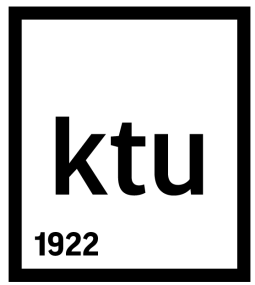
Atminties valdymas C ir C++

Karolis Ryselis

Kauno Technologijos Universitetas



C language combines the portability of the Assembly language with the ease of use of Assembly language.

Paskaitos turinys

C

- C — procedūrinio programavimo kalba.
- Pirmoji versija 1972 m. (Dennis Ritchie).
- Kalba daugiausia naudojama sistemų programavimui.
- Kalba sukurta taip, kad kompiliatorius būtų nesudėtingas, suteiktų prieigą prie žemo lygio operacijų.
- Nepalaiko nei objektinio, nei funkcinio programavimo.
- Paskutinė standarto revizija — C18.

Funkcijos parametrų perdavimas C ir rodyklės

- C parametrai funkcijai perduodami visada reikšme (*pass by value*).
- Perduodant parametrus **visada** daroma perduodamo kintamojo kopija.
- Be įprastų kintamųjų tipų (`int`, `double`, `char` ir kt.) C kalboje egzistuoja visų šių tipų rodyklės (*pointers*).
- Rodyklės tipo reikšmė yra adresas, kur laikomas kintamasis kompiuterio atmintyje.
- C kalboje galima rodyklių aritmetika, t. y., galima prie rodyklės tipo reikšmės pridėti, atimti ir pan.
- Galima sukurti savo struktūras (`struct`), joms taip pat galima naudoti rodyklių tipus.

C rodyklės

- Kadangi funkcijos parametrai visada yra kopijuojami, perdavus kintamąjį į funkciją ir pakeitus funkcijoje jo reikšmę, funkcijos išorėje pasikeitimų nesimatys, nes dirbama su kintamojo **kopija**.
- Perdavus rodyklės tipo kintamąjį funkcijai nurodoma ne kintamojo reikšmė, o vieta atmintyje.
- Kai funkcija žino, kurioje atminties vietoje yra kintamasis, ji gali keisti tą atmintį ir pasikeitimai matysis visur, kur naudojama ta atmintis.
- Dėl to, norint perduoti kintamąjį su galimybe funkcijos viduje jo reikšmę keisti, naudojamos rodyklės.

C rodyklės

- Ne rodyklės tipo kintamojo adresą galima sužinoti pasinaudojus & operatoriumi: `int* foo_ptr = &foo`.
- Rodyklės kintamojo rodomos vietos reikšmę galima sužinoti pasinaudojus * operatoriumi: `int foo = *foo_ptr`.

C rodyklės

```c
int multiply_by_two(int number){
    return number * 2;
}

void multiply_by_two_in_place(int* number) {
    *number = *number * 2;
}
```

C rodyklės

```c
int main() {
    int number = 42;
    printf("Number is %d\n", number);
    int* number_ptr = &number;
    printf("Address of number is %p\n", number_ptr);
    printf("Deferenced number_ptr is %d\n", *number_ptr);
    printf("Doubled number is %d\n", multiply_by_two(number));
    printf("Doubled number is %d\n", multiply_by_two(*number_ptr));
    multiply_by_two_in_place(&number);
    printf("Doubled in place: %d\n", number);
    multiply_by_two_in_place(number_ptr);
    printf("Doubled in place again: %d\n", number);
    return 0;
}
```

C rodyklės

Programos rezultatas:

```
Number is 42
Address of number is 0x7ffeacb9d3ac
Deferenced number_ptr is 42
Doubled number is 84
Doubled number is 84
Doubled in place: 84
Doubled in place again: 168
```

C masyvai

- C masyvai yra rodyklės, masyvo sintaksė (`int array[50]`) yra tik sintaksinis cukrus.
- Naudojantis masyvo sintakse galima sukurti masyvą, bet jis bus sukuriamas dėkle (*stack*) ir masyvui skirta atmintis bus atlaisvinta, kai bus baigta vykdyti funkcija, kurioje masyvas buvo paskelbtas.
- Norint turėti didelį (kupetoje (*heap*) išskirtą) arba funkcijos gyvavimo laiku neapribotą masyvą, galima turėti rodyklę į masyvo pradžią ir žinoti jo ilgį bei masyvo atmintį išskirti rankiniu būdu.

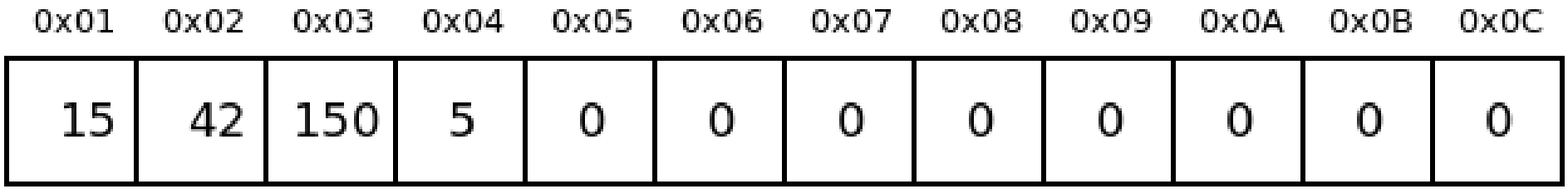
Skirtumai tarp masyvo ir rodyklės

- `int array[] = {1, 2, 3}` sukurs masyvą su trimis elementais, `int* array` galima priskirti tik adresą.
- Jei `array` yra paskelbtas kaip masyvas, `sizeof(array)` grąžins masyvo dydį atsižvelgiant į tai, kiek masyve yra elementų; jei tai rodyklė, grąžins tik rodyklės dydį.

Rodyklių aritmetika

- C kalboje prie rodyklės galima pridėti skaičių; rezultatas bus nauja rodyklė, rodanti į atminties vietą, pastumtą per tiek pozicijų, koks skaičius buvo pridėtas atsižvelgiant į rodyklės tipą.
- Operatorius `[]` su masyvais nesusijęs. Išraiška `array[1]` reiškia tą patį, ką `array + 1`, jei `array` yra rodyklės tipo.
- Kadangi `array + 1 == 1 + array`, tai `array[1] == 1[array]`.

Rodyklės ir atmintis



Tarkime, turime baitų masyvą `array`, kurio pradžia yra adresu `0x01`, bei kintamąjį. Tuomet:

- `array == 0x01;`
- `*array == array[0] == 15;`
- `*(array + 3) == array[3] == 5;`
- `&array[2] == 0x03;`
- Jei kursime rodyklę `short* x = &array[2]`, tada `x == 0x03`, `*x == 150`.

C dinaminio dydžio masyvai

- Norint sukurti dinaminio dydžio masyvą reikia patiems išskirti atmintį.
- Atminties valdymui yra skirtas funkcijų rinkinys. Pagrindinės funkcijos:
  - `malloc` — išskiria atmintį;
  - `calloc` — išskiria ir inicializuoja atmintį;
  - `free` — atlaisvina prieš tai išskirtą atmintį;

Atminties išskyrimas

`void* malloc(size_t size)`

Išskiria atmintyje tiek baitų, kiek nurodyta parametru `size`, ir grąžina rodyklę į išskirtos atminties pradžią.

`void* calloc(size_t nitems, size_t size)`

Išskiria atmintyje vietos `nitems` elementų, kurių kiekvieno dydis yra `size` baitų, išskirtą atmintį užpildo nuliais ir grąžina rodyklę į išskirtos atminties pradžią.

Abi funkcijos grąžina `void*`, todėl dažniausiai naudinga rodyklės tipą pasikeisti į prasmingą, pvz., `int*`, jei norime išskirti atmintį sveikų skaičių masyvui.

Atminties atlaisvinimas

`void free(void* ptr)`

Atlaisvina atmintį, kuri buvo išskirta naudojantis viena iš atminties išskyrimo funkcijų.

`free` tik pažymi, kad atmintį vėl galima naudoti naujiems kintamiesiems, bet jos turinio neišvalo.
`ptr` naudojimas po `free` kreipinio yra *undefined behaviour*.
Neiškvietus `free` atmintis nebus atlaisvinta, tai gali sukelti vis augantį atminties naudojimą (*memory leak*).

C masyvai

```c
int sum(const int *array, size_t array_size) {
    int result = 0;
    for (int i = 0; i < array_size; i++) {
        result += array[i];
    }
    return result;
}

void copy_array(const int *from, int *to, size_t array_size) {
    for (int i = 0; i < array_size; i++) {
        to[i] = from[i];
    }
}
```

C masyvai

```c
int main() {
    size_t n = 1300;
    int array1[n];
    fill_array_with_random_numbers(array1, n);
    int* array2 = malloc(n * sizeof(int));
    copy_array(array1, array2, n);
    int sum1 = sum(array1, n);
    int sum2 = sum(array2, n);
    free(array2);
    printf("Sum is %d\n", sum1);
    printf("Sum is %d", sum2);
    return 0;
}
```

C daugiamačiai masyvai

- C kalboje galima naudotis daugiamačio masyvo kūrimo sintakse: `int array[size1][size2]`.
- Tokiems masyvams galioja tie patys apribojimai, kaip paprastiems C masyvams.
- Galima kurti rodyklių rodyklių tipo kintamuosius: `int** array`.
- Dvigubos rodyklės kintamasis reiškia, kad jame yra saugomas adresas, kuriame yra saugomas kitas adresas.
- Kiekviena rodyklė yra masyvo, kuriame saugomos reikšmės, adresas, o pagrindinė rodyklė saugo adresą, kur saugomas pirmas adresų masyvo elementas.
- Kiekvieną masyvą reikia atskirai inicializuoti su `malloc` ir gautus adresus surašyti į rodyklių masyvą.

C daugiamačiai masyvai

```
long** matrix1 = malloc(sizeof(long*) * HEIGHT);
for (size_t i = 0; i < HEIGHT; i++) {
    matrix1[i] = malloc(sizeof(long) * WIDTH);
}
fill_2d_matrix_with_numbers(matrix1, WIDTH, HEIGHT);
long** result = malloc(sizeof(long*) * HEIGHT);
for (size_t i = 0; i < HEIGHT; i++) {
    result[i] = malloc(sizeof(long) * WIDTH);
}
double_matrix(matrix1, WIDTH, HEIGHT, result);
print_matrix(WIDTH, HEIGHT, result);
for (size_t i = 0; i < HEIGHT; i++) {
    free(matrix1[i]);
}
free(matrix1);
for (size_t i = 0; i < HEIGHT; i++) {
    free(result[i]);
}
free(result);
```

C++

- C++ iš pradžių palaikė procedūrinį ir objektinį programavimą.
- Pirmoji versija 1983 m. (Bjarne Stroustrup).
- C++11 (2011 m.) standartas buvo didelis C++ standartinės bibliotekos išplėtimas, pridėjo funkcinio programavimo elementus, gijų palaikymą, automatinį tipų nustatymą, pakeitė reikšmių priskyrimo veikimą ir kt.

C++ atminties valdymas

- C++ kalboje galioja tie patys atminties valdymo principai, kaip C.
- C++ turi papildomas priemones atminties išskyrimui ir valdymui.

Objektai ir rodyklės

- C++ atsirado objektinis programavimas — klasės ir objektai.
- C kalboje perduodant parametrą į funkciją visada kuriama parametro kopija.
- Norint tą patį padaryti C++ kalboje reikia žinoti, kaip kopijuoti objektus.
- Objektų kopijavimas nevisada yra triviali užduotis — kartais su objektu yra susietas išorinis resursas, kurio būsena yra susijusi su objekto būsena, kartais gali reikėti perkopijuoti rodyklių tipų kintamuosius.
- C++ kopijavimą sprendžia per kopijos konstruktorių — specialų konstruktorių, kuris parametru gauna kitą tos pačios klasės objektą ir sukuria analogišką objektą.
- C++ paprastoms klasėms gali automatiškai sukurti kopijos konstruktorius.

Objektai ir rodyklės

- Sudėtingesnėms klasės, kurioms, pvz., darant objekto kopiją reikia išskirti naują atmintį masyvui, reikia susikurti kopijos konstruktorių pačiam, nes C++ pats atminties neišskirs, o panaudos tą patį masyvą (nukopijuos rodyklę į masyvą).
- Kai kuriais atvejais C++ negali automatiškai sukurti kopijos konstruktoriaus:
  - Kai vienas iš kopijuotinų atributų negali būti nukopijuotas, pvz., jo kopijos konstruktorius yra privatus;
  - Paveldi iš klasės, kuri turi ištrintą kopijos konstruktorių;
  - Paveldi iš klasės, kuri turi ištrintą destruktorių.

Objektai ir rodyklės

- C++ perduodant funkcijai parametru objektą tas objektas turi turėti kopijos konstruktorių, kurį būtų galima iškviesti objekto perdavimo metu.
- Jei tokio konstruktoriaus nėra, tai objekto perduoti parametru negalima.
- Galima funkcijai perduoti rodyklę į objektą — tuomet veikimas bus toks pat, kaip C kalboje perduodant rodyklę.
- Jei C++ turime objektą `obj`, kuriam norime kviesti metodą `test`, tai darome `obj.test()`.
- Jei turime rodyklę į objektą, reikia kviesti `obj->test()`.

Nuorodos

- C++ yra dar vienas būdas perduoti parametrą — nuoroda (angl. *pass by reference*).
- Perduodant nuoroda nedaroma objekto kopija, o perduodamas tas pats objektas.
- Perduodant rodyklę funkcija gauna atminties adresą, perduodant nuorodą funkcija gauna objektą.
- Pagrindiniai skirtumai tarp nuorodų ir rodyklių:
  - Į nuorodą negalima priskirti kitos reikšmės;
  - Nuorodos reikšmė negali būti `NULL`, nes tai nėra adresas;
  - Nuoroda netinka, kai norime perduoti parametru masyvą;

Nuorodos

- Norint perduoti parametrą nuoroda, funkcijos apraše naudojamas simbolis &.
- `void test(MyClass& obj);`

Objekto perdavimas parametru

```
Counter counter;  // count = 0
counter.increase();  // count = 1
decrease_counter_ref(counter);  // count = 0
decrease_counter_copy(counter);  // count = 0
decrease_counter_pointer(&counter);  // count = -1

void decrease_counter_ref(Counter &counter) {
    counter.decrease();
}

void decrease_counter_copy(Counter counter) {
    counter.decrease();
}

void decrease_counter_pointer(Counter *counter) {
    counter->decrease();
}
```

Reikšmės grąžinimas iš funkcijos

- Reikšmės grąžinimas iš funkcijos veikia analogiškai, kaip ir perdavimas parametru.
- Galima grąžinti reikšme (kuriama kopija), nuoroda (grąžinama nuoroda į objektą) arba rodykle (grąžinama rodyklė į atminties vietą).
- C++, kaip ir C, jei objektas kuriamas funkcijos viduje, tai jis sunaikinamas funkcijos pabaigoje.
- Negalima iš funkcijos grąžinti jos viduje sukurto kintamojo grąžinti kaip nuorodos ar rodyklės, nes prieš grąžinant ją iš funkcijos objektas bus sunaikinamas.
- Objektus C++ galima kurti su raktiniu žodžiu `new` — tuomet į kintamąjį priskiriama rodyklė į objektą ir objektas automatiškai nenaikinamas, bet reikia rankiniu būdu iškviesti jo destruktorių.

Reikšmės grąžinimas iš funkcijos

```
Animal* get_chinchilla(const string& name) {
    auto animal = new Animal(name, "chinchilla");
    return animal;
}

Animal& get_chicken(const string &name) {
    auto animal = new Animal(name, "chicken");
    return *animal;
}
```

Reikšmės grąžinimas iš funkcijos

```
auto chinchilla = get_chinchilla("Django");
auto chicken = get_chicken("Rosetta");
auto cow = new Animal("Belle", "cow");
auto sheep = Animal("Dolly", "sheep");
vector<Animal*> animals = {chinchilla, &chicken, cow,
  &sheep};
for_each(animals.begin(), animals.end(),
    [](Animal* animal) {
        cout << animal->get_species() << " named "
            << animal->get_name() << endl;
});
delete(chinchilla);
delete(cow);
```

Kopijavimas ir perkėlimas

- Kopijavimas C++ vykdomas ne tik perduodant parametrą funkcijai reikšme, bet ir darant priskyrimą.
- `auto copy = original;` sukurs kintamojo `original` kopiją. Modifikuojant `copy`, `original` nebus modifikuojamas — tai du atskiri objektai.
- C++ galimas ir kitas variantas — kintamojo perkėlimas.
- `auto moved = move(original);` perkels kintamojo `original` reikšmę į `moved`: kintamasis `original` taps nebenaudotinas.
- Perkėlimas vykdomas visada, kai priskiriama reikšmė yra *rvalue*.

C++ *rvalue*

- *rvalue* tenkina tokias sąlygas:
  - Neįmanoma paimti jos adreso, pvz., `auto a = 42;` — reikšmės „42“ adreso paimti neina; taip pat ir `&i++` arba `&move(item)`.
  - Negalima jos naudoti kairėje priskirimo sakinio pusėje. `i++ = 5;` nesikompiliuos, nes kairėje yra *rvalue*.
  - Iš jos galima sukurti *lvalue*, kurios tipas yra `const&`. Pvz., jei turime funkciją `void function(const int& p)` ir ją kviečiame `function(i++)`, iš `i++` bus sukurta `const int&` tipo reikšmė ir perduota funkcijai.
- Perkėlimui, kaip ir kopijavimui, galima susikurti atskirą konstruktorių.
- C++ `thread` objekto kopijos konstruktorius yra ištrintas ir galima daryti tik objekto perkėlimą.