**მიმთითებლები და მასივები**

**დინამიური მეხსიერება**

**მასივის “გადაგვარება”**

თქვენ უკვე იცით, თუ როგორ უნდა განსაზღვროთ ფიქსირებული მასივი:

|  |  |
| --- | --- |
| 1 | int array [ 5 ] = { 9 , 7 , 5 , 3 , 1 } ; // 5 ელემენტიანი მასივი |

ჩვენთვის, ზემოთ მოცემულია 5 მთელი რიცხვის მასივი, მაგრამ კომპილატორისათვის მასივი არის ცვლადი int [5] ტიპის. ჩვენ ვიცით, რა მნიშვნელობები აქვს array [0], array [1], array [2], array [3] და array [4] (შესაბამისად 9, 7, 5, 3 და 1).

ყველა, გარდა მხოლოდ ორ შემთხვევისა(რომელსაც ქვემოთ მოგახსენებთ), როდესაც ფიქსირებული მასივი გამოიყენება, ფიქსირებული მასივი **“გადაგვარდება“** (აშკარად გარდაიქმნება) მიმთითებლად რომელიც მიუთითებს მასივის პირველ ელემენტზე. მაგალითისათვის განვიხილოთ შემდეგი პროგრამა:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | #include <iostream>    int main ( )  {       int array [ 5 ] = { 9 , 7 , 5 , 3 , 1 } ;         // დავბეჭდოთ მასივის პირველი ელემენტის მისამართი       std :: cout << "Element 0 has address: " << &array [ 0 ] << '\n' ;         // დავბეჭდოთ მიმთითებლის მნიშვნელობა რომელადაც array გარდაიქმნება       std :: cout << "The array decays to a pointer holding address: " << << '\n' ;           return 0 ;  } |

დაიბეჭდება ( შევნიშნოთ რომ სხვადასხვა მანქანაზე მისამართი სხვადასხვა იქნება):

Element 0 has address: 0042FD5C

The array decays to a pointer holding address: 0042FD5C

არსებობს ცრუ მოსაზრება რომ C ++ ში, მასივი და მიმთითებელი მასივზე იდენტურია. ისინი იდენტურნი არ არიან! ზემოთ მოყვანილ შემთხვევაში, მასივი არის "int [5]" ტიპის, ხოლო მისი "მნიშვნელობა" არის თავად მასივის ელემენტები. მიმთითებელი მასივზე კი არის "int \*" ტიპის, ხოლო მისი მნიშვნელობა იქნება მასივის პირველი ელემენტის მისამართი.

ვნახოთ, სადაც გამოვლინდება ეს განსხვავება, თუმცა მასივის ყველა ელემენტზე წვდომა მიმთითებლების საშუალებითაც შესაძლებელია (უნდა გვქონდეს გავლილი თუმცა ამ საკითხს ჩვენ კიდევ შევეხებით), და შესაბამისად ჩვენ ხშირად შეგვიძლია ვისარგებლოთ ისეთი დიზაინით სადაც მიმთითებლებს მასივზე და ფიქსირებული ზომის მასივ ერთნაირ კონტექსტში გამოვიყენებთ, მაგრამ რაც უფრო მნიშვნელოვანია - კერძოდ თუ რამდენ ელემენტიანია მასივი, მასივის მიმთითებლისგან ვერ შევიტყობთ.

განვიხილოთ შემთხვევები როდესაც შვენ შეგვიძლია ვისარგებლოთ იმ ფაქტით როდესაც მასივი „ფუჭდება პოინტერად“ :

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | int array [ 5 ] = { 9 , 7 , 5 , 3 , 1 } ;    // განმისამართება (dereferencing) დაგვიბრუნებს მასივის პირველ ელემენტს  cout << \* array ; // დაბეჭდავს 9 -ს    char name [ ] = "Jason" ; // C-სტილის სტრინგი რომელიც ასევე მასივია  cout << \* name ; // დაბეჭდავს 'J' |

გაითვალისწინეთ, რომ ჩვენ *რეალურად* არ განვიხილავთ მასივს. მასივი (int [5] ტიპის) არაცხადად გარდაიქმნება მიმთითებლად მასივზე (int \*ტიპის), და ჩვენ ვახდენთ მიმთითებლის განმისამართებას რომ მივიღოთ ის მნიშვნელობა რომელიც იმ მისამართზეა რაც მიმთითებლის შიგთავს წარმოადგენს (მასში ინახება) ამ შემთხვევაში მასივის პირველი ელემენტის მნიშვნელობა.

ჩვენ ასევე შეგვიძლია მივანიჭოთ პოინტერს (მიმთითებელს) „მასივი“. კიდევ ერთხელ ავღნიშნავთ რომ ეს შესაძლებელია იმის გამო რომ მასივი „ფუჭდება“ int \* ტიპად შესაბამისად int \* -ს ვანიჭებთ int \* -ს:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | #include <iostream>    int main ( )  {  int array [ 5 ] = { 9 , 7 , 5 , 3 , 1 } ;           std :: cout << \* array ; // დაბეჭდავს 9             int \* ptr = array ;           std :: cout << \* ptr ; // დაბეჭწდავს 9    return 0 ;  } |

**განსხვავებები პოინტერსა და ფიქსირებულ მასივებს შორის**

არსებობს შემთხვევები, როდესაც ფიქსირებულ მასივსა და მიმთითებლებს შორიის სხვაობა თავის როლს თამაშობს და ცხადად ჩანს რომ ისინი ერთიდაიგივენი არ არიან:

ეს კარგად ჩანს sizeof () ოპერატორის გამოყენებისას. ფიქსირებულ მასივისათვის sizeof გამოყენებისას ბრუნდება მთლიანი მასივის ზომა (მასივის სიგრძე \* ელემენტის ზომა), ხოლო როდესაც მიმთითებელს იყენებთ, sizeof აბრუნებს მიმთითებლის ზომას (ბაიტებში) :

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | #include <iostream>    int main ( )  {  int array [ 5 ] = { 9 , 7 , 5 , 3 , 1 } ;    std :: cout << sizeof ( array ) << '\n' ; // გამოიტანს 5 \* 4             int \* ptr = array ;           std :: cout << sizeof ( ptr ) << '\n' ; // გამოიტანს 4    return 0 ;  } |

ამ პროგრამის ბეჭდავს:

20

4

ფიქსირებულმა მასივმა იცის რამდენ ელემენტიანია მასივი. მასივის მიმთითებელმა კი არა.

**ფიქსირებული მასივების ფუნქციისათვის გადაცემა (გადავხედოთ ზემოაღნიშნულის გათვალისწინებით)**

ალბათ გახსოვთ რომ იმის გამო, რომ დიდი მასივების კოპირება შეიძლება იყოს ძალიან ძვირი მეხსიერების თვალსაზრისით, C ++ არ ახორციელებს მთლიანი მასივის არგუმენტად გადაცემას, როგორც ზევით ავღნიშნეთ, მასივი „ფუჭდება“ პოინტერად და ისე გადაეცემა ფუნქციას. განვიხილოთ შემდეგი მაგალითი:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | #include <iostream>    void printSize ( int \* array )  {       // მასივი როგორც პოინტერი       std :: cout << sizeof ( array ) << '\n' ; // დაბეჭდავს პოინტერის ზომას და არა მასივის ზომას    int main ( )  {       int array [ ] = { 1 , 1 , 2 , 3 , 5 , 8 , 13 , 21 } ;       std :: cout << sizeof ( array ) << '\n' ; // დაბეჭდავს მასივის ზომას         printSize ( array ) ; // გადავეცით მასივი რომელიც „პოინტერად გარდაიქმნება“         return 0 ;  } |

გვექნება:

32

4

შეიძლება იფიქროთ რომ ეს მოხდა იმის გამო რომ printSize ( int \* array ) არგუმენტი იყო მიმთითებელი და არა მასივი, თუმცა ადრეა სიხარული ☺ იმ შემთხვევაშიც კი, თუ პარამეტრი გამოცხადებულია როგორც ფიქსირებული მასივი იგივე შედეგი გვექნება:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | #include <iostream>    // C++ არაცხადად გარდაქმნის მასივს მიმთითებლად  void printSize ( int array [ ] )  {       // პოინტერად აღიქმება და არა მასივად       std :: cout << sizeof ( array ) << '\n' ; // დაბეჭდავს პოინტერის ზომას და არა მასივის ზომას  }    int main ( )  {       int array [ ] = { 1 , 1 , 2 , 3 , 5 , 8 , 13 , 21 } ;       std :: cout << sizeof ( array ) << '\n' ; // დაბეჭდავს მასივის ზომას         printSize ( array ) ; // მასივი არგუმენტი გარდაიქმნება პოინტერად         return 0 ;  } |

გვექნება:

32

4

ზემოხსენებულ მაგალითში C ++ მოახდენს პარამეტრების არაცხადად გარდაქმნას მასივის სინტაქსიდან ([]) დან (\*) სინტაქსად. ეს კი იმას ნიშნავს რომ რომ შემდეგი ორი ფუნქციის დეკლარაცია იდენტურია:

|  |  |
| --- | --- |
| 1  2 | void printSize ( int array [ ] ) ;  void printSize ( int \* array ) ; |

ზოგიერთ პროგრამისტს ურჩევნია გამოიყენოს [] სინტაქსი, რადგან ეს ცხადყოფს, რომ ფუნქცია ელოდება მასივს, და არა მხოლოდ მიმთითებლის მნიშვნელობას. ამასთან, უმეტეს შემთხვევაში, იმის გამო, რომ მიმთითებელმა არ იცის თუ რამდენად „მასშტაბურია“ მასივი, თქვენ უნდა იზრუნოთ იმაზე რომ ფუნქციას გააყოლოთ მასივის ზომაც როგორც ცალკე პარამეტრი(სტრინგები გამონაკლისია, რადგან ისინი null-ით სრულდება).

*რეკომენდაცია: მასივის ფუნქციის პარამეტრად გადაცემისას უპირატესობა მიანიჭეთ მიმთითებლის სინტაქსს (\*) რათა ცხადად გესმოდეთ რომ* გარკვეული ოპერაციები, როგორიცაა sizeof (), იმუშავებს ისე, თითქოს პარამეტრი მიმთითებელს წარმოადგენს.

**სავარჯიშო:**

განიხილეთ შემდეგი მაგალითი :

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | #include <iostream>      void changeArray ( int \* ptr )  {       \* ptr = 5 ;  }    int main ( )  {       int array [ ] = { 1 , 1 , 2 , 3 , 5 , 8 , 13 , 21 } ;       std :: cout << "Element 0 has value: " << array [ 0 ] << '\n' ;         changeArray ( array ) ;         std :: cout << "Element 0 has value: " << array [ 0 ] << '\n' ;         return 0 ;  } |

რა გვექნება შედეგად? რატომ ?

**სტრუქტურებში (structs) და კლასებში მასივები არ გარდაიქმნება**

დაბოლოს, უნდა აღინიშნოს, რომ მასივები, რომლებიც სტუქტურის ან კლასის ნაწილია არ გარდაიქმნება როდესაც ხდება სტრუქტურის ან კლასის ფუნქციისათვის პარამეტრად გადაცემა. ამას უფრო დეტალურად კლასებზე საუბრისას განვიხილავთ.

**მიმთებლების (პოინტერების) არითმეტიკა**

C ++ ენა საშუალებას გაძლევთ შეასრულოთ მთელი რიცხვების დამატება ან გამოკლების ოპერაციები პოინტერებზე.  თუ ptr მთელ რიცხვზე მოუთითებს მიუთითებს, ptr + 1 არის მისამართი მეხსიერებაში ptr-ის შემდეგ 1 მთელი რიცხვის ზომით. ptr - 1 არის ptr-ის წინ 1 მთელი რიცხვის ზომის გამოკლებით მიღებული მისამართი.

ანუ თუ ჩავთვლით რომ ინტეჯერი 4 ბაიტია და ptr გვაქვს ინტზე(int\*) მაშინ ptr + 1  არის *მეხსიერების მისამართი* ptr შემდეგ 4 ბაიტის შემდგომ . განვიხილოთ შემდეგი მაგალითი რაც ცხადად დაგვანახებს ამ არითმეტიკას რომელიც **სკალირების** სახელითაა ცნობილი:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | #include <iostream>    int main ( )  {       int value { 7 } ;       int \* ptr { &value } ;         std :: cout << ptr << '\n' ;       std :: cout << ptr + 1 << '\n' ;       std :: cout << ptr + 2 << '\n' ;       std :: cout << ptr + 3 << '\n' ;         return 0 ;  } |

ჩვენს კონკრეტულ მანქანაზე შედეგი ასეთია:

0012FF7C

0012FF80

0012FF84

0012FF88

როგორც ხედავთ, ამ მისამართებიდან თითოეული განსხვავდება 4-ით (7C + 4 = 80 თექვსმეტობითში - მისამართები ხომ თექვსმეტობითში იწერება). ეს იმიტომ ხდება, რომ მთელი რიცხვი 4 ბაიტია ჩვენს მანქანაზე

იგივე პროგრამის თუ გამოყენებით short -ზე  :

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | #include <iostream>    int main ( )  {       short value { 7 } ;       short \* ptr { &value } ;         std :: cout << ptr << '\n' ;       std :: cout << ptr + 1 << '\n' ;       std :: cout << ptr + 2 << '\n' ;       std :: cout << ptr + 3 << '\n' ;         return 0 ;  } |

გვექნება

0012FF7C

0012FF7E

0012FF80

0012FF82

იმის გამო, რომ short 2 ბაიტია და თითოეული მისამართი განსხვავდება 2-ით.

**მასივები ასახულია მეხსიერებაში თანმიმდევრობით**

როგორც ვიცით მასივებს გამოეყოფა მეხსიერების ნაწილი რომელიც ერთადაა განლაგებული ერთმანეთის მიმდევრობით (juxtaposed – ერთმანეთის გვერდით ) ანუ 0, 1, 2,… ელემენტები ყველა ერთმანეთის მიმდებარედ არიან თანმიმდევრობით მოთავსებული. მართლაც:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | #include <iostream>    int main ( )  {       int array [ ] { 9 , 7 , 5 , 3 , 1 } ;         std :: cout << "Element 0 is at address: " << &array [ 0 ] << '\n' ;       std :: cout << "Element 1 is at address: " << &array [ 1 ] << '\n' ;       std :: cout << "Element 2 is at address: " << &array [ 2 ] << '\n' ;       std :: cout << "Element 3 is at address: " << &array [ 3 ] << '\n' ;         return 0 ;  } |

გვექნება:

ელემენტი 0 არის მისამართზე: 0041FE9C

ელემენტი 1 არის მისამართზე: 0041FEA0

ელემენტი 2 არის მისამართზე: 0041FEA4

ელემენტი 3 არის მისამართზე: 0041FEA8

შევნიშნოთ რომ თითოეული ელემენტის მისამართი 4 ბაიტითაა დაშორებული ერთმანეთისაგან, რაც არის მთელი რიცხვის ზომა გამტესტავ კომპიუტერზე.

შესაბამისად ვიცით რა როგორ მუშაობს პოინტერების არითმერიკა და ის თავისებურება რაც მასივის ელემენტებს გააჩნიათ( ერთმანეთის გვერდი-გვერდ არიან განლაგებული) ადვილი გასაგებია მასივზე მიმთითებელს ერთს თუ დავუმატებთ რატომ ავღმოჩნდებით მეორე ელემენტთან:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | #include <iostream>    int main ( )  {       int array [ ] { 9 , 7 , 5 , 3 , 1 } ;         std :: cout << &array [ 1 ] << '\n' ; // ბეჭდავს მეორე ელემენტის მეხსიერების მისამართს       std :: cout << array + 1 << '\n' ; // ბეჭდავს მეორე ელემენტის მეხსიერების მისამართს         std :: cout << array [ 1 ] << '\n' ; // ბეჭდავს 7       std :: cout << \* ( array + 1 ) << '\n' ; // ბეჭდავს 7         return 0 ;  } |

გაითვალისწინეთ, რომ მიმთითებლების არითმეტიკის სისწორისათვის აუცილებელია ფრჩხილები, რადგან ოპერატორს \* უფრო მეტი უპირატესობა აქვს ვიდრე ოპერატორს +.

სინამდვილეში კომპილატორი ოპერატორ [] - ის ინტერპრეტაციას ახდენს პოინტერის არითმერიკისა და განმისამართების(dereference) კომბინაციით : array[n] იგივეა, რაც \*(array + n) , სადაც n არაუარყოფითი მთელი რიცხვია რომელიც მასივის ინდექსს წარმოადგენს.

**მიმთითებლის გამოყენება მასივის ელემენტების იტერაციისათვის**

ჩვენ შეგვიძლია გამოვიყენოთ ისარი და მაჩვენებლის არითმეტიკა მასალის გასაღებად. მიუხედავად იმისა, რომ ჩვეულებრივ არ ხდება ამ გზით (ხელმოწერების გამოყენება ზოგადად უფრო ადვილია წაკითხვისა და შეცდომებისადმი ნაკლებად მიდრეკილება), შემდეგ მაგალითზე მოცემულია შესაძლებელი:

#include <iostream>

using namespace std;

int main()

{

int array []{ 9, 7, 5, 3, 1 };

int \* ptr = array;

for(int \* ptr = array; ptr < array+sizeof(array)/sizeof(array[0]); ptr++)

{

cout<< \*(ptr) << " ";

}

return 0;

}

დავალება განიხილეთ შემდეგი კოდი , რას აკეტეებს და რატომ?

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | #include <iostream>  #include <iterator> // მასივის ზომის (size) გამოყენებისათვის   using namespace std;  bool isVowel ( char ch )  {     switch ( ch )     {     case 'A' :     case 'a' :     case 'E' :     case 'e' :     case 'I' :     case 'i' :     case 'O' :     case 'o' :     case 'U' :     case 'u' :       return true ;     default :       return false ;     }  }    int main ( )  {     char name [ ] { "Mollie" } ;     int arrayLength { static\_cast < int > (size ( name ) ) } ;     int numVowels { 0 } ;       for ( char \* ptr { name } ; ptr < ( name + arrayLength ) ; ++ ptr )     {       if ( isVowel ( \* ptr ) )       {         ++ numVowels ;       }     }    cout << name << " has " << numVowels << " vowels.\n" ;       return 0 ;  } |

count\_if -ით თუ ჩავანაცვლებთ for ციკლს? ანუ ჩავანაცვლებთ ასე

auto numVowels { count\_if (begin ( name ) , end ( name ) , isVowel ) } ;

რა გვექნება? იმუშავებს მასივზე? იმუშავებს მასივის პოინტერზე?

**კითხვა 1**

რატომ მუშაობს შემდეგი კოდი?

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | #include <iostream>    int main ( )  {     int arr [ ] { 1 , 2 , 3 } ;       std :: cout << 2 [ arr ] << '\n' ;       return 0 ;  } |

**დავალება 2**  
ჩაწერეთ ფუნქცია სახელწოდებით find, რომელიც პარამეტრად იღებს მიმთითებელს მასივის დასაწყისზე, მიმთითებელს მასივის დასასრულზე და მნიშვნელობას რომელიც უნდა მოძებნოს მასივში. მან უნდა დააბრუნოს მიმთითებელი პირველივე ნაპოვნ მნიშვნელობაზე ან თუ ასეთი მნიშვნელობა არ მოიძებნა მასივში მიმთიტებელი ბოლო ელემენტზე . გამოიყენეთ შემდეგი კოდი თქვენი ფუნქციის გასატესტად:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | #include <iostream>  #include <iterator>    // ... აქ თქვენი ფუნქციააა    int main ( )  {       int arr [ ] { 2 , 5 , 4 , 10 , 8 , 20 , 16 , 40 } ;         // მოძებნეთ ელემენტი მნიშვნელობით 20.       int \* found { find ( std :: begin ( arr ) , std :: end ( arr ) , 20 ) } ;         // თუ ვიპოვეთ ვბეჭდავთ.       if ( found != std :: end ( arr ) )       {           std :: cout << \* found << '\n' ;       }         return 0 ;  } |

**მითითება:**std::begin და std::end აბრუნებს int\* . find -ის გამოძახება ექვივალენტურია შემდეგის

|  |  |
| --- | --- |
|  | int \* found { find ( arr , arr + std :: size ( arr ) , 20 ) } ; |

პასუხი კითხვა 1

arr[2] იგივეა რაც \*(arr + 2) იგივეა რაც \*(2 + arr) იგივეა რაც 2[arr]

პასუხი დავალება 2

int\* find(int\* begin, int\* end, int value)

{

    for (int\* p{ begin }; p != end; ++p)

    {

        if (\*p == value)

        {

            return p;

        }

    }

    return end;

}

**დინამიური მეხსიერების გამოყოფა C ++ ში როგორ მუშაობს new და delete ოპერატორები**

**შესავალი**

*C ++ მხარს უჭერს მეხსიერების განაწილების სამ ძირითად ტიპს, რომელთაგან ორი უკვე გინახავთ სავარაუდოდ:*

**სტატიკური მეხსიერების გამოყოფა** *ხდება სტატიკური და გლობალური ცვლადისთვის. ამ ტიპის ცვლადისთვის მეხსიერება გამოიყოფა ერთხელ, როდესაც თქვენი პროგრამა ეშვება და მთელი პროგრამის განმავლობაში ნარჩუნდება.*

**მეხსიერების ავტომატური გამოყოფა** *ხდება ფუნქციის პარამეტრებისა და ლოკალური ცვლადისთვის. მეხსიერების ამ ტიპის გამოყოფა ხდება იმდენჯერ რამდენჯერაც საჭიროა შესაბამისი ბლოკის (მაგალითად ფუნქციის გამოძახების) შესრულების დაწყებისათვის და თავისუფლდება ბლოკის დასრულებისას.*

*როგორც სტატიკურ, ასევე ავტომატურ გამოყოფას ორი რამ აქვთ საერთო:*

1. ცვლადის / მასივის ზომა უნდა იყოს ცნობილი კომპილაციის დროს.
2. მეხსიერების გამოყოფა და გაცლა ხდება ავტომატურად (ცვლადის გამოცხადებისას / ცვლადის განადგურებისას).

*უმეტეს შემთხვევაში ასეთი მიდგომა საკმარისია, თუმცა თქვენ შეხვდებით სიტუაციებს, როდესაც ზემოაღნიშნული ორი შეზღუდვის გამო გვექმნება პრობლემა, ეს თავს იჩენს განსაკუთრებით მაშინ როდესაც ინფუთი მოდის გარედან : მომხმარებლისგან ან ფაილიდან.*

*მაგალითად, შეიძლება დაგვჭირდეს რიცხვითი მასივის შექმნა, მომხმარებლის მონაცემების შესანახად მაგრამ წინასწარ არ ვიცით რამდენ მონაცემს მოგაწვდის ის, ან ჩანაწერების წაკითხვა გვინდა ფაილიდან და არ ვიცით რამდენი ჩანაწერია ფაილში. თამაშს ვწერთ სადაც „მტრების“ (რომლებიც მომხმარებელს თავს ესხმიან ) ცვალებადი რაოდენობა გვაქვს.*

*თუ ჩვენ გამოვაცხადებთ ყველაფრის ზომას კომპილაციის დროს, უკეთეს შემთხვევაში შეგვიძლია მიახლოებით ვივარაუდოთ მაქსიმალური ზომა შესაძლო მნიშვნელობებისათვის და იმედი ვიქონიოთ მეტი არ დაგვჭირდება , მაგალითად*

int nums[100]; // ვიმედოვნებთ 100-ზე მეტს არ შემოიტანს!  
Record record[500]; // ვიმედოვნებთ 500-ზე მეტი ჩანაწერი არაა!  
Enemy enemy[40]; // 40 მტერია მაქსიმუმ

ეს არის ცუდი გამოსავალი მინიმუმ ოთხი მიზეზის გამო:

პირველი, ეს იწვევს მეხსიერების ფლანგვას. მაგალითად, თუ ჩვენ გამოვყოფთ 100 ელემენტს ხოლო ჩვენს მომხმარებელს საშუალოდ 10 მნიშვნელობა შემოაქვს? ჩვენ ათჯერ მეტ მეხსიერებას მოვითხოვთ და არ ვიყენებთ, წარმოიდგინეთ ეს ინტების მასივი კი არა ობიექტების მასივია და თანაც ეს ობიექტები საკმაოდ წონადია (რამდენიმე ბაიტი კი არა რამდენიმე ათეული ან ასეული ბაიტია და თან მასივი ათასელემეტინია...) .

მეორე, როგორ განვასხვაოთ რომელი მეხსიერების ნაწილია გამოყენებული და რომელია უქმად? მაგალითად enemy[20] ცოცხალია თუ მკვდარი? ანუ უნდა ვიზრუნოთ იმაზე რომ აქტიური ელემენტები არააქტიურისაგან განვასხვაოთ რაც კოდს უფრო რთულს გახდის.

მესამე, ყველა ჩვეულებრივი ცვლადი (ფიქსირებული მასივების ჩათვლით) გამოიყოფა მეხსიერების ნაწილში, რომელსაც უწოდებენ სტეკს (stack). ზოგადად პროგრამისათვის ამ ტიპოს მოცულობა საკმაოდ შეზღუდულია, მაგალითად ვიზუალ სტუდიოში ასეთი მეხსიერების ნაგულისხმევი ზომა (default) 1 მბ. თუ ამ რიცხვს გადავაჭარბებთ, მოხდება გადავსება (stack overflow) და პროგრამა შეწყდება. მხოლოდ 1 მბ მეხსიერებით შემოფარგლვა პრობლემური იქნება მრავალი პროგრამისთვის, განსაკუთრებით მათთვის, რომლებიც გრაფიკას იყენებს.

მეოთხე და ყველაზე მნიშვნელოვანია შეიძლება ამან გამოიწვიოს ხელოვნური შეზღუდვების დაწესება ან მასივის გადავსება : რა ხდება, როდესაც მომხმარებელი ცდილობს დისკზე 600 ჩანაწერის წაკითხვას, მაგრამ ჩვენ მხოლოდ გამოვყავით მეხსიერება მაქსიმუმ 500 ჩანაწერისთვის? ან მომხმარებელს უნდა გამოვუტანოთ შეცდომა და ვუთხრათ რომ მხოლოდ 500 ჩანაწერი წაიკითოს, ან (უარეს შემთხვევაში, როდესაც ასეთ შემთხვევას საერთოდ არ გავითვალისწინებთ) გადაივსება ჩანაწერების მასივი და ვნახავთ საკმაოდ „ცუდ“ მოვლენებს.

საბედნიეროდ, ამ პრობლემებს მარტივად უმკლავდება დინამიური მეხსიერების ფენომენი. დინამიური მეხსიერების განაყოფა არის საშუალება პროგრამების გაშვებისას, საჭიროების შემთხვევაში, ოპერაციული სისტემიდან გამოვითხოვოთ მეხსიერება. ეს მეხსიერება არ მოდის სტეკ მეხსიერებიდან, ის გამოიყოფა მეხსიერების უფრო დიდი აუზიდან, რომელსაც მართავს ოპერაციული სისტემა და ჰიპ (heap) მეხსიერაბადაა ცნობილი. თანამედროვე აპარატებზე, ამ ტიპის მეხსიერების ზომა შეიძლება გიგაბაიტიც იყოს.

**დინამიური მეხსიერების გამოყოფა C ++ ში**

დინამიური მეხსიერების გამოყოფა(დმგ) C++ საშუალებას გვაძლევს ჩვენი პროგრამის კომპონენტებისათვის მეხსიერება გამოვყოთ პროგრამის გაშვების დროს (run-time). ეს საშუალება ერთერთი მნიშვნელოვანი ინსტრუმენტია C++ში. თუმცა ის გარკვეულ საფრთხეებთანაც არის დაკავშირებული, მაგალითად მეხსიერების „გაჟონვის“(memory leak) პრობლემესთან.

C++ ში დინამიური მეხსიერების გამოყოფა განსხავდება C- სგან, თუმცა ისეთი ფუნქციები როგორიცაა **malloc()**, **calloc()**, **realloc()** და **free()** მუშაობს C++ შიც, თუმცა მათი გამოყენება არაა რეკომენდირებული, სამაგიეროდ გვაქვს 2 ოპერატორი:  **new და delete დინამიური მეხსიერების მართვისათვის.**

შევნიშნოთ რომ დმგ შეუძლია თავად მომხმარებელს, თუმცა მისი მენეჯმენტზე სრულად მომხმარებელია პასუხისგებელი რადგან C++ არ გვაქვს garbage collector - ის ცნება, რომელიც უმიზნოდ დაკავებული მეხსიერების გამონთავისუფლებაში დაგვეხმარებოდა, ამიტომ მომხმარებელმა თავად უნდა იზრუნოს იმაზე რომ ასეთი მეხსიერება გამოანთავისუფლოს.

დინამიური მეხსიერების გამოყობა ხორციელდება მეხსიერების ნაწილში რომელსაც „ჰიპ“-ს (**Heap**) უწოდებენ, ხოლო არა-სტატიკური მეხსიერება და ლოკალური ცვლადები ინახება მეხსიერების ნაწილში რომელსაც „სტეკი“(**Stack**) ეწოდება[[1]](#footnote-1).

**განვიხილოთ მეხსიერების პირობითი რუკა რომელიც ქვემოთაა მოთავსებული:**

|  |  |
| --- | --- |
| **პროგრამა - კოდის სეგმენტი** | **გლობალური ცვლადები - მონაცემების სეგმენტი** |
| **სტეკი**  (გამოიყენება ფუნქციების გამოძახებების დაბრუნებისათვის, მისამარტებისათვის, არგუმენტებისა და ლოკალური ცვლადებისატვის) | |
| **ჰიპი**  (გამოიყენება დინამიური მეხსიერების გამოყოფისათვის) | |

**როგორც ვხედავთ** C++ში გვაქვს მეხსიერების 4 განსხვავებული უბანი.

1. **პროგრამა :**  - **კოდის სეგმენტი** ინახავს პროგრამის გაკომპილირებულ კოდს. აქ ხდება პროგრამის ყველა ფუნქციის ინიცირება (გარკვეული მისამართიდან).
2. **გლობალური ცვლადები (მონაცემების სეგმენტი):** ამ მეხსიერების უბანში ინახება გლობალური ცვლადები, პროგრამის მტელი სასიცოცხლო ციკლის მიმდინარეობისათვის.
3. **სტეკი :**  - ეს უბანი პასუხისმგებელია ფუნქციების გამოძახებებიდან უკან დასაბრუნებელი მისამართების შენახვაზე. ფუნქციისათვის გადასაცემ არგუმენტებზე, ლოკალურ ცვლადებზე. შევნიშნოთ რომ ლოკალური ცვლადები რჩება მეხსიერებაში ფუნქციის სიცოცლხის ციკლის განმავლობაში, ფუნქციის დასრულებისთანავე ისინი მეხიერებიდან იშლება. ამ ყველაფრის გარდა, სტეკი პასუხისმგებელია პროცესორის მიმდინარე მდგომარეობის შენახვაზე.
4. **ჰიპი:**  - ასევე ცნობილია როგორც "თავისუფალი საცავი“ - პასუხისმგებელია დინამიური მეხსიერების გამოყოფაზე.

**ერთი ცვლადისათვის მეხსიერების დინამიურად გამოყოფისათვის ვიყენებთ ოპერატორს new შემდეგი ფორმით:**

new int; /

ზემოთ მოცემულ ინსტრუქციაში ჩვენ ვითხოვთ მთელი რიცხვის შესანახი ზომის (4 ბაიტი ძირითადად) მეხსიერებას ოპერაციული სისტემისგან. ოპერატორი new ქმნის ობიექტს ამ მეხსიერების ფარგლებში(4 ბაიტი) და აბრუნებს მისამართს ამ მეხსიერებისა.

როგორც წესი ჩვენ ვქმნით საკუთარ პოინტერს და მასში ვათავსებთ ამ დინამიურ მეხსიერებაში შექმნის ცვლადს:

int \*ptr = new int;

ასეთ ცვლადებს ანონიმურ ცვლადებს უწოდებენ რადგან მათ სახელი არ აქვთ, ხოლო წვდომა მათზე შესაძლებელია მხოლოდ ptr მიმთითებლის გამოყენებით:

\*ptr = 5; // ანონიმურ ცვლადს მივანიჭეთ მნიშვნელობა 5 პოინტერის საშუალებით

როგორც ვხედავთ დინამიურ მეხსიერებაზე წვდომა მხოლოდ მიმთითებლის საშუალებითაა შესაძლებელი.

**როგორ მუშაობს დინამიური მეხსიერების გამოყოფა?**

თქვენს კომპიუტერს აქვს მეხსიერება (ოპერატიული მეხსიერება) რომელიც ხელმისაწვდომია პროგრამებისათვის. პროგრამის გაშვებისას ის იტვირთება ოპერატიულ მეხსიერებაში, ამ პროცესს ოპერაციული სისტემა კურირებს. თუ რომელი მეხსიერების ნაწილში პროგრამის რა ნაწილს განათავსებს დამოკიდებულია მათ ტიპზე . სად რა თავსდება მოკლედ ზემოთ უკვე მოგახსენეთ. როდესაც თქვენ ოპერაციულ სისტემას სთხოვთ მეხსიერების დინამიურად გამოყოფას თქვენ სინამდვილეში სთხოვთ მას შეგინახოთ ეს მეხსიერება თქვენი პროგრამის აპლიკაციისათვის. თუ მას შეუძლია ამ თხოვნის დაკმაყოფილება ( შესაძლებელია ისეც მოხდეს რომ იმ კონკრეტულ დროს როცა მოითხოვთ საკმარისი თავისუფალი მეხსიერება ოპერაციულ სისტემას არ ქონდეს.) თქვენს აპლიკაციას უკვე შეუძლია მიღებული მეხსიერების „ საკუთარ ნებაზე „ გამოყენება. როცა აპლიკაცია დაასრულებს თავის საქმეს მან წესით უნდა დაუბრუნოს მეხსიერება უკან ოპერაციულ სისტემას რომ მან შეძლოს მისი სხვა აპლიკაციისათვის გადაცემა . აქ მნიშვნელოვანია ვიცოდეთ რომ სტატიკური ან ავტომატური მეხსიერებისგან განსხვავებით, თავად პროგრამაა პასუხისმგებელია დინამიურად გამოყოფილი მეხსიერების მოთხოვნასა და განკარგვაზე შესაბამისად მან უნდა იზრუნოს იმაზე რომ მეხსიერება უკან დააბრუნოს.

**დინამიურად გამოყოფილი ცვლადის ინიციალიზაცია**

როდესაც დინამიურად გამოყოფთ ცვლას, ასევე შეგიძლიათ მისი ინიციალიზაცია უშუალო ინიციალიზაციით ან ერთიანი ინიციალიზაციით (C ++ 11):

int \* ptr1 = new int (5); // გამოიყენეთ პირდაპირი ინიციალიზაცია

int \* ptr2 = new int {6}; // გამოიყენეთ ერთიანი ინიციალიზაცია

**ცვლადის წაშლა**

როდესაც დავასრულებთ დინამიურ ცვლადთან მუშაობას ჩვენ უნდა ცხადად ვუთხრათ კომპილატორს რომ გაანთავისუფლოს დარეზერვებული მეხსიერება , რათა მისი ხელახალი გამოყენება იყოს შესაძლებელი, ეს კი შემდეგნაირად ხორციელდება:

delete ptr; // ვაბრუნებთ დარეზერვებულ ptr- ით მითითებულ მეხსიერებას უკან

ptr = 0; // set ptr to be a null pointer (use nullptr instead of 0 in C++11) // ვარაუდობენ, რომ ptr ადრე გამოიყო ოპერატორ ახალთან

ptr წაშლა; // დაუბრუნეთ ოპერაციულ სისტემას

ptr = 0; // დააყენეთ ptr null მაჩვენებელზე ან გამოიყენეთ nullptr 0-ის ნაცვლად C ++ 11-ში

შევნიშნოთ რომ delete ოპერატორი სინამდვილეში არაფერს შლის. ის უბრალოდ დაუბრუნებს მეხსიერებას ოპერაციუილ სისტემას რათა მან შეძლოს მისი გამოყენება და სხვა აპლიკაციებისათვის გადაცემა. მიუხედავად იმისა, რომ ჩანს თითქოს ჩვენ ვშლით ცვლადს, სინამდვილეში ჩვენ მხოლოდ ვცლით მას, მას ჩვეულებრივად შეგვიძლია მივანიჭოთ ახალი მნიშვნელობა.

1. სტეკი იყენებს სტეკ მონაცემთა სტრუქტურის თავისებურებას (LIFO) ფუნქციების გამოძახებების მართვისათვის. [↑](#footnote-ref-1)