Standard C biblioteket

Ur innehållet:

- Dynamisk minneshantering
- Filoperationer

Målsättningar:

- Att kunna använda dynamisk minneshantering
- Att kunna implementera måldatorberoende funktioner för dynamisk minneshantering med MD407
- Att kunna använda enklare filoperationer

Standard C biblioteket

"The C standard library" eller **`libc`** utgör standardbiblioteket för programspråket C och specificerades ursprungligen i ANSI C standarden från 1989, (C89).
Flera tillägg har gjorts efter detta.

`<assert.h>`	Definierar macro som kan användas för att upptäcka logiska fel och andra felaktigheter I "debug"-versioner av program.
`<ctype.h>`	Deklarerar funktioner för att klassificera och omvandla tecken.
`<errno.h>`	För att kontrollera felkoder från biblioteksfunktionerna.
`<float.h>`	Definierar olika konstanter som definierar implementationsspecifika detaljer i flyttalshanteringen.
`<limits.h>`	Definierar olika konstanter som definierar implementationsspecifika detaljer i heltalshanteringen.
`<locale.h>`	Deklarerar funktioner för lokala anpassningar.
`<math.h>`	Deklarerar vanliga matematiska funktioner..
`<setjmp.h>`	Deklarationer för speciella programflödesändringar.
`<signal.h>`	Deklarerar funktioner för hantering av "signaler".
`<stdarg.h>`	För variabelt antal argument till en funktion.
`<stddef.h>`	Definierar diverse olika typer och macron.
`<stdio.h>`	Deklarerar in- och utmatningsfunktioner.
`<stdlib.h>`	Deklarerar funktioner för numeriska konverteringar, slumptalsgenerering och minneshantering.
`<string.h>`	Deklarerar funktioner för stränghantering.
`<time.h>`	Deklarerar funktioner för datum och tid.

Dynamisk minneshantering

Dynamisk minneshantering tillåter programmet att reservera och återlämna minne under programmets exekvering.
Detta kan ge möjlighet till en effektiv användning av systemets minnesresurser.
Två grundläggande funktioner tillhandahåller tjänsterna.

- `malloc()` Reservera (allokera) minnesutrymme
- `free()` Återlämna (deallokera) minnesutrumme

Funktionsprototyper finns i:

```
#include <stdlib.h>
extern void *malloc (size_t);
extern void free (void *);
```

`size_t` är av samma typ som `sizeof`-operatorn. Det är en måldatorberoende heltalstyp (utan tecken) exempelvis:

```
typedef size_t unsigned int;
```

Dynamisk allokering av minne

Exempel:

```
#include <stdlib.h>
#define TEXT_BUFFER_SIZE   1000

  ...
  char * p;
  p = (char *) malloc(TEXT_BUFFER_SIZE);
  do_textbuffer_job( p );
  free(p);
  ...
```

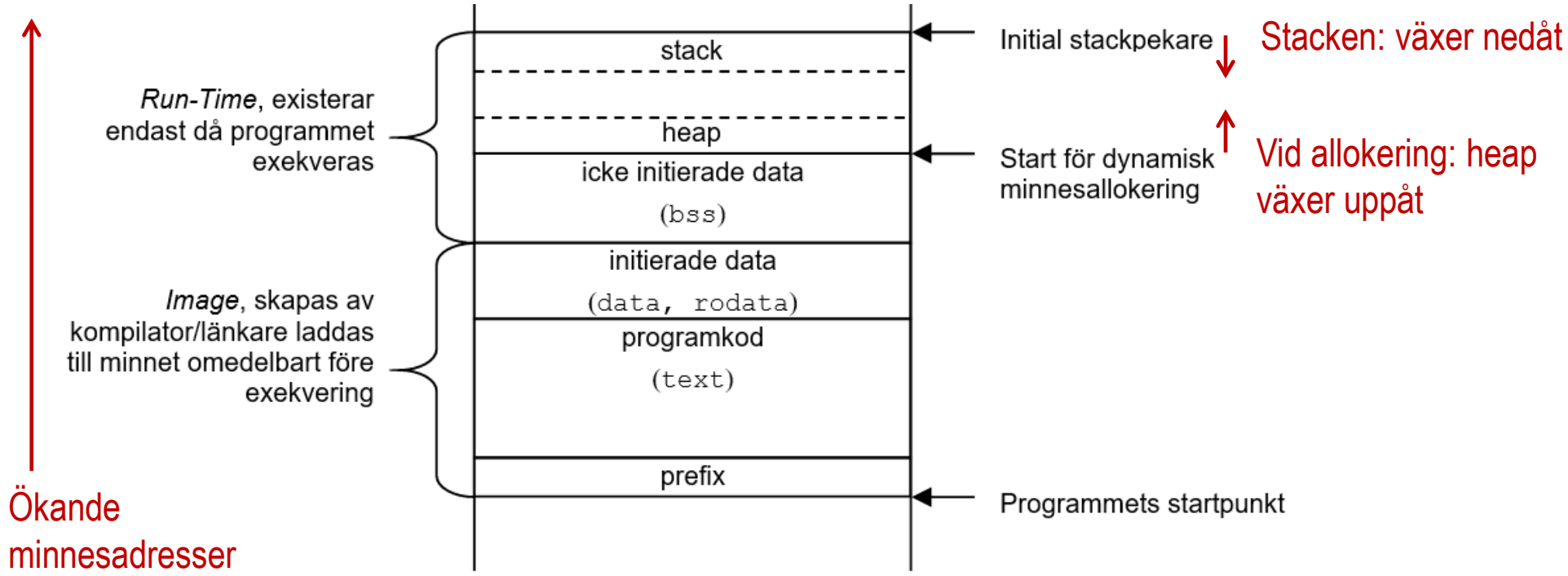
Lämplig pekartyp för minnesblocket

Antal bytes som ska reserveras

Minnet frigörs för nästa allokering

Programmets adressrymd

Exekveringsmiljön för ett program omfattar ett totalt minnesutnyttjande som kan beskrivas med denna figur.



Felkällor - dynamisk minneshantering

```
#include <stdlib.h>
#define TEXT_BUFFER_SIZE  1000

  ...
  char * p;
  p = (char *) malloc(TEXT_BUFFER_SIZE);
  do_textbuffer_job( p );
  free(p);
  ...


void do_textbuffer_job( char *p )
{
  p[0] = '\n'; /* p == 0 ? */
  ...
  c = p[index];  /* index > TEXT_BUFFER_SIZE*/
  ...
  p[index]=c;
  ...
}
```

OUT OF MEMORY

Om minnesutrymmet som avdelats för "heap"-användning är otillräckligt, returnerar malloc 0, därför måste man alltid kontrollera returvärdet

```
if( !p ) /* inget minne, vad göra nu? */
```

OUT OF MEMORY BOUNDS

Vad händer om vi refererar minne utanför det reserverade blocket?

```
p[1001]   /* odefinierat, nonsens */
p[1001] = ...; /* värre, kanske förstör vi något */
```

Typkonverteringar

Returvärdet från `void *malloc (size_t)` ska alltid konverteras till en lämplig (avsedd) pekartyp.

Exempel, vi har:

```
#include <stdlib.h>

typedef struct {
  char *name;
  int   number;
} xyz;

int main() {
  int    *x;
  char   *y;
  double *z;
  xyz    *u;
  // allokera 100 int, char,
  // double respektive xyz...
}
```

```
x = (int *) malloc( sizeof(int) * 100 );
if( !x ) exit(-1);
```

```
y = (char *) malloc( sizeof(char) * 100 );
if( !y ) exit(-1);
```

```
z = (double *) malloc( sizeof(double) * 100 );
if( !z ) exit(-1);
```

```
u = (xyz *) malloc( sizeof(xyz) * 100 );
if( !u ) exit(-1);
```

Andra allokeringsfunktioner

```
void *calloc(size_t antal_element, size_t elementstorlek);
```

`calloc` reserverar *`antal_element*elementstorlek`* och initierar det reserverade blockets samtliga element till 0.

Exempel:

```
p = (int*) calloc( 100, sizeof(int) );
```

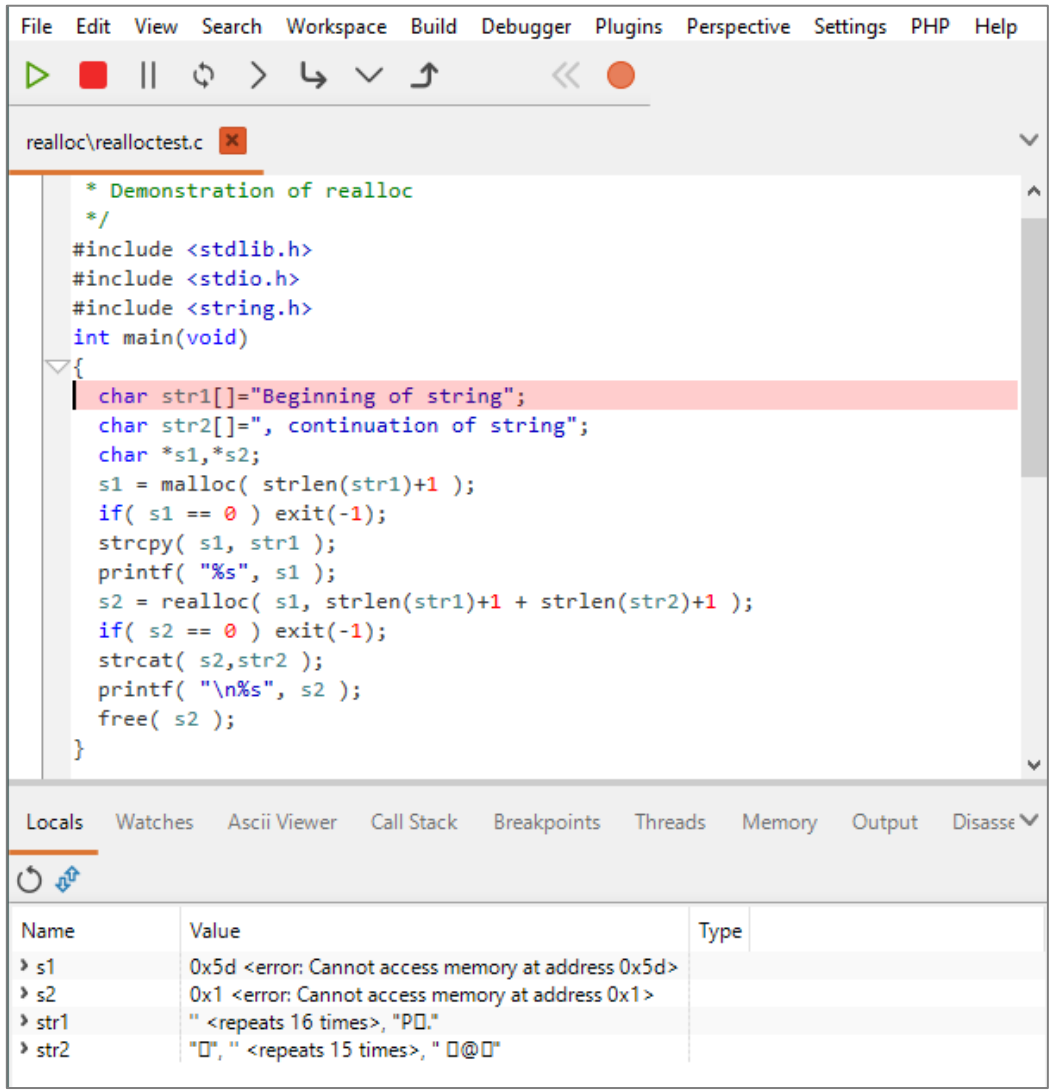
reserverar ett minnesblock som rymmer 100 int och initierar dessa till 0

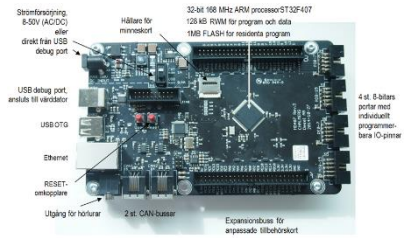
```
void *realloc(void *tidigare_allokering, size_t ny_storlek);
```

`realloc` modifierar en tidigare gjord allokering (`malloc`, `calloc` eller `realloc`) till en ny storlek. Det tidigare blockets innehåll kopieras till det nya blocket, och kan antingen utvidgas med extra utrymme eller förminskas och därmed trunkera det tidigare blocket.

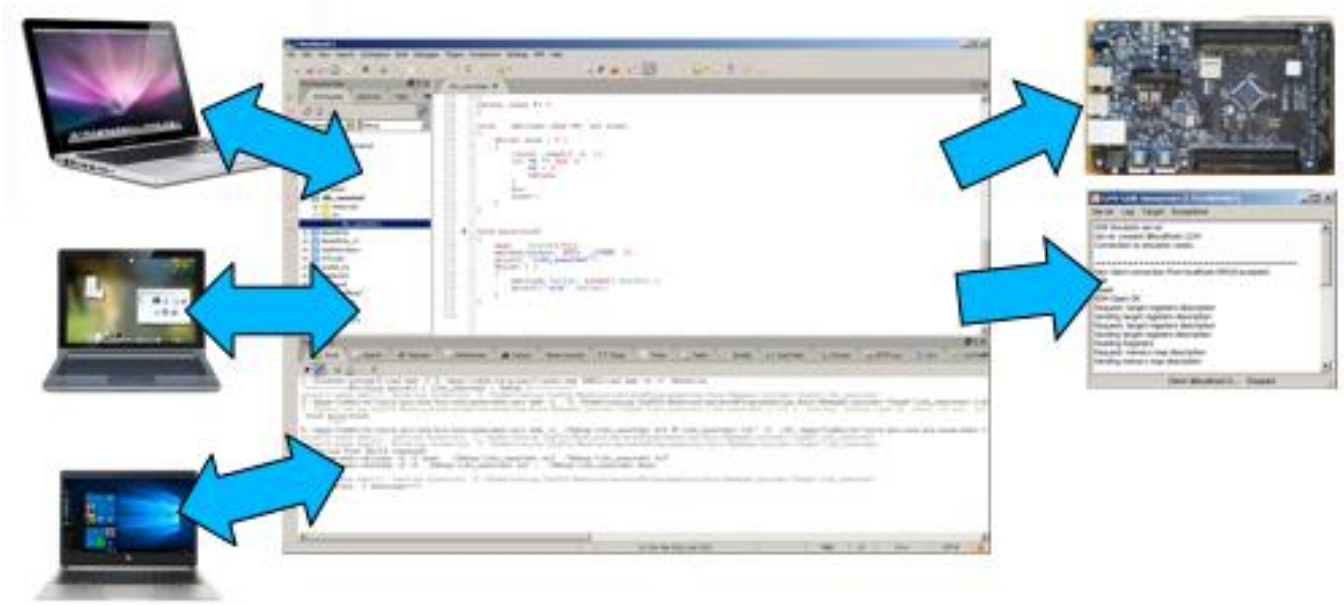
Demonstration: `realloc()`

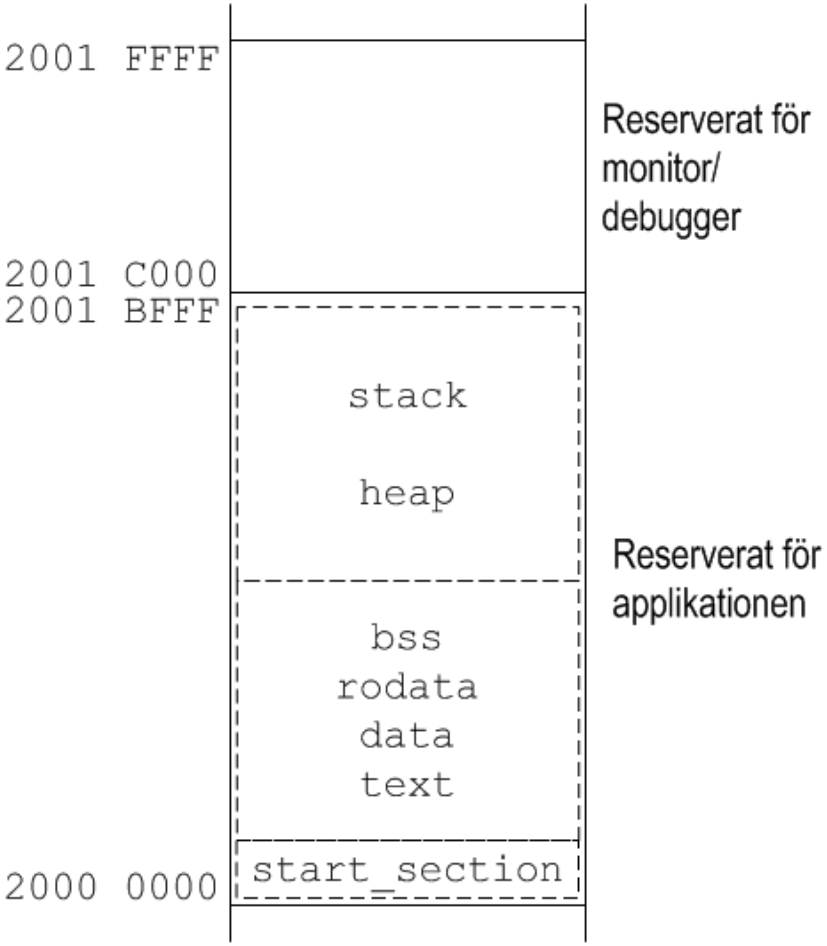
```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(void)
{
  char str1[]="Beginning of string";
  char str2[]=", continuation of string";
  char *s1,*s2;
  s1 = malloc( strlen(str1)+1 );
  if( s1 == 0 ) exit(-1);
  strcpy( s1, str1 );
  printf( "%s", s1 );
  s2 = realloc( s1,strlen(str1)+1 + strlen(str2)+1 );
  if( s2 == 0 ) exit(-1);
  strcat( s2,str2 );
  printf( "\n%s", s2 );
  free( s2 );
}
```





Måldator – minnesdisposition





Dynamisk minneshantering, malloc/free

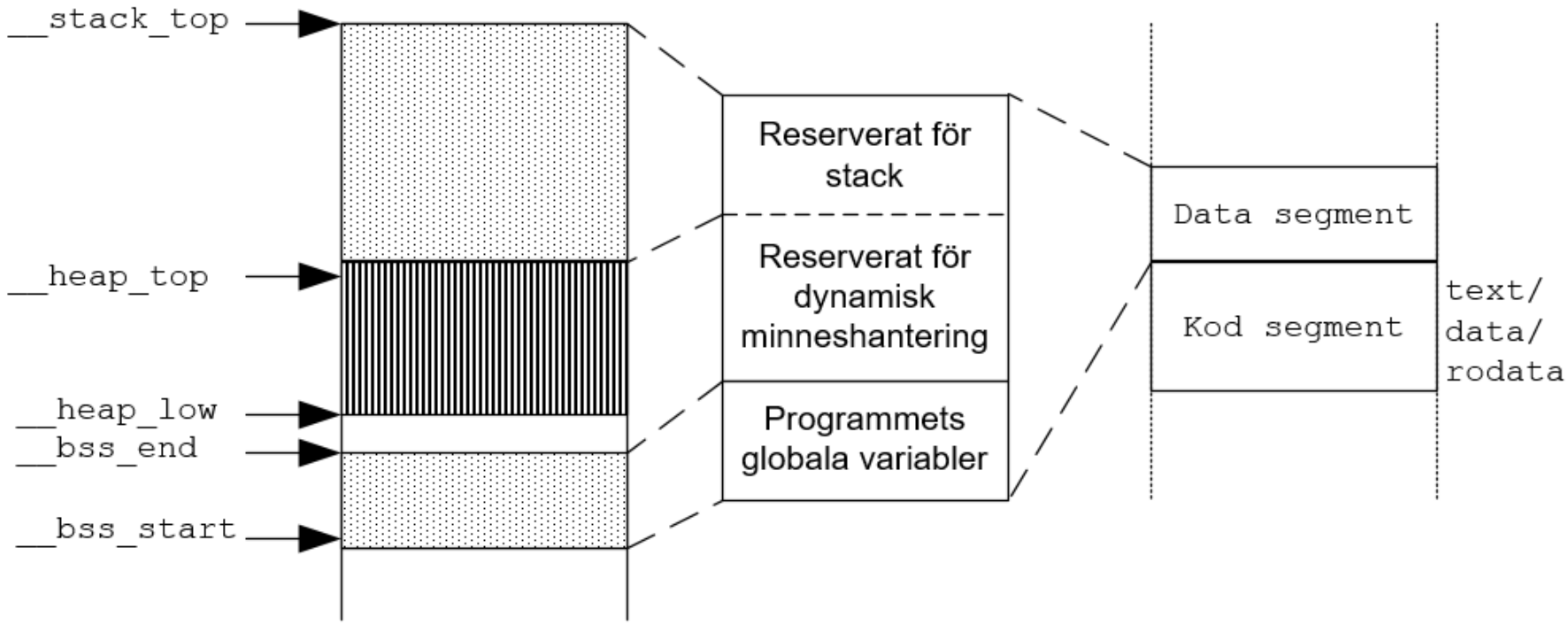
Länkaren skapar de symboler vi behöver för att administrera minnet

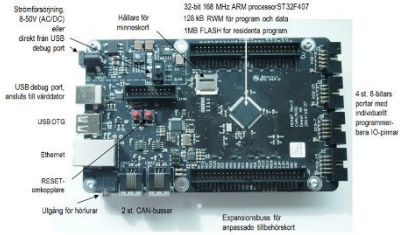
```
...
*(.start_section
*(.text)
*(.text.*)
*(.data)
*(.data.*)
*(.rodata)
*(.rodata.*)
. = ALIGN(4);
__bss_start__ = .;
*(.bss)
*(.bss.*)
__bss_end__ = .;
. = ALIGN(4096);
__heap_low = .;
. = . + 0x400;
__heap_top = .;
. = . + 0x400;
__stack_top = .;
```

C-biblioteket tillhandahåller rutiner som `malloc` och `free` för dynamisk minneshantering men har ingen information om hur måldatorns minne disponeras. Runtime biblioteket måste därför definiera:

```
void *_sbrk (int increment)
```

(*set program break*) som tillhandahåller adresser till minne som är tillgängligt för `malloc`.





Implementering av modifierad startup

Vår startup-sekvens för *md407* behöver modifieras som förberedelse för att använda C-biblioteket.

För applikationsprogram krävs att C-biblioteket initierats av någon funktion som utförts före **`main`**. Standardprocedurer för *pre main* och *after main* finns `crt` (*c-run time)*. Detta motsvaras av den ”startup” vi själva tidigare skapat i våra program.

```
__attribute__((naked))
__attribute__((section (".start_section")) )
void startup ( void )
{
 __asm__ volatile(" LDR  R0,=__stack_top\n");
 __asm__ volatile(" MOV  SP,R0\n");
 __asm__ volatile(" BL   crt_init\n");
 __asm__ volatile(" BL   main\n");
 __asm__ volatile(" BL   crt_deinit\n");
 __asm__ volatile(" .globl _exit\n");
 __asm__ volatile("_exit: B _exit\n");
}
```

```
    ...
    *(.start_section
    *(.text)
    *(.text.*)
    *(.data)
    *(.data.*)
    *(.rodata)
    *(.rodata.*)
    . = ALIGN(4);
    __bss_start__ = .;
    *(.bss)
    *(.bss.*)
    __bss_end__ = .;
    . = ALIGN(4096);
    __heap_low = .;
    . = . + 0x400;
    __heap_top = .;
    . = . + 0x400;
    __stack_top = .;
```

Initiering av runtime funktioner

Standarden säger att icke initierade variabler (dvs. bss-arean) ska initieras till 0 av run-time systemet...

```
void crt_init(void) {
  extern char __bss_start;
  extern char __bss_end;
  extern char __heap_low;
  extern char __heap_top;
  char *s;
  s = &__bss_start;
  while( s < &__bss_end)
    *s++ = 0;
  s = &__heap_low;
  while( s < &__heap_top )
    *s++ = 0;
}
```

```
void crt_deinit(void) {
}
```

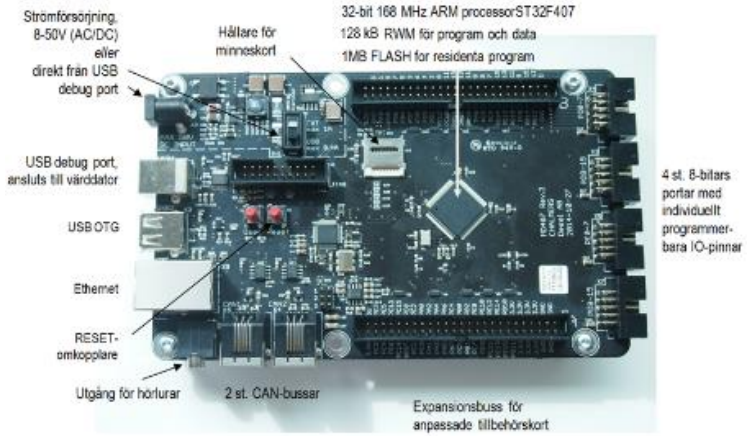
```
...
*(.start_section
*(.text)
*(.text.*)
*(.data)
*(.data.*)
*(.rodata)
*(.rodata.*)
. = ALIGN(4);
__bss_start__ = .;
*(.bss)
*(.bss.*)
__bss_end__ = .;
. = ALIGN(4096);
__heap_low = .;
. = . + 0x400;
__heap_top = .;
. = . + 0x400;
__stack_top = .;
```

Implementering av _sbrk

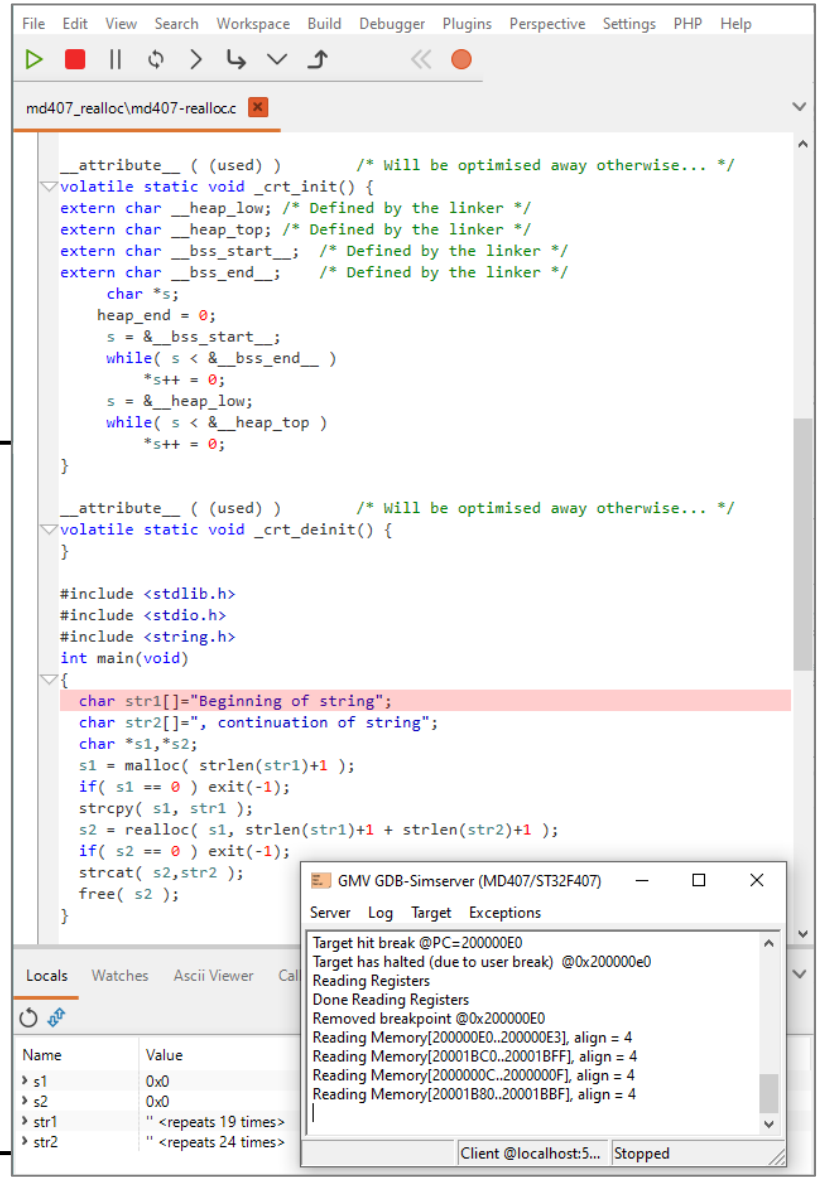
```
#include    <errno.h>
static char *heap_end;
char * _sbrk( int incr) {
  extern char __heap_low;
  extern char __heap_top;
  char *prev_heap_end;
  if (heap_end == 0) {
    heap_end = &__heap_low;
  }
  prev_heap_end = heap_end;

  if (heap_end + incr > &__heap_top) {
    /* Heap and stack collision */
    errno = ENOMEM;
    return (char *)-1;
  }
  heap_end += incr;
  return (char *) prev_heap_end;
}
```

```
    ...
    *(.start_section
    *(.text)
    *(.text.*)
    *(.data)
    *(.data.*)
    *(.rodata)
    *(.rodata.*)
    . = ALIGN(4);
    __bss_start__ = .;
    *(.bss)
    *(.bss.*)
    __bss_end__ = .;
    . = ALIGN(4096);
    __heap_low = .;
    . = . + 0x400;
    __heap_top = .;
    . = . + 0x400;
    __stack_top = .;
```

Demonstration: dynamisk minneshantering i MD407




```c
void main(void)
{
    char str1[]="Beginning of string";
    char str2[]="Continuation of string";
    char *s1,*s2;
    s1 = malloc( strlen(str1)+1 );
    if( s1 == 0 ) exit(-1);
    strcpy( s1, str1 );
    s2 = realloc( s1, strlen(str1)+1 +strlen(str2)+1 );
    if( s2 == 0 ) exit(-1);
    strcat( s2,str2 );
    free( s2 );
}
```



Filoperationer

C-biblioteket tillhandahåller en lång rad funktioner för att hantera filer i filsystemet.

några få exempel:

```
int read(int fd, void *buf, size_t count);
int write(int fd, const void *buf, size_t count);
int open(const char *path, int oflag, ... );
int close( int fd );
```

```
...
fd_in = open("../x1.txt", O_RDONLY , 0777);
fd_out = open("../x2.txt", O_WRONLY | O_CREAT | O_TRUNC, 0777);
result = read(fd_in, buffer, sizeof( buffer) );
if( result != write(fd_out, buffer , result) )
  exit();

close( fd_in );
close( fd_out );
...
```

Demonstration: Filkopiering

```c
int main(void)
{
    char buffer[32];
    int fd_in, fd_out, result;

    fd_in = open("../x1.txt", O_RDONLY , 0777);
    if( fd_in < 0)
        exit(-1);
    fd_out = open("../x2.txt", O_WRONLY | O_CREAT | O_TRUNC, 0777);
    if( fd_out < 0 )
    {
        close( fd_in );
        exit(-1);
    }
    do{
        result = read(fd_in, buffer, sizeof( buffer) );
        if( result != write(fd_out, buffer , result) )
            break;
    }while( result > 0 );
    close( fd_in );
    close( fd_out );
    return 0;
}
```