0. Вступ

Серед багатьох людей, зокрема, навіть серед досвідчених програмістів часто висловлюються дві думки: 1) мова С та мова C++ - це різні мови програмування та мають розглядатись саме як різні мови; 2) мова С є застарілою та не дуже актуальною.

Автор повністю незгоден з другим твердженням – задачі, що потребують використання саме мови С зустрічаються достатньо часто та попит на програмістів саме С все ще великий. Слід також зауважити, що від програмістів на C++ розуміння С вимагається майже завжди.

Лише частково він згоден і з другим твердженням. Так, різниця між двома мовами таки є, але слід зауважити, що крім того, що С є фундаментом на якому будувався і досі будується C++, але достатньо часто конструкції з чистого С потрапляють і в «чистий» C++. Крім того, незважаючи на присутні майже всюди рекомендації виділяти окремо С-код та C++-код, на практиці з різних причин доводиться часто «тягнути» Сі-код в програму, що намагається бути «чистим» C++-кодом. Та й слід завжди мати на увазі що стандарт C++ вимагає того, шо будь-яка програма на мові С повинна компілюватись та працювати так само як на компіляторі С так і на C++.

Отже, позиція, що С - то є частина C++, фактично одним із діалектів, і не найбільш рідко вживаних, мови C++ більш близька автору. Саме з таким поглядом на мову С та C++ і розроблявся даний підручник. Важливою причиною вивчення на початку саме С, а потім вже C++ є можливість тоді для студентів з чистою совістю додавання в резюме двох мов замість однієї.

Даний підручник є частиною результатів роботи його автору над курсом «Мова програмування C++» і є представленням тієї його частини, що стосується саме мови Сі, а не C++. Метою цієї частини, яка передує вивченню саме Сі++, є знайомство студентів з синтаксисом мови, а також практиці написання програм на ній. Оскільки, сама мова С розглядалась як початок знайомства з С-подібним синтаксисом програмування, основна увага в цій частині курсу приділялось саме найбільш уживаним конструкціям мови які застосовуються, зокрема, і в C++, тому деякі підтеми, що часто є частиною курсів по С, автором не розглядалися, або розглядалися не надто детально. Зокрема, це такі конструкції, як перерахування (enums), об’єднання (unions), такі теми, як створення рекурсивних структур тощо. З власного досвіду автора, ці теми зустрічаються в практичному програмуванні не так вже й часто, а матеріалу в інших підручниках та в Інтернеті з цих тем багато, і, як правило, їх засвоєння не є великою проблемою. З іншого боку, деякі теми, такі як збирання проекту, робота з рядками С (null-terminated strings) та динамічною пам’яттю, часто можуть бути описані з практичної точки зору недостатньо, тому автор постарався дати акцент саме на такі теми і старався висвітлити саме їх з одного боку не надто довго, але достатньо детально, щоб можна було грамотно використовувати висвітлені конструкції мови з урахуванням останніх змін у мові програмування Сі.

Автор старався дотримуватися офіційного викладацького стилю розповіді, але інколи старався розбавляти «канцеляріт» більш живою мовою.

Домовленості стилю

Для зручності розуміння в тексті прийняті наступні домовленості.

Звичайний текст має наступний стиль:

Оскільки мова С – це компілятор, то середовище програмування повинно перетворити цю програму (або пакет програм) на файл, що може виконуватися. Для цього вона має виконати наступні дії, які звуться разом побудовою програми (building).

Тестові означення та важливі формулювання:

Програма на С (С++) – *це сукупність текстових файлів*, які зазвичай звуться **заголовочними** та **вхідними** (header та source files), які містять **декларації** (declarations). **Декларації** — це визначення змінних та функцій.

Формат команд або вигляд конструкцій мови:

type variable_name = value; // англійською

чи

ім'я_типу ім'я_змінної =<значення> // українською

або

int fwrite(вказівник_на_масив, розмір_об'єкта, кількість_об'єктів, вказівник_на_файл);

Інформація яка є додатковою та необов'язковою для студента:

При додаванні цілих чисел може виходити переповнення. Проблема полягає в тому, що комп'ютер не видає попередження при їх появі: програма продовжить виконання з невірними даними. Більше того, поведінка при переповненні є визначеною стандартом та фіксованою лише для цілих без знаку (натуральних).

Приклади програмного коду на мові С/С++

```
#include <stdio.h>
int main(){
  printf("Hello, world!\n");
  getchar();  // або int c = getchar();
}
```

Результат роботи програми:

Output is: 2

Некоректні конструкції:

Якщо потрібно вказати в коді некоректну конструкцію, то це буде так:

```
//  printf("%lld\n", -92233720368547758O8); // ПОМИЛКА
```

або якщо вся програма є хибною практикою:

```
#include<stdio.h>
#include<math.h>

int main(){
  double x, y, z;
  scanf("%f %f",&x,&y);
  z = exp(x)*cos(y);
  printf("z=%lf",z);
}
```

Повідомлення компілятору:

Error 1 error C4996: 'scanf': This function or variable may be unsafe. Consider using scanf_s instead. To disable deprecation, use _CRT_SECURE_NO_WARNINGS. See online help for details.

Програмний код не на мові C/C++:

```
cmake_minimum_required (VERSION 2.6)

set (PROJECT hello_world)
project (${PROJECT})
set (HEADERS hello.h)
set (SOURCES hello.cpp main.cpp)
add_executable ($ {PROJECT} $ {HEADERS} $ {SOURCES})
```

Запис командного рядку:

cmake CMakeLists.txt

або

gcc hello1.c <опції, на зразок –o hello –l<бібліотека>>

1. Лінійні програми на С

Програма на С - що це таке з точки зору стандарту С.
Приклад програми на С. З чого вона складається.
Компіляція програми на С.
Структурні частини програми на С: токени, крапки з комою, коментарі, ключові слова, пробіли.
Введення/виведення. С++ введення/виведення. Форматоване введення. Форматоване виведення. Бібліотека stdio.h
Визначення змінної в С. Декларація та ініціалізація.
Локалізація. Бібліотека locale.h
Використання математичної бібліотеки math.h
Дійсні типи даних.
Використання float.h

Програма на С: що це таке з точки зору стандарту С

З метою трохи полякати читача, настроїти його на серйозний лад та познайомити власне зі стилем, на якому написана більша частина документації по мові С, викладемо в цій стилістиці опис того, чим є мова програмування С згідно офіційного стандарту мови. Докладною розшифровкою даних визначень власне буде зміст подальшого підручника.

Програма на С (С++) – це сукупність текстових файлів, які зазвичай звуться **заголовочними** та **вхідними** (header та source files), які містять декларації (declarations). **Декларації** — це визначення змінних та функцій. Файли, що складають програму компілюються для того, щоб утворити виконуваний файл, якщо С++ файли містять головну функцію (main function). Якщо її немає, то програма може бути перетворена в бібліотеку.

Програма на мові С (С++) побудована за допомогою **спеціальних символів** (на кшталт {,}, # і т.п.) та **ключових слів** (keywords). Інші слова можуть служити як ідентифікатори (identifiers). Крім того, програма може містити коментарі (comments), які ігноруються під час компіляції. Деякі символи можуть бути представлені в програмі за допомогою послідовностей символів (escape sequences).

Ідентифікатори (identifiers), на С можуть ідентифікувати сутності (об’єкти) (objects), функції (functions), структури (struct), об’єднаня (union), перерахування (enumeration) та їх члени, типи, що визначені користувачем (typedef names), мітки (labels) та макроси(macros).

Сутності представлені **деклараціями** (declarations), які представляють їх іменами (names) та визначають їх властивості. Декларації визначають всі властивості, що потрібні для використання сутності як визначення (definition). Програма має містити лише одне визначення будь-якої не підставляємої (non-inline) функції або змінної.

Визначення функцій складається з послідовності **тверджень** (statements), деякі з яких включають **вирази** (expressions), які визначають обчислення що повинна виконати програма.

Імена, що виникають в програмі, співставленні з визначеннями змінними, що зберігають ці імена за допомогою **області дії імен** (name lookup). Кожне ім'я діє лише в середині частини програми, що зветься **областю визначення** (scope). Деякі імена можуть мати **зв'язок** (linkage), що визначає посилання до визначених для цього зв'язку сутностей, коли вони посилаються на ті самі сутності що з'являються в різних областях визначення або **одиницях трансляції** (translation units).

Декларації та вирази створюють, знищують, отримують доступ та маніпулюють об'єктами. Кожен об'єкт, функція та вираз в С асоційовані з певним **типом** (type), який може бути базовим (fundamental), похідним типом (compound), або визначеним користувачем (user-defined), повним (complete) або неповним (incomplete), і так далі.

Визначені об'єкти (declared objects) або визначені посилання (declared references), які не є не статичними членами даних (non-static data members) звуться **змінними** (*variables)*.

Програма на С: як це виглядає практично

Приклад програми на С

```
/*
Optional, but recommended: Documentation section
Необов'язкова але рекомендована частина (краще її писати англійською)
i.e. comments, that describe the program, like:
 Заголовочні коментарі, як наприклад:
First C-style program - Перша програма:
Обчислення сінуса
*/
#include <stdio.h> // Link section  - заголовочний файл,
        //бібліотека стандартного вводу-виводу
#include <math.h> // Link section заголовочний файл,
        //бібліотека математичних функцій

//Definition Section and Global declaration section
//Тут можуть бути визначення макросів
//Тут можуть бути визначення глобальних змінних та констант
//Тут можуть бути визначення або декларація функцій

int main() // головна функція (main function): точка входу (entry point)
{
   // Визначення та/або ініціалізація локальних змінних
```

```
    float x; //визначаємо дійсну (одинарної точності) змінну 'x'
    scanf("%F",&x); // введення змінної 'x'
    double y=sin(x); /* Вираз (expression): виклик функції sin,
                обчислення виразу та
               ініціалізація дійсної змінної (подвійною точності) ‘y’
              */
     printf("Result y=%f\n",y); // виведення значення змінної y
}
```

Ще один приклад, що не підключає математичну бібліотеку, але використовує стандартне введення-виведення

```
/*
  Second C-style program  - Обчислення температури по Фаренгейту
*/
#include <stdio.h> // Бібліотека стандартного вводу-виводу
          //(без неї нема printf та scanf)
int main(){    //  точка входу
          // Визначення та/або ініціалізація локальних змінних
  float F, C;   //визначаємо відразу дві дійсні – змінні ‘F’ та ‘C’
   printf(“F=”); // виводимо підказку для користувача
   scanf("%f", &F); // введення змінної ‘F’
  C=(F-32)*5/9;   /*  Обчислення за формулою */
   printf("Celsius C=%e\n",C); /* виведення значення змінної ‘C’ в науковому форматі  */
 }
```

Ці файли потрібно зберігати з розширенням “.c”, для того щоб система та компілятор зрозуміли що це файл для компіляції С-компілятором. Для того, щоб цю програму можна було відкомпілювати компілятором c++ потрібно трохи модифікувати код та зберегти його з розширенням “.cpp”:

```
#include <cstdio> // Link section: , бібліотека стандартного
        // вводу-виводу С інтегрована як С++ бібліотека
#include <cmath> // заголовочний файл,
       //бібліотека математичних функцій з С в С++

int main() // головна функція (main function): точка входу (entry point)
{
   float x; //визначаємо дійсну (одинарної точності) змінну 'x'
   scanf("%F",&x); // введення змінної 'x'
   double y=sin(x); /* Вираз (expression): виклик функції sin,
```

```
            обчислення виразу та
          ініціалізація дійсної змінної (подвійною точності) ‘y’    */
    printf("Result y=%f\n",y); // виведення значення змінної y
}
```

Відмітимо, що мова С – чутлива до регістрів (кейс-сенсітів), тобто регістр символів має значення.

Щодо **коментарів** в коді. В сучасній мові С існує два типи коментарів, тобто ділянок коду, які не компілюються:

- ***Коментар ділянки коду*****,** що може бути як на частину рядка так і на декілька рядків:

```
/*  якийсь текст  тут    */
/*
текст,
знову текст
....
ще щось ....

*/
```

- ***Коментар до кінця рядку:***

```
// текст до кінця рядка ігнорується компілятором
```

Можна побачити також, що всі команди на С завершуються крапкою з комою, а блоки програми виділяються фігурними дужками.

На відміну, наприклад, від Python, немає різниці як розташовувати пробіли та табуляції в текстів програми. Але *вкрай бажано використовувати певний стиль програмування*, схожий до того що використовується в Python – одна команда на рядок, табуляція при виділенні програмних блоків, фігурні дужки на окремих рядках і так далі.

Отриманий файл програми тепер потрібно відкомпілювати та запустити.

Компіляція

Оскільки мова С – це компілятор, то середовище програмування повинно перетворити цю програму (або пакет програм) на файл, що може виконуватися. Для цього вона має виконати наступні дії, які звуться разом побудовою програми (building):

1. **Препроцесінг (Preprocessing):** файли C++ проекту оброблюються таким чином, що текст програми замінюється на зрозумілі компілятору символи, замість #include, #define та інших препроцесорних команд підставляється код відповідних файлів та команд, видаляються коментарі. Результатом

препроцесінгу стає "чистий" C++ файл – проміжне представлення (intermediate representation).

2. **Компіляція (Compilation):** компілятор перетворює проміжний файл (translation unit, intermediate compiled output file) у файл що готовий для компіляції асемблером даного компілятору на даній платформі. Отриманий результат може бути включений до статичних бібліотек.

3. **Збирання (Асемблювання) (Assembly):** (часто цей пункт вказують як частину процесу компіляції) – створює файл як набір асемблерних (машинних) інструкцій (object file) таким чином, що код стає таким, що можна переміщувати.

4. **Лінковка (Linking):** використовує об'єктні файли, що згенервані компілятором та генерує виконуваний файл або бібліотеку (статична чи динамічна лінковка).

Процес компіляції можна запустити в командному рядку або в інтегрованому програмному середовищі.

На Linux в командному рядочку:

Для компіляції С-файлу:

```
gcc hello1.c  <опції, на зразок –o hello –l<бібліотека>>
```

Зокрема для компіляції файлу з першого лістингу потрібно запустити:

```
gcc hello1.c -lm
```

Опція lm потрібна для того, щоб програма використовувала математичну бібліотеку math.h.

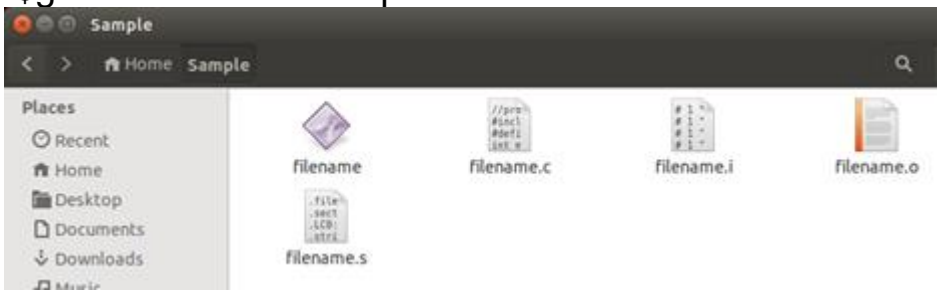
Для компіляції C++:

```
g++ hello1.cpp
```

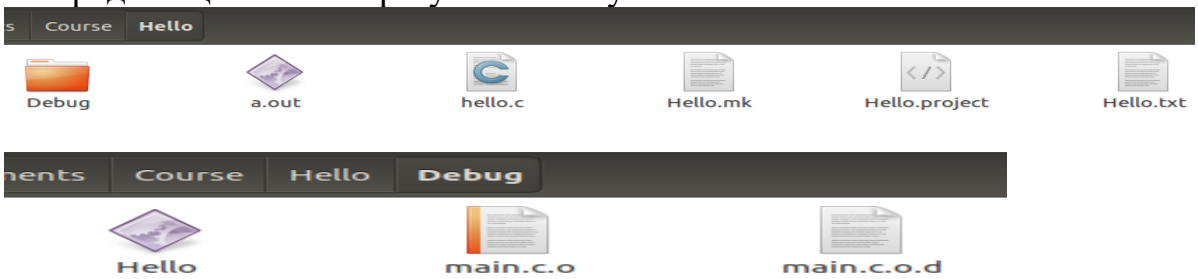
Результатом повинні бути виконуваний файл **hello1** та файл ***hello1.o*** – бібліотека (об'єктний файл)

Більш докладна команда, що показує проміжні етапи видасть результат

```
$gcc –Wall –save-temps filename.c –o filename
```

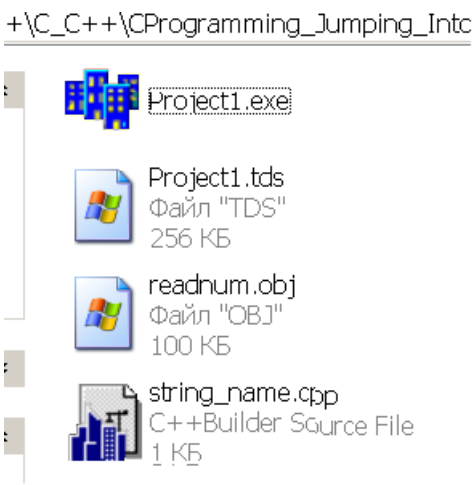


В середовищі CodeLite результат наступний

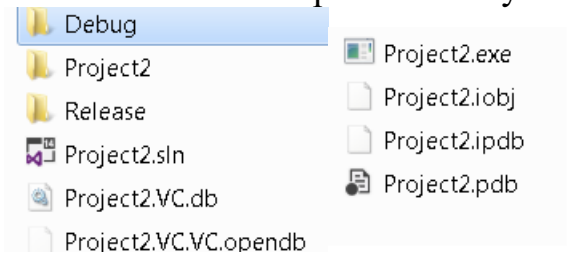


На Windows:

Borland Builder



A Visual Studio 15 зробить наступні папки та вміст ітогової папки Release:



Для запуску в командному рядку можна використати різні компілятори, зокрема:

- MS Visual Studio: cl hello1.c
- MinGW(Eclipse, CodeLite: gcc hello1.c)
- Turbo C: tc hello1.c
- Borland Builder C++: bc hello1.c

Іншим шляхом компіляції може бути створення проекту за допомогою середовища розробки, MS Visual Studio, Eclipse, CodeLite, Code:Blocks, Borland Builder і т.п. (в даному випадку потрібно створити консольний проект) та запустити відповідною кнопкою запуску. В цьому випадку середовище розробки само повинно згенерувати команду запуску та, можливо, створити автоматичний скріпт (більш докладно про це в наступних главах) для запуску програми.

Структурні частини програми на С

Токени С

На мові С програма складається з різних токенів, які є або ключовими словами (keyword), ідентифікаторами (identifier), константами (constant), рядковими одиницями (string literal) або спецсимволами (symbol). Наприклад, printf("Hello, World! \n"); складається з наступних токенів

- printf – команда;
- (– спецсимвол(відкриваючи дужки);
- "Hello, World! \n" – рядкова константа;
-) – спецсимвол (закриваючи дужки);
- ; – спецсимвол (крапка з комою).

Крапки з комою

В мові С крапка з комою – кінець твердження. Таким чином, кожна інструкція повинна закінчуватися нею. Компілятор тоді розуміє, що вона закінчилась і потрібно перевести її в машинний код.

Приклади інструкцій:

- printf("Hello, World! \n");
- x=10;
- return 0;

Коментарі

Коментарі - це допоміжний текст в С програмі який ігнорується компілятором. Вони починаються двома символами /* та закінчуються двома символами */ як показано нижче:

```
/* my first program in C */
```

Неможна коментувати таким стилем всередині коментарю такого самого стилю та всередині літералу.

Інший стиль коментарів, що дозволений в сучасному С – це рядковий коментар, який робиться подвійним символом слеша "//" як в прикладі:

```
float x=1.0;  // ініціалізуємо дійсну (одинарної точності) змінну 'x'
```

Ідентифікатори

Ідентифікатор С – це ім'я, яке використовується для *ідентифікації змінної* (variable), *функції* (function) та інших ідентифікаторов що визначені користувачем. Ідентифікатор починається з латинської літери (або від A до Z, або від a до z) або з нижнього підкреслення '_' за яким слідує 0 або більше літер, нижніх підкреслювань та цифр (від 0 до 9).

Важливо пам'ятати, що мова С не дозволяє символів @, $, та % всередині ідентифікаторів.

Крім того, С – чутлива до регістрів (case-sensitive) програмна мова. Тобто, *Power* та *power* – два різні ідентифікатори в С. Ось приклади ідентифікаторів через кому:

Modx , zara, abc, move_name, a_123, myname50, _temp, j, J, a23b9, retVal.

Ключові слова

В таблиці приведені ключові слова С. Ці слова не можуть бути використані як ідентифікатори чи імена модулів.

Таблиця 1.1

auto	float	signed	_Alignas (з C11)
break	for	sizeof	_Alignof (з C11)
case	goto	static	_Atomic (з C11)
char	if	struct	_Bool (з C99)
const	inline (з C99)	switch	_Complex (з C99)
continue	int	typedef	_Generic (з C11)
default	long	union	_Imaginary (з C99)

do double else enum extern	register restrict (з C99) return short	unsigned void volatile while	_Noreturn (з C11) _Static_assert (з C11) _Thread_local (з C11)

Деякі ключові слова починаються з нижнього підкреслення:

Таблиця 1.2

Ключове слово	Використовується як	Визначено в
_Alignas (з C11)	alignas	stdalign.h
_Alignof (з C11)	alignof	stdalign.h
_Atomic (з C11)	atomic_bool, atomic_int, ...	stdatomic.h
_Bool (з C99)	bool	stdbool.h
_Complex (з C99)	complex	complex.h
_Generic (з C11)	(no macro)	
_Imaginary (з C99)	imaginary	complex.h
_Noreturn (з C11)	noreturn	stdnoreturn.h
_Static_assert (з C11)	static_assert	assert.h
_Thread_local (з C11)	thread_local	threads.h

Також є спецсимволи диграфи <%, %>, <:, :>, %:, та %:%: що представляють альтернативи звичайним токенам.

Директиви, що відповідають макросам (перед ними стоїть символ #):

if	ifdef	include
elif	ifndef	line
else	define	error
endif	undef	pragma
defined		

Цей специфічний токен розпізнається, якщо він знаходиться зовні макросів:

_Pragma(з C99)

Наступні два токена вважаються доповненнями до мови С:

asm
fortran

Пробіли в Сі

Лінія, що містить пробіли (whitespace), можливо з коментарієм, зветься порожньої лінією та ігнорується компілятором С.

Пробілами (Whitespace) в С також звуть порожні лінії, табуляцію, символ переходу на новий рядок та коментарі. Пробіли також відокремлюють елементи однієї інструкції від іншої та дозволяють йому зрозуміти елементи інструкції. Таким чином в інструкції

```
int age;
```
повинен бути хоча б один роздільник (пробіл) між int та age щоб компілятор розрізнив їх. З іншого богу в інструкції
```
fruit = apples + oranges;   // get the total fruit
```
непотрібні пробіли між fruit та =, або між = та apples, хоча їх бажано там ставити для гарного вигляду коду.

Введення/виведення

На C++ введення та виведення можна робити за допомогою команд
```
std::cin>>  // команда введення
std::cout<< // команда виведення
```
Приклад:
```
#include <iostream>
int main(){
  int x;
  std::cin>>x;
  int y = x*2+1;
  std::cout<<"y="<<y;
}
```
Форматоване виведення

Розглянемо класичну запропоновану одним з співавторів С Керніганом програму привітання “Hello, world!”. На С вона виглядатиме наступним чином:
```
#include <stdio.h>
int main(){
  printf("Hello, world!\n");
}
```
Вона виводить текст “Hello, world!” та переводить курсор на новий рядок за допомогою спецсимволу ‘\n’.

Команда виведення інформації на консоль — printf. Як і деякі інші команди вводу/виводу в форматованому стилі для Сі, її опис міститься у заголовочному файлі**<stdio.h>**:

printf (<керуючий рядок>, <список аргументів>);

Керуючий рядок береться у лапки і вказує компілятору вигляд інформації, що виводиться. Вона може містити специфікації перетворення та керуючи або escape-символи.

Специфікація перетворення має такий вигляд:

%<флаг> <розмір поля. точність>специфікація,

- де **флаг** може набувати наступних значень:
  - **“-”** вирівнювання вліво числа, що виводиться (за замовчуванням виконується вирівнювання вправо);
  - **“+”** виводиться знак додатного числа;
- **розмір поля** – задає мінімальну ширину поля, тобто довжину числа. Якщо ширина поля недостатня, автоматично виконується його розширення;

- **точність** – задає точність числа, тобто кількість цифр його дробової частини;
- **специфікація** вказує на вигляд інформації, що виводиться. У таблиці 1.3 наведено основні формати функції друку.

Таблиця 1.3

Формат	Тип інформації, що виводиться
%d	десяткове ціле число
% i	для виведення цілих чисел зі знаком (printf (“a=%i”, -3));
%u	для виводу беззнакових цілих чисел (printf(“s=%u”, s))
%c	один символ
%s	рядок символів
%e	число з плаваючою крапкою (експоненційний запис)
%f	число з плаваючою крапкою (десятковий запис) (printf(“b=%f\n, c=%f\n, d=%f\n”, 3.55, 82.2, 0.555));
%u	десяткове натуральне число

Керуючий рядок може містити наступні **керуючі символи:**
\n – перехід на новий рядок;
\t – горизонтальна і **\v** – вертикальна табуляція;
\b – повернення назад на один символ;
\r – повернення на початок рядка;
\a – звуковий сигнал;
\” –лапки;
\? – знак питання;
\\ - зворотний слеш.

Список аргументів – об’єкти, що друкуються (константи, змінні). Кількість аргументів та їх типи повинні відповідати специфікаціям перетворення в керуючому рядку.

Приклад 1.

```
#include <stdio.h>
const float pi = 3.14158f;
int main(){
int number=5, cost=11000, s=-777;
float bat=255, x=12.345;
printf ("%d students drank %f bottles.\n", number, bat);
```

```
printf ("Value of pi is%f.\n", pi);
printf ("The price is %d%s\n", cost,"y.e");
printf ("x=%-8.4f s=%5d%8.2f ", x, s, x);
}
```

В результаті виконання останньої функції **printf**() на екрані буде виведено:

```
x=12.3450 s= -777 12.34.
```

Локалізація

На жаль, коли ми спробуємо вивести інформацію за допомогою іншого алфавіту, зокрема українського, можливе виникнення такої ситуації, що замість символів алфавіту виведуться якійсь незрозумілі символи. Це пов'язано з тим, що C та навіть C++ розроблявся в ті часи, коли розробка програмного забезпечення велося цілком англійською, а роботу по кодуванню та відображенню символів було залишено на програміста, стандартних функцій для роботи з кодуваннями не було, юнікоду також.

Звісно, таке становище дуже шкодило переносимості програм, і були розроблені стандартні засоби для роботи з різними кодуваннями, для C - бібліотека <locale.h> (clocale.h в C++) із функцією

```
char* setlocale (int category, const char* locale);
```

Ця функція змінює поведінку стандартних функцій C для роботи з рядками у відповідності до локалі і категорії, причому категорії бувають:

LC_COLLATE - впливає на функції strcoll and strxfrm

LC_CTYPE - впливає на функції з cctype, крім isdigit and isxdigit

LC_MONETARY - впливає на інформацію про грошові одиниці з функції localeconv.

LC_NUMERIC - впливає на форматування чисел при вводі-виводі (зокрема, на десяткову кому) і включає LC_MONETARY

LC_TIME - впливає на strftime

LC_ALL - включає все попереднє

Другий параметр - назва локалі - залежить від ОС. Windows та Linux розуміють прості назви на кшталт "Ukrainian" чи "Russian"; іншим системам треба давати більш точні вказівки типу "uk_UA.cp1251" чи "en_US.utf8" (у форматі мова_країна.кодування).

Для C++ була повністю перероблена бібліотека cctype (ctype.h), і тепер є бібліотека locale, що містить всі важливі функції з ctype і ще купу різних функцій.

Крім того, в сучасних стандартах була додана підтримка юнікоду: широкі символи (wchar_t), широкі рядки (L"日本語" буде кодовано в wchar_t, а не в char), бібліотеки <cwchar> (wchar.h) та <cwctype> (wctype.h), класс std::wstring і т.д.

На жаль, при використанні старих версій Ci, незважаючи на зміні в цій бібліотеці, все одно можуть виникнути проблеми, тому краще користуватись свіжими компіляторами для уникнення цих проблем.

Для того, щоб встановити власну локаль корисно користуватись наступними автоматичними локалями

Таблиця 1.4

Ім’я локалі	Опис локалі
"C"	Мінімальна локаль "C"
""	Локаль, яка прописана в системі

Приклад:

```
/* setlocale example */
#include <stdio.h>     /* printf */
#include <locale.h>    /* struct lconv, setlocale, localeconv */

int main ()
{
   printf ("Locale is: %s\n", setlocale(LC_ALL,NULL) );
   setlocale (LC_ALL,"");
   struct lconv *lc;
   lc = localeconv ();
   printf ("Currency symbol is: %s\n-\n",lc->currency_symbol);
   setlocale (LC_ALL,"Ukrainain");
   printf ("наша валюта: %s\n-\n",lc->currency_symbol);
}
```

За допомогою структури lconv можна також змінювати можливість вводу виводу дійсних чисел у форматі крапки чи коми, параметри виводу часу і т.п., а функції localeconv повертає в програму відповідну структуру для роботи з нею.

Зупинка терміналу

Ще одна проблема, яка може виникнути при запуску програмного застосування – це те, що після запуску програми вікно терміналу, що з’явиться після виконання програми, після самого виконання закриється і не дасть насолодитися красотою виведеного тексту. Деякі середовища та платформи залишають термінал чекати спеціального його закриття, деякі дозволяють тримати його в фоні і дають можливість побачити, а деякі відразу закривають термінал після виконання. В останньому варіанті можуть допомогти декілька варіантів трюків. Перший – якщо середовище дозволяє робити відлагодження (дебаггінг, debug) програми, то можна поставити точку зупинки (брейкпойнт) перед останнім *return* і тоді дочекатись коли програма зупиниться перед виходом. Іншим варіантом є додати останню команду вводу символу з клавіатури getchar – ця команда чекає натиснення на клавіатуру:

```
#include <stdio.h>
int main(){
  printf(“Hello, world!\n”);
  getchar();  // або int c = getchar();
}
```

Форматоване введення

Функція **scanf** передбачена для форматного вводу інформації довільного вигляду. Загальний вигляд функції:

scanf (<керуючий рядок>, < список адрес>);

На відміну від функції виводу **printf**(), **scanf**() використовує у списку адреси змінних, для одержання яких перед іменем змінної ставиться символ ”&”, що позначає унарну операцію одержання адреси. Для вводу значень рядкових змінних символ ”&” не використовується. При використанні формату %s рядок вводиться до першого пропуску. Вводити дані можна як в одному рядку через пропуск, так і в різних рядках.

Дану особливість ілюструє відповідна частина програми:

```
int course;
float grant;
char name[20];
printf ( "Вкажіть ваш курс, стипендію, ім’я \n");
scanf ( "%d%f", &course, &grant);
scanf ( "%s", name); /* ”&” відсутній при зазначенні масиву символів */
```

Для введення в форматованому вигляді використовуються майже ті самі флагі та специфікації, що використовуються і для функції **printf.**

В середовищі Visual Studio можливе виникнення наступної проблеми: при компілюванні або зборці програми цей компілятор видасть помилку:

Error 1 error C4996: 'scanf': This function or variable may be unsafe. Consider using scanf_s instead. To disable deprecation, use _CRT_SECURE_NO_WARNINGS. See online help for details.

Ця помилка пов’язана з тим, що команда scanf() допускає так звану проблему переповнення буферу (Buffer Overflow Problem), що дозволяє хакерам ломати комерційні застосування, а Visual Studio, в свою чергу, вимагає більш безпечного коду від програміста. Для компіляції коду в цьому середовищі можливі наступні опції:

1) Поставити при компіляції флаг _CRT_SECURE_NO_WARNINGS у властивості проекту, або макрос #define _CRT_SECURE_NO_WARNINGS на початок проекту
2) Поставити макрос #pragma warning(disable:4996) на початок проекту
3) Скористатись, як радить компілятор, функцією форматованого вводу scanf_s. Зауважимо, що ця функція увійшла в формат С починаючі з версії С11.

```
#include <stdio.h>
int main(){
      int   i, result;
      float fp;    char  c, s[81];
      wchar_t wc, ws[81];
     result = scanf_s( "%d %f %c %C %80s %80S", &i, &fp, &c, &wc, s, ws );
// C4996 warning: consider using scanf_s
      printf( "The number of fields input is %d\n", result );
      printf( "The contents are: %d %f %c %C %s %S\n", i, fp, c, wc, s, ws);
```

```
}
```
Примітка: Більшість специфікаторів враховує пробіли всередині scanf().
Послідовні команди
```
scanf("%d", &a); // scanf_s("%d", &a);
scanf("%d", &b); // scanf_s("%d", &b);
```
зчитують два цілих числа в різних рядках (друге %d вставить новий рядок після вводу) або на той самій лінії відокремлені пробілами або табуляціями (другий %d пропустить пробіли та табуляції). Специфікатори, які не пропускають пробіли, такі як %c, можна примусити приймати їх за допомогою пробілів перед специфікатором:
```
scanf("%d", &a); // scanf_s("%d", &a);
scanf(" %c", &b); // scanf_s("%d", &b);
/* пропустить всі пробіли перед %d, потім введе символ.*/
```
Примітка 2. Якщо у вас більш-менш сучасний компілятор(C11), то краще використовувати замість scanf функцію scanf_s. Якщо ж потрібно забезпечити крос-платформеність, то краще записати власний варіант введення, що використовує scanf_s у випадку сумісного з C11 компілятора, а в іншому випадку, наприклад, форматоване введення рядку з потрібними специфічними опціями.

Визначення змінної в С

Визначення (означення, декларація) змінної призначено для того, що компілятор визначив де та скільки потрібно пам'яті виділити під цю змінну. Для цього в визначенні потрібно вказати тип даної змінної та вказати одну або більше ідентифікаторів змінних через кому.

Тип_змінної список_змінних;

Тут, **Тип_змінної** повинен бути типом мови С включаючи char, w_char, int, float, double, bool або будь-яким визначеним користувачем (user-defined) типом чи об'єктом; а **список_змінних** може складатись з одного або більше ідентифікаторів змінних через кому.

Приклади декларацій змінних:
```
int    i, j1, k_2;
char   c, ch;
float  f, salary;
double d;
```
Рядок **int i, j1, k_2;** визначає цілі змінні i, j1, та k_2; ця інструкція каже компілятору створити змінні, що звуться i, j1 та k_2 типу int.

Змінні можуть бути **ініціалізовані** (initialized), тобто їм може бути присвоєно початкове значення під час визначення. Ініціалізатор складається зі знаку рівності (equal sign), за яким йде вираз:

Тип_змінної ім'я_змінної = <значення>;

Приклади:
```
extern int d = 3, f = 5;     // ініціалізація цілих змінних d і f.
```

```
int d = 3, f = 5;         // ініціалізація d і f.
byte z = 22;              // ініціалізація  z.
char x = 'x';             // змінній x присвоєно значення 'x'.
```

Якщо змінна визначена без ініціалізації: змінні зі статичним типом видимості (static storage duration) ініціалізовані нулем або NULL (всі байти мають значення 0); значення всіх інших змінних невизначено, тобто може бути яким завгодно.

Присвоєння змінної приводить до того, що компілятор гарантує, що існує змінна даного типу та іменем, так що компілятор може проводити операції з цієї змінною без запиту про те, чому ця змінна дорівнює. Це знання зберігається лише під час компіляції, під час лінковки потрібне також визначення цієї змінної.

Різниця між присвоєнням та ініціалізацією С:

Таблиця 1.5

Присвоєння	Ініціалізація
Присвоєння повідомляє компілятору про тип даних та розмір змінної.	Визначення визначає розмір пам'яті під змінну
Змінна може бути перевизначена декілька разів	Може відбутись лише 1 раз.
Присвоєння значення та властивостей змінній	

Вираз: Оператори С

Вираз за допомогою якого компілятор розуміє, що потрібно виконати певне обчислення або операцію над змінною складається з ідентифікаторів, що позначають змінні та позначок операцій, що відповідають певним діям, що компілятор повинен вміти з ними робити.

Позначки операторів – це один або декілька символів, що визначають дію над операндами. Оператори поділяють на унарні, бінарні та тернарні за кількістю операндів, які беруть участь в операції (таблиця 1.6).

Таблиця 1.6

Оператор	Короткий опис
Унарні оператори	
&	Оператор одержання адреси операнду
*	Звернення за адресою (розіменування)
-	Унарний мінус – змінює знак арифметичного операнду
~	Порозрядове інвертування внутрішнього двійкового коду (побітове заперечення)
!	Логічне заперечення (НЕ) значення операнду. Цілочисельний результат 0 (якщо операнд ненульовий, тобто істинний) або 1 (якщо операнд нульовий, тобто хибний). Таким чином: !1 дорівнює 0; !2 дорівнює 0; !(-5)=0; !0 дорівнює 1.

++	Інкремент (збільшення на одиницю): *Префіксна операція (++x)* збільшує операнд на 1 до його використання. *Постфіксна операція (x++)* збільшує операнд на 1 після його використання. int m=1, n=2; int a=(m++)+n; // a=3, m=2, n=2 int b=m+(++n);// b=6, m=2, n=3
--	Декремент (зменшення на одиницю): Префіксна операція (--x) зменшує операнд на 1 до його використання. Постфіксна операція (x--) зменшує операнд на 1 після його використання.
sizeof	Обчислення розміру (в байтах) об'єкта того типу, який має операнд. Має дві форми: 1) sizeof (вираз); sizeof(1.0); // Результат - 8, Дійсні константи за замовчуванням мають тип double; 2) sizeof (тип) sizeof (char); // Результат – 1.
Бінарні оператори	
Арифметичні оператори	
+	Бінарний плюс (додавання арифметичних операндів)
-	Бінарний мінус (віднімання арифметичних операндів)
Мультиплікативні оператори	
*	Добуток операндів арифметичного типу
Оператор	Короткий опис
/	Ділення операндів арифметичного типу (якщо операнди цілочисельні, абсолютне значення результату заокруглюється до цілого, тобто 20/3 дорівнює 6)
%	Одержання залишку від ділення цілочисельних операндів (13%4 = 1)
Оператори зсуву (визначені лише для цілочисельних операндів)	

<<	Зсув вліво бітового представлення значення лівого цілочисельного операнда на кількість розрядів, рівну значенню правого операнда (4<<2 дорівнює 16, тобто код 4 100, а звільнені розряду обнуляються, 10000 – код 16)
>>	Зсув вправо бітового представлення значення правого цілочисельного операнду на кількість розрядів, рівну значенню правого операнду
Порозрядні операції	
&	Порозрядна кон'юнкція (І) бітових представлень значень цілочисельних операндів
|	Порозрядна диз'юнкція (АБО) бітових представлень значень цілочисельних операндів
^	Порозрядне виключне АБО бітових представлень значень цілочисельних операндів
Операції порівняння	
<	Менше ніж
>	Більше ніж
<=	Менше або рівне
>=	Більше або рівне
= =	Рівне
!=	Не дорівнює
Операція	Короткий опис
Логічні бінарні операції	
&&	Кон'юнкція (І) цілочисельних операндів або відношень, цілочисельний результат (0) або (1)
||	Диз'юнкція (АБО) цілочисельних операндів або відношень, цілочисельний результат (0) або (1) (умова 0<x<1 мовою С++ записується як 0<x && x<1)
Тернарний оператор	
Умовна операція	

? :	Вираз1 ? Вираз2 : Вираз3; Першим вираховується значення Виразу1. Якщо воно істинне, тоді обчислюється значенняВиразу2, яке стає результатом. Якщо при обчисленніВиразу1одержуємо 0, тоді в якості результату береться значенняВиразу3. Наприклад: x<0 ? -x : x; //обчислюється абсолютна величина.

Операції присвоювання

Таблиця 1.7

Оператор	Пояснення	Приклад
=	Присвоїти значення виразу-операнду з правої частини операнду лівої частини	P=10.5-3*x
*=	Присвоїти операнду лівої частини добуток значень обох операндів	P*=2 еквівалентно P=P*2
/=	Присвоїти операнду лівої частини результат від ділення значення лівого операнду на значення правого	P/=(2.2-x) еквівалентно P=P/(2.2-x)
%=	Присвоїти лівому операнду залишок від ділення цілочисельного значення лівого операнду на цілочисельне значення правого операнду	P%=3 еквівалентно P=P%3
+=	Присвоїти операнду лівої частини суму значень обох операндів	A+=B еквівалентно A=A+B
-=	Присвоїти операнду лівої частини різницю значень лівого і правого операндів	X-=3.4-y еквівалентно X=X-(3.4-y)

Порядок виконання операцій регулюється наступною таблицею пріоритетів виконання операцій.

Пріоритет виконання операторів

Таблиця 1.8

Ранг	Операції	Напрямок виконання
1	() (виклик функції), [], ->, "."	>>>

2	!, ~, +, - (унарні), ++, --, *, (тип), sizeof, (*new,delete – Ci++)*	<<<
3	.* , ->* - *Ci++*	>>>
4	*, /, % (бінарні)	>>>
5	+, - (бінарні)	>>>
6	<<, >>	>>>
7	<, <=, =>, >	>>>
8	==, !=	>>>
9	& (порозрядна)	>>>
10	^	>>>
11	| (порозрядна)	>>>
12	&& (логічна)	>>>
13	|| (логічна)	>>>
14	?: (тернарний)	<<<
15	=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=,>>=	<<<
16	"," (кома)	>>>

Використання математичної бібліотеки

Звичайно, не хотілося б використанням стандартних операцій та арифметичних дій над змінними обмежуватись. Навіть у далеких від математики галузях, а тим більш при програмуванні штучного інтелекту, задачах шифрування та контроля чи аналізу каналів зв'язку часто-густо треба використовувати відомі математичні функції. Звичайно, не варто (хоча інколи й корисно вміти) кожен раз реалізовувати ці функції, а скористатись бібліотеками, які входять в стандарт мови С.

Для того, щоб використовувати бібліотеку математичних функцій, потрібно підключити бібліотеку math.h. Іншим варіантом є бібліотека tgmath.h, яку варто розглянути окремо. Після того, як ця бібліотека включається в файл програми за допомогою директиви #include в даній програмі можна використовувати найбільш розповсюджені математичні функції.

Основні математичні функції мови С/С++, опис яких міститься у файлі **<math.h>,** наведено в таблиці 1.9.

Таблиця 1.9

Математичний запис	Функція	Пояснення	Приклад
arccos(x)	acos	Повертає арккосинус кута, рівного x радіан	acos(x);
arcsin(x)	asin	Повертає арксинус аргументу x в радіанах	asin(x);
arctg (x)	atan	Повертає арктангенс аргументу x в радіанах	atan(x);
arctg(x/y)	atan2	Повертає арктангенс відношення параметрів x та y в радіанах	atan2(x, y);
]x[	ceil	Заокруглює дійсне значення x до найближчого більшого цілого і повертає його як дійсне	ceil(x);
cos(x)	cos	Повертає косинус кута, рівного x радіан	cos(x);
ch(x)	cosh	Повертає гіперболічний косинус аргументу, рівного x радіан	cosh(x);
e^x	exp	Повертає результат піднесення числа e до степені x	exp(x);
|x|	fabs	Повертає модуль дійсного числа x	fabs(x);
[x]	floor	Заокруглює дійсне число до найближчого меншого числа і повертає результат як дійсний	floor(x);
x mod y	fmod	Повертає залишок ділення x на y. Аналогічна операції %, але працює з дійсними числами	fmod(x, y);
ln (x)	log	Повертає значення натурального логарифму x	log(x);
lg (x)	log10	Повертає значення десяткового логарифму x	log10(x);
x^y	pow	Вираховує значення числа x у степені y	pow(x, y);

sin(x)	sin	Повертає синус кута, рівного x радіан	sin(x);
sh(x)	sinh	Повертає гіперболічний синус кута, рівного x радіан	sinh(x);
$\sqrt{x}$	sqrt	Визначає корінь квадратний числа x	sqrt(x);
tg (x)	tan	Повертає тангенс кута, рівного x радіан	tan(x);
tgh(x)	tanh	Повертає гіперболічний тангенс кута, рівного x радіан	tanh(x);

Приклади:

```
#include<stdio.h>
#include<math.h>

int main(){
  double x, y, z;
  scanf("%lf %lf",&x,&y);
  z = exp(x)*cos(y);
  printf("z=%lf",z);
}
```

Для компіляції файла з математичною бібліотекою деякі компілятори вимагають вказання що компілювати потрібно з цією бібліотекою, наприклад, для gcc під Linux:

```
>>>gcc examle.c –lm
```

або якщо потрібно вказати імя вихідного виконувано файлу:

```
>>> gcc examle.c –o example –lm.
```

Дійсні типи даних

В наступній таблиці приведено дійсні типи даних на С:

Таблиця 1.10

Тип	Розмір типу	Межі значень	Точність
float	4 байти	1.2E-38 - 3.4E+38	6 десяткових знаків
double	8 байтів	2.3E-308 - 1.7E+308	15 десяткових знаків
long double	10 байтів	3.4E-4932 - 1.1E+4932	19 десяткових знаків

В заголовочному файлі float.h визначені точність та інші деталі представлення дійсних типів та їхніх значень:

```
#include <stdio.h>
#include <float.h>
```

```
int main() {
  printf("Storage size for float : %ld \n", sizeof(float));
  printf("Minimum float positive value: %E\n", FLT_MIN );
  printf("Maximum float positive value: %E\n", FLT_MAX );
  printf("Precision value: %d\n", FLT_DIG );
}
```

На 64-бітному Linux наприклад буде наступний результат:

```
Storage size for float : 4
Minimum float positive value: 1.175494E-38
Maximum float positive value: 3.402823E+38
Precision value: 6
```

Константи дійсних типів

Константи дійсних типів мають цілу частину, дробову частину та можуть мати експоненту, згідно специфікації IEEE-754. Вони можуть бути представлені як в десятковому, так і експоненційному (науковому) форматі.

При представленні в десятковій формі потрібно записати цілу частину, десяткову крапку та дробову частину, а потім для типу float можна записати в кінці літеру(суфікс) f або F. Для типу long double в кінці додається суфікс L.

Приклади:

```
float X1 = 123.567;
float X2 = 3.576f;
double X3 =0.2342;
long double x4 =33.56L;
```

Для представлення в експоненційному (науковому) форматі записується спочатку мантиса таким же чином, як і десяткове дійсне число, потім літера e або E і після чього значення експоненти як ціле число, можливо зі знаком.

Приклади:

```
double y1 = 3.14159e0;
double y2 = 314159E-5;
float y3 = 0.314e-1f;
long double y4 = 5.46e235L;
```

Наступні варіанти запису дійсних чисел некоректні:

```
510E       /* Помилка- Illegal: incomplete exponent */
210f       /* Помилка- Illegal: no decimal or exponent */
.e55       /* Помилка- Illegal: missing integer or fraction */
```

Введення/виведення дійсних чисел

Для введення дійсного числа можна використати наступні варіанти викликів функцій scanf:

```
float f1,f2,f3;
scanf("%f", &f1);  // введення в десятковому форматі
scanf("%e", &f2);  // введення в науковому форматі з маленькою e
scanf("%E", &f3);  // введення в науковому форматі з великою E
```

```
double d1,d2,d3;
scanf("%lf", &d1);  // введення в десятковому форматі
scanf("%le", &d2);  // введення в науковому форматі з маленькою e
scanf("%lE", &d3);  // введення в науковому форматі з великою E

long double r1, r2,r3;
scanf("%Lf", &r1);   // введення в десятковому форматі
scanf("%Le", &r2); // введення в науковому форматі з маленькою e
scanf("%LE", &r3); // введення в науковому форматі з великою E
```
Для виведення дійсного числа можна використати наступні варіанти викликів функцій printf:
```
printf("%f", f1);  // виведення в десятковому форматі
printf("%le", f2);  // виведення в науковому форматі з маленькою e
printf("%lE", f3);  // виведення в науковому форматі з великою E
printf("%g", f3);  // виведення в десятковому форматі з найменшою кількістю знаків
printf("%lf", d1);  // виведення в десятковому форматі
printf("%le", d2);  // виведення в науковому форматі з маленькою e
printf("%lE", d3);  // виведення в науковому форматі з великою E
printf("%g", d1);  // виведення в десятковому форматі з найменшою кількістю знаків

printf("%Lf", r1); // виведення в десятковому форматі
printf("%Le", r2); // виведення в науковому форматі з маленькою e
printf("%LE", r3); // виведення в науковому форматі з великою E
printf("%Lg", r1);  // виведення в десятковому форматі з найменшою кількістю знаків
```

Помітимо, що при введенні дійсних чисел важливо не переплутати специфікатори, програма з прикладу наступного вигляду може підрахувати невірне значення:

```
#include<stdio.h>
#include<math.h>

int main(){
  double x, y, z;
  scanf("%f %f",&x,&y);
  z = exp(x)*cos(y);
  printf("z=%lf",z);
```

```
}
```

На C++ можна не враховувати такі нюанси:

```
#include<iostream>
#include<math>

int main(){
  double x, y, z;
  std::cin>>x>>y;
  z = exp(x)*cos(y);
  std::cout<<z;
  printf("z=%lf",z); // в C++ можна використовувати С функції вводу-виводу
}
```

2. Цілі типи. Розгалудження. Цикли

Цілі типи С. Цілі типи фіксованої довжини. Заголовочний файл stdint.h. Цілочисельні константи, введення та виедення. Визначення меж типів за допомогою limits.h.
Переповнення цілих типів та як з ним боротися. Оператор sizeof(). Оператори для роботи з дійсним типом.
Булевий тип на С. Модуль stdbool.h. Логічні операції.
Бітові операції на С.
Розгалудження на С. Умовні конструкції if..else, тернарний оператор, альтернатива(switch).
Цикли на С. Цикл з передумовою, з післяумовою та з лічильником

Цілі типи

В С та C++ існує декілька цілих типів, що відрізняються розміром та типом арифметики (знакова або беззнакова):

- short int (також можна позначити short, можна використати ключове слово signed)
- unsigned short int (також можна позначити unsigned short)
- int (також можна позначити signed int). Це найбільш оптимальний цілий тип для даної платформи, і він гарантовано має розмір не менше 16 bits. Більшість сучасних моделей використовує 32 біти (див. таблицю 2.1).
- unsigned int (також можна позначити unsigned), натуральний або беззнаковий еквівалент int, що використовує модульну арифметику при арифметичних операціях. Зручний для маніпулювання бітами.
- long int (також можна позначити long)
- unsigned long int (також можна позначити unsigned long)
- long long int (також можна позначити long long) (з C99)
- unsigned long long int (також можна позначити unsigned long long) (з C99)

Відмітимо що порядок слів в опису може бути довільним: unsigned long long int та long int unsigned long позначають той самий тип.

Наступна таблиця містить всі цілі типи та їх бітність для найбільш поширених архітектурних моделей даних:

Цілі типи Сі

Таблиця 2.1

Специфікатор типу	**Еквівалент**	**Розмір типу**				
		C standard	**LP32**	**ILP32**	**LLP64**	**LP64**
short	short int	мінімум **16**	**16**	**16**	**16**	**16**
short int						
signed short						
signed short int						
unsigned short	unsigned short int					
unsigned short int						
int	int	мінімум **16**	**16**	**32**	**32**	**32**
signed						
signed int						
unsigned	unsigned int					
unsigned int						
long	long int	мінімум **32**	**32**	**32**	**32**	**64**
long int						
signed long						
signed long int						
unsigned long	unsigned long int					
unsigned long int						
long long	long long int (C99)	мінімум **64**	**64**	**64**	**64**	**64**
long long int						
signed long long						
signed long long int						
unsigned long long	unsigned long long int (C99)					
unsigned long long int						

Стандарт С не гарантує розміри бітів, єдине що він гарантує, це те що 1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long).

Цілочисельна арифметика знакових типів та беззнакових типів відрізняється зокрема в тих випадках, коли розмір результату більше ніж розмір вхідних даних, отже в випадках, коли це можливо треба бути особливо уважним

Також вкрай небажано застосовувати цілий та натуральний операнди одночасно в арифметичних виразах.

Моделі даних

Вибір розмірів типів С визначається моделлю даних(*data model), що в свою чергу визначається архітектурою комп'ютера та його операційною системою*. Найбільш популярні наступні моделі даних:

32 бітові системи:

LP32 або **2/4/4** (int - 16-bit, long та вказівник 32-bit)

Win16 API

ILP32 або **4/4/4** (int, long, та вказівник 32-bit);
Win32 API
Unix та Unix-like systems (Linux, Mac OS X)
64 бітові системи:
LLP64 або **4/4/8** (int та long - 32-bit, вказівник 64-bit)
Win64 API
LP64 або **4/8/8** (int - 32-bit, long та вказівник 64-bit)
Unix та Unix-like systems (Linux, Mac OS X)

Цілі типи фіксованої довжини

Інколи виникає потребі зафіксувати розмір типів для будь-якої платформи. Для цього можна або перевизначити самому типи за допомогою typedef або скористатись бібліотекою <stdint.h> , яка включена в стандарт з C99.

Цілі типи визначені в <stdint.h>

Таблиця 2.2

int8_t **int16_t** **int32_t** **int64_t**	Знаковий цілий тип з розміром 8, 16, 32 і 64 біт відповідно
int_fast8_t **int_fast16_t** **int_fast32_t** **int_fast64_t**	Найбільший знаковий цілий тип з розміром мінімум 8, 16, 32 та 64 біт відповідно
int_least8_t **int_least16_t** **int_least32_t** **int_least64_t**	Найменший знаковий цілий тип з розміром 8, 16, 32 та 64 біт відповідно
intmax_t	Максимальний за розміром цілий тип
intptr_t	Цілий тип в який можна записати вказівник
uint8_t **uint16_t** **uint32_t** **uint64_t**	Натуральний тип з розміром 8, 16, 32 і 64 біт відповідно (лише якщо платформа їх підтримує)
uint_fast8_t **uint_fast16_t** **uint_fast32_t** **uint_fast64_t**	Найбільший натуральний тип з розміром 8, 16, 32 та 64 біт відповідно
uint_least8_t **uint_least16_t** **uint_least32_t** **uint_least64_t**	Найменший натуральний тип з розмірами 8, 16, 32 та 64 біт відповідно
uintmax_t	Максимальний за довжиною натуральний тип
uintptr_t	Натуральний тип в який можна записати вказівник

Цілочисельні константи

Для того щоб визначити константи цілого типу можна обрати наступні варіанти:

5. звичайний ***десятковий*** тип: ненульова десяткова цифра (1, 2, 3, 4, 5, 6, 7, 8, 9), за якою слідує довільна кількість десяткових цифр (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
6. ***вісімкова*** константа: цифра (0) за якою довільна кількість вісімкових цифр (0, 1, 2, 3, 4, 5, 6, 7)
7. ***шістнадцятирічна*** константа: двійний спецсимвол 0x або 0X за яким слідує довільна кількість шістнадцяткових цифр (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C, d, D, e, E, f, F).

Наприклад, наступні змінні ініціалізовані тим самим значенням:

```
int d = 42;
int o = 052;
int x = 0x2a;
int X = 0X2A;
```

Суфікси для форматованого введення/виведення цілих типів

Для того, щоб позначити програмний тип можуть використовуватися суфікси (вони можуть бути в будь-якому порядку):

- натуральний суфікс (unsigned-suffix) (символ u або символ U)
- суфікс довжини (long-suffix) (символ l або символ L) або long-long-suffix (the суфікс ll або LL) (з C99)

Специфікатори для цілочисельних констант

Таблиця 2.3

суфікс	Основа числення 10	Основи числення 8 та 16
no suffix	int long int unsigned long int (until C99) long long int (з C99)	int unsigned int long int unsigned long int long long int(з C99) unsigned long long int(з C99)
u або U	unsigned int unsigned long int unsigned long long int(з C99)	unsigned int unsigned long int unsigned long long int(з C99)
l або L	long int unsigned long int(until C99) long long int(з C99)	long int unsigned long int long long int(з C99) unsigned long long int(з C99)
l/L та u/U	unsigned long int unsigned long long int(з C99)	unsigned long int unsigned long long int(з C99)
ll або LL	long long int(з C99)	long long int(з C99) unsigned long long int(з C99)
ll/LL та u/U	unsigned long long int(з C99)	unsigned long long int(з C99)

Літери в числових константах незалежать від регістру (case-insensitive): 0xDeAdBaBeU та 0XdeadBABEu - те саме число (виключення long-long-suffix, який або ll або LL, але ні lL ні Ll)

Відмітимо, що в С немає від'ємних констант. Вирази такі як -1 визначають унарний оператор (unary minus operator) що застосовується до константи, та можливо приведення до іншого типу (type conversions).

Константи, що закінчується е або E, якщо за ними є знаки + або – повинні бути відокремлені пробілами (щоб не переплутати з науковим представленням дійсного типу):

```
int x = 0xE+2;   // помилка
int y = 0xa+2;   // OK
int z = 0xE +2;  // OK
int q = (0xE)+2; // OK
```

Приклад:

```
#include <stdio.h>
#include <inttypes.h>
int main(){
   printf("123 = %d\n", 123);
   printf("0123 = %d\n", 0123);
   printf("0x123 = %d\n", 0x123);
   printf("12345678901234567890ull = %lu\n", 12345678901234567890ull);
   // цей тип 64-бітовий (unsigned long long або можливо unsigned long)
   // навіть без суфіксу long
   printf("12345678901234567890u = %"PRIu64"\n", 12345678901234567890u );

//  printf("%lld\n", -9223372036854775808); // ПОМИЛКА
// величина 9223372036854775808 за межами типу signed long long, що є найбільшим типом дозволеним для беззнакового та безсуфіксного тип
   printf("%llu\n", -92233720368547758080ull );
   /* унарний мінус застосований     до беззнакового числа віднімає його від костанти 2^64, що дає беззнакове (unsigned)значення 9223372036854775808 */

   printf("%ld\n", -9223372036854775807ull - 1);
   // коректна форма знакового значення (signed value)-9223372036854775808
}
```

Результат:

```
123 = 123
0123 = 83
0x123 = 291
12345678901234567890ull = 1234567890123456789
12345678901234567890u = 1234567890123456789
9223372036854775808
-9223372036854775808
```

Символьний тип

Ще одним варіантом цілого типу є символьні типи:

1. signed char – знакове представлення символьного типу.
2. unsigned char – беззнакове представлення знакового типу. Може використовуватись для представлення довільного об'єкту чи ділянки пам'яті (object representations або raw memory).
3. char – тип для представлення символів. Є еквівалентним або signed char або unsigned char (що залежить від реалізації та може бути визначено через командний рядок, хоча на більшості платформ – це signed char), але це окремий тип, що відрізняється і від signed char та від unsigned char.

Також в сучасному С за допомого typedef визначають типи wchar_t , char16_t, та char32_t (з С11) для представлення символів за допомогою юнікоду.

Макроси бібліотеки limits.h

Бібліотека **limits.h** визначай різні властивості різних типів С. Макроси визначені в цій бібліотеці обмежують значення визначені в char, int та long.

Ці межи зокрема визначають кількість та область різних значень, що може приймати даний тип, наприклад unsigned char може приймати від 0 до 255.

Дані значення можуть прийматись різними та бути визначеними директивою #define , але не можуть бути нижче визначених.

Таблиця 2.4

Макрос	**Значення**	**Опис**
CHAR_BIT	8	Кількість бітів в байті.
SCHAR_MIN	-128	Мінімальне значення signed char.
SCHAR_MAX	+127	Максимальне значення signed char.
UCHAR_MAX	255	Максимальне значення unsigned char.
CHAR_MIN	-128	Мінімальне значення типу char і його значення буде рівне SCHAR_MIN якщо char буде приймати від'ємні значення, інакше воно рівне 0.
CHAR_MAX	+127	Максимальне значення типу char і його значення буде рівне SCHAR_MAX якщо char буде приймати від'ємні значення, інакше воно рівне UCHAR_MAX.
MB_LEN_MAX	16	Максимальна кількість бітів в двубайтовому типі
SHRT_MIN	-32768	Мінімальне значення short int.
SHRT_MAX	+32767	Максимальне значення short int.
USHRT_MAX	65535	Максимальне значення unsigned short int.
INT_MIN	-2147483648	Мінімальне значення int.
INT_MAX	+2147483647	Максимальне значення int.
UINT_MAX	4294967295	Максимальне значення unsigned int.
LONG_MIN	-9223372036854775808	Мінімальне значення long int.
LONG_MAX	+9223372036854775807	Максимальне значення long int.
ULONG_MAX	18446744073709551615	Максимальне значення unsigned long int.

Приклад

Приклад застосування констант з **limits.h**.

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

```
#include <limits.h>

int main() {
  printf("The number of bits in a byte %d\n", CHAR_BIT);

  printf("The minimum value of SIGNED CHAR = %d\n", SCHAR_MIN);
  printf("The maximum value of SIGNED CHAR = %d\n", SCHAR_MAX);
  printf("The maximum value of UNSIGNED CHAR = %d\n", UCHAR_MAX);

  printf("The minimum value of SHORT INT = %d\n", SHRT_MIN);
  printf("The maximum value of SHORT INT = %d\n", SHRT_MAX);

  printf("The minimum value of INT = %d\n", INT_MIN);
  printf("The maximum value of INT = %d\n", INT_MAX);

  printf("The minimum value of CHAR = %d\n", CHAR_MIN);
  printf("The maximum value of CHAR = %d\n", CHAR_MAX);

  printf("The minimum value of LONG = %ld\n", LONG_MIN);
  printf("The maximum value of LONG = %ld\n", LONG_MAX);

  return(0);
}
```

Результат повинен бути:

```
The maximum value of UNSIGNED CHAR = 255
The minimum value of SHORT INT = -32768
The maximum value of SHORT INT = 32767
The minimum value of INT = -2147483648
The maximum value of INT = 2147483647
The minimum value of CHAR = -128
The maximum value of CHAR = 127
The minimum value of LONG = -9223372036854775808
The maximum value of LONG = 9223372036854775807
```

Переповнення цілих чисел

При додаванні цілих чисел може виходити переповнення. Проблема полягає в тому, що комп'ютер не видає попередження при їх появі: програма продовжить виконання з невірними даними. Більше того, поведінка при переповненні є визначеною стандартом та фіксованою лише для цілих без знаку (натуральних).

Переповнення може привести до великих проблем: обнулінню та затиранню даних, можливим експлойтам, помилкам, що буде трудно выдтворити та знайти при відладці, та ці помилки можуть накопичуватися з часом. Розглянемо деякі прийоми боротьби з переповненнями для цілих чисел зі знаком та натуральних чисел.

1. Попередня перевірка даних. З файлу limits.h можна дізнатися максимальне і мінімальне значення для чисел типу int. Якщо обидва числа додатні, то їх сума не більша за INT_MAX, якщо різниця INT_MAX і одного з чисел менше другого числа. Якщо обидва числа від'ємні, то різниця INT_MIN і одного з чисел повинна бути більше іншого. Якщо обидва числа мають різні знаки, то їх сума не більша INT_MAX або INT_MIN.

```
int sum1(int a, int b, int *overflow) {
int c = 0;
if (a > 0 && b > 0 && (INT_MAX - b < a) || a < 0 && b < 0 && (INT_MIN - b > a)){
   *overflow = 1;
}
else{
   *overflow = 0;
   c = a + b;
}
return c;
}
```

В цій функції змінній overflow буде присвоєно значення 1, якщо було переповнення. Функція повертає суму, незалежно від результату додавання.
2. Другий спосіб перевірки – взяти для суми тип, максимальне й мінімальне значення якого відомо що більше суми двох цілих. Після додавання необхідно перевірити, щоб сума була не більшою ніж INT_MAX і не меншою INT_MIN.

```
int sum2(int a, int b, int *overflow) {
signed long long c = (signed long long) a + (signed long long) b;
if (c < INT_MAX && c > INT_MIN) {
*overflow = 0;
c = a + b;
}
else{
   *overflow = 1;
}
return (int) c;
}
```

Зверніть увагу на явне приведення типу. Без нього спочатку пройде переповнення і неправильне число буде записано в змінну c.
3. Третій шлях перевірки платформозалежний, більш того, його реалізація буде різною для різних компіляторів. При переповненні цілих (зазвичай) ставиться флаг переповнення в регістрі флагів. Можна на асемблері перевірити значення флагу відразу ж після виконання додавання.

Робота з натуральними числами без знаку значно простіша: при переповненні виходить обнуління і відомо, що отримане число буде менше кожного з доданків.

```
unsigned usumm(unsigned a, unsigned b, int *overflow) {
   unsigned c = a + b;
if (c < a || c < b) {
   *overflow = 1;
}
```

```
else{
   *overflow = 0;
}
return c;
}
```

Операція sizeof()

Дана операція обчислює розмір пам'яті, необхідний для розміщення в ній виразів або змінних вказаних типів.

Операція має дві форми:

1). **ім'я_типу A; sizeof (A);**

Приклад:

```
unsigned long x;
unsigned long y = sizeof(x);
```

2).**sizeof (ім'я_типу);**

Приклад:

```
unsigned long x;
unsigned long  y = sizeof( unsigned long );
```

Операцію sizeof() можна застосовувати до констант, типів або змінних, у результаті чого буде отримано число байт, що відводяться під операнд. Наприклад, sizeof(int) поверне число байт для розміщення змінної типу int.

Операції над цілими числами

Над цілими числами можна виконувати майже всі дії, що перелічені в таблицях 1.6 -1.7:

!, ~, +, - (унарні), ++, --, *, (тип), sizeof	<<<
*, /, % (бінарні)	>>>
+, - (бінарні)	>>>
<<, >>	>>>
<, <=, =>, >	>>>
==, !=	>>>
& (порозрядний)	>>>
^	>>>
| (порозрядний)	>>>
&& (логічна)	>>>
|| (логічний)	>>>

?: (тернарний оператор)	<<<
=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=,>>=	<<<

Примітка. Відмітимо важливий момент, пов'язаний з операцією ділення. Якщо обидва операнди є цілими числами***, то результат ділення також цілий, тобто операція "/" позначає цілочисельне ділення***. Відповідно % - це остача від цілочисельного ділення. Для того, щоб отримати дійсне число як результат ділення цілих чисел потрібно якимось чином привести один з операндів до дійсного числа. Наприклад

```
int x,y;
scanf("%d %d",&x, &y);

float z = (float)x/y; //1) перетворення чисельнику
printf("z=%f",z);
z = (x+0.0)/y; //2) перетворення значення виразу
printf("z=%f",z);
z = x/(float)y; //3) перетворення знаменнику
printf("z=%f",z);
```

Булевий тип

Булевий тип з'явився в мові С з стандарту С99. Для його підключення потрібна бібліотека <stdbool.h>. Він дозволяє зберігати змінні, які можуть мати значення 0 або 1, які доступні також за допомогою макросів (літералів) true та false.
Ініціалізація або присвоєння може виглядати так:

```
bool x1 = true;  // bool x1 =1;
bool y1 = false;  // bool y1 =0;
```

Відмітимо також, що перетворення до цього типу робиться за тим правилом, що будь-який ненульовий вираз перетворюється на 1(true), і лише вираз рівний нулю перетворюється на 0(false), зокрема:
bool(0.5) дорівнює 1
bool(4) дорівнює 1
bool(0) дорівнює 0.

Операції порівняння

Ще одним варіантом присвоєння значення булевому виразу є присвоєння за допомогою порівняння двох змінних або констант.
Перелік операцій порівняння наведено в таблиці 2.5.

Таблиця 2.5

Операція	**Значення**
<	менше
<=	менше або рівне
==	перевірка на рівність

>=	більше або рівно
>	більше
!=	перевірка на нерівність

Операції порівняння здебільшого використовуються в умовних виразах. Приклади умовних виразів:

- b<0,
- 'b'=='B',
- 'f'!='F',
- 201>=225.

Кожна умова перевіряється: істинна вона чи хибна. Точніше слід сказати, що кожна умова приймає значення "істинно" (true) або "хибно" (false). ***Результатом умовного виразу й цілочисельне арифметичне значення. "Істинно" - це ненульова величина, а "хибно" - це нуль. В більшості випадків в якості ненульового значення "істинно" використовується одиниця. Тобто 1- true, 0 – false.***

Приклад:

```
#include<stdio.h>
int main(){
  int tr, fal;
  tr=(11<=15); /* вираз істинний */
  fal=(1>1); /* вираз хибний */
  printf("true - %d false - %d ",tr,fal);
  return 0;
}
```

Логічні операції

Для маніпуляцій з логічним (булевим) типом, наприклад, для "об'єднання" виразів порівняння, використовуються стандартні логічні операції логічного множення (оператор &&) з правилами логічного І, логічного додавання(оператор ||) за правилами логічного АБО та логічного заперечення (оператор !), а також їх унарні форми (таблиця 2.6.). При цьому як правило використовуються операторні форми, але можливо використання макросів, що вимагає підключення бібліотеки iso646.h. Заголовочний файл iso646.h містить С оператори, що підтриміються ISO646 стандартом (в С++ ця підтримка виконується автоматично.

Логічні операції

Таблиця 2.6

Значення	Оператор	Макрос (з iso646.h)
логічне І (and)	&&	and
логічне АБО (or)	||	or
логічне заперечення (not)	!	not
нерівність	!=	not_eq

Зауваження. Складні логічні вирази обчислюються "раціональним способом") (lazy evaluation). Наприклад, якщо у виразі (A<=B)&&(B<=C) виявилось, що A більше B, то всі вирази, як і його перша частина (A<=B), приймають значення "хибно", тому друга частина (B<=C) не обчислюється.
Таблиця істинності логічних операцій наведена в таблиці 2.7.

Таблиця 2.7

E1	**E2**	**E1&&E2**	**E1||E2**	**!E1**
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Порозрядні операції (побітові операції)

Порозрядні оператори застосовуються тільки до цілочисельних операндів і "працюють" з їх двійковими представленнями. Ці оператори неможливо використовувати із змінними типу double, float, long double. Також ці оператори мають унарні форми та аналогі макроси з бібліотеки iso646.h. Перелік порозрядних операторів наведено в таблиці 2.8.
Примітка. Краще використовувати ці оператори лише для беззнакового типу, для знакового типу звертайте увагу як в вашій архітектурі компілятор представляє унарний знак «мінус» - зазвичай, це перший біт числа, але слід мати на увазі, що, взагалі кажучи, це невизначено стандартом.

Порозрядні операції

Таблиця 2.8

Оператор	**Значення**	**Макрос (з iso646.h)**
~	порозрядне заперечення	compl
& та &=	побітова кон'юнкція (побітове І)	bitand та and_eq
| та |=	побітова диз'юнкція (побітове АБО)	bitor та or_eq
^ та ^=	побітове додавання за модулем 2	xor та xor_eq
<< та <<=	зсув вліво	
>> та >>=	зсув вправо	

Таблиця істинності логічних порозрядних операцій

Таблиця 2.9

E1	**E2**	**E1&E2**	**E1^E2**	**E1|E2**
0011	0101	0001	0110	0111

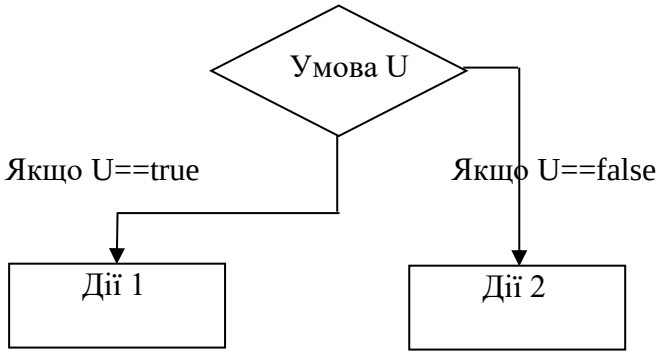
- Порозрядне заперечення ! заміняє змінює в бітовому (двійковому) представленні числа кожну 1 на 0, а 0 на 1.
Приклад: ~(0x9A)=(0x65) тобто ~ (10011010) == (01100101)
- Порозрядна кон'юнкція & (порозрядне І) порівнює послідовно розряд за розрядом два операнди. Для кожного розряду результат рівний 1, якщо тільки два відповідних розряди операндів рівні 1, в інших випадках результат 0.

Приклад: (0x93)&(0x3D)=(0x11) тобто (10010011) & (00111101) == (00010001)
- Порозрядна диз'юнкція | (порозрядне АБО) порівнює послідовно розряд за розрядом два операнди. Для кожного розряду результат рівний 1, якщо хоча б один з відповідних розрядів рівний 1.
Приклад: (0x93)|(0x3D)=(0xBF) тобто (10010011) | (00111101) == (10111111)
- Побітове додавання за модулем 2 порівнює послідовно розряд за розрядом два операнди. Для кожного розряду результат рівний 1, якщо один з двох (але не обидва) відповідних розряди рівні 1.
Приклад: (0x93)^(0x3D)=(0xAE), бо (10010011) ^ (00111101) == (10101110)
На операції побітового додавання за модулем 2 ґрунтується метод обміну значень двох цілочисельних змінних.
a^=b^=a^=b;
Операція зсуву вліво (вправо) переміщує розряди першого операнду вліво (вправо) на число позицій, яке задане другим операндом. Позиції, що звільняються, заповнюються нулями, а розряди, що зсуваються за ліву (праву) границю, втрачаються.
Приклади:

```
unsigned char x =138; //бінарне представлення 138(10001010)
unsigned char y =(x<<2); // це число 40, бо (10001010) << 2 == (00101000)==40
 // перший значущій біт при такий операції видалився.
unsigned char z =(x>>2); // це число 34, бо (10001010) >> 2 == (00100010) =34.
```

Розгалуження

Для написання програм потрібно вміти керувати подіями та операторами програми в залежності від результатів попередніх дій та вхідних даних. Для цього використовуються розгалуження.
Розгалуження будує частину програмного коду, що перевіряє певну умову або булевий вираз та визначає які дії відбуваються, якщо цей вираз дорівнює true, а які дії, якщо він дорівнює false.



На мові С будь-яке значення, що не дорівнює 0 або **null (NULL) вважається true**, а інакше вважається **false**.
Умова хибна, якщо вона дорівнює нулю, в інших випадках (навіть при від'ємних значеннях)вона істинна. *До того ж, умова, що перевіряється, повинна бути скалярною, тобто зводитися до простого значення, яке можливо перевірити на рівність нулю. Взагалі не рекомендується використання змінних*

типу float або double в логічних виразах з використанням рівне та нерівне для перевірки умов з причини недостатньої точності подібних виразів.

На С можна записати розгалуження наступними шляхами

1. Умовна операція ? :

Умовна операція ?: - єдина тернарна операція в мові С. Її синтаксис:

умова ? вираз1: вираз2;

Принцип її роботи такий.

Спочатку обчислюється вираз умови. Якщо цей вираз май ненульове значення, то обчислюється вираз_1. Результатом операції ?: в даному випадку буде значення виразу_1. Якщо вираз умови рівний нулю, то обчислюється вираз_2 і його значення буде результатом операції. В будь-якому випадку обчислюється тільки один із виразів (вираз_1 або вираз_2).

Наприклад, дану операцію зручно використати для знаходження найбільшого з двох чисел x і y: max=(x>y)?x:y;

Приклад 1:

```
#include<stdio.h>
void main(){
int points;
printf("Введіть оцінку [2..5]:");
scanf("%d",&points);
printf("%s",points>3?"Ви добре знаєте матеріал!":"Погано...");
}
```

Приклад 2. Нехай c= 10. Тоді після виконання команди

```
x = (c == 3) ? 2 * c : c - 2;
```

отримаємо $x = 8$, оскільки не дорівнює 3, і тому тут обчислюється значення виразу 2

2. Оператор розгалуження if

Оператор розгалуження призначений для виконання тих або інших дій в залежності від істинності або хибності деякої умови. Основний оператор цього блоку в С - if ... else не має ключового слова then або двокрапки, як в Python, проте обов'язково вимагає, щоб умова, що перевіряється, розміщується у круглих дужках. Оператор, що слідує за логічним виразом є оператором (then- частиною) оператору if...else.

Синтаксис оператора:

if (<умова>) {<оператор1>; …;<операторN>;}
[else {<оператор2;> …, <операторM;>]

У випадку, коли після умови чи ключового слова else слідує лише один оператор можна його не оточувати фігурними дужками, але більшість керівництв зі стилю програмного коду радять робити це завжди, незалежно від кількості операторів для зручності змін в коді та єдиного стилю запису розгалуження.

Приклад 1.

```
/* програма виводить результат ділення двох дійсних чисел */
#include<stdio.h>
int main(){
 float a,b,c;
 printf("Введіть число a: "); scanf("%f",&a);
 printf("Введіть число b: "); scanf("%f",&b);
 if (b==0) printf("Ділення на нуль ! ");
 else {
 c=a/b;
 printf("a: b == %g",c);
 };
}
```

Приклад 2.

```
/* застосування умовного розгалужування */
#include <stdio.h>
int main() {
 int number;
 int ok;
 printf("Введіть число з інтервалу 1..100: ");
 scanf("%d",&number);
 ok=(1<=number) && (number<=100);
 if (!ok)  printf("Не коректно !! ");
 return ok;
}
```

Змінній ok присвоюється значення результату виразу: ненульове значення, якщо істина, і в протилежному випадку - нуль. Умовний оператор if(!ok) перевіряє, якщо ok дорівнюватиме нулю, то !ok дасть позитивний результат й відтоді буде отримано повідомлення про некоректність, виходячи з контексту наведеного прикладу.

Приклад 3.Нехай x= 9. Унаслідок виконання команд

```
if(x > 7)  y = pow(x, 2);
else y = sqrt(x);
if(x <= 5)
        z= exp(x);
else z= ++x;
```

Отримаємо:

y=81, *z*= 10, *x*=10.

Приклад.

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

```
#include <math.h>
int main(){
 float C=1.231;
 float x,y;
 printf("\nx="); scanf_s(“%f”,&x);
 printf("\ny="); scanf_s(“%f”,&x);
 double A;
 if(x<=y){
            A=x*y-C*y*sqrt(y);
 }
 else{
        A=cos(x)+log(y);
 }
 printf("A=%lf\n", A);
 printf("\nx="); scanf_s(“%f”,&x);
 if(x<=y){A=x*y-C*y*sqrt(y);}
else{A=cos(x)+log(y);}
 printf("A=%lf\n", A);
 getchar();
 return 0;
}
```

Результати обчислень:

```
x=1
y=1
A=-0.231
x=2
A=-0.416147
```

3. Оператор switch

switch(<вираз цілого типу>)
{
case <значення_1>: <послідовність_операторів_1>; break;
case <значення_2>: <послідовність_операторів_2>; break;
.....
case <значення_n>: <послідовність_операторів_n>; break;
[default: <послідовність_операторів_n+1>;]
}

Оператор-перемикач switch призначений для вибору одного з декількох альтернативних шляхів виконання програми. Виконання оператора switch починається з обчислення значення виразу (виразу, що слідує за ключовим словом switch у круглих дужках). Після цього управління передається одному з

<операторів>. Оператор, що отримав управління - це той оператор, значення константи варіанту якого співпадає зі значенням виразу перемикача.
Вітка default (може опускатися, про що свідчить наявність квадратних дужок) означає, що якщо жодна з вищенаведених умов не задовольнятиметься (тобто вираз цілого типу не дорівнює жодному із значень, що позначені у case-фрагментах), керування передається по замовчуванню в це місце програми. Треба також зазначити ***обов'язкове застосування оператора break у кожному з case-фрагментів*** (цей оператор застосовують для негайного припинення виконання операторів while, do, for, switch), що негайно передасть керування у точку програми, що слідує відразу за останнім оператором у switch-блоці.
Приклад 1:

```
switch(i){
case -1: n++; break;
case 0: z++; break;
case 1: p++; break;
}
```

Приклад 2 :

```
switch(c){
case 'A': capa++;
case 'a': lettera++;
default: total++;
}
```

В останньому прикладі всі три оператори в тілі оператора switch будуть виконані, якщо значення с рівне 'A', далі оператори виконуються в порядку їх слідування в тілі, так як відсутні оператори break.
Приклад. У п’ятиповерховому будинку на кожному поверсі по чотири квартири. Скласти програму для визначення поверху в залежності від номера квартири.

```
#include <stdio.h>
#include <math.h>
int main(){
  short int n;
 printf("Введіть № квартири: ");
  scanf_s(“%d”,&n);
  switch(n){
    case 1:
    case 2:
    case 3:
    case 4: printf (1-й поверх");
       break;
    case 5:
    case 6:
```

```
   case 7:
   case 8: printf (2-й поверх");;
      break;
   case 9:
   case 10:
   case 11:
   case 12: printf (3-й поверх");;
      break;
   case 13:
   case 14:
   case 15:
   case 16: printf (4-й поверх");;
      break;
   case 17:
  case 18:
  case 19:
  case 20: printf (5-й поверх");;
     break;
  default: printf ("помилковий № квартири");
 }
}
```

Слід відмітити, що розгалуження можна вкладати одне до одного, та будувати послідовні блоки розгалужень.

Приклади:

Обчислення сігнуму (**послідовне** розгалуження):

```
if(x>0){
        signum =1;
}
else if (x==0){
        signum=0;
}
else{ signum=-1;
}
```

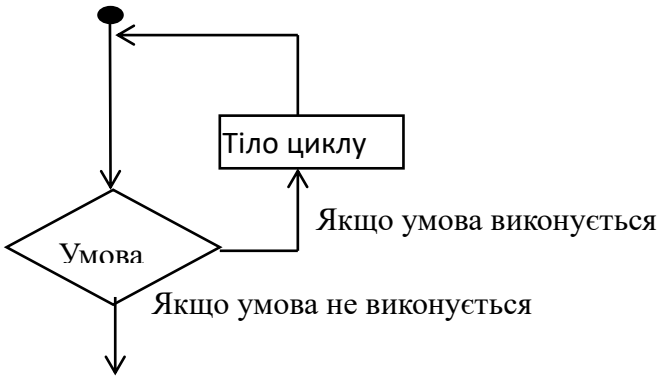
Вкладене розгалуження:

```
if(x>0){
        if(y>0){
             z= sqrt(x)+sqrt(y);
        }
        else{
                z =1;
```

```
        }
}
else{
        z=-1;
}
```

Цикли

Для виконання послідовних обчислень потрібно вміти вказувати команди, які виконуються доки не справджується певна умова виходу з циклу обчислень.



В мові С існують декілька варіантів написання циклів:

- ***Цикл з передумовою (while loop)***
- ***Цикл з лічильником (for loop)***
- ***Цикл з післямовою (do...while loop)***
- ***Вкладені цикли (nested loops)***

Цикл з передумовою

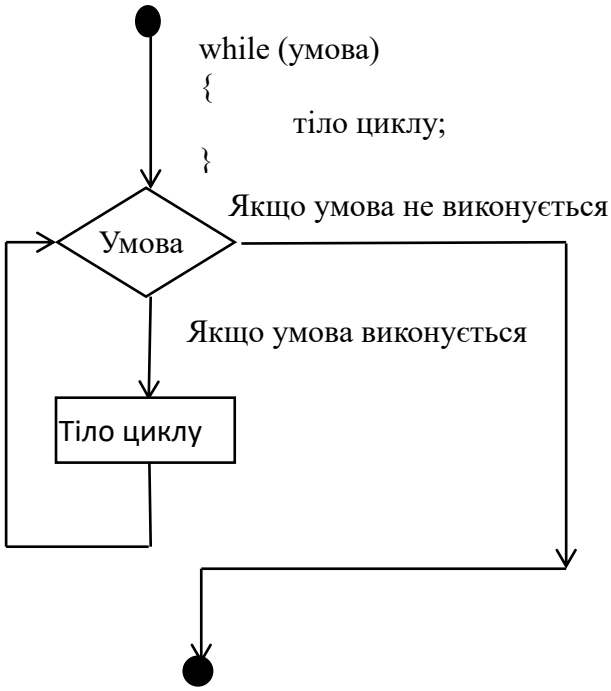
Цикл з передумовою (while loop) виконує певний ланцюжок обчислень – тіло циклу, поки виконується певна умова.

Синтаксис цього циклу наступний

```
while((<умова> condition) {
  <дії>statement(s);
}
```

тут **<дії>statement(s)** може бути однією командою або серією команд. Умова(**condition)** може бути виразом або булевою змінною так само як і умова розгалуження. Тіло циклу виконується поки умова й істинною, тобто дорівнюй true (або вираз умови не дорівнюй 0 або NULL).

Як тільки умова стає false, програма передай керування у місце, що слідує за дужками що обмежують тіло циклу.



Важливою властивістю цього циклу є те, що якщо з самого початку умова в циклі є хибною, то тіло циклу не виконується жодного разу.

Приклад

```
#include <stdio.h>
int main () {
  /* local variable definition */
  int a = 10;
  /* while loop execution */
  while( a < 20 ) {
    printf("value of a: %d\n", a);
    a++;
  }

}
```

Виконання коду дає наступний результат:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Цикл з післяумовою

На відміну від циклів **for** та **while**, які перевіряють умову перед виконанням циклу, цикл з післяумовою **do...while** виконує перевірку умови після проходження тіла циклу.

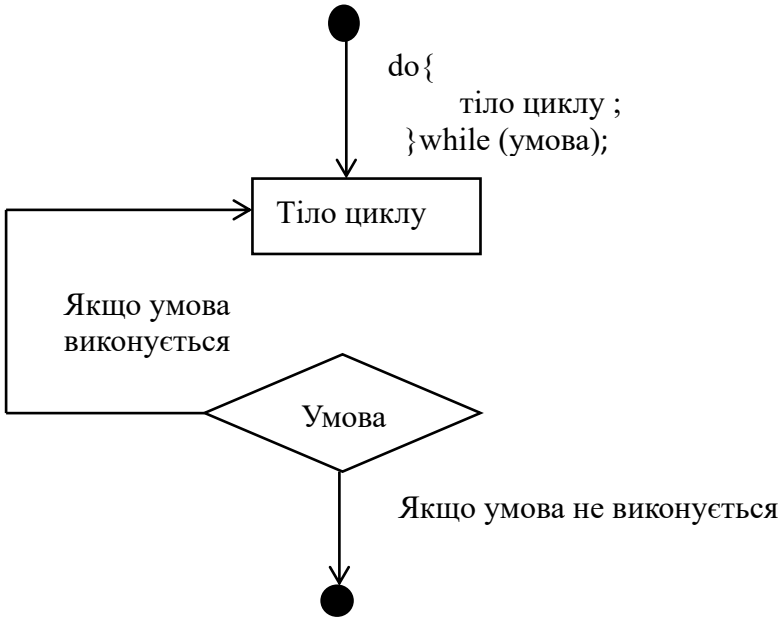
do...while схожий за структурою виконання на цикл з передумовою крім того, що він гарантує виконання тіла циклу хоча б один раз.

Синтаксис **do...while** циклу

```
do {
  <дії>statement(s);
} while(<умова> condition );
```

Помітимо, що спочатку відбувається виконання ланцюгу команд тіла циклу, а потім перевірка умови.

Якщо умова циклу виконується, тобто дорівнюй true, потік виконання передається в початок тіла циклу та виконання циклу починається з початку тіла циклу. Цей процес продовжується поки умова циклу не стає хибною, тобто рівною false.



Приклад.

```
#include <stdio.h>
int main () {
  /* визначення локальної змінної */
  int a = 10;
  /* виконується цикл */
  do {
    printf("value of a: %d\n", a);
    a = a + 1;
  }while( a < 20 );
```

```
}
```
Результат виконання:
```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Цикл з лічильником

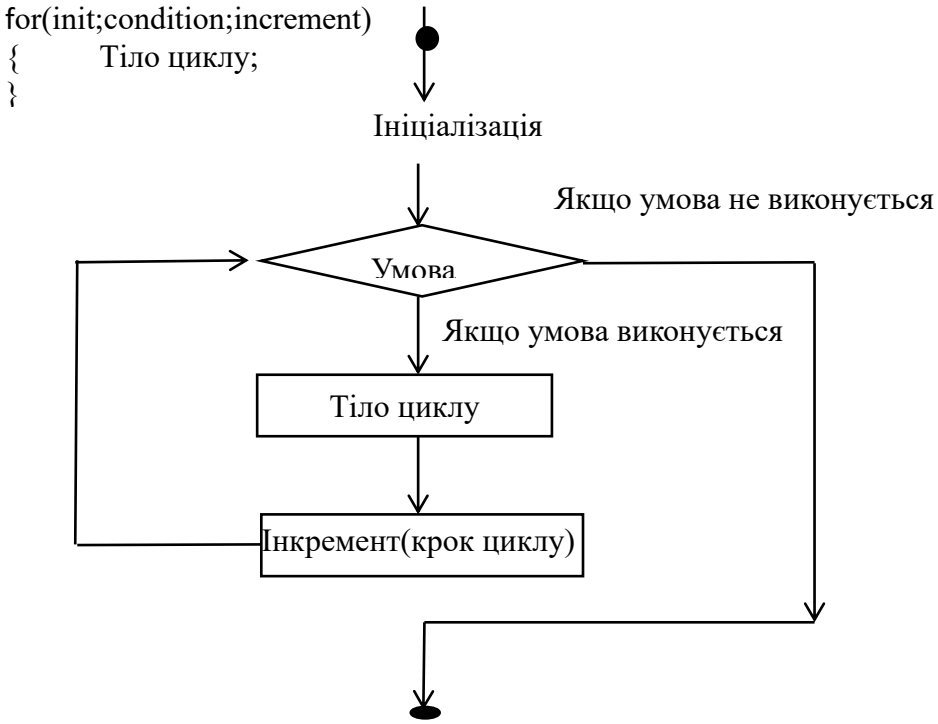
Цикл з лічильником або цикл **for** (**for** loop) – це цикл, що дозволяє ефективно виконувати ланцюжок команд, які виконуються фіксовану кількість разів.

Синтаксис звичайного циклу **for** наступний:

```
for ( <ініціалізація(init)>; <умова(condition)>;<інкеремент(increment)> ) {
  statement(s);
}
```

Потік виконання циклу наступний:

- Крок ініціалізації **init** виконується спочатку та лише один раз. На цьому кроці ініціалізуються змінні (лічильники), що контролюють виконання циклу. Можливі варіанти, коли цей крок порожній тобто нічого не виконується.
- Далі, умова (**condition)** підраховується. Якщо вона істинна (true), тіло циклу виконується. Якщо воно хибне (false), тіло циклу не виконується і потік виконання йде на команду що йде після тіла циклу 'for' (за фігурними дужками).
- Виконується тіло циклу
- Після виконання тіла циклу 'for' , потік виконання йде до виразу інкременту(**increment)**. Команди цього виразу дозволяють змінювати лічильники циклу.
- Умова виконується знову. Якщо вона є істиною (true), то тіло циклу виконується знову (тіло циклу, крок increment , знову умова). Після того, як умова становиться false, цикл 'for' завершує роботу.



Приклад

```
#include <stdio.h>
int main () {
  int a;
  /* виконання циклу for */
  for( a = 10; a < 20; a = a + 1 ){
    printf("value of a: %d\n", a);
  }
}
```

Результат виконання

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Нескінчений цикл

Цикл може стати нескінченим якщо умова циклу ніколи не стане false.
Це можна зробити наступними шляхами:

1) while(1){ ….}
2) do{…} while(1>0);

3) Часто роблять нескінчений цикл за допомогою **for:**

```
#include <stdio.h>
int main () {
  for( ; ; ) {
    printf("This loop will run forever.\n");
  }
}
```

Якщо умова циклу відсутня, вона вважаться, рівною істині .

Примітка: виконання нескінченого циклу зупиняється за допомогою Ctrl + C або (Ctrl+<Pause/Break>).

Керування циклом в тілі циклу

Іноді виникає потреба виходу з циклу посеред виконання ланцюгу команд тіла циклу. Зокрема, якщо умова циклу передбачає нескінчений цикл, а вийти з циклу все ж такі треба. На С існують наступні варіанти керування циклом:

- Команда **break**. Завершує **цикл** або вираз **switch** та передає виконання на місце, що йде після циклу.
- Команда **continue.** Завершує виконання тіла циклу та передає керування на умову циклу (при while, або do..while) або в крок зміни лічильника (increment) в циклі for.
- Команда **goto**. Передає керування на мітку(label), що вказана в програмі. ***Примітка. Сучасні керівництва дуже радять не використовувати цю команду ніколи.***

Примітка*: коли оператор керування залишає тіло циклу всі автоматично створені в тілі циклу змінні знищуються.*

Оператор слідування (кома)

Оператор "кома" (,) називається оператором слідування, яка "зв'язує" два довільних вирази. Список виразів, розділених між собою комами, обчислюються зліва направо.

Наприклад, фрагмент тексту:

```
a=4; b=a+5;
```

можна записати так:

```
a=4, b=b+5;
```

Операція слідування використовується в основному в операторах циклу for().

Для порівняння наводимо приклад з використанням операції слідування (приклад 2) та без неї (приклад 1):

Приклад 1:

```
int a[10],sum,i;
/* ... */
sum=a[0];
for (i=1;i<10;i++)
sum+=a[i];
```

Приклад 2:

```
int a[10],sum,i;
/* ... */
for (i=1,sum=a[0];i<10;sum+=a[i],i++) ;
```

Вкладені цикли

Як і розгалуження, цикли можна вкладати один в одний.

Синтаксис вкладеного циклу з лічильником **nested for loop:**

```
for ( <init1>; <condition1>; <increment1> ) {

  for ( <init2>; <condition2>; <increment2> ) {
    statement(s);
  }
  statement(s);
}
```

Синтаксис вкладеного циклу з передумовою **nested while loop**:

```
while(condition) {

  while(condition) {
    statement(s);
  }
  statement(s);
}
```

Синтаксис вкладеного циклу з післяумовою **nested do...while loop**:

```
do {
  statement(s);

  do {
    statement(s);
  }while( condition );

}while( condition );
```

Помітимо, що й цикли одного виду можна вкладати в цикли іншого та це вкладання можна робити декілька разів (хоча існує стилістичне обмеження таких вкладень – не більше чотирьох)

Приклад

Наступна програма рахує прості числа від 2 до 35 −

```
#include <stdio.h>

int main () {
  /* визначення локальних змінних */
  int i, j;
```

```
    for(i = 2; i<35; i++) {

        for(j = 2; j*j <= i; j++)
                if(!(i%j)) break; // якщо є дільник, то число - не просте
        if(j > (i/j)) printf("%d is prime\n", i);
  }

  return 0;
}
```

Результат:

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
```

3. Масиви та вказівники. Рядки С

Масиви в С. Декларація масивів, ініціалізація масивів. Багатоіндексні масиви Вказівник. Операції над вказівниками. Бібліотека stddef.h. Зв'язок вказівників та масивів. Безрозмірні масиви.
Робота з пам'яттю. Функції з stdlib.h.
Символьний тип та масиви символів.
Рядковий тип. Введення, виведення рядку. Функції для роботи з рядком (string.h, stdlib.h). Бібліотека ctype.h. Перетворення числових типів до рядку та навпаки

Масиви в С

Масив – впорядкований набір фіксованої кількості однотипних елементів, що зберігаються в послідовно розташованих комірках оперативної пам'яті, мають порядковий номер і спільне ім'я, що надає користувач. Масиви в програмуванні подібні до векторів або матриць в математиці.

Якщо простіше, то масив це сукупність однотипних змінних з одним іменем. Він дозволяє зберігати десятки і сотні елементів під одним іменем.

Властивості масиву в С:

- всі елементи мають одне ім'я і один тип даних;
- кожен масив має фіксований розмір – кількість елементів у масиві.
- кожен елемент масиву має власний індекс, по цьому індексу і відбувається звернення до елементів;

- нумерація індексів починається з 0.

Створення масиву
Загальний синтаксис створення масиву:
<тип масиву> ім'я [розмір];

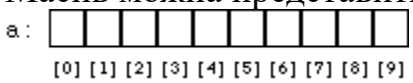
<тип масиву>- будь-який з типів (наприклад: int, bool, char, double) - визначає елементи якого типу, що будуть зберігатися у масиві.
ім'я – аналогічно до змінних та назв функцій – будь-який ідентифікатор. По цьому імені будуть створюватися звернення до окремих елементів.
розмір – натуральне число, що вказує кількість елементів масиву
Приклад:

```
int a [100];
bool apple[1000];
```

При цьому:
кількість елементів завжди є натуральним числом або константною змінною;
замість кількості елементів не можна писати змінні (хоча деякі компілятори це дозволяють).
Таким чином, вираз

```
int a[10];
```

оголошує масив, названий a, що складається з десяти елементів, кожен з яких має тип int. Простіше кажучи, масив є змінною, яка може містити більше одного значення і для того, щоб вказати значення на яке треба звернутись, потрібно використовувати числовий індекс.
Масив можна представити наступним чином:



У С масиви рахуються, починаючи з нуля: десять елементів 10-елементного масиву нумеруються від 0 до 9. Нижній індекс, який вказує єдиний елемент масиву - це просто ціле число у квадратних дужках. Першим елементом масиву є a[0], другий елемент - a[1] і т.д. Можна використовувати ці вирази з індексами масиву всюди в коді, де можна використовувати ім'я простої змінної, наприклад:

```
a[0] = 10;
a[1] = 20;
a[2] = a[0] + a[1];
```

Зауважте, що посилання на індексні масиви (тобто вирази, такі як a[0] і a[1]) можуть з'являтися на будь-якій стороні оператора присвоєння. Індекс не повинен бути константою, як 0 або 1; це може бути будь-який інтегральний вираз. наприклад, звичайний цикл над усіма елементами масиву:
Приклад. Цей цикл встановлює всі десять елементів масиву a до 0.

```
int i;
for(i = 0; i < 10; i++)
a[i] = 0;
```

Декларація масиву

Таким чином, наступні варіанти декларації масиву допустимі в сучасному Ci(Ci++):

1) Декларація масиву сталої довжини:

Тип <ім'я_масиву> [розмір];

Приклад:

```
double ar2[5];
```

2) Використання довжини ініціалізованою макросом #define:

#define РОЗМІР розмір

Тип <ім'я_масиву> [РОЗМІР];

Приклад:

```
#define NMAX 25
int ar2[NMAX];
```

3) Використання довжини ініціалізованою за допомогою константної натуральної(або цілої) змінної:

const <цілий тип> <РОЗМІР> = розмір;

Тип <ім'я масиву> [РОЗМІР];

Приклад:

```
const size_t N = 10;
unsigned ar3[N];
```

4) (Хак) Ще одним варіантом є визначення константи кількості елементів масиву за допомогою перерахування (enum):

enum { **<РОЗМІР> = розмір** };

Тип <ім'я масиву> [РОЗМІР];

Приклад:

```
enum{ N = 10};
unsigned ar4[N];
```

5) Допустима також форма створення безрозмірного масиву вигляду:

Тип <ім'я масиву> [];

Приклад:

```
char ar5[];
```

Але якщо ми далі без деяких попередніх дій будемо звертатись до нього, наприклад, ar4[0] = 'A'; то програма може аварійно завершити роботу (що робити для того, щоб цього не було ми розглянемо трохи пізніше) .

Обмеження масивів на Сі

Масиви - це зручне рішення для багатьох проблем, але багато чого не зробить С з ними автоматично (так як наприклад, numpy в мові Python). Зокрема, не можна одночасно встановлювати всі елементи масиву і не призначати один масив іншому; обидва присвоєння

```
int a[10];
a = 0;              /* Помилка */
```

та

```
int b[10];
b = a;            /* Помилка */
```

некоректні (хоча другий приклад може зкомпілюватись).

Оскільки замість кількості елементів не можна писати змінні (хоча деякі компілятори це дозволяють), то такий код є також не зовсім правильним для всіх платформ:

```
#include <stdio.h>
int main (){
int n=10;
float arr[n];
}
```

Але це правило не поширюється на константи, через те, що значення константи є незмінним. Цілком можливо таке, що розмір ви будете вказувати декілька разів (заповнення масиву, обробка та інші дії) та може знадобитися замінити всі розміри масиву. Якщо всюди вказувати його у вигляді числа, тоді при заміні ви будете змушені замінювати його всюди де він був прописаний, але якщо ви використовували константу, тоді, щоб замінити всі ці розміри достатньо переписати один рядок коду.

Такий код правильний:

Приклад.

```
#include <stdio.h>
int main (){
const int size = 1000;// тип int
double arr[size]; // тип double
}
```

Зауважимо, що з точки зору сучасного програмування для розміру масиву краще брати константну натуральну змінну типу size_t, що створено спеціально для зберігання розмірів масивів та рядків (цей тип визначений в stddef.h, але визначиться й при підключенні стандартного stdio.h чи iostream.h в Сі++).

Приклад

```
#include <stdio.h>
int main (){
```

```
const size_t size = 1000;  /* тип компілятор визначить оптимальним чином (більшість сучасних платформ визначить long unsigned)*/
double arr[size]; // масив типу double
return 0;
}
```

Крім цього, варто зазначити, що компілятори C та C++ не перевіряють вашу програму на таку помилку, як вихід за межі масиву.

Приклад (*поганий*):

```
int a[6]; printf(a[6]);
```

Ця програма зкомпілюється, але завершиться помилкою при виконанні, оскільки у масиві є 6 елементів, то нумерація буде від [0] до [5], а у наступній команді (printf) ми звертаємося до елемента з індексом [6], якого немає. такі помилки називаються помилками на етапі виконання (runtime error).

Заповнення даних: ініціалізація масивів

Основне завдання масивів – зберігання даних, тому часто потрібно відразу їх заповнити - ініціалізувати. Ініціалізувати масиви можна декілька способами.

1 спосіб: заповнення масиву зразу ж після його створення.

Синтаксис такий:

тип_даних ім'я_масиву[к-сть_елементів] = {значення_1, значення_2, … , значення_к-сть_ел-тів};}

Приклад:

```
char str[5] = {'1', 'Q', '!', '$', '@'};
```

Крім цього, при такій ініціалізації дозволяється не вказувати кількість елементів (безрозмірний масив).

```
int abr[] = {1, 15, 876};
```

Якщо ви допустите помилку, як у цьому прикладі

```
double abr[5] = {1};
```

тобто розмір більший за ініціалізовані елементи, тоді наступні елементи після 1 будуть заповнені нулями. Тоді вираз буде мати такий вигляд:

```
double abr[5] = {1,0,0,0,0};
```

У іншому випадку, коли розмір менший за кількість заповнених елементів, виникне помилка на етапі компіляції.

2 спосіб: заповнення масиву за допомогою циклу.

В даному випадку заповнення буде відбуватися з клавіатури. Приклад відповідної програми:

```
#include <stdio.h>
int main (){
const size_t size = 25; /* створили константу - розмір. Якщо ми б її не створили, то в кожному з циклів ми були б змушені писати 25, що не лише непотрібно, але й погано з точки зору зрозумілості та гнучкості коду. У даному випадку для зміни розміру масивів потрібно переписати лише один рядок коду .*/
```

```
double arr[size] = {5}; /* Далі ми заповнили масив за допомогою першого способу */
for (size_t i=0; i<size; ++i){
        printf("arr[%lu]=%lf \n",i,arr[i]);
} /* елементи масиву наступні: 5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 (24 нулі). */
for (size_t i=0; i<size; ++i){
  scanf(&arr[i]); /* далі за допомогою циклу ми виводимо всі елементи масиву*/
}
/* Таким чином, елементи можна виводити на екран. по суті на даному моменті ми можемо вважати елементи масиву як змінні, а змінні можна виводити. */
for (int i=0; i<size; ++i){
        printf("arr[%lu]=%lf,\t" ,i,arr[i]);
}
/* ми вводимо значення елементів у циклі. також варто замітити, що ми вводимо і елемент з індексом 0, який до цього був рівний 5. тобто значення елементів можна змінювати.*/
getchar();
}
```

Розташування масивів у пам’яті та багатоіндексні масиви

Найменшою одиницею пам’яті є біт, але при роботі з адресами в комп’ютері, найменшою одиницею вважається байт. В кожному 1 байті міститься 8 біт. Елементи масиву у пам’яті розташовуються у порядку їх зростання, елемент за елементом. крім цього, кожний елемент має власний розмір, який дорівнює типу масиву. Якщо масив типу int з кількістю елементів N, то кожен елемент буде займати 4 байти. а розмір всього масиву буде дорівнювати 4*N.

Насправді типом масиву може бути не лише базові типи, а й будь-які типи, що може створити користувач. Зокрема, можна створити масив, кожний з яких має тип x, де x - будь-який тип, наприклад, можна створити масив, кожним з елементів якого є інший масив. Ми можемо використовувати ці масиви масивів для тих самих видів завдань, як ми використовуємо багатовимірні масиви в інших комп'ютерних мовах (або матрицях з математики). Природно, ми не обмежуємося масивами масивів; ми могли б мати масив масивів масивів, які діяли б як тривимірний масив і т.д.

Саме таким чином мова програмування С дозволяє створювати та працювати з багатовимірними масивами. Ось загальна форма оголошення багатовимірного масиву:

<ім'я типу> <ім’я масиву> [size1][size2]...[sizeN];

Наприклад, наступне оголошення створює тривимірний масив цілих чисел:

```
int threedim [5][10][4];
```

Такі декларації потрібно читати "зправа-наліво". Наприклад, int a2[5][7]; - - це масив з 5 чогось, і що кожне з цих чогось є масивом з 7 int. Більш коротко, a2 - це масив розміру п’ять складений з масивів сімох int, або a2 - це масив масиву int. Тобто a2 це масив з 5 масивів розміром 7, а не навпаки. Можна також інтерпретувати, що a2 має 5 рядків і 7 стовпців, хоча ця інтерпретація не є обов'язковою. Також можна розглядати "перший" або внутрішній індекс як "x", а другий як "y". Доступ до масиву збігається з тим, що використовується згідно цієї логіки, як у прикладах нижче.

Двовимірні масиви

Найпростішою формою багатовимірного масиву є двовимірний масив. Двовимірний масив, по суті, є списком одновимірних масивів. Щоб оголосити двовимірний цілочисельний масив розміром [x][y], ви повинні написати щось наступне -

type arrayName [x] [y];

де тип type може бути будь-яким дійсним типом даних C і arrayName буде дійсним C ідентифікатором. Двовимірний масив можна розглядати як таблицю, яка матиме x кількість рядків і y кількість стовпців. Двовимірний масив a, який містить три рядки і чотири стовпці, може бути показаний наступним чином -

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Таким чином, кожен елемент у масиві a ідентифікується назвою елемента форми a [i] [j], де 'a' є назвою масиву, a 'i' і 'j' є індексами, які однозначно ідентифікують. кожен елемент у "a".

Ініціалізація двовимірних масивів

Багатовимірні масиви можуть бути ініціалізовані шляхом запису значень в фігурних дужках для кожного рядка. Нижче наведено масив з 3 рядками де кожен рядок має 4 колонки.

```
int a[3][4] = {
{0, 1, 2, 3} ,   /*  рядок з індексом 0 */
{4, 5, 6, 7} ,   /*  рядок з індексом 1 */
{8, 9, 10, 11}   /*  рядок з індексом 2 */
};
```

Вкладені фігурні дужки, які вказують потрібний рядок, є необов'язковими (хоча з точки зору стилю коду гарними). Наступна ініціалізація еквівалентна попередньому прикладу, хоча менш зрозуміла для людини, що буде читати код:

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Доступ до елемента в двовимірному масиві здійснюється за допомогою індексів, тобто індексу рядків і індексу стовпців масиву. Наприклад

```
int val = a[2][3];
```

Вищенаведена інструкція візьме 4-й елемент з 3-го рядка масиву. Ви можете перевірити його значення на малюнку вище. Давайте перевіримо наступну програму, де ми використовували вкладений цикл для обробки двовимірного масиву

Приклад

```
#include <stdio.h>
int main () {   /* масив з 5 рядками та 2 колонками*/
int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
int i, j;    /* введення масиву */
for ( i = 0; i < 5; i++ ) {
        for ( j = 0; j < 2; j++ ) {
                printf("a[%d][%d] = %d\n", i,j, a[i][j] );     }
        }
}
```

Цей код виведе наступний результат:

```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8
```

Щоб проілюструвати використання багатовимірних масивів, ми можемо заповнити елементи вищезгаданого масиву a2 за допомогою цього фрагмента коду:

```
int i, j;
for(i = 0; i < 5; i = i + 1){
        for(j = 0; j < 7; j =j++)
                a2[i][j] = 10 * i + j;
}
```

Ця пара вкладених циклів встановлює a[1] [2] - 12, a[4] [1] - 41 і т.д. оскільки перший розмір a2 дорівнює 5, перша індексна змінна індексу, i, виконується від 0 до 4 аналогічно, другий індекс змінюється від 0 до 6.

ми можемо вивести масив a2 (двовимірним способом, враховуючи його структуру) з подібною парою вкладених циклів:

```
for(i = 0; i < 5; i = i + 1){
  for(j = 0; j < 7; j = j + 1)
     printf("%d\t", a2[i][j]);
  printf("\n");
```

```
}
```
Друк цих елементів:
```
for(j = 0; j < 7; j = j + 1)
    printf("\t%d:", j);
printf("\n");
for(i = 0; i < 5; i = i + 1){
  printf("%d:", i);
  for(j = 0; j < 7; j = j + 1)
    printf("\t%d", a2[i][j]);
  printf("\n");
}
```
Результат:
```
0:    1:    2:    3:    4:    5:    6:
0:    0     1     2     3     4     5     6
1:    10    11    12    13    14    15    16
2:    20    21    22    23    24    25    26
3:    30    31    32    33    34    35    36
4:    40    41    42    43    44    45    46
```
Як пояснювалося вище, ви можете мати масиви з будь-якою кількістю розмірів, хоча ймовірно, що більшість масивів, які ви створюєте, будуть мати розміри один або два.

Вказівники

В C та C++ існує надзвичайно потужний інструмент для роботи зі складними агрегатами даних, який надає загальний підхід до різних на перший погляд програмних об'єктів, таких як масив та рядок. Цей інструмент дозволяє працювати з ділянками пам'яті пристрою та створювати програмні об'єкти великого розміру та зветься вказівник. Коректна робота з вказівником дозволяє швидко керувати пам'яттю пристрою, але, разом з тим, неакуратна робота може привести до значних проблем в коді та при виконанні програми, тому потрібно добре розуміти їх та використовувати акуратно.

Поняття вказівника

Кожна змінна у програмі - це об'єкт, який має ім'я та значення. Після визначення змінної з ініціалізацією всі звернення у програмі до неї за іменем замінюються компілятором на адресу іменованої області оперативної пам'яті, в якій зберігається значення змінної (рис. 1). Програміст може визначити власні змінні для збереження *адрес областей пам'яті.* Такі змінні називають **вказівниками**.

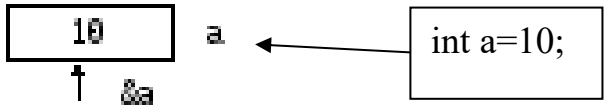


Рис. 1.

Нагадаємо, що кожна змінна в програмі – це об'єкт, що має ім'я та значення. До цього значення можна звернутися та отримати його по імені змінної. Ця змінна,

в свою чергу, десь міститься в пам’яті комп’ютера і адреса пам’яті цієї змінної теж має певне значення (зазвичай довге беззнакове ціле). Вказівник – це змінна, значення якої є адресою (тобто місцем в пам'яті) певній змінній. Іншими словами, вказівник - це символічне представлення адреси змінної в пам’яті.

В С та С++ тип вказівник(інколи звуть покажчиком) може визначити адресу змінної або об'єкта будь-якого типу. Як і будь-яка змінна або константа, потрібно оголосити вказівник, перш ніж використовувати його для зберігання будь-якої адреси змінної. Загальна форма декларації змінної вказівника:

<тип> *< ідентифікатор>;

або

<тип> *< ідентифікатор> = <початкове значення>;

Ще один із способів завдання вказівника є надання безрозмірного масиву вигляду:

<тип> < ідентифікатор> [ ];

Тут **<тип>** є базовим типом вказівника; він повинен бути типом даних С, а **ідентифікатор** - ім'я змінної вказівника. Зірочка *, що використовується для оголошення вказівника, є тією ж зірочкою, яка використовується для множення. Однак у цьому контексті зірочка використовується для позначення змінної як вказівника. Ось деякі правильні декларації вказівника

- `int *ip; /* вказівник на ціле */`
- `double *dp; /* вказівник на подвійне дійсне число */`
- `float* fp; /* вказівник на одинарне дійсне число */`
- `char * ch /* вказівник на символ */`

Фактичний тип даних значення всіх вказівників, будь то цілочисельний, дійсний, символьний або інший, є однаковим - це довге шістнадцяткове число, яке представляє адресу пам'яті. Єдиною відмінністю між вказівниками різних типів даних є тип даних змінної або константи, на яку вказує вказівник.

Приклад:

char ch;	Тут ch- значення символьного типу,
char *cptr;	cptr - вказівник на значення символьного типу,
int val,*ivptr, n;	val і n - значення цілого типу, а *ivptr - вказівник на значення цілого типу;
double r,*rp;	r- змінна дійсного типу подвійної точності, а *rp - вказівник на змінну такого ж типу.

Операції над вказівниками

Нехай змінна типу вказівник має ім'я ptr, тоді в якості значення їй можна присвоїти адресу за допомогою наступного оператора:

ptr=&vr;

В мовах С та С++ при роботі з вказівниками велике значення має операція непрямої адресації (розіменування) *. Операція * дозволяє звертатися до змінної

не напряму, а через вказівник, який містить адресу цієї змінної. Ця операція є унарною та має асоціативність зліва направо. Цю операцію не слід плутати з бінарною операцією множення.
Виведення значення вказівника можна робити як виведення рядка або цілого числа, але за стандартом це робиться за допомогою специфікатора "%p":
printf("Address %p\n", ptr);

Існують наступні операції з вказівниками:

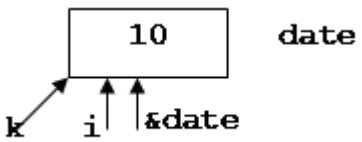
- декларація змінної вказівника;
- присвоєння значення або ініціалізація;
- присвоєння адреси змінній вказівника (оператор &);
- отримання значення (розіменування) за адресою, доступною в вказівнику;
- арифметичні операції:
- інкремент вказівника та додавання цілого числа до вказівника;
- декремент вказівника та віднімання цілого числа від вказівника;
- порівняння вказівників.

Декларація змінної вказівника:

Вказівник - це змінна, значенням якої служить адреса об'єкта конкретного типу. Щоб визначити вказівник треба повідомити, на об'єкт якого типу посилається цей вказівник.

```
char *z;
int *k,*i;
float *f;
```

Оператор присвоювання (=) виконує зворотну дію: імені змінної ставиться у відповідність значення.
Приклад:




```
int date = 10;
int *i, *k, *z;
i = &date;
k = i;
z = NULL;
```

Вказівник на NULL (нульовий вказівник)

Завжди є гарною практикою присвоювати значення змінній NULL у випадку, якщо ви не маєте точної адреси. Це робиться під час оголошення змінної. Вказівник, призначений NULL, називається нульовим вказівником.
В мові С нульова адреса позначається константою NULL, що визначена в заголовному файлі stdio.h.
Примітка. В сучасному С++ радять користуватися замість NULL літералом nullpointer.

Вказівник NULL - це константа зі значенням нуля, визначеним у декількох стандартних бібліотеках. Розгляньте наступну програму -

```
#include <stdio.h>
int main () {
int  *ptr = NULL;
printf("The value of ptr is : %x\n", ptr  );
return 0;
}
```

Результат:

```
The value of ptr is 0
```

У більшості операційних систем програми не мають доступу до пам'яті на адресу 0, оскільки ця пам'ять зарезервована операційною системою. Однак адреса пам'яті 0 має особливе значення; вона сигналізує, що вказівник не призначений для вказування на доступну пам'ять. Але за умовами, якщо вказівник містить нульове (нульове) значення, передбачається, що він не вказує ні на що.

Щоб перевірити, чи є вказівник нульовим, можна використовувати оператор "if" наступним чином -

```
if(ptr)     /* успішний коли p не нульовий p==NULL */
if(!ptr)    /* успішний коли p нульовий p!=NULL */
```

Присвоєння адреси змінній вказівника

Також є **операція визначення адреси - &**, за допомогою якої визначається адреса комірки пам’яті, що містить задану змінну. наприклад, якщо vr - ім’я змінної, то &vr – адреса цієї змінної.

Операція розіменування (*). Операндом цієї операції завжди є вказівник. Результат операції - це той об'єкт, що адресує вказівник – операнд.

Тобто, якщо ptr – вказівник, тоді *ptr — це значення змінної, на яку вказує ptr.

Приклад. Наступний приклад використовує ці операції:

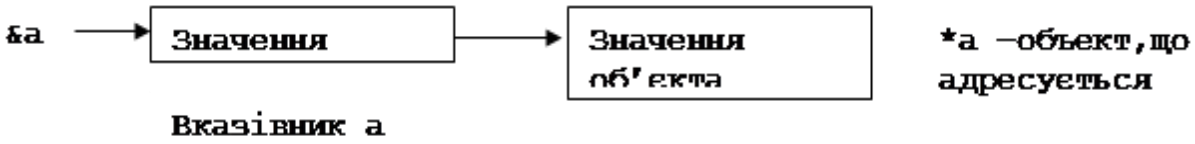
```
#include <stdio.h>
int main () {
int  var = 20;   /* ініціалізація цілої змінної var */
int  *ip;        /* декларація вказівника на ціле число*/
ip = &var;  /* зберігаємо у вказівнику адресу var */
//printf("Address of var variable: %x\n", &var  можна виводити як ціле %x але буде попередження компілятора:  (not int*)
printf("Address of var variable: %p\n", &var  ); //специфікатор адреси %p
printf("Address stored in ip variable: %p\n", ip ); // виводимо значення вказівника
printf("Value of *ip variable: %d\n", *ip ) /* виводимо значення цілої змінної за адресом вказівника */
}
```

Результат роботи коду:

```
Address of var variable: 0x7ffc085bfe18
Address stored in ip variable: 0x7ffc085bfe18
Value of *ip variable: 20
```

Арифметичні операції над вказівниками

Вказівник у С - адреса, що є числовим (цілим) значенням. Отже, ви можете виконувати арифметичні операції на вказівнику так само, як і на цілочисельному значенні (але є певні обмеження пов'язані з фізичним смислом вказівника). Таким чином використовуються чотири арифметичних оператори для вказівників: інкремент(++), декремент(--) та додавання і віднімання цілого числа, а також операції порівняння.
Відзначимо, що подібно будь-яким змінним, змінна типу вказівник має ім'я, адрес у пам'яті і значення.



За допомогою унарних операцій інкремент(++) і декремент(--) числові значення змінних типу вказівник міняються по різному, у залежності від типу даних (точніше від їх розміру), з яким зв'язані ці змінні.
Приклад:

```
char *mychar;
short *myshort,*i,*k;
float *f;
long* mylong;
z++; // значення зміститься на 1 байт
i++; // значення зміститься на 2 байти
f++; // значення зміститься на 4 байти
```

Тобто при зміні вказівника на 1, вказівник переходить до початку наступного (попереднього) поля тієї довжини, що визначається типом об'єкта, який адресується вказівником.
Щоб зрозуміти арифметику вказівника, припустимо, що ptr є вказівником на ціле число, що має адресу 1000. Якщо припустити, що у нас 32-бітні цілі числа, зробимо наступну арифметичну операцію на вказівнику: ptr ++

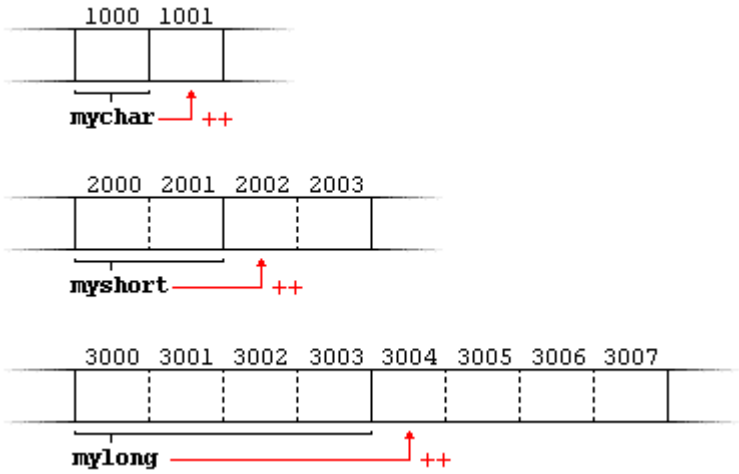


Рис.2

Після зазначеної вище операції, ptr буде вказувати на адресу 1004, оскільки кожен раз коли ptr збільшується на одиницю, він вказує на наступне ціле число, яке становить 4 байти поряд (пам'ятаємо, що ціле число у нас 4 байти) з поточним місцем розташування. Тобто операція інкременту перемістить вказівник на наступну ділянку для цілого числа. Таким чином, якщо ptr вказує на символ, адреса якого є 1000, то вищезгадана операція вкаже на розташування 1001, оскільки наступний символ буде доступний на 1001, а для типу short він перейде на адресу 1002.

Інкремент та збільшення вказівника

Враховуючи вищевказане, можна робити інкремент та збільшення вказівника на ціле число для руху по ділянці пам'яті або масиву.

Наступна програма збільшує вказівник змінної для доступу до кожного наступного елементу масиву

Приклад.

```
#include <stdio.h>
const int MAX = 3;
int main () {
  int  var[] = {10, 100, 200};
  int  i, *ptr;
  ptr = var; /* встановимо адресу (address) вказівника (pointer) */
  for ( i = 0; i < MAX; i++) {
 printf("Address of var[%d] = %x\n", i, ptr );
 printf("Value of var[%d] = %d\n", i, *ptr );
 ptr++; /* рухаємося далі по масиву */
}
}
```

Результат роботи коду:

```
Address of var[0] = bf882b30
Value of var[0] = 10
Address of var[1] = bf882b34
Value of var[1] = 100
Address of var[2] = bf882b38
Value of var[2] = 200
```

Крім того, вказівник дозволяє додавати не лише одиницю за допомогою інкременту, але й будь-яке ціле число подібним чином (за допомогою **оператору += або просто +):**

Приклад.

```
int  var[] = {10, 100, 200};
int  i, *ptr;
printf("Address of var[%d] = %x\n", i, ptr );
printf("Value of var[%d] = %d\n", i, *ptr );
ptr+=2; /*  переходимо на дві позиції вперед */
printf("Address of var[%d] = %x\n", i, ptr );
printf("Value of var[%d] = %d\n", i, *ptr );
```

Результат роботи коду:

Декремент та віднімання цілого числа від вказівника

Так само як інкремент, можна робити й операцію декременту над вказівниками:

```
#include <stdio.h>
const int MAX = 3;
int main () {
  int  var[] = {10, 100, 200};
  int  i, *ptr;
ptr = &var[MAX-1]; /* візьмемо адресу останнього елементу масиву*/
for ( i = MAX; i > 0; i--) {
printf("Address of var[%d] = %x\n", i-1, ptr );
printf("Value of var[%d] = %d\n", i-1, *ptr );
ptr--; /* рухаємося назад по масиву */
}
}
```

Результат:

```
Address of var[2] = bfedbcd8
Value of var[2] = 200
Address of var[1] = bfedbcd4
Value of var[1] = 100
Address of var[0] = bfedbcd0
Value of var[0] = 10
```

Аналогічно, можна віднімати будь-яке ціле число подібним чином (за допомогою **оператору -= або просто -):**

```
ptr = &var[MAX-1]; /* візьмемо адресу останнього елементу масиву*/
for ( i = MAX; i > 0; i--) {
printf("Address of var[%d] = %x\n", i-1, ptr );
printf("Value of var[%d] = %d\n", i-1, *ptr );
ptr = ptr - 2; /* рухаємося назад по масиву з кроком 2*/
}
```

Порівняння вказівників

Вказівники можна порівняти за допомогою реляційних операторів, таких як ==, <, та >. Якщо p1 і p2 вказують на змінні, пов'язані один з одним, такі як елементи одного і того ж масиву, то p1 і p2 можуть бути змістовно порівняні.

Наступна програма змінює попередній приклад, збільшуючи вказівник змінної, доки адреса, на яку він вказує, є меншою або рівною адресі останнього елемента масиву, тобто &var[MAX-1] :

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

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
```

Вкладені вказівники та масиви вказівників

Може виникнути ситуація, коли ми хочемо зберегти масив, який може зберігати вказівники на int або char або будь-який інший доступний тип даних. Далі йде оголошення масиву вказівників на ціле число:

```
int *ptr[MAX];
```

Тут оголошено ptr як масив MAX цілих вказівників. Таким чином, кожен елемент у ptr утримує вказівник на ціле значення. У наступному прикладі використовуються три цілих числа, які зберігаються в масиві вказівників:

```
#include <stdio.h>
static const int MAX = 3;
int main () {
int  var[] = {10, 100, 200};
int i, *ptr[MAX];
for ( i = 0; i < MAX; i++) {
ptr[i] = &var[i]; /* взяти адресу integer. */
    }
    for ( i = 0; i < MAX; i++) {
   printf("Value of var[%d] = %d\n", i, *ptr[i] );
    }
}
```

Результат роботи:

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

Використання вказівників при роботі з масивами

Відмітимо, що масив має визначений розмір, який не може бути змінений, оскільки він є постійним вказівником. Тоді інколи зручно використовувати вказівник замість масиву, якщо треба міняти розмір масиву динамічно в програмі. Ім'я масиву без індексу є вказівником-константою, тобто адресою першого елемента масиву (a[0]).

```
  ↓  a
      *a = = a[0] ;
*(a+1) = = a[1];
. . . . . .
*(a+i) = =a[i];
```

Відповідно до синтаксису в С існують тільки одномірні масиви, але їх елементами , у свою чергу, теж можуть бути масиви.

```
int a[5][5];
```

Для двовимірного масиву:

```
a[m][n] = = *(a[m]+n) = = *(*(a+m)+n);
```

Приклад1. Опис вказівників.

```
int *ptri; //вказівник на змінну цілого типу
char *ptrc; //вказівник на змінну символьного типу
float *ptrf; //вказівник на змінну з плаваючою крапкою
```

Такий спосіб оголошення вказівників виник внаслідок того, що змінні різних типів займають різну кількість комірок пам'яті. При цьому, для деяких операцій з вказівниками необхідно знати об'єм відведеної пам'яті. Тобто операція * в деякому розумінні є оберненою до операції &.

Вказівники використовуються для роботи з масивами. розглянемо оголошення двовимірного масиву:

```
int mas[4][2];
int *ptr;
```

Тоді вираз ptr=mas вказує на першу колонку першого рядка матриці. Записи mas і &mas[0][0] рівносильні. Вираз ptr+1 вказує на mas[0][1], далі йдуть елементи: mas[1][0], mas[1][1], mas[2][0] і т. д.; ptr+5 вказує на mas[2][1].

Двовимірні масиви розташовані в пам’яті так само, як і одновимірні масиви, займаючи послідовні комірки пам’яті

ptr	ptr+1	ptr+2	ptr+3	ptr+4	ptr+5
mas[0][0]	mas[0][1]	mas[1][0]	mas[1][1]	mas[2][0]	mas[2][1]

Масив, розмірність якого стає відомою в процесі виконання програми, називається д**инамічним.**

Робота з пам’яттю в С

В попередніх програмах ми використовували вказівники на вже існуючі змінні. Інколи потрібно передавати у вказівники адреси змінних, що потрібно визначити по ходу виконання програми. Для цього треба виділити місце в пам’яті, де будуть записуватися та зберігатися до кінця програми значення цих змінних.

Розглянемо роботу пам’яті комп’ютера при виконанні роботи програми.

Види пам’яті

Відкомпільована С-програма створює та використовує чотири логічно відокремлені ділянки пам’яті, що мають власні призначення.

Перша ділянка - це пам’ять, що містить код програми.

Друга ділянка призначена для зберігання глобальних змінних.

Третя ділянка зветься **стек** - швидкодоступна ділянка пам’яті, що може використовуватись для різноманітних цілей при виконанні програми. Вона містить адреси повернень функцій, що викликаються, аргументи, що передаються в функцію та локальні змінні. Стек також використовується для зберігання поточного стану процесору.

Четверта ділянка зветься **куча** - це ділянка вільної пам’яті, яку програма може використовувати для динамічного виділення пам’яті під різноманітні програмні об’єкти великого розміру, зокрема класи, структури, масиви й таке інше.

Концептуальна карта пам’яті що виділяє програма на С:

Стек
Купа
Глобальні змінні
Код програми

Запуск програм C / C ++ зазвичай складається з таких дій, які виконуються в описаному порядку:

1) Вимкнути всі переривання.
2) Копіювати будь-які ініціалізовані дані з ПЗУ в ОЗУ.
3) Занулити неініціалізовану область даних.
4) Виділити простір і ініціалізувати стек.
5) Ініціалізувати вказівник стека процесора.
6) Створити й ініціалізувати купу.
7) Виконати конструктори та ініціалізатори для всіх глобальних змінних (лише C ++).
8) Увімкнути переривання.
9) Виклик основної функції.

Як правило, код запуску також буде включати кілька інструкцій після виклику main. Ці інструкції будуть виконуватися тільки в тому випадку, якщо виконується програма мови високого рівня (тобто, виклик основних функцій повертається). Залежно від характеру вбудованої системи, ви можете використовувати ці інструкції для припинення роботи процесора, скидання всієї системи або передачі керування засобом відладки.
Оскільки код запуску не вставляється автоматично, програміст повинен, як правило, сам збирати його і включати отриманий об'єктний файл зі списку вхідних файлів до лінкера. Йому навіть може знадобитися спеціальний параметр командного рядка, щоб змусити його не вставляти звичайний код запуску. Робочий код запуску для різних цільових процесорів можна знайти у пакеті GNU під назвою libgloss.
Кілька потоків живуть в одному процесі в одному просторі, кожен потік виконуватиме конкретне завдання, мати свій власний код, власну пам'ять стека, вказівник інструкцій і спільну пам'ять. Якщо потік має витік пам'яті, він може пошкодити інші потоки і батьківський процес.

Використання пам’яті

Сам вказівник є змінною, що виділяється в програмному стеку. Сам вказівник означає адресу змінної, що виділяє пам’ять «на кучі».
Для того, щоб працювати з динамічним масивом, потрібно виділити відповідну пам’ять під цей масив. Тобто вказати компілятору, щоб він зарезервував місце у відповідній ділянці пам’яті для того, щоб записувати туди дані. Крім того, це потрібно щоб інші змінні, сам компілятор та інші програми також не записували дані в це місце. Якщо цього не зробити, то будуть серйозні помилки, які можуть призвести до зупинки не лише програми, але й всієї системи. Крім того, ще однією помилкою є незвільнення зарезервованої ділянки пам’яті після виконання програми або потрібної функції. На жаль, компілятор C(C++) не звільнить виділену пам’ять автоматично після виходу функції або програми і, таким чином, ділянка пам’яті, що була зарезервована всередині функції, залишиться зайнятою. Це так званий витік пам’яті (Memory leak). Він може призвести до того, що

наступні ділянки програми або повторні виклики чи багатопотокові виклики цієї програми будуть «падати» з невідомих причин або вішати систему. Така помилка не буде повідомлена компілятором, а подальше падіння програми буде виникати не в тому місці, де була зроблена помилка, що ускладнює процес відлагодження програми. Для боротьби з такими помилками потрібні спеціальні інструменти та спеціальні навички дебага, на зразок використання програми Valgrind.

Функція для виділення та знищення пам'яті

Для виділення пам'яті вказівнику можна використовувати такі стандартні функції:

void* malloc(size_t size);
void* calloc(size_t num, size_t size);
void *realloc(void *ptr, size_t new_size);

Функція виділення пам'яті malloc

Метод malloc визначає розміри байтів неініціалізованого сховища. Він визначений в заголовочному файлі <stdlib.h> з наступним інтерфейсом:

void* malloc(size_t size);

Якщо розподіл успішно завершується, метод повертає вказівник на найнижчий (перший) байт у виділеному блоці пам'яті.

Якщо розмір дорівнює нулю, поведінка є визначеною реалізацією (може бути повернуто нульовий вказівник або може бути повернений деякий ненульовий вказівник, який може не використовуватися для доступу до сховища, але повинен бути переданий вільному).

Параметри:

size - розмір(натуральне число) - кількість виділених байт

При успішному завершенні повертає вказівник на початок знову виділеної пам'яті. Щоб уникнути витоку пам'яті, повернений вказівник повинен бути звільнений з free () або realloc ().

При відмові повертає нульовий вказівник.

Приклад:

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
int *p1 = malloc(4*sizeof(int));  // виділяє пам'ять під масив 4 int
int *p2 = malloc(sizeof(int[4])); // аналогічно,
int *p3 = malloc(4*sizeof *p3);   // що один варіант того самого

if(p1) { // перевіряємо, чи коректно виділена пам'ять
for(int n=0; n<4; ++n) // проходимо масив та ініціалізуємо його
```

```
p1[n] = n*n;
for(int n=0; n<4; ++n) // друкуємо значення масиву
        printf("p1[%d] == %d\n", n, p1[n]);
}
   free(p1);
   free(p2);
   free(p3);
}
```

Output:

```
p1[0] == 0
p1[1] == 1
p1[2] == 4
p1[3] == 9
```

Функція ініціалізації виділеної пам’яті calloc

Метод calloc визначає розміри байтів ініціалізованого сховища. Він визначений в заголовочному файлі <stdlib.h> з наступним інтерфейсом:

void* calloc(size_t num, size_t size);

Метод виділяє пам'ять для масиву num об'єктів розміру size і ініціалізує всі байти в виділеному сховищі ***нулями***.

Якщо розподіл успішно завершується, метод повертає вказівник на найнижчий (перший) байт у виділеному блоці пам'яті, який відповідним чином вирівняний для будь-якого типу об'єкту.

Якщо розмір дорівнює нулю, поведінка визначена реалізацією (може бути повернуто нульовий вказівник або може бути повернений деякий ненульовий вказівник, який може не використовуватися для доступу до сховища)

Параметри:

num - кількість об’єктів

size - розмір кожного об’єкту

При успішному завершенні повертає вказівник на початок знову виділеної пам'яті. Щоб уникнути витоку пам'яті, повернений вказівник повинен бути звільнений з free () або realloc ().

При відмові повертає нульовий вказівник.

Примітка: Завдяки вимогам вирівнювання кількість виділених байт не обов'язково дорівнює num * **size**.

Приклад

```
#include <stdio.h>
#include <stdlib.h>

int main(void){
int *p1 = calloc(4, sizeof(int));    // ініціалізуємо пам'ять під 4 цілих
int *p2 = calloc(1, sizeof(int[4])); // ініціалізуємо пам'ять під 4 цілих
```

```
int *p3 = calloc(4, sizeof *p3);    // ініціалізуємо пам'ять під 4 цілих
if(p2) {
for(int n=0; n<4; ++n) // виводимо
        printf("p2[%d] == %d\n", n, p2[n]);
}
   free(p1);
   free(p2);
   free(p3);
}
```

Output:

```
p2[0] == 0
p2[1] == 0
p2[2] == 0
p2[3] == 0
```

Функція перерозподілу пам’яті realloc

Метод **realloc** визначений в <stdlib.h>. Він перерозподіляє задану область пам'яті та має інтерфейс:

void *realloc(void *ptr, size_t new_size);

Пам'ять повинна бути попередньо виділена за допомогою malloc(), calloc () або realloc () і ще не звільнена за допомогою виклику free() або realloc(). В іншому випадку результати не визначені.

Перерозподіл здійснюється зарахунок однієї з двох опцій:

а) розширення або стискання існуючої ділянки, на яку вказують ptr, якщо це можливо. Вміст області залишається незмінним до меншої кількості нових і старих розмірів. Якщо область розширена, вміст нової частини масиву не визначено.

б) виділяючи новий блок пам'яті розміру new_size байтів, копіюючи область пам'яті з розміром, рівним меншому з нових і старих розмірів, і звільняючи старий блок.

Якщо пам'яті недостатньо, старий блок пам'яті не звільняється і повертається нульовий вказівник.

Якщо ptr є NULL, поведінка є такою ж, як виклик malloc (new_size). Якщо new_size дорівнює нулю, поведінка визначена реалізацією (може бути повернуто нульовий вказівник (у цьому випадку старий блок пам'яті може бути або не може бути звільнений), або може бути повернений деякий ненульовий вказівник, який може не використовуватися для доступу до сховища) . Метод realloc є потокобезпечним: він веде себе так, ніби тільки отримує доступ до пам'яті, видимої через її аргумент, а не будь-яку статичну пам'ять.

Параметри:

ptr - вказівник на область пам'яті, яку потрібно перерозподілити

new_size - новий розмір масиву в байтах.

При успішному завершенні повертає вказівник на початок знову виділеної пам'яті. Щоб уникнути витоку пам'яті, повернений вказівник повинен бути звільнений з free () або realloc (). Оригінальний ptr вказівника недійсний, і будь-який доступ до нього є невизначеною поведінкою.
При відмові повертає нульовий вказівник. Оригінальний ptr вказівника залишається і може бути необхідним для звільнення з free () або realloc ().

Приклад

```
#include <stdio.h>
#include <stdlib.h>

int main(){
int *pa = malloc(10 * sizeof *pa); // виділяємо пам'ять під масив 10 цілих
if(pa) {
      printf("%zu bytes allocated. Storing ints: ", 10*sizeof(int));
for(int n = 0; n < 10; ++n)
         printf("%d ", pa[n] = n);
}

int *pb = realloc(pa, 1000000 * sizeof *pb); // перевиділяємо пам'ять під масив
if(pb) {
      printf("\n%zu bytes allocated, first 10 ints are: ", 1000000*sizeof(int));
for(int n = 0; n < 10; ++n)
         printf("%d ", pb[n]); // виводимо масив
      free(pb);
} else { // перевіряємо вказівник перед знищенням
      free(pa);
}
}
```

Output:

```
40 bytes allocated. Storing ints: 0 1 2 3 4 5 6 7 8 9
4000000 bytes allocated, first 10 ints are: 0 1 2 3 4 5 6 7 8 9
```

Функція вирівнювання aligned_alloc

Починаючі зі стандарту C11 в <stdlib.h> існує також функція aligned_alloc:

```
void *aligned_alloc( size_t alignment, size_t size );
```

Ця функція виділяє байти неініціалізованого сховища, вирівнювання якого задається параметром вирівнювання. Параметр розміру повинен бути цілим та кратним до вирівнювання. Метод alignment_alloc є потокобезпечним: він веде себе так, ніби тільки отримує доступ до пам'яті, видимої через її аргумент, а не будь-яку статичну пам'ять. Попередній виклик до free або realloc, який звільняє область пам'яті, синхронізується з викликом вирівнювання, який виділяє ту ж саму або частину тієї ж області пам'яті. Ця синхронізація відбувається після будь-якого доступу до пам'яті за допомогою функції, що звільняється, і перед будь-яким доступом до пам'яті вирівнювання. Існує єдиний повний

порядок всіх функцій розподілу і вивільнення, що діють на кожній конкретній області пам'яті.

Параметри:

alignment - вказує вирівнювання. Має бути допустиме вирівнювання, що підтримується реалізацією.

size - кількість виділених байт. Повинно бути цілим числом кратним вирівнюванню.

При успішному завершенні повертає вказівник на початок знову виділеної пам'яті. Щоб уникнути витоку пам'яті, повернений вказівник повинен бути звільнений функцією free () або realloc (). При відмові повертає нульовий вказівник.

Примітки: Передача розміру, який не є кратним вирівнюванню або вирівнюванням, що не є визначеним або не підтримується реалізацією, призводить до помилки функції і поверненню нульового вказівника.

Звичайний malloc вирівнює пам'ять, придатну для будь-якого типу об'єкта (який на практиці означає, що він вирівнюється до alignof (max_align_t)). Ця функція є корисною для обробки виключних ситуацій, наприклад, для SSE, кешу або VM.

Приклад

```
#include <stdio.h>
#include <stdlib.h>

int main(void){
int *p1 = malloc(10*sizeof *p1);
   printf("default-aligned addr:   %p\n", (void*)p1);
   free(p1);

int *p2 = aligned_alloc(1024, 1024*sizeof *p2);
   printf("1024-byte aligned addr: %p\n", (void*)p2);
   free(p2);
}
```

Можливе виведення:

```
default-aligned addr:   0x1e40c20
1024-byte aligned addr: 0x1e41000
```

Функція звільнення пам’яті free

Для звільнення виділеної пам’яті крім realloc використовується команда free, яка визначена в <stdlib.h>:

void free(void* ptr);

Звільнює простір, попередньо виділений malloc (), calloc (), alignment_alloc, (починаючи з C11) або realloc ().

Якщо ptr є нульовим вказівником, функція нічого не робить.

Поведінка є невизначеною, якщо значення ptr не дорівнює значенню, повернутому раніше malloc (), calloc (), realloc (), або alignment_alloc () (починаючи з C11).

Поведінка є невизначеною, якщо область пам'яті, на яку посилається ptr, вже була звільнена, тобто free() або realloc () вже викликана для вказівника ptr.

Параметри:

ptr - вказівник на пам'ять для звільнення

Поверненого значення немає.
Примітки: функція приймає (і нічого не робить) з нульовим вказівником. Незалежно від того, чи буде успішне виділення, вказівник, що повертається функцією розподілу, може бути переданий до free ()
Приклад:

```
#include <stdlib.h>
int main(void){
int *p1 = malloc(10*sizeof *p1);
free(p1); // все що виділено(allocated) як вказівник повинно бути звільнено
int *p2 = calloc(10, sizeof *p2);
int *p3 = realloc(p2, 1000*sizeof *p3);
if(p3) // якщо p3 не null це значить що p2 звільнено realloc
free(p3);
else // якщо p3 null означає що p2 ще не звільнено
free(p2);
}
```

Робота з пам'яттю в C++

В C++ для роботи з динамічними об'єктами використовують спеціальні операції new і delete. За допомогою операції new виділяється пам'ять під динамічний об'єкт (який створюється в процесі виконання програми), а за допомогою операції delete створений об'єкт видаляється з пам'яті.
Приклад . Виділення пам'яті під динамічний масив.
Нехай розмірність динамічного масиву вводиться з клавіатури. Спочатку необхідно виділити пам'ять під цей масив, а потім створений динамічний масив треба видалити.

```
…
unsigned n;
scanf("%u",&n); // n — розмірність масиву
int *mas=new int[n]; // виділення пам'яті під масив
delete [] mas; // звільнення пам'яті
…
```

В цьому прикладі mas є вказівником на масив з n елементів. Оператор int *mas=new int[n] виконує дві дії: оголошується змінна типу вказівник, далі вказівнику надається адреса виділеної області пам'яті у відповідності з заданим типом об'єкта.
Для цього ж прикладу можна задати наступну еквівалентну послідовність операторів:

```
…
int *mas;
unsigned n;
scanf("%u",&n); // n — розмірність масиву
mas=new int[n];// виділення пам'яті під масив
delete [] mas;// звільнення пам'яті
```

Якщо за допомогою операції new неможливо виділити потрібний об'єм пам'яті, то результатом операції new є 0.
Іноді при програмуванні виникає необхідність створення багатовимірних динамічних об'єктів. Програмісти-початківці за аналогією з поданим способом створення одновимірних

динамічних масивів для двовимірного динамічного масиву розмірності n*k запишуть наступне

```
mas=new int[n][k]; // Невірно! Помилка!
```

Такий спосіб виділення пам’яті не дасть вірного результату. Наведемо приклад створення двовимірного масиву.

```
#include<iostream.h>
#include<stdlib.h>
int main(){
int n;
const int m=5;
printf("input the number";
scanf(&n);
int** a; //a - вказівник на масив вказівників на рядки
a=new int* [n]; //виділення пам’яті для масиву вказівників на n рядків
for(int i=0;i<n;i++)
a[i]=new int [m]; /*виділення пам’яті для кожного рядка масиву розмірністю nxm
…*/
for(int i=0;i<n;i++){
        for(int j=0;j<m;j++)
printf(a[i][j]);
}
for(int i=0;i<n;i++)
delete [] a[i]; //звільнення пам’яті від кожного рядка
delete [] a; //звільнення пам’яті від масиву вказівників
getchar();
return 0;
}
}
```

Рядки (Null-terminated byte strings)

Символьний рядок представляє собою набір з одного або більше символів.
Приклад : "Це рядок"
В мові С немає спеціального типу даних, який можна було б використовувати для опису рядків. Замість цього рядки представляються у вигляді масиву елементів типу *char*. Тобто в С для зберігання рядків використовують символьні масиви. Це такі ж масиви як і ті що розглядалися раніше, але зберігають вони не числові дані, а символьні. Це означає, що символи рядка розташовуються в пам'яті в сусідніх комірках, по одному символу в комірці, як на рисунку:

I	t		i	s		s	t	r	i	n	g	\0

Необхідно відмітити, що останнім елементом масиву є символ '\0'. Це нульовий символ (байт, кожний біт якого рівний нулю). У мові С він використовується для того, щоб визначати кінець рядка.

Можна уявити символи такого масиву розташованими послідовно в сусідніх комірках пам'яті - в кожному осередку зберігається один символ і займає один байт. Один байт тому, що кожен елемент символьного масиву має тип char. Останнім символом кожної такого рядка є символ \ 0 (нульовий символ). Наприклад: «Текст: перший рядок \n\r Text \t Кінець. \0»
Сам текст, включаючи пробіл та спецсимволи, складається з певної кількості символів. Якби в останній комірці перебувала наприклад '.' (крапка), а не нульовий символ '\0' - для компілятора це вже не рядок. І працювати з таким набором символів треба було б, як зі звичайним масивом - записувати дані в кожну клітинку окремо і виводити на екран посимвольно (за допомогою циклу):
Приклад.

```
char str [16] = { 'T', 'e', 'x', 't', ' ', 'a', 'n', 'd', ' ', 's', 't', 'o', 'p', '!' ,'\0'};
for (int i = 0; i <=sizeof(str); i++)
{
   printf("%c",str[i]);
}
printf("\n");
```

На щастя, в С є куди більш зручний спосіб ініціалізації і звернення до символьних масивів – символьний рядок або рядок з нульовим завершенням. Для того, щоб його обмежити, останнім символом такого масиву обов'язково повинен бути нульовий символ '\0'. Саме він робить набір символів рядком, працювати з яким набагато легше, ніж з масивом символів. Цей рядок зветься *символьним рядком* в Сі, або рядком байтів з нульовим завершенням.
Рядок байтів з нульовим завершенням (NTBS) - це послідовність ненульових байтів, за якими слідує байт з нульовим значенням (символ завершення нуля). Кожен байт у рядку байтів кодує один символ деякого набору символів. Наприклад, масив символів {'x63', 'x61', 'x74', '0'} є NTBS, що містить рядок "cat" в кодуванні ASCII.
***Символьний рядок* (*стрінг*)** — це масив символів, що обмежений лапками (“”). Він визначений як масив типу char. Нульовий символ ('\0') автоматично додається останнім байтом символьного рядка та виконує роль ознаки його кінця. Кількість елементів у масиві дорівнює кількості символів у рядку плюс один, оскільки нульовий символ також є елементом масиву. Кожна рядкова константа, навіть у випадку, коли вона ідентична іншій рядковій константі, зберігається у окремому місці пам'яті. Якщо необхідно ввести у рядок символ лапок ("), то перед ним треба поставити символ зворотного слешу ('\'). У рядок можуть бути введені будь-які спеціальні символьні константи, перед якими стоїть символ '\'.
Основні методи ініціалізації символьних рядків.

- `char str1[]= "ABCdef";`
- `char str2[]={'A', 'B', 'C', 'd', 'e', 'f','\0'};`
- `char str3[100]; gets(str3);`

Цей спосіб вважається зараз небезпечним з точки зору Buffer Overflow, тому краще скористатись наступною формою, де вказується довжина рядку:
fgets(str3, 100, stdin);
При цьому потрібно вводити не більше символів, ніж в розмірі буферу (в даному випадку - 100).

```
char str4[100];
scanf("%s",str4);
```

Помітимо, що в цій формі непотрібно використовувати амперсанд перед змінною str4 (подумайте самі, чому). Зауважимо, що ця форма теж може бути Buffer Overflow небезпечною, тому можна використовувати fscanf(stdin,"%s", name);
Помітимо, що при даному способі вводу вводитись буде лише до першого пробілу, що введений в тексті, тому якщо потрібно ввести саме рядок з пробілами, то потрібно використовувати форму scanf(" %[^\n]s",name); (див. таблицю специфікаторів scanf та printf). Подібна стратегія потрібна також і в випадках, якщо потрібно обробити інші роздільники.

Декларація рядку

Оголошується рядок таким чином - створюємо масив типу char, розмір в квадратних дужках вказувати не обов'язково (його підрахує компілятор), оператор = і в подвійних лапках пишемо необхідний текст. Тобто ініціалізуємо масив рядковою константою:

```
#include <stdio.h>
#include <locale.h>
int main (){
   setlocale (LC_ALL, "Ukrainain");
   char str2[] = "Текст"; // '\0' присутній неявно
   printf("str=%s\n",str2);
}
```

Прописувати нульовий символ не треба. Він присутній неявно і додається в кожну таку строкову константу автоматично. Таким чином, при тому, що ми бачимо текст "text" з 4 символів в рядку, розмір масиву буде 5, так як '\ 0' теж символ і займає один байт пам'яті. Займе він останній символ цього символьного масиву.
Примітка. Це може не бути правдою для масиву, що записаний не латинським алфавітом, а наприклад кирилицею, бо кириличний символ може займати більше ніж один байт. Наприклад, на моєму комп'ютері текст з попереднього прикладу займає 2*5+1 =11 байт, бо українські літери займають 2 байти.
Примітка до примітки. В сучасному С для підтримки різних алфавітів та кодувань (наприклад UTF-8) можна використовувати тип символу wchar_t (мультибайтовий символ).

Як бачите, для виведення рядка на екран, досить звернутися до неї по імені: printf("%s",str); буде виводити на екран символ за символом, поки не зустріне в одній з комірок масиву символ кінця рядка \0 і виведення перерветься. Таке звернення для звичайного символьного масиву (масиву без \ 0) не є правильним: Так як компілятор виводив би символи на екран навіть вийшовши за рамки масиву, поки не зустрів би в якійсь комірці пам'яті символ \ 0. Можете спробувати підставити в перший приклад замість циклу оператор виведення рядку і побачите, що вийде. Наприклад **показало ось так:**
Text and stop!#Text and stop!
str=

Зверніть також увагу на відмінність символьної константи (в одинарних лапках - 'f', '@') від рядкової константи (в подвійних лапках "f", "@"). Для першої, компілятором C ++ виділяється один байт для зберігання в пам'яті. Для символу записаного в подвійних лапках, буде виділено два байта пам'яті - для самого символу і для нульового (додається компілятором).
Символьна константа складається з одного символу ASCII між апострофами ('). Приклади спеціальних символів:
Таблиця 4.1

Новий рядок	'\n'
Горизонтальна табуляція	'\t'
Повернення каретки	'\r'
Апостроф	'\''
Лапки	'\"'
Нульовий символ	'\0'
Зворотний слеш	'\'

Усі константи-рядки в тексті програми, навіть ідентично записані, розміщуються за різними адресами в статичній пам'яті. З кожним рядком пов'язується сталий вказівник на його перший символ. Власне, рядок-константа є виразом типу "вказівник на char" зі сталим значенням - адресою першого символу.
Так, присвоювання p="ABC" (*p* - вказівник на *char*) встановлює вказівник p на символ 'A'; значенням виразу *("ABC"+1) є символ 'B'.
Елементи рядків доступні через вказівники на них, тому будь-який вираз типу "вказівник на char" можна вважати рядком.
Необхідно мати також на увазі те, що рядок вигляду "x" - не те ж саме, що символ 'x'. Перша відмінність : 'x' - об'єкт одного з основних типів даних мови C (*char*), в той час, як "x" - об'єкт похідного типу (масиву елементів типу *char*). Друга різниця : "x" насправді складається з двох символів - символу 'x' і нуль-символу.

Функції введення рядків.

Прочитати рядок із стандартного потоку введення можна за допомогою функції *gets*(). Вона отримує рядок із стандартного потоку введення. Функція читає символи до тих пір, поки їй не зустрінеться символ нового рядка '\n', який генерується натисканням клавіші ENTER. Функція зчитує всі символи до символу нового рядка, додаючи до них нульовий символ '\0'.

Синтаксис :

```
char *gets(char *buffer);
```

Як відомо, для читання рядків із стандартного потоку введення можна використовувати також функцію *scanf*() з форматом *%s*. Основна відмінність між *scanf*() і *gets*() полягає у способі визначенні досягнення кінця рядка; функція *scanf*() призначена скоріше для читання слова, а не рядка. Функція *scanf*() має два варіанти використання. Для кожного з них рядок починається з першого не порожнього символу. Якщо використовувати *%s*, то рядок продовжується до (але не включаючи) до наступного порожнього символу (пробіл, табуляція або новий рядок). Якщо визначити розмір поля як *%10s*, то функція *scanf*() не прочитає більше 10 символів або ж прочитає послідовність символів до будь-якого першого порожнього символу.

Що якщо рядок повинен буде ввести користувач за допомогою клавіатури? В цьому випадку необхідно оголосити масив типу char із зазначенням його розміру достатнього для зберігання символів, що вводять, включаючи \0. Не забувайте про цей нульовий символ. Якщо вам треба зберігати 3 символи в масиві, його розмір повинен бути на одиницю більше - тобто 4.

```
#include <stdio.h>
#include <locale.h>
int main (){
  printf("Input 0:");
  char str[20];
  int i=0;
  while((str[i]=getchar()) && str[i]!='\n' && i<19) {i++; }
  str[i] ='\0';
  puts(str);
}
```

Використовуючи порожні лапки при ініціалізації, ми присвоюємо кожному елементу масиву значення \0. Таким чином рядок буде очищений від "сміття" інших програм. Навіть якщо користувач введе рядок, що містить меншу кількість символів, наступний за рядком буде символ \0. Це дозволить уникнути небажаних помилок.

Примітка. Нульовий символ - це не цифра 0; він не виводиться на друк і в таблиці символів ASCII має номер 0. Наявність нульового символу передбачає, що кількість комірок масиву повинна бути принаймні на одну більше, ніж число символів, які необхідно розміщувати в пам'яті. Наприклад, оголошення

char str[10];
передбачає, що рядок містить може містити максимум 9 символів.
Таким чином, можливі наступні **варіанти введення рядків на Сі:**
1) **scanf("%s",name); // scanf_s("%s",name);**
Зауважимо, що ця команда введе рядок лише до першого пробілу або роздільника. Для того щоб ввести далі за допомогою scanf можна використати іншу форму:
scanf_s("%[^\n]s",name); /* цей варіант scanf ігнорує пробіли та нові рядки при вводі*/
Але в цьому варіанті можуть бути ще проблеми при існуванні попереднього вводу. Тоді можна або очистити буфер за допомогою fflush (stdin); або використати scanf з пробілом перед специфікатором
scanf(" %[^\n]s",name);

2) gets(siteName); // не рекомедовано
Ця форма має Buffer Overflow, тому краще скористатись наступною формою, де вказується довжина рядку:
fgets(siteName, 20, stdin);
При цьому потрібно вводити не більше символів ніж в розмірі буферу (в даному випадку - 20).

3) **fscanf(stdin,"%s",name); // fscanf_s(stdin,"%s",name);**
Файлова форма scanf() де ми на початку вказуємо стандартний файл введення з консолі.

4) Ще одна файлова форма введення для якої потрібно визначити змінну Name фіксованої довжини символів:
fread(Name, 1, sizeof(Name),stdin);

Виведення рядків

Тепер розглянемо інші функції виведення рядків. Для виведення рядків можна використовувати функції *puts*() і *printf*().
Синтаксис функції puts():
int puts(char *string);
Ця функція виводить всі символи рядка *string* у стандартний потік виведення. Виведення завершується переходом на наступний рядок.
Синтаксис функції printf() :
printf("%s", string);
Різниця між функціями *puts*() і *printf*() полягає в тому, що функція *printf*() не виводить автоматично кожний рядок з нового рядка.

Стандартна бібліотека мови програмування С містить клас функцій для роботи з рядками, і всі вони починаються з літер str. Для того, щоб використовувати одну або декілька функції необхідно підключити файл string.h (#include<string.h>)
Можна використовувати динамічний масив символів для того, щоб працювати з рядками:

```
#include <stdio.h>
const int MAX = 4;
int main () {
char *names[] = {
"Georgetta",
"Musetta",
"Odetta",
"Kusetta"
};

int i = 0;
for ( i = 0; i < MAX; i++) {
printf("Value of names[%d] = %s\n", i, names[i] );
}
return 0;
}
```

Результат роботи:

```
Value of names[0] = Georgetta
Value of names[1] = Musetta
Value of names[2] = Odetta
Value of names[3] = Kusetta
```

Основні функції роботи з рядками

Визначення довжини рядка. Для визначення довжини рядка використовується функція *strlen*(). Її синтаксис :

size_t strlen(const char *s);

Функція *strlen*() повертає довжину рядка *s*, при цьому завершуючий нульовий символ не враховується.

Приклад :

```
char *s= "Some string";
int len;
```

Наступний оператор встановить змінну *len* рівною довжині рядка, що адресується вказівником *s*:

```
len = strlen(s); /* len == 11 */
```

Копіювання рядків. Оператор присвоювання для рядків не визначений. Тому, якщо *s1* і *s2* - символьні масиви, то неможливо скопіювати один рядок в інший наступним чином.

```
char s1[100];
char s2[100];
s1 = s2; /* помилка Останній оператор (s1=s2;) не зкомпілюється.*/
```

Щоб скопіювати один рядок в інший, необхідно викликати функцію копіювання рядків *strcpy*(). Для двох вказівників *s1* і *s2* типу char * оператор strcpy(s1,s2); копіює символи, що адресуються вказівником *s2* в пам'ять, що адресується вказівником *s1*, включаючи завершуючі нулі.

Для копіювання рядків можна використовувати і функцію *strncpy*(), яка дозволяє обмежувати кількість символів, що копіюються.

strncpy(destantion, source, 10);

Наведений оператор скопіює 10 символів із рядка *source* в рядок *destantion*. Якщо символів в рядку *source* менше, ніж вказане число символів, що копіюються, то байти, що не використовуються, встановлюються рівними нулю.

Примітка. Функції роботи з рядками, в імені яких міститься додаткова літера n мають додатковий числовий параметр, що певним чином обмежує кількість символів, з якими працюватиме функція.

Конкатенація рядків.

Конкатенація двох рядків означає їх об'єднання, при цьому створюється новий, більш довгий рядок. Наприклад, при оголошенні рядка

```
char first[]= "Один ";
```

оператор

strcat(first, "два три чотири!");

перетворить рядок first в рядок "Один два три чотири".

При викликанні функції strcat(s1,s2) потрібно впевнитися, що перший аргумент типу char * ініціалізований і має достатньо місця щоб зберегти результат. Якщо s1 адресує рядок, який вже записаний, а s2 адресує нульовий рядок, то оператор strcat(s1,s2); перезапише рядок s1, викликавши при цьому серйозну помилку.

Функція *strcat*() повертає адресу рядка результату (що співпадає з її першим параметром), що дає можливість використати "каскад" декількох викликів функцій :

```
strcat(strcat(s1,s2),s3);
```

Цей оператор додає рядок, що адресує *s2*, і рядок, що адресує *s3*, до кінця рядка, що адресує *s1*, що еквівалентно двом операторам:

```
strcat(s1,s2);
strcat(s1,s3);
```

Повний список прототипів функцій роботи з рядками можна знайти в таблиці 4.2.

Порівняння рядків.

Функція *strcmp*() призначена для порівняння двох рядків. Синтаксис функції :

int strcmp(const char *s1, const char*s2);

Функція *strcmp*() порівнює рядки *s1* і *s2* і повертає значення 0, якщо рядки рівні, тобто містять одне й те ж число однакових символів. При порівнянні рядків ми розуміємо їх порівняння в лексикографічному порядку, приблизно так, як наприклад, в словнику. У функції насправді здійснюється посимвольне порівняння рядків.

Кожний символ рядка *s1* порівнюється з відповідним символом рядка *s2*. Якщо *s1* лексикографічно більше *s2*, то функція *strcmp*() повертає додатне значення, якщо менше, то - від'ємне.

Прототипи всіх функцій, що працюють з рядками символів, містяться у файлі string.h. У файлі <string.h> оголошена велика кількість функцій для роботи з рядками. Всі функції працюють з рядками, що закінчуються нульовим символом. Нижче наведено оголошення деяких функцій і їхнє призначення:

Таблиця 4.2

char *strcat (char * dest , const char *source)	Функція дописує рядок dest у кінець рядка source
char *strncat(char * dest, const char *source, unsigned n)	Функція дописує рядок dest у кінець рядка source
char *strchr (const char *source, int c)	Функція дописує не більше n символів рядка dest у кінець рядка source
char strrchr (const char *source, int c)	Пошук у рядку source першого входження зліва символу c, повертає вказівник на знайдений символ або NULL
int strcmp (const char *s1, const char *s2)	Пошук у рядку source першого входження зправа символу c, повертає вказівник на знайдений символ або NULL
int strncmp (const char *s1, const char *s2, unsigned n)	Порівнює рядки посимвольно, зліва направо. Повертає 0, якщо рядки s1 і s2 рівні, повертає негативне число, якщо s1 за алфавітом раніш s2, повертає позитивне число, якщо s1 за алфавітом пізніше s2
int stricmp (const char *s1, const char *s2)	Порівнює рядки за першими n символами
	порівнює рядки без обліку регістра символів
char *strcpy (char * dest, const char *source)	Копіювання рядка source у рядок dest
char *strncpy (char * dest, const char *source, unsigned n)	Копіювання не більш n перших символів у рядок dest
int strlen (const char *s)	Повертає довжину рядка s
char *strlwr (char *s)	Перетворює символи рядка в нижній регістр (у маленькі букви)
char *strupr (char *s)	Перетворює символи рядка у верхній регістр (у великі букви)
char *strset (char *s, int c)	Заповнює весь рядок s символом c
char *strnset (char *s, int c, unsigned n)	Заміняє перші n символів рядка s на символ c

char *strrev (char *s)	Розташовує всі символ рядка s, за винятком нуль термінатора, у зворотному порядку
size_t strcspn (const char *s1, const char *s2)	Повертає довжину початкової ділянки рядка s1, що складається тільки із символів, яких немає в s2
size_t strspn (const char *s1, const char *s2)	Повертає довжину початкової ділянки рядка s1, що складається тільки із символів, що є в s2
char *strpbrk (char *s1, char *s2)	Переглядає рядок s1 зліва направо, поки не зустрінеться кожний із символів рядка s2; повертає вказівник на знайдений символ або NULL
char *strtok (char *s1, const char *s2)	Застосовується для послідовного пошуку в рядку s1 фрагментів рядка, обмежених символами з заданої множини *роздільників*. Множина роздільників задається рядком s2. Перше звертання до strtok фіксується у рядок (s1), в якому ведеться пошук. Для пошуку другого і третього і т.д. фрагментів рядка у тім же рядку в якості першого параметру задається NULL. Функція повертає вказівник на знайдений фрагмент рядка або NULL, якщо в рядку токенов не залишилося. Вихідний рядок змінюється – по мірі знаходження фрагментів рядка, перший за ними роздільник заміняється на нуль термінатор
char *strstr (const char *s1, const char *s2)	Шукає перше з ліва входження рядка s2 у рядок s1. Повертає або вказівник на s2 у s1, або NULL, якщо s1 не містить рядка s2
char *strdup (const char *s)	Функція створює копію рядка s. Пам'ять для копії виділяється автоматичним викликом malloc(strlen(s) + 1). Програміст повинний самостійно звільнити цю пам'ять, коли вона стане непотрібною.

Робота з рядками, що не обов'язково закінчуються нульовим байтом

Для роботи з масивом символів, що не обов'язково має у кінці нульовий символ, можна користуватися функціями перетворення буферів. Прототипи цих функцій знаходяться у файлі string.h. Ці функції дозволяють присвоювати кожному байту в межах вказаного буфера задане значення, а також використовуються для порівняння вмісту двох буферів. Наприклад:
memcpy() — копіювання символів з одного буфера у другий, поки не буде скопійований заданий символ або не буде скопійовано визначену кількість символів
memcmp() — порівнює вказану кількість символів з двох буферів
У файлі ctype.h описано прототипи функцій, що призначені для перевірки літер. Ці функції повертають ненульове значення (істина), коли її аргумент задовольняє заданій умові або належить вказаному класу літер, та нуль в іншому випадку. Наприклад:

int islower(int c) — символ с є малою літерою;
int isupper(int c) — символ с є великою літерою;
int isalnum(int c) — символ с є буквою або цифрою;
int isalpha(int c) — символ с є буквою;
int tolower(int c) — перетворення літери у нижній регістр;
int strtol(int c) — перетворення рядку у довге ціле число;

Приклад

Очистити вираз від пробілів та зайвих символів зліва:

```
char *ltrim(char *str, const char *seps)
{
   size_t totrim;
   if (seps == NULL) {
      seps = "\t\n\v\f\r ";
   }
   totrim = strspn(str, seps);
   if (totrim > 0) {
      size_t len = strlen(str);
      if (totrim == len) {
         str[0] = '\0';
      }
      else {
         memmove(str, str + totrim, len + 1 - totrim);
      }
   }
   return str;
}
```

Таблиця 4.3

Методи класифікації символів	
В заголовочному файлі <ctype.h>	
isalnum	символ є літерою або цифрою
isalpha	символ є літерою
islower	символ є малою літерою
isupper	символ є великою літерою
isdigit	символ є цифрою
isxdigit	символ є шістнадцятковою цифрою
iscntrl	символ є контрольним символом
isgraph	символ є графічним символом
isspace	символ є пробілом
isblank(C99)	символ є порожнім символом

isprint	символ є друкованим
ispunct	символ є пунктуаційним
Character manipulation	
tolower	перетворення літери у нижній регістр
toupper	перетворення літери у верхній регістр

Примітка: додаткові функції, чиї імена починаються з c або to в нижньому регістрі, можуть бути додані до заголовка ctype.h і не повинні визначатися програмами, які включають цей заголовок.

В бібліотеці stdlib.h визначаються функції для перетворення рядків у числові типи.

Таблиця 4.4

Перетворення у рядків у числові типи

Заголовочний файл <stdlib.h>	
atof	переведення рядку до дійсного типу
atoi, atol, atoll(C99)	переведення рядку до цілого значення
strtol, strtoll(C99)	переведення рядку до цілого значення
strtoul, strtoull(C99)	переведення рядку до натурального значення
strtof, strtod (C99), strtold(C99)	переведення рядку до дійсного типу
Заголовочний файл <inttypes.h>	
strtoimax(C99), strtoumax(C99)	переведення рядку до intmax_t та uintmax_t
Функції роботи з рядками	
Заголовочний файл <string.h>	
strcpy, strcpy_s(C11)	копіює рядки один в інший
strncpy, strncpy_s(C11)	копіює визначену кількість символів з одного рядку в інший
strcat, strcat_s(C11)	конкатенує два рядки
strncat,strncat_s(C11)	копіює визначену кількість символів двох рядків
strxfrm	ттрансоформує рядок, щоб strcmp мав той же результат, що й strcoll
Інормація по рядкам	
Заголовочний файл <string.h>	
strlen(), strnlen_s(C11)	довжина рядку
strcmp	порівняння двох рядків
strncmp	порівнює визначену кількість символів в рядку
strcoll	порівняння двох рядків в даній локалі
strchr	знаходить першу позицію символу в рядку

strrchr	знаходить останню позицію символу в рядку
strspn	повертає довжину максимального сегменту в одному рядку що збігається з символами в другому рядку
strcspn	повертає довжину максимального сегменту в одному рядку що не містяться у символах в другого рядку
strpbrk	знаходить першу позицію хоч якогось символу з першого рядку в другому рядку
strstr	знаходить першу позицію підрядку
strtok, strtok_s(C11)	знаходить наступний роздільник в рядку
Операції з памяттю	
Заголовок <string.h>	
memchr	знаходить першу позицію символу в байтовому масиві
memcmp	порівнює два буфери
memset, memset_s(C11)	заповнює буфер символом
memcpy, memcpy_s(C11)	копіює два буфери
memmove, memmove_s(C11)	пересуває два буфери
Різне	
Визначено в <string.h>	
strerror, strerror_s(C11) strerrorlen_s (C11)	повертає текстову версію даної помилки

4) Функції на Сі. Області дії та специфікатори функцій

Функції та процедурне програмування. Опис функції на С. Тип void. Прототипи функцій. Рекурсія
Масиви та вказівники як аргументи функцій. Як повернути вказівник з функції. Вказівники на функцію
Функція з довільним числом аргументів (stdarg.h).
Типи змінних та області дії змінних. Локальні змінні. Глобальна змінні. Формальні параметри(аргументи функції).
Змінні оточення.
Шляхи передачі значень у функцію
Головна функція та робота з командним рядком
Специфікатори змінних. Специфікатори зберігання та специфікатори доступу
Специфікатори функцій
Перетворення типів

Функції та процедурне програмування

При написанні програм середнього та високого рівня складності виникає потреба в їх розбитті на частини. Розбиття великої програми на частини дозволяє зменшити ризик виникнення помилок, підвищує читабельність програмного коду завдяки його структуруванню.
Крім того, якщо деякий програмний код повторюється декілька разів (або є близьким за змістом), то є доцільним організувати його у вигляді функції, яку потім можна викликати багатократно за її іменем. Таким чином, відбувається економія пам'яті, зменшення вихідного коду програми, тощо.
Функція – це частина програми, яка має такі властивості чи ознаки:

- є логічно самостійною частиною програми;
- має ім'я, на основі якого здійснюється виклик функції (виконання функції). Ім'я функції підпорядковується правилам задавання **імен ідентифікаторів мови С++**;
- може містити список параметрів, які передаються їй для обробки або використання. Якщо функція не містить списку параметрів, то така функція називається функцією без параметрів;
- може повертати (не обов'язково) деяке значення. У випадку, якщо функція не повертає ніякого значення, тоді вказується ключове слово void;
- має власний програмний код, який береться у фігурні дужки { } і вирішує задачу, яка поставлена на цю функцію. Програмний код функції, реалізований в фігурних дужках, називається "тіло функції".

Використання функцій у програмах дає такі переваги:

- компактна організація програми шляхом зручного виклику програмного коду за його іменем, який у програмі може зустрічатись декілька разів (повторюватись);
- економія пам'яті, розміру вихідного та виконавчого коду і т.д.;
- зменшення ризику виникнення помилок для великих наборів кодів;

- підвищення читабельності програмного коду.

Функцію в С++ можна розглядати:

- як один з похідних типів даних (поряд з масивами і вказівниками);
- як мінімальний виконуваний модуль програми (підпрограму).

Формат визначення функції

Всі функції мають однаковий формат визначення:

[тип результату] ім'я функції ([список формальних аргументів])
{ // *тіло функції*
<опис даних;>
<оператори;>
[return] [вираз];
};

Тип результату:

Функція може повертати деяке (**одне** !) значення. Це значення і є результат виконання функції, який при виконанні програми підставляється в точку виклику функції, де б цей виклик не зустрівся. Допускається також використовувати функції що не мають аргументів і функції що не повертають ніяких значень. Дія таких функцій може полягати, наприклад, в зміні значень деяких змінних, виводі на друк деяких текстів і тому подібне.

Таким чином **[тип результату] -** *будь-який базовий або раніше описаний тип* значення (*за винятком масиву і функції*), що повертається функцією. Функція може нічого і не повертати - це необов'язковий параметр. За відсутності повертання і типу результату тип результату за замовчуванням буде цілий (**int**). Він також може бути описаний ключовим словом (**void**), тоді функція не повертає ніякого значення. Якщо результат повертається функцією, то в тілі функції є необхідним оператор **return вираз**;, де **вираз** формує значення, що співпадає з типом результату. Тип результату, який повертається функцією; в разі, якщо функція не повертає ніякого значення, специфікується типом void і функція називається "порожньою". Найчастіше, це функції, які виводять на екран повідомлення чи виконують деякі зміни параметрів, проте не можуть передати певний результат іншим змінним при виклику.

Ім'я функції

<**ім'я_функції**>- або main для головної функції, або довільний ідентифікатор.

Ім'я функції — ідентифікатор функції, за яким завжди знаходиться пара круглих дужок "**()**", де записуються **формальні аргументи**. Фактично **ім'я функції** — це особливий вид покажчика на функцію, його значенням є адреса початку входу у функцію;

формальні аргументи або *список формальних параметрів*

< **список формальних аргументів** >- або порожній ( ), або список, кожен елемент якого записується як:

<тип> <ім'я_формального_параметру>

Список формальних аргументів має такий вигляд:
([const] тип 1 [параметр 1], [const] тип 2 [параметр 2], . . .)
Список формальних аргументів визначає кількість, тип і порядок проходження переданих у функцію вхідних аргументів, які розділяються комою («,»). У випадку, коли параметри відсутні, дужки залишаються порожніми або містять ключове слово (**void**). Формальні параметри функції локалізовані в ній і недоступні для будь-яких інших функцій.
У списку формальних аргументів для кожного параметра треба вказати його тип **(***не можна групувати параметри одного типу, вказавши їх тип один раз***).**

Наприклад:
```
int k, //  коректний запис одно параметру
char l, char j, int z, // коректний запис трьох параметрів
int x,y // некоректно
```
Тіло функції
Тіло функції – це набір операторів, що виконуються у фігурних дужках {} при виклику функції. Тіло функції може бути складовим оператором або блоком. На С *визначення функцій не можуть бути вкладеними.*
Тіло функції може складатися з описів змінних або **опису даних**.
Опис даних полягає в декларації змінних, що використовуються в функції.
Змінні, що використовуються при виконанні функції, можуть бути **глобальні** і **локальні**. Змінні, що описані (визначені) за межами функції, називають **глобальними**. За допомогою глобальних параметрів можна передавати дані у функцію, не включаючи ці змінні до складу формальних параметрів. У тілі функції їх можна змінювати і потім отримані значення передавати в інші функції.
Змінні, що описані у тілі функції, називаються **локальними** або **автоматичними**. Вони існують тільки під час роботи функції, а після реалізації функції система видаляє локальні змінні і звільняє пам'ять. Тобто між викликами функції вміст локальних змінних знищується, тому ініціювання локальних змінних треба робити щоразу під час виклику функції.
Оператори це будь яка послідовність інструкцій, що може призводити до обчислення результату функції**.**
Повертання результату [return] [вираз];
Для передачі результату з функції у функцію, що її викликала, використовується оператор return. Його можна використовувати у двох формах:
return;
завершує функцію, яка не повертає ніякого значення (тобто перед її іменем вказано тип void)
***Приклад 1*:**
```
void op (int x, int y){
        printf("\nПросто виводимо щось %d %d!\n", x, y);
        return; // цей оператор можна просто не писати
```

}
Метод повернення значення
return <вираз>;
повертає значення виразу, причому *тип виразу повинен співпадати з типом функції*.
Приклад 2.
```
float cube(float d){
return d*d*d;  // тип результату float
}
```

Після визначення функцію можна багаторазово використовувати у програмі для виконання однотипних дій над різними змінними.

Виклик функції
Виклик функції - це вираз, значенням якого є результат, що виробляється функцією. Якщо вираз функції використовується як окремий оператор (тобто не у виразі), то значення, котре повертає функція втрачається. Якщо функція описана з типом **void** (як така, що не повертає значення) використовується в деякому виразі, то її значення вважається невизначеним.
Існує два способи виклику функцій:
<ім'я функції> (<список фактичних параметрів>)
або
(*<вказівник на функцію>)(<список фактичних параметрів>)
Вказівником на функцію є деякий вираз, значенням якого є адреса цієї функції.
Другий спосіб як правило використовується для того, щоб зробити в С певний аналог функціонального програмування і в нашому курсі розглядатись не буде.
Виклик функції має наступний вигляд:
<ім'я_функції>(([список фактичних параметрів])
Список фактичних параметрів - або сигнатура, є переліком виразів, кількість яких дорівнює кількості формальних параметрів функції. Між формальними і фактичними параметрами повинна бути відповідність за типами. В якості фактичних параметрів можна використовувати змінні, визначені та ініціалізовані у програмі, з типами, що відповідають типам формальних параметрів. Якщо функція повертає значення, її виклик можна використати у правій частині операції присвоювання з метою передачі результату функції змінній, тип якої співпадає з типом функції, що викликається.
Наприклад:,
```
void main(){
float s, f=0.55;
s=cube(f);
…}
```

В якості фактичних параметрів також можуть виступати явно задані константні значення:
Наприклад, виклик функції з *прикладу 1* має наступний вигляд:
```
c = op ( '+', 5 ,4 );
```

Тип void
Якщо функція не повертає значення, тоді вона повинна починатися з ключового слова **void**.
Він може позначати як відсутність аргументів у функції так і відсутність результатів функції.
Ключове слово **void** розглядається як окремий тип в C/C++. В стандарті C визначено термін "тип об'єкта". У C99 і раніше; **void** не є типом об'єкта; у C11, він є таким. У всіх версіях стандарту **void** є неповним типом, тобто типом який неможливо завершити (на відміну від інших неповних типів, які можна завершити). Це означає, що ви не можете застосувати оператор sizeof до **void** і не може оголошувати змінну типу **void**, але ви можете мати покажчик на неповний тип.
У C11 змінилося те, що неповні типи тепер є підмножиною типів об'єктів; це лише зміна термінології. (Інший тип типу - тип функції).
Ключове слово **void** також можна використовувати в інших контекстах:
• Як єдиний тип параметра в прототипі функції, як і в int func (**void**), він вказує, що функція не має параметрів. (C ++ використовує пусті дужки для цього, але вони означають щось інше в C.) При використанні для списку параметрів функції void вказує, що функція не приймає ніяких параметрів.
• Як тип повернення функції, як у **void** func (int n), це вказує на те, що функція не повертає результату. При використанні як тип повернення функції ключове слово void вказує, що функція не повертає значення.
• **void** * - це тип вказівника, який не вказує на конкретний тип. При використанні в оголошенні вказівника **void,** він що це "універсальний" вказівник. Тобто якщо тип вказівника є **void** *, вказівник може вказувати на будь-яку змінну, яка не оголошена ключовим словом const або volatile. Вказівник **void** не може бути розіменований, якщо він не буде переданий іншому типу та **void*** може бути перетворений у будь-який інший тип вказівника. Універсальний вказівник може вказувати на функцію, але ми повинні розіменувати його з вказівником **void** * на функцію для подільшої роботи, крім того void не може вказувати на члена класу в C++ .
Примітка. При застосуванні до функцій слово **void** як правило використовують лише для вказання типу результату, але можна і повертати його та вказувати в аргументах (не рекомендовано в C++).
Зокрема, в C наступні функції еквівалентні:
```
void f(){};
void f(void){};
void f(){ return void;}
```

```
void f(void) {return;}
```

Приклад 3. Функція MyFunc1() без параметрів, яка не повертає значення.
Якщо в тілі деякого класу або модуля описати функцію:

```
// опис функції, яка не отримує і не повертає параметрів
void MyFunc1(void){
   // тіло функції - вивід тексту на форму в компонент label1
   label1->Text = "MyFunc1() is called";
   return; // повернення з функції
}
```

тоді викликати цю функцію можна наступним чином:

```
// виклик функції з іншого програмного коду (наприклад, обробника події)
MyFunc1();
...
```

Приклад 4. Функція MyFuncMult2(), яка отримує 1 параметр цілого типу і не повертає значення. Функція здійснює множення отриманого параметру на 2 і виводить результат на форму в компонент label1.

```
// функція, що отримує ціле число, множить його на 2 і виводить результат
void MyFuncMult2(int x){
   int res;     // внутрішня змінна
   res = x * 2; // обчислення результату
   printf("r=%d", res) // вивід результату
}
```

Виклик функції з іншого програмного коду:

```
// виклик функції з обробника події
MyFuncMult2(42); // буде виведено 84
MyFuncMult2(-8); // -16
```

Приклад 5. Функція, яка отримує 2 параметри типу double, знаходить їх добуток і виводить його на форму в елементі управління label1.

```
// функція, що множить параметр x на параметр y і виводить результат на форму
void MyFuncMultDouble(double x, double y){
   double z;
   z = x*y;
   printf("r=%f \n", z) // вивід результату
   return;
}
```

Виклик функції з іншого програмного коду:

```
MyFuncMultDouble(2.5, -2.0); // буде виведено -5
MyFuncMultDouble(1.85, -2.23); // буде виведено -4.1255
```

Опис та використання функцій, що повертають параметр

Функція може отримувати параметр за значенням або за адресою та повертати результат.

Приклад 6. Отримання параметру за значенням. Функція, яка отримує один параметр цілого типу, множить його на 5 і повертає результат. Функція не виконує виведення результату.

```
// функція, що множить параметр на 5
int Mult5(int d){
   int res;
   res = d * 5;
   return res; // повернення результату
}
```

Виклик функції з іншого програмного коду

```
// виклик функції з іншого програмного коду
int x, y;
x = 20;
y = Mult5(x);   // y = 100
y = Mult5(-15); // y = -75
```

Приклад 7. Функція обчислення значення y = sign(x), що визначається за правилом:

Реалізація функції:

```
int sign(float x)
{   if (x<0) return -1;
   if (x==0) return 0;
   if (x>0) return 1;
}
```

Виклик функції з іншого програмного коду:

```
int res;
res = sign(-0.399f); // res = -1
res = sign(0.00f);   // res = 0
res = sign(2.39);    // res = 1
```

Приклади опису та використання функцій, що отримують два і більше параметрів

Приклад 8. Приклад функції MaxFloat(), що отримує 2 параметри типу float і повертає максимальне значення з них.

```
// функція, що знаходить максимум між двома дійсними числами
float MaxFloat(float x, float y)
{
   if (x>y) return x;
   else return y;
}
```

Слід звернути увагу, що у даній функції 2 рази зустрічається оператор return.

Виклик функції MaxFloat() з іншого програмного коду:

```
// виклик функції з іншого програмного коду
float Max; // змінна - результат
Max = MaxFloat(29.65f, (float)30); // Max = 30.0

double x = 10.99;
double y = 10.999;
Max = MaxFloat(x, y); // Max = 10.999
Max = MaxFloat((float)x, (float)y); // Max = 10.999 - так надійніше
```

Приклад 9. Функція MaxInt3(), яка знаходить максимальне значення між трьома цілими числами.

```
// функція, що знаходить максимум між трьома цілими числами
// функція отримує 3 цілочисельних параметри з іменами a, b, c
int MaxInt3(int a, int b, int c){
   int max;
   max = a;
   if (max<b) max = b;
   if (max<c) max = c;
   return max;
}
```

Виклик функції з іншого програмного коду

```
// виклик функції з іншого програмного коду
int a = 8, b = 5, c = -10;
int res;
res = MaxInt3(a, b, c);      // res = 8
res = MaxInt3(a, b+10, c+15); // res = 15
res = MaxInt3(11, 2, 18);    // res = 18
```

Прототипи функції

Важливою особливістю мов С та С++ є можливість прототипування функцій.

Прототип *надає* компіляторові *інформацію про тип та ім'я функції*, а також інформацію *про типи, кількість та порядок розміщення аргументів*, які їй можна передавати. Зважаючи на це, імена формальних параметрів зазначати необов'язково. Прототип являє собою зразок для здійснення синтаксичної перевірки компілятором.

Прототипування функції є за смислом схожим на декларування (визначення) змінних перед використанням. Спочатку створюється прототип функції - тобто її визначення як тип результату, назва функції та типи її аргументів (список формальних параметрів). Після цього компілятор не буде видавати помилку при компіляції програми в місцях де ми будемо її викликати, але лише лінковщик вже буде шукати і підставляти її реалізацію в виконуваний файл. Це зроблено зокрема для того, щоб ми мали можливість додавати функції з інших бібліотек, що не

включені в файл, які компілюється в даний момент. Оскільки визначення функцій не можуть міститися всередині блоків та складових операторів, тобто в інших функціях, у програмі вони можуть розміщуватися як до, так і після функції, яка їх викликає. В останньому випадку перед використанням функції у програмі необхідно розмістити її опис, або прототип, інакше виникатимуть проблеми.
Запис прототипу може містити тільки перелік типів формальних параметрів без імен, а наприкінці прототипу завжди ставиться символ «**;**», тоді як у описі (визначенні) функції цей символ після заголовка не присутній.
Механізм передачі параметрів є основним засобом обміну інформацією між функцією, що викликається, та функцією, яка викликає. Параметри, котрі зазначаються у заголовку опису функції, називаються формальними, а параметри, які записані у операторах виклику функції — фактичними. Наведемо приклад фрагмента програми з використанням функцій:
Приклад:

```
double sqr (double); // прототип функції sqr()
int main( ) // голавна функція
{ //---------------- виклик функції sqr()
 printf("Квадрат числа=%lf \n", sqr (9));
}
double sqr (double p) //реалізація функції sqr()
{ return p*p; } //повернення результату
```

У результаті виконання програми буде виведено:

```
Квадрат числа = 81
```

Функція завжди має бути визначена або оголошена до її виклику. **При оголошенні, визначенні та виклику тієї самої функції типи та послідовність параметрів повинні співпадати.** На імена параметрів обмежень на відповідність не існує, оскільки функцію можна викликати з різними аргументами, а в прототипах імена ігноруються компілятором (вони необхідні тільки для покращення читання програми). Тип значення, що повертає функція, та типи параметрів спільно визначають тип функції.
У найпростішому випадку при виклику функції слід вказати її ім’я, за яким у круглих дужках через кому треба перелічити імена аргументів, що передаються. Виклик функції може здійснюватися у будь-якому місці програми, де за синтаксисом дозволяється вираз того типу, що формує функція. Якщо тип значення, що повертає функція не void, вона може входити до складу виразів або, у поодинокому випадку, розташовуватись у правій частині оператора присвоювання.
Компілятор передусім послідовно перевіряє коректність виклику та використання об'єктів у програмі, при виявленні функції, яка не була описана чи визначена раніше, видасть повідомлення про помилку і вказівку про те, що функція повинна містити прототип. Те саме повідомлення можна побачити на

екрані, якщо використовувати у програмі функцію зі стандартних бібліотек і не під'єднувати заголовний файл у якому вона описана. Прототип функції користувача багато в чому нагадує заголовок функції і має наступний формат:

<тип><ім'я_функції>(<список_формальних_параметрів>);

Головною відмінністю є наявність в кінці опису крапки з комою.

Так, прототипи функцій з прикладів 5 та 6 матимуть вигляд:

```
int Mult5(int d);
int MaxInt3(int a, int b, int c);
```

Прототип функції дозволяє компілятору виконувати більш надійну перевірку типу. Оскільки прототип функції повідомляє компілятору, чого очікувати, компілятор краще зможе позначити будь-які функції, які не містять очікуваної інформації. Прототип функції не описує тіло функції.

Приклад.

```
int getsum (float x, float y);
void getIt(int y);
```

Прототипи найчастіше використовуються в файлах заголовків, хоча вони можуть з'являтися в будь-якому місці програми. Це дозволяє викликати зовнішні функції в інших файлах, а компілятор перевіряти параметри під час компіляції.

Мета прототипу функції

1. Прототип функції гарантує, що виклики функції виконуються з правильним числом і типами аргументів.
2. Прототип функції визначає кількість аргументів.
3. У ньому зазначається тип даних кожного з переданих аргументів.
4. Він дає порядок, в якому аргументи передаються функції.
5. Прототип функції повідомляє компілятору, чого очікувати, що надати функції і чого очікувати від функції.

Переваги прототипів функцій

1. Прототипи зберігають час налагодження.
   Прототипи запобігають проблемам, які виникають при компіляції за допомогою функцій, які не були оголошені.
   Коли відбувається перевантаження функції, прототипи розрізняють версію функції, яку потрібно викликати.
2. Прототип функції дозволяє уникнути помилок при передаванні фактичних параметрів у формальні параметри.
3. Якщо відсутній прототип функції, то компілятор приймає тип формальних параметрів рівним типу фактичних параметрів. Це може призвести до помилки.
4. Прототип функції дає компілятору інформацію про тип формальних параметрів всередині функції. Це важливо, коли тип фактичних параметрів не співпадає з типом формальних параметрів.

Приклад. Нехай дано функцію Div(), яка отримує 2 числа типу long int і повертає результат ділення націло цих чисел. Функція не має прототипу (див. нижче).

```
// функція, що ділить 2 числа націло
```

```
long int Div(long int x, long int y)
{
   long int res;
   res = x / y; // результат - ціле число
   return res;
}
```
Якщо в іншому програмному коді написати так:
```
...
// виклик функції з іншого програмного коду
int a, b, c;
a = 290488;
b = -223;
c = Div(a, b); // правильна відповідь: c = -1302. Значення c може бути помилкове
```

то є ризик виникнення помилкового результату в змінній c. Це пов'язано з тим, що при виклику функції компілятор не має інформації про тип формальних параметрів (x, y), що використовуються у функції. Компілятор вважає, що тип параметрів у функції такий самий як і тип фактичних параметрів (змінні a, b), тобто тип int. Однак, функція Div() використовує значення параметрів типу long int, який в пам'яті має більшу розрядність ніж тип int. Тому, може виникнути спотворення значень.
Щоб уникнути такої помилки рекомендовано давати прототип функції. У даному випадку прототип має вигляд:
```
long int Div(long int, long int);
```
Щоб уникнути такої помилки рекомендовано давати прототип функції. У даному випадку прототип має вигляд:
```
long DivRigth(long, long);
```

А виклик буде наступним:
```
long c2 = DivRigth((long)a, b); // правильна відповідь: c = -1302.
printf("c=%ld",c2);
```

Реалізація буде такою ж самою:
```
long DivRigth(long x, long y){
   long res;
   res = x / y; // результат - ціле число
   return res;
   }
```

На C++ якщо функція описується в класі і викликається з методів цього класу, тоді подібних помилок не буде. Це пов'язане з тим, що в класі прототип функції відомий усім методам класу.

Використання вказівників в функціях

Програмування С дозволяє використовувати вказівник у аргументах функції. Для цього просто потрібно оголосити параметр функції як тип вказівника.
Нижче наведено простий приклад, де ми передаємо вказівник на довгий натуральний тип у функцію і змінюємо значення всередині функції, яке потім можна отримати як результат у функції, що викликається:

```
#include <stdio.h>
#include <time.h>

void getSeconds(unsigned long *par);

int main () {
unsigned long sec;
getSeconds( &sec );
printf("Number of seconds: %ld\n", sec ); /* виводимо значення */
return 0;
}

void getSeconds(unsigned long *par) {
*par = time( NULL ); /* отримаємо кількість секунд в поточному системному часі
*/
}
```

Результат:

```
Number of seconds :1294450468
```

Бачимо, що в цьому випадку значення аргументу змінилося, тобто використання вказівника дозволяє робити ***аргументи-змінні***, тобто це ще один шлях отримати результат з функції окрім просто прийняти return.
Функція, що використовує як аргумент вказівник, також може приймати масив як аргумент.
Тепер розглянемо наступну функцію, яка приймає масив як аргумент разом з іншим аргументом і на основі переданих аргументів, вона повертає середнє з чисел, переданих через масив наступним чином

```
#include <stdio.h>
double getAverage(int *arr, int size); /* декларація функції */

int main () {
        int balance[5] = {1000, 2, 3, 17, 50}; /* масив 5 елементів */
```

```
        double avg;
        avg = getAverage( balance, 5 ) ; //передаємо вказівник на масив як аргумент
        printf("Average value is: %f\n", avg ); /* виводимо результат */
}

double getAverage(int *arr, int size) {
        int  i, sum = 0;   /* реалізація функції: середнє значення*/
        double avg;
        for (i = 0; i < size; ++i) {
                sum += arr[i];
        }
        avg = (double)sum / size;
        return avg;
}
```

Результат:

```
Average value is: 214.40000
```

Ми бачили, що мова програмування C дозволяє повернути масив з функції. Аналогічно, C також дозволяє повернути вказівник функції. Для цього потрібно оголосити функцію, що повертає покажчик, як у наведеному нижче прикладі:

```
int * myFunction() {
***
}
```

По-друге, слід пам'ятати, що не варто повертати адресу локальної змінної поза функцією, тому вам доведеться визначити локальну змінну як *статичну* змінну. Тепер розглянемо наступну функцію, яка генерує 10 випадкових чисел і повертає їх за допомогою імені масиву, який представляє вказівник, тобто адресу першого елемента масиву.

```
#include <stdio.h>
#include <time.h>

/* функція, що генерує масив випадкових чисел */
int * getRandom( ) {
        static int r[10];
        int i;
        srand( (unsigned)time( NULL ) );  /*  встановлюємо сіль(seed) */

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

```
        return r;
}
int main () {
        int *p; /* створюємо вказівник на ціле число */
        int i;
        p = getRandom();

        for ( i = 0; i < 10; i++ ) {
                printf("*(p + [%d]) : %d\n", i, *(p + i) );
        }
}
```

Результат:

```
1523198053
1187214107
1108300978
430494959
1421301276
930971084
123250484
106932140
1604461820
149169022
*(p + [0]) : 1523198053
*(p + [1]) : 1187214107
*(p + [2]) : 1108300978
*(p + [3]) : 430494959
*(p + [4]) : 1421301276
*(p + [5]) : 930971084
*(p + [6]) : 123250484
*(p + [7]) : 106932140
*(p + [8]) : 1604461820
*(p + [9]) : 149169022
```

Таким чином, якщо потрібно передати масив одновимірних параметрів як аргумент у функції, вам доведеться оголосити формальний параметр одним з наступних трьох способів, і всі три методи декларування дають подібні результати, тому що кожен повідомляє компілятору про те, що використовується вказівник на ціле число для отримання результату. Аналогічно, ви можете передати багатовимірні масиви як формальні параметри.

Використання масивів як аргументи функції:

- Формальні параметри як вказівник:

```
void myFunction (int * param) {
}
```

При цьому способі довжина масиву не має значення, оскільки функція приймає вказівник.

- Формальні параметри як масив визначеного розміру:
```
void myFunction (int param [10]) {
}
```
- Формальні параметри як безрозмірний масив:
```
void myFunction (int param []) {
}
```

При цьому способі довжина масиву також не має значення, оскільки функція насправді приймає вказівник і цей спосіб є насправді еквівалентним першому.

Повернення масивів з функції

Мова С не дозволяє повернути весь масив як аргумент функції. Однак, ви можете повернути вказівник на масив, вказавши ім'я масиву без індексу.

Якщо ви хочете повернути одновимірний масив з функції, вам доведеться оголосити функцію, що повертає вказівник, як у наступному прикладі -

```
int * myFunction () {
}
```

По-друге, слід пам'ятати, що С не одобрює повернення адреси локальної змінної за межі функції, тому вам доведеться визначити локальну змінну як статичну змінну, або виділити пам'ять за допомогою функцій malloc, calloc та знищити цю пам'ять після використання функції далі в програмі.

Рекурсія

Рекурсія – це алгоритмічна конструкція, де підпрограма викликає сама себе. Рекурсія дає змогу записувати циклічний алгоритм, не застосовуючи команд циклу.

Пряма рекурсія – прямою (безпосередньою) рекурсією є виклик функції усередині тіла цієї функції.

a() {.....a().....}

Непряма рекурсія – коли функція здійснює рекурсивний виклик функції за допомогою ланцюжка виклику інших функцій. Всі функції, що входять в ланцюжок, теж вважаються рекурсивними.

приклад:

a(){.....b().....}
b(){.....c().....}
c(){.....a().....} .

Всі функції a,b,c є рекурсивними, оскільки при виклику однієї з них, здійснюється виклик інших і самій себе.

Всі функції в мові С можуть бути рекурсивними, тобто будь-яка з них може викликати саму себе.

Приклад: Скласти функцію для обчислення n!, де використовуючи рекурсію, можна так:

```
#include <stdio.h>
long int factorial(int n){
        if (n==0 || n==1) return 1;
        else return n*factorial(n-1);
}
int main(){
        int x;
        printf("Ввести число x=");scanf(“%d”,&x);
        printf("Факторіал x!=%Ld",factorial(x));
}
```

Приклад: Рекурсивна функція обчислення добутку цілих чисел від **a** до **b** має вигляд:

```
int Suma (int a, int b) {
        int S;
        if (a==b) S=a;
        else S=b+Suma(a,b-1);
        return S;
}
```

Для нормального завершення будь-яка рекурсивна функція повинна містити хоча б одну нерекурсивну галузь, що закінчується оператором повернення.

Приклад 3. З функцією обчислення найбільшого спільного дільника:

```
unsigned gcd(unsigned x, unsigned y){
  if(y==0) return x;
  return gcd(y,x%y);
}
…
unsigned m=gcd(450,80);
```

Типи змінних та області дії змінних

Існує три типи змінних у програмі С.

1. **Локальні змінні**
2. **Глобальна змінні**
3. **Формальні параметри(аргументи функції).**
4. **Змінні оточення**

Область видимості – це ділянка програмного коду, де визначена змінна існує та за межами якої змінні не може бути викликана (використана). У С всі ідентифікатори пов’язані лексично (або статично) з певними областями програмного коду. За типами змінних та їх декларації виділяють три місця, де означення змінних може бути використано:

Локальні змінні - всередині функції до **програмного блоку**.

Ззовні всіх функції - глобальні змінні (global variables).
В визначенні функції, в якості аргументів - формальні параметри (formal parameters).

Локальні змінні в С:

Локальні змінні – це змінні що створюються (тобто визначаються, наживо ініціалізуються та присвоюються) всередині функції або блоку програми (тобто в ділянці, що обмежена фігурними дужками).
Такі змінні, що описані у тілі функції, називаються **локальними** або **автоматичними**. Вони існують тільки під час роботи функції, а після реалізації функції система видаляє локальні змінні і звільняє пам'ять. Тобто між викликами функції вміст локальних змінних знищується, тому ініціювання локальних змінних треба робити щоразу під час виклику функції.
Тобто, область дії локальних змінних буде лише в межах функції. Ці змінні оголошуються в межах функції і не можуть бути доступні за межами функції.
У наведеному нижче прикладі змінні m та n мають область дії тільки в межах основної функції. Вони не є видимими для тестової функції.
Аналогічно, змінні a та b мають видимі у тестовій функції. Вони не видимі з основної функції.

```
#include<stdio.h>
void test(); // декларація функції test
int main(){
int m = 22, n = 44;// m, n  локальні змінні
/*m та n мають область дії всередині main().
Функція test їх не бачить.
m =a +b; --- помилка!
При спробі використати тут a або b і в інших функціях,буде помилка 'a' undeclared
та 'b' undeclared */
printf("\nvalues : m = %d and n = %d", m, n);
test(); // Виклик test()
}

void test() { // реалізація функції test
int a = 50, b = 80; // a, b локальні змінні test function
        /*a та b мають область дії лише в цій функції. Їх не бачить main function.
  a = n+m; -- Помилка!
При спробі використати m або n тут і в інших функціях,буде помилка 'm'
undeclared та 'n' undeclared */
printf("\nvalues : a = %d and b = %d", a, b);
}
```

Результат:

```
values : m = 22 and n = 44values : a = 50 and b = 80
```

Глобальні змінні

Змінні, які описані поза всіма функціями, тобто на початку програми називаються глобальними (змінна m). До глобальних змінних можна звернутися з будь-якої функції та блока.

Глобальні об'єкти можуть використовуватися для повернення результату підпрограми. Тобто в підпрограмі можна не використовувати команди return: передати значення головній програмі можна відразу в тілі підпрограми, надавши значення глобальній змінній. Але такий підхід не є професійним, так як в цьому випадку підпрограму стає неможливо багаторазово використовувати для зміни значень різним глобальним об'єктам.

Область дії глобальних змінних буде проходити протягом всієї програми. Доступ до цих змінних можна отримати з будь-якої точки програми. Глобальні змінні також визначається за межами основної функції. Таким чином, вони видимі для головної функції та всіх інших підфункцій.

Приклад програми для глобальної змінної в С:

```
#include<stdio.h>
void test();
int m = 22, n = 44; // Глобальні змінні
int a = 50, b = 80; // Глобальні змінні

int main(){
   printf("All variables are accessed from main function");
   printf("\nvalues: m=%d:n=%d:a=%d:b=%d", m,n,a,b); // всі змінні доступні
test();
}

void test(){
 printf("\n\nAll variables are accessed from test function"); .. // всі змінні доступні
 printf("\nvalues: m=%d:n=%d:a=%d:b=%d", m,n,a,b);
}
```

Результат:

```
All variables are accessed from main function
values : m = 22 : n = 44 : a = 50 : b = 80
All variables are accessed from test function
values : m = 22 : n = 44 : a = 50 : b = 80
```

Навіщо потрібні глобальні та локальні змінні

Змінні можна вважати каналами комунікації в межах програми. Коли встановлюється значення у змінній в одній точці програми, то в іншій точці (або точках) ви читаєте значення знову. Ці дві точки можуть бути у сусідніх висловлюваннях, або вони можуть знаходитися в широко розділених частинах програми. Скільки триває дія змінної? Наскільки широко

розділені можуть бути налаштування та вибірка частин програми, і як довго після встановлення змінної вона зберігається? Залежно від змінної та способу її використання, вам можуть знадобитися різні відповіді на ці запитання. Видимість змінної визначає, скільки решти програми може отримати доступ до цієї змінної. Ви можете організувати, що змінна є видною тільки в межах однієї частини однієї функції, або в одній функції, або в одному вихідному файлі, або в будь-якому місці програми. Чому ви хочете обмежити видимість змінної? Для максимальної гнучкості, чи не було б зручно, якщо б всі змінні були потенційно видимі скрізь? Таке розташування буде занадто гнучким: скрізь у програмі вам доведеться відстежувати імена всіх змінних, які де-небудь вказуються в програмі, щоб ви не випадково повторно використали його. Всякий раз, коли змінна помилково мала неправильне значення, вам доведеться шукати всю помилку в програмі, оскільки будь-яка операція у всій програмі могла б потенційно змінити цю змінну. Ви б постійно кодували в великому обсязі коду, використовуючи загальну назву змінної, як і в двох частинах вашої програми, і наявність одного фрагмента коду випадково перезаписувало значення, що використовуються іншою частиною коду.

Щоб уникнути цієї плутанини, С зазвичай надає змінним найменшу або найменшу видимість. Змінна, оголошена в дужках {} функції, видно тільки в межах цієї функції; змінні, оголошені в межах функцій, називаються локальними змінними.

Якщо інша функція в іншому місці оголошує локальну змінну з тим же ім'ям, то вона є іншою змінною цілком, і обидві не зіткнуться один з одним.

З іншого боку, змінна, оголошена поза будь-якої функції, є глобальною змінною, і вона потенційно може бути видимою в будь-якому місці програми. Використовуйте глобальні змінні, коли хочете, щоб шлях комунікації міг подорожувати до будь-якої частини програми. Коли ви оголошуєте глобальну змінну, ви, як правило, даєте їй більш довге, більш описове ім'я (не щось загальне, як і або x), так що всякий раз, коли ви використовуєте його, ви будете пам'ятати, що це одна і та ж змінна всюди. Іншим словом для видимості змінних є область застосування.

Як довго тривають змінні? За замовчуванням локальні змінні (ті, що оголошені у функції) мають автоматичну тривалість: вони виникають, коли функція викликається, і вони (і їх значення) зникають, коли функція повертається. З іншого боку, глобальні змінні мають статичну тривалість: вони тривають, а збережені в них значення зберігаються, доки програма не закінчує виконання. (Звичайно, ці значення можуть бути перезаписані, тому вони не обов'язково зберігаються назавжди.)

Нарешті, можна розбити функцію на декілька вихідних файлів для полегшення обслуговування. Коли кілька вихідних файлів об'єднані в одну програму, компілятор повинен мати спосіб кореляції глобальних змінних, які можуть бути використані для зв'язку між кількома вихідними файлами. Крім того, якщо глобальна змінна буде корисною для зв'язку, повинна бути одна з них: ви не хочете, щоб одна функція в одному вихідному файлі зберігала значення в одній глобальній змінній з ім'ям globalvar, а потім мали іншу функцію в інший вихідний файл з іншого глобальної змінної з ім'ям globalvar. Таким чином, глобальна змінна повинна мати точно один визначальний екземпляр, в одному місці в одному вихідному файлі. Якщо одна і та ж змінна буде використана де-небудь ще (тобто в іншому вихідному файлі або файлі), змінна оголошується в інших файлах із зовнішнім декларацією, що не є визначальним екземпляром.

У зовнішній декларації дається інформація:

- ***назва***
- ***тип глобальної змінної, що використовується***

Але при цьому мається на увазі що вона не визначена тут та для неї невизначено місця, вона визначена в іншому місці, і є лише посилання на неї тут. Якщо ви випадково маєте два різних

примірника визначення змінної з однаковою назвою, компілятор (або лінкер) скаржиться, що це "multiply defined"
У зв'язку з наявністю глобальних та локальних об'єктів, оболонка капсули є блоком і діє як мембрана, пропускаючи в себе глобальні об'єкти і не випускаючи локальних. Цей ефект в програмуванні носить назву мембранного ефекту. Але правило хорошого стилю програмування забороняє використання глобальних об'єктів в підпрограмі (для передачі значень в підпрограму використовуються параметри, а для повернення значень з підпрограми в головну програму – використовують команду return. Це робить підпрограму універсальною).

Ініціалізація локальних та глобальних змінних

Коли локальна змінна декларується вона не ініціалізується системою, її потрібно ініціалізувати самостійно. Глобальні змінні ініціалізовані автоматично коли їх декларують, таким чином як наведено в таблиці:

Тип даних	Значення при створенні
int	0
char	'\0'
float	0
double	0
вказівник	NULL

Формальні параметри

Формальні параметри - це аргументи які передаються в визначенні функції та перевантажують якщо їх назви співпадають глобальні змінні.
Параметр сполучає підпрограму з її оточенням. Параметри використовуються тільки для передачі інформації між оточенням та підпрограмою. В описі підпрограми присутні **формальні** параметри, які створюються в момент виклику підпрограми. При виклику з головної програми у назві підпрограми перераховуються **фактичні** параметри, які повинні мати значення. Ці значення передаються **формальним** параметрам Локальні ж об'єкти є тимчасовими ресурсами, необхідними для виконання під програмної капсули і є цілком схованими в середині підпрограми. Локальні об'єкти – це здебільшого проміжні змінні, необхідні для розв'язання під задачі, що ставиться перед підпрограмною капсулою. По завершенню роботи підпрограми локальні об'єкти знищуються.

Формальні параметри – це змінні, що приймають значення аргументів (параметрів) функції. Якщо функція має декілька аргументів (параметрів), їх тип та імена розділяються комою ' , '.
список формальних аргументів — визначає кількість, тип і порядок проходження переданих у функцію вхідних аргументів, які розділяються комою («,»). У випадку, коли параметри відсутні, дужки залишаються порожніми або

містять ключове слово (**void**). Формальні параметри функції локалізовані в ній і недоступні для будь-яких інших функцій.
Список формальних аргументів має такий вигляд:
([const] тип 1 [параметр 1], [const] тип 2 [параметр 2], . . .)
У списку формальних аргументів для кожного параметра треба вказати його тип **(не можна групувати параметри одного типу, вказавши їх тип один раз).**
При виклику функції, що має аргументи, компілятор здійснює копіювання копій формальних аргументів в стек.
Приклад 1. Функція MyAbs(), що знаходить модуль числа має один формальний параметр x.

```
// функція, що знаходить модуль дійсного числа
float MyAbs(float x) // x - формальний параметр
{
   if (x<0) return (float)(-x);
   else return x;
}
```

Виклик функції з іншого програмного коду (іншої функції)

```
// виклик функції з іншого програмного коду
float res, a;
a = -18.25f;
res = MyAbs(a); // res = 18.25f; змінна a - фактичний параметр
res = MyAbs(-23); // res = 23; константа 23 - фактичний параметр
```

При виклику функції з іншого програмного коду фігурує фактичний параметр. У даному прикладі фактичний параметр це змінна a та константа 23.
При виклику функції фактичні параметри копіюються в спеціальні комірки пам'яті в стеку (стек – частина пам'яті). Ці комірки пам'яті відведені для формальних параметрів. Таким чином, формальні параметри (через використання стеку) отримують значення фактичних параметрів.
Оскільки, фактичні параметри копіюються в стек, то зміна значень формальних параметрів в тілі функції не змінить значень фактичних параметрів (тому що це є копія фактичних параметрів).

Область видимості формальних параметрів функції визначається межами тіла функції, в якій вони описані. У наведеному нижче прикладі формальний параметр n цілого типу має область видимості в межах фігурних дужок { }.
Приклад. Функція, що знаходить факторіал цілого числа n.

```
// функція, що знаходить n!
unsigned long int MyFact(int n) // початок області видимості формального параметру n
{
   int i;
   unsigned long int f = 1; // результат
```

```
   for (i=1; i<=n; i++)
      f = f*i;

   return f;   // кінець області видимості формального параметру n
}
```

Виклик функції з іншого програмного коду (іншої функції):

```
// виклик функції з іншого програмного коду
int k;
unsigned long int fact;
k = 6;
fact = MyFact(k); // fact = 6! = 720
```

Приклад

```
#include <stdio.h>
int a = 20; /* глобальні змінні */
int sum(int a, int b);
int main () {   /* локальні змінні в main function */
  int a = 10; int b = 20; int c = 0;
  printf ("value of a in main() = %d\n",  a);
  c = sum( a, b);
  printf ("value of c in main() = %d\n",  c);
  return 0;
}

/* функція додавання */
int sum(int a, int b) {
   printf ("value of a in sum() = %d\n",  a);
   printf ("value of b in sum() = %d\n",  b);
   return a + b;
}
```

Результат:

```
value of a in main() = 10
value of a in sum() = 10
value of b in sum() = 20
value of c in main() = 30
```

Таким чином, як підсумок наступні правила областей дії змінних в Сі:

Глобальна дія: Доступ до та з будь-якої точки програми (програма може складатись і з кількох файлів).

```
// файл: file1.c
int a;
int main(void){
```

```
  a = 2;
}

/* файл: file2.c - Коли компілюємо його з file1.c, функції файлу можуть бачити зовнішню (extern) змінну int a; */
int myfun(){
  a = 2;
}
```

Щоб обмежити доступ тільки до поточного файлу, глобальні змінні можуть бути позначені як статичні.

Область блоку: Блок - це набір висловлювань, укладених у ліву та праву дужки ({та} відповідно). Блоки в С можуть бути вкладені (блок може містити інші блоки всередині нього). Змінна, оголошена в блоці, доступна в блоці і всіх внутрішніх блоках цього блоку, але недоступна поза блоком.

*Якщо **внутрішній блок оголошує змінну з такою ж назвою, що і змінна, оголошена зовнішнім блоком**, то видимість змінної зовнішнього блоку закінчується на **початку** декларації внутрішнім блоком.*

```
#include <stdio.h>
int main()
{
  {
     int x = 10, y  = 20;
     {
        // зовнішній блок x та y, тому
        // наступні команди коректні та друкують 10 and 20
        printf("x = %d, y = %d\n", x, y);
        {
           // y декларуються знову, тому зовнішнє y не доступне в цьому блоці
           int y = 40;
           x++;  // Змінює x до 11
           y++;  // Змінює y до 41
           printf("x = %d, y = %d\n", x, y);
        }
        // Ця команда стосується змінних зовнішнього блоку
        printf("x = %d, y = %d\n", x, y);
     }
  }
  return 0;
}
```

Результат:

```
x = 10, y = 20
```

```
x = 11, y = 41
x = 11, y = 20
```

Параметри функції: Сама функція є блоком. Параметри та інші локальні змінні функції виконуються за тими ж правилами блоку. Змінна, оголошена в блоці, може бути доступна тільки всередині блоку і всіх внутрішніх блоків цього блоку. Наприклад, наступна програма виробляє помилку компілятора.

```
int main(){
 {
   int x = 10;
}
{
   printf("%d", x);  // Error: x is not accessible here
}
  return 0;
}
```

Результат:

```
error: 'x' undeclared (first use in this function)
```

Змінні оточення (середовища) С:

Змінна середовища - це змінна, яка буде доступна для всіх програм та застосувань С.

Ми можемо отримати доступ до цих змінних з будь-якої точки програми С, не оголошуючи і не ініціалізуючи у програмі.

Вбудовані функції, які використовуються для доступу, зміни та встановлення цих змінних оточення, називаються функціями середовища.

снують 3 функції, які використовуються для доступу, зміни і присвоєння змінної середовища в С. Це

1.setenv() (int setenv(char* envname))

Функція setenv () повинна оновлювати або додавати змінну в середовищі, де викликається процес. Аргумент envname вказує на рядок, що містить назву змінної середовища, яку потрібно додати або змінити. Змінна середовища повинна бути задана значенням, до якого належать точки envval. Функція повинна вийти з ладу, якщо envname вказує на рядок, який містить символ '='. Якщо змінна середовища з іменем envname вже існує, а значення, що перезаписується не є нулем, функція повертає успіх, і середовище має бути оновлено. Якщо змінна середовища з іменем envname вже існує, а значення перезапису дорівнює нулю, функція повертає успіх, а середовище залишається незмінним.

Якщо програма змінює середовище або покажчики, на які вона вказує, поведінка setenv () не визначена. Функція setenv () повинна оновлювати список покажчиків, на які вказуються точки оточення.

Рядки, описані envname і envval, копіюються цією функцією.

Функція setenv () не повинна бути повторно вбудованою. Функція, яка не повинна бути повторно введеною, не обов'язково має бути потокобезпечною.

Після успішного завершення повертається нуль. В іншому випадку повертається -1, errno встановлюється для вказівки помилки, і середовище не повинно бути зміненим.

2.getenv() (char* getenv(char* name))

name - це рядок С, що містить ім'я запитуваної змінної.

Ця функція повертає С-рядок(що закінчується нулевим символом) із значенням запитаної змінної середовища або NULL, якщо ця змінна середовища не існує

3. putenv()
Функція putenv () встановлює значення змінної оточення, змінюючи існуючу змінну або створюючи нову. Параметр varname вказує на рядок форми var = x, де x - нове значення змінної середовища var.
Ім'я не може містити порожній символ або символ рівного (=). Наприклад,
PATH NAME = / my_lib / joe_user
недійсний через пробіл між PATH і NAME. Аналогічно,
PATH = NAME = / my_lib / joe_user
не дійсний через символ рівності між PATH і NAME. Система інтерпретує всі символи, що йдуть за першим рівним символом, як значення змінної середовища.
Функція putenv () повертає 0 у разі успіху. Якщо putenv () не вдається, то повертається -1 і errno встановлюється для вказівки помилки.

Приклад getenv() в C:
Ця програма отримує значення змінної оточення. Припустимо, що DIR визначено як "/usr/bin/test/".

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
printf("Directory = %s\n",getenv("DIR"));
return 0;
}
```

Результат:

```
/usr/bin/test/
```

Приклад програми з setenv() в C:
Ця програма присвоює значення змінної оточення. Припустимо, що змінна FILE присвоюється "/usr/bin/example.c"

```
#include <stdio.h>
#include <stdlib.h>
int main(){
setenv("FILE", "/usr/bin/example.c",50);
printf("File = %s\n", getenv("FILE"));
return 0;
}
```

File = /usr/bin/example.c
Приклад putenv() в C:
Ця функція модифікує значення змінній оточення

```
#include <stdio.h>
#include <stdlib.h>
int main(){
setenv("DIR", "/usr/bin/example/",50);
printf("Directory name before modifying = " \"%s\n", getenv("DIR"));
putenv("DIR=/usr/home/");
printf("Directory name after modifying = " \"%s\n", getenv("DIR"));
return 0;
}
```

Результат:

```
Directory name before modifying = /usr/bin/example/
Directory name after modifying = /usr/home/
```

Шляхи передачі значень у функцію

В мовах програмування звичайно використовують два способи передачі параметрів: за значенням і за посиланням. При передачі параметра по значенню аргументом може бути довільний вираз, значення якого і передається в підпрограму. При передачі параметра по посиланню значенням може бути тільки змінна (як проста так і структурована).

В цьому випадку в підпрограму передається не значення змінної, а її адреса, для того, щоб в цю адресу можна було б записати нове значення і тим самим змінити значення змінної, котра передавалась в якості параметра.

В різних мовах використовуються різні рішення. В одних всі аргументи передаються по посиланню; в інших використовуються обидва механізми, і тоді в списку параметрів вказуються спеціальні ключові слова, котрі показують, що значення аргументу може змінюватись підпрограмою.

В мові C++ використано третє рішення, тобто всі аргументи передаються по значенню, а для передачі аргументу по посиланню в підпрограму передається адреса потрібної змінної з допомогою операції отримання адреси об'єкта (&).

Вказівники дозволяють нам в одній змінній зберігати адрес іншої змінної. Для роботи з вказівниками використовуються дві операції: отримання адреси змінної (&) та вибір значення змінної з використанням вказівника (*). Щоб вказати компілятору те, що зміна є вказівником на значення деякого типу, перед іменем змінної ставлять зірочку (*).

У мові C++ визначено декілька способів передачі параметрів і повернення результату обчислень функцій, серед них найбільш широке використання набули:

Виклик функції з передачею параметрів

Виклик функції з передачею параметрів за допомогою формальних аргументів-значень або передача аргументів **за значенням** (Call-By-Value). Це є проста передача копій змінних в функцію. У цьому випадку зміна значень параметрів в тілі функції не змінить значення, що передавались у функцію ззовні (при її виклику)

Виклик функції з передачею **параметрів** полягає у тому, що у функцію передаються не самі аргументи, а їх копії. Ці копії можна змінювати всередині функції, і це ніяк не позначиться на значеннях аргументів, що за межами функції залишаться без зміни, наприклад:

```
void fun (int p) // функция fun()
{++p;
printf(“%d”,p);
}

void main ( ) //---------- головна функция
```

```
{
int x = 10;
fun (x); //---------- виклик функції
printf("%d",x);
}
```

Результат роботи цього фрагмента програми:
p=11, x=10,
оскільки для виклику функції fun(x) до неї передається копія значення, що дорівнює **10**. Всередині функції значення копії змінної збільшується на **1**, тобто **(++p)**, і тому виводиться **p = = 11**, але за межами функції параметр **p** не змінюється. У цьому випадку функція не повертає ніякого значення.
При цьому способі для звертання до функції достатньо написати її ім'я, а в дужках значення або перелік фактичних аргументів. Фактичні аргументи повинні бути записані в тій же послідовності, що і формальні, і мати відповідний тип (крім аргументів за замовчуванням і перевантажених функцій).
Якщо формальними аргументами функції є параметри-значення і в ній не використовуються глобальні змінні, функція може передати у програму, що її викликає, лише одне значення, що записується в операторі return. Це значення передається в місце виклику функції. Достроковий вихід з функції можна також організувати з використанням оператора return.

Виклик функції з передачею адрес параметрів
Виклик функції з передачею адреси за допомогою параметрів-вказівників або передача параметру **за посиланням** (Call-By-Reference). Передається посилання (вказівник) на об'єкт (змінну), що дозволяє синтаксично використовувати це посилання як вказівник і як значення. Зміни, внесені в параметр, що переданий за посиланням, змінюють вихідну копію параметра функції, яка викликається.
Приклад. Цей приклад демонструє відмінність між передачею параметрів за значенням, передачею параметрів за адресою та передачею параметрів за посиланням. Описується функція, що отримує три параметри. Перший параметр (x) передається за значенням. Другий параметр (y) передається за адресою (як покажчик). Третій параметр (z) передається за посиланням.

```
// функція MyFunction
// параметр x - передається за значенням (параметр-значення)
// параметр y - передається за адресою
// параметр z - передається за посиланням
void MyFunction(int x, int* y, int& c){
   x = 8; // значення параметра змінюється тільки в межах тіла функції
   *y = 8; // значення параметра змінюється також за межами функції
   c = 8; // значення параметра змінюється також за межами функції
   return;
```

```
}
```
Демонстрація виклику функції MyFunction() з іншого програмного коду:
```
int a, b, c;
a = b = c = 5;
// виклик функції MyFunction()
// параметр a передається за значенням a->x
// параметр b передається за адресою   b->y
// параметр c передається за посиланням c->z
MyFunction(a, &b, c); // на виході a = 5; b = 8; c = 8;
```
Як видно з результату, значення змінної a не змінилось. Тому що, змінна a передавалась у функцію MyFunction() з передачею значення (перший параметр). Однак, значення змінної b після виклику функції MyFunction() змінилось. Це пов'язано з тим, що в функцію MyFunction() передавалось значення адреси змінної b. Маючи адресу змінної b в пам'яті комп'ютера, всередині функції MyFunction() можна змінювати значення цієї змінної з допомогою покажчика y. Також змінилось значення **c** після виклику функції. Посилання є адресою об'єкту в пам'яті. З допомогою цієї адреси можна мати доступ до значення об'єкта.

Виклик функцій з передачею даних за допомогою глобальних змінних

Використовувати глобальні змінні для передачі даних між функціями дуже легко, оскільки вони видимі в усіх функціях, де описані локальні змінні з тими ж іменами. Але такий спосіб не є поширеним, тому що ускладнює налагодження програми та перешкоджає розташуванню функції у бібліотеці загального використання. Слід прагнути, щоб функції були максимально незалежними, а їхній інтерфейс повністю визначався прототипом функції. Наведемо приклад використання глобальних змінних:
```
#include < stdio.h >
int a, b, c;   // глобальні параметри
int sum ( );   //---------- прототип функції

int main ( ) {
 scanf("%d %d",&a,&b);
sum();  //---------- виклик sum()
printf("c=%d",c);
}

int  sum( )   //---------- функція sum()
{ c = a + b; }
```

Виклик функцій з застосуванням параметрів, що задані за замовчуванням, при цьому можна використовувати або всі аргументи, або їх частину (C++).
В сучасних версіях мови C++ з'явилася можливість передавати дані за замовчуванням. У цьому випадку при написанні функції всім аргументам або декільком з них присвоюються початкові значення і задовольняються такі вимоги: коли якому-небудь аргументу присвоєно значення за замовчуванням, то всі аргументи, що розташовані за ним (тобто записані праворуч), повинні мати значення за замовчуванням. Таким чином, список параметрів поділяється на дві частини: параметри, що не мають значення за замовчуванням, і параметри, що мають такі значення.
У випадку виклику функції для параметрів, що не мають значень за замовчуванням, обов'язково повинен бути фактичний аргумент, а для параметрів, що мають значення за замовчуванням, фактичні аргументи можна опускати, коли ці значення не треба змінювати.
Якщо деякий параметр має значення за замовчуванням та для нього відсутній фактичний аргумент, то і для всіх наступних (тобто записаних пізніше) параметрів фактичні аргументи повинні бути відсутні, тобто їхні значення передаються до функції за замовчуванням, наприклад:

```
// C++ код: аргументи за замовченням
void funct1 (float x, int y, int z = 8)
{ std::cout <<"x = " << x << " y = " << y << "z = " << z << endl; }

void funct2 (float x, int y = 2, int z = 10)
{ std::cout <<"x = " << x << "y = " << y << "z = " << z << endl; }

void funct3 (float x = 3.15, int y = 42, int z = 202)
{ std::cout << "x = " << x << "y = " << y << "z = " << z << endl; }

int main ( ) {
funct1 (2.1, 10); //за замовченням є один аргумент — z
funct2 (9.2); // за замовченням є два аргументи — y, z
funct3 ( ); // за замовченням є всі три аргументи
}
```

На екрані буде виведено:
x = 2.1 y = 10 z = 8
x = 9.2 y = 2 z = 10
x = 3.15 y = 42 z = 202
З цієї ілюстраційної програми видно, що аргумент за замовчуванням — це той аргумент, значення якого задане при описі заголовка функції, а при її виклику його можна не вказувати.
Якщо замість параметра, заданого за замовчуванням при звертанні до функції, записується інше значення фактичного параметра, то значення за замовчуванням

перекривається заданим фактичним значенням. Так, наприклад, в останньому програмному фрагменті при виклику функції **funct2 (13.5, 75);** на екрані буде виведено:
x = 13.5 y = 75 z = 10,
тобто лише **z** — прийнято за замовчуванням.

Використання типів передачі даних
При виклику функції локальним змінним відводиться пам'ять в стеку і проводиться їх ініціалізація. Управління передається першому операторові тіла функції і починається виконання функції, яке продовжується до тих пір, поки не зустрінеться оператор **return** або останній оператор тіла функції. Управління при цьому повертається в точку, наступну за точкою виклику, а локальні змінні стають недоступними. При новому виклику функції для локальних змінних пам'ять розподіляється знов, і тому старі значення локальних змінних втрачаються.
Параметри функції передаються за значенням і можуть розглядатися як локальні змінні, для яких виділяється пам'ять при виклику функції і проводиться ініціалізація значеннями фактичних параметрів. При виході з функції значення цих змінних втрачаються. Оскільки передача параметрів відбувається за значенням, в тілі функції не можна змінити значення змінних в зухвалій функції, що є фактичними параметрами. Проте, якщо як параметр передати покажчик на деяку змінну, то використовуючи операцію реадресації можна змінити значення цієї змінної.
Приклад:
```
/* Неправильне використання параметрів */
 void change (int x, int y) { int k=x; x=y; y=k; }
```
У даній функції значення змінних **x** і **y**, що є формальними параметрами, міняються місцями, але оскільки ці змінні існують тільки усередині функції **change**, значення фактичних параметрів, використовуваних при виклику функції, залишаться незмінними. Для того, щоб мінялися місцями значення фактичних аргументів можна використовувати функцію приведену в наступному прикладі.
Приклад:
```
/* Правильне використання параметрів */
void change (int *x, int *y) { int k=*x; *x=*y; *y=k; }
```
При виклику такої функції як фактичні параметри повинні бути використані не значення змінних, а їх адреси **change (&a,&b);**
В мові С можна описувати змінні, котрі містять вказівники на функції, тобто адреси функцій, і здійснювати звертання до функцій з допомогою цих вказівників.
Приклад:
```
double y;
/* Опис функції */
double linefunc (double x,double a, double b);
double *func;
```

```
func=&linefunc;
y=(*func)(2.4,-5.1,7.);
```
Вказівник на цю функцію можна також передати як аргумент в іншу функцію.

Виклик функції зі змінною кількістю параметрів.

При виклику функції із змінним числом параметрів задається будь-яке необхідне число аргументів. У оголошенні і визначенні такої функції змінне число аргументів задається трикрапкою в кінці списку формальних параметрів або списку типів аргументів.

Всі аргументи, задані у виклику функції, розміщуються в стеку. Кількість формальних параметрів, оголошених для функції, визначається числом аргументів, які беруться із стека і привласнюються формальним параметрам.

Програміст відповідає за правильність вибору додаткових аргументів із стека і визначення числа аргументів, що знаходяться в стеку.

Прикладами функцій із змінним числом параметрів є функції з бібліотеки функцій мови C++, що здійснюють операції введення-виводу інформації (printf, scanf і тому подібне).

Головна функція

Тепер, знаючі синтаксис визначення функцій можна зрозуміти, що таке оце main(), що завжди писалося в програмі – це так звана головна функція. При компіляції програми для її виконання потрібно вказати, яка ж з функцій виконується першою. Зауважимо, що якщо нашою метою є створення бібліотеки, то існування цієї головної функції не є необхідною умовою, тобто програмний код для бібліотеки може складатись лише з набору функцій, які просто може виконувати користувач бібліотеки. У той же час, якщо потрібно створити виконуваний файл, то потрібно якось передати комп'ютеру звідки і як потрібно виконувати функції, що імплементовані в програмному коді, тобто описати точку входу. В деяких мовах програмування компілятор для цього обирає першу команду, що записана в потрібному місті, деякі мови вимагають що користувач сам вказав потрібну точку входу за допомогою командного рядку або середовища, але в мові C потрібно вказати одну і лише одну функцію з назвою main().

Формат цієї команди може бути лише одним з вказаних:

```
int main(){ …}
```

```
int main(int argc, char** argv) {…}
```

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

Тут:

argc – натуральне число, що представляє кількість аргументів, що передаються в програму з зовнішього середовища, тобто як аргументи в команді що

передається з програмного середовища, або командного рядка що запускає програміст.

argv – вказівник на масив рядків (рядків С стилю, тобто рядків, що закінчуються нулевим символом (*null-terminated multibyte strings*)), які передаються з програмного середовища, або командного рядка що запускає програміст. Ці рядки – це розділені пробілами рядки, кількість яких дорівнює argc, вигляду argv[0] … argv[argc-1] . Значення argv[argc] повинно бути нульовим.

Примітка. Зауважимо що деякі компілятори дозволяють писати головну функцію у вигляді

```
void main();
```

Однак, це не є офіційно дозволеною формою, зокрема gcc такої форми не дозволяє.

Примітка 2. Наявність типу головної функції не вимагає (хоча деякі старі компілятори такі вимагають) того, щоб тіло головної функції закінчувалося return <ціле значення>; .

Примітка 3. Крім того, оскільки за замовченням будь-яка функція в С повертає цілий 0(окрім старого до С89 стандарту та навпаки нового С11 стандарту), це може виконуватись автоматично і тому така форма теж можлива

```
main();
```

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

Однак, загальна порада все ж таки писати int перед main, бо деякі стандарти цього вимагають.

Якщо використовується перша форма, тобто якщо функція main() визначена без параметрів, то програма не може отримати доступ до команд і аргументів командного рядку.

Щоб отримати доступ до переданих в програму даними, їх необхідно присвоїти змінним. Оскільки аргументи відразу передаються в main (), то її заголовок повинен виглядати таким чином:
int main (int n, char * arr [])
У першій змінній (n) міститься кількість слів, а в другій - покажчик на масив рядків. Часто другий параметр записують у вигляді ** arr.
Приклад

```
#include <stdio.h>
int main (int argc, char** argv) {
int i;
printf ( "% d \ n", argc);
```

```
for (i = 0; i <argc; i ++){
                puts (argv [i]);
        }
}
```

Програма виводить кількість слів у командному рядку при її виклику і кожне слово з нового рядка. Її можна викликати без аргументів командного рядка та з аргументами.
У програмі ми використовували змінні-параметри argc і argv. Прийнято використовувати саме такі імена, але насправді вони можуть бути будь-якими. Та все ж таки краще дотримуватися цього стандарту, щоб ваші програми були більш зрозумілі не тільки вам, а й іншим програмістам.

Специфікатори змінних

В мові C та C++ перед типом змінної при її визначенні може стояти специфікатор або специфікатори, що визначають або область дії змінної або особливості її зберігання в пам'яті комп'ютера. Їх ділять на два типи: специфікатори зберігання та специфікатори дії:

- відсутніть специфікаторів (auto);
- специфікатор static;
- специфікатор register;
- специфікатор extern.

Відсутність специфікаторів(auto) – застосовується для локальних змінних по замовчуванню. *Область видимості – обмежена блоком, в якому вони оголошені*;

Специфікатор static – застосовується як для локальних, так і для глобальних змінних. Область видимості локальної статичної змінної зберігається після виходу з блока чи функції, де ця змінна оголошена. Під час повторного виклику функції змінна зберігає своє попереднє значення.
Цей специфікатор потрібен при необхідності збереження значень які підраховує функція для їх врахування при наступному виклику функції. Наприклад це використовується для підрахунку виклику функцій чи при створенні випадкових чисел щоб не було повторень значень при новому виклику функції. Тоді їх треба описати як статичні за допомогою службового слова static, наприклад:

```
static int x, y;
static float p = 3.25;.
```

Статична змінна схожа на глобальну, але діє тільки у тій функції, в якій вона оголошена.
При цьому специфікаторі змінна є глобальною в тому сенсі, що вона оголошена за межами будь-якої функції, але приватна для одного вихідного файлу, в якому вона визначена. Така змінна видима для функцій у цьому вихідному файлі, але не

для будь-якої функції в будь-яких інших вихідних файлах, навіть якщо вони намагаються надіслати відповідну декларацію.
Таким чином отримується будь-який додатковий контроль, який може знадобитися під час видимості та тривалості життя, і можна розрізнити визначення примірників і зовнішніх декларацій, використовуючи класи зберігання. **Клас зберігання** – це додаткове ключове слово на початку декларації, яке певним чином змінює декларацію. Як правило, клас зберігання (якщо такий є) є першим словом у декларації, що передує назві типу. (Строго кажучи, цей порядок традиційно не був необхідним, і зустрічається код з класом зберігання, назвою типу та іншими частинами декларації в функції.

Специфікатор register – вказує компілятору, що значення слід зберігати в регістрах процесора (не в оперативній пам'яті). Це зменшує час доступу до змінної, що прискорює виконання програми. *Область видимості – обмежена блоком, в якому вони оголошені.*

Специфікатор extern – використовується для передачі для передачі значень глобальних змінних з одного файлу в інший (часто великі програми складаються з кількох файлів). Область дії – всі файли, з яких складається програма.

Підсумуємо використання специфікаторів зберігання

```
int globalvar = 1;
extern int anotherglobalvar;
static int privatevar;
f(){
  int localvar;
  int localvar2 = 2;
  static int persistentvar;
}
```

Тут ми маємо шість змінних, три оголошені зовні і три оголошені всередині функції f ().globalvar - глобальна змінна. Декларація, яку ми бачимо, є її визначальним екземпляром (також відбувається включення початкового значення).
globalvar може бути використаний в будь-якому місці цього вихідного файлу, і він також може бути використаний в інших вихідних файлах (якщо відповідні зовнішні декларації видаються в інших вихідних файлах).
anotherglobalvar є другою глобальною змінною. Тут не визначено визначальний екземпляр для нього (і його ініціалізація) знаходиться десь ще.
privatevar є "приватною" глобальною змінною. Він може використовуватися в будь-якому місці цього вихідного файлу, але функції в інших вихідних файлах не можуть отримати до нього доступ, навіть якщо вони намагаються видати

зовнішні декларації для нього. (Якщо інші вихідні файли намагаються оголосити глобальну змінну під назвою "**privatevar**", вони отримають свої власні, вони не будуть обмінюватися цією.) Оскільки вона має статичну тривалість і не отримує явної ініціалізації, **privatevar** буде ініціалізовано 0.
localvar є локальною змінною у функції f (). Доступ до нього можна отримати тільки в межах функції f (). (Якщо будь-яка інша частина програми оголошує змінну з ім'ям "**localvar"**, то ця змінна буде відмінною від тієї, яку ми розглядаємо тут.). Змінна **localvar** концептуально створюється кожного разу, коли f () викликається, і видаляється, коли f () повертає значення. Будь-яке значення, яке було збережено в **localvar** в останній раз, коли f () було запущено, буде втрачено і не буде доступним наступного разу, коли f () викликається. Крім того, оскільки він не має явного ініціалізатора, значення **localvar** буде взагалі сміттям кожного разу, коли f() викликається.
localvar2 також локальна змінна, і все, що ми говорили про **localvar**, застосовується до нього, за винятком того, що, оскільки його декларація включає в себе явний ініціалізатор, вона буде ініціалізована 2 кожного разу, коли f () викликається.
Нарешті, **persistentvar** знову локальна змінна до f (), але він зберігає своє значення між викликами до f (). Він має статичну тривалість, але не має явного ініціалізатора, тому його початкове значення буде 0.

Специфікатори доступу

Крім специфікаторів зберігання використовуються також специфікатори доступу:

- **специфікатор const** - цей специфікатор вказує, що даній змінна не може бути модифікована в процесі роботи з нею і залишиться константою;
- **специфікатор volatile** – застосовується до глобальних змінних, значення яких можуть надходити від периферійних пристроїв (наприклад від системного таймера);
- **специфікатор mutable** – застосовується для відміни специфікатору const.

Примітка. З стандарту С11 додано що специфікатор _Thread_local для керування доступом в багатопотокових застосуваннях.

Специфікатори volatile та mutable зустрічаються не надто часто, volatile частіше за все в програмах яким потрібно щось робити на низькому рівні, а mutable частіше за все використовується для швидких переробок старого коду, що буває інколи не сумісний з певним використанням специфікатору const.
А ось як раз специфікатор const використовується часто і навіть завжди рекомендується к використанню:
а) коли потрібно визначити змінну що є реальною константою, наприклад число Пі, чи кількість елементів певного масиву;

б) коли в формальних аргументах є структура, клас або масив чи вказівник, що не повинні змінюватись протягом виконання функції. Саме в таких випадках використання const інколи є не лише гарним стилем коду, але й суворою вимогою.

Специфікатори функцій

Деякі специфікатори можуть також відноситись і до функцій:

- специфікатор static;
- специфікатор extern;
- специфікатор inline.

Статичні функції (Static C functions)

У С функції за замовчуванням є глобальними. Ключове слово "static" перед іменем функції робить його статичним. Наприклад, нижче функція fun () є статичною.

```
static int fun(void){
        printf("I am a static function ");
}
```

На відміну від глобальних функцій у С, доступ до статичних функцій обмежується файлом, де вони оголошені. Тому, коли ми хочемо обмежити доступ до функцій, їх роблять статичними. Іншою причиною для використання статичних функцій може бути повторне використання тієї ж назви функції в інших файлах.

Наприклад, якщо зберегти наступну програму в одному файлі file1.c:

```
/* Всередині file1.c */
static void fun1(void){
  puts("fun1 called");
}
```

Та записати наступну функцію в file2.c :

```
/* Всередині file2.c */
int main(void){
  fun1();
  getchar();
  return 0;
}
```

Тепер, якщо ми компілюємо вищевказаний код з командою “gcc file2.c file1.c”, ми отримаємо помилку “undefined reference to `fun1”. Це тому, що функцію fun1 () оголошено статичною функцією у файлі1.c і не може бути використано у файлі2 .c.

Зовнішні функції (Extern function)

Цей специфікатор означає протилежну властивість до «static». За замовчуванням, декларації та визначення функцій в С оголошують властивість "extern" до них.

Це означає, що навіть якщо ми не використовуємо extern з оголошенням / визначенням функцій C, він присутній. Наприклад, коли ми пишемо.

```
int foo(int arg1, char arg2);
```

На початку присутній extern, що прихований, і компілятор розглядає його, як показано нижче.

```
extern int foo(int arg1, char arg2);
```

Те ж саме відбувається і з визначенням функції C (визначення функції C означає написання тіла функції). Отже, коли ми визначаємо функцію C, екстерн присутній у початку визначення функції. Оскільки декларація може виконуватися будь-яку кількість разів і визначення можна робити лише один раз, то оголошення функції можна додавати в декількох файлах C / H або в одному файлі C / H кілька разів. Але ми помічаємо фактичне визначення функції тільки один раз (тобто тільки в одному файлі). Оскільки extern розширює видимість всієї програми, функції можуть використовуватися (декларуватися) в будь-якому файлі всієї програми, якщо відома функція. Таким чином, знаючи декларацію функції, компілятор C знає, що визначення функції існує і йде вперед для компіляції програми.
Незважаючи на те, що специфікатор extern додається автоматично, гарним стилем є використання цього специфікатора при декларації функції в заголовочному файлі.
Підсумовуючи:
1. Декларація може бути зроблена будь-яку кількість разів, але визначення тільки один раз.
2. Ключове слово "extern" використовується для розширення видимості змінних / функцій ().
3. Оскільки функції відображаються по всій програмі за замовчуванням. Використання extern не потрібне для оголошення / визначення функції. Його використання є надмірним.
4. Коли extern використовується з змінною, вона оголошена але не визначена.
5. Як виняток, коли змінна extern оголошується з ініціалізацією, вона також приймається як визначення змінної.

Inline Function

Inline Function (__inline в деяких компіляторах) це ті функції, **чиї** визначення невеликі і автоматично підставляються в місця, де відбувається кожен виклик функції. Підстановка функцій є повністю вибором компілятора.

```
#include <stdio.h>

// Inline function in C
inline int foo()
```

```
{
   return 2;
}

// Driver code
int main()
{

   int ret;

   // inline function call
   ret = foo();

   printf("Output is: %d\n", ret);
   return 0;
}
```

Результат - Compiler Error:

```
In function `main':
undefined reference to `foo'
```

Це один з побічних ефектів GCC, як він обробляє функцію inline. При компіляції GCC виконує вбудоване заміщення як частину оптимізації. Отже, в головному не існує жодного виклику функції (foo).

Як правило, обсяг файлу GCC є "не зовнішнім лінкером". Це означає, що вбудована функція ніколи не надається лінкеру, який викликає помилку лінкера, згадану вище. Щоб вирішити цю проблему, використовуйте “static” перед вбудованим. Використання статичного ключового слова змушує компілятор враховувати цю вбудовану функцію в компонувальнику, і, отже, програма компілюється і виконується успішно.

```
#include <stdio.h>

// Inline function in C
static inline int foo()
{
return 2;
}
// Driver code
int main()
{
int ret;
// inline function call
ret = foo();
printf("Output is: %d\n", ret);
 return 0;
}
```

Output:

```
Output is: 2
```

Перетворення типів даних

Автоматичні перетворення

С виконує автоматичні перетворення типу даних у наступних чотирьох ситуаціях:

• Коли у виразі з'являються два або більше операндів різних типів.

• Коли аргументи типу char, short і float передаються функції, використовуючи стару декларацію стилю.

• Якщо аргументи, які не відповідають точно параметрам, оголошеним у прототипі функції, передаються функції.

• Коли тип даних операнда навмисно перетворений оператором операції.

Звичайні арифметичні перетворення

Наступні правила, які називаються звичайними арифметичними перетвореннями, регулюють перетворення всіх операндів у арифметичні вирази. Ефект полягає в доведенні операндів до загального типу, який також є типом результату. Правила регулюються в такому порядку:

1. Якщо будь-який з операндів не має арифметичного типу, то перетворення не виконується.

2. Якщо будь-який з операндів має тип long double, інший операнд перетворюється на long double.

3. В іншому випадку, якщо один з операндів має тип double, інший операнд перетворюється на double.

4. В іншому випадку, якщо один з операндів має тип float, інший операнд перетворюється у float.

5. В іншому випадку інтегральні промо-акції виконуються на обох операндах і застосовуються наступні правила:

1. Якщо будь-який з операндів має тип unsigned long int, то інший операнд перетворюється на довгий int.

В іншому випадку, якщо один операнд має тип long int, а інший має тип unsigned int, і якщо довгий int може представляти всі значення unsigned int, операнд типу unsigned int перетворюється на long int. Якщо довгий int не може представляти всі значення непідписаного int, обидва операнди перетворюються в довгу int.

3. В іншому випадку, якщо один з операндів має тип long int, інший операнд перетворюється на long int.

4. В іншому випадку, якщо один з операндів має тип unsigned int, інший операнд перетворюється в unsigned int.

5. В іншому випадку обидва операнда мають тип int.

Звичайні правила арифметичного перетворення.

Символи та цілі числа

Поле char, short int або int, або підписаний або без знака, або об'єкт, що має тип переліку, можна використовувати у виразі, де дозволено int або unsigned int. Якщо int може представляти всі значення вихідного типу, значення перетворюється на int. В іншому випадку, він перетворюється в unsigned int. Ці правила перетворення називаються інтегральними промо-акціями.

Ця реалізація інтегрального просування називається збереженням значень, на відміну від непідписаного збереження, в якому непідписані char і unsigned short розширюються до unsigned int. DEC C використовує підтримку збереження цінностей, як це вимагається стандартом ANSI C, якщо не вказано загальний режим C.

Щоб допомогти знайти арифметичні перетворення, які залежать від незмінених правил збереження, DEC C, з включеною опцією перевірки, позначає будь-які інтегральні промоції

непідписаних символів і непідписаних коротких до int, які можуть впливати на підхід збереження цінності для інтегральних промоцій.
Всі інші арифметичні типи не змінюються інтегральними промоціями.
У DEC C змінні типу char - це байти, що розглядаються як ціле число. Коли довше ціле число перетворюється в коротше ціле або до char, воно урізається зліва; надлишкові біти відкидаються. Наприклад:

```
int i;
char c;
i = 0xFFFFFF41;
c = i;
```

Цей код призначає hex 41 ('A') c. Компілятор перетворює коротші цілі числа, що підписуються, на більш довгі з розширенням знаків.
Натуральні та звичайні цілі
Переходи також відбуваються між різними типами цілих чисел.
Коли значення з інтегральним типом перетворюється в інший цілий тип (наприклад, int перетворено в long int) і значення може бути представлено новим типом, значення не змінюється.
Коли знакове ціле число, перетворюється на ціле беззнакове число, що дорівнює або більшому розміру, а значення цілого знаку невід'ємне, його значення не змінюється. Якщо ціле значення відємне, то:
• Якщо тип беззнакового цілого є більшим, спочатку ціле число перетворюється в ціле число, яке відповідає цілому числу без знаку; потім значення перетворюється в беззнакове, додаючи до нього більше, ніж найбільше число, яке може бути представлено в цілому цілому типу.
• Якщо число цілого беззнакового типу є рівним або меншим, ніж тип цілого, що перетоворюється, то значення перетворюється в беззнаковий, додаючи до нього одне більше, ніж найбільше число, яке може бути представлено в цілому цілому типу.
Коли ціле значення знижується до цілого беззнакового числа меншого розміру, результатом є невід'ємний залишок значення, поділений на номер один, більший за найбільше представлене значення без знака для нового інтегрального типу.
Коли ціле значення знижується до цілого числа меншого розміру, то ціле число перетворюється з нього і як це відбудеться стандарт нам не гарантує.

Дійсні та цілі типи (Floating and Integral)

Коли операнд з дійсним типом перетворюється в ціле число, дробову частину відкидають. Коли значення плаваючого типу має бути конвертовано під час компіляції до цілого чи іншого плаваючого типу, і результат не може бути представлений, компілятор повідомляє про застереження в таких випадках:
Перетворення в unsigned int і результат не може бути представлений непідписаним типом int. Перетворення має тип, відмінний від unsigned int, і результат не може бути представлений типом int. Коли значення інтегрального типу перетворюється в дійсний тип, а значення знаходиться в діапазоні значень, які можуть бути представлені, але не зовсім точно, результатом перетворення є або наступне більш високе або наступне нижнє значення, в залежності від того, що є природним результатом перетворення на даному апаратному забезпеченні. Потрібно дивитися документацію для результатів перетворення на вашій платформі.
Дійсні типи
Якщо в виразі з'являється операнд типу float, він розглядається як об'єкт з однією точністю, якщо вираз не включає об'єкт типу double або long double, і в цьому випадку застосовується звичайне арифметичне перетворення.

Коли дійсне число підвищується до подвійного або довгого подвійного, або подвійний підвищується до довгого подвійного, його значення не зміниться. Поведінка є невизначеною, коли double понижений до float чи long double або double до float, якщо значення, що перетворюється, знаходиться поза діапазону значень, які можуть бути представлені.
Якщо значення, яке перетворюється, знаходиться в діапазоні значень, які можуть бути представлені, але не зовсім точно, результат округлюється до наступного більш високого або наступного нижчого значення, яке може бути представлено.

Перетворення вказівників

Хоча два типи (наприклад, int і long) можуть мати однакове представлення, вони все ще різні типи. Це означає, що вказівник на int не може бути присвоєно вказівнику на long без використання приведення. Вказівник на функцію одного типу також не може бути призначений покажчику на функцію іншого типу без використання приведення. Крім того, вказівники на функції, які мають різну інформацію про тип параметрів, включаючи відсутність інформації про тип параметрів старого стилю, є різними типами. У цих випадках, якщо приклад не використовується, компілятор видає помилку. Оскільки існують обмеження на вирівнювання для деяких цільових процесорів, доступ через невизначений покажчик може призвести до набагато більш повільного часу доступу або виключення.
Вказівник на void може бути перетворений в або з покажчика на будь-який неповний або об'єктний тип. Якщо вказівник на будь-який неповний або об'єктний тип перетворюється на вказівник на void і назад, результат порівнюється з вихідним покажчиком.
Інтегральна постійна виразу, що рівна 0, або такий вираз, що передається типу void *, називається константою нульового покажчика. Якщо константа нульового вказівника присвоюється або порівнюється для рівності з покажчиком, константа перетворюється на покажчик цього типу. Такий покажчик називається нульовим покажчиком і гарантовано порівнюється з нерівним покажчиком на будь-який об'єкт або функцію.
Вказівник на масив автоматично перетворюється на вказівник на тип масиву, де він вказує на перший елемент масиву.

Перетворення функцій аргументу

Передбачається, що типи даних аргументів функції відповідають типам формальних параметрів, якщо не існує декларація про прототип функції. При наявності прототипу функції всі аргументи у виклику функції порівнюються для сумісності призначення з усіма параметрами, заявленими в декларації прототипу функції. Якщо тип аргументу не відповідає типу параметра, але є сумісним з призначенням, С перетворює аргумент на тип параметра. Якщо аргумент у виклику функції не є призначенням, сумісним з параметром, оголошеним у декларації прототипу функції, генерується повідомлення про помилку.
Якщо прототипу функції немає, всі аргументи типу float перетворюються на double, всі аргументи типу char або short перетворюються на тип int, всі аргументи типу unsigned char і unsigned short перетворюються в unsigned int, а масив або назва функції перетворюється в адресу імені масиву або функції. Компілятор не виконує ніяких інших перетворень автоматично, а будь-які невідповідності після цих перетворень є помилками програмування.
Вказівник функції - це вираз, який має тип функції. За винятком випадків, коли це операнд оператора sizeof або unary & operator, призначення функції з типом "function returning type" перетворюється на вираз, який має тип "вказівник на функцію повернення типу".

Перетворення типів

Перетворення типів (type casting) - це спосіб перетворення змінної з одного типу даних в інший тип даних. Наприклад, якщо ви хочете зберегти значення 'long' у просте ціле число, ви можете ввести cast 'long' до 'int'. Ви можете перетворити

значення з одного типу на інший, явно використовуючи оператор перетворення (cast) наступним чином

(type_name) вираз

Розглянемо наступний приклад, коли оператор перетворення викликає ділення однієї цілої змінної на іншу, як операцію з дійсними числами:

```
#include <stdio.h>

int main() {
  int sum = 17, count = 5;
  double mean;
  mean = (double) sum / count;
  printf("Value of mean : %f\n", mean );
}
```

Коли вищезгаданий код компілюється і виконується, він дає наступний результат:

```
Value of mean: 3.400000
```

Слід зазначити, що оператор перетворення типу (тип) має пріоритет над діленням, тому значення суми спочатку перетворюється на тип double і, нарешті, ділиться на кількість, даючи double.

Перетворення типу можуть бути неявними, які виконуються компілятором автоматично, або можуть бути вказані явно через використання оператора cast. Вважається гарною практикою програмування використання оператора cast, коли необхідні перетворення типу.

Просування(промошн) в цілому

Цілеспрямоване просування - це процес, за допомогою якого значення цілочисельного типу "менше", ніж int або unsigned int, перетворюються в int або unsigned int. Розглянемо приклад додавання символу з цілим числом

```
#include <stdio.h>
int main() {
  int  i = 17;
  char c = 'c'; /* ascii value is 99 */
  int sum;
  sum = i + c;
  printf("Value of sum : %d\n", sum );
}
```

Коли вищезгаданий код компілюється і виконується, він дає наступний результат:

```
Value of sum: 116
```

Тут значення sum складає 116, оскільки компілятор робить ціле просування і перетворює значення 'c' в ASCII перед виконанням фактичної операції додавання.

Звичайне арифметичне перетворення

Звичайні арифметичні перетворення неявно виконуються для передачі їх значень загальному типу. Компілятор спочатку виконує ціле просування; якщо операнди

все ще мають різні типи, то вони перетворюються на тип, який виглядає найвищим у наступній ієрархії

bool -> char -> short int -> int ->unsigned int -> long -> unsigned ->long long -> float -> double -> long double

Звичайні арифметичні перетворення не виконуються для операторів присвоєння, ані для логічних операторів && і ||. Візьмемо наступний приклад, щоб зрозуміти концепцію

```
#include <stdio.h>

int main() {

  int  i = 17;
  char c = 'c'; /* ascii value is 99 */
  float sum;

  sum = i + c;
  printf("Value of sum : %f\n", sum );
}
```

Коли вищезгаданий код компілюється і виконується, він дає наступний результат:

Value of sum : 116.000000

Тут легко зрозуміти, що перший с перетворюється в ціле число, але оскільки кінцеве значення є подвійним, застосовується звичайне арифметичне перетворення, і компілятор перетворює і та с у 'float' і додає їх, даючи результат 'float'.

Передача масивів у функцію

Якщо в якості параметру функції використовується позначення масиву, необхідно передати до функції його розмірність.

*Приклад 3:*Обчислення суми елементів масиву

```
int sum (int n, int a[] ){
        int i, s=0;
        for( i=0; i<n; i++ )
                s+=a[i];
        return s;
}
int main(){
        int a[]={ 3, 5, 7, 9, 11, 13, 15 };
        int s = sum( 7, a );
        printf("s=%d",s);
}
```

Рядки в якості фактичних параметрів можуть визначатися або як одновимірні масиви типу char[],або як вказівники типу char*. На відміну від звичайних масивів, для рядків немає необхідності явно вказувати довжину рядка, оскільки будь-який рядок обмежується нуль-символом.
При передачі у функцію двовимірного масиву в якості параметру так само необхідно задавати кінцеві розміри масиву у заголовку функції. Робити це можна:

- а) явним чином (a[3][4] тоді функція працюватиме з масивами лише заданої розмірності);
- б) можна спробувати для квадратної матриці через додатковий параметр ввести розмірність (void matrix(double x[][], int n)), де n – порядок квадратної матриці, а double x[][]– спроба визначення двовимірного масиву зі заздалегідь невизначеними розмірами. В результаті на таку спробу компілятор відповість:
  Error…: Size of type is unknown or zero;
- в) найзручнішим вважається спосіб представлення та передачі двовимірної матриці за допомогою допоміжних масивів вказівників на одновимірні масиви, якими в даному випадку виступають рядки двовимірного масиву. Всі дії виконуються в межах рядків, розмір яких передається у функцію за допомогою додаткового формального параметру або з використанням глобальних змінних.

Приклад 4.

```
# include<stdio.h>
//Функція транспонування квадратної матриці
void trans (int n, double*p[])
{double x;
 for (int i=0; i<n-1; i++)
     for(int j=i+1; j<n; j++)
        {x=p[i][j];
        p[i][j]=p[j][i];
        p[j][i]=x;
    }
}
int main(){
// Задано масив для транспонування
double A[4][4]={11, 12, 13, 14
21, 22, 23, 24
31, 32, 33, 34
41, 42, 43, 44};
// Допоміжний двовимірний масив вказівників
double * ptr[]={(double*)&A[0], (double*)&A[1],
(double*)&A[2], (double*)&A[3]};
// Виклик функції
```

```
int n=4;
trans(n, ptr);
for(int i=0; i<n;i++){
        printf(”\n Рядок %d : ”,(i+1)”);
                for(int j=0; j<n; j++){
                        printf(”\t %3.3f ”, A[i][j]);
                }
    }
}
```

5) Типи даних, що визначені користувачем. Структури. Робота з файлами

Структури. Декларація та ініціалізація структур. Анонімна структура. Визначення власного типу (ключове слово typedef). Вказівники на структури. Робота з файлами. Текстові та бінарні файли. Відкриття файлів та файлові змінні.
Робота з символьними та текстовими файлами.
Робота з бінарними файлами.

Структури

Структура — це сукупність різнотипних елементів або логічно зв'язаних змінних, в який входять елементи будь-яких типів (на старому С - за винятком функцій, хоча можна використовувати вказівник на функції, на сучасному С та С++ можна використовувати й функції), яким присвоюється одне ім’я (на мові С воно може бути відсутнім – так звана *анонімна структура*) для зручності подальшої обробки , що займає одну ділянку пам’яті, тобто визначено як окремий тип. Елементи, що складають структуру, називаються *полями*.
На відміну від масиву, який є однорідним об'єктом, структура може бути неоднорідною.
Традиційним прикладом структури служить облікова картка того, що працює: службовець підприємства описується набором атрибутів, таких, як табельний номер, ім'я, дата народження, стать, адрес, зарплата. У свою чергу, деякі з цих атрибутів самі можуть виявитися структурами. Такі, наприклад: ім'я, дата народження, адрес, що мають декілька компонент.

Декларація структури

Змінна типу структура, як і будь-яка змінна, повинна бути описана. Цей опис складається з двох кроків: опису шаблону (тобто складу) або типу структури та опису змінних структурного типу.

Синтаксис декларації структури:
struct [<назва>] {
<список полів структури>}
<декларація1>
[, <декларація2> ...];
або
struct <назва> <декларація1> [, <декларація2> ...];
Декларація ***структури*** задає ім'я типу структури і специфікує послідовність змінних величин, що називаються полями (елементами, членами) структури. Ці поля структури можуть мати різні типи.
Декларація структури починається з ключового слова struct і має дві форми подання, як показано вище. У першій формі подання типи і імена елементів структури специфікуються в списку декларацій елементів < **список полів структури** >. < **назва** > - це ідентифікатор, який іменує тип структури, визначений у списку декларацій елементів.
Кожен <**декларація**> задає ім'я змінної типу структури. Тип змінної в декларatorів може бути модифікований на покажчик до структури, на масив структур або на функцію, яка повертає структуру.
Друга синтаксична форма використовує тег <**назва**> структури для посилання на тип структури. У цій формі декларації відсутній список декларацій елементів, оскільки тип структури визначений в іншому місці. Визначення типу структури має бути видимим для тегу, який використовується в декларації і визначення повинне передувати декларації через тег, якщо тег не використовується для декларації вказівника або структурного типу typedef. В останніх випадках декларації можуть використовувати тег структури без попереднього визначення типу структури, але все ж визначення повинне знаходитися в межах видимості декларації.
Список декларацій елементів < **список полів структури** > - це одне або більше декларацій змінних або бітових полів. Кожна змінна, що об’явлена в цьому списку, називається елементом структурного типу. Декларації змінних списку мають той же самий синтаксис, що і декларації змінних за винятком того, що вони не можуть містити специфікаторів класу пам'яті або ініціалізаторів. Елементи структури можуть бути будь-якого типу: будь-якого базового, масивом, вказівником, об’єднанням або структурою.
Єдине обмеження - поле не може мати тип батьківської структури, в який воно описано. Однак, поле може бути описано, як вказівник на тип структури, до якої він входить, дозволяючи створювати рекурсивні структури, наприклад списки.
Таким чином, опис структури має вигляд:
struct [<ім’я структури>]
{ <тип 1> ім’я поля 1;
<тип 2> ім’я поля 2 . . .;
} p1, p2 . . .;

де **struct** — службове слово;

- **<ім'я структури>** — ім'я типу структура (може бути відсутнім);
- **<тип 1>, <тип 2>** — імена стандартних або визначених типів;
- **ім'я поля 1, ім'я поля 2,…** — імена полів структури;
- **p1, p2 . . .;** — імена змінних типу структура.

Таким чином, створюється змінна нового типу **struct [<ім'я структури>] який можна використовувати в програмі як новий тип, передавати як аргумент та повертати з функції.** ***Однак, зауважимо, що оскільки це є новим типом для нього за замовченням не визначено жодної стандартної функції окрім присвоювання і таким чином потрібно для нової структури визначати потрібні операції додатково*****.**

Примітка: На C++ можна користуватись структурою без використання ключового слова **struct [<ім'я структури>] , а лише вказавши <ім'я структури> у якості типу нової змінної у декларації.**

Наприклад, для зберігання інформації по успішності студента з дисципліни «Програмування» визначимо таку структуру:

```
struct Student  // визначаємо структуру з назвою Student

{ char  name [25]; //  Поле 1:  Прізвище та ініціали – тип рядок з 25 символів
  char group[3]; //  Поле 2:  Група – тип рядок з 3 символів
int pract_mark; //  Поле 3:  Бали за практику – тип ціле
int course_project1; //  Поле 4:  Бали за перший проект – тип ціле
int course_project2; //  Поле 5:  Бали за другий проект – тип ціле
float additional_mark; // Поле 6: середній додатковий бал  -тип дійсне число
}  st1, st2;
```

Ця декларація може бути зроблено як локально так і глобально, і це означає що ми створили дві змінні типу struct Student.

Змінні **st1 і st2** можна оголосити окремим оператором, наприклад:

```
// створюємо глобально структуру з назвою Student
struct Student  {
char  name [25]; //  Поле 1:  Прізвище та ініціали – тип рядок з 25 символів
char group[3]; //  Поле 2:  Група – тип рядок з 3 символів
int pract_mark; //  Поле 3:  Бали за практику – тип ціле
int course_project1; //  Поле 4:  Бали за перший проект – тип ціле
int course_project2; //  Поле 5:  Бали за другий проект – тип ціле
float additional_mark; // Поле 6: середній додатковий бал  -тип дійсне число
} ;
```

Далі в коді глобально або локально визначаємо відповідні змінні

```
struct Student stl, st2;.
```

Приведена в прикладі структура має три частини, які називаються **елементами** або **полями**. Для того, щоб працювати зі структура необхідно засвоїти три основні прийоми:
- встановлення формату структури;
- визначення змінної, яка відповідає даному формату;
- забезпечення доступу до окремих компонентів змінної-структури.

Розглянемо наступний приклад.

```
struct book {
        char nazva[maxnazva];
        char avtor[maxavtor];
        float cina;
};
```

Опис структури, що складається з взятого в фігурні дужки списку описів, починається з службового слова **struct.** За словом struct може записуватися необов'язкове ім'я, яке називається назвою структури (тут це book). Ярлик іменує структури і може використовуватись надалі як скорочений запис докладного опису. Список полів(елементів) знаходиться в фігурних дужках. Кожне поле має свою назву. Після визначення кожного елемента ставиться крапка з комою. Поле структури має будь-який тип даних, а також може включати в себе інші структури. Опис структури завершується крапкою з комою. Опис структури може бути розташований ззовні функції і всередині. Якщо опис поміщено всередину функції, то структура використовується лише всередині функції. Поняття структура " може використовуватись в двох значеннях. Одно з них-**шаблон. Шаблон** вказує компілятору, як представити дані, але для них не виділяється пам'ять, він лише визначає форму структури.

Імена елементів і тегів без яких-небудь колізій можуть співпадати з іменами звичайних змінних (тобто не елементів), оскільки вони завжди помітні по контексту. Більш того, одні і ті ж імена елементів можуть зустрічатися в різних структурах, хоча, згідно хорошого стилю програмування, краще однакові імена давати тільки близьким по сенсу об'єктам.

Приклад:

Так, наприклад для анкети службовця можна вибрати такі імена:

tab_nom - табельний номер;
pib - прізвище, ім'я, по батькові;
stat- стать;
summa - зарплата;

Всі ці поняття можна об'єднати в таку, наприклад, структуру:

```
struct anketa {
        int tab_nom;
        char fio[30];
        char data[10];
        int pol;
```

```
        char adres[40];
        float summa;
        };
```
Друге значення, поняття “структура” - це змінна - структура, яка створюється на наступному етапі. Створення структурної змінної робиться за допомогою такого опису:
```
struct book odna;
```
Обробляючи цей оператор, компілятор створює змінну odna і використовуючи шаблон book виділяє пам’ять для двох символьних масивів змінної дійсного типу. При визначенні змінної-структури шаблон book відіграє таку ж роль, як слова int і float для більш простих описів. Для комп’ютера визначення
```
struct book odna;
```
є скороченим варіантом опису:
```
struct book {
        char nazva[maxnazva];
        char avtor[maxavtor];
        float cina;
} odna;
```
Можна визначити декілька змінних–структур або вказівник на цей вигляд структури:
```
struct book odna, libry, *pbook;
```
Ще один класичний приклад:
1) // Декларація структури Точка.
```
struct Point
{
   int x, y;  // однотипні поля можна обєднувати як змінні через кому
} p1;  // Змінна p1 декларована як 'Point'
```

2)
// Створення нового типу Точка як базового типу
```
struct Point {
        int x, y;
};
int main() {
  struct Point p1;  // Змінна p1 декларована як звичайна змінна
}
```
Примітка: В C++ ключове слово struct *необов’язково* при декларації змінної. В С воно *обов’язкове* (окрім випадку, коли при декларації застосований *typedef)*.

Ініціалізація

Розглянемо тепер ініціалізацію структур і структурних змінних. До останніх версій C++ не можна було ініціалізувати структуру безпосередньо під час декларації. Наприклад, наступна С програма на Сі-компіляторах повинна впасти.

```
struct Point
{
int x = 0;  // COMPILER ERROR:  cannot initialize members here
int y = 0;  // COMPILER ERROR:  cannot initialize members here
};
```

Причина подібної поведінки наступна — коли тип об'явлений, пам'яті під нього ще не виділено. Пам'ять виділяється лише коли створюються відповідні змінні.
В нашому прикладі визначення структури є зовнішнім, а змінна структурного типу описана всередині функції . Для ініціалізації структури використовується синтаксис за допомогою фігурних дужок, подібний тому, який використовується при ініціалізації масивів:

```
struct book odna={“Три мушкетери”, А.Дюма, 3};
```

або

```
struct Point {
   int x, y;
};

int main() {
        // Ініціалізація. Полю x присвоюється значення  0,  а  y  присвоюється 1.
        //  Порядок декларації визначає порядок ініціалізації.
        struct Point p1 = {0, 1};
}
```

Отже, використовується список ініціалізаторів розділених комами і взятий у фігурні дужки. Кожне конкретне значення (ініціалізатор) повинен відповідати типу елемента структури, якому присвоюється початкове значення. Тому можна одночасно присвоїти елементу nazva стрічкове значення, а елементу cina-числове. Для ясності кожному елементу відводиться власна стрічка ініціалізації, але компілятору достатньо того, щоб значення були розділені комами.
Ініціювання полів структури слід здійснювати або при її описі, або в тілі програми. При описі структури ініціювання полів виглядає, наприклад, для описаної вище структури **Student**

```
st1 = {"Молодець І. І.", 40, 10, 10, 3.4f};
st2 = {"Поганенко А. М.", 20, 1, 1, -3.5f};
```

Якщо ініціювання виконується в тілі програми, то для звернення до імені поля треба спочатку записати ім'я структурної змінної, а потім ім'я поля. Ці обидва записи відокремлюються крапкою і являють собою складене ім'я.

Отже, у випадку появи змінної **st1** у програмі для її ініціювання можна записати ініціалізацію за допомогою фігурних дужок, або ініціювання виконується за допомогою доступу до кожного поля.

```
int main() {
        struct Point p1;
        // Доступ до полів point p1
        p1.x = 20;
        p1.y = 10;
        printf ("x = %d, y = %d", p1.x, p1.y);
}
```

Направлена Ініціалізація (Designated Initialization) дозволяє ініціалізувати поля структури в будь-якому порядку. Ця властивість додана в C (не C++) зі стандарту C99 .

Приклад.

```
#include<stdio.h>
struct Point3D {
 int x, y, z;
};
int main()
{
         // приклад designated initialization
        struct Point3D p1 = {.y = 0, .z = 1, .x = 2};
        struct Point3D p2 = {.x = 20};

        printf ("x = %d, y = %d, z = %d\n", p1.x, p1.y, p1.z);
        printf ("x = %d", p2.x);
        return 0;
}
```

Результат роботи:

```
x = 2, y = 0, z = 1
x = 20
```

ця властивість відсутня в C++ тобто присутня лише в C.

Серед полів можна використовувати й вказівники.

Розглянемо ілюстраційну програму:

```
#include <string.h>
#include <stdio.h>
int main ( ){
        // визначили структуру
        struct credit { char* pib; int theory[2]; int tasks[2]; float avg;} st1, st2;
        // потрібно виділити пам'ять під перше поле
        st1.pib=(char*) malloc(30);
        strcpy (st1.pib, "Нездавайло Х.Х.");// та ініціалізувати його
        st1.theory = {2,3}; // ініціалізуємо інші поля
        st1.tasks[0] = 0;
        st1.tasks[1] =3;
```

```
// рахуємо середній бал
st1.avg = (float) (st1.theory[0] +st1.theory[1] + st1.tasks[0] + st1.tasks[1]);
        st2 = st1;  // присвоєння структур - визначено
        puts (st2.pib); // виводимо 1 поле 2-ої структури
free(st1.pib); /* звільнюємо ділянку під прізвище – це звільнить й друге прізвище!!*/

// 2 варіант – більш правильний
        st1.pib=(char*) malloc(30); // виділямо дві ділянки під призвища
        st2.pib=(char*) malloc(30);
        strcpy (st1.pib, "Здавайло У.У");
        strcpy(st2.pib,st1.pib, sizeof(st1.pib));

        free(st1.pib); // звільнюємо дві ділянки під призвища
        free(st2.pib);
}
```

У наведеній програмі організується присвоювання всім полям структури **st1** відповідних значень. Слід зауважити, що поле **st1.pib** одержує значення шляхом застосування функції **strcpy ()**. Структурна змінна **st2** того ж типу, що і **st1**, тому справедлива операція **st2 = st1;.**

Анонімна структура

Якщо функція використовує тільки один структурний тип, то цей тип можна оголосити без імені. Тоді раніше розглянуту структуру можна оголосити таким чином:

```
struct{ char fam [25];
         int mat, fiz, prg;
        float sb;
} stl, st2;
```

Коли при описі структур у деякій функції або в межах видимості змінних у різних функціях є багато (але не всі) однакових полів, то їх слід об’єднати в окрему структуру. Її можна застосовувати при описі інших структур, тобто поля структури можуть самі бути типу **struct**. Це називається **вкладеністю структур** — її можна використати, наприклад, якщо треба обробляти списки студентів та викладачів університету. Студентські списки містять дані: прізвище та ініціали, дата (день, місяць, рік) народження, група та середній бал успішності, а в списках викладачів присутні такі дані: прізвище, ініціали, дата народження, кафедра, посада. У процесі обробки списку студентів і списку викладачів можна оголосити відповідно такі структури:

```
struct stud{
        char fio [25];
        int den, god;
```

```
        char mes [10];
        char grup;
        float sb;
        }
struct prep{
        char fio [25];
        int den, god;
        char mes [10];
        char kaf, dolg;
        }
```

В оголошених типах однакові поля можна об'єднати в окрему структуру і застосовувати її при описі інших типів. Поетапно це виглядає так:

- загальна структура:

```
struct spd{
        char fio [25];
        int den, god;
        char mas[10]; }
```

- структура для опису інформації про студентів :

```
struct stud
{
        spd dr;
        char grup;
        float sb}
st1, st2;
```

- структура для опису інформації про викладачів:

```
struct prep
{       spd dr;
        char kaf [10];
        char dolg [15];
} pr1, pr2;
```

У структурах **stud i prep** для оголошення поля, що містить дані про прізвище і дату народження, використовується раніше описаний тип **spd**. Тепер до поля **fio, den, god, mes** можна звернутися, використовуючи запис **st1.dr.fio**, наприклад, при зверненні до функції введення:

gets (st1.dr.fio);

або **gets (pr1.dr.fio);.**

Визначення власного типу

В мові С є ключове слово **typedef**, яке можна використовувати для визначення власного типу даних. Наприклад таким чином можна зробити щось на зразок аліасингу Пітона, наприклад визначити типе **BYTE** для однобайтового цілого:

```
typedef unsigned char BYTE;
```

Після цього визначення BYTE може використовуватись для визначення типу unsigned char**, наприклад:**

```
BYTE  b1, b2;
```

За домовленістю (стилем коду), заголовочні літери (uppercase letters) використовуються в таких означеннях щоб вказувати користувачам програми що цей тип насправді абревіатура, але насправді можна використовувати й звичайні літери

```
typedef unsigned char byte;
```

Можна і в багатьох випадках потрібно використовувати **typedef** для того щоб декларувати новий тип даних, що ми власне створили в програмі, зокрема при визначенні структури. Наприклад:

```
#include <stdio.h>
#include <string.h>

typedef struct Books {
  char title[50];
  char author[50];
  char subject[100];
  int book_id;
} Book;

int main( ) {
  Book book;
  strcpy( book.title, "C Programming");
  strcpy( book.author, "Cool Teacher");
  strcpy( book.subject, "Programming");
  book.book_id = 111111;

  printf( "Book title : %s\n", book.title);
  printf( "Book author : %s\n", book.author);
  printf( "Book subject : %s\n", book.subject);
  printf( "Book book_id : %d\n", book.book_id);

  return 0;
}
```

Результат роботи:

```
Book  title : C Programming
```

```
Book  author : Cool Teacher
Book  subject : Programming
Book  book_id : 111111
```

Вказівники на структури

Існує, принаймні, три причини використання вказівників на структури:

- вказівниками легше керувати;
- структура не може передаватись функції в якості аргументу, але це можливо для вказівника;
- в деяких оголошеннях даних використовуються структури, які містять вказівники на інші структури.

Оголошення вказівника на структуру:

struct book *dn;

Спочатку йде службове слово struct ,потім назва структури book , зірочка і ім'я вказівника. Це оголошення не створює нову структуру, але дозволяє використовувати вказівник dn для позначення будь-якої існуючої структури типу book.

Вказівник ініціалізується таким чином, що вказує на структуру.

dn=&x1;

Розглянемо приклад:

```
#include <stdio.h>
#define N 20
typedef struct names{ /*перший шаблон*/
        char imya[N];
        char prizv[N];
} names;
typedef struct harakter { /*другий шаблон*/
        struct names druzi; /*вкладена структура*/
        char bludo[N];
        char robota[N];
        float zarob;
        } harakter;
int main() {
        struct harakter x1[2]{{ /*ініціалізація змінної*/
                {“Іван” ,“Петренко”}, “вареники”, “інженер”,30250.00},
                {“Петро”,”Іващенко”},”борщ”,”лікар”,40325.00}};
        struct harakter *x2;
        x2=&x1[0]; /*вказує на структуру*/
        printf(“заробіток: %lf \n”,x2->zarob);
```

```
        x2++; /*вказує на наступну структуру*/
        printf("заробіток: %lf \n", (*x2).zarob);
        printf("улюблена страва %s - %s  \n ",x2→druzi.prizv,x2->bludo);
}
```

Вказівник x2 вказує на елемент x1[0]. Можна використати цей вказівник для отримання значення елемента x1[0].

1. За допомогою нової операції ->. Ця операція складається з дефіса (-) і символу "більше"(>). Іншими словами, вказівник структури з операцією -> має такий самий зміст, які ім'я структури, після якого застосовується операція "крапка".
2. Якщо **x2= =&x1[0],**то і ***x2= =x1[0],** оскільки **&** і * є еквівалентними операціями. Тому можлива заміна

x1[0].zarob= =(*x2).zarob

Дужки необхідні, оскільки пріоритет операції "крапка" вище, ніж пріоритет операції "зірочка".

Так само, як і для базових типів, після оголошення структури як певного типу можна створювати масиви структур.

```
#include<stdio.h>
struct Point
{
        int x, y;
};
int main() {
    // Масив структур
     struct Point arr[10];
      // Доступ до членів масиву
        arr[0].x = 10;
        arr[0].y = 20;

        printf("%d %d", arr[0].x, arr[0].y);
        return 0;
}
```

Результат роботи:

10 20

Після оголошення структурного типу змінних для роботи з їхніми полями можна застосовувати і вказівники, тоді опис структури матиме вигляд:

```
struct stud
{
```

```
        char fam [25];
        int mat, fiz, prg;
        float sb;
} st1, *pst;
```

Доступ до полів може здійснюватися двома способами:

- з використанням операції розіменування «*», тобто
  gets ((*pst).fam); (*pst).fiz = 5;
- з використанням вказівника ->, наприклад,
  gets (pst -> fam); pst-> fiz = 5;
  тощо.

Крім того, до полів змінної **st1** можна звертатися, вказуючи поля через операцію «**.**», як це робилося раніше.

Дані типу структура можна об'єднати в масиви, тоді для розглянутого вище ілюстраційного прикладу з урахуванням кількості студентів, структуру можна записати так:

```
struct stud
{ char fam [20];
        int mat, fiz, prg;
        float sb;
} spis[15], *sp = &spis[0];.
```

У випадку, коли масив описується десь у тексті програми, тобто не саме після опису структури, його можна оголосити у вигляді: **stud spis [15];** — масив типу структура з ім'ям **stud**, що містить відповідну інформацію про групу із **15** студентів.

Доступ до елементів масиву типу структура здійснюється із застосуванням індексу або через покажчик-константу, яким є ім'я масиву, тобто одним з таких способів:

```
strcpy (spis[1].fam, " ");
spis[1].fiz = 5;
```

або

```
strcpy ((sp + 1) -> fam, " ");
```

Приведена в прикладі структура має три частини, які називаються елементами або полями. Для того, щоб працювати зі структура необхідно засвоїти три основні прийоми:

- встановлення формату структури;

- визначення змінної, яка відповідає даному формату;

- забезпечення доступу до окремих компонентів змінної-структури.

Поля структури можуть також бути масивами. Наприклад, у розглянутій структурі **stud** можна оцінки з різних предметів об'єднати в масив. Тоді структуру слід описати у вигляді:

```
struct stud1
{ char fam [25];
```

int pred [3];
float sb
} spis[15], *ps = &spis[0];.
Звернення до полів здійснюватиметься одним із способів:
((*ps).fam)
(ps->pred [0]),
наприклад,
gets ((*ps).fam);
printf("%d %d %d", **ps -> pred[0] , ps -> pred[1] , ps -> pred[2]);**
або
printf("%d %d %d",**ps -> *(pred + 0) >> ps -> *(pred + 1) >> ps -> *(pred + 2)).**
Доцільно також зазначити, що *бібліотека stdlib.h містить спеціальні функції для пошуку та сортування структурних змінних.*
Ще приклад:

```
#include<stdio.h>
typedef struct Point {
        int x, y;
}Point;
int main() {
        Point p1 = {1, 2};  // визначили змінну типу Point
        Point *p2 = &p1;  // p2  -  це вказівник на структуру p1
        // Доступ до полів структури використовуючи вказівник
        printf("%d %d", p2->x, p2->y);

        Point * p_array; //  визначили безрозмірний масив типу Point
        unsigned n;
        scanf(«%u»,&n); // ввели кількість елементів масиву p_array
        p_array = ( Point *) malloc(n*sizeof(Point)); // виділили память

        // ввели масив точок
        for(int i=0;i<n;i++){
                scanf(« %d %d », p_array[i].x, p_array[i].y);
        }
        int i=0;
        Point * ptr = p_array;
        while (i<n){ // вивели масив точок ітеруючись по вказівнику
                i++;
                printf(«%d %d», ptr→x, ptr->y);
                ptr++;
        }
        free(p_array); // не забули звільнити память
```

}

Робота з файлами

Основні концепції роботи з файлами

Файл *– це поіменована сукупність даних, яка зберігається на пристрої.* Будь-який файл являє собою послідовність байтів. В залежності від того, який сенс та призначення мають ці байти, дані поділяють на текстові та бінарні (двійкові). Таким чином, файли поділяються на текстові та бінарні.

Текстовий файл - текстовий потік даних, фактично рядок, що має ім'я та зберігається на пристрої. Текстові файли містять байти, що є кодами алфавітних, цифрових символів та знаків пунктуації (пробілів, табуляцій та символів переходу на новий рядок)

Бінарний файл – це сукупність будь якого типу даних, що так само знаходиться в пристрої. Бінарні файли можуть містити будь-які значення байтів, в тому числі і такі, яким не відповідає графічний знак, що може бути виведений на екран. При цьому для того, щоб зчитувати конкретний файл, потрібно знати дані якого типу та в якому порядку записані на пристрої.

Потоки вводу-виводу - це об'єкти типу FILE, до яких можна отримати доступ і маніпулювати ними лише за допомогою вказівників типу FILE *

Примітка: Хоча можна створити локальний об'єкт типу FILE, використання безпосередньо такої змінної для вводу-виводу в функціях має невизначену поведінку, тобто не визначено стандартом і може призвести до серйозних помилок.

Кожен потік вводу виводу пов'язаний з зовнішнім фізичним пристроєм (*файл, стандартний вхідний потік, принтер, послідовний порт* тощо).

Потоки вводу-виводу можуть використовуватися як для неформатованих, так і для форматованих входів і виходів. Вони чутливі до локалі і можуть виконувати широкі(багатобайтові) перетворення, якщо це необхідно. Всі потоки отримують доступ до одного і того ж локального об'єкта: останнього встановлено з setlocale. Окрім специфічної для системи інформації, необхідної для доступу до пристрою (наприклад, дескриптор файлу POSIX), кожен об'єкт потоку містить наступне:

1) **Ширину символів (починаючі зі стандарту** C95**)**: *невстановлену, вузьку* або *широку*(unset, narrow або wide)
2) **Стан буферизації**: *небуферизований, лінійний буфер, повністю буферизований*.(unbuffered, line-buffered або fully buffered)
3) **Буфер**, який може бути замінений зовнішнім, наданим користувачем буфером.
4) **Режим введення / виводу:** *введення, виведення* або *оновлення* (вхід і вихід).
5) **Індикатор бінарного / текстового режиму**.
6) **Індикатор стану кінця файлу**.
7) **Індикатор стану помилки.**

8) **Індикатор положення файлу** (об'єкт типу fpos_t), який для широких потоків символів включає стан розбору (об'єкт типу mbstate_t (C95)).
9) Блокування повторного вмикання, що використовується для запобігання гонкам даних, коли кілька потоків читають, записують, розміщують або запитують позицію потоку (починаючі зі стандарту C11).

Вузька і широка орієнтація
Відкритий потік не має орієнтації. Перший виклик fwide або до будь-якої функції вводу-виводу встановлює орієнтацію: широка функція введення-виведення робить потік з широким орієнтуванням, вузька функція вводу-виводу робить потік вузьким. Після встановлення ширини її можна змінити лише за допомогою freopen. Звичайні (вузькі) функції вводу-виводу не можна викликати на широкому потоці. Широкі функції вводу-виводу не можна викликати на вузько орієнтованому потоці. Широкі функції вводу-виводу перетворюють між широкими і багатобайтовими символами за допомогою виклику mbrtowc і wcrtomb. На відміну від багатобайтових символьних рядків, багатобайтові послідовності символів у файлі можуть містити вбудовані нулі і не повинні починатися або закінчуватися в початковому стані зсуву. POSIX вимагає, щоб параметр LC_CTYPE поточно встановленої локалі C зберігався в потоковому об'єкті в той момент, коли його орієнтація стала широкою, і використовується для всіх майбутніх входів / виходів цього потоку, поки не буде змінена ширина, незалежно від будь-яких наступних викликів до setlocale .

Текстовий потік є впорядкованою послідовністю символів, що складається з рядків (нуль або більше символів які закінчуються '\n'). Чи вимагає останній рядок закінчення '\n', визначено реалізацію. Символи, можливо, повинні бути додані, змінені або видалені на вході і виході, щоб відповідати умовам для представлення тексту в ОС (зокрема, потоки C на ОС Windows перетворюються на \n на виході, і конвертуються \r\n на вході). Дані, прочитані з текстового потоку, гарантовано порівнюються з даними, які раніше були виписані в цей потік, лише якщо виконується наступне:

- символ кінця рядку - \n
- дані складаються тільки з друкованих символів і контрольних символів і спецсимволів \n (зокрема, в ОС Windows, символам \t та \n відразу передує символ пробілу (пробіли, які виписуються безпосередньо перед останнім символом \0x1A;
- ніякого \n немає безпосередньо перед символом пробілу (пробіли, які виписуються безпосередньо перед \n знищуються

Бінарний (двійковий) потік є впорядкованою послідовністю символів, яка може прозоро записувати внутрішні дані. Дані, що зчитуються з двійкового потоку, завжди дорівнюють даним, які були раніше записані в цей потік. Реалізаціям дозволено лише додавати до кінця потоку ряд нульових символів. Широкий двійковий потік не повинен закінчуватися в початковому стані зсуву.
У реалізаціях POSIX не розрізняють текстові та двійкові потоки (спеціального відображення для будь-яких інших символів немає)

З точки зору програміста робота з файлами виглядає так:
1) Програма оперує не з маркером магнітного диску для зчитування інформації, а з допоміжною змінною (скажемо, f) типу вказівник на певну структуру, через які вже операційна система отримує доступ до конкретного місця на диску.
2) Перш ніж робити з файлом на диску будь-які дії (читати дані з файлу чи писати файл), програма повинна його відкрити. Файл, розташований на диску, зв'язується зі змінною f (тобто до змінної заносяться службові дані для доступу саме до даного файлу).
3) Після цього, коли треба записати чи прочитати дані з файлу, програміст викликає спеціальні функції читання та запису, передаючи їм через один аргумент змінну f канал доступу до файлу, а через решту аргументів – які саме дані читати чи писати.
4) На закінчення роботи програма повинна закрити файл і розірвати зв'язок між змінною f та файлом на диску, при цьому звільняються ресурси операційної системи, що використовуються для доступу до файлу.

Основні функції файлового введення-виведення оголошено в тому ж заголовочному файлі stdio.h, що й функції введення з клавіатури та виведення на екран. Вся подальша робота з файлом здійснюється через змінну f та вказівник на структуру даних типу FILE.

Відкриття файлу

Значення, яке повертає функція fopen – вказівник на службову структуру даних, через яку програма може звертатися до файлу на диску, присвоюється змінній f. Відтепер ця змінна є посередником між програмою та дисковим пристроєм.
Відкриття файла здійснюється за допомогою функції **fopen**, яка має наступний формат:

```
FILE* fopen(const char* filename, const char* mode);
```

тобто

FILE *fopen (“фізичне ім’я файлу”, “режим”);

Тут **FILE** - ім’я типу, описане в заготовочному файлі **stdio.h**;– вказівник на цю структуру.
Функція **fopen** повертає вказівник на файлову змінну, або **NULL**, якщо під час відкриття файла виникла помилка. Фізичне ім’я файлу – це його повне ім’я, включаючи шлях до файлу.
Можливі режими:
“**w”**- текстовий файл відкривається для запису; якщо файл із вказаним іменем існує, то його вміст знищується;
“**r”**- текстовий файл відкривається для читання;
“**a”**- текстовий файл відкривається для доповнення, інформація додається в кінець файлу.

Таблиця 5.1

Параметри доступу до файлу

Тип	Опис
r	Читання. Файл повинен існувати.
w	Запис нового файлу. Якщо файл с з таким іменем вже існує, то його зміст буде знищено. Інакше він створиться
a	Запис в кінець файлу. Операції позиціонування (fseek, fsetpos, frewind) ігноруються. Файл створюється, якщо не існував.
r+	Читання та оновлення. Можна як читати, так і записувати. Файл повинен існувати.
w+	Запис і оновлення. Створюється новий файл. Якщо файл с з таким іменем вже існує, то його зміст буде знищено. Можна як читати, так і записувати.
a+	Запис в кінець файлу й оновлення. Операції позиціонування працюють лише для читання, для запису ігноруються. Якщо файлу не існувало, то він створиться.

Якщо необхідно відкрити файл в бінарному режимі, то в кінець рядка режиму додається літера b, наприклад “rb”, “wb”, “ab”, або, для мішаного режиму “ab+”, “wb+”, “ab+”. Якщо замість b додати літеру t, то файл буде відкриватися в текстовому режимі.

Примітка. В новому стандарті С (С11, 2011) можна додати літеру x, яка означає, що функція fopen для запису повинна завершитися з помилкою, якщо файл вже існує.

Примітка. Компілятори MS можуть зараз вимагати використовувати функцію fopen_s замість fopen з міркувань безпеки (формат fp=fopen_s(FILE* fp,const char* name, const char* mode)). Але хоча в стандарті С11 додали цю функцію також, але «факультативно». Тому не всі компілятори (навіть С11 сумісні) до цієї пори підтримують її, отже потрібно або писати код роздільно для різних платформ, або «примушувати» MS компілятор використовувати fopen.

Як бачимо, режими відкривання файлу можуть бути доповнені специфікатором +, який означає режим **виправлення** – відкриття файлу одночасно і для запису, і для читання, наприклад:

```
FILE *lst;
lst=fopen(“D:\TC\file.txt”,”r+”);
```

Дані команди забезпечують відкриття вже існуючого текстового файлу **D:\TC\file.txt** як для читання, так і для запису.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
//З допомогою змінної file будемо мати доступ до файлу
```

```
FILE *file;
//Відкриваємо текстовий файл з правами на запис
file = fopen("C:/c/test.txt", "w+t");
//Пишемо в файл
fprintf(file, "Hello, World!");
//Закриваємо файл
fclose(file);
}
```

Відкриття файлу може відбутися з успіхом або з неуспіхом. У випадку успіху функція fopen створює в пам'яті службову структуру даних - потік, зв'язаний з потрібним файлом на диску, та повертає вказівникна цей потік. Цей вказівник потрібно зберегти в деякій змінній та потім використовувати в усіх операціях введення-виведення (див. приклад вище).

Неуспіх при спробі відкрити файл може виникнути з таких причин:

- для імені файлу, якого не існує на диску, задано режим r або r+;
- програма намагається відкрити для запису файл, для якого операційна система дає доступ лише для читання;
- програма відкрила надто багато файлів і вичерпала ліміт дозволених ресурсів;
- тощо.

В усіх таких випадках, коли замовлений файл в замовленому режимі відкрити неможливо, функція **fopen** повертає значення NULL.

Поведінка програми при неуспішному відкритті файлу та ще в багатьох схожих ситуаціях заслуговує на окремий розгляд. Гарно написана програма повинна бути стійкою до помилок, що можуть виникати під час виконання, через обставини, які програмісту та його програмі непідконтрольні. Скажімо, якщо програма запитує ім'я файлу у користувача, цілком може статися, що користувач помилково введе ім'я файлу, який насправді не існує. Або програма намагається записати дані у файл, а наявні у користувача права доступу забороняють запис.

Звичайно ж, програма повинна розпізнати таку нештатну ситуацію та коректно обробити її. Наведемо кілька типових рішень. В найпростішому випадку програма, побачивши, що файл відкрити неможливо, просто завершує свою роботу, друкуючи на екран повідомлення.

Зауважимо, що файл також бажано закрити по закінченню роботи за допомогою функції fclose().

Нагадаємо, що функція main повинна повертати ціле число, причому 0 символізує, що програма відпрацювала успішно, а будь-яке інше число означає, що програма завершилася через помилку.

```
#include <stdio.h>
#include <stdlib.h>
```

```
#define ERROR_FILE_OPEN -3

int main() {
        FILE *output = NULL;
        int number;
        output = fopen("D:/c/output.bin", "wb");
        if (output == NULL) {
            printf("Error opening file");
            getchar();
        exit(ERROR_FILE_OPEN);
        }

        scanf("%d", &number);
        fwrite(&number, sizeof(int), 1, output);

        fclose(output);
}
```

Наведений вище фрагмент працює за доволі примітивною логікою, завершуючи програму у разі невдачі. Бажано зробити програму більш гнучкою: у випадку невдачі вона пропонує користувачу ввести інше ім'я файлу або, якщо користувач бажає, завершити роботу.

```
# include <stdio .h>
int main () {
char fileName [ 80 ];
FILE *f;
do {
  printf (" Введіть ім 'я файлу або крапку : " );
  scanf ( "%s", fileName );
  if( strcmp ( fileName , "." ) == 0 )
     return 0;
  f = fopen ( fileName , "r" );
} while ( f == NULL );
/* далі нормальна обробка файлу */
. . .
fclose ( f );
}
```

Читання з файлу

Якщо текстовий файл відкрито в режимі, що допускає читання, то прочитати з нього довільні дані можна за допомогою функції **fscanf**. Це майже повний аналог функції **scanf**. Перший аргумент – вказівник на потік, другий аргумент - форматний рядок, що містить будь-яку кількість специфікаторів формату, решта аргументів (їх кількість відповідає кількості специфікаторів) - вказівники на змінні, в які треба розмістити результати розбору прочитаного тексту.
Формат команди:
int fscanf(const char * filename, const char * format,char* buffer);
тобто
int fscanf (вказівник_на_файл,рядок форматування, перелік змінних);
// fscanf ("Ім'я файлу", "формат як в scanf", змінні через амперсант);

Функція **fscanf** намагається читати символи з потоку та співставляти їх з форматним рядком так само, як функція **scanf** розбирає послідовність символів, введену з клавіатури.
Функція повертає число успішно співставлених специфікаторів.
Нехай, наприклад, дано файл з іменем data.txt з таким вмістом:

```
42  3.14  Magic lantern
```

Розглянемо програму (щоб не засмічувати розгляд, в тексті програми пропущено перевірку на помилку відкриття)

```
 # include <stdio .h>
 # define LEN 256
int main () {
        FILE *f;
        int m, n;
        double dt;
        char s[ LEN ];
        f = fopen (" data . txt ", "r");
        n = fscanf (f , "%d %lf %s", &m, &dt , s);
        printf (" Прочитано %d значень :\n", n);
        printf (" Ціле %d, дійсне %lf , рядок %s\n", m, dt , s);
         fclose ( f );
}
```

Якщо ця програма успішно відкриє файл для читання, то функція **fscanf** прочитає з файлу символи та співставивши їх зі специфікатором %d, перетворить їх на ціле число 42 та запише це число в змінну m. Аналогічним чином функція **fscanf** прочитає наступні знаки та занесе в змінну dt значення 373.14. Далі, обробляючи специфікатор %s, функція буде намагатися знайти у файлі послідовність символів до першого роздільника (пробілу, табуляції або переходу на новий рядок). Таким

чином, функція fscanf прочитає слово Magic, бо за ним йде пробіл, занесе його у масив s, приписавши до кінця нуль-символ кінця рядку. Решту тексту з файлу програма не прочитає взагалі (якби програма далі вводила ще дані цього файлу, вона прочитала б залишок, починаючи зі слова lantern).
Оскільки функція fscanf успішно співставила всі три специфікатори, вона поверне значення 3, яке і буде присвоєно змінній n. Тому програма повинна вивести на термінал наступний результат:

```
Прочитано 3 значень :
Ціле 42, дійсне -3.14 , рядок Magic
```

Для введення текстового рядка, що містить пробіли та табуляції, використовується спеціальна функція fgets. Вона має такий прототип:

char * fgets (char *s, int n, FILE *f);

З прототипу видно, що функція має три аргументи: вказівник на символьний масив, в якому буде розміщений прочитаний з файлу текстовий рядок, ціле число границю довжини рядка, та вказівник на потік у файл, з якого треба читати рядок. Функція **fgets** повертає покажчик на ту ж саму область пам'яті s, в яку розміщено прочитаний рядок. Ця функція читає текст з файлу, літера за літерою, поки не зустріне символ кінця рядка, кінець файлу, або поки не прочитає n-1 символ.
Замість функції fgets можна використовувати fscanf, але потрібно пам’ятати, що вона зчитує дані лише до першого пробілу.

fscanf(file, "%127s", buffer);

Також, замість повторного відкриття та закриття файлу можна скористатися функцією freopen, яка «перевідкриває» файл з новими правами доступу.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
        FILE *file;
        char buffer[128];
        file = fopen("C:/c/test.txt", "w");
        fprintf(file, "Hello, World!");
        freopen("C:/c/test.txt", "r", file);
        fgets(buffer, 127, file);
        printf("%s", buffer);
        fclose(file);
}
```

Функції роботи з файлами на диску

При переході від операцій запису до операцій читання часто необхідно застосовувати функцію **fflush,** яка забезпечує запис ще не збереженої інформації із буфера в файл. Вона має наступний формат:

int fflush (вказівник_на_файл);

Функція повертає EOF у разі помилки, та 0 в разі успішного виконання операції. Після закінчення роботи з файлом його необхідно закрити командою **fclose**, яка має наступний формат:

int fclose(вказівник_на_файл);

Для видалення файла на диску слугує функція **remove**. Вона має наступний формат:

int remove("ім'я_файлу");

Для зміни імені файлу на диску є функція **rename**, яка має наступний формат:

int rename("старе_ім'я_файлу", "нове_ім'я_файлу");

У разі успішності виконання функцій **remove, rename** вони повертають 0, та ненульове значення у іншому випадку.

Виведення/ Запис у файл

Форматоване виведення-введення даних у текстовий файл здійснюється за допомогою функцій **fprint** та **fscanf** віповідно, які мають наступний формат:

int fprintf (вказівник_на_файл,рядок форматування, перелік змінних);

Рядок форматування вміщує об'єкти трьох типів:

- -звичайні символи, які виводяться на екран;
- -специфікації перетворення, кожна з яких викликає виведення на екран значення чергового аргументу зі списку аргументів;
- -керуючі символи (для початку з нового рядка, табуляції, звукового сигналу та ін.).

Специфікація розпочинається символом **%** і закінчується символом, який задає перетворення. Між цими знаками може стояти:

- знак мінус "**-**", який вказує, що параметр при виведенні на екран повинен вирівнюватися по лівому краю. Інакше - по правому;

- рядок цифр, який задає розмір поля для виведення. Крапка, яка відділяє розмір поля від наступного рядку цифр, що визначає розмір поля для виведення розрядів після коми для типів **float** та **double**;

- символ довжини **l**, який вказує на тип **long**;

Далі вказується символ типу інформації виведення (перетворення):

d- десяткове число;

o- вісімкове число;

x- шістнадцяткове число;

c- символ (тип char);

s- рядок символів (string);

e- дійсне число в експоненціальній формі;

f- дійсне число (float);
g- використовується як e і f, але виключає виведення на екран незначущих нулів;
u- беззнаковий тип (unsigned);
p- вказівник (pointer).
Якщо після **%** записано не символ перетворення, то він виводиться на екран.
Керуючі символи:
\a- для короткочасної подачі звукового сигналу (alarm);
\b- для переведення курсору вліво на одну позицію (back);
\n- для переходу на новий рядок (new);
\r- для повернення каретки (курсор на початок поточного рядка) (return);
\t- для горизонтальної табуляції (tabulation);
\v- для вертикальної табуляції (vertical).
Приклад. Відкрити файл для запису. Записати у файл матрицю з 10 рядків і 10 стовпців.

```
#include <stdio.h>
int main(){
  int i,j;
  FILE *lds;
   lds=fopen("epa.txt","w");
/*Відкриття файлу на диску для запису. Якщо він не існує, то створюється автоматично*/
for(i=1;i<=10;i++){
                for(j=1;j<=10;j++){
                        fprintf(lds,"%d%c",i+j-1,((j==10)?'\n':' '));}
                }
/*Запис даних до файлу: Перший аргумент – вказівник на файл, другий і третій такі ж як і для команди printf */
fprintf(lds,"\n");
fclose(lds); /*Закриття файлу*/
}
```

Символьний запис у файл

Для деяких задач буває зручно вводити файл посимвольно.
Для цього слугує функція fgetc з таким прототипом:
int fgetc (FILE *f);
Для посимвольного введення та виведення даних у файл використовуються функції **fgetc** та **fputc** відповідно, які мають наступний формат:
int fgetc(вказівник_на_файл);
int fputc(вказівник_на_файл);
Функція **fgetc** повертає наступний символ із файла, на який вказує відповідний вказівник, або **EOF** в разі помилки. Функція **fputs** записує символ у файл, на який

вказує відповідний вказівник, та повертає записаний символ, або **EOF** у випадку помилки.
Для введення-виведення рядків до файлу слугують функції **fgets** та **fputs**, які мають синтаксис, аналогічний **fgetc, fputc.**

Введення, виведення та повідомлень про помилки, stdin, stdout, stderr.

Для того, щоб засвоїти матеріал цього розділу, треба добре зрозуміти особливу ідеологію, що лежить в основі роботи з пристроями в операційних системах (назаразок UNIX) та у мові С. Основний принцип полягає в тому, що кожен приєднаний до комп'ютера пристрій (текстовий дисплей, клавіатура, принтер, модем, звукова плата) розглядається як своєрідний уявний файл. Наприклад, введення символів з клавіатури це те ж саме, що читання даних з уявного файлу, пов'язаного з клавіатурою, виведення на екран виглядає як запис у пов'язаний з екраном файл. Ця ідея дозволяє в однаковий спосіб обробляти введення з дискового файлу та з пристрою, такого, як клавіатура чи модем. Величезна вигода полягає в тому, що програма може не замислюватися над тим, звідки насправді беруться дані, з диску чи з пристрою у один і той самий текст програми підходить для обох випадків.
В заголовочному файлі stdio.h оголошено спеціальні змінні, в яких містяться вказівники на стандартні потоки введення-виведення. Найголовніші з них:

- **stdin** стандартний потік введення, зв'язаний з клавіатурою;
- **stdout** стандартний потік для виведення звичайної інформації, пов'язаний з дисплеєм;
- **stderr** стандартний потік, також пов'язаний з дисплеєм, але призначений для виведення повідомлень про помилки.
- ***stdprn*** стандартний потік виведення, пов'язаний з принтером.

Функція **printf** насправді виводить символи в стандартний потік виведення stdout, а функція введення **scanf** бере символи зі стандартного потоку введення stdin. Таким чином, наступні два оператори роблять в точності одне й те саме
printf ("Матан, алгебра, програмування \n");
fprintf (stdout , "Матан, алгебра, програмування \n");
Наступні два оператори також:
scanf ("%d", &x);
fscanf (stdin , "%d", &x);
Наведемо деякі корисні поради, пов'язані з таким підходом до обробки файлів.
Для друку повідомлень про помилки під час роботи програми призначено потік stderr.
ним і треба користуватися в гарно написаній програмі замість того, щоб виводити такі домлення функцією printf (в такому випадку повідомлення пішли б в інший потік stdout).
Приклад:

```
 int n, *p;
 printf (" Кількість елементів ");
scanf ( "%d", &n );
p = (int *) malloc ( n * sizeof (int ) );
if( p == NULL ) {
fprintf ( stderr , "Не вистачає пам 'яті \n");
exit ( -1 );
}
 /* нормальна робота */
 . . .
```

Тут повідомлення для користувача, яке супроводжує нормальну роботу програми (на кількість елементів масиву), виводиться функцією printf та потрапляє у потік std повідомлення про помилку (нестача пам'яті) виводиться у спеціальний потік для помилок(хоча обидва ці потоки зрештою пов'язані з одним і тим самим дисплеєм).

Робота з бінарними файлами

Бінарні(двійкові) файли, на відміну від текстових, можуть містити такі коди, які не допускають відображення на екрані. Крім того, інформація у текстових файлах структурується розбиттям на рядки, пробілами між словами, табуляціями, структуру тексту видно людським оком.
Бінарний файл, навпаки, це просто послідовність байтів, не призначена для читання людиною, двійковий файл може обробляти лише програма, яка «знає», який сенс має той чи інший байт.
Для обробки двійкових файлів використовуються ті ж потоки (змінні типу вказівників на структуру **FILE**), що і для текстових файлів. Для відкриття бінарного файлу використовується та ж функція **fopen**, а для закриття функція **fclose**. *Відрізняються лише функції введення та виведення.*
Для того щоб зрозуміти роботу цих функцій, треба зрозуміти декілька фактів.
При роботі з файлами дуже часто доводиться мати справу з числами, сенсом яких є розмір файлу чи його частини. Звичайно ж, розмір вимірюється завжди цілим числом (причому невід'ємним), і можна було б скористатися звичайним типом int. Але в мові С вирішили виділити цей тип окремою назвою, щоб підкреслити його особливе призначення:
size_t – це просто інше ім'я, запроваджене через **typedef**, для довгого беззнакового цілого типу.
При обробці текстових файлів функції введення спочатку читають з файлу символи, а потім розглядають їх як, наприклад, десятковий запис цілого числа і розміщують відповідне значення в пам'яті. В двійковому файлі числа зберігаються вже не в десятковому записі, а в такому ж точно вигляді, в якому вони існують в оперативній пам'яті машини. Іншими словами, записати інформацію до двійкового файлу значить перенести на диск точну копію вмісту деякої області оперативної пам'яті, байт за байтом.
В основу обробки бінарних файлів покладено таку ідею: обмін даними між пам'яттю та файлом здійснюється блоками однакового розміру. За одну операцію виведення можна взяти з пам'яті та скопіювати на диск певну кількість блоків, за одну операцію введення можна, навпаки, кілька блоків прочитати з файлу та записати в пам'ять. Важливо, що введення/виведення здійснюється лише цілими блоками (неможливо, наприклад, ввести півтори блоки).
Для введення-виведення даних до бінарних файлів застосовуються функції **fwrite** та **fread** відповідно.
Важливо, що блоки, які записуються у файл, повинні стояти у пам'яті підряд. Також і при читанні даних – блоки, прочитані з файлу, розташуються у пам'яті

підряд. Для запису даних до двійкового файлу призначена функція **fwrite**, яка має чотири аргументи:

1. **void *p** – вказівник на те місце в оперативній пам'яті, де починається послідовність блоків даних, яку треба записати у файл;
2. **size_t b** – довжина в байтах одного блоку;
3. **size_t n** – число блоків;
4. **FILE *f** – вказівник на потік: до якого файлу записати дані.

Функція бере з пам'яті, починаючи з місця, на яке вказує вказівник, n блоків даних, кожен розміром b байт, та записує їх у файл, пов'язаний з потоком f.
Функція повертає ціле значення число блоків, які їй вдалося записати. При успішній роботі, звичайно ж, це число повинно дорівнювати n, менше значення повинно свідчити про помилку у процесі запису у файл (наприклад, перевовнення диску). В підсумку, функція має прототип:
size_t fwrite (void *p, size_t b, size_t n, FILE *f);
або
int fwrite(вказівник_на_масив, розмір_об'єкта, кількість_об'єктів, вказівник_на_файл);
Функція забезпечує запис об'єктів із масиву, на який вказує **вказівник_на_масив**, кількість об'єктів визначається як **кількість_об'єктів,** розміром **розмір_об'єкта** у файл, на який вказує **вказівник_на_файл**. Функція повертає кількість успішно записаних об'єктів.
Для введення даних з двійкового файлу призначена функція fread. Вона має ті ж аргументи, що й функція fwrite, але передає дані в протилежному напрямку: з потоку f читає n блоків даних, кожен розміром b байт, та розміщує їх в пам'яті, починаючи з місця, на яке вказує вказівник.
Вони мають наступний формат:
size_t fread (void *p, size_t b, size_t n, FILE *f);
або
int fread (вказівник_на_масив, розмір_об'єкта, кількість_об'єктів, вказівник_на_файл);
Функція забезпечує зчитування об'єктів у масив, на який вказує **вказівник_на_масив**, кількість об'єктів визначається як **кількість_об'єктів,** розміром **розмір_об'єкта** із файла, на який вказує **вказівник_на_файл**. Функція повертає кількість успішно записаних об'єктів.
Розглянемо інший приклад.
Програма також записує до файлу числа, але не весь масив за одну операцію, а по одному блоку:

```
 # include <stdio .h>
 # define N 5
int main () {
double w[ N ] = { 2.0 , 1.4142 , 1.1892 , 1.0905 , 1.0443 };
 char fileName [] = " data . dat ";
```

```
FILE * out ;
int i;
out = fopen ( fileName , "w" );
 for ( i = 0; i < N; ++i )
fwrite ( &(w[i]), sizeof(double), 1, out );
fclose ( out );
}
```

Тут функція запису викликається в циклі 5 разів, при кожному виклику вона записує у файл вміст чергового елементу масиву w: аргументи означають, що записувати треба дані, починаючи з тієї адреси, де лежить ця змінна, записати треба один блок такого розміру, який займає в пам'яті дійсне число.

Тепер розглянемо програму, яка читає та роздруковує масив дійсних чисел з файлу data.dat, створеного попередньою програмою.

```
# include <stdio .h>
# define N 5
int main () {
        double w[ N ];
        char fileName [] = "data.dat ";
        FILE *inFile;
        int k, i;
        inFile = fopen ( fileName , "w" );
        k = fread ( w, sizeof ( double ), N, inFile );
        printf ( "З файлу прочитано %d чисел ", k );
        for ( i = 0; i < k; ++i ){
                printf ( "%lf\n", w[i] );
                }
        fclose (inFile );
}
```

Головним місцем програми є виклик функції **fread**, яка з inFile читає N (тобто 5) блоків даних, кожен такого ж розміру, як дійсне число, а ці дані розміщує в пам'яті там, де починається масив w. Отже, програма читає з inFile 5 чисел та присвоює їх елементам масиву. Значенням змінної k при цьому стає кількість вдало прочитаних блоків даних (тобто чисел).

Розглянемо також програму, яка читає та роздруковує числа з файлу один за одним:

```
# include <stdio .h>
int main () {
double x;
char fileName [] = "data.dat";
FILE *outFile;
int k =0;
```

```
in = fopen ( fileName , "r" );
while ( ! feof (outFile) ) {
        fread ( &x, sizeof ( double ), 1, in );
        printf ("%lf\n", x);
         ++k;
 }
printf (" всього %d чисел \n", k);
fclose (outFile );
}
```

Ця програма читає числа по одному, одне число за одну операцію. Аргументи функції fread означають: взяти з файлу (потоку) outFile один блок даних такого ж розміру, як дійсне число, та покласти в оперативну пам'ять, за тією адресою, де розташована змінна x. Цикл означає, що програма буде повторювати вказану операцію доти, поки не закінчиться файл, а кожне значення, прочитане з файлу та занесене до змінної , буде друкуватися на екрані.

Розглянемо роботу з файлами при використанні структур у якості параметрів, що треба прочитати або записати:

```
#define  SURNAME_LEN 30
typedef struct {
        char name[SURNAME_LEN]; int kurs;
        int prog_lab, prog_theory;
 } Student;

int main()
        char fileName [SURNAME_LEN];
        FILE * out ;
        Student s;
        int ans , n =0;
        printf ("Ім 'я файлу ? ");
        fgets ( fileName , SURNAME_LEN, stdin );
        out = fopen ( fileName , "w" );
        do {
                printf ("Ім 'я? "); printf (" Прізвище ? ");
                fscanf (s. surname , SURNAME_LEN , stdin );
                printf (" Оцінки з теорії та лаб . робіт ? ");
                scanf ("%d%d", &(s. prog_theor ), &(s. prog_lab ));
                fwrite ( &s, sizeof (Student), 1, out );
                 ++n;
                printf (" Продовжити (1 так , 0 ні )? ");
                 scanf ("%d", & ans );
        } while ( ans );
```

```
        printf ("У файлі %d записів \n", n);
        fclose ( out );
}
```

як видно, блоком даних є структура типу Student, за кожну операцію до файлу виводиться рівно один блок вміст змінної s. Після завершення програми дані у файлі будуть сформовані так:

Щоб проілюструвати читання структур з файлу в бінарному варіанті, наведемо програму, яка читає базу даних по студентах, сформовану попередньою програмою, та роздруковує її її вміст на екрані у вигляді таблиці з трьох колонок.

```
# include <stdio .h>

# define FNAME_LEN 80
# define SURNAME_LEN 20

 typedef struct tag_student {
char surname [ SURNAME_LEN ];
int prog_theor ;
 int prog_lab ;
} Student;

int main () {
char fileName [ FNAME_LEN ];
FILE *in;
Student s;
int n =1;
printf ("Ім 'я файлу ? ");
fgets ( fileName , FNAME_LEN , stdin );
in = fopen ( fileName , "r" );
while ( ! feof (in) ) {
  fread ( &s, sizeof (Student), 1 );
  printf ("%3d %30 s %10 d %10 d\n", n, s. surname , s. prog_theor , s. prog_lab );
 ++n;
}
fclose ( in );
}
```

Пошук у бінарному файлі: безпосередній доступ

Обробка текстових файлів частіше за все полягає у послідовному, літера за літерою, читанні одного файлу та такому ж послідовному записі результатів в інший файл. При обробці двійкових файлів у багатьох практичних задачах виникає потреба читати чи записувати файл не підряд, а в довільному порядку, час від часу переміщуючись по ньому вперед і назад.
Уявімо, що файл це довга стрічка, поділена на комірки, в кожній з яких може зберігатися один байт. По стрічці переміщується маркер читання-запису. При звичайних (послідовних) операціях читання та запису маркер, прочитавши чи записавши комірку навпроти якої стоїть, переміщується на одну комірку вперед. Разом з тим, в мові С існують функції, які дозволяють рухати маркер на довільну відстань вперед та назад по файлу. якщо встановити голівку на певну комірку, наступна операція читання чи запису буде відноситися саме до цієї комірки.
Функція fseek доволі універсальна: вона дозволяє встановити маркер на n-й байт, рахуючи від початку файла, на n-й байт від кінця файлу, або перемістити маркер на n байтів (вперед чи назад) відносно її поточного місця. Функція має три аргументи:

- Вказівник на потік в якому файлі встановлювати маркер;
- ціле число – на яку відстань переміщувати маркер;
- ціле число, що означає режим переміщення маркеру (рахувати відстань від початку, від кінця файлу, або від поточної позиції маркеру).

Часто при роботі з файлами корисно використовувати поелементний доступ до файлу. Для цього застосовується функція **fseek**, що має формат:
Int fseek(вказівник_файла, зміщення, початкове_значення);
Функція встановлює позицію із **зміщенням** відносно початкового_значення у файлі, який визначається **вказівником файла. Початкове значення** може приймати наступні значення:
SEEK_SET – зміщення відносно початку файла;
SEEK_CUR – зміщення відносно поточного положення;
SEEK_END – зміщення відносно кінця файла;
Функція **ftell** повертає поточну позицію у файлі, та має наступний формат: За допомогою функцій **fseek** та **ftell** легко дізнатися розмір файлу. Для цього достатньо спочатку поставити голівку на кінець файлу, а потім дізнатися її поточну позицію:

```
FILE *f;
long l;
char fileName = "data.dat ";
f = fopen ( fileName , "r" );
fseek ( f, 0L, SEEK_END );
l = ftell ( f );
printf ( " Довжина файлу %ld байтів \n", l );
```

long ftell(вказівник_на_файл);

Приклад 2. Продемонструвати роботу із бінарними файлами.

```
#include <stdio.h>

int main(void){
        const unsigned n=10;
        int a[10];
        FILE *pfile;
        int i,j,pos,start;
        int * pstart;
        pstart=&start; /* вказівник вказує на змінну start */
        clrscr();
        printf("\nFilling vector with numbers...\n\n a=[ ");
        for(i=0;i<10;i++){
        a[i]=i+1;
        printf("%d ",a[i]);

        printf("]\n\nCreating binary file epa.dat for editing...");
        pfile=fopen("epa.dat","w+b");  /*створення нового бінарного файлу для редагування */
        j=fwrite(a,sizeof(int),n,pfile); /* запис елементів вектора а у бінарний файл */
        if(j<n){
                printf("\n\nAn error occured. Only %d of %d elements was written",j,n);
        }
        else printf("\n\nFile was filled with %d elements successfully",j);
        fflush(pfile);
        /* перед зчитуванням даних в режимі редагування обов'язково дописуємо вміст буфера у файл */
        printf("\n\nEnter the number of element you want to read from bin file ");
        scanf("%d",&j);
        fseek(pfile,(j-1)*sizeof(int),SEEK_SET);
        /*перехід до позиції необхідного елемента*/
        pos=ftell(pfile);
        /*Визначення позиції у файлі*/
        fread(pstart,sizeof(int),1,pfile);
        printf("\n\n SEEK_SET: %dth element position is %d, element is %d",j,pos,*pstart);
        printf("\n\nPress any key to exit");
        fclose(pfile); /* закриття файла */
        }
```

Підсумок

Заголовок <stdio.h> надає загальну підтримку операцій з файлами та надає функції введення / виводу звичайних символів.
Заголовок <wchar.h> надає функції для введення / виводу широких символів.

Таблиця 5.2

Функції роботи з файлами

В <stdio.h> **- загальна робота з файлами**	
fopen, fopen_s(C11)	Відкриває файл (функція)
freopen, freopen_s(C11)	Перевідкриває файл (функція)
fclose	Закриває файл
fflush	Сінхронізує вивід з поточним, очищує буфер(функція)
setbuf	Встановлює буфер для потоку
setvbuf	Встановлює буфер та його розмір для потоку
В <wchar.h>	
fwide (C95)	Переключає буфер широких символів I/O та вузьких символів I/O (функція)
Введення, виведення	
В <stdio.h>	
fread	Читає з файлу
fwrite	Записує в файл
fgetc, getc	Читає символ
fgets	Читає рядок
fputc, putc	Записує символ
fputs	Записує рядок
getchar	Читає символ з **stdin** (функція)
gets(до C11), gets_s(з C11)	Читає рядок з stdin (функція)
putchar	Записує символ у stdout (функція)
puts	Записує рядок у **stdout**(функція)
ungetc	Повертає символ обратно в буфер
Широкі символи	
В <wchar.h>	
fgetwc, getwc(C95)	Читає широкий символ з потоку(функція)
fgetws(C95)	Читає широкий рядок з потоку (функція)
fputwc, putwc(C95)	Записує широкий символ в потік (функція)
fputws(C95)	Записує широкий рядок в потік

getwchar(C95)	Читає широкий символ з stdin (функція)
putwchar (C95)	Записує широкий символі до **stdout** (функція)
ungetwc(C95)	Повертає широкий рядок в потік
Форматоване введення/виведення	
В <stdio.h>	
scanf, fscanf, sscanf, scanf_s(C11), fscanf_s(C11), sscanf_s (C11)	Читає форматований рядок з **stdin**, потоку або буферу(функція)
vscanf(C99), vfscanf(C99), vsscanf(C99), vscanf_s(C11), vfscanf_s(C11), vsscanf_s (C11)	Читає форматований рядок з **stdin**, потоку або буферу використовуючи форматований список змінних (функція)
printf, fprintf, sprintf, snprintf(C99), printf_s(C11), fprintf_s(C11), sprintf_s(C11), snprintf_s (C11)	Друкує форматований ввод у **stdout**, потік або буфер(функція)
vprintf, vfprintf, vsprintf, vsnprintf(C99), vprintf_s(C11), vfprintf_s(C11), vsprintf_s(C11), vsnprintf_s (C11)	Друкує форматований ввод у **stdout**, потік або буфер(функція використовуючи форматований список змінних (функція)
Wide character	
Defined in header <wchar.h>	
wscanff(C95),wscanfs(C95),wscanf(C95),wscanf_s(C11),fwscanf_s(C11),swscanf_s (C11)	Читає широкий форматований рядок з **stdin**, файлу або буферу(функція)
vwscanf(C99),vfwscanf(C99),vswscanf(C99), vwscanf_s(C11),vfwscanf_s(C11),vswscanf_s (C11)	Читає широкий форматований рядок з **stdin**, потоку або буфер через список змінних(функція)
wprintf(C95),fwprintfs(C95), wprintf(C95),wprintf_s(C11), fwprintf_ss(C11),wprintf_s(C11),snwprintf_s (C11)	Друкує форматований рядок в **stdout**, потік або буфер(функція)
vwprintf(C95),vfwprintf(C95), vswprintf(C95),vwprintf_s(C11),vfwprintf_s(C11),vswprintf_sм, vsnwprintf_s (C11)	Друкує форматований рядок **stdout**, потік або буфер використовуючи список змінних(функція)
Позиції в файлі	
В <stdio.h>	
ftell	Повертає поточну позицію маркеру (функція)
fgetpos	Повертає позицію у файлах(функція)
fseek	Ставить маркер на потрібну позицію(функція)
fsetpos	Ставить маркер на фіксовану позицію у файлі (функція)

rewind	Повертає маркер на початок файлу(функція)
Обробка помилок	
В <stdio.h>	
clearerr	Помилка очистки буфера(функція)
feof	Перевіряє чи ми на кінці файлу(функція)
ferror	Помилка у файлі(функція)
perror	Рядок з поточної помилки у **stderr** (функція)
Дії над файлами	
В <stdio.h>	
remove	Знищує файл(функція)
rename	Переназиває файл(функція)
tmpfile, tmpfile_s(C11)	Вказівник на поточний файл(функція)
tmpnam, tmpnam_s(C11)	Повертає унікальне ім’я файлу(функція)

Типи

В <stdio.h>

Тип	Означення
FILE	Тип для контролю I/O потоку в С
fpos_t	Тип, що визначає позицію та стан маркера у файлі

Макроси

В <stdio.h>

stdin,stdout,stderr	Вираз типу FILE* асоційований вхідним потоком консолі, Вираз типу FILE* асоційований з вихідним потоком консолі, Вираз типу FILE* асоційований з вихідним потоком консолі для помилок (макрос)
EOF	Цілий константний вираз (constant expression of type int) для кінця файлу(макрос)
FOPEN_MAX	Максимальна кількість файлів, що може бути відкрита одночасно (макрос)
FILENAME_MAX	Розмір потрібний рядку байтів для найбільшого за розміром імені файлу(макрос)
BUFSIZ	Розмір буферу визначений setbuf() (макрос)
_IOFBF,_IOLBF,_IONBF	аргумент setvbuf() позначающій fully buffered I/O аргумент setvbuf()позначающій line buffered I/O аргумент setvbuf()позначающій unbuffered I/O (макрос)

SEEK_SET, SEEK_CUR, SEEK_END	аргумент fseek() – начало файлу; аргумент fseek() поточна позиція; аргумент fseek() – кінець файлу (макрос)
TMP_MAX, TMP_MAX_S (C11)	Максимальна кількість імен в tmpnam Максимальна кількість імен в tmpnam_s (макрос)
L_tmpnam, L_tmpnam_s(C11)	Розмір масиву char для результату tmpnam; Розмір масиву char для результату tmpnam_s (макрос)

6) Модульність. Компіляція та робота програмних застосувань, що складаються з декількох файлів

Модульність. Розділення програм на два чи більше файли. Директива include. Бібліотеки. Статичні та динамічні бібліотеки.
Компіляція у бібліотеки та компіляція разом з бібліотеками в різних середовищах. Makefile-ли та робота з CMake.

Модульність

Зазвичай, в учбових курсах розглядаються лише прості та невеликі за обсягом програми, які містять декілька функцій та розміщуються в одному файлі. Однак в реальній програмістський роботі та в програмних доданках, що розв'язують реальні практичні задачі, часто використовується великий обсяг тексту та сам програмний доданок складається з сотен а то й тисяч функцій. Досвід показує, що записувати всі функції програми в один файл незручно, оскільки зі збільшенням програми в такому файлі стає важко орієнтуватися, а також через те, що при цьому унеможливлюється спільна робота кількох програмістів. Тому грамотна технологія програмування полягає в тому, щоб розбивати велику програму на частини (так звані модулі), кожен з яких складається з порівняно невеликої кількості функцій.

В першому наближенні модуль являє собою окремий файл, в якому містяться визначення, описи структур та класів, тіла декількох функцій.

Переваги модульності, однак, зовсім не вичерпуються згаданою зручністю від розбиття великої кількості функцій на невеликі групи. Програміст може написати такий набір функцій, який може стати корисним для інших програмістів та коли ці функції зроблено достатньо гнучкими та універсальними, щоб їх можна було вставляти в чужі програми. як наслідок, модулі можуть бути самостійним різновидом товару на ринку програмного забезпечення.

Іншими словами, робота програміста може полягати не лише в тому, щоб виготовляти програми для кінцевого споживача, але й в тому, щоб писати модулі для інших програмістів, які будуть вставляти їх у свої програми.

Крім того, до складу одного програмного продукту цілком можуть входити модулі, написані різними мовами програмування. Мови програмування бувають спеціалізованими: кожною мовою найзручніше вирішуються задачі своєї певної області. Якщо ж треба вирішити комплексну задачу, яка розпадається на частини, то найкраще кожну підзадачу реалізувати окремим модулем, написаним найзручнішою для цієї підзадачі мовою.

Отже, модульність стала одним з найвизначніших винаходів в історії програмування і досі залишається наріжним каменем сучасних технологій програмування. Модульність – це не лише суто технічний прийом, а ще й ключовий елемент культури програмування, вона займає центральне місце в парадигмі і, без перебільшення, має світоглядне значення.

Розбиття програми на різні файли

Коли потрібно працювати з достатньо великою програмою часто потрібно розбити її на якийсь певні логічні, функціональні частини які можна, а часто й потрібно винести в різні файли. Наприклад, це робиться для того щоб декілька програмістів працювало над одним проектом, відокремити частини функціоналу та інтерфейсів для різних підпроектів і таке інше.

Розглянемо просту програму:

```
#include <stdio.h>
int add(int x, int y){
     return x+y;
}
int main(){
   printf("The sum of 3 and 4 is: %d \n ", add(3, 4));
}
```

Тут доцільно розбити програму на 2 частини - з функцією add (файл add.c) та головною функцією (файл main.c):

```
// файл 1  add.c
int add(int x, int y){
     return x+y;
}
// файл 2: main.c
#include <stdio.h>
int main(){
   printf("The sum of 3 and 4 is: %d \n ", add(3, 4));
}
```

Однак, якщо ми просто скомпілюємо тепер файл - то програма може не запрацювати:

warning: implicit declaration of function ‘add’ [-Wimplicit-function-declaration]

Повідомлення каже нам, що немає визначення тіла функції add.

Для того, щоб компіляція відбулася без попереджень, можна додати включення першого файлу у другий та скомпілювати його.

```
#include <stdio.h>
#include "add.c"
int main(){
   printf("The sum of 3 and 4 is: %d \n ", add(3, 4));
}
```

В цьому випадку директива #include "add.c" просто додає код файлу add.c до другого файлу main.c, а отже команда компіляції (наприклад gcc main.c) виконає початкову версію файлу.

Включення файлів

Для використання функціоналу інших програмних файлів в мовах C та C++ використовується директива (макрокоманда) #include .

Синтаксис даної директиви наступний: після #include йде ім'я файлу (з розширенням для C та без розширення для Ci++), взяте або в кутові дужки (<>), або в подвійні лапки (“”), тобто має одну з двох форм:

1. **# include < filename.h>**
2. **# include "filename.h"**

Перша форма вказує транслятору, що файл, ім’я якого вказано в кутових дужках, треба шукати в спеціальній директорії (директоріях) – директоріях, що вказуються як системні шляхи, які містять стандартні або ті, що ми хочемо щоб вони трактувались як стандартні, заголовочні файли. Саме тому в кутових дужках вказують імена таких заголовочних файлів, як stdio.h , math.h і т.п., які використовуються компілятором. Друга форма макрокоманди означає, що транслятор повинен шукати файл в директорії з програмою, або в певній користувацький директорії (можливо, зокрема, у тій самій директорії, що й програма модуля).

Тобто директива include дає транслятору команду знайти файл з відповідним іменем та підставити весь вміст цього файлу на місце макрокоманди.

Покажемо це на прикладі. Нехай є файл mydecl.h

```
// mydecl.h
typedef unsigned long int UL;
# define MAX_LEN 32
UL functionSample( int * k , UL x );
```

та ще один файл з іменем mycode.c :

```
// mycode.c
```

```
# include " mydecl . h " /* Включили файл mydecl.h !!! */

int main () {
int m [ MAX_LEN ];
UL s , r =0;
s = functionSample(m , r );
. . . /* ще якісь оператори */
}
```

В файлі mycode.c компілятор прочитавши другий рядок знайде mydecl.h та вставить його на місце директиви. Отже, після такої підстановки компілятор «побачить», ніби файл mycode.c виглядає так:

```
typedef unsigned long int UL;
# define MAX_LEN 32
UL functionSample( int * k , UL x );
int main () {
int m [ MAX_LEN ];
UL s , r =0;
s = functionSample( m , r );
. . . /* ще якісь оператори */
}
```

Більш точно процес роботи директиви виглядає так:

- транслятор читає файл рядок за рядком;
- коли транслятор бачить в тексті директиву #include, він:
- тимчасово припиняє обробляти поточний файл;
- шукає той файл, ім'я якого вказано в лапках після слова include;
- та починає так само, рядок за рядком, опрацьовувати його;
- дійшовши до кінця заголовочного файлу, транслятор знов повертається до недочитаного файлу та продовжує обробляти його з того самого місця, де припинив.

Однак, це не дуже гарне рішення з точки зору створення саме бібліотеки для компіляції. Бажано створити окремий модуль для роботи з створеними функціями.

Для цього можна створити файл з використанням прототипів (попередньої декларації):

```
//main.c (з  попередньою декларацією - forward declaration):
#include <stdio.h>
extern int add(int x, int y); // функція вказує що є функція add()
// без специфікатору extern в принципі можна обійтися
int main(){
   printf("The sum of 3 and 4 is: %d \n ", add(3, 4));
```

```
    return 0;
}
// файл add.c :
int add(int x, int y){
    return x+y;
}
```

Тепер, коли компілятор компілює main.c, він буде знати, який ідентифікатор функції додати і бути скомпільованим. Конпоновщик (лінкер) з'єднає виклик функції, щоб додати в main.cpp визначення функції add в add.c. Але для цього, щоб це відбулося, потрібно скомпілювати обидва файли разом (наприклад, gcc main.c add.c).

Зауважимо, що ще краще додати специфікатор extern до функції int add(int x, int y) в файлі main.c.

Крім того, не завжди зовсім зручно додавати прототипи функцій на початку файлу. Дот того ж хотілося б компілювати один файл замість того, щоб вказувати багато файлів.

Тоді є наступний варіант - додати #include для реалізації функції:

```
#include <stdio.h>
#include "add.c"
int main(){
    printf("The sum of 3 and 4 is: %d \n ", add(3, 4));
}
```

Але це фактично той самий попередній варіант, тут поки що немає бібліотеки з інтерфейсом. На мові С для створення таких багатофайлових застосувань створюють заголовочні файли. Для даної програми це буде тоді виглядати так:

```
//Файл add.c :
int add(int x, int y){
    return x + y;
}
// Файл add.h :
extern int add(int x, int y);
// Файл main.c:
#include <stdio.h>
#include "add.h"
int main(){
    printf("The sum of 3 and 4 is: %d \n ", add(3, 4));
    return 0;
}
```

Компіляція знов іде для обох файлів разом (наприклад, gcc main.c add.c).

Використовуючи цей метод, ми можемо надавати файлам доступ до функцій, які створені в іншому файлі.

Спробуйте самі скомпілювати add.c і main.c . Якщо ви отримали помилку лінкеру, переконайтеся, що ви додали add.c у вашому проекті чи рядку компіляції належним чином.

Проблеми, які виникають при компіляції багатьох файлів

Є багато речей, які можуть піти не так, коли ви вперше намагаєтеся працювати з декількома файлами. Якщо ви спробували вказаний вище приклад і виникли помилки, перевірте наступне:

1. Якщо отримується помилка компілятора про те, що add не визначена в main, то, напевно, був забутий прототип для функції add в main.cpp.

2. Якщо отримується помилка лінкера про не визначену функцію, наприклад, невирішений зовнішній символ "int __cdecl add (int, int)" (? add @@ YAHHH @ Z), на який посилається функція _main, то

2a. - найімовірнішою причиною є те, що add.cpp не додано до вашого проекту правильно. Під час компіляції повинен бути список компільованих файлів як main.cpp, так і add.cpp. Якщо ви бачите тільки main.cpp, то add.cpp точно не збирається. Якщо ви використовуєте Visual Studio або Code :: Blocks, ви повинні побачити add.cpp у списку Solution Explorer / Project з лівого боку IDE. Якщо ви не хочете, клацніть правою кнопкою миші на своєму проекті та додайте файл, а потім спробуйте його знову. Якщо ви компілюєте в командному рядку, не забудьте включити як main.cpp, так і add.cpp у команду компіляції.

2b. - можливо, ви додали add.cpp до неправильного проекту.

2c. - можливо, файл встановлено так, щоб він не компілювався або посилався. Перевірте властивості файлу та переконайтеся, що файл налаштований на компіляцію/лінковку. У Code::Blocks компіляція та посилання є окремими прапорцями, які слід перевірити. У Visual Studio є опція "виключити з побудови", яка повинна бути встановлена на "ні" або залишена порожньою.

3. Не робить #include “add.c” в файлі main.c. Наслідком буде те, що компілятор вставить вміст add.c безпосередньо в main.c, а не розглядати їх як окремі файли. Хоча він може компілюватися і виконуватися для цього простого прикладу, ви будете стикатися з проблемами, якщо використовуєте цей метод.

Висновки

Коли компілятор компілює багатофайлову програму, вона може компілювати файли в будь-якому порядку. Крім того, він компілює кожен файл окремо, не знаючи, що знаходиться в інших файлах.

Реальні проекти завжди працюють з декількома файлами (а часто їх бувають десятки а то і сотні, до того ж з використанням сторонніх бібліотек), тому важливо переконатися, що ви розумієте, як додавати і компілювати декілька файлів.

Нагадування: кожен раз, коли ви створюєте новий файл (.c, .cc або .cpp), вам потрібно буде додати його до свого проекту, щоб його було зібрано.

Модульна організація програм

Щоб добре зрозуміти принцип побудови багатомодульної програми, варто спочатку уявити деяку достатньо велику програму, яка вся міститься в одному файлі. Вона повинна містити декілька функцій, кожна з яких може викликати будь-яку іншу функцію. Оскільки будь-яка функція повинна бути оголошена перед першим викликом, а кожна функція програми може викликати будь-яку іншу функцію, на самому початку програми повинні стояти оголошення усіх функцій (крім main).

Тепер уявімо, що цю ж програму розбито на кілька модулів, тобто тіла декількох функцій перенесено в один файл, тіла кількох інших функцій – у другий, і т.д. Функція, розміщена в одному модулі, може викликати функцію, що міститься у іншому модулі.

А це означає, що на початку кожного модуля повинні міститися оголошення (прототипи) всіх функцій з інших модулів, які викликаються з цього модуля.

Бібліотеки

Бібліотека - це пакет коду, який призначений для повторного використання багатьма програмами. Як правило, бібліотека C ++ складається з двох частин:

1) Файл заголовка, який визначає функціональні можливості, які бібліотека виставляє (пропонує) програмам, що використовують її.

2) Скомпільований бінарний файл, який містить реалізацію цієї функціональності, попередньо скомпільовану в машинну мову.

Деякі бібліотеки можуть бути розділені на кілька файлів та/або мати декілька файлів заголовків.

Бібліотека: Бібліотека - це місце, де реалізована фактична функціональність, тобто вони містять тіло функції. Бібліотеки мають переважно дві категорії:

- **статичні бібліотеки,**
- **динамічні бібліотеки.**

Бібліотеки попередньо компілюються з кількох причин. По-перше, оскільки бібліотеки нечасто змінюються, їх не потрібно часто перекомпілювати. Було б марно витрачати час на перекомпіляцію бібліотеки кожного разу, коли ви писали програму, яка їх використовувала. По-друге, оскільки попередньо скомпільовані об'єкти знаходяться на машинній мові, це запобігає доступу людей або зміні вихідного коду.

Різниця між файлом заголовка та бібліотекою

Файли заголовків - це файли, які повідомляють компілятору, як викликати деякі функціональні можливості (не знаючи, як насправді працює функціональність), називаються заголовочними (хедер) файлами. Вони містять прототипи функцій. Вони також містять типи даних і константи, що використовуються в бібліотеках.

На мові С використовується #include для використання цих файлів заголовків у програмах. Ці файли закінчуються розширенням .h.

Приклад: Math.h - це заголовочний файл, що включає в себе прототип викликів функцій, таких як sqrt (), pow () і т.д. Простими словами, заголовочний файл подібний до візитної картки, а бібліотеки - як справжня людина, тому ми використовуємо візитну картку (файл заголовка), щоб дістатися до фактичної особи (бібліотека).

Представимо цю різницю в таблиці:

Таблиця 6.1

Порівняння заголовочних та бібліотечних файлів

Файли заголовків	Файли бібліотеки
Вони мають розширення .h	Вони мають розширення .lib,.dll, .so, .a
Вони містять декларацію функцій. Вони містять визначення функцій	Вони містять визначення функцій
Вони доступні всередині підкаталогу проекту	Вони доступні в підкаталогу бібліотек
Файли заголовків читаються людиною. Тому що вони мають форму вихідного коду.	Файли бібліотек неможливо читати, оскільки вони мають форму машинного коду.
Файли заголовків в нашій програмі включені за допомогою команди #include, яка внутрішньо обробляється попередньою обробкою.	Файли бібліотеки в нашій програмі включені в останню стадію спеціальним програмним забезпеченням, що називається компоновщик(linker)

Заголовочні файли

Прототипи зовнішніх функцій, звичайно ж, можна було б безпосередньо написати в текст модуля, але за великій кількості функцій це було б не надто зручно. Більш правильно виносити прототипи функцій в окремий заголовочний файл з розширенням .h за допомогою макросу #include та підключати до кожного модуля, який має намір використовувати ці функції.

Отже, створюючи кожен модуль, програміст в один файл записує тіла кількох функцій в файл з розширеннями .c або .cpp(.cc і т.д.), а в інший, з розширенням

.h (тобто заголовочний) - лише прототипи цих функцій. **Про заголовочний файл кажуть, що він містить інтерфейси модуля або прототипи функцій,** а про файл з розширенням *.c – що він є **реалізацією (імплементацією) модуля.**

Трансляція багатомодульної програми має важливу особливість. Компілятор не обробляє всю багатомодульну програму як одне ціле, а окремо модуль за модулем. Модулі компілюються незалежно один від одного, іншими словами, компілятор, поки компілює один модуль не знає про існування інших модулів. Наприклад, коли модуль unit1.c викликає наприклад, деяку функцію int func1(int) модуля unit2.c, то він в даний момент «бачить» лише інтерфейс функції з unit2.h а не її реалізацію в unit2.c.

Тоді компілятор, поки обробляє модуль використовує ім'я функції, що викликається (воно відомо з прототипу), але не щоб дізнатися тіло цієї функції.

В результаті такої роздільної компіляції кожного с-файлу виходить так званий об'єктний файл, який зазвичай має таке ж ім'я, що й відповідний вхідний файл, але з розширенням .obj для windows-подібних систем або .o для Unix-подібних).

В об'єктному файлі операції над даними та оператори управління вже перекладено у машинні коди, але виклики функцій поки що залишено, як прийнято говорити, наприклад, об'єктний файл unit1.obj містить виклик функції int func1(int), але не саму функцію

Якщо реалізація цієї функції відсутня, то лінковка є нерозв'язаною, оскільки неможливо перетворити на команду переходу до реалізації функції, бо модуль не знає нічого про об'єктний файл unit2.obj , в якому знаходиться це тіло.

Нарешті, після того, як модулі відкомпільовано, спрацьовує ще одна спеціальна програма-компоновщик (лінковщик) або редактор зв'язків.

Ця програма бере всі об'єктні файли, що входять до складу програми, та об'єднує їх в один виконуваний файл (тобто файл, готовий для виконання на машині), знаходячи при цьому посилання на імена функцій та замінюючи кожен виклик функції за іменем, що міститься в одному модулі, на конкретну адресу її реалізації, що взята з іншого модуля.

Примітка 1: Мова С сама по собі призначена для того, щоб писати нею модулі, і тільки для цього. Опис того, з яких модулів складається багатомодульна програма, здійснюється не за допомогою мови програмування, а спеціальними засобами, наприклад Makefile, скріпт на мовах типу Bash, Python або за допомогою середовища IDE (MS Visual Studio etc).

Примітка 2: Стандарт та означення мови С не містять правил щодо імен та розширень заголовочних файлів в принципі в директиві підключення можна вказати ім'я файлу з будь-яким розширенням (головне, щоб цей файл містив текст, зрозумілий для транслятора, тобто текст мовою С). Але за усталеною традицією файлам, спеціально призначеним для включення в інші файли, надають розширення "**.h**" від англ. header (заголовок).

Ще раз відмітимо, що в заголовочних файлах прийнято розміщувати оголошення, потрібні в основному тексті програми. Але важливо розуміти, що використання .h – файлів саме для оголошень не синтаксичне правило, а

традиція. Транслятор не перевіряє, знаходяться у підключеному файлі оголошення чи якісь інші конструкції мови С. Єдине, що транслятор вимагає того щоб у підключеному файлі був деякий текст, правильний з точки зору граматики мови С.

Тому, в принципі, правила мови дозволяють і такий трюк, який, на жаль, часто приваблює початківців: розбити довгий програмний текст на кілька частин, кожну оформити в окремому файлі, а в ще одному файлі зібрати ці частини до купи за допомогою директиви include .

Але ж модулі, повинні компілюватися незалежно один від одного, щоб програма як ціле могла збиратися з цих частин на відміну від описаного підходу. Тому описаний вище помилковий підхід є поганим з дуже багатьох причин і його слід завжди уникати.

Увага!

В заголовочному файлі можна розміщувати: прототипи функцій, оголошення типів (особливо структур), означення макросів і всі такі оголошення, які використовуються на етапі компіляції. В жодному разі не слід розміщувати в заголовочному файлі програмний код, що стосується етапу виконання програми, тобто тіла функцій.

Примітка. Насправді, більшість компіляторів дозволять вам написати тіло функцій в заголовочному файлі. Єдине реальне обмеження - це неможливість написання головної функції main в заголовочному файлі.

Проблема подвійного включення

Використовуючи викладені вище засоби, можна зіткнутися з серйозними незручностями.

Наприклад, на практиці можливі ситуації, коли один заголовочний файл прямо чи непрямо підключається до якогось модуля два або більше разів. Це означає, що кожне оголошення з цього заголовочного файлу потрапить в текст програми кілька разів, а це може викликати помилку компіляції. Розглянемо приклад. Нехай в заголовочному файлі один структурний тип:

```
// файл ratio.h
typedef struct tagRatio {
int m , n ;
} TRatio ;
```

Нехай є ще два заголовочні файли, в яких оголошуються функції для роботи з об'єктами TRatio. У файлі ratio_io.h оголосимо функції для введення та виведення, а в ratio_m.h функції для математичних обчислень з ними. В кожному цих двох заголовочних файлів першим рядком йде підключення файлу ratio.h :

```
// Файл ratio_io.h :
# include "ratio.h"
```

```
TRatio ratio_read ();
void ratio_print ( TRatio );
//Файл ratio_m.h :
# include " ratio . h "
TRatio ratio_add ( TRatio , TRatio );
TRatio ratio_mpy ( TRatio , TRatio );
```

Нарешті, нехай є модуль, який має намір використовувати обидва набори функцій:

```
// Файл main.c
# include " ratio_io . h"
# include " ratio_m . h "
/* якийсь програмний текст */
```

Розберемо, як транслятор прочитає цей текст. Натрапивши на перший рядок, він (спрощено кажучи) вставить на то місце файлу де є директива включення файлу ratio.h (#include ratio_io.h) та спробує обробити його, тобто транслятор підставляє вміст цього файлу ratio_io.h. Так само, обробляючи другий рядок, транслятор змушений буде ще раз вставити текст файлу ratio.h .

Отже, з точки зору транслятора, модуль після обробки всіх директив виглядатиме так:

```
typedef struct tagRatio {
int m , n ;
} TRatio ;
TRatio ratio_read ();
void ratio_print ( TRatio );
typedef struct tagRatio {
int m , n ;
} TRatio ;
TRatio ratio_add ( TRatio , TRatio );
TRatio ratio_mpy ( TRatio , TRatio );
/* якийсь програмний текст */
```

Як можна побачити, при компіляції модуля транслятор побачить два означення структурного типу TRatio - а це є помилкою з точки зору граматики мови С.

Отже, як видно з цього прикладу, помилки можуть виникати через те, що один заголовочний файл підключається до кількох інших заголовочних файлів. Коли ці останні підключаються до якогось модуля, весь текст першого файлу включається багаторазово. Тому виникає задача: винайти такий спосіб написання та використання заголовочних файлів, який би гарантовано уберігав програміста від проблем з подвійним включенням. Іншими словами, хотілося б знайти такий засіб, який би примушував компілятор автоматично слідкувати за спробами багатократно підключити один заголовочний файл, щоб цей файл підключався лише один раз, при першій такій спробі.

Умовна компіляція

Мова С містить директиви препроцесора #ifdef та #ifndef , які дозволяють на етапі компіляції в залежності від тієї чи іншої умови вилучити або, навпаки, включити той чи інший фрагмент тексту.

Конструкцію

```
# ifdef ім 'я
/* якийсь програмний текст */
#endif
```

компілятор обробляє так: спочатку він перевіряє, чи означене дане ім'я , тобто чи був десь раніше при компіляції цього ж модуля означений макрос з таким іменем; якщо ні, то весь програмний текст до директиви ігнорується. Директива #ifndef #endif пропускає, що текст не компілюється, а якщо так, то #endif працює навпаки: вилучає з компіляції текст до директиви тоді, коли макрос з вказаним іменем означений.

Ці директиви, що називають директивами умовної компіляції, мають багато різних застосувань, які, однак, далеко виходять за рамки базового курсу. Зараз опишемо лише одну конструкцію, яка дозволяє вирішити проблему повторного включення.

Розглянемо файл ratio.h у такій редакції:

```
# ifndef _RATIO_H_
# define _RATIO_H_
typedef struct tagRatio {
int m , n ;
} TRatio ;
# endif
```

Нехай він, як і раніше, підключається до модуля двічі через посередництво інших заголовочних файлів. При першому включенні компілятор, натрапивши на директиву продовжить компіляцію, оскільки макрос з іменем _RATIO_H_ раніше не означувався. В наступному ж рядку даний макрос стає означеним. Після цього при другому включенні того ж заголовочного файлу компілятор, знов обробляючи директиву _RATIO_H_ ifndef , помітить, що макрос вже означено, отже пропустить весь текст заголовочного файлу. Таким чином, вдалося запобігти повтору раніше відкомпільованих оголошень.

Продемонстрована тут техніка є стандартною для мови С: якщо придивитися до фірмових заголовочних файлів, що входять, наприклад, до комплекту середовища програмування, в них можна побачити той же технічний прийом.

Примітка 1. Зауважимо також, що ця методика дозволяє також визначити чи був підключений конкретний модуль в дану програму за допомогою виклику директиви #ifdef _RATIO_H_

Примітка 2. Деякі середовища пропонують інший варіант боротьби з повторним включенням - додаванню до модуля директиви #pragma (once), але нажаль такий варіант підтримується не всіма компіляторами.

Включення С-модулів в програму на С++

При включенні С-модулів у проект, що компілюється С-компілятором, можуть виникнути деякі проблеми пов'язані з тим, що компілятори С та С++ трошки по різному компілюють код та, наприклад, інтерпретують прототипи функцій. Але є стандартний спосіб створення заголовочних файлів, що дозволяють їм бути підтриманим на С++ компіляторі при цьому коректно працювати на С - компіляторах.

Приклад.

```
//Файл - sum.h:
#ifdef __cplusplus
  extern "C" {
#endif
int sumI (int a, int b);
float sumF (float a, float b);
#ifdef __cplusplus
} // end extern "C"
#endif
```

Якщо включити цей заголовок у вихідний файл С, то він стане:

```
int sumI (int a, int b);
float sumF (float a, float b);
```

Але якщо включити їх із вихідного файлу С++, то він стане:

```
extern "C" {
int sumI (int a, int b);
float sumF (float a, float b);
} // end extern "C"
```

Мова С нічого не знає про директиву extern "C", але С++ вміє її обробити, і він потребує цієї директиви для оголошень С -функції. Це пов'язано з тим, що С++ викликає імена функцій (і методів) та він підтримує перевантаження функції або методів, тоді як С не підтримує.

Зокрема це можна побачити у вихідному файлі С++ з назвою print.cpp:

```
#include <iostream> // std :: cout, std :: endl
#include "sum.h" // sumI, sumF
void printSum (int a, int b) {
  std::cout << a << "+" << b << "=" << sumI (a, b) << std :: endl;
}

void printSum (float a, float b) {
   std:: cout << a << "+" << b << "=" << sumF (a, b) << std :: endl;
}
```

```
    extern "C" void printSumInt (int a, int b) {
    printSum (a, b);
}
    extern "C" void printSumFloat (float a, float b) {
    printSum (a, b);
}
```

Об'єктні файли

Кожен вихідний файл C та C++ потрібно компілювати в об'єктний файл. Об'єктні файли, отримані в результаті компіляції декількох вихідних файлів, потім пов'язуються з виконуваним файлом, спільною бібліотекою або статичною бібліотекою (останній з них є лише архівом об'єктних файлів). Файли C++ зазвичай мають суфікси розширення .cpp, .cxx або .cc.

Вихідний файл C++ може включати інші файли, відомі як файли заголовків, з директивою #include. Файли заголовків мають розширення, такі як .h, .hpp або .hxx, або взагалі не мають розширення, як у типовій бібліотеці C++ та інших заголовках бібліотек. Розширення не має значення для препроцесора C++ , який буквально замінить рядок, що містить директиву #include, на весь вміст включеного файлу.

Першим кроком, який компілятор зробить у вихідному файлі, буде запустити препроцесор (обробку макросів) на ньому. Тільки вихідні файли передаються компілятору (для попередньої обробки та компіляції). Файли заголовків не передаються компілятору. Натомість вони включені з вихідних файлів.

Кожен заголовочний файл може бути відкритий кілька разів під час фази попередньої обробки всіх вихідних файлів, залежно від того, скільки вихідних файлів містять їх, або скільки інших заголовних файлів, включених з вихідних файлів, також включати їх (може бути багато рівнів непрямого) . З іншого боку, вихідні файли відкриваються тільки один раз компілятором (і препроцесором), коли вони передаються йому.

Для кожного вихідного файлу C++ препроцесор побудує модуль трансляції, вставивши в нього вміст, коли одночасно знайде директиву #include, яку він буде видаляти з вихідного файлу коду та заголовків, поки не знайде умовну компіляцію блоки, директиви яких оцінюються як помилкові. Також будуть виконуватися інші задачі, наприклад, заміна макросів.

Після того, як препроцесор завершить створення цієї (іноді величезної) одиниці трансляції, компілятор починає фазу компіляції і створює об'єктний файл.

Щоб отримати цю одиницю перекладу (попередньо оброблений вихідний код), аргумент (опцію) -E можна передати компілятору (наприклад, в g++/gcc разом з опцією -o), щоб вказати бажане ім'я попередньо обробленого вихідного файлу.

Статичні та динамічні бібліотеки

Існує два типи бібліотек: статичні бібліотеки та динамічні бібліотеки.

Статичні бібліотеки містять об'єктний код, пов'язаний з додатком кінцевого користувача, а потім вони стають частиною виконуваного файлу. Ці бібліотеки спеціально використовуються під час компіляції, що означає те, що бібліотека повинна бути у правильному місці, коли користувач хоче скомпілювати свою програму C або C ++. У Windows вони закінчуються розширенням .lib, а для для Unix-подібних систем та MacOS - вони мають розширення .a.

Статична бібліотека (також відома як архів) складається з процедур, які компілюються і безпосередньо пов'язані з вашою програмою. Під час компіляції програми, яка використовує статичну бібліотеку, вся функціональність статичної бібліотеки, що використовує наша програма, стає частиною виконуваної програми. У Windows статичні бібліотеки зазвичай мають розширення .lib, тоді як у Unix-подібних систем статичні бібліотеки зазвичай мають розширення .a (архіву). Однією з переваг статичних бібліотек є те, що потрібно завантажувати або розповсюджувати лише виконуваний файл, щоб користувачі могли запускати вашу програму. Оскільки бібліотека стає частиною вашої програми, це гарантує, що з вашою програмою завжди використовується правильна версія бібліотеки. Крім того, оскільки статичні бібліотеки стають частиною вашої програми, ви можете використовувати їх так само, як і функціональні можливості, які ви написали для своєї програми. З іншого боку, оскільки копія бібліотеки стає частиною кожного виконуваного файлу, це може призвести до великої кількості втраченого простору. Статичні бібліотеки також не можуть бути легко оновлені - для оновлення бібліотеки необхідно замінити весь виконуваний файл.

Динамічна бібліотека (яка також називається спільною бібліотекою, shared library) складається з процедур, які завантажуються у вашу програму під час виконання. Коли ви збираєте програму, яка використовує динамічну бібліотеку, бібліотека не стає частиною вашого виконуваного файлу - вона залишається окремою одиницею. У Windows динамічні бібліотеки, як правило, мають розширення .dll (динамічна бібліотека зв'язків), тоді як у Linux динамічні бібліотеки зазвичай мають розширення .so (shared object). Однією з переваг динамічних бібліотек є те, що багато програм можуть спільно використовувати одну копію, що економить простір. Можливо, більшою перевагою є те, що динамічну бібліотеку можна оновити до більш нової версії без заміни всіх виконуваних файлів, які використовують її.

Спільні або динамічні бібліотеки потрібні лише під час виконання, тобто користувач може компілювати свій код без використання цих бібліотек. Коротше кажучи, ці бібліотеки пов'язані між собою під час компіляції для вирішення невизначених посилань, а потім поширюються до програми, щоб програма могла завантажувати її під час виконання. Наприклад, коли ми відкриваємо наші папки з іграми, можна знайти багато файлів .dll (динамічних бібліотек). Оскільки ці бібліотеки можуть спільно використовуватися кількома програмами, вони також

називаються спільними бібліотеками. Ці файли закінчуються розширеннями .dll або .lib.

Оскільки динамічні бібліотеки не пов'язані з вашою програмою, програми, що використовують динамічні бібліотеки, повинні явно завантажувати та взаємодіяти з динамічною бібліотекою. Цей механізм може бути заплутаним і робить незручною взаємодію з динамічною бібліотекою. Для спрощення використання динамічних бібліотек можна використовувати бібліотеку імпорту.

Бібліотека імпорту - це бібліотека, яка автоматизує процес завантаження та використання динамічної бібліотеки. У Windows це зазвичай робиться через невелику статичну бібліотеку (.lib) з тією ж назвою, що й динамічна бібліотека (.dll). Статична бібліотека пов'язана з програмою під час компіляції, а потім функціональність динамічної бібліотеки може бути ефективно використана, як якщо б вона була статичною бібліотекою. У Linux файл спільного об'єкта (.so) використовується і як динамічна бібліотека, так і як бібліотека імпорту. Більшість компоновщиків може створювати бібліотеку імпорту для динамічної бібліотеки, коли створюється динамічна бібліотека.

Встановлення та використання бібліотек

Процес, необхідний для використання бібліотеки:

Для кожної бібліотеки:

1) Закачайте бібліотеку. Наприклад, завантажте з веб-сайту або за допомогою менеджера пакетів.

2) Встановіть бібліотеку. Розпакуйте її до каталогу або встановіть за допомогою менеджера пакетів, або проінсталюйте її (наприклад, запустить як C/C++ - програму).

3) Вкажіть компілятору, де слід шукати файл(и) та заголовок(ки) для бібліотеки.

Для кожного проекту:

4) Вкажіть лінкеру (компонувальнику), де шукати файл(и) бібліотеки для бібліотеки.

5) Вкажіть лінкеру (компонувальнику), які статичні файли або файли імпортувати.

6) Включіть файли заголовків бібліотеки у вашу програму.

7) Переконайтеся, що програма знає, де можна знайти будь-які динамічні бібліотеки, які використовуються.

Те саме більш докладно:

Встановлення бібліотеки на C ++ зазвичай включає 4 кроки:

1) Завантаження бібліотеки. Найкращим варіантом є завантаження попередньо скомпільованого пакета для вашої операційної системи (якщо він існує), тому вам

не доведеться самостійно компілювати бібліотеку. Якщо для вашої операційної системи немає такого пакету, вам доведеться завантажити доданок з вихідним кодом і скомпілювати його самостійно (компіляція проекту).

У Windows бібліотеки, як правило, розповсюджуються у вигляді файлів .zip або безпосередньо dll. У Linux бібліотеки зазвичай розподіляються як пакети (наприклад, .RPM) або теж архівом. Менеджер пакетів може мати деякі з найпопулярніших бібліотек (наприклад, SDL), перероблені вже для легкого встановлення, тому перевірте їх там і встановить, наприклад, командою apt install.

2) Встановіть бібліотеку. У Linux це зазвичай передбачає виклик менеджера пакетів і дозвіл (наприклад надання системних прав) йому виконати всю роботу. У Windows це зазвичай передбачає розпакування бібліотеки в потрібний каталог. Ми рекомендуємо зберігати всі ваші бібліотеки в одному місці для легкого доступу. Наприклад, використовуйте каталог з назвою C:\\Libs і помістіть кожну бібліотеку в її власний підкаталог.

3) Переконайтеся, що компілятор знає, де шукати файли заголовків для бібліотеки. У Windows, як правило, це підкаталог include каталогу, до якого ви встановили файли бібліотеки (наприклад, якщо ви встановили бібліотеку до C:\\libs SDL-1.2.11, файли заголовків, ймовірно, знаходяться в C:\\libs, а -1.2.11 виключають з назви). На Linux бібліотеки зазвичай встановлюються в /usr/include, які вже повинні бути частиною шляху пошуку файлів. Однак, якщо файли встановлені в іншому місці, вам доведеться сказати компілятору, де їх знайти.

4) Вкажіть компонувальнику, де шукати файли бібліотек. Як і в кроці 3, це зазвичай передбачає додавання каталогу до списку місць, де лінкер шукає бібліотеки, або додайте бібліотеки в ті місця де проект вже шукає бібліотеки (зокрема в системних шляхах) . У Windows, це типово підкаталог /lib каталогу, до якого ви встановили файли бібліотеки. В Linux бібліотеки зазвичай встановлюються в /usr/lib, які вже повинні бути частиною шляху пошуку вашої бібліотеки.

Після того, як бібліотека встановлена та IDE знає, де її слід шукати, для кожного проекту, який бажає використовувати бібліотеку, як правило, потрібно виконати наступні 3 кроки:

5) Якщо ви використовуєте статичні бібліотеки або імпортовані бібліотеки, повідомте компонувальнику, які файли бібліотек повинні зв'язати (gcc -L(l) *).

6) Включіть файли заголовків бібліотеки у вашу програму (директива include). Це повідомляє компілятору про всі функціональні можливості, які пропонує бібліотека, щоб програма могла правильно скомпілюватись.

7) Якщо використовуються динамічні бібліотеки, переконайтеся, що програма знає, де їх можна знайти. У Linux, бібліотеки, як правило, встановлюються в /usr/lib, який знаходиться в шляху пошуку за замовчуванням або каталоги змінної

середовища PATH. У Windows шлях пошуку за замовченням включає в себе каталог, з якого запускається програма, каталоги, встановлені за допомогою виклику SetDllDirectory(), каталоги Windows, System і System32, а також каталоги змінної середовища PATH. Найпростіший спосіб використовувати .dll - скопіювати .dll в розташування виконуваного файлу. Оскільки ви зазвичай розповсюджуєте файл .dll із вашим виконуваним файлом, має сенс тримати їх разом.

Кроки 3-5 включають налаштування вашого IDE - на щастя, майже всі IDE працюють так само, коли йдеться про виконання цих речей. На жаль, оскільки кожна IDE має інший інтерфейс, найскладнішою частиною цього процесу є просто визначення місця, де потрібне місце для виконання кожного з цих кроків. Ми розглянемо, як виконувати всі ці дії для Visual Studio Express 2005 і Code :: Blocks. Якщо ви використовуєте іншу IDE, прочитайте обидва - до того часу, коли ви закінчите, ви повинні мати достатньо інформації, щоб зробити те ж саме з вашим власним середовищем розробки (IDE) з невеликим пошуком.

Visual Studio Express 2005

Кроки 1 і 2 - Отримання та встановлення бібліотеки

Завантажте та встановіть бібліотеку на жорсткий диск.

Кроки 3 і 4 - Скажіть компілятору, де можна знайти заголовки та файли бібліотеки

Ми будемо робити це на глобальній основі, тому бібліотека буде доступна для всіх наших проектів. Отже, наступні кроки потрібно робити лише один раз на бібліотеку.

A) Перейдіть до меню «Інструменти»("Tools") та виберіть "Опції"("Options")

B) Відкрийте вкладку "Проекти та рішення" ("Projects and Solutions") і натисніть "Директорії VC ++" ("VC++ Directories").

C) У верхньому правому куті в розділі "Показати каталоги для:" ("Show directories for") виберіть "Включити файли"("Include Files"). Додайте шлях до файлів .h для бібліотеки.

D) У верхньому правому куті в розділі "Показати каталоги для:" ("Show directories for") виберіть "Бібліотечні файли"("Library Files"). Додайте шлях до файлів .lib для бібліотеки.

E) Натисніть "OK".

Крок 5 - Вкажіть компонувальнику, які бібліотеки використовує ваша програма

Для кроку 5 нам потрібно додати .lib файли з бібліотеки до нашого проекту. Ми робимо це на індивідуальній основі. Visual Studio пропонує 3 різні методи для додавання .lib файлів до нашого проекту:

A) Використовуйте директиву препроцесора #pragma до основного файлу .cpp. Це рішення працює лише з Visual Studio і не є портативним. Інші компілятори ігноруватимуть цей рядок.

```
#include "curses.h"
#pragma comment(lib, "PDCurses.lib")
```

B) Додайте файл .lib до вашого проекту, як якщо б він був файлом .cpp або .h. Це рішення працює з Visual Studio, але не з багатьма іншими компіляторами.

B) Додайте бібліотеку на вхід лінкера. Це найбільш «портативне» рішення в тому сенсі, що кожен IDE забезпечить подібний механізм. Якщо ви коли-небудь переходите до іншого компілятора або IDE, це рішення вам доведеться використовувати. Це рішення потребує 5 кроків:

C-1) У вкладці рішень клацніть правою кнопкою миші на назву проекту з напівжирним шрифтом і виберіть у меню пункт «Властивості»("Properties").

C-2) У спадному меню "Конфігурація:"("Configuration:") виберіть "Всі конфігурації" ("All Configurations").

C-3) Відкрийте вузол “Властивості конфігурації”("Configuration Properties"), вузол “Linker” і натисніть “Input”.

C-4) У розділі "Додаткові залежності"("Additional Dependencies") додайте назву бібліотеки.

C-5) Натисніть "OK".

Кроки 6 і 7 - включають файли заголовків і переконайтеся, що проект може знайти DLL . Просто включіть заголовок (файли) з бібліотеки у ваш проект.

Середовище Code :: Blocks

Кроки 1 і 2 - Отримання та встановлення бібліотеки

Завантажте та встановіть бібліотеку на жорсткий диск.

Кроки 3 і 4 - Скажіть компілятору, де можна знайти заголовки та файли бібліотеки.

Ми будемо робити це на глобальній основі, тому бібліотека буде доступна для всіх наших проектів. Отже, наступні кроки потрібно робити лише один раз на бібліотеку.

A) Перейдіть до меню «Налаштування» та виберіть «Компілятор».

B) Перейдіть на вкладку "Каталоги". Вкладку компілятора вже буде вибрано для вас.

C) Натисніть кнопку «Додати» і додайте шлях до файлів .h для бібліотеки. Якщо ви працюєте з Linux і встановили бібліотеку за допомогою менеджера пакетів, переконайтеся, що тут є /usr/include

D) Перейдіть на вкладку "Linker". Натисніть кнопку "Додати" і додайте шлях до файлів .lib для бібліотеки. Якщо ви працюєте з Linux і встановили бібліотеку за допомогою менеджера пакетів, переконайтеся, що /usr/lib вказаний тут.

E) Натисніть кнопку "OK".

Крок 5 - Скажіть компонувальнику, які бібліотеки використовує ваша програма. Ми повинні додати файли бібліотеки з бібліотеки до нашого проекту. Ми робимо це на індивідуальній основі для кожного проекту.

A) Клацніть правою кнопкою миші на назву напівжирного проекту під робочим місцем за умовчанням (можливо, "Консольне додаток", якщо ви його не змінили). Виберіть у меню пункт "Параметри побудови".

B) Перейдіть на вкладку лінкера. У вікні «Бібліотеки посилань» натисніть кнопку «Додати» та додайте бібліотеку, яку ви бажаєте використовувати у вашому проекті.

C) Натисніть кнопку "OK"

Кроки 6 і 7 - включають файли заголовків і переконайтеся, що проект може знайти DLL. Просто включіть заголовочні файли з бібліотеки у ваш проект.

Компілювання

Зазвичай програми на мовах C або C++ є сукупністю кількох .c (.cpp) файлів з реалізаціями функцій і .h файлів з прототипами функцій і визначеннями типів даних. Як правило, кожному .c файлу відповідає .h файл з тим же ім'ям.

Припустимо, що розробляється програма, яка називається general і вона складається з файлів gen_file1.c, gen_file1.h, gen_file2.c, gen_file2.h, gen_file3.c, gen_file3.h. Розробка програми ведеться в POSIX-середовищі з використанням компілятору GCC.

Найпростіший спосіб скомпілювати програму - вказати всі вихідні .c файли в командному рядку gcc:

```
gcc gen_file1.c gen_file2.c gen_file3.c -o general
```

Компілятор gcc виконає всі етапи компіляції вихідних файлів програми і компоновку виконуваного файлу general. Зверніть увагу, що в командному рядку gcc вказуються тільки .c файли і ніколи не вказуються .h файли.

Компіляція і компоновка за допомогою перерахування всіх вихідних файлів в аргументах командного рядка GCC допустима лише для зовсім простих програм. З ростом числа вихідних файлів ситуація дуже швидко стає некерованою. Крім того, кожен раз все вихідні файли будуть компілюватися від початку до кінця, що в разі великих проектів займає багато часу. Тому зазвичай компіляція програми виконується в два етапи: компіляція об'єктних файлів і компоновка виконуваної програми з об'єктних файлів. Кожному .c файлу тепер відповідає об'єктний файл, ім'я якого в POSIX-системах має суфікс .o. Таким чином, в даному випадку програма general компонується з об'єктних файлів gen_file1.o, gen_file2.o і gen_file3.o наступною командою:

```
gcc gen_file1.o gen_file2.o gen_file3.o -o general
```

Кожен об'єктний файл повинен бути отриманий з відповідного вихідного файлу за допомогою такої команди:

```
gcc -c gen_file1.c
```

Зверніть увагу, що явно задавати ім'я вихідного файлу необов'язково. Воно буде отримано з імені компільованого файлу заміною суфікса .c на суфікс .o. Отже, для компіляції програми general тепер необхідно виконати чотири команди:

```
gcc -c gen_file1.c
gcc -c gen_file2.c
gcc -c gen_file3.c
gcc gen_file1.o gen_file2.o gen_file3.o -o general
```

Хоча тепер для компіляції програми необхідно виконати чотири команди замість однієї, натомість з'являються такі переваги:

• якщо якась зміна внесена в один файл, наприклад, в файл gen_file3.c, немає необхідності перекомпілювати файли gen_file2.o або gen_file1.o; досить перекомпілювати файл gen_file3.o, а потім виконати компоновку програми general;

• компіляція об'єктних файлів gen_file1.o, gen_file2.o і gen_file3.o не залежить один від одного, тому може виконуватися паралельно на багатопроцесорному (багатоядерному) комп'ютері.

У разі декількох вихідних .c і .h файлів і відповідних проміжних .o файлів відстежувати, який файл потребує перекомпіляції, стає складно, і тут на допомогу приходить програма make. За описом файлів і команд для компіляції програма make визначає, які файли потребують перекомпіляції, і може виконувати перекомпіляцію незалежних файлів паралельно.

Файл A залежить від файлу B, якщо для отримання файла A необхідно виконати деяку команду над файлом B. Можна сказати, що в програмі існує залежність файлу A від файлу B. У нашому випадку файл gen_file1.o залежить від файлу gen_file1.c, а файл general залежить від файлів gen_file1.o, gen_file2.o і gen_file3.o. Можна сказати, що файл general транзитивній залежить від файлу gen_file1.c. Залежність файла A від файлу B називається задоволеною, якщо:

• всі залежності файлу B від інших файлів існують;

• файл A існує в файлової системі;

• файл A має дату останньої модифікації не раніше дати останньої модифікації файлу B.

Якщо всі залежності файлу A існують, то файл A не потребує перекомпіляції. В іншому випадку спочатку задовольняються всі залежності файлу B, а потім виконується команда перекомпіляції файлу A.

Наприклад, якщо програма general компілюється в перший раз, то в файлової системі не існує ні файлу general, ні об'єктних файлів gen_file1.o, gen_file2.o, gen_file3.o. Це означає, що залежно файлу general від об'єктних файлів, а також залежно об'єктних файлів від .c файлів не існують, то всі вони повинні бути перекомпіліьовані. В результаті в файлової системі з'являться файли gen_file1.o, gen_file2.o, gen_file3.o, дати останньої модифікації яких будуть більше дат останньої модифікації відповідних .c файлів (в припущенні, що годинник на

комп'ютері йдє правильно, і що в файлової системі немає файлів "з майбутнього"). Потім буде створено файл general, дата останньої модифікації якого буде більше дати останньої модифікації об'єктних файлів.

Якщо конфігурації всіх залежностей всіх файлів один від одного задовільняються, то для компіляції програми general не потрібно виконувати жодних команд

Припустимо тепер, що в процесі розробки був змінений файл gen_file3.c. Його час останньої зміни тепер більше часу останньої зміни файлу gen_file3.o. Залежність gen_file3.o від gen_file3.c стає незадовілененою, і, як наслідок, залежність general від gen_file3.o також стає незадовільненою. Щоб задовольнити залежності необхідно перекомпілювати файл gen_file3.o, а потім файл general. Файли gen_file1.o і gen_file2.o можна не чіпати, тому що залежності цих файлів від відповідних .c файлів задовільненені. Така загальна ідея роботи програми make і, насправді, всіх програм управління складанням проекту: ant (http://ant.apache.org/), scons (http://www.scons.org/) і ін.

Хоча утиліта make присутня у всіх системах програмування, вигляд керуючого файлу або набір опцій командного рядка можуть сильно відрізнятися. Далі буде розглядатися командна мова і опції командного рядка програми GNU make. У дистрибутивах операційної системи Linux програма називається make. У BSD, як правило, програма GNU make доступна під ім'ям gmake.

Файл опису проекту може містити описи змінних, опису залежностей і опису команд, які використовуються для компіляції. Кожен елемент файлу опису проекту повинен, як правило, розташовуватися на окремому рядку. Для розміщення елементу опису проекту на кількох рядках використовується символ продовження \ точно так же, як в директивах препроцесора мови Сі.

Визначення змінних записуються наступним чином:

<Ім'я> = <визначення>

Використання змінної записується в одній з двох форм:

$ (<Ім'я>)

або

$ {<ім'я>}

Ці форми рівнозначні.

Змінні розглядаються як макроси, тобто використання змінної означає підстановку тексту з визначення змінної в точку використання. Якщо при визначенні змінної були використані інші змінні, підстановка їх значень відбувається при використанні змінної (знову так само, як і в препроцесорів мови Сі)

Залежності між компонентами визначаються наступним чином:

<Мета>: <мета1> < мета 2> ... < мета N>,

де <мета> - ім'я мети, яке може бути або ім'ям файлу, або деяким ім'ям, що позначає дію, який не відповідає жодний файл, наприклад clean. Список цілей в правій частині задає цілі, від яких залежить <мета>.

Якщо опис проекту містить циклічну залежність, тобто, наприклад, файл A залежить від файлу B, а файл B залежить від файлу A, такий опис проекту є помилковим.

Команди для перекомпіляції мети записуються після опису залежності. Кожна команда повинна починатися з символу табуляції (\ t). Якщо жодної команди для перекомпіляції мети не задано, будуть використовуватися стандартні правила, якщо такі є. Для визначення, яким стандартним правилом необхідно скористатися, зазвичай використовуються суфікси імен файлів. Якщо жодна команда для перекомпіляції мети не задана і стандартне правило не знайдено, програма make завершується з помилкою.

Для програми general найпростіший приклад файлу Makefile для компіляції проекту може мати вигляд:

```
general: gen_file1.o gen_file2.o gen_file3.o
   gcc gen_file1.o gen_file2.o gen_file3.o -o general

gen_file1.o: gen_file1.c
   gcc -c gen_file1.c

gen_file2.o: gen_file2.c
   gcc -c gen_file2.c

gen_file3.o: gen_file3.c
   gcc -c gen_file3.c
```

Однак, в цьому описі залежностей не враховані .h файли. Наприклад, якщо файл gen_file1.h підключається в файлах gen_file1.c та gen_file2.c, то зміна файлу gen_file1.h повинна приводити до перекомпіляції як gen_file1.c, так і gen_file2.c. Тобто .o файли залежать не лише від .c файлів, але й від .h файлів, які включаються даними .c файлами безпосередньо або опосередковано. З урахуванням цього файл Makefile може отримати наступний вигляд:

```
general: gen_file1.o gen_file2.o gen_file3.o
   gcc gen_file1.o gen_file2.o gen_file3.o -o general

gen_file1.o: gen_file1.c gen_file1.h
   gcc -c gen_file1.c

gen_file2.o: gen_file2.c gen_file2.h gen_file1.h
   gcc -c gen_file2.c

gen_file3.o: gen_file3.c gen_file3.h gen_file1.h
   gcc -c gen_file3.c
```

З урахуванням цих доповнень файл Makefile прийме вигляд:

```
all: general

general: gen_file1.o gen_file2.o gen_file3.o
   gcc gen_file1.o gen_file2.o gen_file3.o -o general

gen_file1.o: gen_file1.c gen_file1.h
   gcc -c gen_file1.c

gen_file2.o: gen_file2.c gen_file2.h gen_file1.h
   gcc -c gen_file2.c

gen_file3.o: gen_file3.c gen_file3.h gen_file1.h
   gcc -c gen_file3.c

clean:
   rm -f general *.o
```

Отримаємо наступний файл:

```
CC = gcc
CFLAGS = -Wall -O2
LDFLAGS = -s

all: general

general: gen_file1.o gen_file2.o gen_file3.o
   $(CC) $(LDFLAGS) gen_file1.o gen_file2.o gen_file3.o -o general

gen_file1.o: gen_file1.c gen_file1.h
   $(CC) $(CFLAGS) -c gen_file1.c

gen_file2.o: gen_file2.c gen_file2.h gen_file1.h
   $(CC) $(CFLAGS) -c gen_file2.c

gen_file3.o: gen_file3.c gen_file3.h gen_file1.h
   $(CC) $(CFLAGS) -c gen_file3.c

clean:
   rm -f general *.o
```

CMake - програма для створення makefile-ів

CMake одна з найпопулярніших (де-факто стандарт) кросплатформених програм для збирання проектів на C та Ci++. CMake не провадить збірку проекту, він лише генерує зі свого файлу збірки make-файл для конкретної платформи. Це може бути проект Microsoft Visual Studio, NMake makefile, проект XCode для MacOS, Unix makefile, Watcom makefile, проект Eclipse або CodeBlocks.

Даний файл зазвичай називається CMakeLists.txt і містить команди зрозумілі CMake. Виконавши

```
cmake CMakeLists.txt
```

ми отримаємо або проект який можна відкрити у вашому середовищі розробки, або makefile, за яким можна зібрати проект, запустивши відповідну систему збирання (make, nmake, wmake). Завжди можна явно вказати, що повинна генерувати програма, вказавши ключ -G з потрібним параметром (просто запустіть cmake, щоб побачити доступні варіанти).

Дистрибутив CMake можна скачати з офіційного сайту (http://cmake.org/cmake/resources/software.html), де є версії для всіх популярних платформ або встановити зі сховищ вашого дистрибутива Linux, швидше за все він там уже є.

```
cmake_minimum_required (VERSION 2.6)

set (PROJECT hello_world)
project (${PROJECT})
set (HEADERS hello.h)
set (SOURCES hello.cpp main.cpp)
add_executable ($ {PROJECT} $ {HEADERS} $ {SOURCES})
```

У першому рядку просто вказана мінімальна версія CMake необхідна для успішної інтерпретації файлу.

У третьому рядку визначається змінна PROJECT і їй задається значення hello_world - так буде називатися наша програма.

Про що і йдеться в п'ятому рядку. Конструкція ${ім'я_змінної} повертає значення змінної, таким чином проект буде називатися hello_world.

У сьомому і десятому рядках вводяться змінні, які містять список файлів необхідних для складання проекту.

І в останньому рядку йде команда зібрати виконуваний файл з ім'ям зазначеному в змінної PROJECT і з файлів, імена яких знаходяться в змінних HEADERS і SOURCES.

Цей шаблон CMake зручно використовувати де завгодно, для цього достатньо змінити назву проекту і імена файлів.

Збирання бібліотеки з CMake

Для збирання бібліотеки потрібно виконати такі самі команди, як і виконуваному файлі, але в останньому рядку замість команди add_executable, вказують команду add_library. У цьому випадку буде зібрана статична бібліотека, для складання динамічної бібліотеки треба вказати параметр SHARED після імені бібліотеки:

```
add_library ($ {PROJECT} SHARED $ {HEADERS} $ {SOURCES})
```

Стильові вимоги

Правила написання CMake-файлів досить лояльні:

- Імена змінних пишуться англійськими літерами у верхньому регістрі
- Команди CMake записуються англійськими літерами в нижньому регістрі
- Параметри команд CMake пишуться англійськими літерами у верхньому регістрі
- Команди CMake відокремлені пропуском від відкриває дужки

Розташування CMakeLists.txt

Дуже зручно в корені кожного проекту мати директорію build з файлами збірки проекту, причому ім'я файлу у всіх директоріях має називатися однаково (назва за замовчуванням CMakeLists.txt відмінно підійде). Це дозволить збирати складні проекти рекурсивно, підключаючи директорії з підпроектів.

Створювати окрему директорію, а не зберігати файл в корені зручно, як з міркувань чітко організованої структури проекту, так і тому, що CMake при роботі створює ряд файлів, які зручно зберігати окремо, щоб не захаращувати проект.

Збирання складного проекту

Ускладнити завдання і зберемо проект, що залежить від бібліотеки. Для цього треба зробити наступні речі:

- Зібрати бібліотеку
- Підключити бібліотеку до проекту

З пунктом 1 складнощів виникнути не повинно, про те як зібрати виконуваний файл або бібліотеку написано вище.

Тепер про те як цю бібліотеку підключити. Зробити це можна додавши в CMakeLists.txt проекту три рядки:

```
include_directories (../)
add_subdirectory (../ІМЯ_БІБЛІОТЕКІ/build bin / ІМЯ_БІБЛІОТЕКІ)
target_link_libraries ($ {PROJECT} ІМЯ_БІБЛІОТЕКІ)
```

Перший рядок потрібен для того, щоб вказати шлях (в даному випадку це корінь проекту) по якому компілятор буде шукати спільні заголовки.

У другому рядку йде вказівка CMake взяти файл з директорії build підпроекту, виконати його і результат роботи покласти в директорію ./bin/ІМЯ_БІБЛІОТЕКІ.

Третій рядок повинна бути в файлі після команди add_executable і вказує, що проект треба лінковані разом із зазначеною бібліотекою.

Можна додавати ці команди вручну, але можна змінити шаблон, щоб було можна змінювати тільки один рядок, вказуючи ім'я бібліотеки:

```
cmake_minimum_required (VERSION 2.6)
set (PROJECT ІМЯ_ПРОЕКТА)
project ($ {PROJECT})
include_directories (../)
set (LIBRARIES ІМ'Я_БІБЛІОТЕКІ_1 ІМ'Я_БІБЛІОТЕКІ_2 ***)

foreach (LIBRARY $ {LIBRARIES})
    add_subdirectory (../${LIBRARY}/build bin / $ {LIBRARY})
endforeach ()

set (HEADERS ../ЗАГОЛОВОЧНІ_ФАЙЛИ)
set (SOURCES ../ФАЙЛИ_С)
add_executable ($ {PROJECT} $ {HEADERS} $ {SOURCES})
target_link_libraries ($ {PROJECT} $ {LIBRARIES})
```

Примітка. Якщо проекті Microsoft Visual Studio не показані заголовки, то потрібно додати в файл два рядки:

```
source_group ( "Header Files" FILES $ {HEADERS})
source_group ( "Source Files" FILES $ {SOURCES})
```

На роботу CMake при генерації файли для інших систем це ніяк не вплине.

Примітка

Попередження компілятора можна включити в такий спосіб (приклад для MS VS і GCC):

```
if (MSVC)
    add_definitions (/ W4)
elseif (CMAKE_COMPILER_IS_GNUCXX)
    add_definitions (-Wall -pedantic)
else ()
    message ( "Unknown compiler")
endif ()
```

Література

1 Об'єктно-орієнтоване програмування мовою С++ / Ю.І. Грицюк, Т.Є. Рак

2 Програмування мовою C++ / Ю.І. Грицюк, Т.Є. Рак
3 Алгоритмічні мови та основи програмування: мова С / В.Ю. Вінник
4 C++. Основи програмування. Теорія та практика / О.Г. Трофименко
5 Ю.А.Бєлов, Т.О.Карнаух, Ю.В.Коваль, А.Б. Ставровський. Вступ до програмування мовою C++. Організація обчислень. Навчальний посібник
6 http://www.cplusplus.com/
7 https://purecodecpp.com/uk/
8 http://cpp-info-ua.blogspot.com/
9 ANSI 89 – American National Standards Institute, American National Standard for Information Systems Programming Language C, 1989.
10Kernighan 78 – B. W. Kernighan and D. M. Ritchie, The C Programming Language, Prentice-Hall: Englewood Cliffs, NJ, 1978. Second edition, 1988.
11Thinking 90 – C* Programming Guide, Thinking Machines Corp. Cambridge Mass., 1990.
12Річі К. Мова програмування С
13Керниган, Б. Язык программирования Си. Задачи по языку Си / Б. Керниган, Д. Ритчи, А. Фьюэр. – М. : Финансы и статистика, 1985. – 279 с.
14С у задачах і прикладах : навчальний посібник із дисципліни "Інформатика та програмування" / А.П. Креневич, О.В. Обвінцев. – К. : Видавничо-поліграфічний центр "Київський університет", 2011. – 208 с.
15Уэйт М. Язык Си / М. Уэйт, С. Прата, Д. Мартин. – М. : Мир, 1988. – 512 с
16https://en.cppreference.com/w/c/io/fscanf
17http://www.c-cpp.ru/content/scanf
18http://www.cplusplus.com/reference/cstdio/scanf/
19https://purecodecpp.com/uk/archives/920