ПЕРЕЛІКИ, СТРУКТУРИ ТА ОБ'ЄДНАННЯ

Переліки

Перелічуваний тип (enumerated type) – *тип даних, множиною значень якого є обмежений список ідентифікаторів – констант цього типу.*

Перелічуваний тип визначається як набір ідентифікаторів, які виконують ту саму роль, що і звичайні іменовані константи, але ідентифікатори – константи перелічуваного типу – будуть пов'язані з цим типом, що дає можливість контролю за «правильним» використанням таких констант. Використання переліків дозволяє зробити текст програми більш зрозумілим.

Перелічуваний тип – це тип, що визначається користувачем та складається із набору цілих констант, які називаються нумераторами (або елементами переліку)

Загальний вигляд команди визначення перелічуваного типу та змінних – переліків (квадратні дужки використовуються для позначення необов'язкового елементу синтаксичної конструкції):

```
enum [ім'я_перелічуваного_типу]
{ список_елементів_переліку }
 [список_змінних_переліків];
```

де

ім'я_перелічуваного_типу – ім'я перелічуваного типу, необов'язкове. Якщо ім'я перелічуваного типу не вказане, то визначається анонімний перелік;

список_елементів_переліку – список ідентифікаторів (нумераторів, елементів переліку), відокремлених комами;

список_змінних_переліків – оголошення змінних-переліків.

Кожний блок визначення даних (на відміну від блоку опису команд) має закінчуватися символом «;»

Кожний ідентифікатор (елемент переліку) в своїй області видимості має бути унікальним, проте їх значення можуть повторюватися.

Областю видимості нумераторів є блок, який їх охоплює (область видимості самого переліку), тобто елементи переліку видимі в тій області, в якій визначено перелік. Наступний фрагмент – не відкомпілюється:

```
enum A {a, b, c};
enum B {a, x, y};
```

оскільки нумератор a повторно визначений і в переліку A, і в переліку B.

Кожний елемент переліку в загальному має наступний вигляд:

ім'я-елементу [= ціла-константа]

Кожному елементу присвоюється ціле значення, яке відповідає певному місцю розташування цього елемента в переліку. За умовчанням, значення присвоюються в порядку зростання, починаючи з нуля: першому елементу присвоюється значення 0, наступному – 1, і т.д.:

```
enum Color { RED, YELLOW, GREEN };
```

Елемент RED має значення 0, YELLOW – значення 1, GREEN – значення 2. Значення для елементу переліку можна надати явно:

```
enum Color { RED=1, YELLOW, GREEN };
```

Елемент RED тепер має значення 1. Якщо наступним елементам переліку не присвоюються явні значення, то вони отримують наступні за порядком значення; – значення попереднього елемента, збільшене на одиницю, тобто елемент YELLOW отримує значення 2, а елемент GREEN – значення 3.

Кожний елемент переліку опрацьовується як константа, він має мати унікальне ім’я в тій області, в якій визначено цей перелік. Значення, які визначаються елементами переліку, можуть бути неунікальними, наприклад:

```
enum Color { RED=2, YELLOW=1, GREEN };
```

– значення елементів RED, YELLOW та GREEN дорівнюють 2, 1 та 2 відповідно. Значення 2 використовується кілька разів, це допускається

Змінні перелічуваних типів

Змінні-переліки (змінні перелічуваних типів) можна оголосити в команді визначення перелічуваного типу

```
enum Color { RED, YELLOW, GREEN } x, y;
```

– визначається перелічуваний тип Color та переліки (змінні) x, y, які можуть набувати значень RED, YELLOW та GREEN.

Або так

```
enum Color { RED, YELLOW, GREEN };
enum Color x, y;
```

чи так

```
enum Color { RED, YELLOW, GREEN };
Color x, y;
```

Можна оголошувати переліки неіменованих (анонімних) типів: ім’я типу не вказується, але можна оголошувати змінні:

```
enum { RED, YELLOW, GREEN } x, y;
```

– змінні x та y – переліки неіменованого перелічуваного типу, які можуть набувати значень RED, YELLOW та GREEN. Якщо при опрацюванні функцією параметра-переліку деяке значення не опрацьовується, компілятор може вивести попередження про пропущене значення.

Елементи переліку неявно перетворюються до типу int

```
int red = RED;
```

При використанні переліків зустрічаються наступні проблеми:

1) Відображення значення елемента переліку в літерний рядок, який містить ім'я цього елемента, тобто RED → "RED".
2) Ітерація по елементах переліку та контроль виходу за межі: скільки б ми не додавали елементів до переліку, завжди є константа, яка на одиницю перевищує значення останнього елемента

Структури

*Структури – це об'єднані дані **різних типів**, які характеризують певну сутність та за цим змістом пов'язані між собою*

Структурний тип – складається із набору елементів, які називаються полями. Загальний вигляд команди визначення структурного типу та змінних – структур:

```
struct [ім'я_структурного_типу]
{ список_полів } [список_змінних];
```

де

ім'я_структурного_типу – ім'я структурного типу, необов'язкове. Якщо ім'я структурного типу не вказане, то визначається анонімна структура;

список_полів – список оголошень полів; оголошення поля аналогічне до оголошення змінних.

Імена полів в структурі мають бути унікальними, але ім'я поля цієї структури можна використовувати як ідентифікатор іншого ресурсу – змінної, типу, функції, поля іншої структури чи об'єднання. *В різних структурах можна використовувати однакові імена полів.*

Команда визначення структурного типу – це команда визначення даних, яка містить блок визначення даних (елементів структури). Кожний блок визначення даних (на відміну від блоку опису команд) має закінчуватися символом «;»

Наприклад:

```
enum Kurs { I = 1, II, III, IV, V, VI };
enum Spec { PC, KI, KN, IT, IK };
struct Student
{
	string surname;
	unsigned birYear;
	Kurs kurs;
	Spec spec;
};
```

Оголошення змінних структур

Як і для переліків, змінні-структури (змінні структурних типів) можна оголосити в команді визначення структурного типу:

```
struct Student
```

```
{
        string surname;
        unsigned birYear;
        int kurs;
} st1, st2;
```

– визначається структурний тип Student та структури (змінні) st1 та st2.

Або так (кращий спосіб)

```
 struct Student
{
        string surname;
        unsigned birYear;
        int kurs;
};
struct Student st1, st2;
```

чи так

```
 struct Student
{
        string surname;
        unsigned birYear;
        int kurs;
};
Student st1, st2;
```

Як і для інших типів, для структурного можна оголошувати масиви структур та вказівники на структури:

```
Student s3[25], * ps;
```

- масив s3 із 25 елементів типу Student та вказівник ps на структуру типу Student.

Вкладені структури

Одне (чи більше) з полів структури, в свою чергу може бути структурою – такі структури називаються **вкладеними**. Для доступу до вкладеної структури необхідно вказати, що вкладена внутрішня структура належить до області видимості зовнішньої структури за допомогою операції :: (дві двокрапки) – операції видимості.

Наприклад:

```
struct Student
{
        string surname;
        unsigned birYear;
        int kurs;
        struct adresStud
        {
                string city;
                int number;
        } a;

};

Student st1;
Student::adresStud adr;
```

Єдиним обмеженням при визначенні структур є те, *що не допускається рекурсія при визначенні структурних типів*, – ні безпосередньо, ні через інші типи – не можна при визначенні поля використовувати ім'я цієї ж структури.

```
struct Student
{
	Student st1;// помилка

};
```

Але

```
struct Student
{
	Student*  st1; ;// дозволено

};
```

Розподіл пам'яті для структур

В пам'яті для структури виділяється неперервна область, розмір якої більший або дорівнює сумі розмірів всіх полів: є нюанс, який називається вирівнюванням – розмір області пам'яті, виділеної для структури, може бути більшим за суму розмірів полів цієї структури.

Поля, вказані при визначенні структури першими, розміщуються в комірках з молодшими адресами.

Ініціалізація структури

Ініціалізація структури виконується аналогічно до ініціалізації масиву:

```
struct Student
{
	string surname;
	unsigned birYear;
	int kurs;
};
Student st1 = {"Petrenko", 2003, 4};
```

– створюємо структуру – змінну типу Student та присвоюємо значення всім її полям.

Порядок дуже важливий при ініціалізації структури, бо компілятор встановлює позиційну відповідність між полями при визначенні структури та константами в ініціалізаторі – порядок значень в ініціалізаторі має збігатися з порядком полів при визначенні структури. *Якщо якесь поле залишилося не заповненим, то воно автоматично заповнюється значенням*

- *0 – для цілих типів,*
- *NULL – для вказівників,*
- *'\0' (нуль-символ) – для літерних рядків*

До елементів структури можна звертатися двома способами:

- **за допомогою імені структури –** здійснюється через операцію . (крапка), загальний вигляд – наступний: структура.поле. Наприклад:

```
#include <iostream>
using namespace std;
```

```
struct Student
 {
      string surname;
      unsigned birYear;
      int kurs;
 };
int main()
{
 Student st1;
 st1.surname = "Petrenko";
 st1.birYear = 2003;
 st1.kurs = 4;
 return 0;
}
```

- **за допомогою вказівника на структуру** – використовує операцію роз-іменування вказівника * (зірочка), загальний вигляд – наступний: (*вказвник-на-структуру).поле. Наприклад:

```
#include <iostream>
using namespace std;

struct Student
 {
      string surname;
      unsigned birYear;
      int kurs;
 };
int main()
{
 Student* p = new Student;
 (*p).surname = "Petrenko";
 (*p).birYear = 2003;
 (*p).kurs = 4;
 return 0;
}
```

– дужки обов'язкові, бо згідно правилу «суфікс важливіший префіксу», операція доступу . (крапка) має більш високий пріоритет, ніж операція роз'іменування * (зірочка).

Операції над структурами

1) присвоєння полю структури значення відповідного типу;

2) присвоєння всій структурі значення того самого структурного типу;

3) отримання адресу структури за допомогою операції &;

4) отримання доступу до будь-якого поля структури;

5) визначачення розміру структури та її полів за допомогою операції sizeof().

Імена структурних змінних можна використовувати в якості операндів операції присвоєння. При цьому обидві структурні змінні (перед та після знаку присвоєння =) мають бути оголошені за допомогою одного і того самого структурного типу:

```
#include <iostream>
using namespace std;

struct Student
 {
      string surname;
      unsigned birYear;
      int kurs;
 };
int main()
{
 Student st1 ={"Petrenko",2003, 1};
 Student st2=st1;
 st2.kurs = 2;
 return 0;
}
```

Передавання структур у функції

Зі структурами можна виконувати всі ті дії при передаванні у функції, що з іншими типами:

1) всю структуру можна передати у функцію:

```
void f1(Student s)
{
 cout << s.surname << " "
      << s.birYear << " "
      << s.kurs << " ";
      }
```

2) можна передати вказівник на структуру:

```
void f2(Student *s)
{
 cout << (*s).surname << " "
      << (*s).birYear << " "
      << (*s).kurs << " ";
      }
```

3) можна передавати елементи структури:

```
void f3(string s, unsigned bYear, int k)
{
 cout << s << " "
      << bYear << " "
      << k << endl;
}
```

4) можна повертати структури як результат функції:

```
Student f4(Student s)
```

```
{
 return s;
}
```

5) можна описати параметри-посилання на структуру – це забезпечить можливість передачі структур із функцій не як результат функції, а як вихідний (результуючий) параметр:

```
void f5(Student& s)
{
 // ...
}
```

Об'єднання

Як і структурний, об'єднуваний тип – це тип, що визначається користувачем та складається із набору елементів, які називаються полями.

Загальний вигляд команди визначення об'єднуваного типу та змінних – об'єднань:

```
union [ім'я_об'єднуваного_типу]
{ список_полів } [список_змінних_об'єднань];
```

де

`ім'я_об'єднуваного_типу` – ім'я об'єднуваного типу, необов'язкове. Якщо ім'я об'єднуваного типу не вказане, то визначається анонімне об'єднання;

`список_полів` – список оголошень полів; оголошення поля аналогічне до оголошення змінних;

В об'єднанні має бути по крайній мірі одне поле.

Імена полів в об'єднанні мають бути унікальними, але ім'я поля цього об'єднання можна використовувати як ідентифікатор іншого ресурсу – змінної, типу, функції, поля іншого об'єднання чи структури. В різних об'єднаннях можна використовувати однакові імена полів.

Команда визначення об'єднуваного типу – це команда визначення даних, яка містить блок визначення даних (елементів об'єднання). Кожний блок визначення даних (на відміну від блоку опису команд) має закінчуватися символом «;»

```
union Pay
 {
      int total;
      double tax;
 };
```

Як і для переліків та структур, змінні-об'єднання (змінні об'єднуваних типів) можна оголосити в команді визначення об'єднуваного типу:

```
union Pay
```

```
{
      int total;
      double tax;
} p1,p2;
```

– визначається об'єднуваний тип Pay та об'єднання (змінні) p1,p2.

Або так

```
union Pay
{
      int total;
      double tax;
};
union Pay p1, p2;
```

чи так

```
union Pay
{
      int total;
      double tax;
};
Pay p1, p2;
```

Як і для інших типів, для об'єднуваного можна оголошувати масиви об'єднань та вказівники на об'єднання:

```
Pay p3[5], *p4;
```

– оголосили масив p3 із 5 елементів типу Pay та вказівник p4 на об'єднання типу Pay.

Як і для структури, так і для об'єднання, одне (чи більше) з полів певної структури (певного об'єднання), в свою чергу може бути структурою чи об'єднанням – такі структури чи об'єднання називаються вкладеними. Все, що стосувалося вкладених структур, стосується і вкладених об'єднань.

Ініціалізація об'єднання

Ініціалізація об'єднання виконується аналогічно до ініціалізації масиву чи структури, проте в команді ініціалізації можна надати значення лише першому полю об'єднання:

```
union Pay
{
      int total;
      double tax;
};
Pay p1 = { 23};

cout << p1.total << endl;
cout << p1.tax << endl;
```

Над об'єднаннями можна виконувати ті самі операції, що і над структурами. Об'єднання передаються у / із функції аналогічно структурам.

Визначення типів

Команда `typedef` визначає синонім типу. Її загальний вигляд:

```
typedef визначення_типу нове_ім'я_типу;
```

де

`визначення_типу` – будь-яке допустиме в C++ визначення типу;

`нове_ім'я_типу` – синонім (нове ім'я), що надається цьому типу.

Наприклад:

`int a[10];` – оголошує змінну a як масив із 10 елементів цілого типу.

Використаємо команду визначення синоніму типу typedef

```
typedef int A[10];
A a;
```

– визначає тип A як сукупність масивів із 10 елементів цілого типу та оголошує змінну a цього типу.

Таким чином, ключове слово typedef перетворює команду оголошення змінної в команду визначення синоніму типу. Команда визначення синоніму типу typedef використовується для того, щоб замінити складні та громіздкі оголошення простішими.

Зокрема, в мові С імена переліків, структур та об'єднань – не повноправні імена типів, а лише теги, які в оголошенні змінних, описі параметрів та результату функцій слід використовувати лише разом із ключовими словами enum, struct та union відповідно.

В таких випадках і використовувалася команда typedef для визначення більш компактного імені. На відміну від оголошень enum, struct та union, команда typedef не вводить нового типу, а лише призначає нове ім'я вже визначеному типу.

Еквівалентність типів

Існує кілька схем для визначення, чи еквівалентні типи двох об'єктів. Найчастіше використовуються схеми, що називаються *структурна еквівалентність типів та іменна еквівалентність типів* (в англомовній термінології structural type system та nominative type system відповідно).

Відповідно до схеми структурної еквівалентності типів два об'єкти належать до одного типу, якщо їх компоненти мають однакові типи.

Відповідно до схеми іменної еквівалентності типів два об'єкти мають один і той самий тип лише тоді, коли вони визначені за допомогою імені цього ж типу.

Таким чином, відповідно до схеми іменної еквівалентності типів, два структурні типи будуть різними навіть тоді, коли вони мають одні і ті самі компоненти. Наприклад:

```
struct s1 { int a; };
struct s2 { int a; };
```

– визначено два різні типи. Змінні цих типів – несумісні:

```
s1 x;
s2 y = x; // помилка
```

Структурні типи – відрізняються від простих, тому наступне присвоєння – помилкове:

```
s1 x;
int i = x; // помилка: невідповідність типів
```

Окремі визначення типів визначають різні типи (навіть якщо ці визначення буквально ідентичні).

Мови C та C++ побудовані на жорсткій іменній еквівалентності типів. Команда визначення синоніму типу `typedef` реалізує схему іменної еквівалентності типів – вона дозволяє задати нове ім’я типу, не визначаючи новий тип: в оголошенні, яке починається ключовим словом `typedef`, описується не змінна, а вводиться нове ім’я для типу