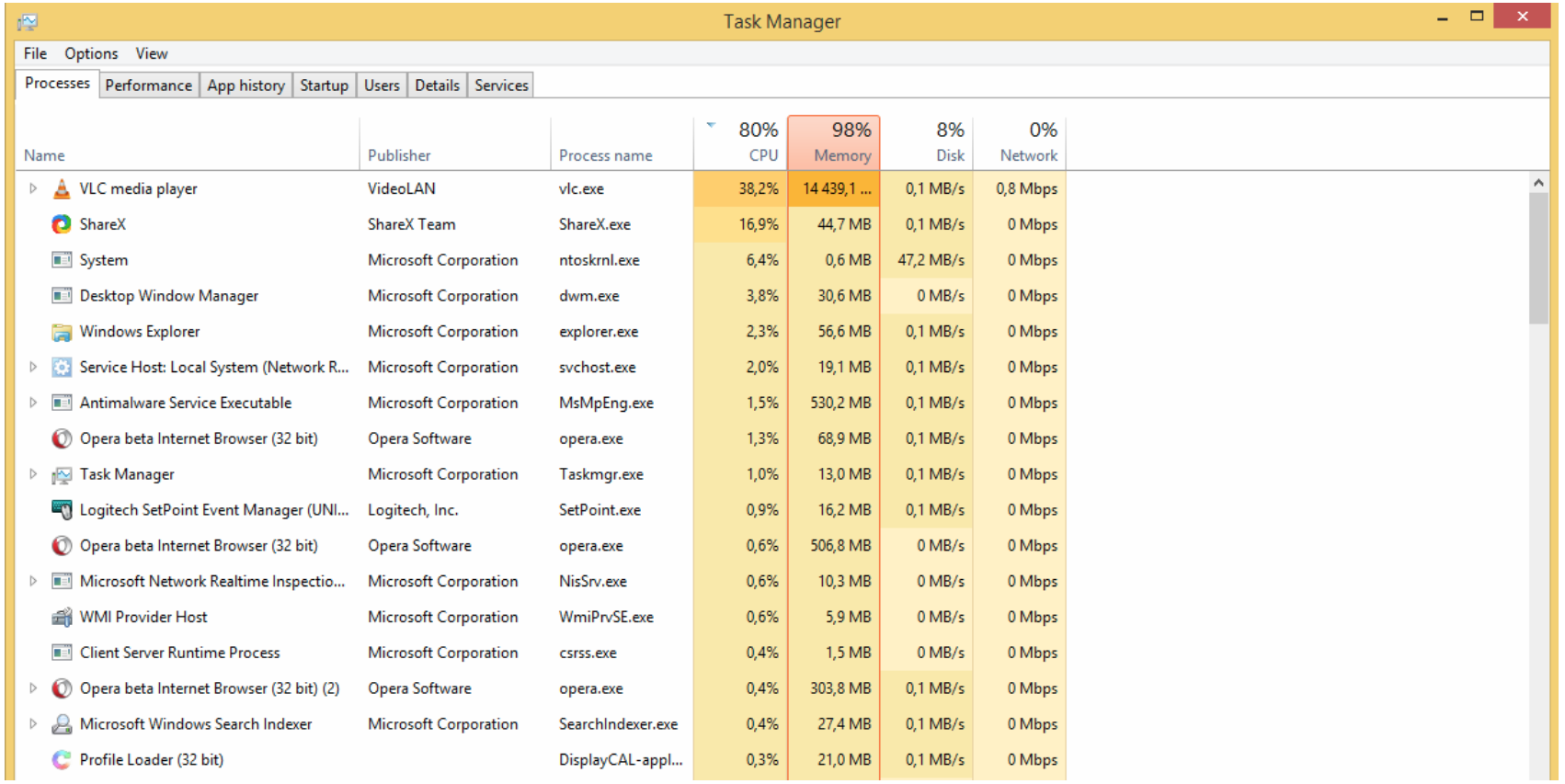
Dünaamiline mälujaotus

MALLOC(), CALLOC() JA FREE()

Ebaõnnestumise hind



Dünaamilisest mälust

Dünaamiline mälu võimaldab programmi töö ajal mälu küsida

- Saab küsida mälu siis kui meil seda vaja on
- Saab vabastada mälu, mida me enam ei vaja
- Saab muuta mälumahtu, mida rakendus tohib kasutada

Staatiliste massiivide asemel saame kasutada dünaamilisi massiive

- Küsides mälu kuhjast (*heap*) saame kasutada oluliselt rohkem mälu programmi tarvis võrreldes pinuga (*stack)*

Eraldatav mälu on järjestikune

- Saame kasutada viida-aritmeetikat või massiivide indekseerimist
- Mälu killustatus tekitab probleeme

Tekivad täiendavad ohud

- Kaotatud viidad, vabastamata mälu – **mälulekked** *(memory leak)*
- NULL- ja kehtetute viitade ligipääs põhjustab rakenduse kokku jooksmist (*segmentation fault*)

Muutuja eluiga

Lokaalmuutujad- ja massiivid omavad piiratud eluiga

Kehtivus vaid samas **skoobis** kus nad on **deklareeritud**

Mälu tuleb **pinust *(stack)*,** mille maht on piiratud (tavaliselt 256 KB – 8 MB, OSist sõltuv, muudetav)

Lokaalselt deklareeritud massiivi tagastada ei saa

Dünaamiliselt küsitud mälu on kättesaadav kuniks vabastamiseni

Viitmuutuja eluiga on piiratud, kuid viidatav mäluala on kasutatav **kuniks me teame selle asukohta**

Dünaamiline mälu tuleb **kuhjast** (*heap)*, saame kasutada kogu arvuti vaba mälu*

Dünaamilist massiivi saab tagastada funktsioonist viidana

Plussid ja miinused

PLUSSID

Saame küsida suuri mäluplokke

Saab kogu programmi raames

Saab kasutada kuniks vabastame

Saab tagastada mälu mida ei kasuta

Saab muuta kasutatavat mälumahtu

-> vabadus

MIINUSED

Vaja viitadel silm peal hoida (ning mitte kaotada!)

Täiendav mälu viitade hoiustamiseks

Ühes tükina küsitud mälu peab olema järjestikune

Mälu tuleb käsitsi vabastada

Mälu küsimine võib ebaõnnestuda

-> vastutus

Dünaamilise mälu kasumine ilma põhjuseta ei ole mõistlik!

Kas on vaja dünaamilist mälu?

JAH

Massiivi pikkus ei ole eelnevalt teada

Massiivi pikkus on varieeruv igal käivitusel

Massiivi pikkus võib programmi käitlemise ajal oluliselt muutuda

Massiiv vajab palju mälu (>100 KiB)

Vaja mälu (massiiv) tagastada funktsioonist

Vaja kasutada abstraktseid andmestruktuure*

EI

Ajutised muutujad

Mälumaht on alati sama

- Nt: Eesti isikukood sõnena

Kasutatav mälumaht on väike

- Üksik muutuja, väike lokaalne massiiv

Rakenduse fookuses kiirus, keerukuse vähendamine

Sardsüsteemid*

Sarnasus failide kasutamisele
`fopen()`, `fclose()`, `NULL`-viida kontroll

Dünaamilise mälu kasutamine

1. Lisa **`<stdlib.h>` teek**
2. Deklareeri sobiliku andmetüübi viit
3. Leia mitu baiti mälu vajad
4. Küsi vajaminev kogus mälu ja salvesta viit saadud mälule
5. Kontrolli, kas mälu küsimine ikka õnnestus (jätka kui jah; kui ei, siis …)

   Ebaõnnestumise korral tagastati NULL-viit
6. Kasuta mälu kuniks seda vajad
7. Vabasta küsitud mälu kui seda enam ei vaja

   Vabastada tuleks kindlasti enne rakenduse sulgemist, kuid võimalusel ka töö kestel

Muutuja suurus

Andmetüübi suurus on sageli varieeruv

- Enamike lihtsate andmetüüpide puhul defineeritakse **miinimum**, tegelik on sageli suurem
  Nt: `int` on standardi kohaselt vähemalt 2 baiti (maksimaalne väärtus 32 767)
- Andmetüübi suurused sõltuvad mikroprotsessori arhitektuurist, kompilaatorist, operatsioonisüsteemist, seadistustest
- Ka viitade suurused on varieeruvad
- Struktuurid polsterdatakse muutujate joondamiseks
- Ühendi suurus on selle kõige suurema liikme suurus

Kasuta `sizeof` operaatorit vajaliku mälumahu leidmiseks!

malloc() funktsioon

```
void* malloc (size_t size);
```

Parameetrid:

- `size` – mitu baiti mälu vaja

Tagastatav väärtus:

- `void` viit hõivatud mäluala algusele
- `NULL` viit juhul kui mälu hõivamine ebaõnnestus

Saadav mälu **on algväärtustamata!**

Näide:

```
pointer = (datatype *)malloc(sizeof(datatype) * n)
```

Tüübiteisendus on **valikuline** C keeles, **kohustuslik** C++ keeles

calloc()

```
void* calloc (size_t num, size_t size);
```

Parameetrid

- `num` – mitu elementi soovid
- `size` – mitu baiti on üks element

Tagastatav väärtus:

- `void` viit hõivatud mäluala algusele
- `NULL` viit juhul kui mälu hõivamine ebaõnnestus

Saadav mälu **on algväärtustatud nullideks!**

Tüübiteisendus on **valikuline** C keeles, **kohustuslik** C++ keeles

Kasutamine:

```
pointer = (datatype *)calloc(n, sizeof(datatype));
```

Staatiline vs dünaamiline massiiv

Staatiline massiiv

```
int num[10];

int num2[N];

int num3[N] = {0};
```

Dünaamiline massiiv

```
int *pNum;
pNum = (int *)malloc(10 * sizeof(int));

int *pNum2 = (int *)malloc(N * sizeof(int));

int *pNum3 = (int *)calloc(N, sizeof(int));
```

`malloc()` annab algväärtustamata mälu
`calloc()` algväärtustab nullideks

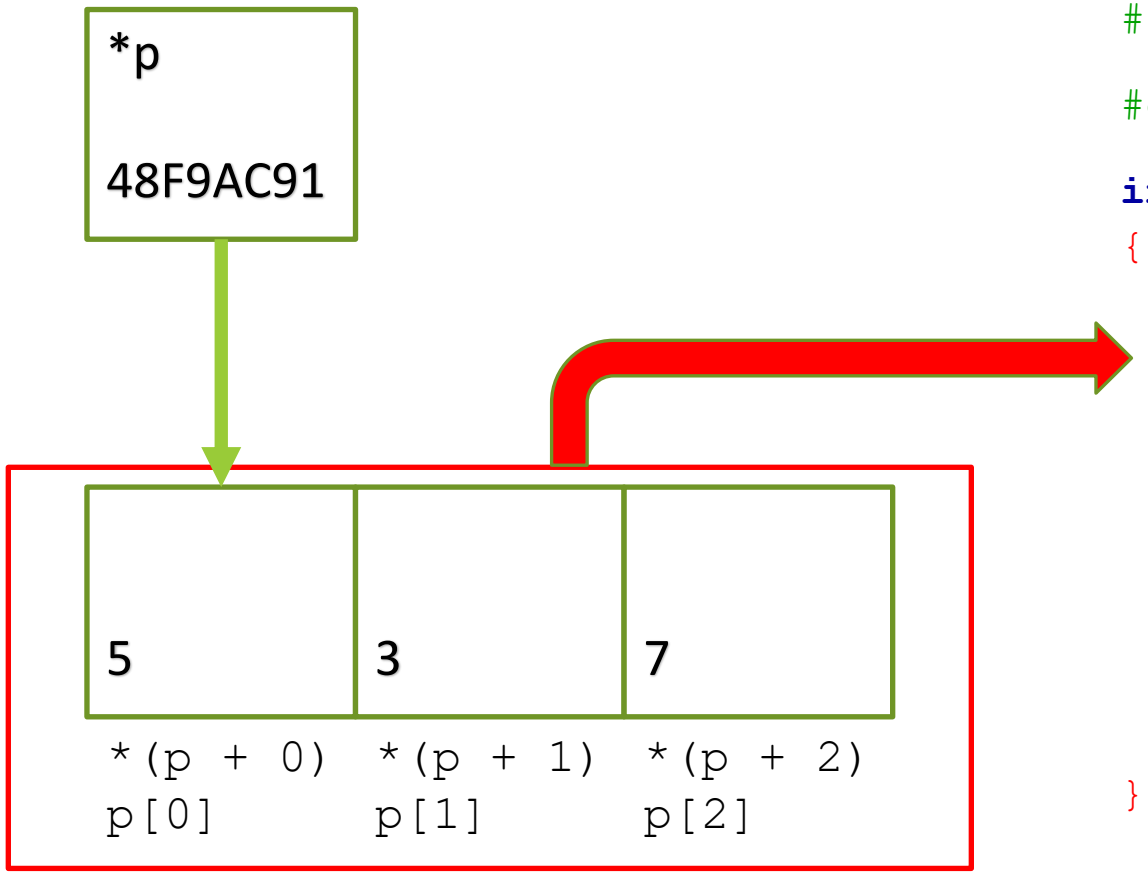
Näide: töövoog

```c
#include <stdio.h>
#include <stdlib.h>

#define N 5

int main(void)
{
    int *pNumbers = (int *)malloc(N * sizeof(int));

    if (pNumbers == NULL)
    {
        fprintf(stderr, "Not enough memory!\n");
        exit(EXIT_FAILURE);
    }

    /* work with the data */

    free(pNumbers);
    return EXIT_SUCCESS;
}
```

1. Lisa **<stdlib.h> teek**
2. Deklareeri sobiliku andmetüübi viit
3. Leia mitu baiti mälu vajad
4. Küsi vajaminev kogus mälu ja salvesta viit saadud mälule
5. Kontrolli, kas mälu küsimine ikka õnnestus
6. Kasuta mälu kuniks seda vajad
7. Vabasta küsitud mälu kui seda enam ei vaja

NB! Pole vahet kas kasutad
`p[i]` või `*(p + i)`

Meenutus viidaaritmeetikast




```c
#include <stdio.h>
#include <stdlib.h>

#define N 3

int main(void)
{
    int *p, i;
    p = (int *)malloc(N * sizeof(int));
    *(p + 0) = 5;
    *(p + 1) = 3;
    *(p + 2) = 7;
    for (i = 0; i < N; i++)
        printf("%p %d\n",(p + i), *(p + i));
    free(p);
    return EXIT_SUCCESS;
}
```

Näide: dünaamilise massiivi tagastamine

```c
#include <stdio.h>
#include <stdlib.h>

#define LEN 10

int *Foo(int n);

int main(void)
{
    int *arr;
    arr = Foo(LEN);

    free(arr);
    return EXIT_SUCCESS;
}
```

```c
int *Foo(int n)
{
    int *pData = (int *)malloc(sizeof(int) * n);
    if (pData == NULL)
    {
        fprintf(stderr, "Not enough memory!\n");
        exit(EXIT_FAILURE);
    }
    return pData;
}
```