Dynamisch geheugen beheer

+ Normaal wordt plaats in het werkgeheugen gereserveerd tijdens de compilatie aan de hand van de declaraties van de variabelen.
+ Deze geheugenreservering is statisch: in het bronbestand van het programma wordt reeds vastgelegd hoe groot bijvoorbeeld een array zal zijn.
+ Hieraan is tijdens de uitvoering van het programma niets meer te veranderen.
+ In plaats daarvan kan geheugen meer dynamisch gereserveerd worden. Hiervoor zijn functies ter beschikking om geheugen tijdens run-time aan te vragen en eventueel later weer vrij te geven.

```
void *malloc( size_t grootte );
void free( void *p );
```

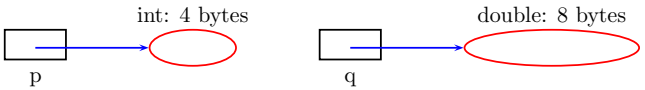
+ Om deze functies te gebruiken moet een include van het `malloc.h` headerbestand gebeuren.

- malloc: memory allocatie;
- argument grootte van de ruimte die men wenst te alloceren, uitgedrukt in aantal bytes.
- Indien de uitvoering van de functie succesvol verloopt, wordt het adres teruggegeven van het begin van de toegekende ruimte. Deze ruimte bestaat uit een opeenvolging van een aantal bytes gelijk aan `grootte`.
  Indien er iets misgaat, wordt een `NULL` waarde teruggeven.
- Returnwaarde van `malloc` is een adres van een geheugenruimte dat zelf geen naam heeft. Deze returnwaarde wordt toegewezen aan een pointer variabele, zodat de anonieme variabele via die pointer gebruikt kan worden.
- free: met argument zo'n pointer variabele die via `malloc` een waarde toegewezen gekregen heeft.
- Het effect van de functie is dat de ruimte waarnaar de pointer wijst, vrij gegeven wordt. Deze ruimte kan dan achteraf door een oproep van malloc terug gealloceerd worden (om voor iets anders gebruikt te worden).
- Functie heeft geen return-waarde, dus als een *procedure* te bestempelen.

- Afhankelijk van de toepassing zal de gealloceerde ruimte data bevatten van een bepaald type.
- Omdat tijdens de uitvoering van `malloc` geen informatie ter beschikking is omtrent het type van de anonieme variabele, wordt door `malloc` een pointer teruggegeven die naar data van een willekeurig type wijst.
- Dit wordt aangegeven door het type `void *`.
- Dit type moet omgezet worden naar het juiste type, door middel van de casting operator.

```
/*
 *      dynvar.c : dynamische geheugen allocatie
 */
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

int main(int argc, char *argv[])
{
    int     *p;
    double  *q;

    p = (int *) malloc(sizeof(int));
    *p = 5;
    printf("De toegekende waarde is %d\n", *p);
    free(p);
```

```
    q = (double *) malloc(sizeof(double));
    *q = 5.25;
    printf("De toegekende waarde is %f\n", *q);
    free(q);
}
```

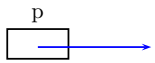


Het gebruik van pointers in programma's leidt vrij dikwijls tot fouten.

Twee bekende fenomenen:

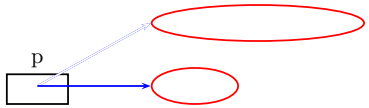
dangling pointer : een variabele die een adres bevat naar een reeds gedeallocceerd object:

```
p = malloc(20);
free(p);
```



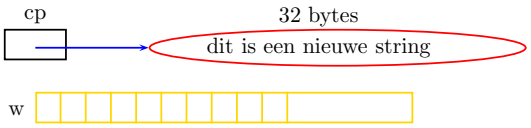
lost object : een gealloceerd dynamische object dat niet langer toegankelijk is vanuit het programma, maar toch nog bruikbare data bevat:

```
p = malloc(40);
p = malloc(4);
```



In plaats van string pointers te initialiseren met adressen van vooraf gedefinieerde karakters en arrays van karakters, kan ook de functie `malloc` gebruikt worden om de geheugenplaatsen voor de string pas tijdens run-time te voorzien.

```
char    *cp;
cp = (char *)malloc(32);
strcpy(cp, "dit is een nieuwe string");
```

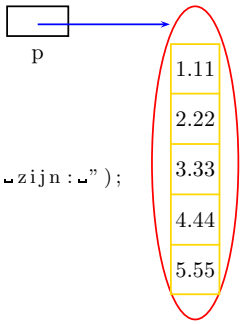


Merk op dat er een verschil is met een gewone array van karakters: **char** w[32];.

Dynamische arrays

```
/*
 *      dynar.c : array met dynamische lengte
 */
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#define VIJF   5

int main(int argc, char *argv[])
{
    double *p;
    int i;

    p = (double *) malloc( VIJF * sizeof(double) );
    memset(p, 0, VIJF*sizeof(double) );
```

```
    *p = 1.11;
    *(p+1) = 2.22;
    *(p+2) = 3.33;
    *(p+3) = 4.44;
    *(p+4) = 5.55;
    printf("De toegekende waarden zijn: ");
    for (i=0; i<VIJF; i++)
        printf("%5.2f ",p[i]);
    printf("\n");
}
```



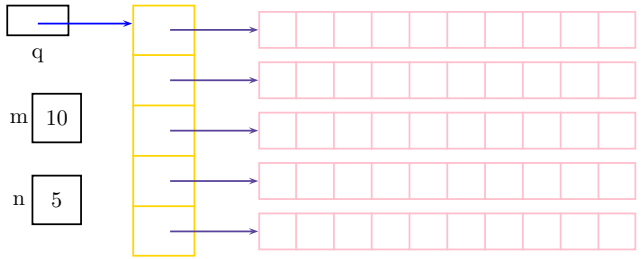
Om een element in de dynamisch gealloceerde array aan te spreken, kan men zowel de pointer notatie (`*(p+i)`) als de array notatie (`p[i]`) gebruiken.

```
    for (i=0; i<n; i++)
        q[i] = (float *) malloc( m*sizeof(float) );
    printf("Geef de elementen in de rij per rij \n");
    for (i=0; i<n; i++)
    {
        for (j=0; j<m; j++)
            scanf("%f", &q[i][j] );
        scanf("%*c");
    }
    /* berekeningen ... */
    for (i=0; i<n; i++)
    {
  printf("%8p : ", q[i] );
        for (j=0; j<m; j++)
            printf("%8.3f ", *(*(q+i)+j) );
        printf("\n");
    }
```

```
/*
 *    dynmat.c : matrix met dynamisch aantal rijen en kolommen
 */
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

int main(int argc, char *argv[])
{
    float   **q;
    int     n, m;
    int     i, j;

    printf("Geef aantal rijen en aantal kolommen : ");
    scanf("%d%d%*c", &n, &m);
    /* reservering geheugen */
    q = (float **) malloc( n*sizeof(float *) );
```

```
    /* vrijgeven geheugen */
    for (i=0; i<n; i++)
        free(q[i]);
    free(q);
}
```

Bij deze dynamische 2-dimensionale array is `q` een pointer naar een pointer; vandaar de twee `*` in de declaratie.



Dynamische structures

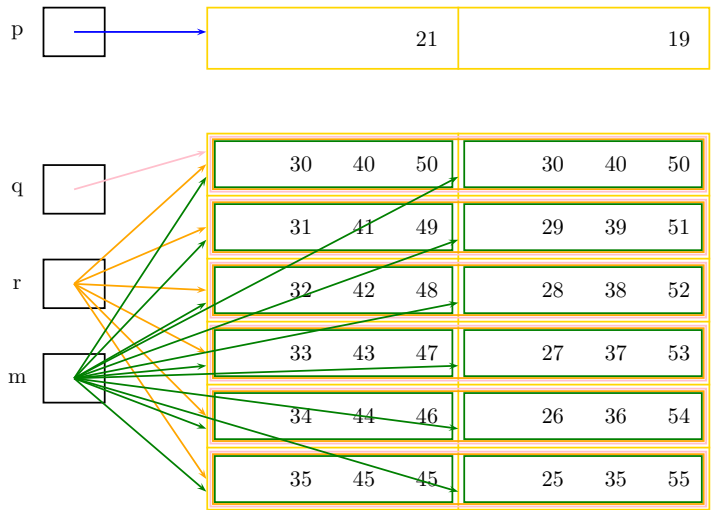
Na definitie (lijn 20) bevat de variabele `p` geen zinnige informatie. Er is alleen plaats voorzien om het resultaat van een `malloc` op te slaan. Wanneer deze toekenning gebeurd is, wijst `p` naar een ruimte van 32 bytes (`sizeof(Koppel)`). Door middel van de casting operator wordt deze ruimte geïnterpreteerd als iets van type `Koppel`. Bij de tweede allocatie wordt ineens ruimte voorzien voor $N = 6$ structures en `q` wijst naar het eerste element hiervan.

Men kan `q` ook interpreteren als een array van N elementen waarbij elk element van type `Koppel` is (`for`-lus vanaf lijn 32). Dit kan natuurlijk ook op een pointer-achtige manier geschreven worden, zie `for`-lus vanaf lijn 41. Met behulp van de `r++` expressie wordt telkens naar het volgende element in de array gewezen.

```
/*
 * dynstru.c : dynamische structuren
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <malloc.h>
#define N 6
typedef struct
        {
             char  naam[14];
             short leeftijd;
        } Mens;
typedef struct
        {
             Mens links;
             Mens rechts;
        } Koppel;
int main(int argc, char *argv[])
{
    Koppel *p, *q, *r;
    Mens   *m;
    int     i;

    p = (Koppel *)malloc( sizeof(Koppel) );
    p->links.leeftijd = 21;
    p->rechts.leeftijd = 19;
    printf("links %d rechts %d\n",
                p->links.leeftijd , p->rechts.leeftijd);

    q = (Koppel *)malloc( N * sizeof(Koppel) );
    memset(q, 0, N*sizeof(Koppel) );
    for (i=0; i<N; i++)
    {
         q[i].links.leeftijd = 30+i;
         q[i].rechts.leeftijd = 30-i;
    }
    for (i=0, r=q; i<N; i++, r++)
         printf("links %d rechts %d\n",
                r->links.leeftijd , r->rechts.leeftijd);

    for (i=0, r=q; i<N; i++, r++)
    {
         (*r).links.leeftijd = 40+i;
         (*r).rechts.leeftijd = 40-i;
    }
    for (i=0, r=q; i<N; i++, r++)
         printf("links %d rechts %d\n",
                r->links.leeftijd , r->rechts.leeftijd);

    m = (Mens *)q;
```

```
    for (i=0; i<2*N; i++, m++)
        m->leeftijd = 50 + (i%2 ? i/2 : -i/2);
    for (i=0, r=q; i<N; i++, r++)
        printf("links %d rechts %d\n",
            r->links.leeftijd , r->rechts.leeftijd);
}
```

In het laatste gedeelte (vanaf lijn 50) wordt de array van N `Koppel` structures geïnterpreteerd als een array van $2N$ `Mens` structures. Hiervoor wordt de `q` pointer van type `Koppel *` via de casting operator omgezet naar de `m` pointer van type `Mens *`.



Een voorbeeld.

In het programma-deel wordt een *vector* object beschreven: wat de elementen van het object zijn en welke operaties op dit object mogelijk zijn.

```
/*
 * vector.h : definities
 */
typedef struct vector
        {
            int lengte;
            double *parr;
        } Vector;
Vector creatie(int lengte, double waarde);
void   teniet(Vector *pv);
void   drukaf(Vector v);
Vector plus(Vector v1, Vector v2);
double inwpro(Vector v1, Vector v2);
```

```
/*
 * vector.c : implementatie
 */
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include "vector.h"

Vector creatie(int lengte, double waarde )
{
    Vector vec = { 0, NULL };
    int    i;

    if ( lengte > 0 )
    {
        vec.lengte = lengte;
        vec.parr = (double *)malloc(lengte*sizeof(double) );
```

```
        for (i=0; i<lengte; i++)
            vec.parr[i] = waarde;
    }
    return vec;
}
void teniet(Vector *pv)
{
    if ( pv->lengte == 0 && pv->parr == NULL )
        return;
    else
    {
        free(pv->parr);
        pv->lengte = 0;
        pv->parr = NULL;
    }
    return;
}
```

```
void drukaf(Vector v)
{
    int i;

    for (i=0; i<v.lengte; i++)
        printf("%10.3f", v.parr[i] );
    printf("\n");
}
Vector plus(Vector v1, Vector v2)
{
    Vector vec;
    int    i;

    if ( v1.lengte != v2.lengte )
        return creatie(0, 0.0);
    vec = creatie(v1.lengte, 0.0);
    for (i=0; i<vec.lengte; i++)
```

```
        vec.parr[i] = v1.parr[i] + v2.parr[i];
    return vec;
}
double inwpro(Vector v1, Vector v2)
{
    double  res = 0.0;
    int     i;

    if ( v1.lengte != v2.lengte )
        return res;
    for (i=0; i<v1.lengte; i++)
        res += v1.parr[i] * v2.parr[i];
    return res;
}
```

Om deze routines te testen kan een testprogramma gemaakt worden:

```
/*
 * testvec.c : testprogramma
 */
#include <stdio.h>
#include <stdlib.h>
#include "vector.h"

int main(int argc, char *argv[])
{
    Vector a, b, c;

    a = creatie(5, 2.5);          b = creatie(5, 6.9);
    c = plus(a,b);                drukaf(c);
    printf("Inwendig product = %f\n", inwpro(a,c) );
    teniet(&a);       teniet(&b);       teniet(&c);
}
```

Denktaak

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

#define LEN 16

int main(int argc, char *argv[])
{
   char *cp;
   char car[LEN];

   if ( argc != 3 )
      exit(1);
   cp = (char *)malloc( LEN*sizeof(char) );
   strcpy(cp, argv[1]);
   strcpy(car, argv[2]);
   cp++;          printf("via pointer %s\n", cp);
   car++;         printf("via array %s\n", car);
   fpoin(cp);     printf("\t\t pointer %s\n", cp);
   farray(car);   printf("\t\t array %s\n", car);
}

fpoin(char *tp)
{
   tp++;        printf("\tfun pointer %s\n", tp);
}

farray(char tar[])
{
   tar++;       printf("\tfun array %s\n", tar);
}
```

Bespreek aan de hand van bovenstaand programma het verschil tussen een array van characters en een pointer naar een character. Leg uit wat er gebeurt bij starten van `a.out joske flup`.

Merk op dat lijn 18 een syntax fout bevat; wat is er verkeerd?