C programos struktūra ir funkcijos

1. C programą sudaro 1 arba daugiau failų (arba kompiliavimo vienetų).

   programa ::= *{ failas }* $_{1+}$

2. C programos failas – tai vienas paskui kitą einantys kintamųjų, funkcijų ir tipų apibrėžimai ir/arba aprašai.

3. Taigi, C programą sudaro 1 arba daugiau funkcijų, esančių (patalpintų į) 1 arba daugiau failų.

4. Visos funkcijos yra tame pačiame (išoriniame) lygyje, t.y. jos negali būti įdėtos viena į kitą.

5. Viena ir tik viena funkcija privalo būti funkcija vardu **`main()`**, kuri ir yra įėjimo į programą taškas. Kitos funkcijos yra iškviečiamos arba iš šios funkcijos arba iš kitų.

6. Funkcijos apibrėžimas susideda iš antraštės ir kūno. Antraštę sudaro funkcijos grąžinamas tipas, vardas bei apskliaustas formalių parametrų sąrašas. Funkcijos kūnas yra blokas. Sintaksiškai tai sudėtinis sakinys.

   funkcijos_apibrėžimas ::= *tipas*$_{opc}$ *vardas* **(** *parametrų_sąrašas*$_{opc}$ **)** *sudėtinis_sakinys*

7. Parametrų sąraše parametrų tipų ir jų vardų poros atskiriamos kableliais.

   parametrų_sąrašas ::= *tipas vardas {*, *tipas vardas }* $_{0+}$

8. Jei funkcija neturi parametrų, parametrų sąraše nurodomas tipas **`void`**.

9. Jei funkcija negrąžina reikšmės (t.y. turime procedūrą), tai kaip grąžinamas nurodomas tipas **`void`**.

10. Sudėtinio sakino pradžioje apibrėžiami lokalūs kintamieji, o paskui rašomi sakiniai. C++

    sudėtinis_sakinys ::= **{** *kintamųjų_apibrėžimai*$_{opc}$ *sakiniai*$_{opc}$ **}**

11. Kadangi sudėtinis sakinys sintaksiškai irgi sakinys, tai galimas sudėtinių sakinių (t.y. blokų) įdėjimas.

12. Funkcijos reikšmė grąžinama naudojant **`return`** sakinį.

13. Kreipinyje į funkciją nurodomas funkcijos vardas ir apskliaustas faktinių parametrų (argumentų) sąrašas. Jei funkcija ir neturi parametrų, kreipinyje į funkciją privalu rašyti skliaustų porą.

14. Prieš kreipiantis į funkciją ji turi būti mažiausiai aprašyta. Jei funkcijos apibrėžimas (t.y. kartu ir aprašas) yra kitame faile arba tame pačiame faile, bet vėliau už kreipinį į funkciją, būtina parūpinti funkcijos aprašą, kuris dar vadinamas funkcijos prototipu.

    funkcijos_aprašas ::= *tipas*$_{opc}$ *vardas* **(** *parametrų_sąrašas*$_{opc}$ **);**

15. Funkcijos prototipe parametrų vardai ignoruojami, todėl yra nebūtini. Kompiliatorius kontroliuoja funkcijos formalių ir faktinių parametrų tipų atitikimą.

16. Argumentai į funkciją perduodami ‘”pagal reikšmę”. Visi kiti argumentų perdavimo mechanizmai yra imituojami šio mechanizmo pagalba. C++ turi perdavimo ‘”pagal nuorodą” mechanizmą.

Baziniai C ir C++ tipai

Rūšis	*Tipas*	*C89* *C99*	*C++*	*Galimi ekvivalentai*
Loginis	**bool**		✓	
	_Bool	✓		
Tuščias	**void**	✓	✓	
Simboliniai	**char**	✓	✓	signed char *arba* unsigned char
	signed char	✓	✓	
	unsigned char	✓	✓	
	wchar_t		✓	
Sveikieji	short	✓	✓	short int signed short signed short int
	unsigned short	✓	✓	unsigned short int
	int	✓	✓	signed signed int
	unsigned	✓	✓	unsigned int
	long	✓	✓	long int signed long signed long int
	unsigned long	✓	✓	unsigned long int
	long long	✓		long long int signed long long signed long long int
	unsigned long long	✓		unsigned long long int
Realūs	**float**	✓	✓	
	double	✓	✓	
	long double	✓	✓	
	_Complex	✓		
	_Imaginary	✓		

C ir C++ operacijos

Nr.	Operacija	C	C++	Prasmė
1	`::`		✓	Globalaus konteksto
	`::`		✓	Klasės arba vardų erdvės konteksto
2	`()`	✓	✓	Kreipinio į funkciją
	`[]`	✓	✓	Masyvo indekso
	`.` `->`	✓	✓	Struktūros (klasės) nario priėjimo turint struktūros kintamąjį arba rodyklę į struktūrą
	`++` `--`	✓	✓	Aritmetinės : postfiksinė inkremento, postfiksinė dekremento
	6 C++ operacijos		✓	Tipo konvertavimo ir nustatymo
3	`+` `-`	✓	✓	Aritmetinės : unarinio pliuso ir minuso
	`++` `--`	✓	✓	Aritmetinės : prefiksinė inkremento, prefiksinė dekremento
	`&`	✓	✓	Adreso
	`*`	✓	✓	Rodyklės
	`!`	✓	✓	Loginė : neiginio
	`~`	✓	✓	Bitinė : neiginio
	`sizeof`	✓	✓	Tipo (reiškinio tipo) dydžio (baitais)
	`(`tipas`)`	✓	✓	Tipo konvertavimo
	`new` `delete`		✓	Dinaminės atminties paskyrimo, panaikinimo
4	`.*` `->*`		✓	Klasės nario priėjimo per rodyklę į klasės narį, turint klasės kintamąjį (objektą) arba rodyklę į klasę
5	`*` `/` `%`	✓	✓	Aritmetinės : daugybos, dalybos, modulio
6	`+` `-`	✓	✓	Aritmetinės : sumos, atimties
7	`>>` `<<`	✓	✓	Bitinės : postūmio į dešinę, į kairę
8	`<` `<=` `>` `=>`	✓	✓	Santykio: mažiau, mažiau arba lygu, daugiau, daugiau arba lygu

9	`==` `!=`	✓	✓	Lygybės : lygu, nelygu
10	`&`	✓	✓	Bitinė : IR
11	`^`	✓	✓	Bitinė : griežtas ARBA
12	`|`	✓	✓	Bitinė : ARBA
13	`&&`	✓	✓	Loginė : IR
14	`||`	✓	✓	Loginė : ARBA
15	`?:`	✓	✓	Sąlygos
16	`=` `+=` `-=` `*=` `/=` `%=` `>>=` `<<=` `&=` `^=` `|=`	✓	✓	Priskyrimo : paprastoji, sudėtinės
17	`throw`		✓	Išskirtinių situacijų apdorojimo
18	`,`	✓	✓	Kablelio

C ir C++ sakiniai

Rūšis	***Sakinys***	***Pastabos***
Tuščias	`;`	Naudojamas, kai to reikalauja sintaksė, bet semantiškai nereikalingas
Reiškinio	*reiškinys*`;`	*reiškinys* - bet koks C/C++ reiškinys. C turi priskyrimo reiškinio sakinį. Paskalis turi priskyrimo sakinį.
Sudėtinis	**{** *sakiniai*$_{op}$ **}**	Sudėtinis sakinys gali būti naudojamas ten, kur naudojamas bet kuris sakinys *sakinys*
Sąlygos	**if (***reiškinys***)** *sakinys*	*reiškinys* - paprastai sąlygos reiškinys, panaudojant lygybės, santykio, logines operacijas. Tačiau gali būti bet kokiu leistinu C/C++ reiškiniu. Jei reiškinys lygus 0 - false. Jei reiškinys nelygus 0 - true.

	if (*reiškinys***)**
	  sakinys
	else
	  sakinys	

	switch (*i-reiškinys***) {**
	  case *ik-reiškinys-1*: *sakiniai*$_{op}$
	  …
	  case *ik-reiškinys-n*: *sakiniai*$_{op}$
	  default: *sakiniai*$_{op}$
	}
		i-reiškinys
		- sveikas reiškinys
		
		ik-reiškinys-n
		- sveikas konstantinis reiškinys

Ciklo	**for (***reiškinys*$_{op}$`;`*reiškinys*$_{op}$`;`*reiškinys*$_{op}$**)** *sakinys*	*reiškinys* (pirmas) - inicijavimo (prieš ciklą) *reiškinys* (antras) - sąlygos (prieš iteraciją) *reiškinys* (trečias) - veiksmo (po iteracijos)
	while (*reiškinys***)** *sakinys*	Jei sąlyga (*reiškinys*) teisinga, vykdoma sekanti iteracija (*sakinys*)
	do *sakinys* **while (***reiškinys***);**	Vykdoma (*sakinys*). Jei sąlyga (*reiškinys*) teisinga, vykdoma sekanti iteracija (*sakinys*)
Šuolio	**return** *reiškinys*`;`	Grąžina funkcijos reikšmę
	break;	Nutraukia ciklo ir `switch` sakinių vykdymą
	continue;	Nutraukia ciklo sakinių iteracijos vykdymą
	goto *žymė*`;`	
Žymės	*žymė*`:`	
Aprašo	*aprašas*`;`	

C įvedimo-išvedimo pagrindai

Pagrindinės simbolių skaitymo-rašymo (įvedimo-išvedimo) funkcijos	
`int getchar(void) ;`	Skaito simbolį iš standartinio įvedimo srauto – ekvivalentiška getc(stdin).
`int putchar(int ch);`	Rašo simbolį į standartinį išvedimo srautą – ekvivalentiška putc(ch, stdout).
`char *gets(char*);`	Skaito eilutę iš standartinio įvedimo srauto. Dėmesio! Neturi parametro, nurodančio buferio dydį (žr fgets()).
`int puts(const char*);`	Įrašo simbolį į standartinį išvedimo srautą ir pabaigoje prideda naujos eilutės simbolį.
`int getc(FILE*);`	Skaito simbolį iš srauto (failo). Gali būti realizuotas kaip makro.
`int putc(int, FILE*);`	Rašo simbolį į srautą (failą). Gali būti realizuotas kaip makro.
`int fgetc(FILE*);`	Skaito simbolį iš srauto (failo). Visuomet realizuojamas kaip funkcija.
`int fputc(int, FILE*);`	Rašo simbolį į srautą (failą). Visuomet realizuojamas kaip funkcija.
`char *fgets(char *, int, FILE*);`	Skaito eilutę iš srauto (failo).
`int fputs(const char*, FILE*);`	Rašo eilutę į srautą (failą).
`int ungetc(int, FILE*);`	Grąžina simbolį atgal į srautą (failą).

Formatuoto skaitymo-rašymo (įvedimo-išvedimo) funkcijos	
`printf(const char*, ...);`	Formatuotas rašymas į standartinį išvedimo srautą. Formatavimo galimybes žr. dokumentacijoje.
`scanf(const char*, ...);`	Formatuotas skaitymas iš standartinio įvedimo srauto.
`fprintf(FILE*, const char*, ...);`	Formatuotas rašymas į srautą (failą).
`fscanf(FILE*, const char*, ...);`	Formatuotas skaitymas iš srauto (failo).
`sprintf(char*, const char*, ...);`	Formatuotas rašymas į eilutę.
`sscanf(const char*,const char*,...);`	Formatuotas skaitymas iš eilutės.

Pagrindinės darbo su failais funkcijos		
`FILE* fopen(const char*, const char*);` Antras parametras nurodo failo atidarymo režimą (būdą)		Srauto (failo) atidarymas.
	`"r"`	Skaitymui iš tekstinio failo. Galima `"rt"`.
	`"w"`	Rašymui į tekstinį failą. Galima `"wt"`.
	`"a`	Tekstinio failo papildymui (*append)* gale.
	`"rb"`	Skaitymui iš dvejetainio failo.
	`"wb"`	Rašymui į dvejetainį failą.
	`"ab`	Dvejetainio failo papildymui.
Visos kombinacijos su "+" ir skaitymui, ir rašymui. Ši opcija dar vadinama atnaujinimo (*update*), pvz.	`"r+"`	Skaitymui ir rašymui į tekstinį failą. Failas turi egzistuoti.
	`"w+`	Rašymui ir skaitymui ir į tekstinį failą. Failas arba sukuriamas, arba jei yra - perrašomas .
`int fclose(FILE*);`		Srauto (failo) uždarymas.
`int fflush(FILE*);`		Srauto (failo) išvalymas.

Rodyklės (I). Įvadas

1. Rodyklės (angl. *pointers*) – tai kintamieji, kurių reikšmės yra atminties adresai.
2. Rodyklės apibrėžiamos:

   ```
   tipas * rodykės_vardas;
   ```

   Rodyklės deklaratorius (simbolis), kaip ir visi C aprašų deklaratoriai, skirtingai nuo Paskalio, siejami su kintamojo vardu, o ne su tipu, pvz.:

   ```
   char *ptr1, *ptr2;
   ```

3. Rodyklės yra tipizuotos - jų apibrėžime nusakoma, kokio tipo objektų adresus jos turi saugoti, pvz. rodyklė į `char` tipą. Netipizuotoms (apibendrintoms) rodyklėms apibrėžti naudojama:

   ```
   void *ptr;
   ```

   Rodyklių tipizavimo savybė riboja (C - pusiau griežtai, C++ - griežtai) netiesioginį tipizuotų rodyklių konvertavimą iš vieno tipo į kitą, pvz.:

   ```
   char *ptrc; int *ptri;
   ptrc=ptri; / *C – perspėjimas arba klaida, C++ - klaida */
   ```

   Tiesa, kiek kitaip yra konvertuojant į(iš) netipizuotas(-ų) rodykles(-ių), pvz.:

   ```
   void *ptr; char *ptrc;
   ptr=ptrc; / *C – gerai, C++ - gerai */
   ptrc=ptr; / *C – gerai, C++ - klaida */
   ```

   Programų pernešamumo (portabilumo) tikslais, C programose **rekomenduojama** (C++ to reikalauja) naudoti tiesioginio tipų konvertavimo operaciją **(***tipas***)**, pvz.:

   ```
   ptrc= (char*) ptri;
   ptrc= (char*) ptr;
   ```

4. Rodyklės yra pats nesaugiausias ir pavojingiausias C mechanizmas. Nors rodyklių apibrėžimas nereikalauja jų inicijavimo, tačiau jas **privalu korektiškai inicijuoti** iki jų panaudojimo. Rodyklės inicijuojamos (arba joms priskiriami) kintamųjų adresai, naudojant adreso operaciją **&**, pvz.:

   ```
   int i,*ptri=&i;
   ```

   arba

   ```
   int *ptri,i;
   ptri=&i;
   ```

   Saugus rodyklių inicijavimas “niekuo” C garantuojamas `NULL` manifestu (C++ - `0`).

   ```
   int *ptri=NULL;
   ```

5. Tiesioginis priėjimo prie objektų (duomenų, kintamųjų, funkcijų) būdas yra nurodyti objekto vardą. Rodyklės parūpina netiesioginį priėjimo prie objektų būdą - per jų adresą, saugomą (tiksliau - gaunamą) rodyklės tipo kintamajame (tiksliau - reiškinyje). Tuo tikslu rodyklės tipo reiškiniui pritaikome rodyklės (arba išadresavimo) operaciją ***** .Pvz. šie veiksmai yra ekvivalentiški:

   ```
   i = 7; *ptri = 7; *&i = 7;
   ```

6. Rodykliniai tipai yra išvestiniai tipai. Jei galima “išvesti” rodyklinį tipą iš bazinio, tai kodėl negalima to padaryti iš kito išvestinio tipo? Todėl prasminga apibrėžti ir manipuliuoti tokių tipų kintamaisiais, pvz.: rodyklė į rodyklę į `char`

   ```
   char c, *ptrc = &c, **ptrptrc = &ptrc;
   ```

   Šie veiksmai yra ekvivalentiški

   ```
   c='A'; *ptrc = 'A'; **ptrptrc = 'A';
   ```

Rodyklės (II). Parametrų perdavimo mechanizmai

1. C realizuoja vienintelį parametrų perdavimo mechanizmą - pagal reikšmę (angl. *by value*; Paskalyje parametrai-reikšmės).
   Šis perdavimo būdas pasižymi tuo, kad į funkciją yra perduodamos argumentų kopijos Tiksliau būtų sakyti, kad funkcijos lokalūs kintamieji (t.y. formalūs parametrai) yra inicijuojami argumentų (faktinių parametrų) reikšmėmis. Todėl naudojant šį mechanizmą patys argumentai funkcijos aplinkoje nėra keičiami.

2. Kitas (plačiąja informatikos prasme kiti) perdavimo mechanizmas - pagal nuorodą (angl. *by reference*; Paskalyje parametrai-kintamieji) gali būti realizuojami (imituojami) parametrų perdavimo pagal reikšmę mechanizmo pagalba.

3. Techniškai argumentų perdavimo pagal nuorodą imitavimą, turint perdavimą pagal reikšmę, galima išskaidyti į 3 žingsnius:

   1) parametrų-reikšmių tipus padarome rodykliniais (tiksliau - pridedam vieną rodyklės deklaratorių), taip paruošdami juos adresinėms reikšmėms (tiksliau – pasakydami, kad parametrai manipuliuos su to tipo kintamųjų adresais);

   2) visur kur naudojami parametrai-reikšmės pritaikome rodyklės (išadresavimo) operaciją;

   3) kreipinyje į funkciją kaip argumentus perduodame kintamųjų adresus.

Perdavimas pagal reikšmę – čia netinkamas	Imituojamas perdavimas pagal nuorodą	C++ perdavimas pagal nuorodą

`void swap_blogas(int x,int y) {`
`  int temp;`
`  temp=x;`
`  x=y;`
`  y=temp;`
`}`
`int main() {`
`  int a=1, b=2;`
`  swap_blogas(a,b);`
`  /* a=1,b=2 */`
`...`
`}`
	`void swap(int* x,int* y) {`
	`  int temp;`
	`  temp=*x;`
	`  *x=*y;`
	`  *y=temp;`
	`}`
	`int main() {`
	`  int a=1, b=2;`
	`  swap(&a,&b);`
	`  /* a=2,b=1 */`
	`...`
	`}`
		`void swap_cpp(int& x,int& y) {`
		`  int temp;`
		`  temp=x;`
		`  x=y;`
		`  y=temp;`
		`}`
		`int main() {`
		`  int a=1, b=2;`
		`  swap_cpp(a,b);`
		`  /* a=2,b=1 */`
		`...`
		`}`

4. C++ turi tikrąjį parametrų perdavimo pagal nuorodą mechanizmą.
   Jis realizuojamas pasitelkiant naują C++ savybę (konstrukciją) – nuorodas (angl. *references*), kurias apibendrintai galima vadinti kintamųjų (objektų) pseudonimais (angl. *aliases*).

Rodyklės (III). Rodyklių aritmetika

1. Vienas svarbiausių rodyklių privalumų yra tas, kad su rodyklėmis (t.y. su rodyklinių tipų kintamaisiais ir reiškiniais) gali būti atliekami kai kurie aritmetiniai veiksmai. Todėl naudojamas terminas rodyklių aritmetika. Pagrindinis šios aritmetikos ypatumas yra tas, kad joje vienetu laikomas ne sveikasis skaičius 1, o rodyklės rodomojo tipo duomens atmintyje užimamas baitų skaičius.

2. Leistinos (o kartu ir prasmingos) rodyklių aritmetikos operacijos:

   2.1. Prie rodyklės galima pridėti/atimti sveiką skaičių. Tokio reiškinio rezultatas yra taip pat rodyklės tipo, o jo skaitinė reikšmė padidinama/sumažinama dydžiu lygiu tas sveikas skaičius kart rodomojo tipo duomens užimamas baitų skaičius, pvz.

   ```
   int i1=1, i2=2, *ptri=&i1;
   ```

   Tarkime, int duomenys užima 4 baitus, o rodyklės ptri reikšmė po inicijavimo yra adresas `1000`. Tuomet reiškinio `ptri+1` reikšmė lygi `1004`. (nors, aišku, ptri reikšmė ir toliau lygi `1000`). Techniškai šią operaciją galima interpretuoti kaip adreso postūmį per antrajame operande nurodytą rodomojo tipo duomenų skaičių.

   Svarbu pastebėti, kad šiam reiškiniui galima taikyti rodyklės (išadresavimo) operaciją `*(ptri+1)`. Ši operacija realizuoja netiesioginį kreipinį į int duomenį, esantį adresu `ptri+1`.

   2.2. Kadangi rodyklei-kintamajam galima priskirti to paties rodyklės tipo reiškinį, tai yra leistinos operacijos `++`, `--`, o taip pat `+=` ir `-=`, jei antras operandas sveiko tipo. Pvz.: po `ptri++;` šio kintamojo reikšmė bus lygi `1004`.

   Verta panagrinėti sakinius

   ```
   *ptri++; (*ptri)++;
   ```

   Pirmame sakinyje pagal operacijų prioritetus postfiksinio inkrementavimo operacija taikoma rodyklei ptri , o ne jos rodomam int duomeniui. Todėl ji ir bus didinama atlikus visus šio sakinio veiksmus. Taigi, viso reiškinio reikšmė po sakinio įvykdymo yra lygi rodyklės rodomo duomens reikšmei (t.y. 1), o pati rodyklė bus lygi `1004`.

   Antrame sakinyje skliaustų pagalba sukeista operacijų taikymo tvarka, t.y. postfiksinio inkrementavimo operacija bus taikoma rodyklės rodomam duomeniui (t.y. `*ptri`). Todėl jis ir bus didinamas atlikus visus šio sakinio veiksmus. Taigi, viso reiškinio reikšmė po sakinio įvykdymo yra lygi rodyklės rodomo duomens reikšmei padidintai vienetu (t.y. 2), o pati rodyklė liks lygi `1000`.

   2.3. Dviejų to paties tipo rodyklių skirtumas, pvz.: `ptri2-ptri1` , kurio rezultatas yra sveiko tipo ir lygus rodomojo tipo duomenų, kuriuos galima būtų patalpinti tarp dviejų adresų, skaičiui (teigiamam arba neigiamam). Paprastai taikomas apribojimas, kad abi rodyklės rodytų į to paties masyvo elementus.

   2.4. Dviejų to paties tipo rodyklių palyginimas, naudojant lygybės `==` ir `!=` operacijas ir santykio `<` , `<=` , `>` , `>=` operacijas. Taip pat paprastai taikomas apribojimas, kad abi rodyklės rodytų į to paties masyvo elementus.

3. Neleistinos rodyklių aritmetikos operacijos:

   3.1. Rodyklių negalima sudėti, pvz.: `ptri2+ptri1`.

   3.2. Su rodyklės negalima atlikti kitų “elementarių” aritmetinių veiksmų, t.y. padauginti ar padalinti iš sveiko skaičiaus ar kitos rodyklės ir t.t.

Rodyklės (IV). Konstantos ir rodyklės

1. Konstantos (angl. *constants*) - tai nekeičiami (nekintantys) programos duomenys. C ir C++ konstantas realizuoja skirtingai. Pagrindinis skirtumas tas, kad C prasme - tai "kintamieji, kurių negalima pakeisti", o C++ - tai tiesiog pastovūs duomenys. Techniškai tariant, C konstantoms visada išskiriama atmintis, o C++ - nebūtinai. Antra, pagal nutylėjimą C konstantos turi išorinį surišimą (žr. Programos objektų surišimas), o C++ - vidinį. Kitaip tariant, C konstantos turi programos matomumą, o C++ - failo. Plačiau apie tai galima paskaityti puikios B.Eckel knygos "Thinking in C++" puikioje 8 dalyje "Constants".

1.1. Konstantos apibrėžiamos rezervuoto kalbos žodžio **const** pagalba. Formali konstantų apibrėžimo sintaksė yra komplikuota. Pvz. bazinių tipų konstantas galima apibrėžti 2 būdais:

```
const int ci1, ci2;     /* arba int const ci3, ci4; */
                        /* bet ne int const ci5, const ci6; */
```

1.2. Nepaisant kai kurių subtilumų ir skirtumų, galima laikytis taisyklės, kad C ir C++ (ypatingai C++) konstantos turi būti inicijuojamos apibrėžimo metu (C++ atveju aukščiau pateikti 2 apibrėžimo būdai be inicijavimo bus tikrai klaidingi), pvz.:

```
const int ci1=1, ci2=2;
```

1.3. Kaip minėta, konstantų negalima keisti, t.y. neleistini veiksmai `ci1=2` , `ci1++` ir t.t.

2. Konstantos (t.y. pastovūs duomenys arba kintamieji) gali būti ne tik bazinių, bet ir išvestinių tipų. Ypatingo dėmesio vertas rodyklių ir konstantų santykis (plačiąją prasme - rodyklių ir konstantiškumo aspektas), nes jis pateikia papildomas tipizavimo galimybes.

2.1. Bandant apibrėžime panaudoti **const** ir ***** specifikatorius "netikėtai" iškyla klausimas. Taip, bet kokiu atveju apibrėžiame rodyklę, tačiau kam pritaikyti konstantiškumo požymį: pačiai rodyklei (t.y. taip apibrėžtume konstantinę rodyklę į įprastąjį rodomojo tipo duomenį) ar rodyklės rodomajam duomeniui (t.y. taip apibrėžtume įprastą rodyklę į konstantinį rodomojo tipo duomenį)? Juk abiejų objektų apibrėžimas yra prasmingas. Sintaksiškai pirmą ir antrą kintamąjį apibrėžiame atitinkamai pvz.:

```
int * const cpi;
const int * pci; /* arba int const * pci; */
```

2.2. Antrasis yra ir ypatingai svarbus. Pasirodo, kad tokios rodyklės tipas yra nesuderinamas (C – pusiau griežtai, C++ - griežtai) su įprastos rodyklės į įprastą rodomojo tipo duomenį tipu, t.y.

```
int * pi;
```

Tokį nesuderinamumą griežtai tipizuotose kalbose sąlygoja būtinumas apsaugoti konstantinius objektus nepriklausomai nuo to, ar jie pasiekiami tiesiogiai, ar netiesiogiai (t.y. rodyklės pagalba). Tačiau toks apsaugojimas yra prasmingas tik į vieną pusę. Rodyklė į konstantinį duomenį negali būti (netiesiogiai) konvertuota į rodyklę į įprastą to paties tipo duomenį, tačiau atvirkštinis (netiesioginis) konvertavimas leidžiamas, nes "nėra pavojaus, jei apsaugosime tai, ko nereiktų apsaugoti", tuo tarpu "pavojinga (neleistina) neapsaugoti to, ką reikia apsaugoti" t.y.

```
pi = pci; /* C:Warning,C++:Error */
pci = pi; /* OK */
```

2.3. Galima apibrėžti ir konstantinę rodyklę į konstantinį duomenį, pvz.:

```
const int * const cpci; /* arba int const * const cpci;*/
```

2.4. Tiek rodykles, tiek konstantas arba rekomenduojama, arba privalu apibrėžiant inicijuoti. Medžiagos įtvirtinimui siūloma savarankiškai pasiaiškinti pavyzdį:

```
int i; const int ci=1;
int * pi1=&i, * const cpi1=&i;                  /* OK */
int * pi2=&ci, * const cpi2=&ci;                /* C:Warning,C++:Error */
const int * pci1=&ci, * const cpci1=&ci;        /* OK */
const int * pci2=&i, * const cpci2=&i;          /* OK */
```

2.5. Taigi, rodyklių (tiek įprastųjų, tiek konstantinių) pagalba galima netiesiogiai pasiekti konstantinius duomenis. Svarbu pastebėti, kad antrieji sakiniai yra klaidingi, o pirmieji – ne:

```
*pi1++; (*pi1)++; *pci1++; (*cpi1)++;           /* OK */
(*pci1)++; *cpi1++; (*cpci1)++; *cpci1++;       /* C: Error,C++:Error */
```

Masyvai (I). Įvadas

1. Masyvai (angl. *arrays*) – tai agregatai, kurių elementai yra to paties tipo (t.y. masyvai – tai homogeniniai agregatai).
2. Masyvai apibrėžiami:

   ```
   tipas masyvo_vardas[masyvo_dydis];
   ```

   Skirtingai nei Paskalyje, C ir C++ nėra sintaksinės konstrukcijos masyvo tipui (kaip vartotojo apibrėžiamam išvestiniam tipui) apibrėžti – masyvus apibrėžiame kaip agregatinius kintamuosius, o ne kaip vartotojo apibrėžiamo agregatinio tipo kintamuosius. Ši ir daugelis kitų C ir C++ masyvų savybių sąlygoja tai, kad juos teisingiau būtų įvardinti pseudomasyvais.
3. C kalboje masyvo_dydis privalo būti konstantinis sveikas reiškinys, pvz.:

   ```
   #define DYDIS 10
   int ai1[DYDIS], ai2[DYDIS*10];
   ```

   C++ *masyvo_dydis* reiškinyje galima naudoti ir sveiko tipo konstantas, pvz.:

   ```
   const int dydis=10;
   int ai3[dydis], ai4[dydis*DYDIS*10];
   ```

   Nei C, nei C++ nepalaiko kintamo dydžio masyvų.
4. Masyvų elementai indeksuojami nuo `0`, t.y. kreipinys į pirmą masyvo elementą yra *masyvo_vardas*`[0]`, o į paskutinį - *masyvo_vardas*`[`*masyvo_dydis*`-1]`, tačiau masyvo ribos nėra kontroliuojamos, todėl yra leistini (nors paprastai neteisingi ir net pavojingi) kreipiniai *masyvo_vardas*`[-1]` ir *masyvo_vardas*`[`*masyvo_dydis*`]`.
5. Apibrėžiant masyvus galima inicijuoti visus arba kelis pirmuosius jo elementus, pvz.:

   ```
   int ai1[3]={1,2,3}, ai2[3]={1};
   ```

   Antruoju atveju paskutinieji masyvo elementai bus inicijuoti `0`. Ši savybė naudojama norint "apnulinti" visus masyvo elementus. Kadangi lokalūs masyvai kaip ir bet kokie lokalūs duomenys pagal nutylėjimą nėra inicijuojami (priešingai - globalūs duomenys yra "apnulinami") tai lokalaus masyvo "apnulinimui" pakanka `int ai[3]={0};`.
   Masyvų apibrėžimui su elementu inicijavimu patogu naudoti "atvirąją" formą:

   ```
   int ai[]={1,2,3};
   ```

   Kompiliatoriui pakanka šios informacijos nustatant masyvo dydį, t.y. tipizuojant apibrėžiamą objektą bei išskiriant jam vietą atmintyje.
6. Simbolių eilutės (toliau - tiesiog eilutės, angl. *strings*) – tai masyvai, kurių elementai yra simbolinio (`char`) tipo. C ir C++ neturi "aukšto" lygio vidinių kalbos priemonių (t.y. neturi bazinio eilutės tipo, pvz. `string`; tiesa, C++ turi standartinį eilutės šabloną `basic_string`) tekstinei informacijai apdoroti, todėl naudojami `char` masyvai. Eilutės-konstantos (t.y. tekstas) užrašomos sintaksine forma *"tekstas"* . C ir C++ eilučių formatas – ASCIIZ, t.y. kiekvienas simbolis masyve saugomas ASCII kodu, o masyvas užbaigiamas dvejetainiu nuliu, kurio simbolinė išraiška `'\0'` (nepainioti su simboliu `'0'`). Taigi, eilutės-konstantos atmintyje užimamas baitų skaičius yra vienetu didesnis negu teksto ilgis. Eilutėmis-konstantomis leidžiama inicijuoti simbolių masyvus, todėl svarbu nurodyti pakankamą masyvo dydį, pvz.:

   ```
   char ac[4]="abc";
   ```

   kas yra ekvivalentiška tokiam masyvo inicijavimui:

   ```
   char ac[4]={'a','b','c','\0'};
   ```

   Būtent tokiais atvejais patogiausia naudoti "atvirąją" apibrėžimo formą:

   ```
   char ac[]="abc";
   ```
7. Masyvai programavimo kalbose yra itin patogi informacijos saugojimo (agregavimo) ir apdorojimo (manipuliavimo) priemonė, tačiau skirtingai nuo kitų kalbų, C ir C++ masyvus, kaip kalbos konstrukcijas, palaiko silpnai. Du svarbiausi ypatumai:
   - masyvais beveik negalima manipuliuoti kaip agregatais (žr. "Masyvų apdorojimas"),
   - darbą su masyvais sistema užtikrina rodyklių pagalba (žr. "Masyvai ir rodyklės").

Masyvai (II). Masyvai ir rodyklės

1. Masyvo vardas identifikuoja rodyklės tipo duomenį, o ne masyvą kaip agregatą. T.y. masyvo vardas kaip reiškinio (nepamirškime – C reiškinių kalba!) tipas yra rodyklė į masyvo elementus, o jo reikšmė lygi pirmojo masyvo elemento adresui, t.y. yra ekvivalentiški (žymėsime ==) šie kreipiniai
   masyvo_vardas == **&***masyvo_vardas***[**0**]**

   Masyvo vardą galima priskirti rodyklės tipo kintamajam, pvz.:
   ```
   int ai[10], *pi, ai2[10];
   pi = ai;
   ```
   tačiau atvirkštinis veiksmas
   ```
   ai = pi;
   ```
   neleistinas, nes masyvo vardas nėra rodyklės tipo kintamasis. Masyvo vardą galima traktuoti kaip rodyklės tipo konstantą (žr. Rodyklės ir konstantiškumas).

2. Pasinaudojant rodyklių aritmetikos ypatumais į masyvo elementus galima kreiptis keliais ekvivalenčiais būdais. Tegu
   ```
   int ai[10], *pi=ai;
   ```
   Taip inicijavus rodyklę, ji rodo į pirmąjį masyvo elementą. Tada `pi+1` rodo į antrąjį masyvo elementą `ai[1]` ir t.t. Kitaip tariant, yra ekvivalentiški šie kreipiniai
   ```
   ai[0] == *(pi + 0) == *pi
   ai[1] == *(pi + 1)
   ```
   ir t.t. Bet jeigu `ai` yra rodyklė tai kiekvienam masyvo indeksui `i` teisinga
   ```
   ai[i] == *(ai + i)
   ```
   Pilnumo sumetimais (pabrėžiant glaudžią rodyklių ir masyvų sąsają) leidžiama pritaikyti indeksavimo operaciją ir "paprastai" rodyklei, t.y.
   ```
   pi[i] == *(pi + i)
   ```
   Taigi, matome, kad masyvų indeksavimo operacija sistemoje realizuojama žemesnio lygio primityvų ir mechanizmų pagalba, t.y. rodyklėmis ir rodyklių aritmetika.

3. Rodyklės ypatingai dažnai naudojamos eilučių apdorojime. Įdomus tas faktas, kad eilutė-konstanta, pvz. `"abc"` (žr. Masyvai(I).Įvadas), kaip reiškinys yra rodyklė tipo, o jo reikšmė yra atminties srities, kurioje patalpinama eilutė (kaip masyvas), adresas. Todėl eilute-konstanta galima inicijuoti ne tik `char` masyvą, bet ir rodyklės į `char` tipo kintamąjį, pvz.:
   ```
   char ac[]="abc", *pc="xyz";
   ```
   Tačiau nepaisant rodyklių ir masyvų "panašumo" šie du apibrėžimai nėra ekvivalentiški. Pirmuoju atveju yra sukuriamas 4 baitų masyvas vardu `ac` , kuris yra inicijuojamas (t.y. užpildomas) eilutės simboliais. Antru atveju sukuriami du objektai: "bevardis" 4 baitų masyvas, kuriam kompiliatorius kaip globaliam duomeniui atmintį išskiria duomenų segmente ir rodyklės tipo kintamasis `pc` , kuris inicijuojamas šio masyvo adresu.
   Kadangi abu apibrėžimai gali būti panaudoti sukuriant tiek lokalius, tiek globalius objektus, svarbu pastebėti, kad pirmasis apibrėžimas lokalaus `ac` atveju yra labai neefektyvus, nes kiekvieno funkcijos iškvietimo metu masyvas bus kuriamas lokaliai (steke). Tuo tarpu antrasis apibrėžimas yra žymiai efektyvesnis, nes lokaliai bus kuriama tik rodyklė ir inicijuojama globalaus "bevardžio" masyvo adresu.
   Kita vertus, eilučių apdorojime rodykles reikia naudoti labai atsargiai. Pirma, priskyrimas `pc=ac;` paprastai reiškia eilutės-konstantos, kaip netiesiogiai pasiekiamo duomens, praradimą, nes pakartotinis priskyrimas `pc="xyz";` negarantuoja, kad abi konstantos yra ta pati atminties sritis, o tai reiškia, kad neefektyviai išnaudojama atmintis. Antra, būtina atkreipti dėmesį į tai, kad rodyklės sukūrimas (ir net jos inicijavimas) dar nereiškia jos paruošimo manipuliacijoms su eilutėmis. Pvz. funkcijos fragmentas
   ```
   char ac[]="abc", *pc=ac;
   gets(pc);
   ```
   dažniausiai baigsis sistemos klaida (programos "lūžiu"), jei eilutės skaitymu į atmintį nebus tinkamai pasirūpinta ir ji bus skaitoma į "uždraustą" atmintį (plačiau žr. Dinaminė atmintis).

Masyvai (III). Masyvų apdorojimas

1. Masyvų kopijavimo (priskyrimo) negalima kaip kitose kalbose užrašyti, pvz.
```
int ai1[10], ai2[10];
ai1=ai2;
```
nes `ai1` ir `ai2` , kaip jau buvo minėta, yra rodyklės (konstantinės), o ne masyvai kaip agregatai. Kadangi rodyklės yra konstantinės šis sakinys yra klaidingas.

2. Masyvų kaip agregatų negalima nei perduoti į funkciją, nei grąžinti iš jos - galima perduoti ir grąžinti tik masyvų adresus. Tačiau jei sintaksinė konstrukcija (imituojanti funkcijos grąžinančios masyvą prototipą)
```
int[10] f1(int);
```
neleidžiama (leidžiama Java kalboje, kur masyvai palaikomi stipriau), tai konstrukcija
```
int f1(int[10]);
```
yra leistina, nors, griežtai tariant, informacija joje yra perteklinė ir netgi klaidinanti. Kadangi į funkciją perduodamas masyvo adresas, tai parametras turėtų būti rodyklės tipo, o ne pats masyvas t.y.
```
int f1(int*);
```
Šis masyvų-paramametrų apibrėžimo būdas atspindi techninę realizaciją, o aukščiau panaudotas, kaip ir šis
```
int f1(int[]);
```
leidžiami ir naudojami pagal sutarimą skaitomumo sumetimais ir yra ekvivalentūs.

3. Kadangi į funkciją galima perduoti tik masyvo adresą, faktiniu parametru nurodant masyvo vardą, tai norint parūpinti "visą" informaciją apie masyvą, reikia perduoti ir masyvo elementų skaičių. Paprastai tam naudojamas antras (kitas) funkcijos parametras. Pvz. funkcijos skaičiuojančios `float` tipo masyvo elementų sumą rekomenduojama (kadangi lengviau skaitoma) realizacija
```
float afsuma1 (float a[], int size){
  int i; float sum;
  for ( i=0,sum=0; i<size ;sum+=a[i++] );
  return sum;
}
```
Nors techniškai ji ekvivalenti tokiai realizacijai
```
float afsuma1 (float* a, int size){
  int i; float sum;
  for ( i=0,sum=0; i<size ;sum+=*(a+i++) );
  return sum;
}
```

4. Perduodant masyvo elementų skaičių į apdorojančią funkciją, patogu pasinaudoti bemaž vienintele sistemos savybe, kai masyvas traktuojamas (šiuo atveju – kompiliatoriaus) kaip agregatas. Operacijos **sizeof** *masyvo_vardas* rezultatas yra baitų skaičius, kurį masyvas užima atmintyje, t.y. *masyvo_vardas* šiuo atveju netraktuojamas kaip rodyklė. Pasinaudojant šia savybe galima užrašyti "universalų" kreipinį į masyvą apdorojančią funkciją, pvz.:
```
afsuma1(ai, sizeof ai/sizeof ai[0]);
```

5. Kiek kitaip yra su eilutėmis, nes eilutės pasižymi tuo, kad turi (masyvo) pabaigos požymį, t.y. dvejetainį nulį, kurio simbolinė išraiška `'\0'`. Taigi, eilutę apdorojanti funkcija, nors ir gavusi tik jos adresą, gali pati nustatyti jos pabaigą. Pvz. funkcijos paskaičiuojančios eilutės ilgį galima realizacija (pastaba: standartinėje bibliotekoje yra funkcija `strlen()`)
```
unsigned lenstr (char* str){
 unsigned len;
  for (len=0; *str++ ;len++ );
  return len;
}
```

Papildomos tipizavimo galimybės. `typedef` ir `enum`

1. C (o kartu ir C++) turi priemonę tipų (tiek bazinių, tiek išvestinių) sinonimams (pseudonimams) apibrėžti. Tam tarnauja specializuota aprašo sakinio forma panaudojant rezervuotą žodį **`typedef`**.

   1.1. Bazinių tipų sinoniminiams apibrėžti naudojama sintaksė
   ```
   typedef tipo_vardas sinoniminis_vardas;
   ```
   Pvz.:
   ```
   typedef unsigned short int USHORT;
   ```

   1.2. **`typedef`** nesukuria naujo tipo. Šios konstrukcijos nereikia painioti ir su preprocesoriaus direktyva **`#define`**.

   1.3. Bendru atveju sinoniminio vardo apibrėžimo sintaksė analogiška kintamojo apibrėžimui. Pavyzdžiui, kad apibrėžti rodyklės į `char` tipo sinonimą rašome konstrukciją labai panašią į rodyklės į `char` tipo kintamojo apibrėžimą, pvz.:
   ```
   typedef char *STRING;
   ```

   1.4. Sinoniminių tipų vardus galima panaudoti kintamųjų apibrėžime, pvz.:
   ```
   USHORT u;
   STRING s;
   ```
   Tačiau svarbu atkreipti dėmesį, kad `s` yra rodyklė (nors tokiame apibrėžime to ir nesimato) ir jai taikytina rodyklės operacija `*s`.

   1.5. **`typedef`** yra nors ir primityvi, tačiau pakankamai lanksti abstragavimosi priemonė, pvz.:
   ```
   typedef float PINIGAI; typedef float PLOTAS;
   ```

   1.6. **`typedef`** pagalba kintamųjų aprašų sudėtingumą patogu "paslėpti" tipo apraše, pvz.:
   ```
   #define DIM 10
   typedef int IVECTOR[DIM];
   typedef IVECTOR IMATRIX[DIM];
   ```
   Šie du apibrėžimai yra ekvivalentiški
   ```
   IMATRIX m; int m[DIM][DIM];
   ```

2. C (o kartu ir C++) turi galimybę paskelbti tipą išvardinant baigtinį rinkinį konstantų, kurių reikšmės yra sveiki skaičiai. Taip apibrėžtus tipus vadinsime išvardinamaisiais arba **`enum`** tipais.

   2.1. Sintaksė
   ```
   enum tipo_vardas_opt {konstantų_sąrašas} tipo_kintamieji_opt;
   ```

   2.2. C kalboje `tipo_vardas` yra tik etiketė (*angl. tag*) , todėl pilnam tipo nurodymui naudojama
   ```
   enum tipo_vardas tipo_kintamieji;
   ```
   C++ `tipo_vardas` yra pilnavertis naujo tipo vardas, todėl leidžiama
   ```
   tipo_vardas tipo_kintamieji;
   ```

   2.3. Konstantas galima inicijuoti sveikais skaičiais. Jei to nepadaro programuotojas, tai atlieka kompiliatorius remdamasis 2 taisyklėmis. Pati pirmoji konstanta, jei ji neinicijuota tiesiogiai, yra inicijuojama `0`, o kiekviena kita neinicijuota konstanta inicijuojama prieš ją rinkinyje esančios konstantos reikšme padidinta vienetu, pvz.:
   ```
   enum Boolean {False=0,True=1};
   enum Boolean {False=0,True};
   enum Boolean {False,True};
   ```
   apibrėžimai yra ekvivalentiški.

   2.4. Konceptualiai išvardinimo tipo kintamieji turėtų operuoti tik savo reikšmių aibės reikšmėmis, t.y. išvardintomis konstantomis, o prasmingos operacijos turėtų būti priskyrimo ir palyginimo. Tačiau esant akivaizdžiam ryšiui tarp `int` ir `enum` C kalboje išvardinimo tipo operandai reiškiniuose traktuojami kaip sveiko tipo. C++ elgiasi griežčiau, laikydamas `enum` nauju pilnaverčiu tipu ir riboja išvardinimo tipo operandų ekvivalentiškumą su sveikojo tipo operandais. Pvz.:
   ```
   enum Diena {Pir=1,Ant,Tre,Ket,Pen,Ses,Sek};
   enum Diena d1=Ant;          /* OK */
   enum Diena d1=0, d2=2;      /* C:OK, C++:Warning arba Error */
   ```

3. C kalboje "ilgam" `enum` vardui "sutrumpinti" dažnai apibrėžiamas bendravardis sinonimas, pvz.:
   ```
   typedef enum Diena {Pir=1,Ant,Tre,Ket,Pen,Ses,Sek} Diena;
   ```

Struktūros (I). Įvadas

1. Struktūros (angl. *structures*) – tai agregatai, kurių elementai yra (tiksliau, gali būti) skirtingų tipų (t.y. struktūros – tai heterogeniniai agregatai). Struktūrų elementai dar vadinami nariais (angl. *members*) . C struktūros didžiąja dalimi yra analogiškos Paskalio įrašams (angl. *records*).
2. Skirtingai nei C ir C++ masyvai, struktūros apibrėžiamos kaip naujų, vartotojo apibrėžiamų, tipų kintamieji, t.y. iš pradžių apibrėžiamas (teisingo apibrėžimo ir aprašo terminų naudojimo prasme, tiksliau būtų sakyti aprašomas arba specifikuojamas) struktūros tipas (vadinsime, struktūrinis tipas) ir tik paskui apibrėžiama struktūra, kaip struktūrinio tipo kintamasis.
3. Supaprastintą sintaksę galima pateikti taip:

```
struct struktūrinio_tipo_vardas {
     nario_1_tipas narys_1;
     ...
     nario_N_tipas narys_N;
};
struct struktūrinio_tipo_vardas struktūrinio_tipo_kintamasis;
```

   Pvz.:

```
struct Studentas {
     int   kursas;
     char  pavarde[20];
};
struct Studentas studentas;
```

   Svarbu suprasti (dažna pradedančiųjų klaida), kad `Studentas` vardo apibrėžimas yra tipo, kaip kintamųjų šablono, aprašas (būtent aprašas, nes atmintis nėra išskiriama), tuo tarpu `studentas` vardo apibrėžimas yra "tikras" kintamojo apibrėžimas, nes yra išskiriama atmintis, kuri pažymima (įvardinama) šiuo vardu.
4. Bendresnę sintaksę galima pateikti taip:

```
struct struktūrinio_tipo_vardas_opt {
     nario_1_tipas narys_1 ,kiti_nariai_1_opt;
     ...
     nario_N_tipas narys_N ,kiti_nariai_N_opt;
} struktūrinio_tipo_kintamieji_opt;
```

   Matome, kad struktūras (kintamuosius) galima apibrėžti iš karto po tipo aprašo, pvz.:

```
struct Studentas {
     int   kursas, nr;
     char* vardas;
     char  pavarde[20];
} studentas, studente, studentai[10];
```

5. Kaip ir išvardinimo `enum` tipų atveju, C kalboje struktūrinio tipo vardas yra tik etiketė (*angl. tag*), todėl nepakanka

```
Studentas studentas;
```

   Tuo tarpu C++ kalboje `Studentas` yra pilnavertis naujo tipo identifikatorius, todėl toks apibrėžimas yra leidžiamas.

   C kalboje "ilgam" struktūrinių tipų vardui "sutrumpinti" (o kartu ir pasiekti C++ efektą) dažnai apibrėžiamas bendravardis sinonimas (žr. `typedef` ir `enum`), pvz.

```
typedef struct Complex {double re,im;} Complex;
```

Struktūros (II). Struktūrų apdorojimas

1. Struktūros (struktūriniai kintamieji) inicijuojamos, kaip ir masyvai, panariui pvz.:

```
struct Studentas {
     int metai, nr;
     int kursas;
     char  pavarde[20];
     char* vardas;
} fuksas = {2000, 1, 1, "Pavarde", "Vardas"};
```

2. Struktūrų narius galima pasiekti naudojant 2 priėjimo operacijas.

2.1. Jei turime įprastą struktūrinio tipo kintamąjį (trumpiau, struktūrą)

```
struct Studentas nebefuksas = fuksas;
```

naudojame . operaciją, pvz.:

```
nebefuksas.kursas = 2;
nebefuksas.pavarde[5] = 'g';
*nebefuksas.vardas = 'V';      /* *(nebefuksas.vardas) = 'V'; */
*(nebefuksas.vardas + 3) = 'g';
```

Kam lygi reiškinio `*nebefuksas.vardas + 3` reikšmė?

2.2. Jei turime rodyklės į struktūrinį tipą kintamąjį (trumpiau, struktūros rodyklę)

```
struct Studentas *pstudentas = &fuksas;
```

naudojame -> operaciją, pvz.:

```
pstudentas->kursas = 2;
```

kas naudojant . operaciją yra ekvivalentu (tačiau painu)

```
(*pstudentas).kursas = 2;
```

Pratęsdami praeito punkto priskyrimus, galime rašyti (pasistenkite užrašyti ir ekvivalentus)

```
pstudentas->pavarde[5] = 'g';
*pstudentas->vardas = 'V';
*(pstudentas->vardas + 3) = 'g';
```

2.3. Užduotis. Apibrėžkite struktūrinio tipo masyvą, pvz. `studentai[10]`, ir modifikuokite vieną iš jo elementų, kaip tai padaryta praeituose punktuose.

3. Skirtingai nei su masyvais, su struktūromis galima manipuliuoti kaip su agregatais.

3.1. To paties tipo struktūras galima priskirti vieną kitai (arba vieną inicijuoti kita, žr.2.1), pvz.:

```
struct Studentas fuksas, fukse, studentas;
studentas = fuksas;
```

Tiesa, jų negalima palyginti (nes neįmanoma nusakyti pirmumo-viršumo kriterijų)

```
if (fukse == fuksas) /* klaida */
```

3.2. Struktūras galima perduoti į funkcijas ir struktūras galima grąžinti iš funkcijų. Kai struktūra yra perduodama į funkciją kaip argumentas, ji yra perduodama pagal reikšmę, t.y. funkcijoje yra sukuriama lokali struktūra (struktūra-parametras) , kuri yra panariui inicijuojama struktūra-argumentu. Iš funkcijos struktūra grąžinama taip pat panariui. Pvz.:

```
struct Studentas perkelti(struct Studentas s) {
  s.kursas++;
  return s;
}
...
studentas = perkelti(studentas);
```

3.3. Toks struktūrų (kaip agregatų) apdorojimas funkcijose nors ir yra žingsnis į priekį lyginant su masyvais, tačiau tuo pačiu yra ir neefektyvus, nes struktūros gali būti labai dideli agregatai. Todėl į funkcijas rekomenduojama perduoti jų adresus. Pvz.:

```
void perkelti(struct Studentas* ps) {
  ps->kursas++;
}
...
perkelti(&studentas);
```

Dinaminės atminties valdymas. C ir C++ priemonės.

1. C neturi vidinių kalbos priemonių darbui su dinamine atmintimi. Manipuliacijoms su dinamine atmintimi naudojamos standartinės bibliotekos funkcijos, aprašytos <stdlib.h> ir/arba <alloc.h>.

1.1. Dvi dinaminės atminties paskirstymo

```
void * malloc(unsigned size);
void * calloc(unsigned num , unsigned size);
```

ir viena perpaskirstymo

```
void * realloc(void * ptr , unsigned size);
```

funkcijos grąžina dinaminėje atmintyje paskirtos srities adresą. Šis adresas yra netipizuotas. Funkcijoms `malloc()` ir `realloc()` reikalingos atminties dydis baitais nusakomas parametre `size`. Funkcijai `calloc()` - parametruose `size` ir `num`, todėl paskirtos atminties dydis bus lygus parametrų sandaugai. Be to, `calloc()` apnulina paskirtą atmintį. `realloc()` vietoj parametre `ptr` (adresu) nurodytos atminties paskiria naują (tik tuo atveju, jei reikia!), perkopijuoja senosios turinį į naująją ir atlaisvina senąją.
Visos trys funkcijos grąžina `NULL` , jei paskirti atminties nepavyko (pvz. jos nebeliko).

1.2. Paskirta atmintis atlaisvinama funkcija (procedūra)

```
void free(void * ptr);
```

1.3. Naudingos manipuliavimo su dinamine atmintimi funkcijos yra aprašytos <alloc.h>, pvz.:

```
void * memcpy(void * dest , void * src , unsigned count);
void * memset(void * dest , int c , unsigned count);
```

Detaliau susipažinkite patys, tačiau naudodami jas būkite atsargūs – adresai `dest` ir `src` turi būti dinaminės atminties (jokiais būdais ne statinės ar automatinės) adresais.

1.4. Pavyzdys:

```
int * ptr;
ptr = /*(char*)*/ malloc(sizeof(int));
free(ptr);
```

1.5. Pavyzdys:

```
int dim; char * memptr;
printf("Kiek atminties reikia eilutei?\n");
scanf("%d",&dim);
if (NULL == (memptr = (char*)calloc(dim,sizeof(char)))) {
  printf("Neliko laisvos atminties\n"); exit(1); }
...
free(memptr);
```

2. C++ turi dinaminės atminties paskirstymo ir atlaisvinimo operacijas: **`new`** ir **`delete`**.

2.1. Kreipinio į dinaminės atminties paskyrimo operaciją sintaksė **`new`** *`tipas;`* , pvz.:

```
int * ptr;
ptr = new int;
```

Lyginant su C funkcijomis, galima pastebėti 3 `new` operacijos privalumus: pirma, nereikia vargintis dėl reikalingos atminties dydžio nurodymo – jį netiesiogiai nusako operandas *`tipas`*; antra, operacija grąžina tipizuotą rodyklę – jos tipas yra rodyklė į *`tipas`*; trečia, rodyklės galima inicijuoti dinaminės atminties adresais jų apibrėžimo metu, pvz.:

```
int * ptr = new int;
```

2.2. Dinaminę atmintį galima paskirti ir masyvui, pvz.:

```
scanf("%d",&dim);
memptr = new char[dim];
```

2.3. Paskirtos dinaminės atminties atlaisvinimo sintaksė įprastam tipo duomeniui **`delete`** *`adresas;`* ir **`delete []`** *`adresas;`* masyvui, pvz.:

```
delete ptr; delete [] memptr;
```

2.4. Skirtingai nei kitose kalbose (aplinkose), C ir C++ sistemos neatlieka automatinio nebenaudotinos dinaminės atminties atlaisvinimo (ši kitų kalbų savybė angliškai vadinama *garbage collection*). Todėl programuotojas turi pats pasirūpinti ne tik dinaminės atminties paskyrimu, bet ir deramu ir savalaikiu jos atlaisvinimu.

Dinaminės atminties valdymas. Dvimačio masyvo pavyzdys.

```
#include <stdio.h>
#include <stdlib.h>

void init_array (long**, int, int);
void print_array(long**, int, int);

int main() {
    long** matrix;
    int rows, cols;
    int i;

    scanf("%d%d", &rows, &cols);

    matrix = (long**)malloc(sizeof(long*)*rows);
    for (i=0 ; i < rows ; i++) {
        matrix[i] = (long*)malloc(sizeof(long)*cols);
    }

    init_array (matrix, rows, cols);
    print_array(matrix, rows, cols);

    for (i=0 ; i < rows ; i++) {
        free(matrix[i]);
    }
    free(matrix);

    return 0;
}

void init_array(long** matrix, int rows,int cols) {
    int i,j;

    for (i=0 ; i < rows ; i++)
    for (j=0 ; j < cols ; j++)
        matrix[i][j] = i+j+2;
/*      *(*(matrix+i)+j) = i+j+2;       */
}
void print_array(long** matrix, int rows,int cols) {
    int i,j;

    for (i=0 ; i < rows ; i++) {
      for (j=0 ; j < cols ; j++)
        printf ("%ld\t", matrix[i][j]);
      printf ("\n");
    }
}
```

```
2       3       4       5
3       4       5       6
4       5       6       7
5       6       7       8
```

Dinaminės atminties valdymas. Programos pavyzdys.

// str_hand.c

```
#include <string.h>
#include <stdlib.h>

void cpystr (char* dest, const char* sour) {
  while (*dest++=*sour++);
}
void dupstr01 (char* dest, const char* sour) {   /* klaidinga */
  dest = (char*) calloc(strlen(sour)+1,sizeof (char));
  while (*dest++=*sour++);
}
void dupstr02 (char** dest, const char* sour) {  /* klaidinga */
  *dest = (char*) calloc(strlen(sour)+1,sizeof (char));
  while (*(*dest)++=*sour++);
  /* bet ne  while (**dest++=*sour++); */
}
void dupstr11 (char** dest, const char* sour){
 char *ptrtmp;
  ptrtmp = *dest = (char*) calloc(strlen(sour)+1,sizeof (char));
  while (*ptrtmp++=*sour++);
}
void dupstr12 (char* dest[], const char sour[]) {
 int i;
  *dest = (char*) calloc(strlen(sour)+1,sizeof (char));
  for (i=0; (*dest)[i]=sour[i] ;i++);
}
void dupstr13 (char** dest, const char* sour) {
 int i;
  *dest = (char*) calloc(strlen(sour)+1,sizeof (char));
  for (i=0; *(*dest+i)=*(sour+i) ;i++);
}
char* dupstr (const char* sour) {
 int i;char* dest;
  dest = (char*) calloc(strlen(sour)+1,sizeof (char));
  for (i=0; *(dest+i)=*(sour+i) ;i++);
  return dest;
}
```

// str_hand.h

```
void  cpystr   (char* dest, const char* sour);
void  dupstr01 (char* dest, const char* sour);
void  dupstr02 (char** dest, const char* sour);
void  dupstr11 (char** dest, const char* sour);
void  dupstr12 (char* dest[], const char sour[]);
void  dupstr13 (char** dest, const char* sour);
char* dupstr   (const char* sour);
```

// str_test.c

```
#include <stdio.h>
#include <stdlib.h>
#include "str_hand.h"
#define  MY_PRINT(PTR) printf("\n\n%p %s",PTR,PTR)
#define  MY_PRINTD(PTR) printf("\n%p %08X",PTR,*PTR)

 char gach[]="Masyvas1", *gstr="Eilute1";

int main (void){
 char lach[]="Masyvas2", *lstr="Eilute2", *pch=lach;

  MY_PRINT(gach);MY_PRINTD(&gach);
  MY_PRINT(gstr);MY_PRINTD(&gstr);
  MY_PRINT(lach);MY_PRINTD(&lach);
  MY_PRINT(lstr);MY_PRINTD(&lstr);
  MY_PRINT(pch);MY_PRINTD(&pch);

 cpystr(pch,lach);
  MY_PRINT(pch);MY_PRINTD(&pch);
 dupstr01(pch,gstr);
```

MY_PRINT(pch);MY_PRINTD(&pch); /* free(pch); - Error:Exception */

```
 dupstr02 (&pch,lstr);
  MY_PRINT(pch);MY_PRINTD(&pch); free(pch);
 dupstr11 (&pch,gstr);
  MY_PRINT(pch);MY_PRINTD(&pch); free(pch);
 dupstr12 (&pch,lstr);
  MY_PRINT(pch);MY_PRINTD(&pch); free(pch);
 pch = dupstr (gach);
  MY_PRINT(pch);MY_PRINTD(&pch); free(pch);
return 0;
}
```

```
0040A128 Masyvas1
0040A128 0040A128

0040A141 Eilute1
0040A134 0040A141

0012FF80 Masyvas2
0012FF80 0012FF80

0040A149 Eilute2
0012FF7C 0040A149

0040A128 Masyvas1
0012FF78 0040A128
```

```
0040A128 Masyvas2
0012FF78 0040A128

0040A128 Masyvas2
0012FF78 0040A128
```

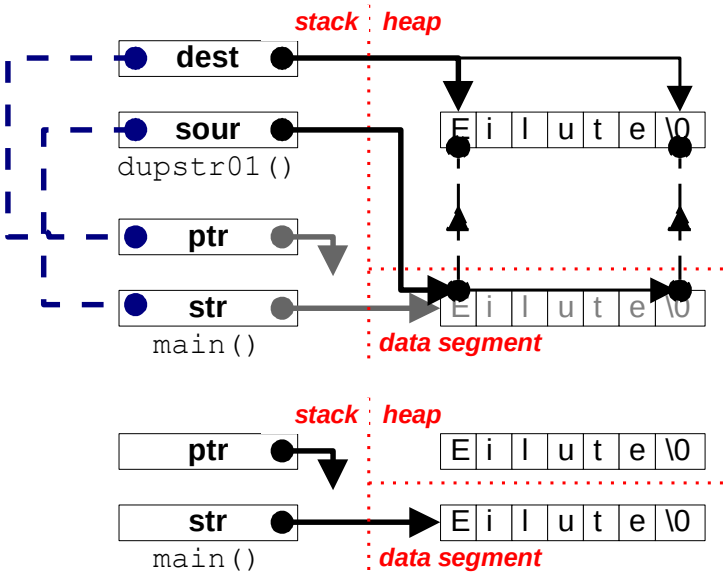
008928B4

```
0012FF78 008928B4

008928BC Eilute1
0012FF78 008928BC

008928BC Eilute2
0012FF78 008928BC

008928BC Masyvas2
0012FF78 008928BC
```



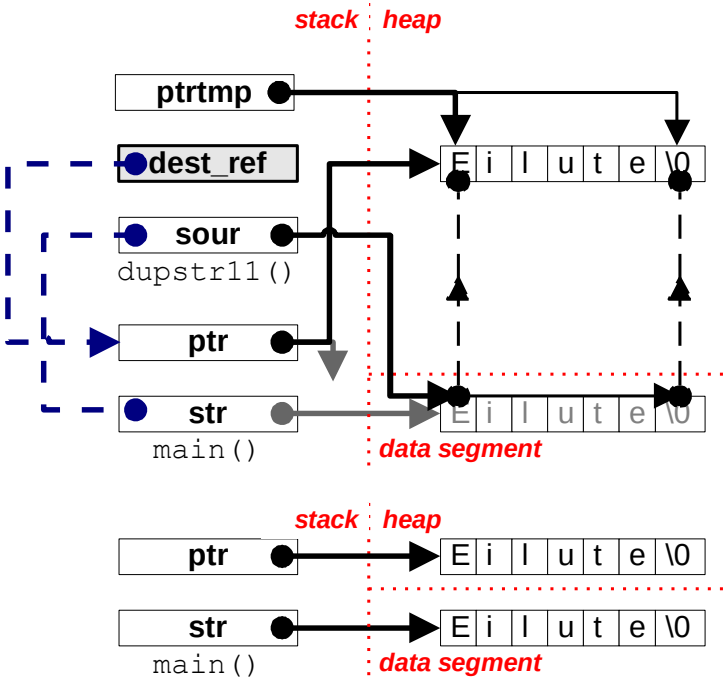

```
#include <stdlib.h>
#include <string.h>

void dupstr01 (char* dest, const char* sour) {
 dest = (char*) calloc(strlen(sour)+1,sizeof (char));
 while (*dest++=*sour++);
}

int main (void){
  char *str="Eilute";
  char *ptr=NULL;

  dupstr01(ptr,str);

  return 0;
}
```




```
#include <stdlib.h>
#include <string.h>

void dupstr11 (char** dest_ref, const char* sour){
 char *ptrtmp;
 ptrtmp=*dest_ref=(char*)calloc(strlen(sour)+1,sizeof(char));
 while (*ptrtmp++=*sour++);
}

int main (void){
  char *str="Eilute";
  char *ptr=NULL;

  dupstr11 (&ptr,str);

  return 0;
}
```

Dinaminės duomenų struktūros. Vienpusio sąrašo pavyzdys (0).

```
#include <stdio.h>
#include <stdlib.h>

struct LinkedList{
          char data;
          struct LinkedList *next;
};

struct LinkedList* Head=NULL;

void Push_element(char data) {
  struct LinkedList* to_new=NULL;

  to_new = (struct LinkedList*) malloc(sizeof(struct LinkedList));
  to_new->data= data;
  to_new->next= Head;
  Head=   to_new;
}

char Pop_element(void) {
  struct LinkedList* to_delete = NULL;
  char pop_data;

  /* reiktu patikrinti ar sarasas jau nera tuscias */
  to_delete =       Head;
  pop_data =        Head -> data;
  Head =  Head -> next;
  free(to_delete);
  return pop_data;
}

void String2List (char* str) {


  while (*str) Push_element(*str++);

}

void List2Prn (void)  {
  while (NULL != Head) printf("%c >> ",Pop_element());
  printf("NULL\n");
}


int main (void) {

  String2List("Globali");List2Prn();

  return 0;
}
```

```
i >> l >> a >> b >> o >> l >> G >> NULL
```

Dinaminės duomenų struktūros. Vienpusio sąrašo pavyzdys (1a).

```
#include <stdio.h>
#include <stdlib.h>

struct LinkedList{
          char data;
          struct LinkedList *next;
};



void push_element(struct LinkedList** headRef,char data) {
  struct LinkedList* to_new = NULL;

  to_new = (struct LinkedList*) malloc(sizeof(struct LinkedList));
  to_new->data= data;
  to_new->next= *headRef;
  *headRef=         to_new;
}

char pop_element(struct LinkedList** headRef) {
  struct LinkedList* to_delete = NULL;
  char pop_data;

  /* reiktu patikrinti ar sarasas jau nera tuscias */
  to_delete =       *headRef;
  pop_data =        (*headRef) -> data;
  *headRef =        (*headRef) -> next;
  free(to_delete);
  return pop_data;
}

struct LinkedList* string2list (char* str) {
  struct LinkedList* head=NULL;

  while (*str) push_element(&head,*str++);
  return head;
}

void list2prn (struct LinkedList* head)  {
  while (NULL != head) printf("%c >> ",pop_element(&head));
  printf("NULL\n");
}


int main (void) {
  struct LinkedList* head=NULL;
  head = string2list("lokali"); list2prn(head);

  return 0;
}
```

```
i >> l >> a >> k >> o >> l >> NULL
```

Dinaminės duomenų struktūros. Vienpusio sąrašo pavyzdys (1b).

```
#include <stdio.h>
#include <stdlib.h>

typedef   char Data, *String;
struct    LinkedList {
          Data data;
          struct LinkedList *next;
};
typedef   struct LinkedList Element, *Link;

void push_element(Link* headRef,Data data) {
  Link to_new = NULL;
  to_new = (Link) malloc(sizeof(Element));
  to_new->data= data;
  to_new->next= *headRef;
  *headRef=         to_new;
}

Data pop_element(Link* headRef) {
  Link to_delete = NULL;
  Data data;
  /* reiktu patikrinti ar sarasas jau nera tuscias */
  to_delete =       *headRef;
  data =  (*headRef) -> data;
  *headRef =        (*headRef) -> next;
  free(to_delete);
  return data;
}

Link string2list (String str) {
  Link head=NULL;
  while (*str) push_element(&head,*str++);
  return head;
}

void list2prn (Link head)  {
  while (NULL != head) printf("%c >> ",pop_element(&head));
  printf("NULL\n");
}

int main (void) {
  Link head1 = NULL;
  Link head2 = NULL;
  head1 = string2list("pirma"); list2prn(head1);
  head2 = string2list("antra"); list2prn(head2);
  return 0;
}
```

```
a >> m >> r >> i >> p >> NULL
a >> r >> t >> n >> a >> NULL
```

Dinaminės duomenų struktūros. Vienpusio sąrašo pavyzdys (2).

// list.h

```
typedef char Data;

struct LinkedList {
          Data data;
          struct LinkedList *next;
};
typedef struct LinkedList Element, *Link;

void push(Link*, Data);
Data pop(Link*);
void add(Link);     /* void add(Link*);          */
void delete(Link);  /* void delete(Link*);       */
int Count(Link);
void delete_list(Link*);
```

```
Link* create_list(void);
...
void delete_list(Link**);
```

// list.c

```
#include <stdlib.h>
#include "list.h"
```

```
void push(Link* headRef,Data d) {...*headRef=(Link)malloc(sizeof(Element));}
```

```
Data pop(Link* headRef) {...}
void add(Link* headRef) {...}
void delete(Link* headRef) {...}
int  Count(Link head) {...}
void delete_list(Link* headRef) {... *headRef = NULL;}
```

```
Link* create_list(void) {... return (Link*) malloc(sizeof(Link));}
...
void delete_list(Link** ptr_headRef) {... free(*ptr_headRef);
                                          **ptr_headRef = NULL;}
```

// c_list_driver.c

```
#include "list.h"

int main (void) {
  Link static_head=NULL;
  Data d;
  push(&static_head,'?');

  d = pop(&static_head);
  if ('?' != d)            ...blogai parašiau push() arba pop()
  if (NULL != static_head) ...blogai parašiau delete_list() arba delete()

  delete_list(&static_head);
}
```

```
  Link* ptr_dynamic_head=NULL;
  ptr_dynamic_head = create_list();
  push(ptr_dynamic_head,'!');
  ...
  delete_list(&ptr_dynamic_head);
}
```

// s2l.c

```
#include <stdio.h>
#include <stdlib.h>
typedef char Data;

struct LinkedList {
          Data data;
          struct LinkedList *next;
};
typedef struct LinkedList Element;
typedef Element *Link;

typedef char *String;

Link s2l_iter (String);
Link s2l_recur (String);
void l2prn_iter (Link);
void l2prn_recur (Link);

int main (void) {
  String str = "Iteracija";
  Link list_head=NULL;

  list_head =s 2l_iter(str); l2prn_iter(list_head);

  list_head = s2l_recur("Recursija"); l2prn_recur(list_head);

  return 0;
}

Link s2l_iter (String str) {
...
}

Link s2l_recur (String str) {
...
}

void l2prn_iter (Link head)  {
...
}

void l2prn_recur (Link head)  {
...
}
```

// s2l.c

```
#include <stdio.h>
#include <stdlib.h>
#include "s2l.h"

Link s2l_iter (String str) {
  Link head=NULL,tail=NULL;

  if ('\0' != *str) {
    head = (Link) malloc(sizeof(Element)); /* pasitikriname */
    head->data = *str++;
    tail = head;
    while (*str) {
      tail->next = (Link) malloc(sizeof(Element)); /* pasitikriname */
      tail = tail->next;
      tail->data = *str++;
    }
    tail->next = NULL;
  }
  return head;
}

Link s2l_recur (String str) {
  Link head=NULL;

  if (*str) {
          head = (Link) malloc(sizeof(Element)); /* pasitikriname */
          head->data = *str++;
          head->next = s2l_recur (str); /* rekursija */
          return head;
  }
  else
          return NULL;
}

void l2prn_iter (Link head)  {

  while (NULL != head) {
    printf("%c >> ", head->data);
    head = head->next;
  }
  printf("nil\n");
}

void l2prn_recur (Link head)  {

  if (head) {
    printf("%c >> ", head->data);
    l2prn_iter(head->next);
  }
  else
    printf("nil\n");
}
```

// s2l.h

```
typedef char Data;

struct LinkedList {
          Data data;
          struct LinkedList *next;
};
typedef struct LinkedList Element;
typedef Element *Link;

typedef char *String;

Link s2l_iter (String);
Link s2l_recur (String);
void l2prn_iter (Link);
void l2prn_recur (Link);
```

// s2l_driver.c

```
#include "s2l.h"

int main (void) {
  String str = "Iteracija";
  Link list_head=NULL;

  list_head =s 2l_iter(str); l2prn_iter(list_head);

  list_head = s2l_recur("Recursija"); l2prn_recur(list_head);
  /*
  l2prn_recur(list_head = s2l_recur("Recursija"));
  l2prn_recur(s2l_recur("Recursija"));
  */

return 0;
}
```

```
I >> t >> e >> r >> a >> c >> i >> j >> a >> nil
R >> e >> c >> u >> r >> s >> i >> j >> a >> nil
```

Atminties klasių specifikatoriai.

Objektas	***Apibrėžimo*[1] *vieta***	***Atminties klasės specifikatorius***	***Atminties klasė***	***Gyvavimo laikas***	***Galiojimo sritis***	***Surišimas***	**Matomas *(Matomumas)***	***Kaip gali būti pasiektas kituose failuose***
Object	*Location of definition*	*Storage class specifier*	*Storage class*	*Lifetime, Duration*	*Scope*	*Linkage*	*Visible (Visibility)*	
Kintamasis/ duomuo	funkcijos/bloko viduje	**`auto`**	automatinė (stekas)	funkcijos/ bloko	lokali	joks [none]	Šioje funkcijoje/ bloke[2]	**Nepasiekiamas**
		`register`	registrinė					
		`static`	**statinė**	**programos**				
		`nenurodytas`	žr. **`auto`**					
	funkcijos išorėje	*`nenurodytas`*[3]	statinė (duomenų segmentas)	programos	globali	išorinis [external]	Šiame[4] ir kituose failuose	Aprašius (bet neapibrėžus !) kintamąjį su **`extern`**.
		`extern`[3]						
		`static`				**vidinis [internal]**	**Tik šiame faile**	Nepasiekiamas
Funkcija	faile	**`extern`**	statinė (kodo segmentas)	programos	globali	išorinis [external]	Šiame[4] ir kituose failuose	Pateikus funkcijos prototipą su **`extern`** arba be.
		`static`				**vidinis [internal]**	**Tik šiame faile**	Nepasiekiamas
		`nenurodytas`	žr. **`extern`**					

[1] Apibrėžimas - tai instrukcija, kuri nurodo, kad instrukcijoje įvardinamam objektui turi būti išskirta atmintis. Atminties dydis ir klasė (rūšis) priklauso nuo instrukcijos panaudojimo programoje vietos bei objekto apibūdinančių tipo, atminties klasės (ir galbūt kitų) specifikatorių. Aprašas - tai instrukcija, kuri vienareikšmiškai apibūdina, kažkur programoje apibrėžtą objektą, taip sudarydama galimybes pasiekti ir kreiptis į objektą už jo apibrėžimo matomumo ribų. Programoje objektą (funkciją ar kintamąjį) pasižymintį tomis pačiomis savybėmis (vardu, tipu ir t.t) galima apibrėžti tik vieną kartą, tuo tarpu aprašyti galima tiek kartų kiek reikia, tačiau ten kur to reikalaujama.

[2] C++ - nuo apibrėžimo taško iki funkcijos/bloko pabaigos, nes kintamąjį galima apibrėžti bet kurioje funkcijos/bloko vietoje.

[3] C kalboje forma su **`extern`** ar be jo yra ekvivalentiškos, C++ - ne visai. Abejose kalbose "tikru" kintamojo apibrėžimu bus traktuojama instrukcija su kintamojo inicijavimu, nepaisant to panaudotas **`extern`** ar ne, tuo tarpu C++ apibrėžimu traktuoja ir formą be **`extern`**.

[4] Nuo apibrėžimo taško iki failo pabaigos, išskyrus funkcijas/blokus, kuriose tokiu vardu apibrėžti lokalūs kintamieji. C++ globalius kintamuosius, kuriuos "slepia" lokalūs kintamieji tais pačiais vardais, galima pasiekti naudojant globalaus konteksto operaciją : :