Preprocesor języka C (CPP)

- Przetwarza pliki zanim zobaczy je kompilator
- Linie zaczynające się od znaku #
- Preprocesor "nie zna" C
- Wyświetlenie efektów działania CPP: **`gcc -E my_file.c`**
- Funkcjonalność:
  - Dołączanie plików
  - Stałe
  - Makra
  - Kompilacja warunkowa

Włączanie plików

- CPP zastępuje **`#include`** zawartością włączanego pliku
- Pliki nagłówkowe
  - **`#include <stdio.h>`**
  - Poszukiwane w katalogach systemowych
  - W systemie UNIX, szuka w **`/usr/include`**
  - Nie należy używać **`<>`** do włączania plików nagłówkowych użytkownika
- Pliki nagłówkowe użytkownika
  - **`#include "my_pow.h"`**
  - Przeszukuje najpierw bieżący katalog, a następnie systemowy
- Dyrektywy **`#include`** można zagnieżdżać
- opcja **`-I`** służy do wyspecyfikowania dodatkowych katalogów, które kompilator ma przeszukiwać

Stałe

- Proste zastępowanie tekstu

```
#define  name                replacement text
```

Przykłady:

```
#define PI                   3.14
#define MAX                  1000
#define TRUE                 1
#define SECS_PER_DAY         (60 * 60 * 24)
#define READ_ACCESS          0x01
#define WRITE_ACCESS         0x02
#define RW_ACCESS            (READ_ACCESS | WRITE_ACCESS)
#undef PI /* un-defines a constant/macro */
```

Makrodefinicje

```
#define max(A, B)      ((A) > (B) ? (A) : (B))

x = max(p+q, r+s);
```

staje się

```
x = ((p+q) > (r+s) ? (p+q) : (r+s));
```

Uwaga, wyrażenia obliczane dwukrotnie!

- **`stdio.h`** definiuje m.in. następujące makra:

```
#define getchar()      getc(stdin)
#define putchar(c)     putc((c), stdout)
```

Nie umieszczaj spacji między nazwą makra i '('

Nie kończ średnikiem

Zamiana na łańcuch

- **`#parameter_name`** umieszcza **`parameter_name`** w cudzysłowiu

```
#define PRINT_STR_VAR(S) printf(#S " = %s\n", S)

char szDrink[] = "Okocim";
PRINT_STR_VAR(szDrink);
```

- staje się

```
printf("szDrink" " = %s\n", szDrink);
```

co następnie, po konkatenacji łańcuchów, zamienia się na:

```
printf("szDrink = %s\n", szDrink);
```

ostatecznie wynik wydruku to:

```
szDrink = Okocim
```

Zamiana na łańcuch

```
typedef enum { red, green, blue, white, black } EColor;

#define COLOR_CASE(x) case x: return "The color is " #x

const char* Color2Name(EColor clr) {
    switch (clr) {
        COLOR_CASE(red);
        COLOR_CASE(green);
        COLOR_CASE(blue);
        COLOR_CASE(white);
        COLOR_CASE(black);
        default: return "INVALID COLOR!";
    }
}
```

Zamiana na łańcuch

- Aby zamienić na łańcuch wynik rozwinięcia argumentu makrodefinicji, należy użyć dwóch poziomów makrodefinicji.

```
#define XSTR(S) STR(S)
#define STR(S) #S
#define FOO 4
STR (FOO)
   ==> "FOO"
XSTR (FOO)
   ==> XSTR (4)
   ==> STR (4)
   ==> "4"

#define FIELDSIZE 4
char* str="long string";
printf("%" XSTR(FIELDSIZE) "s",str);
```

Konkatenacja

- Aby skonkatenować dwa argumenty makrodefinicji, używamy operatora **##**:

```
#define DOUBLE(A,B) A##B
printf("%f\n",DOUBLE(3,e6));
   ==> printf("%f\n",3e6);

#define DOUBLE(A,B) AB
printf("%f\n",DOUBLE(3,e6));
   ==> printf("%f\n",AB); /* invalid */

#define DOUBLE(A,B) A B
printf("%f\n",DOUBLE(3,e6));
   ==> printf("%f\n",3 e6); /* invalid */
```

Predefiniowane makra

- **__LINE__** Bieżący numer linii w pliku źródłowym
- **__FILE__** Nazwa pliku właśnie kompilowanego
- Użyteczny do obsługi błędów, uruchamiania i logowania

```
#define DEBUG_STR(S)\
      printf(#S " = %s [File: %s, Line: %d]\n",\
      S, __FILE__, __LINE__);

DEBUG_STR(szDrink)
```

- wynik wydruku:

```
szDrink = Okocim [File: debug_str.c, Line: 16]
```

Kompilacja warunkowa

- **`#if, #ifdef, #ifndef, #else, #endif`**
- Przydatny dla oddzielenia kodu do celów uruchamiania programu od kodu w wersji produkcyjnej
- **`assert()`**, debug_alloc, itd.

```
#ifdef MY_DEBUG
#define DEBUG_STR(S)\
  printf(#S " = %s [File: %s, Line: %d]\n",\
  S, __FILE__, __LINE__);
#else
#define DEBUG_STR(S)
        /* do nothing in release builds */
#endif /* MY_DEBUG */
```

- Można stosować opcję kompilatora **`-Dmacro=value`**:
- **`$ gcc -DMY_DEBUG=1 debug_str.c -o debug_str && ./debug_str`**
- **`szDrink = Okocim [File: debug_str.c, Line: 16]`**

Makra dla celów przenośności programów

- Kod specyficzny dla kompilatora, procesora, systemu operacyjnego lub API

```
#ifdef _WIN32
OutputDebugString(szOutMessage);
#else  /* _WIN32 */
_CrtOutputDebugString(szOutMessage);
#endif  /* _WIN32 */
```

Prawie jak funkcja

- Aby zdefiniować makro, które może być użyte jak funkcja w większości kontekstów można użyć poniższej konstrukcji

```
#define fun(x) do { printf(#x "=%d\n",x); x++;} while (0)
```

Zapewni to działanie w poniższym przykładzie wymagającym średnika po wywołaniu makrodefinicji lub funkcji

```
if (condition)
  fun(t);
else
  fun(z);
```

- Prostsza wersja poniżej nie zadziała

```
#define fun(x) { printf(#x "=%d\n",x); x++;}
```

Problemy w dużych programach

- Małe programy $\rightarrow$ jeden plik
- “Nie tak małe" programy:
  - Wiele linii kodu
  - Wiele komponentów
  - Więcej niż jeden programista
- Problemy:
  - Trudniej zarządzać długimi plikami (zarówno programistom, jak i komputerom)
  - Każda zmiana wymaga długiej kompilacji
  - Wielu programistów nie może modyfikować jednocześnie tego samego pliku
  - Pożądany jest podział na komponenty

Wiele modułów

- Podzielenie projektu na wiele plików
  - Dobry podział na komponenty
  - Mniej kompilacji przy zmianach w programie
  - Łatwiejsze zarządzanie strukturą i zależnościami w projekcie
- Język C posiada konstrukcje służące do odwołania do innych plików (**`extern`**)
- Podczas kompilacji można skonsolidować (zlinkować) wiele plików w celu stworzenia jednego programu.
- Można kompilować wiele plików źródłowych, linkować wiele prekompilowanych plików pośrednich lub połączyć te opcje.

Wiele modułów

```
/* greetings.h */
void greetings(void);
```

```
/* main.c */
#include <stdio.h>
#include "greetings.h"

int main()
{
  greetings();
  return 0;
}
```

```
/* greetings.c */
#include <stdio.h>
#include "greetings.h"

void greetings(void)
{
  printf ("Hello user !\n");
}
```

kompilacja

konsolidacja

```
gcc -g -Wall -pedantic -c main.c -o main.o
gcc -g -Wall -pedantic -c greetings.c -o greetings.o
gcc -g main.o greetings.o -o hello
```

Wiele modułów - zmiana w `greetings.c`

```
/* greetings.h */
void greetings(void);
```

```
/* main.c */
#include <stdio.h>
#include "greetings.h"

int main()
{
  greetings();
  return 0;
}
```

```
/* greetings.c */
#include <stdio.h>
#include <stdlib.h>
#include "greetings.h"

void greetings(void)
{
  printf ("Hello %s!\n",
   getenv("LOGNAME"));
}
```

```
gcc -g -Wall -pedantic -c greetings.c -o greetings.o
gcc -g main.o greetings.o -o hello
```

Wiele modułów - dalsze zmiany

```
/* greetings.h */
int greetings(void);
```

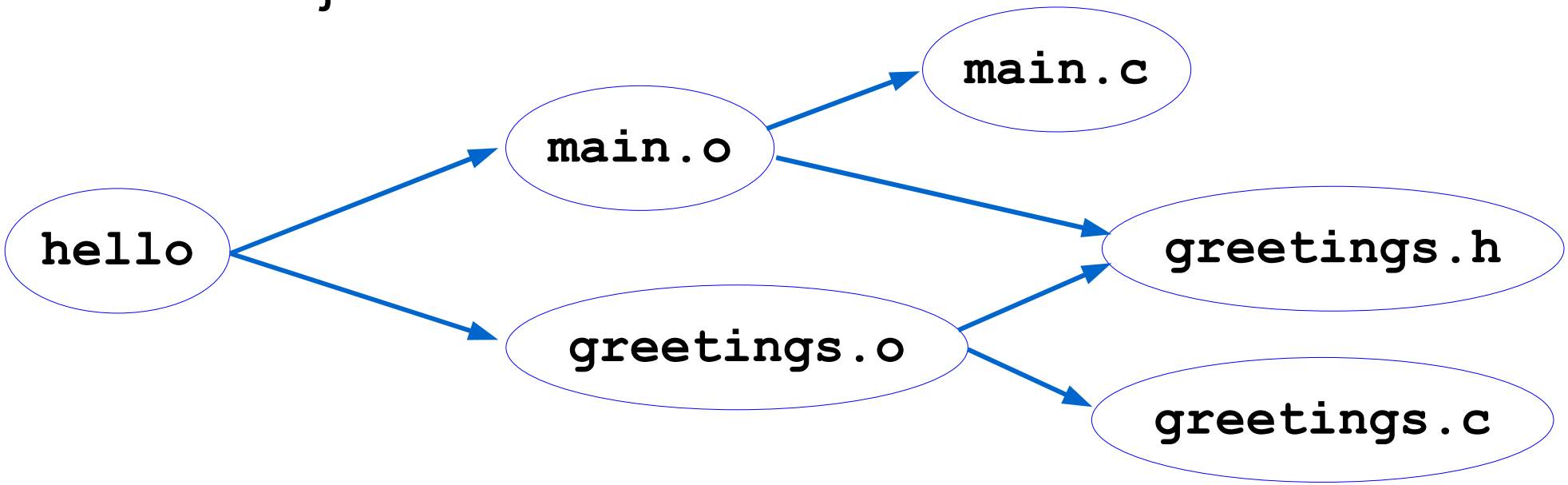
```
/* main.c */
#include <stdio.h>
#include "greetings.h"

int main()
{
  greetings();
  return 0;
}
```

```
/* greetings.c */
#include <stdio.h>
#include <stdlib.h>
#include "greetings.h"

int greetings(void)
{
  return printf ("Hello %s!\n",
   getenv("LOGNAME"));
}
```

```
gcc -g -Wall -pedantic -c main.c -o main.o
gcc -g -Wall -pedantic -c greetings.c -o greetings.o
gcc -g main.o greetings.o -o hello
```

Zależności między modułami

- Moduł A jest zależny na etapie kompilacji od modułu B, jeżeli **`A.c`** wymaga **`B.h`** w celu kompilacji
- Program A jest zależny na etapie konsolidacji od modułu B, jeżeli A wymaga **`B.o`** w celu konsolidacji
- Unikaj zależności cyklicznych (A $\rightarrow$ B i B $\rightarrow$ A)
- Zarządzanie złożonością oprogramowania wymaga m.in. minimalizacji zależności



Makefile i `make`

- Łatwo jest zmienić plik nagłówkowy, ale zapomnieć o rekompilacji wszystkich zależnych od niego plików źródłowych
- Pliki Makefile pomagają w zarządzaniu zależnościami
- **`make`** jest programem często używanym do kompilacji pakietów oprogramowania złożonych z wielu plików źródłowych
  - określa automatycznie które pliki trzeba wygenerować ponownie
  - używa pliku konfiguracyjnego (zazwyczaj zwanego **`makefile`** lub **`Makefile`**), określającego reguły i zależności dotyczące generacji każdego z plików
- Plik konfiguracyjny może mieć też inną nazwę, podaną jako argument polecenia make

  `bash# make`

  `bash# make -f myfile.mk`
- pierwsza wersja używa domyślnego pliku konfiguracyjnego (**`makefile`** lub **`Makefile`**)
- druga wersja używa **`myfile.mk`** jako pliku konfiguracyjnego

Prosty plik `makefile`

```
hello: main.o greetings.o
        gcc -g main.o greetings.o -o hello
main.o: main.c greetings.h
        gcc -g -Wall -pedantic -c main.c -o main.o

greetings.o: greetings.c greetings.h
        gcc -g -Wall -pedantic -c greetings.c -o greetings.o
```

linia zależności

linia operacji

linia zależności + operacja = reguła

Linia zależności musi zaczynać się w kolumnie 1

Linia operacji musi zaczynać się znakiem tabulacji

```
$ make
gcc -g -Wall -pedantic -c main.c -o main.o
gcc -g -Wall -pedantic -c greetings.c -o greetings.o
gcc -g main.o greetings.o -o hello
$ make
make: `hello' is up to date.
```

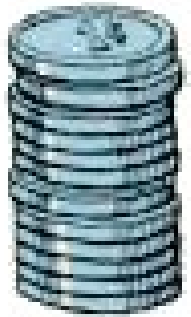
Wielokrotne włączanie tego samego pliku

- Kiedy włączane pliki nagłówkowe włączają inne pliki, niektóre pliki nagłówkowe mogą być dołączone wielokrotnie
  - Czasami nie jest to szkodliwe
    - np. wielokrotne deklaracje zmiennych z uzyciem **`extern`**
  - Czasami powoduje błędy
    - np. nie wolno wykonywać **`typedef`** dwukrotnie dla tego samego typu
  - Zawsze wydłuża to czas kompilacji
- W celu zapobieżenia takim sytuacjom, używa się specjalnej konstrukcji, zwanej **`#include`** guard

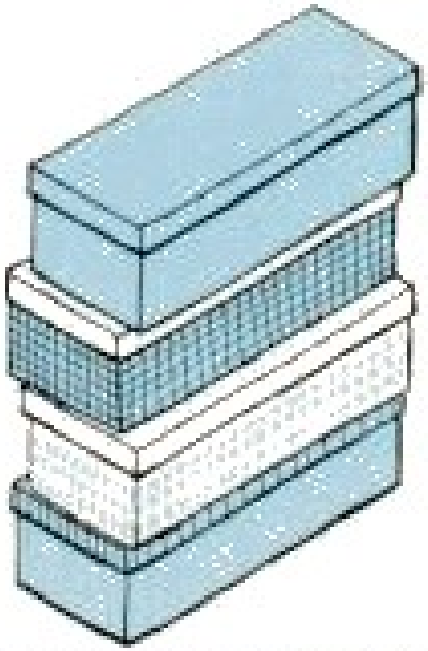
```
/* header.h */

#ifndef _HEADER_H_

#define _HEADER_H_

/* header body */

typedef some_type some_name;

#endif /* _HEADER_H_ */
```

Stosy w życiu codziennym

A stack of
cafeteria trays



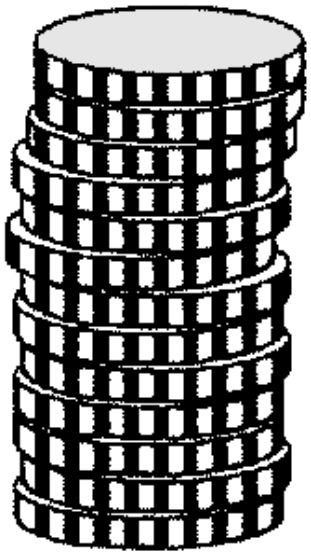
A stack
of pennies



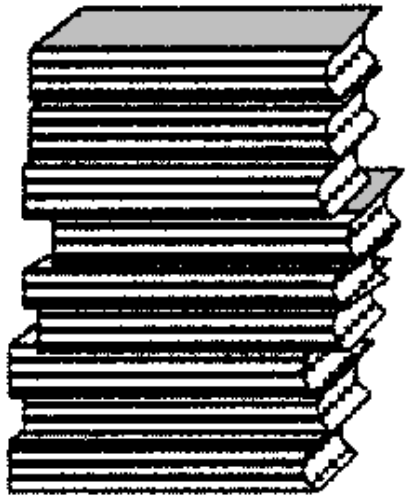
A stack of shoe boxes

A stack of
neatly folded shirts

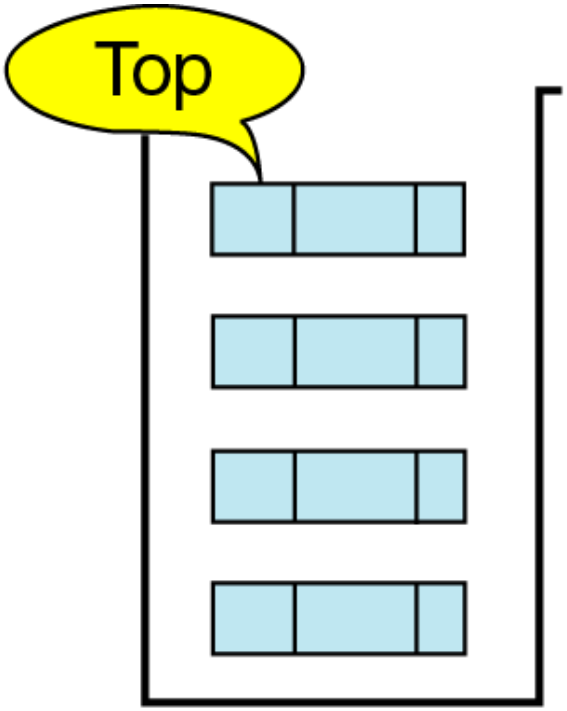
Więcej stosów



Stack of coins



Stack of books

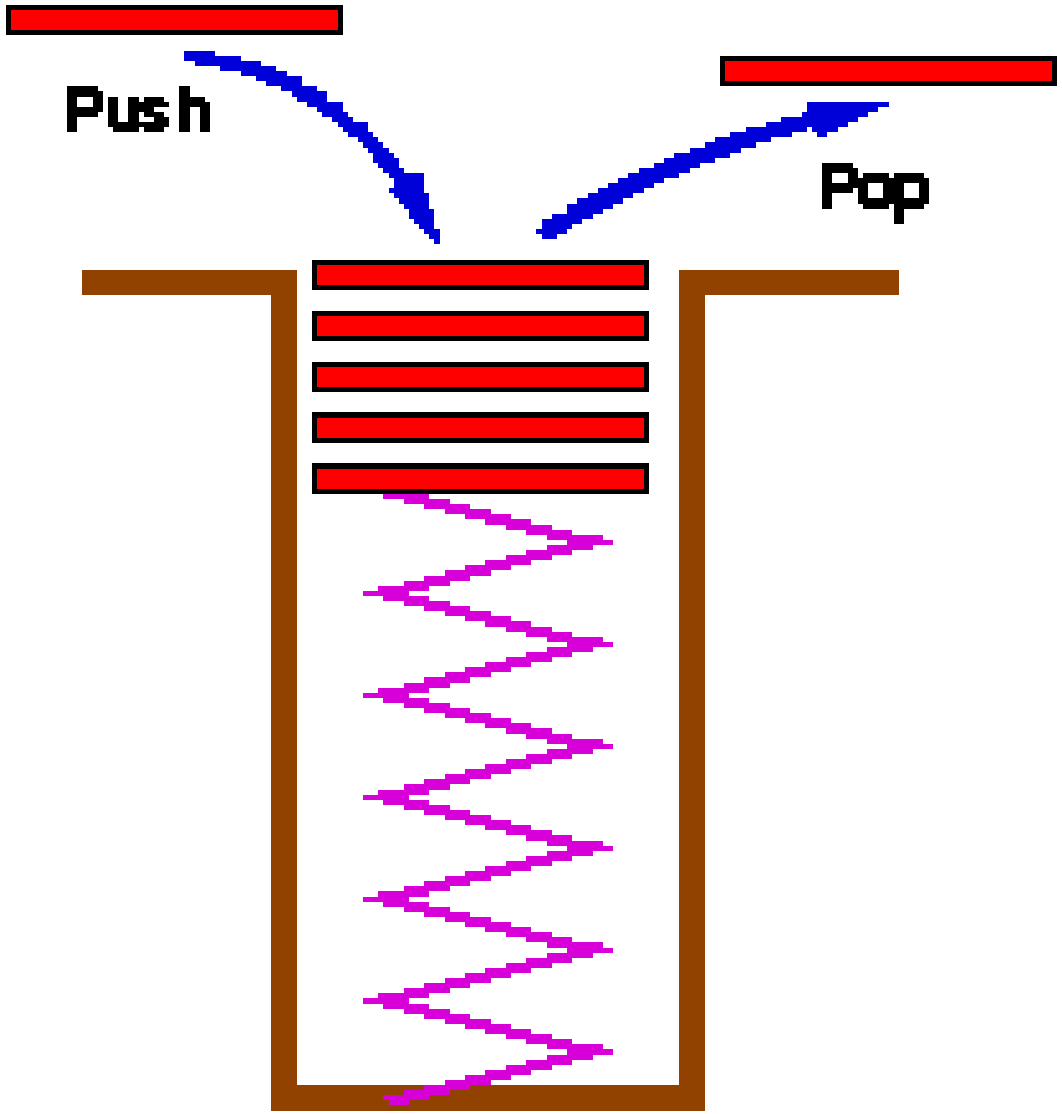


Computer stack

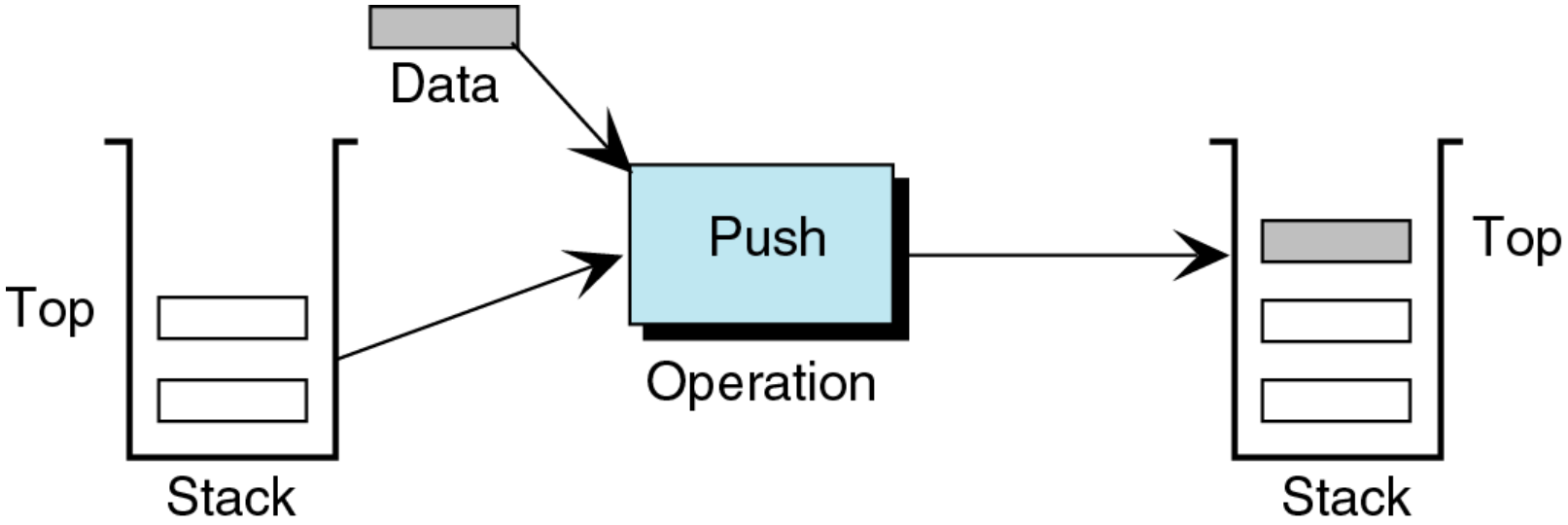
- Stos to struktura LIFO: Last In First Out

Podstawowe operacje na stosie

- Push
  - Dodanie elementu
    - Przepełnienie
- Pop
  - Zdjęcie elementu
    - *Underflow*
- Stack Top
  - Co jest na wierzchu
    - Stos może być pusty

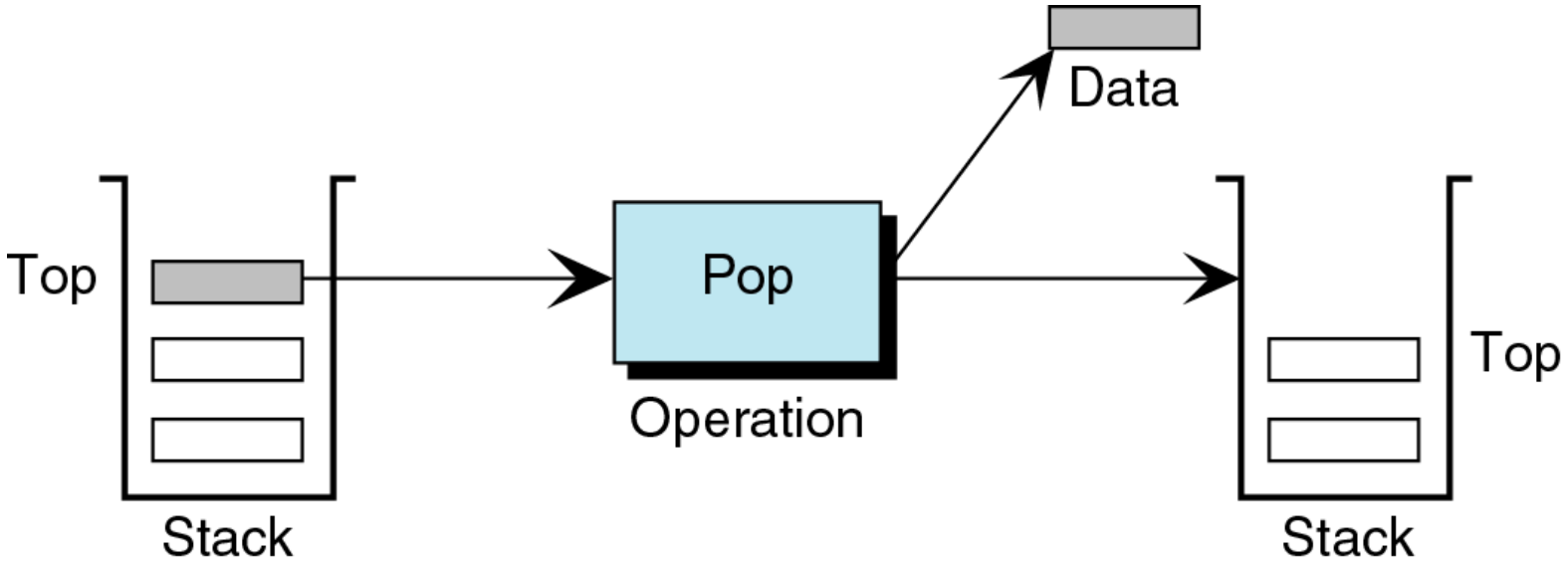


Push



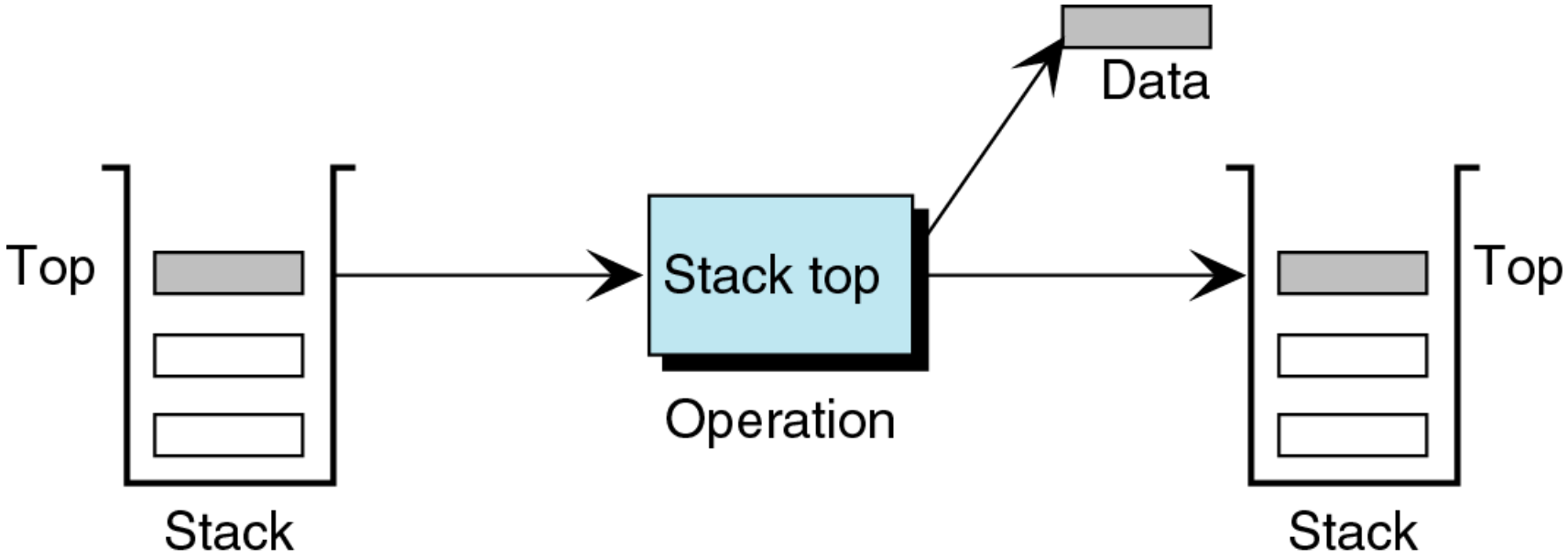
- Dodaje nowy element na wierzchołek stosu

Pop



- Zdejmuje element z wierzchołka stosu

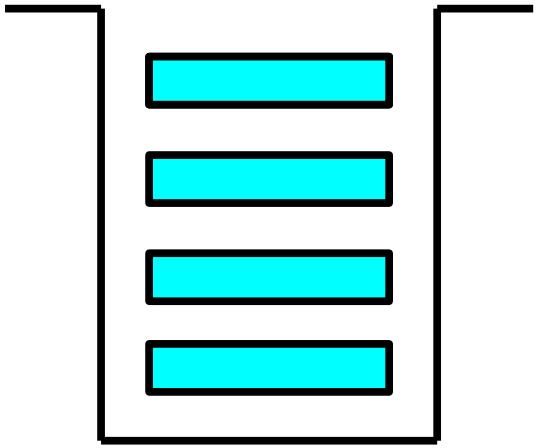
Stack Top



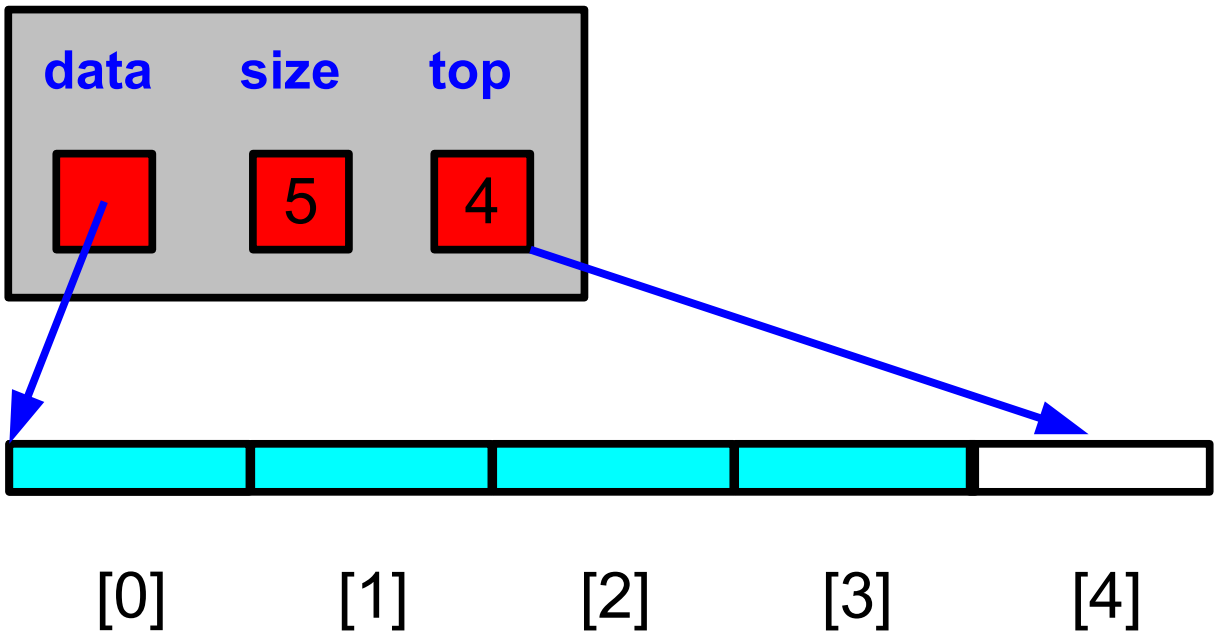
- Odczytuje górny element stosu. Nie modyfikuje stosu.

Implementacja stosu przy pomocy tablicy

- Stos można zaimplementować w oparciu o tablicę
  - Ta implementacja ma ustaloną pojemność



Koncepcja



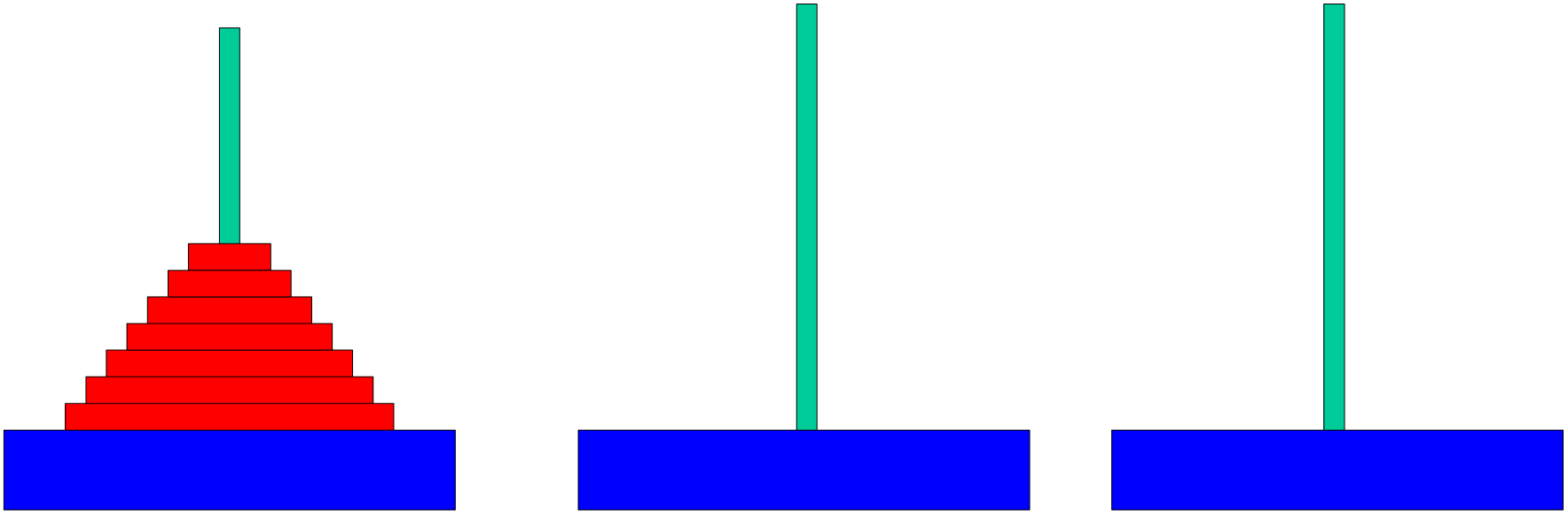
Fizyczna implementacja

Implementacja stosu

```
/* stack.h */
void push(int a);
int pop(void);
int peek(void);
```

```
/* stack.c */
#include <assert.h>
#include "stack.h"

#define STACKSIZE 20
static int top; /* first empty slot on the stack */
static int size=STACKSIZE;
static int data[STACKSIZE];

void push(int a)
{
        assert(top<size);
        data[top++]=a;
}

int pop(void)
{
        assert(top>0);
        return data[--top];
}

int peek(void)
{
        assert(top>0);
        return data[top-1];
}
```

Przykład: wieże w Hanoi



- Celem gry jest przełożenie zestawu krążków, ułożonych w stos na wieży A, na wieżę C przy użyciu pomocniczej wieży B; wolno nam przekładać jedynie pojedyncze krążki, nie wolno kłaść krążka większego na mniejszym
- Stosy mogą być zaimplementowane jako tablica dwuwymiarowa **`towers[3][7]`**

Uruchamianie programów (debugging)

- Często programy nie działają zgodnie z oczekiwaniami
- Potrzebujemy sposobu na przejście przez program krok po kroku innej, niż patrzenie na wydruk
- Znajomość przepływu sterowania i zmian wartości zmiennych podczas wykonywania programu pomaga w znalezieniu błędów
  - Można wstawić do programu instrukcje **`printf`** wypisujące wartości zmiennych w strategicznych miejscach
  - Użycie debugera jest zazwyczaj wygodniejsze
- Debuger jest programem przy pomocy którego można uruchamiać inne programy, sterować nimi i drukować wartości zmiennych w celu znalezienia i poprawienia błędów
- Najpopularniejszym debugerem dla systemów UNIX jest GDB, the GNU debugger

Podstawowe cechy debugera

- Debuger pozwala stwierdzić
  - W której instrukcji lub wyrażeniu program się zatrzymał
  - Jeżeli podczas wykonywania funkcji wystąpił błąd, która linia programu zawiera wywołanie tej funkcji i z jakimi parametrami
  - Jakie są wartości zmiennych w danym miejscu podczas wykonywania programu
  - Jaki jest wynik obliczenia dowolnego wyrażenia w danym momencie wykonywania programu

Używanie GDB

- Program kompilujemy z opcją **`-g`**
  - **`gcc -Wall -pedantic -g -c hello.c`**
  - **`gcc -g hello.o -o hello`**
- Program uruchamiamy z użyciem debugera
  - **`gdb hello`**
- Ustawiamy pułapkę w funkcji **`main`**
  - **`break main`**
- Uruchamiamy program
  - **`run`**

Polecenia GDB

- **`run` *`command-line-arguments`***
  - Uruchamia program jak po napisaniu z linii poleceń

    `hello command-line-arguments`
  - Możemy napisac

    `run <file1 >file2`

  aby przekierować standardowe wejście i wyjście do plików
- **`break` *`place`***
  - Zastawia pułapkę; program zatrzyma się we wskazanym miejscu kiedy do niego dojdzie. Najczęściej pułapki ustawia się na początku funkcji, jak np.
    - **`break main`**
  - Można również zastawić pułapkę w konkretnej linii pliku źródłowego
    - **`break 20`**
    - **`break hello.c:10`**

Polecenia GDB

- **`delete` *`N`***
  - Usuwa pułapkę numer *N*. Pominięcie argumentu usuwa wszystkie pułapki.
  - **`info break`** wypisuje informacje na temat zastawionych pułapek
- **`help` *`command`***
  - Opisuje pokrótce polecenie lub zagadnienie związane z GDB
  - **`help`** wypisuje listę mozliwych tematów
- **`step`**
  - Wykonuje bieżącą linię programu i zatrzymuje się przed wykonaniem kolejnej linii
- **`next`**
  - Jak **`step`**, z tym że jeżeli bieżąca linia programu zawiera wywołanie funkcji, przechodzi przez całe wywołanie funkcji i zatrzymuje się w następnej linii (nie wchodzi do wnętrza funkcji)

Polecenia GDB

- **`finish`**
  - Kontynuuje wykonanie programu do końca bieżącej funkcji
- **`continue`**
  - Kontynuuje wykonywanie programu do jego normalnego zakończenia lub napotkania pułapki
- **`where`**
  - Drukuje stos wywołań - łańcuch wywołań funkcji które przywiodły program do bieżącego miejsca
  - Równoważne polecenie to **`backtrace`**

GDB Commands

- **`print` *`E`***
  - drukuje wartość E w bieżącym miejscu programu, gdzie *E* jest poprawnym wyrażeniem w języku C (zwykle po prostu zmienną)
  - **`display`** działa podobnie, ale wyrażenie zamiast jednokrotnie jest drukowane po każdym zatrzymaniu programu w oparciu o uaktualnione wartości zmiennych
- **`quit`**
  - Kończy działanie GDB

Debugowanie "post-mortem"

- Można spowodować, że program będzie *zrzucał zawartość pamięci* po wywołaniu funkcji **`abort`**
- Polega to na utworzeniu pliku **`core`** na dysku w katalogu programu, zawierającego obraz pamięci programu w momencie awaryjnego zakończenia
- Moża wykorzystać ten plik w celu znalezienia przyczyny zatrzymania programu
- W celu włączenia tej opcji przy korzystaniu z powłoki bash, możemy napisać w linii poleceń

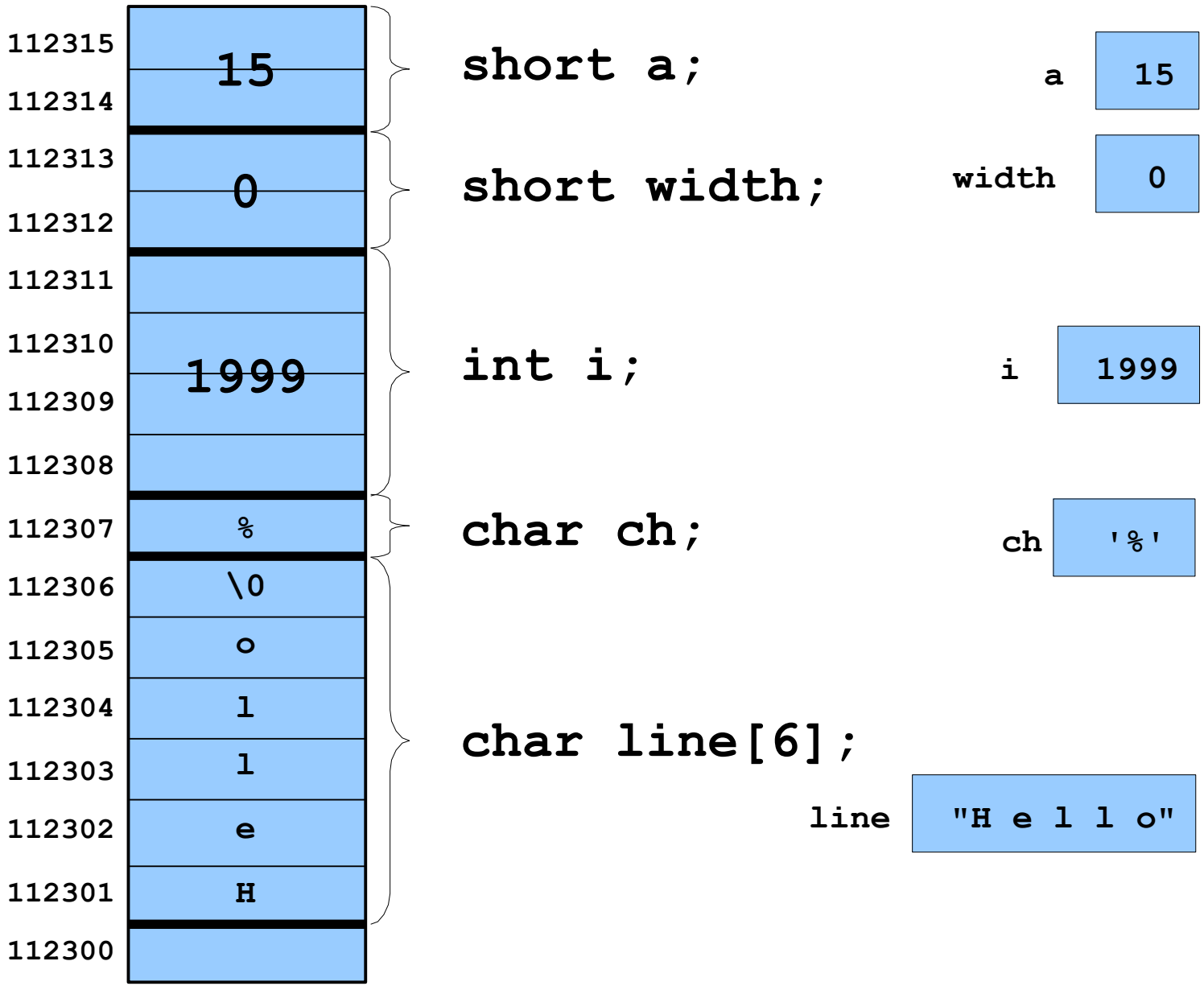
  `ulimit -c unlimited`
- Aby wykorzystać plik **`core`** w programie GDB używamy składni

  `gdb progname corefilename`

Pamięć komputera

- Pamięć komputera to duża tablica kolejno ponumerowanych komórek pamięci
- Każda komórka może przechować bajt danych
- Adres komórki pamięci to jej indeks w tablicy
- Dane różnych typów zajmują jedną lub więcej kolejnych komórek (bajtów)

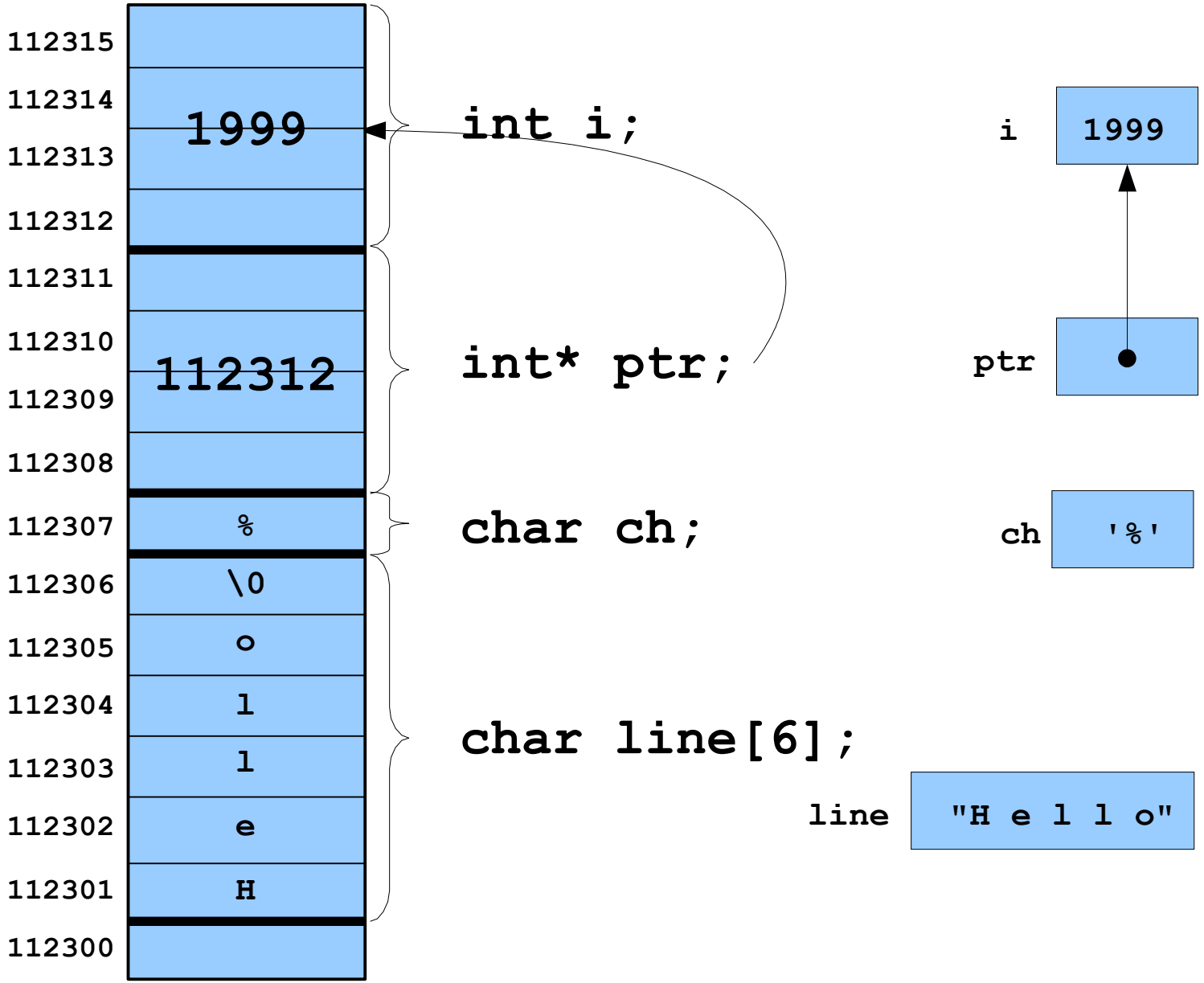
Pamięć komputera



Wskaźniki

- Wskaźnik to zmienna, która może zawierać adres innej zmiennej (np., 112300)
- Mówimy, że wskaźnik wskazuje na zmienną znajdującą się w pamięci pod tym adresem
- Wskaźnik zazwyczaj zajmuje 4 bajty pamięci (może wtedy adresować komórki o adresach od 0 do 2^{32}-1)

Pamięć komputera



Deklarowanie wskaźników

- W języku C określamy typ zmiennej na jaką może wskazywać wskaźnik:

```
int *ad; /* pointer to int */
char *s; /* pointer to char */
float *fp; /* pointer to float */
char **s; /* pointer to variable that is a
 pointer to char */
```

Operacje na wskaźnikach

- Możemy przypisać adres do wskaźnika

```
int *p1, *p2; int a, b;
p1 = &a;
```

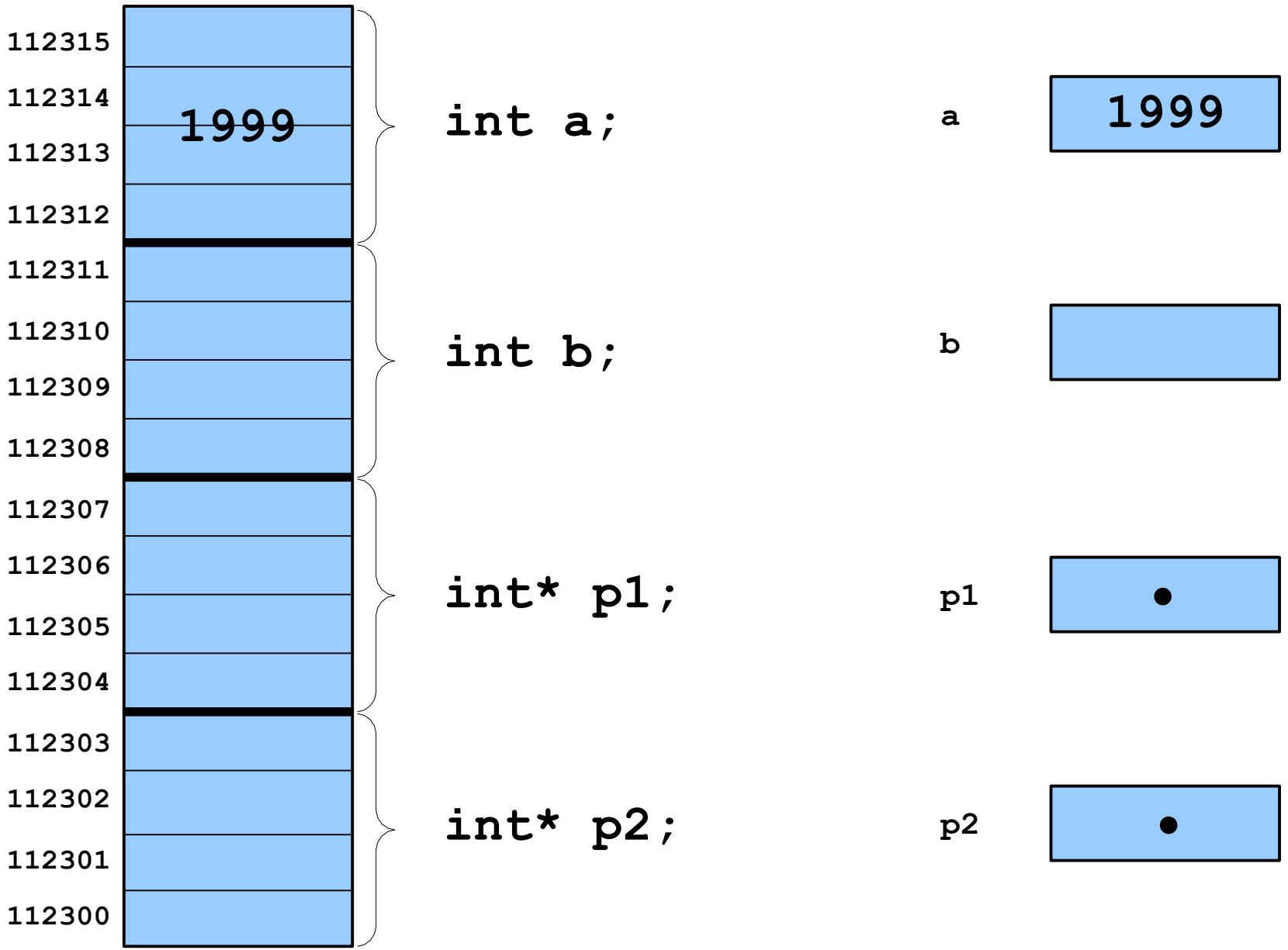
- Możemy przypisywać wskaźniki do siebie nawzajem

```
p2 = p1;
```

- Możemy *wyłuskiwać* wskaźniki

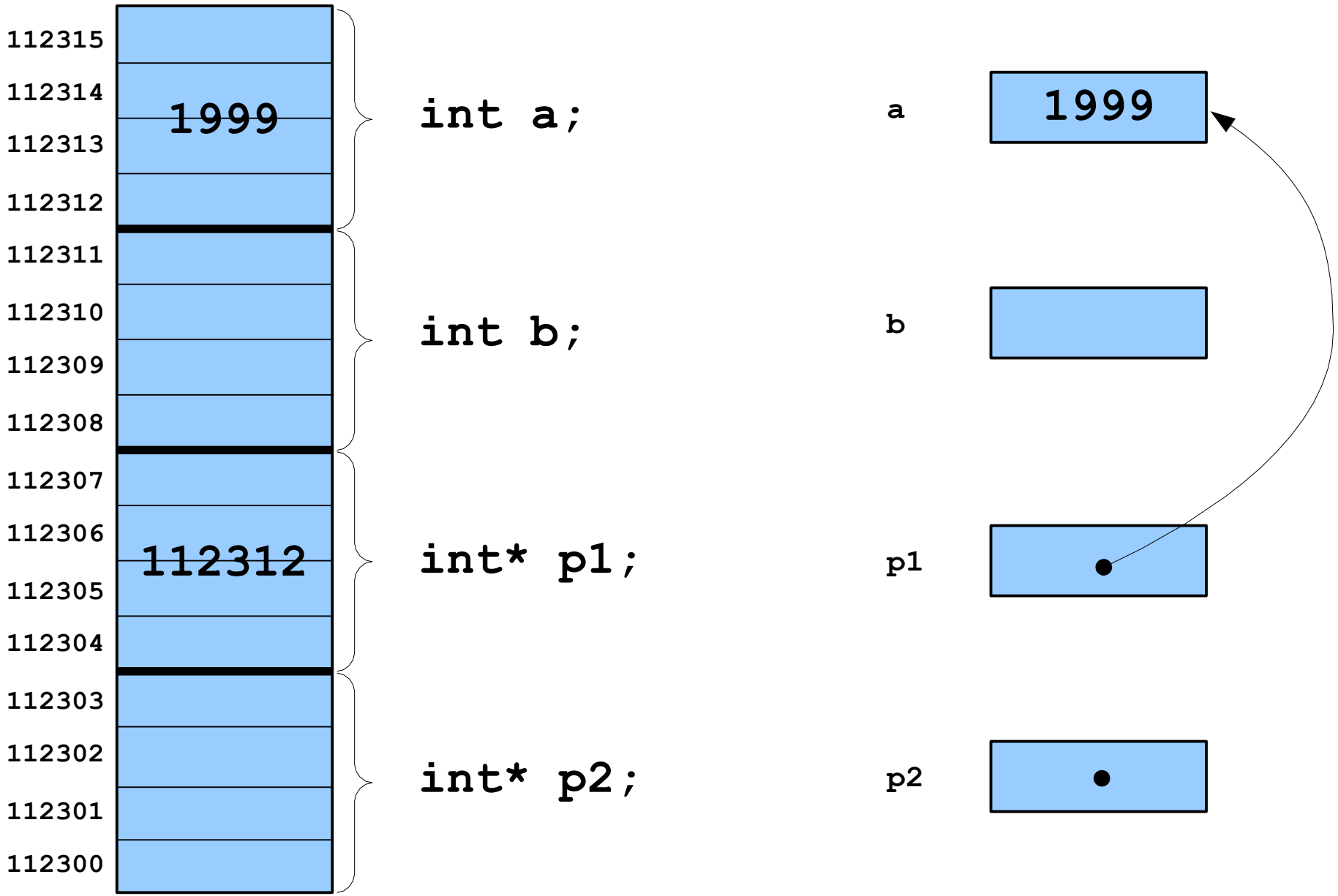
```
*p1 = 3; /* same as a = 3; */
b = *p1; /* same as b = a; */
```

Operacje & i *



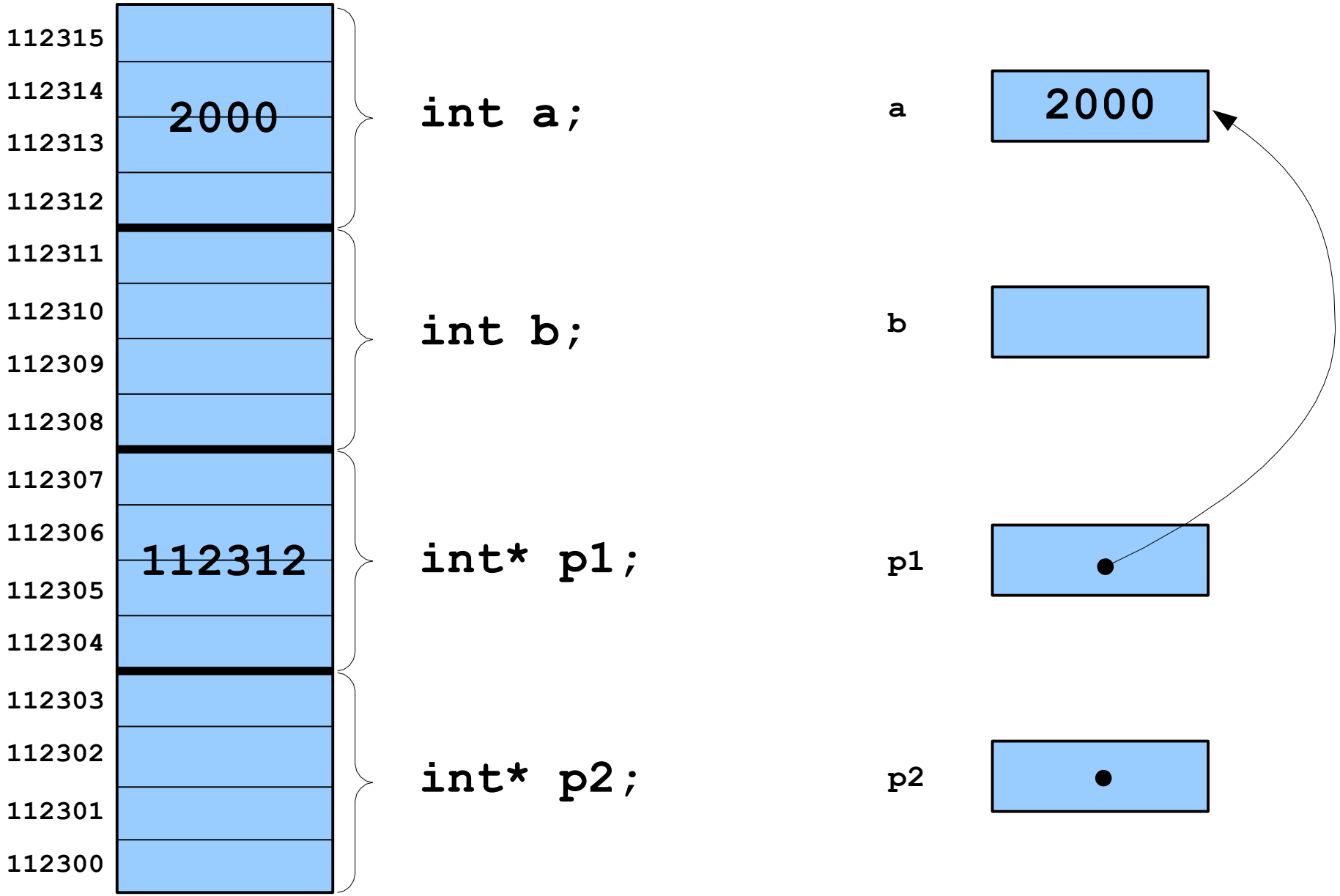

```
a = 1999;
```

Operacje & i *



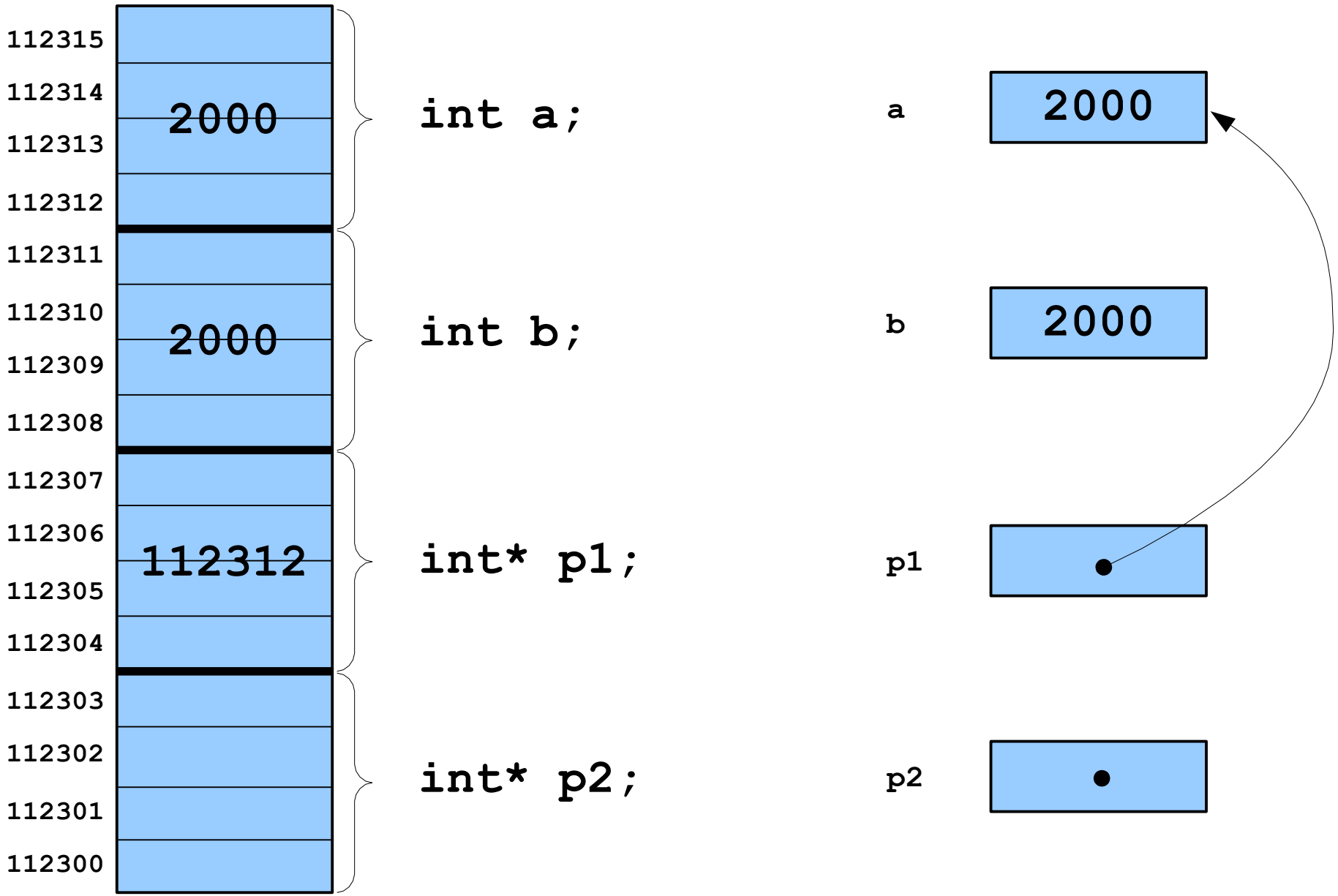

```
p1 = &a;
```

Operacje & i *



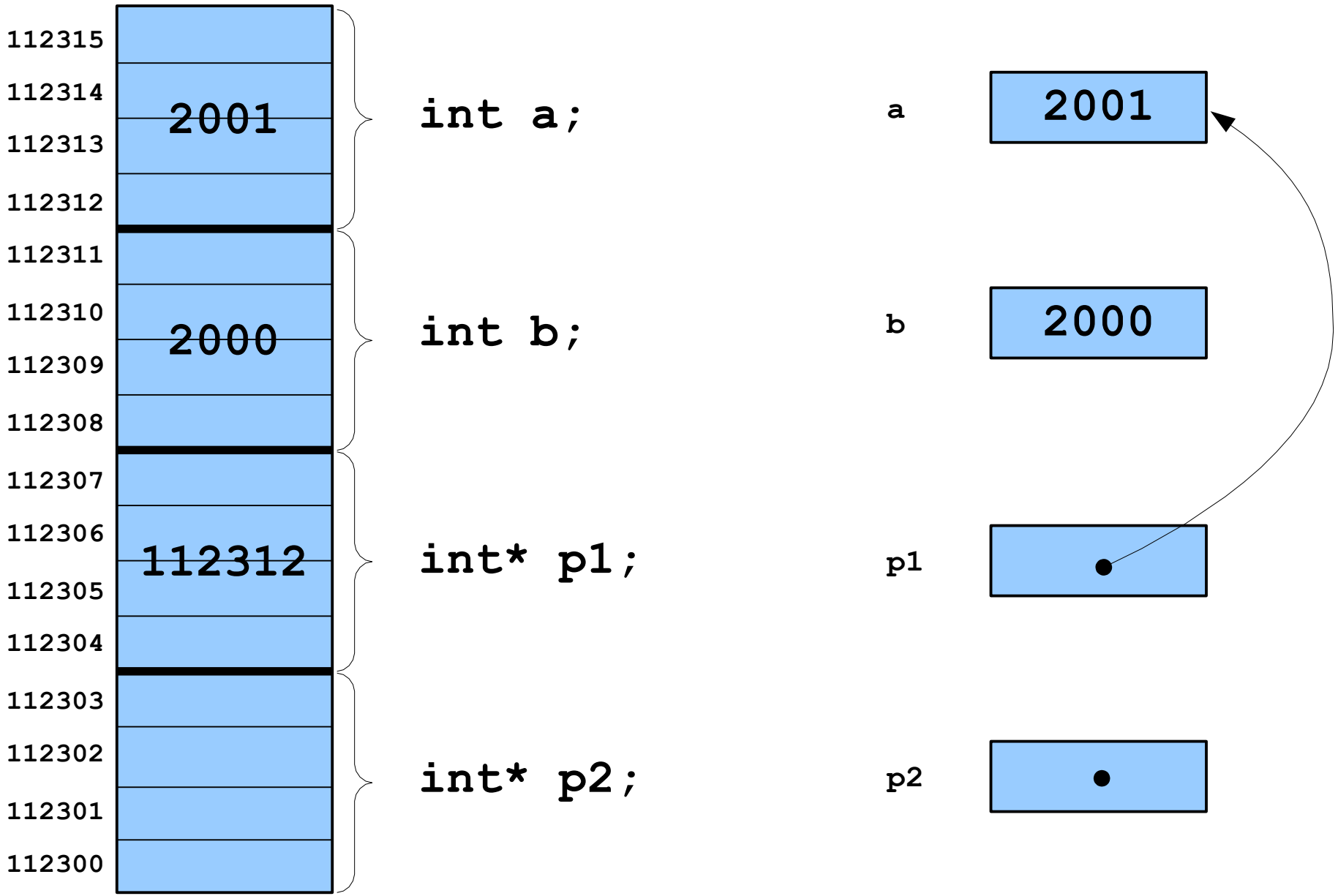

```
*p1 = 2000;
```

Operacje & i *



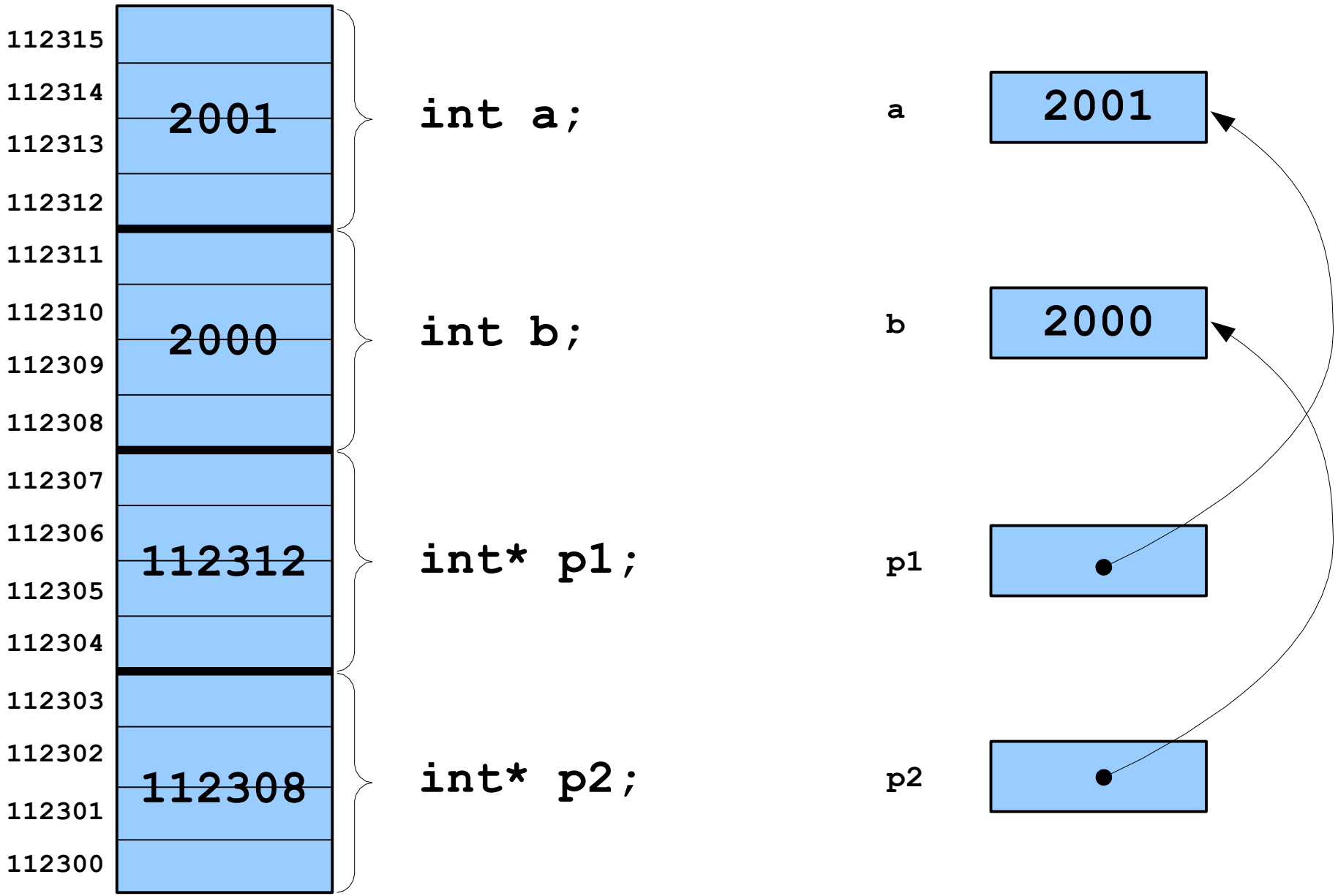

```
b=*p1;
```

Operacje & i *



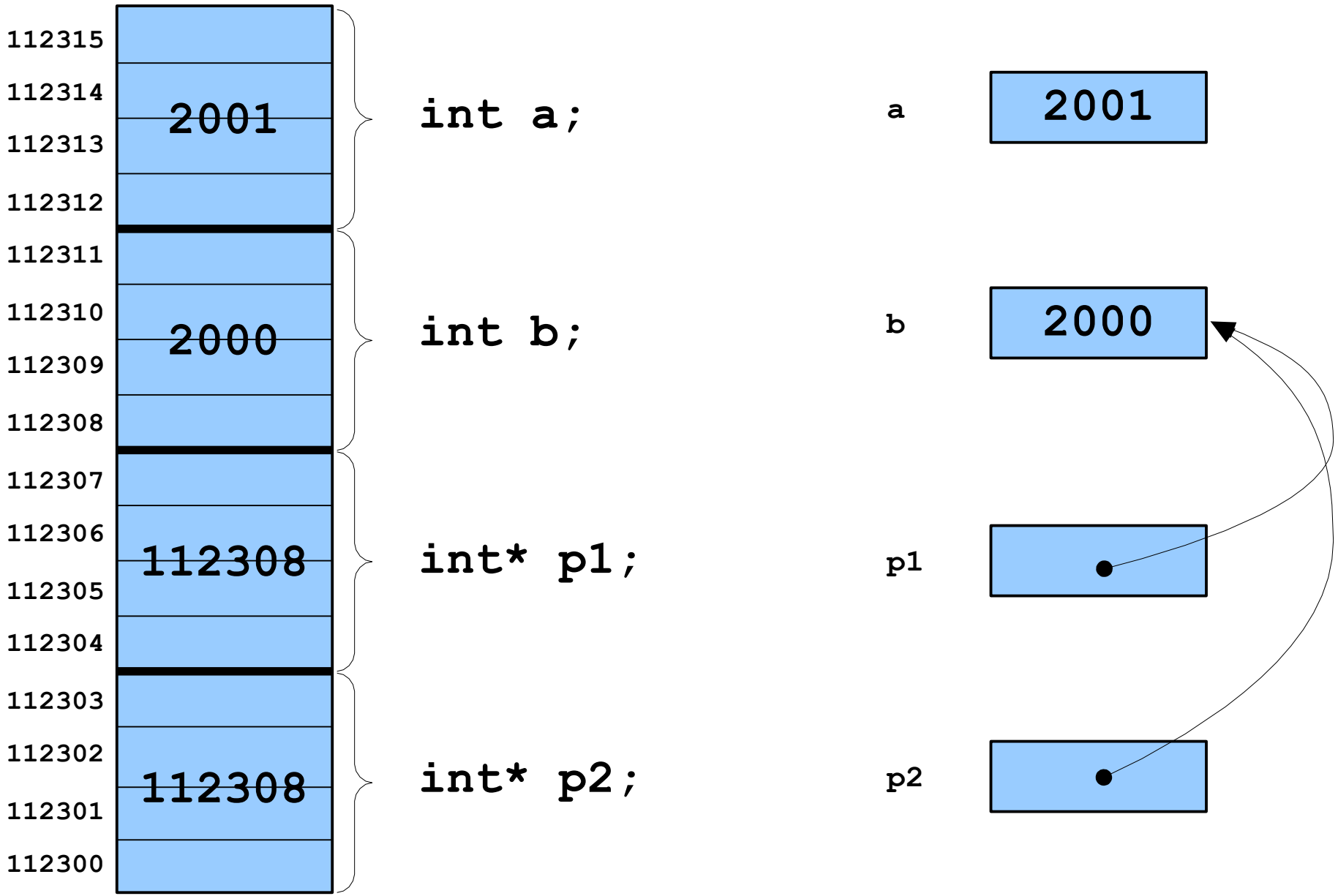

```
(*p1)++;
```

Operacje & i *



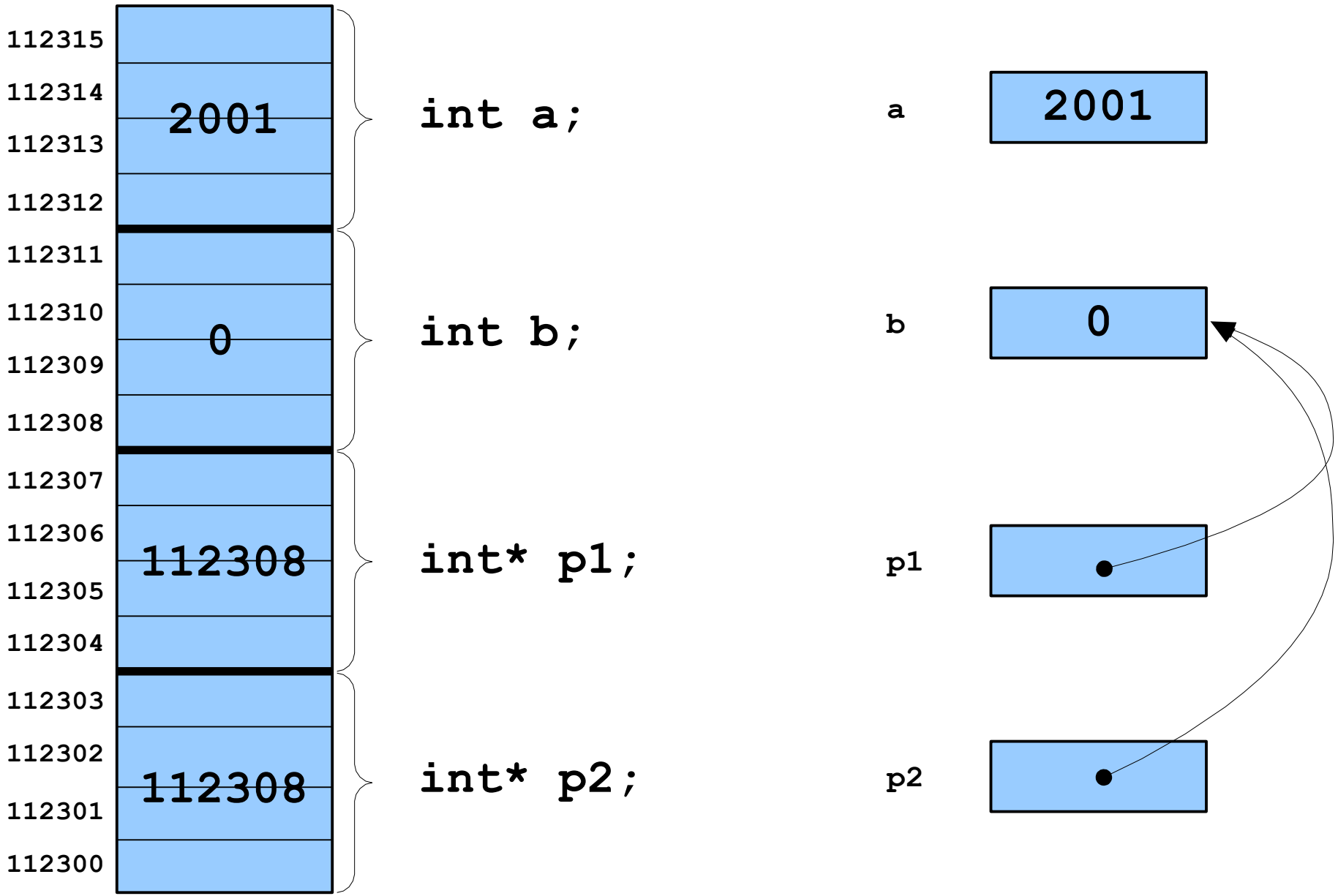

```
p2=&b;
```

Operacje & i *



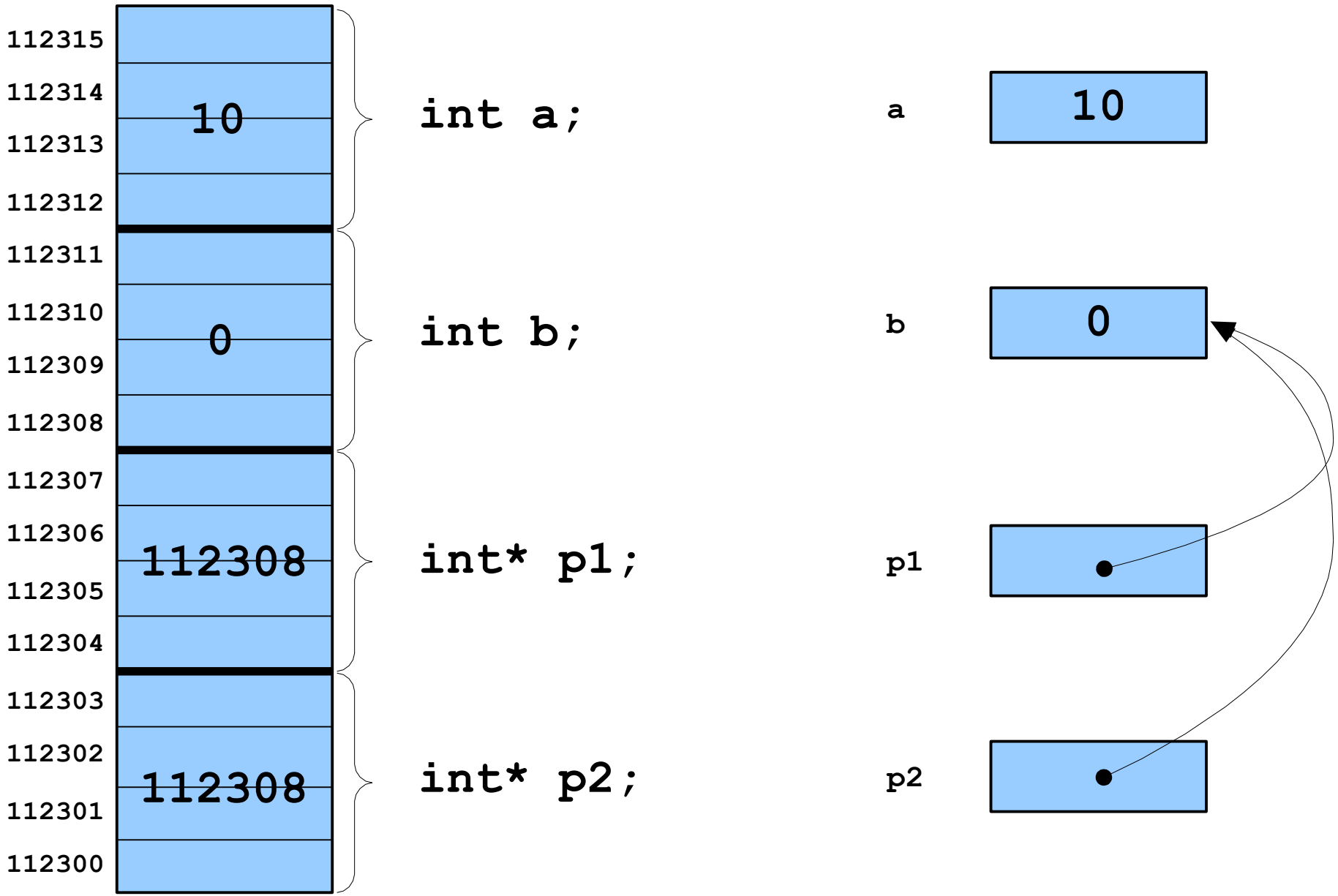

```
p1=p2;
```

Operacje & i *




```
*p1=0;
```

Operacje & i *




```
a=*p2+10;
```

Wskaźniki jako argumenty funkcji

- Argumenty funkcji w języku C są przekazywane przez wartość
- Możemy zasymulować przekazywanie parametrów przez referencję poprzez przekazanie wskaźnika
- Przydaje się to jeżeli potrzebujemy:
  - Wsparcia dla parametrów in/out (dwukierunkowych) (np. swap, find-replace)
  - Zwrócić kilka różnych wartości (jedna wartość zwracana nie wystarcza)
  - Przekazywać duże obiekty (tablice i struktury)

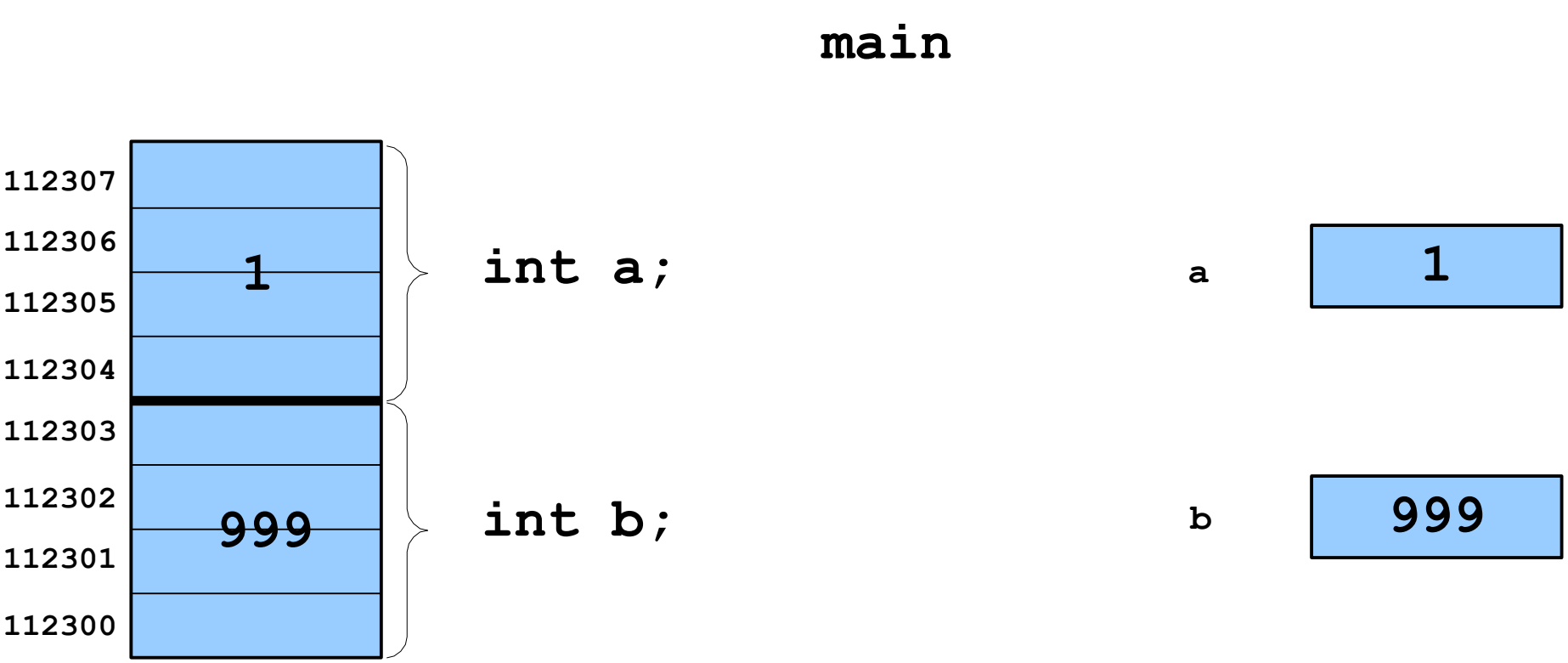
Wskaźniki jako argumenty funkcji

```
/* bad example of swapping
a function can't change parameters */
void bad_swap(int x, int y)
{
    int temp;

    temp = x;
    x = y;
    y = temp;
}

/* good example of swapping - a function can't change parameters,
   but if a parameter is a pointer it can change the value it points to */
void good_swap(int *px, int *py)
{
    int temp;

    temp = *px;
    *px = *py;
    *py = temp;
}
```

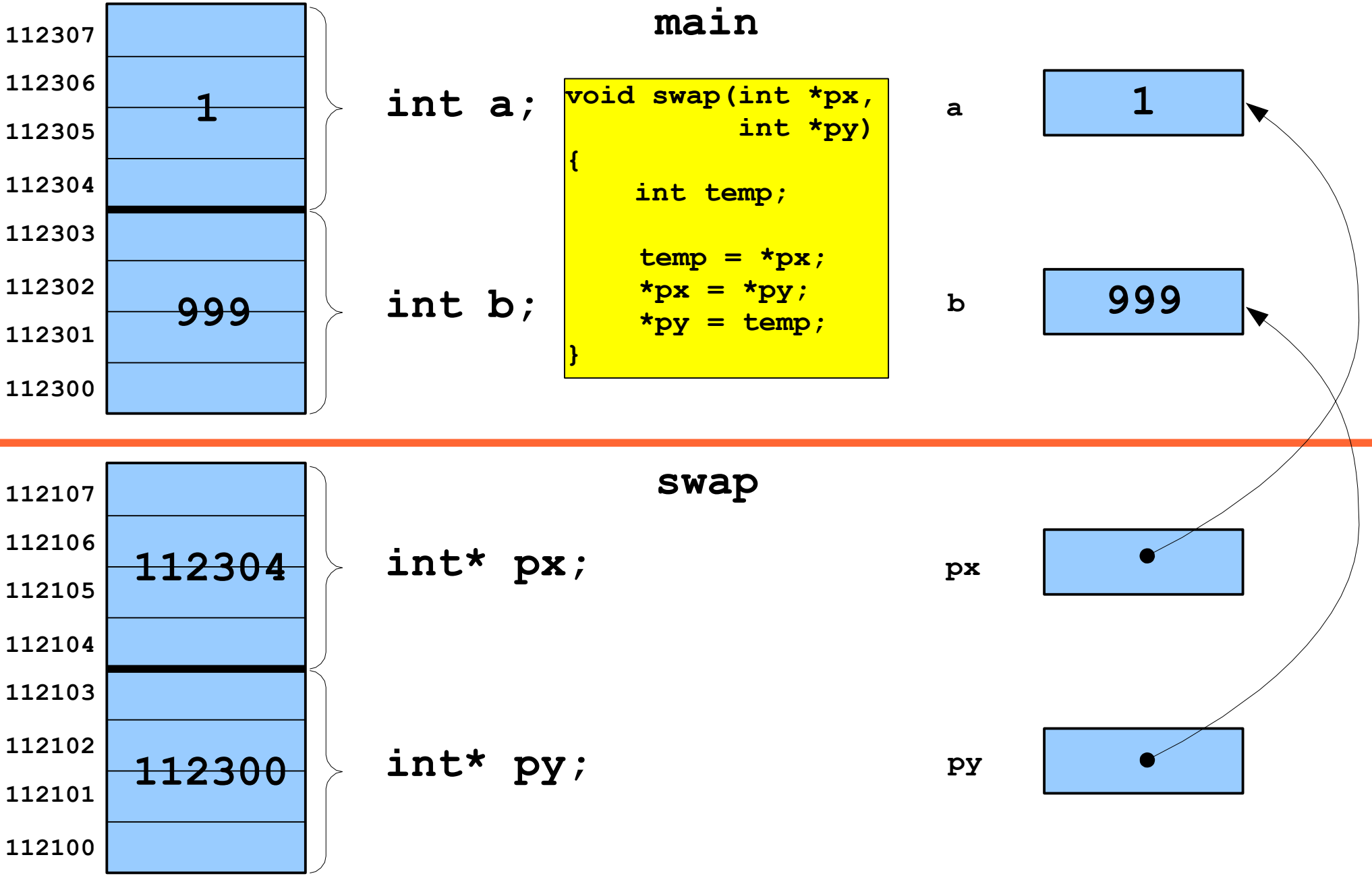
Wskaźniki jako argumenty funkcji

```
#include <stdio.h>
void bad_swap(int x, int y);
void good_swap(int *p1, int *p2);

main() {
    int a = 1, b = 999;

    printf("a = %d, b = %d\n", a, b);
    bad_swap(a, b);
    printf("a = %d, b = %d\n", a, b);
    good_swap(&a, &b);
    printf("a = %d, b = %d\n", a, b);
}
```

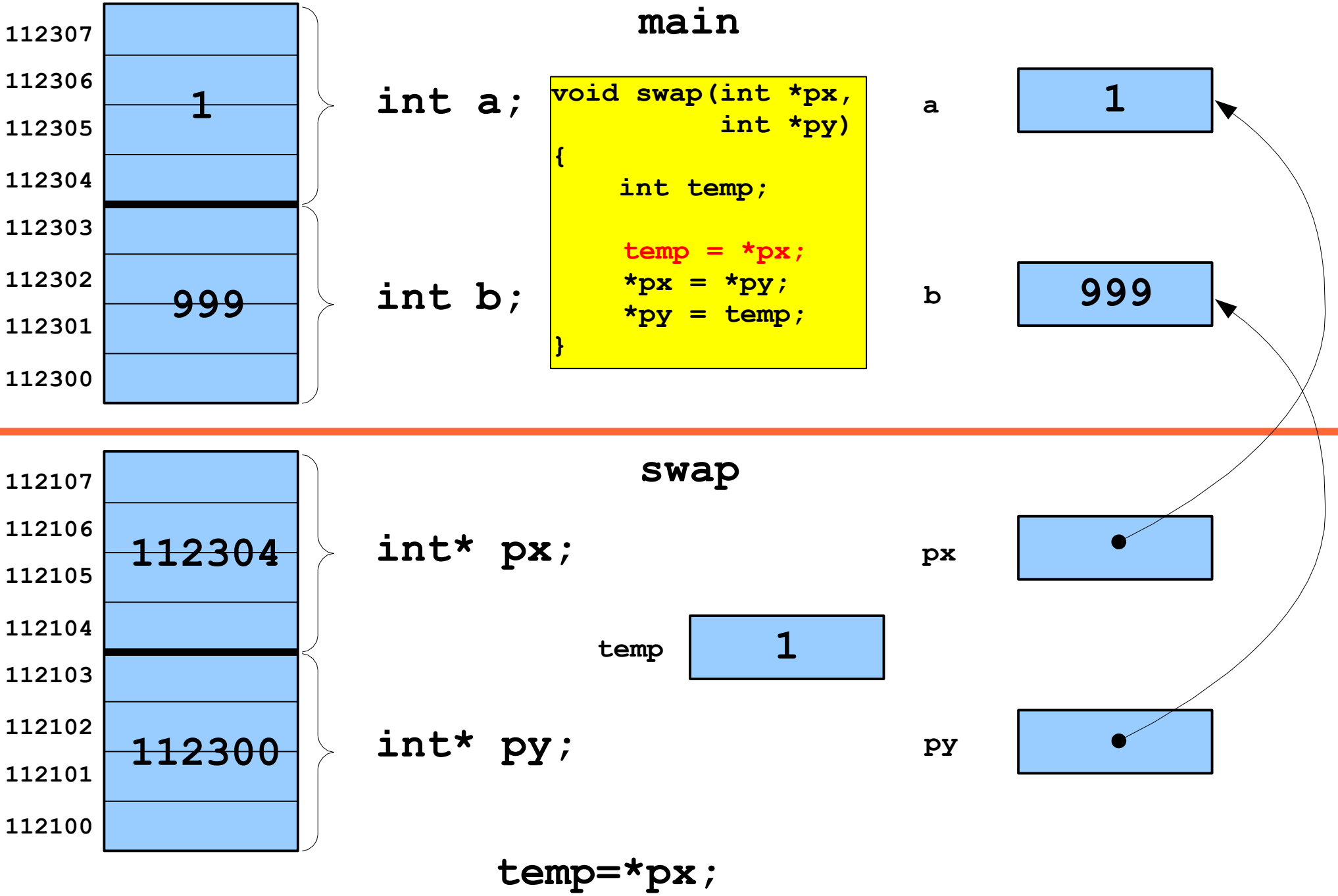
Pod maską


main
112307
112306
112305
112304
112303
112302
112301
112300
1
999
int a;
int b;
a
1
b
999

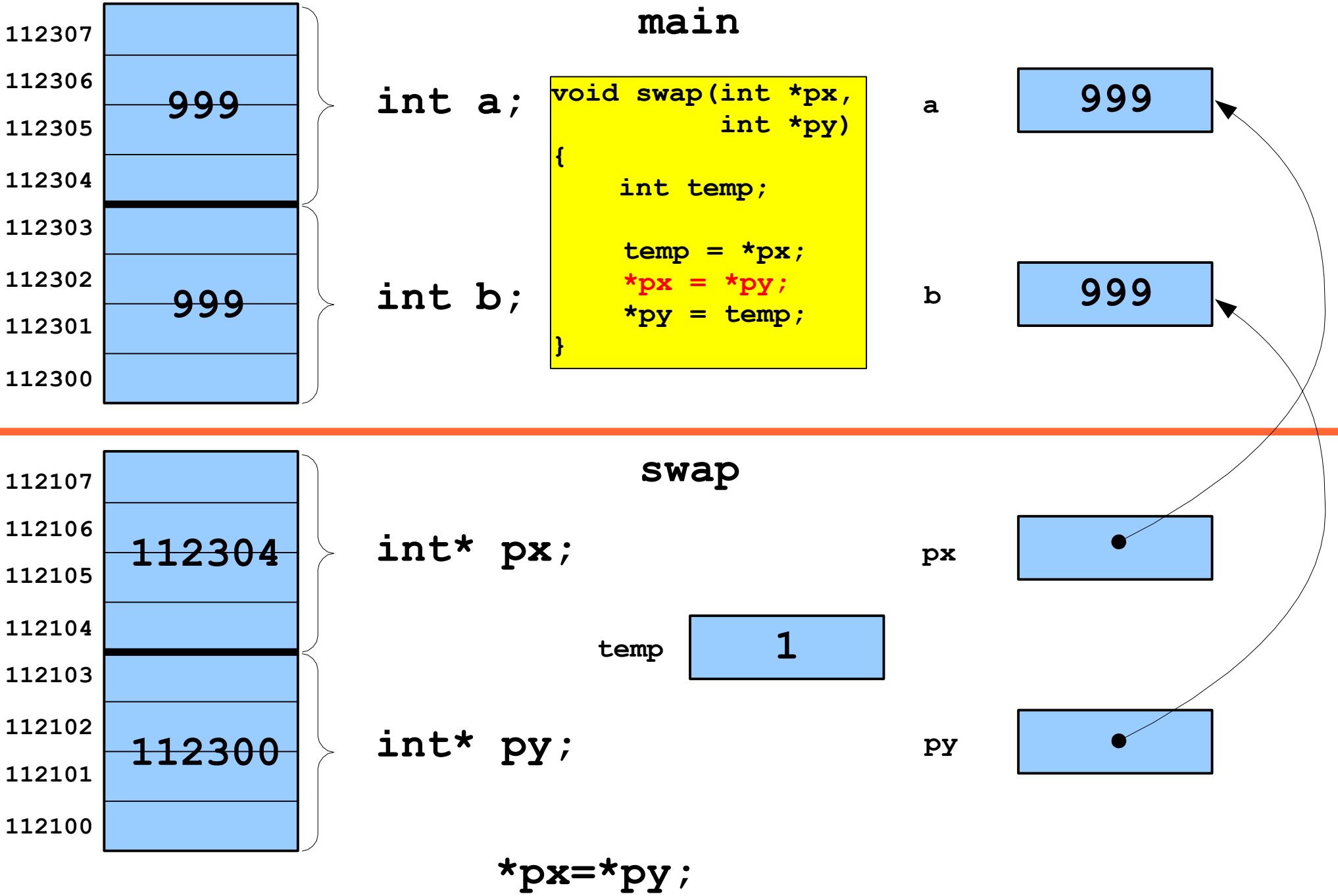
Pod maską


main
112307
112306
112305
112304
112303
112302
112301
112300
1
999
int a;
int b;
void swap(int *px,
int *py)
{
int temp;
temp = *px;
*px = *py;
*py = temp;
}
a
b
1
999
swap
112107
112106
112105
112104
112103
112102
112101
112100
112304
112300
int* px;
int* py;
px
py
swap(&a, &b);

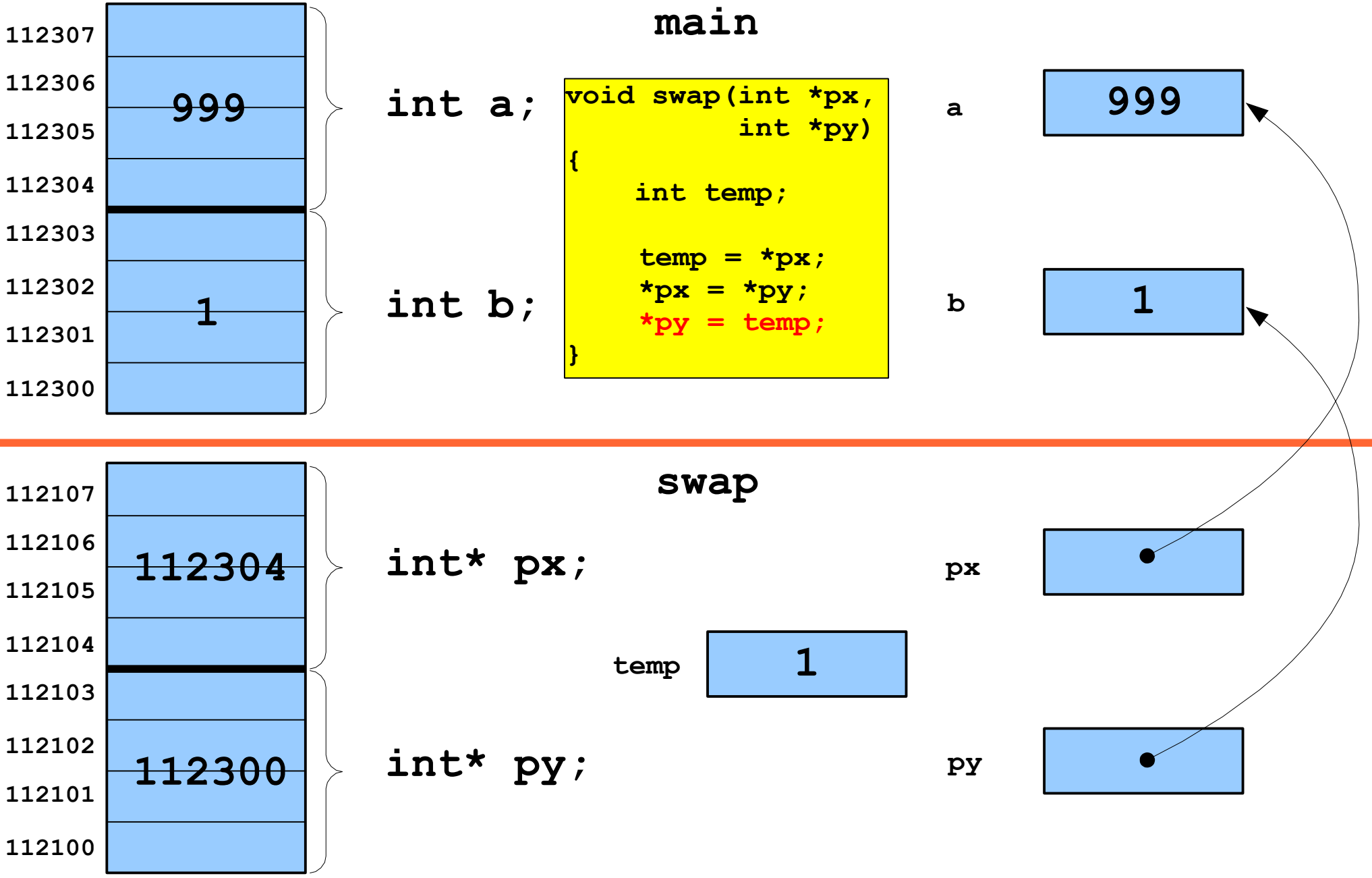
Pod maską


main
112307
112306
112305
112304
112303
112302
112301
112300
1
999
int a;
int b;
void swap(int *px,
int *py)
{
int temp;
temp = *px;
*px = *py;
*py = temp;
}
a
b
1
999
swap
112107
112106
112105
112104
112103
112102
112101
112100
112304
112300
int* px;
int* py;
temp
1
px
py
temp=*px;

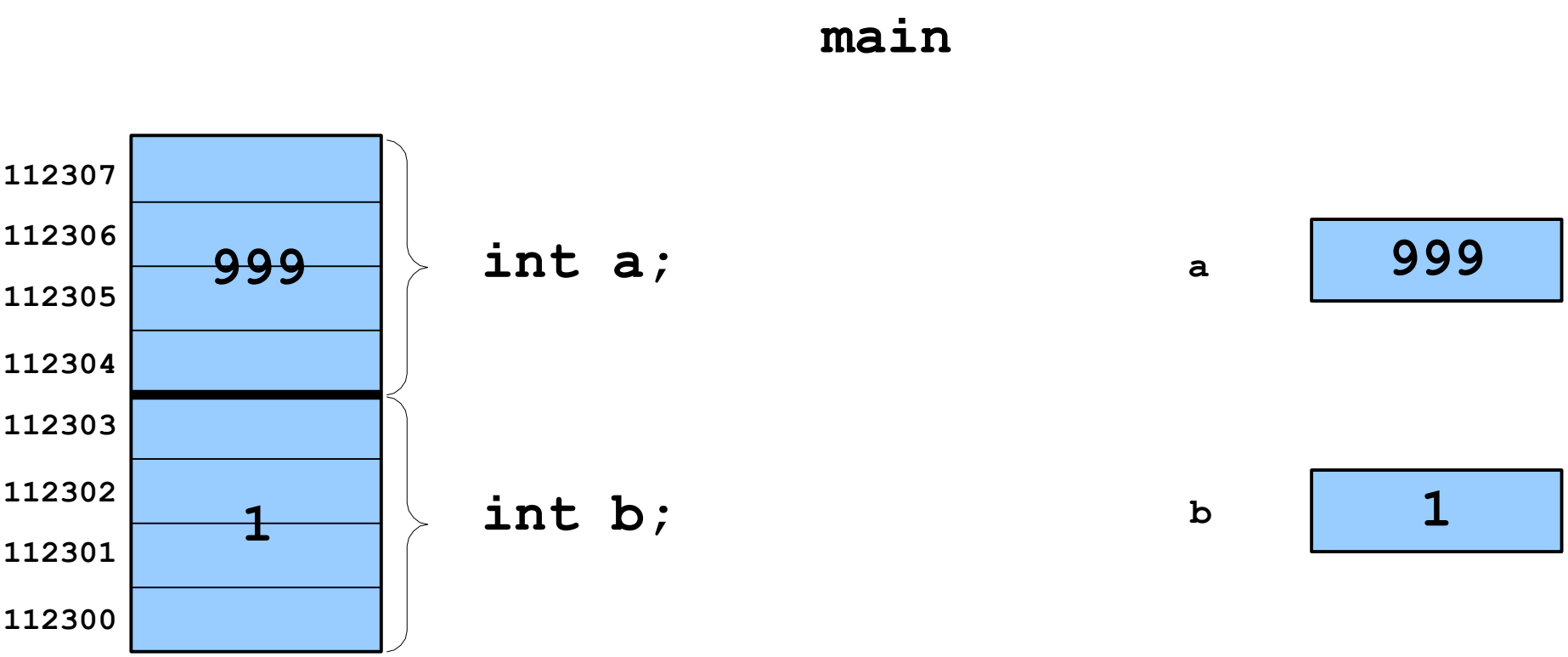
Pod maską


main
112307
112306
112305
112304
112303
112302
112301
112300
999
999
int a;
int b;
void swap(int *px,
int *py)
{
int temp;
temp = *px;
*px = *py;
*py = temp;
}
a
b
999
999
swap
112107
112106
112105
112104
112103
112102
112101
112100
112304
112300
int* px;
int* py;
temp
1
px
py
*px=*py;

Pod maską


main
112307
112306
112305
112304
112303
112302
112301
112300
999
1
int a;
int b;
void swap(int *px,
int *py)
{
int temp;
temp = *px;
*px = *py;
*py = temp;
}
a
b
999
1
swap
112107
112106
112105
112104
112103
112102
112101
112100
112304
112300
int* px;
int* py;
temp
1
px
py
*py=temp;

Pod maską


main
112307
112306
112305
112304
112303
112302
112301
112300
999
1
int a;
int b;
a
b
999
1

Wskaźniki jako argumenty funkcji

```
#include <stdio.h>   /* printf() */
#include <math.h>    /* pow() */

static void CalcAreaVol(double dRad, double* pdArea,
                        double* pdVol)
{
    *pdArea = 4.0 * M_PI * pow(dRad, 2.0);
    *pdVol  = 4.0/3.0 * M_PI * pow(dRad, 3.0);
}

int main()
{
   double area = 0.0, vol = 0.0;
   CalcAreaVol(3.5, &area, &vol);
   printf("Radius=%g Area=%g Vol=%g\n", 3.5,
     area, vol);
   return 0;
}
```

Czytanie z wejścia - `scanf`

- Jeżeli potrzebujemy wczytać wartość ze standardowego wejścia do zmiennej, możemy użyć funkcji **`scanf`**
- **`scanf`** może umieścić wczytaną wartość w zmiennej ponieważ przekazujemy jej adres zmiennej

```
int capital = 0;
scanf("%d", &capital);
```

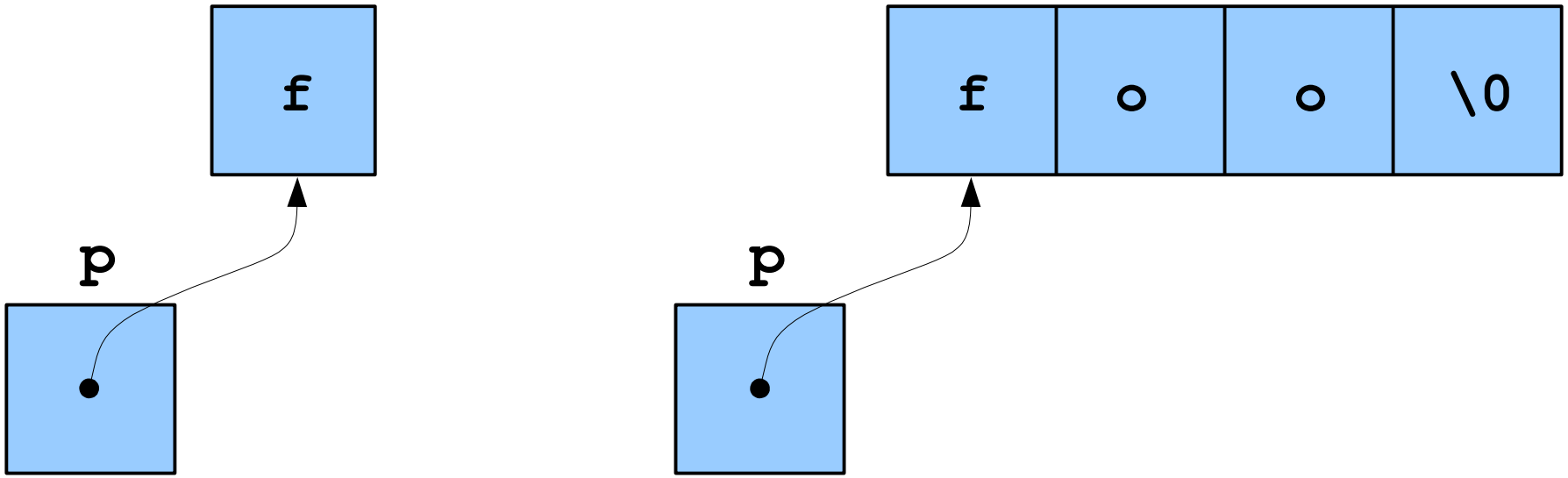
- Co by się stało, gdybyśmy pominęli &?

```
scanf("%d", capital);
```

- **`scanf`** potraktowałby wartość zmiennej **`capital`** jako adres, pod którym należy umieścić dane!

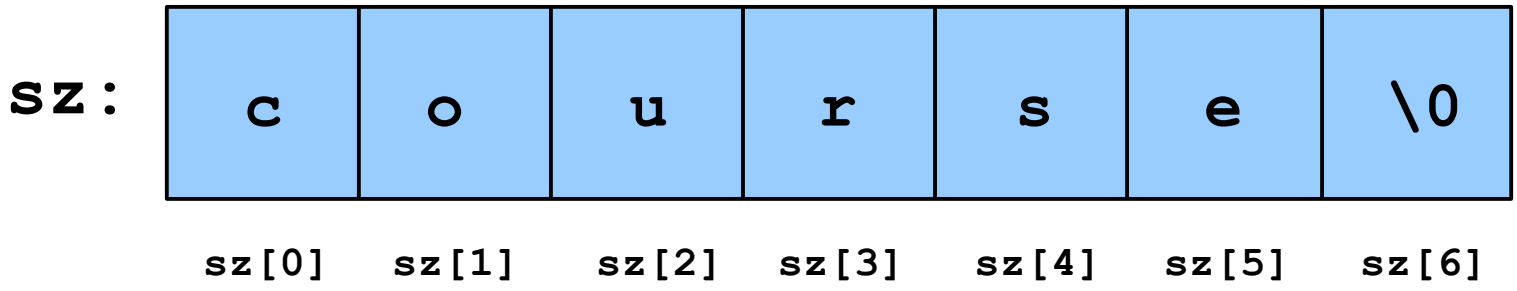
Wskaźniki do tablic

- Wskaźnik może wskazywać na:
  - pojedynczą zmienną (**`int`**, **`char`**, **`float`**, **`double`**, **`struct`**, itd.)
  - tablicę zmiennych
- Może to być mylące
- Rozważmy: **`char* p`**
- Czy musimy wiedzieć, czy wskaźnik wskazuje na pojedynczy znak, czy pierwszy element tablicy znaków?

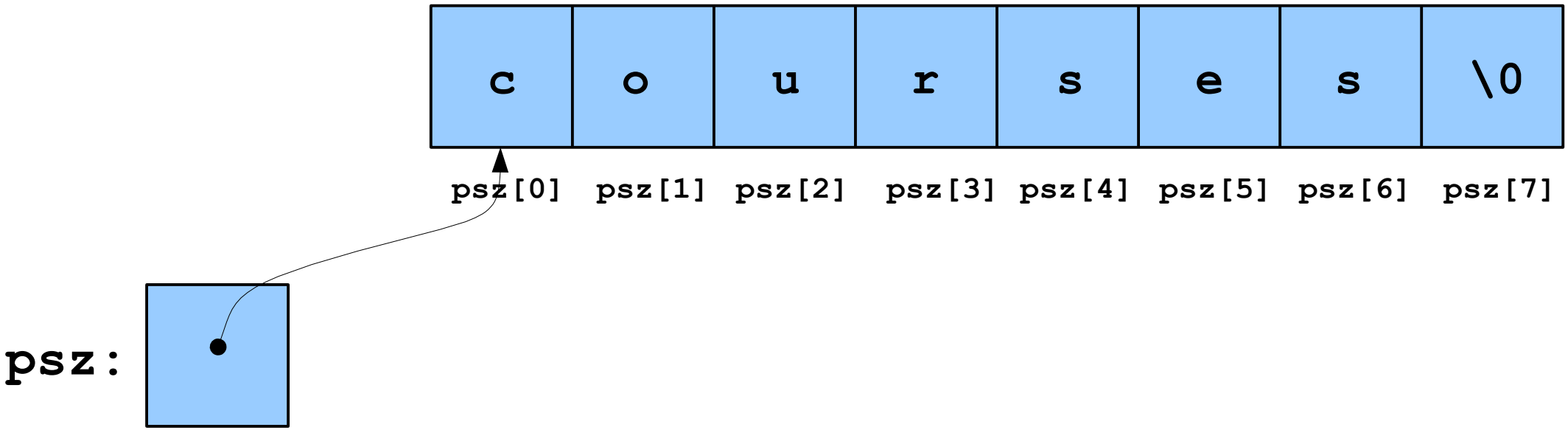


Znowu łańcuchy

```
char sz[] = "course";
```



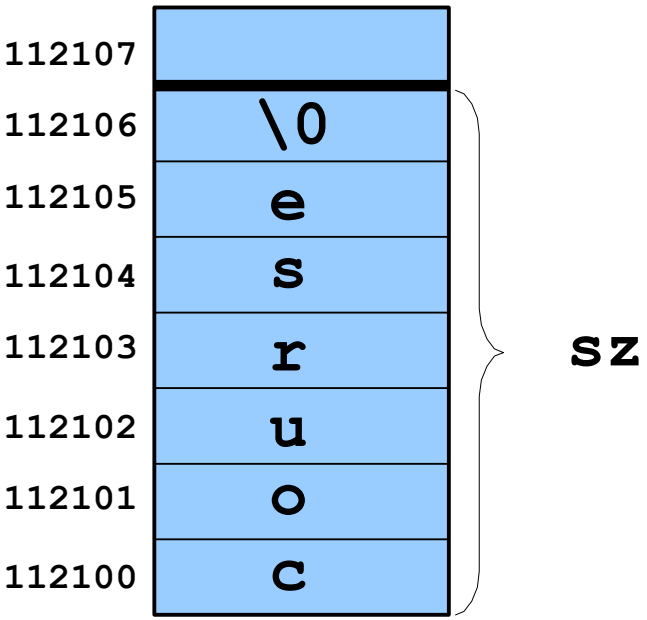

```
char* psz = "courses";
```

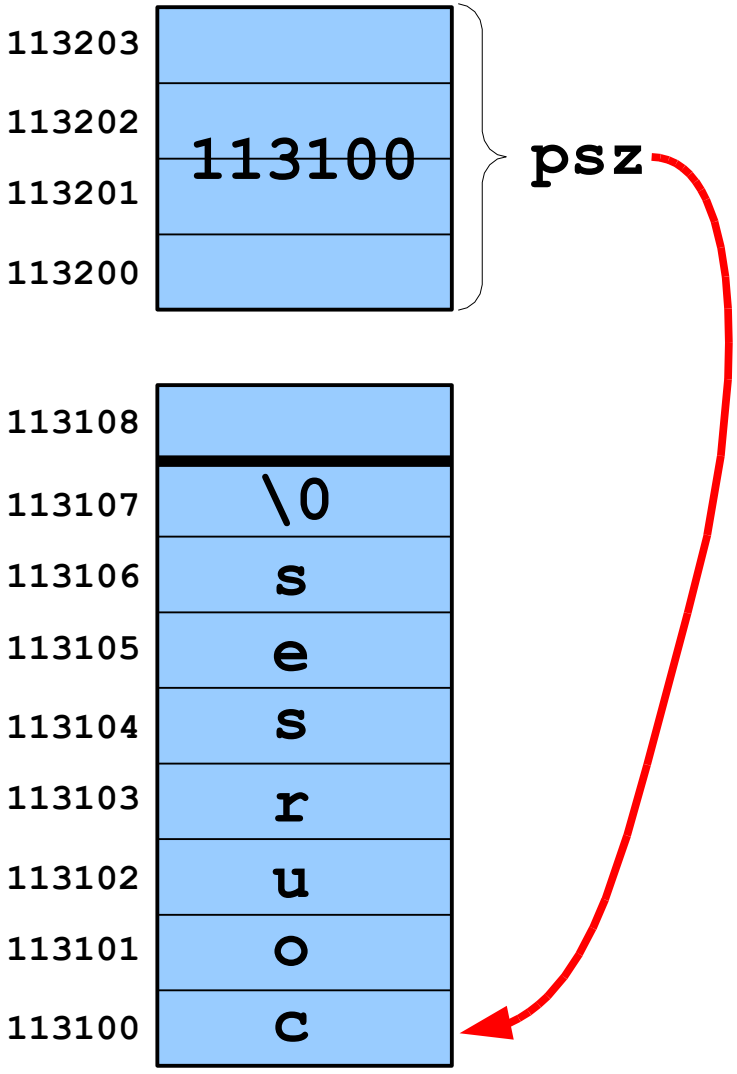


Znowu łańcuchy

```
char sz[] = "course";
char* psz = "courses";
```

- **sz** jest tablicą znaków
- **psz** jest wskaźnikiem do tablicy znaków



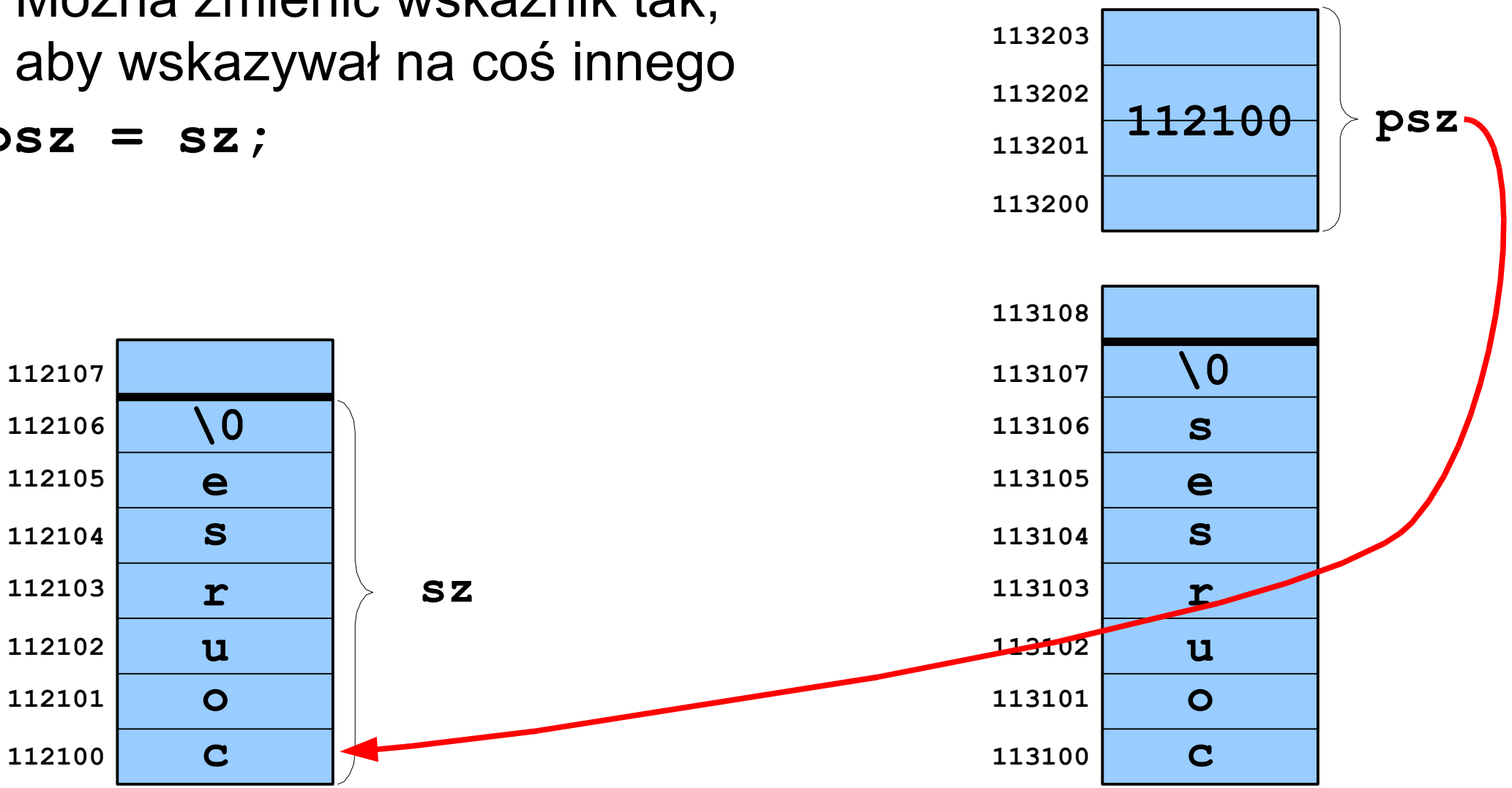


- **sizeof(sz)**? **sizeof(psz)** ?

Znowu łańcuchy

- Można zmienić wskaźnik tak, aby wskazywał na coś innego

```
psz = sz;
```



Znowu łańcuchy

- Można modyfkować znaki w **sz**

```
sz[0] = 'C'; /* Okay */
```

- Nie można zmieniać znaków w **psz**

```
psz[0] = 'C'; /* Danger, Will Robinson! */
```

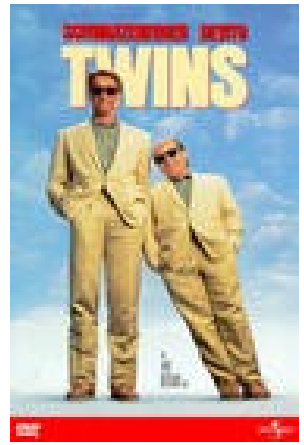
- Spod **psz** można tylko czytać. Dlaczego?
  - Wskazuje na literał łańcuchowy ("courses") który zwykle jest umieszczony w pamięci tylko do odczytu
  - Efekt wykonania operacji **psz[0] = 'C'** jest niezdefiniowany
  - W systemie Linux powoduje błąd ochrony pamięci
  - słowo kluczowe **const** służy do zapobieżenia niepożądanym modyfikacjom

```
const char* psz = "courses";
```

  - Można również użyć **const** dla **sz**

```
const char sz[] = "course";
```

Podobieństwa `[]` i `*`



- Deklaracja z `stdio.h`:

```
size_t strlen(const char*);
```

- Można przekazać tablicę wszędzie tam, gdzie spodziewany jest wskaźnik

```
strlen(sz);
strlen(psz);
strlen("Greetings!");
```

- Równoważna deklaracja:

```
size_t strlen(const char[]);
```

  - Nazwy tablic są niejawnie konwertowane na wskaźnik do ich pierwszego elementu
  - `strlen(sz);` ⇒ `strlen(&sz[0]);`
  - `psz = sz;` ⇒ `psz = &sz[0];`

Podobieństwa `[]` i `*`

- Z tablicą można zrobić dwie rzeczy
  - Określić jej rozmiar z użyciem **`sizeof()`**
  - Uzyskać wskaźnik do jej pierwszego elementu
- Wszystkie inne operacje są wykonywane przy pomocy wskaźników
- Operacje na indeksach są automatycznie konwertowane na operacje na wskaźnikach

Arytmetyka na wskaźnikach

- Można użyć indeksowania w celu uzyskania dostępu do elementów: **`p[i]`**

```
char sz[] = "course";
char* psz = sz;
sz[0]  = 'C';          ⇒ "Course"
psz[5] = 'E';          ⇒ "CoursE"
```

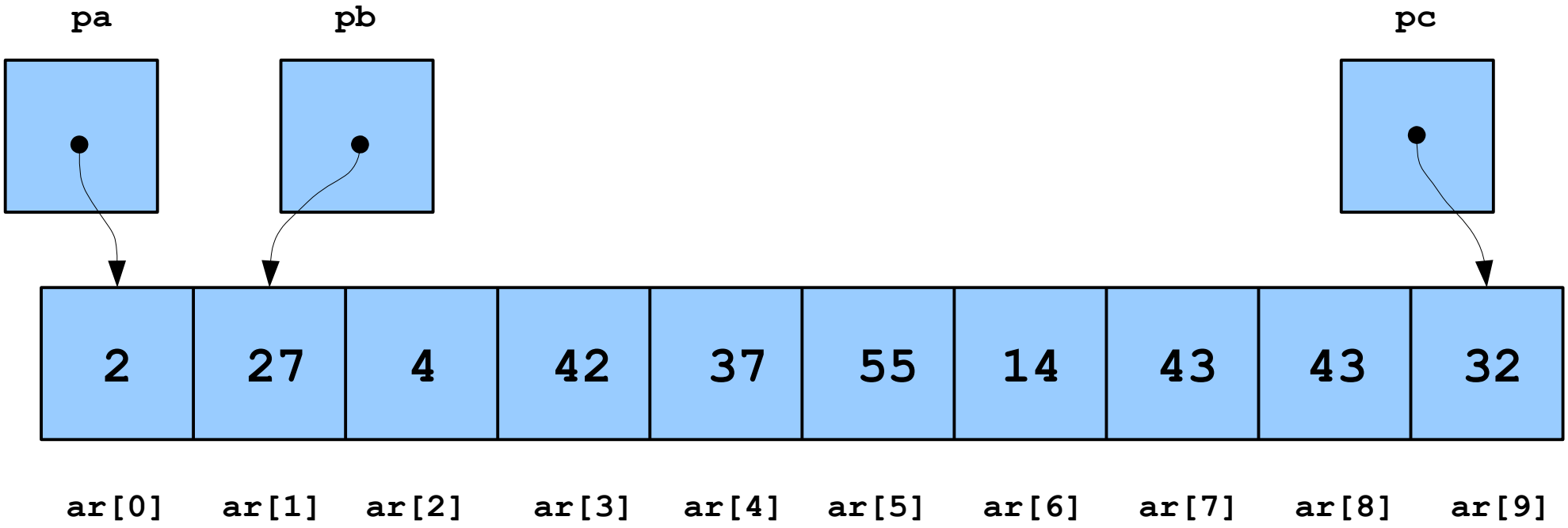
- Można również użyć arytmetyki na wskaźnikach: **`*(p + i)`**

```
*(sz + 0)  = 'C';
*(psz + 5) = 'E';
```

Arytmetyka na wskaźnikach

- Działa to również dla "większych" typów

```
int ar[10];
int *pa, *pb, *pc;
pa = &ar[0] /* same as pa = ar */
pb = pa + 1; pc = pa + 9;
ar[3]=42; *(ar+4)=37; *pa=2; *pb=27; *pc=32;
```



Arytmetyka na wskaźnikach

- Do wskaźników można dodawać i odejmować liczby całkowite - w wyniku uzyskujemy wskaźnik do innego elementu tego samego typu

```
int *pa; char *s;
```

`s-1` $\Rightarrow$ wskazuje na **`char`** przed **`s`** (odjęte 1)

`pa+1` $\Rightarrow$ wskazuje na następny **`int`** (dodane 4!)

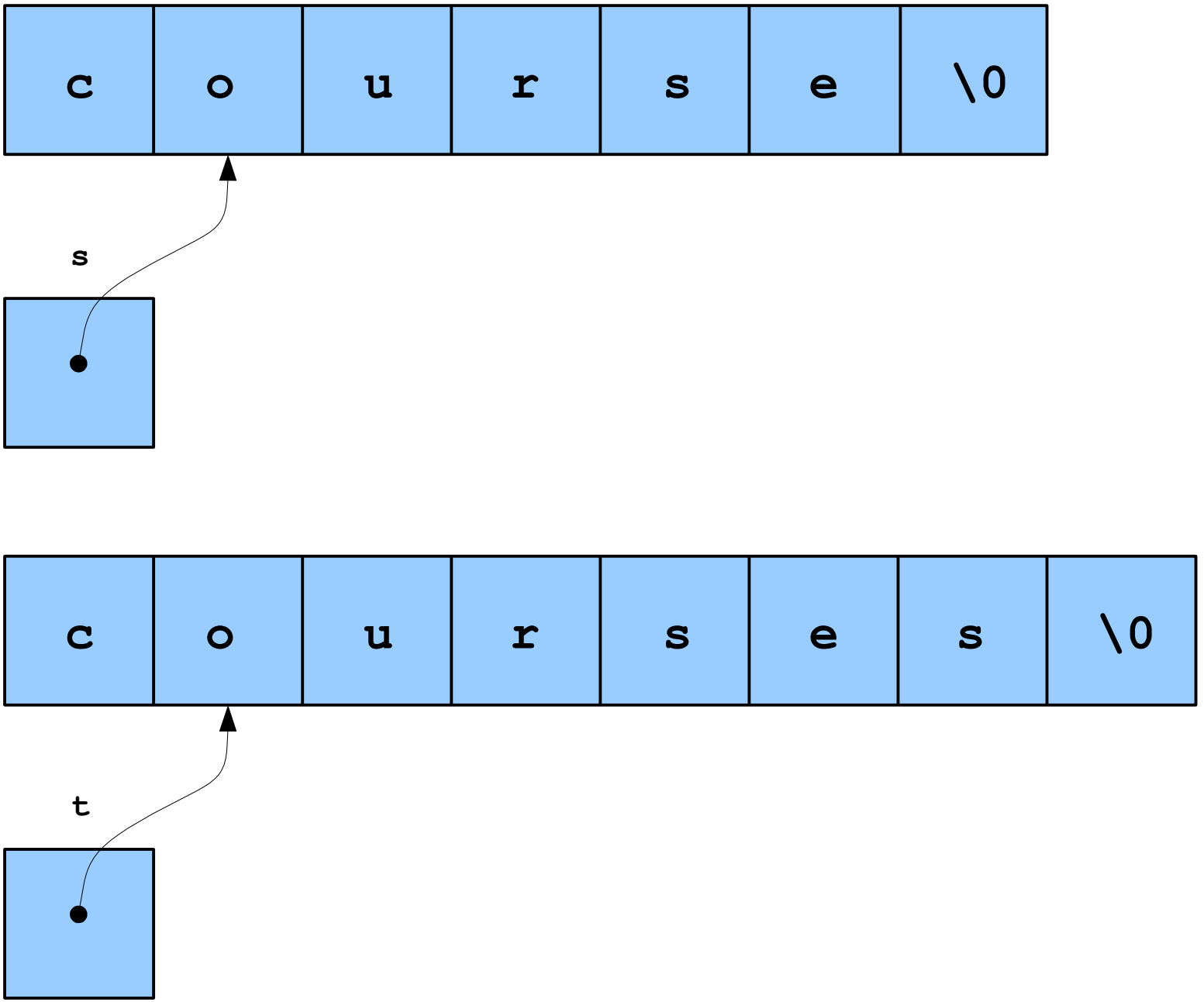
`s+9` $\Rightarrow$ wskazuje na 9. **`char`** za **`s`** (dodane 9)

`++pa` $\Rightarrow$ zwiększa **`pa`** tak, że wskazuje na kolejny **`int`**

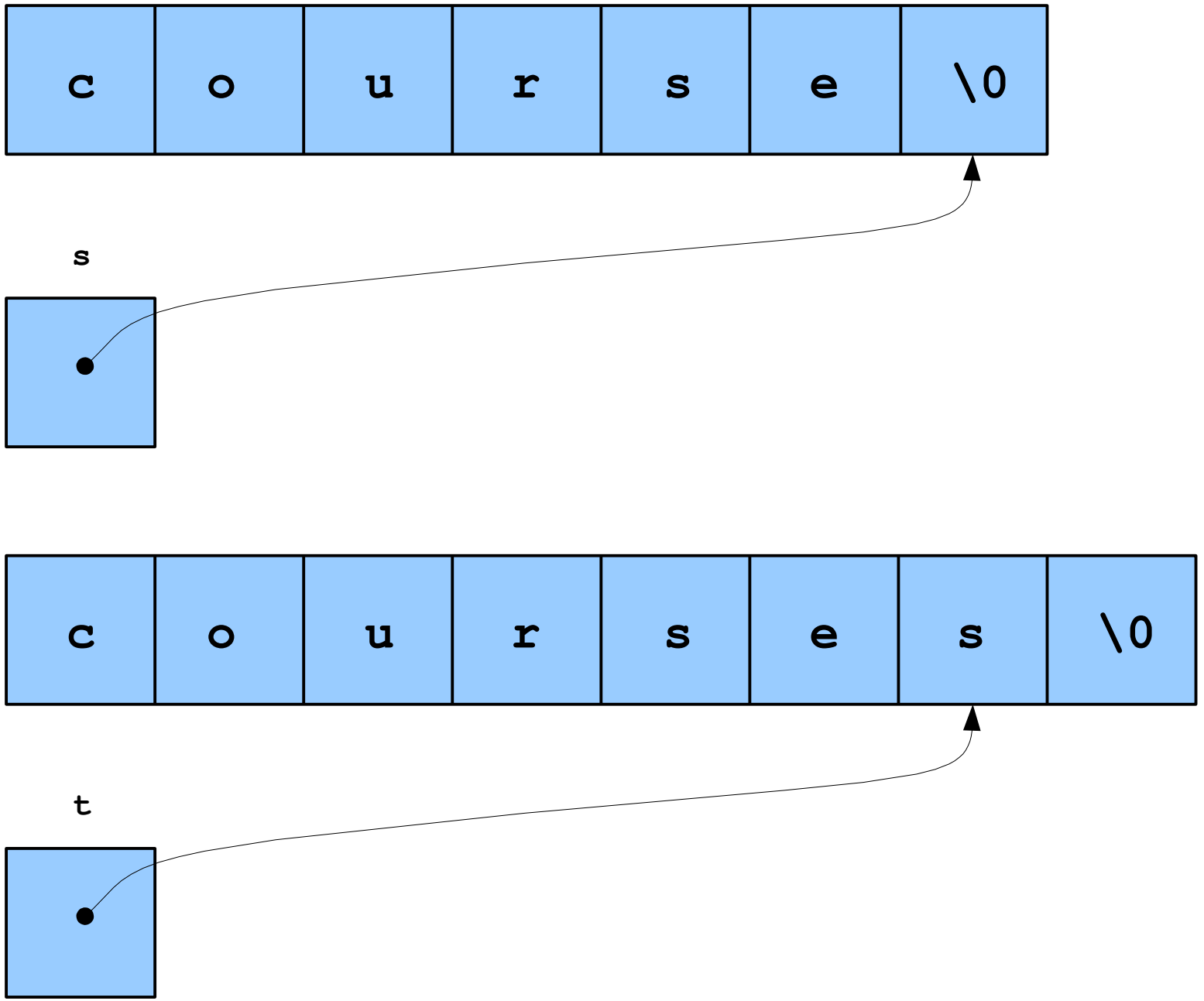
Przykład - porównanie łańcuchów

```
#define MAXLEN 100
int strcmp(char *p1, char *p2);
/*exactly the same as int strcmp(char p1[], char p2[]);*/
int getline(char *line, int max);
/*exactly the same as int getline(char line [], int max);*/

int main() {
    char a[MAXLEN], b[MAXLEN];
    int comp;
    getline(a, MAXLEN);
    getline(b, MAXLEN);
    comp = strcmp(a, b);
    if(comp > 0)
            printf("First line is greater than second\n");
    else if (comp < 0)
            printf("First line is less than second\n");
    else printf("These lines are equal!\n");
    return 0
}
```

Przykład - porównanie łańcuchów



Przykład - porównanie łańcuchów



Porównanie porównań łańcuchów

```
/* pointer version */
/* strcmp: return <0 if s<t,
   0 if s==t, >0 if s>t */
int strcmp(char *s, char *t)
{
   for( ; *s == *t; s++, t++)
      if(*s == '\0')
         return 0;
   return *s - *t;
}
```

```
/* array version */
/* strcmp: return <0 if s<t,
   0 if s==t, >0 if s>t */
int strcmp(char *s, char *t)
{
   int i;

   for(i = 0 ; s[i] == t[i]; i++)
      if(s[i] == '\0')
         return 0;
   return s[i] - t[i];
}
```

Odejmowanie wskaźników

- Można również wykonac odejmowanie wskaźników

```
size_t strlen(const char* s)
{
        const char* p = s;
        while(*p != '\0')
            ++p;
        return p - s;
}
```

Wskaźnik pusty `NULL`

- 0 jest specjalną wartością którą można przypisać do wskaźnika w celu zaznaczenia, że nie wskazuje na żaden obiekt
  - najczęściej używamy stałej symbolicznej **`NULL`**
- Standard gwarantuje, że żaden prawidłowy adres nie jest równy 0
- Wewnętrzna reprezentacja wskaźnika pustego nie musi zawierać samych zer
  - zwykle zawiera
  - zależy to od architektury procesora
- Na wielu systemach próba wyłuskania wskaźnika pustego powoduje błąd ochrony pamięci

Pułapki związane ze wskaźnikami

- Wskaźnik pusty to nie to samo, co pusty łańcuch

```
const char* psz1 = 0;
const char* psz2 = "";
assert(psz1 != psz2);
```

- Przed wyłuskaniem wskaźnika upewnij się, że nie jest to wskaźnik pusty

```
if (psz1)
        /* use psz1 */
```

- **`sizeof(psz1)`** nie podaje liczby elementów w tablicy, na którą wskazuje **`psz1`**. Do tego celu potrzeba dodatkowej zmiennej.

Błąd wiszącego odwołania

- Nie zwracaj wskaźnika do automatycznej zmiennej lokalnej
- Zmienne te przestają istnieć po powrocie z funkcji

```
#include <ctype.h> /* toupper */

const char* upcase(const char* s)
{
            char szBuf[100];
            char* p = szBuf;

            while(*p++ = toupper(*s++))
                    /* empty */ ;
            return szBuf;
}
```

Błąd wiszącego odwołania - kiepskie rozwiązanie

- Zmienne statyczne istnieją dalej po powrocie z funkcji
- To rozwiązanie wprowadza inne problemy

```
#include <ctype.h> /* toupper */
#include <stdio.h> /* printf */

const char* upcase(const char* s)
{
            static char szBuf[100];
            char* p = szBuf;

            while(*p++ = toupper(*s++))
                    /* empty */ ;
            return szBuf;
}

int main()
{
            printf("%s %s\n", upcase("ala"), upcase("ola"));
            return 0;
}
```

Wskaźniki do funkcji

- Funkcja nie jest zmienną
- Natomiast można pobrać adres funkcji
  - Przechować go w zmiennej
  - Przechować go w tablicy (np. emulacja vtable z C++)
  - Przekazać go jako parametr do innej funkcji (np. **`bsearch`**, **`qsort`**). Rozwiązanie znane jako "callback".
  - Zwrócić go z funkcji (np. **`GetProcAddress()`** w systemie Windows)
- Uniezależnia to kod od nazwy funkcji i jej implementacji.

Zmienne przechowujące wskaźniki do funkcji

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

int main()
{
    float radians = 0, result = 0;
    char op = '\0';

    double (*pfn)(double) = 0; /* pfn is a function ptr variable */

    printf("Enter radians, a space, then s, c, or t: ");
    scanf("%g %c", &radians, &op);

    if (op == 's')      pfn = &sin;
    else if (op == 'c') pfn = &cos;
    else if (op == 't') pfn = &tan;

    result = (*pfn)(radians); /* Call function that pfn points to*/
    printf("Result=%g\n", result);
    return 0;
}
```

typedef dla wskaźnika do funkcji

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

typedef double (*PFN_OPER)(double);

int main()
{
    float radians = 0, result = 0;
    char op = '\0';

    PFN_OPER pfn = 0; /* pfn is a function ptr variable */

    printf("Enter radians, a space, then s, c, or t: ");
    scanf("%g %c", &radians, &op);

    if (op == 's')      pfn = &sin;
    else if (op == 'c') pfn = &cos;
    else if (op == 't') pfn = &tan;

    result = (*pfn)(radians); /* Call function that pfn points to*/
    printf("Result=%g\n", result);
    return 0;
}
```

Skomplikowane deklaracje wskaźników

- **`char** argv`**
  - wskaźnik do wskaźnika do znaku
- **`int *x[15]`**
  - tablica 15 wskaźników do liczb całkowitych
- **`int (*x)[15]`**
  - wskaźnik do tablicy 15 liczb całkowitych

Skomplikowane deklaracje

- **`int *f()`**
  - funkcja zwracająca wskaźnik do **`int`**
- **`int (*f)()`**
  - wskaźnik do funkcji zwracającej **`int`**
- **`void *f()`**
  - funkcja zwracająca wskaźnik **`void`** (wskaźnik do nieznanego typu)
- **`void (*f)()`**
  - wskaźnik do funkcji zwracającej **`void`**
- program **`cdecl`** może tu pomóc

Argumenty wywołania programu

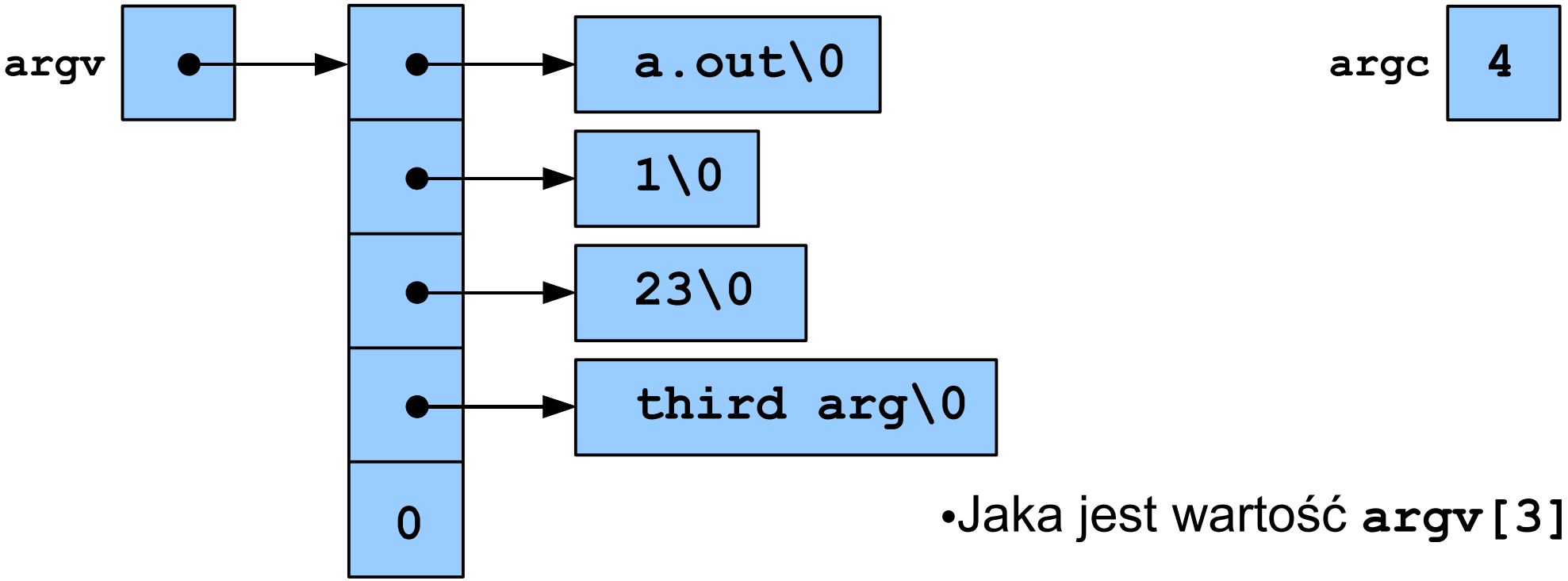
- Pełna deklaracja funkcji **`main`** ma postać

```
int main(int argc, char* argv[]);
```

- **`argv`** przechowuje argumenty wywołania programu, **`argc`** ich liczbę

```
$ a.out 1 23 "third arg"
```

- **`argv`** jest wskaźnikiem do tablicy łańcuchów



- Jaka jest wartość **`argv[3][3]`**?

Argumenty wywołania programu

- Wypisanie argumentów wywołania programu:

```
#include <stdio.h>
int main(int argc, char* argv[])
{
        int i;
        for(i = 1; i < argc; ++i)
                printf("%d: %s\n", i, argv[i]);
        return 0;
}
```