МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Н. Д. Любашенко

ПРОГРАМУВАННЯ-2.
Мова С

КОНСПЕКТ ЛЕКЦІЙ

*Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для студентів,
які навчаються за спеціальністю 113 «Прикладна математика»,
освітньою програмою «Наука про дані (Data Science) та математичне
моделювання»*

Київ
КПІ ім. Ігоря Сікорського
2019

ПРОГРАМУВАННЯ-2. Мова С [Електронний ресурс] : навч. посіб. для студ. спеціальності 113 «Прикладна математика», освітньої програми «Наука про дані (Data Science) та математичне моделювання» / КПІ ім. Ігоря Сікорського ; уклад.: Н. Д. Любашенко. – Електронні текстові дані (1 файл: 1,6 Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2019. – 144 с.

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського (протокол №6 від 31.01.2020 р.)
за поданням Вченої ради факультету прикладної математики (протокол № 8 від 27.01.2020 р.)

Електронне мережне навчальне видання

ПРОГРАМУВАННЯ-2.
МОВА С

Укладачі: *Любашенко Наталія Дмитрівна*, старший викладач

Відповідальний редактор: *Сирота С.В.*, канд. техн. наук, доц.

Рецензенти: *Пакриш О.Є.*, канд. техн. наук, доц.
Віштань Д.М., канд. техн. наук, старш. наук. співроб.

Посібник адресований студентам закладів вищої освіти зі спеціальності «Прикладна математика», спеціалізації «Наука про дані (Data Science) та математичне моделювання». Матеріал відповідає змісту міжнародного стандарту ISO/IEC з мови С, при цьому форма його подання є наочною. Посібник складається з 16-ти розділів. Детально розглянуто концепції та конструкції мови С. Особливу увагу приділено концепції типізації даних та функцій, деклараціям об'єктів та функцій. Окремі розділи присвячено середовищу трансляції та виконання, а також питанню якнайширшого використання потужної стандартної бібліотеки С-програм. Для кращого сприйняття формальних описів наводяться фрагменти програм та відповідні пояснення. У кінці кожного розділу містяться контрольні питання. У додатках знаходяться словник термінів та інформація про стандартні заголовки С-бібліотеки.

Викладений матеріал відповідає діючим стандартам та навчальним програмам з дисципліни «Програмування».

© КПІ ім. Ігоря Сікорського, 2019

З М І С Т

В С Т У П **6**

Лекція 1. Актуальність мови С. Огляд конструкцій та концепцій мови С. Приклад програми на мові С **8**

Актуальність мови С *8*
Об'єкт. Функція *10*
Огляд конструкцій мови С *11*
Огляд концепцій мови С *11*
Базовий приклад *16*

Лекція 2. Лексичні елементи (частина 1). Декларації та визначення даних і функцій (частина 1) **22**

Лексичні елементи. Ключові слова *22*
Лексичні елементи. Ідентифікатори *22*
Лексичні елементи. Універсальні імена символів *24*
Декларації та визначення об'єктів, функцій *25*

Лекція 3. Концепції мови С (частина 1) **29**

Концепція типу *29*
Класифікація типів *29*
Похідний тип масив *32*
Похідний тип вказівник *34*
Робота з вказівниками *34*
Похідний тип структура *39*

Лекція 4. Вирази. Перетворення і кастинг типів **41**

Вирази в мові С *41*
Пріоритет операторів. Асоціативність операторів. Порядок обчислень у виразах *42*
Первинні вирази *44*
Постфіксні оператори *44*
Унарні оператори *48*
Перетворення і кастинг типів *51*

Лекція 5. Декларації та визначення даних і функцій (частина 2). Лексичні елементи (частина 2) **53**

Правило "праворуч-ліворуч" *53*
Кваліфікатори типу *55*
Лексичні елементи. Константи *56*

Лекція 6. Функції в мові С **62**

Декларації та визначення функцій *62*
Механізм передачі аргументів, повернення результату *63*
Виконання виклику функції *64*
Вказівники як параметри функції *68*
Повернення вказівника з функції *69*

Вказівники на функції ... 69
Бібліотека функцій мови С. Стандартні заголовки. Функції управління пам'яттю ... 70

Лекція 7. Введення / виведення даних (частина 1) ... **73**

Заголовок <stdio.h> ... 73
Потоки. Файли ... 74
Операції над файлами ... 77
Функції доступу до файлів ... 77
Функції форматованого введення та виведення ... 79
Функції символьного введення та виведення ... 80

Лекція 8. Твердження та блоки ... **82**

Твердження в мові С ... 82
Складене твердження (блок) ... 82
Твердження-вирази та пусті твердження ... 82
Твердження вибору ... 83
Твердження ітерації ... 84
Твердження швидкого переходу ... 86
Твердження з міткою та goto ... 87

Лекція 9. Директиви препроцесора (частина 1) ... **89**

Директиви препроцесора в мові С ... 89
Умовна підстановка ... 89
Включення початкового файлу ... 91
Макропідстановки ... 92

Лекція 10. Введення та виведення даних (частина 2) ... **94**

Функції прямого введення та виведення ... 94
Функції позиціювання у файлах ... 94
Функції обробки помилок ... 95

Лекція 11. Середовище програмування ... **97**

Структура С-програми ... 98
Запуск С-програми. Функція main ... 99
Припинення програми ... 100

Лекція 12. Виконання програми ... **101**

Поведінка програми ... 101
Побічні ефекти ... 102
Точки упорядкованості ... 104

Лекція 13. Концепції мови С (частина 2) ... **107**

Концепція меж дії ідентифікаторів ... 107
Концепція з'єднання ідентифікаторів ... 109
Концепція простору імен ... 114
Концепція тривалості зберігання об'єктів ... 115

Лекція 14. Декларації та визначення даних і функцій (частина 3) ... **117**

Специфікатори класу зберігання ... *117*
Декларатори ... *121*
Імена (назви) типів ... *121*
Статичні судження ... *122*
Зовнішні визначення ... *122*
Лекція 15. Директиви препроцесора (частина 2) ... 125
Управління рядком ... *125*
Директива помилки ... *125*
Директива Pragma ... *126*
Пуста директива ... *128*
Зарезервовані імена макросів ... *128*
Оператор Pragma ... *129*
Лекція 16. Лексичні елементи (частина 3). Концепції мови С (частина 3) ... 130
Лексичні елементи. Літерали – рядки ... *130*
Лексичні елементи. Пунктуатори ... *130*
Лексичні елементи. Імена заголовків ... *130*
Лексичні елементи. Препроцесингові числа ... *131*
Лексичні елементи. Коментарі ... *131*
Концепція представлення типів ... *132*
Концепція сумісного типу та складеного типу ... *132*
Концепція вирівнювання об'єктів ... *133*
Додаток 1. Словник стандартних термінів ... 135
Додаток 2. С-бібліотека ... 142
Література ... 144

В С Т У П

Універсальна мова програмування С вражає живучістю. Вона не просто залишилась актуальним засобом системного програмування, а й розвинулась під впливом сучасних вимог [1]. Нею користуються як для розробки програм на низьких рівнях, наприклад, драйверів пристроїв, операційних систем, апаратних прошивок, так і прикладних програм високого рівня.

Програмісту-початківцю мова програмування С може видатись недружелюбною, громіздкою. Проте після певних зусиль програміст починає змінювати думку і по-новому оцінювати С. Тепер мова стає гнучким, ефективним, універсальним інструментом реалізації програмних проектів.

Мовою С опікуються солідні міжнародні організації з уніфікації програмних і технічних засобів: ISO (the International Organization for Standardization) та IEC (the International Electrotechnical Commission). Вони виробили поточний стандарт для С [2]. Мета стандарту – сприяти мобільності, надійності, ефективному відлагодженню і виконанню С-програм в різноманітних обчислювальних системах.

Цей навчальний посібник складено на основі лекцій та лабораторних занять, які читаються студентам денної форми навчання спеціальності 113 “Прикладна математика” освітньо-кваліфікаційного рівня “Бакалавр”.

Особливістю посібника є те, що його структура максимально наближена до оригінального тексту останньої версії стандарту ISO/IEC мови С [2]. При цьому матеріал довідкового характеру подано у зручній табличній формі.

При складанні посібника було враховано термінологічні розбіжності в книгах та онлайн-ресурсах по мові С. Авторський варіант перекладів та визначень термінів подано в словнику (додаток 1).

Базовий приклад, наведений в кінці першої лекції, – це приклад програми, посилання на яку міститимуться у наступних лекціях для ілюстрації деяких конструкцій та концепцій мови С.

Кожна лекція подана в окремому розділі, містить як теоретичний матеріал, так і достатню кількість прикладів. У кінці кожного розділу наведені контрольні питання.

На рисунку зображено узагальнену ілюстрацію по вивченню мови С.

В таблиці розміщено тематику відповідних лекцій.

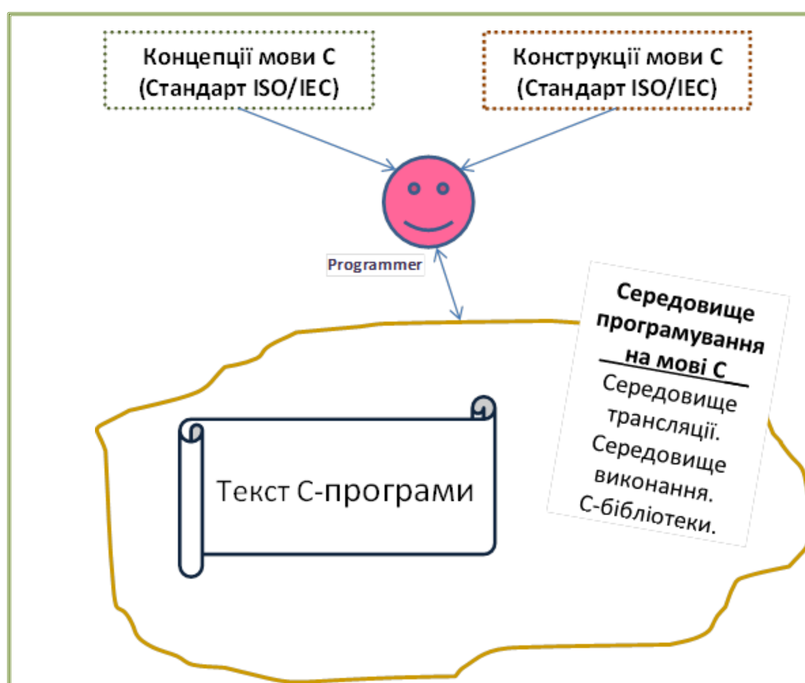


Рисунок – Ознайомлення програміста з мовою С

Таблиця – Теми лекцій з програмування на мові С

<table>
<tr><th colspan="6">Тематика лекцій з мови програмування С</th></tr>
<tr><th>Вступ</th><th colspan="3">Середовище С-програмування</th><th>Конструкції мови С</th><th>Концепції мови С</th></tr>
<tr><td rowspan="2">Стандарт ISO/IEC 9899:2018 – Information technology – Programming languages – C.</td><td>Середовище трансляції</td><td>Середовище виконання</td><td>С-бібліотеки</td><td rowspan="2">Лексичні елементи.
Декларації об’єктів та функцій.
Вирази.
Твердження та блоки.
Директиви препроцесора.</td><td rowspan="2">Об’єкт. Функція.
Пріоритет операторів.
Асоціативність операторів.
Порядок обчислень у виразах.
Межі дії ідентифікаторів.
З’єднання ідентифікаторів.
Простори імен.
Тривалість зберігання об’єктів.
Типи.
Перетворення типів.
Представлення типів.
Сумісний тип та складений тип.
Вирівнювання об’єктів.</td></tr>
<tr><td colspan="3">Структура програми.
Запуск програми.
Функція main.
Поведінка програми.
Побічні ефекти.
Точки упорядкованості.
Припинення програми.
С-бібліотеки.
Введення та виведення даних.
Управління оперативною пам’яттю.</td></tr>
</table>

Лекція 1. Актуальність мови С. Огляд конструкцій та концепцій мови С. Приклад програми на мові С

Актуальність мови С

На рис. 1.1 показано беззаперечні переваги мови С. Це, перш за все, ефективність та швидкість коду, яка не притаманна багатьом іншим мовам. Також слід відначити широку типізацію даних, переносимість на різні платформи, потужну бібліотеку програм.

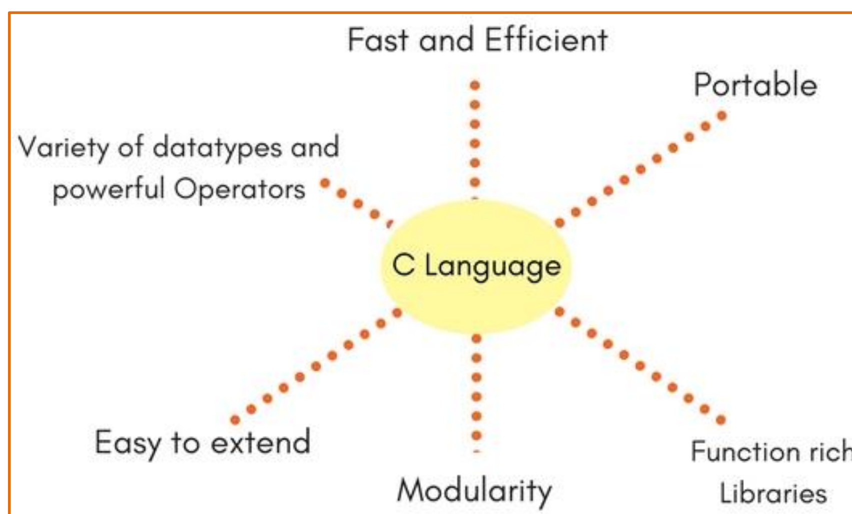


Рисунок 1.1 – Переваги мови С

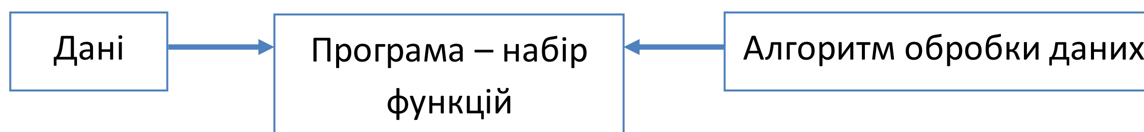
Мова С реалізує парадигму процедурного програмування.

Основною ідеєю парадигми процедурного програмування є команда. Виконання кожної команди змінює стан середовища виконання програми, наприклад, вміст оперативної пам’яті. При цьому результат роботи програми залежить від порядку виконання команд. Якщо команди поміняти місцями, результат зміниться.

Обробка даних організується програмістом як виконання набору процедур. Процедура є логічною одиницею алгоритму. Дані, які обробляються процедурами, виокремлюються ще на етапі алгоритмізації, і програміст повинен піклуватися про них не менше, ніж про сам алгоритм. В мові С процедура називається функцією.

Реалізація алгоритму зводиться до викликів функцій, передачі їм потрібних даних – аргументів, отримання в місці виклику функції результату її виконання, подальшої обробки цього результату.

Схематично цей процес можна зобразити так:



В тексті типової С-програми присутні типові частини (табл. 1.1). Це, по-перше, документація, яка повинна супроводити будь-який програмний продукт. Далі йдуть твердження препроцесора. Третя частина містить зовнішні декларації для об'єктів, які доступні в інших трансляційних одиницях. Далі бачимо функції автора програми. Нарешті, йде функція main, з якої починається виконання С-програми. Відповідний приклад наведено в кінці лекції, підрозділ "Базовий приклад".

Таблиця 1.1 – Частини типової С-програми

Секція документації по програмі
Директиви препроцесора
Секція зовнішніх визначень об'єктів та прототипів функцій
Визначення функцій Декларації Твердження та блоки
Функція main() Декларації Твердження та блоки

Згідно стандарту, реалізація (імплементація) мови С - це два середовища.

Одне середовище називають середовищем трансляції. В ньому можна виконати трансляцію початкового файлу .c, компоновку об'єктних модулів .obj у виконуваний файл .exe.

Друге середовище називають середовищем виконання С-програми, і в ньому відбувається безпосереднє виконання програми.

Щоб написати текст програми на мові С і успішно відлагодити цю програму, програмісту доведеться, крім мовних концепцій та конструкцій, звернути увагу на особливості середовища, в якому виконуватиметься програма. Можливо, доведеться читати супровідну документацію.

На рис. 1.2 зображено етапи обробки С-програми в середовищі програмування.

Для роботи ви можете обрати серед сучасних інтегрованих середовищ розробки (IDE) С-програм:

- Eclipse CDT (C/C++ Development Tooling);
- Code::Blocks;
- GNAT Programming Studio (GPS);
- Visual Studio Code;
- CodeLite IDE;
- Netbeans C++ IDE.

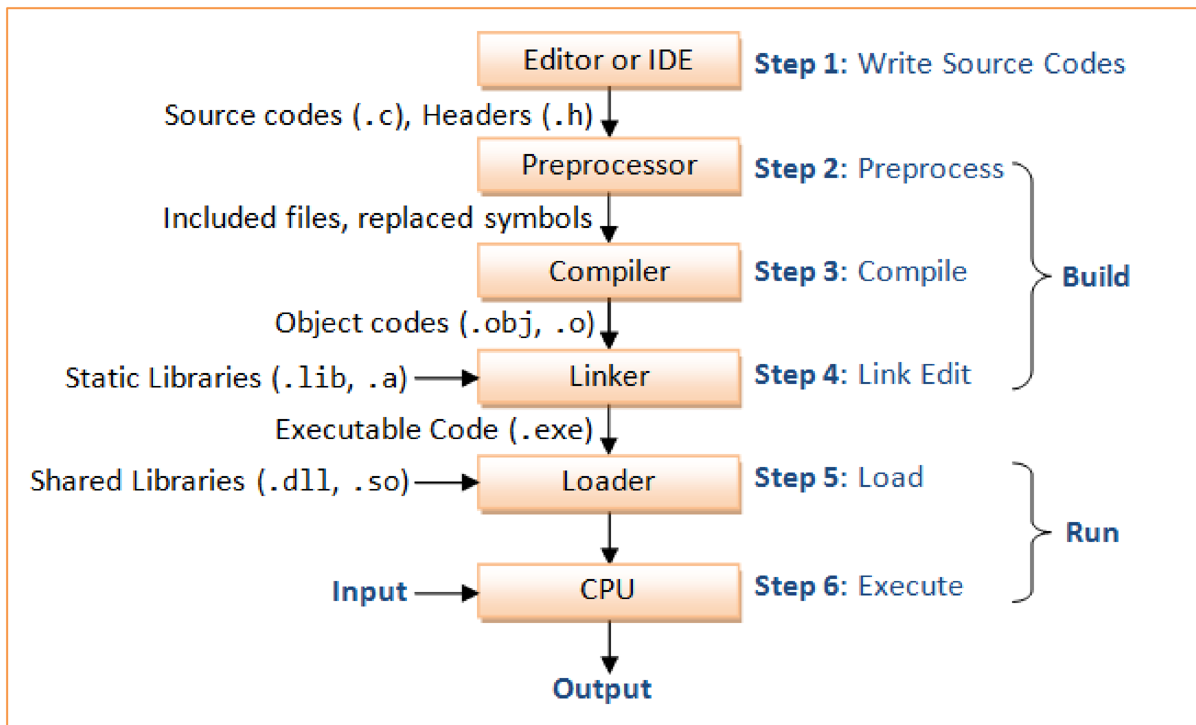


Рисунок 1.2 – Етапи обробки С-програми

Об'єкт. Функція

Щоб створювати ефективні програми, програміст перш за все має ретельно ознайомитись з концепціями мови та її конструкціями.

Першими поняттями, які треба засвоїти С-програмісту, є об'єкт та функція.

Об'єкт в мові С – це область зберігання даних в середовищі виконання, вміст якої може бути конкретним значенням. Якщо адресуватись до об'єкта, тоді об'єкт можна інтерпретувати як такий, що має певний тип.

В стандарті С поняття об'єкта узагальнило декілька понять, зокрема, змінної, масиву, структури, віднісши останні до категорії типів.

Функція - це автономний блок тверджень мови С для розв'язання якоїсь задачі, при цьому твердження формуються з об'єктів. Кожна С-програма складається хоча б з однієї функції, яка має стандартну назву main.

Переваги використання функцій:

- окрему функцію можна розробляти і переробляти відповідно до потреб;
- функція забезпечує код багаторазового використання;
- полегшується налагодження та редагування великих програм, якщо вони мають модульну структуру.

Огляд конструкцій мови С

В довідковій таблиці (табл. 1.2), що відповідає стандарту [2], наведено всі конструкції мови С - граматичні ‘цеглини’, з яких будується текст С-програми, яка реалізує той чи інший алгоритм обробки тих чи інших даних.

Цими граматичними “цеглинами” є:

- директиви препроцесора (п’ятий стовпець),
- послідовність тверджень та блоків (третій стовпець), які відтворюють алгоритм,
- вирази (другий стовпець), які може написати програміст у твердженнях,
- для всіх даних та функцій програміст має надати їх опис через декларації та зовнішні визначення (четвертий стовпець),
- лексеми (перший стовпець), тобто елементарні мовні елементи, з яких будуються всі перелічені вище конструкції.

Огляд концепцій мови С

Щоб створювати ефективні програми, програміст перш за все має ретельно ознайомитись з концепціями мови, які покладені в основу організації обробки даних С-програмою. Наступна таблиця (табл. 1.3) містить перелік концепцій, тобто властивостей мови, її характерних особливостей застосування.

Таблиця 1.2 – Конструкції мови С

Лексичні елементи (Токени) Lexical elements (Tokens)	**Вирази** Expressions **Константні вирази** Constant expressions	**Твердження та блоки** Statements and blocks	**Декларації** Declarations **Зовнішні визначення** External definitions	**Директиви препроцесора** Preprocessing directives
Ключові слова Keywords ************ **Ідентифікатори** Identifiers ************ **Універсальні імена символів** Universal character names ************ **Константи** Constants Цілі константи Integer constants Константи з плаваючою точкою Floating constants Константи переліку Enumeration constants Символьні константи Character constants ************ **Літерали - рядки** String literals ************ **Пунктуатори** (роздільники)	**Первинні вирази** Primary expressions Ідентифікатор Identifier Константа Constant Рядок- літерал String-Literal (Вираз) (Expression) Загальний відбір Generic-selection ************ **Постфіксні оператори** Postfix operators Індексація масиву Array subscripting Виклики функції Function calls Елементи структури і об'єднання Structure and union members	**Твердження з міткою** Labeled statements ************ **Складене твердження (блок)** Compound statement (block) ************ **Твердження-вирази та пусті твердження** Expression and null statements ************ **Твердження вибору** Selection statements Твердження if The if statement Твердження switch The switch statement ************ **Твердження ітерації** Iteration statements Твердження while The while statement Твердження do The do statement	**Специфікатори класу зберігання** Storage-class specifiers typedef, extern, static, _Thread_local, auto, register ************ **Специфікатори типу** Type specifiers void, char, short, int, long, float, double, unsigned, signed, _Bool, _Complex Специфікатори структури та об'єднання Structure and union specifiers struct, union Специфікатори переліку Enumeration specifiers enum Теги (бирки) Tags	**Умовна підстановка** Conditional inclusion #if, #else, #elif, #ifdef, #ifndef, #endif, #if defined ************ **Включання початкового файлу** Source file inclusion #include ************ **Макропідстановки** Macro replacement ************ **Управління рядком** Line control ************ **Директива помилки** Error directive ************ **Директива Pragma** Pragma directive ************ **Пуста директива** Null directive ************ **Зарезервовані імена макросів**

Punctuators ************ **Імена заголовків** Header names ************ **Препроцесингові числа** Preprocessing numbers ************ **Коментарі** Comments ************	Оператори постфіксних інкремента і декремента Postfix increment and decrement operators Складені літерали Compound literals ************ **Унарні оператори** Unary operators Оператори префіксних інкремента і декремента Prefix increment and decrement operators Оператори адреси та вмісту за адресою Address and indirection operators Унарні арифметичні оператори Unary arithmetic operators Оператори sizeof та _Alignof The sizeof and _Alignof operators ************ **Оператори кастингу (перетворення)** Cast operators ************ **Мультиплікативні оператори** Multiplicative operators ************ **Аддитивні оператори** Additive operators ************ **Оператори побітового зсуву** Bitwise shift operators ************	Твердження for The for statement ************ **Твердження швидкого переходу** **Jump statements** Твердження goto The goto statement Твердження continue The continue statement Твердження break The break statement Твердження return The return statement ************	Специфікатори атомного типу Atomic type specifiers _Atomic Визначення типу Type definitions typedef ************ **Кваліфікатори типу** Type qualifiers const, restrict, volatile, _Atomic ************ **Специфікатори функції** Function specifiers Inline, _Noreturn ************ **Специфікатори вирівнювання** Alignment specifier _Alignas ************ **Декларатори** Declarators Прямий декларатор Direct declarator Декларатор масиву Array declarators	Predefined macro names ************ **Оператор Pragma** Pragma operator ************

	Оператори відношення Relational operators ************ **Оператори рівності** Equality operators ************ **Оператор побітового І** Bitwise AND operator ************ **Оператор побітового виключного АБО** Bitwise exclusive OR operator ************ **Оператор побітового включного АБО** Bitwise inclusive OR operator ************ **Логічний оператор І** Logical AND operator ************ **Логічний оператор АБО** Logical OR operator ************ **Умовний оператор** Conditional operator ************ **Оператори присвоєння** Assignment operators ************ **Оператор кома** Comma operator ************ **Константні вирази** Constant expressions		Декларатор функції Function declarators Декларатор вказівника Pointer declarators ************ **Імена (назви) типів** Type names ************ **Ініціалізація (присвоєння початкових значень)** Initialization ************ **Статичні судження** Static assertions _Static_assert ************ **Зовнішні визначення** External definitions Визначення функції Function definitions Зовнішні визначення об’єкта External object definitions	

Таблиця 1.3 – Концепції мови С

Назва концепції Concepts	**Опис концепції**
Межі дії ідентифікаторів Scopes of identifiers	Для кожного об'єкта, який позначається ідентифікатором, цей ідентифікатор є видимим (тобто може бути використаний) тільки в частині тексту програми, яка називається межами дії ідентифікатора. Існує чотири види меж дії, або видимості: - межі функції, - межі файлу, - межі блоку, - межі прототипу функції.
З'єднання ідентифікаторів Linkages of identifiers	Ідентифікатор, оголошений в різних межах або більше одного разу в одніх межах, може, тим не менш, посилатись на один і той же об'єкт або функцію за допомогою процесу з'єднання. Ідентифікатори класифікуються як з'єднані зовнішньо, з'єднані внутрішньо або не з'єднані.
Простори імен Name spaces of identifiers	Якщо в програмі наявні однакові ідентифікатори для різних об'єктів, то для забезпечення однозначності їх співставлення існує механізм просторів імен.
Тривалість зберігання об'єктів Storage durations of objects	Період, протягом якого під час виконання програми за об'єктом зберігається шматок пам'яті.
Типи Types	Смисл значення, що зберігається в об'єкті або повертається функцією, визначається типом виразу, який використовується для доступу до цього значення. Найпростішим таким виразом є ідентифікатор, тип вказується в декларації ідентифікатора.
Представлення типів Representations of types	Машинне представлення різних типів не специфікується стандартом С, за винятком кількох випадків.
Сумісний тип та складений тип Compatible type and composite type	Сумісність між типами означає схожість двох типів один з одним. Сумісність типів є важливою в процесі перетворення типів і виконання операцій. С-компілятор зводить сумісні типи до одного, так званого складеного типу.
Вирівнювання об'єктів Alignment of objects	Вирівнювання - це визначене реалізацією ціле число, яке дорівнює кількості байтів між послідовними адресами оперативної пам'яті для даних програми. Іншими словами, вирівнювання обмежує адреси пам'яті, які можна віддати для об'єктів програми.

Зауваження. Концепція меж дії - це властивість, оброблювана компілятором, тоді як концепція з'єднання є властивістю, яку обробляє лінкер, а концепція тривалості зберігання стосується етапу виконання програми.

Концепція мови С	**Етап обробки програми**
Межі дії ідентифікаторів	Компіляція
З'єднання ідентифікаторів	Лінкування (редагування зв'язків)
Тривалість зберігання об'єктів	Виконання

Базовий приклад

Програма хешування даних з усуненням колізій

Постановка задачі.

Хеш-таблиця - це абстрактна структура даних у вигляді пар (ключ, значення). Потрібна для організації швидкого доступу до даних, які зберігаються у так званих відерцях.

Потрібне значення можна знайти в масиві відерець за допомогою хеш-функції, яка обчислює за ключем відповідний індекс.

На рис. 1.3 показано телефонний довідник як хеш-таблиця. Її елементами є пари (ім'я абонента, номер телефону).

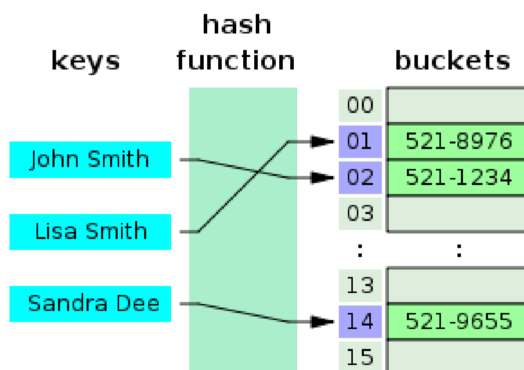


Рисунок 1.3 – Хешування

При хешуванні можуть виникати колізії, коли різним даним відповідає однакове значення хеш-функції. Як обробити колізію?

Розв'язання.

Можна вирішити цю проблему за допомогою зв'язаного списку, який зберігатиме дані при виникненні колізії.

Що таке список як форма організації даних в програмуванні?

На рис. 1.4 зображено зв'язаний список з 3-х елементів. Кожний елемент (node) – вузол списку - містить не тільки якісь дані (умовно позначено info), але й адресу наступного елемента (link). Таким чином, цей список займає не неперервний шматок пам'яті, а розкиданий по області доступної пам'яті. Голова списку (head pointer) – це елемент, який містить тільки адресу першого елемента списку (head node). Останній елемент містить пустий вказівник (null).

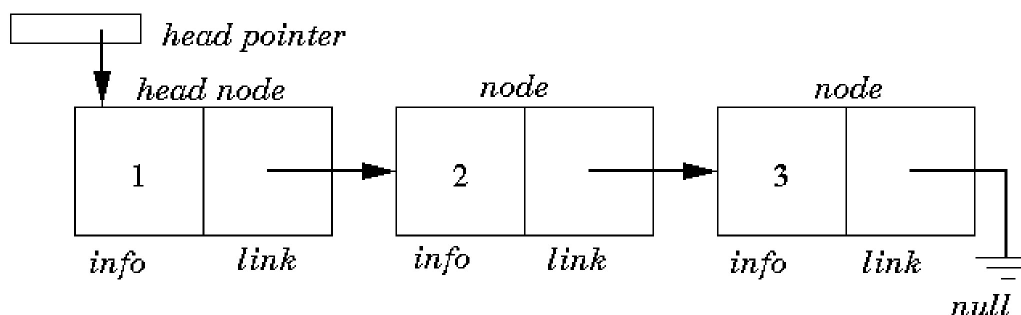


Рисунок 1.4 – Список з 3-х елементів

Всякий раз, коли нове значення даних віднесено до відерця, яке вже зайнято, це значення додасться у список, асоційований з цим відерцем.

Недоліком методу є зниження його ефективності, коли багато значень збігаються з одним відерцем.

Приклад.

Хеш-функція – це остача від ділення націло значення ключа на довжину таблиці**.** На рис. 1.5 показано приклад хеш-таблиці з колізіями і їх усунення за допомогою списку. Хеш-функція - ключ%5, де 5 – довжина таблиці.

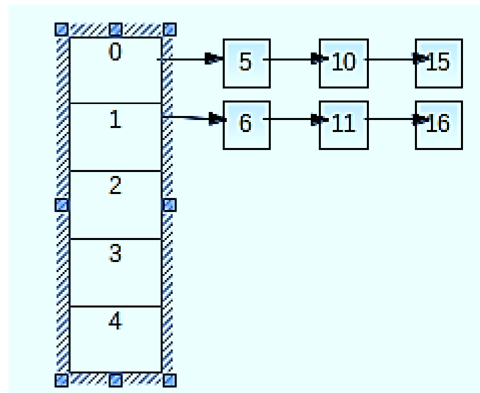


Рисунок 1.5 - Приклад колізій

С-програма.

Нижче наведено С-код відповідної програми [3] та результати виконання контрольного прикладу. Секції програми виділено різними кольорами. По суті, програма реалізує алгоритм роботи зі списком, його створення в оперативній пам'яті, вставку елементів, пошук елементів.

```
    /* Секція документації */
1   /* автор коду - codingstreet.com */
2
    /* Директиви препроцесора */
3   #include<stdio.h>
4   #include<stdlib.h>
5
    /* Секція зовнішніх визначень об'єктів */
6   struct hash          { struct hashnode *ptr;  };
7   struct hashnode
8                        { int val;
9                          struct hashnode *next;  };
10
    /* Визначення функцій */
11  struct hash *create_hash(int size)
12  {    struct hash *htable;
13       int i;
14       htable = (struct hash *) malloc(sizeof(struct hash) * size);
15       if  (htable == NULL) {    printf("\n Out of Memory !!");
16                                 return NULL;                  }
17       for (i = 0; i < size; i++)
18            htable[i].ptr = NULL;
19       return htable;
20   }
21
22   struct hashnode *create_hashnode(int val)
23  {    struct hashnode *tmp;
24       tmp = (struct hashnode *) malloc(sizeof(struct hashnode));
```

```
25      if  (tmp == NULL)  printf("\n System out of Memory");
26      tmp->next = NULL;
27      tmp->val = val;
28  }
29
30  int insert_value(struct hash **htable, int val, int hsize)
31  {   int slot;
32      struct hash *htptr = (*htable);
33      struct hashnode *tmp;
34      if ((*htable) == NULL) {printf("\n Hash Table is not created");
35                              return 0;                        }
36      slot = (val) % hsize;
37      if (htptr[slot].ptr == NULL) {   tmp = create_hashnode(val);
38                                       htptr[slot].ptr = tmp;        }
39      else {        tmp = create_hashnode(val);
40                    tmp->next = htptr[slot].ptr;
41                    htptr[slot].ptr = tmp;              }
42      return 1;
43  }
44
45  void print_hashtable(struct hash **htable, int size)
46  {     int i;
47        struct hashnode *tmp;
48        struct hash *htmp = (*htable);
49        for (i = 0; i < size; i++) {      tmp = htmp[i].ptr;
50                                          while (tmp != NULL)
51                                              {         printf(" %d ", tmp->val);
52                                                        tmp = tmp->next;          }
53                                          printf("\n");
          }
54  }
55
56  int seach_hashtable(struct hash **htable, int val, int size)
57  {      int slot, found = 0;
58         slot = val % size;
59         struct hashnode *tmp;
60         tmp = (*htable)[slot].ptr;
61         while (tmp != NULL)
62                   {         if (tmp->val == val) {       found = 1;
63                                                          break;    }
64                             tmp = tmp->next;                          }
65         return found;
66  }
67
```

/* Функція main() */

```
68  int main()
69  {
70  struct hash *hashtable1;
71  int hashtable1size;
72  printf("\n Enter the size of HashTable : ");
73  scanf("%d", &hashtable1size);
74  hashtable1 = create_hash(hashtable1size);
75  if (hashtable1 != NULL) printf ("\nTable Created Successfully");
76  int key, val, found;
77  do {
78            printf("\n Hash Table Collision Resolution by Codingstreet.com");
79            printf("\n 1. Insert value ");          printf("\n 2. Search Value");
```

```
80          printf("\n 3. Print Table");          printf("\n 4. Exit");
81          printf("\n Enter the key [1-4] :");
82          scanf("%d", &key);
83  switch (key) {
84          case 1:
85                  printf("\n Enter Val : ");    scanf("%d", &val);
86                  insert_value(&hashtable1, val, hashtable1size);
87                  break;
88          case 2:
89                  printf("\n Enter Val : ");    scanf("%d", &val);
90                  found = seach_hashtable(&hashtable1, val, hashtable1size);
91                  if (found == 0)
92                          printf("\n Val not found");
93                  else
94                          printf("\n %d found at slot %d ", val, val % hashtable1size);
95                  break;
96          case 3:
97                  print_hashtable(&hashtable1, hashtable1size);
98                  break;
99          case 4:
100                 printf("\n Thank you !!! ");
101                 break;
102                 default:
103                 break;
104                 }
105 }       while (key != 4);
106 printf("\n Thanks for using Codingstreet.com 's Data structure solution\n");
107 return 0;
108 }
```

Контрольний приклад.

Програма пропонує задати розмір таблиці. Користувач вводить число 3 , що означає три рядки для майбутньої таблиці. Отже, хеш-функція - **ключ%3**.

Користувач обирає пункт 1 меню для введення значень (які одночасно є ключами): 1, 5, 10, 15, 11, 5, 6. Програма будує хеш-таблицю. Користувач обирає пункт 3 меню, щоб побачити таблицю на моніторі. Буде виведено три рядочки:

6 15
10 1
5 11 5

Перший рядок – дані з остачею 0 від ділення на 3. Другий рядок – дані з остачею 1 від ділення на 3. Третій рядок – дані з остачею 2 від ділення на 3.

```
Enter the size of HashTable : 3

Table Created Successfully
 Hash Table Collision Resolution by Codingstreet.com
 1. Insert value
 2. Search Value
 3. Print Table
 4. Exit
 Enter the key [1-4] :1

 Enter Val : 1

 Hash Table Collision Resolution by Codingstreet.com
 1. Insert value
 2. Search Value
```

```
3. Print Table
4. Exit
Enter the key [1-4] :1

Enter Val : 5

Hash Table Collision Resolution by Codingstreet.com
1. Insert value
2. Search Value
3. Print Table
4. Exit
Enter the key [1-4] :1

Enter Val : 10

 Hash Table Collision Resolution by Codingstreet.com
1. Insert value
2. Search Value
3. Print Table
4. Exit
Enter the key [1-4] :1

Enter Val : 15

 Hash Table Collision Resolution by Codingstreet.com
1. Insert value
2. Search Value
3. Print Table
4. Exit
Enter the key [1-4] :1

Enter Val : 11

 Hash Table Collision Resolution by Codingstreet.com
1. Insert value
2. Search Value
3. Print Table
4. Exit
Enter the key [1-4] :1

Enter Val : 5

 Hash Table Collision Resolution by Codingstreet.com
1. Insert value
2. Search Value
3. Print Table
4. Exit
Enter the key [1-4] :1

Enter Val : 6

Hash Table Collision Resolution by Codingstreet.com
1. Insert value
2. Search Value
3. Print Table
4. Exit
Enter the key [1-4] :3
6  15
10  1
```

```
5 11 5
```

Література: [1, 2, 3].

КОНТРОЛЬНІ ПИТАННЯ

1. Поясніть парадигму процедурного програмування.
2. Чому розробка стандартів мов програмування є важливою для програмістів?
3. Назвіть переваги та недоліки мови програмування С.
4. З яких частин складається типова С-програма?

Лекція 2. Лексичні елементи (частина 1). Декларації та визначення даних і функцій (частина 1)

В табл. 2.1 у відповідності зі стандартом наведено всі види лексем, тобто мінімальних лексичних одиниць, з яких формується текст будь-якої С-програми.

Таблиця 2.1 – Лексеми мови С

Лексичні елементи (Токени)	
Ключові слова **Ідентифікатори** **Універсальні імена символів** **Константи** Цілі константи Константи з плаваючою точкою Константи переліку Символьні константи	**Літерали - рядки** **Пунктуатори** (роздільники) **Імена заголовків** **Препроцесингові числа** **Коментарі**

Розглянемо детально ключові слова, ідентифікатори та універсальні імена символів.

Лексичні елементи. Ключові слова

Ключове слово – це кожне з наступних 44-х слів, які зарезервовані розробниками мови С, тому програміст використовує ці слова тільки у визначеному контексті.

auto break case char const continue default do double	else enum extern float for goto if inline int long	register restrict return short signed sizeof static struct switch	typedef union unsigned void volatile while _Alignas _Alignof _Atomic	_Bool _Complex _Generic _Imaginary _Noreturn _Static_assert _Thread_local

Лексичні елементи. Ідентифікатори

Згідно стандарту, ідентифікатор призначений для того, щоб програміст міг позначати сутності, задіяні в алгоритмі створюваної програми. В табл. 2.2 показано, що саме програміст може позначати ідентифікатором.

Таблиця 2.2 – Використання ідентифікаторів в мові C

Ідентифікатор позначає:	Приклад (відповідні ідентифікатори виділено)
об'єкт	int **slot**, **found** = 0; struct hash ***htable**; union Data { int i; char str[20]; } **data**; enum week {Mon, Tue, Wed} **day**;
функцію	int **seach_hashtable**(struct hash **htable, int val, int size)
тег структури	struct **hash** *htable; struct **hashnode** { int val; struct hashnode *next; };
тег об'єднання	union **Data** { float f; char str[20]; } data;
тег переліку	enum **week** {Mon, Tue, Wed} day;
елемент структури	struct hashnode { int **val**; struct hashnode * **next**; };
елемент об'єднання	union Data { float **nums**; char **str**[4]; } ;
елемент переліку	enum week {**Mon**, **Tue**, **Wed**} day;
назву за typedef	typedef unsigned char **BYTES**;
назву мітки	**fini**: printf("passing from the goto statement\n");
ім'я макросу, параметр макросу	#define **minimum**(**a**, **b**) ((a) < (b) ? (a) : (b))

Наступні символи можна використовувати для ідентифікатора:
_ a b c d e f g h i j k l m n o p q r s t u v w x y z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Цифри можуть бути в ідентифікаторі, крім його початку: 0 1 2 3 4 5 6 7 8 9

Малі та великі літери відрізняються.

Максимальна довжина ідентифікатора не встановлена.

Приклад.

1) m_v та M_V є різними ідентифікаторами.

2) num, _address , user_name , email_1 - вірні ідентифікатори.

3) Недійсними є ідентифікатори:

- 1host - ідентифікатор не може починатися з цифри,
- m v - ідентифікатор не може містити пробіл,
- int - ідентифікатор не може бути ключовим словом,
- S# - символ # не дозволено для ідентифікатора.

Лексичні елементи. Універсальні імена символів

Наші програмісти звикли до використання невеликої множини символів ASCII. Проте в сучасному глобальному світі програмування вимагає доступу до будь-яких символів.

Універсальні імена символів – це коди, які можуть подати будь-який символ. Програміст може вільно вибирати ці символи, якщо йому уже не вистачає звичного набору символів для ідентифікаторів.

Фактично універсальне ім’я символу є кодом і подається як оминаюча послідовність (Escape Sequence). Назва означає, що в таких послідовностях втрачається (оминається) зміст окремо взятого символа, проте набуває певного значення саме їх група.

Універсальні імена символів записуються з префіксом \U, за яким іде восьмизначна кодова точка Юнікода, або з префіксом \u, за яким іде чотиризначна кодова точка Юникода.

На рис. 2.1 показано кодування шести ієрогліфів. Кожен з них може бути ідентифікатором, як, наприклад, більш звичний для наших програмістів ідентифікатор A.

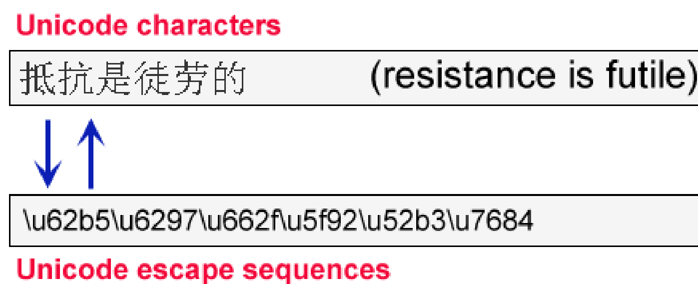
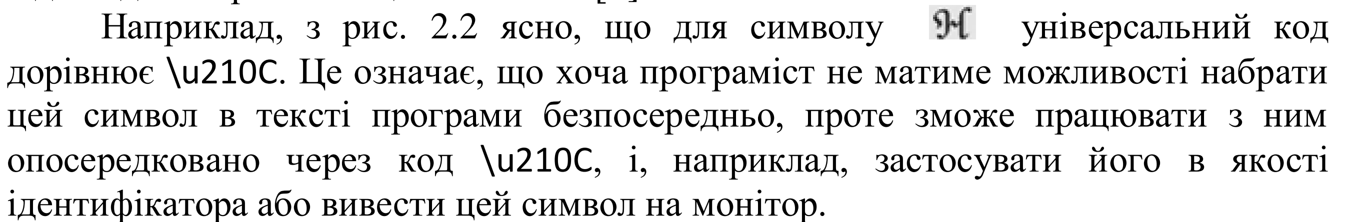


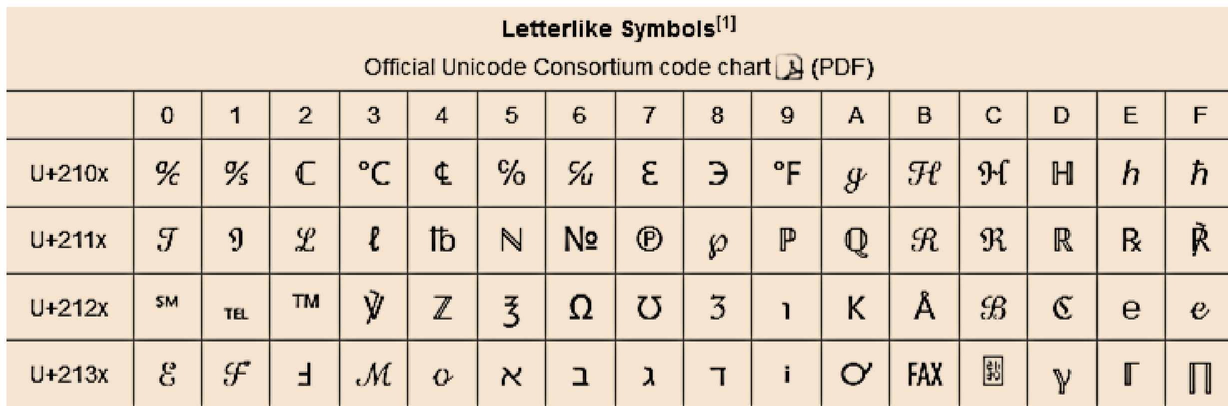
Рисунок 2.1 – Приклад кодування ієрогліфів

На рис. 2.2 та рис. 2.3 подано універсальні імена деяких символів та відповідні зображення цих символів [4].

Наприклад, з рис. 2.2 ясно, що для символу  універсальний код дорівнює \u210C. Це означає, що хоча програміст не матиме можливості набрати цей символ в тексті програми безпосередньо, проте зможе працювати з ним опосередковано через код \u210C, і, наприклад, застосувати його в якості ідентифікатора або вивести цей символ на монітор.

Наступні діапазони номерів Юнікода можна використовувати як універсальні імена символів в ідентифікаторі: 00A8, 00AA, 00AD, 00AF, 00B2-00B5, 00B7-00BA, 00BC-00BE, 00C0-00D6, 00D8-00F6, 00F8-00FF, 0100-02FF, 0370-167F, 1681-180D, 180F-1DBF, 1E00-1FFF, 200B-200D, 202A-202E, 203F-2040, 2054, 2060-206F, 2070-20CF, 2100-218F, 2460-24FF, 2776-2793, 2C00-2DFF, 2E80-2FFF, 3004-3007, 3021-302F, 3031-303F, 3040-D7FF, F900-FD3D, FD40-FDCF, FDF0-FE1F, FE30-FE44, FE47-FFFD, 10000-1FFFD, 20000-2FFFD, 30000-3FFFD, 40000-4FFFD, 50000-5FFFD, 60000-6FFFD, 70000-7FFFD, 80000-8FFFD, 90000-9FFFD, A0000-AFFFD, B0000-BFFFD, C0000-CFFFD, D0000-DFFFD, E0000-EFFFD.

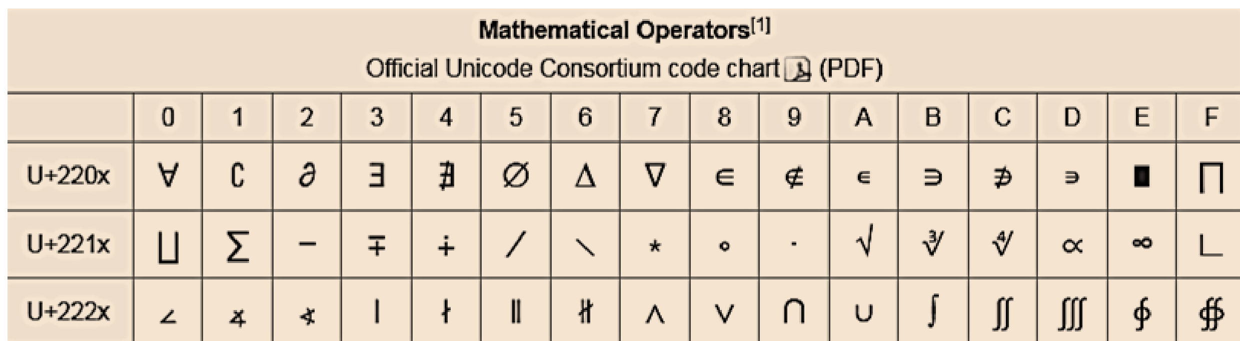
Наступні діапазони номерів кодових точок Юнікоду також можна використовувати як універсальні імена символів для будь-якого символу в ідентифікаторі, крім першого: 0300-036F, 1DC0-1DFF, 20D0-20FF, FE20-FE2F.



Letterlike Symbols[1]
Official Unicode Consortium code chart (PDF)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
U+210x	℀	℁	ℂ	℃	℄	℅	℆	ℇ	℈	℉	ℊ	ℋ	ℌ	ℍ	ℎ	ℏ
U+211x	ℐ	ℑ	ℒ	ℓ	℔	ℕ	№	℗	℘	ℙ	ℚ	ℛ	ℜ	ℝ	℞	℟
U+212x	℠	℡	™	℣	ℤ	℥	Ω	℧	ℨ	℩	K	Å	ℬ	ℭ	℮	ℯ
U+213x	ℰ	ℱ	Ⅎ	ℳ	ℴ	ℵ	ℶ	ℷ	ℸ	ℹ	℺	℻	ℼ	ℽ	ℾ	ℿ

Рисунок 2.2 – Фрагмент таблиці Юнікоду для буквоподібних символів



Mathematical Operators[1]
Official Unicode Consortium code chart (PDF)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
U+220x	∀	∁	∂	∃	∄	∅	∆	∇	∈	∉	∊	∋	∌	∍	∎	∏
U+221x	∐	∑	−	∓	∔	∕	∖	∗	∘	∙	√	∛	∜	∝	∞	∟
U+222x	∠	∡	∢	∣	∤	∥	∦	∧	∨	∩	∪	∫	∬	∭	∮	∯

Рисунок 2.3 – Фрагмент таблиці Юнікоду для математичних символів

Приклад. Рядок програмного тексту містить ідентифікатор, який складається з 7-ми символів: 6-ти символів латиниці m, y, n, a, m, e та 7-го символа Ṁ (велика латинська M з точкою зверху), записаного через своє універсальне ім'я 1E40.

```
int myname\u1E40 = 5;
```

Декларації та визначення об'єктів, функцій

Декларації (declaration) в С-програмі потрібні транслятору для того, щоб знати, в якому форматі, де, яким способом та на який термін розміщувати, зберігати та обробляти дані.

В табл. 2.3 наведено всі можливі елементи декларації в мові С.

Фактично декларація починається з ідентифікатора, яким програміст іменує об'єкт або функцію.

Визначення даних (definition) – це коли програміст або система надає декларованим даним конкретні значення.

Визначення функції – це декларація функції плюс тіло функції.

Згідно стандарту, формально декларація складається зі списку розділених комою деклараторів, кожен з яких може мати додаткову інформацію про тип та/або ініціалізацію, а перед цим списком стоять специфікатори, які вказують на з'єднання, тривалість зберігання і тип сутностей, які позначені деклараторами.

Таблиця 2.3 – Елементи декларацій в мові С

<table>
<tr><th colspan="2">Декларації. Зовнішні визначення</th></tr>
<tr><td>
Специфікатори класу зберігання
typedef, extern, static,
_Thread_local, auto, register

Специфікатори типу
void, char, short, int, long,
float, double,
unsigned, signed,
_Bool, _Complex

Специфікатори структури та об'єднання
struct, union

Специфікатори переліку
enum

Теги (бирки)

Специфікатори атомного типу
_Atomic

Визначення типу
Typedef
</td><td>
Кваліфікатори типу
const, restrict, volatile, _Atomic

Специфікатори функції
inline
_Noreturn

Специфікатори вирівнювання
_Alignas

Декларатори
Прямий декларатор
Декларатори масиву
Декларатори функції
Декларатор вказівника

Імена (назви) типів

Ініціалізація
(присвоєння початкових значень)

Статичні судження
_Static_assert

Зовнішні визначення
Визначення функції
Зовнішні визначення об'єкта
</td></tr>
</table>

Приклад. Написана програмістом декларація стосується сутності з ідентифікатором data, який є прямим декларатором. Справа від декларатора стоїть ініціалізатор =10 , а зліва – специфікатор типу int, специфікатор класу зберігання static, кваліфікатор типу volatile. Ця декларація є також визначенням, оскільки для об'єкта data резервується пам'ять.

```
volatile static int data = 10;
```

Приклад (зі стандарту). Написана програмістом декларація містить декларатор масиву fa[11] типу float та декларатор масиву *afp[17] типу вказівник на float.

```
float fa[11], *afp[17];
```

Примітка. Ідентифікатори fa та afp є прямими деклараторами.

На рис. 2.4, 2.5, 2.6 зображено приклади декларацій об'єктів p, r_t_c та функції fahr.

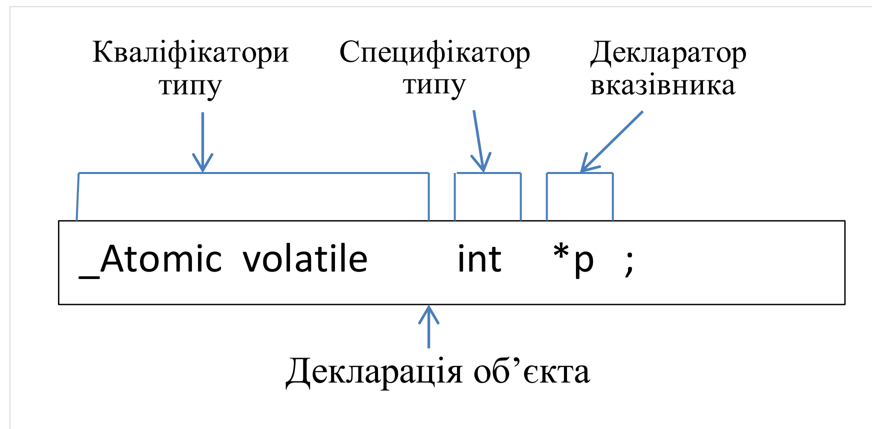


Рисунок 2.4 – Декларація об'єкта p

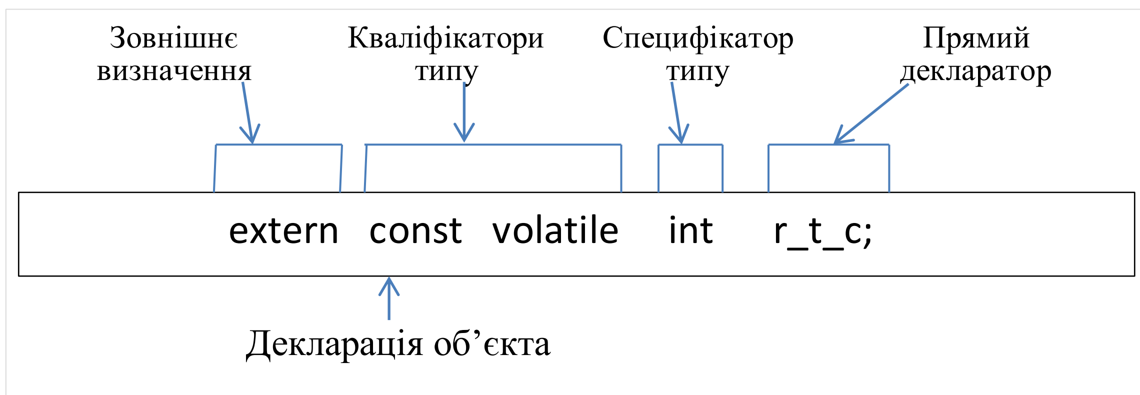


Рисунок 2.5 – Декларація об'єкта r_t_c

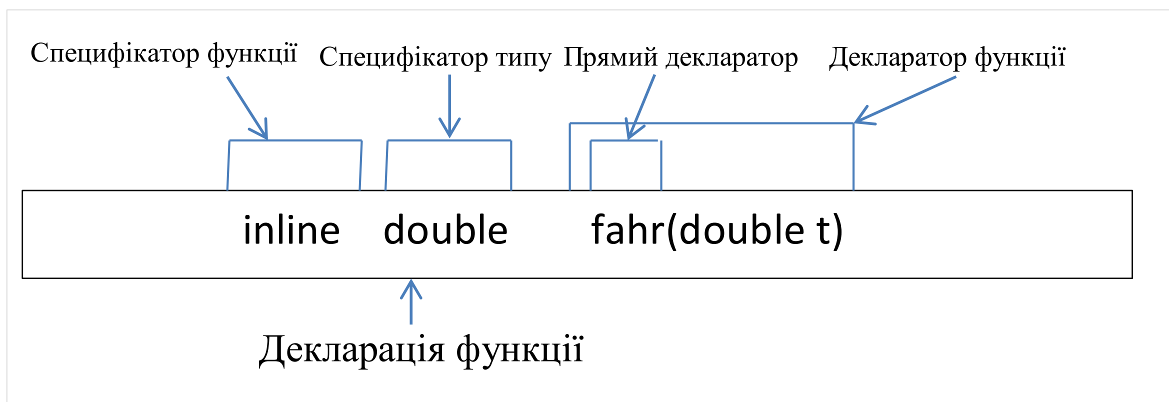


Рисунок 2.6 – Декларація функції fahr

У табл. 2.4 наведено різницю між декларацією та визначенням.

Таблиця 2.4 – Різниця між декларацією та визначенням об'єкта

Декларація	Визначення
Об'єкт або функцію можна декларувати в програмі будь-яку кількість разів	Об'єкт або функцію можна визначити в програмі лише один раз
При декларуванні пам'ять не виділяється	При визначенні виділяється пам'ять
Приклади	

`int f(int);`
є декларацією функції. Ця декларація призначена лише для інформування компілятора про те, що функція з ім'ям `f` буде використовуватися у програмі.
	`int f(int a)`
	`{ return a; }`
	
	Система розподіляє пам'ять під вищевказане визначення функції.

`extern int n;` говорить компілятору, що об'єкт `n` декларовано, тоді як пам'ять для нього буде виділено пізніше в тому ж файлі або в іншому файлі.	
`int a;` є одночасно і декларацією, і визначенням.	

Література: [2], стор. 42-50.

КОНТРОЛЬНІ ПИТАННЯ

1. Як визначається об'єкт в мові С?
2. Що таке універсальні імена символів, і як вони використовуються в мові С?
3. В базовому прикладі знайдіть лексеми різних видів.
4. Нехай підключено файл <complex.h> С-бібліотеки. Що означає наступна декларація? Чи є тут визначені об'єкти?

```
int i = 3.5;
double complex c = 5 + 3 * i;
```

Лекція 3. Концепції мови С (частина 1)

Концепція типу

Дані є предметом особливої уваги С-програміста.

На програміста повністю покладається відповідальність за формування, опис, організацію і перевірку даних в С-програмі.

Програміст повинен явно або неявно описати типи всіх об’єктів програми, а також типи значень, які повертаються функціями. Від типу залежатиме смисл значення, що зберігається в об'єкті або повертається функцією. Тип пишеться програмістом в декларації об’єкта або функції.

Класифікація типів

Існує певна класифікація типів даних в С. На цій основі програміст може придумати безліч типів для якнайкращої розробки своєї програми.

По–перше, типи розділені на дві групи в залежності від того, що вони описують – об’єкти або функції (табл.3.1).

Таблиця 3.1 – Типізація в мові С

Типи об'єктів object types	**Типи функцій** function types

По-друге, основна класифікація типів в мові С для практичного ужитку наведена у табл. 3.2.

Специфікаторами базових типів є ключові слова void, char, short, int, long, float, double, unsigned signed, _Bool, _Complex.

Специфікаторами типів структур та об’єднань є ключові слова struct, union.

Специфікатором переліку є ключове слово enum.

Також у стандарті С описано функціональне групування типів, яке наведено у табл. 3.3.

Таблиця 3.2 - Класифікація типів даних в мові С

<table>
<tr><td rowspan="4">ТИП ПЕРЕЛІКУ
enumerated type

enum</td><td colspan="5">БАЗОВІ ТИПИ
basic types</td><td>ПОХІДНІ ТИПИ
derived types</td></tr>
<tr><td rowspan="3">char</td><td colspan="2">Стандартні цілі типи
standard integer types</td><td colspan="2">Типи з плаваючою точкою
floating types</td><td rowspan="3">Масив
array type

Структура
structure type

Об’єднання
union type

Функція
function type

Вказівник
pointer type

Атомарний
atomic type</td></tr>
<tr><td>Стандартні цілі знакові типи
standard signed integer types

signed char
short int
int
long int
long long int</td><td>Стандартні цілі беззнакові типи
standard unsigned integer types

unsigned char
unsigned short int
unsigned int
unsigned long int
unsigned long long int
_Bool</td><td rowspan="2">Дійсні типи
real floating types

float
double
long double</td><td rowspan="2">Комплексні типи
complex types

float _Complex
double _Complex
long double _Complex</td></tr>
<tr><td>Розширені цілі знакові типи, визначені реалізацією
implementation-defined extended signed integer types</td><td>Розширені цілі беззнакові типи, визначені реалізацією
implementation-defined extended unsigned integer types</td></tr>
<tr><td colspan="7">Тип void охоплює порожній набір значень</td></tr>
</table>

Таблиця 3.3 – Функціональні групи типів

Символьні типи character types	**Цілі типи** integer types	**Дійсні типи** real types	**Арифметичні типи** arithmetic types	**Скалярні типи** scalar types	**Агреговані типи** aggregate types	**Похідні типи декларатора** derived declarator types
char **signed char** **unsigned char**	**char** Цілі знакові типи signed integer types Цілі беззнакові типи unsigned integer types Типи переліку Enumerated types	Цілі типи integer types Дійсні типи з плаваючою точкою real floating types	Цілі типи integer types Типи з плаваючою точкою floating types	Арифметичні типи arithmetic types Вказівники pointer types	Масиви array types Структури structure types	Масиви array types Функції function types Вказівники pointer types

Похідний тип масив

Згідно стандарту, тип масив описує непорожню множину об'єктів, яка розміщується в неперервному шматку пам'яті, з певним типом кожного об'єкта, який називається типом елемента.

Тип масив характеризується типом елементів і кількістю елементів в масиві. Тип є похідним тому, що походить з типу його елементів, і якщо тип елемента - int, тип масив часто називають "масивом int".

Якщо вираз типу масив зустрічається в будь-якому контексті (окрім окремих випадків, наприклад, операндів оператора адреси &, операндів у sizeof), то він піддається компілятором неявному перетворенню у вказівник на його перший елемент.

Існує кілька варіантів масивів (табл. 3.4), які визначаються виразом в квадратних дужках, значення якого буде розміром масиву:

- масиви відомого постійного розміру,
- масиви змінної довжини,
- масиви невідомого розміру.

Таблиця 3.4 – Види масивів в мові С

Варіант масиву	**Опис**
Масиви відомого постійного розміру	Якщо вираз в деклараторі масиву є цілочисельним константним виразом зі значенням більше нуля, а тип елемента є типом з відомим постійним розміром, то декларатор оголошує масив відомого постійного розміру.
Масиви змінної довжини	Якщо вираз в деклараторі масиву не є цілим числом, то декларатор призначений для масиву змінного розміру. Кожен раз, коли процес виконання програми охоплює декларацію, вираз в деклараторі знову обчислюється, і під масив знову виділяється пам'ять. Відповідно, час життя масиву змінного розміру закінчується, коли відбувається вихід за межі дії ідентифікатора масива.
Масиви невідомого розміру	Якщо вираз в деклараторі масиву опущено, він декларує масив невідомого розміру. Такий тип є неповним типом. Проте є винятки, коли тип вважатиметься повним: це - список параметрів функцій (де такі масиви перетворюються на вказівники), а також коли в декларації є ініціалізатор. Зауваження. Масив змінної довжини з зазначеним розміром * є повним типом.

Приклад (зі стандарту). Декларація описує x як об'єкт типу одновимірний масив. Хоча кількість елементів не вказано в квадратних дужках, тобто формально маємо масив невідомого розміру, проте задано три ініціалізатори, тому фактично масив отримує розмір – три елементи.

```
int x[] = { 1, 3, 5 };
```

Приклад (зі стандарту). В мові С багатовимірний масив інтерпретується як масив масиву масиву і т.д. У даному фрагменті програми маємо об'єкт matrix типу масив розміру 4, кожний елемент якого має тип масив розміру 3, кожний елемент якого має тип int.

Декларація є визначенням з ініціалізацією. Числа 10, 20 і 30 ініціалізують перший елемент масиву matrix (формально – масив matrix[0]), а саме matrix[0][0], matrix[0][1], matrix[0][2]. Аналогічно наступні два рядки чисел ініціалізують matrix[1] і matrix[2]. Ініціалізатор закінчується зарано, тому matrix[3] ініціалізується нулями за замовчуванням.

```
int matrix[4][3] = {
                    { 10, 20, 30 },
                    { 40, 50, 60},
                    { 70, 80, 90},      };
```

Точно такого ж ефекту можна досягти, написавши

```
int y[4][3] = { 1, 3, 5, 2, 4, 6, 3, 5, 7 };
```

Приклад (зі стандарту). Декларація ініціалізує перший стовпчик масиву z числами 1, 2, 3, 4, а інші елементи будуть заповнені нулями.

```
int z[4][3] = { { 1 }, { 2 }, { 3 }, { 4 } };
```

Приклад. На рис. 3.1 показано матрицю цілих чисел, якій відповідає декларація масиву matrix. На рис. 3.2 показано розміщення масиву в пам'яті.

```
int matrix[4][3]= {{10, 20, 30}, {40, 50, 60}, {70, 80, 90}, {100, 110, 120}};
```

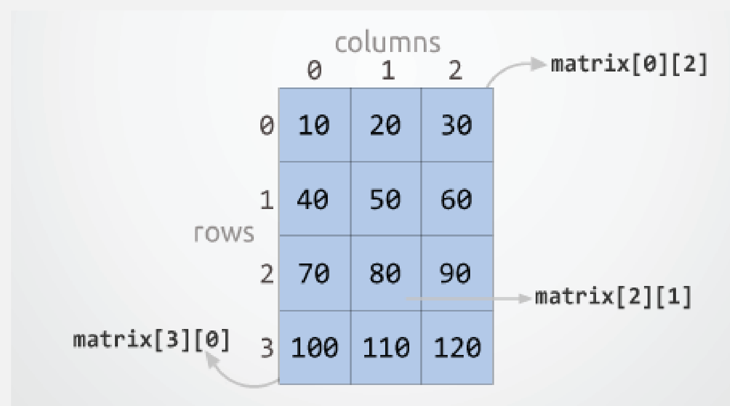


Рисунок 3.1 – Матриця цілих чисел

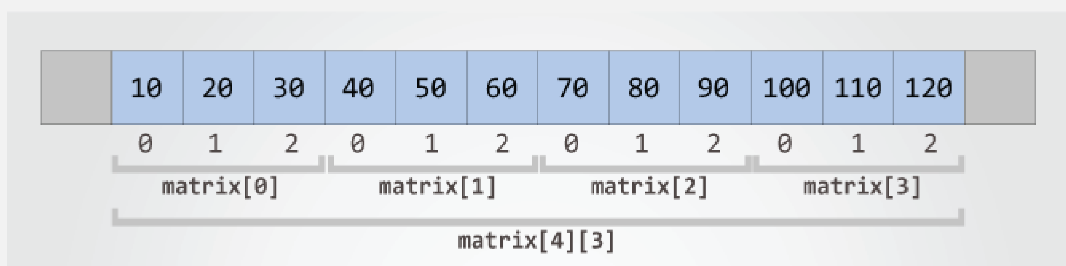


Рисунок 3.2 – Розміщення масиву в оперативній пам'яті

Приклад. Фрагмент програми є блоком у фігурних дужках. Межі дії ідентифікатора n – даний блок. При виконанні фрагменту декларація int a[n] обробляється не одноразово, і при зміні значення n виділяється інша кількість пам'яті. При виході за межі блоку життя масиву

закінчується, що означає звільнення відповідного шматка пам'яті. Вираз sizeof a / sizeof *a обчислює кількість елементів масиву a.

```
{
     int n = 1;
label:
    int a[n];                    // пам'ять різного розміру виділяється 10 разів
    printf("The array has %zu elements\n", sizeof a / sizeof *a);
    if (n++ < 10) goto label;    // при виході за межі блоку пам'ять звільняється
}
```

Приклад.

```
extern int x[];          // тип об'єкта x – масив int невідомого розміру
int a[] = {1,2,3};       // тип об'єкта a – масив int з трьох елементів
```

Приклад. У фрагменті бачимо декларації об'єктів n, m, nm типу масив відомого постійного розміру.

```
int n[100];
char m[sizeof(double)];
enum { MAX=100 };
int nm[MAX];
```

Похідний тип вказівник

Згідно стандарту, тип вказівник походить з типу функції або з типу об'єкта. В цьому контексті тип функції чи тип об'єкта називається типом, на який посилаються (в оригіналі - the referenced type).

Оскільки в програмуванні посилання означає адресування до об'єкта, тобто звертання за конкретною адресою в оперативній пам'яті, стає зрозумілим, що робота програміста з типом вказівник є роботою з адресами комірок пам'яті.

Для скорочення назви типу замість фрази "тип вказівник, що походить від типу int, на який можна посилатись", кажуть коротко "вказівник на int".

Робота з вказівниками

Програмісти активно застосовують вказівники.

Перш за все треба знати три базові дії в роботі з вказівниками.

Дія 1. Написання декларації вказівника, де буде префіксний символ *.

Дія 2. Присвоєння вказівнику значення-адреси за допомогою унарного оператора &.

Дія 3. Отримання доступу до вмісту комірки, адреса якої міститься у вказівнику, за допомогою унарного оператора *.

Приклад. У двох рядках декларацій описано об’єкт x типу int та об’єкт px типу вказівник на int. На рис. 3.3 дана ілюстрація розміщення цих об’єктів в пам’яті під час виконання програми.

```
int x = 10;
int* px = &x;
```

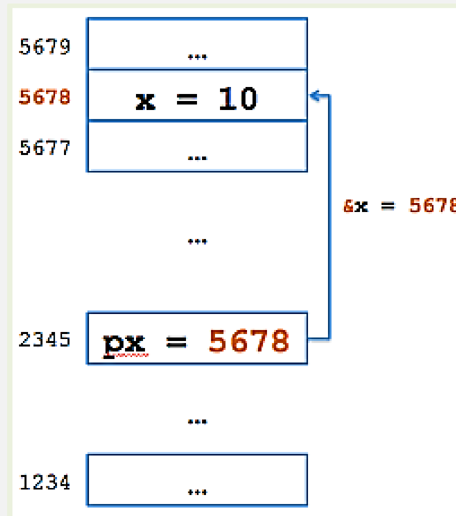


Рисунок 3.3 – Ілюстрація роботи з вказівником

Приклад.

```
#include <stdio.h>
int main () {
  int  var = 90;
  int  *ip;                          /* декларація об’єкта - вказівника */
  ip = &var;                         /* адреса, за якою розміщено var – в ip */
  printf("Address of var : %x\n", &var  );
  printf("Address stored in ip : %x\n", ip );
  printf("Value of *ip : %d\n", *ip );
  return 0;
}
```

Результат роботи програми:

```
Address of var : bffd8b3c
Address stored in ip : bffd8b3c
Value of *ip : 90
```

Далі ми розглянемо смисл вказівника NULL, арифметику вказівників (адресну арифметику), типи вказівник на вказівник та масив вказівників.

Тип NULL вказівник виділено для того, щоб врахувати ситуацію при виконанні програми, коли якийсь вказівник ні на що не вказує.

Макрос NULL визначається як ((void *) 0) в заголовних файлах більшості реалізацій компілятора C. Але стандарт говорить, що 0 також є константним нульовим вказівником. Це означає, що код int * ptr = 0; є цілком законним, хоча результат програми буде під питанням.

Згідно стандарту, макропідстановка NULL визначається реалізацією. Це означає, що внутрішнє представлення нульового вказівника може бути

ненульовим бітовим шаблоном. Реалізація компілятора може відображати константу – нульовий вказівник, наприклад, як одиниці у всіх бітах або якось інакше.

Вам не потрібно турбуватися про внутрішнє представлення нульового вказівника, допоки ви не берете участь у кодуванні компілятора або кодуванні нижчого рівня.

Використання програмістом NULL корисно в таких випадках:

а) ініціалізувати вказівник;

b) перед доступом до вмісту за адресою, яка є значенням вказівника, впевнитись, що вказівник на щось вказує;

c) передати нульовий вказівник як аргумент функції, коли не треба передавати будь-яку дійсну адресу пам'яті.

Адресна арифметика – це арифметичні дії над числами-адресами.

Адреси оперативної пам'яті можуть бути значеннями об'єктів типу вказівник.

Додавання або віднімання числа n до вказівника означає просування по комірках пам'яті, а саме збільшення або зменшення адреси на величину n*sizeof(тип-вказівника) байтів.

Щоб краще зрозуміти арифметику вказівників, розглянемо ptr - вказівник на int. Нехай ptr містить адресу 1000. Також нехай тип int займає чотири байти. Тоді вираз ptr++ матиме значення 1004. При цьому вміст пам'яті за цими адресами не зміниться.

Аналогічно, якщо ptr вказує на об'єкт типу char, адреса якого є 1000, то ptr++ вкаже на місце з адресою 1001.

До вказівників можна застосовувати чотири арифметичних оператора **++ -- + -** та оператори відношення **== < >.**

Треба зауважити, що вказівники не можна додавати. Це пояснюється тим, що вказівники містять адреси, а додавання двох адрес не має сенсу, тому що ви не знаєте, на що ця сума вказуватиме при виконанні програми.

Проте ми можемо відняти два вказівника. Це пояснюється тим, що різниця між двома вказівниками дає кількість елементів відповідного типу даних, які можуть зберігатися між двома адресами.

Приклад.

```
#include <stdio.h>
const int MAX = 3;
int main ()
{       int  var[] = {10, 100, 200};
        int  i, *ptr;      /* декларація вказівника ptr */
        ptr = var;         /* ptr присвоєно адресу початку масиву var */
        i = 0;
        while ( ptr <= &var[MAX - 1] ) {        /* порівнюються адреси */
                printf ("Address of var[%d] = %x\n", i, ptr );
                printf ("Value of var[%d] = %d\n", i, *ptr );
                ptr++;
```

```
                i++;
                                        }
 return 0;
}
```

Результат:

```
Address of var[0] = bfdbcb20
Value of var[0] = 10
Address of var[1] = bfdbcb24
Value of var[1] = 100
Address of var[2] = bfdbcb28
Value of var[2] = 200
```

Приклад (зі стандарту). Декларація

```
char s[] = "abc", t[3] = "abc";
```

визначає об'єкти s та t типу масив char, елементи яких ініціалізовані літералами-рядками символів.

Ця декларація є тотожна декларації

```
char s[] = { ’a’, ’b’, ’c’, ’\0’ },
t[] = { ’a’, ’b’, ’c’ };
```

Вміст цих масивів можна модифікувати, наприклад, через оператор присвоєння:

```
s[1]=’d’;
```

З іншого боку, декларація

```
char *p = "abc";
```

визначає об'єкт p типу вказівник на char та ініціалізує його адресою початку масиву типу char довжиною 4 і значеннями елементів з рядка-літерала. При спробі модифікувати вміст масиву за допомогою вказівника p поведінка програми є невизначеною. Тут рядок "abc" інтерпретується транслятором як константа.

Приклад. Вказівники дуже корисні для обробки масиву символів з рядками різної довжини. На рис. 3.4 зображено економію пам’яті при використанні вказівників для роботи з масивами.

```
char *name[3] = {
                "Adam",
                "chris",
                "Deniel"
                };
//Тепер той же масив, але без вказівників
char name[3][20] = {   "Adam",
                       "chris",
                       "Deniel"
                      }
```

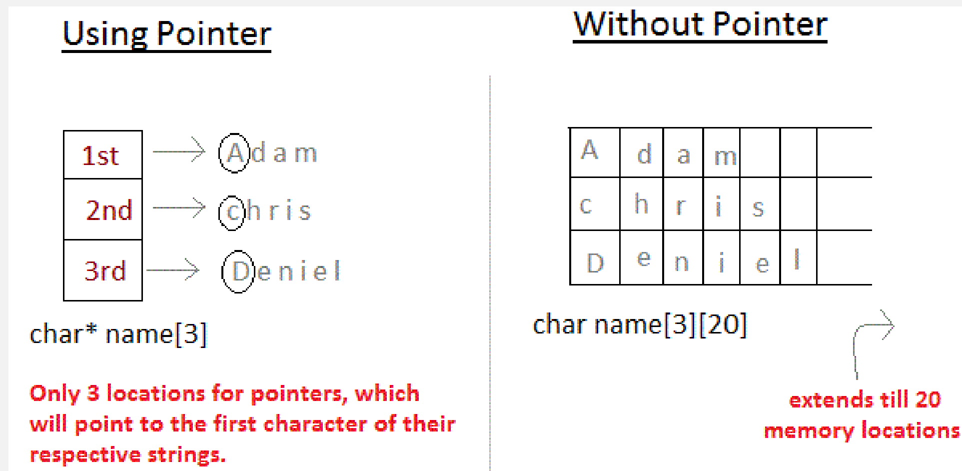


Рисунок 3.4 – Економія пам'яті при використанні вказівників

Тип даних вказівник на вказівник є розповсюдженим типом при реалізації алгоритмів обробки даних.

Приклад. На рис. 3.5 наведено ілюстрацію розподілу пам'яті під дані програми.

```
#include <stdio.h>
int main() {
   int  a = 10;
   int  *p1;              // об'єкт  p1  має тип вказівник на int
   int  **p2;             // об'єкт  p2  має тип вказівник на вказівник на int
   p1 = &a;               // значенням об'єкта  p1  стає адреса об'єкта  a
   p2 = &p1;              // значенням об'єкта  p2  стає адреса об'єкта  p1
   printf("Addr of a = %u\n", &a);
   printf("Addr of p1 = %u\n", &p1);
   printf("Addr of p2 = %u\n\n", &p2);
   printf("Value at the addr stored by p2 = %u\n", *p2);
   printf("Value at the addr stored by p1 = %d\n\n", *p1);
   printf("Value of **p2 = %d\n", **p2);
   return 0;
}
```

Результат:

```
Addr of a = 2686724
Addr of p1 = 2686728
Addr of p2 = 2686732
Value at the addr stored by p2 = 2686724
Value at the addr stored by p1 = 10
Value of **p2 = 10
```

Рисунок 3.5 – Робота з типом вказівник на вказівник

Переваги використання вказівників**:**

1. Вказівники ефективні при обробці масивів і структур.

2. Вказівники дозволяють посилання на функцію і тим самим допомагають при передачі функції в якості аргументів для інших функцій.

3. Вказівники зменшують час виконання програми.

4. Механізм вказівників дозволяє підтримувати динамічне управління пам'яттю.

Похідний тип структура

Згідно стандарту, структурний тип описує послідовно розташований непустий набір об'єктів - елементів, кожен з яких має додатково вказане ім'я і, можливо, окремий тип.

Структура - це агрегований похідний тип даних, який дозволяє об'єднувати елементи даних різних типів. Є аналогом запису в базах даних.

Приклад. Декларація містить ідентифікатор book. Його тип – структура з тегом Books, яка складається з 4-х елементів title, author, subject, book_id різних типів.

```
struct Books {
                    char  title[50];
                    char  author[50];
                    char  subject[100];
                    int   book_id;
             } book;
```

Щоб отримати доступ до елементів структури, можна скористатись операторами:

- оператор елемента . ;

- оператор вказівника на структуру -> .

Приклад. Фрагмент програми ілюструє, крім роботи зі структурним типом даних, концепцію простору імен. Для доступу до елемента Tag структури з тим же ідентифікатором Tag використано оператор точку.

```
#include <stdio.h>
struct Tag;
struct Tag
{
  int Tag;
};
int main(void)
{
  struct Tag Tag = { .Tag = 10 };
  printf( "Tag.Tag = %d\n", Tag.Tag );
  return 0;
}
```

Результат:

```
Tag.Tag = 10
```

Приклад. Для доступу до елементів структури person1 використано оператор -> для вказівника personP.

```
#include <stdio.h>
struct person
        {  int age;
           float height;
        };

int main()
{
   struct person *personP, person1;
  personP = &person1;
  printf("Enter age:");
  scanf("%d", &personP->age);
  printf("Enter height:");
  scanf("%f", &personP->height);
  printf("Age: %d\n",  personP -> age);
  printf("Height: %f",  personP -> height);
  return 0;
}
```

Приклад. Структура з тегом complex є вкладеною в структуру з тегом number.

```
struct  complex
{
 int  imag;
 float  r;
};
struct  number
{
  struct complex  comp;
  int  i;
}   num1, num2;
…   … … … … … … … … … … … … … … …
num2.comp.imag = 19;
```

Література: [2], стор. 31-33, 100-105.

КОНТРОЛЬНІ ПИТАННЯ

1. Як організовано типізацію даних в мові С?
2. Що таке похідний тип?
3. Наведіть приклади використання масивів різних типів в С-програмі.
4. Наведіть приклад декларації структурного типу.

Лекція 4. Вирази. Перетворення і кастинг типів

Вирази в мові С

Згідно стандарту, вираз - це послідовність операторів та операндів, яка:
- або визначає обчислення значення,
- або позначає об'єкт чи функцію,
- або генерує побічні ефекти,
- або виконує комбінацію попередніх варіантів.

Вирази є основою будь-якої С-програми, бо входять до декларацій, тверджень, параметрів функцій.

В табл. 4.1 наведено всі види операторів та первинні вирази мови С. Ми розглянемо деякі з них.

Таблиця 4.1 – Первинні вирази та оператори в мові С

Оператори	
Первинні вирази Ідентифікатор Константа Рядок- літерал (Вираз) Загальний відбір **Постфіксні оператори** Індексація масиву **[ ]** Виклики функції **( )** Елементи структури і об'єднання **-> .** Оператори постфіксних інкремента і декремента **++ --** Складені літерали **Унарні оператори** Оператори префіксних інкремента і декремента **++ --** Оператори адреси & та вмісту за адресою * Унарні арифметичні оператори **+ - ! ~** Оператори sizeof та _Alignof	**Оператори кастингу (type)** **Мультиплікативні оператори * / %** **Адитивні оператори + -** **Побітові оператори зсуву << >>** **Оператори відношення < > <= >=** **Оператори рівності == !=** **Побітовий оператор &** **Побітовий оператор ^** **Побітовий оператор |** **Логічний оператор &&** **Логічний оператор ||** **Умовний оператор ?:** **Оператори присвоєння** **= += -= *= /= %= >>= <<= &= ^= |=** **Оператор кома ,** **Константні вирази**

Пріоритет операторів. Асоціативність операторів. Порядок обчислень у виразах

Поняття “пріоритет операторів” і “асоціативність операторів” стосуються етапу компіляції С-програми і є не залежними від поняття “порядок обчислень”, який пов’язаний з етапом виконання С-програми.

На рис. 4.1 показано таблицю пріоритетів та асоціативності, за якою компілятор фактично “розставляє дужки” у виразах для підготовки до обчислень цих виразів під час виконання програми.

Оператори, які знаходяться вище в таблиці, мають більший пріоритет, тобто компілятор С виокремлює їх першими. Оператори з однаковим пріоритетом оцінюються у порядку їх асоціативності.

Precedence	Operator	Description	Associativity
1	`++ --`	Suffix/postfix increment and decrement	Left-to-right
	`()`	Function call	
	`[]`	Array subscripting	
	`.`	Structure and union member access	
	`->`	Structure and union member access through pointer	
	`(type){list}`	Compound literal(C99)	
2	`++ --`	Prefix increment and decrement[note 1]	Right-to-left
	`+ -`	Unary plus and minus	
	`! ~`	Logical NOT and bitwise NOT	
	`(type)`	Type cast	
	`*`	Indirection (dereference)	
	`&`	Address-of	
	`sizeof`	Size-of[note 2]	
	`_Alignof`	Alignment requirement(C11)	
3	`* / %`	Multiplication, division, and remainder	Left-to-right
4	`+ -`	Addition and subtraction	
5	`<< >>`	Bitwise left shift and right shift	
6	`< <=`	For relational operators < and ≤ respectively	
	`> >=`	For relational operators > and ≥ respectively	
7	`== !=`	For relational = and ≠ respectively	
8	`&`	Bitwise AND	
9	`^`	Bitwise XOR (exclusive or)	
10	`|`	Bitwise OR (inclusive or)	
11	`&&`	Logical AND	
12	`||`	Logical OR	
13	`?:`	Ternary conditional[note 3]	Right-to-Left
14[note 4]	`=`	Simple assignment	
	`+= -=`	Assignment by sum and difference	
	`*= /= %=`	Assignment by product, quotient, and remainder	
	`<<= >>=`	Assignment by bitwise left shift and right shift	
	`&= ^= |=`	Assignment by bitwise AND, XOR, and OR	
15	`,`	Comma	Left-to-right

Рисунок 4.1 – Пріоритет операторів

Приклад. Обчислення виразу 1 > 2 + 3 && 4 в програмі відбудеться згідно пріоритету операторів (рис. 4.1): > (пріоритет 6), + (пріоритет 4), && (пріоритет 11), тобто ((1 >(2 + 3)) && 4), де спочатку виконується 2 + 3, результат 5, потім 1 > 5, результат 0 (false), потім 0 && 4, результат 0**.**

Приклад. Обчислення виразу 1 == 2 != 3 відбудеться згідно асоціативності зліва направо операторів == та !=, оскільки вони мають однаковий прворитет. Спочатку виконується 1 == 2, результат 0 (false), потім 0 != 3, результат 1 (true).

Приклад (зі стандарту). Як обчислюється виділене твердження – вираз у наступному фрагменті?

```
#include <stdio.h>
int sum;
char *p;
/* ... */
sum = sum * 10 - ’0’ + (*p++ = getchar());       /* твердження – вираз */
```

На етапі трансляції вираз групується з врахуванням пріоритетів операторів:

```
sum = (((sum * 10) - ’0’) + ((*(p++)) = (getchar())));
```

Детальніше, розстановка пріоритетів виглядає так:

++ – унарний оператор інкремента – пріоритет 1;

getchar() – виклик функції – пріоритет 1;

***** – унарний оператор вмісту за адресою – пріоритет 2;

***** – бінарний оператор множення – пріоритет 3;

- – бінарний оператор віднімання – пріоритет 4;

+ – бінарний оператор додавання – пріоритет 4;

= у виразі (*p++ = getchar()) – пріоритет 14;

= у виразі sum = sum * 10 - ’0’ + (*p++ = getchar()) – пріоритет 14.

А от на етапі виконання порядок обчислення операндів може бути різним, наприклад, фактичне збільшення змінної p може відбутись будь-коли, але (лекція 12, точки упорядкованості) не після досягнення кінця виразу (;), і виклик функції getchar може відбутися раніше, аніж обчислення дістануться місця, де буде потрібне повернуте значення функції (у прикладі – це оператор присвоєння).

Приклад. У програмному фрагменті програміст використав наступні конструкції мови C: декларації, вирази, твердження.

```
{                       /* початок складеного твердження (блоку) */
int x;                  /* декларація */
int y;
int z;
x = 2;                   /* 2 - первинний вираз, x - первинний вираз, x = 2 є виразом,
                        додана крапка з комою робить вираз твердженням */
y = 2 * x + 5;          /* вираз з операторами +, * міститься у виразі з оператором = */
if (y == 9)             /*  y == 9  є виразом; if else – твердження вибору */
```

```
  { z = 52; }          /*  z = 52  є виразом */
else
  { z = 0;  }          /*  z = 0 - вираз, z = 0; -  твердження, { z = 0;  } - блок */
func("fparam");                /* виклик функції func("fparam") є виразом */
}                      /* кінець блоку */
```

Базовий приклад. Рядки 14, 15, 18, 24-27, 36, 39-41 і т.д..

Первинні вирази

До первинних виразів належать: ідентифікатор, константа, рядок- літерал, (вираз), загальний відбір.

Загальний відбір _Generic –це спосіб вибору одного з кількох виразів під час компіляції на основі типу вказаного виразу.

Приклад. Функція main обчислює кубічний корінь числа. В залежності від типу константи, який визначається на етапі трансляції, обирається один з трьох бібліотечних макросів cbrt, cbrtf, cbrtl.

```
1  #include <stdio.h>
2  #include <math.h>
3
4  // Possible implementation of the tgmath.h macro cbrt
5  #define cbrt(X) _Generic((X), \
6                long double: cbrtl, \
7                    default: cbrt,  \
8                      float: cbrtf  \
9  )(X)
10
11 int main(void)
12 {
13     double x = 8.0;
14     const float y = 3.375;
15     printf("cbrt(8.0) = %f\n", cbrt(x)); // selects the default cbrt
16     printf("cbrtf(3.375) = %f\n", cbrt(y)); // converts const float to float,
17                                             // then selects cbrtf
18 }
```

Результат виконання програми:

```
cbrt(8.0) = 2.000000
cbrtf(3.375) = 1.500000
```

Постфіксні оператори

Варто нагадати, що в компьютерних науках термін 'постфіксний' стосується форми запису виразів. Існує три форми, які відрізняються розташуванням операторів та операндів:

- інфіксний запис (знак оператора розміщується між операндами), наприклад, A + B, A + B * C;

- префіксний запис (знак оператора розміщується перед операндами), наприклад, + A B, + A * B C;

- постфіксний запис (знак оператора розміщується після операндів), наприклад, A B +, A B C * +.

Індексація масиву – це постфіксний вираз вигляду вираз1[вираз2], який позначає відповідний елемент масиву.

Для успішної роботи з масивами програміст має вивчити тип вказівник, оператор адреси & , оператор опосередкованості (вмісту за адресою) *, а також розуміти адресну арифметику.

В мові С оператор індексу [ ] пов'язаний з вказівниками тому, що вираз **вираз1[вираз2]** є ідентичним виразу **(* ((вираз1) + (вираз2)))** і перетворюється в останній при виконанні програми. Згідно стандарту, один з цих виразів має бути масивом (еквівалентно, вказівником на перший елемент), а другий – типу int. Тоді оператор + виконує дію адресної арифметики. Результатом є адреса. Вміст пам'яті за цією адресою буде відповідним елементом масиву.

Приклад. Записи mas[5] та 5[mas] – ідентичні і означають позначення 6-го елементу масиву mas. Після обробки транслятором вираз mas[5] стане виразом (* ((mas) + (5))), а при виконанні програми значенням виразу стане 6-й елемент масиву mas.

Приклад. В програмі використано квадратні дужки, тобто оператор індексації, для доступу до елементів масиву для їх ініціалізації. У першому виклику функції printf використано оператор опосередкованості *, операндом якого є ідентифікатор масиву. Оскільки для компілятора ідентифікатор масиву суть адреса його початку, то результат виконання оператора * - дані за цією адресою, тобто перший елемент масиву, значення якого 23. У другому виклику функції printf використано оператор індексації масиву для звертання ло третього елемента age[2]. Компілятор С перетворює вираз age[2] у вираз адресної арифметики (* (age + 2)), значенням якого є число 65 (рис. 4.2).

```
#include <stdio.h>
int main()
{
  short age[4];
  age[0]=23;
  age[1]=34;
  age[2]=65;
  age[3]=74;
  printf("%d\n", *age);
  printf("%d\n", age[2])
  return 0;   }
```

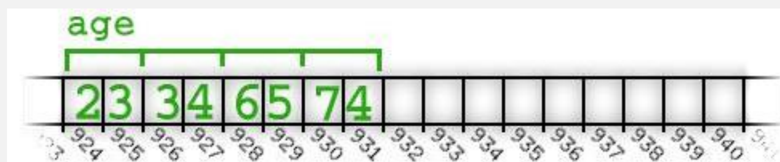


Рисунок 4.2 - Ілюстрація розміщення масиву в пам'яті

Приклад (зі стандарту). У фрагменті показано декларацію об'єкта x.

```
int x[3][5];
```

Згідно правилу "праворуч-ліворуч", x - це масив з трьох елементів, кожен з яких є масивом з п'яти елементів типу int.

Як оброблятиметься звертання до елемента x[i][j] ?

Вираз x [i] є еквівалентним виразу (*((x) +(i))).

При обчисленні значення цього виразу застосовуються правила адресної арифметики. Спочатку x перетворюється на вказівник на перший з трьох масивів довжиною п'ять елементів int кожний. Потім число i множиться на розмір (в байтах) масиву з п'яти int. Результати додаються, отримується результуюча адреса, доступ до вмісту якої здійснюється оператором опосередкованості *.

У виразі x[i][j] цей масив x [i], у свою чергу, перетворюється на вказівник на перший з п'яти int, далі аналогічно обчислюється адреса доступу до j-го елемента типу int.

Виклик функції – це постфіксний оператор, в якому першим операндом є вказівник на функцію (позначення функції також є прийнятним, тому що воно буде перетворено у вказівник на функцію), а всі інші операнди, якщо такі є, називаються аргументами виклику функції. Значення – результат виконання викликаної функції - повертається у місце виклику функції.

Приклад.

```
int myFunc(int x, int y)                  // визначення (тобто декларація + тіло) функції myFunc
{
   return x * 2 + y;
}
int (*fn)(int, int) = &myFunc;         /* декларація fn як вказівника на функцію з двома
                                                параметрами типу int та ініціалізація адресою функції myFunc */
int x = 42;
int y = 123;
printf("(*fn)(%i, %i) = %i\n", x, y, (*fn)(x, y));   /* виклик функції fn як аргумента функції printf */

printf("fn(%i, %i) = %i\n", x, y, fn(x, y));              /* Інша форма виклику функції fn */
```

Результат:

```
(*fn)(42, 123) = 207
 fn(42, 123) = 207
```

Приклад. Порядок обчислення аргументів не визначений у С при виклику функції. У наведеному фрагменті при виклику функції function він може бути зліва направо або справа наліво, що може вплинути на результат. Порядок залежить від реалізації.

```
#include <stdio.h>

void function(int a, int b)
{
   printf("%d %d\n", a, b);
}
int main(void)
{
   int a = 1;
```

```
    function(a++, ++a);
    return 0;
}
```

Приклад. У C всі параметри функції передаються за значенням.

```
#include <stdio.h>
void modify(int v) {
    printf("modify 1: %d\n", v);
    v = 12;
    printf("modify 2: %d\n", v);
}
int main(void) {
    int v = 0;
    printf("1: %d\n", v);
    modify(v);
    printf("2: %d\n", v);
    return 0;
}
```

Результат:

```
1:0
modify 1:0
modify 2:12
2:0
```

Приклад. Щоб виклик функції міг змінювати локальні об'єкти викликаючої функції, програмісти використовують в якості аргументів вказівники.

```
#include <stdio.h>
void modify(int* v)
{
    printf("modify 1: %d\n", *v);
    *v = 12;
    printf("modify 2: %d\n", *v);
}
int main(void) {
    int v = 0;
    printf("1: %d\n", v);
    modify(&v);
    printf("2: %d\n", v);
    return 0;
}
```

Результат:

```
1:0
modify 1:0
modify 2:12
2:12
```

Складений літерал – це постфіксний вираз, який складається з назви типу у дужках, за яким слідує список ініціалізаторів у фігурних дужках. Він конструює об'єкт без імені, значення якого задається списком ініціалізації.

Складені літерали використовуються в основному зі структурами і особливо корисні при передачі змінних структур до функцій. Ми можемо передати об'єкт структуру, не визначивши його.

Приклад. В програмі використано складений літерал (struct Point){2, 3}, який є аргументом функції при її виклику. Це економить час виконання програми, оскільки не потрібно створювати окрему змінну та передавати її в якості аргумента.

```
#include <stdio.h>
struct Point                    /* тип структура, призначений для зберігання координат точки
{
  int x, y;
};
void printPoint(struct Point p)             /* функція, що виводить координати точки */
{
  printf("%d, %d", p.x, p.y);
}
int main()
{
 printPoint((struct Point){2, 3});          /* виклик функції з безпосередньою передачею
                                                елементів структури, без створення
                                                окремої змінної типу struct Point  */

/* без складеного літерала вищезазначене твердження було б написано як
                      struct Point temp = {2, 3};
                      printPoint(temp);                */

return 0;
}
```

Результат програми:
2, 3

Унарні оператори

Нижче подано всі унарні оператори, згідно стандарту:

&	адреса операнда
*	опосередкованість (вміст за адресою)
+	унарний плюс
-	унарний мінус
~	побітове заперечення
!	логічне заперечення
++	інкремент (збільшення операнда на одиницю)
--	декремент (зменшення операнда на одиницю)
sizeof	розмір операнда
_Alignof	вирівнювання

Тут можна детальніше описати оператори префіксного інкременту та префіксного декременту. Ці оператори збільшують або зменшують значення об'єкта відповідно і повертають посилання на результат. Постфіксний інкремент (декремент) створює копію об'єкта, збільшує (або зменшує) значення об'єкта, а повертає копію з початковим значенням.

Приклад. Наступний фрагмент коду виробляє суперечливі результати, оскільки відомо, що порядок обчислення операндів оператору присвоєння є наперед не визначеним. Якщо спочатку збільшується змінна i, а потім обчислюється індекс масиву x, то x[2] присвоїться 1. Якщо спочатку обчислюється індекс, то x [1] присвоїться 1.

```
int x[4] = { 0, 0, 0, 0 };
int i = 1;
x[i] = i++;              /* проблемний рядок коду */
```

Приклад. У фрагменті програми застосовано оператор унарного мінуса, змінна x отримує нове значення з протилежним знаком, а саме, -987.

```
short x = 987;
x = -x;
```

Приклад. У фрагменті програми застосовано оператор побітового заперечення, новим значенням y стає 0x5555 як побітове заперечення значення 0xAAAA.

```
unsigned short y = 0xAAAA;
y = ~y;
```

Приклад. У фрагменті є вираз з оператором логічного заперечення. Якщо x більше або дорівнює y, результат виразу дорівнює 1 (true). Якщо x менше y, результат дорівнює 0 (false).

```
if ( !(x < y) ) …
```

Приклад. У фрагменті застосовано оператор інкремента, змінна j зменшується на 1, перш ніж вона буде використана як індекс.

```
if( line[--j] != '\n' )
   return;
```

Приклад. У фрагменті є оператори адреси та опосередкованості. Адреса шостого елемента масиву a стає значенням вказівника pa. Оператор опосередкованості використовується для доступу до значення за адресою, що зберігається в pa. Це значення типу int присвоюється x.

```
int *pa, x;       // *pa - декларатор вказівника
int a[20];
double d;
pa = &a[5];       /* оператор &
x = *pa;          /* оператор *
```

Приклад. У фрагменті використано унарні оператори & та *. На рис. 7.3 зліва зображено фрагмент програми, а справа – ілюстрація вмісту виділеної пам'яті.

```
int x=5;        /* декларація об'єкта x типу int та його ініціалізація операцією присвоєння,
                правим операндом є константа 5 */

int *ptr=&x;  /* декларація об'єкта ptr типу вказівник на int та його ініціалізація операцією
                присвоєння, правим операндом є унарний оператор адреси об'єкта x */

int copy=*ptr; /* декларація об'єкта copy типу int та його ініціалізація операцією
                присвоєння, правим операндом якої є оператор опосередкованості ptr */
```

Code	Variable	Address	Value
int x = 5;	x	0x04	5
int* ptr = &x;	ptr	0x08	0x04
int copy = *ptr;	copy	0x0C	5

Рисунок 7.3 – Унарні оператори & та *

Приклад. У програмі використано макрос alignof з метою отримання параметрів вирівнювання даних різних типів.

```
#include <stdio.h>
#include <stddef.h>
#include <stdalign.h>
 int main(void)
{
   printf("Alignment of char = %zu\n", alignof(char));
   printf("Alignment of max_align_t = %zu\n", alignof(max_align_t));
   printf("alignof(float[10]) = %zu\n", alignof(float[10]));
   printf("alignof(struct{char c; int n;}) = %zu\n",
         alignof(struct {char c; int n;}));
}
Можливий результат:
Alignment of char = 1
Alignment of max_align_t = 16
alignof(float[10]) = 4
alignof(struct{char c; int n;}) = 4
```

Приклад. У програмі оператор sizeof повертає довжину (в байтах) вказаного в дужках операнда. Результат залежатиме від машинної архітектури.

```
#include<stdio.h>
int main()
{
      int a = 0;
      double d = 10.21;
```

```
    printf("%d", sizeof(a+d));
    return 0;
}
```

Можливий результат:

8

Перетворення і кастинг типів

В мові С активно застосовується перетворення і кастинг типів. Обидві дії виконують задачу перетворення одного типу даних в інший. Наприклад, перед обчисленням виразів транслятор виконує перетворення типів даних (при потребі).

Перетворення типу (type conversion) – це неявне перетворення, виконується компілятором.

Просування типу (type promotion) – особливий випадок неявного перетворення значення вужчого діапазону у тип з більш широким діапазоном. Транслятор С виконує просування автоматично для об'єктів типу _Bool, char, enum та short int, які просуваються до int, і для об'єктів типу float, які підвищуються до double. Спочатку компілятор просуває char в int. Якщо операнди все ще мають різні типи даних, вони перетворюються до найвищого типу даних, який відображається в наступній ієрархічній діаграмі (рис. 4.4).

Рисунок 4.4 – Неявне перетворення типів

Приклад. У виразі sum = num + c перш за все змінна c перетворюється в int, далі компілятор перетворює num та c до типу float і додає їх для отримання результату типу float.

```
#include <stdio.h>
main() {
  int  num = 13;
  char c = 'k';                    /* ASCII-код символу k - 107 */
  float sum;
  sum = num + c;    /* перед обчисленням sum відбувається перетворення num та c  */
  printf("sum = %f\n", sum );}
```

Результат:

```
sum = 120.000000
```

Приклад. На перший погляд, вираз (a * b) / c викликає переповнення розрядної сітки, оскільки знакові символи можуть мати значення лише в діапазоні від -128 до 127 (у більшості компіляторів C), а значення підвиразу (a * b) дорівнює 1200 > 127. Але просування типу забезпечить відповідний результат без переповнення.

```
#include <stdio.h>
int main()
{
   char a = 30, b = 40, c = 10;
   char d = (a * b) / c;
   printf ("%d ", d);
   return 0;
}
```

Результат:

```
120
```

Кастинг типу (type casting) – це явне перетворення типу, виконується програмістом, за допомогою оператора кастингу **(type)**.

Приклад.

```
#include<stdio.h>
int main()
{
          double x = 1.2;
          int sum = (int) x + 1;            // явне перетворення x з double до int
          printf("sum = %d", sum);
          return 0;
}
```

Результат:

```
sum = 2
```

Література: [2], стор. 55-77.

КОНТРОЛЬНІ ПИТАННЯ

1. Яку роль відіграють пріоритет та асоціативність операторів в організації обчислень у виразах мови C?
2. Опишіть види перетворень даних в мові C.
3. В якому порядку буде обчислено вираз 12 + 3 - 4 / 2 < 3 + 1?
4. Наведіть програмний фрагмент з використанням оператора кастингу.

Лекція 5. Декларації та визначення даних і функцій (частина 2). Лексичні елементи (частина 2)

Правило "праворуч-ліворуч"

Програмісту корисно знати правило "праворуч-ліворуч", яке полегшує розуміння та написання простих чи складних декларацій для опису будь-яких типів даних та функцій.

Перш за все треба знати наступні символи та їх розташування в деклараціях відносно ідентифікатора.

Символ * - ознака вказівника - завжди ліворуч.

Символ [] - ознака масиву - завжди праворуч.

Символ () - ознака функції - завжди праворуч.

Тепер саме правило.

КРОК 1. Знайдіть ідентифікатор. Це ваша відправна точка. Тоді скажіть: "Ідентифікатор …". Ви почали вашу декларацію.

КРОК 2. Подивіться на символи праворуч від ідентифікатора. Якщо ви знайдете там (), то це - декларація для функції. Таким чином, ви кажете "Ідентифікатор функції…". Якщо ви знайшли там [], ви кажете "Ідентифікатор об'єкта типу масив…". Продовжуйте рухатись праворуч, поки або не вичерпаються символи або не з'явиться права дужка). Якщо ви натрапляєте на ліву дужку, це початок символу ().

КРОК 3. Тепер подивіться на символи ліворуч від ідентифікатора. Якщо це не один з трьох вищезазначених символів (а, скажімо, щось на зразок "int"),то просто скажіть це. В іншому випадку формуйте опис, як на кроці 2. Продовжуйте йти ліворуч, поки не вичерпаються символи або не з'явиться ліва дужка (.

КРОК 4. Повторіть кроки 2 і 3, доки не сформуєте свою декларацію.

Приклад. Потрібно "розшифрувати" наступну декларацію.

```
int *p[];
```

1) Знаходимо ідентифікатор. Отже, початок розшифровки: "Ідентифікатор p …".

2) Рухаємося по декларації вправо. Бачимо символ [] масиву. Отже, "Ідентифікатор p об'єкта типу масив…".

3) Далі рухатися вправо не можемо, оскільки досягнуто кінець декларації.

4) Рухаємось ліворуч від ідентифікатора і знаходимо символ * . Отже, "Ідентифікатор p об'єкта типу масив вказівників…".

5) Продовжуємо йти ліворуч і знаходимо int . Отже, декларація читається так:

"Ідентифікатор p об'єкта типу масив вказівників на int".

Іншими словами, у нас є тип масив, кожен елемент якого – вказівник на ціле.

Приклад. Потрібно "розшифрувати" наступну декларацію.

```
int *(*func())();
```

1) Знаходимо ідентифікатор func.

2) Переміщуємося вправо. Символ () означає функцію. Отже, початок розшифровки: “Ідентифікатор func функції …”

3) Продовжуємо рух вправо. Оскільки з’явилася права дужка, рух праворуч перериваємо.

4) Починаємо рух вліво від ідентифікатора. Бачимо * - символ вказівника. Отже - “Ідентифікатор func функції, що повертає вказівник на…” .

5) Не можна більше рухатися вліво через появу лівої дужки, так що повертаємось до руху вправо і поновлюємо його з місця попереднього переривання. Бачимо символ () функції. Отже, “Ідентифікатор func функції, що повертає вказівник на функцію...”.

5) Рух вправо закінчується, тому що досягнуто кінець декларації.

6) Відновлюємо рух ліворуч. * означає, що “Ідентифікатор func функції, що повертає вказівник на функцію, яка повертає вказівник на….”.

6) Продовжуємо йти вліво і бачимо int. Кінець опису типу.

Отже, остаточний опис типу:

“Ідентифікатор func функції, яка повертає вказівник на функцію, яка повертає вказівник на int”.

Приклад. Потрібно “розшифрувати” наступну декларацію.

```
int (*(*f_one)(char *,double))[9][20];
```

Опис типу:

f_one є вказівником на функцію, що має два параметри (char *, double) і повертає вказівник на масив (довжини 9) масиву (довжини 20) типу int.

Приклад. Наведено приклади складних декларацій, серед них деякі є неправильними. Для скорочення пояснень в деяких деклараціях опущено слова декларатор, специфікатор.

int i;	i - прямий декларатор (ідентифікатор), int – специфікатор типу
int *p;	*p - декларатор вказівника зі специфікатором типу
int a[];	a[] - декларатор масиву зі специфікатором типу int
int f();	f() - декларатор функції зі специфікатором типу int значення, яке повертається
int **pp;	декларатор вказівника на вказівник зі специфікатором типу
int (*pa)[];	декларатор вказівника на масив типу int
int (*pf)();	декларатор вказівника на функцію, яка повертає значення типу int
int *ap[];	декларатор масиву int-вказівників
int aa[][];	масив масивів типу int
int af[]();	масив функцій, які повертають int (ЗАБОРОНЕНО)
int *fp();	функція, яка повертає int-вказівник
int fa()[];	функція, яка повертає масив типу int (ЗАБОРОНЕНО)
int ff()();	функція, яка повертає функцію, яка повертає значення типу int (ЗАБОРОНЕНО)
int ***ppp;	вказівник на вказівник на int -вказівник
int (**ppa)[];	вказівник на вказівник на масив типу int
int (**ppf)();	вказівник на вказівник на функцію, яка повертає тип int
int *(*pap)[];	вказівник на масив int - вказівників
int (*paa)[][];	вказівник на масив масивів типу int
int *(*pfp)();	вказівник на функцію, яка повертає int-вказівник
int **app[];	масив вказівників на int - вказівники
int (*apa[])[];	масив вказівників на масиви типу int
int (*apf[])();	масив вказівників на функції, які повертають int

int *aap[][];	масив масивів int-вказівників
int aaa[][][];	масив масивів масивів типу int
int aaf[][]();	масив масивів функцій, які повертають int (ЗАБОРОНЕНО)
int *afp[]();	масив функцій, які повертають int-вказівник (ЗАБОРОНЕНО)
int **fpp();	функція, яка повертає вказівник на int-вказівник
int (*fpa())[];	функція, яка повертає вказівник на масив типу int
int (*fpf())();	функція, яка повертає вказівник на функцію, яка повертає int
int *fap()[];	функція, яка повертає масив int-вказівників (ЗАБОРОНЕНО)
int faa()[][];	функція, яка повертає масив масивів типу int (ЗАБОРОНЕНО)
int faf()[]();	функція, яка повертає масив функцій, які повертають int (ЗАБОРОНЕНО)
int *ffp()();	функція, яка повертає функцію, яка повертає int-вказівник (ЗАБОРОНЕНО)

Кваліфікатори типу

В мові С є додаткові опції для опису типів, а саме, кваліфікатори типу: const, restrict, volatile, _Atomic.

Кваліфікатор volatile інформує компілятор, что значення змінної може змінюватись ззовні. Це може відбутись під управлінням операційної системи, апаратури або іншого потоку.

Навіщо потрібен кваліфікатор _Atomic?

Операція в загальній області пам'яті називається атомарною, якщо вона завершується в один крок щодо інших потоків, які мають доступ до цієї пам'яті. Під час виконання такої операції над об'єктом жоден інший потік не може втрутитись до її повного завершення. Неатомарні операції не дають такої гарантії.

Отже, кваліфікатор _Atomic гарантує, що читання (запис) даних не буде "розірвано" в часі і не виникне ситуація "гонок". Об'єкти атомних типів - це єдині об'єкти, вільні від "гонки" даних, тобто вони можуть бути модифіковані паралельно двома потоками або модифіковані одним і прочитані іншим.

Приклад.

```
_Atomic const int * p1;                    // p1 – вказівник на atomic const int
```

Приклад. Об'єкт декларовано так, що він може модифікуватись апаратними засобами (volatile), але не може бути модифікованим через присвоєння, інкремент або декремент (const).

```
extern const volatile int real_time_clock;
```

Приклад. Порядок розташування специфікаторів та кваліфікаторів в декларації може вплинути на остаточний тип даних. Правило "праворуч-ліворуч" буде корисним в цьому контексті. Наступна пара декларацій демонструє різницю між вказівником на постійне значення та постійним вказівником на значення int. Вміст будь-якого об'єкта, на який вказує ptr_to_constant, не може модифікуватись за допомогою цього вказівника, але сам ptr_to_constant може бути змінений, щоб вказувати на інший об'єкт. Аналогічно, вміст int, на яке вказує constant_ptr, може бути змінено, але constant_ptr сам по собі завжди повинен вказувати на одне і те ж місце.

```
const int *ptr_to_constant;
int *const constant_ptr;
```

Лексичні елементи. Константи

Константа – це незмінюваний об’єкт в програмі, який в тексті програми виглядає як число, символ або символьний рядок. Константи мають значення і тип. Тип визначається значенням.

В табл. 5.1 наведено можливі типи констант, які програміст може записати в тексті програми на мові С.

Таблиця 5.1 – Типи констант в мові С

<table>
<tr><th colspan="5">Константи в мові С</th></tr>
<tr><th>Ціла константа</th><th>Константа з плаваючою точкою</th><th>Константа переліку</th><th colspan="2">Символьна константа</th></tr>
<tr><td>десяткова</td><td rowspan="8"></td><td rowspan="8"></td><td>звичайна</td><td>широка</td></tr>
<tr><td>вісімкова</td><td colspan="2">послідовність символів</td></tr>
<tr><td rowspan="6">шістнадцяткова</td><td colspan="2">оминаюча послідовність символів (ОПС)</td></tr>
<tr><td colspan="2">елементарна ОПС</td></tr>
<tr><td colspan="2">вісімкова ОПС</td></tr>
<tr><td colspan="2">шістнадцяткова ОПС</td></tr>
<tr><td colspan="2">універсальне ім’я символа</td></tr>
</table>

Далі розглянемо детально всі види констант.

Ціла константа подає ціле число в десятковій, вісімковій або шістнадцятковій системі числення.

Приклад. Число 28 програміст може подати в тексті своєї програми по-різному:

```
int a=28;          /* десяткове число 28 */
int a=0x1C; /* число 28 у шістнадцятковій системі, оскільки починається з 0x */
int a=034;   /* число 28 у вісімковій системі, оскільки починається з 0  */
```

В табл. 5.2 наведено типи, до яких приводяться задані в тексті програми цілі константи в залежності від форми їх запису та значення [2].

Таблиця 5.2 – Визначення типу цілої константи

Суфікс	Тип десяткової константи	Тип вісімкової або шістнадцяткової константи
none	int long int long long int	int unsigned int long int unsigned long int long long int unsigned long long int
u or **U**	unsigned int unsigned long int unsigned long long int	unsigned int unsigned long int unsigned long long int

l or **L**	long int long long int	long int unsigned long int long long int unsigned long long int
Both **u** or **U** and **l** or **L**	unsigned long int unsigned long long int	unsigned long int unsigned long long int
ll or **LL**	long long int	long long int unsigned long long int
Both **u** or **U** and **ll** or **LL**	unsigned long long int	unsigned long long int

Приклад. Далі показані десяткові, вісімкові і шістнадцяткові константи у фрагменті програмного коду. Для деяких типів програміст має додати відповідні суфікси. Суфікси i64 і I64 використовуються тільки в системах Microsoft для 64-бітових даних.

```
                         /* десяткові константи */
 int a = 28;
unsigned  b = 400024u;
long  c = 20000022l;
unsigned long  d = 40000000ul;
long long  e = 90000000LL;
unsigned long long  f = 9000000001ull;
__int64  g = 90000000002I64;
unsigned __int64  h = 900000000004ui64;
                         /* вісімкові константи */
 int  pm = 024;
unsigned  pm1   = 040000024u;
long  pm2   = 020000022l;
unsigned long  pm3  = 040000000UL;
long long  pm4  = 044000000000000ll;
unsigned long long        pm5 = 04440000000000001Ull;
__int64  pm6 = 0444400000000000002i64;
unsigned __int64  pm7 = 0444400000000000004uI64;
                         /* шістнадцяткові константи */
int  KPI = 0x2a;
unsigned  KPI1 = 0XA0000024u;
long  KPI2 = 0x20000022l;
unsigned long  KPI3 = 0XA0000021uL;
long long  KPI4 = 0x8a000000000ll;
unsigned long long        KPI5 = 0x8A40000000010uLL;
__int64  KPI6 = 0x4a44000000020I64;
unsigned __int64  KPI7 = 0x8a44000000040Ui64;
```

Константа з плаваючою точкою - це десяткове число, яке представляє знакове дійсне число. Загалом, подання знакового дійсного числа містить цілу частину, дробову частину, експоненту, суфікс.

Константи з плаваючою точкою відносяться до типу float, double або long double (табл. 5.3).

Таблиця 5.3 – Визначення типу константи з плаваючою точкою

Суфікс	Тип константи
none of **f**, **F**, **l**, **L**	double
f or **F**	float
l or **L**	long double

В табл. 5.4 показано приклади різних вірних форм запису дійсних чисел.

Таблиця 5.4 – Форми запису дійсних чисел

Форма запису константи в програмі	Значення константи	Тип константи
.0	0.000000	double
0.	0.000000	double
2.	2.000000	double
2.5	2.500000	double
2e1	20.00000	double
2E1	20.00000	double
2.E+1	20.00000	double
2e+1	20.00000	double
2e-1	0.200000	double
2.5e4	25000.00	double
2.5E+4	25000.00	double
2.5F	2.500000	float
2.5L	2.500000	long double

Константи переліку. Перелік є типом даних і складається з набору іменованих цілих констант. Змінна з типом переліку може мати одне зі значень набору.

Приклад. В тексті програми є декларація змінної workday типу enum DAY. Іменовані цілі константи містяться у фігурних дужках.

```
enum DAY        /* Тип    */
{
   saturday,       /* Іменована константа, зв'язана зі значенням 0 за замовчуванням */
   sunday = 0,     /* Іменована константа, зв'язана зі значенням 0 явним присвоєнням */
   monday,         /* Іменована константа, зв'язана зі значенням 1 за замовчуванням */
   tuesday,        /* Іменована константа, зв'язана зі значенням 2 за замовчуванням */
   wednesday,      /* Іменована константа, зв'язана зі значенням 3 за замовчуванням */
   thursday,       /* Іменована константа, зв'язана зі значенням 4 за замовчуванням */
   friday          /* Іменована константа, зв'язана зі значенням 5 за замовчуванням */
} workday;         /* Змінна типу enum DAY  */
```

Символьна константа в базовій мові С - це символ, записаний в одиночних лапках. Наприклад, в рядку програми

```
char u1 = 'x';
```

програміст написав символьну константу 'x'.

Значення символьної константи, яке буде використовуватись програмою, – це відповідний код цієї константи, тобто число, яке міститиметься в пам'яті, відведеній під цю константу. Цей момент є важливим для розуміння того, як виконуватиметься програма.

Так, у вище наведеному прикладі 'x' – однобайтова символьна константа. Значення константи, яке буде використовуватись при виконанні програми, – це ціле число 111 1000 у двійковій системі, 170 - у вісімковій, 120 - у десятковій, 78 - у шістнадцятковій.

Інтернаціоналізація програмування викликала потребу в розширенні таблиці кодувань символів. Наприклад, основною складністю для комп'ютерного середовища з азіатськими мовами є величезна кількість ідеограм, які потрібно вводити / виводити. Для подання цих символів не вистачає одного байту. Щоб допомогти азіатським програмістам, в мові С введено поняття мультибайтових символів та широких символів.

Термін "мультибайтовий символ" обрано світовим стандартом для позначення послідовності байтів, в якій міститься код ідеограми, незалежно від того, яка схема кодування використовується.

Отже, наразі в мові С ціла символьна константа - це послідовність одного або кількох мультибайтових символів, записана в одиночних лапках.

Таким чином, базовий однобайтний символ став просто окремим випадком мультибайтового символу.

Широкі символи введено в мову С з метою уніфікації розміру пам'яті для мультибайтових констант. Так, 16 або 32 біти – цілком доречний варіант для цілочисельних кодів усіх потрібних для програміста символів, навіть якщо це буде повний китайський алфавіт, що містить більше 65 000 ідеограм. Широкі константи мають цілий тип wchar_t, який визначено у файлі <stddef.h>.

Широка константа записується з відповідним префіксом (табл. 5.5) .

Таблиця 5.5 – Визначення типу символьної константи

Префікс	Тип символьної константи
none	звичайна константа unsigned char
L	широка константа цілого типу wchar_t
u	широка константа цілого беззнакового типу char16_t
U	широка константа цілого беззнакового типу char32_t

Оминаюча послідовність символів (ОПС) – ще один спосіб подання констант. Символьні константи можуть подаватись як ОПС: елементарні ОП (табл. 5.6), вісімкові ОП, шістнадцяткові ОП.

Таблиця 5.6 – Елементарні ОП

Символ	Оминаюча послідовність
звуковий сигнал alert (bell)	\a
повернення на одну позицю backspace	\b
зміна сторінки form Feed	\f

зміна рядка new line	\n
повернення каретки carriage Return	\r
горизонтальна табуляція horizontal Tab	\t
вертикальна табуляція vertical Tab	\v
обернена коса риска backslash	\\
одинарні лапки single Quote	\'
подвійні лапки double Quote	\"
знак питання question Mark	\?

Вісімкова ОП - це обернена риска, за якою йде вісімковий еквівалент символу.

У шістнадцятковій ОП символ подається оберненою рискою, за якою йде x та шістнадцятковий еквівалент.

Приклад. Наведено еквівалентні записи в тексті програми символьної константи A.

```
// безпосереднє відтворення символу A – послідовність з одного символу
char u1 = 'A';
// код символу A у вісімковій системі числення - вісімкова ОПС
char u2 = '\101';
// код символу A у шістнадцятковій системі числення - шістнадцяткова ОПС
char u3 = '\x41';
//  універсальне ім’я символу A
char u4 = '\u0041';
//  універсальне ім’я символу A
char u5 = '\U00000041';
```

Приклад. Шістнадцяткова оминаюча послідовність '\x5A' еквівалентна символу Z .

Приклад. Символьна константа ℌ подається універсальним іменем '\u210C'.

Приклад. У наступному виразі масив ініціалізується оминаючими послідовностями, які еквівалентні рядку "Hello!".

```
/*  Ініціалізація масиву рядком "Hello!"  */
char x[] = {'\110','\145','\154','\154','\157','\41','\0'};
```

Приклад. Об’єкт wmc ініціалізується широкою константою. L'ABCD' – це мультибайтова широка символьна константа, про що програміст повідомив префіксом L. Значення цієї константи залежатиме від компілятора.

```
wchar_t  wmc = L'ABCD';
```

Приклад. У наступному фрагменті програми об’єкту (змінній) з ідентифікатором var типу wchar_t (тобто широкий символьний тип) присвоюється початкове значення – широка символьна константа A. Код цієї константи розміщується в 4-х байтах.

```
wchar_t  var = L'A';
```

```
printf ("Wide character value=", var) ;
printf ("Size of the wide char is", sizeof(var));
return 0;
Результат:
Wide character value= 65
Size of the wide char is 4
```

Література: [2], стор. 78-105.

КОНТРОЛЬНІ ПИТАННЯ

1. Розшифруйте наступні декларації:

```
int *(*a)[];   int b[][][];   int (*c[])();   int *d[];   int e;   const _Atomic(int) * f;
const int *g;   extern const volatile int h;
```

2. Наведіть приклади використання специфікаторів та кваліфікаторів типів.
3. Наведіть приклади символьних констант різних видів.

Лекція 6. Функції в мові С

Декларації та визначення функцій

Функціональний тип в мові С є похідним типом, який характеризується типом значення, що повертається функцією, кількістю та типом параметрів.

Декларація функції в програмі містить ідентифікатор функції і (необов'язково) типи параметрів функції.

Декларації функцій можуть з'являтися як в межах блоку, так і в межах файлу.

На рис. 6.1 зображено приклад декларації функції fahr. Тут програміст вказав, що для виконання функція потребує значення одного параметру типу double, результат повертається з типом double. Специфікатор inline означає, що у програміста є прохання до компілятора забезпечити якнайшвидше виконання викликів функції. Стандарт С покладає виконання цього прохання на конкретну реалізацію компілятора.

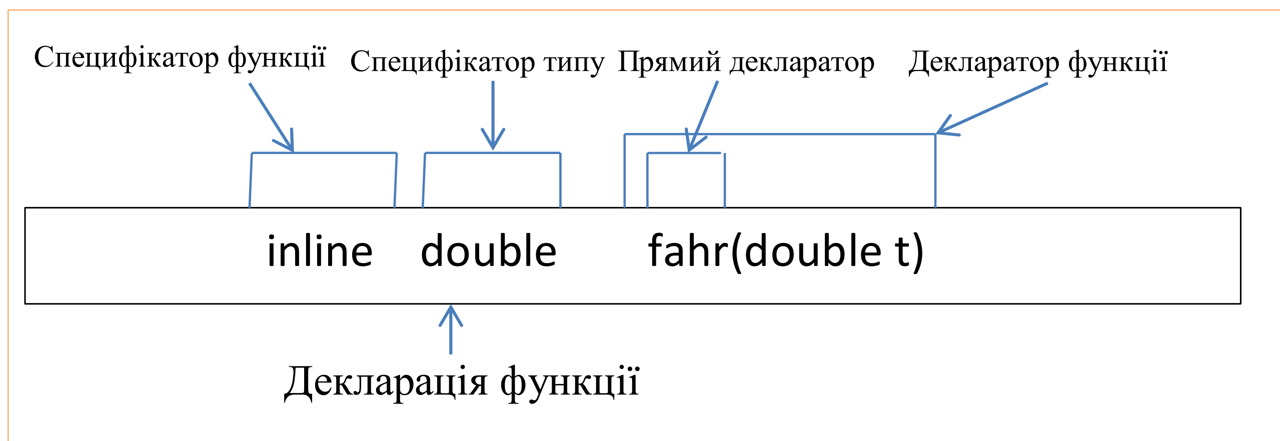


Рисунок 6.1 – Декларація функції fahr

Визначення функції – це декларація плюс тіло функції у фігурних дужках.

Тіло функції – це блок, який містить різні твердження для обробки параметрів та повернення результату у викликаючу функцію.

За замовчуванням ідентифікатор функції має зовнішнє з’єднання, тобто функція доступна в усій програмі. Щоб зробити функцію видимою тільки в межах одного файлу, треба вказати специфікатор static.

Програміст повинен забезпечити обмін даними між функціями, які містяться в його програмі. Для цього є кілька способів.

Спосіб 1. Механізм передачі параметрів.

Спосіб 2. Дані, які зберігаються на зовнішніх носіях у формі файлів, можуть потрапити до функції за допомогою бібліотечних функцій введення / виведення.

Спосіб 3. В С є концепції управління даними, а саме, тривалість зберігання об’єкта та з’єднання ідентифікаторів, які дають доступ до об’єкта з усіх функцій, зокрема, на весь час виконання програми.

Механізм передачі аргументів, повернення результату

Рис. 6.2 ілюструє поняття параметра (формального аргумента) функції, яке стосується декларації та визначення функції add, та поняття аргумента (фактичного параметра), яке стосується виклику функції add та означає фактичне значення параметра, яке передається функції в момент її виклику. Тут і параметри, і аргументи мають однакові імена a, b.

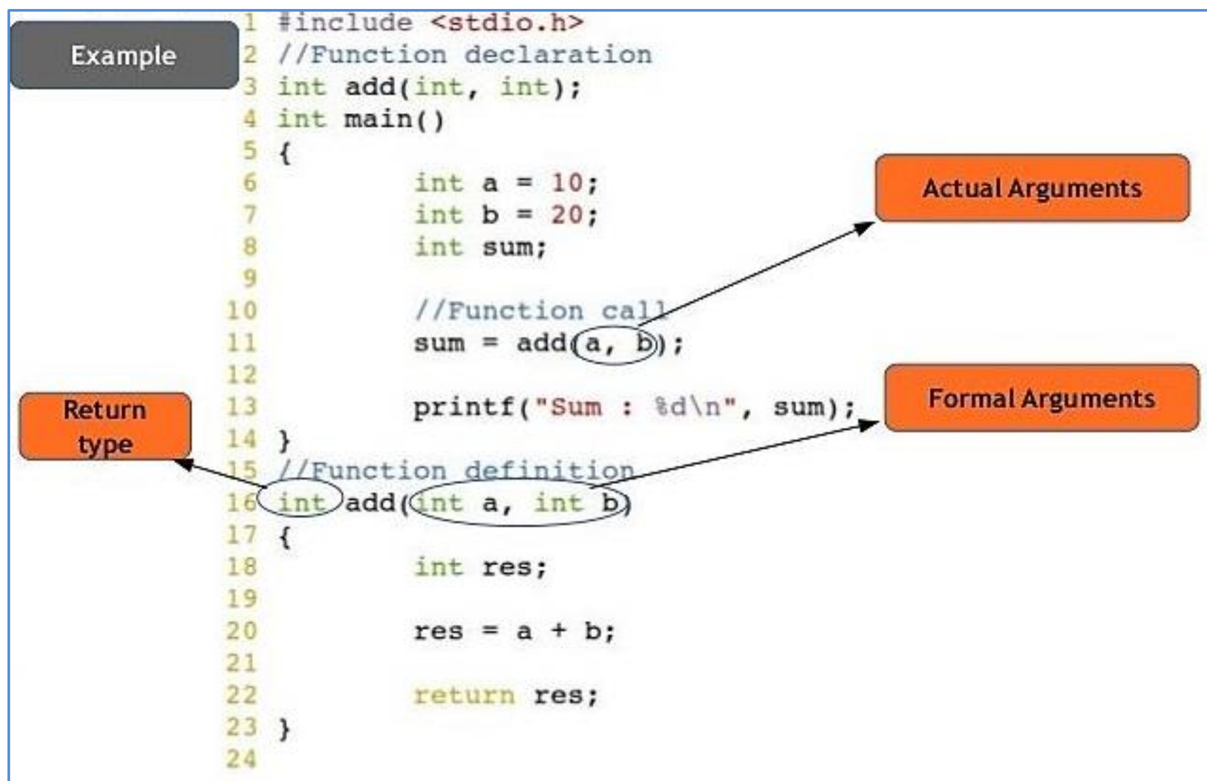


Рисунок 6.2 – Приклад декларації, визначення та використання функції в програмі

Приклад (зі стандарту). У програмному фрагменті дано визначення функції max з двома параметрами; функція повертає значення типу extern int, де extern – це специфікатор класу зберігання, int – специфікатор типу. max(int a, int b) – це декларатор функції, блок { return a > b ? a: b; } – це тіло функції.

```
extern int max(int a, int b)
{
return a > b ? a: b;
}
```

Наступне подібне визначення використовує форму списку ідентифікаторів для декларації параметрів:

```
extern int max(a, b)
int a, b;
{
return a > b ? a: b;
}
```

Різниця між цими двома визначеннями функції полягає в тому, що перша форма виступає як декларація прототипу, що призведе до перетворення типів аргументів при наступних викликах функції, тоді як друга форма цього не виконує.

В табл. 6.1 дано опис двох механізмів передачі параметрів: за значенням та за посиланням. В мові С реалізовано перший механізм.

Таблиця 6.1 – Механізми передачі аргументів

Передача за значенням	Передача за посиланням
Цей метод при виклику функції копіює значення аргумента у параметр функції.	Цей метод при виклику функції копіює адресу аргумента у параметр.
Зміни значення параметра всередині функції не вплавають на значення аргумента.	Всередині функції для доступу до значення аргумента використовується його адреса, а це означає, що зміни значення параметра призведуть до зміни аргумента.

На рис. 6.3 зображено розподіл оперативної пам'яті під час виконання програм, зокрема, для програми P4.

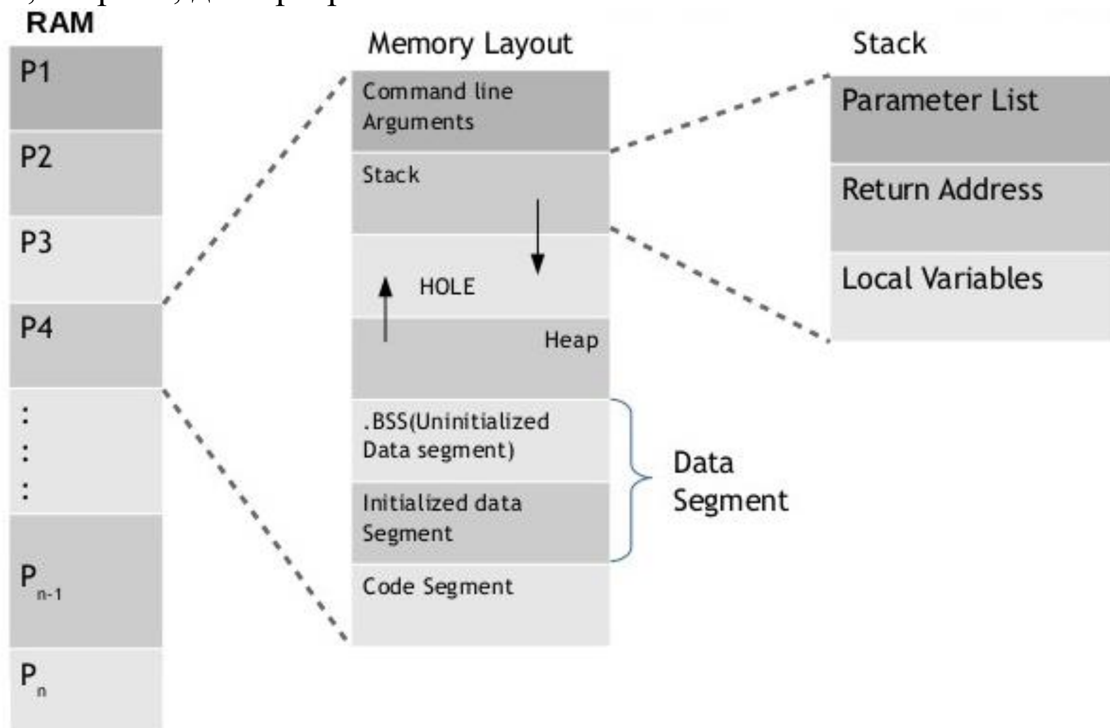


Рисунок 6.3 – Схема розподілу оперативної пам'яті під час виконання програм

Виконання виклику функції

Згідно стандарту, виклик функції виконується так:

1. Спочатку обчислюються вирази для розміру кожного параметра – масива змінної довжини.

2. Потім значення кожного аргументу - виразу перетворюється на тип відповідного параметра (аналогічно як в операторі присвоєння). Якщо

аргументами є вирази - масиви і позначення функцій, то вони перетворюються у відповідні вказівники.

3. Після цього виконується складений оператор (блок), що становить тіло функції.

4. По закінченню виконання функції управління повертається у викликаючу функцію, у місце виклику.

В табл. 6.2 вказано різні варіанти опису функцій, коли програмісту потрібні або не потрібні параметри для передачі даних у функцію, потрібно або не потрібно отримати певний результат виконання функції для подальших обчислень.

Таблиця 6.2 – Варіанти декларацій функцій

Варіанти С функцій	Синтаксис
З параметрами та зі значенням, що повертається	Декларація функції: int function (int); Виклик функції: function (a); Визначення функції: int function(int a) { ...; return a; }
З параметрами та без значення, що повертається	Декларація функції: void function (int); Виклик функції: function (a); Визначення функції: void function(int a) { ...; }
Без параметрів та без значення, що повертається	Декларація функції: void function ( ); Виклик функції: function ( ); Визначення функції: void function( ) { ...; }
Без параметрів та зі значенням, що повертається	Декларація функції: int function ( ); Виклик функції: function ( ); Визначення функції: int function( ) { ...; }

Приклад. Функція test без параметрів та без значення, що повертається.

```
#include<stdio.h>
void test();
int main()
{
	test();
	return 0;
}
void test()
{
	int a = 55, b = 88;
	printf("\n values a = %d and b = %d", a, b);
}
```

Результат:

values a = 55 and b = 88

Приклад. Функція sum без параметрів та зі значенням, що повертається.

```
#include<stdio.h>
int sum();
int main()
{
	int add;
	add = sum();
	printf("\n Sum of two values = %d", add);
	return 0;
}
int sum()
{
	int a = 55, b = 88, sum;
	sum = a + b;
	return sum;
}
```

Результат:

Sum of two values = 143

Приклад. Функція function з параметрами та без значення, що повертається.

```
#include<stdio.h>
void function(int, int[], char[]);
int main()
{
	int a = 20;
	int arr[5] = {10,20,30,40,50};
	char str[30] = "See you soon";
	function(a, &arr[0], &str[0]);
	return 0;
}
```

```
void function(int a, int *arr, char *str)
{
        int i;
        printf("value of a is %d \n",a);
        for (i=0;i<5;i++)
        {
                printf("value of arr[%d] is %d \n",i,arr[i]);
        }
        printf("\n value of str is %s\n",str);
}
```

Результат:

```
value of a is 20

value of arr[0] is 10
value of arr[1] is 20
value of arr[2] is 30
value of arr[3] is 40
value of arr[4] is 50

value of str is See you soon
```

Приклад. Функція function з параметрами та зі значенням, що повертається.

```
#include<stdio.h>
#include<string.h>
int function(int, int[], char[]);
int main()
{
        int i, a = 20;
        int arr[5] = {10,20,30,40,50};
        char str[30] = "See you soon"";
        printf("value of a is %d\n",a);
        for (i=0;i<5;i++)
        {
            printf("value of arr[%d] is %d\n",i,arr[i]);
        }
        printf("value of str is %s\n",str);
        printf("\n values after modification \n");
        a= function(a, &arr[0], &str[0]);
        printf("value of a is %d\n",a);
        for (i=0;i<5;i++)
                {
                  printf("value of arr[%d] is %d\n",i,arr[i]);
                }
printf("value of str is %s\n",str);
return 0;
}

int function(int a, int *arr, char *str)
```

```
{
      int i;
      a = a+20;
      arr[0] = arr[0]+50;
      arr[1] = arr[1]+50;
      arr[2] = arr[2]+50;
      arr[3] = arr[3]+50;
      arr[4] = arr[4]+50;
      strcpy(str,"See you");
      return a;
}
```

Результат:

```
value of a is 20
value of arr[0] is 10
value of arr[1] is 20
value of arr[2] is 30
value of arr[3] is 40
value of arr[4] is 50
value of str is See you soon
values after modification
value of a is 40
value of arr[0] is 60
value of arr[1] is 70
value of arr[2] is 80
value of arr[3] is 90
value of arr[4] is 100
value of str is See you
```

Вказівники як параметри функції

Вказівник використовується для передачі адрес аргументів під час виклику функції, і тим самим реалізується механізм передачі аргументів за посиланням.

Приклад. Функція swap переставляє значення двох змінних m та n, визначених у викликаючій функції main. Це є можливим, оскільки у функцію swap передаються не самі значення, а адреси, за якими вони зберігаються.

```
#include <stdio.h>
void swap(int *a, int *b);
int main()
{   int m = 10, n = 20;
    printf("m = %d\n", m);     printf("n = %d\n\n", n);
    swap(&m, &n);
    printf("After swapping:\n\n"); printf("m = %d\n", m); printf("n = %d", n);
   return 0;
}

void swap(int *a, int *b)
```

```
{  int temp;
   temp = *a;
   *a = *b;
   *b = temp;
}
```
Результат:
```
m = 10
n = 20
After swapping:
m = 20
n = 10
```

Повернення вказівника з функції

Функція також може повернути вказівник у викликаючу функцію. У цьому випадку треба бути обережним з вказівником на локальну змінну, тому що локальні змінні функції не живуть поза функцією, вони мають межі дії тільки всередині функції.

Приклад.

```
#include <stdio.h>
int* larger(int*, int*);
void main()
{  int a = 15;
   int b = 92;
   int *p;
   p = larger(&a, &b);
   printf("%d is larger",*p);
}

int* larger(int *x, int *y)
{  if(*x > *y)
      return x;
   else
      return y;
}
```
Результат:
```
92 is larger
```

Вказівники на функції

В мові C можна декларувати вказівник на функцію, який потім використати як аргумент в іншій функції.

Приклад.

```
int (*sum)();            // декларація вказівника sum на функцію, яка повертає int
int *sum();              // декларація функції sum, що повертає вказівник на int
```

Приклад. Тут s є вказівником на функцію. Тепер функція sum може бути викликана за допомогою вказівника s разом з необхідними значеннями аргументів 50 та 80.

```
int sum(int, int);
int (*s)(int, int);
s = sum;
s (50, 80);
```

Приклад.

```
#include <stdio.h>
int sum(int x, int y)
{
   return x+y;
}
int main( )
{
   int (*fp)(int, int);
   fp = sum;
   int s = fp(10, 15);
   printf("Sum = %d", s);
   return 0;
}
```

Результат:

Sum = 25

Бібліотека функцій мови С. Стандартні заголовки. Функції управління пам'яттю

Однією з переваг мови С є наявність багатої бібліотеки функцій.

Кожна функція С-бібліотеки декларується у відповідному так званому заголовку. Заголовок (заголовний файл) – це файл, текст якого декларує набір функцій, плюс необхідні типи та додаткові макроси для полегшення їх використання. Вміст заголовків додається у програму директивою препроцесора #include. Ім'я заголовків має вигляд < ім'я.h> або "ім'я.h".

Нижче наведено перелік стандартних заголовних файлів, їх призначення.

Детальніше розглянуто заголовок <stdlib.h>, який містить серед інших функції управління оперативною пам'яттю. Наведено фрагмент зі стандарту С, де детально описано ці функції.

Стандартні заголовки наступні.

<assert.h> - Діагностика.
<complex.h> - Комплексна арифметика.
<ctype.h> - Робота з символами.
<errno.h> - Помилки.
<fenv.h> - Середовище з плаваючою точкою.
<float.h> - Характеристики плаваючих типів.

<inttypes.h> - Перетворення формату цілих типів.
<iso646.h> - Альтернативні написання.
<limits.h> - Розміри цілих типів.
<locale.h> - Локалізація.
<math.h> - Математика.
<setjmp.h> - Нелокальні переходи.
<signal.h> - Управління сигналами.
<stdalign.h> - Вирівнювання.
<stdarg.h> - Змінні аргументи.
<stdatomic.h> - Атоми.
<stdbool.h> - Булеви типи.
<stddef.h> - Загальні визначення.
<stdint.h> - Цілі типи.
<stdio.h> - Введення/виведення.

<stdlib.h> - Основні утиліти.

Функції числових перетворень.
Функції генерації псевдовипадкової послідовності.
Функції управління пам'яттю.

7.22.3.2 The calloc function

Synopsis

1 **#include** <stdlib.h>
void ***calloc**(**size_t** nmemb, **size_t** size);

Description

2 The **calloc** function allocates space for an array of nmemb objects, each of whose size is size. The space is initialized to all bits zero.

Returns

3 The **calloc** function returns either a null pointer or a pointer to the allocated space.

7.22.3.3 The free function

Synopsis

1 **#include** <stdlib.h>
void free(**void** *ptr);

Description

2 The **free** function causes the space pointed to by ptr to be deallocated, that is, made available for further allocation. If ptr is a null pointer, no action occurs. Otherwise, if the argument does not match a pointer earlier returned by a memory management function, or if the space has been deallocated by a call to **free** or **realloc**, the behavior is undefined.

Returns

3 The **free** function returns no value.

7.22.3.4 The malloc function

Synopsis

1 **#include** <stdlib.h>
void ***malloc**(**size_t** size);

Description

2 The **malloc** function allocates space for an object whose size is specified by size and whose value is indeterminate.

Returns

3 The **malloc** function returns either a null pointer or a pointer to the allocated space.

Спілкування з середовищем.
Утиліти пошуку і сортування.
Функції арифметики цілих чисел.
Функції перетворення широких символів.

Функції перетворення широких рядків.

<stdnoreturn.h> - _Noreturn.
<string.h> - Управління рядками.
<tgmath.h> - Математика з родовим типом
<threads.h> - Потоки.
<time.h> - Дата та час.
<uchar.h> - Утиліти для Юнікода.
<wchar.h> - Утиліти для розширених багатобайтових та широких символів.
<wctype.h> - Утиліти для класифікації та відображення широких символів.

Література: [1], [2], стор. 113-115, 131-332.

КОНТРОЛЬНІ ПИТАННЯ

1. Як здійснюється передача аргументів у функцію та повернення результату її виконання в мові С?

2. Для чого потрібні стандартні заголовки?

3. Напишіть фрагмент С-програми з використанням функцій управління пам’яттю.

Лекція 7. Введення / виведення даних (частина 1)

Заголовок <stdio.h>

Введення (вв) — це процес передачі даних під час виконання програми з зовнішніх пристроїв в оперативну пам'ять, а виведення (вив) — зворотній процес пересилки даних з оперативної пам'яті назовні.

Вв/вив даних реалізовано у мові С не спеціальними операторами, як у деяких інших мовах, а функціями.

Декларації цих функцій містяться у стандартній бібліотеці, і їх треба підключати за допомогою заголовного файлу <stdio.h>. Вміст stdio.h залежить від реалізації. Тіла функцій, тобто програмний код, як правило, не містяться в <stdio.h>, вони поставляються у прекомпільованому вигляді, підключаються під час лінкування С-програми.

Згідно стандарту, заголовок <stdio.h>:
- визначає кілька макросів (зокрема, NULL, який розширюється до визначеного реалізацією нульового вказівника-константи; _IOFBF, _IOLBF, _IONBF, які розширюються до цілих констант, придатних для використання в якості третього аргументу функції setvbuf; EOF, який розширюється до цілої константи типу int та від'ємним значенням, що повертається декількома функціями при виявленні кінця файлу; stderr, stdin, stdout, які є виразами типу вказівник на тип FILE і які вказують на FILE-об'єкти, асоційовані відповідно зі стандартними потоками помилок, введення та виведення;
- декларує три типи (один з них FILE);
- декларує багато функцій для виконання введення та виведення.

У табл. 7.1 наведено перелік функцій для вв/вив згідно стандарту.

Таблиця 7.1 – Функції для введення та виведення даних в мові С

Функції введення / виведення The input/output functions		
Функції введення широких символів The wide character input functions	Функції виведення широких символів The wide character output functions	Функції введення / виведення байтів The byte input/output functions
fgetwc, fgetws, getwc, getwchar, fwscanf, wscanf, vfwscanf, vwscanf	fputwc, fputws, putwc, putwchar, fwprintf, wprintf, vfwprintf, vwprintf	fgetc, fgets, fprintf, fputc, fputs, fread, fscanf, fwrite, getc, getchar, printf, putc, putchar, puts, scanf, ungetc, vfprintf, vfscanf, vprintf, vscanf

Потоки. Файли

В мові С використано поняття потік (stream) та файл (file) для програмування процесів обміну даними (рис. 7.1).

Файл — це або розміщений на пристрої довільний блок інформації з унікальним іменем, або сам фізичний пристрій (термінал, клавіатура, диск, флешка, мережа тощо), ототожнюваний з файлом.

Згідно стандарту, потік даних - абстрактне поняття для узагальнення процесу передачі даних для різних типів пристроїв та файлів.

Якими б не були пристрої чи файли, дані відображаються або у текстові потоки, або у двійкові потоки (рис. 7.1).

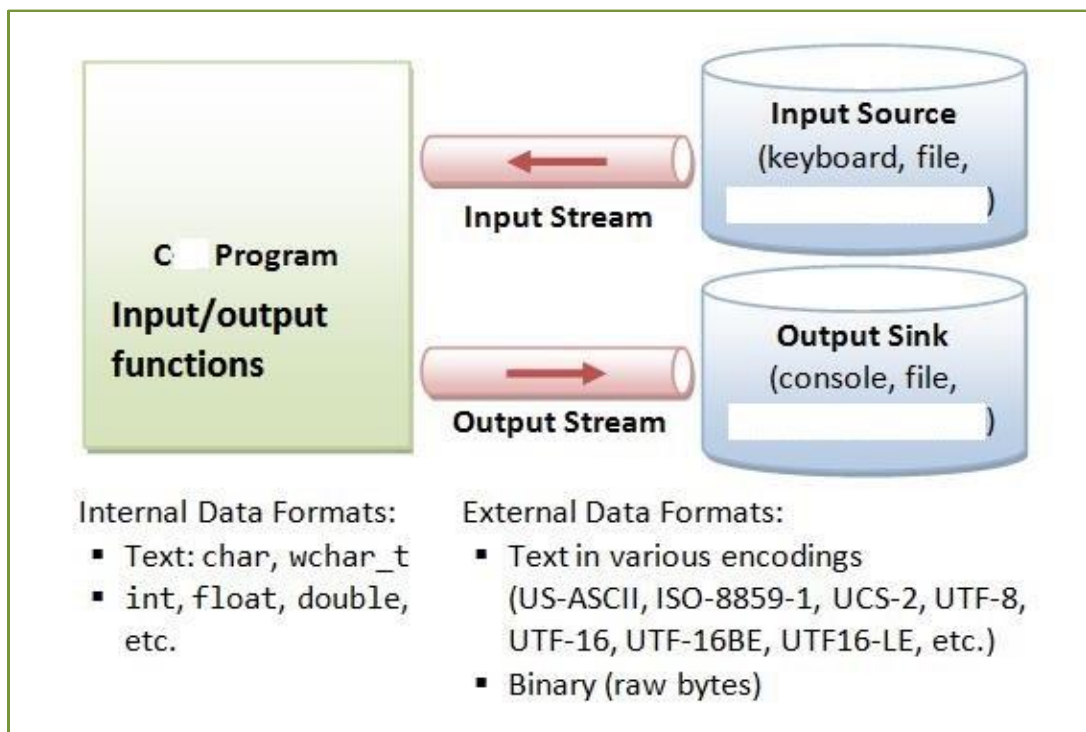


Рисунок 7.1 – Організація обміну даними в мові С

Отже, програмісту надається наступна “формула” для оволодіння інструментом обміну даними в С-програмах:

ФАЙЛ -------> ПОТІК -------> ФУНКЦІЇ ВВ/ВИВ.

Для роботи з потоками програмісту треба ознайомитись з похідним типом даних FILE.

Згідно стандарту, тип FILE є типом даних для представлення об’єктів – потоків.

Об’єкт типу FILE містить інформацію для управління потоком, зокрема:

- індикатор позиції у файлі,
- вказівник на асоційований буфер (якщо такий є),
- індикатор помилки, яка відбулася при вв/вив,
- індикатор кінця файлу, в який записується, чи був досягнутий кінець файлу.

Якщо потік не буферизований (unbuffered), символи мають передатись (ввестися / вивестися) якнайшвидше.

Якщо потік буферизований, символи можуть накопичуватись у буфері і передаватись як блок.

Якщо потік повністю буферизований (fully buffered), символи передадуться при наповненні буфера. Якщо потік буферизується рядком (line buffered), символи передадуться блоком, коли зустрінеться символ кінця рядка.

Ось для прикладу фрагмент файлу stdio.h, який показує декларацію типу даних FILE. З декларації видно, що тип даних FILE – це похідний тип – структура з тегом _iobuf. Значення елементів структури використовуються середовищем виконання для управління потоком.

```
…
typedef struct  _iobuf
{
   char*   _ptr;
   int _cnt;
   char*   _base;
   int _flag;
   int _file;
   int _charbuf;
   int _bufsiz;
   char*   _tmpfname;
} FILE;
…
```

Тип FILE запропоновано для зручності, для підвищення рівня програмування вв/вив в С. Програмісту аж ніяк не потрібно вникати в деталі елементів структури, тим більше, що вміст stdio.h залежить від реалізації.

Коли запускається функція main, у неї вже є три попередньо визначені стандартні потоки, відкриті та доступні для використання. Ці потоки декларуються у файлі заголовка stdio.h:

FILE * stdin	- стандартний потік введення, наприклад, з клавіатури
FILE * stdout	- стандартний потік виведення, наприклад, на монітор
FILE * stderr	- стандартний потік для виведення повідомлень про помилки та діагностики при виконанні програми

Отже, програміст має виконати такі кроки для організації роботи програми з файлами:

1. Визначитися з іменами потрібних файлів. Ім’я файлу - це рядок символів.

2. Написати декларації вказівників на асоційовані з файлами потоки. Вказівник потоку - це вказівник на об'єкт типу FILE; цей об'єкт описує асоційований з файлом потік. Для стандартних потоків введення з клавіатури та виведення на екран цей пункт пропускається, оскільки підтримується системою.

3. Відкрити файли за допомогою відповідних функцій. Отримати від цих функцій значення вказівників на асоційовані потоки.

4. Застосувати вказівники при викликах функцій вв/вив, обраних для організації подальшого “руху” даних.

5. Подбати про дані, які могли залишитись в тимчасовому буфері. Закрити потоки і тим самим закрити файли. Процеси обміну даними закінчено, задіяні ресурси вивільнено.

Приклад. У програмі декларовано два вказівники p, q на тип FILE, і обидва вони стосуються одного і того ж файлу з іменем day_care.txt, але призначені для роботи в різних текстових потоках в різних режимах.

```
#include<stdio.h>
struct baby
{
   char name[5];
   int age;
};
void main()
{
   struct baby e;
   FILE *p,*q;
   p = fopen("day_care.txt", "a");      /* відкриття файлу в режимі "a", отримання значення
                                           вказівника першого текстового потоку */
   q = fopen("day_care.txt", "r");      /* відкриття файлу в режимі "r", отримання значення
                                           вказівника другого текстового потоку */
   printf("Enter baby's name and age:");   /* вив на консоль рядка – запрошення для вв;
                                              потік – стандартний stdout */
   scanf("%s %d", e.name, &e.age);   /* вв даних і їх розміщення у вказаних об'єктах ОП;
                                        потік – стандартний stdin */
   fprintf(p,"%s %d", e.name, e.age);        /* вив даних у файл */
   fclose(p);                                /* закриття потоку і файлу */
   do
   {
      fscanf(q,"%s %d", e.name, e.age);    /* вв даних з файлу */
      printf("%s %d", e.name, e.age);       /* вив даних на консоль */
   }
   while(!feof(q));              /* ітерації продовжуються до виявлення кінця файлу */
}
```

Приклад. У фрагменті програми є запис рядка символів "Complete Ghost Stories by James" у бінарний потік bfp (пов'язаний з файлом "novel.txt") за допомогою С-функції fwrite.

```
const char *text = "Complete Ghost Stories by James";
FILE *bfp= fopen("novel.txt", "wb");
if (bfp)                                   /* перевірка успішності відкриття файлу */
{
   fwrite(text, sizeof(char), strlen(text), bfp);
   fclose(bfp);
}
```

Операції над файлами

Табл. 7.2 містить опис бібліотечних функцій, які здійснюють наступні операції над файлами:
- видалення файлу,
- перейменування файлу,
- створення тимчасового файлу,
- генерація унікального тимчасового імені файлу.

Таблиця 7.2 – Функції операцій над файлами

Функція	Декларація функції	Призначення функції
remove повертає нуль, якщо операція успішна	#include <stdio.h> int remove(const char *filename);	задане в аргументі ім'я файлу більше не буде доступним
rename повертає нуль, якщо операція успішна	#include <stdio.h> int rename(const char *old, const char *new);	файл, чиє ім'я задано в першому аргументі, буде доступним надалі за іменем, заданим другим аргументом
tmpfile повертає вказівник на потік створеного файлу	#include <stdio.h> FILE *tmpfile(void);	створює тимчасовий двійковий файл, який відрізняється від будь-якого іншого існуючого файлу і буде автоматично видалений під час закриття або при завершенні програми
tmpnam повертає вказівник на рядок імені; якщо генерація не вдасться, то null	#include <stdio.h> char *tmpnam(char *s);	генерує рядок, який є допустимим тимчасовим унікальним іменем файлу

Функції доступу до файлів

Табл. 7.3 містить опис бібліотечних функцій доступу до файлів:
- відкриття файлу,
- закриття файлу,
- очищення буферу,
- буферізація потоку.

Таблиця 7.3 - Функції доступу до файлів

Функція	Декларація функції	Призначення функції
fopen повертає вказівник на об'єкт типу FILE, який ідентифікує асоційований з файлом потік. Якщо операція не вдалася, повертається null-вказівник	#include <stdio.h> FILE *fopen(const char * restrict filename, const char * restrict mode);	відкриває файл, чиє ім'я – рядок задано в першому аргументі; асоціює з файлом потік. Режим відкриття задається другим аргументом. Можливі значення режиму: r - відкриття текстового файлу для читання; w - обрізка до нульової довжини або створення текстового файлу для запису; wx - створення текстового файлу для запису; a - відкриття або створення текстового файлу

		для запису в кінець файлу; rb – відкриття двійкового файлу для читання; wb - обрізка до нульової довжини або створення двійкового файлу для запису; wbx - створення двійкового файлу для запису; ab - відкриття або створення двійкового файлу для запису в кінець файлу; r+ - відкриття текстового файлу для оновлення (тобто читання та запису); w+ - обрізка до нульової довжини або створення текстового файлу для оновлення; w+x - створення текстового файлу для оновлення; a+ - відкриття або створення текстового файлу для оновлення, запис в кінець файлу; r+b або rb+ - відкриття двійкового файлу для оновлення; w+b або wb+ - обрізка до нульової довжини або створення двійкового файлу для оновлення; w+bx або wb+x - створення двійкового файлу для оновлення; a+b або ab+ - відкриття або створення двійкового файлу для оновлення, запис в кінець файлу
fclose повертає нуль, якщо потік був успішно закритий, або **EOF**, якщо трапились помилки	#include <stdio.h> int fclose(FILE *stream);	вказаний в аргументі потік очищується, а пов'язаний з ним файл закривається
fflush повертає нуль при успіху операції, або **EOF**, якщо трапились помилки	#include <stdio.h> int fflush(FILE *stream);	будь-які недозаписані для вказаного вихідного потоку дані доставлятимуться до хост-середовища для їх запису у файл
freopen повертає значення потоку. Якщо виникне помилка, то null-вказівник	#include <stdio.h> FILE *freopen(const char * restrict filename, const char * restrict mode, FILE * restrict stream);	пов'язує нове ім'я файлу (перший аргумент) з уже відкритим потоком (третій аргумент) і в той же час закриває старий файл у потоці
setbuf нічого не повертає	#include <stdio.h> void setbuf(FILE * restrict stream, char * restrict buf);	еквівалентна функції setvbuf, яка викликається зі значеннями **_IOFBF** для типу буферізації та **BUFSIZ** для розміру буфера, або (якщо buf є нульовим вказівником), зі значенням **_IONBF** для типу
setvbuf повертає нуль при успішному завершенні або не нуль в іншому випадку	#include <stdio.h> int setvbuf(FILE * restrict stream, char * restrict buf, int mode, size_t size);	визначає, як буде буферізовано вказаний потік (перший аргумент). Другий аргумент задає адресу буфера; якщо адреса - **NULL**, то функція виділяє буфер зазначеного розміру (четвертий аргумент) автоматично. Третій аргумент вказує тип буферізації і може мати значення **_IOFBF** (повна буферізація вв/вив), **_IOLBF** (буферізація по рядках) або **_IONBF** (буферізації немає)

Функції форматованого введення та виведення

Табл. 7.4 містить опис бібліотечних функцій форматованого вв/вив.

Форматоване виведення перетворює внутрішнє двійкове представлення даних у символи, які записуються у файл.

Форматоване введення зчитує символи з файлу і перетворює їх у внутрішню форму.

Форматоване вв/вив дуже портативне: файли з форматованими даними легко переносити на різні комп'ютери з різними операційними системами, звісно, якщо всі вони використовують стандартний набір символів (ASCII). Форматовані файли є читабельними для людей і можуть бути набрані з клавіатури або відредаговані за допомогою текстового редактора.

Проте для роботи з форматованими даними потрібно більше ресурсів часу та пам'яті, аніж з неформатованими.

Таблиця 7.4 - Функції форматованого введення та виведення

Функція	Декларація функції	Призначення функції
fprintf повертає кількість переданих символів або від'ємне значення при виникненні помилки	#include <stdio.h> int fprintf(FILE * restrict stream, const char * restrict format, ...);	записує виведення у потік, на який вказує перший аргумент. Виведення даних відбувається під управлінням рядка - формату, на який вказує другий аргумент. Формат визначає, як перетворюватимуться наступні аргументи перед виведенням. Якщо аргументів для формату недостатньо, поведінка не визначена. Якщо формат вичерпаний, а аргументи залишаються, то надлишкові аргументи оцінюються (як завжди), але надалі ігноруються
fscanf повертає значення макросу **EOF**, якщо вхідна помилка відбувається до завершення першого перетворення (якщо таке є). В іншому випадку функція повертає кількість введених елементів	#include <stdio.h> int fscanf(FILE * restrict stream, const char * restrict format, ...);	читає введення з потоку, на який вказує перший аргумент. Введення відбувається під управлінням рядка - формату, на який вказує другий аргумент. Формат визначає, як перетворюватимуться введені дані при присвоєнні наступним аргументам. Ці аргументи є вказівниками на об'єкти, в які попадуть перетворені дані. Якщо аргументів для формату недостатньо, поведінка програми не визначена. Якщо формат вичерпаний, а аргументи залишаються, то надлишкові аргументи оцінюються (як завжди), але надалі ігноруються
printf повертає кількість переданих символів або від'ємне значення при виникненні помилки	#include <stdio.h> int printf(const char * restrict format, ...);	еквівалентна функції fprintf з першим аргументом stdout, який є стандартним потоком і не вказується явно
scanf повертає значення макросу **EOF**, якщо вхідна помилка відбувається до завершення першого перетворення (якщо таке є). В іншому випадку функція повертає кількість	#include <stdio.h> int scanf(const char * restrict format, ...);	еквівалентна функції fscanf з першим аргументом stdin, який є стандартним потоком і не вказується явно

введених елементів		
snprintf	#include <stdio.h> int snprintf(char * restrict s, size_t n, const char * restrict format, ...);	опис функцій можна знайти в тексті стандарту
sprintf	#include <stdio.h> int sprintf(char * restrict s, const char * restrict format, ...);	
sscanf	#include <stdio.h> int sscanf(const char * restrict s, const char * restrict format, ...);	
vfprintf	#include <stdarg.h> #include <stdio.h> int vfprintf(FILE * restrict stream, const char * restrict format, va_list arg);	
vfscanf	#include <stdarg.h> #include <stdio.h> int vfscanf(FILE * restrict stream, const char * restrict format, va_list arg);	
vprintf	#include <stdarg.h> #include <stdio.h> int vprintf(const char * restrict format, va_list arg);	
vscanf	#include <stdarg.h> #include <stdio.h> int vscanf(const char * restrict format, va_list arg);	
vsnprintf	#include <stdarg.h> #include <stdio.h> int vsnprintf(char * restrict s, size_t n, const char * restrict format, va_list arg);	
vsprintf	#include <stdarg.h> #include <stdio.h> int vsprintf(char * restrict s, const char * restrict format, va_list arg);	
vsscanf	#include <stdarg.h> #include <stdio.h> int vsscanf(const char * restrict s, const char * restrict format, va_list arg);	

Функції символьного введення та виведення

Табл. 7.5 містить опис бібліотечних функцій вв/вив символів.

Таблиця 7.5 - Функції символьного введення та виведення

Функція	Декларація функції	Призначення функції
fgetc повертає наступний символ з потоку введення або **EOF**, якщо	#include <stdio.h> int fgetc(FILE *stream);	отримує символ як тип unsigned char, перетворює у тип int, просуває індикатор позиції у файлі для вказаного в аргументі потоку

трапилась помилка або досягнуто кінець файлу		
fgets повертає вміст s або null-вказівник при помилці читання	#include <stdio.h> char *fgets(char * restrict s, int n, FILE * restrict stream);	читає з вказаного потоку (третій параметр) в масив, на який вказує перший параметр. Кількість прочитаних символів, включаючи символ нового рядка, не перевищує значення другого аргумента. Null -символ записується відразу після того, як в масив прочитано останній символ
fputc повертає записаний символ. Якщо виникає помилка запису, повертається EOF	#include <stdio.h> int fputc(int c, FILE *stream);	пише символ, заданий першим аргументом (перетворений в unsigned char), до заданого потоку, в позицію, яка відповідає значенню індикатора
fputs повертає EOF, якщо виникає помилка запису; в іншому випадку повертає невід'ємне значення	#include <stdio.h> int fputs(const char * restrict s, FILE * restrict stream);	записує рядок, на який вказує перший аргумент, до вказаного потоку (другий аргумент). Завершаючий Null -символ не записується
getc	#include <stdio.h> int getc(FILE *stream);	опис функцій можна знайти в тексті стандарту
getchar	#include <stdio.h> int getchar(void);	
putc	#include <stdio.h> int putc(int c, FILE *stream);	
putchar	#include <stdio.h> int putchar(int c);	
puts	#include <stdio.h> int puts(const char *s);	
ungetc	#include <stdio.h> int ungetc(int c, FILE *stream);	

Література: [2], стор. 217-247.

КОНТРОЛЬНІ ПИТАННЯ

1. Що таке поток в мові С?
2. Як реалізовано введення та виведення даних в мові С?
3. Що означає форматування введення та виведення даних в мові С?
4. Наведіть фрагмент програми з використанням типу FILE.

Лекція 8. Твердження та блоки

Твердження в мові С

Послідовністю тверджень (операторів) програміст описує дії для реалізації алгоритму.

Кожна С-програма складається з тверджень і виконується твердження за твердженням. Кожне твердження, як правило, складається з певних виразів. В свою чергу, вираз може додатково містити підвирази.

В табл. 8.1 наведено всі доступні в мові С інструменти для організації обчислювального процесу.

Таблиця 8.1 – Перелік видів тверджень в мові С

Твердження та блоки	
Твердження з міткою **Складене твердження (блок)** **Твердження-вирази та пусті твердження** **Твердження вибору** Твердження if Твердження switch	**Твердження ітерації** Твердження while Твердження do Твердження for **Твердження швидкого переходу** Твердження goto Твердження continue Твердження break Твердження return

Складене твердження (блок)

Базовий приклад. Все, що взято у фігурні дужки, є блоком. Рядки 77-105.

Твердження-вирази та пусті твердження

Вираз, до якого додається крапка з комою, стає твердженням.

Приклад. Фрагмент програми складається з 4-х тверджень.

```
x + y;                          /* твердження-вираз, де x + y є виразом */
5 + 7;                          /* твердження-вираз, де 5+7 є виразом */

result = (float) (x * y) / (u + v);                 /* твердження-вираз */
okey = (x * y) / (u + w) * (z = a + b + c + d);     /* твердження-вираз */

/*  (z = a + b + c + d) є підвиразом */
```

Пусте твердження складається з крапки з комою і не виконує жодних операцій.

Пусті твердження зазвичай використовуються як заповнювачі в твердженнях ітерації або як твердження з міткою в кінці функцій або складених тверджень.

Приклад. Наступне твердження ітерації знаходить перший елемент масиву, який має значення 0. Твердження for виконується тільки для його побічних ефектів; тіло циклу є пустим, точніше, містить пусте твердження.

```
for (i=0; array[i] != 0; i++) ;
```

Твердження вибору

До вибору належать твердження if та switch. Порядок виконання кожного з них наведено на рис. 8.1, 8.2.

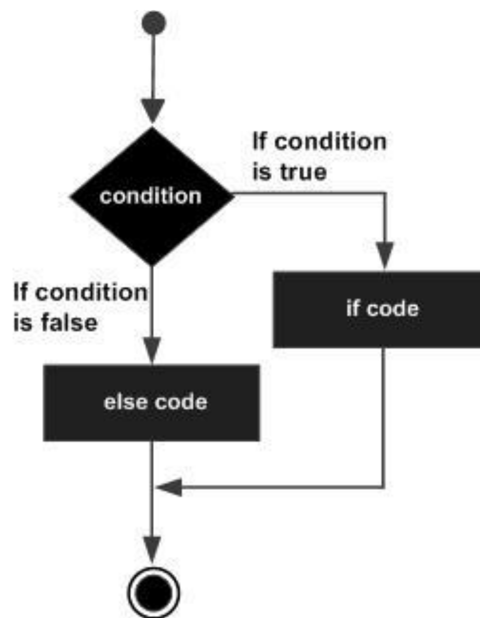


Рисунок 8.1 – Порядок виконання if

Базовий приклад. Рядки 15, 25, 34, 37, 62, 75, 91.

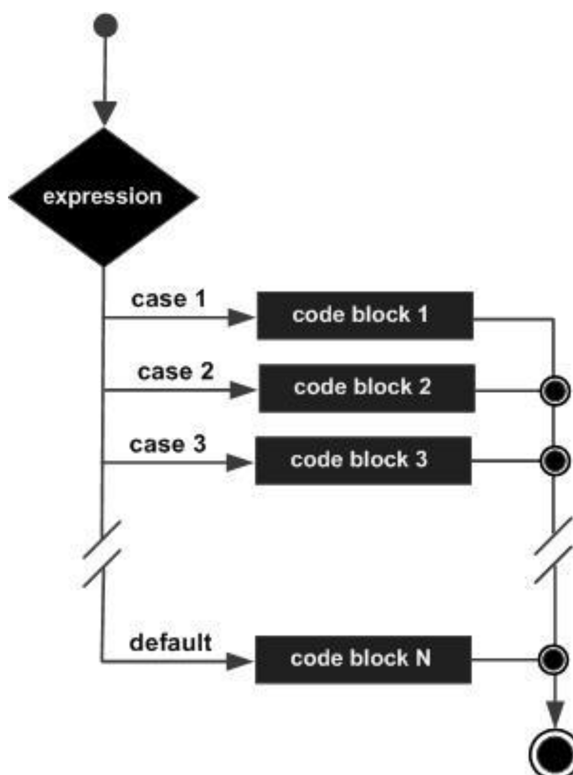


Рисунок 8.2 – Порядок виконання switch

Базовий приклад. Рядки 83-105.

Твердження ітерації

До ітерацій належать твердження while, do, for. Порядок виконання кожного з них наведено на рис. 8.3, 8.4, 8.5.

Базовий приклад. Рядки 50, 61, 105.

Базовий приклад. Рядки 77-105.

Базовий приклад. Рядки 17, 49.

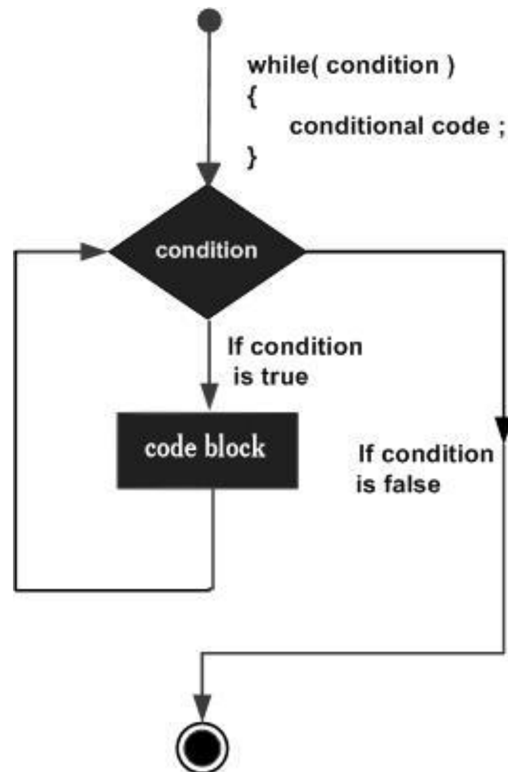


Рисунок 8.3 – Порядок виконання while

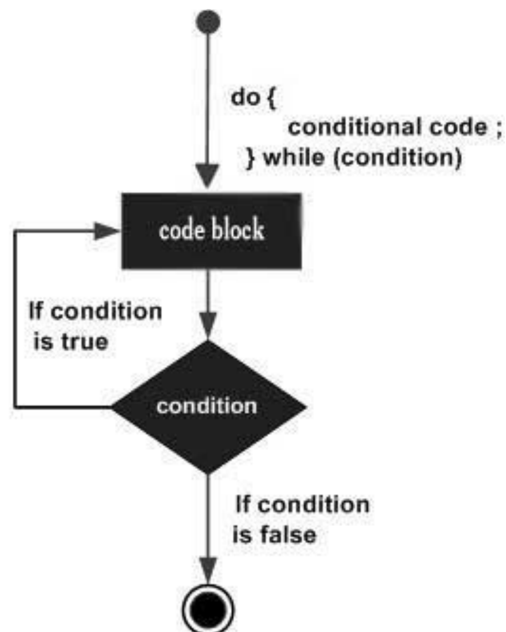


Рисунок 8.4 – Порядок виконання do

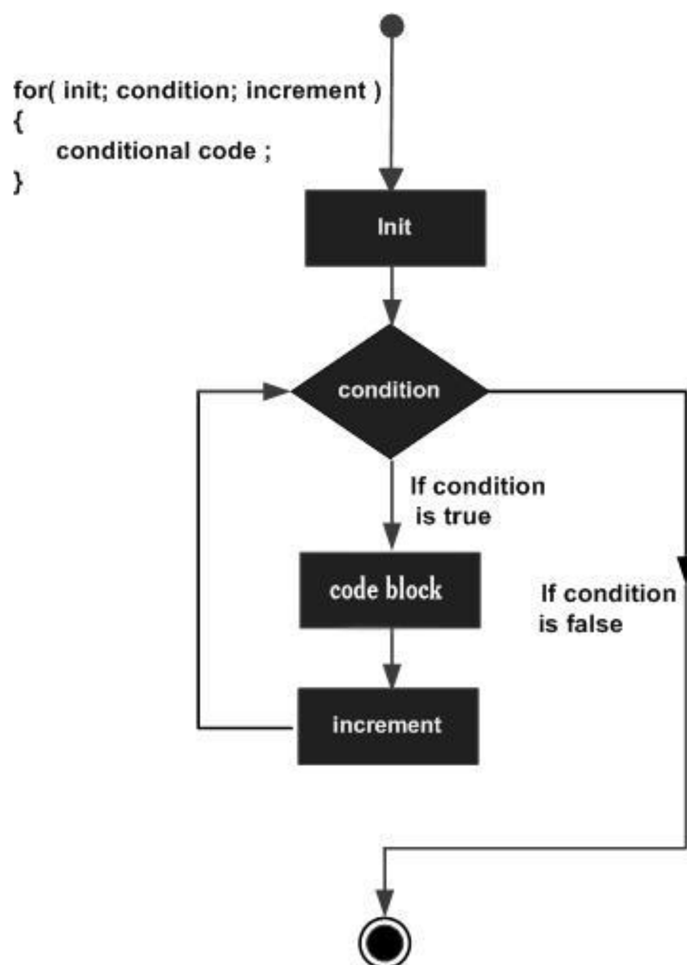


Рисунок 8.5 – Порядок виконання for

Твердження швидкого переходу

До швидкого переходу належать твердження goto, continue, break, return.

Твердження break призводить до завершення виконання ітераційних тверджень do, for або while, а виконання програми продовжується з наступного твердження.

В контексті твердження switch твердження break зазвичай використовується в кінці кожного кейса для його завершення (запобігаючи ситуації fall-through, коли, крім потрібного, виконуються також і нижче розташовані кейси).

Твердження return завершує виконання всієї функції, в якій знаходиться ітерація, а виконання продовжується в точці, наступній після виклику функції.

Твердження continue дозволяє відразу перейти в кінець тіла ітерації, пропускаючи весь код, який знаходиться під ним. Це корисно в тих випадках, коли ми хочемо завершити поточну ітерацію завчасно.

Приклад. У програмі використано твердження continue.

```
#include<stdio.h>
int main()
{ int i;
  for(i=0;i<5;i++)
 {
   if(i==3)
  {
    continue;
   }
printf("%d",i\t);
  }
}
```

Результат : 0 1 2 4

Базовий приклад. Твердження break. Рядки 63, 87, 95, 98, 101, 103.

Базовий приклад. Твердження return. Рядки 16, 19, 35, 42, 65, 107.

Твердження з міткою та goto

Приклад. Прикладом використання оператора goto є відокремлення основної логіки програми від логіки обробки помилок. У функції f() замість обробки помилок в різних частинах цієї функції, програмістом використано оператор goto і функцію handle_errors () для обробки помилок.

```
void f()
{
   int error;
   /* алгоритм */
   if ( /* error */)
      {
            error = 1;
            goto error_handler;
      }
    /* алгоритм */
   if ( /*error */)
      {
      error = 2;
            goto error_handler;
       }
    /* алгоритм */
   if ( /*error */)
      {
            error = 3;
            goto error_handler;
       }
```

```
error_handler:  handle_errors(error);                    /* твердження з міткою */
    ;
}
```

Література: [2], стор. 106-112.

КОНТРОЛЬНІ ПИТАННЯ

1. Яке призначення пустих виразів в мові С?
2. Наведіть приклади використання тверджень ітерації.
3. Наведіть приклади використання тверджень вибору.
4. Наведіть приклади використання тверджень швидкого переходу.

Лекція 9. Директиви препроцесора (частина 1)

Директиви препроцесора в мові С

Згідно стандарту, за допомогою директив препроцесора можна обробляти та оминати частини початкових С-файлів, додавати вміст інших файлів, працювати з макросами. Ці можливості називаються препроцесінгом тому, що концептуально вони реалізуються перед трансляцією.

Препроцесор має свій власний синтаксис, який відрізняється від синтаксису С. Всі директиви препроцесора починаються зі знака хешу (#), наприклад, #define, #include.

В табл. 9.1 наведено всі наявні в мові С директиви препроцесора, згідно стандарту.

Таблиця 9.1 – Перелік директив препроцесора в мові С

Директиви препроцесора
Умовна підстановка #if, #else, #elif, #ifdef, #ifndef, #endif, #if defined
Включання початкового файлу #include
Макропідстановки
Управління рядком
Директива помилки
Директива Pragma
Пуста директива
Зарезервовані імена макросів
Оператор Pragma

Умовна підстановка

У С програмуванні програміст може проінформувати препроцесор, чи слід включати в остаточний текст програми перед трансляцією певний шматок коду чи ні. Для цього є так звані умовні директиви #if, #else, #elif, #ifdef, #ifndef і #endif. Вони схожі на твердження if. Однак, є велика різниця, яку потрібно зрозуміти. Твердження if перевіряється під час виконання програми, щоб визначити, чи

повинен бути виконаний той чи інший блок коду. Умовна ж директива #if використовується до виконання програми на етапі трансляції.

Програміст використовує умовні директиви, коли:

- потрібні дещо різні коди залежно від компьютера, операційної системи;
- треба компілювати один і той же початковий файл різними програмами;
- потрібно виключити певний код з програми, але зберегти його для майбутнього.

Приклад.

```
#define ABCD 2
#include <stdio.h>
 int main(void)
{
 #ifdef ABCD
    printf("1: yes\n");
#else
    printf("1: no\n");
#endif
             #ifndef ABCD
             printf("2: no1\n");
             #elif ABCD == 2
             printf("2: yes\n");
             #else
              printf("2: no2\n");
             #endif
 #if !defined(DCBA) && (ABCD < 2*4-3)
    printf("3: yes\n");
#endif
}
```

Після обробки препроцесором текст функції main матиме вигляд:

```
int main(void)
{

   printf("1: yes\n");
                     printf("2: yes\n");
   printf("3: yes\n");
}
```

Результат:

```
1: yes
2: yes
3: yes
```

Включення початкового файлу

Директивою #include програміст може додати в поточний початковий файл тексти інших початкових файлів, які в цьому контексті мають назву заголовних (від англійського header). Текст вставиться, починаючи з рядка відразу після директиви #include. Альтернативою цьому способу є ручний набір в кожен файл.

Є два типи заголовних файлів: файли, які створює сам програміст, і файли, що входять до бібліотеки С-компілятора.

Приклад. stdio.h - стандартний файл, myprog.h – ім’я користувацького файлу.

```
#include <stdio.h>
#include "myprog.h"
```

Практика програмування на С полягає в тому, що програміст зберігає у файлах заголовків:

- константи,
- макроси,
- глобальні змінні,
- прототипи функцій,
- функції зі специфікатором inline або static.

Програмісту-початківцю можна порекомендувати наступне розроблення с-файлів та h-файлів. Якщо програма не дуже маленька:

- розмістити тексти функцій в декількох файлах з розширенням .с;
- для кожного файлу .c створити файл з тим же ім'ям, але з розширенням h;
- у файл .h набрати прототип для кожної функції, визначеної у файлі .c;
- у файлі .c набрати команду #include для файлу .h (це гарантуватиме, що прототипи функцій збігаються з визначеннями функцій).

Надалі можна включити ці файли .h у будь-який інший файл .c, який використовує будь-яку з цих функцій (це гарантує, що код викличе кожну функцію правильно, з точки зору кількості параметрів, типів параметрів та типу значення, що повертається).

Приклад. Нехай у файлі head.h програміст помістив такий текст:

```
char *test (void);
```

а у файлі program.c – текст основної програми:

```
int x;
#include "head.h"
int main (void) {
puts (test ());
                }
```

На компілятор поступає оновлений текст – результат обробки тексту основної програми препроцесором:

```
int x;
```

```
char *test (void);
int main (void) {
  puts (test ());
}
```

Зауваження. Не рекомендується розміщувати будь-який виконуваний код (наприклад, тіло функції) у заголовному файлі.

Макропідстановки

Програміст може визначити макрос, використовуючи директиву препроцесора #define. Макрос по суті - це фрагмент коду, якому дано ім'я.

Приклад. Програміст використав директиву для опису макроса з іменем PI.

```
#include <stdio.h>
#define PI 3.1415       /* макровизначення */

int main()
{
   float radius, area;
   printf("Enter the radius: ");
   scanf("%d", &radius);

  area = PI*radius*radius;     /* замість PI препроцесор підставить 3.1415
   printf("Area=%.2f",area);
   return 0;
}
```

Можна також визначити макроси з параметрами, які працюють як аналог виклику функції, так звані функціональні макроси.

Приклад. Рядок тексту С-програми `area = circleArea(radius);` препроцесор замінить на рядок `area = 3.1415*radius*radius`.

```
#include <stdio.h>
#define PI 3.1415
#define circleArea(r) (PI*r*r)           /* макрос з іменем circleArea та параметром r */
int main()
{
   int radius;
   float area;
   printf("Enter the radius: ");
   scanf("%d", &radius);
   area = circleArea(radius);            /* макропідстановка */
   printf("Area = %.2f", area);
   return 0;
}
```

Після обробки препроцесором текст функції `main` матиме вигляд:

```
int main()
```

```
{
   int radius;
   float area;

   printf("Enter the radius: ");
   scanf("%d", &radius);
   area = 3.1415*radius*radius;                    /* застосування макроса circleArea */

   printf("Area = %.2f", area);

   return 0;
}
```

Література: [2], стор. 117-126.

КОНТРОЛЬНІ ПИТАННЯ

1. Що таке препроцесор?
2. Яке призначення у директиви умовної підстановки?
3. Чому С-програміст повинен використовувати заголовні файли?
4. Напишіть фрагмент програми із застосуванням макровизначення #define Max(a,b) ((a) > (b) ? (a) : (b)).

Лекція 10. Введення та виведення даних (частина 2)

Функції прямого введення та виведення

Табл. 10.1 містить опис стандартних бібліотечних функцій прямого вв/вив.

Неформатоване вв/вив є основною формою введення / виведення. Неформатоване вв/вив передає внутрішнє двійкове представлення даних безпосередньо між пам'яттю і файлом.

Неформатоване вв/вив є найменш портативною формою введення / виводу. Неформатовані файли можна легко переміщувати лише між комп'ютерами, які мають однакове внутрішнє представлення даних. Неформатоване вв/вив не є читабельним для людини, тому ви не можете набрати його на моніторі або редагувати за допомогою текстового редактора.

Таблиця 10.1 – Функції прямого введення та виведення в мові С

Функція	Декларація функції	Призначення функції
fread повертає кількість успішно прочитаних елементів	#include <stdio.h> size_t fread(void * restrict ptr, size_t size, size_t nmemb, FILE * restrict stream);	читає з заданого потоку (четвертий аргумент) в масив, на який вказує перший аргумент , задану третім аргументом кількість елементів, розмір кожного з яких задано другим аргументом
fwrite повертає кількість успішно записаних елементів	#include <stdio.h> size_t fwrite(const void * restrict ptr, size_t size, size_t nmemb, FILE * restrict stream);	записує в заданий потік (четвертий аргумент) з масиву, на який вказує перший аргумент , задану третім аргументом кількість елементів, розмір кожного з яких задано другим аргументом

Функції позиціювання у файлах

Табл. 10.2 містить опис стандартних бібліотечних функцій позиціонування:

- отримання значення індикатора поточної позиції у файлі для заданого потоку;

- встановлення значення індикатора поточної позиції у файлі для заданого потоку.

Таблиця 10.2 – Функції позиціювання у файлах

Функція	Декларація функції	Призначення функції
fgetpos у разі успіху повертає нуль; у разі невдачі повертає ненульове значення і зберігає визначене реалізацією позитивне значення в errno	#include <stdio.h> int fgetpos(FILE * restrict stream, fpos_t * restrict pos);	отримує значення індикатора поточної позиції у файлі і записує його в об'єкт, на який вказує вказівник pos (другий аргумент)
fseek повертає ненульове значення, якщо запит не може бути задоволений	#include <stdio.h> int fseek(FILE *stream, long int offset, int whence);	встановлює індикатор позиції у файлі для вказаного у першому аргументі потоку. Для двійкового потоку нова позиція

		від початку файлу отримується шляхом додавання зміщення offset та позиції, визначеної whence (третій аргумент). Якщо третій аргумент - SEEK_SET, то додавання буде проводитись від початку файлу, якщо SEEK_CUR, то від поточного значення, якщо SEEK_END, то від кінця файлу. Для текстового потоку offset повинен бути або нульовим, або значенням, поверненим раніше успішним викликом функції ftell в потоці, асоційованому з тим самим файлом, а третій аргумент повинен мати значення SEEK_SET
fsetpos у разі успіху повертає нуль; при невдачі повертає ненульове значення	#include <stdio.h> int fsetpos(FILE *stream, const fpos_t *pos);	встановлює індикатор позиції у файлі для заданого потоку відповідно до значення об'єкта, на який вказує другий параметр. Це значення повинно бути значенням, отриманим від попереднього успішного виклику функції fgetpos для потоку, асоційованого з цим же файлом
ftell повертає поточне значення індикатора позиції у файлі; при невдачі повертає -1L	#include <stdio.h> long int ftell(FILE *stream);	отримує поточне значення індикатора позиції у файлі для вказаного потоку. Для двійкового потоку значенням є кількість символів від початку файлу. Для текстового потоку індикатор позиції містить невизначену інформацію, яку, тим не менш, можна використовувати для fseek
rewind не повертає значення	#include <stdio.h> void rewind(FILE *stream);	встановлює індикатор позиції у файлі на початок файлу

Функції обробки помилок

Табл. 10.3 містить опис стандартних бібліотечних функцій обробки помилок.

Таблиця 10.3 – Функції обробки помилок

Функція	Декларація функції	Призначення функції
clearerr не повертає значення	#include <stdio.h> void clearerr(FILE *stream);	очищує для вказаного потоку індикатори кінця файлу та помилки
feof повертає ненульове значення тільки тоді, коли встановлений індикатор кінця файлу	#include <stdio.h> int feof(FILE *stream);	перевіряє індикатор кінця файлу для вказаного потоку
ferror повертає ненульове значення тільки тоді, коли встановлений індикатор помилки	#include <stdio.h> int ferror(FILE *stream);	перевіряє індикатор помилки для вказаного потоку

perror не повертає значення	#include <stdio.h> void perror(const char *s);	виводить у стандартний error потік заданий рядок; далі номер останньої помилки (береться з стандартного виразу errno) і її опис

Приклад. Нехай файл langs.txt містить наступні дані:

Java
Python
C
C++
Ruby
Perl

Наступна програма читає ці дані і потім виводить на консоль для перевірки, також обчислюється кількість прочитаних знаків, включаючи символи нового рялка.

```
#include<stdio.h>
int main() {
  char buffer[40];
  FILE * stream;
  stream = fopen("langs.txt", "r");
  int count = fread(&buffer, sizeof(char), 30, stream);
  fclose(stream);
  printf("Data read from file: %s \n", buffer);
  printf("Read: %d", count);
  return 0;
}
```

Результат:
Data read from file:
Java
Python
C C++
Ruby
Perl
Read: 27

Література: [2], стор. 217-247.

КОНТРОЛЬНІ ПИТАННЯ

1. Що таке пряме введення/виведення даних?
2. Напишіть фрагмент програми з функцією прямого введення або виведення.
3. Напишіть фрагмент програми з функцією позиціювання.
4. Для чого призначені стандартні функції обробки помилок?

Лекція 11. Середовище програмування

Згідно стандарту, реалізація (імплементація) мови C - це два середовища. Одне середовище називають середовищем трансляції. В ньому можна виконати трансляцію початкового файлу .c, компоновку об'єктних модулів .obj у виконуваний файл .exe. Друге середовище називають середовищем виконання C-програми. Саме характеристики цих середовищ визначають і обмежують результати виконання відповідних C-програм (рис. 11.1).

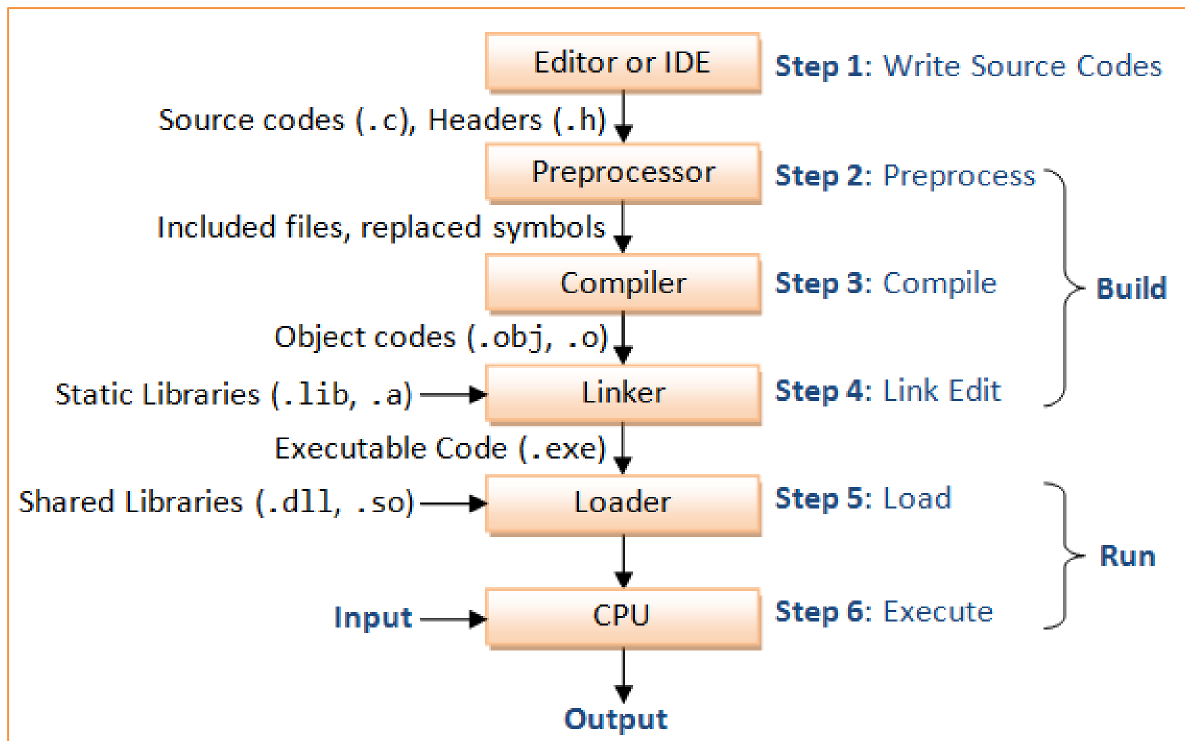


Рисунок 11.1 - Елементи середовища C-програмування

Обробка C-програми в обраному програмістом середовищі здійснюється в декілька етапів:

Етап 1. Ви починаєте з набору тексту своєї програми в якомусь редакторі або інтегрованому середовищі розробки. Зберігаєте у файлах з розширенням .c (текстовий файл) та .h (так званий файл заголовків).

Етап 2. Компіляція тексту програми, тобто переклад на машинну мову. Першим відпрацьовує препроцесор – частина транслятора. Препроцесор знаходить у вихілному коді призначені саме для нього команди, обробляє їх та передає отриманий текст (так звану трансляційну одиницю) далі на основний транслятор.

Етап 3. C-програма компілюється, компілятор генерує об'єктний код, який зберігається у файлах з розширенням .obj або .o.

Етап 4. Після генерації об'єктного коду компілятор викликає системну програму компонувальник (лінкер). Лінкер приймає на вхід об'єктні модулі і збирає їх в один виконуваний модуль.

Одним з головних завдань для лінкера є надання коду функцій бібліотеки (наприклад, printf (), scanf (), sqrt ()) для вашої програми. Лінкер може виконати це завдання двома способами:

- статичним лінкуванням, тобто скопіювати код бібліотечної функції в об'єктний код;
- динамічним лінкуванням, тобто, при певній попередній підготовці, зробити так, щоб повний код функцій не копіювався, а став доступним під час виконання.

Прикладами статичних бібліотек є файли .a у Linux та файли .lib в Windows. Прикладами динамічних бібліотек є .so в Linux та .dll в Windows.

Ви можете як користатись стандартними бібліотеками, так і створювати свої, статичні або динамічні.

Етап 5. Починається виконання вашої програми. Лінкер “підтягує” потрібні функції з динамічних бібліотек .dll, .so.

Етап 6. Ваша програма виконується, ви отримаєте результати ваших зусиль.

Структура С-програми

Згідно стандарту, С-програма не обов’язково повинна бути відтрансльована вся і відразу.

Текст програми зберігається в одиницях, що називаються початковими файлами (або файлами попередньої обробки).

Початковий файл разом з усіма заголовками та файлами, включеними через директиву препроцесора #include, названо препроцесинговою трансляційною одиницею. Після здійснення препроцесингу отримуємо так звану трансляційну одиницю.

Попередньо відтрансльовані одиниці можуть зберігатися окремо або в бібліотеках. Далі вони можуть під'єднуватись до виконуваної програми.

Отже, програміст може писати свою С-програму по частинах і запускати на трансляцію також частинами (рис. 11.2). Звичайно, ніхто не відміняв можливість помістити весь текст в єдиний як завгодно великий файл.

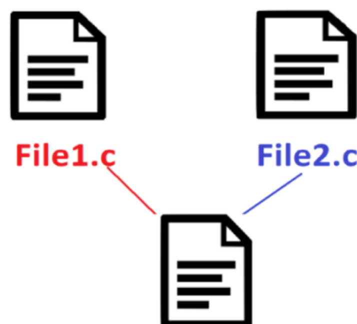


Рисунок 11.2 - Структура С-програми

Окремі трансляційні одиниці програми комунікують через, наприклад, виклики функцій, зовнішні об'єкти, операції з файлами даних.

Запуск С-програми. Функція main

Перша функція, що викликається при запуску будь-якої С-програми, називається main.

Імплементація не описує прототип для цієї функції.

Функція main буде визначена:

1) з типом int для значення, яке повертається цією функцією в середовище виконання,

2) або без параметрів; або з двома параметрами (називаються далі argc і argv, хоча можуть використовуватися будь-які імена, вони локальні до функції, в якій вони оголошені); або іншим способом, визначеним реалізацією.

Приклад. Коротюсінька програма, в якій програміст вказує, що функція працює без отримання аргументів.

```
int main (void)
{/ * ... * /}
```

Приклад. Коротюсінька програма, в якій програміст вказує, що функція працює зі списком параметрів.

```
int main (int argc, char * argv [])
{/ * ... * /}
```

Приклад. Коли операційна система викликає програму main, то може передати аргументи:

argc - кількість аргументів, переданих програмі з середовища виконання, і які містяться в другому аргументі argv;

argv - вказівник на перший елемент масиву з argc + 1 вказівників, з яких останній є нульовим, а попередні, якщо такі є, вказують на рядки, які містять аргументи, передані програмі з з середовища виконання. Якщо argv [0] не є нульовим вказівником (тобто argc> 0), він вказує на рядок, який містить ім'я програми.

```
#include <stdio.h>
int main(int argc, char *argv[])
{
   printf("argc = %d\n", argc);
   for(int  n = 0; n != argc; ++n)
      printf("argv[%d] - %s\n", n,argv[n]);
   printf("argv[argc] = %p\n", (void*)argv[argc]);
}
```

Можливий результат:

```
argc = 3
argv[0] - ./prog.out
argv[1] - infile
argv[2] - outfile
```

```
argv[argc] = (nil)
```

Припинення програми

Припинення виконання програми відбудеться, коли досягнеться:

- або твердження return з типом значення, сумісним з типом int;
- або виклик функції exit з відповідним значенням аргумента типу int;
- або остання дужка } програми, при досягненні якої в середовище обробки повертається значення 0.

Якщо тип повернення не сумісний з int, статус завершення є неспецифікованим.

Література: [2], стор. 9-27.

КОНТРОЛЬНІ ПИТАННЯ

1. Яку структуру може мати програма на мові С?
2. Що таке середовище трансляції?
3. Що таке середовище виконання?
4. В чому полягають особливості функції main?

Лекція 12. Виконання програми

Поведінка програми

Особливості виконання С-програм в деталях подано в тексті стандарту. Окремі моменти розглянуто в цій лекції.

Поведінка програми – це видимі результати її виконання.

Програміст повинен враховувати, що поведінка його С-програми може суттєво відрізнятися від очікуваної. Причин для цього вистачає: різні типи помилок в програмі, різні версії компілятора, різні архітектури середовищ виконання, різні платформи розробки тощо.

Цілий розділ стандарту присвячено опису програмних станів, які належать до тієї чи іншої поведінки. Нас же цікавить загальна концепція. Зокрема, три види поведінки програми (табл. 12.1):

1) невизначена поведінка програми;
2) неспецифікована поведінка;
3) поведінка, визначена реалізацією.

Таблиця 12.1 – Види поведінки С-програми

Вид поведінки С-програми	**Опис**
невизначена поведінка undefined behavior	Поведінка при використанні непереносної (non-portable), помилкової конструкції у програмі або помилкових даних, до яких Міжнародний стандарт не висуває жодних вимог. Результат виконання програми може бути будь-яким, як очікуваним, так і з неочікуваними помилками.
неспецифікована поведінка unspecified behavior	Використання не специфікованого значення або інша поведінка, коли Міжнародний стандарт надає дві або більше можливостей і не висуває жодних додаткових вимог до вибору з цих можливостей в будь-якому випадку. З цього випливає, що поведінка програми може змінитись на іншій архітектурі процесора, між поколіннями мікропроцесорів, під впливом налаштування оптимізації компіляторів або з будь-якої іншої причини. На перший погляд, така неоднозначність може здатися небажаною, але насправді це дозволяє підтримувати С на багатьох різних апаратних платформах, не жертвуючи продуктивністю. За відсутності вимог конкретної поведінки, розробники та продавці компіляторів можуть вільно робити все, що має сенс. Зазвичай це означає те, що є найпростішим для компілятора або найефективнішим для цільової апаратної платформи.
поведінка, визначена реалізацією implementation-defined behavior	Неспецифікована поведінка, але кожна реалізація документує спосіб вибору з можливостей. Це надає розробнику компілятора свободу реалізації, а користувачам, тобто програмістам, - гарантію поточної поведінки їх С-програм, навіть якщо ця поведінка може змінитись на іншій платформі або в наступній версії компілятора.

Приклади невизначеної поведінки. Невизначена поведінка варіюється від повного ігнорування ситуації з непередбачуваними результатами, до поведінки у задокументованій манері під час трансляції або виконання програми (з видачею діагностичного повідомлення або без нього), і до припинення трансляції чи виконання (з видачею діагностичного повідомлення).

Приклади невизначеної поведінки:

- поведінка при переповненні розрядної сітки;
- будь-яка спроба модифікувати об'єкт з обмеженим часом життя поза межами цього часового інтервалу призведе до невизначеної поведінки;
- операції з вказівником, який до цього отримав некоректне значення (наприклад, адресу пам'яті поза межами масиву), викличуть невизначену поведінку.

Приклади поведінки, визначеної реалізацією:

- поведінка, визначена особливостями трансляції, діагностичних повідомлень транслятора;
- якийсь альтернативний варіант визначення функції main;
- особливості виконання операцій з даними цілого типу та з плаваючою точкою;
- поширення знакового біта, коли знакове ціле зсувається вправо.

Прикладом неспецифікованої поведінки є порядок, в якому обчислюються значення аргументів функції.

Ця невизначеність, неуточненість або залежність від середовища розробки може зробити перенесення С-коду з однієї платформи на іншу складною місією. Навіть при написанні нового коду для незнайомої платформи програмісту може здаватися часом, що він опинився в знайомому, але трохи викривленому світі.

Зауважимо, шо тільки окремі програмісти вільно володіють концепціями та специфікаціями мови С, ознайомившись зі стандартом, але для більшості програмістів, наприклад, усвідомлення зв'язку між переносним С і невизначеною, неуточненою або залежною від реалізації поведінкою може бути відсутнім. Можна писати код, навіть не здогадуючись, що він відтвориться у тій чи іншій поведінці.

Отже, шановні С програмісти, будьте уважні та обережні!

Побічні ефекти

Програмісту треба знати, що для більшості операторів С порядок, у якому обчислюються вирази, не специфікується, тобто не уточнюється. Цей порядок визначається конкретним компілятором залежно від контексту, тому можуть виникнути деякі неочікувані результати при використанні певних операторів. Ці несподівані результати викликані так званими побічними ефектами.

Згідно стандарту, побічні ефекти при виконанні С-програми – це зміни стану середовища виконання. Іншими словами, будь-яка операція, яка змінює стан комп'ютера, зачіпає пам'ять, виділену під операнд, або взаємодіє з "зовнішнім світом", має побічний ефект.

Приклади побічних ефектів конкретної функції:

- модифікація нелокальної змінної,
- модифікація статичної локальної змінної,
- модифікація аргументу, переданого за допомогою посилання,
- виконання введення /виведення,
- виклик інших функцій з побічними ефектами.

Побічні ефекти можуть бути навмисно викликані програмістом для отримання бажаного результату; з іншого боку, оператор присвоєння по своїй суті є джерелом побічних ефектів.

За наявності побічних ефектів поведінка програми може залежати від історії, тобто порядок виконання обчислень має значення. Це треба враховувати при відлагодженні програми.

Небажані побічні ефекти зазвичай відбуваються, коли один і той самий об'єкт використовується в двох або більше місцях в одному і тому ж виразі і при цьому модифікується.

Приклад. Функція має побічний ефект виведення даних. Тут програміст розробив функцію не для того, щоб отримати значення, яке вона повертає; функція потрібна і викликається для зв’язку з "зовнішнім світом", стан якого зміниться, а саме, на зовнішньому пристрої з’явиться рядок символів (побічний ефект виконання програми).

```
int Write(const char* s)
{ return printf("Output: %s\n", s); }
```

Приклад. Щоб краще зрозуміти приклад, нагадаємо, як працюють оператори інкременту/декременту. Оператори префіксного інкременту та префіксного декременту збільшують або зменшують значення об'єкта відповідно і повертають посилання на результат. Постфіксний інкремент (декремент) створює копію об'єкта, збільшує (або зменшує) значення об'єкта, а повертає копію з початковим значенням.

Наступний фрагмент коду виробляє суперечливі результати, оскільки порядок обчислення операндів оператору присвоєння є невизначеним. Якщо спочатку збільшується змінна i, а потім обчислюється індекс масиву x, то значення x[2] дорівнюватиме 1. Якщо спочатку обчислюється індекс, то значення x [1] дорівнюватиме 1.

```
int x[4] = { 0, 0, 0, 0 };
int i = 1;
x[i] = i++;
```

Приклад. Тут значення змінної Z збільшиться на 1, і у вираз повернеться її нове значення 3. Отже, X прийме значення 2, Z - значення 2.

```
Z=1;
X=++Z;
```

Приклад. Тут значення змінної Z збільшиться на 1, а у вираз повернеться її початкове значення 3. Отже, X прийме значення 3, а Z - значення 4.

```
Z=3;
X=Z++;
```

Приклад. Мова С не гарантує порядок, у якому обчислюються аргументи виклику функції. Отже, у наступному фрагменті функція func може отримати значення 7 і 8 або 8 і 8 для своїх параметрів, в залежності від того, обчислюються вони зліва направо або справа наліво.

```
int i = 7;
func( i, ++i );
```

Приклад. Поведінка наступної програми є невизначеною, чому?

Причиною невизначеної поведінки програми є оператор "+", який не має стандартизованого порядку обчислення своїх операндів. Якщо першою виконається функція f1, то результатом програми буде виведення "KPIinKYIV". Якщо першою виконається функція f2,то результатом програми буде виведення "inKYIVKPI". Отже, результати роботи цієї програми можуть бути різними для різних компіляторів і різних машин.

```
#include <stdio.h>
int f1()
{ printf ("KPI"); return 1;}
int f2()
{ printf ("inKYIV"); return 1;}
int main()
{
        int p = f1() + f2();
        return 0;
}
```

Наведені приклади можуть ще більше розчарувати програміста-початківця. Проте, не здавайтесь! Не все так напружено в С – ідеології. Попередня інформація призвана не налякати, а попередити та озброїти програміста знаннями, які дозволять йому уникнути грубих помилок. Тим більше, що стандарт все-таки може надати нам якісь гарантії.

Далі розглянемо пов'язані з побічними ефектами так звані точки упорядкованості в програмі.

Точки упорядкованості

Згідно стандарту, точка упорядкованості - це будь-яке місце у виконанні програми, в якому гарантовано, що всі побічні ефекти попередніх обчислень вже виконані, а жоден з побічних ефектів наступних обчислень ще не відбувся.

У стандарті детально розписані всі можливі місця точок упорядкованості в програмі. Зокрема, точками упорядкованості є:

1. Кінець першого операнду таких операторів:
   а) логічне І && ;
   б) логічне АБО || ;
   в) умовний ? ;
   d) кома , .
2. Кінець повного виразу.

Приклад. Наступні три програми функціонально аналогічні програмі з невизначеною поведінкою з останнього прикладу. Проте програміст, заручившись підтримкою точок упорядкованості, написав їх так, щоб уникнути неоднозначності виконання.

Перша програма.

Оскільки в операторі && є точка упорядкованості, а саме - після першого операнда, то гарантується, що функція f1, а не f2, викликається і виконується першою.

```
#include <stdio.h>
int f1()
{ printf ("KPI"); return 1;}
int f2()
{ printf ("inKYIV"); return 1;}

 int main()
{
       int p = f1() && f2() ;    // Точки упорядкованості: після першого операнда;
                                 //        після точки з комою – кінця повного виразу.
       return 0;
}
```

Друга програма.

Оскільки оператор кома є точкою упорядкованості, то гарантується, що функція f1, а не f2, викликається і виконується першою.

```
#include <stdio.h>
int f1()
{ printf ("KPI"); return 1;}
int f2()
{ printf ("inKYIV"); return 1;}

 int main()
{
        p = (f1(), f2());
       return 0;
}
```

Третя програма.

Оскільки оператор ? є точкою упорядкованості, то гарантується, що функція f1, а не f2, викликається і виконується першою.

```
#include <stdio.h>
int f1()
{ printf ("KPI"); return 1;}
int f2()
{ printf ("inKYIV"); return 1;}
```

```
int main()
{
	int p = f1()? f2(): 3;
	return 0;
}
```

Література: [2], стор. 3-6, 10-16.

КОНТРОЛЬНІ ПИТАННЯ

1. Коротко охарактеризуйте три види можливої поведінки С-програм.
2. Що таке побічні ефекти? Наведіть приклад.
3. Що таке точки упорядкованості? Наведіть приклад.

Лекція 13. Концепції мови C (частина 2)

Концепція меж дії ідентифікаторів

Згідно стандарту, ідентифікатор призначений для того, щоб програміст міг позначати сутності, задіяні в алгоритмі створюваної програми. В табл. 13.1 показано, що саме програміст може позначати ідентифікатором.

Таблиця 13.1 – Призначення ідентифікаторів в мові C

Ідентифікатор позначає:	Приклад (відповідні ідентифікатори виділено)
об'єкт	int **slot**, **found** = 0; struct hash ***htable**; union Data { int i; char str[20]; } **data**; enum week {Mon, Tue, Wed} **day**;
функцію	int **seach_hashtable**(struct hash **htable, int val, int size)
тег структури	struct **hash** *htable; struct **hashnode** { int val; struct hashnode *next; };
тег об'єднання	union **Data** { float f; char str[20]; } data;
тег переліку	enum **week** {Mon, Tue, Wed} day;
елемент структури	struct hashnode { int **val**; struct hashnode * **next**; };
елемент об'єднання	union Data { float **nums**; char **str**[4]; } ;
елемент переліку	enum week {**Mon**, **Tue**, **Wed**} day;
назву за typedef	typedef unsigned char **BYTES**;
назву мітки	**fini**: printf("passing from the goto statement\n");
ім'я макросу, параметр макросу	#define **minimum**(**a**, **b**) ((a) < (b) ? (a) : (b))

Ідентифікатор є видимим (тобто може бути використаний) тільки в певній частині тексту С-програми. Ця частина називається межами дії (видимості) ідентифікатора.

Якщо різні сутності позначені одним і тим самим ідентифікатором, то вони або мають різні межі дії, або знаходяться в різних просторах імен.

Отже, один і той же ідентифікатор може позначати різні об'єкти в різних точках програми. Це забезпечується саме концепцією меж ідентифікаторів.

Існує чотири види меж дії:

- межі функції,
- межі файлу,
- межі блоку,
- межі прототипу функції.

Сучасні компілятори С підтримують сучасні стандарти, які дозволяють програмісту декларувати змінну в будь-якому місці. Межа дії змінної починається від точки декларації до кінця блоку (кінець блоку – наступна найближча закриваюча дужка).

Приклад. Межі дії змінної z починаються всередині блоку.

```
if ( x < 10 )
{
  printf("%d", 11);              // тут змінна z не є видимою
  int z = 42;                    // початок меж дії (видимості) z
  printf("%d", z);
}                                // кінець меж дії (видимості) z
```

Приклад. Декларація int i=0 всередині твердження ітерації для ініціалізації. Об'єкт і буде існувати тільки в межах циклу. Перевага - економія пам'яті.

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

Приклад. Межі дії x та y - різні блоки.

```
int main()
{  int x = 0 ;
   while (x<10)
        {  if (x>5)
                {
                        int y[x];
                        y[0] = x;
                        printf("%d %d\n",y[0],y[1]);
                }
                x++;
        }
}
```

Приклад. Розглядається програма, в якій одним і тим же ідентифікатором hih названі чотири різні сутності, і межі їх дії перетинаються. Дозволяється тільки одна форма перетину – вкладення, і тоді декларація, яка з'являється у внутрішніх межах, приховує декларацію, яка з'являється у зовнішніх межах.

```
// простір імен тут - звичайні ідентифікатори

int hih;              // тут починаються межі дії (до кінця даного файлу) ідентифікатора hih
```

```
void f(void)
{
   int hih = 1;          // тут починаються межі дії (до кінця даного блоку) ідентифікатора hih,
                                      попередній hih з файловими межами приховується.
   {
     int hih = 2;                  // тут починаються межі дії (до кінця даного вкладеного блоку)
                                                  внутрішнього hih, зовнішній hih приховується.
     printf("%d\n", hih);      // виведеться 2 - значення внутрішнього hih
   }                                  // тут закінчуються межі дії внутрішнього hih
   printf("%d\n", hih);                  // виведеться 1 - значення зовнішнього hih
}                              // тут закінчуються межі дії зовнішнього hih
 void g(int hih);      // тут ім'я hih має межі дії у прототипі функцій;
                                      інший hih (з межами дії до кінця файлу) приховується
```

Приклад. У програмі однакове ім'я n позначає і параметр функції, і об'єкт в твердженні ітерації. Межі дії першого n – функція, межі дії другого n – блок.

```
void f(int n)                              // починаються межі дії  параметру n
{                                                     // починається тіло функції
       ++n;                                          // тут n – параметр функції
       for(int n = 0; n<10; ++n) {       // починаються межі дії  внутрішнього імені n
       printf("%d\n", n);                   // виведення 0 1 2 3 4 5 6 7 8 9
       }                                         // закінчуються межі дії  внутрішнього імені n
                                                 // n знову позначає параметр функції
       printf("%d\n", n);                          // виведення значення n
}                                                // закінчуються межі дії  параметру n
int a = n;      // Помилка: n поза межами блоку
```

Приклад. У функції f межами дії мітки label є вся функція, включаючи всі вкладені блоки, не залежно від місця декларації мітки. Такі правила дійсні виключно для міток.

```
void f()
{
  {
        goto label;       // мітка label знаходиться у межах дії
        label:;
  }
  goto label;              // мітка label знаходиться у межах дії
}
void g()
{
   goto label;            // помилка: межами дії мітки label не є функція g()
}
```

Концепція з'єднання ідентифікаторів

Для того, щоб С-програміст міг повністю управляти ресурсами через їх ідентифікатори, концепція меж ідентифікатора доповнюється концепцією з'єднання ідентифікаторів (табл. 13.2).

Згідно стандарту, навіть якщо ідентифікатор декларовано з різними межами дії, або його декларовано кілька разів в одних межах, то процес з'єднання може зробити ідентифікатор таким, що відноситься до одного і того ж об'єкта (або функції).

Ідентифікатори класифікуються як:
- з'єднані зовнішньо,
- з'єднані внутрішньо,
- не з'єднані.

Таблиця 13.2 – З'єднання ідентифікаторів в мові С

Тип з'єднання	Опис
Зовнішнє з'єднання	Всі декларації ідентифікатора з зовнішнім з'єднанням позначають один і той же об'єкт або функцію в межах всієї програми, тобто у всіх одиницях трансляції та бібліотеках, що належать програмі. До ідентифікатора з зовнішнім з'єднанням можна звертатись з інших трансляційних одиниць. Ідентифікатор доступний лінкеру. Коли відбувається друга декларація того ж самого ідентифікатора з зовнішнім з'єднанням, лінкер пов'язує ідентифікатор з тим же об'єктом або функцією. Змінні та функції з зовнішніми з'єднаннями також мають мовне з'єднання, що дає можливість зв'язувати одиниці трансляції, написані на різних мовах програмування.
Внутрішнє з'єднання	Всередині одного файлу всі декларації одного ідентифікатора з внутрішнім зв'язком позначають один і той же об'єкт. До ідентифікатора з внутрішнім з'єднанням можна звертатись з усіх меж дії у поточній одиниці трансляції. Лінкер не має інформації про ідентифікатори з внутрішнім з'єднанням.
Ніякого з'єднання	Унікальна сутність, доступна тільки у межах дії свого ідентифікатора, не має жодного з'єднання. Ідентифікатор можна використовувати тільки в межах його дії. Якщо ідентифікатор не має з'єднання, то будь-яка подальша декларація з використанням цього ідентифікатора декларує щось нове, наприклад, нову змінну або новий тип.

У стандарті детально описано правила визначення типу з'єднання.

Зокрема, якщо декларація ідентифікатора об'єкта або функції з межами дії - файлом - містить специфікатор класу зберігання static, то цей ідентифікатор має внутрішнє з'єднання.

А от для ідентифікатора зі специфікатором класу зберігання extern, в межах, в яких є видимою попередня декларація цього ж ідентифікатора, діє такий порядок визначення типу з'єднання:

- якщо попередня декларація вказує внутрішнє або зовнішнє з'єднання, з'єднання ідентифікатора у наступній декларації буде таким, як зазначено у попередній декларації;

- якщо ж попередньої видимої декларації немає, або якщо для попередньої декларації з'єднання немає, то ідентифікатор має зовнішнє з'єднання.

Рис. 13.1 описує вид з'єднання, призначене об'єкту, який декларується двічі в одній одиниці трансляції. Колонка позначає першу декларацію, а рядок - повторну декларацію.

	static	No linkage	extern
static	Internal	Undefined	Internal
No linkage	Undefined	No linkage	External
extern	Undefined	Undefined	External

Рисунок 13.1 – З'єднання при повторному декларуванні

Приклад. У програмному фрагменті ідентифікатори i2 та i5 мають як зовнішнє, так і внутрішнє з'єднання. Подальше використання обох ідентифікаторів призводить до невизначеної поведінки.

```
int i1 = 10;          /* definition, external linkage */
static int i2 = 20;   /* definition, internal linkage */
extern int i3 = 30;   /* definition, external linkage */
int i4;               /* external linkage */
static int i5;        /* internal linkage */

int i1;               /* valid declaration */
int i2;               /* linkage disagreement with previous */
int i3;               /* valid  declaration */
int i4;               /* valid */
int i5;               /* linkage disagreement with previous */

int main(void) {
 /* ... */
 return 0;
}
```

Приклад. У фрагменті немає конфліктуючих декларацій.

```
int i1 = 10;               /* external linkage */
static int i2 = 20;        /*  internal linkage */
extern int i3 = 30;        /*  external linkage */
int i4;                    /*  external linkage */
static int i5;             /* internal linkage */

int main(void) {
 /* ... */
 return 0;
}
```

Приклад зовнішнього з'єднання. Текст програми міститься у двох файлах. Програміст хоче, щоб змінна Amount була доступною в обох файлах, тобто позначала один і той же об'єкт. Перший варіант тексту програми декларує ідентифікатор Amount у файлі **client.c**. На етапі компіляції цієї програми ми отримаємо помилку компілятора. Ця помилка компілятора виникає

тому, що коли компілятор при обробці файлу Calculation.c доходить до Amount, він бачить його як недекларований ідентифікатор.

Файл client.c	Файл calculation.c

#include <stdio.h>
int Amount = 0;
int main()
{
Addition();
printf("%d\n", Amount);
return 0; }
	
	void Addition()
	{
	int a = 0, b = 0;
	printf("Enter the value\n");
	scanf("%d%d",&a,&b);
	Amount = a + b; }

Отже, потрібно задекларувати Amount у Calculation.c; механізм з'єднання запускається за допомогою ключового слова extern. Після цього, якщо ми компілюємо код, він чудово компілюється. Компілятор відмічає змінну Amount як "unresolved".

Коли обидва об'єктні файли передаються лінкеру, лінкер визначає значення посилань з поміткою "unresolved" з інших об'єктних файлів і формує правильний exe-код. Тепер при виконанні програми ідентифікатор Amount означатиме одне й те саме місце в пам'яті, тобто один і той же (зовнішній) об'єкт.

Файл client.c	Файл calculation.c

#include <stdio.h>
int Amount = 0;
int main()
{
Addition();
printf("%d\n", Amount);
return 0; }
	extern int Amount;
	void Addition()
	{
	int a = 0, b = 0;
	printf("Enter the value\n");
	scanf("%d%d",&a,&b);
	Amount = a + b; }

Ідентифікатори функцій мають зовнішнє з'єднання (за замовчуванням). Для обмеження доступності функцій програміст використовує специфікатор класу зберігання static.

Приклад різних з'єднань. У програмі тільки ідентифікатор функції callable має зовнішнє з'єднання і, отже, є видимим ззовні. Інші імена діють в межах до кінця файлу і мають внутрішнє з'єднання. Зокрема, ідентифікатор length позначає той самий об'єкт і в тілі функції callable, і в тілі функції fillup.

```
static length;
static void fillup(void);
int callable ()
{
    if (length ==0){
        fillup ();
    }
    return (buf [length--]);
}

static void fillup (void)
{
    while (length <100){
```

```
                buf [length++] = 0;
                                }
        }
```

Приклад внутрішнього з'єднання. Ключове слово static в декларації забезпечує видимість об'єкта з ідентифікатором Amount тільки в межах поточної трансляційної одиниці (файлу).

```
#include <stdio.h>
 static int Amount = 10;
 int Display(void)
{
   printf("%d ",Amount);
}

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

Приклад відсутнього з'єднання. В наведеній далі програмі один і той же ідентифікатор Amount позначає різні сутності, має різні межі дії і посилається на різні об'єкти.

```
#include <stdio.h>
int Display1(void)
{       int Amount = 20;
        printf("%d ", Amount);
}
int Display2(void)
{       int Amount = 30;
        printf("%d ", Amount);
}
int main()
{       int Amount = 10;
        Display1();
        Display2();
        return 0;
}
```

Результат виконання: 20 30.

Приклад різних з'єднань. У фрагменті коду декларовано ідентифікатори ppp, bbb, mmm, ddd. Деякі з них з'єднані, деякі – ні.

Для визначення типів з'єднання скористаємось правилами зі стандарту.

Розглянемо ідентифікатор ppp. Бачимо в тексті програми дві декларації цього імені. Перша - зі специфікатором static. Межі дії – до кінця файлу. Тип з'єднання – внутрішнє, тому що є специфікатор static. Друга декларація ppp - у функції func0() - також має внутрішнє з'єднання, оскільки попередня декларація цього ж імені має внутрішнє з'єднання. Обидві декларації відносяться до одного і того ж об'єкта.

Ідентифікатор bbb декларується в програмі тричі. Перша декларація ідентифікатора bbb означає межі дії – до кінця файла, а також зовнішнє з'єднання завдяки специфікатору extern. Друга декларація bbb у функції func0() не має з'єднання і посилається на унікальний об'єкт. Третя декларація bbb у функції func1() завдяки специфікатору extern також має зовнішнє з'єднання і відноситься до того ж (зовнішнього) об'єкту, що і перша декларація.

Перша декларація ідентифікатора mmm має зовнішнє з'єднання, так само, як і декларація mmm у func0(). Тобто, ці дві декларації одного ідентифікатора в одному файлі відносяться до одного і того ж об'єкта. А от третя декларація mmm у func1() не має з'єднання і таким чином посилається на унікальний об'єкт.

Нарешті, декларації ddd у функціях func0() і func1() відносяться до різних об'єктів, оскільки у першій декларації ідентифікатор не має з'єднання, а той же ідентифікатор у другій декларації має зовнішнє з'єднання.

```
static int ppp;
extern int bbb;
int mmm;
int func0() {
    extern int mmm;
    extern int ppp;
    static int ddd;
    int bbb;
    ...
}
int func1() {
    static int mmm;
    extern int ddd;
    extern int bbb;
    ...
}
```

Концепція простору імен

Якщо в якійсь точці програми видно більше однієї декларації конкретного ідентифікатора, то для однозначного визначення об'єкта, якого стосується цей ідентифікатор, транслятор використовує синтаксичний контекст.

В мові встановлено окремі простори імен для різних категорій ідентифікаторів:

- імена міток – це окремий простір;
- теги структур, об'єднань і переліків – це окремий простір;
- елементи структур або об'єднань – це окремий простір;
- всі інші ідентифікатори, які називаються звичайними ідентифікаторами (описуються у звичайних деклараторах) – це окремий простір.

Приклад. У програмному фрагменті всі ідентифікатори id вказують на різні об'єкти.

```
/* тег структури */
struct id {
        int id;         /* елемент структури */
        };
```

```
/* інший тег структури */
struct id2 {
        char id;        /* елемент структури */
        };
/* звичайний ідентифікатор */
id()
{
        id:             /* мітка */
}
```

Концепція тривалості зберігання об'єктів

Для того, щоб С-програміст міг повністю управляти ресурсами через їх ідентифікатори, концепції меж дії та з'єднання ідентифікаторів доповнюється концепцією тривалості зберігання ідентифікаторів (об'єктів).

Тривалість зберігання об'єкта визначає його "час життя" - частину виконання програми, під час якої гарантовано резервування пам'яті під цей об'єкт: об'єкт існує, має постійну адресу і зберігає своє останнє значення.

Програміст може не помітити, що якийсь операнд містить звертання до об'єкта (наприклад, змінної чи масиву), час життя якого уже закінчився в процесі виконання програми, і тим самим допускає помилку в управлінні ресурсами. Така ситуація призводить до невизначеної поведінки програми.

Існують чотири види тривалості зберігання (табл. 13.3):

- потокова (thread),
- статична (static),
- автоматична (automatic),
- призначена (allocated).

Таблиця 13.3 – Тривалість зберігання

Тип	Опис
Потокова	Об'єкт, ідентифікатор якого декларовано зі специфікатором класу зберігання _Thread_local, має тривалість зберігання потоку. Його час життя - це повне виконання потоку, для якого він був створений, а його значення ініціалізується при запуску потоку.
Статична	Об'єкт, ідентифікатор якого декларовано або із зовнішнім з'єднанням, або з внутрішнім з'єднанням, або зі специфікатором класу зберігання static або extern, має статичну тривалість зберігання. Час життя - це весь період виконання програми, тобто пам'ять для зберігання об'єкта виділяється, коли програма починається, і звільняється, коли програма закінчується.
Автоматична	Об'єкт, ідентифікатор якого декларовано без з'єднання і без специфікатора класу зберігання static або зі специфікатором auto (явно практично не вказується), має автоматичну тривалість зберігання. Пам'ять для об'єкта виділяється на початку блоку, в якому декларується ім'я об'єкта, і вивільняється при виході з блоку.
Призначена	Об'єкт має призначену тривалість зберігання, якщо пам'ять для нього виділяється і звільняється при виконанні програми за допомогою викликів відповідних функцій динамічного розподілу пам'яті.

Приклад. Програма виводить різні адреси для одного і того ж ідентифікатора A, який позначає два різних об'єкти з різними межами дії, різним з'єднанням, різною тривалістю зберігання.

```
#include <stdio.h>
#include <stdlib.h>
 int A;                                  /* статична тривалість зберігання об'єкта A */
 int main(void)
{
  printf("&A = %p\n", (void*)&A);
  int A = 1;                             /* автоматична тривалість зберігання, інший об'єкт A */
  printf("&A = %p\n", (void*)&A);
  int *ptr_1 = malloc(sizeof(int));   /* початок призначеної тривалості зберігання */
  printf("address of int in allocated memory = %p\n", (void*)ptr_1);
  free(ptr_1);                           /* кінець призначеної тривалості зберігання */

}
```

Можливий результат програми:

```
&A = 0x600ae4
&A = 0x7ffefb064f5c
address of int in allocated memory = 0x1f28c30
```

Література: [2], стор. 28-37.

КОНТРОЛЬНІ ПИТАННЯ

1. Розгляньте текст будь-якої С-програми і визначте межі дії всіх ідентифікаторів.
2. Розгляньте текст будь-якої С-програми і визначте з'єднання для всіх ідентифікаторів.
3. Розгляньте текст будь-якої С-програми і знайдіть простори імен для різних категорій ідентифікаторів.
4. Розгляньте текст будь-якої С-програми і визначте тривалість зберігання різних об'єктів.

Лекція 14. Декларації та визначення даних і функцій (частина 3)

Специфікатори класу зберігання

Клас зберігання визначає:
- чи об'єкт має внутрішнє, зовнішнє або відсутнє з'єднання;
- чи об'єкт повинен зберігатися в пам'яті або в регістрі;
- чи об'єкт отримує або не отримує початкове значення за замовчуванням;
- чи тривалість зберігання об'єкта підтримується протягом усього часу виконання програми або лише під час виконання блоку, де визначений об'єкт.

Специфікатор класу зберігання явно вказується програмістом для уточнення декларації об'єкта, функції чи параметрів, якщо його не влаштовує ситуація за замовчуванням. Якщо специфікатора класу зберігання в декларації немає, то клас призначається за замовчуванням.

Мова С має шість ключових слів, які згруповані як специфікатори класу зберігання: typedef, extern, static, _Thread_local, auto, register.

Термін клас зберігання пов'язаний з розглянутими в попередніх лекціях концепціями меж дії ідентифікаторів (етап компіляції), з'єднання ідентифікаторів (етап лінкера), тривалості зберігання (етап виконання програми) (табл. 14.1).

Таблиця 14.1 – Класи зберігання

Специфікатор класу зберігання	**Характеристика**
Специфікатор **auto** (специфікатор за замовчуванням)	Об'єкти (змінні) з цим специфікатором мають: межі дії – блок; тип з'єднання – ніякого; тривалість зберігання – автоматична, тобто пам'ять виділяється, коли виконання програми входить до блоку, що містить декларацію змінної, і звільняється при виході з блоку. За замовчуванням, така змінна має значення "сміття".
Специфікатор **register**	Вказує компілятору, що об'єкт повинен зберігатися в регістрі компьютера. Зазвичай задається програмістом для сильно використовуваних змінних, таких як індекс ітерації, в надії підвищити продуктивність програми за рахунок мінімізації часу доступу. Однак, компілятор не зобов'язаний виконувати цей запит. Через обмежений розмір і кількість регістрів на більшості систем, тільки декілька змінних можуть бути фактично розміщені в регістрах. Якщо компілятор не може виділити регістр, то об'єкт розглядається як такий, що має специфікатор класу зберігання auto. Об'єкти з цим специфікатором мають: межі дії – блок; тип з'єднання – ніякого; тривалість зберігання – автоматична.
Специфікатор **static**	Пам'ять для змінної з цим специфікатором виділяється під час компіляції і зберігається, доки програма працює. Змінна може мати зовнішнє, внутрішнє або ніякого з'єднання.

	Коли змінна декларується зовнішньо для будь-якої функції у файлі, вона має межі дії - файл, зовнішнє з'єднання та статичну тривалість зберігання. Якщо ви додасте ключове слово static до такої декларації, ви отримаєте змінну зі статичною тривалістю зберігання, межами дії - файлом та внутрішнім з'єднанням. Якщо ви оголошуєте змінну всередині функції та використовуєте ключове слово static, змінна має статичну тривалість зберігання, межі дії – блок та ніякого з'єднання. Якщо не ініціалізована, така змінна встановлюється в 0 за замовчуванням.
Специфікатор **extern**	Всі декларації функцій є extern за замовчуванням. Для об'єкта з цим специфікатором: межі дії – файл; тип з'єднання – зовнішнє або внутрішнє; тривалість зберігання – статична.
Специфікатор **_Thread_local**	Тривалість зберігання – потокова. Застосовується при паралельних обчисленнях.
Специфікатор **typedef**	Називається "специфікатором класу зберігання" лише для синтаксичної зручності.

Приклад. Програма розміщена у двох файлах parta.c, partb.c.

```
// parta.c --- різні класи зберігання

#include <stdio.h>
void report_count();
void accumulate(int k);
int count = 0;          // статична тривалість зберігання; межі дії – файл; зовнішнє з'єднання
int main(void)
{
   int value;            // межі дії – функція; автоматична тривалість зберігання
   register int i;       // тип з'єднання – ніякого; автоматична тривалість зберігання
   printf("Enter a positive integer (0 to quit): ");
   while (scanf("%d", &value) == 1 && value > 0)
   {
      ++count;
      for (i = value; i >= 0; i--)
           accumulate(i);
         printf("Enter a positive integer (0 to quit): ");
   }
   report_count();
   return 0;
}
void report_count()
{
   printf("Loop executed %d times\n", count);
}
```

```
// partb.c – друга частина програми

#include <stdio.h>
extern int count;                // зовнішнє з’єднання, статична тривалість зберігання
static int total = 0;            //  статична тривалість зберігання, межі дії – файл, внутрішнє з’єднання
void accumulate(int k);          // декларація функції

// визначення функції; параметр k має автоматичну тривалість зберігання
void accumulate(int k)
{
   static int subtotal = 0;  // статична тривалість зберігання, межі дії – блок;  ніякого з’єднання
   if (k <= 0)
   {
      printf("loop cycle: %d\n", count);
      printf("subtotal: %d; total: %d\n", subtotal, total);
      subtotal = 0;
   }
   else
   {
      subtotal += k;
      total += k;
   }
}
```

Результат:

```
Enter a positive integer (0 to quit): 5
loop cycle: 1
subtotal: 15; total: 15
Enter a positive integer (0 to quit): 10
loop cycle: 2
subtotal: 55; total: 70
Enter a positive integer (0 to quit): 2
loop cycle: 3
subtotal: 3; total: 73
Enter a positive integer (0 to quit): 0
Loop executed 3 times
```

Приклад. Наступна програма демонструє, як один ідентифікатор позначає різні об’єкти з різними тривалістю життя і межами дії. Специфікатор auto на практиці опускається.

```
#include <stdio.h>
int main( )
{
   auto int i = 1;
    {
     auto int i = 2;
       {
         auto int i = 3;
         printf ( "\n%d ", i);
       }
     printf ( "%d ", i);
```

```
    }
   printf( "%d\n", i);
 }
```

Результат:
3 2 1

Приклад. Наступний код програми визначає статичну змінну і у двох блоках всередині функції staticDemo (). Функція staticDemo () викликається двічі з головної функції. Під час другого виклику статичні змінні зберігають свої старі значення, вони не ініціалізуються знову у другому виклику staticDemo ().

```
#include <stdio.h>
 void staticDemo()
{
   static int i;
   {
     static int i = 1;
     printf("%d ", i);
     i++;
   }
   printf("%d\n", i);
   i++;
}
 int main()
{
   staticDemo();
   staticDemo();
}
```

Результат:
1 0
2 1

Приклад. У наступній С-програмі, якщо ви видалите extern int x, то отримаєте помилку "*Undeclared identifier x*", оскільки змінна x визначена пізніше, ніж вона була використана у виклику функції printf. У цьому прикладі специфікатор extern повідомляє компілятору, що змінна x вже була визначена в іншому місці.
Також, якщо змінити вираз extern int x; на вираз extern int x = 50; , то знову отримаєте помилку " Redefinition of x", тому що зі специфікатором extern змінна не може бути ініціалізована, якщо вона визначена в іншому місці. Якщо ж ні, то декларація extern стає визначенням.

```
#include <stdio.h>
 extern int x;
 int main()
{
  printf("x: %d\n", x);
}
 int x = 10;
```

Приклад. Змінна n має цілий тип. Також в багатопотоковому процесі створюється унікальний екземпляр цієї змінної для кожного потоку, який його використовує, і знищується, коли потік закінчується.

```
#include <threads.h>
__thread int n;
```

Приклад. Після рядків

```
typedef int MILES, KLICKSP();
typedef struct { double hi, lo; } range;
```

нижченаведені конструкції є валідними деклараціями:

```
MILES distance;
extern KLICKSP *metricp;
range x;
range z, *zp;
```

Декларатори

В декларації відправною точкою є декларатор. Це може бути:

- прямий декларатор (ідентифікатор, декларатор масиву (рис. 5.2), декларатор функції (рис. 5.3));
- декларатор вказівника (рис. 5.1).

Приклад. Бачимо написану програмістом декларацію. Вона містить два декларатори масивів: s[] та t[3]. Справа від деклараторів розташовані ініціалізатори: = "abc", = "bca". Зліва від деклараторів – специфікатор типу (char) елементів масивів.

```
char s[] = "abc", t[3] = "bca";
```

Приклад. Бачимо написану програмістом декларацію. Вона містить декларатор * sequence[5] масиву вказівників, а також специфікатор та кваліфікатор типу цього масиву, тобто його елементів.

```
volatile int * sequence[5];
```

Імена (назви) типів

У деяких контекстах необхідно повністю описати тип даних. У мові С цей опис називається назвою типу і синтаксично є декларацією, в якій опущено ідентифікатор.

Імена типів використовуються, наприклад, в таких ситуаціях: явне перетворення типу оператором cast, в операторі sizeof, у складеному літералі.

Приклад.

Конструкції, які розташовані в деклараціях:

(a) int

(b) int *
(c) int *[3]
(d) int (*)[3]
(e) int (*)[*]
(f) int *()
(g) int (*)(void)
(h) int (*const [])(unsigned int, ...)

називають, відповідно, типи:

(a) цілий тип int,
(b) вказівник на int,
(c) масив з трьох вказівників на int,
(d) вказівник на масив з трьох int,
(e) вказівник на масив змінної довжини,
(f) функція без специфікації параметрів, яка повертає вказівник на int,
(g) вказівник на функцію без параметрів, яка повертає int,
(h) масив неуточненої кількості константних вказівників на функції, кожна з яких повертає тип int і має один параметр типу unsigned int і неуточнену кількість інших параметрів.

Статичні судження

Приклад. Програмісту потрібно, щоб ще на етапі компіляції впевнитись, що розмір цілого типу менше за символьний тип. Якщо це не так, очікується заданий текст повідомлення this program requires that int is less than char.

```
1  #include <assert.h>
2  int main(void)
3  {
4      
5      // This will produce an error at compile time.
6      static_assert(sizeof(int) < sizeof(char),
7                    "this program requires that int is less than char");
8  }
```

Compiler messages:

```
n file included from main.cpp:1:
ain.cpp: In function 'main':
ain.cpp:6:5: error: static assertion failed: "this program requires that int is less than char"
```

Зовнішні визначення

Після обробки препроцесором текст C-програми фактично складається з послідовності зовнішніх декларацій. Вони називаються зовнішніми, оскільки розташовані поза будь-якою функцією (і, отже, межі їх дії - файл).

Зовнішнє визначення - це зовнішня декларація, яка також є визначенням об'єкта або функції (тобто призводить до резервування пам'яті).

Декларація ідентифікатора для об'єкта, яка діє у межах файлу, без ініціалізатора, без специфікатора класу зберігання або зі специфікатором static, - це так зване орієнтовне визначення (a tentative definition).

В усій програмі має бути тільки одне зовнішнє визначення ідентифікатора.

Очевидно, специфікатори класу зберігання auto та register не можуть з'явитись серед специфікаторів у зовнішній декларації.

За замовчуванням ідентифікатор функції має зовнішнє з'єднання, тобто функція доступна в усій програмі. Щоб зробити функцію видимою тільки в межах одного файлу, треба вказати специфікатор static.

Приклад (зі стандарту). У програмному фрагменті наведено визначення функції max. extern (специфікатор за замовчуванням) – це специфікатор класу зберігання, int – специфікатор типу значення, що повертається, max(int a, int b) – це декларатор функції, блок { return a > b ? a: b; } – це тіло функції.

```
extern int max(int a, int b)
{
return a > b ? a: b;
}
```

Приклад (зі стандарту).

```
int i1 = 1;             // визначення, зовнішнє з'єднання
static int i2 = 2;      // визначення, внутрішнє з'єднання
extern int i3 = 3;      // визначення, зовнішнє з'єднання
int i4;                 // орієнтовне визначення, зовнішнє з'єднання
static int i5;          // орієнтовне визначення, внутрішнє з'єднання
int i1;                 // орієнтовне визначення, відноситься до попереднього
int i2;                 // невизначеність по з'єднанню
int i3;                 // орієнтовне визначення, відноситься до попереднього
int i4;                 //  орієнтовне визначення, відноситься до попереднього
int i5;                 // невизначеність по з'єднанню
extern int i1;          // відноситься до попереднього, з'єднання якого є зовнішнім
extern int i2;          // відноситься до попереднього, з'єднання якого є внутрішнім
extern int i3;          // відноситься до попереднього, з'єднання якого є зовнішнім
extern int i4;          // відноситься до попереднього, з'єднання якого є зовнішнім
extern int i5;          // відноситься до попереднього, з'єднання якого є внутрішнім
```

Приклад (зі стандарту). Якщо в кінці трансляційного блоку міститься int i[], а масив i все ще має неповний тип, тобто не має конкретної довжини, тоді неявний ініціалізатор змусить його мати один елемент, який обнуляється при подальшому запуску програми.

```
int i[];
```

Література: [2], стор. 78-105.

КОНТРОЛЬНІ ПИТАННЯ

1. Які класи зберігання об'єктів існують в мові C?
2. Наведіть приклади назв типів та опишіть їх. Наприклад, назва типу int означає цілий тип int.
3. Що таке зовнішнє визначення та зовнішня декларація?

Лекція 15. Директиви препроцесора (частина 2)

Управління рядком

Компілятор використовує номер рядка та ім'я файлу для позначення помилок, які він знаходить під час компіляції.

Директива #line дозволяє змінити номер поточного рядка та ім'я файлу, який компілюється.

Якщо змінити номер рядка і ім'я файлу, компілятор ігнорує попередні значення і продовжує обробку з новими значеннями.

Директиву #line зазвичай використовують при відлагодженні програми, щоб повідомлення про помилки стосувались початкового С-файлу, а не згенерованої програми.

Приклад.

```
#include <stdio.h>                       /*line 1*/
                                         /*line 2*/
int main(){                              /*line 3*/
                                         /*line 4*/
    printf("Hello\n");                   /*line 5*/
                                         /*line 6*/
    printf("Line: %d\n",__LINE__);       /*line 7*/
    #line 16                             /*перенумерація рядка*/
    printf("Line: %d\n",__LINE__);       /*line 16*/
    printf("Thanks \n");                 /*line 17*/
    return 0;                            /*line 18*/
}                                        /*line 19*/
```

Результат:

```
Hello
Line: 7
Line: 16
Thanks
```

Приклад. У наступному фрагменті номер рядка не змінюється, але ім'я компільованого файлу змінюється на new_file_name.abc. При цьому фактичне (фізичне) ім'я файлу залишається незмінним, проте компілятор вважає, що компілює файл з новим ім’ям (і яке він поверне за допомогою макросу __FILE__).

```
#line __LINE__ "new_file_name.abc"
```

Директива помилки

Директива #error - дуже корисна і часто недовикористовувана директива препроцесора.

Призначення: директива #error припиняє компіляцію і виводить текст, вказаний в директиві.

Компіляцію доцільно припинити у таких випадках:
- код є неповним;
- код вимагає особливих версій бібліотеки;
- код використовує інструменти, залежні від компілятора;
- код має специфічні вимоги до компілятора.

Приклад. Програма знаходить ще не реалізовані програмістом функції на етапі компіляції. Компіляцію буде припинено для будь-якого компілятора.

```
int my_function( void )
{
 #error my_function not implemented
 return 0;
}
```

Насправді може бути кориснішим дозволити компіляцію під час розробки та відлагодження програми, але переривати компіляцію для версії готового програмного продукту, щоб упевнитись у відсутності "порожніх" функцій.

Приклад. Нехай макрос DEBUG десь визначено (для етапу відлагодження програми). Під час розробки ми можемо компілювати код, але коли ми робимо випускову версію (в якій DEBUG уже відсутній), то відловимо нереалізовані функції з пустим тілом.

```
int my_function( void )
{
#ifndef DEBUG
 #error my_function not implemented
#endif
 return 0;
}
```

Іноді С-код залежить від конкретних версій бібліотеки. Корисно зупинити компіляцію, якщо включена непідходяща версія бібліотеки.

Приклад. Фрагмент містить перевірку версії бібліотеки.

```
#if library_version < 2
#error requires library_version 2 or better
#endif
```

Директива Pragma

Директива #pragma використовується для інструктування компілятора використовувати ті чи інші функції, наприклад, які залежать від реалізації компілятора. Директива не виконує жодних дій, а лише змінює поведінку компілятора.

Приклад. Директива #pragma pack(n) вирівнює елементи структури на границю мінімального з двох чисел –числа n (в байтах) та стандартної границі (1, 2, 4 або 8).

У програмному фрагменті показано стандартне вирівнювання та розміщення об’єкта типу структура.

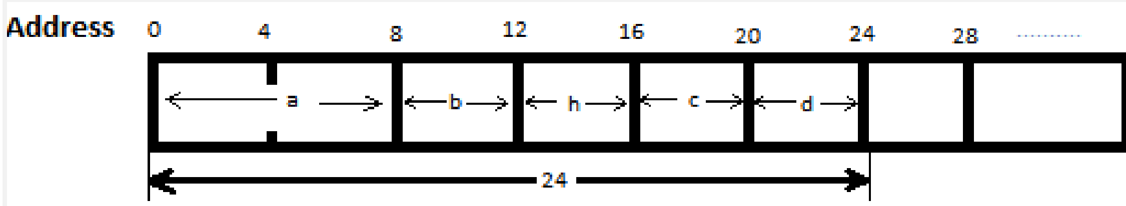
```
#include <iostream>
struct school
{       double a;
        int b;
        char h;
        int c;
        char d;
};

int main()
{       struct school students;
        printf("size of struct %d \n",sizeof(students));
        return 0;
}
```

Результат виконання та розміри виділеної пам’яті наведено нижче.

```
size of struct 24

--------------------------------
Process exited after 0.01123 seconds with return value 0
Press any key to continue . . .
```



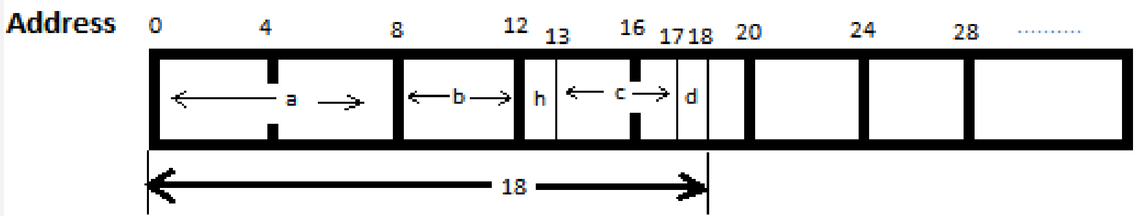
У наступному фрагменті директива # pragma pack (1) призводить до зміни розмірів пам'яті під об’єкт типу структура.

```
#include <iostream>
# pragma pack (1)
struct school
{       double a;
        int b;
        char h;
        int c;
        char d;
};
int main()
{       struct school students;
        printf("size of struct %d \n",sizeof(students));
        return 0;
}
```

Результат виконання та розміри виділеної пам’яті наведено нижче.

```
size of struct 18

--------------------------------
Process exited after 0.008896 seconds with return value 0
Press any key to continue . . .
```



Пуста директива

Директива препроцесора у формі # не має ніякого ефекту.
Існує практика використання цієї директиви для покращення читабельності початкового коду. Підтримується стандартом С.

Приклад.

```
#
#include "myfile.h"
#
```

Зарезервовані імена макросів

Наперед визначений макрос - це макрос, який є зрозумілим для препроцесора С і його не потрібно визначати в програмі.

Згідно стандарту, обов'язковими є такі зарезервовані імена макросів:

`__FILE__` - буде підставлено ім'я поточного початкового файлу у формі рядка-літерала;

`__LINE__` - буде підставлено номер поточного рядка у формі цілочисельної константи;

`__DATE__` - буде підставлено дату компіляції у формі рядка-літерала;

`__TIME__` - буде підставлено час компіляції у формі рядка-літерала.

`__FILE__` , `__LINE__` , `__func__` особливо корисні для цілей відлагодження програми.

Приклад. У стандартний файл stderr можна занести інформацію про місце потенційної помилки у виконуваній програмі, наприклад, коли знаменник отримає нульове значення. Тепер програмісту буде легше віднайти місце в тексті програми, де виникла помилка.

```
fprintf(stderr, "%s: %s: %d: Denominator 0", __FILE__, __func__, __LINE__);
```

Оператор Pragma

Оператор _Pragma є альтернативним методом визначення директиви #pragma.

Приклад. Наступні записи еквівалентні.

```
#pragma comment(copyright, "FAM2019")
_Pragma("comment(copyright, \" FAM2019\")")
```

Література: [2], стор. 126-129.

КОНТРОЛЬНІ ПИТАННЯ

1. Для чого корисна директива #error?
2. В чому полягає призначення директиви #pragma?
3. Коли використовується пуста директива?
4. Назвіть деякі зарезервовані імена макросів та їх призначення.

Лекція 16. Лексичні елементи (частина 3). Концепції мови С (частина 3)

Лексичні елементи. Літерали – рядки

Літерал-рядок - це послідовність, яка складається з нуля або більшої кількості мультибайтових символів, взята у подвійні лапки. Широкий рядковий літерал відрізняється префіксом **L**, **u** або **U**.

Приклад. У наступному виразі міститься символьний рядок (у лапках).

```
char *cst = "This is a literal";
```

Лексичні елементи. Пунктуатори

Пунктуатори **-** символи, що мають незалежну синтаксичну та смислову значимість. Пунктуатор – це кожен з наступних символів:

```
[    ]    (    )    {    }    .    ->
++   --   &    *    +    -    ~    !
/    %    <<   >>   <    >    <=   >=   ==
!=   ^    |    &&   ||
?    :    ;    ...
=    *=   /=   %=   +=   -=   <<=  >>=
&=   ^=   |=   ,    #    ##
<:   :>   <%   %>   %:   %:%:
```

Шість токенів - диграфів <: :> <% %> %: %:%: ведуть себе, як і шість токенів [ ] { } # ## відповідно.

Лексичні елементи. Імена заголовків

Якщо подивитись код будь-якої С-програми, то у перших рядках тексту можна побачити імена, обмежені дужками < та > або " та ". Це так звані імена заголовків. Вони є іменами заголовних файлів, які зазвичай містять визначення типів даних, прототипи функцій та команди препроцесора С.

Приклад. У тексті програми бачимо три імені стандартних заголовків <stdio.h>, <stdlib.h>, <string.h>, один користувацький заголовок "neuron.h", які позначають відповідні заголовні файли.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "neuron.h"
int main(int argc, char *argv[]) {…}
```

Лексичні елементи. Препроцесингові числа

Так зване препроцесингове число (пп-число) має досить дивне визначення. Воно включає в себе всі звичайні цілі константи і константи з плаваючою точкою. Проте є й інші ситуації, в яких спочатку навіть можна не побачити число.

Препроцесингове число, як лексична одиниця, розпізнається на етапі роботи препроцесора і не має типу або значення. Потім на певному етапі трансляції у нього можуть з’явитись і тип, і значення, як у звичайної константи.

Формально препроцесингове число починається з десяткової цифри, перед якою може стояти точка, а після - будь-яка послідовність букв, цифр, підкреслення, точок і двосимвольних експонент e +, e-, E+, E- p+, p-, P+, P-. Експоненти, які починаються з "p" або "P", використовуються для шістнадцяткових констант з плаваючою точкою.

По суті, пп-число є послідовністю символів.

Метою цього незвичайного визначення є полегшення роботи з різноманітністю числових констант. Наприклад, препроцесору дуже складно розрізняти лексично допустимі та недопустимі числа з плаваючою точкою.

Визначення також дозволяє розділити ідентифікатор в будь-якій позиції і отримати точно два токени, які потім можна склеїти оператором ##.

Препроцесингове число також може бути частиною тексту, яка пропускається чи залишається умовною директивою.

Приклад. Валідні препроцесингові числа:

```
.321        5E          0
765         321E0F      3e+xy
111LU       0.123E-005  1for1
```

Приклад. 23 – препроцесингова константа.

```
#define  AGE  23
```

Приклад. У макровизначенні R препроцесингове число 2e не є валідним числом. Однак оператор препроцессора ## "склеїть" його з препроцесинговим числом 3 в місцях виклику R, в результаті чого з'являється препроцесингове число 2e3, яке є валідним числом.

```
#define  R  2e ## 3
```

Лексичні елементи. Коментарі

В мові С прграміст може додавати коментарі в текст програми двома способами:

- за допомогою обмежувачів /* та */ початку та кінця коментарів відповідно;

- за допомогою обмежувачів // та символу нового рядка.

Вміст коментаря розглядається транслятором лише для виокремлення багатобайтових символів і для знаходження символу завершення коментаря.

Приклад зі стандарту. Наявність бекслешу \ в кінці рядка коментаря вказує транслятору, що дію наступного символу, а це є символ нового рядка, слід ігнорувати, так що наступний рядок обробляється так, ніби він є частиною поточного рядка. Отже, пара коментарів розташована програмістом на двох рядках.

```
"a//b"                    - чотирисимвольний рядок-літерал
#include "//e"            - невизначена поведінка
// */                     - коментар,  не синтаксична помилка
f = g/**//h;              - еквівалентно  f = g / h;
//\                       - перший рядок дворядкового коментаря
i();                      - другий рядок дворядкового коментаря
/\                        - перший рядок дворядкового коментаря
/ j();                    - другий рядок дворядкового коментаря
#define glue(x,y) x##y
glue(/,/) k();            - синтаксична помилка, не коментар
/*//*/ l();               - еквівалентно  l();
m = n//**/o
+ p;                      - еквівалентно  m = n + p;
```

Концепція представлення типів

Машинне представлення різних типів не специфікується стандартом C, за винятком кількох випадків.

Концепція сумісного типу та складеного типу

У стандарті C детально розглядається питання про так звані сумісні типи та складені типи. Тут ми зробимо загальний огляд цих важливих концепцій.

Поняття сумісного типу і складеного типу були введені, щоб дозволити програмісту ситуації, в яких декларації типу не повинні бути ідентичними.

Зокрема, у C-програмі декларації, що посилаються на один і той самий об'єкт або функцію в різних трансляційних одиницях, не повинні використовувати один і той же тип. Вони повинні лише використовувати досить схожі типи, формально відомі як сумісні типи. Сумісність типів дуже важлива в процесі перетворення типів і виконанні операцій програмою.

З дозволу для програміста використовувати сумісні типи виникає задача для розробників C-компілятора звести ці сумісні типи до одного, так званого складеного типу, щоб потім програма могла успішно виконатись. Складений тип формується компілятором з сумісних типів і є сумісним з кожним з цих типів.

Отже, складені типи формуються не програмістом. Програміст має тільки знати правила сумісності типів, викладені в стандарті.

Приклад.

```
// Перша трансляційна одиниця (скорочено ТО1)

#include <stdio.h>
struct s {int i;};              // сумісний з типом s в ТО3, але не в ТО2
extern struct s x = {0};        // сумісний з типом x в ТО3
```

```
extern void f(void);            // сумісний з типом f в ТО2

int main()
{
  f();
  return x.i;
}

// Друга трансляційна одиниця (скорочено ТО2)

struct s {float f;};            // сумісний з типом s в ТО4, але не в ТО1
extern struct s y = {3.14};     // сумісний з типом y в ТО4

void f()                        // сумісний з типом f в ТО1
{
  return;
}

// Третя трансляційна одиниця (скорочено ТО3)

struct s {int i;};              // сумісний з типом s в ТО1, але не в ТО2
extern struct s x;              // сумісний з типом x в ТО1

// Четверта трансляційна одиниця (скорочено ТО4)

struct s {float f;};            // сумісний з типом s в ТО2, але не в ТО1
extern struct s y;              // сумісний з типом y в ТО2
```

Приклад. У програмному фрагменті є дві декларації функції f, які трохи відрізняються, але є сумісними. Виходячи з правил, представлених у стандарті, компілятор визначає єдиний складений тип, який працює для обох декларацій, і буде використаний як на етапі трансляції, так і на етапі роботи С-програми.

```
// Дві декларації функції в однакових межах дії – в межах файлу

int f(int (*)(), double (*)[3]);
int f(int (*)(char *), double (*)[]);
```

Після обробки компілятором отримано складений тип:

```
int f(int (*)(char *), double (*)[3]);
```

Концепція вирівнювання об'єктів

Вирівнювання - це визначене реалізацією ціле число, яке дорівнює кількості байтів між послідовними адресами оперативної пам'яті для даних програми.

Іншими словами, вирівнювання обмежує адреси пам’яті, які можна віддати для об’єктів програми.

Література: [2], стор. 50-54.

КОНТРОЛЬНІ ПИТАННЯ

1. Навіщо потрібні коментарі в програмі?
2. Що таке препроцесингові числа в мові С, і яке їх призначення?
3. Розкрийте концепцію сумісного та складеного типів.
4. Що означає вирівнювання пам’яті?

Додаток 1. Словник стандартних термінів

Переклад деяких термінів відрізняється від запропонованих в інших джерелах, які стосуються процедурного програмування.

Наприклад, запропоновано переклад слова *statement* як *твердження*, а не як *оператор*, *команда*, *операція*, *інструкція*. По-перше, для останніх 4-х термінів є однозначні загальноприйняті англомовні варіанти. По-друге, запропонований переклад вміщує семантику інших перекладів.

Подані в табл. 1 терміни є важливими, оскільки від їх освоєння залежить розуміння програмістом поведінки та ефективності його С-програми.

Таблиця 1 – Терміни стандарту мови С

Термін стандарту С	**Український переклад**	**Визначення терміну в контексті мови С**
behavior	поведінка	Поведінка програми – це зовнішній прояв або дія виконуваної програми.
compound statement (block)	складене твердження (блок)	Блок дозволяє згрупувати набір декларацій і операторів в одну синтаксичну одиницю.
declaration	декларація	Декларація в С-програмі описує атрибути та інтерпретацію ідентифікаторів.
declarator	декларатор	Ідентифікатор в декларації. Кожен декларатор декларує один ідентифікатор і стверджує, що коли операнд тієї ж форми, що й декларатор, з'являється у виразі, він позначає функцію або об'єкт з межами дії, тривалістю зберігання і типом, вказаними специфікаторами в декларації.
declare	декларувати	Написати С-декларацію.
define	визначати	Надати формальне визначення .
definition	визначення	Визначення ідентифікатора - це декларація для цього ідентифікатора, яка: - для об'єкта спричине виділення ресурсу пам’яті під цей об’єкт; - для функції містить тіло функції; - для константи переліку є єдиною декларацією ідентифікатора; - для назви в typedef є першою (або єдиною) декларацією ідентифікатора.

Термін стандарту С	**Український переклад**	**Визначення терміну в контексті мови С**
designation, designator	позначення	В мові С позначення має місце в деяких ситуаціях. Так, позначення функції - це вираз, який має тип функції. За винятком випадків, коли це операнд оператора sizeof або унарного оператора &, позначення з типом "функція, що повертає тип" перетворюється транслятором на вираз, який матиме тип "вказівник на функцію, що повертає тип ". Наприклад, в декларації int function_a(int a); ідентифікатор function_a є позначенням функції. У виклику функції (*pf[f1()]) (f2(), f3() + f4()); позначеннями функцій є f1, f2, f3, f4. Хоча виклик функції визначений тільки для вказівників на функції, він працює з позначеннями функцій завдяки неявному перетворенню типу функції на тип вказівника на функцію.
enumeration	перелік	Тип даних в мові С.
escape sequence	оминаюча послідовність	Послідовність символів, в якій втрачається (оминається) зміст окремо взятого символа, проте набуває певного значення саме їх група, наприклад, для кодування китайського ієрогліфа. Unicode characters 抵抗是徒劳的 (resistance is futile) ↓↑ \u62b5\u6297\u662f\u5f92\u52b3\u7684 Unicode escape sequences
explicit	явний	Наприклад, явне перетворення типів даних.
expression	вираз	Вираз являє собою послідовність операторів і операндів. Призначення виразу в програмі: - описує обчислення значення, - позначає об'єкт або функцію,

Термін стандарту С	Український переклад	Визначення терміну в контексті мови С
		- генерує побічні ефекти, - виконує комбінацію попередніх пунктів. Перед обчисленням результату оператора проводиться упорядкування обчислень значень операндів.
implementation	імплементація, реалізація	Певне програмне забезпечення, яке працює в певному трансляційному середовищі з конкретними параметрами управління, яке виконує трансляцію С-програм і підтримує виконання С-функцій в певному середовищі виконання.
implementation-defined behavior	поведінка, визначена реалізацією	Поведінка програми, яка залежатиме від конкретної реалізації середовища компіляції та виконання. При цьому програміст може знайти відповідну інформацію в документації по цій реалізації.
implicit	неявний	Наприклад, неявне перетворення типів у виразах, яке здійснюється автоматично без втручання програміста.
linkage	з’єднання	Визначає, чи можуть декларації в різних областях посилатися на одну й ту ж сутність.
locale-specific behavior	поведінка, обумовлена місцевими особливостями	Поведінка програми, яка залежить від умовностей національності, культури і мови, що описані в документах по реалізації.
name space	простір імен	Механізм просторів імен придумано для ситуації, коли більше однієї декларації певного ідентифікатора видно в будь-якій точці програми при її трансляції, і тоді виникає неоднозначність застосування одного й того ж ідентифікатора до різних сутностей. Різні категорії ідентифікаторів попадають у відповідні простори імен, а саме: - імена міток; - теги структур, об'єднань і переліків; - елементи структури або об'єднання; кожна структура або об'єднання

Термін стандарту C	**Український переклад**	**Визначення терміну в контексті мови C**
		має окремий простір імен для своїх елементів; - всі інші ідентифікатори, так звані звичайні ідентифікатори.
object	об’єкт	Об’єкт в мові C – це область зберігання даних в середовищі виконання, вміст якої може бути конкретним значенням. Якщо адресуватись до об’єкта, тоді об'єкт можна інтерпретувати як такий, що має певний тип. Отже, об’єкт ототожнюється зі шматком пам’яті.
operand	операнд	Операнд - це сутність, на яку діє оператор.
operator	оператор	Спеціальний символ, один з пунктуаторів, який повідомляє транслятору, яку потрібно виконати операцію над вказаними операндами.
preprocessing file	препроцесинговий файл	Те ж саме, що і початковий файл.
punctuator	пунктуатор, роздільник	Символ, що має незалежну синтаксичну та смислову значимість. Наприклад, він може вказувати операцію, яка повинна виконуватися, в цьому випадку він відомий як оператор.
qualifier	кваліфікатор, визначник	Кваліфікатор типу. Існує три кваліфікатора: const, volatile, restrict. Кожен тип даних може мати кілька кваліфікованих версій свого типу, що відповідають комбінаціям одного, двох або всіх трьох кваліфікаторів.
scope	зона, межі, границі	Межі дії ідентифікаторів. Частина трансляційної одиниці, в якій ідентифікатор є видимим, тобто, діє його декларація.
sequence point	точка упорядкованості	Будь-яке місце у виконанні програми, в якому гарантовано, що всі побічні ефекти попередніх обчислень вже виконані, а жоден з побічних ефектів наступних обчислень ще не відбувся.
side effect	побічний ефект	Побічні ефекти при виконанні C-програми – це зміни стану середовища виконання. По-простому, побічний ефект – це коли щось десь змінюється. Доступ до об'єкта з кваліфікатором volatile, модифікація об'єкта, модифікація файлу або виклик функції, яка виконує будь-яку з цих операцій, - це побічні ефекти. В загальному випадку, обчислення виразу

Термін стандарту C	**Український переклад**	**Визначення терміну в контексті мови C**
		включає як обчислення значення виразу, так і ініціювання побічних ефектів. Побічні ефекти часто ускладнюють розуміння поведінки програми.
source file	початковий файл	Файл, в якому міститься текст C-програми.
specifier	специфікатор, опис	Специфікатор – частина декларації. Він додатково описує з'єднання сутностей, тривалість зберігання сутностей та частину типу сутностей.
specify	описувати, специфікувати, уточнювати	Детально описувати, уточнювати те, що не регламентується або не уточнюється стандартом C.
statement	твердження	Конструкція мови C, що описує дію, яку слід виконати програмі.
storage duration	тривалість зберігання	Час життя об'єкта - це період часу при виконанні програми, в якому об'єкт має для себе валідну пам'ять.
tag	тег	Тег є ідентифікатором, який використовується для даних зі специфікаторами структури, об'єднання або переліку (struct, union, enum).
token	токен, лексема	Лексична одиниця мови.
translation unit	одиниця трансляції	Код, створений препроцесором з початкового файлу і заголовних файлів.
type promotion	підвищення типу	Особливий випадок неявного перетворення типу даних, коли компілятор автоматично розширює двійкове подання цілочисельних об'єктів або об'єктів з плаваючою точкою. На відміну від деяких інших перетворень типів, при цьому перетворенні ніколи не втрачається точність і не змінюються значення перетворюваних об'єктів.
undefined behavior	невизначена поведінка	Поведінка програми при використанні програмістом помилкових конструкцій або помилкових даних, до яких стандарт не пред'являє жодних вимог.

Термін стандарту C	**Український переклад**	**Визначення терміну в контексті мови C**
unqualified	не обмежений умовами	Тип без кваліфікатора типу.
unspecified behavior	не специфікована, не уточнена поведінка	Поведінка програми при наявності не специфікованого значення; або поведінка, для якої стандарт надає кілька можливостей, але не встановлює вимоги щодо їх документування.
wide character	широкий символ	Тип символьних даних, який зазвичай має розмір, більший за традиційний 8-бітовий символ типу `char`.

Додаток 2. С-бібліотека

Кожна функція бібліотеки програм на мові С декларується у відповідному заголовному файлі.

Вміст заголовків додається у програму директивою препроцесора #include.

Далі наведено заголовки, а також функції управління пам’яттю, які містяться у файлі <stdlib.h> загальних утиліт, у відповідності до стандарту С.

Diagnostics. Діагностика.	**<assert.h>**
Complex arithmetic. Комплексна арифметика.	**<complex.h>**
Character handling. Робота з символами.	**<ctype.h>**
Errors. Помилки.	**<errno.h>**
Floating-point environment. Середовище з плаваючою точкою.	**<fenv.h>**
Characteristics of floating types. Характеристики плаваючих типів.	**<float.h>**
Format conversion of integer types. Перетворення формату цілих типів.	**<inttypes.h>**
Alternative spellings. Альтернативні написання.	**<iso646.h>**
Sizes of integer types. Розміри цілих типів.	**<limits.h>**
Localization. Локалізація.	**<locale.h>**
Mathematics. Математика.	**<math.h>**
Nonlocal jumps. Нелокальні переходи.	**<setjmp.h>**
Signal handling. Управління сигналами.	**<signal.h>**
Alignment. Вирівнювання.	**<stdalign.h>**
Variable arguments. Змінні аргументи.	**<stdarg.h>**
Atomics. Атоми.	**<stdatomic.h>**
Boolean type and values. Булеви типи.	**<stdbool.h>**
Common definitions. Загальні визначення.	**<stddef.h>**
Integer types. Цілі типи.	**<stdint.h>**
Input/output. Введення/виведення.	**<stdio.h>**

General utilities. Загальні утиліти. **<stdlib.h>**

Numeric conversion functions. Функції числових перетворень.

Pseudo-random sequence generation functions. Функції генерації псевдовипадкової послідовності.

Memory management functions**. Функції управління пам'яттю.**

7.22.3.2 The calloc function

Synopsis

1 **#include** <stdlib.h>
void ***calloc**(**size_t** nmemb, **size_t** size);

Description

2 The **calloc** function allocates space for an array of nmemb objects, each of whose size is size. The space is initialized to all bits zero.

Returns

3 The **calloc** function returns either a null pointer or a pointer to the allocated space.

7.22.3.3 The free function

Synopsis

1 **#include** <stdlib.h>
void free(**void** *ptr);

Description

2 The **free** function causes the space pointed to by ptr to be deallocated, that is, made available for further allocation. If ptr is a null pointer, no action occurs. Otherwise, if the argument does not match a pointer earlier returned by a memory management function, or if the space has been deallocated by a call to **free** or **realloc**, the behavior is undefined.

Returns

3 The **free** function returns no value.

7.22.3.4 The malloc function

Synopsis

1 **#include** <stdlib.h>
void ***malloc**(**size_t** size);

Description

2 The **malloc** function allocates space for an object whose size is specified by size and whose value is indeterminate.

Returns

3 The **malloc** function returns either a null pointer or a pointer to the allocated space.

Communication with the environment. Спілкування з середовищем.
Searching and sorting utilities. Утиліти пошуку і сортування.
Integer arithmetic functions. Функції арифметики цілих чисел.
Multibyte/wide character conversion functions. Функції перетворення широких символів.
Multibyte/wide string conversion functions. Функції перетворення широких рядків.

_Noreturn.	**<stdnoreturn.h>**
String handling. Управління рядками.	**<string.h>**
Type-generic math. Математика з родовим типом.	**<tgmath.h>**
Threads. Потоки.	**<threads.h>**
Date and time. Дата та час.	**<time.h>**
Unicode utilities. Утиліти для Юнікода.	**<uchar.h>**
Extended multibyte and wide character utilities. Утиліти для розширених багатобайтових та широких символів.	**<wchar.h>**
Wide character classification and mapping utilities. Утиліти для класифікації та відображення широких символів.	**<wctype.h>**

Література

1. Язык программирования C / Б. Керниган, Д. Ритчи. – Москва:Вільямс, 2013. – 304 с.

2. International Standard ISO/IEC 9899:2018 – Information technology – Programming languages – C. URL: https://web.archive.org/web/20181230041359if_/http://www.open-std.org/jtc1/sc22/wg14/www/abq/c17_updated_proposed_fdis.pdf.

3. Collision Resolution in Hash Table (by linked list). URL: http://codingstreet.com/category/hash/.

4. Letterlike Symbols. URL: https://unicode.org/charts/PDF/U2100.pdf.