14. Процедурно програмиране - основни конструкции

Процедурното програмиране представлява програмна парадигма, която включва реализиране на поведението на програмата чрез процедури (функции), които се извикват една друга. Получената програма представлява поредица от стъпки, които образуват йерархия от извиквания на съставящите я процедури.

Процедурното програмиране се класифицира като императивно програмиране, защото включва директни команди за изпълнение.. То включва понятията блок и област на действие (scope), които са от съществено значение за ефективното структуриране на кода.

1. **Принципи на структурното програмиране**
   Основната идея на структурното програмиране е създаването на ясна структура в кода чрез изграждане на хубави структури, които посочват отделни, ясно дефинирани същности в нашия код.
   Представлява изграждане на програми с отделни компоненти с ясно дефинирани задачи и граници. Такива структури биха били например функциите. Метод, който ни помага да получим такава структура е декомпозицията.

   Основен принцип на структурното програмиране е принципът за модулност - разпределяне на функционалността на програмата на независими, взаимнозаменяеми модули, така че всеки от тях да съдържа всичко необходимо за изпълнението само на един аспект от желаната програма

2. **Управление на изчислителния процес**
   Процесът преминава през стъпки в определен ред, с ясно определени начална и крайна точка. В C++, изпълнението на програмата започва от main функцията. От своя страна тя може да извиква други функции, а нормалното изпълнение на програмата завършва с края на изпълнението на main функцията. Тя има следната дефиниция:
   $$int\ main(\ \underbrace{int\ argc}_{\substack{\text{размер на}\\ \text{масива } argv}}\ ,\ \underbrace{char * argv[]}_{\substack{\text{параметрите,}\\ \text{(подадени по терминала)}\\ \text{при стартиране на програмата}}}\ )$$
   Имплементации на цикли и условни оператори дават повече контрол върху хода на изпълнение на програмата.

3. **Основни управляващи конструкции - условни оператори, оператори за цикъл**
   C++ предоставян няколко оператора за контрол на потока на изпълнение, които позволяват на програмиста да промени нормалния път на изпълнение на програмата. Такива оператори са условните оператори и операторите за цикъл.

   Условен оператор
   Синтаксис:
   $if\ (\langle$условие_0$\rangle)\ \langle$оператор_0$\rangle$
   $\{else\ if\ (\langle$условие_i$\rangle)\ \langle$оператор_i$\rangle\}$
   $[else\ \langle$оператор_else$\rangle]$

   $if\ (\langle$условие$\rangle)\ \langle$оператор_1$\rangle\ [else\ \langle$оператор_2$\rangle]$

   Позволява ни да изпълним една (или повече) линии код само ако дадено условие е вярно. Частта в {} скобите може да присъства произволен брой пъти с различни условия и оператори. Частта в [] скобите е опционална (може да я има 0 или 1 път).

   Операторът се изпълнява се по следния начин:
   - Ако ⟨условие_0⟩ се оценява като булево true, то се изпълнява ⟨оператор_0⟩ и останалата част от условния оператор се прескача, иначе:
   - Ако ⟨условие_i⟩ се оценява като булево true, то се изпълнява ⟨оператор_i⟩ и останалата част от оператора се прескача, ако се оценява като false се проверява ⟨условие_i + 1⟩
   - Ако всички условия са се оценили като булево false, то се изпълнява ⟨оператор_else⟩

Условна операция

⟨булев_израз⟩ ? ⟨израз_1 ⟩ : ⟨израз_2⟩

Представлява триместна операция. Пресмята се ⟨булев_израз⟩

- При оценка true се пресмята ⟨израз_1⟩ и се връща резултатът
- При оценка false се пресмята ⟨израз_2⟩ и се връща резултатът

Пример: $x = y < 2 \ ? \ y + 1 : y - 2;$

- Ако у има стойност например 1, то на х ще бъде присвоена стойността 2
- Ако у има стойност 2, то на х ще бъде присвоена стойността 0.

Оператор за многозначен избор switch

switch (⟨израз⟩) {
{ *case* ⟨константен_израз_*i*⟩: {⟨оператор_*i*⟩}
[*default*: {⟨оператор⟩}]
}

Служи за проверка на променлива или израз за равенство спрямо набор от различни стойности.
Ако оценената стойност на ⟨израз⟩ е равна на някоя от стойностите след case (някой от ⟨константен_израз_*i*⟩), то се изпълнението започва от съответният ⟨оператор_*i*⟩ за дадения case и продължава последователно до настъпване на някое от следните условия за прекъсване:

- Достигнат е краят на switch блока
- Друг оператор за контрол на потока на изпълнение е срещнат (като break или return)
- Нещо друго прекъсне нормалното изпълнение на програмата (като ОС)

Това че има други case след изпълнения не е терминиращо събитие, без break или return изпълнението ще продължи и с техните оператори.
Ако не е намерена съответстваща стойност и съществува default етикет, то се изпълнява зададеният от него ⟨оператор⟩.

Ползата спрямо използване на множество if-else if-else е, че зададеният ⟨израз⟩ се оценява само веднъж и че същият ⟨израз⟩ се проверява за равенство всеки път.

Съществува оператор за прекъсване break, който може да бъде използван в тялото на някой от case операторите, за да прекрати по-нататъшно изпълнение на оператора switch

Оператори за цикъл

Циклите са конструкции за управление на изчислителния процес, които ни позволяват да изпълняваме многократно част от кода, докато не се изпълни някакво условие. Такива оператори са for и while.

Оператор for

for (⟨инициализация⟩ ; ⟨условие⟩ ; ⟨корекция⟩) ⟨тяло⟩

Първо се изпълнява ⟨инициализация⟩, то се случва само веднъж, когато цикълът е инициализиран. Обикновено се използва за декларация и инициализация като създадените променливи имат област на действие в цикъла (съществуват до края на блока на цикъла).

Второ, за всяка итерация на цикъла се оценява условието, ако резултатът е true, тялото на цикъла се изпълнява. Ако резултатът е false, то изпълнението на цикъла приключва.

След всяко изпълнение на тялото се изпълнява ⟨корекция⟩ - израз обикновено използван за увеличаване или намаляне на променливите на цикъла, дефинирани в ⟨инициализация⟩. След това отново се изпълнява Стъпка две.

Оператор while

while (⟨условие⟩) ⟨тяло⟩

При изпълнение на оператора while, първо се оценява стойността на ⟨условие⟩, ако е true, то се изпълнява ⟨тяло⟩ и след това се връща в началото на оператора като процесът се повтаря. Ако е false изпълнението на while приключва.

Операторът do while е сходен на while, но първо изпълнява ⟨тяло⟩ и после проверява ⟨условие⟩.

3. **Променливи - видове: локални променливи, глобални променливи; инициализация на променлива; оператор за присвояване**

Променливата представлява именувана област в паметта. С нея се свързва
- Име (идентификатор)
- Място в паметта (адрес)
- Тип
- Стойност

Областта на действие на променливата започва от мястото на дефинирането й до края на блока, в който е дефинирана.

Дефиниция на променливи

⟨тип⟩ ⟨идентификатор⟩ [= ⟨израз⟩] {,
⟨идентификатор⟩ [= ⟨израз⟩] };

Създаване на променлива от даден тип. Казваме, че сме я инициализирали, ако сме й задали начална стойност (чрез = ⟨израз⟩).

Може да извършим дефиниция и инициализация в една стъпка, използвайки следният синтаксис:
⟨тип⟩ ⟨идентификатор⟩ {⟨израз⟩} , където {} са част от синтаксиса, а не пожелателен елемент.
Или синтаксиса:
⟨тип⟩ ⟨идентификатор⟩ (⟨израз⟩)

Така директно се инициализира, докато като използване на = първо се дефинира променливата и след това й се присвоява стойност (сору инициализация)

Ако не инициализираме променливата, то нейната стойност е непредвидима - представлява стойността, съдържана в адреса, който се определя за променливата.

Оператор за присвояване

⟨идентификатор⟩ = ⟨израз⟩;
⟨$lvalue$⟩ = ⟨$rvalue$⟩;

Запазване на дадена стойност (оценката на ⟨израз⟩) в променлива с посочения идентификатор. Като резултат от присвояването се връща стойността на променливата.

- lvalue - място в паметта със стойност, която може да се променя. Пример: променлива
- rvalue - временна стойност, без специално място в паметта. Пример: константа, литерал, резултат от пресмятане

Присвояването е дясноасоциативна операция.
Пример: $a = b = c = 2$; се възприема като $a = \big(b = (c = 2)\big)$

Видове променливи спрямо областта на действие

Областта на действие на променлива определя къде в кода тя може да бъде достъпвана.

Локалните променливи имат блокова област на действие, което значи, че са в област на действие от мястото на дефинирането им до края на блока, в който са дефинирани.

Глобалните променливи са променливи, които са дефинирани извън тялото на функция. Областта им на действие е от мястото на дефиниция до края на файла, в който са дефинирани.

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

Типове на променливи

Типът на променливата указва на компилатора как да интерпретира съдържанието на паметта и носи семантична информация. Променливите от един и същи тип имат подобни характеристики - като размер, допустими операции.
- скаларни - булев, целочислен, символен, изброен, числа с плаваща запетайка, указател, референция (препратка)
- съставни - масив, структура (struct), клас

Функции и процедури. Параметри - видове параметри
Функцията е самостоятелен фрагмент от програмата, представляващ поредица от команди, които могат да се използват многократно и са предназначени за изпълнение на определена задача.

Функцията извършва пресмятане и връща резултат, докато процедурата изпълнява поредица от оператори. Двете понятия понякога се ползват взаимнозаменяемо. В C++ се наричат "функция".

Дефиниране на функция:
⟨тип_резултат⟩ ⟨идентификатор⟩ (⟨формални_параметри⟩) {⟨тяло⟩}
Където ⟨тип_резултат⟩ може да бъде void, ако не връща резултат или някой от вече споменатите типове. Ако се пропусне се подразбира типът int.

Формалните параметри са от вида:
- ⟨празно⟩ | *void* - функцията не приема параметри
- ⟨тип⟩ [⟨идентификатор⟩] {, ⟨тип⟩ [⟨идентификатор⟩] }

Извикване на функция:
⟨име⟩ (⟨фактически_параметри⟩)

Фактически параметри (аргументи) са от вида:
- ⟨празно⟩ | *void* - функцията не приема параметри
- ⟨израз⟩ {, ⟨израз⟩}
Типът на фактическите параметри се съпоставя с типа на формалните параметри. Ако се налага се прави преобразуване на типовете.

Връщане на резултат:
return [⟨израз⟩];

Оператор за връщане на резултат (стойността на ⟨израз⟩) от извикването на функцията

Типът на ⟨израз⟩ се съпоставя с типа на резултата на функцията. Ако се налага се прави преобразуване на типовете. Работата на функцията прекратява незабавно.

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

Функцията може само да се декларира като не се зададе тяло на функцията. Декларацията представлява "обещание" за дефиниция на функция.

Предаване на параметри - по име и по стойност
При предаване по стойност се пресмята стойността на фактическия параметър, в стековата рамка на функцията се създава копие на стойността. Така всяка промяна на стойността остава локална за функцията. При завършване на функцията, предадената стойност и всички промени над нея изчезват.

Предаване с препратка/референция се използва, когато искаме промените във формалните параметри (параметрите) да се отразят върху фактическите параметри (аргументите).
Трябва да обявим, че искаме фактическите параметри да могат да бъдат променяни. Това се случва със следния синтаксис:
⟨тип⟩& ⟨идентификатор⟩

Пример:
$int\ add5(int\&\ x)$ { x += 5; return x; }
$int\ a = 3;\ \ cout \ll add5(a) \ll$ " " $\ll$ a;

Резултатът ще бъде "8 8"
В този случай фактическите параметри трябва да бъдат lvalue.

По време на компилация се прави проверка за съответствие на типовете на формалните параметри със съответните фактически параметри и на типа на резултата на функцията.

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

Символен низ наричаме последователност от символи. В C++ се представя като масив от символи (char), в който след последния символ в низа е записан терминиращият символ '**\0**'.

- '\0' е първият символ в ASCII таблицата с код 0.

Пример:
$char\ word[] = \{'H', 'e', 'l', 'l', 'o', '\0'\};$

$char\ word1[100] =$ "$Hello$"; - ще се постави терминиращ символ след символите (подобно на горния пример), но ще се задели памет за общо 100 символа (тоест ще може да съхраним 99 символа и '\0').

Основни операции със символни низове

- Вход($>>$, $cin.getline(\langle$низ$\rangle, \langle$число$\rangle)$)
  - $>>$ въвежда до разделител (интервал, табулация, нов ред)
  - $cin.getline(\langle$низ$\rangle, \langle$число$\rangle)$ - въвежда до нов ред, но не повече от $\langle$число$\rangle$ - 1 символа
- Изход ($<<$)
  - Пример: $cout \ll$ "$Hello$";
- Индексиране ([ ])
  - Достъпване на i-тия символ в низа
    Пример: от по-горе word: $word[1]$ ще върне 'e'.

Вградени функции за низове (#include <cstring>)

- $strlen(\langle$низ$\rangle)$ - връща дължината на $\langle$низ$\rangle$, т.е. броят символи до срещане на '\0'.
- $strcpy(\langle$буфер$\rangle, \langle$низ$\rangle)$ - прехвърля всички символи от $\langle$низ$\rangle$ в $\langle$буфер$\rangle$, връща $\langle$буфер$\rangle$
- $strcmp(\langle$низ$_1\rangle, \langle$низ$_2\rangle)$ - сравнява два низа лексикографски, връща
  - 0, ако двата низа съвпадат
  - Число <0, ако $\langle$низ$_1\rangle$ е преди $\langle$низ$_2\rangle$
  - Число >0, ако $\langle$низ$_1\rangle$ е след $\langle$низ$_2\rangle$
- $strcat(\langle$низ$_1\rangle, \langle$низ$_2\rangle)$ - конкатенация на низове, записва символите на $\langle$низ$_2\rangle$ в края на $\langle$низ$_1\rangle$ като старият терминиращ символ се изтрива и се записва нов. Връща $\langle$низ$_1\rangle$.
  - Отговорност на програмиста е да осигури, че в $\langle$низ$_1\rangle$ има достатъчно място за всички символи на $\langle$низ$_2\rangle$
- $strchr(\langle$низ$\rangle, \langle$символ$\rangle)$ - търси $\langle$символ$\rangle$ в $\langle$низ$\rangle$, връща адреса на първото срещане на $\langle$символ$\rangle$ в $\langle$низ$\rangle$ (суфикс на $\langle$низ$\rangle$). Връща 0 (null pointer), ако $\langle$символ$\rangle$ не се среща.
- $strstr(\langle$низ$\rangle, \langle$подниз$\rangle)$ - търси $\langle$подниз$\rangle$ в $\langle$низ$\rangle$, връща адреса на първото срещане на $\langle$подниз$\rangle$ в $\langle$низ$\rangle$ или 0 (null pointer), ако не се среща.

Примерна реализация на strlen:

```
size_t strlen(const char* str) {
    int length = 0;
    while (str[length] != '\0') {
        ++length;
    }
    return length;
}
```