**Вказівники**

(лекція 1)

У відповідних частинах оперативної пам'яті зберігаються:

* вхідні дані;
* результати роботи програми;
* проміжні дані тощо.

**Усі дані (крім абсолютних констант) мають адресу розташування в оперативній пам'яті комп'ютера.**

Можливість доступитися до даного за його адресою з використанням спеціального типу даних:

|  |  |
| --- | --- |
| +Паскаль і похідні;  +С і похідні | -Basic  -Visual Basic  -Java |

**Потреба в застосуванні вказівників:**

1. доступ до елементів масиву, бо такий спосіб є швидшим від операції індексації;
2. передавання аргументів у функцію, в якій ці аргументи зазнають змін;
3. передання у функцію масивів і стрічкових змінних;
4. виділення пам'яті під час виконання програми;
5. створення складних динамічних структур (списки, дерева, графи).

**Адреса змінної. Оператор &**

**Операція встановлення адреси & (амперсанд)** - унарна операція для визначення адреси певного даного в ОП.

Числове зображення адреси змінної у пам'яті комп'ютера:

* подається у 16-ій системі числення;
* у середовиші MSVS є 32-ох розрядна;
* адреса кожного байту ОП - вісім 16-их цифр.

Якщо оголошено змінні чи поіменовані константи, тоді операцією & можна встановити відповідні адреси та вивести іх на екран.

**Зверніть увагу!**

Пам'ять не виділяється під:

* абсолютну константу: сout< <&2;
* окремий вираз: соut< <& (а+b);

**Синтаксис оголошення вказівника**

**Вказівник** - це комірка пам'яті, що призначена для зберігання адреси даного;

* **змінна-вказівник** (або просто вказівник) - зберігає адресу змінної;
* **вказівник на константу** - зберігає адресу поіменованої константи;
* **константа вказівник**-вказівник, адресний вміст у якому змінюватися не може;

**Якого типу сам вказівник???**

* **Нe iснує** окремого зарезервованого слова, що встановлює приналежність даного до **типу вказівник**.
* Вказівник містить **адресу даного конкретного типу.**

**тип\_даного \* ім'я\_змінної\_вказівника;**

* тип-даного - будь-який зі стандартних типів даних мови, а також тип користувача, який задекларований раніше;
* у MSVS під змінну - вказівник комп'ютер виділяє **4 байти** (32 розряди);
* **ВАЖЛИВО**, щоби при оголошенні вказівника символ \* був наявним **обов'язково!**

int \* ptr\_int\_1,ptr\_int\_2;

int \* ptr1; int \* ptr2; іnt \* ptr3; int \* ptr4, \* ptr5;

**Вказівники повинні мати значення!**

У випадку використання **неініціалізованого вказівника** ("сміттєвої" адреси) відповідна адреса може виявитись адресою будь-чого: **від коду програми до коду операційної системи.**

**Надання вказівнику значення:**

* Ініціалізація адресою вже існуючого даного під час оголошення;
* наступне присвоєння конкретної адреси даного після оголошення;
* надання адресній комірці значення NULL чи просто 0;
* присвоєння значення адреси, яка зберігається в комірці, що також є вказівником на дане такого ж самого типу.

**Звертання до змінної за її адресою**

**Чи можна за відомою адресою існуючої змінної встановити вміст інформації за цією адресою?**

**Непряме звертання -** використання адреси змінної замість її ідентифікатора для звертання до самої змінної, до її значення, яке міститься у відповідній комірці. **Операція розадресації** - операція, яка дозволяє втілювати непряме звертання.

**Операція розадресації (\*)**

**\* вказівник**

* є **унарною** адресною операцією;
* виконується над даним, яке є вказівником;
* дозволяє використати **дане**, яке є за адресою, записаною в комірці, що є вказівником;
* **може бути:**
  + у **правій** частині оператора присвоєння - використовується значення змінної за її адресою;
  + у **лівій** частині оператора присвоєння - змінюється вмістиме комірки, адреса якої записана у відповідному вказівнику;
  + окремим **операндом** у виразі;
  + **фактичним параметром** функції;
* дозволяє **використовувати і змінювати** значення змінної, прямий доступ до якої неможливий.

**Оператор присвоєння для вказівників**

**Синтаксис:**

* ім'я \_змінної\_вказівник\_1 = & ім'я\_змінної;
* iм'я \_змінної \_вказівник \_2 = ім'я\_змінної \_вказівник\_1;

де обидва вказівники на змінні **однакового** типу.

**Зверніть увагу!**

Неявного приведення типу для вказівників не передбачено!

**Вказівник на void**

**Вказівник на уoid** - тип вказівника, який може вказувати на **дане будь-якого типу**

void \* iм'я\_вказівника;

* Призначені для передавання параметрів-вказівників у функції, що працюють незалежно від типу даних, на які вказує вказівник.
* Вказівникові на void **можна** присвоювати як значення конкретного вказівника, так і адресу змінної довільного типу.
* Вказівникові на конкретний тип присвоювати значення адреси, яка була записана у вказівник на void **НЕ можна**.
* **НЕ можна** звертатися за відповідним вмістом через операцію розадресації \*.

**Зв'язок між масивами та вказівниками**

* Ім'я статичного масиву є коміркою, яка містить адресу області оперативної пам'яті, що виділена для зберігання однотипних даних, об'єднаних у масив.
* Доступ до елементів кожного масиву здійснюється за операцією індексації **[ ]**, тобто розадресації:

**ім'я\_масиву[індекс елемента масиву].**

* **Вказівнику** на відповідний тип, можна присвоїти **адресу області, де зберігається масив** даних такого типу.
* Використовуючи операцію індексації до змінної-вказівник, можна мати доступ до кожного елемента масиву:

**\*(ім'я\_вказівника + лічильник).**

**Зв'язок між масивом даних певного типу та вказівником на цей тип.**

* Записи

ar\_int [i], \*(ar\_int + i), ptr\_ar\_1[i], \*(ptr\_ar\_1 + i), ptr\_ar\_2[i] та \*(ptr\_ar\_2 + i)

**є майже еквівалентними;**

* Виведення елементів масиву можна подати так:

for (int i = 0; i < n; i++)

сout < <\*(ar\_int +i)<< "\t"<<\*(ptr\_ar\_1 + i) << "\t"<<\*(ptr\_ar\_2+ i)<< "\n";

* Перший елемент будь-якого одновимірного масиву може бути взятий так

\*ім'я\_масиву.

**Вказівники та посилання**

(лекція 2)

**Операції над вказівниками**

**Унарні операції над вказівниками**

* Операція **інкременту** (++) збільшує вказівник на одиницю обсягу пам'яті, яку займає дане, що на нього вказує вказівник;
* Операція **декременту** (- -) зменшує вказівник на одиницю обсягу пам'яті, яку займає відповідне дане;
* **Префіксний** варіант - спершу змінюється значення у відповідній комірці, а тоді використовується сама комірка;
* **Постфіксний** варіант - спершу використовується значення з комірки, а тоді змінюється вміст комірки.

Унарні операції мають однаковий пріоритет, але виконуються справа наліво.

**Зауваження**

Операції інкременту та декременту можна виконувати й над вказівниками, що містять адреси простих змінних, але адресне зміщення не завжди дасть нам значення наступної змінної, оскільки в MSVS між локальними змінними є 8 байтів пам'яті.

**Бінарні операції над вказівниками**

* **адитивні операції** додавання чи віднімання цілого числа (ptr±i) - зміна адреси на величину, яка рівна кількості байт пам'яті, що їх займає вказане число даних того типу, на який вказує вказівник;
* **встановлення різниці** двох вказівників (віднімання вказівників: ptr1-ptr2) - результатом є число елементів даного типу, які можуть бути записані в області між вказаними двома адресами.

**Логічні операції**

* **Операції порівняння** - значення адреси можуть співпадати чи не співпадати.
* **Логічні операції (!, &&, ||)** - значення вказівника, що не дорівнює NULL, дає істинне значення виразу, а для порожньої адреси хибне.

**Вказівники на константу**

const тип\_даного \* ім'я\_вказівника;

* вважається змінною-вказівником на константу певного типу;
* можна одразу присвоїти адресу оголошеної раніше константи або зробити це пізніше звичайним присвоєнням;
* дозволено виконувати усі операції, як і над вказівниками на змінні;
* вмістиме комірки за цією адресою змінювати **НЕ можна.**

**Константні вказівники**

**Константний вказівник на змінну**

тип\_даного ім'я\_змінної;

тип\_даного \* const ім'я\_вказівника = &iм'я\_змінної:

* при оголошенні вказівникові одразу слід надали адресу попередньо оголошеної змінної;
* змінювати адресу, яка міститься у вказівнику, на адресу іншої змінної **НЕ можна**;
* константний вказівник **НЕ можна** ініціалізувати адресою константи відповідного типу;
* вмістиме комірки за цією адресою можна змінювати за допомогою операції розадресації.

**Константний вказівник на константу**

const тип\_даного ім'я\_константи = значення;

тип\_даного \* const ім'я\_вказівника = &ім'я\_константи;

Над константним вказівником на константу **НЕ можна виконувати жодної операції,** яка передбачала би зміну як адреси, так і значення самої константи, вказівником на яку ми оперуємо.

**Поняття динамічної пам'яті**

**Статична пам'ять**

* відбувається виділення пам'яті при оголошенні змінної (числової, символьної, адресної);
* відповідна змінна займає обсяг ОП доти, поки це визначено класом пам'яті змінної.

**Динамічна пам'ять**

* пам'ять під змінну виділяється **за потребою під час виконання програми;**
* змінна "живе" в комп'ютера доки пам'ять не звільнять програмно або до кінця роботи програми;
* область ОП, яка виділяється, називають **"купою"** ("heap");
* початкове значення такої змінної є непередбачуване, тому вимагає чіткої ініціалізації.

**Засоби мови для роботи з динамічною пам'яттю**

**При роботі з масивами**

1. необхідно ще до запуску на виконання знати, **який обсяг пам'яті слід зарезервувати під масив**;
2. необхідно оголосити розмірність масиву за допомогою абсолютної чи поіменованої константи;
   1. якщо розмір масиву є **меншим** за зарезервовану пам'ять, то частина пам'яті не використовується;
   2. якщо зарезервований обсяг пам'яті є **недостатнім**, то потрібно вносити зміни у код, перекомпільовувати його;
3. потрібно мати інструмент, для виділення бажаного розміру ОП у процесі виконання програми.

**Оператор new**

**Операція виділення динамічної пам'яті** передбачає виділення в області динамічної пам'яті, достатньої для зберігання даного відповідного розміру (типу).

* адреса першого байта виділеної області пам'яті записується у вказівник на відповідний тип;
* доступ до даного здійснюється не через ім'я комірки, а через її адресу, яка зберігається у вказівнику,

**Синтаксис**

**тип\_даного \* ім'я вказівника;**

**тип\_вказівника = new тип\_даного;**

або

**тип\_даного \* ім'я\_вказівника = new тип\_даного;**

* Назви типу даного при оголошенні вказівника і при виконанні операції nеw є **однаковими!**
* Доступ до вмісту цієї області здійснюється **виключно** операцією розадресації, бо ця область імені не має

**Динамічне виділення пам'яті під масив**

**тип\_даного \* ім'я\_вказівника;**

**iм'я\_вказівника = new тип\_даного [розмір масиву];**

* Розмір масиву - додатне ціле значення (змінна, поіменована чи абсолютна константа).
* В області heap буде виділено суцільну область, розміром:

розмір\_масиву × sizeof (тип\_даного).

**Важливо!**

* **НЕ можна** виділити область розміром, кратним розміру типу даного, наприклад:

prt\_dbl\_ar = new double \* n;

prt\_dbl\_ar = new n \* double;

* Якщо випадково у комірку з адресою **динамічного масиву** внести якусь іншу адресу, то виділена частина динамічної пам'яті стане **НЕДОСТУПНОЮ** як програмі, так і системі аж до моменту завершення програми.

**Отже,** наявність такого потужного інструменту дозволяє раціонально використовувати пам'ять під час виконання програми.

**Оператор delete**

**Операція звільнення динамічної пам'яті** передбачає звільнення області динамічної пам'яті для майбутнього використання з під даних, які перестали бути потрібними.

**delete iм'я\_вказівника;**

або

**delete [ ] iм'я\_вказівника;**

**Важливо!**

* Якщо при звільненні пам'яті, виділеної під масив, не вказати [ ], то в область heap буде повернено лише одну комірку (1-ий елемент масиву), а решта стануть **НЕДОСТУПНИМИ** до кінця виконання програми.
* **Втрата пам'яті** - недоступність даних у динамічній пам'яті через втрату адреси цієї області, а також при незвільненні області з-під зайвої інформації.

**Поняття посилання у С++**

**Посилання (reference)** — це інше ім'я (псевдо) вже існуючої, оголошеної змінної чи вказівника:

* використовується для передавання аргументів у функцію;
* має ту ж саму адресу, що й змінна, на яку воно оголошене;
* оголошується **НЕ** на тип, а на дане конкретного типу;
* кожне посилання оголошується тільки для конкретної змінної;

**Якщо над посиланням на змінну виконається деяка операція, то ця операція виконається над самою змінною.**

**Синтаксис оголошення посилання**

**тип\_даного & ім'я\_посилання = ім'я\_існуючої\_змінної;**

* **Посилання повинно бути ініціалізованим при оголошенні.**
* Посилання **НЕ** можна оголошувати для абсолютної константи.
* Посилання можна оголошувати на вказівник i використовувати це друге, альтернативне ім'я для доступу до змінної чи динамічного виділення пам'яті.

**Синтаксис оголошення const посилання**

**const тип\_даного ім'я\_константи = значення;**

**const тип\_даного & ім'я\_посилання = ім'я\_константи;**

* оголошується для константи;
* дозволяє передати функцію параметр, який є константою за означенням;
* поіменована константа, якщо це не формальний параметр функції, вже має бути оголошеною.

**Передача параметрів у функцію. Багатовимірні динамічні масиви**

(лекція 3)

**Передача аргументів у функцію**

**Під час виконання функції:**

* записуються у стек **копії** змінних чи констант, перелічених у списку аргументів (формальні та фактичні параметри екрануються);
* викликається процедура з поверненням одного результату через ім'я функції.

**Способи передачі параметрів у функцію:**

**мова С**

1. за значенням;
2. за передаванням адреси або вказівником;

**мова С++** за посиланням

**Передача параметрів за значенням**

**Передача за значенням** - це спосіб передачі аргументів, при якому функція створює копії переданих значень.

* передаються копії змінних у функцію;
* відсутня жодна можливість впливу функції на змінні в точці виклику;
* оригінали фактичних аргументів містяться у зовсім іншій області пам'яті, ніж копії, які опрацьовує і змінює функція;
* здійснюється, якщо кількість значень, що необхідно повернути (не масиви) є не більше одного.

double gorner (double a[ ], int n, double x)

{

double rez = a[0];

for (int i = 1; i <= n; rez = rez\*x + a[i++]);

return rez;

}

* функція **gorner** повертає одне значення **rez**;
* для всіх параметрів передаються копії
  + для масиву а-копія адреси області, в якій є масив;
  + для n i х - копії відповідних типів.

**Передача параметрів за вказівником**

**Передача за вказівником** - це спосіб передачі аргументів, при якому в якості параметрів функції передаються не копії змінних, а копії їх адрес розташування у пам'яті.

* через фактичний параметр передається копія адреси комірки пам'яті, де зберігається змінна;
* функція може змінити значення змінних у точці виклику; передані адреси при цьому не змінюються;
* *формальні параметри -* вказівники, *фактичні параметри* адреси змінних;
* для доступу до змінної в тілі функції використовують операцію розадресації.

**Передача аргументів у функцію**

**за значенням**

* коли треба захистити аргументи від несанкціонованого доступу;
* коли функція не має наміру змінювати фактичні параметри;

**за вказівником;**

* коли є потреба у зміні аргументів коли треба передати аргументи, що займають значний обсяг пам'яті;

**за посиланням**

* коли треба повернути змінену адресу.

**Передача параметрів за посиланням**

**Передача за посиланням** - це спосіб передачі аргументів, який не передбачає створення копії **HI** адреси, **HI** самого даного, бо оперує напряму з адресою даного.

* у функцію передається посилання на змінну;
* функція має прямий, безпосередній доступ до значень аргументів, переданих за посиланням;
* можна повертати не одне, а декілька значень параметрів;
* *формальні параметри* - посилання на змінні (позначаються символом & після вказання типу параметра), *фактичні параметри* - імена змінних;
* в якості фактичного параметра **НЕ** може бути абсолютна константа.

**Вказівник на функцію**

**Поняття адреси функції**

* Об'єктний код функції розташований в певній області ОП.
* Ця область має адресу, яка асоційована з іменем функції.

**Вказівник на функцію** - комірка, що містить адресу функції.

**Необхідність у використанні вказівника на функцію**

* Якщо список формальних параметрів містить функцію.
* Фактичним параметром такої функції може бути ім'я будь-якої функції, тип результату якої співпадає з означеними.
* Дозволяє створити статичний масив вказівників на функції.

**Синтаксис оголошення**

**тип\_результату ( \* ім'я\_вказівника) (список параметрів);**

* Ім'я вказівника на функцію з **обов'язковим** символом \* перед ідентифікатором береться у дужки (тобто це вказівник на функцію, а не функція, що повертає вказівник на щось).
* Тип результату може бути довільним, в тому числі й void.
* Другі дужки зі списком чи без списку формальних параметрів є **обов'язковими**, бо є ознакою функції, яка передається.

**Масив вказівників на функції** - функції, що повертають однаковий результат і мають однаковий список формальних параметрів, можна об'єднати у масив

**тип\_результату (\*ім'я\_вказівника [розмір]) (список параметрів);**

**Поняття багатовимірного масиву**

**Багатовимірний масив** - це одновимірний масив, кожен елемент якого є масивом на одиницю меншої розмірності.

**Ім'я масиву** - це вказівник на дане того типу, якого є елементи цього масиву.

* Якщо масив **статичний**, то ім'я масиву — це константний вказівник.
* Якщо масив **динамічний**, то адреса виділеної області пам'яті зберігається у змінній-вказівнику;
* **Якщо елементи масиву:** 
  + **дані**, то ім'я масиву є вказівником на їхній тип;
  + **вказівники**, то ім'я масиву є вказівником на вказівник.

**Вказівник на вказівник**

**Синтаксис оголошення**

* оголошення вказівника на змінну певного типу:

**тип \* ім'я\_вказівника;**

* оголошення вказівника на вказівник:

**тип\_даного \*\* ім'я\_вказівника\_на\_вказівник;**

* символів \* буде стільки, якою є "зануреність вказівниковості".

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **2001** |  | **4001** |  | **6001** |
|  |  |  |  |  |
| **Value**  **3** |  | **Value**  **2001** |  | **Value**  **4001** |
| **а** |  | **ptr** |  | **pptr** |

**Доступ до даних**

Щоб отримати доступ до даного (а), що його адреса зберігається у змінній-вказівнику (рtr), адреса якого міститься в іншій комірці (рptr), використовують операцію непрямого доступу.

**\*\* ім'я\_вказівника\_на\_вказівник**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **2001** |  | **4001** |  | **6001** |
|  |  |  |  |  |
| **Value**  **3** |  | **Value**  **2001** |  | **Value**  **4001** |
| **а** |  | **ptr** |  | **pptr** |

**Передача вказівника у функцію**

**Вказівник на вказівник використовують для повернення зміненої адреси у викликаючу функцію:**

* змінена адреса **НЕ** повертається у викликаючу функцію, оскільки передається копія адреси даного, а не копія адреси розташування даного:

**тип\_результату ім'я\_функції (тип \* ім'я вказівника);**

* змінена адреса повертається у викликаючу функцію, якщо *формальний   
  парамет*р - вказівник на вказівник, а *фактичний* - адреса комірки, за якою міститься адреса даного:

**тип\_результату ім'я\_функції (тип \*\* ім'я вказівника);**

* для повернення зміненої адреси в С+ + можна використати механізм   
  посилань - тип даного \* &.

**Багатовимірні динамічні масиви**

**Недоліки використання статичних масивів:**

1. нераціональне використання пам'яті:

* виділення недостатнього чи зайвого обсягу пам'яті при оголошенні масиву

1. у випадку багатовимірних масивів (масивів масивів) масиви, які є відповідними елементами, повинні бути **однакової**, заданої при оголошенні розмірності;

**Двовимірні динамічні масиви**

**Оголошення двовимірного динамічного масиву**

* оголошується вказівник на вказівник на дане такого типу, які формуватимуть динамічний масив:

**тип\_даних \*\* ідентифікатор;**

* використовуючи операцію new, виділяється пам'ять під масив вказівників потрібної розмірності;
* в кожну комірку динамічного масиву вноситься адреса нововиділених областей пам'яті під відповідні масиви даних **необов'язково** однакової розмірності.

double \*\* ptr\_ptr;

int n = 5, m = 4;

ptr\_ptr = new double \* [n];

for (int i = 0; i < n; i++)

ptr\_ptr[i] = new double[m];

**Звільнення пам'яті**

* при зміні вмістимого вказівника втрачається можливість доступу до раніше виділеної області;
* звільнення динамічної пам'яті здійснюється в порядку оберненому до виділення:

1. спочатку розриваються зв'язки кожного з масивів найменшої вимірності:
2. вкінці звільняється пам'ять від масиву вказівників;

for (int i =0; i< n; i++)

delete[ ] ptr\_ptr[i];

delete[ ] ptr\_ptr;

**Важливо!**

1. Доступ до елемента масиву можна здійснювати

* операцією індексації:

ptr\_ptr\_i[i][j]

* операцією розадресації:

\*(\*(ptr\_ptr\_i + )+j)

1. Якщо при звільненні динамічної памягі одразу написати

delete[ ] ptr\_ptr;

то система до завершення роботи програми **НЕ** матиме доступу до сегментів динамічної пам'яті, в яких були відповідні "рядки" матриці.

**Розташування масиву в ОП**

* У комірку, що є вказівником на вказівник записується адреса суцільно розподіленої області динамічної пам'яті.
* У кожну комірку цього масиву записуються адреси суцільних областей, але кожна область є виокремленою і незв'язаною ні з попередньою, ні з наступною.
* Кожна область має зв'язок лише зі своїм вказівником.

**Отже,** двовимірний динамічний масив **НЕ** є суцільно розподіленою областю як статичний, а є **фрагментарною областю**.

|  |  |
| --- | --- |
| **Розташування масиву в ОП** |  |

**Тривимірні динамічні масиви**

**Оголошення тривимірного динамічного масиву**

1. Оголошується вказівник на тип даних з трьома зірочками;
2. Виділяється пам'ять під масив, кожен елемент якого є вказівником на вказівник, адреса цієї області й записується в оголошений вказівник;
3. У циклі кожну комірку масиву заповнюють адресами областей під одновимірні масиви вказівників;
4. У двох вкладених циклах здійснюється виділення області динамічної пам'яті під одновимірні масиви для даних відповідного типу.

**Приклад оголошення тривимірного динамічного масиву**

тип\_даних \*\*\* ім'я = new тип\_даних \*\* [зозмір\_1];

for (int i = 0; i < розмір\_1; i++)

ім'я [i] = new тип\_даних \*[розмір\_2];

for (int i = 0; i < розмір\_1; i++);

for (int j = 0; j < розмір\_2; j++)

ім'я[i][j] = new тип\_даних \*[розмір\_3];

**Звільнення пам'яті** відбувається в **оберненому** порядку до порядку виділення.

**Стрічки** (лекція 4-5)

**Поняття стрічки**

**При розв'язуванні значного числа завдань потрібно**

* опрацьовувати числові, кількісні дані;
* обробляти символьну, текстову інформацію.

**Інструментарій для опрацювання текстової інформації**

+ тип даних string - Turbo Pascal, Object Pascal, Vizual Basіc;

+ клас string з відповідними методами опрацювання - C ++;

- **НЕ** має спеціального типу даних - мова C:

* містить заголовковий файл string.h з функціями, призначеними для роботи з інформацією текстово-символьного характеру.

**Стрічка як масив символів**

**Стрічка (стрічка символів)** - послідовність символів, яка у середовищі C розглядається, як масив даних типу char.

**Нуль-термінатор (нуль-символ) ' < 0'** - ознака кінця стрічки; останнім елементом у масиві.

**Константа стрічкового типу (стрічковий літерал)** - обмежена лапками послідовності символів:

“Programming language C”, “Ukraine”, “\tKyiv”, “\t”, “etc”.

* У стрічкових літералах нуль-термінатор дописується автоматично.

**Оголошення стрічок**

**Статичний масив символів -** оголошувати так само, як масив із елементами будь-якого відомого компіляторові типу:

**char ім'я \_ стрічки [довжина \_ рядка \_ символів];**

* вказується тип елементів, ім’я, у квадратних дужках його розмір;
* розмір є абсолютною чи поіменованою додатною цілочисельною константою;
* необхідно врахувати, що останнім елементом масиву символів є нуль-символ.

**Наприклад:**

char text [81];

char strichka [BUFSIZ];

const int size = 128;

char str [size];

* text призначений для зберігання не більше 80 символів;
* strichka може містити включено з нуль-символом BUFSIZ елементів.

BUFSIZ - це ідентифікатор означної у середовищі MSVS цілочисельної константи, яка становить 512.

* str може містити не більше 127 символів;
* кожен символ, як елемент масиву, займає один байт;

**Доступ до символів стрічки** здійснюється як до елемента масиву через:

* операцію індексації: text [i];
* непряме звернення за операцією розадресації: \* (text + i);
* індекс першого символу у стрічці дорівнює 0.

**Динамічне виділення пам'яті** оголошується динамічний масив символів

**char \* str\_d = new char [81];**

в якому також слід резервувати місце під символ завершення стрічки '\0' .

**Оголошення стрічка з ініціалізації**

**Перший спосіб**

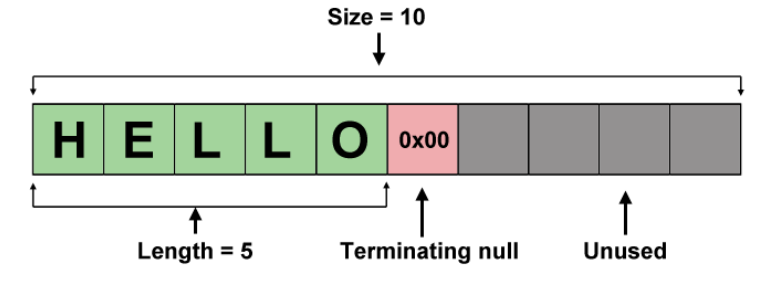
* початкове задання елементів масиву, як перелік значень, взятих у фігурні дужки

та відокремлених одне від одного комою:

char s1 [10] = { 'H' , 'E' , 'L' , 'L' , 'O' };

* під масив s1 виділяється 10 байтів пам'яті, з яких інформацією заповнено лише 5

перших комірок і символ '\0' ;

**+ МОЖНА** явно додавати нуль-термінатор:

char s2 [10] = { 'H' , 'E' , 'L' , 'L' , 'O' , '\0' };

- **НЕ МОЖНА** забувати про місце під нуль-термінатор:

char s3 [10] = { 'H' , 'E' , 'L' , 'L' , 'O' , ',' , ' ' , 'М' , 'О' , 'М'};

- **НЕ МОЖНА**, щоб кількість символів при ініціалізації була більшою, ніж

заданий розмір:

char s4 [9] = { 'H' , 'E' , 'L' , 'L' , 'O' , ',' , ' ' , 'М' , 'О' , 'М'};

+ **МОЖНА** не задавати розмірності, але потрібно **явно** прописувати символ

завершення стрічки:

char s5 [] = { 'H' , 'E' , 'L' , 'L' , 'O' , '\0' };

**Другий спосіб**

► ініціалізація стрічок через стрічкові літерали:

char str1 [10] = “Hello”;

char str2 [] = “mom” ;

► нуль-термінатор додається автоматично;

+ **МОЖНА** оголосити вказівник на даному символічному типі, який запишеться

адреса розташування літератури в ОП:

const char \* str3 = “Hello” ;

► стрічкові літератури є константами, тому вказівник str3 можна використовувати для

доступу до стрічки, але змінити її - **НЕ** можна.

**Введення та виведення стрічок**

**Особливості введення та введення стрічок:**

► символи пробілу, таблиці чи "вводу" є розділювачі при введеному числі

інформації;

► для стрічкових даних такі символи можуть бути змістовними;

► при введені-виведені стрічковій чи посимвольної інформації використовують:

1. інструкція вводу-виводу cin>>, cout << ;

2. функція стрічкового вводу та виводу, оголошені у заголовкових файлах

stdіo.h, conіo.h, тощо;

**Небуферизований ввід-вивід**

1. функція для введення символів (файл conіo.h)

int \_getche ( void ) та int \_getche ( void )

зчитують символ безпосередньо з консолі (клавіатури), без використання буфера;

* + при \_getche () - уведений символ відображається на екрані;
  + при \_getch () - уведений символ НЕ відображається на екрані;

1. функція для виведення символів без буферизації

int \_putch ( int c)

повертає символ є або у випадку неуспіх EOF, ( EOF = -1).

**Буферизований ввід-вивід**

1. функція потокового вводу символів (файл stdіo.h)

int getchar ( void )

читає символ з буферизованого стандартного потоку вводу - stdіn;

1. функція потокового (буферизованого) виводу символів

int putchar ( char )

записує символ у стандартному потік виводу stdout і повертає код записаного

символу.

Для адекватного **посимвольного** введення доречніше використовувати функції

небуферизованого вводу / виводу символів.

**Пострічкове введення даних**

1. об'єкт cout для виведення стрічки:

cout << стрічка;

1. функція виведення стрічки (файл stdіo.h):

int puts ( char \*);

* виводить стрічку (сталу чи зміну) на екран монітора та перевести курсор на наступний рядок;
* повернути невід'ємне значення (0), якщо не виникло проблем із виведенням, або EOF (-1) - у протилежному випадку;

**Пострічкове введення даних**

1. об'єкт cin для введення стрічки: cin>>стрічка;
   * якщо стрічка **НЕ містить** розділювачів, то **УСЕ**, що введено до ENTER,   
     буде з буфера перенесено в область, яку виділили під стрічку при   
     оголошенні;
   * якщо стрічка **містить** розділювачі (пробіли, табуляція, ввід), то виділена область пам'яті заповниться **лише** символами до розділювача, решта - залишаться у буфері;
   * cin зручно використовувати, коли потрібно розробити стрічки на слова.
2. функція буферизованого стрічкового введення (файл stdіo.h): char\*gets\_s (char \*);
   * повертає введену стрічку, яка заповнює масив символів, встановленого при оголошенні розміру;
   * замінює символ ' \ n' на символі ' \ 0' , викинувши його з буфера;
   * пробіли та табуляція сприймаються як значимі символи;
   * якщо розмір оголошеної стрічки є недостатність для запису введеної інформації, то виникає помилка етапу виконання.
3. функція файлового введення стрічки (файл stdіo.h):

char \* f gets\_s ( char \* \_Buf, int \_MaxCount, FILE \* \_file) ;

* + записує у \_Buf стрічку, яку читає з файлу \_file до ознаки кінця стрічки;
  + довжина зчитаної стрічки не перевищує вказане число символів \_MaxCount;
  + не замінює символ ' \ n' на символ ' \ 0' , а дописує нуль-термінатор вкінці стрічки;
  + для введення з консолі імені пристрою зчитування є стандартний потік вводу stdin .

1. методи, визначені для об'єкта cin : get () і getline ()

cin.get ( char & c) - повертає окремий введений символ

cin.get ( void ) - повертає окремий введений символ

cin.get ( char \* s, int n) - повертає стрічку максимальної довжини n з урахуванням

місця під нуль-термінатор

cin.get ( char \* s, int n, char delim) - повертає стрічку або довжини n або до

символу завершення вводу delim

* + якщо символ завершення вводу не вказаний, то це ' \ n' ;
  + символ завершення з буфером **НЕ** вилучається і може приєднатися до

наступної стрічки;

* + доречно викинути цей символ ( n символів) з буферу методом

cin.ignore ( int n, char sumvol);

cin.getline ( char \* s, int n) - повертає стрічку максимальної довжини n

cin.getline ( char \* s, int n, char delim) - повертає стрічку або довжини n або до

символу завершення вводу delim

* + адресу початку розташування стрічки в пам'яті;
  + якщо символ завершення введення вводу не вказаний, то це ' \ n' ;
  + символ завершення ' \ n' з буфера вилучається, але інші символи потрібно

випробуватихувати самостійно.

**Бібліотечні функції опрацювання стрічок** (лекція 6)

**Функції з бібліотеки string.h**

* Заголовковий файл string.h або cstring містить оголошення функції опрацювання стрічок;
* При використанні цих функцій можуть виникати застереження щодо неповного використання функцій, тому доречно використовувати директиву процесора:

#pragma warning (disable:4996);

**Довжина стрічки**

**Функція**

size\_t strlen(const char\*)

визначає кількість символів у стрічці без урахування нуль-термінатора.

* Не слід плутати поняття розміру виділеної пам'яті під стрічку з довжиною заповненої символами стрічки .

**Копіювання стрічок**

**Особливості копіювання стрічок**

* **НЕ можна** просто операцією присвоєння надати одній стрічці вміст іншої.
* Можна у вказівник записати адресу оригінальної стрічки,але це не буде копія.
* Розмір копії повинен бути достатнім для розміщення оригіналу.
* Скопійована стрічка повинна завершувати нуль-символом.

**Функція**

char\* strcpy(char\* d, const char\*s)

повертає вказівник на початок скопійованої стрічки.

* Символ нуль-термінатор також записується (копіюється) у результуючу стрічку.

**Функція**

char\* strcpy(char\* d, const char\*s)

копіює задане число символів max\_len з другої стрічки s у першу d

* Повертається вказівник на початок першої стрічки d.
* Якщо довжина стрічки s є меншою за кількістю копійованих символів max\_len,то у стрічку d після змістовних символів дописуються нуль-символи до довжини max\_len.

**Важливо!**

Функції strcpy i strncpy повертають результат і через ім'я функції,і через її перший параметр,що дозволяє використовувати їх як аргумент інших функцій.

**Конкатенація (об'єднання) стрічок**

**Функція**

char\* strcat(char\* d, const char\* s)

дописує копію другої стрічки s в кінці першої стрічки d.

* Нова перша стрічка стає об'єднанням двох стрічок-аргументів.
* Нуль-термінатор дописується після другої стрічки.
* Друга стрічка не змінюється.
* Повертається вказівник на початок першої стрічки d.

**Важливо!**

Розмір пам'яті, виділених під першу стрічку, повинен бути достатнім, для запису у неї потрібної інформації.

**Функція**

char\* strncat(char\*d, const char\*s, size\_t max\_len)

дописує задане число символів max\_len другої стрічки s в кінці другої стрічки d.

* Повертає вказівник на початок першої стрічки d.
* Якщо довжина стрічки s є меншою за кількістю max\_len, то у стрічку d після змістовних символів дописуються нуль-символи до довжини max\_len.
* Під перший аргумент потрібно виділити не менше max\_len байт.

**Важливо!**

Функції об'єднання стрічок можуть бути аргументами інших функцій.

**Порівняння стрічок**

**Функція**

char\* strcmp(const char\*s1, const char\*s2)

порівнює дві стрічки s1 і s2.

* Повертається 0,якщо обидві стрічки-аргументи однакові.
* Якщо стрічки не співпадають, то повертається ненульове значення, яке відповідає різниці кодів перших неоднакових символів:
  + менше 0-якщо s1 менше s2;
  + більше 0-якщо s1 більше s2;
* Порівнюються стрічки, а не масиви символів, тому аргументами можуть бути стрічки, які зберігаються у масивах різної розмірності.

**Важливо!**

Функцію strcmp використовують для порівняння стрічок, але **НЕ** символів

* Використовуйте strcmp() для порівняння стрічок:

if (strcmp(word, ”quit”) == 0)

puts(“Bye!”);

* Використовуйте операції відношення для порівняння символів:

if (ch == 'q')

puts(“Bye!”);

**Функція**

char\* strncmp(const char\*s1, const char\*s2, size\_t max\_count)

порівнює перших max\_count символів стрічок s1 i s2:

**Функція**

char\* stricmp(const char\*s1, const char\*s2)

порівнює дві стрічки s1 i s2 без урахування регістрі.

**Функція**

char\* strincmp(const char\*s1,const char\*s2, size\_t max\_count)

порівнює перших max\_count символів стрічок s1 i s2 без урахування регістру.

**Пошук символів у стрічці**

**Функції**

char\* strchr(const char\*s, int c);

char\* trrchrcat(const char\*s, int c);

шукають перше і останнє входження символу **c** у стрічку **s**, відповідно.

* Якщо символ буде знайдено, то повертається підстрічка, що починається із шуканого символу;
* Якщо символ відсутній-то повертається NULL.

**Пошук підстрічки у стрічці**

**Функція**

char\* strstr(const char\*s1, char s2);

визначає входження чи не входження стрічки s2 у стрічку s1.

* Якщо підстрічка s2  **повністю** входить у стрічку s1,то повертається вказівник на ту частину стрічки s1,що містить стрічку s2;
* У протилежному випадку-повертається NULL.

**Виділення лексем у стрічці**

**Функція**

char\* strtok(char\*s1, const char\* s2);

відокремлює зі стрічки s1 її частини (лексеми), що обмежені наперед вказаними розділювачами, які записані у стрічку s2.

* Якщо лексему виокремлено, то функція повертає вказівник на початок першої виділеної в s1 лексеми.
* Відразу після завершеної лексеми у стрічку s1 вноситься '/0'.
* Наступні виклики функції здійснюються з першим аргументом NULL і будуть повертати вказівники на наступні лексеми в s1.
* Кожного разу у стрічку s1 замість розділювачів будуть вноситися нуль-термінатори.
* Якщо вже всі лексеми виокремлені, то функція повертає значення NULL.

**Функція з бібліотеки ctype.h**

Заголовковий файл ctype.h містить функції, які встановлюють приналежність символу до відповідної групи:

* літера;
* цифра;
* буквенно-цифровий символ;
* символ пунктуації;
* керуючий символ і т д.

**Встановлення характеру символу**

|  |  |
| --- | --- |
| **Функція** | **Повертає true, якщо аргумент** |
| isalnum() | Літера або цифра |
| isalpha() | Літера |
| iscntrl() | Керуючий символ |
| isdigit() | Цифра |
| isgraph() | Символ, що відображається на екрані |
| islower() | Маленька літера |
| isprint() | Символ, що не відображається на екрані |
| ispunct() | Символ пунктуації |
| isspace() | Символ пробілу, табуляції |
| isupper() | Велика літера |

**Структури** (лекція 7)

**Потреба у використанні структур**

**Типи даних:**

* стандартні  **прості** (скалярні) типи даних: цілі, дійсні, символьні, логічні;
* впорядковані сукупності однотипних даних - **масиви:** статичні, динамічні, одновимірні, багатовимірні;
* інші складені (агреговані) типи даних, які користувач оголошує за певним синтаксичним правилом - **типи користувача**:
  + у мові Pascal і похідних - записи;
  + у мові C і похідних - структури;
  + в об'єктноорієнтованих мовах - класи.

**Прості типи даних** - вказують на одну характеристику чи властивістю даного.

**Як об'єднати в одне ціле різнотипні дані???**

1. Точка в просторі
   * назва точки;
   * координати (абсциса, ордината, апліката);
2. Дата:
   * рік;
   * місяць;
   * день;
3. Дані про працівника, пацієнта, клієнта:
   * особові дані;
   * адреса тощо.

**Що таке структура у мові С/С++?**

**Структура -** складений (агрегований) тип даного, який об'єднує дані різних типів (як простих, так і складених) в один тип.

* типи змінних, які об'єднуються повинні бути відомі компіляторові на момент оголошення структури;
* зміні об'єднуються в одну область пам'яті, яка має для простоти одне ім'я.

**Поля структури -** змінні, що об'єднанні у структуру.

Термінові ”структури”відповідають **два змістово різні поняття:**

**Структурна змінна (змінна типу структура)** - місце в пам'яті, де міститься інформація.

**Шаблон (pattern) структури** - правила формування структурної змінної, що використовуються компілятором для виділення місця у пам'яті під цю змінну та організації доступу до її полів.

**Синтаксис оголошення структур**

**Як описати змінну типу структуру???**

1. задати шаблон структури (оголосити тип “структура”);
2. оголосити змінну типу структура.

**Задання шаблону структури** - задання типу даного з відповідною організацією і з описом полів, які характеризують різні властивості змінних означуваного типу.

struct ім'я\_типу\_структура

{

тип\_поля\_1 ім'я\_поля\_1;

тип\_поля\_2 ім'я\_поля\_2, ім'я\_поля\_3;

тип\_поля\_N ім'я\_поля\_M;

};

**Оголошення типу “структура”**

**Iм'я\_типу\_структура -** це надалі ідентифікатор типу даних, що будуть оголошуватися.

* кожен шаблон структури має власне ім'я, щоби компілятор міг розрізняти різні шаблони;
* ім'я\_типу\_структурає **унікальним** в межах області оголошення.

**Список полів структури -** перелік властивостей (полів) структури.

* кожне ім'я в списку полів є унікальним;
* типи різних полів можуть співпадати;
* поля однакового типу можна оголошувати аналогічно до опису змінних однакового типу;
* якщо є декілька оголошених типів “структура”, то імена полів різних структур можуть співпадати;
* полем структури **МОЖЕ** бути:
  + масив;
  + раніше оголошена структура;
  + вказівник на тип, в тому числі на оголошуваний тип ”структура”;
* Полем структури **НЕ МОЖЕ** бути оголошувана структура.

**Приклади оголошення типу “структура”**

**Завдання точки у тривимірному просторі:**

|  |  |
| --- | --- |
| struct Point\_3D  {  double x; /\*абсциса\*/  double y; /\*ордината \*/  double z; /\*імпліката\*/  }; | або  struct Point\_3D  {  double x,y,z;  }; |

**Задання інформації про книгу:**

struct Book

{ char author[40]; /\*Автор \*/

char title[80]; /\*Назва \*/

int year; /\*Рік видання\*/

float price; /\*Ціна\*/

};

**Важливо!**

При заданні шаблону структури пам'яті **НЕ** відбувається.

**Область видимості шаблону структури**

* Якщо опис шаблону здійснено у блоці { } - **локальний шаблон**, який є видимий лише в межах цього блоку.
* Якщо опис шаблону є ппл за межами блоків, то шаблон є видимим в усіх функціях нижче точки опису до кінця файлу.

**Безіменний шаблон** - це шаблон, який не має ім'я\_типу\_структура.

* Після закриваючої фігурної дужки оголошення шаблону **повинен бути** список змінних

struct

{

список полів структури;

}список змінних безіменного типу структури;

**Приклад оголошення безіменної структури**

**Завдання інформації про книгу ( безіменний шаблон ):**

struct

{

char author[40];

char title[80];

int year;

float price;

} book1, book2;

**Важливо!**

При оголошенні безіменної структури і заданні змінних такого типу відбувається виділення пам'яті під ці зміни.

**Безпосереднє оголошення змінних**

struct ім'я\_типу\_структура

{

список полів структура;

}список зміних;

**Оголошення зміних раніше створеного типу ”структура”**

* у середовищі C

struct ім'я\_типу\_структура список змінних;

* у середовищі C++

ім'я\_типу\_структура список змінних;

**Задання інформації про особу:**

struct Person

{char \*FirstName; /\*Ім'я \*/

char \*LastName; /\*Прізвище \*/

int BirthYear; /\*Рік народження \*/

} he, she;

struct Person he1, she1; /\*у середовищі C \*/

Person he2, she2; /\*у середовищі C++ \*/

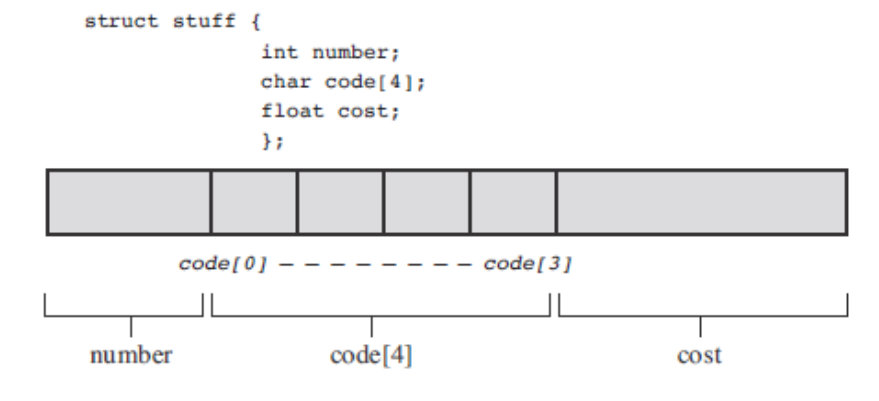
**Важливо!**

Змінні типу структура можуть бути локальні та глобальні.

* Якщо змінні **глобальні**, то початкове значення полів є нульовим (нуль відповідного типу).
* Якщо змінні **локальні**, то вміст полів невизначений, сміттєвий.

**Виділення пам'яті під змінну типу “структура”**

* При оголошенні змінної типу структура виділяється пам'ять у відповідній області залежно від області видимості змінної.
* Розмір, який займатиме змінна - це сума розмірів усіх її полів, які вирівняні на межу найбільшого розміру поля стандартного типу.



**Ініціалізація структур**

Змінній типу “структура” **можна** надавати значення при оголошенні, тобто ініціалізувати.

* Початкові значення кожного поля структури задаються через кому у фігурних дужках.
* Потрібно враховувати заданий порядок полів структури.
* Слід дотримуватися відповідності типів полів та типів їх значень.

**Доступ до полів структури**

Коли змінна типу “структура” оголошена, то доступ до її полів здійснюється через операцію крапка (.)

ім'я\_змінної. ім'я\_поля

Така конструкція **мо**же:

* входити у вираз, якщо поле ініціалізоване;
* бути аргументом функції;
* бути лівою частиною оператора присвоєння, якщо це поле не оголошено константним.

**Важливо!**

Над кожним з полів змінна типу “структура” можуть виконуватися операції, які визначенні для даних відповідного типу.

**Операція непрямого доступу до полів структури**

Якщо оголошено вказівник на змінну типу “структура”, тоді доступ до відповідного поля здійснюється:

* поєднанням операції крапка з операцією розадресації ()

(\*вказівник\_наструктуру) . ім'я\_поля\_структури

* символьним зображенням операції стрілка вліво (->)

вказівник \_наструктуру->ім'я\_поля\_структури

**Важливо!**

Операція взяття поля структури (.) має вищий пріоритет від операції розадресації (\*), тому операція розадресації береться у дужки.

Над структурами **можна** виконувати операції:

* визначення розміру структури;
* встановлення адреси структури;
* присвоєння одній змінній типу “структура” значення іншої:безпосередньо

змінна1 = змнна2;

чи за операцією розадресації за вказівником;

* доступ до поля структури безпосередньо (.) чи за адресою (->);
* виконувати над кожним полем структури ті операції, що дозволенні відповідно до типу поля.

Над структурами **НЕ можна** виконувати, зокрема:

* арифметичних операцій;
* операцій порівняння чи логічних операцій об'єднання;
* структура не може входити у вираз, що містить більше однієї складової, або таких, що формує умову.

**Масиви структур**

**Оголошення статичного масиву структур**

Синтаксис **оголошення статичного масиву структур** аналогічній оголошенню масивів будь-якого відомого компіляторів типу даних.

* Спочатку оголошуємо поіменовану структуру:

struct ім'я\_структури

{

список полів структури

};

* Далі оголошуємо масив одним із можливих варіантів:

struct ім'я\_структури ім'я\_масиву [розмір];

struct ім'я\_структури ім'я\_масиву [розмір] = {значення};

struct ім'я\_структури ім'я\_масиву [ ] = {значення};

struct ім'я\_структури ім'я\_масиву [розмір1] [розмір2];

**Доступ до поля елемента масиву структур**

відбувається за правилом:

ім'я\_масиву [індекс] . ім'я\_поля

або

(ім'я\_масиву+індекс)-> ім'я\_поля

**Ініціалізація масиву структур**

* Кількість елементів у списку значень не повинна перевищувати задану розмірність.
* Список значень береться у фігурні дужки, елементи відокремлюються комами.
* оскільки кожен елемент масиву структур є структурною змінною, то кожен елемент списку також береться у фігурні дужки.
* якщо полем структури є статичний масив, то його елемент теж береться у дужки.

Над масивами структур **можна** виконувати ті ж самі дії, що й над масивами стандартних типів даних:

* визначити розмір виділеної пам'яті;
* взяття окремого елемента масиву за операцію індексації чи розадресації;
* встановлення адреси масиву;

**Структури (частина 2)**

(лекція 8)

**Вкладені структури**

**Структури дозволяють вкладеність однієї структури в іншу**

* Полями структур можуть бути **користувацькі типи,** в тому числі й попередньо оголошені структури.
* Доступ до полів вкладених структур здійснюється при послідовному застосуванні операції крапка чи стрілка.
* Ініціалізація вкладених структур відбувається подібно до звичайних не вкладених структур.
* Теоретично обмеження на глибину вкладеності немає.

**Важливо!**

* Структура **НЕ** може бути своїм власним полем, тобто **НЕ** може вкладатися у себе.
* **Проте** структура може містити поле - вказівник на оголошену структуру.

**Структури, як аргумент функцій**

Змінну типу “структура” **МОЖНА** передавати як аргумент у функцію.

**Особливості передачі змінної типу “структура” у функцію:**

1. за значенням
   * значення змінної не змінюється
   * доречно використовувати, коли структура займає невеликий обсяг пам'яті
2. за передаванням адреси або вказівника
   * дозволяє повертати змінене значення змінної
   * доречно використовувати, коли структура займає великий обсяг пам'яті
   * доступ до відповідного поля здійснюється операцією ->
3. за посиланням (&)
   * дозволяє повертати змінене значення змінної
   * якщо змінювати вміст **фактичного параметра** у тілі функції непотрібно, то відповідний **формальний** параметр супроводжують модифікатором const
   * економія пам'яті, за рахунок того, що не створюється копія структурної змінної
4. повернення результату через ім'я функції
   * оголошений структурний тип **МОЖЕ** виступати типом результату функції.

**Функція як поля структур**

Структур може містити не лише змінні, але й **функції**

**Такі функції (методи структури):**

* можуть опрацьовувати як окремі поля, так і структурну змінну в цілому;
* дозволяють зменшити число параметрів, що передається чи повертається функцією;
* можуть повертати довільний тип результату;
* **оголошуються** за допомогою прототипів і тілі структури, **означуються** - за її межами через операцію розширення контексту (: :);
* можуть бути самостійною простою інструкцією, входити у вираз, бути аргументом іншої функції;
* звертання до такої функції здійснюється через операцію доступу до поля структури.

**Застосування typedef для структур**

**Ключове слово typedef** дозволяє створювати власне ім'я для типу даних

* надає символьні імена **лише типам,** але не значенням;
* інтерпретація **typedef** виконується компілятором, а не процесором;
* область видимості ідентифікатор тип залежить від місцезнаходження оператора **typedef**

1. якщо оголошення виконано у функції - область видимості **локальна** (у межах цієї функції)
2. якщо оголошення виконано поза функціями - область видимості **глобальна**

* при оголошенні **typedef** часто використовують великі літери, щоби нагадати користувачу, що ім'я типу - це насправді символьне скорочення.

**Синтаксис оголошення**

typedef оголошення \_ типу нове \_ ім'я \_ типу;

typedef unsigned long long ULL;

// оголошення нового імені ULL для типу unsigned long long

ULL b, c;

// тепер ULL можна використовувати для оголошення змінних

const int rd = 20, sc = 25;

typedef int MATRIX [rd][sc];

// оголошення нового імені MATRIX для двовимірного масиву

MATRIX a1, a2;

// змінні a1, a2 - двовимірний масив

**Задання комплексного числа**

typedef struct complex

{ double Re;

double Im; }

COMPLEX;

COMPLEX number = {1, 2};

**Можна не вказувати імені структури:**

|  |  |
| --- | --- |
| typedef struct  { double x, y;  } RECT;  RECT r1 = {3.0, 6.0};  RECT r2; | struct  { double x, y;  } r1 = {3.0, 6.0};  struct  { double x, y; } r2; |

**Шаблонні структури**

**Перезавантаження функцій**

* дозволяє при різних реалізаціях виконавчого файлу отримувати розв'язання задач для даних різного типу;
* передбачає кількаразове повторення оголошень і означень функцій, які виконують аналогічні дії над даними різних типів.

**Шаблонування функцій -** функції, у списку параметрів яких є один або декілька формальних параметрів не конкретного типу, а **узагальненого** чи абстрактного типу.

* Можуть повертати результат як конкретного, так і загального типів при умові, що у списку формальних параметрів задекларований такий узагальнений тип.
* Використовуються, коли однаковий алгоритм застосовується для даних різного типу.

**Який принцип роботи шаблонних функцій?**

* При виклику шаблонної функції компілятор виконує аналіз фактичних параметрів, а тоді генерується об'єктний код, який зберігається у відповідному місці ОП.
* При звертанні до цієї ж функції з фактичними параметрами **іншого типу** генерується **новий** код і виконується відповідний варіант функції.
* Якщо ж звертаються **повторно** до функції з параметрами таких типів, які вже опрацьовувалися, тоді використовується вже існуюча копія (екземпляр шаблону) коду функції.

**Такий підхід називають *узагальненим програмуванням.***

**Синтаксис оголошення і задання шаблонних функцій**

використовує ключові слова template i class чи typename

* прототип шаблону функії має вигляд:

template **<**class iм'я\_узагальненого\_типу>

тип\_функції ім'я\_функції(список\_формальних\_параметрів);

або

template **<**typename iм'я\_узагальненого\_типу>

тип\_функції ім'я\_функції(список\_формальних\_параметрів);

* У списку формальних параметрів замість конкретного типу чи типів зазначають узагальнений тип.
* Якщо необхідно вказати два і більше різних узагальнених типів, то у кутових дужках (< >) кожен тип прописується через кому з використанням ключового слова class чи typename.

**Синтаксис оголошення і задання шаблонних функцій**

* означення функції:

template **<**class iм'я\_узагальненого\_типу>

тип\_функції ім'я\_функції(список\_формальних\_параметрів) {тіло\_функції}

або

template **<**typename iм'я\_узагальненого\_типу>

тип\_функції ім'я\_функції(список\_формальних\_параметрів) {тіло\_функції}

**Шаблонна функція для обміну двох значень**

template **<**class AnyType>

void Swap(AnyType &a, AnyType &b)

{

AnyType temp;

temp = а;

а = b;

b = temp;

}

* I Описується шаблон функції з узагальненим типом **AnyType**.
* Жодних функцій шаблон не створює, лише дає компілятору вказівку щодо означення функції.

**Важливо!**

Шаблон функції може містити означення двох і більше узагальнених типів, при цьому кожен із них повинен бути у списку формальних параметрів.

Шаблонування можна застосовувати і для задання структур.

**Шаблонні структури** - структури із шаблонними (узагальненими) типами полів.

**Оголошення шаблонних структур**

здійснюється в області оголошення глобальних змінних.

1. Зазначається намір про встановлення шаблону через ключове слово template i список узагальнених типів полів;
2. Оголошується нове ім'я структурного типу;
3. Прописується взірець самої структури із зазначенням полів.

**Синтаксис оголошення шаблонних структур**

template **<**class A, class B>

struct T\_Struct

{

A a; B b;};

**Синтаксис оголошення змінної**

T\_Struct <int, int> dane;

* Оголошення змінної **повинно (!)** включати вказання конкретного типу поля.

**Директиви препроцесора**

(лекція 9)

**Поняття препроцесора**

**Етапи виконання програми, написаної мовою С/С++:**

1. спочатку лістинг програми опрацьовується **препроцесором** спеціальною програмою, яка завершує формування вихідного тексту:
   * замінює символьні абревіатури;
   * під'єднує інші файли за бажанням розробника;
   * дозволяє обирати, яка частина коду буде видимою для компілятора;
2. далі сформований текст опрацьовується **компілятором**, який перетворює текст програми з мови високого рівня на об'єктний (машинний) код;
   * якщо синтаксичні помилки відсутні, то компіляція завершується генеруванням об'єктного коду.

**Директиви препроцесора**

**Директиви препроцесора -** характерні інструкції, які виконуються на початку компіляції програми:

* починаються спеціальним символом #, перед яким у даному (директивному) рядку немає жодних інших символів;
* **НЕ** мають жодного завершального символу, в тому числі крапки з комою (;).

**Виділяють:**

* **директиви під'єднання файлів -** додають текст з інших текстових файлів, що містять прототипи бібліотечних і створених користувачем функцій, декларації користувацьких типів тощо;
* **директиви означення** - описують макро, що дозволяють задавати поіменованиі константи чи означені ідентифікатори;
* **директиви умовного компілювання файлів проєкту** - організовують умовну компіляцію, тобто залежно від зазначених в командному рядку параметрів дозволяють одержувати різний програмний код;
* директиви, що розширюють можливості мови програмування.

**Директива під'єднання файлів  #include**

**Директива #include -** вказує препроцесору на необхідність під'єднання до вихідного коду проєкту вміст іншого файлу - **заголовкового** (header).

* Вмітиме файлу підставляється на місце директиви #include.
* Системні заголовкові файли є в папці INCLUDE:

1. С заголовкові файли  з розширенням .h;

#include <string.h>

1. С++ заголовкові файли  без розширення:

#include <iostream>

**Синтаксис директиви #include**

**#include <назва\_файлу>     або     #include “назва\_файлу”**

* Для кожного файлу вказується своя директива.
* Директива під'єднання розташовується поза межами будь-якого виконавчого блоку чи тіла функції на початку коду відповідного файлу.
* Якщо НЕ під'єднати відповідний файл, то з'явиться повідомлення про синтаксичну помилку етапу компіляції.

**Як працює директива #include?**

Препроцесор знаходить у тексті програми директиву #include:

* створює копію вказаного файлу;
* під'єднує цю копію до коду;
* також під'єднує файл чи файли із відповідними бібліотечними функціями.

**Бібліотечні файли**

містять машинний код бібліотечних функцій.

* Зазвичай мають розширення .lib і розташовані у папці LIBRARY або LIB;
* Містять означення функцій, оголошених у відповідному заголовковому файлі.

**Важливо!**

1. Якщо назва\_файлу взята в кутові дужки < >:
   * під'єднується стандартний заголовковий (header) файл, який розташований в одній із системних папок (наприклад INCLUDE)
   * коли такого файлу у стандартних папках НЕ знайдено, то видається повідомлення про помилку.
2. Якщо назва\_файлу взята у лапки “ ”:
   * пошук починається з папок поточного проєкту, папок користувача, а тоді переходить до перегляду системних папок.
3. Якщо файл створений користувачем і може міститися у папці Header даного проєкту або в іншій несистемній папці, то його назву **ОБОВ'ЯЗКОВО** задають у лапках.

**Зауваження**

Можна під'єднувати і файли з розширенням cpp, але потрібно пам'ятати, що

* заголовкові файли (.h) містять оголошення (declaration);
* файли .ccp містять означення (denition);
* лише ОДИН файл може мати функцію main().

**Приклад розбиття проєкту на декілька файлів**

**decl\_struct.h** - заголовковий файл, в якому оголошуються структури та прототипи функцій;

**dcl.cpp** - означуються функції;

**main.cpp** - файл головної програми;

**Директива препроцесора #define**

**Директива #define** - визначає макро, що дозволяє формувати вбудовані функції, задавати поіменовані константи, чи просто оголошувати ідентифікатори:

#define ідентифікатор\_макро тіло\_макро

* Препроцесор переглядає вихідний текст і замінює кожне входження лексеми ідентифікатор\_макро лексемою тіло\_макро.
* Якщо лексема є стрічкою, символьною константою чи коментарем, то заміна **НЕ** відбувається.
* Можна формувати вкладені макро.
* **НЕ** може бути однакових ідентифікаторів макро, а от тіла макро **формально** можуть бути однакові.

**Зауваження**

Замість #define для означення **поіменованих констант** на мові С++ рекомендують

використовувати

const тип ім'я = значення;

* тип даного вказується явно;
* видимість даного можна обмежувати окремими функціями чи файлами;
* ідентифікатор const можна використовувати для складніших типів (масивів, структур тощо).

**Псевдоніми типів**

У С++ передбачено два способи встановлення псевдоніму для типу даного:

* через препроцесор;
* з використанням ключового слова typedef.

typedef ім’я\_типу ім’я\_псевдоніму;

Наприклад

#define BYTE char // препроцесор замінює BYTE на char

або

typedef char byte; // byte стає псевдонімом char

**Зауваження**

Важливо пам'ятати, що препроцесор просто підставляє лексеми.

Наприклад

typedef char\* byte\_pointer;

робить byte\_pointer псевдонімом для вказівника на char, **АЛЕ**

#define FLOAT\_POINTER float\*

FLOAT\_POINTER pa, pb;

оголошує вказівником тільки pa, тоді як pb - це просто float.

**Вбудовані функції -** це функції, код яких при компіляції підставляється (вбудовується) у виконавчий код програми.

* Збільшується   виконавчого коду програми, бо кожне звертання до такої функції спричиняє появу повної копії її коду.
* Не витрачаються часові ресурси на передавання й повернення значень через стек, як для звичайних функцій.
* Зазвичай це дуже короткі функції, які містять лише кілька простих інструкцій.

**Вбудовані функції в С/С++**

можна оголосити як макро з параметрами

#define ім'я\_функції (список аргументів) (тіло макро)

* При звертанні до вбудованих функцій такого типу компілятор **НЕ** виконує перевірку на відповідність типів.
* У тілі макро аргументи, над якими виконуються операції, доречно брати у дужки.
* Якщо тіло макро не може поміститися в одному рядку, його можна розділити на наступні рядки, використовуючи символ лівої похилої риски (\).
* Символ # перед параметром у тілі макро, вказує, що змінну потрібно трактувати як стрічку.

**Вбудовані функції в С++**

можна оголосити за допомогою ключового слова inline.

inline тип\_функції ім'я\_функції (список параметрів)

* Оголошується аналогічно до звичайних функцій.
* Компілятор сам приймає рішення про визначення даної функції як вбудованої.
* Аргументи за значеннями передаються, як у звичайних функціях.

**Директива препроцесора #undef**

**Директива препроцесора #undef -** директива відміни оголошення #define.

#define ідентифікатор\_макро тіло\_макро

...

#undef ідентифікатор\_макро

* Компілятор вважає оголошеним ідентифікатор з директиви #define до тих пір, поки не з'явиться директива #undef.
* Директива #undef дозволяє переозначувати макро різними тілами.

**Умовна компіляція**

**Умовна компіляція** - дозволяє залежно від значення чи оголошення відповідних ідентифікаторів включати в проєкт потрібні програмні складові.

* Секції умовної компіляції починаються однією з директив #if, #ifdef, #ifndef.
* Секції умовної компіляції завершуються **обов'язково** директивою #endif.
* Зазвичай використовується для уникнення **помилки повторного включення** означеної змінної, функції чи заголовкового файлу у багатофайлових проєктів.

**Директива #if**

Загальний вигляд секції умовної компіляції

#if логічний\_вираз

TRUE - секція

#endif

* Якщо логічний\_вираз має істинне (ненульове) значення, тоді компілюються інструкції TRUE  секції, а далі інструкції після директиви #endif.
* Якщо ж значення виразу є хибне (нульове), то компілюються інструкції після директиви #endif.

**Директива #else**

Умовна компіляція може мати й розгалуження:

#if логічний\_вираз

TRUE - секція

#else

FALSE  секція

#endif

* При істинності логічного виразу передбачається компілювання одних інструкцій, а при хибності - інших.

**Директива #elif**

Якщо ж розгалуження є множинним, тоді в секції директиви #else знову можна

Сформулювати умову, тобто використати директиву #elif.

#if Логічний\_вираз\_1

TRUE - секція\_1

#elif логічний\_вираз\_2

TRUE - секція\_2

.

.

.

#else

FALSE - секція

#endif

* Дозволеною є вкладеність директиви секцій #if одна в одну на довільну глибину, однак кожному #if повинна відповідати своя директива #endif.

**Директиви #ifdef та #ifndef**

встановлюють чи наявний опис деякого ідентифікатора, тобто перевіряють чи описане ім'я макро.

* Ім'я макро вважається описаним, якщо цей ідентифікатор зустрівся попередньо у директиві#define
* При цьому не обов'язково ідентифікатору було надано значення, тобто макророзширення може бути й порожнім.
* Дозволяє уникнути **повторного включення заголовкових файлів**, повторного задання імен тощо. Тому заголовковий файл доречно починати саме з такої директиви, причому ім'я макро, яке перевіряється на неоголошеність, носить фіктивно-формальний характер.

**Директиви, що розширюють можливості мови**

**#error text** - вказує, яке повідомлення text повинно з'явитися на екрані у випадку

помилки;

**#line ціле\_число** - змінює лічильник рядків програми компілятора. Якщо програма

складається з декількох файлів, то можна явно задати номери перших рядків файлів,

що під'єднуються;

**#pragma** - надає можливість за допомогою спеціальних команд керувати можливостями компілятора.

**Файловий ввід-вивід у С**

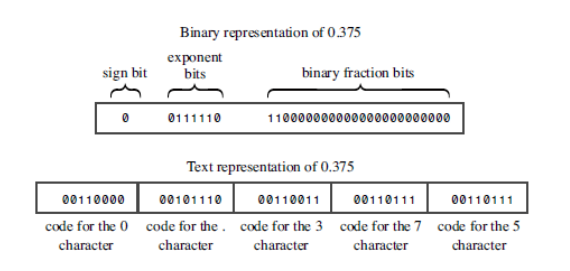
(лекція 10)

**Поняття файлу. Типи файлів**

Файл - іменована область на зовнішньому носії, яка служить для збереження даних.

**Які бувають файли?**

1. **Текстові файли (text)**
   * усі дані  навіть числа  зберігаються у   тексту;
   * можуть бути створені у довільному текстовому редакторі чи програмно;
   * рядки тексту (інформації) відокремлюються один від одного **символом завершення рядка**, який може відрізнятися для різних ОС.
2. **Двійкові файли (binary)**
   * усі дані зберігаються у вигляді внутрішнього комп'ютерного представлення;
   * створюються і редагуються лише програмно.
   * для символів двійкове представлення співпадає з текстовим  код символу;
   * для чисел - ці представлення відрізняються.



**Як працювати з файлами?**

**При роботі з файлами**

* оголошується відповідна змінна - **файлова змінна** або **логічне ім'я файлу**;
  + у мові С - це **вказівник** на дане типу структура, яка має переозначену назву FILE;
  + у мові С++ - **об'єкт** одного з класів istream або ostream;
* кожному відкритому в програмі файлу надається унікальне ціле беззнакове число - **дескриптор файлу**, за яким процес звертається до цього файлу;
* коли файл закривається, середовище забирає наданий номер.

**Відкриття файлу**

Функція fopen()

повертає інформацію потоку вводу-виводу, який прикріплено до вказаного файлу

FILE\* fopen(const char\* file\_path, const char\* file\_mode);

const char\* file\_path - шлях доступу до файлу: стрічковий літерал, поіменована

стрічкова константа або стрічкова змінна:

* може містити логічну назву диску, назви папок і назву самого файлу з розширенням;
* складові шляху відокремлюються символами: однією правою похилою (\) або двома лівими похилими (//);

**Шлях доступу до файлу**

* може бути повним;
* може містити лише “ім'я.розширення”.

**Важливо!**

* Якщо не вказано повного шляху, то за замовчуванням цей файл міститься в поточній папці (папці проєкту).
* Якщо файлу не знайдено, тоді файлова змінна набуває значення NULL і подальша робота з таким файлом є неможлива.

FILE\* fopen(const char\* file\_path, const char\* file\_mode);

const char\* file\_mode - режим, у якому відкривається файл:

* “r” - читання;
* “w” - запис;
* “а” - доповнення;

Також уточнюється характер файлу:

* “b” - двійковий файл;
* “t” - текстовий файл

**Важливо!**

За замовчуванням файл вважається текстовим.

**Комбінований режим відкриття файлу**

коли після основного символу режиму стоїть знак “+” або другий символ:

* “r”, “rb”, “r+b” - файл відкривається для читання (і запису):
  + маркер (курсор) виставляється на **початок** файлу;
  + файл **повинен вже існувати**;
  + якщо файл **НЕ існує** або не знайдено шляху до нього, то функція повертає значення NULL;
* “w”, “wb”, “w+b”  файл відкривається для запису (і читання)
  + маркер файлу виставляється на **початок** файлу;
  + якщо файл **існує**, то його вміст очищується;
  + якщо файл **НЕ існує**, тоді за вказаним шляхом створюється файл, куди будуть записуватися дані;
  + функція повертає значення NULL, якщо не існує вказаної директорії чи драйвера;
* “a”, “ab”, “a+b”  файл відкривається для доповнення (і читання):
  + маркер файлу виставляється на **кінець** файлу
  + якщо файл **існує**, то його вміст не очищується, а курсор виставляється перед кінцем файлу для дописування даних;
  + якщо файл **НЕ існує**, то за вказаним шляхом створюється файл, куди будуть записуватися дані

**Важливо!**

Якщо файл відкривається в режимі з розширенням (з символом “+”), то ввід і вивід можна виконувати в одному потоці тільки з використанням функцій означення позиції.

**Закриття файлу**

**Функція fclose()**

закриває потік вводу-виводу, який прикріплено до вказаного файлу

int fclose(FILE \*fp);

* Записує всю інформацію з буфера у файл і закриває потік fp.
* При успішному закритті файлу повертає нуль, у протилежному випадку EOF (-1).
* При нормальному завершенні програми функція fclose() автоматично викликається для кожного відкритого для читання файлу.
* Для файлів запису ця функція є обов'язковою, бо створює ознаку кінця файлу.

**Функція fcloseall()**

закриває декілька файлів

int fcloseall(void);

* Відокремлює всі відкриті потоки від зв'язаних з ними файлів.
* Завжди повертає нульове значення.

**Функція fflush()**

виштовхує буфери потоку у файл

int fflush(FILE \*fp);

* Усі дані з буфера виведення записуються у файл fp, при чому потік залишається відкритим.
* Якщо fp - NULL-вказiвник, то буфер просто очищується.

**Встановлення ознаки кінця файлу**

**Функція feof()**

перевіряє чи досягнуто ознаку кінця файлу, тобто чи встановлено індикатор EOF для даного потоку

int feof(FILE \*fp);

* повертає значення нуль, якщо кінця файлу не досягнуто;
* повертається значення EOF (-1), якщо кінця файлу досягнуто.

**Запис даних та їх читання**

**Функції читання та запису в файл**

поділяються на категорії:

* форматований ввід-вивід даних;
* блоковий ввід-вивід даних;
* посимвольний ввід-вивід;
* ввід-вивід стрічкових даних (масиву символів).

**Форматований файловий ввід-вивід даних**

**Функція fprintf()**

здійснює форматований запис у файл (повертає кількість успішно виведених даних)

int fprintf(FILE\* fp, const char\* format\_str [,output]);

**format\_str** - форматуюча стрічка;

**output** - список виводу (необов'язковий параметр).

* діє аналогічно функції консольного форматованого виводу printf();
* для двійкових файлів форматування не передбачене.

**Функція fscanf()**

здійснює форматований ввід даних із файлу (повертає кількість успішно введених даних)

int fscanf(FILE\* fp, const char\* format\_str, input\_adr);

**format\_str** - форматуюча стрічка;

**input\_adr** - список адрес вводу  одна або   відокремлених комами адрес, за якими введені дані будуть внесені в ОП;

* діє аналогічно функції консольного форматованого вводу scanf().

**Зауваження**

Файл stdio.h пов'язує три вказівники на три стандартні файли, автоматично відкриті C-програмами:

* stdin - файл стандартного вводу (клавіатура);
* stdout - файл стандартного виводу (екран);
* stderr - файл стандартних помилок (екран).

Усі ці вказівники **можна** використовувати як аргументи стандартних функцій

файлового вводу-виводу

**Ввід-вивід агрегованих типів даних**

**Функції блокового вводу-виводу**

дозволяють здійснювати однією інструкцією читання чи запис даних агрегованого

(складеного) типу у двійковий файл.

**Функції fread() i fwrite()**

повертають число успішно прочитаних даних

size\_t fread(void\* ptr, size\_t d, size\_t n, FILE\* f);

size\_t fwrite(const void\* ptr, size\_t d, size\_t n, FILE\* f);

**ptr** - адреса області пам'яті, призначеної для даних, які читаються з файлу або записуються у файл;

**d** - розмір пам'яті, яку займає одне дане блоку;

**n** - кількість даних, що будуть переміщені (кожне займає d байт);

**f** - вказівник на файл, асоційований з потоком вводу і виводу, відповідно.

**Зауваження**

Дані типу структура зручно зберігати у двійкових файлах, оскільки функції блокового вводу-виводу дозволяють одразу записати все дане у файл чи зчитати його в ОП комп'ютера.

**Файловий ввід-вивід символів**

**Функція fgetc()**

читає з потоку символ, повертає його і збільшує відповідний вказівник (якщо він є) для читання наступного символу

int fgetc(FILE\* fp); //читає з файлу

**Функція fputc()**

повертає байт, записаний у потік, або в разі помилки  EOF

int fputc(int c, FILE\* fp); //записує у файл

**Зауваження**

Якщо для функцій посимвольного введення чи виведення вибрати файлом стандартний файл вводу (stdin) чи виводу (stdout), то їх можна використовувати замість функції getchar() чи putchar(), відповідно.

**Функція fgets()**

читає   str з потоку fp, який прикріпленій до файлу

char\* fgets(char\* str, int n, FILE\* fp); //читає з файлу

**n** - кількість символів стрічки, що будуть збережені.

* Повертає вказівник на стрічку str, або NULL, у випадку помилки чи кінця файлу.
* Символи зчитуються, починаючи з поточної позиції fp, до символу нового рядка '\n'  або доки число зчитаних символів не сягне значення n-1.
* str доповнюється '\0', але символ '\n' не замінюється.

**Функція fputs()**

записує у заданий потік fp вмістиме стрічки, на яку вказує str

int fputs(char\* str, FILE\* fp); //записує у файл

* Повертає останній записаний символ, або EOF  у разі помилки.
* Якщо рядок, що вводиться, є порожнім, то повертає значення 0.
* Нуль-термінатор не записується у файл.

**Зауваження**

У текстовому файлі можливе неоднозначне відображення стрічки. У двійкових файлах  рядок відображатиметься **однозначно**.

**Позиціювання у файлі**

**Позиціювання у файлі -** встановлення курсору файлу у відповідну позицію.

**Функція fseek()**

встановлює курсор файлу в позицію, на яку вказує offset, відносно початкової позиції

origin

int fseek(FILE\* fp, long offset, int origin);

Значення origin може бути:

* 0 - точкою відліку є початок файлу;
* 1 - точкою відліку є поточна позиція у файлі;
* 2 - точкою відліку є кінець файлу.

offset вказує на скільки байт зміститься курсор у файлі відносно точки відліку: може

бути додатне (для 0, 1) чи від'ємне (для 1, 2).

**Функція rewind()**

переводить курсор на початок файлу для читання даних

void rewind(FILE\* fp);

rewind(FILE\* fp); та fseek(FILE\* fp, 0L, 0);

є однаковими, АЛЕ

* rewind() - не повертає жодного значення;
* fseek() - повертає 0 в разі успіху й не нуль у протилежному випадку.

**Зауваження**

У файлах, відкритих для читання та запису, після виклику цих двох функцій допустимими є як дії читання, так і запису.

**Файловий ввід-вивід у С++**

(лекція 11)

**Файловий ввід-вивід даних у С++**

|  |  |
| --- | --- |
| Для С  використовуються функції вводу-виводу з бібліотеки <stdio.h> | Для С++   * підтримуються функції вводу-виводу для С з бібліотеки <stdio.h> * використовується ввід-вивід із заголовкових файлів <iostrea> та <fstream> |

**Як працювати з файлами у C++?**

**При роботі з файлами**

* приєднується потік вводу-виводу (потік байт) до програми;
* цей потік зв'язується з файлом на зовнішньому носії.

**Простий файловий ввід-вивід**

1. Створити об'єкт ifstream чи ofstream  для керування потоком вводу чи виводу, відповідно;
2. Поставити цей об'єкт у відповідність одному з файлів;
3. Використовувати цей об'єкт, як cin чи cout (ввід-вивід буде здійснюватися у файл замість екрана).

**Простий файловий ввід-вивід**

**Простий файловий вивід**

ofstream fout; //створення об'єкту fout типу ofstream

fout.open(“f1.txt”); // зв'язування fout з файлом f1.txt

або

ofstream fout(“f1.txt”); // створення об'єкту та зв'язування з файлом

fout < < ...; // виведення у файл, як cout

П**ростий файловий ввід**

ifstream fin; // створення об'єкту fin типу ifstream

fin.open(“f2.txt”); // зв'язування fin з файлом f2.txt

або

ifstream fin(“f2.txt”); // створення об'єкту та зв'язування з файлом

fin > >...; // читання з файлу, як cin

**Зауваження**

* Класиifstream і ofstream використовують буферизований ввід-вивід.
* При створенні кожного об'єкту створюється свій буфер, який заповнюється даними.
* Після заповнення буферу його вмітиме записується у файл.
* Таке переміщення даних є швидше ніж побайтове.

**Закриття файлу**

* З'єднання з файлом розривається автоматично при завершенні роботи програми.
* З'єднання з файлом можна закрити явно методом close():

fout.close(); //закриття з'єднання виводу з файлом

fin.close(); //закриття з'єднання вводу з файлом

* При цьому потоки НЕ знищуються тобто об'єкти fout i fin, як і відповідні буфери існують і їх можна під'єднати до іншого файлу.
* Закриття файлу очищає буфер і оновлює файл.

**Стани потоку**

**Стан потоку -**  член даних, який містить інформацію про те чи успішною була остання операція виконана з цим потоком:

* складається з трьох окремих елементів (бітів) eofbit, badbit, failbit, кожен з яких може приймати значення 0 або 1;
* якщо усі три біти стану встановлені в 0, то **усе гаразд;**
* якщо:

eofbit = 1 досягнуто кінця файлу;

badbit = 1 потік був пошкоджений (наприклад, помилка при читанні з файлу);

failbit = 1  збій вводу-виводу (наприклад, файл недоступний або не існує).

**Методи опрацювання станів потоку**

|  |  |
| --- | --- |
| good()  eof()  fail()  rdstate()  clear()  setstate()  is\_open() | повертає значення true, якщо всі біти нульові;  повертає true, якщо eofbit = 1;  повертає true, якщо badbit = 1 або failbit = 1;  повертає поточний стан потоку;  встановлює стан потоку (за замовчуванням 0);  встановлює певні біти стану потоку (інші біти стану потоку залишаються без змін);  повертає true, якщо фал успішно відкрито. |

**Перевірка успішності відкриття файлу**

fin.open(...);

if(fin.fail()) … // файл НЕ відкрито

if(!fin.good()) … // файл НЕ відкрито

if (!fin) … // файл НЕ відкрито

if (!fin.is\_open()) … // файл НЕ відкрито

**Зауваження**

Методи is\_open() і good(), на відміну від інших, дозволяють точніше відслідкувати причину проблеми з відкриттям файлу, наприклад, невідповідність режиму відкриття.

**Відкриття декількох файлів**

* Кількість файлів, яку можна відкрити одночасно залежить від операційної системи.
* Якщо декілька файлів опрацьовуються **одночасно**, то потрібно створювати окремий потік для кожного файлу.
* Якщо декілька файлів опрацьовуються **послідовно**, то можна відкрити єдиний потік і по черзі з'єднувати його з кожним з файлів.

ifstream fin; //створення потоку для читання

fin.open(“f1.txt”); //зв'язування потоку з f1.txt

...

fin.close(); //закриття з'єднання потоку з f1.txt

fin.clear(); //скидання fin (не обов'язково)

fin.open(“f2.txt”); //зв'язування потоку з f2.txt

...

fin.close(); //закриття з'єднання потоку з f2.txt

**Режими відкриття файлу**

**Режим відкриття файлу - о**писує як сама буде опрацьовуватися файл (для читання, запису, доповнення тощо).

* задається при розв'язуванні потоку з файлом:

ifstream ім'я\_потоку(назва\_файлу, режим);

* задається за допомогою метода open():

ifstream ім'я\_потоку;

ім'я\_потоку.open(назва\_файлу, режим);

**Константи режимів файлу**

|  |  |
| --- | --- |
| **ios\_base::in**  **ios\_base::out**  **ios\_base::app**  **ios\_base::trunc**  **ios\_base::binary** | файл відкривається для читання  файл відкривається для запису  файл відкривається для дописування в кінці  вмістиме файлу очищується, якщо файлу існує  двійковий файл |

**Зауваження**

* Метод open() і конструктор ifstream мають за замовчуванням режим ios\_base::in
* Метод open() і конструктор ofstream мають за замовчуванням режим ios\_base::out

| ios\_base::trunc (“|” - бітове “АБО”);

* Для fstream режим за замовчуванням НЕ передбачений.

**Відповідність між режимами у С та С++**

|  |  |
| --- | --- |
| **Режим С**  “r”  “w”  “b”  “w”  “a”  “r+”  “w+” | **Режим С++**  ios\_base::in  ios\_base::out  ios\_base::binary  ios\_base::out|ios\_base::trunc  ios\_base::out|ios\_base::app  ios\_base::in|ios\_base::out  ios\_base::in|ios\_base::out|ios\_base::trunc |

**Двійкові файли**

**Ввід-вивід агрегованих типів даних**

Функції-члени read() i write()  дозволяють здійснювати однією інструкцією читання чи запис у двійковий файл даних агрегованого (складного) типу.

**Запис даного pl у двійковий файл f1.dat**

ofstream fout (“f1.dat”, ios\_base::out | ios\_base::app | ios\_base::binary);

fout.write( (char \*) &pl, sizeof pl);

**Читання даного pl у двійковий файл f2.dat**

ifstream fin(“f2.dat”, ios\_base::in | ios\_base::binary);

fin.read( (char \*) &pl, sizeof pl);

**Зауваження**

Функції-члени read() i write() доповнюють одна одну. Функція read() використовується для читання даних з двійкового файлу, які були записані туди функцією write().

**Позиціювання у файлі**

**Позиціювання у файлі** - передбачає можливість переміщення у довільну позицію у файлі замість послідовного переміщення по ньому.

* Який підхід найпростіше реалізувати, якщо усі записи у файлі мають однакову довжину.
* Оскільки fstream використовує буфери для проміжного зберігання даних, то насправді курсор переміщається у буфері, а НЕ у реальному файлі.

**Метод seekg()**

Переміщує курсор ВВОДУ на streamoff байтів від позиції origin (застосовується для об'єктів ifstream)

fin.seekg (streamoff, origin);

* ios\_base::beg - зміщення проводиться від початку файлу;
* ios\_base::cur  - зміщення проводиться від поточної позиції;
* ios\_base::end -.зміщення проводиться від кінця файлу.

**Метод  seekp()**

переміщує курсор ВИВОДУ (застосовується для об'єктів ofstream)

**Зауваження**

За замовчуванням зміщення відраховується від початку файлу (0).

**Методи tellg() i tellp()**

Повертають поточну позицію файлового курсора в байтах, відрахованих від початку файлу, для вхідних і вихідних потоків, відповідно.

* При створенні об'єкту fstream курсори вводу і виводу переміщуються в тандемі тому tellg() i tellp() повертають однакові значення;
* При створенні об'єкту ifstream для вхідного потоку та ofstream -  для вихідного tellg() i tellp()  можуть повертати різні значення.

**Лінійні списки** (лекція 12-13)

**Лінійний список -** це скінченна послідовність однотипних елементів (вузлів)

**Довжина списку -** це кількість елементів у лінійному списку.

* У процесі роботи програми **довжина списку МОЖЕ змінюватися.**

**Наприклад**

* Лінійний список S1 =< a1, a2, . . . , an > складається з n елементів:  
  a1, a2, . . . , an
* Лінійний список S2 =< 2, 3, 1 > складається з трьох елементів:   
  2, 3, 1;
* Лінійний список S3 =< > - порожній, не містить елементів.

**Можливі операції з лінійними списками:**

1. знаходження елементу із заданою властивістю;
2. визначення першого елементу у списку;
3. виставка нового елементу перед або після зазначеного вузла ;
4. виключення (видалення) елементу зі списку;
5. впорядкування вузлів лінійного списку.

**Зауваження**

У мовах програмування НЕМАЄ певної структури даних для представлення лінійного списку так, щоб усі зазначені операції над ним виконувалися однаково ефективно.

**Методи збереження лінійних списків -** обирають таким чином, щоб забезпечити **максимальну ефективність** і за часом виконання програми, і за обсягом необхідної пам'яті.

**Метод послідовного збереження -** елементи лінійного списку розташовуються в масиві (статичному чи динамічному) фіксованого розміру.

* Розмір масиву повинен бути відомий наперед.
* Розмір масиву обмежує максимальні розміри лінійного списку.

**Послідовне збереження лінійних списків**

struct film {

char title[45];

 int rating;

};

struct film \* movie; // оголошення вказівника на структуру

movie = new film[5]; // виділення пам'яті під масив структур

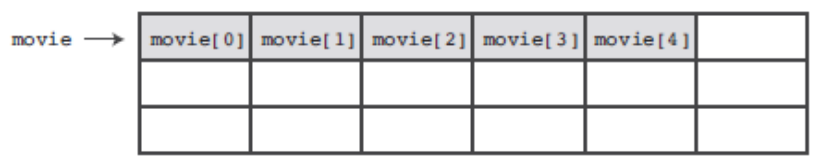


Рис.1: Приклад розташування в пам'яті масиву структур.

struct film \* movies[5]; // оголошення масиву вказівників

for (int i = 0; i < 5; i++)

            movies[i] = new film;

 //виділення пам'яті під кожну структуру окремо

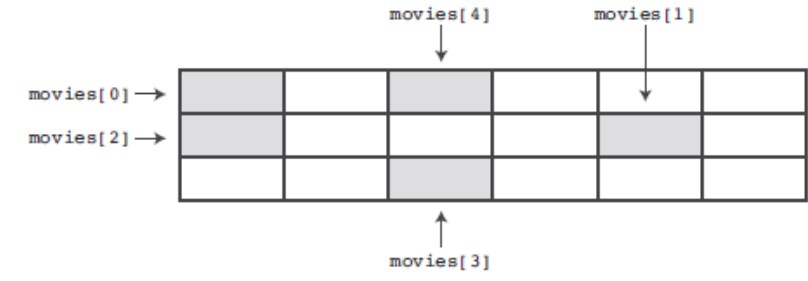


Рис. 2: Приклад розташування в пам'яті окремих структур.

**Зв'язане збереження лінійних списків**

**Метод зв'язаного збереження -** в ролі елементів збереження даних використовується зв'язані у ланцюг  структури.

* На початок цього ланцюжка (перший вузол) вказує вказівник (**head**).
* Така структура, крім елементу збереження списку, містить вказівник на сусідній (наступний) елемент (**next**).

struct film { // структура елемента збереження

 char title[45]; // елемент списку

 int rating;

 struct film \*next; // вказівник на елемент збереження

} \*head, \*ptr1, \*ptr2; // вказівник на список

 //формування першого елемента  (вершини) списку

head = new film;

 strcpy(head->title, Modern Time); head->rating = 10;

 head->next = NULL;

 //формування другого елементу списку

 ptr1 = new film;

strcpy(ptr1->title, “Midnight in Paris”); ptr1->rating = 8;

 ptr1->next = NULL;

 //під'єднання другого елементу до списку

 head->next = ptr1;

 //формула третього елементу списку

 ptr2 = new film;

 strcpy(ptr1->title, “Star Wars”); ptr2->rating = 9;

 ptr2->next = NULL; ptr1->next = ptr2;

 // head - вершина (початок)списку;ptr1 - останній елемент;

 //ptr2 - додає новий елемент в кінці списку

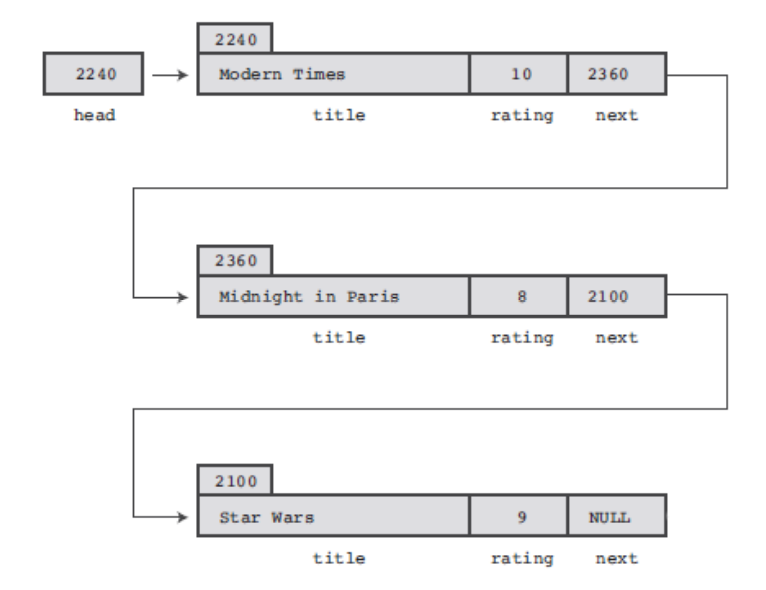


Рис.3: Приклад розташування в пам'яті зв'язного списку.

**Список типу черга (FIFO  First In First Out)** -

зв'язний список типу “перший увійшов - перший вийшов”, тобто

* елементи можуть видалятися тільки з початку списку;
* кожний новий елемент дописується в кінці списку.

**Список типу стек  (LIFO  Last In First Out)** -

зв'язний список типу “останній ввійшов - перший вийшов” , тобто

* видалення і дописування елементів може відбуватися тільки з кінця списку.

//формування першого елемента (основи) списку

 head = NULL; //список порожній

 for (int i = 0; i < 5; i++) //цикл формування списку

{

            ptr1 = new film;

            strcpy(ptr1->title, ”Film”);  ptr1->rating = i;

            ptr1->next = head; //під'єднання елемента до списку

            head = ptr1; //ptr1 знову вказує на вершину списку

}

 //head  - вершина стеку;

 //ptr1 - “нарощує” елементи списку над вершиною

**Зауваження**

В останньому елементі списку вказівник на наступний елемент (поле next) має значення NULL.

**Операції над списками**

**Порівняння послідовного та зв'язаного збереження:**

* Послідовне збереження:
* Підтримується напряму середовищем.
* Довільний доступ до елементів даних.
* Розмір визначення під час компіляції.
* Значні затрати часу при встановленні чи видаленні елементу
* Зв'язаного  збереження :
* Розмір визначається під час виконання.
* Вставлення чи видалення елементу виконують швидко.
* Довільний доступ до елементів неможливий.
* Користувач повинен забезпечити програму підтримку.

**Доступ до елементу списку**

* при **послідовному збережені** передбаченій ***довільний доступ*** до елементів, через їхні індекси у масиві ;
* при **зв'язаному збереженні** можливий лише ***послідовний доступ*** до елементів, коли потрібно починати спочатку списку і послідовно рухатися від одного вузла до іншого, поки буде досягнутий потрібний елемент.
* послідовного доступу цілком достатньо, якщо потрібно, наприклад, відобразити УСІ елементи списку ;
* послідовний доступ вкрай незручний, коли потрібно здійснити пошук конкретного елементу.

 //вивід значення і-го елементу списку

 struct Node {//структура елемента збереження

        double val; // елемент списку

        struct Node \*next ; //вказівник на елемент збереження

 } \*head, \*ptr1, \*ptr2; //head вказує на вершину списку

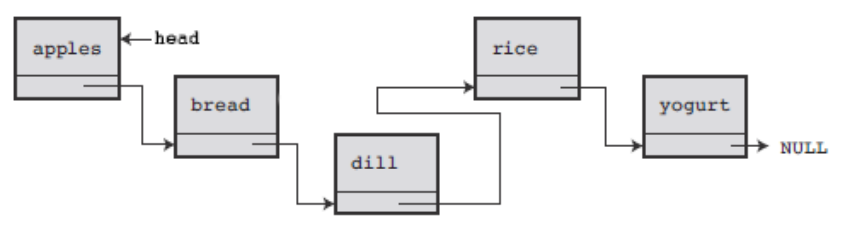
 …

 ptr1 = head; j = 1; //ptr1 також вказує на вершину списку

 while (ptr1 != NULL && j! = i) { ptr1 = ptr1->next; j++;}

 if (ptr1 == NULL) cout < <” \n  Немає вузла” < < i;

 else cout < <” \n i-елемент рівний” < < ptr1->val;

ВВ

**Вставлення нового елементу у списку**

* при **послідовному збереженні** потрібно перемішувати повні елементи масиву, щоб звільнити місце для нового елементу;

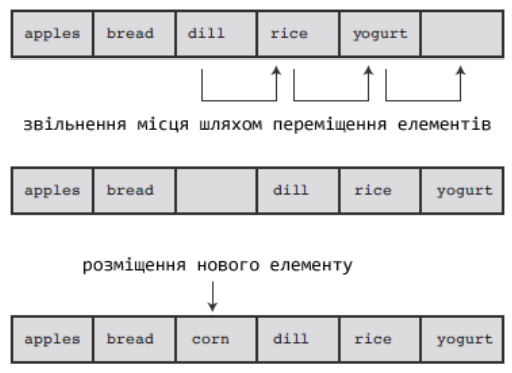


Рис. 4 :Вставлення елементу в масив

**Вставлення нового елементу у список**

* при **зв'язаному  збереженні** достатньо перевстановити вказівники.

//встановлення нового елементу після вузла ptr1

struct Node {структура елемента збереження

       double val; // елемент списку

       struct Node \*next ; //вказівник на елемент збереження

 } \*head, \*ptr1, \*ptr2; //вказівники на список

 //head вказує на вершину списку

...

ptr2 = new Node; ptr2->val = ...; //формула нового вузла

 ptr2->next = ptr1->next;

 ptr1->next= ptr2; //включення ptr2 у список

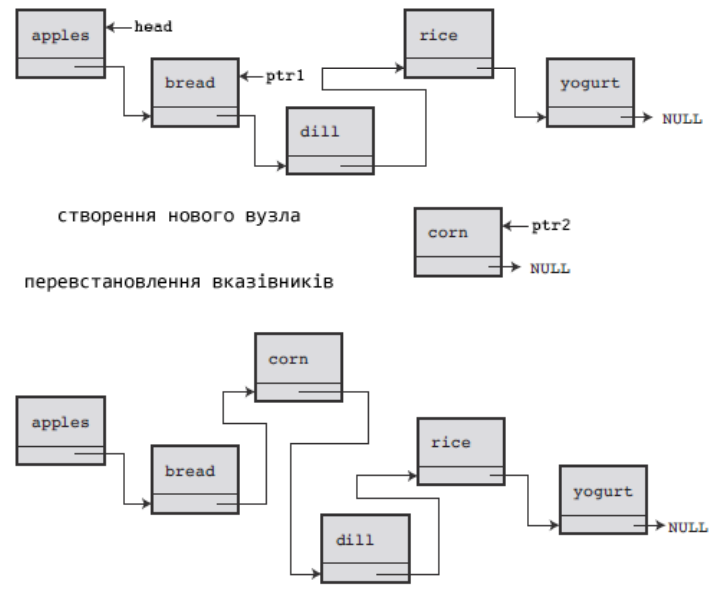


Рис. 5: Встановлення елементу в зв'язаний список

**Видалення елементу зі списку**

* при **послідовному збереженні** потребує повної зміни розташування елементів;
* при **зв'язаному збереженні** достатньо перевстановити вказівники і звільнити пам'ять, яку  займав видалений елемент.

//видалення елементу після вузла ptr1

 ptr1 = head; j = 1; //ptr1 вказує на вершину списку

//пошук і-го вузла;  ptr2 вказує на попередню вершину

 while(ptr1 != NULL && j != i)

        { ptr2 = ptr1; ptr1 = ptr1-> next;}

 if (ptr2 == NULL) cout < <” \n Немає вузла” < < i);

else {ptr2->next = ptr1-> next;

 //ptr2->next вказує на  i+1 елемент списку

 delete ptr1; } //видалення  і - го елементу

Циклічне збереження списку

**Циклічний список -** зв'язане збереження лінійного списку, коли його останній елемент вказує на перший елемент, а вказівник head - на вершину списку.

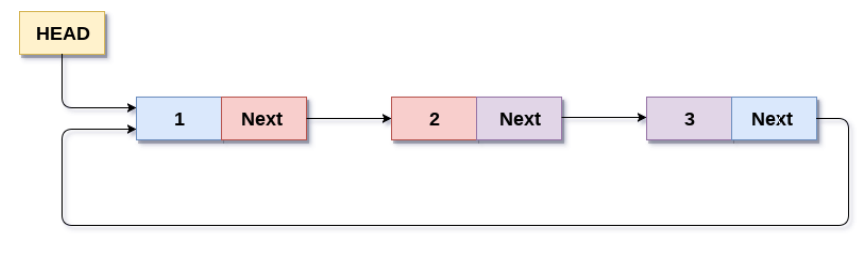


Рис. 6: Схема циклічного списку.

**Організація двозв'язних списків**

**Двозв'язний список -** зв'язане збереження лінійного списку, коли кожен елемент збереження має два вказівники (посилання на попередній і наступний елемент лінійного списку).

struct Node {//структура елементу збереження

       double data; //інформаційна частина

       struct Node \*next ; //вказівник на наступний елемент

       struct Node \*prev ; //вказівник на попередній елемент

 } \*Head, \*Tail, \*ptr; //вказівники на список

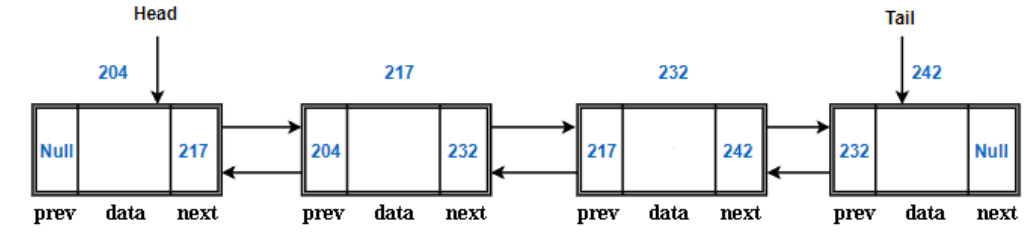


Рис. 7:Схема двозв'язного списку

//вставлення нового елементу після вузла  ptr1

 struct Node {//структура елемента збереження

        double data; // інформаційна частина

        struct Node \*next, \*prev ; // вказівники на елемент

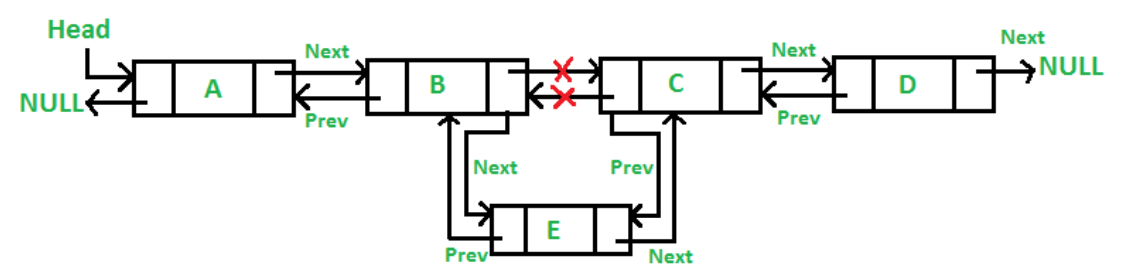
} \*Head, \*Tail, \*ptr1, \*ptr2; //вказівники на список

...

ptr2 = new Node; ptr2->data = ...; //формула нового вузла

ptr2->next = ptr1->next; (ptr1->next)->prev = ptr2;

ptr2->prev= ptr1; ptr1->next= ptr2;//включення ptr2 у список



//встановлення нового елементу після вузла ptr1

 struct Node {//структура елемента збереження

 double data; //інформаційна частина

struct Node \*next, \*prev ; //вказівники на елемент

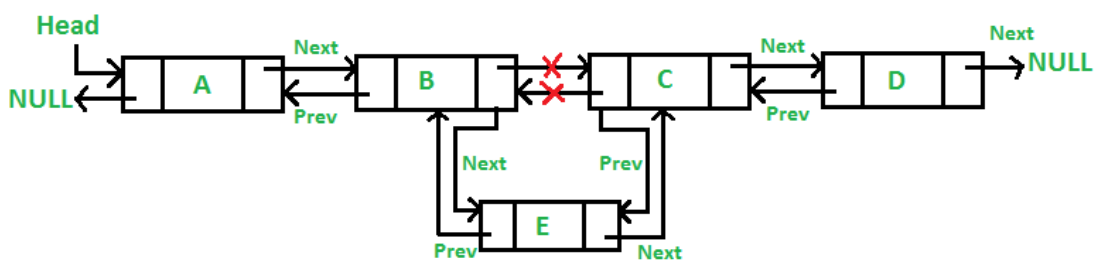
 } \*Head, \*Tail, \*ptr1, \*ptr2; //вказівники на список

 …

ptr2 =new Node; ptr2 -> data = …; //формування нового вузла

ptr2 -> next = ptr1 -> next; (ptr1 -> next) -> prev= ptr2;

ptr2 -> prev =  ptr1; ptr1 -> next = ptr2; çi ñïèñêó delete ptr2; //включення  ptr2 у список



//видалення елементу після вузла ptr1

 struct Node {//структура елемента збереження

        double data; // інформаційна частина

        struct Node \*next, \*prev ; //вказівники на елемент

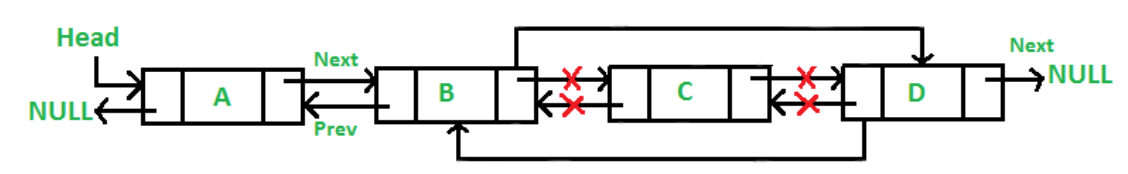
} \*Head, \*Tail, \*ptr1, \*ptr2; //вказівники на список

 ...

ptr2 = ptr1->next; ptr1->next = (ptr1->next)->next;

(ptr1->next)->prev= ptr1;//виключення  ptr2 зі списку

delete ptr2;//видалення вузла



**Стек -** це лінійний список, де операції додавання, видалення чи доступ до елемента виконується тільки через вершину списку (як стопку книг на столі, де додавання або виймання книги можливе тільки зверху).

**Черга -** це лінійний список, де елементи видаляються з початку списку, а додаються наприкінці списку (як звичайна черга в магазині).

**Двостороння черга -** це лінійний список, у якому операція додавання, видалення чи доступ до елемента можливі як спочатку, так і наприкінці списку (як послідовність книг на полиці, так що доступ до них можливий з обох кінців).

**Основи аналізу алгоритмів** (лекція 14)

**Ефективність алгоритмів**

Одну й ту ж задачу можуть розв'язувати багато алгоритмів.

**Аналіз алгоритмів**

1. Чи даний алгоритм правильно розв'язує поставлену задачу (задовольняє властивість правильності)?
2. Чи ефективний алгоритм? Скільки “часу” необхідно для його виконання?

**Обчислювальна складність алгоритму -** це приблизна кількість значимих операцій, які виконуються алгоритмом.

* Характеризує (вимірює) *відносний час* виконання алгоритму.

***Реальний час, необхідний для розв'язку задачі, непридатний для аналізу ефективності алгоритму***

* Алгоритм не стає *кращим*, якщо його перенести на *більш швидкий* комп'ютер.
* Алгоритм не стає *гіршим*, якщо його виконувати  на *більш повільному* комп'ютері.
* Фактична кількість операцій алгоритму для тих чи інших вхідних даних не дає багато інформації про ефективність алгоритму.

**Швидкість росту алгоритму -** це залежність кількості значимих операцій конкретного алгоритму від розміру вхідних даних.

* При невеликому розмірі вхідних даних алгоритм А може вимагати меншої кількості операцій, ніж алгоритм В.
* Але при зростанні обсягу вхідних даних, ситуація може змінитися на протилежну.

**Зауваження** Можна порівнювати характеристики тільки тих алгоритмів, що розв'язують одну і ту ж задачу.

**Приклад** Розглянемо два алгоритми знаходження максимального значення з чотирьох величин  a, b, c, d.

|  |  |
| --- | --- |
| Варіант 1 (послідовний перебір):  max = a;   if (b > max)        max = b;   if (c > max)        max = c;  if (d > max)      max = d;   cout < < max; | Варіант 2 (метод вибору):  if (a > b)      if (a > c)          if (a > d) cout < < a;          else cout < < d;   else        if (c > d) cout < < c;        else cout < < d;   else        if (b > c)             if (b > d) cout < < b;             else cout < < d;   else         if (c > d) cout < < c;         else cout < < d; |

**При аналізі потрібно розглядати різні можливі набори вхідних значень.**

* Якщо обмежитись лише одним вхідним набором, то він може виявитись таким, на якому алгоритм є найшвидшим або найповільнішим.

**Наприклад,** алгоритм пошуку найбільшого елементу  в списку  з n елементів:

max = mas[0];

 for (i = 1; i < n; i++)

       if (mas[i] > max)

       max = mas[i];

 cout < < max;

* виконає єдине присвоєння перед початком циклу (в тілі циклу присвоювань не буде), якщо список впорядкований по *спаданню*.
* виконає n присвоєнь (одне перед початком циклу і (n-1 в циклі), якщо  список впорядкований по *зростанню.*

**Класи вхідних даних**

**Клас вхідних даних -** це сукупність вхідних наборів, на кожному з яких алгоритм має однакову обчислювальну складність.

* Розбиття на класи дозволяє зменшити кількість варіантів вхідних даних, які необхідно розглядати.
* Для аналізу алгоритму достатньо знати лише кількість класів та обсяг роботи  на кожному з них.
* При цьому необхідно підібрати такі вхідні дані, які забезпечують **найшвидше і найповільніше** виконання алгоритму, а також оцінювати **середню ефективність** алгоритму на всіх можливих вхідних наборах.

**Приклад**

Для алгоритму пошуку максимального значення у списку з 10 фіксованих унікальних елементів розбиття на класи мого б ґрунтуватися на місцезнаходженні максимального елемента.

1. Якщо максимальний елемент знаходиться на першому місці - необхідно виконати  лише 1 присвоєння.
2. Якщо максимальний елемент знаходиться на другому місці - необхідно виконати 2 присвоєння.

…

Отже, для цього алгоритму всі вхідні набори необхідні розбити на 10 класів!(=362880) наборів у кожному з класів.

**Операції порівняння**

* Усі операції порівняння вважаються еквівалентними.
* Враховуються в алгоритмах пошуку та сортування.

**Арифметичні операції**

* Адитивні операції: додавання, Віднімання, збільшення та зменшення лічильника.
* Мультиплікативні операції: множення, ділення і залишок від ділення.

**Приклад**

При обчисленні значення многочленна

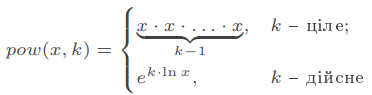
https://lh5.googleusercontent.com/dtuF1SmIMq5vg2sRwEdji5WAQ3eoQ08eZ-Mf1lVzk7sdFmgUBGjKW2TdBJpcRgDQ1vu1I7-IwV1fgW8QolBX5OgPuxOWsADyCgbFka1kfDXrxyPuliMFS5CIeRpe08dBQ-wyc3Jq

можна використати прямий метод:

                      for (double sum = 0, int i = 0; i <= n; i++)

                                    sum += a[i] \* pow(x, n-i);

Оскільки функції:



то число операції множення необхідне для реалізації алгоритму

https://lh3.googleusercontent.com/7e8s_UZWZ6giKZrmnAzw-3T0NHA5oCDhef8nJ1b3a9-E_pC_ub75SCTMnF8Y67LBpcEoBmvt92s8xGjOvVz9yccP5eh39HYv7QXCR0fnIujOupD_v3Jxcldk5HKUIau_okeT5mPK

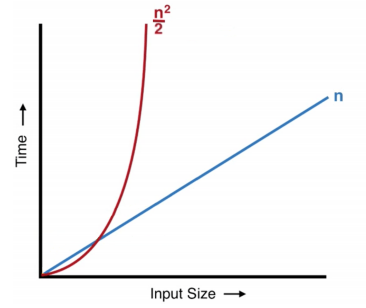
**Приклад**

При використанні схеми Горнера для обчислення значення многочленна

Pn(x) = (. . . (((a0x + a1) x + a2) x) + · · · + an−1) x + an

for (double sum = 0, int i = 0; i <= n; sum \*= x, sum += a[i++]);

необхідно лише n операцій множення.



**Середня ефективність алгоритму**

**Найкращий випадок -** це такий клас вхідних даних, на якому алгоритм виконується за мінімальний час (найменша обчислювальна складність).

* На такому наборі даних алгоритм   виконує найменше дій.
* З точки зору аналізу найкращий випадок  алгоритму є мало актуальним.

**Найгірший випадок -** це такий клас вхідних даних,  на якому алгоритм виконується за максимальний час (найбільша обчислювальна складність).

* Час роботи в найгіршому випадку дає верхню межу час роботи алгоритму незалежно від вхідних даних.
* У деяких алгоритмах найгірший випадок трапляється доволі часто.

**Середня ефективність алгоритму (середній випадок) -** це середній час виконання алгоритму (середня обчислювальна складність) на усіх можливих  наборах вхідних даних.

* Середній випадок є найскладнішим, оскільки вимагає врахування багатьох  різноманітних деталей.

*Для аналізу середнього випадку необхідно:*

1. розбити набір вхідних множин алгоритму на класи;
2. визначити ймовірність, з якою вхідний набір  належить кожному з класів;
3. підрахувати час роботи алгоритму на даних з кожного класу.

Середня ефективність алгоритму обчислюється за формулою

https://lh5.googleusercontent.com/rJ9wQTpGodH33nmsmrERIXrsgONWZpjBVDM_4A14pKyYcoRP-in1L8FrUQs6Ay1JCKkK11v8ev2R2SdpqxV0SEv8vnMuDB_ipYxvf24_GO2AN9SvRwqu_buO6qBtJTXk4-l-vbmq

n - розмір вхідних даних,

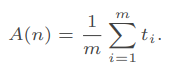
m - число класів,

p*і* - ймовірність того, що вхідні дані належать до  *і-ого класу,*

t*і -* час, який необхідний  алгоритму для обробки даних з *і-* ого класу.

* Розбити на класи і значення параметрів p*і*  та t*і* залежить від розміру вхідних даних n.

В окремих випадках можна вважати, що ймовірність попадання вхідних даних у кожен клас є однакова, тоді.



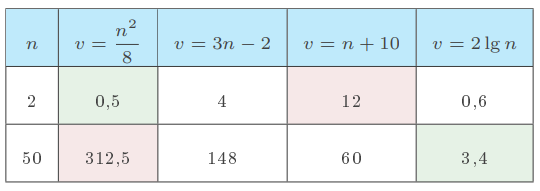
**Швидкість росту алгоритму**

**Швидкість росту алгоритму -** це швидкість зростання кількості значимих операцій алгоритму (обчислювальної складності) при зростанні розміру (об'єму) вхідних даних.

**Клас швидкості росту -** функціональна залежність, за якою наростає кількість операцій  v = f(n), де n - розмір вхідних даних.

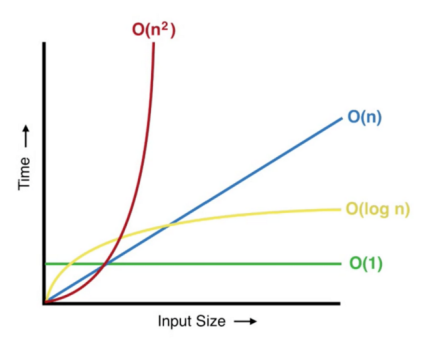
* При аналізі важливим є клас швидкості росту алгоритму, а не точна кількість значимих операцій, які він виконує.

Розглянемо чотири функції: Пор v = n^2/8, v = 3n − 2, v = n + 10, v = 2lg(n).Порівнюємо значення цих функцій при  n = 2  n = 50.



Як бачимо, функія v = n^2/8 має найбільшу швидкість росту.

* З двох алгоритмів, ефективнішим буде алгоритм  з меншою швидкістю росту, при умові, що розмір вхідних даних n є достатньо великим.



* Функції з більшою швидкістю росту при великих значеннях аргументу домінують  над функціями з меншою швидкістю росту.
* Якщо обчислювальна складність  алгоритму представляє собою суму декількох функцій, то в більшості випадків функціями з меншою швидкість росту можна  знехтувати.

*Наприклад,*  якщо алгоритм виконує

n^3 − 100n

операцій порівняння, то клас швидкості росту цього алгоритму становить  n^3.

**Порядком алгоритму (функцій) -** це домінуючий член формули, який характеризує швидкість росту обчислювальної складності алгоритму.

**Приклади оцінки складності алгоритмів**

Розглянемо алгоритм сортування списку з *N* елементів у порядку зростання.

**Алгоритм сортування Неймана (метод бульбашки)**

полягає у відштовхуванні менших значень на початок  списку в той час, як більші значення опускаються в кінець списку.

* При кожному проході по списку відбувається порівняння двох елементів.
* Якщо порядок сусідніх елементів неправильний, то вони міняються місцями.
* При кожному приході ближче  до свого місця  просовуються зразу декілька елементів, але гарантовано займає правильне положення лише  один.
* Якщо на певному проході не відбулося жодної перестановки елементів, то всі вони розміщені  в потрібному порядку.

double\* bubble(double\* mas, int n) //сортування методом бульбашки

{

  int flag = 1;

 while (flag && n > 1)

 {

 flag = 0; //обміну не було

 n- -; //зменшення кількості елементів, що переглядаються

for (int i = 0; i < n; i++)

       if (mas[i] > mas[i + 1])//ïîðiâíÿííÿ ñóñiäíiõ åëåìåíòiâ

      {

                                 //обмін сусідніх елементів списку

            double b = mas[i];

  mas[i] = mas[i + 1];

  mas[i + 1] = b;

    flag = 1; //обмін відбувся

 }

}

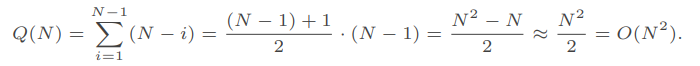
          return mas;

**Аналіз найкращого випадку**

* Найкращим набором даних буде список, відсортований по зростанню.
* При відсутності перестановок на першому проході, буде виконано  N − 1 порівняння.

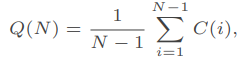
**Аналіз найгіршого випадку**

* Список відсортований у  зворотному порядку (по спаданню) буде найгіршим випадком.
* При такому варіанті список максимальним буде як число порівняння, так і число перестановок (N − 1 і порівнянь та перестановок на і-ому проході).
* Складність в найгіршому випадку (кількість порівнянь) становить

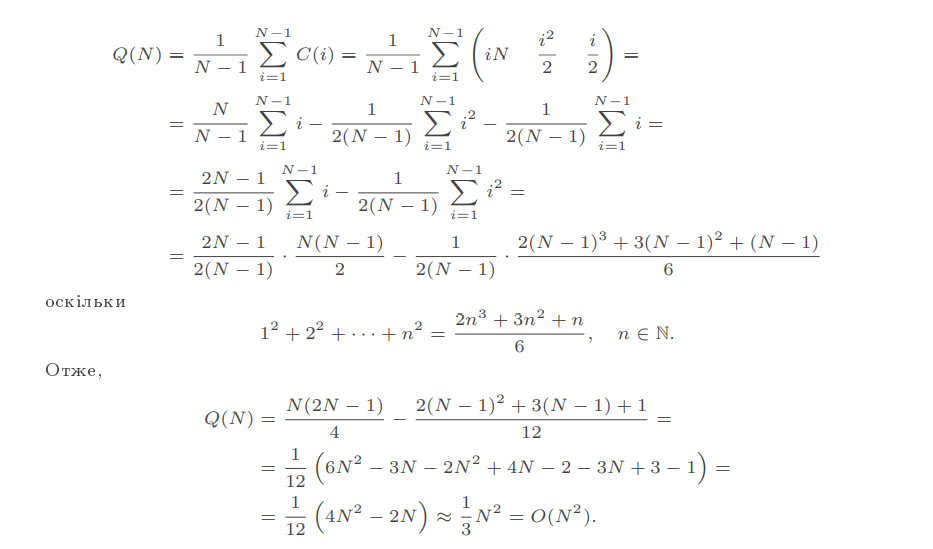
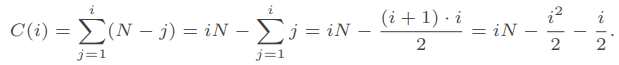


**Аналіз середнього випадку**

* У середньому випадку вважаємо, що на кожному з  N − 1 проходів рівноймовірна поява приходу без перестановки елементів, тобто алгоритм завершує свою роботу.
* Складність алгоритму в середньому випадку



де C(i) - число порівнянь, виконаних на перших і проходах:



**Зауваження**

Середній випадок часто дуже близький до найгіршого .

**Сортування і злиття списків** (лекція 15-16)

**Сортування списків**

**Задачі пошуку**

Нехай у лінійному списку потрібно знайти певний елемент.

**Послідовний пошук**

* Переглядається кожен елемент лінійного списку, поки не знайдеться потрібний.
* Єдиний можливий алгоритм пошуку, якщо список **НЕ впорядкований**
* *найкращий* випадок - шуканий елемент *перший* у списку;
* *найгірший* випадок - шуканий елемент *останній* у списку.

**Інші алгоритми пошуку**

* Для **впорядкованого** лінійного списку існують більш ефективні алгоритми пошуку.

**Сортування методом вибору**

**Сортування списку -** задача перестановки елементів списку у визначеному порядку.

**Сортування списку методом вибору елемента**

* *мінімального -* якщо сортуємо за не спаданням;
* *максимального -* якщо сортуємо за не зростанням.

**Алгоритми**

* При поточному перегляді списку вибирається найменший/найбільший елемент із тих, що ще невпорядковані.
* Після цього здійснюється обмін місцями поточного елементу і знайденого найменшого/найбільшого.
* Далі береться наступний елемент і знову здійснюється аналогічний пошук.
* Список переглядається на один раз менше, ніж кількість елементів у ньому.
* При кожному перегляді, може відбутися не більше одного обміну.
* Необхідна кількість порівнянь Q = N(N-1)/2=O(n^2)
* Не вимагає додаткової пам'яті

B =< 20, 10, 8, −5, 7 >, B’ =< >

B =< 10, 8, 20, 7 >,       B’ =< −5 >

B =< 8, 20, 10 >,           B’ =< −5, 7 >

B =< 20, 10 >,               B’ =< −5, 7, 8 >

B =< 20 >,                     B’ =< −5, 7, 8, 10 >

B =< >,                          B’ =< −5, 7, 8, 10, 20 >

**Сортування вставкою**

**Алгоритм**

базується на додаванні чергового елементу в потрібне місце вже відсортованого списку.

* 1-ий елемент вважається відсортованим списком довжини 1.
* До нього у потрібне місце додається 2-ий елемент вихідного списку.
* На наступному кроці у потрібне місце додається 3-ій елемент вихідного списку.
* Цей процес повторюється до тих пір, поки всі елементи вихідного списку не опиняться у відсортованій частині списку.
* При сортуванні вставкою потрібно N^2 порівнянь.
* Не вимагає додаткової пам'яті

B =< 20, −5, 10, 8, 7 >,   B’ =< >

B =< −5, 10, 8, 7 >,         B’ =< 20 >

B =< 10, 8, 7 >,               B’ =< −5, 20 >

B =< 8, 7 >,                     B’ =< −5, 10, 20 >

B =< 7 >,                         B’ =< −5, 8, 10, 20 >

B =< >                             B’ =< −5, 7, 8, 10, 20 >

**Сортування Неймана (метод бульбашки)**

**Алгоритм**

полягає у виштовхуванні менших значень на початок списку в той час, як більші значення опускаться в кінець списку.

* При кожному проході по списку відбувається порівняння двох елементів.
* Якщо порядок сусідніх елементів неправильний, то вони міняться місцями.
* При кожному переході ближче до самого місця просуваються зразу декілька елементів, але гарантовано займає правильне положення лише один.
* Якщо на певному проході не відбувається жодної перестановки елементів, то всі вони розміщені в потрібному порядку.
* У *найкращому випадку* (список є відсортованим по зростанню) потрібно виконати N-1 порівнянь.
* У *найгіршому випадку* (список є відсортованим по спаданню) потрібно виконати ~ 0,5 \* N^2 порівнянь.
* Примітивний і повільний алгоритм сортування

B =< 20, −5, 10, 8, 7 >

B1 =< −5, 10, 8, 7, 20 >

B2 =< −5, 8, 7, 10, 20 >

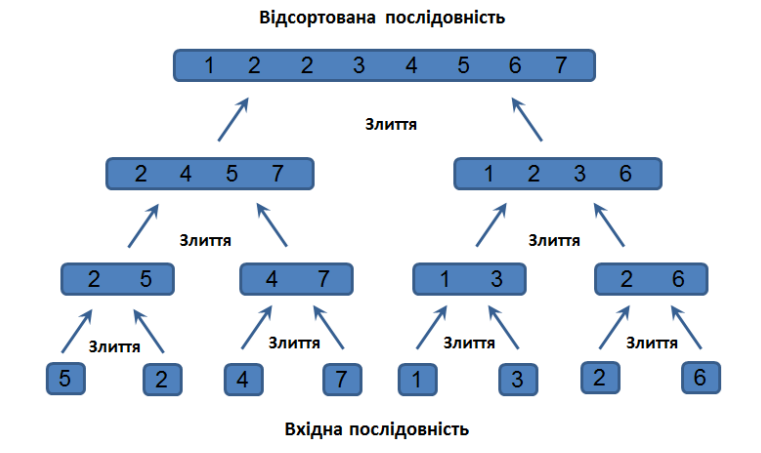
B3 =< −5, 7, 8, 10, 20 >

**Сортування шляхом злиття**

**Алгоритм**

рекурсивний алгоритм сортування, в основі якого лежить злиття двох упорядкованих ділянок масиву в дну впорядковану ділянку іншого масиву.

* Вихідний список розбивається на половинки, доки у кожному підсписку опиниться лише один елемент (список з одного елементу вже впорядкований).
* Отримані підмасиви зливаються, формуючи відсортований білий масив.
* Проходи повторюються до тих пір, поки не буде сформована одна послідовність елементів списку.
* Необхідна кількість дій Q = N \* log2 (N).
* Є дуже ефективним і часто застосовується при великих N.

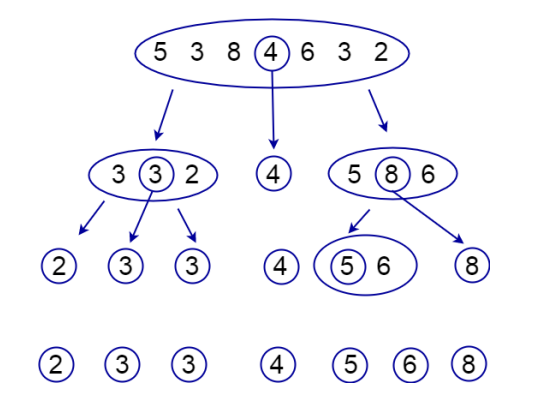


**Швидке сортування поділом**

**Алгоритм**

полягає в переставленні елементі списку таким чином, щоб його можна було розділити на дві частини і кожний елемент з першої частини був не більший за будь-який елемент з другої.

* Вибирається опорний елемент, який розділяє список на дві частини.
* Отримані підсписки частково впорядковуються: *лівий* містить елементи не більші за даний, *правий* - елементи не менші з даний.
* Далі до двох підсписків знову застосовується впорядкування швидким сортуванням.
* Необхідна кількість дій Q = 2N \* ln (N).
* Вимагає додаткової пам'яті для виконання рекурсивної функції.



**Сортування розподілом (за розрядами)**

**Алгоритм**

полягає у впорядкуванні усіх елементів списку з розрядами: спочатку молодший, потім другий, третій і так далі аж до найстаршого.

* Процедура складається з двох процесів:
  1. *Розподіл.* Формуються допоміжні підсписки B0, B1,. . . , B9, що зберігають елементи, у яких m-ті розряди однакові.
  2. *Збирання.* Допоміжні підсписки B0, B1,. . . , B9 об'єднуються у цьому ж поряду, утворюючи єдиний список.
* Кількість дій, необхідних для сортування N t - цифрових чисел Q = Nt.
* Вимагає використання додаткової пам'яті для підсписків.

B =< 07, 18, 03, 52, 04, 06, 08, 05, 13, 42, 30, 35, 26 >

***Розподіл - 1 (по останній цифрі):***

B0 =< 30 >, B1 =< >, B2 =< 52, 42 >, B3 =< 03, 13 >, B4 =< 04 >,

B5 =< 05, 35 >, B6 =< 06, 26 >, B7 =< 07 >, B8 =< 18, 08 >, B9 =< > .

***Збирання - 1:***

B =< 30, 52, 42, 03, 13, 04, 05, 35, 06, 26, 07, 18, 08 > .

***Розподіл - 1 (по другій справа цифрі):***

B0 =< 03, 04, 05, 06, 07, 08 >, B1 =< 13, 18 >, B2 =< 26 >, B3 =< 30, 35 >,

B4 =< 42 >, B5 =< 52 >, B6 =<>, B7 =<>, B8 =<>, B9 =<> .

***Збирання - 2:***

B =< 03, 04, 05, 06, 07, 08, 13, 18, 26, 30, 35, 42, 52 > .

**Задачі пошуку**

**Послідовний пошук**

передбачає послідовний перегляд усіх елементів списку в порядку їхнього розташування, поки не знайдеться  потрібний елемент.

* Якщо достовірно невідомо, чи такий елемент присутній в списку, то необхідно стежити за тим, щоб пошук не вийшов за межі списку.
* *Найгірший випадок -* шуканий елемент останній або його у списку взагалі *немає.*

**Двійковий пошук**

**Алгоритм**

полягає в порівнянні шуканого значення із центральним елементом списку.

1. Визначається центральний елемент у списку і порівнюється із шуканим.
2. Якщо вони співпадають, то елемент - знайдено.
3. У противному порядку - пошук продовжується в одній із половин списку.

* Застосовується **ЛИШЕ** для відсортованих списків.
* Необхідна кількість дії Q = log2 (N).

**Двійкове дерево пошуку**

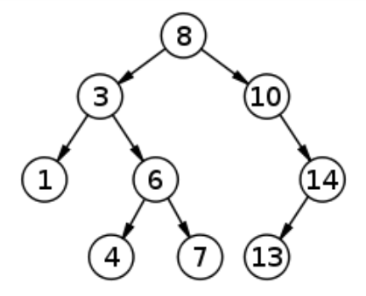
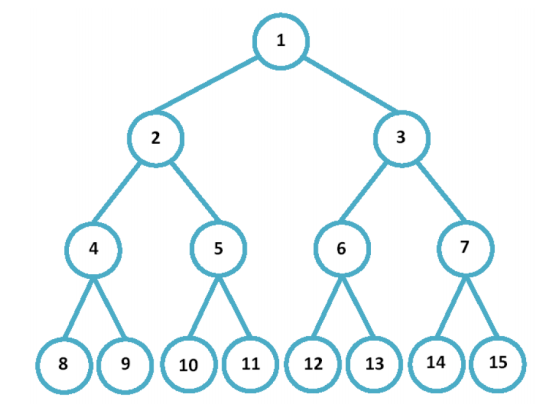
**Двійкове (бінарне) дерево -** це зв'язана структура, в якій кожен вузол (вершина) має не більше двох нащадків і лише одного батька.

* На додаток до key і супутніх даних, кожен вузол містить атрибути  left, right i p, які вказують на вузли, що відповідають лівій дитині, правій дитині і батькові, відповідно.
* Якщо дитини або батька немає, відповідний атрибут містить значення NULL.

**Корінь дерева -** найвищий вузол двійкового дерева (у нього відсутній  вузол-батько).

* Єдиний вузол у дереві.батько якого NULL.

**Повне двійкове дерево -** це двійкове дерево, в якому кожен вузол має двох нащадків.



**Властивість двійкового дерева пошуку**

Нехай х - це вузол двійкового дерева пошуку.

* Якщо у - вузол лівого піддерева, то y.key <= 6 x.key.
* Якщо у - вузол правого піддерева, то y.key => 6 x.key.

**Сортування серединним обходом дерева**

В такому алгоритмі кожна вершина виводиться між виведенням лівої та правої дитини.

* Вивести значення ключів лівого піддерева.
* Вивести ключ кореня.
* Вивести значення ключів правого піддерева.

B =< 1, 3, 4, 6, 7, 8, 10, 13, 14 >

**Пошук у двійковому дереві пошуку**

Для пошуку вузла із заданим ключем використовуються процедуру, яка отримує вказівник на корінь дерева, ключ k та повертає вказівник на вузол із ключем  k, якщо такий існує, або NULL.

* Пошук починається з коренів і рухається вниз по дереву.
* Для кожного зустрічного вузла х його ключ порівнюється з ключем k.
* Якщо  k == x.key, пошук завершується.
* Якщо  k < x.key, пошук продовжується у *лівому* піддереві.
* Якщо  k > x.key, пошук продовжується у *правому* піддереві.

**Зауваження**

Час виконання такої процедури становить O(h), де h - висота дерева.