



Uniwersytet
Wrocławski

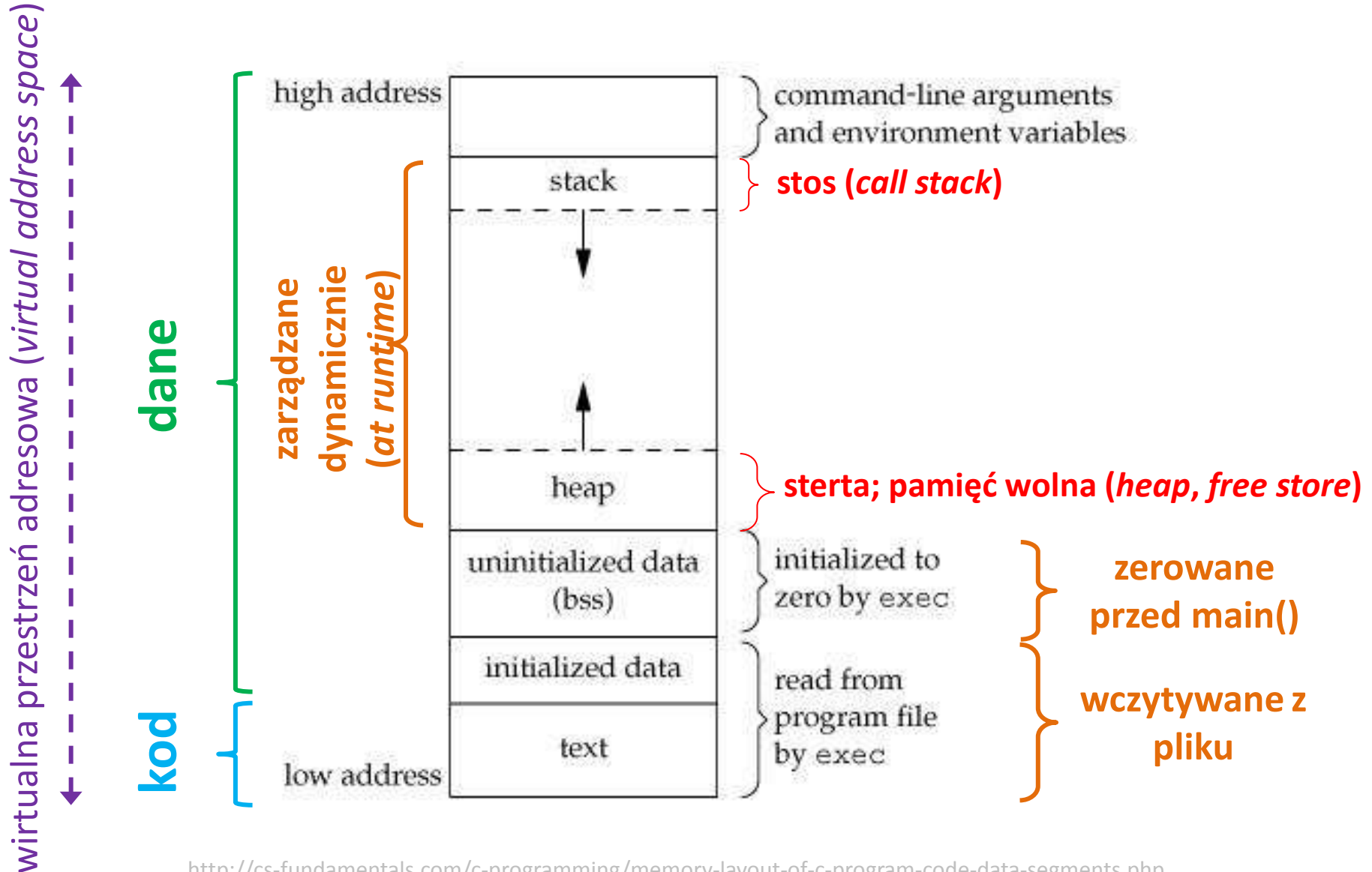
C++: *model pamięci*

Zbigniew Koza
Wydział Fizyki i Astronomii

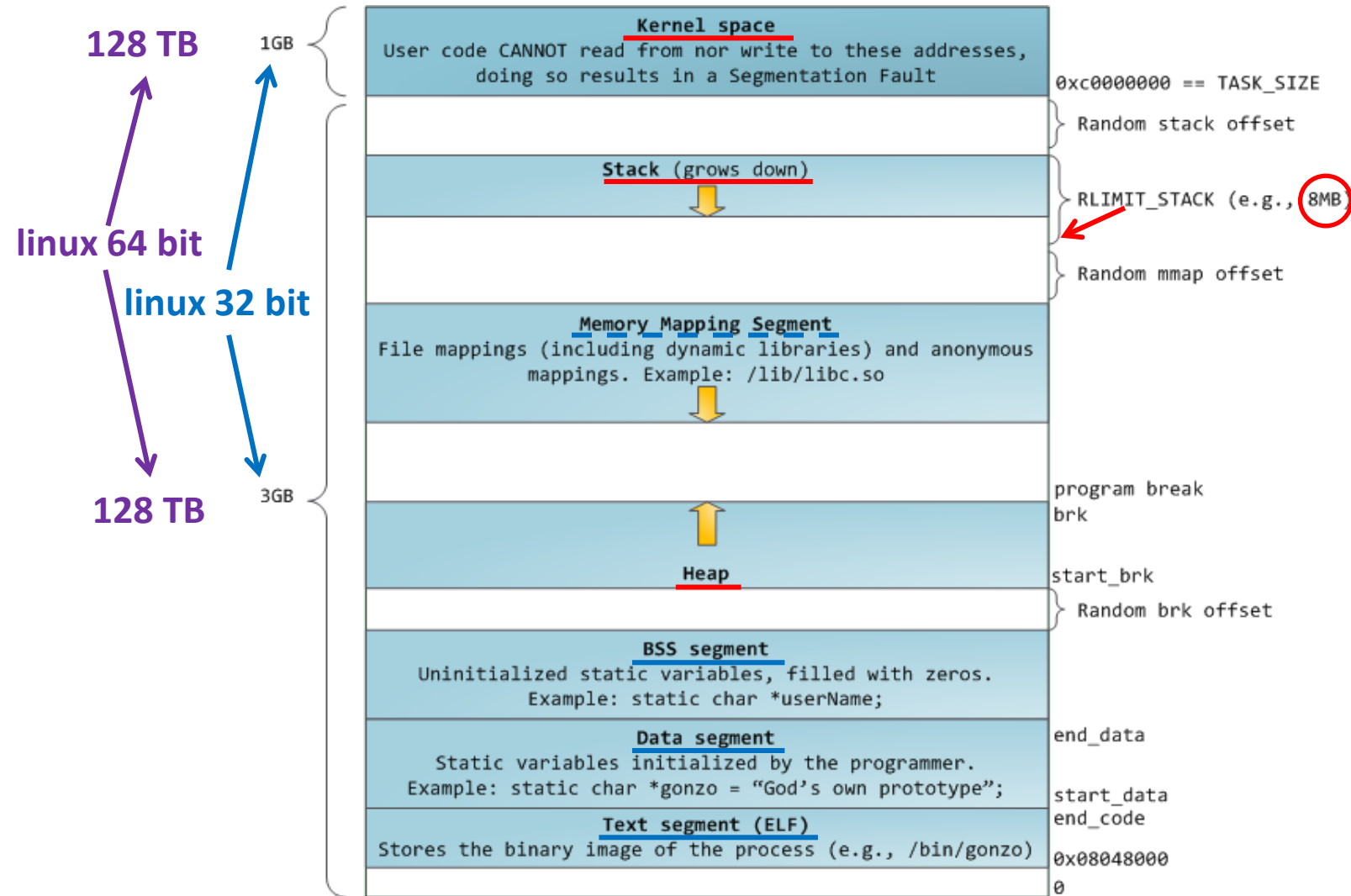
Wrocław

MODEL PAMIĘCI

Model pamięci w C/C++



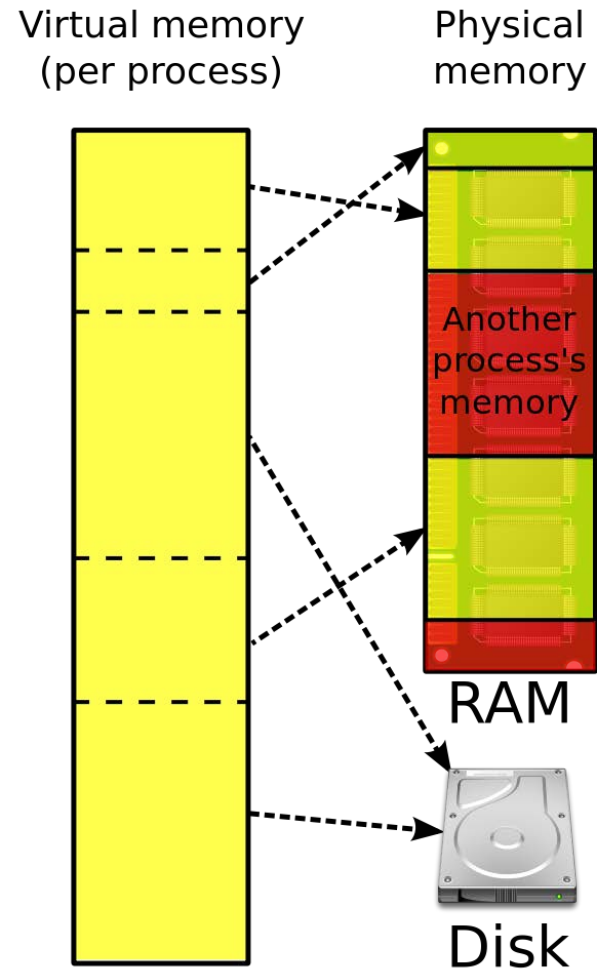
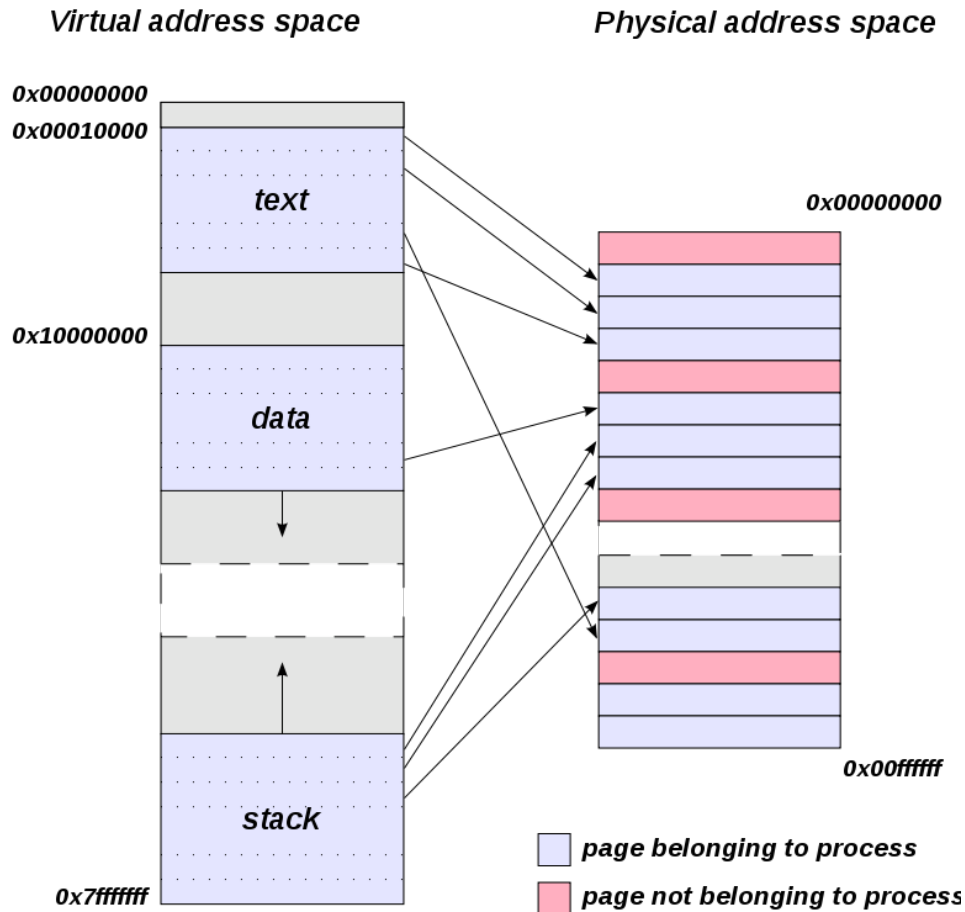
Model pamięci w C/C++ (Linux)



Wirtualna przestrzeń adresowa

- W wielozadaniowych systemach operacyjnych każdy **proces** "widzi" identyczną, **wirtualną przestrzeń adresową**
 - 2^{32} B = 4 GB – systemy 32-bitowe
 - 2^{43} - 2^{48} B = 8-256 TB – systemy 64-bitowe
- Pozwala to:
 - Całkowicie **odseparować od siebie różne procesy** (których mogą być setki).
 - **Odseparować program od sprzętu**

Wirtualna a fizyczna przestrzeń adresowa





Wirtualna a fizyczna przestrzeń adresowa

- Adresy z wirtualnej przestrzeni adresowej są tłumaczone na adresy w przestrzeni fizycznej sprzętowo (CPU)
 - wsparcie systemu operacyjnego pozwala rozszerzyć pamięć fizyczną o pamięć na dysku (*swap*)
 - pamięć dzielona jest na **segmenty** (i/lub **strony**)
 - segmenty mają przypisaną długość i flagi (prawo odczytu, zapisu, wykonania, lokalizacja fizyczna), co zapewnia podstawową ochronę systemu

Segmentacja i *swap*...

- Już domyslamy się, dlaczego Windows potrzebuje dużo miejsca na dysku C:

 hiberfil.sys	5,95 GB	2016-04-14 18:27:20	Plik systemowy	AHS
 <u>pagefile.sys</u>	<u>7,94 GB</u>	2016-04-14 18:27:20	Plik systemowy	AHS

- (Linux używa osobnej partycji):

Number	Start	End	Size	File system	Name	Flags
1	17,4kB	50,0MB	50,0MB			bios_grub
4	50,0MB	600MB	550MB	ext4		msftdata
2	600MB	10,6GB	10,0GB	<u>linux-swap (v1)</u>		
3	10,6GB	120GB	109GB	ext4		msftdata

- i skąd się biorą takie komunikaty:

```
zkoza@zbyszek:~/tmp$ ./a.out  
Segmentation fault (core dumped)
```


Kernel/user space

- Wirtualna przestrzeń adresowa podzielona jest na dwie części:
 - pamięć jądra (górna połowa adresów)
 - pamięć użytkownika (dolna połowa adresów)
- Tylko funkcje systemowe mają dostęp do pamięci jądra (\Rightarrow bezpieczeństwo systemu i innych użytkowników; por. hermetyzacja danych w klasach)
 - dlatego pamięć jądra zostawiamy na boku...

.text

- Sekcja `.text` zawiera **kod programu** (oznaczony flagami "wykonywalny", "tylko do odczytu")
- może zawierać **wartości stałych**, zwłaszcza napisów ("tylko do odczytu")

```
int main()  
{  
    char* a = "Ala ma kota!";  
    *a = 'O'; // Ola ma kota??  
}
```

Stały napis (const char*)

Ojej!

```
zkoza@zbyszek:~/tmp$ ./a.out  
Segmentation fault (core dumped)
```

przykład

```
zkoza@zbyszek:~/tmp$ cat dump.cpp
```

```
int main()
{
    char * a = "Ala ma kota!";
    *a = '0';
    return *a;
}
```

tekst programu **1.**

```
zkoza@zbyszek:~/tmp$ g++ dump.cpp -O2 -std=c++11
```

```
dump.cpp: In function 'int main()':
```

```
dump.cpp:3:15: warning: deprecated conversion from string  
constant to 'char*' [-Wwrite-strings]
```

```
    char * a = "Ala ma kota!";
```

^

nigdy nie lekceważ ostrzeżeń kompilatora! **2.**

```
zkoza@zbyszek:~/tmp$ grep "Ala ma kota!" a.out
```

```
Binary file a.out matches
```

kod binarny zawiera napis! **3.**

```
zkoza@zbyszek:~/tmp$ ./a.out
```

```
Segmentation fault (core dumped)
```

ojej! **4.**

.data + .bss

- Sekcje `.data` i `.bss` zawierają zmienne **alokowane statycznie**, czyli przez kompilator

```
int N = 100; // zainicjalizowana zmienna globalna (.data)
int main()
{
    static int m = X::x;
    static double q; // domyślnie inicjalizowana na 0
}
```

`.data` ← `static int m = X::x;`
składowa statyczna klasy X;
nie wiemy czy
zainicjalizowana i/lub stała

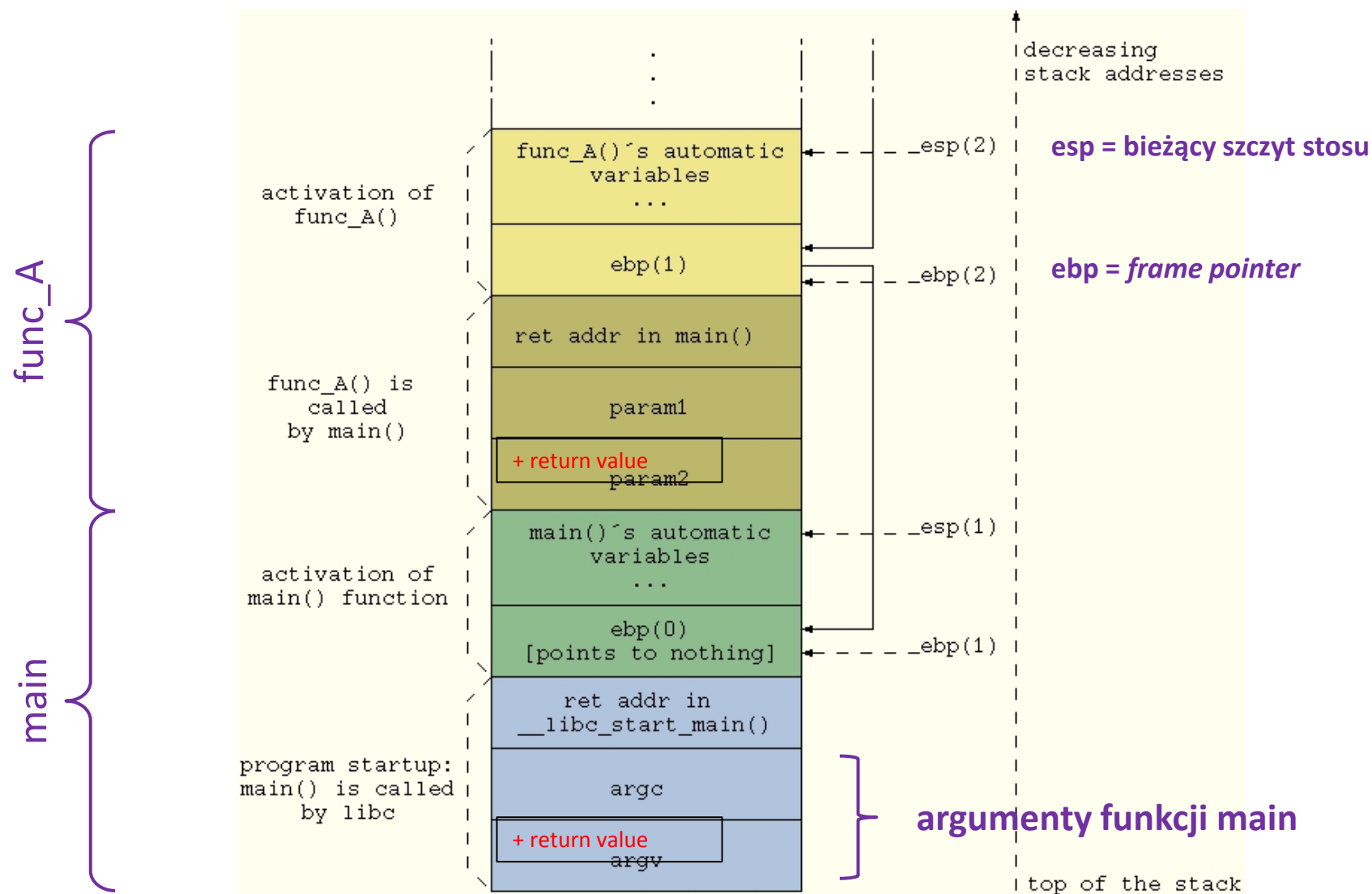
`.bss` ← `static double q;`
domyślnie inicjalizowana na 0

- `.data`: zmienne o znanych wartościach początkowych (tu: `N` i `m`)
- `.bss`: zmienne niezainicjalizowane; program uruchomieniowy wypełnia je zerami (tu: `q`)

.data + .bss

- Wniosek: niezainicjalizowane zmienne i obiekty globalne, a także składowe statyczne klas oraz zmienne statyczne w funkcjach są automatycznie inicjalizowane zerami
- Inicjalizacja takich obiektów wykonywana jest raz (inicjalizator może być funkcją)
- Czas życia tych danych = czas życia procesu

Stos funkcji (*call stack*)



Stos funkcji...

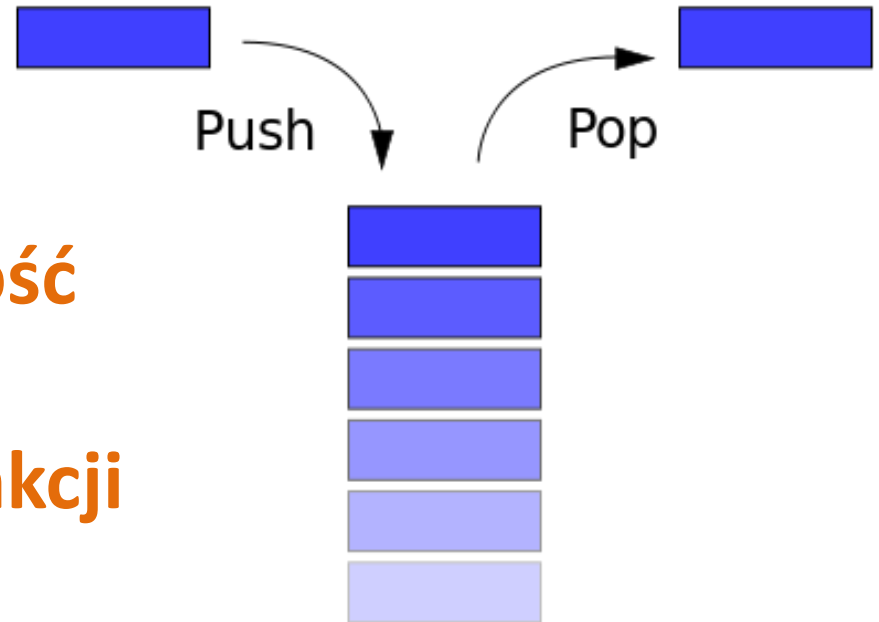
- Miejsce na:
 - **adres powrotu z funkcji**
 - wartość funkcji
 - argumenty funkcji
 - zmienne lokalne funkcji
 - zmienne tymczasowe (potrzebne podczas opracowywania niektórych wyrażeń)
 - wartość pseudoskażnika **this**
 - inne parametry związane z wywołaniem funkcji (stan rejestrów, informacje niezbędne do poprawnej obsługi wyjątków, etc.)

Stos funkcji

- Zalety:
 - Prostota, efektywna implementacja
 - Szybki dostęp do danych
 - Umożliwia stosowanie rekurencji
 - **Pełna automatyzacja** obsługi pamięci przez kompilator
 - Ułatwia inspekcję stanu programu:
podczas działania debuggera i *post mortem*

Zasada działania stosu

- LIFO: *Last in, First Out*
- Konsekwencje: **kolejność destrukcji obiektów automatycznych w funkcji jest odwrotna do kolejności ich konstrukcji** (fundamentalna cecha C++)



Stos funkcji a zakresy

- **Zakres** (ang.: *scope*): każdy fragment kodu ujęty w nawiasy klamrowe, **{ }**

```
...  
if (...) {  
    std::vector<int> v(100);  
}  
...
```

początek zakresu

zmienna lokalna w zakresie

koniec zakresu: destruktor v

- Wyjście programu z zakresu \Rightarrow destrukcja obiektów zdefiniowanych w tym zakresie (w odwrotnej kolejności do ich konstrukcji)

Destrukcja obiektów tymczasowych

- Jeżeli w jakiejś instrukcji , np.

`f(std::string("Ala")) ;`



obiekt tymczasowy

koniec zakresu:
destruktor `std::string("Ala")`



występuje obiekt tymczasowy,
to jego destruktor wykonywany jest przed
rozpoczęciem kolejnej instrukcji

Rozmiar stosu

- Rozmiar stosu jest ograniczony
 - Linux64: 8MB (obecnie)

```
zkoza@zbyszek:~$ ulimit -s  
8192
```

- W aplikacjach wielowątkowych każdy wątek ma własny stos

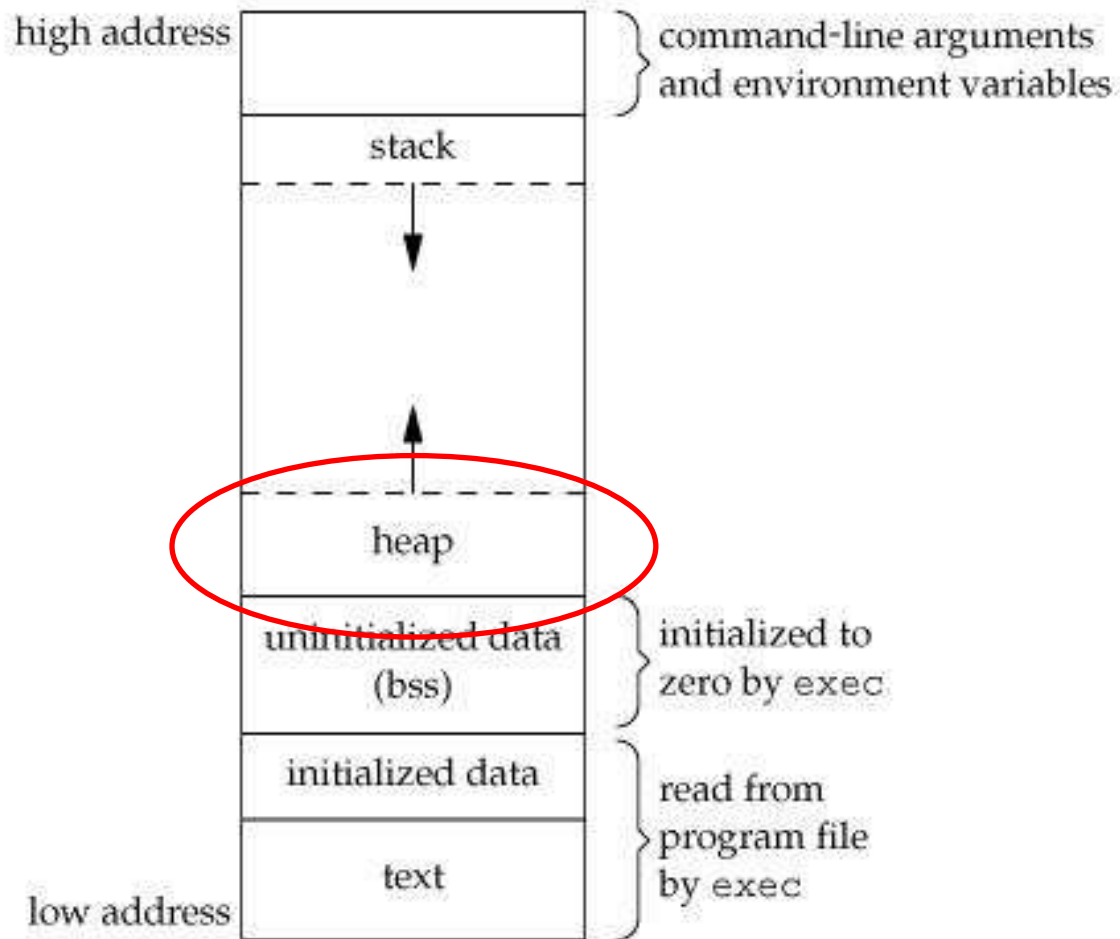
System	
Dojścia	30467
Wątki	<u>1253</u>
Procesy	91
Czas pracy	1:01:15:55
Zadeklarowane (GB)	3 / 15

Pamięć ...	Wątki	Opis
152 K	126	NT Kernel & System
10 124 K	113	AVG Resident Shield Service
770 288 K	69	Firefox
234 096 K	52	Thunderbird
5 780 K	49	Bluetooth Stack Server
32 636 K	43	AVG Scanning Core Module - Server Part
12 624 K	42	AVG Watchdog Service
26 336 K	36	Proces hosta dla usług systemu Windows
10 632 K	34	AVG Online Shield Service
24 108 K	32	Eksplorator Windows

Rozmiar stosu jest ograniczony...

- Nie deklaruuj wielkich tablic jako zmiennych lokalnych
- Pilnuj, aby stopień zagnieżdżenia funkcji rekurencyjnych nie był zbyt duży

Sterta (*free store; heap*)



Sterna (*free store; heap*)

- Niemal cała pozostała pamięć wirtualna zarezerwowana jest na **stertę**
- **Sterna** to fragment pamięci operacyjnej zarządzany ("ręcznie"/"dynamicznie") przez program/programistę
- Czas życia obiektów na sterce nie jest ograniczony przez zakres, w którym zostały utworzone
- Pamięć dynamiczna/na żądanie

Środki dostępne w języku C++

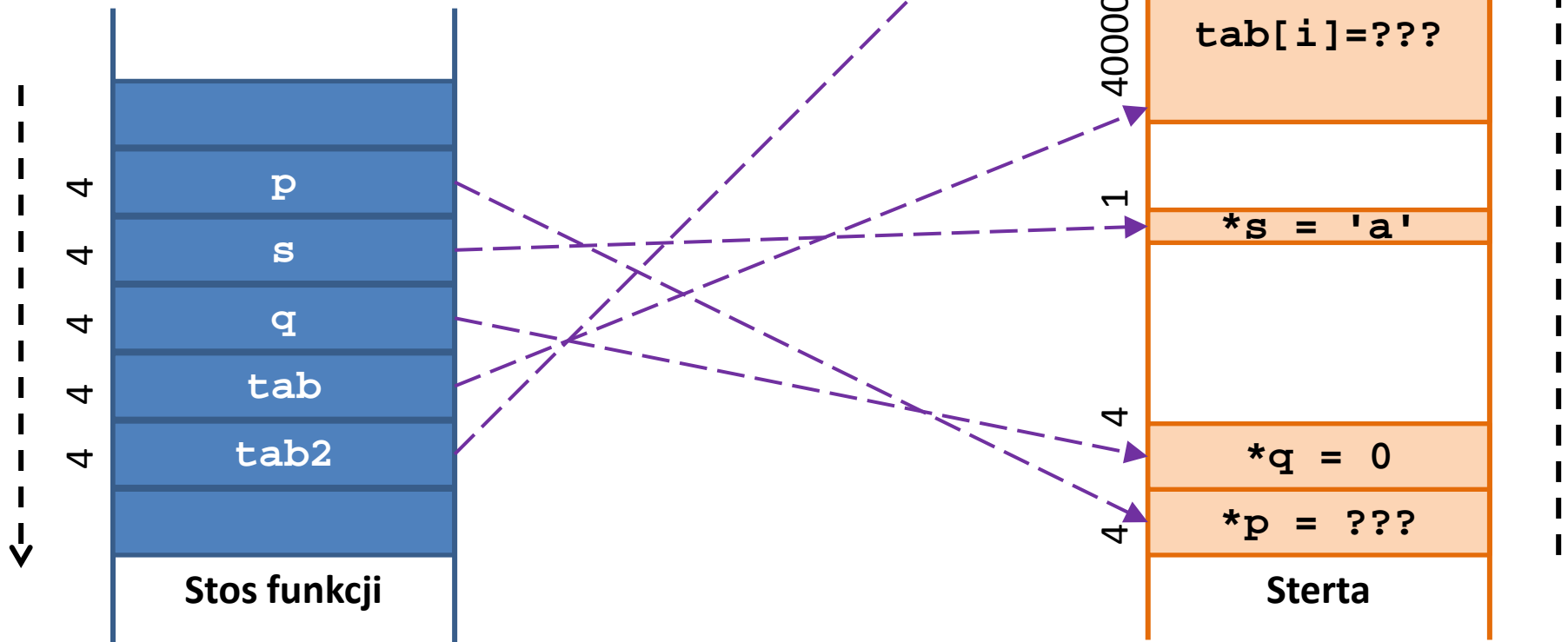
<i>przydział pamięci</i>	<i>zwolnienie pamięci</i>	<i>uwagi</i>
Język C		
malloc	free	
calloc	free	przydzielona pamięć jest zerowana
Język C++		
new TYP	delete ADRES	konstruktor/destruktor
new TYP [LICZBA]	delete [] ADRES	konstruktory/destruktory

```
int* p = new int;  
double* tab = new double[100000];  
*p = 9;  
tab[16] = 1.0/3;  
...  
delete p;  
delete[] tab;
```

} zwykle w innej funkcji niż new

Obiekt na sterpie

```
int* p = new int; // *p = śmieć  
int* s = new char('a');  
int* q = new int();  
int* tab = new int[100000];  
double* tab2 = new double[3]();
```



Zalety sterty...

- Bezpośredni dostęp do praktycznie całej fizycznie dostępnej pamięci
- Pełna kontrola nad momentem zwolnienia pamięci
 - Dane umieszczone na sterckie mogą być dostępne dłużej niż dane umieszczone na stosie funkcji
 - Gwarancja wywołania destruktora
- Interfejs do komunikacji między wątkami procesu

Wady sterty...

- Brak automatycznej kontroli poprawności użycia zasobów sterty
- Znaczny narzut czasowy na alokację nowych bloków pamięci, zwłaszcza jeśli biblioteka obsługująca stertę musi skomunikować się z systemem operacyjnym
- Nagie wskaźniki (w C/C++)...

Stos a sarta

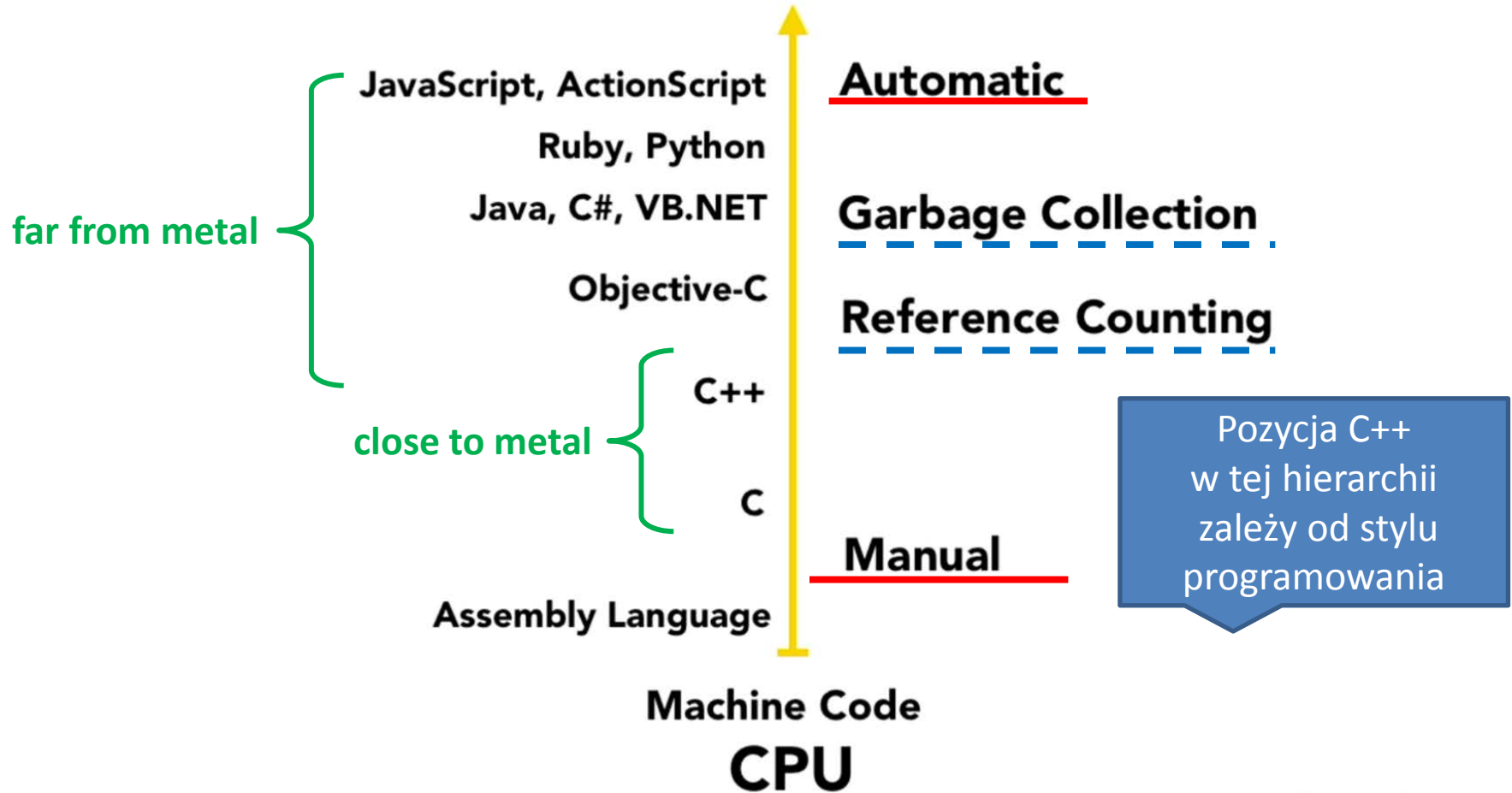
Stos

- Automatyczne zarządzanie (kompilator)
- Tylko zmienne lokalne
- Szybka alokacja zasobów
- Pamięć nie ulega fragmentacji
- Ograniczony rozmiar
- Oddzielny dla każdego wątku
- Umożliwia rekurencję

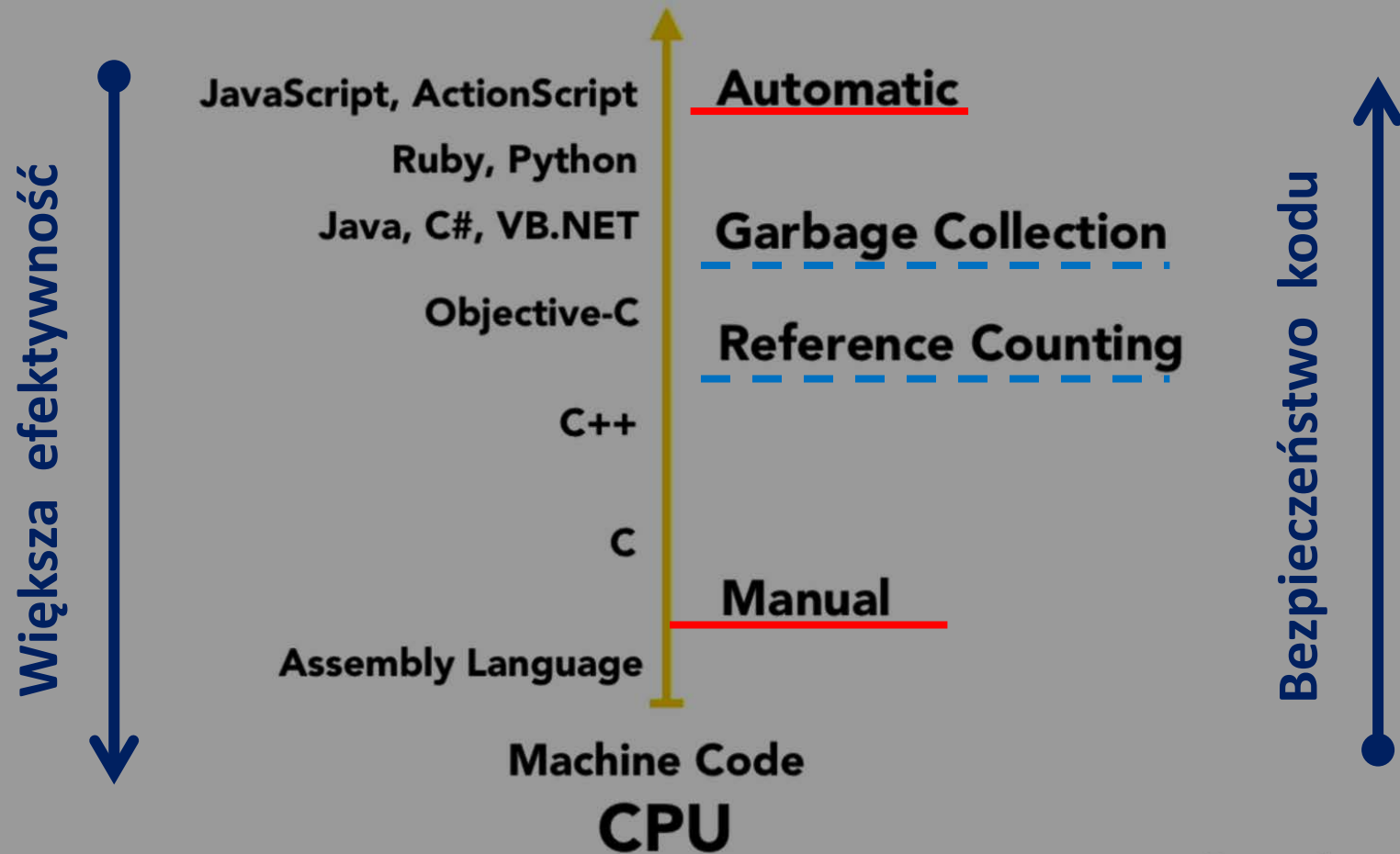
Sarta

- Ręczne zarządzanie
- Pamięć dostępna wszędzie i zawsze, wspólna dla wszystkich wątków
- Bez ograniczeń na rozmiar
- Zarządzanie (new/delete) jest relatywnie czasochłonne (dla CPU)
- Pamięć może ulec fragmentacji
- Wskaźniki... (C/C++)

Dlaczego sposób obsługi pamięci jest tak ważny?



Dlaczego sposób obsługi pamięci jest tak ważny?



Dlaczego sposób obsługi pamięci jest tak ważny?

- Bo dla wielu firm sposób (koszt) rozwiązania tego problemu może być decydującym czynnikiem, determinującym wybór użytej technologii
- Brak automatycznej obsługi pamięci jest jednym z głównych argumentów krytyków C++

Co na to C++?

- Jeśli naprawdę nie musisz, nie używaj **new/delete** (!!!) – czyli unikaj ręcznego alokowania obiektów na stercie
- Jeśli potrzebujesz dużych ilości pamięci, co wymaga użycia sterty, używaj **BIBLIOTEK**, które zapewniają transparentną i bezpieczną obsługę alokacji i dealokacji pamięci

O jakie biblioteki chodzi?

Biblioteka standardowa, kontenery:

- `std::vector<T>`
- `std::array<T, N>`
- `std::deque<T>`
- `std::list<T>`
- `std::forward_list<T>`
- `std::queue<T>`
- `std::priority_queue<T>`
- `std::stack<T>`
- `std::bitset<N>`
- `std::map<KEY, T>`
- `std::set<KEY>`
- `std::multimap<KEY,T>`
- `std::multiset<KEY>`
- `std::unordered_map<KEY, T>`
- `std::unordered_set<KEY>`
- `std::unordered_multimap<KEY,T>`
- `std::unordered_multiset<KEY>`
- `std::string`

Inne biblioteki, np. Qt...

- QList<T>
- QLinkedList<T>
- QVector<T>
- QStack<T>
- QQueue<T>
- QSet<T>
- QMap<Key, T>
- QMultiMap<Key, T>
- QHash<Key, T>
- QMultiHash<Key, T>
- QString

Deklaruj swoje obiekty na stosie...

- Zamiast

```
int* p = new int[100000];  
// używaj p jak nawy tablicy int-ów  
delete [] p;
```

- Używaj na przykład

```
std::vector<int> p(100000);
```

lub

```
std::deque<int> p(100000);
```

Jeśli musisz jawnie używać new/delete...

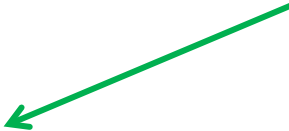
- Poczytaj o:

- `unique_ptr<T>`

np:

```
unique_ptr<int> p = new int{8};
```

wskaźnik
bezpiecznie
opakowany
w obiekt
na stosie



- `shared_ptr<T>`

- **RAII** (*Resource Aquisition is Initialisation*)

Nie tylko pamięć

- Podobnych technik używaj do zarządzania innymi **zasobami**, np.:

– Język C:

`FILE*` **f** = fopen("plik.txt", "r");



...

`fclose(f);`

– Język C++:

`std::ifstream` **f**("plik.txt");

