Wykład 9: Przydział i zarządzanie pamięcią.

dr inż. Andrzej Stafiniak

Wrocław 2023

Pamięć komputerowa

architektura von Neumanna - vs. - architektura harwardzka ?

Pamięć komputerowa

- **Pamięć komputerowa** - ?

Pamięć komputerowa

- **Pamięć komputerową** - stanowią „*moduły komputera, służące do przechowywania danych i programów*”, czyli informacji. [*wikipedia.org*]
- Ogólny podział **pamięci komputerowej**: ?

Pamięć komputerowa

- **Pamięć komputerową** - stanowią „*moduły komputera, służące do przechowywania danych i programów*”, czyli informacji. [*wikipedia.org*]

- Ogólny podział **pamięci komputerowej**:

Pamięć **wewnętrzna** (podstawiowa, primary storage) – bezpośrednio dostępna dla procesora. Znajdują się w niej informacje, które są potrzebne w trakcie pracy komputera dla bieżąco wykonywanych programów. Jest to najczęściej pamięć ulotna RAM (*Random Access Memory*):
- pamięć operacyjna,
- rejestry procesora,
- pamięć podręczna procesora,

ale również pamięć nieulotna ROM (*Read Only Memory*):
- BIOS.

Pamięć **zewnętrzna** (masowa, secondary storage) – procesor nie ma do niej bezpośredniego dostępu. Zaliczamy do tej grupy zarówno nośniki pamięci w formie wewnętrznych i zewnętrzach dysków twardych (HDD, SDD) oraz przenośne nośniki danych takie jak półprzewodnikowe pamięci USB (pendrive’y), płyty CD, DVD, Blu-ray.

W celu obsługi informacji zawartych na tej pamięci musi być ona najpierw wczytana do pamięci operacyjnej.

Pamięć komputerowa

- **Pamięć komputerową** - stanowią „*moduły komputera, służące do przechowywania danych i programów*”, czyli informacji. [*wikipedia.org*]

- Ogólny podział **pamięci komputerowej**:

Pamięć **wewnętrzna** (podstawiowa, primary storage) – bezpośrednio dostępna dla procesora. Znajdują się w niej informacje, które są potrzebne w trakcie pracy komputera dla bieżąco wykonywanych programów. Jest to najczęściej pamięć ulotna RAM (*Random Access Memory*):

- pamięć operacyjna,
- rejestry procesora,
- pamięć podręczna procesora,

ale również pamięć nieulotna ROM (*Read Only Memory*):

- BIOS.

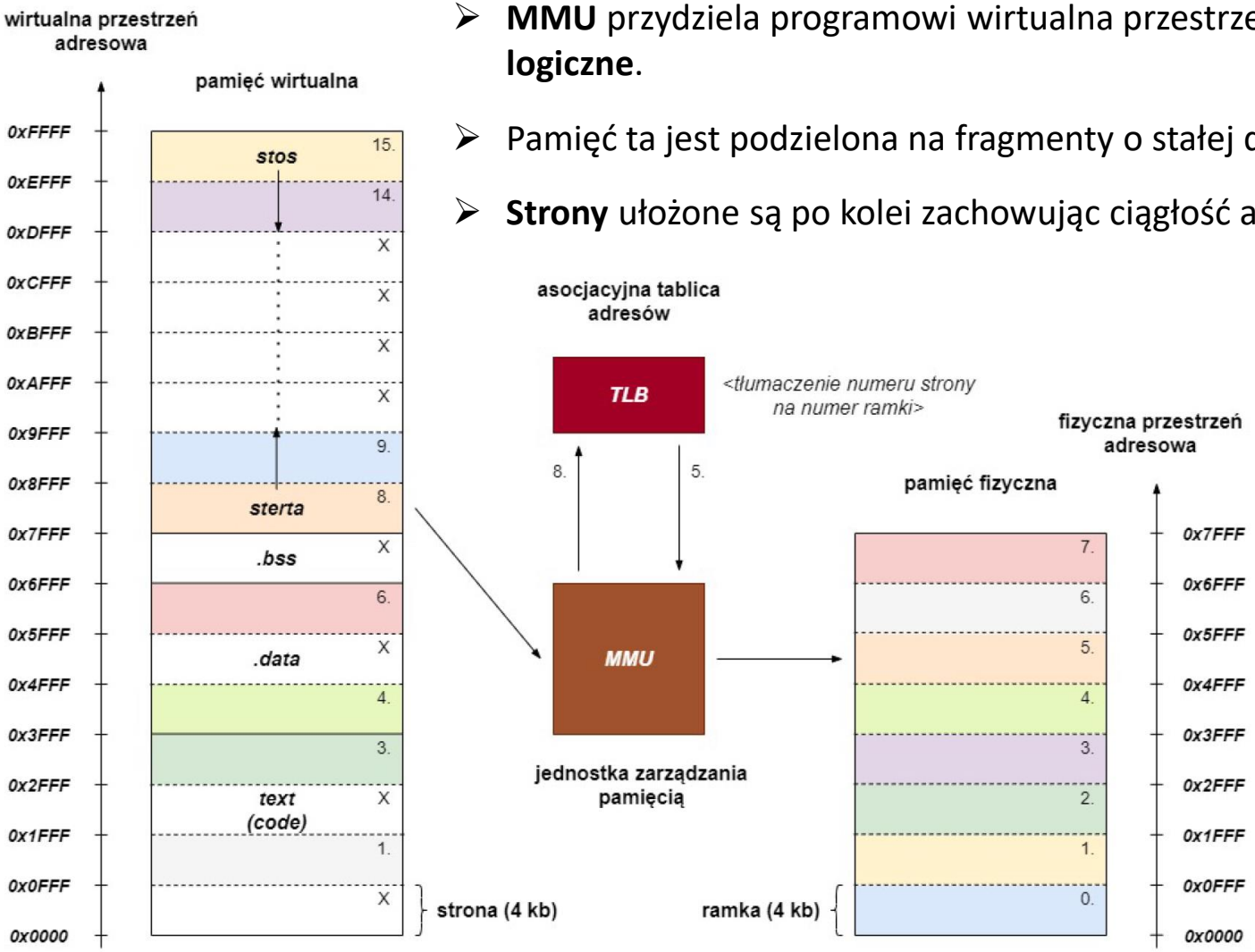
Pamięć **zewnętrzna** (masowa, secondary storage) – procesor nie ma do niej bezpośredniego dostępu. Zaliczamy do tej grupy zarówno nośniki pamięci w formie wewnętrznych i zewnętrzach dysków twardych (HDD, SDD) oraz przenośne nośniki danych takie jak półprzewodnikowe pamięci USB (pendrive’y), płyty CD, DVD, Blu-ray.

W celu obsługi informacji zawartych na tej pamięci musi być ona najpierw wczytana do pamięci operacyjnej.

Pamięć komputerowa

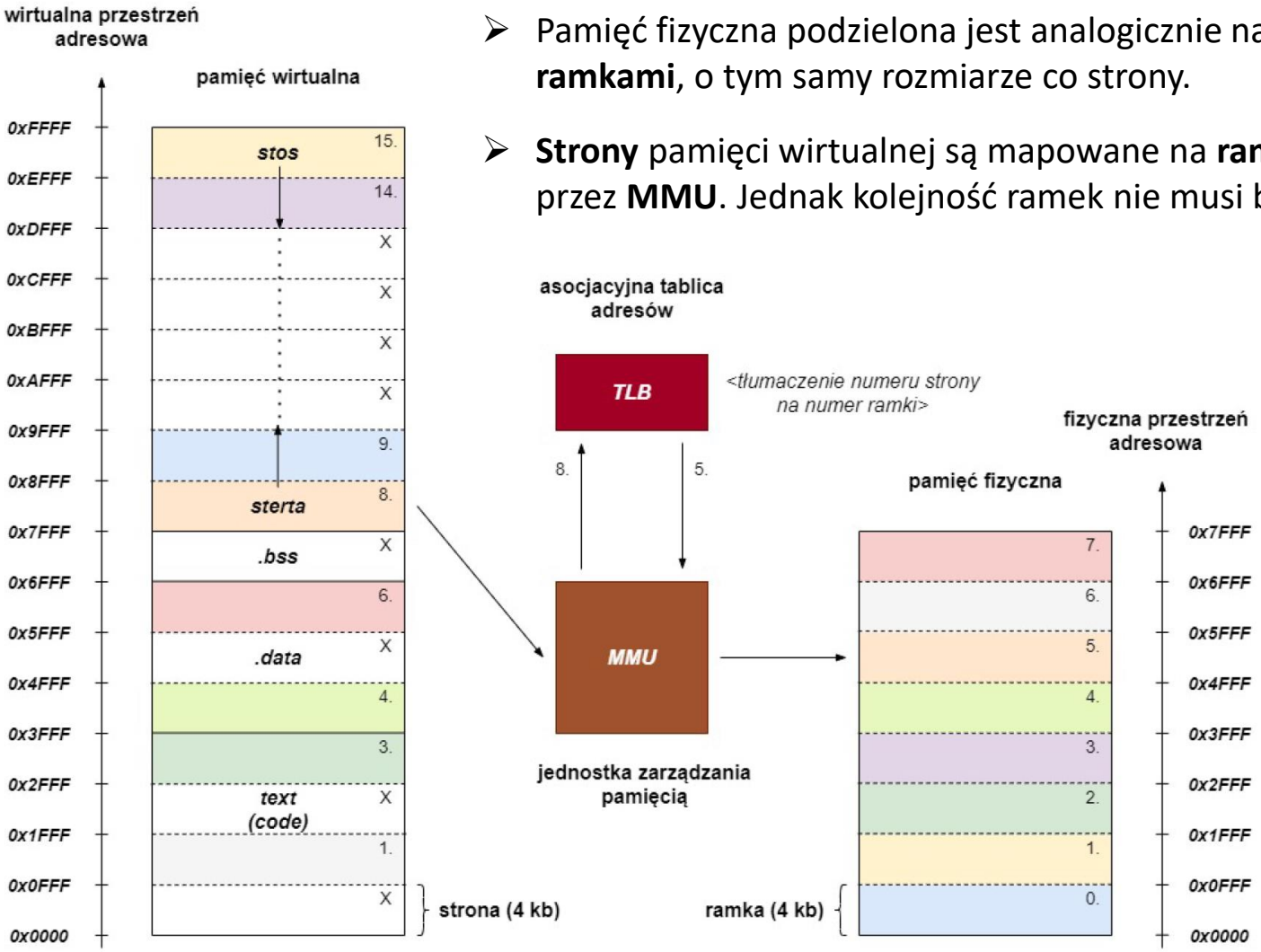
- W wyniku uruchomienia programu na komputerze, do jego obsługi zostaje przydzielona pamięć,
- jednak **nie** jest to **pamięć fizyczna** (komórki pamięci operacyjnej),
- tylko pamięć wirtualna.
- Zarządzanie pamięcią wirtualną i fizyczną jest realizowane przez **jednostkę zarzadzania pamiecią MMU** (ang. Memory Management Unit) z wykorzystaniem asocjacyjnej tablicy adresów **TLB** (ang. Translation Lookaside Buffer).

Translacja adresów logicznych na fizyczne



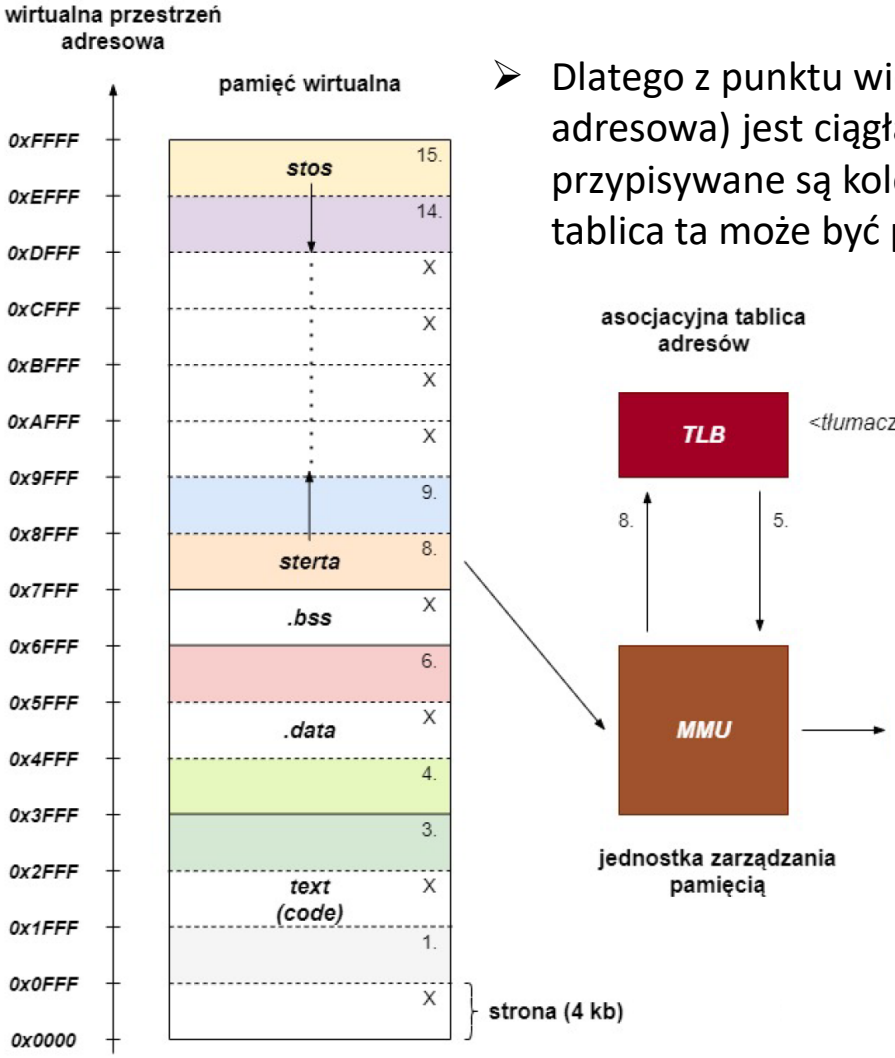
- **MMU** przydziela programowi wirtualna przestrzeń adresową czyli **adresy logiczne**.
- Pamięć ta jest podzielona na fragmenty o stałej długości zwane **stronami**.
- **Strony** ułożone są po kolei zachowując ciągłość adresów.

Translacja adresów logicznych na fizyczne



- Pamięć fizyczna podzielona jest analogicznie na fragmenty zwane **ramkami**, o tym samy rozmiarze co strony.
- **Strony** pamięci wirtualnej są mapowane na **ramki** pamięci fizycznej przez **MMU**. Jednak kolejność ramek nie musi być zachowana.

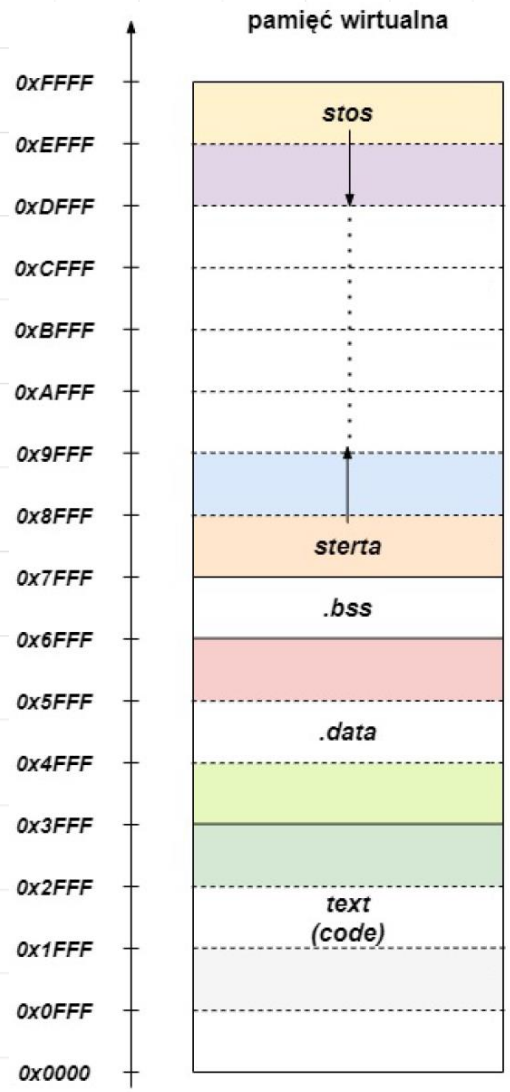
Translacja adresów logicznych na fizyczne



- Dlatego z punktu widzenia programu pamięć (wirtualna przestrzeń adresowa) jest ciągła (przykładem są elementy tablicy, którym przypisywane są kolejne adresy komórek pamięci), ale w rzeczywistości tablica ta może być podzielona na różne sektory pamięci operacyjnej RAM.

Nr strony	Zakres adresów wirtualnych	Nr ramki	Zakres adresów fizycznych
0	0x0000 – 0x0FFF	–	–
1	0x1000 – 0x1FFF	6	0x6000 – 0x6FFF
2	0x2000 – 0x2FFF	–	–
3	0x3000 – 0x3FFF	2	0x2000 – 0x2FFF
4	0x4000 – 0x4FFF	4	0x4000 – 0x4FFF
5	0x5000 – 0x5FFF	–	–
6	0x6000 – 0x6FFF	7	0x7000 – 0x7FFF
7	0x7000 – 0x7FFF	–	–
8	0x8000 – 0x8FFF	5	0x5000 – 0x5FFF
9	0x9000 – 0x9FFF	0	0x0000 – 0x0FFF
10	0xA000 – 0xAFFF	–	–
11	0xB000 – 0xBFFF	–	–
12	0xC000 – 0xCFFF	–	–
13	0xD000 – 0xDFFF	–	–
14	0xE000 – 0xEFFF	3	0x3000 – 0x3FFF
15	0xF000 – 0xFFFF	1	0x1000 – 0x1FFF

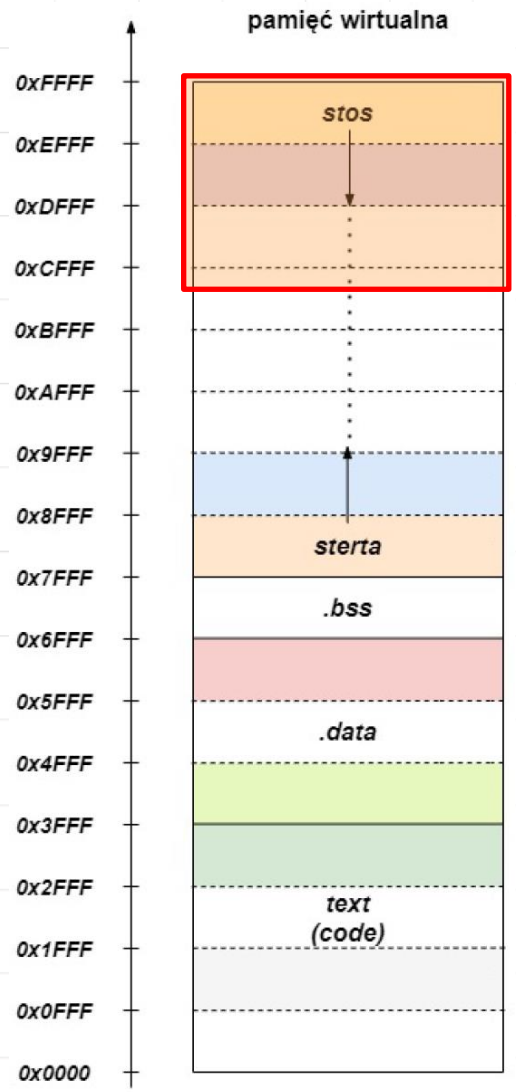
Segmenty pamięci wirtualnej



W zależności od przeznaczenia, pamięć wirtualna podzielona jest na **segmenty** :

- **stos** (*stack*)
- **sterta** (*heap*)
- **.bss** (*block started by symbol).*
- **.data**
- **text (code)**

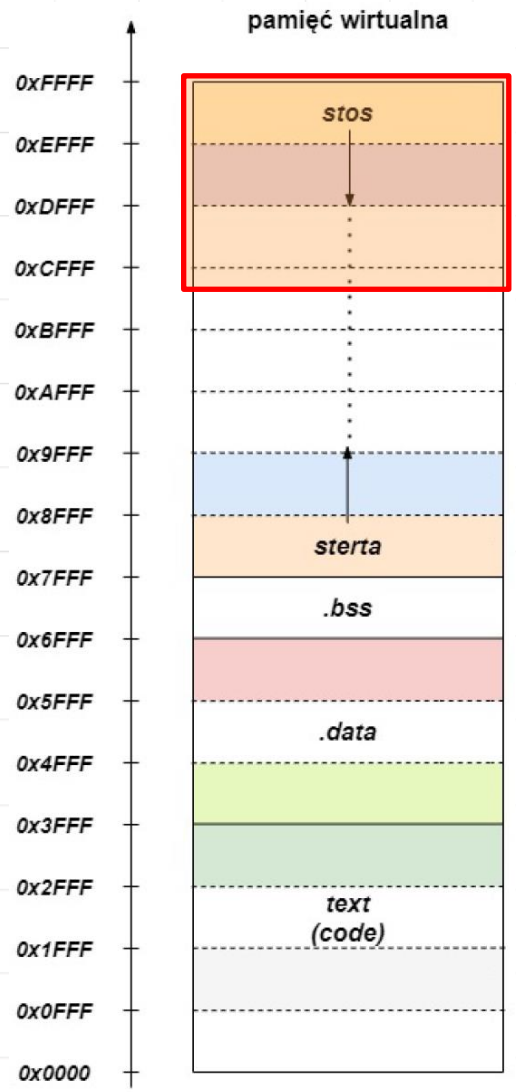
Segmenty pamięci wirtualnej



Segment **stos** (*stack*):

- Jest to liniowa struktura danych w realizacji bufora LIFO, w której elementy dodawane i usuwane sa z wierzchołka stosu (Last In, First Out – ostatni element na wejściu jest pierwszym elementem na wyjściu)
- Najczęściej **stos** ulokowany jest w górnym zakresie wirtualnej przestrzeni adresowej.
- Rozmiar stosu znany jest na etapie kompilacji, a dane w nim alokowane są zwalniane automatycznie.
- Kolejne elementy dodawane na stos powodują jego rozrost w kierunku niższych adresów.

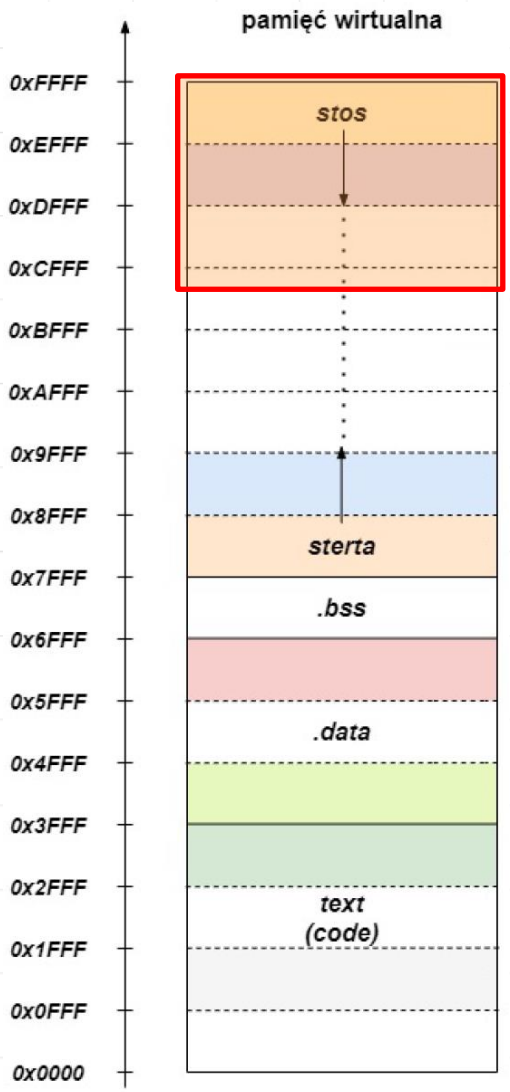
Segmenty pamięci wirtualnej



Segment **stos** (*stack*):

- na **stosie** przydzielana jest pamięć dla zmiennych lokalnych (automatycznych, zmiennych alokowanych statycznie - **ale nie zmiennych statycznych**! czas życia zmiennych lokalnych automatycznych - od definicji do końca bloku instrukcji, a zmiennych lokalnych statycznych (`static`) od deklaracji do końca programu)
- na stosie umieszcza się wartości zwracane przez funkcje oraz adresy powrotu z funkcji,
- na stosie umieszcza kopie argumentów wywołania funkcji.

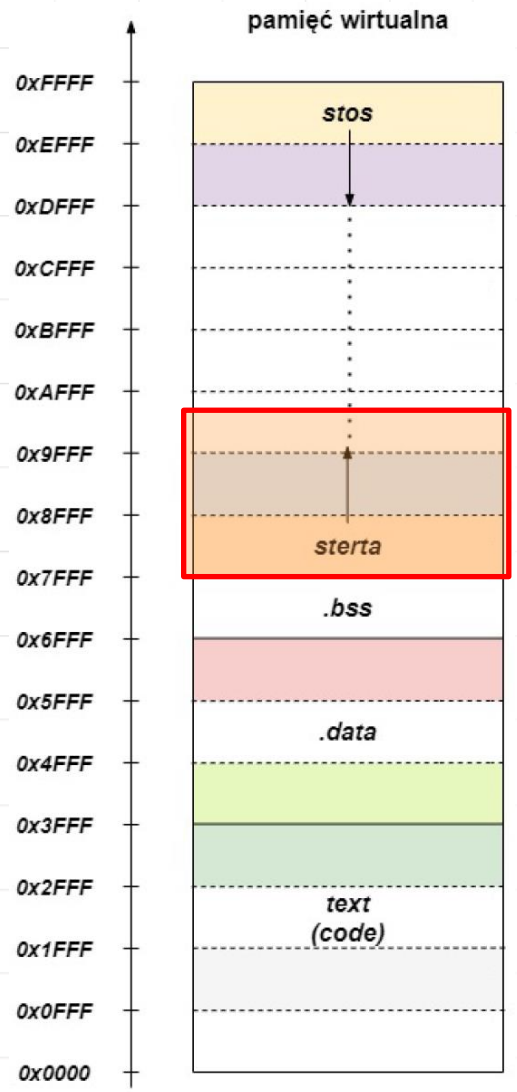
Segmenty pamięci wirtualnej




```c
#include <stdio.h>

void rozrostStosu1(int a, int b, int c, int d, int e){
    int x1 = 1, x2 = 2, x3 = 3;
    int x4 = 1, x5 = 2, x6 = 3;
    printf("\nWydrukuj adresy zmiennych alokowanych na stosie:\n");
    printf("&x1 - %#x \n", &x1);
    printf("&x2 - %#x \n", &x2);
    printf("&x3 - %#x \n", &x3);
    printf("&x4 - %#x \n", &x4);
    printf("&x5 - %#x \n", &x5);
    printf("&x6 - %#x \n", &x6);
    printf("&a - %#x \n", &a);
    printf("&b - %#x \n", &b);
    printf("&c - %#x \n", &c);
    printf("&d - %#x \n", &d);
    printf("&e - %#x \n", &e);
}

    void rozrostStosu2(int aa, int bb, int cc, int dd, int ee){
    int xx1 = 1, xx2 = 2, xx3 = 3;
    int xx4 = 1, xx5 = 2, xx6 = 3;
    printf("\nWydrukuj adresy zmiennych alokowanych na
    printf("&xx1 - %#x \n", &xx1);
    printf("&xx2 - %#x \n", &xx2);
    printf("&xx3 - %#x \n", &xx3);
    printf("&xx4 - %#x \n", &xx4);
    printf("&xx5 - %#x \n", &xx5);
    printf("&xx6 - %#x \n", &xx6);
    printf("&aa - %#x \n", &aa);
    printf("&bb - %#x \n", &bb);
    printf("&cc - %#x \n", &cc);
    printf("&dd - %#x \n", &dd);
    printf("&ee - %#x \n\n", &ee);
}

int main(){
    rozrostStosu1(1, 2, 3, 4, 5);
    rozrostStosu2(6, 7, 8, 9, 0);
    return 0;
}
```

```
Wydrukuj adresy zmiennych alokowanych na stosie:
&x1 - 0x61feec
&x2 - 0x61fee8
&x3 - 0x61fee4
&x4 - 0x61fee0
&x5 - 0x61fedc
&x6 - 0x61fed8
&a - 0x61ff00
&b - 0x61ff04
&c - 0x61ff08
&d - 0x61ff0c
&e - 0x61ff10

Wydrukuj adresy zmiennych alokowanych na stosie:
&xx1 - 0x61feec
&xx2 - 0x61fee8
&xx3 - 0x61fee4
&xx4 - 0x61fee0
&xx5 - 0x61fedc
&xx6 - 0x61fed8
&aa - 0x61ff00
&bb - 0x61ff04
&cc - 0x61ff08
&dd - 0x61ff0c
&ee - 0x61ff10
```

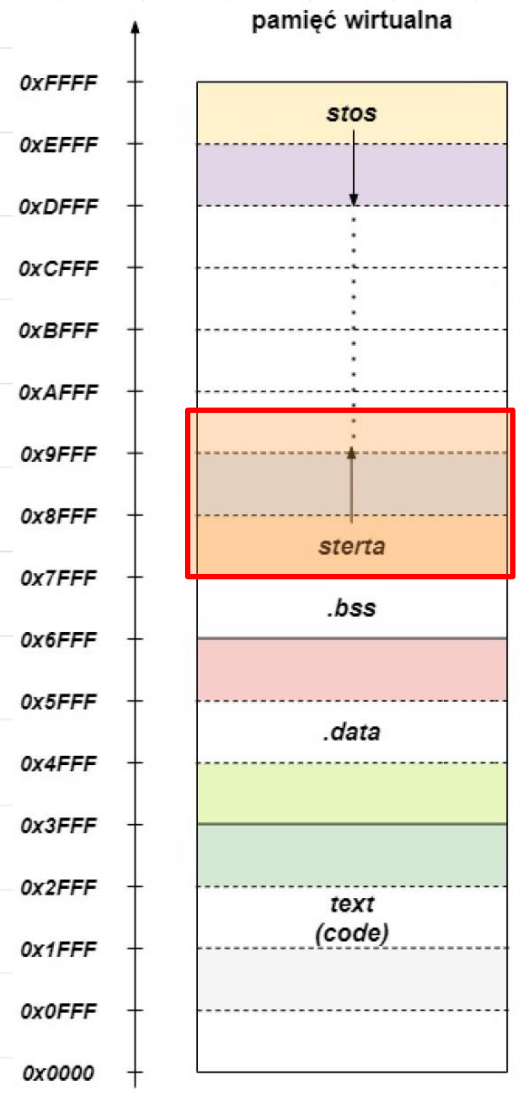
Segmenty pamięci wirtualnej



Segment **sterta** (*heap*):

- jest to największy obszar pamięci dostępny dla programu,
- służy do przechowywania zmiennych alokowanych dynamicznie (rozmiar pamięci alokowanej na stercie nie jest znany w czasie kompilacji programu),
- przechowuje niepowiązane ze sobą elementy (w sposób nieuporządkowany, możliwa fragmentacja pamięci), do których można odwołać się z każdego miejsca w programie, jeżeli zna się ich adresy,

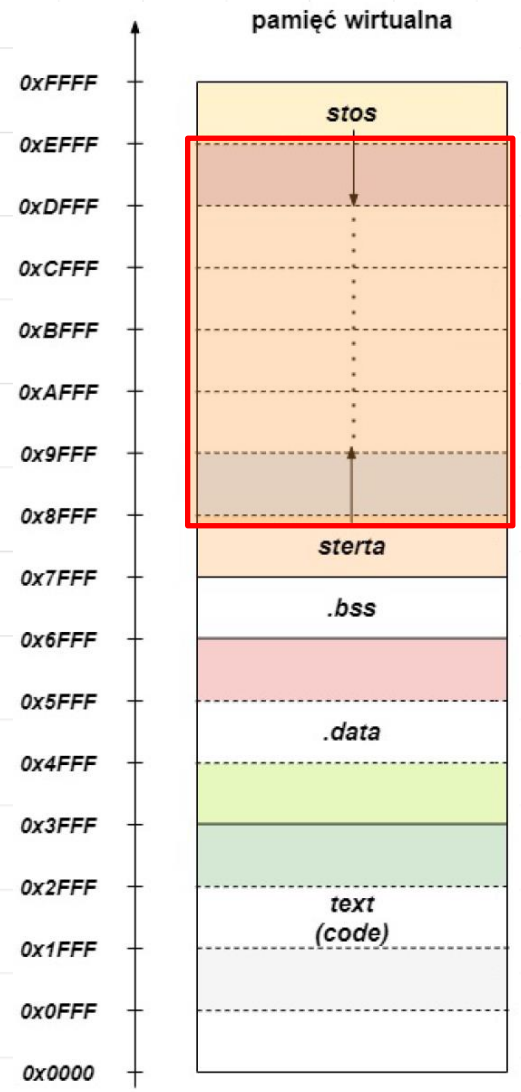
Segmenty pamięci wirtualnej



Segment **sterta** (*heap*):

- do alokacji pamięci na stercie wykorzystuje się wskaźniki oraz dedykowane funkcje/operatory:
- funkcje (język C) - **`malloc(), calloc(), realloc(),`**
- operatory (język C++) - **`new, new[],`**
- pamięć zaalokowana na stercie nie jest zwalniana w sposób automatyczny, język C/C++ nie zapewniają mechanizmu automatycznego odśmiecania (ang. **garbage collector**) znanego z języków wysokiego poziomu,
- musimy pamiętać o zwolnieniu zalakowanej pamięci, funkcja **`free()`** (język C), operator **`delete, delete [].`**

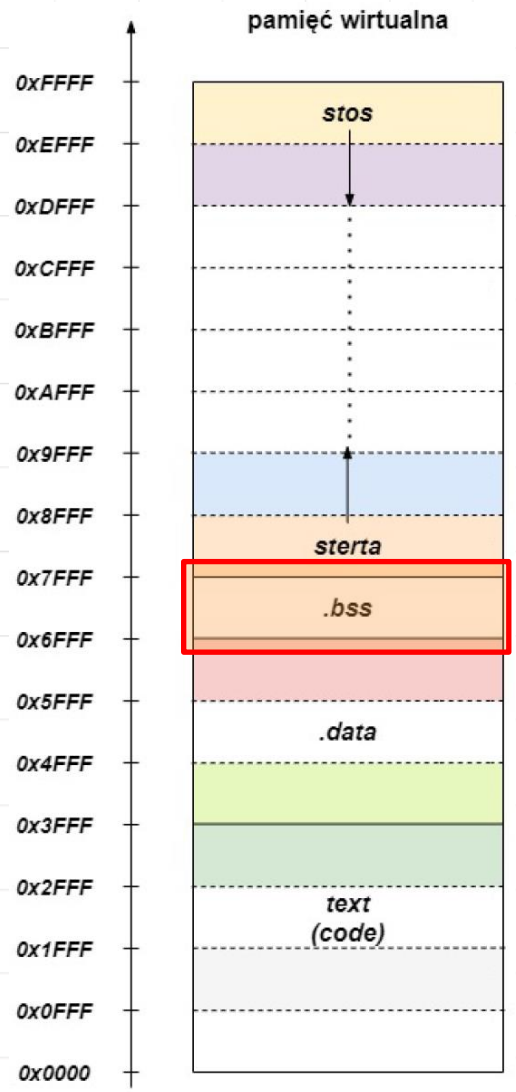
Segmenty pamięci wirtualnej



Segment **sterta** (*heap*):

- **sterta i stos** współdzieli dostępny (na rozrost) obszar pamięci ze sobą,
- najczęściej adresy **stosu** i **sterty** „rosną” w przeciwnych kierunkach. Stos „rośnie” w kierunku niższych adresów, a sterta w kierunku wyższych adresów, więc istnieje możliwość nałożenia się przestrzeni adresowych obu segmentów.

Segmenty pamięci wirtualnej

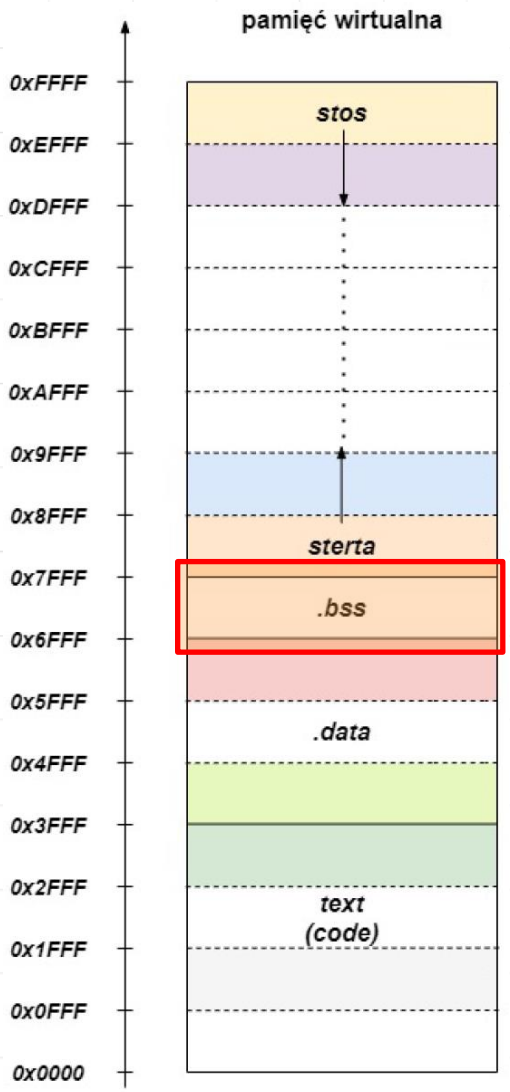


Segment **.bss** (*block started by symbol*):

- przechowuje niezainicjalizowane zmienne **globalne** i **statyczne**,
- zmienne **globalne** i **statyczne** jawnie zainicjalizowane wartością 0.

W języku C/C++ **globalne** i **statyczne** zmienne są automatycznie inicjalizowane wartością 0 przed wywołaniem funkcji main().

Segmenty pamięci wirtualnej

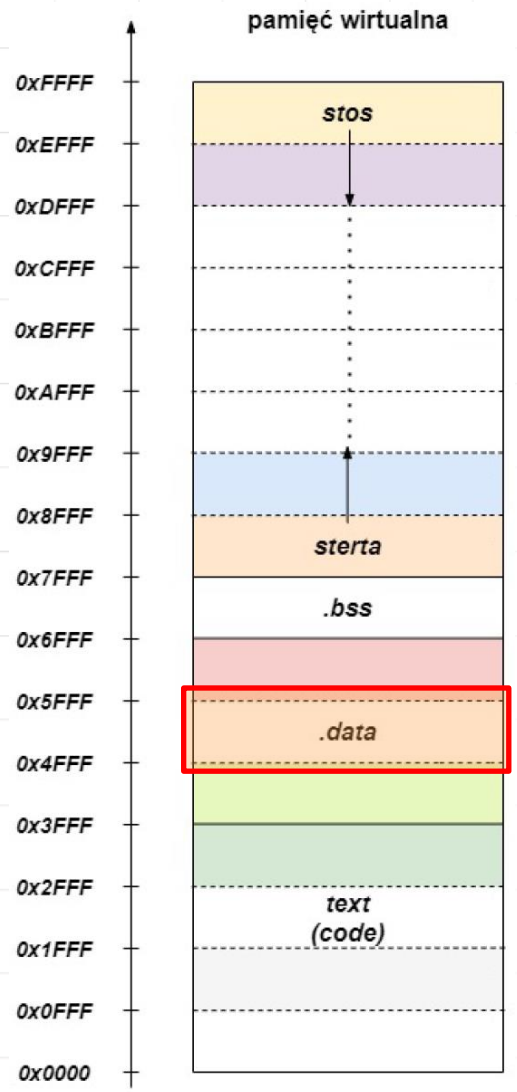


Segment **.bss** (*segment danych niezainicjalizowanych*):

- zmienne alokowane w segmencie **.bss**:

```
// Niezainicjalizowana zmienna globalna
float x;
// Zmienna globalna zainicjalizowana do 0
int y = 0;
// Niezainicjalizowana statyczna zmienna globalna
static short z;
// Statyczna zmienna globalna zainicjalizowana do 0
static unsigned int w = 0;
// Zmienne umieszczone w anonimowej przestrzeni nazw
    sa rownowazne statycznym zmiennym globalnym
namespace {
    // Zmienna niezainicjalizowana
    double a;
    // Zmienna zainicjalizowana do 0
    long b = 0;
}

int main() {
    // Niezainicjalizowana statyczna zmienna lokalna
    static long long c;
    // Statyczna zmienna lokalna zainicjalizowana do
    0
    static unsigned short d = 0;
    return 0;
}
```

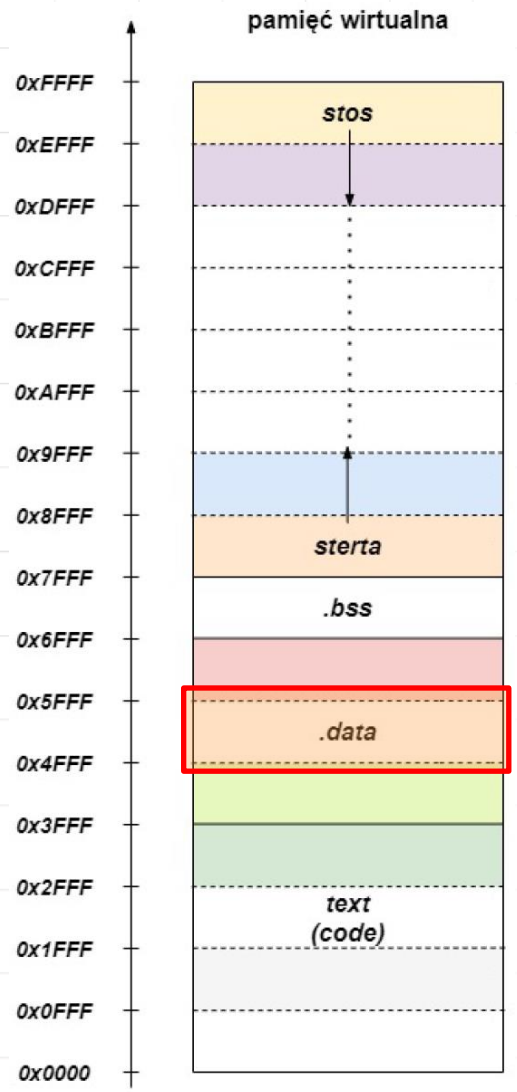
Segmenty pamięci wirtualnej



Segment **.data** (*segment danych zainicjalizowanych*):

- przechowuje globalne i statyczne zmienne jawnie zainicjalizowane przez programistę (wartością inną od 0).
- W segmencie **.data** można wydzielić obszar pamięci, w którym alokowane są zmienne tylko do odczytu – **.rodata** (*read-only data*), gdzie alokowane są stałe.
- Rozmiar segmentu **.data** nie zmienia się w trakcie działania programu.

Segmenty pamięci wirtualnej

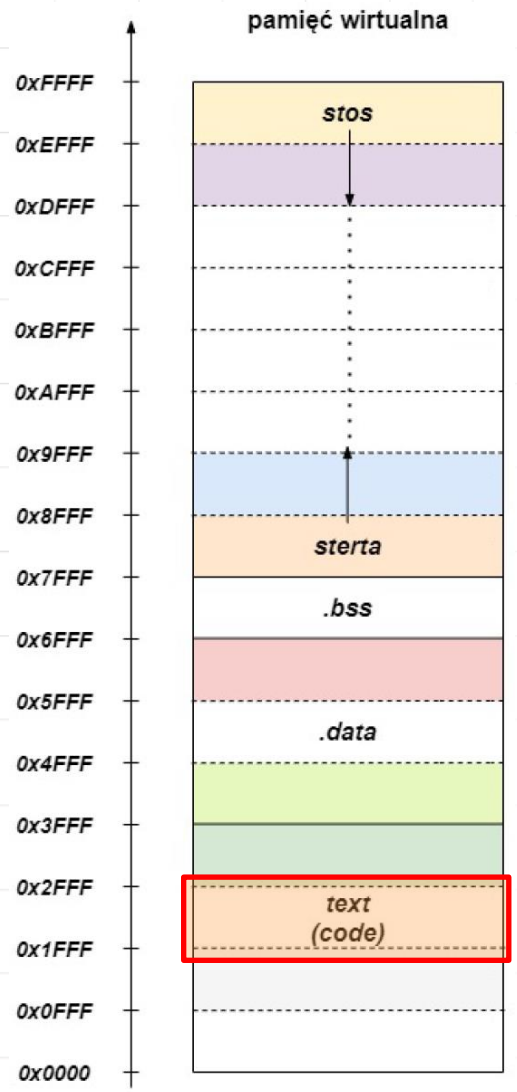


Segment **.data** (*segment danych zainicjalizowanych*):

- zmienne alokowane w segmencie **.data**:

```
// Zainicjalizowana zmienna globalna
float x = 3.5;
// Zainicjalizowane statyczne zmienne globalne
static short y = 100;
namespace {
    double z = 5.5;
}
// Globalna stala (.rodata)
const int a = 111;
// Statyczna globalna stala (.rodata)
static const float b = 4.0;

int main() {
    // Zainicjalizowana statyczna zmienna lokalna
    static long w = 666;
    // Lokalna stala (.rodata)
    const unsigned short c = 1;
    return 0;
}
```

Segmenty pamięci wirtualnej



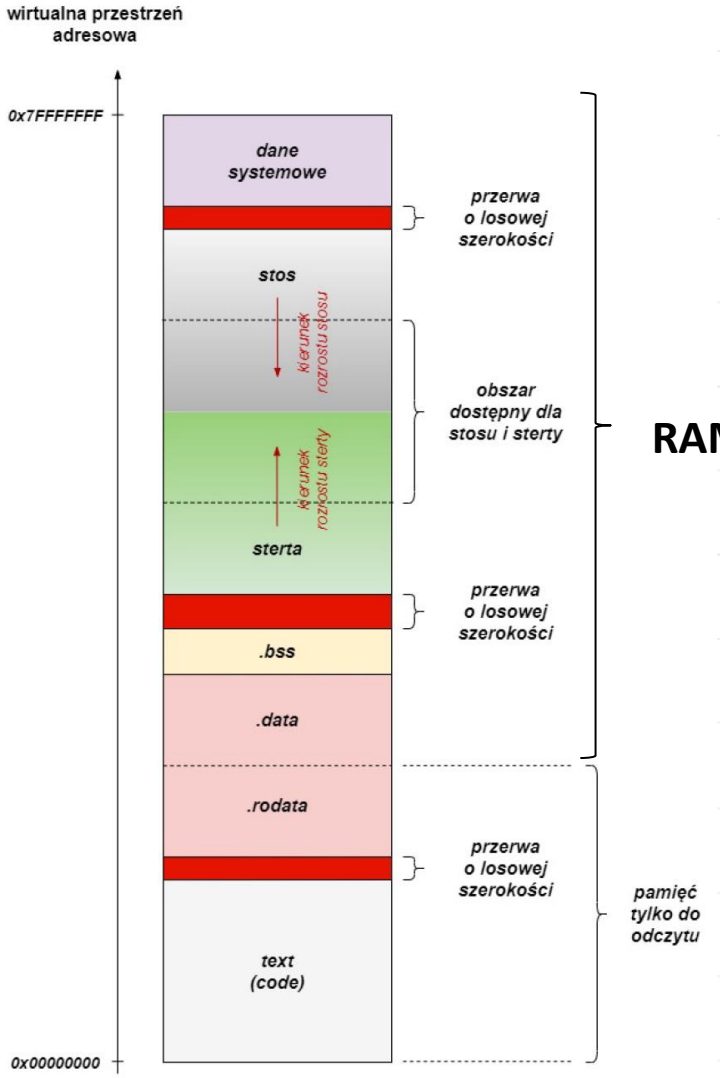
Segment **text** (**code**):

- służy do przechowania skompilowanego kodu maszynowego programu,
- ma stały rozmiar,
- w większości przypadków segment **text** jest obszarem pamięci tylko do odczytu, aby zapobiec przed nieintencjonalna modyfikacja jego zawartości w trakcie działania programu.
- W segmencie **text** położone są również adresy funkcji zdeklarowanych w kodzie programu.

Segmenty pamięci wirtualnej

- Segmenty pamięci są dodatkowo rozdzielone przerwami o losowej szerokości. Ma to na celu utrudnienie manipulacji z zewnątrz w wykonywanym kodzie, dzięki wprowadzeniu elementu losowości w rozmieszczeniu segmentów pamięci.

Typowy układ pamięci:



Dynamiczna alokacja pamięci -

- inaczej przydział pamięci podczas działania programu. Polega na alokowaniu pamięci na **stercie,** a rozmiar przydzielanej pamięci nie musi być znany czasie kompilacji programu.

Zastosowanie dynamicznej alokacji pamięci:

- gdy rozmiar wymaganej pamięci nie jest znany w czasie kompilacji, np. ilość elementów tablicy jest podawana przez użytkownika,
- gdy struktury danych takiej jak tablice mogą zwiększać swój rozmiar w trakcie działania programu (optymalizacja),
- w celu implementacji złożonych struktury danych takich, jak np. listy
- w celu wydłużenia czasu życia zmiennych z bloku funkcji(dla zmiennych lokalnych), ale nie na czas działania programu (zmienne globalne czy statyczne), np. gdy zmienna zdefiniowana wewnątrz funkcji ma być dostępna dla innych funkcji.

Dynamiczna alokacja pamięci – język C, <stdlib.h>

W języku C realizacja przydziału pamięci podczas działania programu odbywa się za pomocą trzech funkcji zdeklarowanych w **<stdlib.h>**:

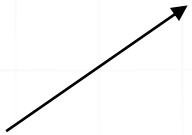
malloc(), calloc(), realloc()

Zwolnienie pamięci przydzielonej tymi funkcjami może się odbyć wyłącznie za pomocą funkcji **free().**

Dynamiczna alokacja pamięci – język C, <stdlib.h>

Nagłówek funkcji:

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



Typ zwracany - wskaźnik na początek przydzielonej pamięci. Typ `void *` może przechowywać adres zmiennej dowolnego typu i może być rzutowana na konkretny typ wskaźnikowy.

Wielkość alokowanej pamięci w bajtach. Typ `size_t` (alias dla `unsigned int`) - stosowany do reprezentowania rozmiarów i liczb.

Funkcja **`malloc()`** rezerwuje na **stercie** niezainicjalizowaną pamięć o rozmiarze **`size`**. W przypadku poprawnego wywołania zwracany jest **wskaźnik** na początek zaalokowanego bloku pamięci. W innym wypadku zwracana jest wskaźnik zerowy - `NULL`.

Dynamiczna alokacja pamięci – język C, <stdlib.h>

Nagłówek funkcji:

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

wskaźnik na początek przydzielonej pamięci

Ilość **num** elementów o rozmiarze **size** w bajtach .

Funkcja **calloc()** rezerwuje na **stercie** pamięć, oraz inicjalizuje zerami dla **num** elementów o rozmiarze **size**. W przypadku poprawnego wywołania zwracany jest **wskaźnik** na początek zaalokowanego bloku pamięci. W innym wypadku zwracana jest wskaźnik zerowy - NULL.

Dynamiczna alokacja pamięci – język C, <stdlib.h>

Nagłówek funkcji:

```
void * realloc(void * ptr , size_t size)
```

wskaźnik na początek przydzielonej pamięci

Wskaźnik na wcześniej przydzieloną pamięć za pomocą funkcji `malloc()`, `calloc()`, `realloc()`, jeżeli wartość wskaźnika `NULL` - to alokuje pamięć o rozmiarze `size` (jak `malloc()`)

nowy rozmiar przydzielanej pamięci

Funkcja **`realloc`**`()` alokuje segment pamięci o rozmiarze `size` w obszarze wskazywanym przez wskaźnik `ptr`, który został wcześniej zaalokowany i nie został zwolniony z użyciem funkcji `free()`. W ten sposób możliwe jest rozszerzanie bądź zmniejszanie zaalokowanego obszaru pamięci (optymalizacja). Należy mieć na uwadze, ze funkcja `realloc()`, w pierwszej kolejności zwalnia pamięć wskazywaną przez wskaźnik `ptr`.

Dynamiczna alokacja pamięci – język C, <stdlib.h>

Nagłówek funkcji:

```
void free(void * ptr)
```

Wskaźnik na wcześniej przydzieloną pamięć za pomocą jednej z funkcji `malloc()`, `calloc()`, `realloc()`.

Funkcja **`free`**`()` zwalnia pamięć zaalokowana za pomocą jednej z funkcji `malloc`(), `calloc`() albo `realloc`(), którą wskazuje `ptr` (jeśli `ptr` jest wskaźnikiem na pamięć zaalokowaną w inny sposób, to wynik działania funkcji jest niezdefiniowany. Jeżeli `ptr` ma wartość `NULL` to funkcja nie robi nic.

Funkcja `free()` nie zmienia wartości wskaźnika (nie ustawia wartości na `NULL`) ani nie zeruje zalakowanej wcześniej pamięci !!!

Dynamiczna alokacja pamięci – język C, <stdlib.h>

```c
#include <stdio.h>
#include <stdlib.h> /* realloc, calloc, malloc, free, exit, NULL */

int main ()
{
    int liczba=0, l = 1;    //liczba i l-licznik
    int * tabInt = NULL;
    int * aktTabInt = NULL;

    /*
    printf ("Wprowadz rozmiar tablicy dynamicznej");
    scanf ("%d", &liczba);
    tabInt = (int*) malloc(liczba*sizeof(int));
    tabInt = (int*) calloc(liczba, sizeof(int));
    */

    do{
        printf ("Wprowadz liczbe calkowita (666 - stop): ");
        scanf ("%d", &liczba);

        if (liczba == 666)
            break;
        aktTabInt = (int*) realloc(tabInt, l*sizeof(int));

        if(aktTabInt!=NULL){
            tabInt=aktTabInt;
            tabInt[l-1]=liczba;
        }
        else{
            free(tabInt);
            printf("Blad utworzenia lub optymalizacji tablicy");
            exit(EXIT_FAILURE);
        }
        ++l;
    }while (1);

    printf ("Wprowadzone liczby to: ");
    for (int i=0;i<l-1;++i)
        printf ("%d, ",aktTabInt[i]);

    free(aktTabInt);
    return 0;
}
```

```
Wprowadz liczbe calkowita (666 - stop): 1
Wprowadz liczbe calkowita (666 - stop): 2
Wprowadz liczbe calkowita (666 - stop): 3
Wprowadz liczbe calkowita (666 - stop): 4
Wprowadz liczbe calkowita (666 - stop): 5
Wprowadz liczbe calkowita (666 - stop): 6
Wprowadz liczbe calkowita (666 - stop): 666
Wprowadzone liczby to: 1, 2, 3, 4, 5, 6,
```

Dynamiczna alokacja pamięci – język C, <stdlib.h>

```c
#include <stdio.h>
#include <stdlib.h> /* realloc, calloc, malloc, free, exit, NULL */

int main ()
{
    int liczba=0, l = 1;    //liczba i l-licznik
    int * tabInt = NULL;
    int * aktTabInt = NULL;

    /*
    printf ("Wprowadz rozmiar tablicy dynamicznej");
    scanf ("%d", &liczba);
    tabInt = (int*) malloc(liczba*sizeof(int));
    tabInt = (int*) calloc(liczba, sizeof(int));
    */

    do{
        printf ("Wprowadz liczbe calkowita (666 - stop): ");
        scanf ("%d", &liczba);

        if (liczba == 666)
            break;
        aktTabInt = (int*) realloc(tabInt, l*sizeof(int));

        if(aktTabInt!=NULL){
            tabInt=aktTabInt;
            tabInt[l-1]=liczba;
        }
        else{
            free(tabInt);
            printf("Blad utworzenia lub optymalizacji tablicy");
            exit(EXIT_FAILURE);
        }
        ++l;
    }while (1);

    printf ("Wprowadzone liczby to: ");
    for (int i=0;i<l-1;++i)
        printf ("%d, ",aktTabInt[i]);

    free(aktTabInt);
    return 0;
}
```

W języku C/C++ nie ma możliwości sprawdzenia rozmiaru dynamicznej tablicy za pomocą operatora **`sizeof`**. ☹

Wartość tą musimy zapamiętać oraz ewentualnie przekazać.

```
Wprowadz liczbe calkowita (666 - stop): 1
Wprowadz liczbe calkowita (666 - stop): 2
Wprowadz liczbe calkowita (666 - stop): 3
Wprowadz liczbe calkowita (666 - stop): 4
Wprowadz liczbe calkowita (666 - stop): 5
Wprowadz liczbe calkowita (666 - stop): 6
Wprowadz liczbe calkowita (666 - stop): 666
Wprowadzone liczby to: 1, 2, 3, 4, 5, 6,
```

Dynamiczna alokacja pamięci – język C++ <iostrem>

Nagłówek funkcji:

```
void * operator new(std::size_t size);
void operator delete(void * ptr);
```

Domyślne operatory są specjalnymi składnikami biblioteki standardowej, a ich deklaracja znajduje się w globalnej przestrzeni nazw. Można dołączyć nagłówek <new> lecz nie jest to konieczne.

Operator **new** alokuje dynamicznie pamięć na **stercie** dla zmiennej/obiektu o rozmiarze **size** bajtów. Jednak rozmiar nie jest to podany jawnie tylko po przez wskazanie typu. W przypadku powodzenia zwracany jest wskaźnik na początek zaalokowanego bloku pamięci. W innym przypadku wyrzucany jest wyjątek `bad_alloc` (przy odpowiedniej konstrukcji `new(std :: nothrow)` zwracany jest `nullptr)`.
Operator **delete** zwalnia pamięć wskazywaną przez wskaźnik **ptr**, która została przydzielona za pomocą operatora **new**. Jeżeli **ptr** ma wartosc `nullptr` , to operator **delete** nie robi nic, w każdej innej sytuacji zachowanie jest niezdefiniowane.

Dynamiczna alokacja pamięci – język C++ <iostrem>

Nagłówek funkcji:

```
void * operator new[](std::size_t size);
void operator delete[](void * ptr);
```

Oba poprzednie operatory występują również w wersji do dynamicznej alokacji pamięci dla tablic obiektów, `new[], delete[]`. Ważne aby operatorem `delete[]` zwalniać pamięć przydzieloną za pomocą operatora `new[].`

Operatory `new, new[], delete, delete[]` mogą być przeciążane (przeciążanie/ przeładowanie operatorów pozwala na nadanie im nowych funkcji)

Dynamiczna alokacja pamięci – język C++ <iostrem>

```cpp
#include <iostream>

int main(){

    unsigned int tabSize;
    std::cout << "Podaj rozmiar tablicy tab[n][n] = ";
    std::cin >> tabSize;

    // Deklaracja dynamicznej tablicy dwuwymiarowej
    int ** tab = new int * [tabSize];
    for (unsigned int i = 0; i<tabSize; ++i)
        tab[i] = new int [tabSize];

    // Inicjalizacja tablicy
    std::cout << "Wprowadz kolejne elementy tablicy tab[i][j] rozdzielone spacja:";
    std::cout << std::endl;

    for ( unsigned int i = 0; i < tabSize; ++i){
        for ( unsigned int j = 0; j < tabSize; ++j){
            std :: cin >> tab[i][j];
        }
    }

    // Wyswietlenie zawartosci tablicy
    std::cout << "Wprowadzone wartosci do tab[i][j] to:" << std::endl;
    for ( unsigned int i = 0; i < tabSize; ++i){
        for ( unsigned int j = 0; j < tabSize; ++j){
            std :: cout << tab[i][j] << "   ";
        }
        std::cout << std::endl;
    }

    // Zwolnienie pamieci
    for (int i = 0; i<tabSize; i++)
        delete [] tab[i];

    delete [] tab ;
}
```

```
Podaj rozmiar tablicy tab[n][n] = 3
Wprowadz kolejne elementy tablicy tab[i][j] rozdzielone spacja:
1 2 3 4 5 6 7 8 9
Wprowadzone wartosci do tab[i][j] to:
1   2   3
4   5   6
7   8   9
```