7

Типове указател и псевдоним

7.1 Тип указател

Променлива, това е място за съхранение на данни, което може да съдържа различни стойности. Идентифицира се с дадено от потребителя име (идентификатор). Има си и тип. Дефинира се като се указват задължително типът и името ѝ. Типът определя броя на байтовете, в които ще се съхранява променливата, а също и множеството от операциите, които могат да се изпълняват над нея. Освен това, с променливата е свързана и стойност – неопределена или константа от типа, от който е тя. Нарича се още **rvalue**. Мястото в паметта, в което е записана rvalue, се идентифицира с адрес, който се нарича **адрес на променливата** или **lvalue**. По-точно адресът е адреса на първия байт от множеството байтове, отделени за променливата.

Пример: Фрагментът

```
int i = 1024;
```

дефинира променлива с име i и тип int. Стойността ѝ (rvalue) е 1024. i именува място от паметта (lvalue) с размери 4 байта, като lvalue е адреса на първия байт на това място.

Намирането на адреса на дефинирана променлива става чрез унарния префиксен дясно-асоциативен оператор & (амперсанд). Приоритетът му е същия като на унарните оператори +, -, !, ++, -- и др. Фиг. 7.1 описва оператора.

Оператор &

Синтаксис

&<променлива>

където <променлива> е вече дефинирана променлива.

Семантика

Намира адреса на <променлива>.

Фиг. 7.1 Оператор &

Пример: &i е адреса на променливата i и може да се изведе чрез оператора cout << &i;

Операторът & не може да се прилага върху константи и изрази, т.е. &100 и &(i+5) са недопустими обръщения. Не е възможно също прилагането му и върху променливи от тип масив, тъй като те имат и смисъла на *константни* указатели.

Адресите могат да се присвояват на специален тип променливи, наречени променливи от тип указател или само указатели.

Дефиниране

Нека T е име или дефиниция на тип. За типа T, T* е тип, наречен указател към T. T се нарича **указван тип** или **тип на указателя**.

Примери:

int* е тип указател към тип int;

enum {a, b, c}* е тип указател към тип enum {a, b, c}.

Множество от стойности

Състои се от адресите на данните от тип T, дефинирани в програмата, преди използването на T*. Те са константите на типа T*. Освен тях съществува специална константа с име NULL, наречена **нулев указател**. Тя може да бъде свързвана с всеки указател независимо от неговия тип. Тази константа се интерпретира като “сочи към никъде”, а в позиция на предикат е false.

Променлива величина, множеството от допустимите стойности, на която съвпада с множеството от стойности на типа T*, се нарича **променлива от тип T***, **променлива от тип указател към тип T** или само **указател към тип T**. Дефинира се по общоприетия начин. Фиг. 7.2 показва синтаксиса на дефиницията на променлива от тип указател.

Дефиниция на променлива от тип указател

T* <променлива> [= <стойност>]$_{опц}$
{,<променлива> [= <стойност>]$_{опц}$}$_{опц}$|
T *<променлива> [= <стойност>]$_{опц}$
{*<променлива> [= <стойност>]$_{опц}$}$_{опц}$;

където

- T е име или дефиниция на тип;
- <променлива> е идентификатор;
- <стойност> е шестнадесетично число, представляващо адрес на данна от тип T или NULL.

Фиг. 7.2 Дефиниция на променлива от тип указател

T определя типа на данните, които указателят адресира, а също и начина на интерпретацията им.

Забелязваме, че фрагментите

<променлива> [=<стойност>] и

*<променлива> [=<стойност>]

могат да се повтарят. За разделител се използва запетаята. В първия случай на дефиницията (Фиг. 7.2) обаче има особеност – операторът * се свързва с T само за първата променлива. Дефиницията

T* a, b;

е еквивалентна на

T* a;
T b;

т.е. само променливата a е от тип указател към тип T.

Примери: Дефиницията

int *pint1, *pint2;

задава два указателя към тип int, а

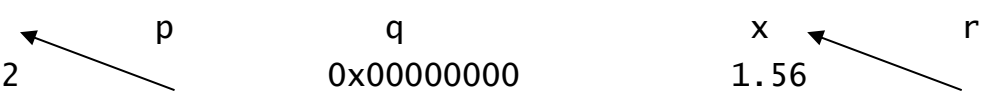
```
   int *pint1, pint2;
```
- указател pint1 към int и променлива pint2 от тип int.

Дефиницията на променлива от тип указател предизвиква в ОП да се отделят 4В, в които се записва някакъв адрес от множеството от стойности на съответния тип, ако дефиницията е с инициализация и неопределено или NULL, ако дефиницията не е с инициализация. (За реализацията Visual C++ 6.0 е неопределено). Този адрес е **стойността** на променливата от тип указател, а записаното на този адрес е **съдържанието** ѝ.

Пример: В резултат от изпълнението на дефинициите

```
   int i = 12;
   int* p = &i;      // p е инициализирано с адреса на i
   double *q = NULL; // q е инициализирано с нулевия указтел
   double x = 1.56;
   double *r = &x;   // r е инициализирано с адреса на x
```

ОП има вида:



Съвет: Всеки указател, който не сочи към конкретен адрес, е добре да се свърже с константата NULL. Ако по невнимание се опитате да използвате нулев указател, програмата ви може да извърши нарушение при достъп и да блокира, но това е по-добре, отколкото указателят да сочи към кой знай къде.

Операции и вградени функции

Извличане на съдържанието на указател

Осъществява се чрез префиксния, дясноасоциативен унарен оператор * (Фиг. 7.3). * има приоритет на унарен оператор.

Оператор *
Синтаксис

*<променлива_от_тип_указател>

Семантика

Извлича стойността на адреса, записан в <променлива_от_тип_ указател>, т.е. съдържанието на <променлива_от_тип_ указател>.

Фиг. 7.3 Оператор *

Като използваме дефинициите от примера по-горе, имаме:

*p е 12 // 12 е съдържанието на p

*r е 1.56 // 1.56 е съдържанието на r

Освен, че намира съдържанието на променлива от тип указател, обръщението

*<променлива_от_тип_указател>

е променлива от тип T. (Всички операции, допустими за типа T, са допустими и за нея.

Като използваме дефинициите от примера по-горе, *p и *r са цяла и реална променливи, съответно. След изпълнение на операторите за присвояване

```
*p = 20;
*r = 2.18;
```

стойността на i се променя на 20, а тази на r – на 2.18.

Аритметични и логически операции

Променливите от тип указател могат да участват като операнди в следните аритметични и логически операции +, -, ++, --, ==, !=, >, >=, < и <=. Изпълнението на аритметични операции върху указатели е свързано с някои особености, заради което аритметиката с указатели се нарича още адресна аритметика. Особеността се изразява в т. нар. мащабиране. Ще го изясним чрез пример.

Да разгледаме фрагмента

```
int *p;
double *q;
...
p = p + 1;
q = q + 1;
```

Операторът p = p + 1; свързва p не със стойността на p, увеличена с 1, а с p + 1*4, където 4 е броя на байтовете, необходими за записване на данна от тип int (p е указател към int). Аналогично, q = q + 1; увеличава стойността на q не с 1, а с 8, тъй като q е указател към double (8 байта са необходими за записване на данна от този тип).

Общото правило е следното: Ако p е указател от тип T*, p+i е съкратен запис на p + i*sizeof(T), където sizeof(T) е функция, която намира броя на байтовете, необходими за записване на данна от тип T.

Въвеждане

Не е възможно въвеждане на данни от тип указател чрез оператора cin. Свързването на указател със стойност става чрез инициализация или оператора за присвояване.

Извеждане

Осъществява се по стандартния начин - чрез оператора cout.

Допълнение

Типът, който се задава в дефиницията на променлива от тип указател, е информация за компилатора относно начина, по който да се интерпретира съдържанието на указателя. В контекста на последния пример *p са четири байта, които ще се интерпретират като цяло число от тип int. Аналогично, *q са осем байта, които ще се интерпретират като реално число от тип double.

Следващата програма илюстрира дефинирането и операциите за работа с указатели.

```
#include <iostream.h>
int main()
{int n = 10;   // дефинира и инициализира цяла променлива
 int* pn = &n; // дефинира и инициализира указател pn към n
 // показва, че указателят сочи към n
 cout << "n=  " << n << "  *pn= " << *pn << '\n';
```

```
 // показва, че адресът на n е равен на стойността на pn
 cout << "&n= "<< &n << "  pn= " << pn << '\n';
 // намиране на стойността на n чрез pn
 int m = *pn;  //  m == 10
 // промяна на стойността на n чрез pn
 *pn = 20;
 // извеждане на стойността на n
 cout << "n= " << n << '\n'; // n == 20
 return 0;
}
```

В някои случаи е важна стойността на променливата от тип указател (адресът), а не нейното съдържание. Тогава тя се дефинира като указател към *тип void*. Този тип указатели са предвидени с цел една и съща променлива - указател да може в различни моменти да сочи към данни от различен тип. В този случай, при опит да се използва съдържанието на променливата от тип указател, ще се предизвика грешка. Съдържанието на променлива - указател към тип void може да се извлече само след привеждане на типа на указателя (void*) до типа на съдържанието. Това може да се осъществи чрез операторите за преобразуване на типове.

Пример:

```
int a = 100;
void* p;                // дефинира указател към void
p = &a;                 // инициализира p
cout << *p;             // грешка
cout << *((int*) p);    // преобразува p в указател към int
                        // и тогава извлича съдържанието му.
```

В C++ е възможно да се дефинират указатели, които са константи (T* const <идентификатор>), а също и указатели, които сочат към константи (const T* <идентификатор>). И в двата случая се използва запазената дума const, която се поставя пред съответните елементи от дефинициите на указателите. Стойността на елемента, дефиниран като const (указателя или обекта, към който сочи) не може да бъде променяна. В дефиницията:

T* const <идентификатор>;

<идентификатор> е константен указател към тип T и не може да бъде променяна стойността му. В дефиницията:

const T* <идентификатор>;

<идентификатор> е указател към константа от тип T и не може да бъде променяно съдържанието му.

Пример:

```
int i, j = 5;
int *pi;              // pi е указател към int
int * const b = &i;   // b е константен указател към int
const int *c = &j;    // c е указател към цяла константа.
b = &j;               // грешка, b е константен указател
*c = 15;              // грешка, *c е константа
```

7.2 Указатели и масиви

В C++ има интересна и полезна връзка между указателите и масивите. Изразява се в това, че имената на масивите са указатели към техните "първи" елементи. Последното позволява указателите да се разглеждат като алтернативен начин за обхождане на елементите на даден масив.

Указатели и едномерни масиви

Нека a е масив, дефиниран по следния начин:

int a[100];

Тъй като a е указател към a[0], *a е стойността на a[0], т.е. *a и a[0] са два различни записа на стойността на първия елемент на масива. Тъй като елементите на масива са разположени последователно в паметта, a + 1 е адреса на a[1], a + 2 е адреса на a[2] и т.н. a + n-1 е адреса на a[n-1]. Тогава *(a+i) е друг запис на a[i] (i = 0, 1, ..., n-1).

Има обаче една особеност. Имената на масивите са константни указатели. Заради това, някои от аритметичните оператори, приложими над указатели, не могат да се приложат над масиви. Такива са ++, -- и присвояването на стойност.

Следващата програма показва два начина за извеждане на елементите на масив.

```
#include <iostream.h>
int main()
{int a[] = {1, 2, 3, 4, 5, 6};
 int i;
 for (i = 0; i <=  5; i++)
    cout << a[i] << '\n';
 for (i = 0; i <=  5; i++)
    cout << *(a+i) << '\n';
 return 0;
}
```

Операторът

```
for (i = 0; i <= 5; i++)
{cout << *a << '\n';
 a++;
}
```

съобщава за грешка заради a++ (a е константен указател и не може да бъде променян). Може да се поправи като се използва помощна променлива от тип указател към int, инициализирана с масива a, т.е.

```
int* p = a;
for (i = 0; i <= 5; i++)
{cout << *p << '\n';
 p++;
 }
```

Използването на указатели е по-бърз начин за достъп до елементите на масива и заради това се предпочита. Индексираните променливи правят кода по-ясен и разбираем. В процеса на компилация всички конструкции от вида a[i] се преобразуват в *(a+i), т.е. операторът за индексиране [] се обработва от компилатора чрез адресна аритметика. Полезно е да отбележим, че операторът [] е ляво-асоциативен и с по-висок приоритет от унарните оператори (в частност от оператора за извличане на съдържание *).

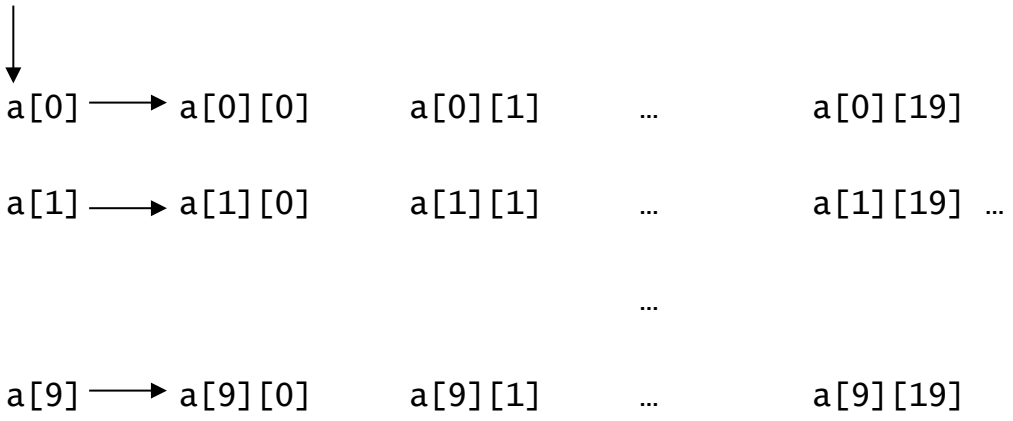
Указатели и двумерни масиви

Името на двумерен масив е константен указател към първия елемент на едномерен масив от константни указатели. Ще изясним с пример казаното.

Нека a е двумерен масив, дефиниран по следния начин:

```
int a[10][20];
```

Променливата a е константен указател към първия елемент на едномерния масив a[0], a[1], ..., a[9], като всяко a[i] е константен указател към a[i][0] (i = 0,1, ..., 9), т.е.



Тогава

```
**a == a[0][0]
```

a[0] == *a a[1] == *(a + 1) ... a[9] == *(a + 9),

т.е.

```
a[i] == *(a + i)
```

Като използваме, че операторът за индексиране е лявоасоциативен и с по-висок приоритет от оператора *, получаваме:

```
a[i][j] == (*(a+i))[j] == *(*(a+i)+j).
```

Задача 67. Да се напише програма, която въвежда по редове правоъгълна матрица Anxk от цели числа. Конструира матрица, образувана от редовете на A от четна позиция, след което я извежда като увеличава с 1 всеки неин елемент. Конструира матрица, образувана от редовете на A от нечетна позиция, след което я извежда като увеличава с 2 всеки неин елемент. И накрая, ако n е четно,

извежда сумата на матриците от редовете от четните и нечетните позиции на A (без увеличенията с 1 и 2).

Програма Zad67.cpp решава задачата. Матрицата от редовете на A от четна позиция е конструирана в *масива от указатели p*, а матрицата от редовете на A от нечетна позиция е конструирана в *масива от указатели q*. Конструирането на масива p е реализирано чрез фрагмента:

```
int m = -1;
  for (i = 0; i <= n-1; i = i+2)
  {m++;
   *(p+m) = *(a+i);
  }
```

Стойността на p[0] е адреса на a[0][0], стойността на p[1] е адреса на a[2][0], стойността на p[2] е адреса на a[4][0] и т.н. Масивът q е конструиран по аналогичен начин, но стойността на q[0] е адреса на a[1][0], стойността на q[1] е адреса на a[3][0], стойността на q[2] е адреса на a[5][0] и т.н. Елементите на масива A не се копират на друго място в паметта. Така се прави икономия на ОП.

```
// Program Zad67.cpp
#include <iostream.h>
#include <iomanip.h>
int main()
{int a[20][100];
 int* p[20];
 int* q[20];
 cout << "n, k = ";
 int n, k;
 cin >> n >> k;
 if (!cin)
 {cout << "Error! \n";
  return 0;
 }
 // въвеждане на матрицата
 int i, j;
 for (i = 0; i <= n-1; i++)
```

```
 for (j = 0; j <= k-1; j++)
 {cout << "a[" << i << "][" << j << "]= ";
  cin >> *(*(a + i) + j);
 }
// извеждане на матрицата
 for (i = 0; i <= n-1; i++)
 {for(j = 0; j <= k-1; j++)
   cout << setw(10) << *(*(a + i) + j);
  cout << '\n';
 }
 cout << "\n\n  Нова матрица от редовете в четни позиции \n";
 // конструиране
 int m = -1;
 for (i = 0; i <= n-1; i = i+2)
 {m++;
  *(p+m) = *(a+i);
 }
 // извеждане с увеличение на елементите с 1
 for (i = 0; i <= m; i++)
 {for(j = 0; j <= k-1; j++)
     cout << setw(10) << *(*(p + i) + j) + 1;
  cout << '\n';
 }
 cout << "\n\n  Нова матрица от редовете в нечетни позиции \n";
 // конструиране
 int l = -1;
 for (i = 1; i <= n - 1; i = i + 2)
 {l++;
  q[l] = a[i];
 }
 // извеждане с увеличение на елементите с 2
 for (i = 0; i <= l; i++)
 {for(j = 0; j <= k - 1; j++)
     cout << setw(10) << *(*(q + i) + j) + 2;
  cout << '\n';
 }
```

```
  // сумиране на двете матрици ако n е четно
  if (n%2 == 0)
   for (i = 0; i <= m; i++)
   {for (j = 0; j <= k-1; j++)
     cout << setw(10) << *(*(p + i) + j) + *(*(q + i) + j);
    cout << '\n';
   }
 return 0;
}
```

7.3 Указатели и низове

Низовете са масиви от символи. Името на променлива от тип низ е константен указател, както и името на всеки друг масив. Така всичко, което казахме за връзката между масив и указател е в сила и за низ – указател.

Следващият пример илюстрира обхождане на низ чрез използване на указател към char. Обхождането продължава до достигане на знака за край на низ.

```
#include <iostream.h>
int main()
{char str[] = "C++Language"; // str е константен указател
 char* pstr = str;      // pstr е указател към низа str
 while (*pstr)
 {cout << *pstr << '\n';
  pstr++;
 }    // pstr вече не е свързан с низа “C++Language”.
 return 0;
}
```

Тъй като низът е зададен чрез масива от символи str, str е константен указател и не може да бъде променяна стойността му. Затова се налага използването на помощната променлива pstr. Ако низът е зададен чрез указател към char, както е в следващата програма, не се налага използването на такава.

```
#include <iostream.h>
int main()
```

```
{char* str = "C++Language";  // str е променлива
 while (*str)
 {cout << *str << '\n';
  str++;
 }
 return 0;
}
```

Примерите показват, че задаването на низ като указател към char има предимство пред задаването като масив от символи. Ще отбележим обаче, че дефиницията

```
char* str = "C++Language";
```

не може да бъде заменена с

```
char* str;
cin >> str;
```

следвани с въвеждане на низа “C++Language”(ще напомним, че не е възможно въвеждане на стойност на променлива от тип указател чрез оператора cin), докато дефиницията

```
char str[20];
```

позволява въвеждането му чрез cin, т.е.

```
cin >> str;
```

е допустимо.

Има още една особеност при дефинирането на низ като указател към char за реализацията Visual C++ 6.0. Ще я илюстрираме с пример.

Нека дефинираме променлива от тип низ по следния начин:

```
char s[] = “abba”;
```

Операторът

```
*s = ‘A’;
```

е еквивалентен на s[0] = ‘A’ и ще замени първото срещане на символа ‘a’ в s с ‘A’. Така

```
cout << s;
```

ще изведе низа

```
Abba
```

Да разгледаме съответната дефиницията на s чрез указател към char

```
char* s = “abba”;
```

Операторът

```
*s = ‘A’;
```

би трябвало да замени първото срещане на символа ‘a’ в s с ‘A’, тъй като s съдържа адреса на първото ‘a’. Тук обаче реализацията на Visual C++ съобщава за грешка – нарушение на достъпа. Последното може да се избегне като опцията на компилатора /ZI се замени с /Zi. Това се реализира като в менюто Project се избере Settings, след това C/C++, където Category трябва да има опция General. Накрая, в Project Options се промени /ZI на /Zi. Опцията /Zi се грижи за проблемите при нарушаване на достъпа - едно неудобство, което трябва да се има предвид.

7.4 Тип псевдоним

Чрез псевдонимите се задават алтернативни имена на обекти в общия смисъл на думата (променливи, константи и др.). В тази част ще ги разгледаме малко ограничено (псевдоними само за променливи).

Дефиниране

Нека T е име на тип. Типът T& е тип псевдоним на T. T се нарича **базов тип** на типа псевдоним.

Множество от стойности

Състои се от всички имена на дефинирани вече променливи от тип T.
Пример: Нека програмата съдържа следните дефиниции:

```
int a, b = 5;
...
int x, y = 9, z = 8;
...
```

Множеството от стойности на типа int& съдържа имената a, b, x, y, z. Променлива величина, множеството от допустимите стойности на която съвпада с множеството от стойности на даден тип псевдоним, се нарича променлива от този тип псевдоним. Фиг. 4 илюстрира дефиницията.

Дефиниране на променлива от тип псевдоним

<дефиниране_на_променлива_от_тип_псевдоним> ::=
T& <променлива> = <вече_дефинирана_променлива_от_тип_T>
{,<променлива> = <вече_дефинирана_променлива_от_тип_T>}$_{опц}$;|
T &<променлива> = <вече_дефинирана_променлива_от_тип_T>
{, &<променлива> = <вече_дефинирана_променлива_от_тип_T>}$_{опц}$;

където T е име тип. Променливата след знак = се нарича инициализатор.

Фиг. 7.4 Дефиниране на променлива от тип псевдоним

Забелязваме, че е възможно фрагментите:

<променлива> = <вече_дефинирана_променлива_от_тип_T> и

&<променлива> = <вече_дефинирана_променлива_от_тип_T>

да се повтарят многократно. За разделител се използва символът запетая. В първият случай има една особеност. Дефиницията

```
T& a = b, c = d;
```

е еквивалентна на

```
T& a = b;
T c = d;
```

т.е. операторът & след типа T се отнася само за първата променлива след него.

Пример: Дефинициите

```
int a = 5;
int& syna = a;
double r = 1.85;
double &syn1 = r, &syn2 = r;
int& syn3 = a, syn4 = a;
```

определят syna и syn3 за псевдоними на цялата променлива a, syn1 и syn2 за псевдоними на реалната променлива r и syn4 за променлива от тип int.

Дефиницията на променлива от тип псевдоним задължително е с инициализация – дефинирана променлива от същия тип като на базовия тип на типа псевдоним. След това не е възможно променливата-псевдоним да стане псевдоним на нова променлива. Затова тя е "най-константната" променлива, която може да съществува.

Пример:

```
 ...
 int a = 5;
 int &syn = a;  // syn е псевдоним на a
 int b = 10;
 int& syn = b;    // error, повторна дефиниция
 ...
```

Операции и вградени функции

Дефиницията на променлива от тип псевдоним свързва променливата-псевдоним с инициализатора и всички операции и вградени функции, които могат да се прилагат над инициализатора, могат да се прилагат и над псевдонима му и обратно.

Примери:

```
1. int ii = 0;
   int& rr = ii;
   rr++;
   int* pp = &rr;
```

Резултатът от изпълнението на първите два оператора е следния:

ОП ii, rr

...	0	...

Операторът rr++; не променя адреса на rr, а стойността на ii и тя от 0 става 1. В случая rr++ е еквивалентен на ii++.

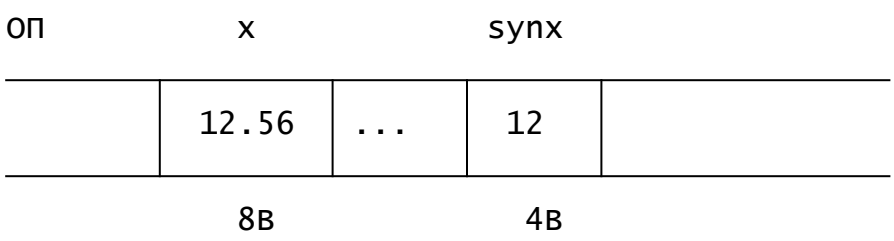
Адресът на rr е адреса на ii. Намира се чрез &rr. Чрез дефиницията

```
int* pp = &rr;
```

pp е определена като указател към int, инициализирана с адреса на rr.

```
2. int a = 5;
   int &syn = a;
   cout << syn << "  " << a << '\n';
   int b = 10;
   syn = b;
```

```
      cout << b << "  " << a << “  “ << syn << '\n';
```
извежда
```
   5   5
   10  10   10
```
Операторът syn = b; е еквивалентен на a = b;.
```
   3. int i = 1;
      int& r = i; // r и i са свързани с 1
      cout << r; // извежда 1
      int x = r;  // x има стойност 1
      r = 2;   // еквивалентно е на i = 2;
```
Допълнение: Възможно е типът на инициализатора да е различен от този на псевдонима. В този случай се създава нова, наречена **временна**, променлива от типа на псевдонима, която се инициализира със зададената от инициализатора стойност, преобразувана до типа на псевдонима. Например, след дефиницията
```
   double x = 12.56;
   int& synx = x;
```
имаме



Сега x и псевдонимът й synx са различни променливи и промяната на x няма да влияе на synx и обратно.

Константни псевдоними

В C++ е възможно да се дефинират псевдоними, които са константи. За целта се използва запазената дума const, която се поставя пред дефиницията на променливата от тип псевдоним. По такъв начин псевдонимът не може да променя стойността си, но ако е псевдоним на променлива, промяната на стойността му може да стане чрез промяна на променливата.

Пример: Фрагментът

```
int i = 125;
const int& syni = i;
cout << i << " " << syni << '\n'; // i и syni имат стойност 125
syni = 25;
cout << i << " " << syni << '\n';
```

ще съобщи за грешка (syni е константа и не може да е лява страна на оператор за присвояване), но фрагментът

```
int i = 125;
const int& syni = i;
cout << i << " " << syni << '\n';
i = i + 25;
cout << i << " " << syni << '\n';
```

ще изведе

```
125 125
150 150
```

Последното показва, че константен псевдоним на променлива защитава промяната на стойността на променливата чрез псевдонима.

Допълнителна литература

1. Ст. Липман, Езикът C++ в примери, “КОЛХИДА ТРЕЙД” КООП, София, 1993.
2. B. Stroustrup, C++ Programming Language. Third Edition, Addison-Wesley, 1997.