УРОК 03

1

Добре дошли в **част 1** на урок 03 от нашия курс по програмиране на C. В този урок ще обсъдим управлението на реда за изпълнение, което включва контролиране на реда, в който се изпълняват операторите в програмата. Правилното управление на реда на изпълнение е от решаващо значение за правилното функциониране на всяка програма.

Условно изпълнение и разклоняване : :

Условното изпълнение позволява на програмата да изпълни определени изрази само ако е изпълнено определено условие. Най-често използваният условен оператор в C е операторът "if". Ето един пример:

```
int x = 10;
if (x > 5) {
printf("x is greater than 5\n");
}
```

В този пример операторът „if“ проверява дали стойността на „x“ е по-голяма от 5. Ако условието е вярно, операторът във фигурните скоби ще бъде изпълнен, което ще отпечата съобщението „x е по-голямо от 5 ". Ако условието е невярно, изразът във фигурните скоби ще бъде пропуснат.

C също има операторите "else if" и "else", които позволяват проверка на множество условия. Ето един пример:

Ex_01

```
int x = 10;
if (x > 20) {
printf("x is greater than 20\n");
}
else if (x > 5) {
printf("x is greater than 5 but less than 20\n");
}
```

```
else {
printf("x is less than or equal to 5\n");
}
```

В този пример първото условие в оператора "if" е невярно, така че програмата преминава към следващото условие в оператора "else if". Тъй като "x" е по-голямо от 5, но по-малко от 20, се отпечатва съобщението "x е по-голямо от 5, но по-малко от 20". Ако и двете условия "if" и "else if" са неверни, програмата ще изпълни операторите вътре в блока "else".

Повторение на командите /цикли/ : :

С предлага няколко вида цикли за многократно изпълнение на блок от код. Най-често използваните цикли са цикълът "for", цикълът "while" и цикълът "do-while".

Цикълът "for" се използва, когато знаем броя на итерациите предварително. Ето един пример:

Ex_02

```
for (int i = 0; i < 10; i++) {
printf("%d\n", i);
}
```

В този пример цикълът ще повтори 10 пъти и стойността на "i" ще се увеличава с 1 всеки път. Цикълът ще продължи, докато стойността на "i" вече не е по-малка от 10.

Цикълът "while" се използва, когато не знаем броя на итерациите предварително. Ето един пример:

Ex_03

```
int i = 0;
while (i < 10) {
printf("%d\n", i);
i++;
}
```

В този пример цикълът ще продължи да се повтаря, докато стойността на "i" вече не е по-малка от 10. Стойността на "i" се увеличава с 1 при всяко повторение на цикъла.

Цикълът "do-while" е подобен на цикъла "while", но винаги ще се изпълнява поне веднъж, дори ако условието е невярно. Ето един пример:

Ex_04

```
int i = 0;
do {
printf("%d\n", i);
i++;
} while (i < 10);
```

В този пример цикълът ще се повтори, докато стойността на "i" вече не е по-малка от 10. Стойността на "i" се увеличава с 1 при всяко повторение на цикъла. Дори ако стойността на "i" първоначално е по-голяма или равна на 10, цикълът пак ще се изпълни поне веднъж

Преходи - 'break' и 'continue'

Преходите се отнасят за прескачане от една част на програма в друга. С предоставя два типа преходи: "прекъсване" и "продължаване".

Операторът "break" се използва за преждевременно прекратяване на цикъл или оператор за превключване. Ето един пример:

Ex_05

```
for (int i = 0; i < 10; i++) {
if (i == 5) {
break;
}
printf("%d\n", i);
}
```

В този пример цикълът ще приключи преждевременно, когато "i" е равно на 5. Това означава, че ще бъдат отпечатани само стойностите от 0 до 4.

Операторът "continue" се използва за пропускане на текущата итерация на цикъл и преминаване към следващата итерация. Ето един пример:

Ex_06

```
for (int i = 0; i < 10; i++) {
if (i == 5) {
continue;
}
printf("%d\n", i);
}
```

В този пример, когато "i" е равно на 5, текущата итерация ще бъде пропусната и цикълът ще премине към следващата итерация. Това означава, че всички стойности от 0 до 9, с изключение на 5, ще бъдат отпечатани.

Заключение:

В този урок разгледахме управлението на реда за изпълнение в C програмирането. Обсъдихме условно изпълнение и разклоняване с помощта на изразите „if“, „else if“ и „else“, както и циклите „for“, „while“ и „do-while“ за многократно изпълнение на блок от код. И накрая, обсъдихме преходите, включително изразите "break" и "continue" за прекратяване на цикъл или пропускане на определена итерация.

Тези концепции са от решаващо значение за овладяването на програмирането на C и ви насърчавам да практикувате писане на код, като използвате тези техники, за да засилите разбирането си за тях. Благодаря ви за вниманието и се надявам, че сте намерили този урок за информативен.

2

Добре дошли в урок 03 **част 2** от нашия курс по програмиране на C. В този урок ще обсъдим **структурата на функциите в C**. Функциите играят решаваща роля в организирането на кода и насърчаването на повторната употреба, което ги прави основна концепция за овладяване.

		Декларация и дефиниция на функция:
		В C функцията обикновено се декларира преди първата си употреба. Декларацията определя името на функцията, връщания тип и типовете параметри (ако има такива). Ето пример за декларация на функция:
		int add(int a, int b);
		В този пример ние декларираме функция с име "add", която приема два целочислени параметъра и връща цяло число. Декларацията на функцията обикновено се поставя в заглавен файл или в началото на изходния файл.
		След като декларираме функция, трябва да дефинираме нейната реализация (имплементация). Дефиницията на функцията предоставя действителния код, който се изпълнява при извикване на функцията. Ето пример за дефиниция на функция за функцията "добавяне":
	Ex_07	int add(int a, int b) {
		int sum = a + b;
		return sum;
		}
		В този пример дефиницията на функцията започва с върнатия тип "int", последван от името на функцията и списъка с параметри, ограден в скоби. Вътре в тялото на функцията ние дефинираме логиката на функцията, която изчислява сумата от двата параметъра и връща резултата.
		Функционално повикване и връщане:
		След като функцията е декларирана и дефинирана, можем да я извикаме от други части на нашата програма. За да извикаме функция, използваме нейното име, последвано от скоби, и предоставяме необходимите аргументи (ако има такива). Ето пример за извикване на функцията "добавяне":
		int result = add(3, 5);
		В този пример извикваме функцията "add" с аргументи 3 и 5. Върнатата стойност на функцията (сумата от двете числа) се присвоява на променливата "result".

		Когато се извика функция, програмата прескача към дефиницията на функцията, изпълнява операторите в тялото на функцията и след това връща контрола на извикващия код. Операторът return се използва за указване на стойността, която да бъде върната от функцията. В предишния пример редът return sum;връща стойността на променливата "сума" обратно на повикващия.
		Функционални параметри и аргументи:
		Функциите могат да имат нула или повече параметри, които са заместители за стойности, предадени на функцията. Параметрите се декларират в скобите в декларацията и дефиницията на функцията. Ето един пример:
	Ex_08	int multiply(int x, int y) {
		int product = x * y;
		return product;
		}
		В този пример функцията "умножение" приема два целочислени параметъра, "x" и "y". В тялото на функцията тези параметри се използват за изчисляване на произведението на двете числа.
		Когато извикваме функция, предоставяме действителните стойности, наречени аргументи, които съответстват на параметрите на функцията. Ето един пример:
		int result = multiply(4, 6);
		В този пример извикваме функцията „умножение“ с аргументи 4 и 6. Функцията ще използва тези стойности като параметри „x“ и „y“ в своята реализация.
		Заключение:
		В този урок разгледахме структурата на функциите в C програмирането. Научихме за декларация и дефиниция на функция, извикване и връщане на функция, както и параметри и аргументи на функция. Функциите са мощни инструменти за организиране на код и насърчаване на повторното използване, което ни позволява да пишем модулни и ефективни програми.
		Насърчавам ви да практикувате писане на функции и да експериментирате с различни типове параметри и връщани стойности, за да подсилите разбирането си. Благодаря ви за вниманието и се надявам, че сте намерили този урок за информативен.

	3	*Добре дошли в **част 3** на урок 03 от нашия курс по програмиране на C. В този урок ще изследваме **връзката между аргументи и параметри във функциите.** Разбирането на тази връзка е от съществено значение за предаването и манипулирането на данни във функциите.*
		Параметри в декларации и дефиниции на функции:
		Параметрите са контейнери в декларациите и дефинициите на функции, които представляват стойностите, предадени на функция, когато тя бъде извикана. Те дефинират входовете, които функцията очаква да получи. Параметрите се декларират в скобите след името на функцията.
		Ето пример за декларация на функция с параметри:
		int add(int a, int b);
		В този пример функцията add( ) е декларирана с два параметъра a и b, и двата от тип int.
		*Когато дефинирате функцията, **параметрите са включени в сигнатурата на функцията** и служат като променливи в обхвата на функцията:*
	Ex_09	*int add(int a, int b) {*
		int sum = a + b;
		return sum;
		}
		В този пример параметрите a и b се използват във функцията за изчисляване на сумата от двете стойности.
		Аргументи при извиквания на функции:
		Аргументите са действителните стойности, предавани на функция, когато тя бъде извикана. Те съответстват на параметрите, декларирани в декларацията или дефиницията на функцията. Аргументите предоставят данните, с които работи функцията.
		Ето пример за извикване на addфункцията с аргументи:
		int result = add(3, 5);

В този пример стойностите 3 и 5 се предават като аргументи на add функцията. Функцията ще използва тези стойности, за да изчисли сумата.

Съответствие между аргументи и параметри:

Съответствието между аргументи и параметри се установява въз основа на тяхната позиция и типове данни. Когато се извика функция, аргументите се съпоставят с параметрите в същия ред.

Например в add декларацията на функцията int add(int a, int b) първият аргумент, предоставен в извикването на функцията, ще бъде присвоен на параметъра a, а вторият аргумент ще бъде присвоен на параметъра b.

Важно е да се гарантира, че типовете данни и редът на аргументите в извикването на функцията съответстват на типовете параметри и реда в декларацията или дефиницията на функцията. Несъответстващите типове или ред могат да доведат до грешки или неочаквано поведение.

Броят на аргументите и параметрите:

Броят на аргументите и параметрите трябва да съвпада, за да се поддържа съответствието. Ако дадена функция очаква параметри, но са предоставени по-малко аргументи, това може да доведе до грешки при компилиране.

Например, разгледайте следната декларация на функция:

```
void printMessage(char* message);
```

Ако извикаме тази функция с повече от един аргумент по този начин:

```
printMessage("Hello", "World");
```

Компилаторът ще генерира грешка, защото функцията очаква само един аргумент.

Заключение:

		В този урок изследвахме връзката между аргументи и параметри в програмирането на C. Научихме, че параметрите действат като заместители в декларациите и дефинициите на функции, дефинирайки входните данни, които функцията очаква да получи. Аргументите са действителните стойности, предавани на функция, когато тя е извикана и съответстват на параметрите.
		Разбирането на тази връзка е от решаващо значение за предаването и манипулирането на данни във функциите. Уверете се, че съвпадат броят, редът и типовете данни на аргументите и параметрите, за да гарантирате съответствието.
		Насърчавам ви да практикувате писане на функции с различни аргументи и параметри, за да подсилите разбирането си. Благодаря ви за вниманието и се надявам, че сте намерили този урок за информативен.
	4	Добре дошли в урок 03 **част 4** от курса по програмиране на C. В този урок ще се потопим в **концепцията за ПОИНТЕРИТЕ (указатели към стойности) в C**. Указателите са мощни инструменти, които ни позволяват директно да манипулираме и осъществяваме достъп до местоположения в паметта. Разбирането как да използвате указатели ефективно е от решаващо значение за напредналото програмиране на C.
		Указатели и адреси на паметта:
		Указателят е променлива, която съхранява адреса на паметта на друга променлива. Той "посочва" мястото, където действителната стойност се съхранява в паметта. Указателите се декларират с помощта на * символа (звездичка).
		Ето пример за деклариране на указател:
		int *ptr;
		В този пример ние декларираме указател с име ptr, който може да съхранява адреса на паметта на целочислена променлива.
		Присвояване и достъп до адреси на паметта:
		За да присвоим адрес на паметта на указател, използваме оператора адрес на &. Този оператор връща адреса на паметта на променлива.

		Ето един пример:
		int num = 10;
		int *ptr = #
		В този пример присвояваме адреса на паметта на променливата num към указателя ptr с помощта на & оператора.
		За достъп до стойността, съхранена на адреса на паметта, към който сочи указател, използваме оператора за дереферирane *. Този оператор извлича стойността от адреса на паметта.
		Ето един пример:
	Ex_10	int num = 10;
		int *ptr = #
		int value = *ptr;
		В този пример стойността, 10 **съхранена в адреса на паметта, посочен от ptr** се извлича и присвоява на променливата value.
		Манипулиране на стойности чрез указатели :
		Една от мощните характеристики на указателите е способността да се променят стойности директно в паметта с помощта на указателя.
		Ето един пример:
	Ex_11	int num = 10;
		int *ptr = #
		*ptr = 20;
		В този пример стойността на num първоначално е 10. Чрез дереферирane на указателя ptr и присвояване на нова стойност на 20, **ние променяме стойността директно в паметта**. Следователно стойността на num също ще бъде 20.
		Указатели като функционални аргументи:

		Указателите често се използват като аргументи на функцията за постигане на семантика за преминаване по препратка, което позволява на функциите да променят променливи директно от извикващия код.
		Ето един пример:
	Ex_12	void square(int *numPtr) {
		*numPtr = (*numPtr) * (*numPtr);
		}
		int main() {
		int num = 5;
		square(&num);
		// The value of num will be modified to 25
		return 0;
		}
		В този пример square( ) функцията приема указател към цяло число като аргумент. Чрез дереферирane на указателя и квадратиране на стойността, ние променяме оригиналната променлива num директно от функцията.
		Нека разгледаме сценарий, при който имате масив от цели числа и искате да обърнете реда на елементите в този масив с помощта на указатели. Вместо да създавате отделен обърнат масив или да използвате допълнителни временни променливи, можете да постигнете това, като манипулирате оригиналния масив с помощта на указатели.
	Ex_13	#include <stdio.h>
		void reverseArray(int* arr, int length) {
		int* start = arr;
		int* end = arr + length - 1;
		while (start < end) {
		// Swap the values pointed by start and end

```
int temp = *start;
*start = *end;
*end = temp;
// Move the pointers towards the center
start++;
end--;
}
}
int main() {
int arr[] = {1, 2, 3, 4, 5};
int length = sizeof(arr) / sizeof(arr[0]);
printf("Original Array: ");
for (int i = 0; i < length; i++) {
printf("%d ", arr[i]);
}
printf("\n");
reverseArray(arr, length);
printf("Reversed Array: ");
for (int i = 0; i < length; i++) {
printf("%d ", arr[i]);
}
printf("\n");
return 0;
}
```

В този пример reverseArray функцията приема масив (arr) и неговата дължина като параметри. Той използва два указателя start и end, инициализирани съответно към първия и последния елемент на масива.

		Чрез итерация през масива от двата края и размяна на стойностите, посочени от startu end, ние ефективно обръщаме реда на елементите. След това указателите startu endce преместват към центъра при всяка итерация, докато се срещнат.
		Чрез директно манипулиране на елементите на масива с помощта на указатели, ние избягваме нуждата от допълнителна памет или временни променливи, което води до ефективно и умно решение.
		Когато стартирате тази програма, ще видите отпечатан оригиналния масив, последван от обърнатия масив.
		Надявам се да намерите този пример за умен и проницателен в демонстрирането на силата и гъвкавостта на указателите в програмирането на C.
		Заключение:
		В този урок изследвахме концепцията за указатели към стойности в C програмирането. Научихме, че указателите съхраняват адреси на паметта и ни позволяват директно да манипулираме и осъществяваме достъп до стойности в паметта. На указателите могат да се присвояват адреси на паметта с помощта на &оператора и могат да имат достъп до стойности с помощта на *оператора. Видяхме също как указателите могат да се използват като аргументи на функции за постигане на семантика за преминаване по препратка.
		Указателите са мощни инструменти, но изискват внимателно боравене, за да се избегнат проблеми, свързани с паметта. Препоръчвам ви да практикувате използването на указатели и да се запознаете с техните свойства и действия. Благодаря ви за вниманието и се надявам, че сте намерили този урок за информативен.
	5	Добре дошли в **част 5** на урок 03 от нашия курс по програмиране на C. В този урок ще изследваме концепцията за **рекурсивни функции**. Рекурсивните функции са функции, които се самоизвикват, позволявайки елегантни и ефективни решения на проблеми, които могат да бъдат разделени на по-малки подпроблеми. Рекурсивното мислене е важно умение, което трябва да развиете като програмист.

Рекурсивна функционална структура:

Рекурсивната функция се състои от два основни компонента: основен случай и рекурсивен случай.

Базовият случай /дъно на рекурсията/ е условието, при което функцията не прави друго рекурсивно извикване и връща конкретна стойност. Той действа като критерий за спиране на рекурсията.

Рекурсивният случай е състоянието, при което функцията се самоизвиква, обикновено с модифицирана версия на оригиналния проблем. Той разбива първоначалния проблем на по-малки подпроблеми, докато стигне до основния случай.

Пример: факториелна функция:

Нека да разгледаме пример за рекурсивна функция за изчисляване на факториела на число. Факториелът на неотрицателно цяло число n, означено с n!, е произведението на всички положителни цели числа, по-малки или равни на n.

Ex_14

```
int factorial(int n) {
if (n == 0 || n == 1) {
return 1; // Base case
}
else {
return n * factorial(n - 1); // Recursive case
}
}
```

В този пример основният случай е, когато не равно на 0 или 1. В тези случаи функцията незабавно връща 1.

В рекурсивния случай функцията се умножава n по факториела на n - 1, като по този начин разбива проблема на по-малки подпроблеми. Рекурсивното извикване продължава до достигане на основния случай, в който момент резултатите се умножават заедно, за да се изчисли факториелът.

Пример: Ред на Фибоначи:

Друг класически пример за рекурсия е генерирането на последователността на Фибоначи. Последователността на Фибоначи е поредица от числа, в които всяко число е сбор от двете предходни, обикновено започващи с 0 и 1.

Ex_15

```
int fibonacci(int n) {
if (n == 0) {
return 0; // Base case
}
else if (n == 1) {
return 1; // Base case
}
else {
return fibonacci(n - 1) + fibonacci(n - 2); // Recursive case
}
}
```

В този пример основните случаи са, когато n е равно на 0 или 1. В тези случаи функцията веднага връща съответно 0 или 1.

В рекурсивния случай функцията изчислява числото на Фибоначи чрез събиране на двете предходни числа на Фибоначи (изчислени рекурсивно). Рекурсията продължава до достигане на базовите случаи.

Рекурсивен срещу итеративен подход:

Рекурсията предоставя елегантен и кратък начин за решаване на определени проблеми. Важно е обаче да се отбележи, че рекурсивните решения може да не винаги са най-ефективните.

В някои случаи итеративният (базиран на цикъл) подход може да бъде по-ефективен по отношение на паметта и производителността. Рекурсивните функции консумират пространство в стека с всяко рекурсивно извикване и прекомерната рекурсия може да доведе до грешки при препълване на стека.

Когато решавате дали да използвате рекурсия или итерация, помислете за сложността на проблема, използването на ресурси и компромисите между простота и ефективност.

Заключение:

В този урок изследвахме концепцията за рекурсивни функции в C програмирането. Научихме, че рекурсивните функции се самоизвикват, като разбиват проблемите на по-малки подпроблеми, докато достигнат базов случай. Рекурсивните функции предоставят елегантни решения за определени проблеми, като например изчисляване на факториели или генериране на последователността на Фибоначи.

Важно е да разберете основния случай и рекурсивния случай в рекурсивните функции, за да избегнете безкрайна рекурсия и да осигурите прекратяване. Освен това имайте предвид потенциалните компромиси между простота и ефективност, когато избирате между рекурсия и итерация.

Насърчавам ви да практикувате прилагането на рекурсивни функции и да изследвате как те могат да решават различни проблеми. Благодаря ви за вниманието и се надявам, че сте намерили този урок за информативен.

6

Добре дошли в урок 03 **част 6** от нашия курс по програмиране на C. В този урок ще разгледаме **типове функции и описания на прототипи на функции**. Разбирането на типове функции и прототипи е от решаващо значение за писането на добре организиран и модулен код.

Типове функции:

В програмирането на C функциите могат да бъдат класифицирани в различни типове въз основа на тяхната върната стойност и типовете на техните параметри. Нека разгледаме три често срещани типа функции:

Функции с връщане на стойност: Тези функции изпълняват някои операции и връщат стойност на повикващия. Върнатият тип е посочен в декларацията и дефиницията на функцията.

Ex_16

```
int add(int a, int b) {
return a + b;
}
```

В този пример функцията add приема два целочислени параметъра a и b и връща сумата от двете числа като цяло число.

		Функции без връщана стойност: Тези функции, известни също като void функции, не връщат никаква стойност. Те обикновено се използват за изпълнение на задачи или операции, без да е необходимо да връщат резултат.
	Ex_17	void printMessage() {
		printf("Hello, world!\n");
		}
		В този пример функцията printMessageняма никакви параметри и просто отпечатва съобщение на конзолата.
		Функции без параметри: Тези функции не изискват никакви входни параметри. Те извършват конкретно действие или изчисление и могат или не могат да върнат стойност.
	Ex_18	int getRandomNumber() {
		return rand() % 100; // Returns a random number between 0 and 99
		}
		В този пример функцията getRandomNumberняма никакви параметри и генерира и връща произволно число.
		Прототипът на функцията описва интерфейса на функцията, предоставяйки информация за нейното име, тип връщане и типове параметри. Той служи като декларация, която позволява на компилатора да провери използването на функцията преди нейното действително дефиниране.
		Прототипът на функция обикновено се появява преди главната функция или в началото на изходния файл, или в заглавния файл.
		Ето пример за прототип на функция:
	Ex_19	int add(int a, int b);
		В този пример прототипът на функцията декларира функция с име add, която приема два целочислени параметъра аи bвръща цяло число.

		Функционалните прототипи са особено полезни, когато функциите са дефинирани в отделни файлове или когато функциите се извикват една друга.
		Когато използвате функции, дефинирани в други файлове, добра практика е да включите съответния заглавен файл, който съдържа прототипите на функциите. Това гарантира, че компилаторът има необходимата информация за функциите, преди да ги използва.
		Заключение:
		В този урок разгледахме типове функции и описания на прототипи в C програмирането. Научихме за функции с връщани стойности, функции без връщани стойности (празни функции) и функции без параметри. Също така проучихме концепцията за прототипи на функции и тяхната роля в декларирането на интерфейса на функциите.
		Разбирането на типове функции и прототипи е от съществено значение за писането на добре организиран и модулен код. Той позволява правилна декларация на функции, проверка на типа и разделяне на дефиниции на функции в множество файлове.
		Насърчавам ви да практикувате използването на различни типове функции и прототипи, за да подсилите разбирането си. Благодаря ви за вниманието и се надявам, че сте намерили този урок за информативен.
	7	Добре дошли в урок 03 **част 7** от нашия курс по програмиране на C. В този урок ще се потопим в описания на **прототипи на функции**. Разбирането на функционалните прототипи е от решаващо значение за писането на модулен и добре организиран код.
		Какво е прототип на функция?
		Прототипът на функция е декларация, която описва интерфейса на функция, включително нейното име, тип на връщане и типове параметри. Той служи като план за функцията и предоставя съществена информация на компилатора преди действителната дефиниция на функцията.

Функционалните прототипи често се поставят преди основната функция в изходния файл или в заглавните файлове. Те позволяват на компилатора да проверява използването на функцията, да гарантира съвместимост на типа и да открива всякакви грешки или несъответствия в началото на процеса на компилиране.

Защо да използвате прототипи на функции?

Има няколко предимства от използването на прототипи на функции във вашия код:

Осигуряване на правилно използване на функцията: Чрез предоставяне на необходимата информация за името на функцията, типа на връщане и типовете параметри, прототипите на функции помагат да се гарантира, че функциите се използват правилно в цялата програма.

Откриване на грешки: Функционалните прототипи позволяват на компилатора да проверява за несъответствия, като несъответстващи типове аргументи или неправилни връщани типове, което може да помогне за улавяне на грешки преди изпълнение.

Разрешаване на отделна компилация: Функционалните прототипи улесняват модулното програмиране, като позволяват функциите да бъдат дефинирани в отделни файлове. Прототипите гарантират, че правилните функционални сигнатури ("подписи") са известни на други части на програмата, дори ако дефинициите на функциите са в различни файлове.

Синтаксис на прототипи на функции:

Синтаксисът за прототип на функция е подобен на декларация на функция, с пропускане на тялото на функцията. Ето един пример:

```
int add(int a, int b);
```

В този пример add е името на функцията, int е връщаният тип и int a, int b са типовете параметри. Точката и запетая в края показва, че това е декларация на прототип.

Пример: Прототипи на функции на практика:

Нека разгледаме сценарий, при който имаме програма с множество изходни файлове. За да осигурим правилно разделяне и модулност, можем да използваме функционални прототипи.

Да кажем, че имаме функция calculateAverage( ), дефинирана във файл с име statistics.c и нейния съответен прототип на функция в заглавен файл с име statistics.h. Прототипът на функцията ще изглежда така:

```
double calculateAverage(int numbers[], int length);
```

В основния изходен файл, като main.c, ще включим statistics.h заглавния файл и ще използваме calculateAverage( ) функцията, без да знаем действителното й изпълнение:

Ex_20

```
#include "statistics.h"

int main() {
int numbers[] = {5, 10, 15, 20, 25};
int length = sizeof(numbers) / sizeof(numbers[0]);

double average = calculateAverage(numbers, length);
printf("The average is: %lf\n", average);

return 0;
}
```

Като включим statistics.h заглавния файл, имаме достъп до прототипа на функцията, което ни позволява да използваме функцията calculateAverage( ) правилно във main функцията. Реалното изпълнение на calculateAverage( ) функцията може да бъде дефинирано във statistics.c файла.

Заключение:

В този урок разгледахме значението на прототипите на функции в програмирането на C. Научихме, че прототипите на функциите служат като декларации, които описват интерфейса на функцията, включително нейното име, тип връщане и типове параметри. Функционалните прототипи помагат да се гарантира правилното използване на функцията, откриват грешки и позволяват отделно компилиране.

Чрез използването на функционални прототипи можете да пишете модулен и добре организиран код, който насърчава повторното използване и поддръжката. Насърчавам ви да практикувате използването на прототипи на функции във вашите програми, за да подобрите разбирането си за тази основна концепция.

Благодаря ви за вниманието и се надявам, че сте намерили този урок за информативен.

8

Добре дошли в **част 8** на урок 03 от нашия курс по програмиране на C. В този урок ще изследваме концепциите за **динамични и статични локални променливи във функции и блокове**. Разбирането на разликите между тези променливи е от съществено значение за управлението на паметта и контролирането на обхвата на променливите във вашите програми.

Статични локални променливи :

Статичните локални променливи са променливи, декларирани във функция или блок, които запазват стойностите си между извикванията на функция. Те се инициализират само веднъж и стойностите им се запазват при множество извиквания на функции.

Ето един пример:

Ex_21

```
void countCalls() {
static int counter = 0;
counter++;
printf("Function called %d times\n", counter);
}

int main() {
countCalls(); // Function called 1 times
countCalls(); // Function called 2 times
countCalls(); // Function called 3 times
return 0;
}
```

В този пример статичната локална променлива counter се инициализира на 0, когато функцията е извикана за първи път. С всяко следващо извикване на функция, стойността на counter се увеличава и се запазва при извиквания на функция.

Статичните локални променливи са полезни, когато искате да запазите информация между извикванията на функции, като например поддържане на състояние на преброяване или проследяване.

Динамични локални променливи :

Динамичните локални променливи са променливи, разпределени с помощта на функции за динамично разпределение на паметта като malloc, calloc или realloc. Тези променливи не се освобождават автоматично, когато функцията или блокът приключи, и техният живот се простира извън обхвата на функцията или блока.

Ето един пример:

Ex_22

```
void createDynamicArray() {
int* arr = malloc(5 * sizeof(int));
// Use the dynamically allocated array
free(arr);   // Deallocate the memory when no longer needed
}

int main() {
createDynamicArray();
return 0;
}
```

В този пример динамичната локална променлива arr е разпределена с помощта на malloc. Паметта за масива продължава дори след края на функцията createDynamicArray . Важно е да освободите паметта, free когато вече не е необходима, за да избегнете изтичане на памет.

Динамичните локални променливи са полезни, когато трябва да разпределите динамично паметта и да имате контрол върху нейния живот.

Статични срещу динамични локални променливи :

Статичните локални променливи и динамичните локални променливи имат някои прилики, но се различават по важни аспекти:

Живот: Статичните локални променливи имат живот, който обхваща цялото изпълнение на програмата, като запазва стойностите си при извиквания на функции. Динамичните локални променливи имат по-дълъг живот от функцията или блока, в който са дефинирани, но изискват ръчно освобождаване.

Разпределяне на памет: Статичните локални променливи се разпределят автоматично в паметта и се съхраняват на фиксирано място. На динамичните локални променливи се разпределя памет в купчината с помощта на функции за динамично разпределение на паметта и могат да бъдат преоразмерени или освободени ръчно.

Обхват: както статичните, така и динамичните локални променливи са достъпни само в рамките на функцията или блока, в който са дефинирани. Те не са видими или достъпни извън своя обхват.

Заключение:

В този урок изследвахме концепциите за динамични и статични локални променливи в програмирането на C. Статичните локални променливи запазват стойностите си при множество извиквания на функции, докато динамичните локални променливи се разпределят в купчината и имат по-дълъг живот.

Разбирането на разликите между динамичните и статичните локални променливи е от решаващо значение за управлението на паметта и контролирането на обхвата на променливите във вашите програми. Позволява ви да използвате ефективно паметта и да запазвате необходимата информация.

Насърчавам ви да практикувате използването на динамични и статични локални променливи във вашите програми, за да затвърдите разбирането си за тези концепции. Благодаря ви за вниманието и се надявам, че сте намерили този урок за информативен.

	9	Добре дошли в урок 03 **част 9** от нашия курс по програмиране на C. В този урок ще изследваме ролята на **програмния стек в програмирането на C**. Разбирането на програмния стек е от решаващо значение за разбирането как работят функциите, управлението на извикванията на функции и обработката на локални променливи.
		Какво представлява програмният стек?
		Програмният стек, известен също като **стек за повиквания или стек за изпълнение**, е област от паметта, използвана от програмата за проследяване на извиквания на функции, локални променливи и адреси за връщане. Работи на принципа „**Последен влязъл, първи излязъл**“ (LIFO).
		Когато се извика функция, нейните локални променливи и адресът за връщане се изпращат в стека. Докато функцията се изпълнява, на локалните променливи се разпределя памет в стека, а извикванията на функции в тази функция се подреждат отгоре.
		Когато дадена функция завърши своето изпълнение, нейните локални променливи се освобождават и адресът за връщане се изважда от стека, което позволява на програмата да продължи изпълнението от мястото, където е спряла.
		Рамки за извикване на функция и стек:
		Всяко извикване на функция създава стекова рамка, известна още като запис за активиране или стекова рамка. Рамката на стека съдържа локалните променливи, параметри и адрес за връщане на функция. Той представлява състоянието на функцията в определен момент от изпълнението на програмата.
		Когато се извика функция, се създава нова рамка на стека и се избутва в стека. Рамката на стека включва локалните променливи на функцията, параметрите и адреса за връщане. Тъй като се правят повече извиквания на функции, нови стекови рамки се добавят върху предишните.
		Когато дадена функция завърши изпълнението си, нейната стекова рамка се премахва от горната част на стека и програмата продължава с предишната стекова рамка.
		Пример: Разбиране на програмния стек:
		Нека разгледаме един пример, за да разберем програмния стек в действие:

Ex_23

```
void outerFunction() {
int outerVariable = 10;
innerFunction(); // Calling innerFunction()
}

void innerFunction() {
int innerVariable = 20;
}

int main() {
outerFunction();
return 0;
}
```

В този пример, когато се извика outerFunction , за него се създава нова стекова рамка. Разпределя се outerVariable памет в рамките на този стеков кадър. Когато се извика innerFunction от рамките на outerFunction, друг кадър на стека се създава върху предишния. Разпределя се innerVariable памет в тази нова стекова рамка.

Докато програмата се изпълнява, стековите рамки се избутват и изваждат от стека, управлявайки локалните променливи и потока на изпълнение.

Значение на програмния стек :

Разбирането на програмния стек е от решаващо значение поради няколко причини:

Управление на извиквания на функции: Програмният стек гарантира, че функциите се извикват и връщат в правилния ред, поддържайки потока на изпълнение.

Работа с локални променливи: Стекът осигурява разпределение на паметта за локални променливи, което позволява на всяка функция да има свой собствен набор от променливи без намеса от други функции.

Поддръжка на рекурсия: Програмният стек играе жизненоважна роля в поддържането на рекурсивни извиквания на функции чрез създаване и управление на множество стекови рамки.

		Ефективно управление на паметта: Естеството на LIFO на стека позволява ефективно разпределяне и освобождаване на паметта, тъй като кадрите на стека се избутват и изваждат при необходимост.
		Заключение:
		В този урок изследвахме ролята на програмния стек в програмирането на C. Програмният стек е област от паметта, която управлява извиквания на функции, локални променливи и адреси за връщане. Той работи на принципа LIFO, създавайки и управлявайки стекови рамки за всяко извикване на функция.
		Разбирането на програмния стек е от решаващо значение за разбирането как работят функциите, управлението на извикванията на функциите и ефективната обработка на локалните променливи. Той ви позволява да разберете потока на изпълнение и гарантира правилното управление на паметта във вашите програми.
		Препоръчвам ви да проучите повече за програмния стек и неговото въздействие върху извикванията на функции и управлението на паметта. Благодаря ви за вниманието и се надявам, че сте намерили този урок за информативен.
	10	Добре дошли в урок 03, **част 10** от нашия курс по програмиране на C. В този урок ще разгледаме концепцията за **„Указатели към функции“**. Указателите към функции са мощни конструкции, които ни позволяват да съхраняваме и манипулираме функции като данни в C, осигурявайки гъвкавост и позволявайки съвременни техники за програмиране.
		Какво представляват указателите към функциите?
		В програмирането на C функциите са блокове от код, които могат да бъдат извикани и изпълнени. Указателите са променливи, които съхраняват адреси в паметта на данни.
		Указателите към функции, както подсказва името, са променливи, които съхраняват адреси в паметта на функции вместо на данни.
		Използвайки указатели към функции, можем да третираме функциите като ги предаваме като аргументи на други функции, връщаме ги от функциите и дори динамично избираме коя функция да извикаме по време на изпълнение.

		Деклариране и използване на указатели към функции:
		За да декларирате указател към функция, трябва да посочите връщания тип на функцията и нейните типове параметри. Ето един пример:
	Ex_24	int (*addPtr)(int, int); // Pointer to a function that returns an int and takes two int parameters
		В този пример addPtr е указател към функция, която приема два int параметъра и връща int.
		За да присвоите адреса на функция към указателя, можете директно да използвате името на функцията (без скоби), което автоматично ще доведе до указател на функция:
		int add(int a, int b) {
		return a + b;
		}
		addPtr = add; // Assigning the address of the function add/без скоби/ to addPtr
		Сега, когато addPtr съдържа адреса на add функцията, можете да го използвате, за да извикате функцията:
		int result = (*addPtr)(3, 5); // Calling the function add using the function pointer
		Като алтернатива можете да използвате съкратена нотация, за да извикате функцията чрез показалеца:
		int result = addPtr(3, 5); // Shorthand notation to call the function add using the function pointer
		Използване на указатели към функции като обратни извиквания:
		Едно от най-мощните приложения на указатели към функции е използването им като обратни извиквания. Обратното извикване е функция, която се предава като аргумент на друга функция и се изпълнява в контекста на тази функция.
		Ето пример за използване на указател към функция като обратно извикване:
	Ex_25	void process(int data, int (*callback)(int)) {
		int result = callback(data);

```
printf("Result: %d\n", result);
}

int doubleNumber(int num) {
return num * 2;
}

int main() {
process(5, doubleNumber); // Calling process and passing doubleNumber as a callback
return 0;
}
```

В този пример process функцията приема цяло число data и указател към функция callback. Вътре process извикваме функцията за обратно извикване и обработваме нейната върната стойност.

Когато извикваме process(5, doubleNumber), предаваме 5 като данни и doubleNumber като функция за обратно извикване. Функцията doubleNumber се изпълнява в контекста на process функцията, като ефективно удвоява числото и отпечатва резултата.

Заключение:

В този урок изследвахме концепцията за "указатели към функции" в програмирането на C. Указателите към функции ни позволяват да третираме функциите като първокласни граждани, осигурявайки гъвкавост и позволявайки съвременни техники за програмиране.

Научихме как да декларираме и използваме указатели към функции, да присвояваме функционални адреси на указатели и да извикваме функции чрез указатели. Видяхме също как указателите към функции могат да се използват като мощни обратни извиквания, предавайки функции като аргументи на други функции.

Използването на указатели към функции отваря широка гама от възможности за създаване на по-гъвкав и модулен код. Насърчавам ви да изследвате по-нататък и да експериментирате с указатели към функции във вашите собствени програми.