

# Základní datové typy, výčtové typy, struktury a unie v jazyce C a C++

## Logický typ

bool – false a true. V paměti zabírá jeden byte. Platí false < true.

Jestliže je očekávána logická hodnota, tak libovolné nenulové číslo konvertuje na true, nula na false,

použije-li se místo celého čísla logická hodnota, automaticky se převede na celé číslo - false na 0, true na 1

## Znakové typy

1bytové znakové typy: char, unsigned char a signed char

2bytový wchar\_t - pro práci s asijskými znaky nebo s kódem UNICODE.

znakové konstanty, např 'a' jsou typu char (v jazyce C jsou typu int).

## Celá čísla

celočíslné typy (stdint.h):

long long int, unsigned long unsigned long int (int lze vynechat), long, unsigned long

- typy s přesně určenou šířkou
  - int8\_t, int16\_t
  - uint8\_t, uint16\_t
- typy s určitou minimální šířkou – typy se šířkou alespoň N bitů
  - int\_least8\_t, int\_least16\_t
  - uint\_least8\_t, uint\_least16\_t
- typy s určitou minimální šířkou, pro něž jsou v dané architektuře výpočty nejrychlejší
  - int\_fast8\_t, int\_fast16\_t
  - uint\_least8\_t, uint\_fast16\_t
- typům které mohou obsahovat ukazatele n objekty, neboli do nichž lze uložit hodnotu typu void\* a pak ji přenést zpět aniž se změní
  - intptr\_t
  - uintptr\_t
- typy s maximální šířkou
  - intmax\_t
  - uintmax\_t

## Celočíslné konstantní výrazy

je to výraz, který dokáže překladač vyhodnotit již v době překladu (makro, konstanta, sizeof)

## Ukazatele

void\* nelze v C++ přiřadit ukazateli s doménovým typem

```
void * v;
```

```
int a,*b;
```

```
v=&a; // C++ a C
```

```
b=v; // C lze ale v C++ nikoliv
```

## Výčtové typy

deklarace\_výčtového\_typu: **enum** *jmenovka*<sub>nep</sub> { *seznam\_enumerátorů* } *seznam\_deklarátorů*<sub>nep</sub>;

## Jazyk C a C++

Např.

```
enum barvy { cerna = 22, bila = -33, modra = 2*cerna, zelena };
```

C: enum barvy b=cerna; C++ stačí: barvy b=cerna;

Aby se v jazyce C mohlo klíčové slovo enum vynechat, musí se deklarovat výčtový typ pomocí klíčového slova typedef.

*Struktury a unie* v C++ patří mezi objektové typy.

### *Struktury*

```
struct jmenovkanep { deklarace_složek } seznam_deklarátorůnep;
```

Např.:

```
struct TOsoba {  
char Jmeno[30];  
int Cislo;  
};
```

V jazyce C se na strukturu musí odvolávat „úplným zápisem“, tj. konstrukcí struct TOsoba Osoba;

V C++ stačí jmenovka, tj. TOsoba Osoba;

Aby se v jazyce C mohlo klíčové slovo struct vynechat, musí se deklarovat nový typ pomocí klíčového slova typedef: typedef struct TOsoba TOSoba;

### *Unie*

V C++ lze deklarovat anonymní unie, v jazyku C nejsou k dispozici. Ke složce anonymní unie se přistupuje pomocí samotného identifikátoru složky. Globální anonymní unie musí být statické.

## **Příkazy jazyka C a C++ - kategorizace, základní charakteristiky**

### *Příkazy*

Příkazem je též deklarace, tj. deklarace se může vyskytnout všude tam, kde syntaxe jazyka povoluje příkaz. Blok (složený příkaz) v C++ je definován následovně: blok: posloupnost\_příkazů

Blok v jazyce C je definován takto: blok\_v\_C: { posloupnost\_deklarací posloupnost\_příkazů }

Deklarace proměnné je dále povolena ve výrazu podmínka těchto příkazů: if, switch, while, for  
Ve všech těchto příkazech je deklarovaná proměnná lokální proměnou v rámci tohoto příkazu (bloku).

### *Skoky*

Nelze přeskočit deklaraci s inicializací, pokud se nepřeskočí celý blok, ve kterém se takový příkaz nachází. Toto omezení se týká příkazů goto a switch.

Následující zápis je nesprávný:

```
if (x) goto Pokracovani;  
int y = 10;  
//...
```

Pokracovani:

```
// ...
```

Následující zápisy jsou správné:

```
if (x) goto Pokracovani;  
int y; y = 10; //...
```

Pokracovani:

```
// ...
```

```
if (x) goto Pokracovani;
```

```
{ int y = 10; //...  
}
```

Pokracovani:

## Jazyk C a C++

// ...

Nesprávná konstrukce příkazu switch:

```
switch (c) { case 0: int x = 10; // ...  
break; case 1: // ...  
}
```

Správné konstrukce příkazu switch: `switch (c) { case 0: int x; x = 10; // ... break; case 1: // ... } switch (c) { case 0: { int x = 10; // ... break; } case 1: // ... }`

## Pole, ukazatele a dynamické proměnné v jazyce C a C++ - deklarace a použití polí a ukazatelů, alokace a dealokace v paměti C a C++

### *Pole*

Pole má přesně takovou velikost, aby obsáhlo všechny své prvky. Prvky pole mají indexy od 0 do "počet prvků pole - 1". Velikost pole musí být během překladu konstantní. Překladač musí znát při překladu kolik místa alokovat pro pole. Nemůžeme tedy použít proměnnou k určení velikosti pole.

Např. `int mojePole[5];` Deklarace pole inicializací: `h[15] = {10, 20, 50, 100, 200, 500, 1000, 2000};`

Řetězce v programech C++ jsou reprezentovány poli datového typu `char`. Např. znakovému poli lze přiřadit řetězec takto: `char text[] = "Toto je řetězec.";`

Tím alokujeme 17 slabik v paměti a uložíme řetězec na toto místo. Pokud spočítáme znaky našeho řetězce, zjistíme, že jich je pouze 16. Poslední 17 slabika je alokována pro uložení ukončujícího znaku řetězce (tento znak je na konec řetězce vložen automaticky). Ukončující znak je speciální znak, který je reprezentován konstantou `'\0'`, což je ekvivalentní s číslem 0.

### *Ukazatel*

dvě základní kategorie: ukazatele na data (objekty) a ukazatele na funkce.

Oba dva typy ukazatelů jsou speciální objekty, které obsahují adresy v paměti. Mají různé vlastnosti, účely použití a pravidla zacházení. Obecně řečeno, ukazatelé na funkce se používají pro přístup k funkcím a pro předávání funkcí jako parametrů jiným funkcím. S těmito ukazateli není možné provádět žádné aritmetické operace. Ukazatele na objekty můžeme inkrementovat a dekrementovat tak, jak je to při prohlížení polí nebo složitějších struktur potřeba.

Např. po deklaraci `void (*fce)(int);` je `fce` ukazatel na funkci s parametrem typu `int`, která nic nevrací.

### *Dynamické proměnné*

C pro alokaci a uvolnění paměti funkce: `malloc`, `calloc`, `realloc` a `free`

Ty lze použít i v C++, ale zpravidla se místo nich používají operátory `new` a `delete`.

Výsledkem operátoru `new` je ukazatel na označení typu.

Bez inicializace: `int* a = new int;` Výraz `*a` má nedefinovanou hodnotu.

S inicializací: `int* b = new int(10);` Výraz `*b` má hodnotu 10.

Alokace jednorozměrných polí `int *c=new int [10]`

Alokuje pole 10 prvků typu `int`. Vrací ukazatel na první prvek pole. Inicializace pomocí `new` nelze provést. Musí se provést cyklem `for`.

Alokace vícerozměrných polí matice (10x10)

```
Matice=new int* [10];
```

```
for( i=0;i<10;i++) Matice[i]=new int [10];
```

Pokud se alokace nepodaří, operátor `new` vrací 0, novější překladače vyvolávají výjimku `bad_alloc`.

## Funkce v jazyce C a C++ - deklaráce, parametry, použití

Funkce je část kódu, která provádí jednoduché, dobře definované akce.

Parametr je hodnota předávaná funkci, která ji použije ve svých operacích.

Prototyp je deklaráce funkce, která je uvedena před definicí funkce.

Volání funkce je vyvolání příkazů tvořících funkci.

Rekurze je proces, ve kterém funkce volá sama sebe.

Lokální proměnná je proměnná deklarovaná uvnitř funkce (existuje pouze do návratu z funkce).

Globální proměnná je proměnná deklarovaná mimo funkci (existuje do ukončení programu).

Funkci která nemá parametry lze zapsat v C++ dvěma způsoby.

```
int f(void);
```

```
int f();
```

V jazyce C je přípustná pouze první varianta. Druhá varianta znamená funkci, u níž není znám počet a typ parametrů.

Informativní *deklaráce*=prototyp funkce.

V C++ musí být před voláním funkce deklarován alespoň její prototyp. V jazyce C může volání funkce předcházet její deklaraci.

Parametry funkce mohou být předávány - hodnotou – v C a C++

- odkazem – pouze v C++

### *Implicitní hodnoty parametrů*

V C++ se mohou definovat implicitní hodnoty parametrů funkcí.

```
void f(int a, int b=0, int c=getch())
```

Předepíše-li se implicitní hodnota pro některý z parametrů, musí se předepsat implicitní hodnoty i pro všechny parametry které za ním následují. Pokud se některý z parametrů vynechá, musí se vynechat i následující.

```
f(3,5); - pro c se použije implicitní hodnota
```

### *Referenční funkce*

Funkce která vrací referenci.

### *Vložené funkce*

Je-li tělo funkce malé – jednořádkové, volání takové funkce není příliš efektivní, neboť počítač spotřebuje více času na předávání parametrů, skok do funkce a návrat z ní, než na vykonání samotného těla funkce. V C++ lze použít vloženou funkci, která se deklaruje pomocí specifikátoru *inline*. Překladač tak místo hlavičky funkci vloží samotné tělo.

### *Přetěžování funkcí*

V C++ lze v jednom programu definovat několik funkcí se stejným identifikátorem, pokud se liší počtem nebo typem parametrů.

```
void f();
```

```
int f(int);
```

```
double f(double);
```

```
void f(int,double);
```

Při volání vybere překladač funkci, jejíž formální parametry nejlépe odpovídají skutečným parametrům v zápisu funkce. Pokud typy skutečných parametrů neodpovídají přesně typům formálních parametrů žádné z funkcí, vybere překladač tu, pro kterou je konverze nejsnazší.

Výsledek rozhodování však musí být jednoznačný, jinak překladač oznámí chybu.

# Objektové typy – deklarace, složky a jejich přístupová práva, konstruktory a destruktory, použití

*Objektové typy* v C++: třída, struktura, unie

Třídy a struktury jsou v C++ téměř totéž, liší se jen implicitními přístupovými právy.

Tělo třídy obsahuje seznam deklarací složek třídy, kterými mohou být:

- atributy = datové složky
- metody = funkční složky
- typy – tzv. vnořené typy
- šablony

První způsob deklarace představuje informativní deklaraci, která se může vyskytnout i vícekrát. Další způsoby představují definiční deklaraci.

Deklarace objektového typu může být:

- globální
- lokální v deklaraci jiné třídy – vnořená třída
- lokální ve funkci

Pokud se v deklaraci vynechá jmenovka, jedná se o nepojmenovanou třídu. V deklaraci nepojmenované třídy se nesmí vynechat seznam deklarátorů.

## *Atributy*

Deklarace atributů je analogická deklaraci obyčejných proměnných.

## *Inicializace bez konstrukturu*

Instance objektového typu lze inicializovat stejným způsobem jako složky struktury nebo unie v jazyku C, pokud tento objektový typ:

nemá konstruktor, všechny atributy jsou veřejné a nekonstantní, ani jeden z jeho nestatických atributů není referencí, nemá předka, nemá virtuální funkci

## *Metody*

Deklarace metody je buď prototyp a nebo definiční deklarace. Pokud se uvede pouze prototyp, musí se metoda později definovat za deklarací třídy. Definuje-li se metoda mimo tělo třídy, musí se její jméno kvalifikovat jmenovkou třídy s použitím rozlišovacího operátoru.

## *Přístupová práva*

k omezení přístupu ke složkám třídy slouží specifikátory `private`, `protected` a `public` a deklarace přátel.

`public` – veřejně přístupné složky – jsou přístupné pro kteroukoliv část programu bez omezení

`private` – soukromé složky – jsou přístupné pouze pro metody této třídy a její přátele

`protected` – chráněné složky – jsou přístupné pouze pro metody této třídy a jejich potomky a pro jejich přátele

Složky, jejichž deklaraci nepředchází specifikace přístupu jsou implicitně

- soukromé ve třídách
- veřejné ve strukturách

## *Lokální proměnné a atributy*

Identifikátor lokální proměnné definované v metodě včetně parametru této metody může být shodný s identifikátorem atributu třídy. V takovém případě lokální proměnná metody zastihuje atribut třídy. Pokud je potřebné v takovéto metodě přistoupit k atributu třídy, musí se atribut třídy kvalifikovat jmenovkou třídy a rozlišovacího operátoru nebo pomocí ukazatele `this`.

### *Statické atributy a metody*

Mohou se volat v době, kdy neexistuje žádná instance patřičné třídy. Mimo metody dané třídy se musí kvalifikovat a to jednou možností

- jmenovkou třídy a rozlišovacím operátorem
- identifikátorem instance třídy a operátorem kvalifikace nebo operátorem nepřímé kvalifikace

### *Přátelé*

Přítelem dané třídy může být funkce nebo třída, která není členem dané třídy, které je povoleno přistupovat ke všem složkám dané třídy. `friend` `prototyp_funkce`;

- Přátelství není tranzitivní – pokud je třída B přítelem třídy A a třída C je přítelem B, není pravda že třída C je přítelem třídy A.
- Přátelství není ani dědičné.
- Přátelství se nevztahuje ani na vnořené třídy.

### *Konstruktory a Destruktory*

Konstruktory se volají při vytváření instance v rámci její definiční deklarace nebo alokace pomocí operátoru `new`, při konverzích, při předávání parametrů objektových typů hodnot apod.

Specifické vlastnosti konstruktorů a destruktůrů:

- jméno konstruktoru je tvořeno identifikátorem třídy
- deklarace konstruktoru nesmí obsahovat specifikaci typu vrácené hodnoty a to ani `void`
- nedědí se – odvozená třída ovšem použije ke své konstrukci konstruktory předků
- nesmí být virtuální, statické konstantní a nestále
- nelze získat jejich adresu
- mohou být volány pro instance deklarované s `cv`-modifikátory

inicializační část konstruktoru

Slouží pro inicializaci nestatických atributů instancí a k předání parametrů konstruktorům předků. Zapisuje se mezi hlavičkou a tělo konstruktoru. Statické atributy se v inicializační části nesmí uvádět.

`TA() : x(0), y(0) {}`

Definiční deklarace instance

Definice instance třídy znamená vždy volání konstruktoru. parametry konstruktoru se zapisují za identifikátor instance do kulatých závorek podobně jako skutečné parametry. Pokud se má volat implicitní konstruktor, definuje se instance bez prázdných závorek.

Implicitní konstruktor

konstruktor, který může být volán bez parametrů. Může mít parametry, ale musejí mít předepsány implicitní hodnoty.

Pokud třída nemá uživatelem deklarovaný konstruktor, překladač vytvoří implicitně deklarovaný implicitní konstruktor. Jestliže třída obsahuje alespoň jeden, žádný se nevytváří a pokud je potřeba, generuje se chyba.

Kopírovací konstruktor

Kopírovací konstruktor je konstruktor, který lze volat s jedním parametrem typu reference na danou třídu. Při nenadefinování vlastního kopírovacího konstrukturu a zároveň je atribut pole, přiřazení B=T nastane fakt, že pole z instance B bude ukazovat na stejné pole jako A.

#### Konverzní konstruktory

konstruktor třídy TA bez modifikátoru explicit, který lze volat s jedním parametrem, může překladač použít k implicitním nebo explicitním konverzím typu prvního parametru na typ třídy TA.

#### Explicitní konstruktory

Explicitní konstruktor je deklarován s modifikátorem explicit. Explicitní konstruktor na rozdíl od konverzního konstrukturu nelze použít k implicitním konverzím, ale pouze k explicitním.

#### Destruktory

Destruktory jsou metody, které se volají automaticky při zániku:

- lokální instance –ve chvíli kdy program opouští období platnosti
- globální instance a lokální statické instance – při ukončení programu
- dynamické instance - při volání delete

Destruktory jsou metody třídy s těmito specifiky:

- jméno destrukturu je tvořeno identifikátorem, před kterým je ~
- nemá žádný parametr
- deklarace konstrukturu nesmí obsahovat specifikaci typu vrácené hodnoty a to ani void
- nedědí se – odvozená třída ovšem použije ke své destrukci destruktory svých předků
- nesmí být statické, konstantní nebo nestálé
- nelze získat jejich adresu

Pokud není deklarovaný, překladač vytvoří implicitně deklarovaný destruktorem.

Po provedení těla destrukturu dané třídy se volají destruktory atributů objektových typů dané třídy a destruktory předků.

Skončí-li program voláním funkce exit(), nezavolají se destruktory lokálních nestatických instancí.

Globální a lokální statické instance budou zrušeny obvyklým způsobem.

Skončí-li program voláním funkce abort(), nezavolají se žádné destruktory.

V případě dynamických instancí se musí volat operátor delete.

## Odvozené třídy – význam, druhy dědění, polymorfni a abstraktní třídy

Dědičnost slouží k rozšiřování vlastností tříd.

Jazyk C++ podporuje vícenásobnou dědičnost, takže jedna třída může mít několik předků.

Má-li daná třída předky, v deklaraci odvozené třídy se za její jmenovkou uvede dvojtečka a seznam jmenovka předků oddělených čárkou. Před každou jmenovkou předka může být uveden jeden ze specifikátorů public,protecte nebo private a případně klíčové slovo virtual.

V seznamu předků nesmí být uvedena jmenovka právě deklarované třídy nebo vícekrát stejná jmenovka předka.

Přístupová práva:

- public – zděděné složky budou mít stejná přístupová práva jako má předek
- protected –veřejně přístupné a chráněné složky budou chráněné a soukromé zůstanou soukromé
- private – všechny zděděné složky budou v odvozené třídě soukromé

Jazyk C a C++

Pokud se neuvede specifikace přístupu, jsou defaultně:

- private – u class
- public- u struct

Přístupová práva na složku jdou v potomkovi změnit, pokud u předka nejsou soukromé.

Nevirtuální a virtuální dědění

jedna třída nemůže být vícekrát přímým předkem jiné třídy. Může se ale stát, že od jedné třídy např. TA budou odvozeny dvě třídy TB a TC a ty budou přímými předky třídy TD. Podobjekt třídy TA bude obsažen ve třídě TD jednou nebo dvakrát v závislosti na typu dědění.

Statické složky třídy jsou v paměti uloženy pouze jednou bez ohledu na typ dědění, počtu instancí dané třídy a jejich potomků. Obdobně identifikátory vnořených typů a výčtových konstant dané třídy existují v jejich potomcích pouze jednou bez ohledu na typ dědění.

Potomek může zastoupit předka.

Překladač může automaticky konvertovat instanci potomka na předka, jestliže:

- toto přetypování je jednoznačné
- předek je v místě operace přístupný

Pořadí konstrukce a destrukce

Při vytváření instance odvozené třídy se nejprve zkonstruuji zděděné podobjekty. Přitom se nejprve volají konstruktory virtuálních předků a potom nevirtuálních. V obou případech se postupuje podle pořadí zapsání v deklaraci odvozené třídy. destruktory jsou volané v opačném pořadí.

Při TD D bude výpis u nevirtuálního dědění

```
TATBTATCTD  
TDTCTATBTA
```

*Polymorfismus - POLYMORFNÍ TŘÍDY*

S instancemi se pracuje často pomocí ukazatelů, přitom zpravidla vznikají situace, kdy není známý přesný typ instance, na kterou daný ukazatel ukazuje a je potřebné vyvolat metodu skutečné instance. Třídy obsahující virtuální metody, se nazývají polymorfni.

Jestliže je v určité třídě TA deklarována metoda jako virtuální, bude virtuální i ve všech potomcích třídy TA. Při nové definici metody v potomkovi se metoda nemusí (ale může) deklarovat se specifikátorem virtual.

Virtuální metoda, která je v odvozené třídě nově definována, se nazývá předdefinovaná virtuální metoda.

Virtuální metodou mohou být i destruktory, když nemají v předkovi a v potomkovi stejné jméno.

Virtuální metody nemohou být konstruktory, statické metody ani spřátelené funkce.

Virtuální metody musí být v předkovi a potomkovi deklarovány se stejným jménem, počtem a typem parametrů. Typ vrácené hodnoty virtuální metody by měl být identický v předkovi i v potomkovi.

*Abstraktní třída*

Je třída, která může být použita jen jako předek jiné třídy. Nelze deklarovat instanci abstraktní třídy, ale lze deklarovat ukazatel nebo referenci na abstraktní třídu. Třída je abstraktní, jestliže má alespoň jednu čistou virtuální metodu. Čistá virtuální metoda nemá definici a její prototyp má tvar `virtual` prototyp =0; kde prototyp je vlastní prototyp nevirtuální metody.

## Přetěžování operátorů – význam, základní pravidla, kategorizace, použití

jazyk C++ umožňuje rozšířit definice většiny operátorů na objektové a výčtové typy.

Operátory se dělí do čtyř skupin

- operátor podmíněného výrazu `?:`, rozlišovací operátor `::`, operátor přímé kvalifikace `..`
- operátor indexování `[]`, volání funkce `()`, přiřazení `=`, nepřímé kvalifikace `->` a přetypování (typ) lze přetěžovat pouze jako nestatické metody objektových typů.
- operátory `new` a `delete` lze přetěžovat jako obyčejné funkce nebo jako statické metody objektových typů
- všechny ostatní operátory můžeme přetěžovat jako nestatické metody objektových typů nebo jako obyčejné funkce, které mají alespoň jeden parametr objektového nebo výčtového typu.

Přetížený operátor může změnit svůj význam, který má pro vestavěné typy, např. operátor `+` lze definovat jako násobení apod., ale zpravidla je snahou neměnit smysl přetíženého operátoru, aby se dal odhadnout jeho výsledek.

Přetížený operátor se deklaruje jako operátorová funkce. Pro parametry operátorové funkce nelze předepisovat implicitní hodnoty. Unární operátor se definuje jako obyčejná funkce s jedním parametrem nebo jako metoda bez parametru. Binární operátor se definuje jako obyčejná funkce se dvěma parametry nebo jako metoda s jedním parametrem.

Lze volat

```
TKomplexCislo z=a+b;
```

```
TKomplexCislo z=a.operator + (b);
```

*Operátor ++ --*

Lze přetížít prefixovou i postfixovou verzi.

Prefixový operátor se deklaruje jako obyčejná funkce s jedním parametrem nebo jako metoda bez parametrů.

Postfixový operátor se deklaruje jako obyčejná funkce se dvěma parametry, z nichž druhý je typu `int`, nebo jako metoda s jedním parametrem typu `int`. Parametr `int` slouží pouze k rozlišení prefixové a postfixové verze a nelze jej v operátorové funkci použít.

*unární operátor*

bitový posun - `~`

*binární operátor +, -*

Přetížení binárních operátorů `+` a `-` automaticky neznamena přetížení složených přiřazovacích operátorů `+=` `-=`. Tyto operátory se musí přetížit samostatně.

operátory přetěžované jen jako metody

*operátor přiřazení =*

Operátor se nedědí.

implicitní operátor přiřazení `TA& TA::operator = (const TA&)`

Aby bylo možné přetížený operátor přiřazení zřetěžit, musí vracet referenci na jeho levý operand podobně jako implicitně deklarovaný operátor přiřazení.

Kopírovací operátor přiřazení má smysl definovat, pokud se jedná o třídu, pro níž nemohl překladač vytvořit implicitní kopírovací operátor přiřazení nebo pokud obsahuje dynamicky alokovaný atribut.

#### *operátor indexování []*

Operátor indexování je binární operátor. Levým operandem je instance objektového typu, pravým operandem je index zapsaný mezi závorky. Přetížený operátor indexování je nestatická Metoda má tvar: `operator [] (parametr)`

Aby mohl být operátor indexování použit na levé straně, musí operátorová funkce vracet referenci na typ prvku

`int &operator [] (int i)`

Pokud má fce vracet pouze int, lze operátor použít jen na pravé straně.

#### *Operátor volání funkce()*

Přetížený operátor volání funkce je nestatická metoda mající tvar `operator ()` (seznam parametrů)

Operátor volání funkce vrací referenci, aby mohl být použit na levé straně přiřazovacího příkazu.

#### *operátor nepřímé kvalifikace ->*

Považuje se za unární. Operátor musí vracet buď ukazatel na nějakou třídu nebo instanci jiné třídy, v níž je operátor také přetížen.

#### *operátory new a delete*

## **Šablony – význam, deklarace, použití**

Šablony umožňují popsat najednou celou množinu funkcí, které se liší jen například typem jejich parametru, nebo množinou objektových typů, které se liší nejen například typem jejich atributu. Mají podobný význam jako makra, ale poskytují více možností. Na rozdíl od maker jsou šablony zpracovávány překladačem. Šablona představuje vzor, podle kterého překladač vytvoří funkci nebo objektový typ. Takto vytvořená funkce nebo objektový typ se nazývá instance šablony.

*Deklarace šablony* se v programu může objevit na úrovni souboru, uvnitř objektového typu nebo uvnitř šablony objektového typu. Šablona se nemůže deklarovat jako lokální v bloku.

`template <seznam parametru> deklarace`

Seznam parametrů představuje jednotlivé formální parametry oddělené čárkami, podobně jako v deklaraci funkce. Formální parametry šablony mohou být **hodnotové typy, formální typy nebo šablony objektových typů**.

Deklarace je

- deklarace nebo definice obyčejné funkce
- deklarace nebo definice třídy
- definice metody třídy
- definice vnořené třídy
- definice statického atributu šablony třídy nebo definice statického atributu třídy vnořené uvnitř šablony třídy
- definice vnořené šablony

## Jazyk C a C++

### Parametry šablony

- class
- typename
- template

### Typové parametry

Předepisují se pomocí klíčového slova typename nebo class.

```
template <class T, class V=int> class TA;
```

Pro parametr V je předepsána implicitní hodnota typu int. Když je uvedeno class, může se dosadit i neobjektový typ.

### hodnotové parametry

hodnotové typy se deklarují podobně jako formální parametry funkcí, musí však být jednoho z následujících typů

- celočíselný typ  
template <class T, int i> class TA {...};  
TA <double, 10\*5> A;
- výčtový typ
- ukazatel na objekt
- reference na objekt
- ukazatel na funkci
- třídní ukazatel

### Šablonové parametry

parametrem šablony může být i jiná šablona třídy. Skutečným parametrem v tomto případě musí být jméno existující šablony třídy.

```
template <class T, template<class U> class W> class TA;
```

### Třídy

šablona objektového typu může obsahovat stejné druhy složek, jako objektový typ.

### Metody

Metody, které nejsou definovány přímo v těle šablony, se musí definovat jako šablony. Jméno metody se musí kvalifikovat jmenovkou třídy, za níž následují v lomených závorkách jména formálních parametrů v témže pořadí.

```
template <class T, class U> void TA <T,U>::f1(){...}
```

### Funkce

Šablona funkce umožňuje popsat najednou celou množinu funkcí, které se liší jen například typem jejich parametrů.

```
template <class T>  
T Max(T a, T b){return a>b?a:b;}  
Max (1,2);
```

### Přetěžování

V jednom oboru viditelnosti lze deklarovat několik šablon funkcí se stejným jménem. Lze také deklarovat šablonu funkce a obyčejnou funkci se stejným jménem. Pravidla pro rozlišování mezi

šablonami jsou podobná jako pravidla pro rozlišování přetížených funkcí. Má-li překladač volbu mezi funkcí a šablonou, dá přednost funkci, pokud parametry přesně odpovídají.

*Explicitní specializace* šablony umožňuje deklarovat specifickou verzi šablony pro její určité skutečné parametry. Explicitní specializace může být deklarována pro:

- šablonu obyčejné funkce
- metodu šablony třídy
- statický atribut třídy
- třídu vnořenou do šablony třídy
- šablonu třídy vnořenou do jiné šablony třídy
- šablonu metody vnořenou do šablony třídy

*Parciální specializace šablon třídy* poskytuje alternativní definici k primární definici šablony pro určité druhy parametrů. Primární šablona musí mít alespoň informativní deklaraci před deklaraci parciálních specializací šablony. Parciální specializované deklarace se od primární deklarace liší tím, že za jménem šablony následují v lomených závorkách formální parametry, které určují způsob specializace.

Každá parciální specializace šablony představuje odlišnou šablonu, a proto musí být kompletně definována. Může obsahovat jiné složky než primární šablona.

## **Výjimky – význam, deklarace a použití výjimek dle normy C++**

Výjimka představuje situaci, která nastane v průběhu normálního chodu programu a způsobí, že program nemůže obvyklým způsobem dále pokračovat. Jinými slovy, výjimka je chyba v běhu programu.

Pokud vznikne výjimka, je třeba ji programově ošetřit.

V normě jazyce C++ lze pracovat pouze s tzv. synchronními (softwarovými) výjimkami, které vzniknou uvnitř programu a nejsou závislé na operačním systému. Pomocí výjimek v C++ tedy nelze zpracovávat výjimky jako např. dělení nulou, aritmetického přetečení, přístup na neplatnou adresu apod.. Tyto výjimky se nazývají asynchronní (hardwarové) a lze je ošetřovat pomocí mechanismu strukturovaných výjimek nebo výjimek nadstavbové knihovny.

Všechny operace, které by mohly vyvolat výjimky, se musí provádět v tzv. pokusném bloku (angl. try block). Pokusný blok se skládá ze složeného příkazu (angl. compound statement) a z jednoho či několika handlerů (angl. exception handler).<sup>1</sup>

Pokud při provádění operací ve složeném příkazu pokusného bloku nevznikne výjimka, proběhnou řádně všechny příkazy tohoto bloku a po jeho ukončení bude program pokračovat za hlídaným blokem – handlery se přeskočí.

Jestliže v průběhu některé z operací ve složeném příkazu pokusného bloku nastane výjimka, provádění příkazů tohoto bloku se předčasně skončí v místě, kde výjimka vznikla a řízení se přesune do některého z handlerů.

Pokud handler neukončí běh programu, může program po provedení tohoto handleru pokračovat za pokusným blokem, v němž se handler nachází.

Výjimka může vzniknout přímo ve složeném příkazu pokusného bloku nebo v některém z vnořených bloků či v některé z funkcí volaných ve složeném příkazu pokusného bloku. Pokusný

blok může být vnořen do jiného pokusného bloku a výjimku, která vznikne ve vnořeném pokusném bloku, může ošetřit handler v nadřazeném pokusném bloku.

Když vznikne výjimka, začne systém hledat vhodný handler, který by ji ošetřil. Přitom postupuje podle posloupnosti vnořených pokusných bloků a volání funkcí od místa vzniku výjimky k funkci main. Dochází k tzv. šíření výjimky do nadřazeného bloku resp. do volající funkce.

Při vyvolání výjimky se vytváří datový objekt, který ponese informaci o povaze výjimky a okolnostech jejího vzniku. Často jde o instanci určitého objektového typu. Typ hodnoty, která je posílána handleru při vyvolání výjimky, se nazývá *typ výjimky*.

Vyvolání výjimky se provádí pomocí výrazu throw. Hodnota výrazu přiřazovací\_výraz ponese informaci o výjimce. Typ výjimky je určen typem hodnoty tohoto výrazu. hodnotu tohoto výrazu lze použít v handleru a podle ní se rozhodnout, jak se výjimka ošetří. Z pokusného bloku lze vyskočit příkazem return, break, continue nebo goto.

### Handler

Handler začíná klíčovým slovem catch, za kterým je v závorkách specifikován typ výjimky.

Specifikace typu výjimky se podobá specifikaci jednoho formálního parametru funkce.

Při hledání odpovídajícího handleru se prochází handlersy v pořadí jejich uvedení za složeným příkazem pokusného bloku.

Parametr handleru lze předávat i jako konstantu nebo referenci. Předávání odkazem (referencí) se používá zejména u parametru objektového typu, protože umožňuje používat virtuální metody a nedochází ke zbytečnému kopírování objektu.

Specifikaci typu výjimky v deklaraci handleru lze nahradit výpustkou. Takový handler se nazývá univerzální. Zachytí totiž všechny výjimky jakéhokoliv typu. Musí být proto uveden jako poslední v pokusném bloku.

### Specifikace výjimek

```
void f(); - může se vyvolat jakákoliv výjimka a rozšířit se z ní
void g() throw(); - může vzniknout jakákoliv, ale žádná se nemůže rozšířit
void h() throw(TVyjimka, int); - může vzniknout jakákoliv, ale rozšířit se může pouze TVyjimka a int
typedef void (*tuf)() throw (int); // chyba - nelze použít typedef
int (TA::*ug)() const throw (); - deklarace třídního ukazatele, ze kterého se nesmí rozšířit žádná výjimka
```

Všechny deklarace včetně definice určité funkce nebo metody musí mít specifikovaný stejný seznam výjimek.

### Konstruktory a destruktory

Pokud vznikne výjimka v konstruktoru instance třídy, nebude se pro ni volat destruktory. Budou se ale volat destruktory jejich nestatických složek a předků, které již byly zcela zkonstruovány a nebyla započata jejich destrukce voláním jejich destruktory.

### Neošetřené a neočekávané výjimky

Neošetřená výjimka vznikne pokud:

- výjimku nezachytí žádný handler
- se rozšíří výjimka z destruktory během úklidu zásobníku
- se rozšíří výjimka z konstruktoru nebo destruktory globální instance třídy

V takových případech program zavolá funkci terminate (), která zavolá funkci abort().

Pokud se z nějaké funkce rozšíří výjimka, která není uvedena ve specifikaci Výjimek této funkce, jedná se o neočekávanou výjimku. V případě vzniku se volá void unexpected()

## Vstupy a výstupy v jazyce C a C++ - kategorizace, základní charakteristiky

Jazyk C++ obsahuje všechny nástroje pro vstupní a výstupní operace, které jsou dostupné v jazyku C, a navíc jeho součástí jsou prostředky založené na objektových datových typech.

Pro vstup a výstup se v jazyku C++, ale i v jazyku C používají tzv. datové proudy. Datový proud se stará o přenos dat od zdroje ke spotřebiči. Zdrojem může být program a spotřebičem soubor, obrazovka apod. Součástí proudu bývá zpravidla vyrovnávací paměť. Při přenosu může docházet k transformaci dat, např. k převodu z binární do znakové podoby apod.

### Standardní datové proudy

V jazyce C a C++ existují tři standardní datové proudy:

- `stdin` – slouží pro vstup ze standardního vstupního souboru. Tím je zpravidla konzola počítače, tedy klávesnice. Může být ale přeměřován prostředky operačního systému (z příkazového řádku při spuštění programu)
- `stdout` – je určen pro výstup do standardního výstupu souboru. Tím je zpravidla konzole, tedy obrazovka monitoru. Může být ale přeměřován prostředky operačního systému.
- `stderr` – je určen pro výstup do standardního souboru chyb. Tím je zpravidla opět konzole. Tento proud však nelze pod operačním systémem DOS/Windows přeměřovat.

Všechny tyto proudy se automaticky otevírají při spuštění programu a zavírají při jeho ukončení.

### Objektové datové proudy

Objektová koncepce datových proudů přináší řadu výhod, mezi něž patří možnost definovat vlastní vstupní operátor `>>` a výstupní operátor `<<` pro své vlastní datové typy, definovat vlastní manipulátor apod.

Datové proudy jazyka C++ jsou založeny na dvou hierarchiích objektových typů – `ios_base`.

#### `ios_base`

Třída je definovaná v hlavičkovém souboru `<ios>`. Ve třídě `ios_base` jsou definovány vlastnosti, které jsou společné všem objektovým datovým proudům. Obsahuje definici těchto vnořených typů:

- třída `failure` – odvozená ze třídy `exception`
- tři typy bitových masek – `fmtflags`, `iostate`, `openmode`
- výčtový typ `seekdir`
- třída `Init`

#### `iostate`

obsahuje stavové příznaky

- |                      |   |  |
|----------------------|---|--|
| <code>goodbit</code> | - | stav bez chyby - hodnota 0   |
| <code>badbit</code>  | - | indikuje, že nějaká operace, jiná než vstupní a výstupní, byla neúspěšná |
| <code>eofbit</code>  | - | operace dosáhla konce vstupní sekvence                                   |
| <code>failbit</code> | - | vstupní nebo výstupní operace byla neúspěšná                             |

#### `seekdir`

Příznaky pro přesun ukazatele v proudu:

- |                  |   |  |
|------------------|---|--|
| <code>beg</code> | - | přesun ukazatele a pozici relativní vzhledem k začátku         |
| <code>cut</code> | - | přesun ukazatele a pozici relativní vzhledem k aktuální pozici |
| <code>end</code> | - | přesun ukazatele a pozici relativní vzhledem ke konci          |

Jazyk C a C++

### **Init**

Slouží ke konstrukci a destrukci tříd cout, cin, clog, cerr.

### **Manipulátory**

manipulátory jsou funkce, které lze volat s použitím operátoru >> nebo <<. Např. příkaz

```
cout<<setw(10)<<a;
```

je použit manipulátor setw, který nastaví šířku vyhrazeného prostoru stejně jako kdyby se volala metoda cout.width(10)

manipulátory jsou obyčejné funkce deklarované v různých hlavičkových souborech, v závislosti na tom, pro jaké proudy je lze použít. Mohou být bez parametrů nebo s parametry a lze definovat i vlastní manipulátory

dec, hex, oct, fixed, left, right, boolalpha, setfill, flush

### **Šablona třídy basic\_ifstream**

Slouží pro čtení ze souboru typu FILE.

### **Šablona třídy basic\_ofstream**

Slouží pro zápis do souboru typu FILE.

### **Šablona třídy basic\_ofstream**

slouží pro čtení i zápis

### **Vlastní vstupní a výstupní operátory**

Pro formátované vstupy a výstupy slouží přetížené operátory >> a <<. Pro vestavěné typy jsou definovány jako metody šablon tříd basic\_istream a basic\_ostream. Pro uživatelské typy se musí definovat jako obyčejné funkce nebo jako metody třídy odvozené od šablony basic\_istream resp. basic\_ostream.

Má-li operátor vstupu >> pro typ X fungovat podobně jako vestavěné operátory >>, musí se definovat buď jako šablona obyčejné operátorové funkce nebo jako obyčejná operátorová funkce

## **Generické programování v C++ - principy, koncepty, modely, všeobecné koncepty, iterátory**

Mezi programovací styly patří:

strukturované programování – programování pomocí funkcí, které operují nad daty – neobjektový přístup,

objektově orientované programování,

**generické programování** – je založeno na vytváření abstraktních vzorů funkcí a tříd pomocí generických konstrukcí (šablon).

### *Koncepty a modely*

Jazyk C++ umožňuje definováním šablony funkce popsat velkou množinu funkcí lišících se typem parametrů. Neumožňuje však rozhodování, zda daný typ je vhodný pro příslušnou šablonu.

*Koncept* je tedy množina požadavků na typové parametry generických konstrukcí.

## Jazyk C a C++

*Model konceptu* je typ, který vyhovuje danému konceptu. *Zjemnění konceptu* – koncept  $X$  je zjemněním konceptu  $Y$ , pokud množina požadavků konceptu  $Y$  je podmnožinou požadavků konceptu  $X$ . Koncept  $X$  tedy vyhovuje konceptu  $Y$  a obsahuje navíc další požadavky. Určitý koncept může být zjemněním i několika jiných konceptů. Zjemnění konceptu je podobné dědičnosti tříd.

minimalizovatelné typy (zkr. MT), minimalizovatelné kopírovatelné typy (zkr. MKT)

V knihovně STLport jsou definovány koncepty kontejnerů, iterátorů a dalších objektů.

### *Všeobecné koncepty*

Norma jazyka C++ popisuje všeobecné požadavky na typové parametry šablon. Knihovna STLport pro tyto požadavky zavádí koncepty.

#### **Koncept Assignable**

Typ je modelem konceptu Assignable, pokud je možné kopírovat objekty tohoto typu a přiřadit hodnotu do objektu tohoto typu. Model konceptu musí vyhovovat výrazům uvedeným v následující tabulce.

Výraz	Návratový typ	Po provedení výrazu
$X(x)$	$X$	$X(x)$ je kopií $x$
$x(y); x = y;$		$x$ je kopií $y$
$x = y$	$X\&$	$x$ je kopií $y$

Norma jazyka C++ definuje pro koncept Assignable pouze poslední z uvedených požadavků, tj.  $x = y$ .

#### **Koncept DefaultConstructible**

Typ je modelem konceptu DefaultConstructible, jestliže má implicitní konstruktor, který, jeli to možné, provede konstrukci objektu bez inicializace jeho složek.

Model konceptu musí vyhovovat výrazům uvedeným v následující tabulce.

Výraz	Návratový typ
$X()$	$X$
$X x;$	

#### **Koncept EqualityComparable**

Typ je modelem konceptu EqualityComparable, jestliže objekty tohoto typu lze porovnávat pomocí operátoru  $==$ , který musí splňovat vlastnosti relace rovnosti.

Model konceptu musí vyhovovat výrazům uvedeným v následující tabulce.

Výraz	Návratový typ	Význam
$x == y$	lze implicitně konvertovat na typ bool	
$x != y$	lze implicitně konvertovat na typ bool	ekvivalent výrazu $!(x == y)$ .

Operátor  $!=$  nemusí být v modelu konceptu definován, ale je-li definován, musí být uvedeným ekvivalentem operátoru  $==$ .

Relace  $==$  má vlastnosti uvedené v následující tabulce.

Vlastnost relace $==$	Platí
identita	jestliže $\&x == \&y$ , potom $x == y$
reflexivita	$x == x$
symetrie	jestliže $x == y$ , potom $y == x$
tranzitivita	jestliže $x == y$ a zároveň $y == z$ , potom $x == z$

## **ITERÁTORY**

Iterátor je zevšeobecněním pojmu ukazatel: je to nějaký objekt, který ukazuje na jiný objekt. Slouží zejména k procházení nějakého rozsahu objektů.

## Jazyk C a C++

Iterátory jsou centrální součástí generického programování, protože představují rozhraní mezi kontejnery (např. `vector`) a generickými algoritmy. Tyto algoritmy mají zpravidla jako své parametry iterátory a ne samotné kontejnery, takže kontejnery musí poskytovat přístup ke svým prvkům pomocí iterátorů.

Většina algoritmů pracuje s dvojicí iterátorů, které se zpravidla označují `first` a `last`. Ty představují *rozsah* (angl. *range*) všech iterátorů od `first` do `last` vyjma `last`. Rozsah se označuje intervalem `[first, last)`. Rozsah je prázdný, pokud `first == last`. Rozsah je platný, pokud iterátor `last` je dosažitelný z iterátoru `first`.

### Triviální iterátor

Triviální iterátor může být dereferencován, aby se zpřístupnila reference objektu, na který iterátor „ukazuje“. Může být měnitelný nebo konstantní. Není přímo využit žádným algoritmem. Je základním konceptem jiných iterátorů. Je zjemněním konceptů `DefaultConstructible`, `Assignable`, `EqualityComparable`.

Všechny operace mají konstantní časovou složitost.  
Koncept musí splňovat následující vlastnost:

<b>Vlastnost</b>	<b>Platí</b>
identita	$x == y$ jen v tom případě, pokud platí $\&*x == \&*y$

Modelem triviálního konceptu je např. ukazatel na objekt, který není součástí pole objektů.

### Vstupní iterátor

Vstupní iterátor slouží k procházení prvků kontejneru za účelem čtení objektů obsažených v kontejneru (nemodifikuje objekty kontejneru). Může být:

- dereferencován, aby se zpřístupnila reference objektu, na který iterátor ukazuje,
- inkrementován, aby se přesunul na následující iterátor nějaké posloupnosti.

Je zjemněním triviálního iterátoru. Je konstantní.

Po provedení výrazu `++i` nelze použít kopii staré hodnoty `i`. Pokud platí `i == j`, nemusí platit `++i == ++j`. Algoritmy nesmí dvakrát procházet přes stejný iterátor. Musí se jednat o tzv. jednopřechodové (angl. *single-pass*) algoritmy. Iterátor totiž může při inkrementaci vyjmout prvek z kontejneru, např. ze vstupního datového proudu. Modelem vstupního iterátoru je např. šablona třídy `istream_iterator`. Vstupní iterátor používá např. algoritmus `find`, který provádí sekvenční hledání hodnoty `value` v rozsahu `[first, last)`. Vrací iterátor, který ukazuje na hodnotu `value`. Jestliže hodnotu `value` nenajde, vrací `last`. Algoritmus `find` jakož i další generické algoritmy jsou definovány v hlavičkovém souboru `<algorithm>`.

Vstupním iterátorům vyhovují např. obyčejné ukazatele (jsou modelem iterátoru s náhodným přístupem).

Příklad

V následujícím příkladu se hledá hodnota 7 v poli 10 celých čísel:

```
int pole[10];
// naplnění pole
int* i = find(pole, pole+10, 7);
if (i == pole+10) cout << "Hodnota 7 v poli neexistuje";
else cout << "Hodnota 7 se poli nachází na indexu " << i-pole;
```

Výrazy `pole` a `pole+10` představují měnitelné iterátory. Konstantní iterátory jsou typu ukazatel na konstantní `int`, např.:

Jazyk C a C++

```
const int *first = pole;  
const int *last = pole+10;  
const int *i = find(first, last, 7);
```

Iterátory různých kontejnerů knihovny C++ také vyhovují vstupním iterátorům (vyhovují iterátorům, které jsou zjemněním vstupního iterátoru). Kontejnery obsahují dvě metody begin() a end(). Metoda begin() vrací iterátor, který ukazuje na první prvek kontejneru, metoda end() vrací koncový iterátor.

## **Výstupní, Dopředný, Dvousměrný, S náhodným přístupem,**

### **Třídy iterátorů**

#### **Inverzní iterátor**

Inverzní iterátor (angl. reverse iterator) je tzv. adaptér iterátoru, který prochází prvky kontejneru v opačném pořadí než normální iterátor.

Adaptér (angl. adaptor) je třída nebo funkce, která konvertuje jedno rozhraní na jiné rozhraní. Adaptér iterátoru konvertuje určité rozhraní na rozhraní používané iterátory.

Inverzní iterátor lze použít pro dvousměrný iterátor nebo iterátor s náhodným přístupem. Operátor inkrementace aplikovaný na objekt inverzního iterátoru provede totéž co operátor dekrementace aplikovaný na objekt korespondujícího normálního iterátoru.

Základní vztah mezi inverzním iterátorem a jeho korespondujícím normálním iterátorem  $i$  je dán následující rovností:

$$\&*(reverse\_iterator(i)) == \&(i - 1)$$

Tento vztah je dán faktem, že koncový normální iterátor ukazuje za poslední platný prvek normální posloupnosti, přičemž k němu odpovídající inverzní iterátor musí ukazovat na první platný prvek opačné posloupnosti, tedy ukazuje na poslední platný prvek normální posloupnosti. Naopak k normálnímu iterátoru, který ukazuje na první platný prvek normální posloupnosti, odpovídá inverzní iterátor, který ukazuje za poslední platný prvek opačné posloupnosti, tedy ukazuje před první platný prvek normální posloupnosti.

#### **Vstupní iterátor datového proudu**

Vstupní iterátor datového proudu je šablona třídy istream\_iterator, která čte objekty typu T pomocí operátoru >> ze vstupního datového proudu basic\_istream resp. z jeho potomka.

Šablona je modelem konceptu vstupního iterátoru.

Při každé inkrementaci přečte jeden objekt typu T z datového proudu a uloží jej. Přečtený objekt je přístupný pomocí dereference iterátoru. Jestliže se dosáhne konce datového proudu, iterátor obsahuje speciální hodnotu, která indikuje koncový iterátor.

#### **Výstupní iterátor datového proudu**

Výstupní iterátor datového proudu je šablona třídy ostream\_iterator, která zapisuje objekty typu T pomocí operátoru << do výstupního datového proudu basic\_ostream resp. do jeho potomka. Šablona je modelem konceptu výstupního iterátoru. Za každý zapsaný objekt typu T může být zapsán ještě řetězec znaků, který je parametrem konstruktora.

```
copy(IntList.begin(),IntList.end(),ostream_iterator<int>(cout,""));
```

# Generické programování v C++ - funktory, adaptabilní funktory, adaptéry funktorů, koncepty a modely kontejnerů

Funktor (angl. functor nebo function object) je jakýkoli objekt, který lze použít po vzoru funkce s kulatými závorkami. Funktorem je tedy ukazatel nebo reference na obyčejnou funkci (včetně operátorové funkce) nebo instance třídy, která má přetížený operátor volání funkce. Řada generických algoritmů má jako svůj parametr funktor.

```
template <class InputIterator, class UnaryFunction>
UnaryFunction for_each(InputIterator first, InputIterator last,UnaryFunction f) {
for (; first != last; ++first)
f(*first);
return f;}

```

Algoritmus `for_each` pro každý prvek rozsahu `[first, last)` volá funktor `f`, který nesmí modifikovat prvek rozsahu. V normě C++ je druhý typový parametr této šablony pojmenován `Function`, ale přesnější je pojmenování `UnaryFunction` uvedené v knihovně `STLport`, protože funktor je volán s jedním parametrem.

## *Adaptabilní funktory a adaptéry funktorů*

Adaptabilní funktory jsou funktory, které obsahují deklaraci vnořených typů pro návratový typ a parametry funktoru. Musí obsahovat pouze deklaraci vnořeného typu `result_type`, který představuje návratový typ generátoru.

Adaptabilní unární funkce a adaptabilní predikát musí obsahovat deklaraci dvou vnořených typů:

- `result_type` – návratový typ funktoru,
- `argument_type` – typ parametru funktoru.

Adaptabilní binární funkce a adaptabilní binární predikát musí obsahovat deklaraci tří vnořených typů:

- `result_type` – návratový typ funktoru,
- `first_argument_type` – typ prvního parametru funktoru,
- `second_argument_type` – typ druhého parametru funktoru.

Význam adaptabilních funktorů spočívá v tom, že je mohou využít adaptéry funktorů.

Adaptér funktoru (angl. function object adapter) je adaptabilní funktor, který transformuje rozhraní jiného adaptabilního funktoru. Z toho vyplývá, že lze adaptéry řetězit a určitému algoritmu lze předat jako parametr místo funktoru např. adaptér adaptéru funktoru.

## **Standardní funktory**

pro operátory vestavěných datových typů nelze získat adresu, která by se mohla použít jako ukazatel na funkci při volání nějakého generického algoritmu. Pro tento účel se musejí volat pomocí funktorů. V hlavičkovém souboru `<functional>` je definováno velké množství adaptabilních funktorů, které implementují velké množství aritmetických, relačních nebo logických operátorů. Jsou založeny na binárních a unárních funkcích.

## **Negátory**

adaptér funktoru který neguje výsledek volání adaptabilního predikátu. unární neguje unární predikát a je modelem adaptabilního predikátu, Binární je model adaptabilního binárního predikátu a neguje binární predikát (not1,not2)

## **Vazače**

jsou adaptéry funktorů, které transformují binární funkci na unární tak, že jeden z parametrů binární fce je fixní. jsou modelem adaptabilní unární funkce (binder1st, binder2nd)

## **Adaptéry obyčejných funkcí**

Aby mohla být obyčejná funkce modelem adaptabilního funktoru, musí obsahovat deklarace vnořených typů. To se provede vytvořením adaptéru, který požadované vnořené typy deklaruje.  
pointer\_to\_unary\_function pointer\_to\_binary\_function

## **Adaptéry metod objektových typů**

adaptéry metod umožňují použít na místě volání funktoru volání metody určité třídy. Obsahují třídní ukazatel na určitou metodu, který je inicializován konstruktorem. Operátor volání funkce volá prostřednictvím třídního ukazatele příslušnou metodu, a to pro instanci, která je prvním parametrem operátoru volání funkce. Existuje osm typů těchto adaptérů. Liší se typem prvního parametru operátoru volání funkce.

## **KONCEPTY KONTEJNERŮ**

Kontejner (angl. container) je objekt, který obsahuje jiné objekty, tzv. prvky (angl. elements) a obsahuje metody pro zpřístupnění jeho prvků.

V normě jazyka C++ jsou definovány společné požadavky na všechny kontejnery, které odpovídají konceptu dopředného kontejneru a konceptu DefaultConstructible. Dále jsou v normě definovány požadavky na sekvenci (přibližně odpovídají konceptu Sekvence) a asociativní kontejner.

## **Všeobecné koncepty kontejnerů**

### **Koncept Kontejner**

Základním konceptem kontejnerů je Kontejner (angl. Container). Ostatní koncepty kontejnerů jsou přímo nebo nepřímo jeho zjemněním. Kontejner musí mít mj. sdružený typ iterator, který slouží k procházení prvků kontejneru. Není zaručena neměnnost pořadí prvků kontejneru při jeho iteraci. Není garantováno, že v určitém okamžiku bude aktivní pouze jeden iterátor daného kontejneru. Kontejner vlastní své prvky. Prvky zániku kontejneru zaniknou ukazatele, ale objekty, na které ukazatele ukazují, zaniknout nemusí. Velikost (angl. size) kontejneru je počet prvků kontejneru. Koncept kontejneru je zjemněním konceptu Assignable.

### **Dopředný kontejner**

Dopředný kontejner je kontejner, u kterého se pořadí prvků při iteraci nemění. Tím pádem mohou být dva dopředné kontejnery porovnávány pomocí relačních operátorů ==, !=, <, <=, > a >=.

Dopředný kontejner je tedy zjemněním nejen konceptu Kontejner, ale i konceptů

LessThanComparable a EqualityComparable. Při porovnávání dvou kontejnerů se porovnávají jejich prvky, a proto prvky kontejneru musí být modelem nejen konceptu Assignable, ale i konceptu LessThanComparable a EqualityComparable.

Iterátor dopředného kontejneru musí být modelem dopředného iterátoru nebo jeho zjemnění. Tím pádem může být kontejner používán víceprůchodovými algoritmy. V jednom okamžiku může být aktivních více iterátorů stejného kontejneru.

### **Reverzibilní kontejner**

Reverzibilní kontejner je dopředný kontejner, jehož iterátor je dvousměrný. Modelem tohoto konceptu je kontejner list. Kontejner list je navíc modelem konceptů sekvence s vkládáním na začátek a sekvence s vkládáním na konec.

### **Kontejner s náhodným přístupem**

Kontejner s náhodným přístupem je reverzibilní kontejner, jehož iterátor je iterátor s náhodným přístupem.

Modelem tohoto konceptu jsou kontejnery vector a deque. Kontejner vector je navíc modelem konceptu sekvence s vkládáním na konec a kontejner deque je navíc modelem konceptů sekvence s vkládáním na začátek a sekvence s vkládáním na konec.

## **KONCEPTY KONTEJNERŮ – SEKVENCE**

### **Sekvence**

Sekvence je kontejner, jehož velikost se může měnit a jehož prvky jsou uchovávány v lineárním pořadí. Sekvence je tedy koncept nějaké lineární datové struktury. Je zjemněním konceptu dopředného kontejneru a konceptu DefaultConstructible. Typ prvků sekvence musí být modelem konceptů Assignable, LessThanComparable, EqualityComparable a DefaultConstructible.

### **Sekvence s vkládáním na začátek**

Sekvence s vkládáním na začátek je sekvence, která umožňuje vkládání prvku na začátek kontejneru a odstranění prvního prvku z kontejneru s konstantní časovou složitostí. Modelem tohoto konceptu jsou kontejnery list a deque. Tyto kontejnery jsou navíc modelem dalších konceptů.

### **Sekvence s vkládáním na konec**

Sekvence s vkládáním na konec je sekvence, která umožňuje přidání prvku na konec kontejneru, odstranění posledního prvku z kontejneru a získání hodnoty posledního prvku kontejneru s konstantní časovou složitostí. Modelem tohoto konceptu jsou kontejnery vector, deque a list. Tyto kontejnery jsou modelem i dalších konceptů.

### **Šablona třídy vector**

Šablona třídy vector je sekvence, která má iterátor s náhodným přístupem. I když norma jazyka C++ nspecifikuje vnitřní strukturu kontejneru, jedná se o šablonu abstraktní datové struktury jednorozměrného pole neboli vektoru. Vektor je dynamicky alokovaný a po jeho vytvoření lze do něj prvky přidávat nebo z něj odstraňovat. Vložení a odstranění prvku na konci vektoru má konstantní časovou složitost, zatímco vložení a odstranění prvku v jiném místě vektoru má lineární časovou složitost. Vektor může být alokován na více prvků, než kolik jich právě obsahuje. Celkový počet prvků, které vektor může obsahovat bez potřeby realokace, se nazývá kapacita vektoru. Kapacitu vektoru lze jen zvýšit. Kapacita se nesníží ani při odstranění všech prvků z vektoru. Pokud je kapacita vektoru rovna velikosti vektoru a má se do něj vložit další prvek, kapacita vektoru se určitým způsobem zvýší. Při realokaci vektoru (při změně jeho kapacity) se zneplatní všechny iterátory a ukazatele na prvky vektoru. Šablona je definována v hlavičkovém souboru <vector>.

### **Šablona třídy vector<bool>**

Šablona třídy vector<bool> je parciální specializací šablony třídy vector pro prvky typu bool. Každý prvek tohoto vektoru zabírá jeden bit.

Šablona obsahuje vnořenou třídu reference, která představuje referenci na prvek vektoru.

Konstantní reference je deklarována jako synonymum pro typ bool. Reference má tyto složky:

- neveřejný konstruktor – nelze tedy vytvořit instanci tohoto typu mimo třídu reference (kromě šablony vector, která je přítelem reference),
- operátor přetypování na typ bool,
- operátor přiřazení z hodnoty typu bool,
- kopírovací operátor přiřazení,
- metoda flip() – neguje (překlopí) bit, který je s referencí spjat, tj. z hodnoty 0 vytvoří hodnotu 1 a naopak.

Oproti primární definici šablony vector obsahuje šablona vector<bool> dvě další metody. static void swap(reference x, reference y); Vymění hodnoty (bity) dvou referencí void flip(); Neguje (překlopí) všechny bity (hodnoty prvků) vektoru.

### Šablona třídy deque

Šablona třídy deque představuje obousměrnou frontu. Jméno deque (vyslovuje se stejně jako anglické slovo „deck“) je zkratkou textu „double-ended queue“. Je podobná šabloně třídy vector. Oproti vektoru je navíc modelem sekvence s vkládáním na začátek, tj. lze s konstantní časovou složitostí vložit nebo odstranit prvek na začátku fronty. S konstantní časovou složitostí lze tedy provést i operace vložení a odstranění prvku na konci fronty a operaci náhodného přístupu k prvku. Operace vložení a odstranění prvku uvnitř fronty má lineární časovou složitost. Ačkoli šablona deque provádí s konstantní časovou složitostí stejné operace jako šablona vector, šablona vector danou operaci provede zpravidla rychleji, protože její implementace je jednodušší než šablony deque. Interní reprezentace obousměrné fronty závisí na typu použité knihovny. Může se jednat o jednorozměrné pole s platnými prvky od určitého po určitý index pole. Nebo se může jednat o pole ukazatelů na pole prvků. Všechny iterátory fronty se zneplatní při vložení prvku na jakoukoli pozici do fronty (včetně na začátek a konec) a při odstranění prvku ze středu fronty. Pokud se odstraní prvek ze začátku nebo konce fronty, zneplatní se jen iterátor, který ukazuje na odstraněný prvek. Šablona je definována v hlavičkovém souboru <deque>. Deklarace šablony je následující:

```
template <class T, class Allocator = allocator<T> > class deque;
```

Význam typových parametrů je stejný jako u šablony vector. Narozdíl od vektoru nemá metody capacity a resize, ale jako vektor obsahuje dvě metody assign – jejich význam je stejný jako u vektoru.

### Šablona třídy list

Šablona třídy list je sekvence, která má dvousměrný iterátor. Představuje abstraktní datovou strukturu obousměrného zřetěženého seznamu. Konkrétní typ seznamu záleží na implementaci v dané knihovně. Je modelem konceptu reverzibilního kontejneru, sekvence s vkládáním na začátek a na konec.

Vložení a odstranění prvků z kterékoli pozice v seznamu má konstantní časovou složitost.

Zpřístupnění prvku uvnitř seznamu má lineární časovou složitost. Iterátory seznamu zůstanou platné při vložení prvku na jakoukoli pozici do seznamu. Při odstranění prvku ze seznamu se zneplatní pouze iterátor, který ukazuje na odstraněný prvek. Šablona třídy list je definována v hlavičkovém souboru <list> a její deklarace je následující:

```
template <class T, class Allocator = allocator<T> >  
class list;
```

### **Šablona třídy `list`**

Šablona třídy `list` je sekvence, která má dopředný iterátor. Představuje abstraktní datovou strukturu jednosměrného zřetěženého seznamu. Šablona není součástí normy jazyka C++, je uvedena pouze v knihovně `STLport`. Je modelem konceptu sekvence s vkládáním na začátek. Bližší informace viz nápověda ke knihovně `STLport`.

### **Šablona třídy `basic_string`**

Šablona třídy `basic_string` je sekvence znaků neboli řetězec znaků. Je modelem stejných konceptů jako šablona třídy `vector`, tj. konceptu kontejneru s náhodným přístupem a sekvence s vkládáním na konec. Šablona je definována v hlavičkovém souboru `<string>`.

### **Asociativní kontejner**

Asociativní kontejner je kontejner s proměnlivou velikostí, který umožňuje rychlé hledání prvků podle klíče. Umožňuje vložení a odstranění prvku, ale na rozdíl od sekvence není možné zadat pozici, na kterou se má prvek vložit. Prvky asociativního kontejneru jako u ostatních kontejnerů jsou typu `value_type`. Prvky jsou svázány s klíčem typu `key_type`. U jednoduchých asociativních kontejnerů (koncept Jednoduchý asociativní kontejner a jeho zjemnění) je typ `value_type` totožný s typem `key_type`, tj. hodnota prvku je jeho klíčem. U ostatních asociativních kontejnerů klíč tvoří určitou část hodnoty prvku.

Prvky jsou do kontejneru ukládány podle jejich klíče. Klíče prvků nelze modifikovat. U jednoduchých asociativních kontejnerů jsou tudíž hodnoty prvků konstantní. U ostatních asociativních kontejnerů lze modifikovat tu část hodnoty prvku, která netvoří klíč. Z toho vyplývá, že asociativní kontejnery nemohou mít měnitelné iterátory. Měnitelný iterátor umožňuje přiřazení hodnoty do prvku výrazem `*i = t`, kde `i` je iterátor a `t` objekt typu hodnoty prvku.

U jednoduchých asociativních kontejnerů jsou vnořené typy `iterator` a `const_iterator` totožné. U ostatních asociativních kontejnerů není sice typ `iterator` měnitelným iterátorem (nelze napsat `*i = t`), ale poskytuje mechanismus modifikace části hodnoty prvku, která netvoří klíč, např. výrazem `(*i).second = d`.

Některé asociativní kontejnery mají klíč jedinečný, u jiných kontejnerů může existovat více prvků se stejným klíčem.

Norma jazyka C++ obsahuje pouze koncept Asociativní kontejner, který popisuje vlastnosti všech konceptů asociativních kontejnerů kromě hašovacích. Modely hašovacích kontejnerů nejsou v normě jazyka C++ obsaženy. Koncept Asociativní kontejner je zjemněním konceptů Dopředný kontejner a `DefaultConstructible`.

Prvky se stejným klíčem jsou při procházení kontejneru pomocí iterátorů za sebou. To znamená, že jestliže iterátory `p` a `q` ukazují na prvky se stejným klíčem a jestliže iterátor `p` předchází iterátoru `q`, potom každý prvek rozsahu `[p, q)` má stejný klíč.

### **Jednoduchý asociativní kontejner**

Jednoduchý asociativní kontejner je kontejner s prvky, jejichž hodnota tvoří jejich klíč. Je zjemněním konceptu asociativního kontejneru. Vnořený typ `key_type` je totožný s typem `value_type`. Kontejner nemá měnitelný iterátor, má jen konstantní iterátor, tudíž typy `iterator` a `const_iterator` jsou stejné. Hodnoty prvků kontejneru jsou konstantní, nelze je modifikovat. Lze je pouze vkládat a odstraňovat. Důvodem je skutečnost, že hodnota prvku je zároveň jeho klíč. Modelem konceptu jednoduchého asociativního kontejneru jsou kontejnery `set`, `multiset`, `hash_set` a `hash_multiset`, které jsou modelem i jiných konceptů.

### **Párový asociativní kontejner**

Párový asociativní kontejner je kontejner s prvky, jejichž hodnota se skládá ze dvou částí: datové složky a klíče. Je zjemněním asociativního kontejneru.

Kromě vnořených typů požadovaných v konceptech, kterého je zjemněním, musí mít deklarován vnořený typ `mapped_type` a specifikuje další požadavky na typy `key_type` a `value_type`. Modelem konceptu párového asociativního kontejneru jsou kontejnery `map`, `multimap`, `hash_map` a `hash_multimap`, které jsou modelem i jiných konceptů.

### **Tříděný asociativní kontejner**

Tříděný asociativní kontejner má klíče prvků utříděné vzestupně. Je zjemněním konceptů `reversible_container` a `associative_container`.

Většina operací v tříděném asociativním kontejneru má maximálně logaritmickou časovou složitost.

### **Jednoznačný asociativní kontejner**

Jednoznačný asociativní kontejner je kontejner s prvky, jejichž klíče jsou jedinečné. Nemůže v něm existovat více prvků se stejným klíčem. Je zjemněním konceptu asociativního kontejneru.

### **Víceznačný asociativní kontejner**

Víceznačný asociativní kontejner je kontejner, ve kterém se může vyskytovat více prvků se stejným klíčem. Je zjemněním konceptu asociativního kontejneru.

### **Jednoznačný tříděný asociativní kontejner**

Jednoznačný tříděný asociativní kontejner je tříděný kontejner s prvky, jejichž klíče jsou jedinečné. Nemůže v něm existovat více prvků s ekvivalentním klíčem. Je zjemněním konceptu jednoznačného asociativního kontejneru a tříděného asociativního kontejneru. Prvky v jednoznačném tříděném asociativním kontejneru jsou uspořádány podle svých klíčů vzestupně. To znamená, že jestliže iterátor `i` předchází iterátoru `j`, potom výraz `a.value_comp>(*i, *j)` je vždy pravdivý. Modelem konceptu jednoznačného tříděného asociativního kontejneru jsou kontejnery `set` a `map`, které jsou modelem i jiných konceptů.

### **Víceznačný tříděný asociativní kontejner**

Víceznačný tříděný asociativní kontejner je tříděný kontejner, v němž může existovat více prvků s ekvivalentním klíčem. Je zjemněním konceptu víceznačného asociativního kontejneru a tříděného asociativního kontejneru. Modelem konceptu víceznačného tříděného asociativního kontejneru jsou kontejnery `multiset` a `multimap`, které jsou modelem i jiných konceptů.

### **Šablona třídy `set`**

Šablona třídy `set` je asociativní kontejner, který má dvousměrný iterátor. Je modelem konceptů jednoduchého asociativního kontejneru a jednoznačného tříděného asociativního kontejneru. To znamená, že objekty ukládané do kontejneru jsou zároveň klíčem, klíče prvků jsou utříděné vzestupně a v kontejneru neexistují dva prvky s ekvivalentním klíčem.

Při vložení prvku do šablony `set` zůstanou iterátory této šablony platné. Při odstranění prvku ze šablony se zneplatní pouze iterátor, který ukazuje na odstraněný prvek.

Anglické slovo „set“ lze přeložit do češtiny jako „množina“. Jedná se tedy o kontejner obsahující množinu rozdílných prvků. Interně je implementována jako určitý typ binárního stromu. Šablona je definována v hlavičkovém souboru `<set>`.

### **Šablona třídy multiset**

Šablona třídy multiset je asociativní kontejner, který má dvousměrný iterátor. Je modelem konceptů jednoduchého asociativního kontejneru a víceznačného tříděného asociativního kontejneru. To znamená, že objekty ukládané do kontejneru jsou zároveň klíčem, klíče prvků jsou utříděné vzestupně a v kontejneru může existovat více prvků s ekvivalentním klíčem.

Při vložení prvku do šablony multiset zůstanou iterátory této šablony platné. Při odstranění prvku ze šablony se zneplatní pouze iterátor, který ukazuje na odstraněný prvek. Interně je implementována jako určitý typ binárního stromu. Šablona je definována v hlavičkovém souboru <set>.

### **Šablona třídy map**

Šablona třídy map je asociativní kontejner, který má dvousměrný iterátor. Je modelem konceptů párového asociativního kontejneru a jednoznačného tříděného asociativního kontejneru. To znamená, že objekty ukládané do kontejneru se skládají z datové složky a klíče. Prvky jsou utříděné podle svých klíčů vzestupně a v kontejneru neexistují dva prvky s ekvivalentním klíčem.

Při vložení prvku do šablony map zůstanou iterátory této šablony platné. Při odstranění prvku ze šablony se zneplatní pouze iterátor, který ukazuje na odstraněný prvek. Interně je implementována jako určitý typ binárního stromu. Šablona je definována v hlavičkovém souboru <map>.

### **Šablona třídy multimap**

Šablona třídy multimap je asociativní kontejner, který má dvousměrný iterátor. Je modelem konceptů párového asociativního kontejneru a víceznačného tříděného asociativního kontejneru. To znamená, že objekty ukládané do kontejneru se skládají z datové složky a klíče. Prvky jsou utříděné podle svých klíčů vzestupně a v kontejneru může existovat více prvků s ekvivalentním klíčem.

Při vložení prvku do šablony multimap zůstanou iterátory této šablony platné. Při odstranění prvku ze šablony se zneplatní pouze iterátor, který ukazuje na odstraněný prvek. Interně je implementována jako určitý typ binárního stromu. Šablona je definována v hlavičkovém souboru <map>.