**Лекція 7 Алгоритми і дані**

**Структурування і абстракція програм.**

Будь-які достатньо великі програми проектуються шляхом декомпозиції задачі – *виділення в задачі деяких структур і їхніх абстракцій*. Абстрагування від проблеми  передбачає ігнорування ряду деталей для того, щоб звести задачу до більш простої. Задачі абстрагування і наступна декомпозиція типові для процесу створення  програм: декомпозиція використовується для поділу програм на компоненти;  абстрагування ж передбачає продуманий вибір компонентів для цієї задачі.

*Структурний підхід* до даних і алгоритмів дає можливість структурування  складної програми. Розробляти сучасну програмну методом „все відразу”  неможливо, вона повинна бути представлена в вигляді деякої структури – складових  частин і зв’язків між ними. Правильне структурування дає можливість на кожному  етапі розробки зосередити увагу розробника на одній оглядовій її частині або  доручити реалізацію різних її частин різним виконавцям.

При структуруванні програм можна застосовувати підхід, який базується на  структуризації алгоритмів і відомий, як „низхідне” проектування – „програмування з верху в низ”, або підхід, який базується на структуризації даних і  відомий, як „висхідне” проектування – „програмування з низу в верх”.

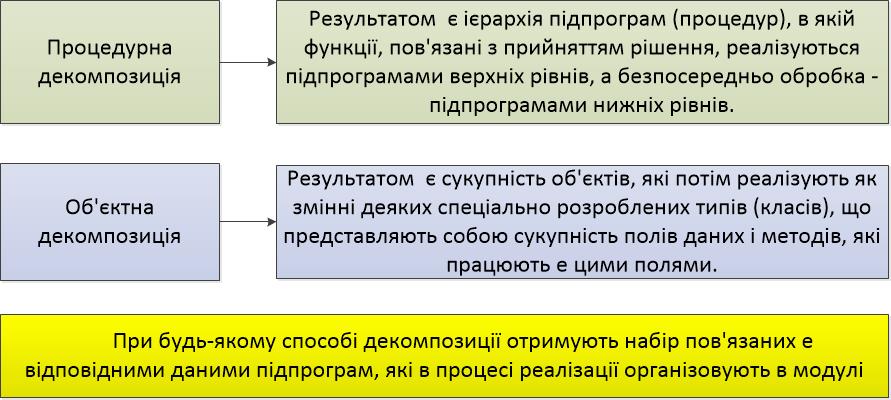
У першому випадку – структуризації алгоритмів – структурують перш за все  дії, які повинна виконувати програма. Велику і складну задачу представляють у  вигляді декількох задач меншого обсягу. Таким чином, модуль самого верхнього  рівня, який відповідає за вирішення задачі в цілому, отримується достатньо простим  і він забезпечує тільки послідовність звертань до модулів, які вирішують задачі нижчого рівня. Потім кожна задача в свою чергу підлягає декомпозиції за такими ж правилами. Процес дроблення продовжується до тих пір, поки на черговому рівні  декомпозиції не отримують задачу, реалізація якої буде досить простою. Для цього  випадку будь-яку програму можна представити набором наступних функціональних  абстракцій. *Аналізуючи таке представлення програми, можна отримати узагальнену абстракцію функції*.

Інший підхід базується на  структуризації даних (програмування – це обробка даних). У програмах можна  застосовувати різні алгоритми, але в реальної програми завжди є замовник, який маючи вхідні дані, хоче, щоб за  ними можна було отримати вихідні дані, а якими засобами це забезпечується – його  не цікавить. Таким чином, *задачею будь-якої програми є перетворення вхідних  даних у вихідні*.

* При проектуванні досить складного програмного забезпечення після визначення його загальної структури виконують декомпозицію компонентів відповідно до обраного підходом до отримання елементів, які, на думку проектувальника, в подальшій декомпозиції не потребують.
* В даний час використовують два способи декомпозиції розроблюваного програмного забезпечення:

• процедурний (або структурний);

• об'єктний.



Мови програмування представляють набір базових типів даних і  операції над ними. Інтегруючи базові типи, програміст створює більш складні типи  даних і визначає нові операції над складними типами. В ідеалі *останній крок  композиції дає типи даних, які відповідають вхідним і вихідним даним задачі, а операції над цими типами реалізують в повному обсязі задачу проекту*. Розглядаючи  програму не як набір функцій, а перш за все, як деякий набір даних, кожен, із яких  має дозволену групу функцій, отримують наступне абстрактне представлення  програми.

Аналіз такого представлення  програми дозволяє також отримати абстракцію даних. Отримані абстракції алгоритмів і  даних лежать в основі багатьох  формалізованих методів розробки  програмного забезпечення.

Не можна протиставляти низхідне проектування висхідному і дотримуватись  одного вибраного підходу. *Реалізація будь-якого реального проекту завжди ведеться  зустрічними шляхами, причому, з постійною корекцією структур алгоритмів за  результатами розробки структур даних і навпаки*.

Ще одним досить продуктивним технологічним методом, який зв’язаний з  структуризацією даних є *інкапсуляція*. Зміст її полягає у тому, що сконструйований  новий тип даних – „будівельний блок” – оформляється таким чином, що його  внутрішня структура стає недоступною для програміста – користувача цього типу.  Програміст, який використовує цей тип даних у своїй програмі, може оперувати з  даними цього типу тільки через виклик функцій, які визначені для цього типу.  Новий тип даних представляється для нього у вигляді „чорного ящика” для якого  відомі входи й виходи, але вміст – невідомий і недоступний.

Сучасні мови програмування блочного типу мають достатньо розвинуті можливості побудови програм з модульною структурою і управління доступом модулів до даних і функцій. Розширення ж мов додатковими можливостями конструювання типів і їх інкапсуляції робить мову об’єктно-орієнтованою. Сконструйовані й повністю закриті типи даних представляють собою класи, а функції, які працюють з їх внутрішньою структурою – методи роботи з класами.  При цьому в значній мірі міняється й сама концепція програмування. Програміст,  який оперує об’єктами, вказує в програмі „що” потрібно зробити з об’єктом, а не  „як” це потрібно зробити.

**Концепція** **структур даних.**

Структури даних і алгоритми є тими матеріалами, з яких будуються програми.  Більше того, сам комп’ютер складається з структур даних і алгоритмів. Вбудовані  структури даних представлені регістрами й словами пам’яті, де зберігаються бінарні  величини. Закладені в конструкцію апаратури алгоритми – це реалізація в  електронних логічних схемах жорстких правил, за якими поміщені в пам’ять дані  інтерпретуються як команди, що підлягають виконанню. Тому *в основі роботи будь- якого комп’ютера лежить вміння оперувати лише з одним видом даних – з  окремими бітами*. Працює ж з цими даними комп’ютер тільки у відповідності з  незмінним набором алгоритмів, які визначаються системою команд центрального  процесора.

Задачі, які вирішуються за допомогою комп’ютера, рідко представляються мовою бітів. Як правило, дані мають форму чисел, літер, текстів, символів і більш  складних структур типу послідовностей, списків і дерев. Структура даних відноситься за своєю суттю до „просторових” понять: її можна звести до схеми організації інформації в пам’яті комп’ютера. Алгоритм ж є  відповідним процедурним елементом в структурі програми – він служить рецептом  розрахунку.

Структури даних, які застосовуються в алгоритмах, можуть бути досить складними. Вибір правильного представлення даних часто служить ключем до вдалого програмування і може в більшій мірі впливати на продуктивність програми, ніж деталі реалізації використовуваного алгоритму. Але, мабуть, ніколи не появиться загальна теорія вибору структур даних, у кожному конкретному випадку  потрібно підходити до цього творчо.

Незалежно від змісту і складності будь-які дані в пам’яті комп’ютера представляються послідовністю бінарних розрядів, а їх значеннями є відповідні бінарні числа. Дані, які розглядаються у вигляді послідовності бітів, мають дуже просту організацію, тобто є слабо структурованими. Більш крупні й змістовніші, ніж біт, „будівельні блоки” для організації довільних даних отримуються на основі поняття „структури даних”.

Поняття „*фізична структура даних” відображає спосіб фізичного представлення даних в пам’яті машини і називається ще структурою зберігання*.

Розгляд структури даних без врахування її представлення в машинній пам’яті називається абстрактною або логічною структурою. В загальному випадку між логічною і відповідною їй фізичною структурами існує відмінність, міра якої залежить від самої структури і особливостей того середовища, в якому вона повинна бути відображена.

Будь-яка структура даних може описуватися, таким чином, на трьох різних рівнях:

**-** *Функціональна специфікація* – вказує для деякого класу імен операції, які дозволені з цими іменами, і властивості цих операцій;

**-** *Логічний опис* – задає декомпозицію об’єктів на більш елементарні об’єкти і декомпозицію відповідних операцій на більш елементарні операції;

**-** *Фізичне представлення* – дає метод розміщення в пам’яті комп’ютера тих величин, які складають структуру, і відношення між ними, а також спосіб кодування операцій на мові програмування.

Одній і тій же функціональній специфікації можуть відповідати декілька логічних описів, які в свою чергу можуть реалізовуватися декількома фізичними представленнями. Проте, потрібно мати впевненість, що декомпозиція кожного нового рівня достатньо добре відображає декомпозицію безпосередньо вищого рівня.

**Класифікація структур даних.**

*Під структурою даних в загальному випадку розуміють множину елементів даних і множину зв’язків між ними*.

Розрізняються прості (базові) структури даних і інтегровані (структуровані). *Простими* називаються такі структури даних, які не можуть бути розділені на складові частини більші, ніж біти. З логічної точки зору прості дані є неподільними одиницями.

*Інтегрованими* називаються такі структури даних, складовими частинами яких є інші структури даних – прості але в свою чергу інтегровані. Інтегровані структури даних конструюються програмістом з використанням засобів інтеграції даних, які представляються мовами програмування.

У залежності від наявності чи відсутності явно заданих зв’язків між елементами даних потрібно розрізняти *незв’язні структури і зв’язні структури*.

Важлива ознака структури даних – її *змінність – зміна кількості* елементів і/або зв’язків між елементами структури. За ознакою змінності розрізняють структури *статичні, напівстатичні, динамічні*.

Ще одна важлива ознака структури даних – *характер впорядкованості* її елементів. За цією ознакою структури можна поділити на *лінійні й нелінійні структури*.

**Операції над структурами даних.**

Над будь-якими структурами даних можуть виконуватися чотири загальні операції: створення, знищення, вибір (доступ), поновлення.

Операція *створення* полягає у виділенні пам’яті для зберігання структури даних. Пам’ять може виділятися в процесі виконання програми або на етапі компіляції. У деяких мовах для структурованих даних, які конструюються програмістом, операція створення включає в себе також встановлення початкових значень параметрів, створюваної структури (ініціалізація). Незалежно від використовуваної мови програмування, наявні в програмі структури даних не появляються „з нічого”, а явно або неявно оголошуються операторами створення.

Операція *знищення* структур даних протилежна до операції створення – вона звільнює пам’ять, яку займала структура, для подальшого використання. Операція знищення дозволяє ефективно використовувати пам’ять.

Операція *вибору* використовується програмістами для доступу до даних в самій структурі. Форма операції доступу залежить від типу структури даних, до якої здійснюється звертання. Метод доступу – одна з найбільш важливих властивостей структур, тому що вона має безпосереднє відношення до вибору конкретної структури даних.

Операція поновлення дозволяє змінити значення даних в структурі даних. Прикладом операції поновлення є операція присвоєння, або, більш складна форма – передача параметрів. Ці чотири операції обов’язкові для всіх структур і типів даних. Крім цих загальних операцій для кожної структури даних можуть бути визначені специфічні операції, які працюють тільки з даними конкретного типу чи структури.

**ПРОСТІ СТРУКТУРИ ДАНИХ.**

**Арифметичні типи**

Стандартні типи даних часто називають арифметичними, оскільки їх можна  використовувати в арифметичних операціях. Для опису основних типів визначено ключові слова: цілий, символьний, логічний, дійсний. За допомогою цілих чисел можна представити кількість об’єктів, яка є  дискретною за своєю природою (тобто кількість об’єктів можна перерахувати).

Внутрішнє представлення величин цілого типу – ціле число в бінарному коді. Внутрішнє представлення дійсного типу складається з двох частин – мантиси і  порядку.

Результати логічного типу отримуються при порівнянні даних будь-яких типів. Величини логічного типу можуть приймати тільки значення true і false. Внутрішня форма представлення значення false – 0 (нуль). Будь-яке інше значення інтерпретується як true.

Значенням символьного типу є символи з деякої наперед визначеної множини. В більшості сучасних персональних комп’ютерів цією множиною є ASCII. Ця множина складається з 256 різних символів, які впорядковані певним чином і містить символи великих і малих букв, цифр і інших символів, включаючи спеціальні керуючі символи. Значення символьного типу займає в пам’яті 1 байт. Іншою широко використовуваною множиною для представлення символьних даних є код Unicode. В Unicode кожний символ кодується двома байтами.

Над арифметичними типами, як і над всіма іншими, можливі перш за всі чотири основних операції: створення, знищення, вибір, поновлення. Специфічні операції над числовими типами – додавання, віднімання, множення і ділення. Ще одна група операцій над арифметичними типами – операції порівняння: „рівно”, „не рівно”, „більше”, „менше” і т.п. Особливість виконання порівнянь на рівність/нерівність дійсних чисел полягає в тому, що оскільки ці числа представляються в пам’яті з деякою точністю, порівняння їх не завжди може бути абсолютно достовірним.

**Перерахований тип**

При написанні програм часто виникає потреба визначити декілька іменованих констант, для яких потрібно, щоб усі вони мали різні значення. Для цього зручно використовувати перерахований тип.

Перерахований тип представляє собою впорядкований тип даних, який визначається програмістом, тобто програміст перераховує всі значення, які може приймати змінна цього типу.

Діапазон значень перечислень визначається кількістю біт, які необхідні для  представлення всіх його значень. Будь-яке значення цілого типу можна явно  привести до перерахованого типу, але *при виході за межі його діапазону результат  буде невизначеним*. При відсутності ініціалізації перша константа приймає нульове значення, а кожній наступній присвоюється на одиницю більше значення від  попереднього.

На фізичному рівні над змінними перерахованого типу визначені операції  створення, знищення, вибору, поновлення. При цьому виконується визначення  порядкового номера ідентифікатора за його значенням і, навпаки, за номером  ідентифікатора його значення. При виконанні арифметичних операцій перечислення перетворюються у ціле.  Оскільки перечислення є типом, який визначається користувачем, для нього можна  вводити власні операції.

**Покажчики.**

Тип „покажчик” представляє собою адресу комірки пам’яті (в більшості сучасних обчислювальних систем розмір комірки – мінімальної адресованої одиниці пам’яті – складає один байт). При програмуванні на низькому рівні робота з адресами складає значну частину програм. При вирішенні прикладних задач з використанням мов високого рівня найбільш часті випадки, коли програміст може використовувати покажчики, наступні:

1. При необхідності представити одну й ту ж ділянку пам’яті, а значить, одні й ті  ж фізичні дані, як дані різної логічної структури.

2. При роботі з динамічними структурами даних.

В С++ розрізняють три види покажчиків – покажчик на об’єкт, на функцію і  на пустий тип, які відрізняються властивостями і набором допустимих операцій.  Покажчик не є самостійним типом, він завжди зв’язаний з якимось іншим типом. У програмах на мовах високого рівня покажчики можуть бути типізованими і нетипізованими. При оголошені типізованого покажчика визначається й тип об’єкту в пам’яті, який адресується цим покажчиком.

Хоча фізична структура адреси не залежить від типу й значення даних, які зберігаються за цією адресою, компілятор вважає покажчики на різні типи такими, що мають різний тип. Таким чином, коли йде мова про *типізовані покажчики*, правильніше говорити не про єдиний тип даних „покажчик”, а про цілу сім’ю типів: „покажчик на ціле”, „покажчик на символ”.

*Нетипізований покажчик* використовується для представлення адреси, за якою містяться дані невідомого типу. Робота з нетипізованими покажчиками суттєво обмежена, вони можуть використовуватися тільки для збереження адреси, звертатися за такою адресою не можна.

Основними операціями, в яких беруть участь покажчики є присвоєння, отримання адреси, вибір.

Перерахованих операцій достатньо для вирішення задач прикладного програмування тому набір операцій над покажчиками в більшості мов програмування цим й обмежується. Системне програмування потребує більш гнучкої роботи з адресами, тому в С/C++ доступні також операції адресної арифметики.

**СТАТИЧНІ СТРУКТУРИ ДАНИХ.**

Статичні структури відносяться до класу структур, які представляють собою  структуровану множину примітивних, базових, структур. Оскільки статичні структури відрізняються відсутністю змінності, пам’ять для них виділяється автоматично – як правило, на етапі компіляції, або при виконанні – в момент активізації того програмного блоку, в якому вони описані. Ряд мов програмування допускають розміщення статичних структур в пам’яті на етапі виконання за явною вимогою програміста, але й у цьому випадку обсяг виділеної пам’яті залишається незмінним до знищення структури. Виділення пам’яті на етапі компіляції є такою зручною властивістю статичних структур, що у ряді задач програмісти використовують їх навіть для представлення об’єктів, які мають властивість змінності. Наприклад, коли розмір масиву невідомий наперед, для нього резервується максимально можливий розмір.

Статичні структури в мовах програмування зв’язані із структурованими типами. Структуровані типи в мовах програмування є тими засобами інтеграції, які дозволяють будувати структури даних будь-якої складності. До таких типів відносяться масиви, структури та їхні похідні типи.

**Масиви.**

Логічно *масив об’єднує елементи одного типу даних*, тобто належить до однорідного типу даних. Більше формально його можна визначити як впорядковану сукупність елементів деякого типу, які адресуються за допомогою одного або декількох індексів. Масиви можна класифікувати за кількістю розмірностей масиву, їх поділяють на одновимірні масиви (вектори), двохвимірні (матриці) і багатовимірні (трьох, чотирьох і більше).

Логічно *масив – це така структура даних, яка характеризується*:

**-** фіксованим набором елементів одного і того ж типу;

**-** кожний елемент має унікальний набір значень індексів;

**-** кількість індексів визначають мірність масиву;

**-** звернення до елемента масиву виконується за ім’ям масиву і значенням індексів для даного елемента.

*Фізична структура масиву* – це спосіб розміщення елементів масиву в пам’яті комп’ютера. Під елемент масиву виділяється кількість байт пам’яті, яка визначається базовим типом елемента цього масиву. Кількість елементів масиву і розмір базового типу визначають розмір пам’яті для зберігання масиву.

Сама найважливіша операція фізичного рівня над масивом – доступ до заданого елемента. Як тільки реалізовано доступ до елемента, над ним може бути виконана будь-яка операція, що має сенс для того типу даних, якому відповідає елемент. Перетворення логічної структури масиву у фізичну називається процесом *лінеаризації*, в ході якого багатовимірна логічна структура масиву перетвориться в одновимірну фізичну структуру.

Адресою масиву є адреса першого байту початкового компоненту масиву. Індексація масивів в С/С++ обов’язково починається з нуля. До операцій логічного рівня над масивами необхідно віднести такі як сортування масиву, пошук елемента за ключем.

**Розріджені масиви.**

На практиці зустрічаються масиви, які через певні причини можуть займати пам’ять не повністю, а частково. Це особливо актуально для масивів великих  розмірів, таких що для їхнього зберігання в повному об’ємі пам’яті може бути недостатньо. *Розріджений масив – це масив, більшість елементів якого рівні між собою, так що зберігати в пам’яті достатньо лише невелику кількість значень відмінних від основного (фонового) значення інших елементів.* При роботи з розрідженими масивами питання розташування їх в пам’яті реалізуються на логічному рівні з врахуванням їхнього типу.

Розрізняють два типи розріджених масивів:

**-** масиви, в яких розташування елементів із значеннями відмінними від фонового, можуть бути описані математично;

**-** масиви з випадковим розташуванням елементів.

До масивів з математичним описом розташування елементів відносяться масиви, в яких існує закономірності в розташуванні елементів із значеннями відмінними від фонового. Елементи, значення яких є фоновими, називають нульовими; елементи, значення яких відмінні від фонового, – ненульовими. Фонове значення не завжди рівне нулю. Ненульові значення зберігаються, як правило, в одновимірному масиві, а зв’язок між розташуванням у розрідженому масиві і в новому, одновимірному, описується математично за допомогою формули, що перетворює індекси масиву в індекси вектору.

На практиці для роботи з розрідженим масивом розробляються функції:

**-** для перетворення індексів масиву в індекс вектору;

**-** для отримання значення елемента масиву з його упакованого представлення за індексами;

**-** для запису значення елемента масиву в її упаковане представлення за індексами.

До масивів з випадковим розташуванням елементів відносяться масиви, в яких не існує закономірностей у розташуванні елементів із значеннями відмінними від фонового.

*Один з основних способів зберігання подібних розріджених матриць* полягає в запам’ятовуванні ненульових елементів в одновимірному масиві і ідентифікації кожного елемента масиву індексами. Дане представлення масиву скорочує вимоги до об’єму пам’яті більш ніж в 2 рази. Спосіб послідовного розподілу має також ту перевагу, що операції над матрицями можуть бути виконані швидше, ніж це можливо при представленні у вигляді послідовного масиву, особливо якщо розмір матриці великий. Методи послідовного розміщення для представлення розріджених матриць звичайно дозволяють швидше виконувати операції над матрицями і більш ефективно використати пам’ять, ніж методи із зв’язаними структурами. Проте послідовне представлення матриць має певні недоліки. Так включення і виключення нових елементів матриці викликає необхідність переміщення великої кількості інших елементів. Якщо включення нових елементів і їхнє виключення здійснюється часто, то повинен бути вибраний метод зв’язаних структур.

Метод зв’язаних структур, проте, переводить структуру даних, що представляється, в інший розділ класифікації. При тому, що логічна структура даних залишається статичною, фізична структура стає динамічною.

**Множини.**

Тип даних „множина” не реалізований як стандартний тип мови програмування С/С++, але дуже часто використовується в програмуванні.

*Множина* – така структура, яка є набором даних одного і того ж типу, що не повторюються (кожен елемент множини є унікальним). Порядок слідування елементів множини на має принципового значення. До множин застосовується стандартний принцип виключення. Це означає, що конкретний елемент або є членом множини, або ні. Множина може бути пустою, таку множину називають нульовою.

Множина є підмножиною іншої множини, якщо в цій другій множині можна  знайти усі елементи, які є в першій множині. Відповідно, множина вважається  надмножиною іншої множини, якщо вона містить усі елементи цієї другої множини. Кожен окремий елемент є членом множини, якщо він входить до складу елементів множини.

Над *множинами визначені наступні специфічні операції*:

1. Об’єднання множин. Результатом є множина, що містить елементи початкових множин.

2. Перетин множин. Результатом є множина, що містить спільні елементи початкових множин.

3. Різниця множин. Результатом є множина, яка містить елементи першої множини, які не входять в другу множину.

4. Симетрична різниця. Результатом є множина, яка містить елементи, які входять до складу однієї або другої множини (але не обох).

5. Перевірка на входження елемента в множину. Результатом цієї операції є значення логічного типу, що вказує чи входить елемент в множину.

**Структури.**

На відміну від масивів чи множин, усі елементи яких однотипні, структура може містити елементи різних типів. *Елементи структури називаються полями структури і можуть мати довільний тип, крім типу цієї ж структури, але можуть бути покажчиками на неї*. Якщо при описі структури відсутній тип структури, обов’язково повинен бути вказаний список змінних, покажчиків або масивів визначеної структури. *Звернення до окремих полів структури замінюються на їхні адреси ще на етапі компіляції*.

Самою найважливішою операцією для структури є операція доступу до вибраного поля структури – операція кваліфікації. Над вибраним полем структури можливі будь-які операції, які допустимі для типів цього поля.

Більшість мов програмування підтримує деякі операції, які працюють із структурою, як з єдиним цілим, а не з окремими її полями. Це операція присвоєння значення одного запису іншому однотипному запису, при цьому відбувається поелементне копіювання.

**Об**’**єднання.**

Об’єднання представляють собою *частковий випадок структури, усі поля якої розміщуються за однією ж і тою ж адресою*. Формат опису такий же, як і в структури. Довжина об’єднання рівна найбільшій із довжин його полів. У кожен момент часу в змінній типу об’єднання зберігається тільки одне значення, і відповідальність за його правильне використання лягає на програміста.

Об’єднання застосовуються для економії пам’яті в тих випадках, коли відомо, що більше одного поля одночасно не потрібно, а також для різної інтерпретації одного і того ж бітового представлення.

Дуже часто деякі об’єкти програми відносяться до одного й того ж класу, відрізняючись лише деякими деталями. У цьому випадку застосовують комбінацію структурного типу і об’єднання. Об’єднання використовують як поля структури, при цьому в структурі включають поле, яке визначає, який саме елемент об’єднання використовується в кожний момент.

У загальному випадку змінна структура буде складатися з трьох частин: набір спільних компонентів, мітки активного компоненту і частини зі змінними компонентами.

**Бітові типи.**

В ряді задач може стати в нагоді робота з окремими бінарними розрядами даних. Частіше всього такі задачі виникають в системному програмуванні, коли, наприклад, окремий розряд зв’язаний з станом окремого апаратного перемикача або окремої шини передачі даних. Дані такого типу представляються у вигляді набору бітів, які упаковані в байти або слова, і логічно не зв’язаних один з одним. Операції над такими даними забезпечують доступ до вибраного біта даного.

Бітові поля – це особливий вид полів структури. Вони використовуються для компактного розміщення даних, наприклад, прапорців типу „так/ні”. Мінімально адресована комірка пам’яті – 1 байт, а для зберігання прапорця достатньо одного біта. Бітові поля описуються за допомогою структурного типу. Бітові поля можуть бути довільного цілого типу. Ім’я поля може бути відсутнім, такі поля використовуються для вирівнювання на апаратну межу. Доступ до поля здійснюється звичайним способом – за іменем.

Над бітовими типами можливі три групи специфічних операцій: операції алгебри логіки, операції зсуву, операції порівняння. Операції бульової алгебри – НІ („~”), АБО („|”), І („&”), виключне АБО („^”). Ці операції і за назвою, і за змістом подібні на операції над логічними аргументами, але відмінність у їх застосуванні до бітових аргументів полягає в тому, що операції виконуються над окремими розрядами. В мові C/С++ для побітових і загальних логічних операцій використовуються різні позначення. Операції зсуву виконують зміщення бінарного коду на задану кількість розрядів ліворуч або праворуч. Із трьох можливих типів зсуву (арифметичний, логічний, циклічний) в мовах програмування частіше реалізується лише логічний.

**Таблиці.**

Елементами векторів і масивів можуть бути інтегровані структури. Одна з таких складних структур – таблиця. З фізичної точки зору таблиця є вектором, елементами якого є структури. Характерною логічною особливістю таблиць є те, що доступ до елементів таблиці проводиться не за номером (індексом), а за ключем – значення однієї з властивостей об’єкту, який описується структурою-елементом таблиці. Ключ – це властивість, що ідентифікує дану структуру в множині однотипних структур і є, як правило, унікальним в даній таблиці. Ключ може включатися до складу структури і бути одним з його полів, але може і не включатися в структуру, а обчислюватися за деякими її властивостями. Таблиця може мати один або декілька ключів.

Основною операцією при роботі з таблицями є операція доступу до структури за ключем. Вона реалізовується процедурою пошуку. Оскільки пошук може бути значне більш ефективним в таблицях, впорядкованих за значеннями ключів, досить часто над таблицями необхідно виконувати операції сортування.

Іноді розрізняють таблиці з фіксованою і із змінною довжиною структури. Очевидно, що таблиці, які об’єднують структури ідентичних типів, будуть мати фіксовані довжини структур. Необхідність в змінній довжині може виникнути в задачах, подібних до тих, які розглядалися для об’єднань. Як правило таблиці для таких задач і складаються із структур до складу яких входять об’єднання, тобто зводяться до фіксованої (максимальній) довжини структури. Значно рідше зустрічаються таблиці з дійсно змінною довжиною структури. Хоча в таких таблицях і економиться пам’ять, але можливості роботи з такими таблицями обмежені, оскільки за номером структури неможливо визначити її адресу. Таблиці із  структурами змінної довжини обробляються тільки послідовно – в порядку  зростання номерів структур.

Доступ до елемента такої таблиці звичайно  здійснюється в два кроки. На першому кроці вибирається постійна частина  структури, в якій міститься, – в явному чи неявному вигляді – довжина структури.  На другому кроці вибирається змінна частина структури у відповідності з її  довжиною. Додавши до адреси поточної структури її довжину, одержують адресу  наступної структури.

**НАПІВСТАТИЧНІ СТРУКТУРИ ДАНИХ.**

**Характерні особливості напівстатичних структур.**

Напівстатичні структури даних характеризуються наступними ознаками:

**-** вони мають змінну довжину і прості процедури її зміни;

**-** зміна довжини структури відбувається в певних межах, не перевищуючи якогось  максимального (граничного) значення.

Якщо напівстатичну структуру розглядати на логічному рівні, то це  послідовність даних, зв’язана відносинами лінійного списку. Доступ до елемента  може здійснюватися за його порядковим номером.

Фізичне представлення напівстатичних структур даних в пам’яті – це  звичайно послідовність комірок в пам’яті, де кожний наступний елемент розташований в пам’яті в наступній комірці. Фізичне представлення може мати також вид одно-направленого зв’язного списку (ланцюжки), де кожний наступний  елемент адресується покажчиком, який знаходиться в поточному елементі. У цьому  випадку обмеження на довжину структури менш строгі.

До напівстатичних структур відносяться стеки, черги, деки, лінійні списки, мультисписки, ***стрічки*** (string)

**ДИНАМІЧНІ** **СТРУКТУРИ ДАНИХ.**

**Зв'язне** **представлення** **даних** **в** **пам'яті.**

Динамічні структури за визначенням характеризуються відсутністю фізичної  суміжності елементів структури в пам’яті, непостійністю і непередбачуваністю  розміру (кількість елементів) структури в процесі її обробки. Оскільки елементи динамічної структури розташовуються за не  передбачуваними адресами пам’яті, адресу елемента такої структури не можна обчислити за адресою початкового або попереднього елемента. Для встановлення  зв’язку між елементами динамічної структури використовуються покажчики, через  які встановлюються явні зв’язки між елементами. Таке представлення даних в  пам’яті називається зв’язним. Елемент динамічної структури складається з двох  полів:

**-** інформаційного поля або поля даних, в якому містяться ті дані, заради яких і створюється структура;

**-** поле зв’язку, в якому міститься один або декілька покажчиків, які зв’язують даний елемент з іншими елементами структури.

Коли зв’язне представлення даних використовується для вирішення прикладної задачі, для кінцевого користувача „видимим” робиться тільки вміст інформаційного поля, а поле зв’язку використовується тільки програмістом- розробником.

Переваги зв’язного представлення даних:

**-** можливість забезпечення значної змінності структур;

**-** розмір структури обмежується тільки доступним об’ємом машинної пам’яті;

**-** при зміні логічної послідовності елементів структури потрібно виконати не переміщення даних в пам’яті, а тільки корекцію покажчиків.

Разом з тим зв’язне представлення не позбавлене й недоліків, основні з яких:

**-** робота з покажчиками вимагає більш високої кваліфікації від програміста;

**-** на поля зв’язку витрачається додаткова пам’ять;

**-** доступ до елементів зв’язної структури може бути менш ефективним за часом.

Останній недолік є найбільш серйозним і саме ним обмежується застосування  зв’язного представлення даних. Якщо в суміжному представленні даних для  обчислення адреси будь-якого елемента у всіх випадках достатньо номера елемента  і інформації, яка міститься в описі структури, то для зв’язного представлення адреса  елемента не може бути обчислена з початкових даних. Опис зв’язної структури  містить один або декілька покажчиків, які дозволяють увійти до структури, далі  пошук необхідного елемента виконується проходженням ланцюжком покажчиків  від елемента до елемента. Тому зв’язне представлення практично ніколи не  застосовується в задачах, де логічна структура даних має вигляд вектора або масиву  – з доступом за номером елемента, але часто застосовується в задачах, де логічна  структура вимагає іншої початкової інформації доступу (таблиці, списки, дерева і  т.д.).

**НЕЛІНІЙНІ СТРУКТУРИ ДАНИХ**

До нелінійних структур відносяться графи та дерева.