Тема 14. Процедурно програмиране - основни конструкции

1. Принципи на структурното програмиране

Основен принцип на структурното програмиране е ***модулния*** - програмата се разделя на "подходящи" взаимносвързани части(функции, модули, същности), всяка от които се реализира чрез определени средства.

2. Управление на изчислителния процес. Основни управляващи конструкции - условни оператори, оператори за цикъл

2.1 Начало и край на изчислителния процес

Обикновено в езиците за процедурно програмиране, програмите имат начална(главна) функция, от която започва изпълнението - в C++ тя се нарича main:

```
int main(int argc, char** argv)
```

main функцията, както всяка друга фунцкия, може да извиква и други функции;
Съществено е, че нормалното изпълнение на програмата завършва с края на изпълнението на main функцията.

По-конкретно, управляването на логиката на програмата се осъществява чрез управляващи конструкции, характерни за процедурния стил на програмиране.

2.2 Основни управляващи конструкции - условни оператори, оператори за цикъл

2.2.1 Условни оператори

Чрез тези оператори реализираме разклоняващи се изчислителни процеси.
Оператор, който дава възможност да се изпълни (или не) един или друг оператор в зависимост

от някакво условие, се нарича условен.
Ще разгледаме следните условни оператори: **if/else** и **switch**.

i. `if/else`

Синтаксис

```
if (<условие>) <оператор1> else <оператор2>
```

Семантика

Пресмята се стойността на булевия израз, представящ условието. Ако резултатът е `true`, изпълнява се `<оператор1>`. В противен случай се изпълнява `<оператор2>`.

Пример

```
int the_answer;
std::cin >> the_answer;
if(the_answer != 42) {
    std::cout << "You are mistaken!" << std::endl;
} else {
    std::cout << "I see you have the answer to everything" << std::endl;
}
```

ii. `switch`

Синтаксис

```
switch (<израз>) {
    case <израз 1> : <редица_от_оператори 1>
    case <израз 2> : <редица_от_оператори 2>
    …
    case <израз n-1> : <редица_от_оператори n-1>
    [default : <редица_от_оператори n>]
}
```

- `switch`, `case` и `default` са запазени думи
- `<израз>` е израз от допустим тип (типовете `bool`, `int` и `char` са допустими(целочислен), реалните типове `double` и `float` - не)
- `<израз 1>`, `<израз 2>`, …, `<израз n-1>` са константни изрази, задължително с различни стойности
- `<редица_от_оператори i>`, i = 1, 2, …, n, се дефинира така:

```
<редица_от_оператори> ::= <празно>|
                          <оператор>|
                          <оператор><редица_от_оператори>
```

Семантика

Намира се стойността на switch-израза. Получената константа се сравнява последователно със стойностите на етикетите `<израз1>`, `<израз2>`, …

При съвпадение се изпълняват операторите на съответния вариант и операторите на всички варианти, разположени след него, до срещане на оператор `break`. В противен случай, ако участва `default` вариант, се изпълнява редицата от оператори, която му съответства и в случай, че не участва такъв – не следват никакви действия от оператора `switch`.

Пример

```cpp
int main() {
    int year;
    int month;

    std::cin >> year;
    std::cin >> month;

    if (month < 1 || month > 12) {
        return 1;
    }

    bool is_leap = (year % 4 == 0 && year % 100 != 0) || (year % 400 == 0);
    std::cout << "The month has: ";

    switch(month) {
        case 1: case 3: case 5: case 7: case 8: case 10: case 12:
            std::cout << 31; break;
        case 2:
            std::cout << 28 + is_leap; break;
        default:
            std::cout << 30;
    } std::cout << " days.\n";

    return 0;
}
```

2.2.2 Оператори за цикъл

Операторите за цикъл се използват за реализиране на циклични изчислителни процеси, т.е. многократно повтаряне на група операции, докато не се изпълни дадено условие.

i. Оператор за цикъл `for`

Синтаксис

```
for (<инициализация>; <условие>; <корекция>)
<тяло>
```

Семантика

0. Изпълнява се инициализацията, тя се изпълнява точно веднъж. Създадените променливи на това място имат област на действие в цикъла.
1. Оценява се условието

- Ако се оцени до `true`, се стига до 2.
- Ако се оцени до `false`, се приключва изпълнението

2. Изпълява се тялото на цикъла, след това се извършва `корекция`, после се стига до стъпка 1.

Пример

```
#include <iostream>

int main()
{
    int result = 1;
    int n = 10;
    for(int prev = 0, k = 1, temp = 42; k < n; ++k) {
        temp = result;
        result += prev;
        prev = temp;
    }

    std::cout << result << std::endl; // 55, 10-то Фибоначиево число

    return 0;
}
```

ii. Оператор за цикъл `while`

Синтаксис

```
while (<условие>) <тяло>
```

Семантика

1. Оценява се условието. Ако е се оцени до `true` , стига се до стъпка 2. Иначе приключва изпълненението.
2. Изпълнява се тялото, след това се преминва към стъпка 1.

Пример

```
int main() {
    int n = 100;

    while(n) {
        std::cout << n << '\n';
        n /= 2;
    }
    return 0;
} // 100 50 25 12 6 3 1
```

Операторът `do-while` е подобен, с разликата че първо се изпълнява тялото и после се прави оценка на условието.

3. Видове променливи

Променлива е именувана област в паметта, която има **име**(идентификатор), **адрес** - място в паметта, **тип** и **стойност**.

```
<дефиниция> ::= <тип> <идентификатор> [ = <израз> ] {,
                      <идентификатор> [ = <израз> ] };
<присвояване> ::= <идентификатор> = <израз>;
```

Декларация заедно със задаване на стойност наричаме инициализация.
Можем и да използваме операторите {} и () за инициализация на променлива, освен оператор

"=".
В C++ неинициализирана променлива има **недефинирана стойност**.

Оператор за присвояване

```
<индетификатор> = <израз>;
<lvalue> = <rvalue>;
<lvalue> = <lvalue>;
```

Запазва се оценката на израза в променлива с посочения идентификатор.

`lvalue` е място в паметта със стойност, която може да се променя, напр. променлива
`rvalue` може да бъде временна стойност, без специално място в паметта, напр. константа, литерал, резултат от пресмятане

Операторът връща присвоената стойност(референция към a). Той е дясноасоциативен - напр. a = b = c = 42 $\iff$ a = (b = (c = 42)); chaining

3.1 Спрямо областта на действие

Областта на действие на променлива определя къде в кода тя може да бъде достъпвана. Локална променлива е такава, която е дефинирана в тялото на функция.

- локални променливи - имат блокова област на действие, т.е. имат област на действие от мястото им на дефиниране до края на блока, в който са дефинирани
- глобални променливи - не е декларирана в тялото на функция и са видими(освен ако не са скрити по правилото локалното скрива глобалното) из цялата програма; **жизнен цикъл** - до края на изпълнението на програмата.

4. Функции и процедури. Параметри - видове параматери. Предаване на параметри - по име и по стойност. Типове и проверка за съотвествие на тип

4.1 Типове

Типът на променлива указва на компилатора как да интерпретира съдържанието на паметта и носи семантична информация. Променливите от един един същи тип има подобни характеристики - размер, допустими операции.

- скаларни типове
  - интегрални - булев(`bool`), целочислен(`int` ,..), символен(`char` ,..), изброен(`enum`)
  - други - плаваща запетайка(`float` , `double`), указател(`T*`), референция(`T&`)
- съставни типове - масив(`[]` , в частен случ. низ `char[]`), запис(`struct`), клас(`class`) и обединение(`union`)

4.2 Функции и процедури. Параметри - видове параматери. Предаване на параметри - по име и по стойност.

4.2.1 Функции и процедури. Параметри - видове параматери.

В процедурното програмиране използваме понятията функция и процедура взаимозаменяемо в смисъла на относително независима част от програмата, извършваща определено пресмятане чрез последователност от оператори, която може да бъде използвана многократно.

Функция - именована последователност от инструкции

Функции в C++

```
<тип върната стойност> <име(идентификатор)> (<формални параметри>) {<тяло>}
```

- `void` = празен тип, не връща резултат
- ако типът на резултата се пропусне, подразбира се `int`

```
<формални_параметри> ::= <празно> | void | <параметър> {, <параметър> }
<параметър> ::= <тип> [<идентификатор>]
<тяло> ::= { <оператор> } // последователност от оператори
```

- ако `<идентификатор>` се пропусне, параметърът няма име и не се използва
  напр. : f (x, y) = x + 5

Извикване на функция

```
<име>(<фактически_параметри>)
<фактически_параметри> ::= <празно> | void | <израз> {, <израз> }
```

❗ ***забл.*** Типът на фактическия параметър се съпоставя с типа на съответния формален параметър. Ако се налага, прави се преобразуване на типовете

Връщане на резултат

```
return [<израз>];
```

- оператор за връщане на резултат на функция
- типът на `<израз>` се съпоставя с типа на резултата на функцията и ако се налага, прави се преобразуване на типовете
- работата на функцията се прекратява незабавно
- стойността на `<израз>` е резултатът от извикването на функцията

При извикване на функция се заделя нова стекова рамка(намира се на програмния стек), в която се пазят фактическите параметри, адрес за връщане и локалните променливи на функцията.

В зависимост от това как се пазят фактическите параметри, разглеждаме два варианта.

4.2.1 Предаване на параметри - по име и по стойност

Предаване по стойност(call by value)

- Пресмята се стойността на фактическия параметър и в стековата рамка се създава **копие** на стойността.
- Всяка промяна остава **локална** за функцията. При завършване на функцията предадената стойност и всички промени над нея изчезват.

Предаване по име(call by name, call by reference)

- Понякога искаме промените във формалните параметри да се отразяват и върху фактическите параметри
  Обявяваме, че фактическите параметри могат да бъдат променяни така:
  `<параметър> ::= <тип>& <идентификатор>`

Ако няма да правим промени, но не искаме да правим копия - const reference

f(5) при f приемаща const reference - ОК,
f(5) при примемаща не const reference - НЕ Е ОК, трябва lvalue

напр.

```
int add5(int& x) {
    x += 5;
    return x;
}

int add5_pointer(int* a) {
    return *(a++);
}
int a = 3; cout << add5(a) << ' ' << a; // 8 8
```

❗ ***забл.*** Фактическият параметър трябва да е `lvalue`, иначе ще се създаде копие и няма да имаме call by reference.

Проверка за съответствие на типовете на съответните фактически и формални параметри, както и за типа на връщаната стойност от функцията се прави по време на компилация(в C++).

❗ ***забл.*** Call-by-name може да се реализира и чрез предаване по адрес(указател).

5. Символни низове. Представяне в паметта. Основни операции със символни низове

Символен низ наричаме последователност от символи. В C++ най-базовото им представяне е това като масиви от символи, които завършват с терминиращия символ '\0'(завършваща нула, последният значещ силвол).

напр.

```
const char* str_literal = "Hello there"; // низов литерал
char my_str[] = {'H', 'e', 'l', 'l', 'o', '\0'};
// ако искаме да посочим размер, трябва да съобразим, че ни трябва място за '\0'
char my_str_again[6] = "Hello";
```

Основни операции със символни низове

Ще използваме C++, за да демонстрираме основните операции с низове.

- Вход
  - `>>` въвежда до разделител (интервал, табулация, нов ред)
  - `std::cin.getline(<низ>, <число>)` въвежда до нов ред, но не повече от `<число>-1` символа

- Изход - с `<<`
- Индексиране - с `[]`

Останалите характерни операции за работа с низове се съдържат в библиотеката `cstring` :

- `std::strlen(str)` - връща дължината на низ str, без терминиращия символ

```
// примерна реализация
int strlen(const char* str) {
    int len = 0;
    while(*str++ && ++len);
    return len;
}
```

- `std::strcpy(destination, source)` - копира низа `source` в масива `destination`

```
// примерна реализация
void strcpy(char* destination, const char* source) {
    while(*destination++ = *source++);
}
```

- `std::strcmp(str1, str2)` - сравнява двата низа лексикографски; връща 0, ако са еднакви посимволно, отрицателно число, ако първия предхожда лексикографски втория и положително число иначе.

```
// примерна реализация
int strcmp(const char* str1, const char* str2) {
    while(*str1 && *str1 == *str2) {
        ++str1;
        ++str2;
    }
    return *str1 - *str2;
}
```

- `std::strcat(str1, str2)` - конкатенация на низове, добавя втория низ след края на първия(презаписва се теминиращия символ на първия)

```
// примерна реализация
void strcat(char* str1, const char* str2) {
    while(*str1) {
        ++str1;
    }
    while(*str1++ = *str2++);
}
```

- `std::strchr(str, chr)` - търсене на символ **chr** в низ **str**; връща суфикса на низа от първото срещане на **chr**, а ако няма такъв символ, връща `nullptr`
- `std::strstr(str, substr)` - търси подниз **substr** на низа **str** и връща сифукса на низа от първото срещане на подниза, а ако няма такова - `nullptr` .