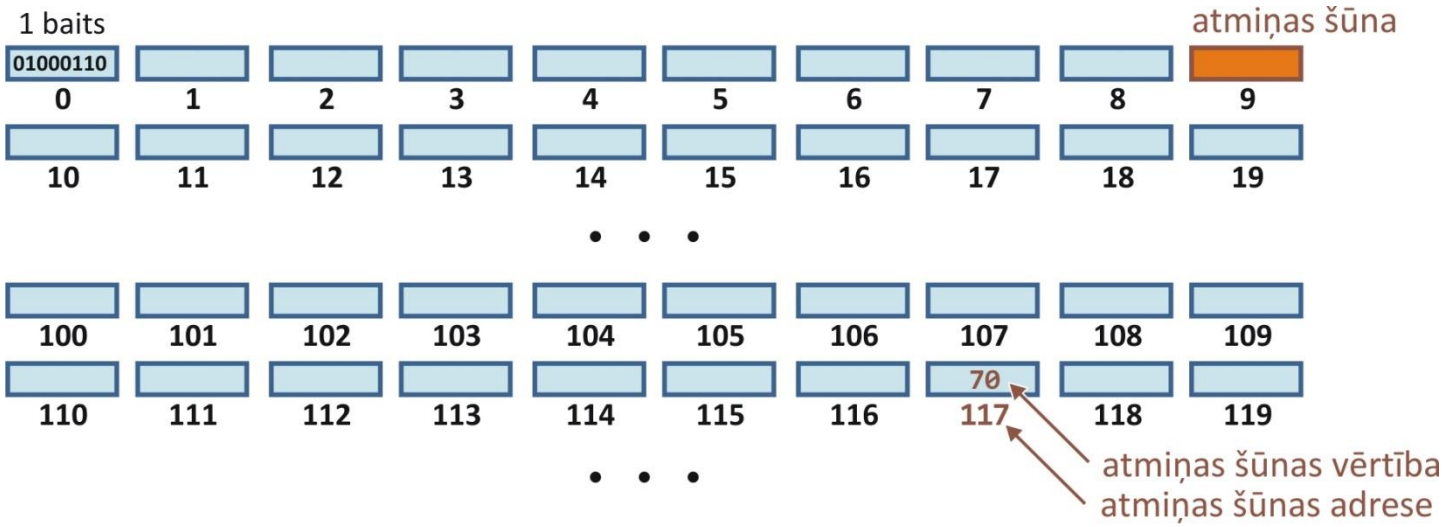
Programmētāju skola
3. līmeņa grupa

Dinamiskie mainīgie

Igors Ščukins
RTU Daugavpils

Operatīva atmiņa

Datora operatīva atmiņa ir organizēta kā baitu šūnu secīgums. Katru atmiņas šūnu nosaka atšķirīgs unikāls numurs jeb **adrese**. Atmiņas adrešu numerācija sākas ar **0** un secīgi pieaug par **1**. Lai piekļūtu datiem atmiņā, procesoram jānorāda tā atrašanās vietas adrese (**Random Access**).



Mainīgo izvietojums atmiņā

Jebkāds mainīgais tiek obligāti izvietots operatīvā atmiņā. Praktiski mainīgais – tas ir datora operatīvas atmiņas fragments un tiek raksturojams ar tā sākumadresi un izmēru.

Mainīgais	Izmērs	Sākumadrese
`int a = 0x12345678;`	4	0x00371F40
`char b = 'A';`	1	0x00371F44
`char c[] = "Hi!";`	4	0x00371F48
`char d[10] = "World";`	10	0x00371F4C
`int e[] = {65, 66, 67, 68, 69};`	20	0x00371F58

Atmiņas adreses	Atmiņas dati	
	Heksadecimālā veidā	ASCII veidā
00371F40	78 56 34 12 41 00 00 00 48 69 21 00 57 6f 72 6c	xV4.A...Hi!.Worl
00371F50	64 00 00 00 00 00 00 00 41 00 00 00 42 00 00 00	d.......A...B...
00371F60	43 00 00 00 44 00 00 00 45 00 00 00 00 00 00 00	C...D...E.....
00371F70	29 23 be 84 e1 6c d6 ae 52 90 49 f1 f1 bb e9 eb	)#..įlÖ®R.Ińń»éė
00371F80	b3 a6 db 3c 87 0c 3e 99 24 5e 0d 1c 06 b7 47 de	.¦Ū<..>™$^....·GŽ
00371F90	b3 12 4d c8 43 bb 8b a6 1f 03 5a 7d 09 38 25 1f	..MČC».¦..Z}.8%.
00371FA0	5d d4 cb fc 96 f5 45 3b 13 0d 89 0a 1c db ae 32	]ÕĖü–õE;.....Ū®2
...	...	...

Mainīgā izmēra pārsniegšana

```
int a = 0x12345678;
char b = 'A';
char c[] = "Hi!";
char d[10] = "World";
int e[] = {65, 66, 67, 68, 69};

int main()
{
      c[3] = ' ';
      puts(c);
}
```

Atmiņas adreses	Atmiņas dati	
	Heksadecimālā veidā	ASCII veidā
00371F40	78 56 34 12 41 00 00 00 48 69 21 20 57 6f 72 6c	xV4.A...Hi! Worl
00371F50	64 00 00 00 00 00 00 00 41 00 00 00 42 00 00 00	d.......A...B...
00371F60	43 00 00 00 44 00 00 00 45 00 00 00 00 00 00 00	C...D...E.......
00371F70	29 23 be 84 e1 6c d6 ae 52 90 49 f1 f1 bb e9 eb	)#..įlÖ®R.Ińń»éė
...	...	...

Hi! World

Rādītāji

Rādītājs – tas ir mainīgais, kas var glābāt cita mainīgā vai operatīvas atmiņas apgabala adresi.

Deklarācija

```
type* name;
```

```
float* pf;
int* pi, * pj;
void* p;
```

Rādītājam var piešķirt

Atbilstoša tipa mainīgā adresi, ko var saņemt, izmantojot operāciju **&**

```
int i; pi = &i;
float f; pf = &f;
p = &f;
```

~~pf = &i;~~
~~pi = &f;~~

Atbilstoša tipa cita rādītāja vērtību

```
pj = pi;
p = pf;
```

~~pi = pf;~~

Tukšo rādītāju, kas C valodā ir apzīmējams ar konstanti **NULL** (C++ - **nullptr**)

```
pf = nullptr;
if (pf == nullptr)...
```

Netieša piekļuve

Netieša piekļuve (dereference, разыменование) – tā ir piekļuve mainīgajam vai atmiņas apgabalam, kuras adrese ir saglabāta rādītājā.

Sintakse

```
*name
```

Piemēram

```
int a = 0, b = 0, c = 0, d = 0, * p = &a;
*p = 10;
p = &b; a = *p + 20;
p = &c; for (*p = 0; *p < 10; *p++) std::cout << *p;
p = &d; std::cin >> *p;
```

Piemēram

```
void clr(int * ptr) { *ptr = 0; }
void main()
{
     int a, b[5];
     clr(&a);
     for (int i = 0; i < 5; i++) clr(&b[i]);
}
```

Operācijas ar rādītājiem

C atbalsta rādītāju palielināšanu un samazināšanu par vērtību, kas ir kārtīga tipa izmēram, uz kuru norāda rādītājs.

```
ptr ++          ptr --
ptr += num      ptr -= num
ptr + num       ptr - num
```

Piemēram

```
char* pc;       pc++;           // +1
int* pi;        pi -= 2;        // -8
double* pd;     pd = pd + 5;    // +40
```

Piemēram

```
int a[10], * p = &a[0];
for (int i = 0; i < 10; i++) *(p++) = 0;
```

Masīvi

C valodā masīva identifikators apzīmē rādītāju uz pirmo masīva elementu un operācija **[ ]** nozīmē palielināta rādītāja dereferenci.

```
&a[0] ≡ a
&a[i] ≡ a + i
 a[i] ≡ *(a + i)
```

Piemēram

```
int a[5] = { 0 };
*a = 10;              //a[0] = 10
*(a + 1) = 20;        //a[1] = 20
int* p = a + 2;       //p -> a[2]
*p = 30;              //a[2] = 30
p[1] = 40;            //a[3] = 40
(p + 1)[1] = 50;      //a[4] = 50
```

Piemēram

```
int a[] = { 1, 2, 3, 4, 0 };
for (int *p = a; *p != 0; p++) printf("%d ", *p);
```

Operācijas ar simbolu rindām

Strādājot ar simbolu rindām C stilā (**char***) ir jāņem vēra ka rindas mainīgais – tas ir rādītājs.

Nederīgas operācijas

```
char str1[100], str2[100];
```

~~`str1 = str2;`~~

`strcpy(str1, str2);`

~~`str1 = "Hello, world!";`~~

`strcpy(str1, "Hello, world!");`

~~`if (str1 == str2)...`~~

`if (strcmp(str1, str2) == 0)...`

~~`str1 = str1 + str2;`~~

`strcat(str1, str2);`

Cikla uzbūve

```
for (int i = 0; str[i]; i++) printf("%c", str[i]);

for (char* s = str; *s; s++) printf("%c", *s);
```

Operācijas ar simbolu rindām

Rindas piešķiršana – nokopēt visus simbolus no rindas str2 uz rindu str1

```
void strCpy(char* str1, char* str2) {
     while (*(str1++) = *(str2++));
}
```

Rindas konkatenācija – nokopēt visus simbolus no rindas str2 uz rindas str1 beigām

```
void strCat(char* str1, char* str2) {
     while (*str1) str1++;
     while (*(str1++) = *(str2++));
}
```

Rindas salīdzināšana – pārbaudīt vai rinda str1 ir vienāda ar rindu str2

```
bool strCmp(char* str1, char* str2) {
     while (*str1 || *str2) if (*(str1++) != *(str2++)) return false;
     return true;
}
```

References

C++ piedāvā alternatīvo netiešas piekļuves sintaksi – **references** (***reference, ссылка***).
References darbojas gandrīz kā rādītāji ar divām galvenajām atšķirībām:

- references nepieprasa izpildīt dereferenci, lai piekļūtu datiem;
- referenci pēc inicializācijas nevar pārvirzīt uz citu adresi.

Deklarācija

```
type & name = variable;
```

```
float f;
float& rf = f;
```

```
std::string s{"Hello"};
std::string& rs = s;
```

```
int i = 0;
int* pi = &i;
int*& rpi = pi;
```

Piekļuve

```
int i1 = 1, i2 = 2;
int& ri = i1;
```

```
ri = 10; // i1 = 10, i2 = 2
ri = i2; // i1 = 2,  i2 = 2
```

Funkcijas parametru nodošana

```
void f(int* v) {
    *v = 10;
}
```

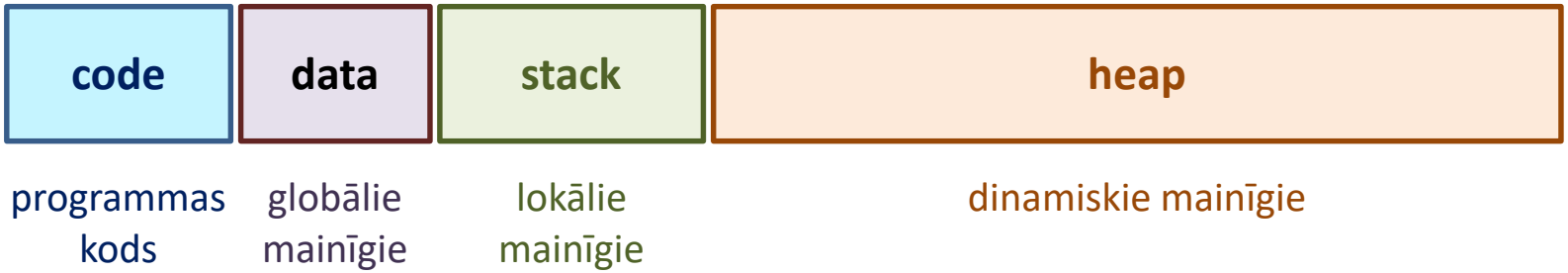
```
int i;
f(&i);
cout << i;
```

```
void f(int& v) {
    v = 10;
}
```

```
int i;
f(i);
cout << i;
```

Atmiņas sadalījums

Kad programma ir ielādēta atmiņā un sāk strādāt, operētājsistēma iedala tai vairākus operatīvas atmiņas apgabalus (segmentus), kur programma darba laikā var glābāt savu kodu, datus un dienestu informāciju.



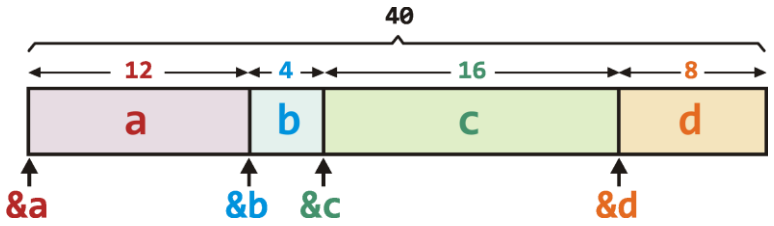
Katrā segmentā programma sadala un izmanto atmiņu atšķirīgi, t.i. pielieto dažādus atmiņas sadalījumu principus.

Atmiņas sadalījums tas ir programmas mainīgo izveidošanas, glabāšanas un dzēšanas veids. Parasti pielieto trīs sadalījuma veidus: **statiskais**, **automātiskais** un **dinamiskais**.

Statiskais atmiņas sadalījums

Statiskais sadalījums paredz mainīgo izveidošanu programmas palaišanas momentā un dzēšanu – pēc programmas pabeigšanas. Tāds sadalījums ir pielietojams globālajiem mainīgajiem un lokāliem mainīgajiem, kuri ir deklarēti ar moifikatoru `static`.

```
char a[12];   // 12 bytes
int b;        //  4 bytes
int c[4];     // 16 bytes
double d;     //  8 bytes
```



Automātiskais atmiņas sadalījums

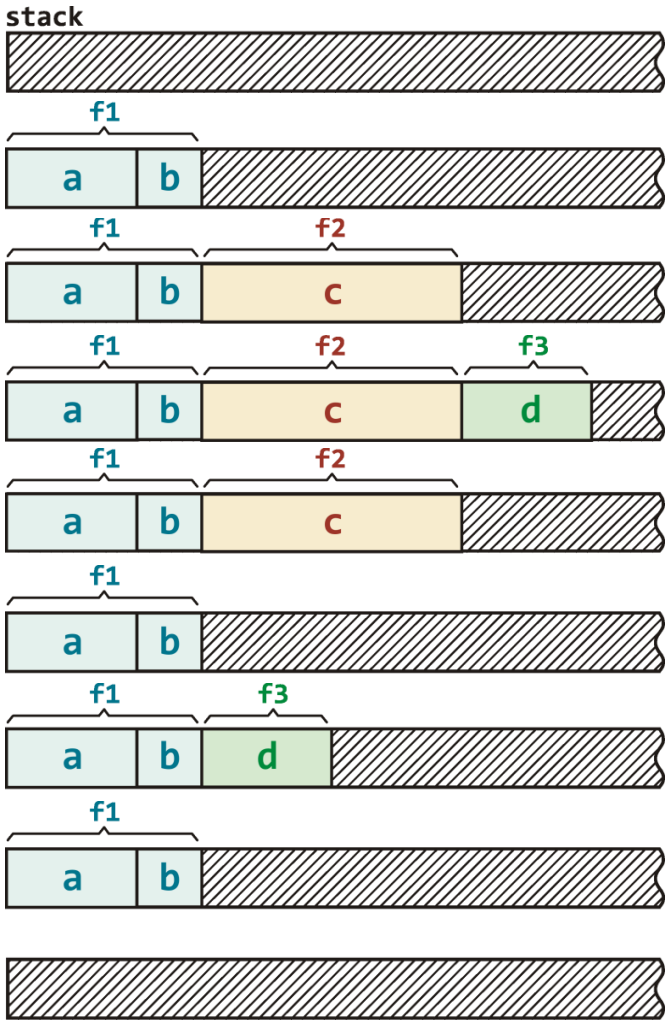
Automātiskais sadalījums tiek pielietots lokālajiem mainīgajiem. Tādi mainīgie tiek izveidotie funkcijas izsaukšanas momentā un tiek dzēsti, kad funkcija pabeidz darbu. Automātisko mainīgo glabāšanai ir paredzēts speciālais atmiņas apgabals, ko dēvē par **steku (stack)**.

```
void f1() {
    double a;
    int b;
    f2();
    f3();
}

void f2() {
    int c[4];
    f3();
}

void f3() {
    char d[8];
}
```

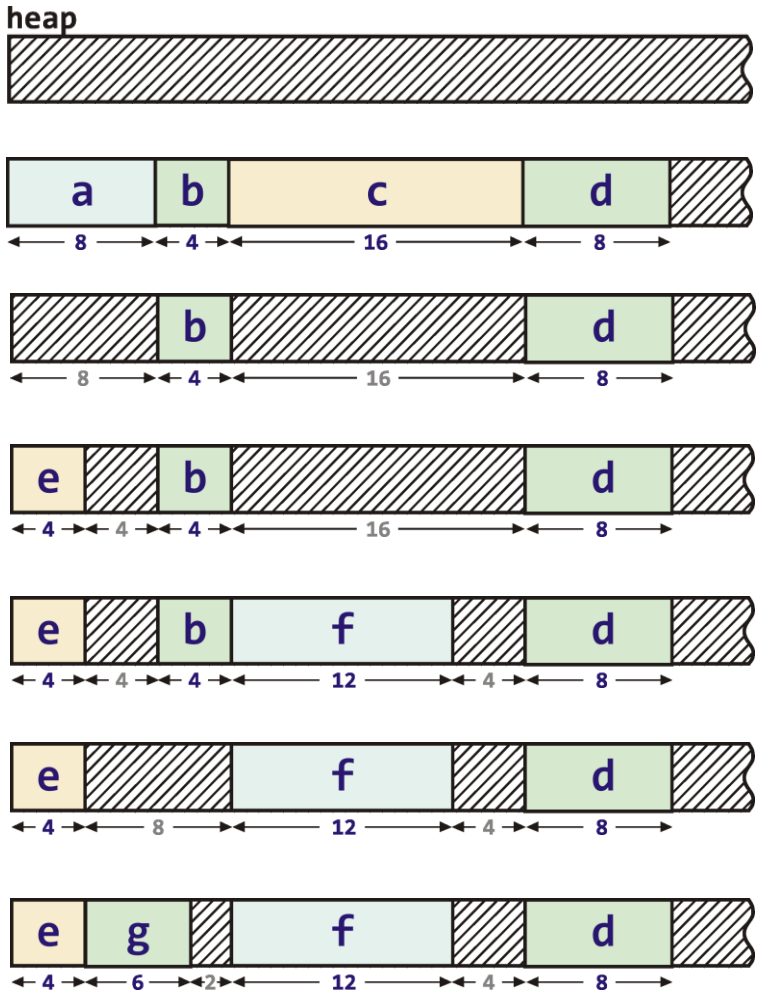
```
call f1
|
|  call f2
|  |
|  |  call f3
|  |  |
|  |  exit f3
|  |
|  exit f2
|
|  call f3
|  |
|  exit f3
|
exit f1
```



Dinamiskais atmiņas sadalījums

Dinamiskais sadalījums paredz manuālo mainīgo izveidošanu un dzēšanu ar speciālo komandu palīdzību. Tādu mainīgo glabāšanai ir paredzēts speciālais atmiņas apgabals, ko dēvē par **kaudzi (heap)**.

```
double a;
int b;
double c[2];
char d[8];
int e;
int f[3];
char g[6];
```



Dinamiskie mainīgie

Dinamiskais mainīgais – tas ir mainīgais, ko izveido un dzēš dinamiski programmas darba laikā ar speciālas komandas palīdzību. C++ valodā tam izmanto operatorus **new** un **delete.**

Izveidošana

```
ptr = new type;
```

Dzēšana

```
delete ptr;
```

Piemēram

```
int* a, * b;
a = new int;
b = new int;
printf("a = "); scanf("%i", a);
printf("b = "); scanf("%i", b);
printf("a + b = %i\n", *a + *b);
delete a;
delete b;
```

Dinamiskie masīvi

Dinamiskais masīvs atšķiras no statiskā vai automātiskā masīva ar to, ka tā izmērs ir jānoteic izveidošanas, bet ne deklarācijas laikā. Bet kā arī statiskā masīva, dinamiskā masīva izmēru nevar mainīt pēc izveidošanas.

Izveidošana

```
ptr = new type[size];
```

Dzēšana

```
delete[] ptr;
```

Piemēram

```
int* a, n;
printf("array size = "); scanf("%i", &n);
a = new int[n];
for (int i = 0; i < n; i++) a[i] = rand() % 100;
...
delete[] a;
```

Dinamiskās simbolu rindas

```
char* strNew(char* str) {
    return strcpy(new char[strlen(str) + 1], str);
}


void strDel(char* str) {
    delete[] str;
}

char* strGet(const char* msg = "Enter a string: ") {
    char s[100];
    printf("%s", msg); gets_s(s, 100);
    return strNew(s);
}

char* input = strGet();
printf("You entered <%s>\n", input);
strDel(input);
```

Dinamisko rindu masīvs

```
char* input[10];

for (int i = 0; i < 10; i++) {
    input[i] = strGet();
}

for (int i = 0; i < 10; i++) {
    printf("%2i: %s\n", i, input[i]);
}

for (int i = 0; i < 10; i++) {
    strDel(input[i]);
}
```

Divdimensiju dinamiskais masīvs

Deklarācija

```
int** array;
```

Izveidošana

```
int** create(int rows, int cols) {
    int** a = new int*[rows];
    for (int r = 0; r < rows; r++) a[r] = new int[cols];
    return a;
}
```

Dzēšana

```
void destroy(int** array, int rows) {
    for (int r = 0; r < rows; r++) delete[] array[r];
    delete[] array;
}
```

Iterēšana

```
void clear(int** array, int rows, int cols) {
    for (int r = 0; r < rows; r++)
        for (int c = 0; c < cols; c++) array[r][c] = 0;
}
```

Tipiskās kļūdas

Neinicializētu rādītāju dereference

```
int* a;
*a = 10; // UB
```

Darbs ar mainīgo pēc dzēšanas (*karājošs rādītājs*, *dangling pointer*)

```
int* a = new int;
*a = 10;
delete a; // a - dangling pointer
*a++; // UB
```

Atmiņas noplūde (*memory leak*)

```
int* a;
for (int i = 0; i < 100; i++) a = new int; // memory leak
*a = 10;
delete a;
```

Kaudzes fragmentēšana