Лекція 3

Складні типи

У цій главі...

3.1. Оператори typedef і sizeof

З типами даних тісно зв'язані два дуже корисних оператори.

3.1.1. Оператор typedef

Цей оператор дозволяє перейменувати один з існуючих типів даних. На перший погляд, це зовсім зайве. Якщо тип уже має одне ім'я, навіщо йому інше? Як правило, це роблять, щоб полегшити подальшу модифікацію програми. Наприклад, деяка програма виконує обчислення з числами типу `float`. У якійсь ситуації можуть знадобитися більш точні обчислення. Якщо не використовувати перейменування типу, прийдеться пройтися по всій програмі і замінити `float` на `double`. А от якщо в програмі тип `float` був перейменований, то його псевдонім, скажемо `real`, стає деяким параметром, якому можна знову перевизначити. Для того щоб перейти до обчислень з підвищеною точністю, досить змінити лише один оператор: замість

```
typedef float real;
```

варто написати

```
typedef double real;
```

Тоді програма примет наступний вид.

Перейменування типу

```
#include <iostream.h>

int main()
 {
   typedef double real;
   real a=1.0e300, b=1.0e200, c=a*b;
   cout << c << endl;
   return 0;
}
```

Ця програма дуже підступна. Значення змінних `a` і `b` лежать у припустимому діапазоні типу `double`, але їхній добуток виходить за його межі. Однак у мові C++ такі ситуації за замовчуванням не відслідковуються. Програма без коливань виводить на екран невірну відповідь.

```
1.127201e-231
```

Зовсім очевидно, що нам необхідно розширити діапазон, задавши тип `long double`. У приведеному вище прикладі для цього досить замінити оператор

```
typedef double real;
```

оператором

```
typedef long double real;
```

Тепер відповідь буде вірною.

```
1.0e+500
```

Звичайно, у такій короткій програмі застосування оператора `typedef` не занадто помітно, однак у великій програмі, що складається з багатьох файлів, він дозволяє уникнути стомлюючої перевірки і заміни оголошень змінних типу `double` на змінні типу `long double`.

Варто мати на увазі, що оператор `typedef` не створює новий тип, він просто створює псевдонім існуючого типу. До речі, це перейменування діє в межах усієї програми. Інакше кажучи, один тип можна перейменувати тільки один раз.

Розглянемо приклад.

Повторне перейменування

```
#include <iostream.h>

int main()
 {
   typedef double real;
   real a=1.0e300, b=1.0e200, c=a*b;
   cout << c << endl;

   typedef long double real; // Помилка! Повторне перейменування!
   real d=1.0e300, e=1.0e200, f=d*e;
   cout << f << endl;
   return 0;
}
```

У той же час використання псевдоніму не забороняє застосування вихідного імені типу.

Застосування псевдоніма і старого імені

```
#include <iostream.h>

int main()
 {
   typedef long double real;
   real a=1.0e400, b=1.0e500, c=a*b;
   cout << c << endl;

   long double d;
   d=c*c;
   cout << d << endl;
   return 0;
}
```

У результаті обчислень на екран будуть виведені числа.

```
 1.0e+900
1.0e+1800
```

Оператор `typedef` дуже часто зустрічається в заголовних файлах. Він забезпечує машинну незалежність програм. Зокрема, саме за допомогою оператора `typedef` визначені типи `wchar_t` і `size_t`.

3.1..2. Оператор sizeof

Цей оператор дозволяє обчислити розмір змінної чи типу в байтах. Операндом цього оператора може бути ім'я типу, змінна чи вираз.

Проілюструємо сказане прикладом.

Застосування оператора sizeof

```
#include <iostream.h>

int main()
{
   cout <<"sizeof(char)             = "<< sizeof(char)<< endl;
   cout <<"sizeof(int)              = "<< sizeof(int)<< endl;
   cout <<"sizeof(long int)         = "<< sizeof(long int)<< endl;
   cout <<"sizeof(float)            = "<< sizeof(float)<< endl;
   cout <<"sizeof(double)           = "<< sizeof(double)<< endl;
   cout <<"sizeof(long double)      = "<< sizeof(long double)<< endl;

   char aChar;
   int anInt;
   long aLong;
   float aFloat;
   double aDouble;
   long double aLongDouble;

   cout <<"sizeof(aChar)            = "<< sizeof(aChar)<< endl;
   cout <<"sizeof(anInt)            = "<< sizeof(anInt)<< endl;
   cout <<"sizeof(aLong)            = "<< sizeof(aLong)<< endl;
   cout <<"sizeof(aFloat)           = "<< sizeof(aFloat)<< endl;
   cout <<"sizeof(aDouble)          = "<< sizeof(aDouble)<< endl;
   cout <<"sizeof(aLongDouble)      = "<< sizeof(aLongDouble)<< endl;
```

```
    cout <<"sizeof(2*anInt+20)        = " << sizeof(2*anInt+20)<<endl;
    cout <<"sizeof(2*aLong+100)       = " << sizeof(2*aLong+100)<<endl;
    cout <<"sizeof(2*aFloat+1.0)      = " << sizeof(2*aFloat+1.0)<<endl;
    cout <<"sizeof(aDouble+3)         = " << sizeof(aDouble+3)<<endl;
    cout <<"sizeof(aLondDouble-4)     = " << sizeof(aLongDouble-10)<<endl;

    cout <<"sizeof((char)anInt        = " << sizeof((char)anInt)<< endl;
    cout <<"sizeof((char)aFloat       = " << sizeof((char)aFloat)<< endl;
    cout <<"sizeof((char)aDouble      = " << sizeof((char)aDouble)<< endl;
    cout <<"sizeof((int)aLong         = " << sizeof((int)anInt)<< endl;
    cout <<"sizeof((int)aFloat        = " << sizeof((int)aFloat)<< endl;
    cout <<"sizeof((int)aDouble       = " << sizeof((int)aDouble)<< endl;
    cout <<"sizeof((float)aDouble     = " << sizeof((float)aDouble)<< endl;
    cout <<"sizeof(double)aLongDouble = " << sizeof((double)aLongDouble)<<
endl;

    return 0;
}
```

Результати роботи цієї програми приведені нижче.

```
 sizeof(char)              = 1
sizeof(int)               = 2
sizeof(long int)          = 4
sizeof(float)             = 4
sizeof(double)            = 8
sizeof(long double)       = 10
sizeof(aChar)             = 1
sizeof(anInt)             = 2
sizeof(aLong)             = 4
sizeof(aFloat)            = 4
sizeof(aDouble)           = 8
sizeof(aLongDouble)       = 10
sizeof(2*anInt+20)        = 2
sizeof(2*aLong+100)       = 4
sizeof(2*aFloat+1.0)      = 4
sizeof(aDouble+3)         = 8
sizeof(aLondDouble-4)     = 10
sizeof((char)anInt        = 1
sizeof((char)aFloat       = 1
sizeof((char)aDouble      = 1
sizeof((int)aLong         = 2
sizeof((int)aFloat        = 2
sizeof((int)aDouble       = 2
sizeof((float)aDouble     = 4
sizeof(double)aLongDouble = 8
```

Оператор `sizeof` повертає значення типу `size_t`, що є різновидом цілого типу. Оскільки цілий тип на різних машинах може мати різну довжину, оператор `sizeof` дозволяє забезпечити машинну незалежність програми. У заголовному файлі цей тип визначений за допомогою оператора `typedef`. Варто особливо підкреслити, що вираз усередині оператора `sizeof` *не обчислюється* — для визначення типу результату це не обов'язково.

3.2. Перерахування

Перерахування — це визначений користувачем тип, що дозволяє іменувати літерали інтегральних типів, тобто символьні і цілочисельні. Наприклад, оператор

```
enum {ZERO,ONE, TWO, THREE, FOUR, FIVE};
```

еквівалентний операторам

```
 const int ZERO  = 0;
const int ONE   = 1;
const int TWO   = 2;
const int THREE = 3;
const int FOUR  = 4;
const int FIVE  = 5;
```

Як бачимо, за замовчуванням значення, привласнені константам, починаються з нуля і послідовно збільшуються на одиницю. У той же час елементам перерахування можна привласнити будь-яке припустиме інтегральне значення, не обов'язкове ціле. Наприклад, як перерахування можна використовувати символьні літерали. Правда, вони все рівно інтерпретуються як цілі числа. Розглянемо приклад.

Перерахування символьних літералів

```
#include <iostream.h>

int main()
{
   enum { PLUS = '+', MINUS = '-'};
   cout << PLUS;

   return 0;
}
```

Тут на екран виводиться ASCII-код символу `'+'`. Для того щоб вивести на екран сам символ, буде потрібно виконати приведення типу.

Приведення константи перерахування

```
#include <iostream.h>

int main()
{
   enum { PLUS = '+', MINUS = '-'};
   cout << (char)PLUS;

   return 0;
}
```

Значення лічильника, не задане явно, перевищує значення попереднього лічильника на одиницю. Наприклад, оператор

```
enum { ZERO = 10, ONE, TWO, THREE, FOUR, FIVE};
```

привласнює літералам `ZERO`, `ONE`, `TWO`, `THREE`, `FOUR` і `FIVE` значення 10–15 відповідно.

Елементи перерахування не обов'язково повинні бути послідовними.

```
enum { ZERO = 0, ONE=1, TWO=5, THREE, FOUR=7, FIVE};
```

Перераховані літерали тепер будуть мати наступні значення: `ZERO=0`, `ONE=1`, `TWO=5`, `THREE=6`, `FOUR=7`, `FIVE=8`.

Іменовані перерахування утворять новий інтегральний тип. Наприклад, якщо між ключовим словом `enum` і списком перерахування вказати ім'я `Numbers`, виникне новий тип.

```
enum Numbers {ZERO, ONE, TWO, THREE, FOUR, FIVE}
```

Це значить, що можна оголосити змінну `aNumber`

```
Numbers aNumber;,
```

яка може приймати лише значення 0–5. По-перше, це дозволяє обмежити діапазон припустимих значень змінної і, по-друге, робить програму більш читабельної. Крім того, змінну типу `Numbers` можна визначити одночасно з оголошенням перерахування.

```
enum Numbers {ZERO, ONE, TWO, THREE, FOUR, FIVE} aNumber;
```

При визначенні перерахування можна використовувати не тільки літерали, але й арифметичні вирази.

Арифметичні вираз при визначенні перерахування

```
#include <iostream.h>

int main()
{
   enum { ONE = 1, TEN = ONE + 9, THIRTY_ONE = 3*TEN + ONE};
   cout << ONE << " " << TEN << " " << THIRTY_ONE << endl;

   return 0;
}
```

Ця програма виводить на екран числа `1 10 31`.

Змішане перерахування

```
#include <iostream.h>

int main()
{
   enum { ONE = 1, TWO, TEN = ONE + 9, THIRTY_ONE = 3*TEN + ONE};
   cout << ONE << " " << TWO <<" " << TEN << " " << THIRTY_ONE << endl;

   return 0;
}
```

Ця програма виводить на екран числа `1 2 10 31`.

Область видимості літералів, оголошених усередині списку перерахувань, обмежена блоком, у якому визначене саме перерахування.

Повторне визначення константи перерахування

```
#include <iostream.h>
int main()
{
   enum {ONE, TWO};
   enum {TWO, THREE}; // Помилка! Повторне визначення літерала TWO
   cout << ONE << " " << TWO << endl;

   return 0;
}
```

На жаль, у перерахуванні неможлива зміна кроку збільшення літералів. Він завжди дорівнює одиниці. Якщо все необхідно перенумерувати літерали з другим кроком, це прийдеться зробити явно.

```
enum Numbers {ZERO=0, ONE=10, TWO=20, THREE=30, FOUR=40, FIVE=50}
```

3.3. Масиви

Вище ми описали всі убудовані типи даних, що існують у мові C++. Зрозуміло, у реальних додатках простими змінними обійтися неможливо. Як правило, обробка даних вимагає їхній відповідний організації. Для цього в мові C++ передбачені засоби для створення масивів, структур і об'єднань.

Масив являє собою сукупність однотипних змінних, розташованих у послідовно пронумерованих суміжних комірках пам'яті. Номер елемента масиву задається індексом. Індексація елементів масиву починається з нуля. Найменший індекс відноситься до першого елемента масиву, а найбільший — до останнього.

У залежності від способу зв'язування індексів з комірками пам'яті, масиви розділяються на три категорії: статичний, фіксований автоматичний, динамічний.

Статичним називається масив, у якому зв'язування індексів і розміщення в пам'яті виконуються на етапі компіляції програми. Статичні масиви дуже ефективні, оскільки для їхнього створення і знищення не потрібно додаткових операцій.

Фіксованим автоматичної іменується масив, у якому індекси зв'язуються статично, а розміщення в пам'яті виконується при обробці оголошень усередині функцій.

Динамічним називається масив, де зв'язування індексів і розміщення в пам'яті здійснюються під час виконання програми. Цей спосіб організації даних дуже гнучкий: розміри динамічного масиву можуть збільшуватися і зменшуватися під час виконання програми в міру необхідності.

Крім того, масиви можуть бути одномірними і багатомірними.

3.3..1. Одномірні масиви

Оголошення одномірного масиву виглядає в такий спосіб.

тип ім'я_масиву[*розмір*]

Наприклад, оголошення статичного масиву `numbers`, що має тип `int` і складається з 10 елементів, виглядає так.

```
int numbers[10];
```

Мова C++ містить спеціальний оператор доступу до елемента масиву `[]`. Операндами цього оператора є ім'я масиву й індекс елемента. Наприклад, щоб привласнити першому елементу масиву `numbers` число 1000, необхідно виконати наступний оператор.

```
number[0] = 1000;
```

Варто мати на увазі, що індексація масиву починається з нуля, отже, останнім елементом масиву `numbers` є елемент `numbers[9]`, а не `numbers[10]`.

Обсяг пам'яті, займаний масивом, залежить від його типу і розміру. Він дорівнює розміру типу, помноженому на кількість елементів.

На жаль, у мові C++ не передбачена перевірка виходу індексу масиву за межі припустимого діапазону. Вихід за межі пам'яті, відведеної для масиву, ніяк не контролюється. Отже, під час виконання програми легко помилитися і записати дані в сусідні осередки, знищивши попередню інформацію. Наслідки можуть бути непередбаченими.

Масиви тісно зв'язані з вказівниками. Власне кажучи, ім'я масиву є вказівником на перший його елемент. Інакше кажучи, ім'я масиву `numbers` можна використовувати як базу для зсуву вказівника. Так, вираз

```
numbers[4] = 1;
```

еквівалентно виразу

```
*(numbers+4) = 1;.
```

Зрозуміло, масиви можна розглядати як послідовність осередків, що згодом будуть заповнені корисною інформацією. Однак частіше в процесі роботи передбачається, що масив із самого початку містить деякі дані.

Для того щоб заповнити масив початковими даними, необов'язково виконувати присвоювання в циклі, досить виконати *ініціалізацію* під час оголошення.

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

Слід зазначити, що розмір масиву під час ініціалізації задавати необов'язково. Побачивши рядок

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

компілятор автоматично обчислить розмір масиву `numbers`. Однак ініціалізація вимагає обережності. Достаточно легко зробити наступні типові помилки. По-перше, припустимо, що масив повинний містити 10 цілих чисел, але під час ініціалізації розмір не зазначений і задане менша кількість значень.

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

Зовсім очевидно, що компілятор буде вважати, що масив `numbers` містить 9 чисел. І спроба присвоювання

```
numbers[10]=9;
```

призведе до дуже неприємної ситуації. Справа в тім, що практично всі компілятори не виявляють виниклої помилки. Більш того, під час виконання програми помилка виявляється не відразу. У цьому і полягає підступництво мови C++ при роботі з масивами — немає засобів для виявлення виходу за межі припустимого діапазону. Розроблювачі компіляторів самі вирішують, як надійти в цій ситуації. Одні перекладають проблеми на плечі програміста, а інші створюють код, що при виконанні створює виняткову ситуацію.

По-друге, програміст може помилково задати більше чисел, чим передбачено оголошенням.

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

На щастя, компілятор розпізнає помилки такого роду, видаючи повідомлення: “Занадто багато ініціалізаторів”.

Крім того, до масиву можна застосовувати оператор `sizeof`. Результатом цієї операції є ціле число, що представляє собою кількість байтів, займаних масивом.

```
int numbers[10] = {0, 1 ,2 3, 4, 5, 6, 7, 8, 9};
cout << sizeof(numbers);
```

Цей фрагмент програми виводить на екран число 40.

Якщо масив не підлягає змінам, його можна оголосити константним. Для цього перед його оголошенням варто помістити ключове слово `const`.

```
const int numbers[] = {0, 1 ,2 3, 4, 5, 6, 7, 8, 9};
```

3.3.2. Рядки

У мові C++ передбачено два види рядків: *рядка, що завершуються нульовим байтом* (вони являють собою одномірні масиви символів, що завершуються нульовим байтом), і клас `string`. Рядка, що завершуються нулем, іноді позначають абревіатурою NBTS — Null Byte Terminated String.

Якщо масив використовується як рядок, його розмір повинний на одиницю перевищувати кількість символів — з урахуванням нульового байта. Наприклад, масив `symbols`, що представляє собою рядок з 33 символів, повинний з'являтися в такий спосіб.

```
char symbols[33];
```

Однак оголошений у такий спосіб масив ще не є рядком — адже в його останньому осередку немає нульового байта. Щоб записати символи в масив, існує декілька можливостей. Одна з найбільш розповсюджених — присвоєння *строкової константи*.

```
char symbols[] = "Строкова константа";
```

Відзначимо декілька особливостей такого оголошення, що супроводжується ініціалізацією масиву. По-перше, розмір масиву `symbols` визначається автоматично. Він дорівнює кількості символів у рядку “Строкова константа” плюс одиниця. По-друге, нульовий байт автоматично додається компілятором.

Конечно, масив символів можна ініціалізувати як звичайний масив.

```
char symbols[]={'a','b','c','d','e','f','g','h','i','j','\0'};
```

Зверніть увагу на те, що тепер програміст сам повинний подбати про завершення рядка, помістивши в останню позицію символ `'\0'`. Якщо цього не зробити, працювати з масивом символів як з рядком стане неможливо.

У мові C++ немає операторів для роботи з рядками. Всі операції над рядками виконуються за допомогою функцій, оголошених у стандартному заголовному файлі `string.h`.

Прототип функції	Призначення
`char* strcat(char* dest, const char* source);`	Приписує рядок `source` до кінця рядка `dest`
`char* strncat(char* dest, const char* source, unsigned n);`	Приписує `n` символів рядка `source` до кінця рядка `dest`
`char* strchr(char* s, int ch);` `const char* strchr(const char* s, int ch);`	Знаходить перше входження символу `ch` у рядок `s`
`int strcmp(const char* s1, const char* s2);`	Порівнює рядка. Повертає 0, якщо `s1==s2`; `-1` — якщо `s1<s2`; і `1` — якщо `s1>s2`

`int strncmp(const char* s1, const char* s2, int n);`	Порівнює перші `n` символів рядків `s1` і `s2`. Повертає 0, якщо `s1==s2`; `-1` — якщо `s1<s2`; і `1` — якщо `s1>s2`
`int stricmp(const char* s1, const char* s2);`	Порівнює рядка, ігноруючи регістр букв. Повертає 0, якщо `s1==s2`; `-1` — якщо `s1<s2`; і `1` — якщо `s1>s2`
`int strnicmp(const char* s1, const char* s2, int n);`	Порівнює перші `n` символів рядків `s1` і `s2`, ігноруючи регістр букв. Повертає 0, якщо `s1==s2`; `-1` — якщо `s1<s2`; і `1` — якщо `s1>s2`
`char* strcpy(char* dest, const char* source);`	Копіює рядок `source` у рядок `dest`
`char* strncpy(char* dest, const char* source, int n);`	Копіює перші `n` символів рядка `source` у рядок `dest`
`size_t strlen(const char* s);`	Обчислює кількість символів, що входять у рядок, без обліку нульового символу наприкінці
`char* strlwr(char* s);`	Переводить рядок `s` у нижній регістр
`char* strupr(char* s)`	Переводить рядок `s` у верхній регістр
`char* strdup(const char* s);`	Виділяє пам'ять для рядка `s`
`char* strset(char* s, int ch);`	Заповнює рядок `s` символом `ch`
`char* strnset(char* s, int ch, int n);`	Заміняє перші `n` символів рядка `s` символом `ch`
`char* strrev(char* s);`	Переставляє символи рядка `s` у зворотному порядку
`size_t strcspn(const char* s1, const char* s2);`	Повертає довжину початкового відрізка рядка `s1`, що складає только із символів, що не входять у рядок `s2`
`const char* strpbrk(const char* s1, const char* s2);`	Переглядає рядок `s1`, поки не знайде символ, що належить рядку `s2`
`char* strrchr(char* s, int c);` `const char* strrchr(const char* s, int c);`	Переглядає рядок `s1`, поки не знайде останній символ, заданий молодшим байтом параметра `c`
`size_t strspn(const char* s1, const char* s2);`	Повертає довжину початкового відрізка рядка `s1`, що складає только із символів, що входять у рядок `s2`
`char* strtok(char* s1, const char* s2);`	Виділяє фрагменти рядка, використовуючи роздільники. При послідовних викликах функція повертає вказівник на перший символ кожного фрагмента

3.3..3.3.Двомірні масиви

Мови програмування призначені для збереження й обробки інформації. Один з найбільш розповсюджених способів збереження інформації — використання таблиці, чи матриці. Для їхнього представлення в мові C++ передбачені багатомірні масиви.

Розглянемо для початку найпоширеніший варіант — двомірний масив, якому можна представити у виді масиву одномірних масивів. Його оголошення виглядає так.

тип ім'я_масиву[*розмір1*][*розмір2*]

У C++ прийняте наступне звертання до елементів двомірного масиву:

ім'я_масиву[*індекс1*][*індекс2*].

Ініціалізація двомірного масиву виконується по рядкам.

```
int numbers[3][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
```

Як бачимо, таке оголошення не занадто наочне. Для того щоб спростити представлення матриці, можна здійснити угруповання даних.

```
int numbers[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
```

Варто мати на увазі, що будь-який масив зберігається в послідовних комірках пам'яті. Таким чином, завжди існує можливість представити двомірний масив у виді одномірного, скориставшись формулою для обчислення індексів. Отже, допустимо, що індекс `k` — це індекс одномірного масиву, що має розмір `n`, а індекси `i` і `j` задають положення елемента двомірного масиву. Тоді значення елемента `numbers[i][j]` можна обчислити за допомогою вираз `*(numbers+i*n+j)`. Крім того, результатом вираз `*(numbers[i])` є перший елемент `i`-го рядка. Отже, значення елемента `numbers[i][j]` можна обчислити і так: `*(numbers[i]+j)`.

3.3.4. Вільні масиви

Як ми вже бачили, поняття масиву і вказівника тісно зв'язані, оскільки ім'я будь-якого масиву є ім'ям вказівника на його перший елемент. Отже, індексація елементів масиву здійснюється на основі адресної арифметики.

Однак зв'язок масивів і вказівників цим не обмежується. Представимо, що в осередках масиву зберігаються не змінні одного з числових типів, а вказівники. Це відкриває можливість створювати масиви, що, будучи одномірними, насправді є двомірними, причому друга розмірність такого масиву є змінної. Такі масиви називаються *вільними*.

Найбільше широко розповсюдженою реалізацією цієї можливості є масиви рядків. Розглянемо відповідну демонстраційну програму.

Статичний вільний масив рядків

```
#include <iostream.h>
#include <string.h>

int main()
{
   char *strings[] = { "Перший рядок",
                       "Другий рядок",
                       "Третій рядок",
                       "Четвертий рядок",
                       "П'ятий рядок"};
  int size = 2;
  for(int i = 0; i<size; i++)
  {
    cout << strings[i] << endl;
    for (int j =0; j<strlen(strings[i]); j++)
    cout << strings[i][j] << " ";
    cout << endl;
  }

  return 0;
}
```

Ця програма спочатку виводить на екран первісний уміст вільного масиву `strings`, а потім роздруковує поточну рядок, переміжаючи її символи пробілами.

```
Перший рядок
П е р ш и й  р я д о к
 Другий рядок
Д р у г и й  р я д о к
```

У той час як масив адресується вказівниками на його елементи, переміщення по масиві вказівників здійснюється за допомогою вказівників на вказівники. Такий спосіб називається непрямою *адресацією*. Таким чином, програму можна переписати в такий спосіб (тепер масив вказівників є не статичним, а динамічним).

Динамічний вільний масив рядків

```
 #include <iostream.h>
#include <string.h>

int main()
{
   int size = 5;
   char **strings = new char* [size];
   strings[0]= "Перший рядок";
   strings[1]= "Другий рядок";
   strings[2]= "Треться рядок";
   strings[3]= "Четвертий рядок";
   strings[4]= "П'ятий рядок";

   for(int i = 0; i<size; i++)
   {
       cout << strings[i] << endl;
       for (int j =0; j<strlen(strings[i]); j++)
       cout << strings[i][j] << " ";
       cout << endl;
   }
```

```
    return 0;
}
```

Вільні масиви не обов'язково складаються з рядків. Як приклад розглянемо програму, у якій обробляється матриця, що складається з цілих чисел.

Вільний масив цілих чисел

```
#include <iostream.h>

int size = 3;
int** numbers = new int*[size];

int main()
{
   int val;

   for(int i=0; i<size; i++)
   {
       numbers[i]= new int[size];
       for (int j=0; j<size; j++)
       {
           cin >> val;
           *(numbers[i]+j)=val;
       }
   }


   for(i=0; i<size; i++)
   {
       for (int j=0; j<size; j++)
           cout << numbers[i][j];
       cout << endl;
   }
   return 0;
}
```

Варто мати на увазі, що еквівалентом вираз `numbers[i][j]` є вираз `*(*(numbers+i)+j)`, а не `**(numbers+size*i+j)` чи яке-небудь інше. Інакше кажучи, у масиві `numbers` зберігаються лише вказівники на початок одномірних масивів цілих чисел, і лише через ці вказівники можна одержати їхні конкретні значення. Непряма адресація не забезпечує прямого доступу до елементів одномірних масивів.

Основною перевагою вільних масивів є ефективне використання пам'яті: незважаючи на те що довжина рядків вільного двомірного масиву варіюється, жодного зайвого осередку не виділяється.

3.4. Структури POD

Структура POD (Plain Old Data) являє собою сукупність різнотипним змінним, об'єднаним загальним ім'ям. *Оголошення структури* є схемою, по якій створюється її екземпляр. Змінну, утворюючу структуру, називаються її *дан-членами*. Ця термінологія не зовсім зручна (спробуйте провідміняти — дані-члени, даних-членів і тощо!), тому надалі ми будемо використовувати терміни чи поле *елемент* структури.

Поняття структури тісно зв'язано з основним поняттям об’єктно-орієнтованого програмування — *класом*. У відповідності зі стандартом мови C++ структура може складатися не тільки з полів, але і з функцій-членів. Таким чином, структура — це найпростіший клас, усі поля і функції-члени якого за замовчуванням відкриті. Однак оскільки мова C++ є багатошаровим і поєднує в собі декілька парадигм програмування — процедурну, об'єктно-орієнтовану й узагальнену — прийнято розрізняти так називані структури POD і класи. З цієї причини ми не будемо зараз аналізувати об’єктно-орієнтовані властивості структур і зосередимося на її властивостях, успадкованих від мови C.

Логічна схема, покладена в основу структури, цілком визначається програмістом. Як правило, у структуру включаються взаємозалежні дані. Наприклад, структури широко застосовуються для представлення часу і дати (день, місяць, рік, година, хвилини, секунди).

Оголошення структури виглядає в такий спосіб.

```
struct ім'я_структури
{
 тип1 поле, ..., поле;
 тип2 поле, ..., поле;
 .
 .
 .
```

```
   тип поле, ..., поле;
  } імена_екземплярів;
```

Зверніть увагу на декілька моментів. По-перше, на відміну від мови С, у С++ однотипні поля можна повідомляти в одному рядку. По-друге, імена полів можуть збігатися з іменами інших змінних, визначених поза структурою, — імена полів завжди уточнюються ім'ям структури. По-третє, після закриваючої фігурної дужки можна визначити один чи декілька екземплярів. Якщо цього не зробити, оголошення структури залишиться лише логічною схемою і пам’ять виділена не буде. У цьому випадку визначення екземпляра виконується окремо. По-четверте, навіть якщо жодний екземпляр структури не визначений, після оголошення обов'язково варто поставити точку з коми — адже оголошення структури є оператором.

Наприклад, структура, що зберігає дату і час, може бути оголошена в такий спосіб.

```
 struct TimeData
{
 unsigned int day;     // 01...31
 char month;           // 1...12
 unsigned int year;    // 0-65535
 unsigned int hour;    // 00-24
 unsigned int minute;  // 00-60
 float second;         // 00.00 - 60.00
}A,B,C;
```

У даному випадку ім'ям структури є ідентифікатор `TimeData`, а іменами її екземплярів — букви `A`, `B` і `C`. Відкладене визначення екземплярів структури виглядає так.

```
 struct TimeData
{
  unsigned int day;     // 01 ... 31
  char month;           // 1 ... 12
  unsigned int year;    // 0-65536
  unsigned int hour;    // 00-24
  unsigned int minute;  // 00-60
  float second;         // 00.00 - 60.00
};
...
 TimeData A,B,C;
```

Якщо в програмі створюється лише один екземпляр структури, її ім'я після ключового слова `struct` указувати не обов’язково.

```
 struct
 {
  unsigned int day;     // 01...31
  char month;           // 1...12
  unsigned int year;    // 0-65536
  unsigned int hour;    // 00-24
  unsigned int minute;  // 00-60
  float second;         // 00.00 - 60.00
 }unique;
```

Доступ до елементів екземпляра структури здійснюється за допомогою оператора `.` (“точка”).

```
A.day = 10;
A.month = 1;
A.year = 2003;
unique.second = 10.35;
```

3.4.1. Операції над структурами

Перш ніж використовувати екземпляр структури в програмі, його випливає ініціалізувати. Зробити це безпосередньо в оголошенні неможливо, оскільки оголошення структури являє собою лише абстрактний опис схеми структури і не зв'язано з виділенням конкретної пам'яті для її полів. Отже, початкові значення полів просто нікуди записати. Ініціалізація можлива лише при визначенні екземпляра структури.

Инциализация структури: перший варіант

```
#include <iostream.h>

int main()
{
   struct Complex
   {
      double Re, Im;
   } A = {0, 1};
   cout << A.Re << endl << A.Im;
```

```
   return 0;
}
```

Можливий також інший варіант.

Ініціалізація структури: другий варіант

```
#include <iostream.h>

int main()
{
   struct Complex
   {
      double Re, Im;
   } A;
   Complex A = {0, 1};
   cout << A.Re << endl << A.Im;
   return 0;
}
```

Структури можна привласнювати один одному. Присвоювання можливе лише в тому випадку, коли структури мають однаковий тип. Розглянемо приклад, що ілюструє присвоювання комплексних чисел.

Присвоювання структур

```
#include <iostream.h>

int main()
{
   struct Complex
   {
      double Re, Im;
   } A = {0,1}, B={2,3};
   cout << A.Re << endl << A.Im;
   cout << B.Re << endl << B.Im;
   B = A;
   cout << B.Re << endl << B.Im;
   return 0;
}
```

У підсумку, екземпляр структури B буде зовсім ідентичний екземпляру A.

Якщо над структурами необхідно виконати більш складну операцію, наприклад порівняти два екземпляри, програміст повинний виразити цю операцію через операції над полями. Наприклад, операцію екземплярів на рівність можна виразити за допомогою операторів порівняння перевірки їхніх полів.

```
if((X.Re == Y.Re)&& (X.Im == X.Im)) cout << "Екземпляри збігаються";
```

3.4.2. Масиви і структури

З одного боку, структури можуть бути елементами масивів, а з інший, — масиви можуть бути членами структур.

Визначення масиву структур нічим не відрізняється від визначення масиву убудованого типу. Природно, оголошення (логічна схема) структури повинне передувати її визначенню.

Масив структур

```
#include <iostream.h>

int main()
{
   struct Complex
   {
      double Re, Im;
   };
   Complex A[100];
   return 0;
}
```

Для доступу до екземпляра структури, що є елементом масиву, використовується її індекс. Наприклад, щоб вивести на екран дійсну і уявну частини 50-го елемента масиву A, можна виконати наступну інструкцію.

```
cout << A[49].Re << "+ i*" << A[49].Im;
```

Зверніть увагу на те, що індекс 50-го елемента масиву структур, як і будь-якого іншого масиву, дорівнює 49, оскільки нумерація починається з нуля.

Якщо структура містить масив, її оголошення виглядає в такий спосіб.

```
struct Record
 {
```

```
    char array[30];
    double b;
};
```

Звертання до 50-му елемента масиву `array` виглядає так.

```
Record.array[49];
```

3.4.3. Вкладені структури

Структура може бути елементом структури іншого типу. При цьому вкладення структур однакового типу не допускається. Наприклад, структуру, що містить дані про студента, можна представити в наступному виді.

Вкладені структури

```
#include <iostream.h>

int main()
{
   // Дата надходження
   struct Date
   {
     unsigned int day;
     unsigned int month;
     unsigned int year;
   };

   // Персональні дані
   struct Personal
   {
     char* surname;
     char* name;
     double score;
   };
   struct Record
   {
     struct Date a;
     struct Personal b;
   } record={{1,9,2003}, {"Іванов","Іван",4.8}};
   cout << record.a.day     << "."
        << record.a.month   << "."
        << record.a.year    << " "
        << record.b.surname << " "
        << record.b.name    << ": "
        << record.b.score;

   return 0;
}
```

На екран виводиться наступна рядок.

```
1.9.2003 Іванов Іван: 4.8
```

Зверніть увагу на те, що ініціалізувати екземпляр структури `Record` екземплярами структур `Date` і `Personal` неможливо. Таким чином, ініціалізація наступного виду неможлива.

```
struct Data
 {
   unsigned int day;
   unsigned int month;
   unsigned int year;
 }data = {1, 9, 2003};
 struct Personal
 {
   char* surname;
   char* name;
   double score;
 }personal = {"Іванов", "Іван", 4.8};
 struct Record
 {
   struct Data a;
   struct Time b;
 } record={Date, Personal}}; // Помилка
```

3.4.4. Вказівники на структури

Екземпляр структури є повноцінної змінний. Отже, на нього можна встановлювати вказівники. Однак вказівники на структури декілька відрізняються від звичайних.

По-перше, щоб оголосити вказівник на структуру, необхідно поставити символ * перед ім'ям екземпляра структури.

```
Complex *Z;
```

Оскільки вказівник повинний містити адресу змінної, для посилання на екземпляр структури необхідно обчислити адресу цього екземпляра. Для цього використовується оператор узяття адреси &. Доступ до поля структури за допомогою вказівника забезпечується оператором -> (“стрілка”).

Посилання на поле структури

```
#include <iostream.h>

int main()
{
   struct Complex
   {
      double Re, Im;
   }Z;
   Complex *p;
   p = &Z;
   p->Re = 0;
   p->Im = 1;
   cout << p->Re << endl << p->Im;

   return 0;
}
```

Можливі і більш складні варіанти, коли доступ до полів структури забезпечується як через вказівник, так і безпосередньо.

Розглянемо наступну програму, що ілюструє доступ до вкладених структур за допомогою вказівників, а до їх полів — безпосередньо.

Змішаний доступ до полів структури

```
#include <iostream.h>
#include <malloc.h>

int main()
{
   struct Date
   {
     unsigned int day;
     unsigned int month;
     unsigned int year;
   };
   struct Personal
   {
     char* surname;
     char* name;
     double score;
   };
   struct Record
   {
     struct Date a;
     struct Personal b;
   };

   Record* pRecord = (Record*) malloc(sizeof(Record));

   pRecord->a.day   = 1;
   pRecord->a.month = 4;
   pRecord->a.year  = 2003;
   pRecord->b.surname  = "Іванов";
   pRecord->b.name   = "Іван";
   pRecord->b.score   = 4.8;

   cout << pRecord->a.day     << "/"
```

```
          << pRecord->a.month   << "/"
          << pRecord->a.year    << " "
          << pRecord->b.surname << " "
          << pRecord->b.name    << ": "
          << pRecord->b.score;

    return 0;
}
```

На екран будуть виведені дата і час.

```
1.9.2003 Іванов Іван: 4.8
```

3.5. Бітові поля

Мовам програмування властива деяка марнотратність при роботі з цілими числами. Оскільки мінімальний розмір адресованої одиниці пам'яті дорівнює 8 байт, при роботі з цифрами велика кількість бітів дорівнює нулю і є не інформативним. Для подолання цього недоліку в мові C++ передбачена підтримка *бітових полів*. Сфера їхнього застосування досить велика: при роботі з логічними змінними `true` і `false`, при роботі з індикаторами стану фізичних пристроїв (портів і т.п.) і при шифруванні.

В основу бітових полів покладене поняття структури. Бітове поле є різновидом елемента структури, розмір якого можна задати в бітах. Визначення бітового поля має наступний вид.

```
struct ім'я_структури
{
  тип ім'я1:кількість_біт;
  тип ім'я2:кількість_біт;
.
.
.
  тип имя:кількість_біт;
} список_змінних;
```

Членами бітового поля можуть служити лише змінні типів `char`, `int` з відповідними модифікаціями. Якщо розмір бітового поля дорівнює одиниці, його тип можна вказати за допомогою специфікатора `unsigned`. Наприклад, бітове поле `ftime`, визначене в заголовному файлі `io.h`, виглядає в такий спосіб.

```
struct  ftime
{
    unsigned    ft_tsec  : 5;   /* Двосекундний інтервал */
    unsigned    ft_min   : 6;   /* Хвилини */
    unsigned    ft_hour  : 5;   /* Години */
    unsigned    ft_day   : 5;   /* Дні */
    unsigned    ft_month : 4;   /* Місяці */
    unsigned    ft_year  : 7;   /* Роки */
};
```

Оскільки бітове поле, власне кажучи, є різновидом структури, при роботі з ним використовуються ті ж оператори: `.` (“точка”) і `->` (“стрілка”). Наприклад, щоб змінити установку часу, можна виконати наступні оператори.

```
ftime.ft_min = 30;
ftime.ft_hour = 12;
```

Якщо на бітове поле встановлений вказівник, для доступу до його елементів застосовується оператор `->`.

Елементи бітового поля іменувати не обов’язково. Це дає можливість виділяти інформативні біти й ігнорувати непотрібні. Наприклад, у попереднім бітовому полі можна “відключити” елементи, що зберігають інформацію про час.

```
struct  ftime
{
    unsigned    : 5;            /* Двосекундний інтервал */
    unsigned    : 6;            /* Хвилини */
    unsigned    : 5;            /* Години */
    unsigned    ft_day   : 5;   /* Дні */
    unsigned    ft_month : 4;   /* Місяці */
    unsigned    ft_year  : 7;   /* Роки */
};
```

Якщо потрібно відключити біти, що випливають після інформації про час, їх можна просто не вказувати.

```
struct  ftime
{
    unsigned    ft_tsec  : 5;   /* Двосекундний інтервал */
    unsigned    ft_min   : 6;   /* Хвилини */
```

```
    unsigned    ft_hour  : 5;   /* Години */
};
```

Бітові поля можуть бути елементами звичайних структур.

```
struct Data
 {
   unsigned int day   : 5;
   unsigned int month : 4;
   unsigned int year  : 7;
 };
 struct Time
 {
   unsigned int hour  : 5;
   unsigned int min   : 6;
   unsigned int sec   : 5;
 };
 struct TimeData
 {
   struct Data a;
   struct Time b;
 };
```

Крім того, у звичайній структурі можна використовувати поля з зазначеними розмірами.

```
struct TimeData
{
  unsigned int day    : 5
  char month          : 4;          // 1 ... 12
  unsigned int year   : 7;
  unsigned int hour   : 5;
  unsigned int minute : 6;
  float second;
};
```

У 32-розрядній операційній системі розмір цієї структури дорівнює 16 байт. Якби бітові полючи не використовувалися, її розмір був би дорівнює 24 байт.

Бітові полючи є машинно-орієнтованими. Отже, їхнє застосування супроводжується певними обмеженнями. Перелічимо деякі з них.

1. Адреса бітового полючи одержати неможливо.
2. Бітові поля не можуть бути елементами масиву.
3. Бітові поля не можуть бути статичними.

3.6. Об'єднання

Об'єднання — це область пам'яті, у якій у різний час можуть зберігатися змінні різних типів. На відміну від структури, елементи якої записуються послідовно друг за другом, елементи об'єднання перекривають один одного в одній і тій же області пам'яті. Таким чином, у той час як розмір структури дорівнює сумі розмірів її елементів, розмір об'єднання дорівнює максимальному розміру серед усіх його елементів.

Оголошення об'єднання виглядає в такий спосіб.

```
union тип_об'єднання
{
  тип ім'я_члена1;
  тип ім'я_члена2;
  .
  .
  .
  тип ім'я_члена;
} імена_екземплярів_об'єднання;
```

Розглянемо приклад. У ньому об'єднання містить дві структури, перші елементи яких перекривають один одного, утворити “ярлички”.

Об'єднання з “ярличками”

```
#include <iostream.h>
#define Check();    (var.tag == 0)?cout << var.First.i << endl\
                   :cout << var.Second.f << endl;

int main()
{
   union Combine
```

```
    {
       int tag:1;
       struct
       {
          int tag: 1;
          int i;
       }First;
       struct
       {
          int tag: 1;
          double f;
       }Second;
     };

     Combine var;

     var.tag = 0;
     var.First.i = 5;

     Check();

     var.tag=1;
     var.Second.f = 50.5;

     Check();
     return 0;
}
```

Оскільки в різні моменти часу на тому самому місці можуть виявитися різні структури, их якось потрібно розрізняти. Для цього досить привласнити першим елементам цих структур різні значення. Програмісту досить перевірити перший елемент, щоб визначити, що випливає за ним — ціле чи дійсне число.

Як і структури, об'єднання не обов'язково іменувати. У цьому випадку компілятор сам розпізнає, що елементи об'єднання зберігаються в одній і тій же області пам'яті, але екземпляр створювати не стане. Таким чином, оператор доступ (“точка”) стає зайвим. Наприклад, у попередньому випадку ми створили тільки один екземпляр структури типу `Combine` — змінну `var`, хоча можна було б обійтися і без цього.

Безіменне об'єднання

```
#include <iostream.h>
#define Check();    (var.tag == 0)?cout << var.First.i << endl\
                   :cout << var.Second.f << endl;


int main()
{
   union
   {
      int tag:1;
      struct
      {
         int tag: 1;
         int i;
      }First;
      struct
      {
         int tag: 1;
         double f;
      }Second;
    }var;

    var.tag = 0;
    var.First.i = 5;

    Check();

    var.tag=1;
    var.Second.f = 50.5;

    Check();
```

```
    return 0;
}
```

В іншому безіменні об'єднання нічим не відрізняються від іменованих.

3.7. Резюме

- Оператор `typedef` дозволяє перейменувати один з існуючих типів даних. Його першим операндом є тип, що переіменовується, а другим його нове ім'я.
- Оператор дозволяє `sizeof` обчислити розмір змінної чи типу в байтах. Операндом цього оператора може бути ім'я типу, змінна чи вираз.
- *Перерахування* — це визначений користувачем тип, що дозволяє іменувати літерали інтегральних типів, тобто символьні і цілочисельні. За замовчуванням значення, привласнені константам перерахування, починаються з нуля і послідовно збільшуються на одиницю. Елементам перерахування можна привласнити будь-яке припустиме інтегральне значення, не обов'язкове ціле, але вони все рівно інтерпретуються як цілі числа.
- *Масив* являє собою сукупність однотипних змінних, розташованих у послідовно пронумерованих суміжних комірках пам'яті. Номер елемента масиву задається індексом. Індексація елементів масиву починається з нуля. Найменший індекс відноситься до першого елемента масиву, а найбільший — до останнього.
- *Статичним* називається масив, у якому зв'язування індексів і розміщення в пам'яті виконуються на етапі компіляції програми.
- *Фіксованим автоматичної* іменується масив, у якому індекси зв'язуються статично, а розміщення в пам'яті виконується при обробці оголошень усередині функцій.
- *Динамічним* називається масив, де зв'язування індексів і розміщення в пам'яті здійснюються під час виконання програми.
- У мові C++ передбачено два види рядків: *рядка, що завершуються нульовим байтом* (вони являють собою одномірні масиви символів, що завершуються нульовим байтом), і клас `string`.
- *Структура POD* (Plain Old Data) являє собою сукупність різнотипним змінним, об'єднаним загальним ім'ям. *Оголошення структури* є схемою, по якій створюється її екземпляр. Змінну, утворюючу структуру, називаються її *даним-членом*.
- В основу бітових полів покладене поняття структури. Бітове поле є різновидом елемента структури, розмір якого можна задати в бітах.
- Членами бітового поля можуть служити лише змінні типів `char`, `int` з відповідними модифікаціями. Якщо розмір бітового поля дорівнює одиниці, його тип можна вказати за допомогою специфікатора `unsigned`. Адреса бітового полючи одержати неможливо. Бітові поля не можуть бути елементами масиву. Бітові поля не можуть бути статичними.
- *Об'єднання* — це область пам'яті, у якій у різний час можуть зберігатися змінні різних типів. На відміну від структури, елементи якої записуються послідовно друг за другом, елементи об'єднання перекривають один одного в одній і тій же області пам'яті. Таким чином, у той час як розмір структури дорівнює сумі розмірів її елементів, розмір об'єднання дорівнює максимальному розміру серед усіх його елементів.

3.8. Контрольні питання

1. Що таке оператор `typedef`?
2. Що таке оператор `sizeof`?
3. Що таке перерахування?
4. Що таке масив?
5. Які масиви називаються статичними?
6. Які масиви називаються автоматичними?
7. Які масиви називаються динамічними?
8. Які два види рядків існують в мові C++?
9. Що таке структура POD?
10. Що таке бітові поля?
11. Які змінні можуть бути членами бітових полів?
12. Що таке об'єднання?