**Керування ресурсами.**

1. **Види пам’яті:**

* **Автоматична (стек)** – використовується для оголошення об’єктів, час життя яких обмежено скоупом (полем видимості) даного об’єкту**.** При виході з поля видимості об’єкт буде знищено.
* **Статична** – існує увесь час роботи програми. Розміщується в спеціальному сегменті даних.
* **Динамічна** – heap. Це та пам’ять, яку потрібно окремо виділяти із вільної пам’яті. Слово «динамічна» тут означає «динамічно-розподілювана». Тобто, на відміну від статичної пам’яті (яка постійна впродовж всього часу роботи програми), динамічна пам’ять характеризується своєю непостійністю – вона виділяється та звільнюється впродовж виконання програми. Статична пам’ять виділяється лише один раз при завантаженні програми (модуля) та звільнюється при її вивантаженні. Динамічна пам’ять використовується коли завчасно невідомо, скільки даних потрібно в ній розмістити. Наприклад, якого розміру буде файл, який треба прочитати, або скільки даних прийде з сокету, тощо. В цьому випадку ресурси виділяються не завчасно, а тоді, коли вони будуть потрібні на етапі виконання програми.

1. **Керування динамічною пам’яттю**

* **Оператори для виділення та звільнення пам’яті:**
  + - new/delete
    - new[]/delete[]
    - malloc/free

**Відповідність операторів**. Якщо ви використовуєте [] в new, то повинні також використовувати [] при вивільненні з delete. Якщо квадратні дужки не використовуються в new, то при вивільненні цієї пам’яті з delete також не треба їх використовувати. Якщо ви використовуєте malloc(), то для вивільнення треба використовувати free().

1. **R.A.I.I. та Guard’и**

* Щоб уникнути витоку ресурсів, використовуйте об’єкти RAII, які «захоплюють» ресурси в своїх конструкторах, та вивільняють в деструкторах.
* Копіювання RAII-об’єктів призводить до копіювання ресурсів, якими вони керують. Поведінка RAII-об’єкту визначає поведінку ресурсу.
* Зазвичай при реалізації RAII класів застосовується одна з систем – заборона копіювання, або дозвіл на копіювання з підрахунком посилань. Але можливі і інші варіанти.

1. **Гарантії безпеки виключень**

* Базова гарантія – при виникненні будь-якого виключення в деякому методі, стан програми повинен залишатись узгодженим, не повинно виникати витоків ресурсів.
* Строга гарантія – якщо при виконанні операції виникає виключення, то це не повинно ніяк вплинути на стан програми. Наприклад, якщо виключення викинуто при вставці елемента в STL контейнер, то останній залишається в тому ж стані, що і до операції вставки.
* Гарантія відсутності виключень – ні при яких обставинах функція не повинна генерувати виключення.

1. **Розумні вказівники**

* **std::auto\_ptr** – розумний вказівник, призначений для «одноосібного» володіння ресурсом:
* при копіюванні передає володіння ресурсом.
* не може бути використаним в STL контейнерах.
* не можна використовувати з іншими типами ресурсів, відмінними від пам’яті.
* **std::unique\_ptr** – покращена версія std::auto\_ptr; також призначений для одноосібного володіння ресурсом.
* забороняє копіювання.
* передача володіння ресурсом відбувається явно через std::move.
* вміє працювати як з вказівниками на окремий об’єкт, так і з масивами.
* можна використовувати з STL контейнерами.
* вміє працювати з custom deleter’ами, тому його можна використовувати з іншими ресурсами, відмінними від пам’яті (файлові дескриптори, сокети, etc).
* **std::shared\_ptr** – розумний вказівник для розділеного володіння ресурсами:
* дозволяє декільком об’єктам посилатись на один і той же ресурс.
* підтримує custom deleter.
* Розмір в два рази більше розміру звичайного вказівника, так як містить вказівник на сам ресурс, та вказівник на лічильник посилань.
* **std::weak\_ptr** – дозволяє уникнути проблем при циклічній залежності з використанням std::shared\_ptr.

1. **Рекомендації по роботі з розумними вказівниками**

* Використовуйте розумні вказівники завжди, коли це можливо.
* Не використовуйте std::auto\_ptr.
* Надавайте перевагу unique\_ptr проти shared\_ptr (якщо це можливо).
* Надавайте перевагу make\_\* функціям, замість передачі вказівників при конструюванні.
* При копіюванні shared\_ptr, не конструюйте його із звичайного вказівника. Натомість, копіюйте саме вже існуючий об’єкт shared\_ptr.

1. **Допоміжні матеріали**

* <https://en.cppreference.com/w/cpp/memory/unique_ptr>
* <https://en.cppreference.com/w/cpp/memory/shared_ptr>
* <https://en.cppreference.com/w/cpp/memory/weak_ptr>
* Скотт Мейерс – Найбільш ефективне використання С++