RODYKLĖS IR NUORODOS

RODYKLĖS Į FUNKCIJAS

C++ kalboje funkcijos vardas yra, iš esmės, rodyklė į šią funkciją, t. y. įėjimo į funkciją adresas, nuo kurio pradedami vykdyti funkcijos veiksmai. Tai pastovi rodyklė (konstanta), tačiau programuotojas gali apibrėžti savo rodyklę į funkciją. Bendra tokio apibrėžimo forma:

<tipas> **(*rodyklės_vardas) (**argumentų_tipų_sąrašas);

Pvz.,

*char (*Fptr) (int, int);*

skelbia *Fptr* rodyklę, kuri rodo į simbolinio tipo funkciją, turinčią du sveikojo tipo argumentus. Kai rodyklė į funkcija yra paskelbta, ją galima naudoti kreipinyje į funkciją vietoj funkcijos vardo:

```
char symb;
int year=2015;
int month=11;
symb = Fptr (year, month);
```

Funkcijos adresas

Funkcijos adresą galima rasti nurodžius funkcijos vardą be argumentų sąrašo ir be skliaustų dešinėje pusėje priskyrimo operatoriaus. Kairėje priskyrimo pusėje turi būti rodyklė į atitinkamo tipo funkciją.

```
int Func(int i, int j);
int (*ptr)(int, int) = Func;    //randamas funkcijos adresas
ptr(10,20);                     //iškviečiama funkcija
```

Rodyklių į funkcijas naudojimas

Rodyklės į funkcijas dažnai tarnauja kaip kitų funkcijų parametrai. Taip atsiranda galimybė rengti universalias funkcijas, kurios supaprastina sudėtingų programų tekstą, ypač įvairiose skaičiavimo metodų įgyvendinimo programose, tokiose, kaip algebrinių lygčių ir sistemų sprendimas, diferencijavimas, integravimas, diferencialinių lygčių sprendimas. Kai kurios standartinių bibliotekų funkcijos savo parametrų sąraše taip pat naudoja rodykles į funkcijas.

Pavyzdys 9.1. Rodyklių į funkcijas panaudojimas.

//Semestro nustatymas pagal mėnesio numerį

```
#include <iostream>
using namespace std;

bool isSpringTerm(int);
bool isFallTerm(int);

int main() {
   int month=1;
   bool (*pFunc)(int) = isFallTerm;

   if (isFallTerm(month))
      cout<<"Rudens semestras 1"<<endl;
   pFunc = isSpringTerm;
   if (pFunc(month))
      cout<<"Pavasario semestras 1"<<endl;;
   ++month;

   pFunc = isFallTerm;
   if (pFunc(month))
      cout<<"Rudens semestras 2";
   pFunc = isSpringTerm;
   if (pFunc(month))
      cout<<"Pavasario semestras 2";
   return 0;
}

bool isSpringTerm(int mn) {
   return (mn > 1) && (mn < 7);
}
bool isFallTerm(int mn) {
   return (mn > 8) && (mn <=12) || (mn == 1);
}
```

NUORODOS

Nuoroda (angl. *reference*) – tai dar vienas specifinis duomenų tipas, skirtas darbui atmintyje. Nuoroda – tai tarsi paslėpta rodyklė, kurią naudojant iš karto gaunama kintamojo reikšmė. Reiškia nuorodą galima naudoti tiesiog

kaip kitą kintamojo vardą (pseudonimą). Skelbiant nuorodą, ją reikia iš karto inicializuoti nurodomo objekto vardu:

<tipas> **&**nuorodos_vardas = kintamojo vardas;

Pvz.,

char raide = 'K';
char &nuor = raide;

Nuorodos nenaudoja papildomos atminties ir gali nurodyti bet kokio tipo objektą programoje. Bet koks nuorodos pasikeitimas reiškia ir objekto į kurį rodo nuorodą atitinkamą keitimą. Pavyzdžiui:

int year=2015;
int &ref1=year;
ref1 += 5; // dabar ref1 = year = 2020

Nuorodos negali rodyti į objektą, kuris turi reikšmę *NULL* ar *nullptr*. Jei koks nors objektas gali tapti „nuliniu", darbui su juo reikia naudoti rodykles. Kadangi nuoroda rodo į patį objektą, bet ne į jo tipą, nuorodai negalima priskirti kitą objektą (kad ir to paties tipo reikšmę). Jei taip nutinka, tai tik reiškia, kad pirmam objektui priskirtas šis antras objektas.

PARAMETRŲ PERDAVIMAS

Kaip jau minėta yra du pagrindiniai parametrų perdavimo į funkciją mechanizmai: reikšmės perdavimas ir perdavimas pagal reikšmės adresą, t. y. pagal nuorodą.

Kai vyksta funkcijos **argumentų perdavimas pagal reikšmę**, kompiliatorius sukuria laikiną perduodamo objekto kopiją ir ją talpino specialioje programos atminties srityje, vadinamoje *steko atmintyje*, kurioje saugomi lokalūs objektai. Iškviečiama funkcija dirba būtent su šiais lokaliais objektais (originalių objektų kopijomis) ir, reiškia nedaro jokios įtakos originaliems objektams. Tokių funkcijų prototipuose nurodomi parametrų (objektų) tipai, bet ne jų adresai, pvz.:

int max2(int, int);

Tačiau, kai reikia, kad funkcija pakeistų originalius objektus, naudojamas **perdavimas pagal adresą (nuorodą)**. Tokiu atveju pats objektas nėra perduodamas į funkciją, jai tik suteikiamas jo adresas, o visi funkcijos veiksmai vykdomi su originaliais objektais. Tai leidžia ne tik sutaupyti

atminties (steko, lokalios) ir laiko, bet ir gražinti iš funkcijos ne tik vienintelę funkcijos reikšmę, bet daug galimų rezultatų. Tik reikia turėti omenyje, kad tokių funkcijų naudojimas gali sukelti šalutinį efektą, kai funkcija atlieka programuotojo nenumatytus pakeitimus pvz., keičia globalių kintamųjų reikšmes.

C++ kalboje parametrų perdavimas pagal adresą galimas dviem būdais: naudojant rodykles ir nuorodas. Kai naudojamos nuorodos, funkcijos parametrų sąraše pateikiamos nuorodos į objekto tipą, pvz.:

void sukeisti(int& x, int& y);

Kreipinyje į tokią funkciją jos parametrų sąraše nereikia nurodyti & ženklo, pvz.:

sukeisti(x, y);

Jei būtų naudojamos rodyklės, funkcijos prototipas atrodytų taip:

void sukeisti(int, int*);*

Funkcijos grąžinama reikšmė taip pat gali būti rodyklė arba nuoroda. Pvz., funkcijų, kurios grąžina atitinkamai rodyklę ir nuorodą į sveikojo tipo reikšmę, prototipai:

**int count(int);*

&int increase();

Tačiau, čia reikia atidumo, nes gana dažna klaida, kai bandoma gražinti nuorodą į lokalų kintamąjį, kuris išeina iš matomumo srities (čia, pvz., *mn*):

```
int& f()
{
   int mn;
   return mn;
}
```

Argumentų (parametrų) perdavimas pagal adresą ypač efektyvus, kai dirbama su struktūriniais duomenų tipais ir perduodami dideli objektai, pvz., masyvai. Tokiu atveju taupoma ir programos naudojama atmintis, ir jos vykdymo laikas.

Tais atvejais, kai perduodami dideli duomenų objektai, bet jų modifikacija nėra numatyta, rekomenduojama naudoti konstantos tipo rodyklę, pvz., funkcija *Count()*

***const** int* Count (int* **const** number);*

gauna, ir grąžina rodyklę į pastovų sveikojo tipo objektą. Bandymas keisti šio tipo objektą funkcijos viduje iššauktų klaidą. Vietoje pastovių rodyklių galima naudoti ir pastovias nuorodas.

Reikėtų paminėti C++ kalbos sintaksės netobulumą, kai dirbama su pastoviomis rodyklėmis ir nuorodomis. Žemiau pateikti aiškinamieji pavyzdžiai:

```
int number;
const int count=0;

// pastovi rodyklė
int* const n1=&number;

// rodyklė rodo į konstantą
const int* n2=&count;

// ir rodyklė, ir nurodomą reikšmė yra konstantos
const int* const n3=&count;

// rodyklė rodo į pastovią eilutę
const char* str1=”text”;

// pati rodyklė į eilutę yra pastovi
char* const str2=”text”;

// ir rodyklė, ir nurodoma eilutė yra konstantos
const char* const str3=”text”;
```

FUNKCIJŲ PARAMETRAI – MASYVAI

Skirtingai nuo kitų duomenų tipų masyvai visada perduodami pagal adresą. Kai funkcija iškviečiama, kompiliatorius transformuoja masyvo tipo parametrą į rodyklę į masyvo tipą. Tokiu būdu, kai funkcija dirba su tokio masyvo-parametro elementais, visi pakeitimai vyksta originaliame masyve. Bendra kreipinio į tokią funkciją forma galėtų būti tokia:

Funkcijos_vardas(Masyvo_vardas);

Tada funkcijos prototipe reikia nurodyti masyvo tipą ir indeksavimo operatorių, pvz.:

Funkcijos_vardas(int[ ]);

Kitas galimas variantas – kartu su tipu nurodyti adreso ėmimo operaciją:

int *Funkcijos_vardas(int&);*

Tokiu atveju pasikeičia ir funkcijos kvietimo forma:

*Funkcijos_vardas(*Masyvo_vardas);*

Pavyzdys 9.2. Masyvo perdavimo į funkciją būdai.

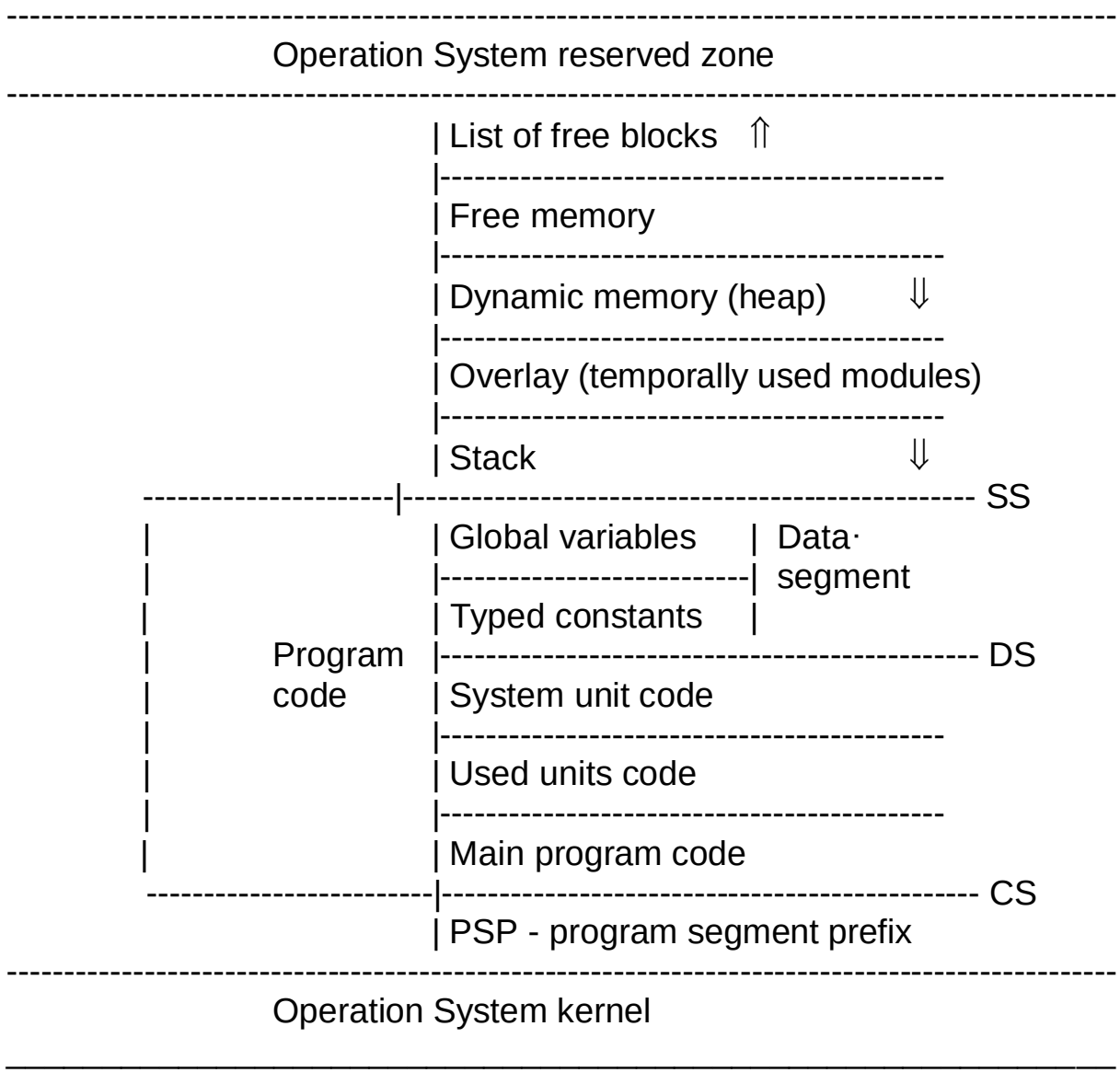
```
#include <iostream>
using namespace std;

void Spausdinti_1 (int[ ], int);
void Spausdinti_2 (int&, int);

int main() {
    int masyvas[ ]={10, 15, -20, 5, 30, 25, -10, 50};
      int n=sizeof(masyvas)/sizeof(masyvas[0]);
        cout<<"Pirmos funkcijos rezultatas"<<endl;
    Spausdinti_1(masyvas, n);
        cout<<endl<<"Antros funkcijos rezultatas"<<endl;
    Spausdinti_2(*masyvas, n);
    return 0;
}

void Spausdinti_1 (int mas[ ], int m) {
       for(int i=0; i<m; i++)
          cout<<mas[i]<<"  ";
}
void Spausdinti_2 (int& mas, int m) {
     for(int i=0; i<m; i++)
          cout<<*(&(mas) + i)<<"  ";
}
```

ATMINTIES SKIRSTYMAS – MEMORY ALLOCATION

Kiekvienas programos kintamasis gali būti talpinamas vienoje iš trijų atminties sričių: programos duomenų srityje, steke, dinaminėje atmintyje (krūvoje, angl. *heap*). Atmintis kintamiesiems gali būti skirstoma statiškai (programos kompiliavimo-pakrovimo metu) arba dinamiškai (programos vykdymo metu).

Kai programos duomenų apimtis nėra didelė, paprastai užtenka statinės atminties, t. y. programos duomenų srities ir stekinės atminties. Tačiau, jei, pavyzdžiui, naudojami keli didelės apimties masyvai, kurie reikalauja nepertraukiamų atminties sričių, šios atminties gali pritrūkti. Tokiu atveju C ir C++ programuotojau naudoja dinaminę atmintį.

PROGRAMOS ATMINTIES SRITYS



DINAMINĖS ATMINTIES SKIRSTYMAS (*ANSI C*)

Standartinėje C kalbos bibliotekoje yra funkcijos darbui su dinamine atmintimi, kurios leidžia išskirti atmintį programos objektams ir išlaisvinti užimamą atmintį. Šios funkcijos aprašytos antraštiniuose failuose *stdlib.h* arba *alloc.h*. Atminties skirstymas vykdomas *malloc()* ir *alloc()* funkcijomis, išlaisvinimas funkcija *free()*.

Funkcija

*void *malloc(size_t size);*

rezervuoja *size* dydžio atminties bloką dinaminėje atmintyje. Jei rezervavimas įvyko sėkmingai, funkcija grąžina rodyklę į šio atminties bloko pradžią, jei nesėkmingai – funkcija grąžina NULL. Konstanta NULL grąžinama ir kai nurodyta funkcijos argumento *size* reikšmė yra 0.

Čia pateiktas tipas *size_t* (kuris yra paskelbtas antraštiniame faile *stddef.h*) yra sinonimas tipo, grąžinamo *sizeof()*, t.y. *unsigned char*.

Praktikoje paprastai prireikia gražinamos reikšmės tipo *void* transformavimo į konkretų tipą, pvz., skirstant atmintį penkiems objektams sveikojo tipo, kreipinys galėtų būti toks:

int pint = (int)malloc(5*sizeof(int));*

Jei reikia paskirti atmintį dinaminiam dvimačiam sveikajam masyvui *Array[20][30]*:

Array = (int)malloc(20*30*sizeof(int));*

Funkcija

*void *calloc(size_t num, size_t size);*

be atminties skirstymo dar inicializuoja masyvo elementus nulinėmis reikšmėmis, masyvo elementų skaičių nurodo parametras *num*, o vieno elemento dydį baitais – *size*.

Jau rezervuotos dinaminės atminties dydį galima pakeisti, tiek padidinti, tiek sumažinti. Tam naudojama funkcija

*void *realloc(void *ptr, size_t new_size);*

kuri perskirsto rodyklės *ptr* nurodytą atminties sritį. Ši dinaminės atminties sritis anksčiau turėjo būti paskirta funkcijomis *malloc(), calloc()* arba

realloc() ir dar neturėjo būti atlaisvinta naudojant funkciją *free()*. Priešingu atveju, funkcijos rezultatas nėra apibrėžtas. Pvz., aukščiau apibrėžtai rodyklei *pint*

int pint = (int)malloc(5*sizeof(int));*

galima iš naujo rezervuoti keturgubai didesnę atminties sritį:

pint_new = (int)realloc(pint, sizeof(int)*20);*

Originali rodyklė *pint* dabar jau nebegalioja. Tačiau, jei perskirstymas nepavyksta, funkcija gražins nulinę rodyklę, o originali rodyklė *pint* lieka galioti.

Atmintis išlaisvinama funkcija *free()*

*void free(void *block);*

kurios parametras – tai rodyklė į išvalomą atminties bloką.

DINAMINĖS ATMINTIES SKIRSTYMAS *C++* KALBOJE

C++ naudoja naują dinaminės atminties skirstymo mechanizmą (standartinės bibliotekos komponentė *new*). Dinaminė atmintis paskirstoma naudojant operatorių *new()*, o išlaisvinama operatoriumi *delete*. Skirtingai nuo C kalbos funkcijų *malloc()* ir *calloc()*, operatorius *new()* grąžina rodyklę į tipą, kuriam skirstoma atmintis, todėl papildomų tipo transformacijų daryti nereikia. C++ kalba taip pat leidžia programuotojui perkrauti operatorius *new* ir *delete* ir tokiu būdu papildyti jų funkcionalumą pagal konkrečios programos poreikius. Be to, kalba pateikia du standartinius mechanizmus, skirtus patikrinti, ar paskirstymas buvo sėkmingas.

Yra dvi operatorių *new* ir *delete* formos, skirtos atitinkamai vieno objekto arba masyvo objektų atminčiai valdyti. Kai dirbama su pavieniais objektais, šių operatorių forma yra:

objekto_tipas *vardas = ***new*** objekto_tipas;

delete vardas;

čia *new* grąžina rodyklę į naujai paskirstyto atminties bloko pradžią. Kai reikalingas atminties blokas masyvui:

objekto_tipas *vardas = ***new*** objekto_tipas **[**objektų_skaičius**];**

delete[ ] vardas;

Konkrečiam objektui taikomo *delete* operatoriaus forma būtinai turi atitikti

operatoriaus *new*, kuris šiam objektui buvo panaudotas, formą, pvz.:

```
int * mas;
mas = new int [50];
. . .
delete[] mas;
```

čia sistema dinamiškai paskirsto atminties vietą sveikajam 50-ties elementų masyvui ir grąžina rodyklę į pirmąjį masyvo elementą, kuris priskiriamas *mas* rodyklei.

Vienas iš mechanizmų, skirtų patikrinti, ar atminties skirstymas buvo sėkmingas, yra naudoti *išimtis* (angl. *exception*). Naudojant šį metodą, jei nepavyksta paskirstyti atmintį, išmetama *bad_alloc* tipo išimtis. Detaliau apie išimčių valdymo mechanizmą bus kalbama Objektinio programavimo kurso dalyje.

Kitas metodas yra žinomas kaip *nothrow*. Čia, nepavykus atminties paskirstymui, užuot išmetus *bad_alloc* išimtį ir nutraukus programą, *new* grąžina nulinę rodyklę, o programa gali tęsti darbą toliau. Naudojant šį metodą, operatoriui *new* reikia nurodyti argumentą – specialų objektą *nothrow*:

```
int * mas;
mas = new(nothrow) int [50];
if (mas == nullptr) {
  cout<<"Nepavyko paskirstyti dinaminį atmintį"<<endl;
  // tam tikrų priemonių taikymas
}
```

Pavyzdys 10.1. Dinaminės atminties skirstymas.

```
#include <iostream>
#include <new>
using namespace std;

int main () {
   int i, n;
   int *p;
   cout << "Kiek masyvo elementų norite įvesti? ";
   cin >> i;
   p= new (nothrow) int[i];
   if (p == nullptr)
```

```
            cout << "Klaida: atmintis nėra paskirstyta";
   else  {
         for (n=0; n<i; n++)  {
            cout << "Įveskite elementą (sveiką skaičių): ";
            cin >> p[n];
         }
     cout << "Masyvo elementai: ";
     for (n=0; n<i; n++)
            cout << p[n] << ", ";
     delete[] p;
     }
  return 0;
}
```