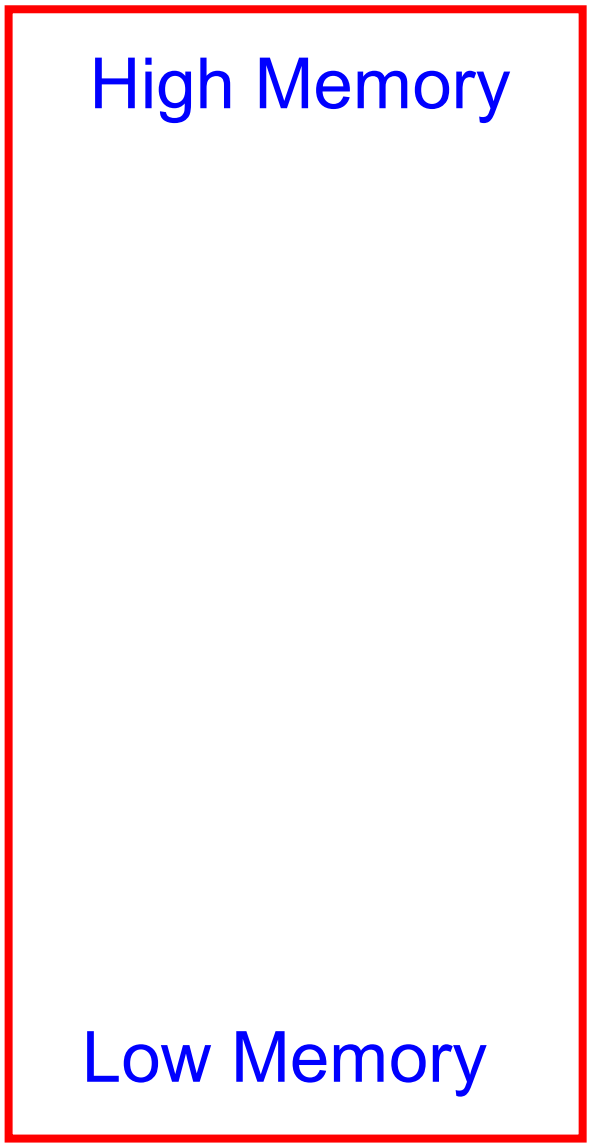
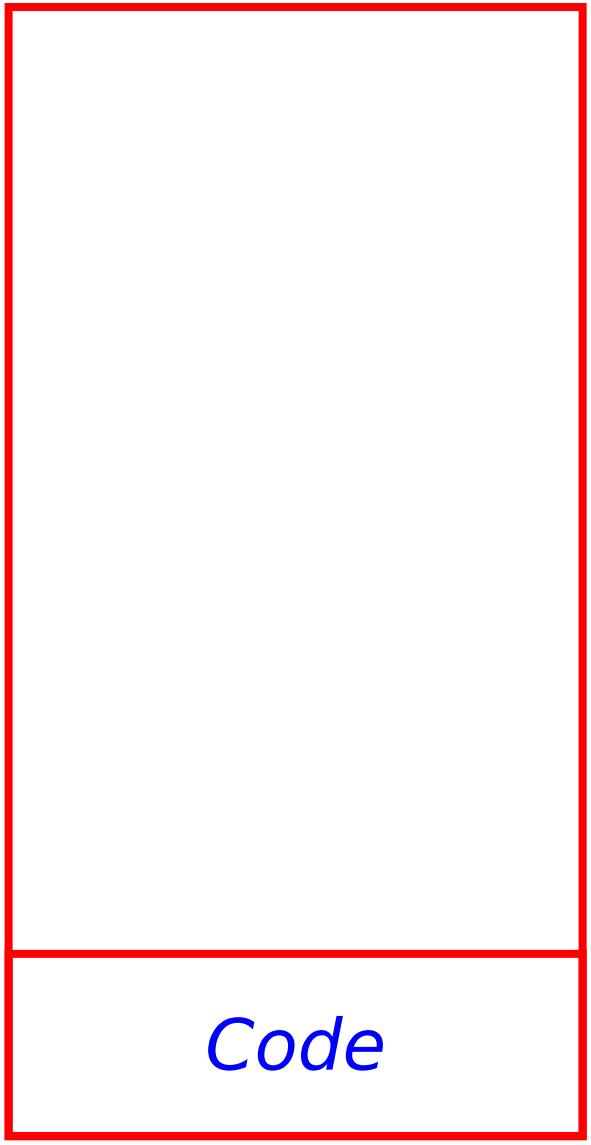
Graficzna wizualizacja pamięci

- Będziemy używać diagramów przedstawiających pamięć komputera lub konkretnego programu w postaci układu prostokątów
- Zachowamy konwencję, w której niskie adresy znajdują się na dole rysunku, a wysokie na górze rysunku
- Diagramy zwykle nie zachowują skali



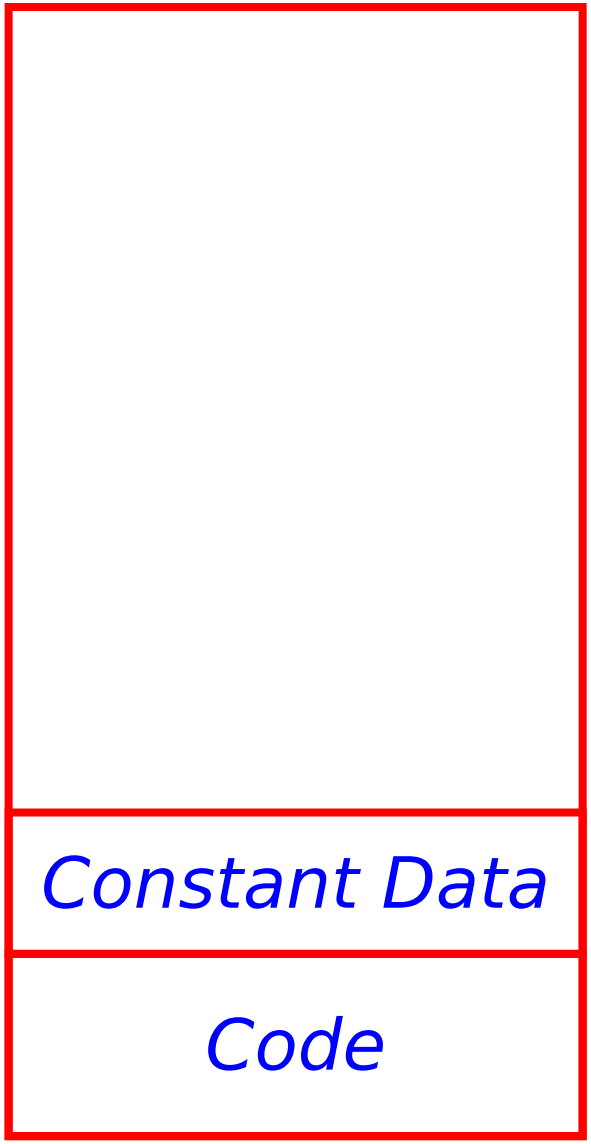
Typowy układ pamięci programu

- Zazwyczaj kod programu (instrukcje kodu maszynowego) jest umieszczany na początku pamięci



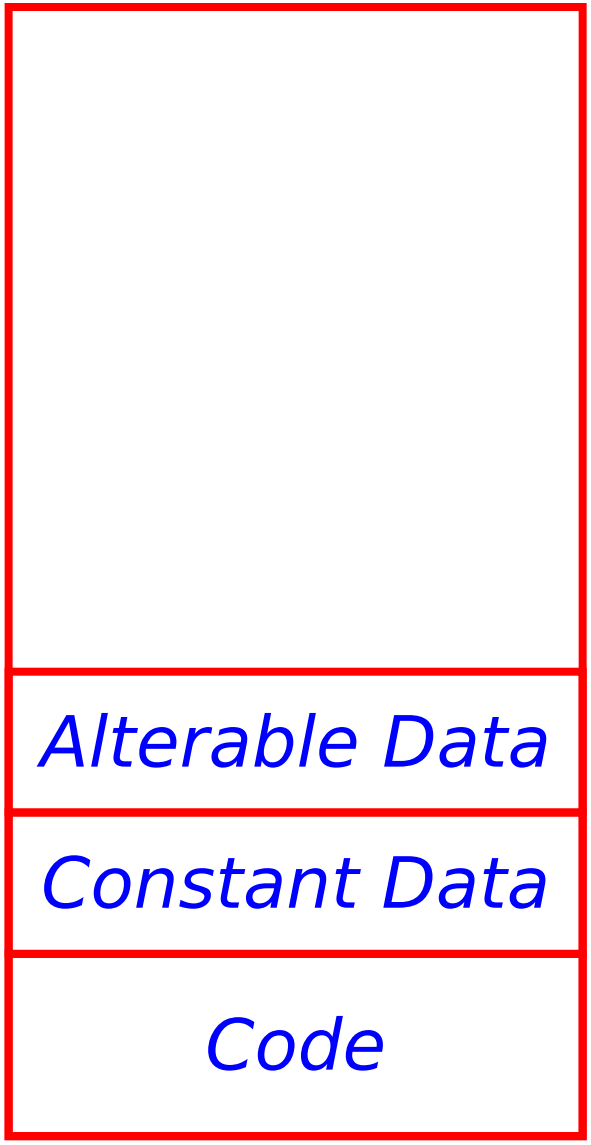
Typowy układ pamięci programu

- Dalej następuje obszar pamięci przechowujący dane stałe



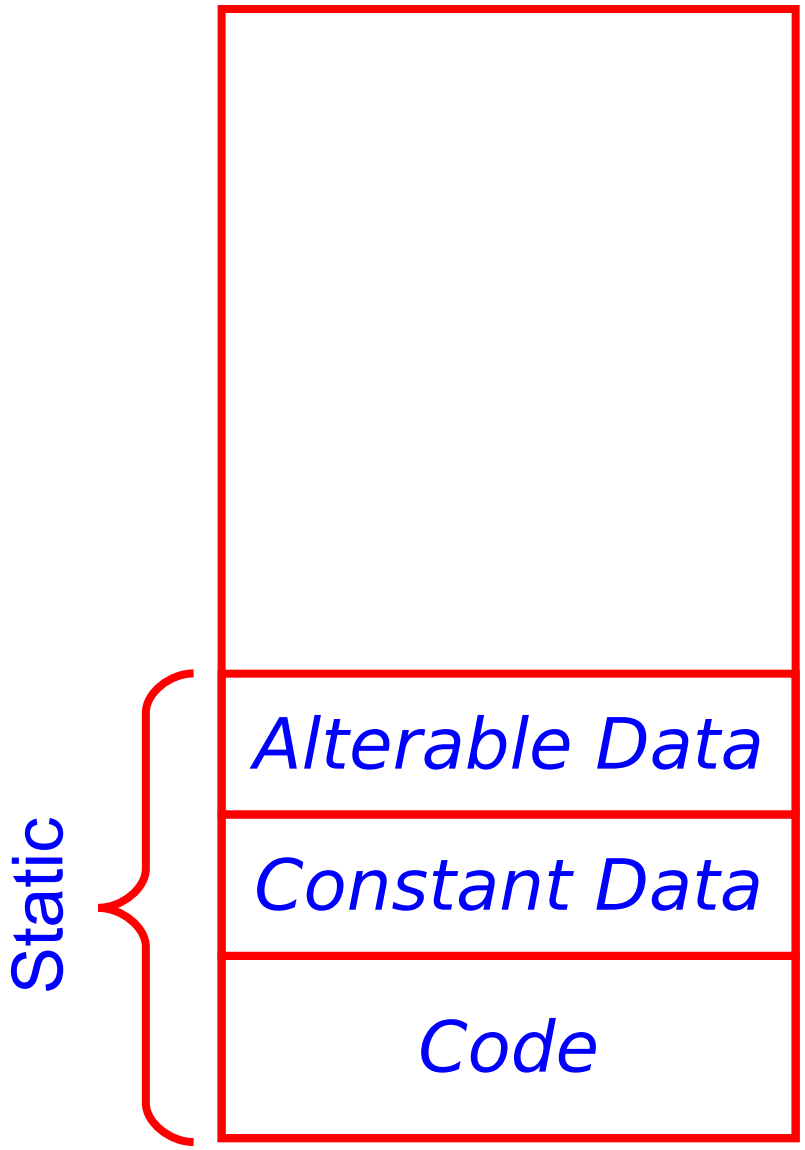
Typowy układ pamięci programu

- Dane modyfikowalne zajmują kolejny obszar



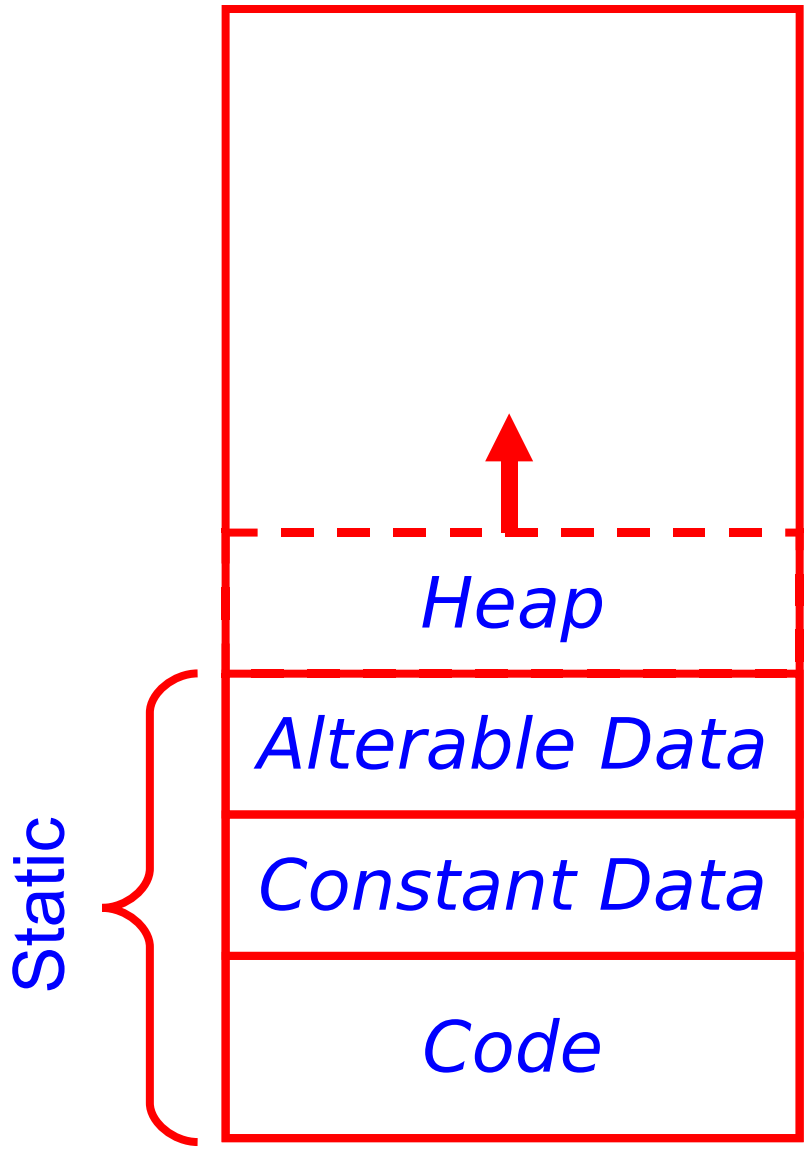
Typowy układ pamięci programu

- Te trzy elementy obejmują statyczny obszar pamięci. Szczegóły dotyczące statycznego obszaru pamięci (rozmiar, adresy funkcji, stałych i zmiennych) są znane na etapie kompilacji.



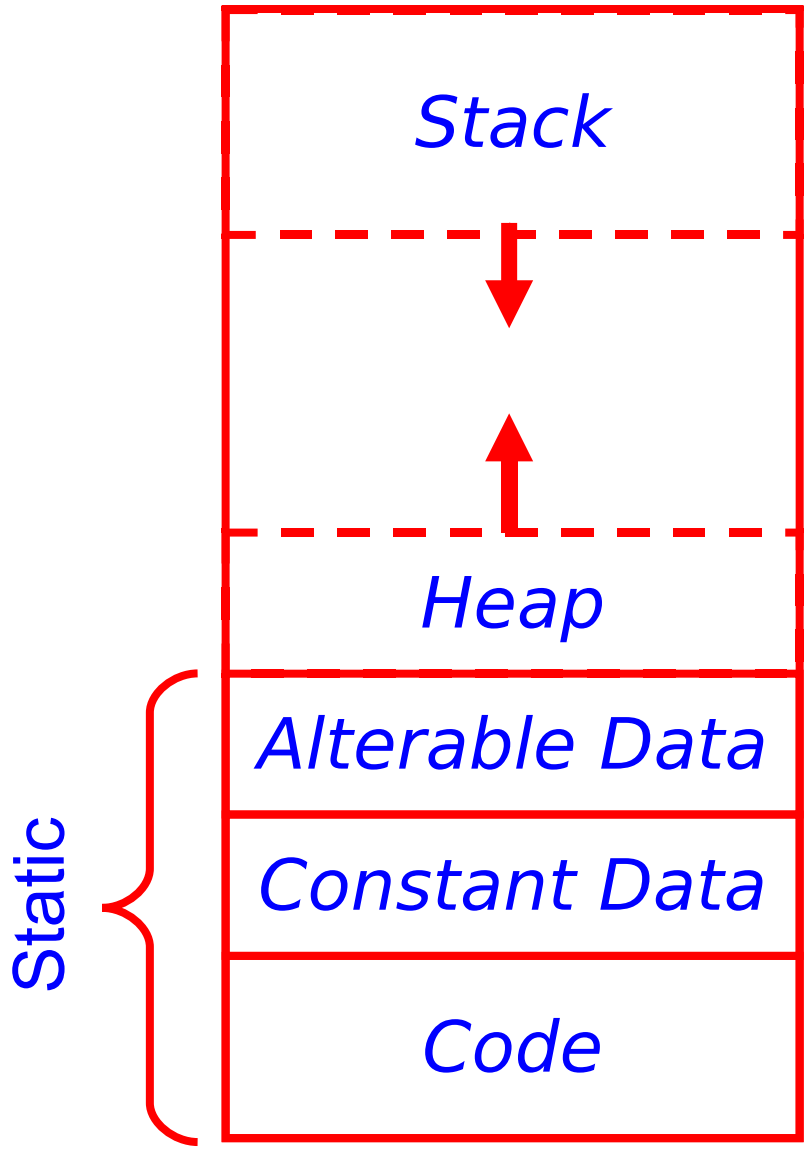
Typowy układ pamięci programu

- Ponad obszarem danych statycznych znajduje się sterta
- Sterta może rozszerzać się w górę w miarę jak program dynamicznie alokuje pamięć



Typowy układ pamięci programu

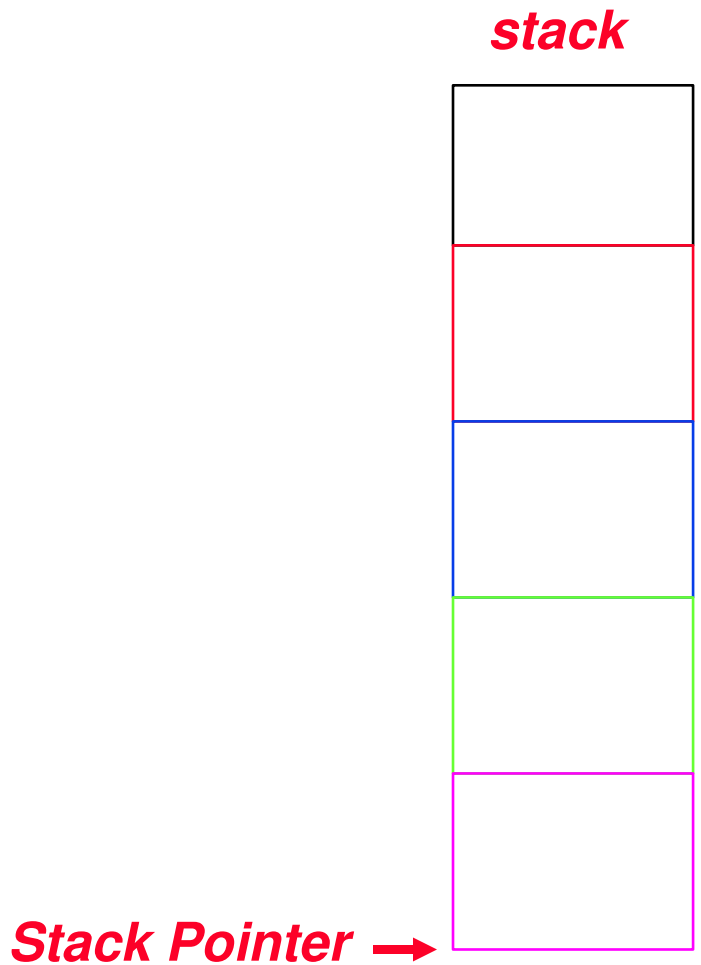
- Na samej górze pamięci znajduje się stos, który może rozszerzać się w dół.
- Stos rozszerza się za każdym wywołaniem funkcji i kurczy się po każdym powrocie z funkcji
- Elementy znajdujące się na stosie obejmują
  - Zmienne lokalne
  - Parametry funkcji
  - Wartości zwracane



Stos

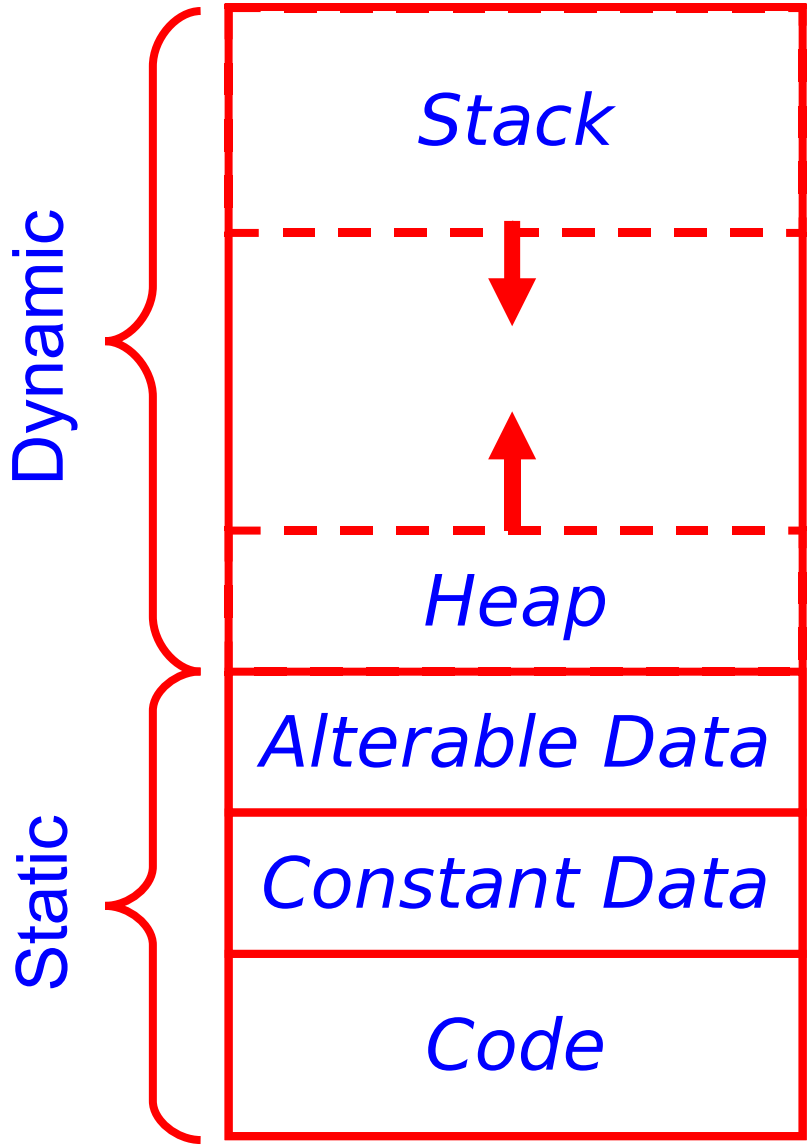
- Last In, First Out (LIFO)

```
main ()
{ a(0);
}
  void a (int m)
  { b(1);
  }
    void b (int n)
    { c(2);
    }
      void c (int o)
      { d(3);
      }
        void d (int p)
        {
        }
```



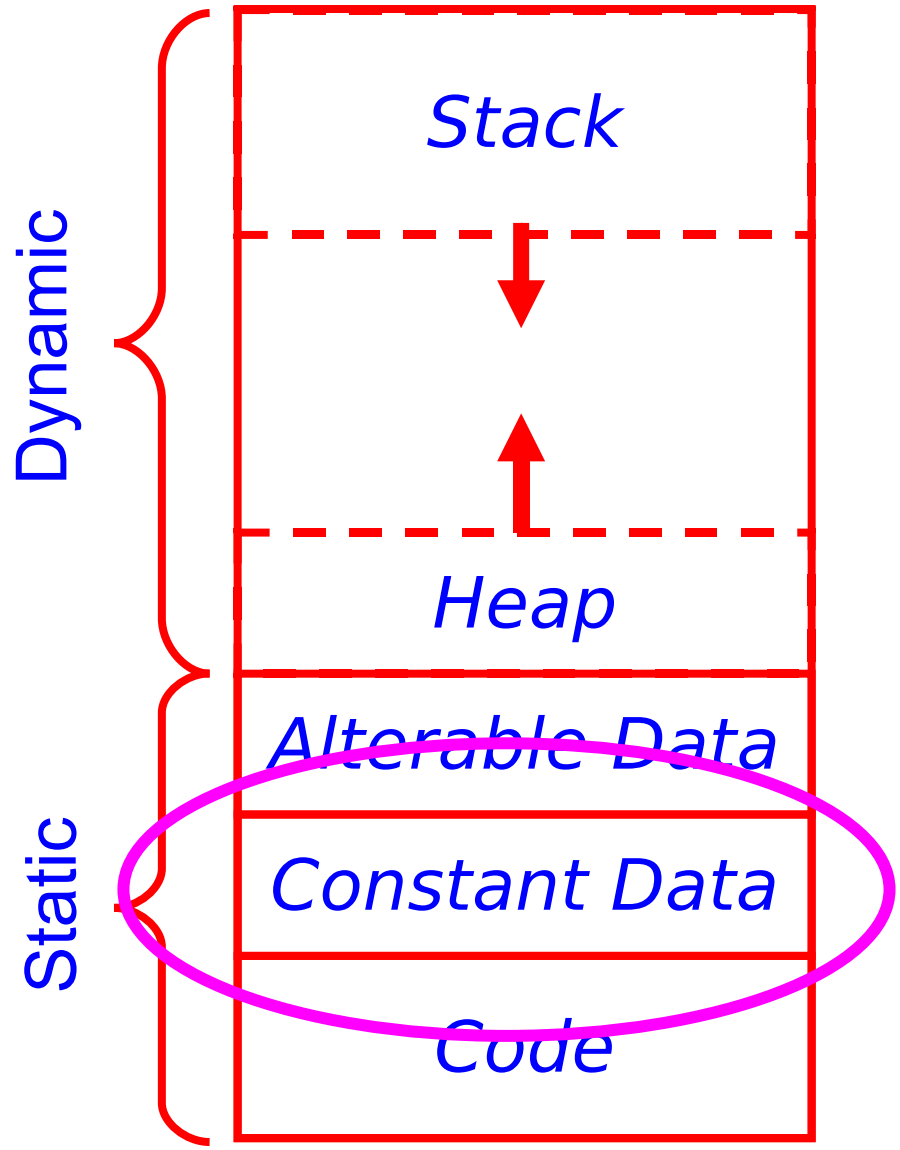
Typowy układ pamięci programu

- Dwa wymienione obszary w górnej części diagramu zmieniają się podczas wykonywania programu
- Dlatego region ten zwany jest obszarem danych dynamicznych



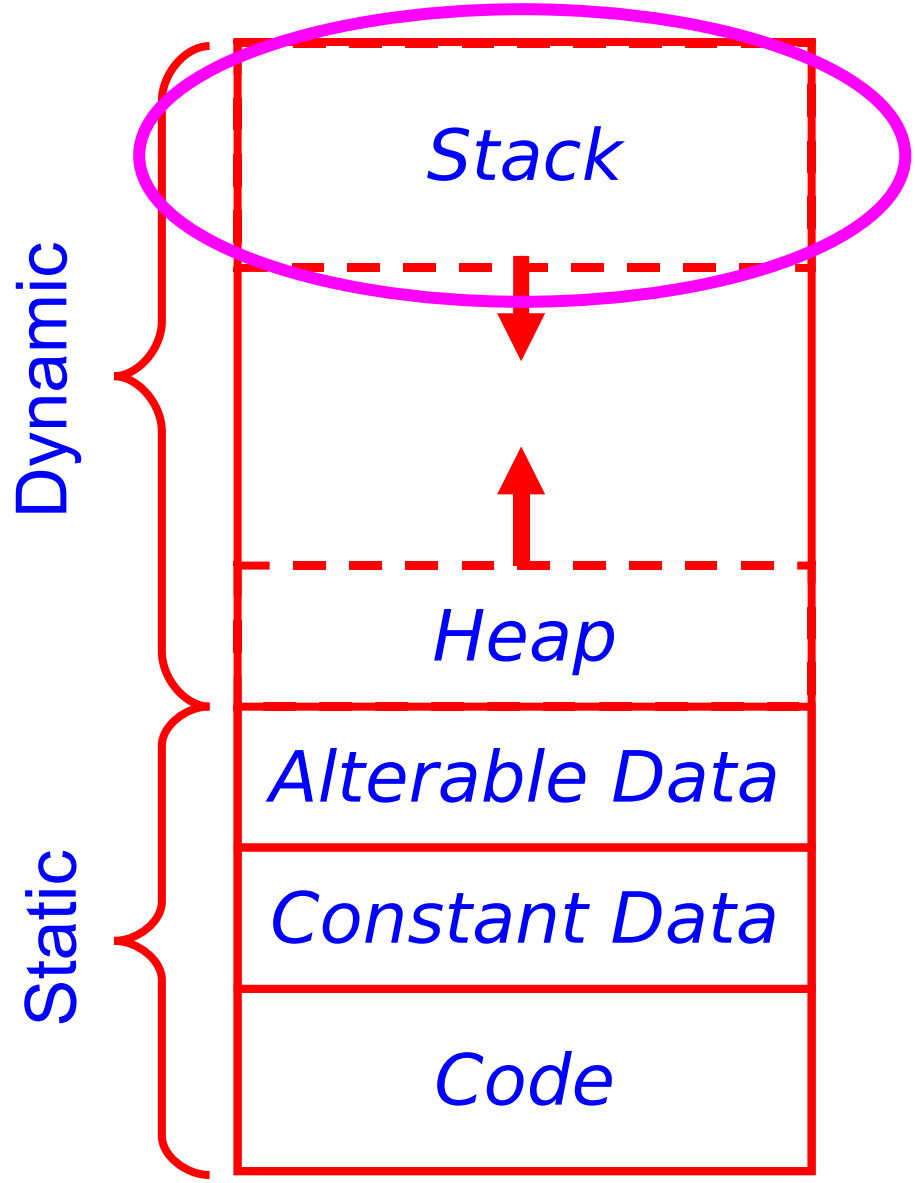
const

- Elementy które mają być umieszczone w obszarze danych stałych są oznaczane przez programistę przy pomocy słowa kluczowego **`const`**
- Umieszczane są w nim również stałe łańcuchowe
- UWAGA!!! W niektórych systemach zmiana danych z kwalifikatorem **`const`** jest możliwa (i relatywnie prosta)



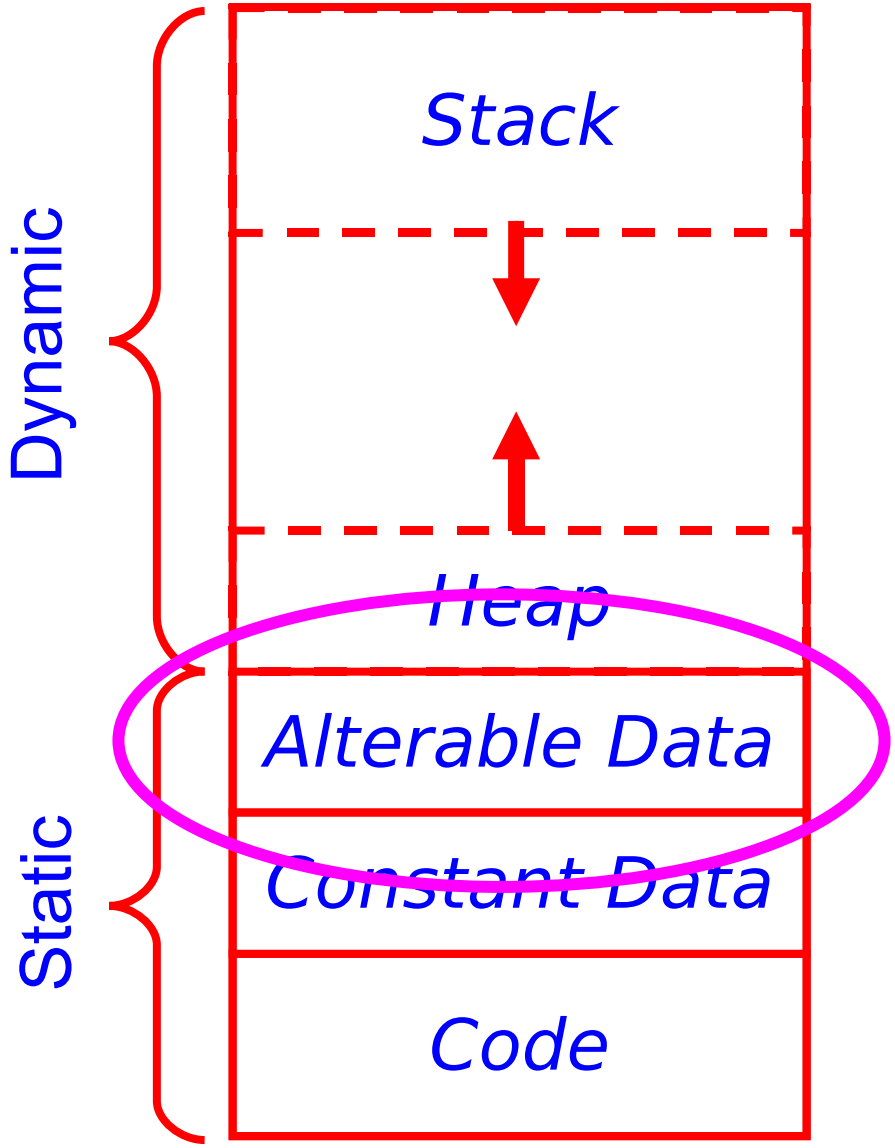
auto

- Zmienne o klasie pamięci auto (automatyczne) są umieszczane na stosie. Słowo kluczowe auto zazwyczaj nie jest używane.
- Miejsce na stosie jest alokowane i zwalniane automatycznie bez udziału porogramisty.



`static` i `extern`

- Zmienne o klasie pamięci static i extern są umieszczane w obszarze danych modyfikowalnych
- **`static`** może mieć różne znaczenia w zależności od kontekstu



Dynamiczna alokacja pamięci

- Obiekty o stałym rozmiarze, znanym w czasie kompilacji, są umieszczane na stosie albo w obszarze danych statycznych
- Czasami rozmiar tablicy jest nieznany w czasie kompilacji
- Programista może alokować pamięć dynamicznie, w czasie wykonania programu, ze sterty.
- Dynamiczna alokacja może być również używana do rezerwacji pamięci na jeden obiekt (int, struktura itd.)

Funkcje do dynamicznej alokacji pamięci

- Funkcje do dynamicznej alokacji pamięci:
  - **`malloc`** – alokuje niezainicjalizowany obszar pamięci
  - **`calloc`** – alokuje obszar pamięci zainicjalizowany zerami
  - **`realloc`** – realokuje obszar pamięci
  - **`free`** – dealokuje (zwalnia) obszar pamięci
- Zadeklarowane w **`<stdlib.h>`**
- Każdemu wywołaniu **`malloc`** , **`calloc`** , **`realloc`** powinno odpowiadać wywołanie **`free`**
- W przeciwnym przypadku mamy tzw. wyciek pamięci

malloc

```
int *ip; /*  define a pointer  */
ip = malloc(10 * sizeof(int));
/*  memory for 10 elements of type int allocated */
if(ip == NULL)
{
  /* Handle Error! */
}
```

- Opcje obsługi błędu
  - Zakończenie programu
  - Ponowne żądanie
  - Zapisanie danych użytkownika
  - Prośba o mniejszy obszar
  - Zwolnienie jakiegoś obszaru

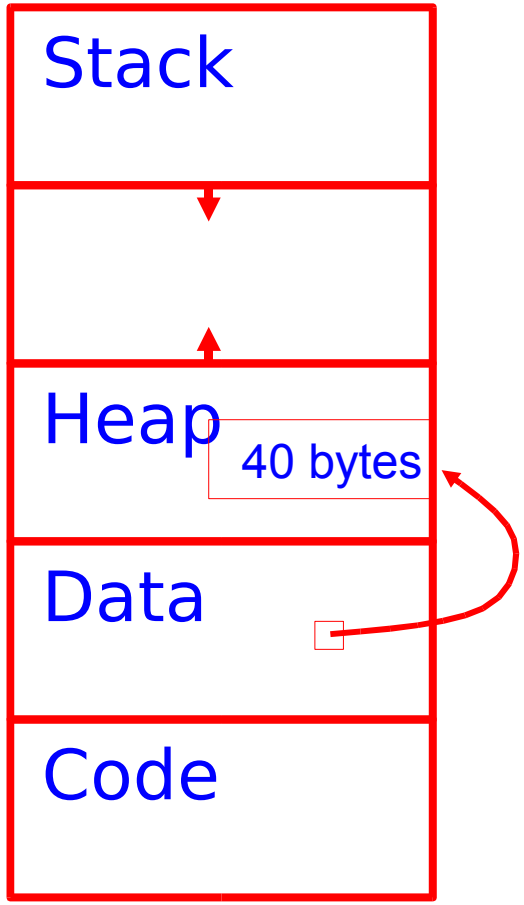
malloc

```
int *ip;
ip = malloc(10 * sizeof(int));
if(ip = NULL)
{
    /* Handle Error! */
}
```

- Użycie operatora przypisania zamiast porównania byłoby katastrofalne!
- Wskaźnik będzie ustawiony na NULL
- Kod obsługi błędów nie będzie wykonany
- Niektórzy programiści aby uniknąć tego błędu piszą: `NULL == ip` lub `!ip`

malloc – co się dzieje?

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



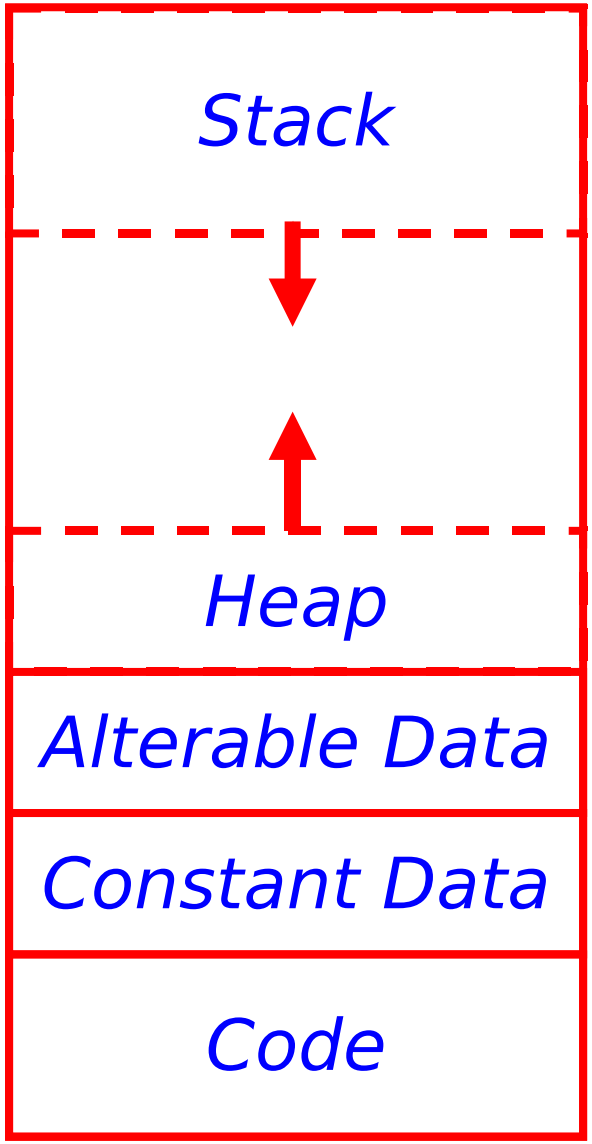
Przykład układu pamięci programu

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

char ga[]="etext";
static char gb[]="stext";
const static char gc[]="sctext";
int i;

int
main ()
{
    static char a[]="sltext";
    static const char b[]="slctext";
    const char* p1="text";
    const char* const p2="text";
    char* p3=malloc(10);
    int j;
    printf("main=    %p\n",(void*)main);
    printf("sctext=  %p\n",(void*)gc);
    printf("slctext= %p\n",(void*)b);
    printf("p1=      %p\n",(void*)p1);
    printf("p2=      %p\n",(void*)p2);
    printf("etext=   %p\n",(void*)ga);
    printf("stext=   %p\n",(void*)gb);
    printf("sltext=  %p\n",(void*)a);
    printf("&i=      %p\n",(void*)&i);
    printf("p3=      %p\n",(void*)p3);
    printf("&p1=     %p\n",(void*)&p1);
    printf("&p2=     %p\n",(void*)&p2);
    printf("&p3=     %p\n",(void*)&p3);
    printf("&j=      %p\n",(void*)&j);
    return 0;
}
```

```
main=     0x8048394
sctext=   0x8048534
slctext=  0x804853b
p1=       0x8048543
p2=       0x8048543
etext=    0x804960c
stext=    0x8049612
sltext=   0x8049618
&i=       0x8049728
p3=       0x8049738
&p1=      0xbffff498
&p2=      0xbffff494
&p3=      0xbffff490
&j=       0xbffff48c
```



Użycie zaalokowanego obszaru

```
int i;
int *ip;
if((ip = malloc(10*sizeof(int))) == NULL)
{
    /* Handle Error Here */
}
for(i = 0; i < 10; i++)
    ip[i] = i;
```

Elastyczność

```
#define MAX 10
int *ip;
ip = malloc(MAX * sizeof(int));
```

- Co będzie jeżeli zmienimy deklarację `int *ip` na short???

```
#define MAX 10
int *ip;
ip = malloc(MAX * sizeof(*ip));
```

Prototypy

- `void *malloc(size_t n);`
- `void free(void *p);`
- `void *realloc(void *p, size_t n);`
- Co to jest ten tajemniczy wskaźnik do **`void`**?

Wskaźnik do `void`

- Oryginalnie nie występował w C
- Względnie nowy dodatek
- Wskaźnik do czegokolwiek
- Przeznaczony do użytku w zastosowaniach takich jak **`free`**, gdzie blok pamięci umieszczony pod jakimś adresem będzie zwolniony bez konieczności znajomości jego dokładnego typu

Potężny i niebezpieczny

```
void *vp;
char *cp;
int *ip;
ip = cp; /* illegal */
```

- Zamiast tego

```
ip = (int *)cp;
```

- lub

```
vp = cp;  /* Legal, powerful and */
ip = vp;  /* dangerous!!! */
```

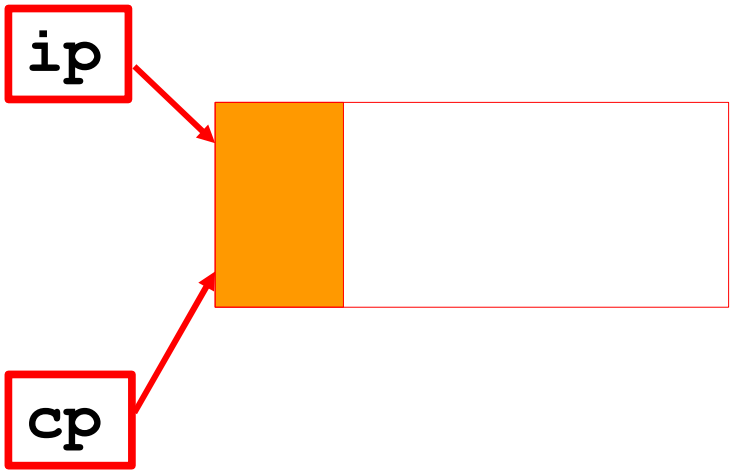
- Po co takie przypisanie?

Rzutowanie

- Zwykle rzutowanie między typami wskaźnikowymi nie jest wymagane
- Może maskować problem

```
int *ip;
char *cp;
...
*cp = 'x';
*ip = ???

*ip = 42;
*cp = ???
```



Ostrzeżenia

- Używanie wskaźnika do **`void`** jako protezy w celu uniknięcia rzutowania nie jest dobrym pomysłem!
- **`malloc`** nie dba o to, co programista robi z zaalokowanym blokiem pamięci. Odpowiedzialny za to jest programista.
- Przekazywanie przypadkowych wartości do **`free`** nie jest dobrym pomysłem!
- **`free`** może zmienić zawartość zwalnianego bloku pamięci
- **`free`** nie zmienia wskaźnika
- Po wywołaniu **`free`** zazwyczaj można robić ze zwolnionym blokiem pamięci wszystko to, co przed jego zwolnieniem!!!
- Zdecydowanie nie jest to dobry pomysł!!!

Inicjalizacja pamięci

- Używamy **`malloc`** kiedy nie chcemy inicjalizować pamięci:

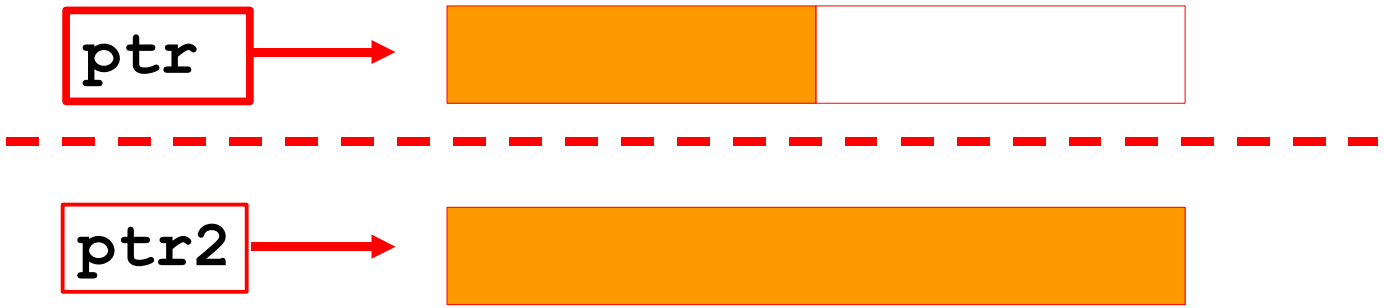
```
double *a;  /*  define a pointer  */
a = malloc(100*sizeof(double));
 /*  memory for 100 elements of type double
 allocated – they have "random" values*/
a[5] = 4.5; /*  use a as an array  */
```

- Używamy **`calloc`** kiedy chcemy zainicjalizować zaalokowaną pamięć:

```
double *a;  /*  define a pointer  */
a = calloc(100, sizeof(double));
 /*  memory for 100 elements of type double
 allocated and initialized with 0s*/
a[5] = 4.5; /*  use a as an array  */
```

Realokacja pamięci

- ptr2 = realloc(ptr, num_bytes);
- Koncepcyjnie wykonuje następujące czynności:
  - Znajduje miejsce na nowy obszar pamięci
  - Kopiuje dane oryginalne w nowe miejsce
  - Zwalnia stary obszar pamięci
  - Zwraca wskaźnik do nowego obszaru



Alokacja dynamiczna

- `int *ip = malloc(...);`
- `malloc` może zaalokować więcej niż od niego żądano
- Dlaczego?
- Dla poprawy wydajności
- Zazwyczaj gdy poprosimy o 1 bajt otrzymamy 8.
- Rozważmy linię programu:
- `char *cp = malloc(1);`
- Co jest bardziej prawdopodobne
  - Program będzie używał zaalokowanego obszaru 1 bajtu
  - Program w przyszłości powiększy obszar przy pomocy `realloc`
- Jak wiele można bezpiecznie używać?

Bezpieczeństwo

- Program powinien używać jedynie obszaru pamięci, którego zażądał
- Poważny problem
- Program który wykracza poza zaalokowany obszar może działać
- Czasem!
- Rozważmy...

```
char *cp = malloc(1);
```

ADDR SIZE

cp 8 (być może!)

- Następnie...

```
realloc(cp,6);
```

- zwróci ten sam wskaźnik, więc...

realloc

- Może zwrócić ten sam wskaźnik, który otrzymał i nie jest to błąd
- Używanie pamięci poza zaalokowanym obszarem może działać
- Czasami
- Zazwyczaj działa w momencie testowania przez programistę, a przestaje u klienta

realloc

- Realloc może zwrócić
  - ten sam wskaźnik
  - inny wskaźnik
  - NULL
- Czy to dobry pomysł:

```
cp = realloc(cp, n);
```

- Nie!
- Jeżeli **realloc** zwróci NULL tracimy poprzednią wartość **cp**
- Wyciek pamięci!

Jak to zrobić poprawnie

```
void *tmp;
if((tmp = realloc(cp,...)) == NULL)
{
    /* realloc error */
}
else
{
    cp = tmp;
}
```

Dodatkowe informacje

- **`realloc(NULL, n)` ≡ `malloc(n);`**

- **`realloc(cp, 0)` ≡ `free(cp);`**

- Powyższe zależności mogą być wykorzystane przy używaniu **`realloc`** w pętli w celu budowy dynamicznej struktury takiej jak lista z dowiązaniami.
- Niektórzy programiści definiują funkcje opakowujące funkcje alokujące pamięć

```
void* xmalloc(size_t size)
{
 void* ptr = malloc(size);
 if(!ptr) abort(); else return ptr;
}
```

Dynamiczny stos

```
/* stack.h */
void push(int a);
int pop(void);
int peek(void);
void clear(void);
void init(void);
void finalize(void);
int empty(void);
```

```
/* stack.c */
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include "stack.h"
static unsigned int top;
                  /* first free slot on the stack */
static int *data;
static unsigned int size;
void init(void)
{
        top=0;
        size=0;
        data=0;
}
void finalize(void)
{
  free(data);
}
void clear(void)
{
        top=0;
}
int empty(void)
{
  return(top==0);
}
```

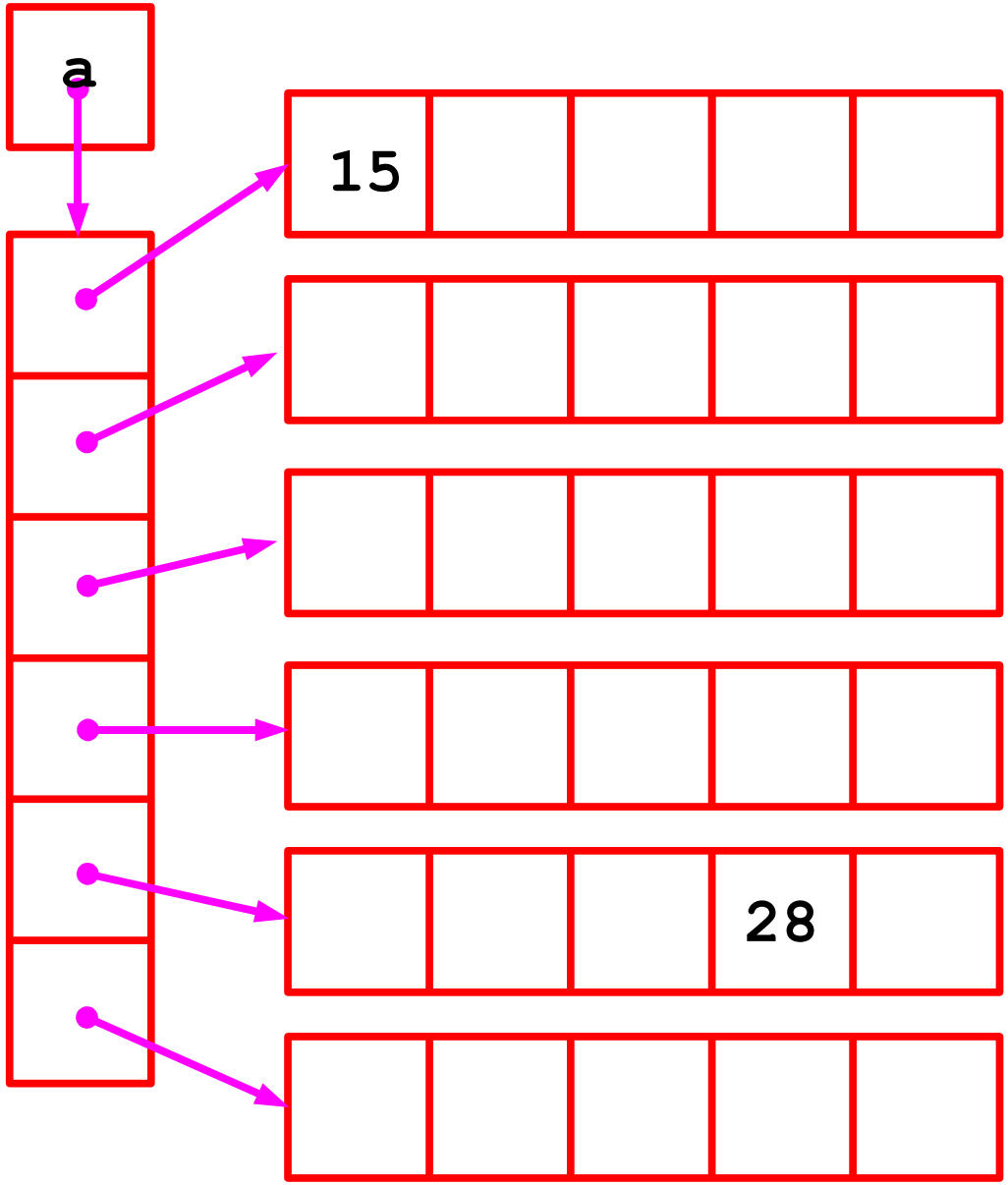
Dynamiczny stos

```
void push(int a)
{
        if(top>=size)
        {
          int newsize=(size+1)*2;
          int* ndata=realloc(data,newsize*sizeof(int));
          if(ndata)
            data=ndata;
          else
          {
            free(data);
            abort();
          }
          fprintf(stderr,"Stack size %d -> %d\n",size,newsize);
          size=newsize;
        }
        data[top++]=a;
}
int pop(void)
{
        assert(top>0);
        return data[--top];
}
int peek(void)
{
        assert(top>0);
        return data[top-1];
}
```

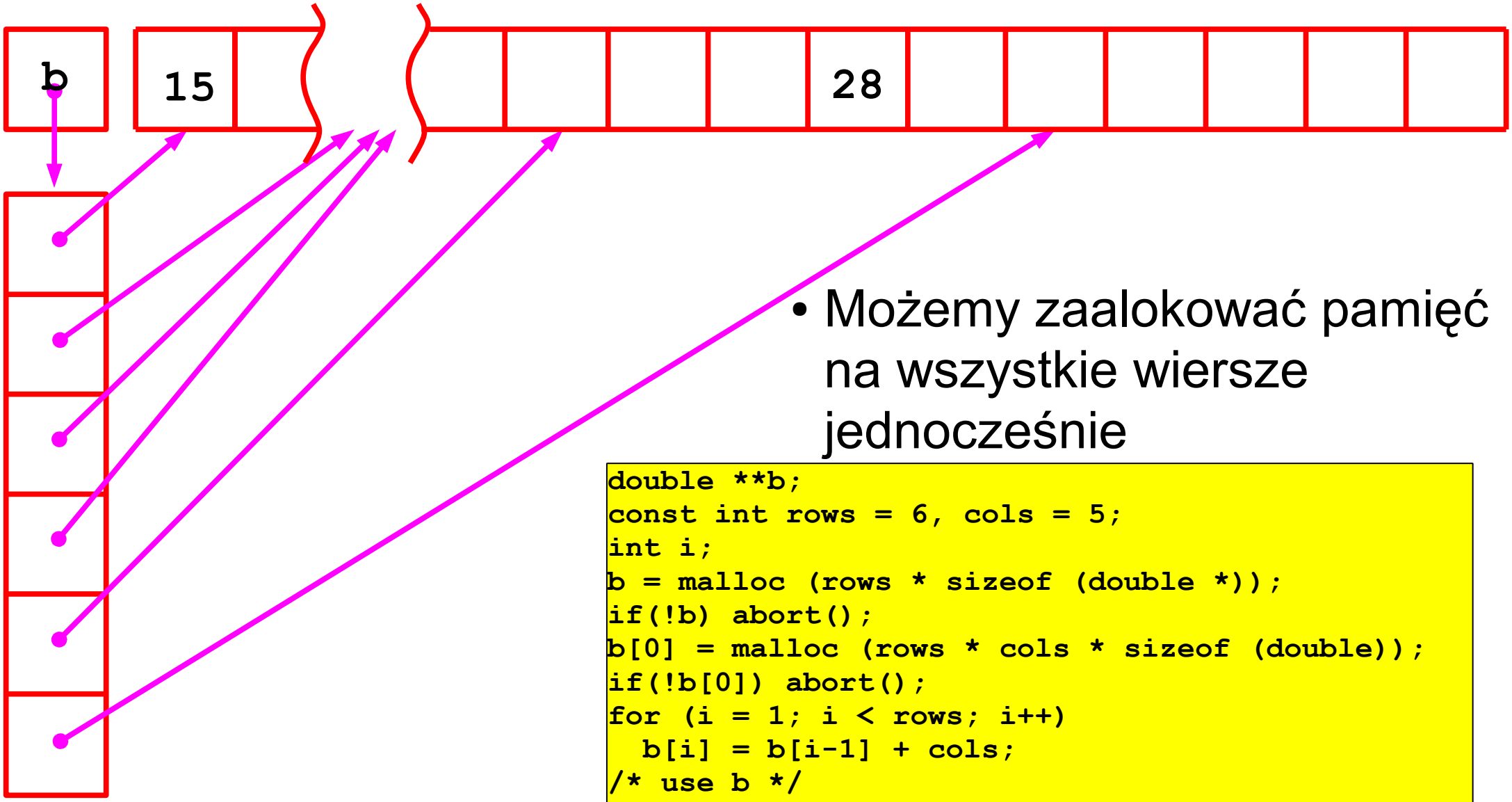
Tablice dwuwymiarowe

- Wskaźnik do wskaźnika
- Najpierw alokujemy pamięć na wskaźniki do wierszy, a potem na ich zawartość

```
int i;
const int rows = 6, cols = 5;
double **a;
a = malloc (rows * sizeof (double *));
if(!a) abort();
for (i = 0; i < rows; i++)
{
  a[i] = malloc (cols * sizeof (double));
  if(!a[i]) abort();
}
/* use a */
a[0][0]=15;
a[4][3]=28;
/* free a */
for (i = 0; i < rows; i++)
  free (a[i]);
free (a);
```



Tablice dwuwymiarowe



- Możemy zaalokować pamięć na wszystkie wiersze jednocześnie

```
double **b;
const int rows = 6, cols = 5;
int i;
b = malloc (rows * sizeof (double *));
if(!b) abort();
b[0] = malloc (rows * cols * sizeof (double));
if(!b[0]) abort();
for (i = 1; i < rows; i++)
  b[i] = b[i-1] + cols;
/* use b */
b[0][0]=15;
b[4][3]=28;
/* free b */
free (b[0]);
free (b);
```

strdup

- **strdup** tworzy kopię łańcucha
- To nie jest funkcja standardowa, ale jest dostępna w wielu systemach
- Jeżeli nasz system nie ma tej funkcji, możemy zdefiniować ją sami

```
char* strdup(const char* s)
{
 char* p = 0;
 p = malloc(strlen(s)+1);
 if (p)
  strcpy(p, s);
 return p;
}
```

- Dlaczego +1? Dlaczego sprawdzamy p? Kto jest właścicielem wskaźnika?

strdup

- Wywołanie **strdup** ...

```
int main()
{
 /* Make a copy: strdup allocates memory! */
 char* copy = strdup("surgeon");

 /* Use a copy */
 printf("Like a %s\n", copy);

 /* Deallocate memory */
 free(copy);
 copy = NULL; /* So we don't accidentally use it */
 return 0;
}
```

free

- Musimy zwolnić pamięć kiedy przestajemy jej używać
- Zawsze ustawiajmy zmienną na NULL po jej zwolnieniu (Dlaczego?)

```
char* psz = strdup("Hello");
free(psz);
psz = 0;
```

- Nie wolno zwalniać tego samego bloku pamięci dwukrotnie:

```
char* psz = strdup("Hello");
free(psz);
free(psz);/* boom! */
```

- Nie wolno zwalniać wskaźnika wskazującego na obszar pamięci zaalokowany statycznie:

```
char* psz = "Hello";
free(psz);/* bye bye! */
```

Wycieki pamięci

- Wycieki pamięci pojawiają się, kiedy zapomnimy wywołać **`free`**
- W przeciwieństwie do języka Java, C nie ma automatycznego odśmiecania
- Szczególnie szkodliwe w przypadku długotrwale wykonujących się procesów które wykonują wiele alokacji (np. serwery, demony)
- Zazwyczaj to wynik
  - Zapominalstwa
  - Wielu ścieżek powrotu z funkcji
  - Przypisania nowej wartości do wskaźnika przed wywołaniem **`free`**
  - Niezwolnienie elementów struktury po zwolnieniu struktury
  - Nieświadomość, że funkcja wywoływana alokuje pamięć, którą funkcja wywołująca powinna zwolnić

Wycieki pamięci

- Wycieki pamięci mogą być bardzo trudne do wyśledzenia
- Musimy starannie prześledzić zmienne związane z alokacją pamięci, linia po linii, od narodzin do śmierci
- Istnieją narzędzia, które w tym pomagają – ElectricFence, valgrind
- Co jeszcze może wyciekać poza pamięcią?

Alokacja dynamiczna - co może się nie udać

- Alokowanie bloku pamięci i użycie zawartości bez inicjalizacji
- Zwolnienie bloku, ale dalsze uzywanie jego zawartości
- Wywołanie **`realloc`** w celu rozszerzenia bloku pamięci i używanie starego adresu
- Alokacja bloku pamięci i zgubienie go poprzez zgubienie wartości wskaźnika
- Odczyt lub zapis poza granicami bloku
- NIEZAUWAŻENIE LUB ZIGNOROWANIE BŁĘDÓW

Typowe błędy i `valgrind`

```
#include <stdio.h>          /* 1*/
#include <stdlib.h>         /* 2*/
#include <assert.h>         /* 3*/
                            /* 4*/
int                         /* 5*/
main ()                     /* 6*/
{                           /* 7*/
  char* p1, *p2;            /* 8*/
                            /* 9*/
  p1=malloc(10);            /*10*/
  printf("%c\n",p1[0]);     /*11*/
  free(p1);                 /*12*/
  *p1='a';                  /*13*/
  p1=malloc(10);            /*14*/
  p2=realloc(p1,10000);     /*15*/
  *p1='b';                  /*16*/
  malloc(30);               /*17*/
  p2[10000]='c';            /*18*/
  p1=malloc(2000000000);    /*19*/
  *p1='c';                  /*20*/
  return 0;                 /*21*/
};                          /*22*/
```

```
$ valgrind --leak-check=yes ./errors 2>rep
```

Typowe błędy i valgrind

```
#include <stdio.h>          /* 1*/
#include <stdlib.h>         /* 2*/
#include <assert.h>         /* 3*/
                            /* 4*/
int                         /* 5*/
main ()                     /* 6*/
{                           /* 7*/
  char* p1, *p2;            /* 8*/
                            /* 9*/
  p1=malloc(10);            /*10*/
  printf("%c\n",p1[0]);     /*11*/
  free(p1);                 /*12*/
  *p1='a';                  /*13*/
  p1=malloc(10);            /*14*/
  p2=realloc(p1,10000);     /*15*/
  *p1='b';                  /*16*/
  malloc(30);               /*17*/
  p2[10000]='c';            /*18*/
  p1=malloc(2000000000);/*19*/
  *p1='c';                  /*20*/
  return 0;                 /*21*/
};                          /*22*/
```

```
$ valgrind --leak-check=yes ./errors 2>rep

Conditional jump or move depends on uninitialised value(s)
 at 0x4027ACE2: _IO_vfprintf (in /lib/libc-2.2.5.so)
 by 0x402823B5: _IO_printf (in /lib/libc-2.2.5.so)
 by 0x8048428: main (errors.c:11)
 by 0x4024814E: __libc_start_main (in /lib/libc-2.2.5.so)

Syscall param write(buf) contains uninitialised or unaddressable byte(s)
 at 0x402F2404: __libc_write (in /lib/libc-2.2.5.so)
 by 0x40298E87: (within /lib/libc-2.2.5.so)
 by 0x40298DE5: _IO_do_write (in /lib/libc-2.2.5.so)
 by 0x4029913F: _IO_file_overflow (in /lib/libc-2.2.5.so)
 Address 0x40228000 is not stack'd, malloc'd or free'd
```

Typowe błędy i valgrind

```
#include <stdio.h>          /* 1*/
#include <stdlib.h>         /* 2*/
#include <assert.h>         /* 3*/
                            /* 4*/
int                         /* 5*/
main ()                     /* 6*/
{                           /* 7*/
  char* p1, *p2;            /* 8*/
                            /* 9*/
  p1=malloc(10);            /*10*/
  printf("%c\n",p1[0]);     /*11*/
  free(p1);                 /*12*/
  *p1='a';                  /*13*/
  p1=malloc(10);            /*14*/
  p2=realloc(p1,10000);     /*15*/
  *p1='b';                  /*16*/
  malloc(30);               /*17*/
  p2[10000]='c';            /*18*/
  p1=malloc(2000000000);    /*19*/
  *p1='c';                  /*20*/
  return 0;                 /*21*/
};                          /*22*/
```

```
$ valgrind --leak-check=yes ./errors 2>rep

Invalid write of size 1
 at 0x8048437: main (errors.c:13)
 by 0x4024814E: __libc_start_main (in /lib/libc-2.2.5.so)
 by 0x8048350: (within /home/gwj/pdsc/03-lecture04/04-memerrors/errors)
 Address 0x41050024 is 0 bytes inside a block of size 10 free'd
 at 0x4002698D: free (vg_replace_malloc.c:231)
 by 0x8048433: main (errors.c:12)
 by 0x4024814E: __libc_start_main (in /lib/libc-2.2.5.so)
 by 0x8048350: (within /home/gwj/pdsc/03-lecture04/04-memerrors/errors)
```

Typowe błędy i valgrind

```
#include <stdio.h>          /* 1*/
#include <stdlib.h>         /* 2*/
#include <assert.h>         /* 3*/
                            /* 4*/
int                         /* 5*/
main ()                     /* 6*/
{                           /* 7*/
  char* p1, *p2;            /* 8*/
                            /* 9*/
  p1=malloc(10);            /*10*/
  printf("%c\n",p1[0]);     /*11*/
  free(p1);                 /*12*/
  *p1='a';                  /*13*/
  p1=malloc(10);            /*14*/
  p2=realloc(p1,10000);     /*15*/
  *p1='b';                  /*16*/
  malloc(30);               /*17*/
  p2[10000]='c';            /*18*/
  p1=malloc(2000000000);    /*19*/
  *p1='c';                  /*20*/
  return 0;                 /*21*/
};                          /*22*/
```

```
$ valgrind --leak-check=yes ./errors 2>rep

Invalid write of size 1
 at 0x8048462: main (errors.c:16)
 by 0x4024814E: __libc_start_main (in /lib/libc-2.2.5.so)
 by 0x8048350: (within /home/gwj/pdsc/03-lecture04/04-memerrors/errors)
 Address 0x41050060 is 0 bytes inside a block of size 10 free'd
 at 0x40026C58: realloc (vg_replace_malloc.c:310)
 by 0x804845B: main (errors.c:15)
 by 0x4024814E: __libc_start_main (in /lib/libc-2.2.5.so)
 by 0x8048350: (within /home/gwj/pdsc/03-lecture04/04-memerrors/errors)
```

Typowe błędy i valgrind

```
#include <stdio.h>          /* 1*/
#include <stdlib.h>         /* 2*/
#include <assert.h>         /* 3*/
                            /* 4*/
int                         /* 5*/
main ()                     /* 6*/
{                           /* 7*/
  char* p1, *p2;            /* 8*/
                            /* 9*/
  p1=malloc(10);            /*10*/
  printf("%c\n",p1[0]);     /*11*/
  free(p1);                 /*12*/
  *p1='a';                  /*13*/
  p1=malloc(10);            /*14*/
  p2=realloc(p1,10000);     /*15*/
  *p1='b';                  /*16*/
  malloc(30);               /*17*/
  p2[10000]='c';            /*18*/
  p1=malloc(2000000000);    /*19*/
  *p1='c';                  /*20*/
  return 0;                 /*21*/
};                          /*22*/
```

```
$ valgrind --leak-check=yes ./errors 2>rep

Invalid write of size 1
 at 0x8048479: main (errors.c:18)
 by 0x4024814E: __libc_start_main (in /lib/libc-2.2.5.so)
 by 0x8048350: (within /home/gwj/pdsc/03-lecture04/04-memerrors/errors)
 Address 0x410527AC is 0 bytes after a block of size 10000 alloc'd
 at 0x40026C58: realloc (vg_replace_malloc.c:310)
 by 0x804845B: main (errors.c:15)
 by 0x4024814E: __libc_start_main (in /lib/libc-2.2.5.so)
 by 0x8048350: (within /home/gwj/pdsc/03-lecture04/04-memerrors/errors)

VG_(get_memory_from_mmap): request for 2000003072 bytes failed.
VG_(get_memory_from_mmap): 14933038 bytes already allocated.

This may mean that you have run out of swap space,
since running programs on valgrind increases their memory
usage at least 3 times.  You might want to use 'top'
to determine whether you really have run out of swap.
If so, you may be able to work around it by adding a
temporary swap file -- this is easier than finding a
new swap partition.  Go ask your sysadmin(s) [politely!]

VG_(get_memory_from_mmap): out of memory!  Fatal!  Bye!
```

Typowe błędy i valgrind

```
#include <stdio.h>          /* 1*/
#include <stdlib.h>         /* 2*/
#include <assert.h>         /* 3*/
                            /* 4*/
int                         /* 5*/
main ()                     /* 6*/
{                           /* 7*/
  char* p1, *p2;            /* 8*/
                            /* 9*/
  p1=malloc(10);            /*10*/
  printf("%c\n",p1[0]);     /*11*/
  free(p1);                 /*12*/
  *p1='a';                  /*13*/
  p1=malloc(10);            /*14*/
  p2=realloc(p1,10000);     /*15*/
  *p1='b';                  /*16*/
  malloc(30);               /*17*/
  p2[10000]='c';            /*18*/
                            /*19*/
                            /*20*/
  return 0;                 /*21*/
};                          /*22*/
```

```
$ valgrind --leak-check=yes ./errors2 2>rep

ERROR SUMMARY: 5 errors from 5 contexts (suppressed: 2 from 1)
malloc/free: in use at exit: 10030 bytes in 2 blocks.
malloc/free: 4 allocs, 2 frees, 10050 bytes allocated.
For counts of detected errors, rerun with: -v
searching for pointers to 2 not-freed blocks.
checked 3413768 bytes.

30 bytes in 1 blocks are definitely lost in loss record 1 of 2
 at 0x400266DE: malloc (vg_replace_malloc.c:153)
 by 0x8048470: main (errors2.c:17)
 by 0x4024814E: __libc_start_main (in /lib/libc-2.2.5.so)
 by 0x8048350: (within /home/gwj/pdsc/03-lecture04/04-memerrors/errors2)

10000 bytes in 1 blocks are definitely lost in loss record 2 of 2
 at 0x40026C58: realloc (vg_replace_malloc.c:310)
 by 0x804845B: main (errors2.c:15)
 by 0x4024814E: __libc_start_main (in /lib/libc-2.2.5.so)
 by 0x8048350: (within /home/gwj/pdsc/03-lecture04/04-memerrors/errors2)

LEAK SUMMARY:
 definitely lost: 10030 bytes in 2 blocks.
 possibly lost:   0 bytes in 0 blocks.
 still reachable: 0 bytes in 0 blocks.
 suppressed: 0 bytes in 0 blocks.
 Reachable blocks (those to which a pointer was found) are not shown.
 To see them, rerun with: --show-reachable=yes
```

Struktury

- Kolekcja składowych
- Bardzo przydatna do grupowania powiązanych danych

```
struct Person
{
        char* name;
        int age;
};
```

Stworzenie osoby

- Dostęp do składowych uzykujemy przy pomocy kropki (.)

```
int main()
{
        struct Person artist;
        artist.name = strdup("Kayah");
        artist.age = 37;
       return 0;
}
```

- Czy ktoś zauważył problem?

Inna metoda inicjalizacji struktury

- Podobnie jak w przypadku tablicy, możemy użyć listy inicjalizatorów
- W C90, lista inicjalizatorów może zawierać jedynie stałe
- W C99, możemy przenieść **`strdup`** do listy inicjalizatorów

```
int main()
{
   struct Person artist = { NULL, 37 };
   artist.name=strdup("Kayah");
   if(!artist.name) abort();
   ... use artist ...

   /* Free memory allocated by strdup */
   free(artist.name);
   artist.name = 0;
   return 0;
}
```

Stworzenie osoby dynamicznie

- Możemy zaalokować obszar na strukturę używając **`malloc()`**
- Dostęp do składowych struktury za pomocą wskaźnika uzyskujemy przy pomocy ->

```
int main()
{
  struct Person* artist = malloc(sizeof(struct Person));
  if(!artist) abort();
  artist->name = strdup("Kayah");
  if(!artist->name) abort();
  artist->age = 37;

  ... exploit artist ...

  /* First, free the member variables */
  free(artist->name);
  artist->name = 0; /* So we don't use it */

  /* Then, free the structure */
  free(artist);
  artist = 0; /* So we don't use it */
  return 0;
}
```

Struktury i `typedef`

- Możemy uprościć użycie struktur przy pomocy typedef:

```
typedef struct Person SPerson;
```

- Podobnie jak w przypadku typów wyliczeniowych możemy połączyć deklarację struktury i typedef:

```
typedef struct Person {
  char* name;
  int age;
} SPerson, *SPersonPtr;
```

- Odtąd możemy pisać **`SPerson`** zamiast **`struct Person`**

Zagnieżdżone struktury

- Struktury można dowolnie zagnieżdżać

```
typedef struct Date {
  int mon, day, year;
} SDate, *SDatePtr;

typedef struct Person {
  char* name;
  SDate dob;
} SPerson, *SPersonPtr;
```

Zagnieżdżone struktury

- Dostęp do składowych poprzez (.) i (->):

```
int
main ()
{
  SPerson *artist = malloc (sizeof (SPerson));
  if(!artist) abort();

  artist->name = strdup ("Kayah");
  if(!artist->name) abort();
  artist->dob.day = 5;
  artist->dob.mon = 11;
  artist->dob.year = 1967;

  /* Free memory allocated by strdup */
  free (artist->name);
  artist->name = 0;

  /* Free the person */
  free (artist);
  artist = 0;
  return 0;
}
```

Struktury rekursywne

- Struktura może mieć składowe typu wskaźnikowego do właśnie deklarowanego typu:

```
typedef struct Person {
  char* name;
  SDate dob;
  struct Person* parents[2];
} SPerson, *SPersonPtr;
```

Struktury rekursywne

```
SPerson parents[2];
SPerson artist;

parents[0].name = strdup("Kayah's Mother");
parents[1].name = strdup("Kayah's Father");

artist.name = strdup("Kayah");
artist.parents[0] = &parents[0];
artist.parents[1] = &parents[1];

printf("%s's parents are %s and %s\n",
        artist.name, artist.parents[0]->name,
        artist.parents[1]->name);
```

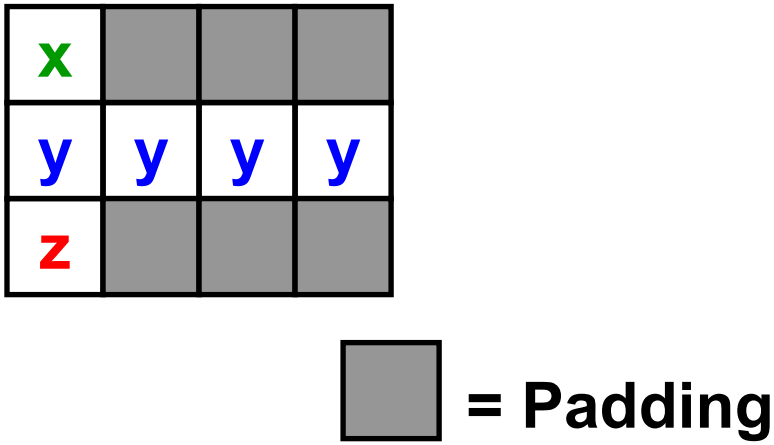
- Najczęściej w takich przypadkach używa się dynamicznej alokacji pamięci

Struktury i sizeof

- Z powodu ograniczeń na wyrównanie poszczególnych pól, rozmiar struktury >= sumy rozmiarów wszystkich pól

```
struct blah {
  char x;
  int  y;
  char z;
};
```

Memory layout



- Zawsze używajmy sizeof w celu określenia rozmiaru struktury
- **`sizeof(struct blah)` ≡ 12 bytes**

Pola bitowe w strukturach

- Mówiliśmy wcześniej o flagach bitowych i maskach
- Pola bitowe są użyteczne, kiedy potrzebujemy upakować wiele flag lub obiektów w najmniejszym możliwym obszarze pamięci
- Pola bitowe nieco to ułatwiają kosztem przenośności

```
struct argb {
  unsigned int alpha : 8;
  unsigned int red   : 8;
  unsigned int green : 8;
  unsigned int blue  : 8;
};
```

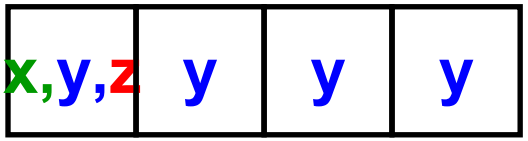
- Zależne od implementacji!

Unie

- Składniowo równoważne strukturom
- Wszystkie składowe zajmują ten sam obszar pamięci
- Programista odpowiada za dostęp do właściwych składowych we właściwym czasie
- Rozmiar unii to rozmiar jej największej składowej

```
union UBlah {
  char x;
  int  y;
  char z;
};
```

Memory layout



Unie

```
union {
    char   x;
    int  y;
    char*  z;
} utype;

utype.x = 'c';
printf("%c\n", utype.x);
utype.z = "Hello";
printf("%s\n", utype.z);
printf("%d\n", utype.y); /* Undefined! */
```