytvořit buď objekt pevně zadaného jména a nebo vytvořit objekt jména, které je uloženo např. v proměnné typu name.

```
objptr o1 = new bomba;
```

Přímé vytvoření objektu typu bomba. Volá se constructor objektu bomba.

```
name n = bomba;
objptr o2 = new vartype n;
```

Totéž, ale nepřímo přes proměnnou typu name. Za ***new*** je pak třeba použít klíčové slovo ***vartype***.

3.2.4. konstruktory a destruktor

Existují metody, které mají v objektu zvláštní postavení – odpovídají právě těm známým jménům, která se píší bez zavináče:

metoda	známé jméno	provádí
constructor	Constructor	inicializaci atributů
lconstructor	LoadConstructor	inicializaci při nahrávání do levelu
cconstructor	CopyConstructor	vytváří identickou kopii objektu
destructor	Destructor	deinicializace při rušení objektu
uconstructor	-	totéž jako constuctor a lconstructor dohromady

Více informací o tom kdy a za jakých okolností se tyto metody volají lze nalézt v dokumentu Jak psát skripty. Zde bych jen rád zdůraznil, že žádná metoda nemá parametry ani návratový typ a každou lze explicitně zavolat. Konstruktorů a destruktorů lze definovat libovolné množství a v daný okamžik se potom volají všechny najednou. ***uconstructor*** je pouze syntaktická zkratka a odpovídá situaci, kdy bychom napsali totožný constructor a lconstructor. Pro jistotu nyní ještě uvedu příklad definice constructoru:

```
constructor()
{
     triger = onull;
     mforce = onull;
}
```

3.2.5. edit-tagy objektu

Objekty existují nejen v textové podobě ve zdrojovém souboru skriptu, ale i jako zcela konkrétní a viditelné entity v editoru levelů. Editor potřebuje znát celou řadu informací o objektu a jeho atributech. Pokud chceme ovlivnit chování objektu v editoru, použijeme k tomu některý z tzv. ***objektových edit-tagů***. Objektové edit-tagy zadáváme tak, že ještě před definicí prvního atributu, resp. metody použijeme klíčové slovo ***edit***.

```
object oDracek
{
     edit {InMap, UserName = "můj krásný dráček", Comment = "co
          mi jen dal práce..."}
}
```

Do složených závorek za klíčovým slovem edit pak již můžeme psát jednotlivé edit-tagy oddělené čárkou. Následuje přehled všech objektových edit-tagů a jejich stručný popis.

edit-tag	za ním následuje	význam
UserName	= “string”	Uživatelské jméno objektu, které editor zobrazí.
Comment	= “string”	Komentář.
InMap	-	Objekt je umístitelný do mapy, zarovnává se podle mřížky.
OutMap	-	Objekt je umístitelný mimo mapu.
NoGrid	-	Objekt je umístitelný do mapy, nezarovnává se podle mřížky.
Editor	-	Objekt zajímavý pro editor. InMap a OutMap automaticky nastavují i tag Editor.
CollReplace	-	Kolize objektů se řeší vyhozením starého objektu z mapy (default).
CollDontPlace	-	Kolize objektů se řeší neumístěním nového objektu do mapy.
CollIgnore	-	Kolize objektů se neřeší a v mapě budou oba najednou.

3.2.6. zjištění typu objektu

Z pointeru na objekt je možné získat jméno objektu pomocí operace ***typeof***:

```
objptr o = new bomba;

...

name n = typeof(o);       // n == bomba
```

3.2.7. rušení objektu

Objekt zrušíme operátorem ***delete***, kterému předáme pointer na objekt:

```
objptr o = new bomba;

delete o;
```

3.3. Atributy

Data objektu nazýváme atributy. Atribut může být definován k libovolnému datovému typu. Atributy mají platnost v rámci objektu a tedy v rámci objektu nemohou existovat dva atributy stejného jména – ovšem je třeba uvažovat o jménech atributů tak jak je eviduje kernel:

_KSOV_*jméno objektu*_*verze objektu*__**M**_*jméno atributu*_*verze atributu*

3.3.1. definování atributu

Atribut definujeme podobně jako například v C++. Uvnitř objektu uvedeme typ a jméno atributu. Samozřejmě je možné definovat za jedním typem více atributů, jejichž jména jsou oddělena čárkami, pouze je třeba dát pozor na pointery, protože například definice

```
object oDracek
{
     int* px, py;
}
```

definuje atribut *px* typu *int*,* ale atribut *py* pouze typu *int*.

3.3.2. přístup k atributům

Přístup k atributu je možný pouze z objektu, ve kterém je atribut definovaný. Vyjádřeno terminologií C++ to znamená, že všechny atributy jsou vždy definovány jako soukromé (private). K atributům tedy mohou přistupovat pouze metody objektu.

3.3.3. atributy známých jmen

Již jsme se setkali se známými jmény. Rovněž některé atributy mohou být kernelu známé a vyžadovat z jeho strany speciální zacházení. Kernel ví, jak chápat hodnotu v atributu známého jména a může ji jak číst, tak měnit. Jména takových atributů se opět uvozují znakem zavináč. Plný seznam a popis atributů známých jmen se nachází v referenční příručce ke kernelu.

3.3.4. dědění atributů

V KRKAL C je implementován mechanismus dědičnosti pro atributy i metody. Ovšem na rozdíl od C++ se nic nedědí implicitně, nýbrž se vždy vyžaduje explicitní určení, že se příslušný atribut nebo metoda mají zdědit. Otázkou je kam se vlastně budou dědit? Zatím nepadla ani zmínka o tom, že by se dal vytvořit potomek objektu. Odpověď je jednoduchá – dědí se v případě, že mezi jmény objektů byla stanovena závislost. Je dobré si pamatovat, že atributy i metody se dědí vždy *proti směru šipek* (viz << na konci následujícího příkladu). Že se má atribut zdědit určíme uvedením klíčového slova ***inherit*** v typu atributu. Tedy například:

```
objectname o1, o2;

object o1
{
     inherit int a;
     int b;
}
```

```
object o2
{
     int c;
}

depend o1 << o2;
```
// díky této závislosti zdědí objekt o2 atribut o1::a

Stejně postupujeme i v případě, že chceme zdědit metodu. Kdybychom chtěli zdědit všechny atributy i metody v objektu – přesněji v příslušné objektové závorce – bylo by zbytečně pracné uvádět u každého atributu i metody slovo *inherit*. V takovém případě můžeme prohlásit, že v dané objektové závorce se dědí úplně vše – uvedením klíčového slova ***inherit*** ještě před otevírající závorkou definice objektu. Tedy kdybychom v minulém příkladu definovali objekt *o1* takto:

```
object o1 inherit
{
     int a;
     int b;
}
```

pak bychom již nemuseli psát *inherit* u jednotlivých atributů a oba by se zdědily automaticky.

3.3.5. rozlišení děděných atributů

Díky struktuře identifikátoru atributu tak jak jej chápe kernel (viz začátek této kapitoly) je zřejmé, že v objektu mohou existovat i atributy, jejichž krátká jména jsou stejná. Tato situace snadno nastane například právě při dědění:

```
objectname o1, o2;
depend o1 << o2;

object o1 inherit
{
     int a;
     int b;
}

object o2
{
     int a;
}
```

Objekt *o1* má dva atributy:

```
_KSID_o1_0001_FFFF_0000_0001__M_a_0001_FFFF_0000_0001
_KSID_o1_0001_FFFF_0000_0001__M_b_0001_FFFF_0000_0001
```

Objekt o2 má celkem 3 atributy:

```
_KSID_o1_0001_FFFF_0000_0001__M_a_0001_FFFF_0000_0001
_KSID_o1_0001_FFFF_0000_0001__M_b_0001_FFFF_0000_0001
_KSID_o2_0001_FFFF_0000_0001__M_a_0001_FFFF_0000_0001
```

Je zřejmé, že pro kernel tato situace nepředstavuje žádný problém, protože v každém objektu se identifikátory atributů liší. Nicméně pokud budeme nyní chtít v objektu *o2* přistoupit k atributu *a*, a napíšeme v tělu některé z metod objektu *o2* například metody:

```
a = 0;
```

pak není zcela jasné, který atribut se vybere – zda ten zděděný z objektu *o1* a nebo ten nově definovaný v *o2*. Platí , že přednost mají atributy, které vznikly ve stejném objektu, nicméně přistupovat můžeme i k těm zděděným, a to sice pomocí operátoru **::**
Tedy příkaz

o1::a = 0;

změní hodnotu atributu s identifikátorem

```
_KSID_o1_0001_FFFF_0000_0001__M_a_0001_FFFF_0000_0001
```

3.3.6. atributové edit-tagy

Podobně jako objekty jsou i atributy objektu často středem zájmu editoru levelů. A stejně jako u objektů by se rád dozvěděl některé informace i o atributech. Proto máme ***atributové edit-tagy***, které můžeme uvést do složených závorek za klíčové slovo ***edit***.

```
int trigger edit {Editable, UserName = "Časovač", DefaultValue = 100}
```

Následující tabulka uvádí úplný seznam atributových edit-tagů spolu s jejich významem:

edit-tag	za ním následuje	význam
Auto	= hodnota typu name	Atribut ovlivňuje automatismus.
Comment	= "string"	Komentář.
DefaultMember	= hodnota dle typu	Defaultní hodnota neinicializovaných položek pole.
DefaultValue	= hodnota dle typu	Defaultní hodnota atributu.
Editable	-	Atribut je editovatelný.
EditType	= typ editoru	Upřesnění typu pro editor – viz následující tabulka.
Exclusive	-	U intervalů určuje otevřenost (interval je ostrý).
IncludeNull		
InMap	-	U objektu nebo pole objektů určuje, že se objekt vybírá z mapy.
Interval	= {h1, h2} dle typu	Určuje interval hodnot, kterých může atribut nabývat.
Is	***op*** hodnota	***op*** může být libovolný relační operátor (<,<=,>,>=). Spolu s hodnotou určují jednostraný interval hodnot, kterých může atribut nabývat.
LevelLoad	-	Atribut se nahrává z levlu.
List	= {h1, h2, ... hn}	Seznam hodnot, kterých může atribut nabývat.
NoLevelLoad	-	Vyruší tag LevelLoad a atribut se tedy nebude nahrávat z levelu.
OutMap	-	Má podobně jako InMap význam pouze u objektů a polí objektů. Určuje, že objekt se bude vybírat mimo mapu.
PlanarNames	-	Má význam při zobrazení atributů ve stromě object

		browseru. Všechny atributy, které mají tento tag nastaven, se zobrazí ve stejné úrovni stromu.
SpecialEdit	-	Atribut je editovatelný speciálním způsobem – například výběrem oblasti z mapy.
UserName	= "string"	Uživatelské pojmenování atributu, které editor zobrazí.

Sada typů, kterou KRKAL C disponuje, je pro účely editoru poměrně chudá. Proto se dá pomocí edit-tagu ***EditType*** typ atributu blíže specifikovat. Seznam všech typů editoru je uveden v další tabulce, ze které je také patrné, se kterými základními typy se mohou použít.

typ editoru	smí se použít se základními typy	znamená
number	char, int, chara, inta	hodnota bude chápána jako číslo
letter	char, chara	hodnota bude chápána jako znak
bool	char, int, chara, inta	hodnota bude chápána jako boolean
string	chara	hodnota bude chápána jako string
void	name, objptr, namea, objptra	Tyto položky lze kombinovat operátorem | (bitové or). Výsledná hodnota určuje filtr na typ jména. anyname znamená libovolné jméno.
object		
method		
param		
automatism		
anyname		

3.3.7. skupiny atributů

Občas je vhodné se na několik atributů dívat jako na jednu skupinu. Příkladem může být následující objekt s atributy *x*, *y* a *z*:

```
object o
{
     int x, y, z;   // souradnice
}
```

Jestliže atributy opravdu označují souřadnice, pak bude jistě vhodné, aby se např. v editoru nastavovaly najednou, například výběrem souřadnic v mapě. Rovněž se dá očekávat, že edit-tagy bude chtít uživatel nastavovat společně všem třem atributům najednou a ne každému zvlášť. Proto je možné definovat tzv. ***skupiny atributů***.

Skupinu atributů definujeme uvnitř objektu jedním z následujících klíčových slov:

klíčové slovo	skupina označuje	počet atributů	význam atributů ve skupině
point2D	bod ve 2D	2	souřadnice bodu x a y
point3D	bod ve 3D	3	souřadnice bodu x, y a z
cell2D	buňka ve 2D	2	souřadnice buňky x a y
cell3D	buňka ve 3D	3	souřadnice buňky x, y a z
area2D	oblast ve 2D	4	souřadnice levého horního rohu x1 a y1 a pravého dolního rohu x2 a y2
area3D	oblast ve 3D	6	souřadnice levého horního rohu x1, y1 a z1 a pravého dolního rohu x2, y2 a z2
cellarea2D	oblast buněk ve 2D	4	souřadnice levého horního rohu x1 a y1 a pravého dolního rohu x2 a y2

cellarea3D	oblast buněk ve 3D	6	souřadnice levého horního rohu x1, y1 a z1 a pravého dolního rohu x2, y2 a z2

Například definici předchozího objektu bychom mohli upravit následovně:

```
object o
{
     cell3D souradniceObjektu edit {UserName = "Moje", DefaultValue = 0}
     {
          int x, y, z;   // souradnice
     }
}
```

Tímto zavádíme do objektu skupinu *souradniceObjektu*, jejímiž členy jsou atributy *x*, *y* a *z*. Na přístupu k těmto atributům se nic nemění, tj. jsou to stále v první řadě atributy objektu *o*. Jméno *souradniceObjektu* je ve skriptech nedostupné – skupina se nedá adresovat jako celek. Skupiny se nedají vnořovat, ale lze je dědit pomocí klíčového slova ***inherit***, podobně jako ostatní atributy. Počet atributů ve skupině je závazný, tj. musí jich v ní být definováno přesně tolik, kolik je uvedeno pro daný typ skupiny v tabulce. Navíc všechny atributy ve skupině musí být výhradně typu int. Skupiny mohou, ale nemusejí být pojmenované.

Pro skupinu je možné určit edit-tagy stejně jako u atributů. Tyto edit-tagy se potom nastaví všem členům skupiny jako defaultní, tedy uvnitř skupiny je lze ještě „přebít" dalšími deklaracemi *edit*. Výjimkou jsou edit-tagy *UserName* a *Comment*, které se nedědí dovnitř skupiny, ale zůstávají platné pro skupinu jako celek.

3.3.8. skriptované proměnné

Dalším speciálním typem atributů jsou tzv. *skriptované proměnné*. Skriptovanou proměnnou definujeme v objektu následovně:

```
object o
{
     scripted x constructor m(17) edit {UserName = "skriptovaná"};
}
```

Povinné je klíčové slovo ***scripted*** následované jménem skriptované proměnné. Poté povinně následuje klíčové slovo ***constructor*** a jméno nějaké safe metody. Tato metoda bude sloužit jako konstruktor námi definované skriptované proměnné.

Nepovinně může ještě následovat celočíselný konstantní výraz , který napíšeme do závorek za jméno metody – tato hodnota se potom konstruktoru při vytváření proměnné předá. Dále může ještě následovat klíčové slovo ***edit*** a uvedení edit-tagů skriptované proměnné.

Skriptované proměnné mají význam především v editoru. Ve skriptech nejsou jako položky objektů programátorovi přístupné. Podobně jako skupiny atributů lze i skriptované proměnné dědit, stačí před jejich jménem použít klíčové slovo ***inherit***.

3.4. Metody

Rozlišujeme dva základní typy metod – ***direct*** a ***safe***. Pro podrobný popis dějů, které probíhají během jejich volání, odkazuji na dokumentaci kernelu, kompilátoru a interpretu. Zde se budeme zabývat pouze syntaxí volání a jaký rozdíl je mezi nimi z pohledu programátora.

3.4.1. direct metody

Direct metody se definují stejně jako metody v C++ . Protože však v KRKAL C neexistují hlavičkové soubory, je třeba ihned za seznamem parametrů uvést i tělo metody.

```
void direct Odjisti() { odjistena = 1; }
```

Důležité je nezapomenout na klíčové slovo ***direct***, protože implicitně jsou metody považovány za ***safe***.

V objektu nemůže existovat více direct metod stejného jména. Může však nastat stejná situace jako při dědění atributů a potom je možné mezi metodami vybírat pomocí operátoru **::** (viz kapitola dědění atributů).

Direct metody se nejvíce podobají funkcím z klasického C. Mají pevně stanovený počet parametrů a jejich typy. Všechny parametry musí být při volání zadány a typy parametrů musejí souhlasit. Direct metody mohou vracet jakýkoliv základní datový typ, kromě typu string. Volání direct metody je hračka, obecná syntaxe volání vypadá takto:

```
[Obj->]Met([v1, v2, ...vn]);
```

Obj je objekt, jehož metodu chceme volat. Neuvedeme-li jej explicitně, bude kompilátor předpokládat, že chceme zavolat metodu v aktuálním objektu.

Met je jméno metody. To může být složené z několika částí a zahrnovat operátor **::**

V kulatých závorkách následuje seznam parametrů, pokud ovšem metoda nějaké má. Hodnoty *v1* až *vn* jsou metodě při volání předány. Jestliže chce direct metoda přes parametry vracet hodnotu, musí si nechat předat pointer.

3.4.2. safe metody

Safe metody se nevolají přímo, ale přes kernel – díky tomu se také získaly označení safe. Safe metody mají složitější syntaxi, než direct metody, a příliš se nepodobají klasickým funkcím v C.

3.4.2.1. definice safe metody

Safe metodu nemůžeme definovat jen tak – uvedením jejího jména, protože mechanismus safe volání vyžaduje, aby každá safe metoda měla své jméno. Můžeme využít například definici ***methodname***. Kromě toho můžeme použitím klíčového slova ***decl*** vynutit definování safe metody přímo v těle objektu. Zatím budeme pro jednoduchost definovat jen safe metody bez parametrů.

```
methodname MyMethod1;
...
object o
{
```

```
     void safe MyMethod1() { }       // OK, deklarováno pomocí methodname
     void decl safe MyMethod2() { }// OK, použito decl
     void safe MyMethod3() { }       // chyba – ani deklarace ani decl
}
```

Kromě toho lze pomocí operátoru **::** metodu definovat lokalizovaně vůči objektu – v tom případě není nutné metodu předem deklarovat a nemusí se uvádět ani klíčové slovo *decl*.

```
object o
{
     void safe ::MyMethod4() { }
}
```

Takto definovaná metoda bude mít jméno složené ze dvou částí: o::MyMethod4.

3.4.2.2. volání safe metod

Všechny safe metody bychom volali v rámci objektu o (například z těla nějaké další metody) následovně:

```
MyMethod1();
MyMethod2();
MyMethod3();
::MyMethod4();
```

případně

```
o::MyMethod4();
```

ovšem pokus o zavolání

```
o::MyMethod1();
```

by skončil chybou, neboť ačkoliv byla metoda MyMethod1 definována v objektu *o*, má - stejně jako její jméno – globální platnost v celém programu a její jméno je MyMethod1 nikoliv o::MyMethod1.

3.4.2.3. návratová hodnota safe metody

Narozdíl od direct metod, může existovat více safe method stejného jména – resp. k tomuto jménu existuje více různých těl metod. Při volání takové několikanásobně definované metody se zavolají všechna těla v nedefinovaném pořadí. V tomto případě však může nejvýše jedna metoda z takové skupiny vracet hodnotu. Případně může více metod vracet hodnotu typu int, za předpokladu, že mají všechny nastavený stejný návratový modifikátor, a to jeden z:

```
retand,
retor,
retadd.
```

V takovém případě se na všechny návratové hodnoty použije operace bitové and nebo bitové or a nebo se tyto hodnoty sečtou.

3.4.2.4. safe metoda jako zpráva

Safe metodu je také možné volat ne okamžitě, ale jako zprávu. V tom případě ale nemůže safe metoda vůbec vracet návratovou hodnotu, ani využívat vracení parametrů hodnotou. Pokud chceme předat safe metodu kernelu jako zprávu – a pro názornost předpokládejme, že bychom rádi předali jako zprávu metodu *MyMethod1* – použijeme některý z následujících zápisů:

```
MyMethod1() message;
MyMethod1() end;
MyMethod1() nextturn;
MyMethod1() nextend;
MyMethod1() timed <int time>;
MyMethod1() callend <objptr obj>;
```

Všech šet variant se liší klíčovým slovem za seznamem parametrů metody. Kernel má fronty, do kterých ukládá požadavky na volání metod a tyto požadavky postupně a v pravý čas vyřizuje. Pro bližší popis této problematiky odkazuji na dokumentaci kernelu, zde bych vše jen zjednodušil a shrnul:

Message znamená, že metoda se bude volat na začátku aktuálního taktu kernelu, *end* znamená volání na konci aktuálního taktu, *nextturn* volání na začátku příštího taktu a *nextend* volání na konci příštího taktu.

Varianta *timed* vyžaduje ještě celočíselný parametr <*time*>. Kernel takovou metodu zavolá v okamžiku, když uplyne zadaný počet jednotek času kernelu (ms zaokrouhlené s ohledem na délku taktu 33 ms).

Varianta s *callend* se zavolá okamžitě, jakmile v objektu <*obj*> nepoběží žádná metoda.

3.4.2.5. parametry safe metody

Parametry safe metod nemají pevně stanovené pořadí, zato však musí být jejich jméno definováno, podobně jako jméno safe metody. Stejně jako u safe metod lze jméno parametru

a) předem definovat, například pomocí ***paramname***
b) definovat uvedením klíčového slova ***decl*** teprve v okamžiku použití v seznamu parametrů příslušné safe metody
c) definovat parametr lokalizovaně vůči metodě

Všechny tři varianty jsou znázorněny v následujícím příkladu:

```
paramname pa;

void decl safe MyMethod5(int pa, decl double pb, objptr ::pc)
{
     int i;
     i = pa + 1;
     pc->MyMethod5(pa: i, pb: pb+1, ::pc : pc)
}
```

Podíváme-li se dovnitř těla metody, pak zjistíme i to, jak se k parametrům přistupuje a jak probíhá volání safe metody s parametry.

3.4.2.6. volání safe metody s parametry

Uvnitř těla metody používáme jména parametrů naprosto standardním způsobem, pouze před lokalizovanými argumenty nepíšeme **::** Ovšem v okamžiku, kdy safe metodu voláme, nepíšeme do seznamu parametrů pouze hodnoty – tak jak tomu je u direct metod, ale vždy zapisujeme dvojici:

jméno argumentu **:** hodnota

V případě, že zapisujeme jméno argumentu, tak před lokalizovanými argumenty **::** píšeme.

Nemusíme dodržet pořadí argumentů definované v hlavičce metody, protože kernel si parametry dokáže podle jmen identifikovat a předat je vždy ve správném pořadí, nicméně bývá to dobrým zvykem.

Jméno parametru, kterému hodnotu při volání předáváme, nemusí být zadáno přímo, ale může se nacházet například v proměnné typu name:

```
name n1 = pa, n2 = pb, n3 = MyMethod5::pc;

MyMethod5(n1: 1, n2 : 36.4, n3 : onull)
```

3.4.2.6.1. defaultní hodnoty parametrů

U safe metod mohou mít parametry zadanou defaultní hodnotu, kterou jim kernel nastaví v případě, že při volání nebyl příslušný parametr zadán.

```
void decl MyMethod5(int ::p = 43) { }
```

3.4.2.6.2. kontrola přiřazení parametru

V těle safe s parametry je možné testovat, zda byl daný parametr při volání předán. K tomu slouží volání ***assigned***

```
void decl MyMethod6(int ::p = 43, objptr ::o = onull)
{
     if(!assigned(o))
          return;

     o->SetTrigger(value : ::p)
}
```

Do závorek za *assigned* je třeba zadat jméno parametru, u nějž chceme zjistit stav přiřazení. Volání vrátí nulu, pokud příslušný parametr nebyl při volání předán a nenulovou hodnotu v opačném případě.

3.4.2.6.3. vracení parametrů hodnotou

V parametrech safe funkcí může funkce vracet hodnotu. Stačí parametr deklarovat s některým návratovým modifikátorem (viz návratová hodnota funkce). V případě parametrů je však ještě potřeba zadat modifikátor i při samotném volání.

```
void decl MyMethod7(ret int ::a)
{
     a = a + 8;
}

…

int i = 4;

o->MyMethod7( ret ::a : i)

//   i == 12
```

Opět je možné používat i ***retand***, ***retor*** a ***retadd***, pokud jsou však u všech takových parametrů dodrženy stejné podmínky jako při stejných modifikacích u návratové hodnoty metody.

3.5. Verze

Prozatím jsme předpokládali, že více či méně pracujeme v souboru jedné verze. Nicméně v reálné situaci můžeme pracovat s objekty z různých verzí, a dokonce jednotlivé metody a atributy mohou mít v rámci jednoho objektu různé verze. Běžně necháváme verze být a ve zdrojovém souboru s nimi přímo nepracujeme. Někdy však nastane situace, že je potřeba rozlišit identifikátor právě podle verze.

3.5.1. verze v identifikátorech jmen

Mějme objekt *o* definovaný nezávisle ve dvou různých verzích, viz následující příklad:

```
#version 1111_1111_1111_1111

objectname o;  // KSID_o_1111_1111_1111_1111

object o
{
     int a;      // KSOV_o_1111_1111_1111_1111__M_a_1111_1111_1111_1111
}

...

#version 2222_2222_2222_2222

objectname o; // KSID_o_2222_2222_2222_2222

object o
{
     int a;      // KSOV_o_2222_2222_2222_2222__M_a_2222_2222_2222_2222
}
```

Jestliže do našeho programu includujeme obě verze, pak se obě verze sloučí a budeme mít dva objekty *o*, každý s jedním atributem *a*. Příslušná kernelí jména jsou zapsána v komentáři příkladu. Chceme-li rozlišit mezi identifikátory objektů, musíme explicitně zadat číslo verze – použijeme k tomu operátor zavináč a řetězec s verzí:

```
name n, m;

n = o@"1111_1111_1111_1111";
m = o@"2222_2222_2222_2222";
```

Tím, že jsme explicitně určili verzi objektu, jsme také vyřešili nejednoznačnost, nicméně verzi je možné explicitně určovat v každé části složeného jména, takže následující zápis (uvnitř některé metody druhého objektu o) je sice zbytečně zdlouhavý, ale přesto platný:

```
int j = o@"2222_2222_2222_2222"::a@"2222_2222_2222_2222";
```

3.5.1.1. specifikátor „this version“

Verzi aktuálně překládaného souboru nemusíme zapisovat celým 20 znakovým identifikátorem, ale zkratkou „this version“. Použití by v předchozím případě mohlo vypadat třeba takto:

```
j = o@"this version"::a;      // "this version" místo dvojek
```

3.5.1.2. příkaz with

Kromě toho je možné použít příkaz ***with***, který způsobí, že v průběhu následujícího příkazu se bude ve chvílích nejasností za defaultní verzi považovat nikoliv aktuální verze, ale verze, kterou zadáme příkazu with.

```
with("1111_1111_1111_1111")
{
     j = o::a;
     ...
}
```

3.5.2. nečisté operace

Současná verze systému KRKAL pracuje pouze s otevřenými verzemi, což znamená, že všechny zdrojové soubory je možné měnit. Počítali jsme však s tím, že vytvořená dokončená a odladěná verze hry se uzavře a dá k dispozici širšímu okruhu programátorů, kteří by mohli využívat objekty této verze a na jejich základě programovat svoje vlastní hry.

V ideálním případě by tito programátoři nikdy nepotřebovali cokoliv v uzavřené verzi měnit. Nicméně dá se snadno odhadnout, že by mohla vzniknout potřeba uzavřenou verzi změnit – ať už kvůli nalezené chybě, nebo kvůli novým záměrům programátora.

Jazyk KRKAL C obsahuje syntaxi pro takzvané nečisté operace. Jedná se o dodatečné zrušení metody, atributu nebo závislosti, modifikaci metody či atributu a o dodatečné přidání specifikace dědění inherit.

Tyto operace jsou pochopitelně velmi drastické a kompilátor se s nimi nemůže vypořádat jinak, než opakováním celého překladu. V praxi by měla být maximální snaha se těmto operacím vyhnout, nicméně počítali jsme i s tím, že někdy prostě nebude zbytí.

3.5.2.1. rušení metody, atributu nebo závislosti

Metoda a atribut se ruší uvedením následující deklarace na nejvyšší úrovni zdrojového souboru:

```
remove „kernelí identifikátor metody nebo atributu";
```

V případě atributu používáme identifikátor začínající *_KSOV,* u metody *_KSM* a u závislostí dvojici identifikátorů *_KSID*.

Příklad:

```
remove „_KSOV_o_1111_1111_1111_1111__M_a_1111_1111_1111_1111"
```

Závislost zrušíme tak, že za klíčové slovo *remove* uvedeme postupně obě kernelí jména tvořící závislost:

```
remove „_KSID_jmeno1_1111_1111_1111_1111" „_KSID_jmeno2_1111_1111_1111_1111"
```

3.5.2.2. modifikace metody nebo atributu

Modifikace je uvozena klíčovým slovem ***modify*** a za identifikátorem následuje kompletní nová definice metody nebo atributu. Ukončena je klíčovým slovem ***endmodify***

Příklad:

```
modify „_KSOV_o_1111_1111_1111_1111__M_a_1111_1111_1111_1111“
        double a edit {UserName = „modifikovaný atribut”}
endmodify
```

3.5.2.3. dodatečná specifikace dědění (čistá operace)

Dodatečně lze specifikovat, že se má některá metoda zdědit. Stačí uvést kdekoliv na nejvyšší úrovni klíčové slovo ***inherit*** a za ním kernelí jméno entity, kterou si přejeme zdědit

Příklad:

```
inherit „_KSOV_o_1111_1111_1111_1111__M_a_1111_1111_1111_1111“
```

Tato operace není ve svojí podstatě „nečistá“, neboť není destruktivní a kompilátor kvůli ní nemusí opakovat překlad. Uvádím jí zde proto, že mechanismus jejího použití se podobá tomu u nečistých operací.