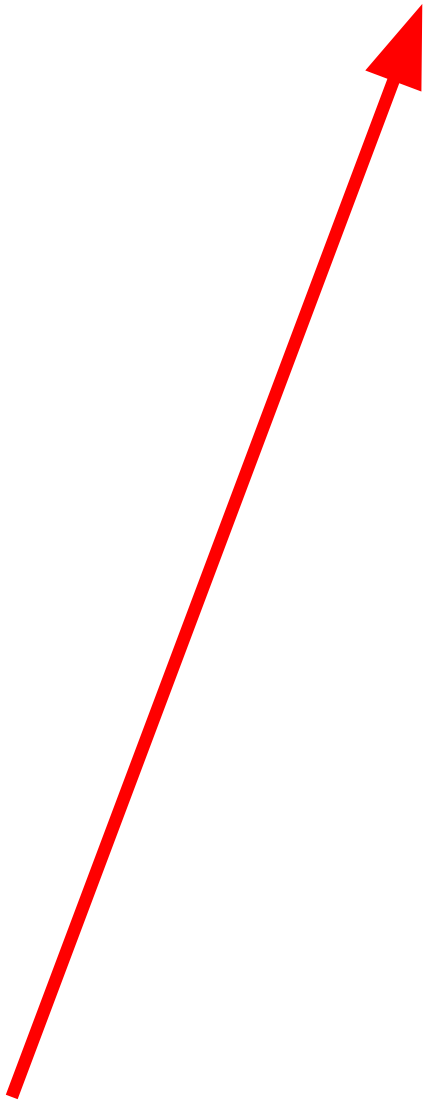
Алгоритми сортування

Алгоритм сортування - це алгоритм для упорядкування елементів в списку.

Сортування з пошуком мінімального (максимального) значення

```
int N = StrToInt(Edit1->Text);
 int *M = new int [N];
 int *MARK = new int [N];
 int *SORT = new int [N];
 Memo1->Clear();
 for(int i = 0; i < N; i++){
       M[i] = random(100);
       Memo1->Lines->Add(IntToStr(M[i]));
 }

 int min;
 bool f;
 for(int i = 0; i < N; i++){

       //Get min
       min = 1000;
       for(int j = 0; j < N; j++){
         f = 1;
         for(int k = 0; k < i; k++){
               if(MARK[k] == j) f = 0;
         }

         if(f && M[j] < min){
```

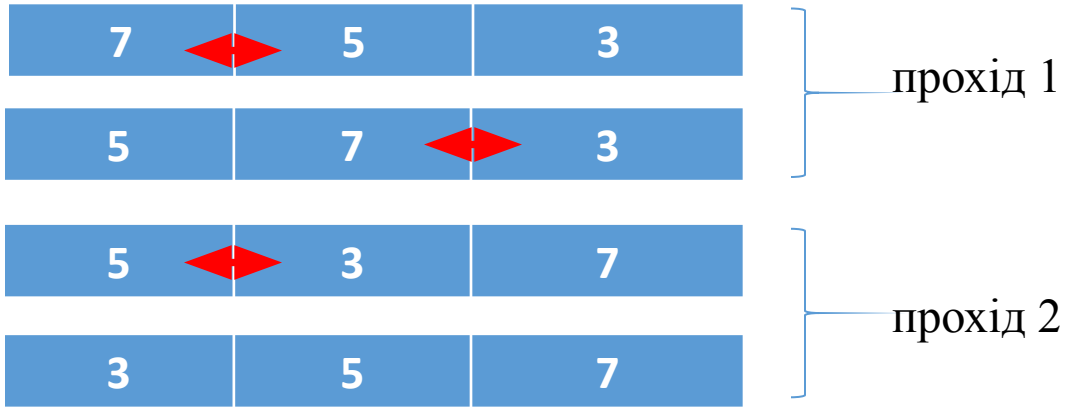


Алгоритм 1

Сортування перестановкою сусідніх елементів

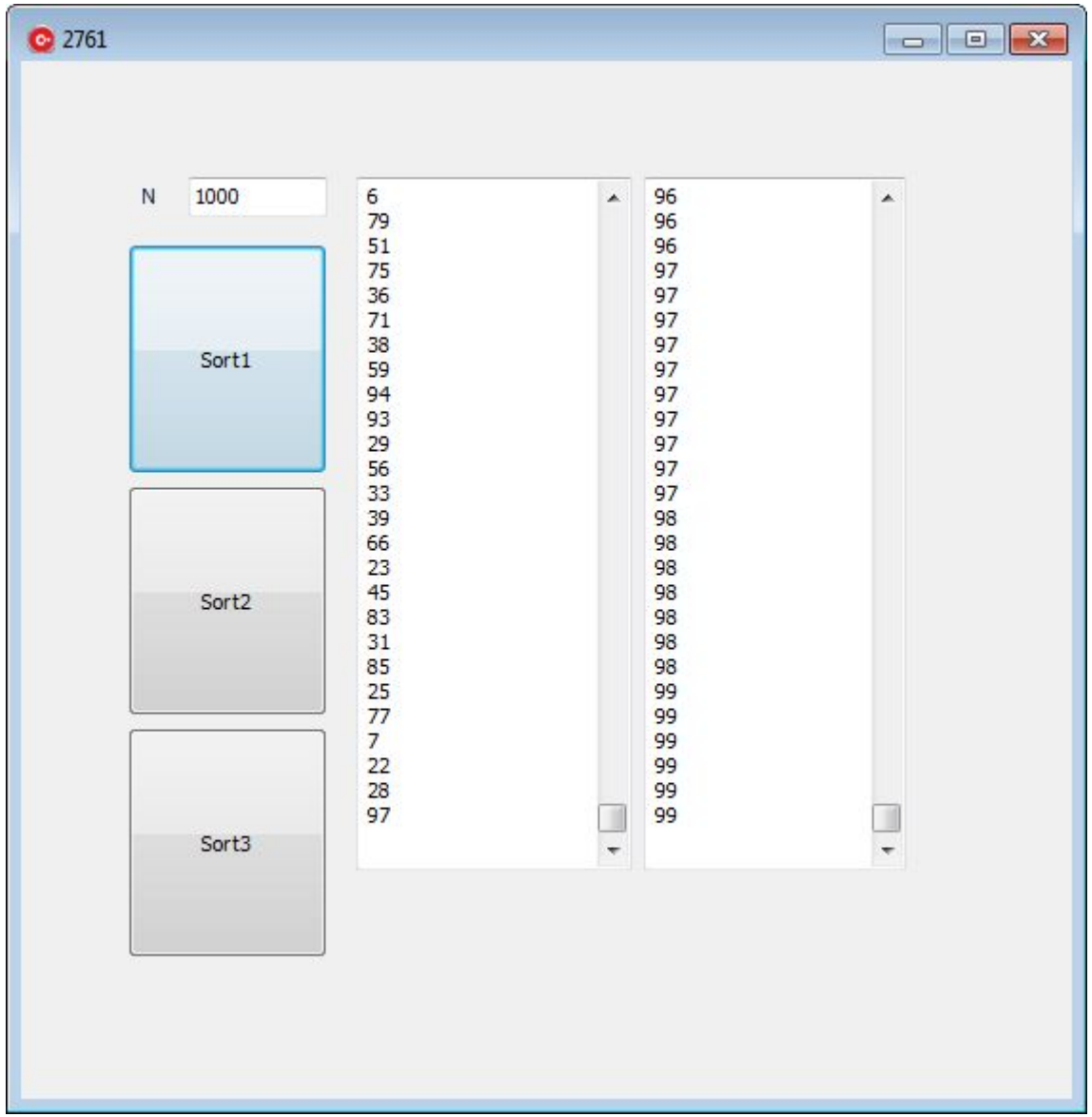
```
int N= StrToInt(Edit1->Text);
int *M = new int [N];

Memo1->Clear();
for(int i = 0; i < N; i++){
      M[i] = random(100);
      Memo1->Lines->Add(IntToStr(M[i]));
}



int  t;
for(int j = 0; j < N-1; j++){
   for(int i = 0; i < N-1; i++){
        if(M[i] > M[i+1]){
             t  = M[i];
             M[i] =  M[i+1];
             M[i+1] = t;
        }
   }
}
delete[] M;
```

Якщо наступне значення менше попереднього, то робиться перестановка

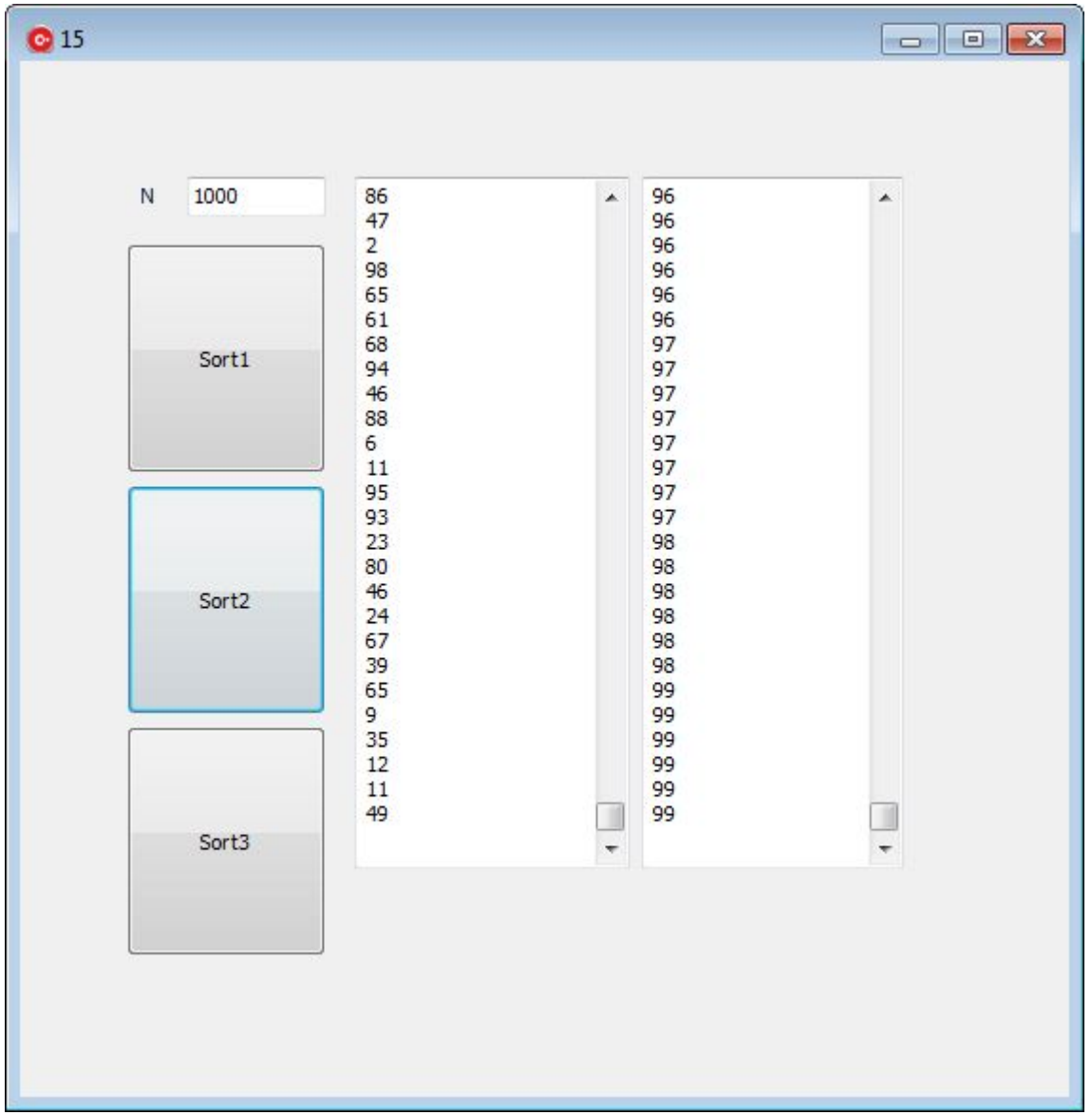


Алгоритм 2

Результат виконання програми



2,761



0,015

Тема 6. Структури.

Об’ява структури. Об’єднання. Бітові поля. Змінні із змінною структурою.

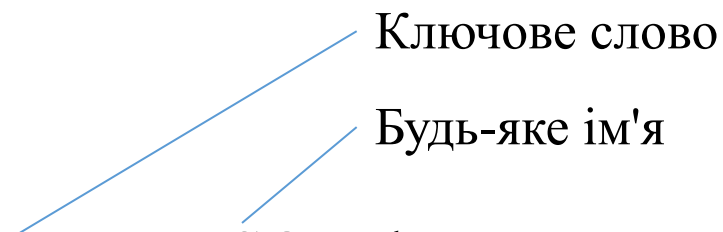
Оператори циклу з предумовою

Структури - це складений об'єкт, в який входять елементи будь-яких типів, за винятком функцій. На відміну від масиву, який є однорідним об'єктом, структура може бути неоднорідною.
Тип структури визначається записом виду:

struct {
  список визначень
}

У структурі обов'язково повинен бути зазначений хоча б один компонент.

Приклад




```
struct PERSON {
  char name[25]; // Ім'я
  int age;       // Вік
  float weight;     // Вага
  float tall;          //Зріст
  ……….
  ……….

} member1; // Обява структури
```

```
struct PERSON {
  char name[25]; // Ім'я
  int age;       // Вік
  float weight;     // Вага
  float tall;          //Зріст
  ……….
};
……..
```

Десь у програмі

```
PERSON member1; // Створення екземпляру
```

Або так…

```
PERSON base[10]; // Створення масиву з 10 екземплярів струтури
```

Звертання до елементів структури

member1.weight = 72,5 //запис параметра

float m = member1.weight //зчитування параметра

Якщо обявлено масив структур

member[i].weight = 72,5 //запис параметра

float m = member[i].weight //зчитування параметра

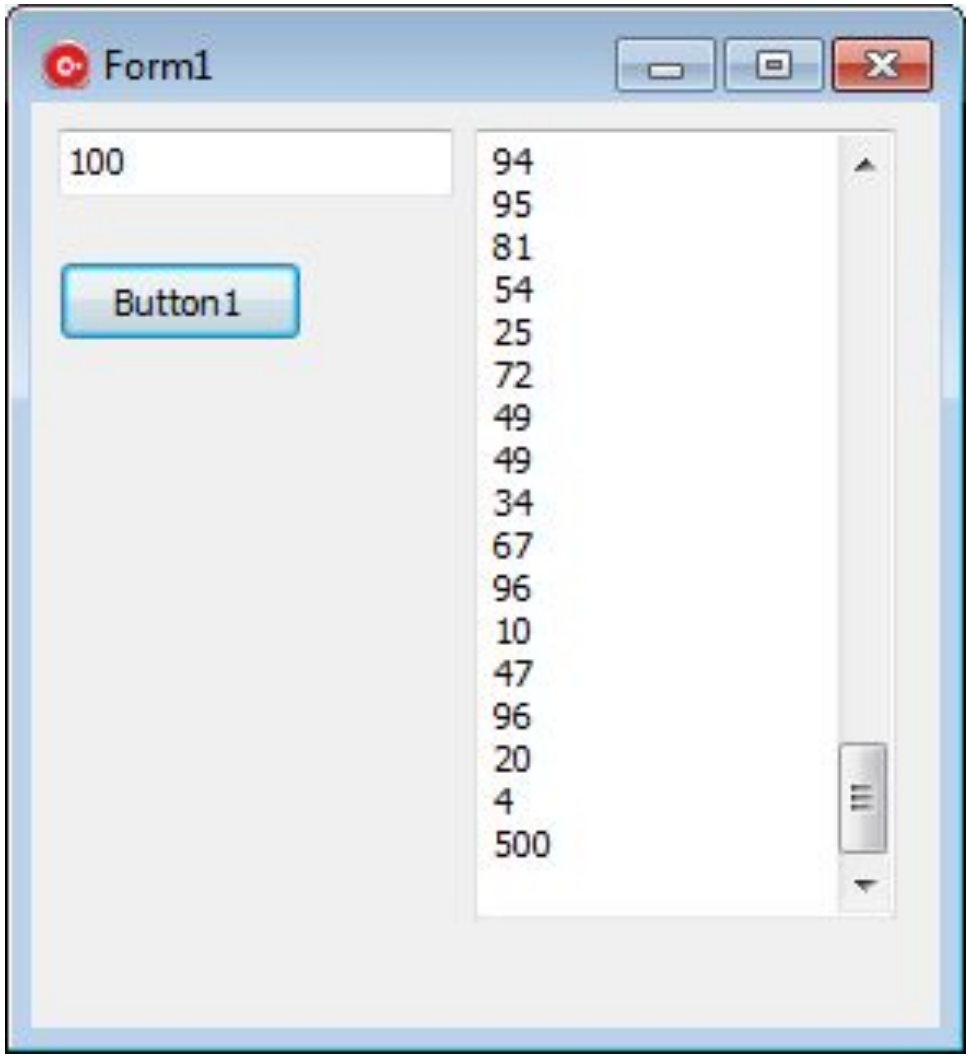
Приклад використання структур. Клас vector.

Ініціалізація

```
/#include <iostream>
//#include <vector>

using namespace std;

struct ma{
   int waight;
};
```

Десь у програмі

```
int N= StrToInt(Edit1->Text);
vector <ma> M(N);

M.resize(N);

Memo1->Clear();
for(int i = 0; i < N; i++){
     //i = 0...99
     M[i].waight = random(100);
}

int s = M.size();
M.resize(101);
M[100].waight = 500;

for(int i = 0; i < M.size(); i++)
     Memo1->Lines->Add(IntToStr(M[i].waight));
```



Об'єднання

Об'єднання - це тип класу, в якому всі дані поділяють одну й ту ж саму область пам'яті. У мові **C++** об'єднання може включати як функції-, так і дані-члени. Всі члени об'єднання відкриті за замовчуванням. Для створення закритих елементів необхідно використовувати ключове слово **private**. Загальна форма оголошення об'єднання виглядає наступним чином.

```
union ім'я_класу {

  // Відкриті члени за замовчуванням.

  private: // Закриті члени.
} Список об'єктів;
```

У мові C об'єднання можуть містити тільки дані-члени і ключове слово private не підтримується.

Приклад.

```
union name {
  char ch;
  int x;
} T;

T.x = 5; //
```

Як бачите, програма звертається до елементів об'єднання за допомогою селектора (точки), аналогічна запис використовувалася при зверненні до елементів структури.

Об'єднання

Головною особливістю об'єднання є те, що для кожного з оголошених елементів виділяється одна і та ж область пам'яті, тобто вони перекриваються. Хоча доступ до цієї області пам'яті можливий з використанням будь-якого з елементів.

Доступ до елементів об'єднання здійснюється у такий же спосіб, що і до структур.

Об'єднання застосовується для наступних цілей: економія пам'яті, перетворення типів, доступ до окремих байтів.

Пам'ять, яка відповідає змінної типу об'єднання, визначається величиною, необхідної для розміщення найбільш довгого елемента об'єднання. Коли використовується елемент меншої довжини, то змінна типу об'єднання може містити невикористану пам'ять. Всі елементи об'єднання зберігаються в одній і тій же області пам'яті, починаючи з однієї адреси.

Бітові поля

C++ має можливість працювати з окремими бітами. Бітові поля корисні з кількох причин. Нижче наведені три з них:

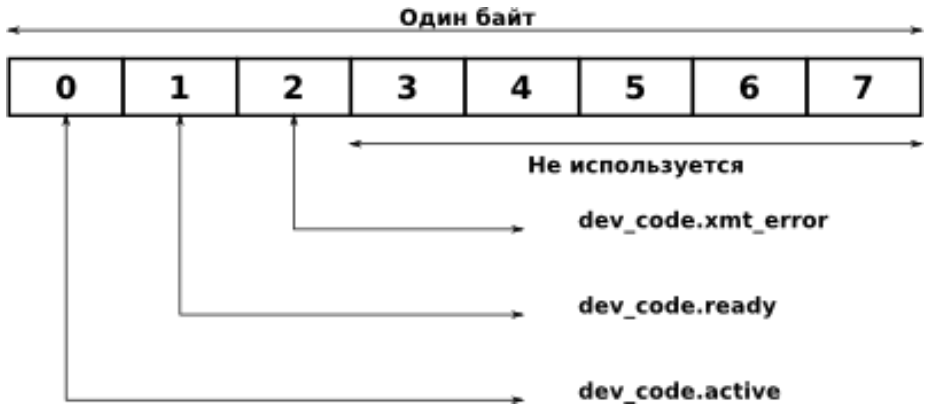
1. Якщо обмежена місце для зберігання інформації, можна зберегти кілька логічних (істина / неправда) змінних в одному байті.
2. Деякі інтерфейси пристроїв передають інформацію, закодований біти в один байт.
3. Деяким процедурам кодування необхідно отримати доступ до окремих бітам в байті.

Хоча всі ці функції можуть виконуватися за допомогою бітових операторів, бітові поля можуть внести велику ясність в програму.

Бітові поля

Метод використання бітових полів для доступу до бітів заснований на структурах. Бітове поле, насправді, - це просто особливий тип структури, яка визначає, яку довжину має кожен член. Стандартний вид об'яви бітових полів наступний:

```
struct ім'я структури {
  тип імя1: довжина;
  тип імя2: довжина;
  ...
  тип іменN: довжина;
}
```



Бітові поля повинні оголошуватися як int, unsigned або signed. Бітові поля довжиною 1 повинні оголошуватися як unsigned, оскільки 1 біт не може мати знак. Бітові поля можуть мати довжину від 1 до16 біт для 16-бітних середовищ, від 1 до 32 біт для 32-бітних середовищ і від 1 до 64 біт для 64-бітних середовищ.

Розглянемо наведене нижче визначення структури:

```
struct device {
  unsigned active: 1;
  unsigned ready: 1;
  unsigned xmt_error: 1;
} dev_code;
```

Десь у програмі

```
while(!dev_code.ready){}
```

Або

```
dev_code.active = 0;
```

Змінні із змінною структурою

Дуже часто деякі об'єкти програми відносяться до одного і того ж класу, з різницею лише деякими деталями. Розглянемо, наприклад, уявлення геометричних фігур. Загальна інформація про фігури може включати такі елементи, як площа, периметр. Однак відповідна інформація про геометричні розміри може виявитися різною в залежності від їх форми.

Розглянемо приклад, в якому інформація про геометричні фігури представляється на основі комбінованого використання структури і об'єднання.

```
enum figure_type {
  CIRCLE,
  BOX,
  TRIANGLEB общем случае каждый объект типа figure будет состоять из трех компонентов: area, perimetr, type.
};

struct figure {
            double area, perimetr; / * Загальні компоненти * /
            figure_type type;       / * Мітка компонента * /
            union{                  / * перерахування компонент * /
               double radius;       / * Круг * /
               double a [2];        / * Прямокутник * /
               double b [3];        / * Трикутник * /
            } geom_fig;
} Fig1, fig2;
```

Компонент type називається міткою активного компонента, так як він використовується для вказівки, який з компонентів об'єднання geom_fig є активним в даний момент. Така структура називається змінною структурою, тому що її компоненти змінюються в залежності від значення мітки активного компонента (значення type).

Вказівник на функцію

Адресний вираз, що стоїть перед дужками визначає адресу функції, що викликається. Це означає, що функція може бути викликана через вказівник на функцію.

Приклад:

int (*fun) (int x, int * y);

Тут оголошена змінна fun як вказівникна функцію з двома параметрами: типу **int** і вказівником на **int**. Сама функція повинна повертати значення типу **int**. Круглі дужки, що містять ім'я вказівника fun і ознака вказівника *, обов'язкові, інакше запис

int *fun (int x, int * y);

буде інтерпретуватися як обявлення функції fun, що повертає вказівник на **int**.

Виклик функції можливий тільки після ініціалізації значення вказівник а fun і має вигляд:

(* fun) (i, & j);

У цьому виразі для отримання адреси функції, на яку посилається вказівник fun, використовується операція разадресаціі *.

вказівникна функцію може бути переданий в якості параметра функції. При цьому разадресація відбувається під час виклику функції, на яку посилається вказівник на функцію.

Вказівник на функцію

Приклад:

```
double (* fun 1) (int x, int y);
double fun2 (int k, int l);
fun1 = fun2; / * Ініціалізація вказівника на функцію * /
(* fun1) (2,7); / * Звернення до функції * /
```

У розглянутому прикладі вказівник на функцію fun1 описаний як вказівник на функцію з двома параметрами, що повертає значення типу double, і також описана функція fun2.

Адресна арифметика

Під адресною арифметикою розуміються дії над вказівниками, пов'язані з використанням адрес пам'яті. Розглянемо операції, які можна застосовувати до вказівників.

1. Привласнення. Вказівником можна привласнити значення адреси. Будь-яке число, присвоєне вказівником, трактується як адреса пам'яті:

```
int *u, *adr;
int N;
u = &N; // Вказівником присвоєно адресу змінної N
```

Взяття адреси. Так як вказівник є змінною, то для отримання адреси пам'яті, де розташований вказівник, можна використовувати операцію взяття адреси &:

```
int * a * b;
 a = & b; // Вказівником a присвоєно адресу вказівника b
```

Адресна арифметика. Непряма адресація.

Для того, щоб отримати значення, що зберігається за адресою, на який посилається вказівник, використовується операція непрямої адресації *:

```
int * uk;
int n;
int m = 5;
uk = &m;  // uk присвоєно адресу змінної m
n = *uk;    // Змінна n прийме значення 5
```

Адресна арифметика. Перетворення типу.

Вказівник на об'єкт одного типу може бути перетворений на вказівник іншого типу. При цьому слід враховувати, що об'єкт, що адресується перетвореним вказівником, буде інтерпретуватися по-іншому.

Операція перетворення типу вказівника застосовується у вигляді
(<тип> *) <вказівник>:

```
int i, *ptr;
 i = 0x8e41;          //0x41 – 65 (dec)
 ptr = & i;
 Caption = IntToStr(  *((char *)ptr));      // ptr перетворений до типу char,

// Друкується: 65 * /
```

Адресна арифметика. Індексація.

Вказівник може індексуватися застосуванням до нього операції індексації, що позначається в **C++** квадратними дужками []. Індексація вказівника має вигляд <вказівник> [<індекс>], де <індекс> записується цілочисельним виразом.

З цієї адреси витягується або в цю адресу надсилається, в залежності від контексту застосування операції, значення, тип якого інтерпретується відповідно до типу вказівника. Розглянемо наступний приклад:

```
int * uk1;
int b, k;
uk1 = & b; // В uk1 адреса змінної b
k = 3;
b = uk1 [k];     // Змінної b присвоюється значення int,
                 // Взяте з адреси на 6 більшого, ніж
                 // Адреса змінної b; в uk1 адреса не змінився
uk1 [k] = -14; // На адресу на 6 більший, ніж адреса змінної b,
               // Записується -14
```

Операція індексації не змінює значення вказівника, до якого вона застосовувалася.

Адресна арифметика. Збільшення / зменшення.

Якщо до вказівника застосовується операція збільшення ++ або зменшення -, то значення вказівника збільшується або зменшується на розмір об'єкта, який він адресує:

```
long b;       // b - змінна типу int довжиною 4 байта
long * ptr;   // ptr - вказівник на об'єкт int довжиною 4 байта
ptr = & b;    // b ptr адреса змінної b
ptr ++;       // b ptr адресу збільшився на 4
ptr--;        // b ptr адреса зменшився на 4
```