1. Wprowadzenie

W ramach kolejnej części kursu z programowania w języku C++ zapoznamy się z pojęciem *wskaźników*. Wskaźniki są bardzo ważnym mechanizmem pojawiającym się w wielu językach programowania, umożliwiającym między innymi sprawną obsługę tzw. zmiennych dynamicznych czy praktyczną realizację dynamicznych struktur danych, o których będziemy mówili w dalszej części kursu. W niniejszym temacie zajmiemy się następującymi zagadnieniami:

- definicjami zmiennych wskaźnikowych w języku C++,
- przykładami zastosowania zmiennych wskaźnikowych oraz ich interpretacją, zarówno tą nisko jak i wysokopoziomową,
- definicjami i przykładami zastosowania operatorów wyłuskania, adresu i rzutowania.

Zapoznanie się z mechanizmem wskaźników pozwala na bardziej pełne zrozumienie sposobu niskopoziomowego działania programów komputerowych (aplikacji). To z kolei znacznie ułatwia efektywną implementację algorytmów w sposób uwzględniający ograniczenia fizyczne działania systemów komputerowych.

2. Zmienne wskaźnikowe

Oprócz omówionych już wcześniej wbudowanych typów danych prostych oraz strukturalnych, język C++ oferuje jeszcze jeden dodatkowy typ danych, tzw. *typ wskaźnikowy*. Zmienne typu wskaźnikowego, zwane krótko *wskaźnikami*, pozwalają na odwoływanie się do innych zmiennych występujących w programie, w sposób pośredni, poprzez tzw. wskazania. Jak sugeruje sama nazwa, wskaźniki służą do wskazywania innych zmiennych obecnych w programie i pozwalają na odwoływanie się do nich, tzn. odczyt lub zapis ich wartości, niebezpośrednio – poprzez wskazania. Podajmy najpierw formalną składnię definicji zmiennej wskaźnikowej w języku C++, jest ona następująca:

```
1 typ *nazwa_zmiennej_wskaźnikowej [ = wartość_początkowa ];
```

Listing 1. *Składnia definicji zmiennej wskaźnikowej w języku C++.*

Definicja zmiennej wskaźnikowej w języku C++ zawiera typ danych *zmiennych wskazywanych*, tzn. takich, które będą potencjalnie wskazywane przez zmienną wskaźnikową w trakcie realizacji programu oraz nazwę samej zmiennej wskaźnikowej. Podobnie jak podczas deklaracji zmiennych innych typów danych, zmienną wskaźnikową można też zainicjować wartością początkową, ten aspekt deklaracji zmiennych wskaźnikowych zostanie dokładnie omówiony w późniejszej części niniejszego kursu. Warto w tej chwili wyraźnie stwierdzić, że z formalnego punktu widzenia, typem zmiennej wskaźnikowej jest wyrażenie: **`typ *`**, a więc pomimo, że często operator definicji wskaźnika **`*`** jest oddzielony znakiem spacji od nazwy typu zmiennej wskazywanej, to jest on immanentną częścią definicji typu danych zmiennej wskaźnikowej w języku C++. Podajmy kilka przykładów deklaracji zmiennych wskaźnikowych.

```
1 int *px, *py = NULL;
2 char *pc;
3 double *pd = NULL;
```

Listing 2. *Przykładowe deklaracje zmiennych wskaźnikowych w języku C++.*

W linii 1 Listingu 2 zdefiniowane zostały dwie zmienne wskaźnikowe typu **`int *`**, będące wskaźnikami, które potencjalnie będą mogły wskazywać w trakcie realizacji programu na inne zmienne, których typem danych jest typ **`int`**. Innymi słowy, jeśli w trakcie realizacji programu zajdzie potrzeba ustawienia wartości jednego z tych wskaźników, tak aby wskazywał inną zmienną, to zmienna ta **musi być zmienną typu całkowitego**, tzn. typu **`int`**. Ponadto, druga ze zdefiniowanych w linii 1 zmiennych została ustawiona na wartość **`NULL`**, tzn. faktycznie została jej nadana wartość początkowa zero. Literał **`NULL`** (*ang. pusty*) nie jest częścią wbudowaną składni języka C++. Jest on zdefiniowany w bibliotece standardowej **`<cstdlib>`** jako tzw. wskaźnik pusty, to znaczy wskaźnik o wartości zero. Dokładną interpretację wartości zerowej zmiennej wskaźnikowej podamy w dalszej części niniejszego kursu. W linii 2 Listingu 2 zdefiniowana została zmienna wskaźnikowa typu **`char *`** i, podobnie jak wcześniej, jest to wskaźnik, który potencjalnie w trakcie realizacji programu będzie wskazywał na inne zmienne, o ile typem tych zmiennych będzie typ znakowy **`char`**. W linii 3 Listingu 2 został zdefiniowany

wskaźnik do zmiennych typu **`double`** i został on zdefiniowany wstępnie jako wskaźnik pusty. Warto na koniec dodać, że często w języku C++ używa się konwencji nazewnictwa zmiennych wskaźnikowych poprzez rozpoczęcie ich nazw od litery „p" od angielskiego określenia *pointer*, czyli właśnie wskaźnik. Tak też jest w przypadku zmiennych zdefiniowanych na Listingu 2, gdzie nazwy wszystkich zdefiniowanych zmiennych rozpoczynają się właśnie od tej litery. Jest to jedynie konwencja, która nie musi być bezwzględnie stosowana (i rzeczywiście nie zawsze będziemy się jej trzymać w dalszych przykładach), ale jej przestrzeganie znacznie ułatwia przyszłą analizę kodów źródłowych programów zawierających wskaźniki, ponieważ często zdarza się że wskaźnikowy charakter zmiennej nie zawsze może być w prosty sposób wydedukowany tylko na podstawie kontekstu jej użycia w analizowanym wyrażeniu języka C++, w którym ona występuje. Użycie litery „p" na początku nazwy zmiennej wskaźnikowej natychmiast sugeruje, że mamy do czynienia ze wskaźnikiem, bez konieczności sprawdzania definicji tej zmiennej, która może znajdować się w odległej, od analizowanej aktualnie, części kodu źródłowego programu.

3. Wskaźniki – interpretacja niskopoziomowa

Jak wspomnieliśmy w poprzednim rozdziale wskaźniki, jak sugeruje sama nazwa, są zmiennymi służącymi do wskazywania innych zmiennych. Choć początkowo może wydawać się, że wprowadzenie wskaźników do zasobu funkcjonalności języka C++ jest dyskusyjne i, być może, komplikuje jedynie konstrukcję programów pisanych w tym języku, to jednak okazuje się, że są one nieodzownym elementem wielu programów, pozwalającym na wyjątkowo efektywne wykorzystanie struktur danych, na których operują tworzone algorytmy, zarówno w sensie szybkości ich działania jak i wykorzystania dostępnej pamięci. Postaramy się udowodnić tę tezę w dalszej części niniejszego kursu. Aby naprawdę dobrze zrozumieć istotę wskaźników warto przyjrzeć się nie tylko logicznemu czy formalnemu aspektowi ich funkcjonowania w języku C++, ale także uświadomić sobie jak działają one na poziomie fizycznym, na poziomie maszynowym systemu komputerowego. Takie spojrzenie pozwala wyzbyć się wszelkich wątpliwości i niedopowiedzeń związanych z ich działaniem. Poniżej posłużymy się dość prostym przykładem, który pokazuje jak można interpretować pojęcie wskaźnika, zarówno na poziomie logicznym działania programu w języku C++ jak i z niskopoziomowego punktu widzenia. Przejdźmy zatem do przedstawienia wspomnianego przykładu.

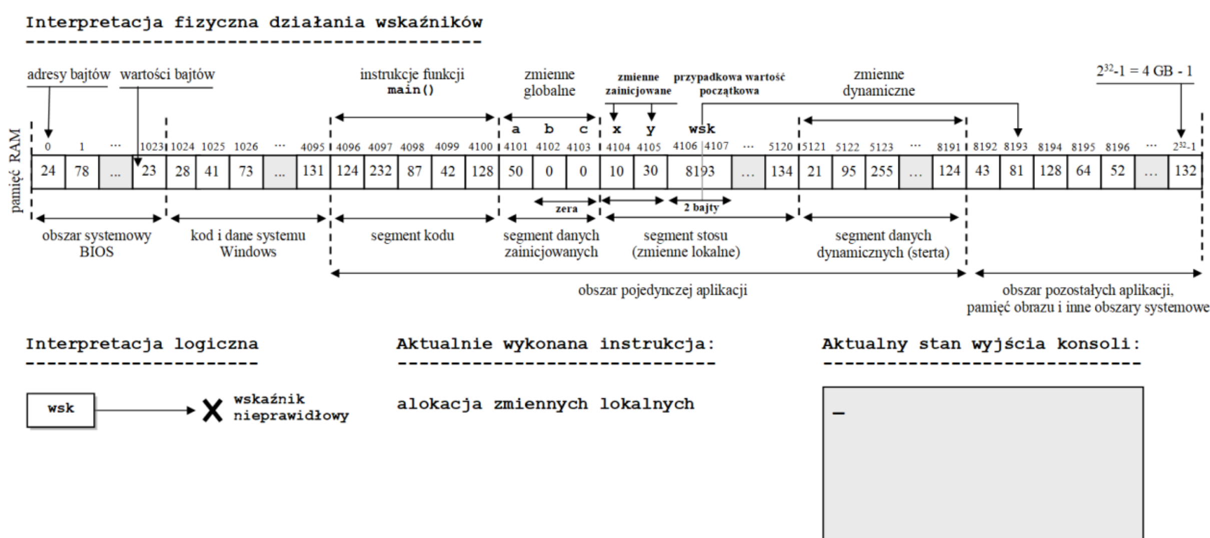
```
 1 #include <iostream>
 2 using namespace std;
 3
 4 int a = 50, b, c;                      // zmienne globalne
 5
 6 int main()                             // funkcja główna
 7 {
 8  int x = 10, y = 30;                   // zmienne lokalne
 9  int *wsk;                             // deklaracja zmiennej wskaźnikowej
10  wsk = &x;                             // & - operator adresu
11  cout << "*wsk = " << *wsk << endl;    // * - operator wyłuskania (odczyt)
12  *wsk = 20;                            // * - operator wyłuskania (zapis)
13  cout << " x   = " <<  x   << endl;    // wyświetlenie wart. zmiennej x
14  cout << " wsk = " <<  wsk << endl;    // wyświetlenie wart. zmiennej wsk
15  wsk = &y;                             // & - operator adresu
16  cout << "*wsk = " << *wsk << endl;    // * - operator wyłuskania (odczyt)
17  *wsk = 40;                            // * - operator wyłuskania (zapis)
18  cout << " y   = " <<  y   << endl;    // wyświetlenie wart. zmiennej y
19  cout << " wsk = " <<  wsk << endl;    // wyświetlenie wart. zmiennej wsk
20  wsk = (int *)1;                       // (typ) - operator rzutowania
21  *wsk = 60;                            // ustawienie wartości drugiego
22  return 0;                             // bajtu pamięci RAM komputera
23 }                                      // w systemie Windows - błąd
```

Listing 3. *Program demonstrujący działanie wskaźników w języku C++.*

W programie z Listingu 3 znajdują się deklaracje trzech *zmiennych globalnych*, **`a`**, **`b`** oraz **`c`**. Zmienne te są tworzone w momencie rozpoczęcia działania programu oraz usuwane z pamięci RAM w momencie jego zakończenia. Są one **zmiennymi wspólnymi** i dostępnymi dla wszystkich funkcji zdefiniowanych w programie. Jeśli jedna z funkcja dokona zapisu wartości zmie-

nnej globalnej w pewnym momencie działania programu, to zmiana ta będzie „widoczna” dla pozostałych funkcji zdefiniowanych w naszym programie. Zupełnie inaczej zachowują się *zmienne lokalne*, tzn. zmienne zdefiniowane wewnątrz funkcji obecnych w naszym programie. W programie przykładowym są to zmienne **`x`**, **`y`** oraz **`wsk`**, zdefiniowane w funkcji głównej **`main`**. Są one tworzone w momencie rozpoczęcia działania danej funkcji i usuwane z pamięci RAM w momencie jej zakończenia. Nie są one dostępne do odczytu czy zapisu (nie są „widoczne”) z poziomu kodu innych funkcji zdefiniowanych w naszym programie, a jedynie lokalnie, wewnątrz funkcji w której się znajdują – stąd ich nazwa. To rozróżnianie, oprócz aspektów wymienionych przed chwilą, ma także znaczenie z punktu widzenia fizycznego rozmieszczenia poszczególnych części naszego programu w pamięci RAM komputera po jego kompilacji i uruchomieniu przez system operacyjny.

Dla uproszczenia dalszego opisu przyjmijmy, że nasz program został skompilowany i uruchomiony w systemie Windows na komputerze zgodnym z *architekturą Intel*, tzn. najogólniej mówiąc, na komputerze z procesorem tej firmy (sytuacja jest bardzo podobna w przypadku systemów komputerowych opartych o inne architektury i działających pod kontrolą innych systemów operacyjnych). W takim przypadku, po kompilacji, załadowaniu przez system operacyjny do pamięci RAM i uruchomieniu, struktura programu w pamięci RAM będzie dla typowego scenariusza wyglądała tak[1] jak przedstawiono to na Rys 1.



Rysunek 1. *Obraz pamięci RAM komputera po załadowaniu i uruchomieniu programu przykładowego.*

Bardzo ważne jest uświadomienie sobie faktu, że z punktu widzenia procesora, **cała pamięć RAM komputera traktowana jest jak tablica bajtów**. *Bajt* jest najmniejszą jednostką informacji jaką procesor może odczytać lub zapisać w pamięci RAM i może on przechowywać liczby całkowite z przedziału od 0 do 255 (razem 256 różnych wartości, co wynika z faktu, że bajt składa się z 8-miu bitów – zer lub jedynek, zaś wszystkich możliwych kombinacji 8-miu zer

[1] podany rysunek zawiera wiele uproszczeń i nie należy go traktować w sposób dosłowny, niemniej dobrze oddaje rzeczywistą strukturę rozmieszczenia poszczególnych części programu jak i innych składników systemu komputerowego w pamięci RAM komputera zgodnego z architekturą Intel i pracującego pod kontrolą systemu operacyjnego Windows

i jedynek jest dokładnie $2^8 = 256$). A zatem z punktu widzenia procesora pamięć RAM jest pojedynczą, dużą tablicą liczb całkowitych, z których każda, nazywana *komórką*, posiada swój unikalny indeks zwany *adresem* (podobnie jak zwykła tablica w języku C++, której każdy element posiada swój indeks). Procesor może odwołać się (tzn. zapisać i/lub odczytać) do dowolnej komórki pamięci RAM posługując się jej adresem. Warto zauważyć, że w pamięci RAM w postaci liczbowej przechowywane są nie tylko dane, tzn. np. wartości różnych zmiennych naszego programu, ale również instrukcje procesora, które z jego punktu widzenia też są zwykłymi liczbami całkowitymi, z tą różnicą, że każda z nich reprezentuje kod odpowiedniej operacji (tzn. kod *instrukcji procesora*, takiej np. jak dodawanie, mnożenie czy porównanie) jaką w danym momencie procesor musi wykonać, aby zrealizować kolejny krok przebiegu aktualnie wykonywanego programu. A zatem z punktu widzenia procesora, wszystkie informacje zawarte w pamięci RAM, zarówno instrukcje do wykonania jak i dane, są poindeksowanymi liczbami całkowitymi. W dużym uproszczeniu można powiedzieć, że pojemność pamięci RAM zainstalowanej w naszym komputerze determinuje zarazem adres ostatniego bajtu w niej zawartego, np. na Rys 1. adresem tym jest $2^{32} - 1 = 4$ GB -1 (gigabajty), przy założeniu, że na komputerze zainstalowano 4 GB pamięci RAM (odjęcie liczby jeden od maksymalnego adresu bajtu zawartego w pamięci RAM wynika z faktu, że adresy numerowane są poczynając od liczby zero, podobnie jak ma to miejsce w przypadku indeksacji tablic w języku C++).

Mając na uwadze wszystkie powyższe spostrzeżenia możemy wywnioskować z Rys. 1, że po załadowaniu programu do pamięci RAM będzie on podzielony na tzw. *segmenty*. Każdy z segmentów przeznaczony jest do zapamiętania odrębnych informacji wchodzących w skład programu. I tak, s*egment kodu* zawiera wszystkie instrukcje procesora składające się na kod maszynowy naszego programu, powstały w wyniku jego kompilacji. Są tam zawarte instrukcje procesora realizujące operacje, skompilowane do postaci jego kodu maszynowego, zdefiniowane w kodzie źródłowym naszego programu w ramach wszystkich umieszczonych w nim funkcji. Z kolei *segment danych zainicjowanych* jest fragmentem pamięci RAM naszego programu, przeznaczonym do przechowywania wszystkich zmiennych globalnych, które w nim występują. Kolejnym segmentem jest tzw. *segment stosu* (lub krótko *stos procesora, ang. stack*), który jest obszarem pamięci RAM, przeznaczonym m.in. do przechowywania wartości zmiennych lokalnych aktualnie wykonywanych przez procesor funkcji naszego programu. Wreszcie *segment danych dynamicznych* czyli tzw. *sterta* (*ang. heap*), jest fragmentem pamięci RAM, przydzielonym przez system operacyjny naszemu programowi, służącym do przechowywania wartości tzw. *zmiennych dynamicznych*, o których mówić będziemy dokładnie w dalszej części niniejszego kursu. Tak w dużym uproszczeniu przedstawia się struktura **każdej aplikacji** działającej w systemie komputerowym zgodnym z architekturą Intel, pracującym pod kontrolą systemu operacyjnego Windows.

Oprócz opisanej powyżej struktury, widzimy z Rys. 1, że w pamięci RAM komputera w danym momencie obecne są też inne, istotne z punktu widzenia działania systemu komputerowego, obszary. Np. pierwszy kilobajt, czyli pierwsze $2^{10} = 1024$ bajty, tzn. bajty pamięci RAM o adresach od 0 do 1023, tworzą obszar systemowy, w którym zawarte są krytyczne dane służące systemowi komputerowemu do komunikacji z urządzeniami zewnętrznymi (takimi jak np. klawiatura, drukarka, monitor czy pory USB). Z naszego punktu widzenia określenie „krytyczne” oznacza tyle, że próba np. zmiany wartości bajtów obecnych w tym obszarze pamięci RAM może prowadzić do natychmiastowego zawieszenia się całości systemu komputerowego ze wszystkimi tego konsekwencjami, np. utratą danych nad którymi aktualnie pracowaliśmy. Na szczęście, w systemie Windows, taka próba zawsze zakończy się błędem działania aplikacji

prowadzącym do natychmiastowego jej zamknięcia, gdyż system Windows blokuje wszelkie próby zapisu lub nawet odczytu wartości znajdujących się w tym obszarze pamięci RAM przez aplikacje użytkownika, uniemożliwiając tym samym zakłócenie działania systemu komputerowego błędnie działającym aplikacjom (czy też np. wirusom, świadomie podejmującym dla swych celów, prób zapisu lub odczytu wspomnianego obszaru pamięci RAM). Niemniej, w innych systemach operacyjnych (takich jak np. system operacyjny DOS firmy Microsoft), które nie posiadają żadnych mechanizmów zabezpieczających, twórca aplikacji w języku C++ ma całkowitą swobodę dostępu co całego obszaru pamięci RAM komputera i może tego dokonać właśnie z użyciem wskaźników. Widać tu, że z jednej strony mechanizm wskaźników daje użytkownikowi potencjalnie pełną kontrolę nad działaniem całego systemu komputerowego, z drugiej zaś, z tego samego powodu, uważany jest za mechanizm „niebezpieczny". Z tego też powodu wiele języków programowania nie zawiera w swej składni bezpośredniej implementacji mechanizmu wskaźników, zastępując go innymi, bezpieczniejszymi rozwiązaniami (np. mechanizmem *referencji*, który jest też dostępny w języku C++). Kończąc ten wątek rozważań, warto dodać, że przekonamy się w podanym programie przykładowym, że rzeczywiście, błędy popełnione w naszej aplikacji w związku z nieprawidłowym wykorzystaniem wskaźników, będą powodowały natychmiastową reakcję systemu operacyjnego.

Widzimy też z Rys. 1, że kod samego systemu operacyjnego (tzn. instrukcje procesora będące częścią algorytmów zaimplementowanych w systemie operacyjnym) oraz dane na których pracuje system operacyjny, również znajdują swoje miejsce w pamięci RAM (w naszym przykładzie są to komórki o adresach od 1024 do 4095[2]. Pozostałe obszary tej pamięci zarezerwowane są dla innych aplikacji pracujących aktualnie w systemie – jak wiadomo w systemie Windows w jednym czasie może działać wiele aplikacji, jest on tzw. *systemem wielozadaniowym*, a także do innych celów związanych z funkcjonowaniem systemu komputerowego, tzw. obszary systemowe, takie jak np. obszar pamięci obrazu, z którego karta graficzna pobiera wszelkie dane potrzebne do wygenerowania i wyświetlenia obrazu na monitorze czy wyświetlaczu podłączonym do komputera (w naszej przykładowej mapie pamięci RAM są to obszary rozpoczynające się od adresu 8192 i rozciągające się do końca dostępnej pamięci, zainstalowanej w przykładowym systemie komputerowym).

Mając tak sformułowany ogląd fizycznej struktury składników systemu komputerowego, uwzględniającej ich postać widzianą z niskopoziomowego punktu widzenia procesora i podłączonej do niego pamięci RAM, możemy już w stosunkowo łatwy sposób dokładnie zrozumieć istotę działania mechanizmu wskaźników.

Na Rys. 1 widzimy obraz zawartości pamięci RAM (tzw. *zrzut* lub *mapa* pamięci RAM) jaka byłaby w niej obecna natychmiast po załadowaniu do pamięci przez system operacyjny naszej przykładowej aplikacji z Listingu 3 i uruchomienia przez system funkcji głównej. Segment kodu zawierałby wszystkie instrukcje zdefiniowane w funkcji **main** (adresy od 4096 do 4100). W segmencie zmiennych globalnych pod adresami 4101, 4102 i 4103 znajdowałyby się zmienne globalne, odpowiednio **a**, **b** oraz **c**, zdefiniowane w naszym kodzie źródłowym. Zauważmy, że początkowe wartości niezainicjowanych zmiennych globalnych (tzn. zmiennych **b** oraz **c**) będą zerowe. Zupełnie inaczej wygląda sytuacja dla zmiennych lokalnych **x**, **y** i **wsk**, których wartości początkowe będą przypadkowe, jeśli nie zostały one zainicjowane w ramach swych definicji. W naszym przykładzie zmiennym **x** oraz **y**, będącym zwykłymi zmiennymi typu całkowitego, zostały nadane wartości początkowe (odpowiednio 10 oraz 30), natomiast zmienna wskaźnikowa **wsk** nie została zainicjowana i będzie ona zawierała wartość przypadkową –

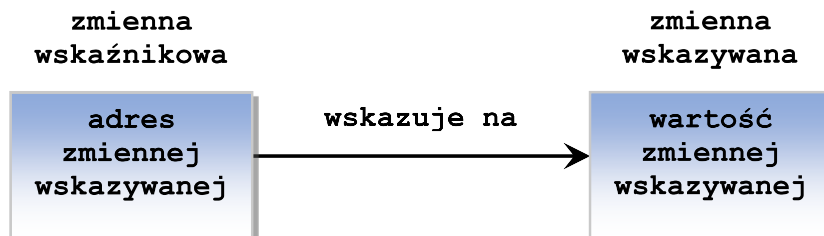
[2] w rzeczywistości jest to obszar zajmujący o wiele większą przestrzeń w pamięci RAM

8193. Przy okazji warto już tutaj zauważyć, że **z niskopoziomowego punktu widzenia wartościami zmiennych wskaźnikowych są również zwykłe liczby całkowite**, to bardzo ważny fakt, który znakomicie ułatwia dalsze zrozumienie działania mechanizmu wskaźników. Należy też dodać, że przypadkowe wartości początkowe niezainicjowanych zmiennych lokalnych (u nas jedyną taką zmienną jest zmienna `wsk`) wynikają z faktu, że instrukcja deklaracji zmiennej (np. instrukcja `int *wsk;`) sprowadza się *de facto* (w uproszczeniu) do polecenia systemowi operacyjnemu rezerwacji odpowiedniej liczby bajtów w pamięci RAM potrzebnych do zapamiętania wartości tej zmiennej (w przypadku naszej zmiennej przykładowej `wsk` będą to 2 bajty, patrz Rys. 1). Skoro tak, to początkowa wartość niezainicjowanej zmiennej lokalnej będzie taka, jaka znajdowała się w rezerwowanym obszarze pamięci RAM bezpośrednio przed momentem jego rezerwacji na skutek utworzenia (deklaracji) rozważanej zmiennej w pamięci RAM (system operacyjny nie „zeruje" rezerwowanego na potrzeby tworzonej zmiennej lokalnej obszaru pamięci RAM). Efekt jest taki, że przy każdym nowym uruchomieniu naszej aplikacji, wartości początkowe zmiennych lokalnych będą wyglądały na przypadkowe, gdyż nie jesteśmy w stanie przewidzieć jaka była zawartość pamięci RAM przed załadowaniem i uruchomieniem naszej aplikacji, ani też stwierdzić do jakiego konkretnie obszaru pamięci RAM nasza aplikacja zostanie załadowana – zależy to wyłącznie od systemu operacyjnego. Z tego powodu początkowa wartość 8193 naszej zmiennej wskaźnikowej `wsk` może być uważana za przypadkową i, zgodnie z interpretacją logiczną wartości wskaźnika (patrz Rys. 1), reprezentuje ona wskaźnik nieprawidłowy, nie związany z żadną inną zmienną naszej aplikacji i wskazujący obszar pamięci RAM znajdujący się poza obszarem przydzielonym naszej aplikacji przez system operacyjny. Zanim przejdziemy do dokładnej analizy naszego programu przykładowego z Listingu 3, warto na koniec dodać, że zarówno na Rys. 1 jak i na pozostałych rysunkach dalszej części rozważanego przykładu, znajdują się pola o następującym znaczeniu:

- **`Interpretacja logiczna`** – logiczna interpretacja wartości wskaźnika `wsk` występującej na danym etapie realizacji programu przykładowego.
- **`Aktualnie wykonana instrukcja`** – instrukcja będąca częścią kodu źródłowego naszej aplikacji przykładowej, bezpośrednio po wykonaniu której możemy zaobserwować stan pamięci RAM podany na wybranym rysunku.
- **`Aktualny stan wyjścia konsoli`** – aktualna zawartość ekranu konsoli naszej aplikacji, obecna bezpośrednio po wykonaniu aktualnej instrukcji rozważanej na wybranym rysunku.

4. Analiza programu przykładowego

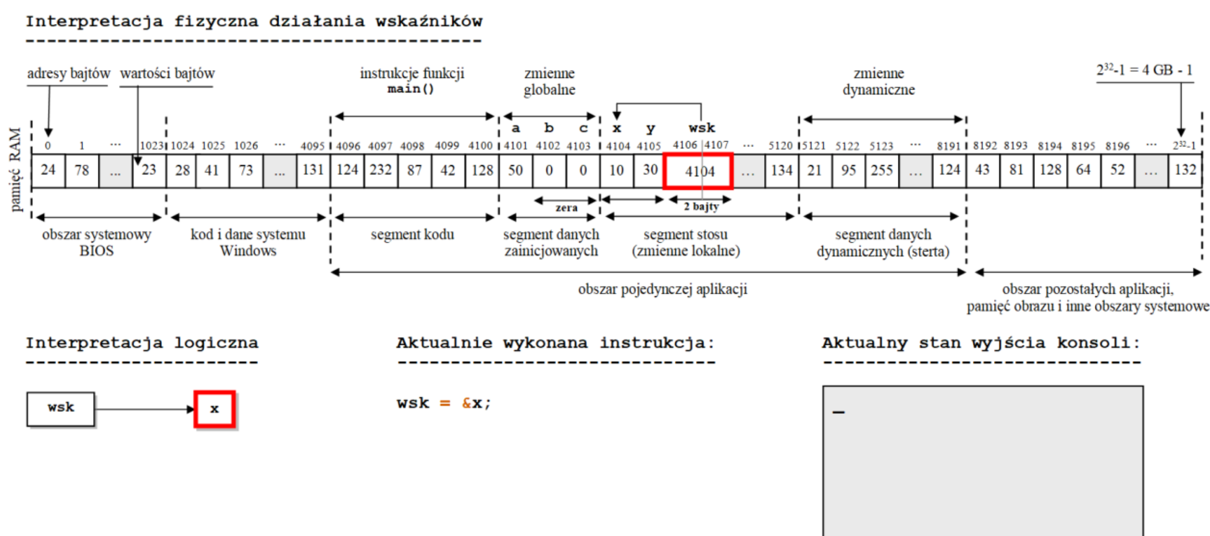
Na początku niniejszego rozdziału przypomnijmy logiczną interpretację wskaźników w języku C++, zgodnie z tym, co zostało powiedziane w Rozdziale 2 można stwierdzić, *że wskaźniki w języku C++ służą do wskazywania innych zmiennych obecnych w programie*. Interpretację tę ilustruje poniższy rysunek.



***Rysunek 2.** Logiczna interpretacja działania zmiennej wskaźnikowej w języku C++.*

Dzięki wskaźnikowi możemy odwołać się do innej zmiennej, na którą w danym momencie wskazuje wskaźnik, tzn. np. odczytać jej wartość, bądź zapisać do niej nową wartość za pośrednictwem wskaźnika. Wartością samej zmiennej wskaźnikowej jest natomiast adres zmiennej wskazywanej w pamięci RAM, na którą wskazuje zmienna wskaźnikowa w danym momencie wykonania programu.

Aby dokładnie zrozumieć powyższe stwierdzenia popatrzmy ponownie na program przykładowy z Listingu 3 oraz mapę pamięci RAM widoczną na Rys. 1, odwzorowującą sytuację jaka ma miejsce zaraz po załadowaniu i uruchomieniu naszej aplikacji przez system operacyjny. Na początku wartość zmiennej wskaźnikowej **`wsk`** jest przypadkowa (patrz poprzedni rozdział). W takiej sytuacji wskaźnik **`wsk`** nie jest związany z żadną inną zmienną obecną w naszym programie i można go uznać jego stan za nieprawidłowy. Zaraz po wykonaniu instrukcji z linii nr 10 Listingu 3, sytuacja ulega zmianie. Nowy stan naszej aplikacji po wykonaniu tej instrukcji zobrazowany jest na Rys. 3.



***Rysunek 3.** Obraz pamięci RAM po wykonaniu instrukcji z linii nr 10 programu przykładowego.*

Instrukcja **`wsk = &x;`** powoduje przypisanie do zmiennej wskaźnikowej **`wsk`** adresu zmiennej całkowitej **`x`**, zdefiniowanej w funkcji **`main`**. Przypisanie to wykonujemy używając w tym celu *operatora adresu* języka C++, oznaczonego symbolem '**`&`**', którego składnia jest następująca:

```
1 &zmienna_wskazywana
```

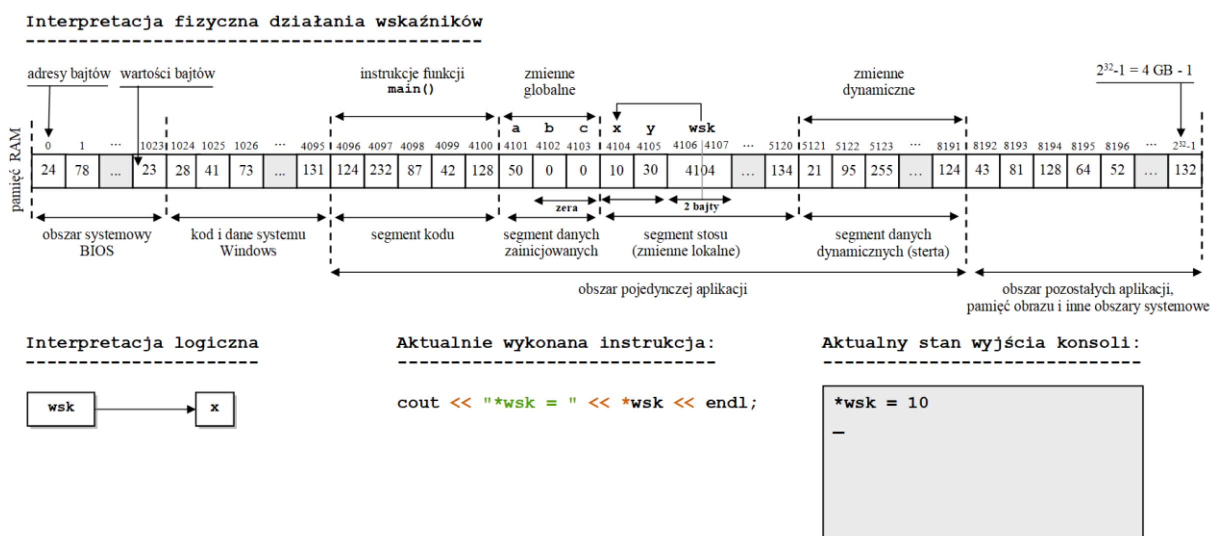
Listing 4. *Składnia operatora adresu języka C++.*

Wartością operatora adresu języka C++ jest adres w pamięci RAM zmiennej stojącej bezpośrednio po jego prawej stronie. Zauważmy (patrz Rys. 3), że teraz wartość wskaźnika **`wsk`** uległa zmianie (co zostało symbolicznie podkreślone kolorem czerwonym) i wynosi obecnie 4104, co równe jest adresowi zmiennej **`x`** w pamięci RAM, z którą od tego momentu związany jest wskaźnik **`wsk`**. Z logicznego punktu widzenia, wartość wskaźnika **`wsk`**, równa adresowi zmiennej **`x`**, powoduje, że wskaźnik ten wskazuje teraz na zmienną **`x`** (co symbolizuje pole **`Interpretacja logiczna`** na Rys. 3) i będzie można od tej pory odwoływać się do tej zmiennej używając wskaźnika **`wsk`**. Użyjmy zatem tego wskaźnika w celu wyświetlenia na ekranie konsoli wartości przechowywanej przez zmienną **`x`**. Aby to zrobić użyjemy kolejnego operatora języka C++, zwanego *operatorem wyłuskania*, którego składnia w języku C++ jest następująca:

```
1 *zmienna_wskaźnikowa
```

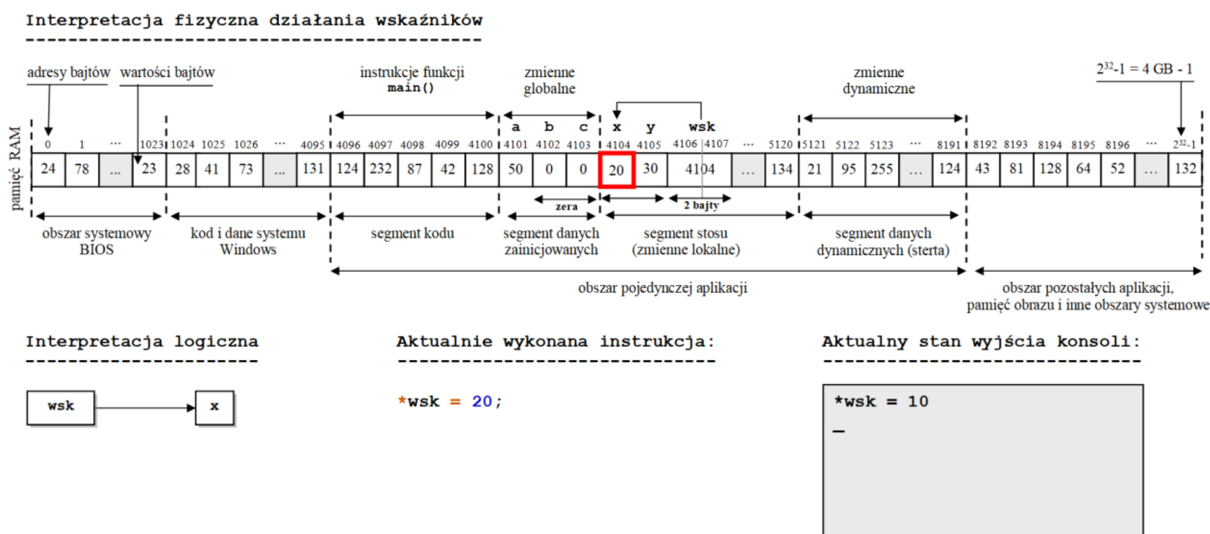
Listing 5. *Składnia operatora wyłuskania języka C++.*

Wartością operatora wyłuskania języka C++, oznaczonego symbolem '**`*`**', zastosowanego na rzecz zmiennej wskaźnikowej stojącej po jego prawej stronie jest wartość zmiennej wskazywanej przez tę zmienną wskaźnikową w danym momencie wykonania programu. W naszym przykładzie, ponieważ wskaźnik **`wsk`** wskazuje obecnie na zmienną **`x`**, to wartością operatora wyłuskania zastosowanego na rzecz wskaźnika **`wsk`** będzie wartość zmiennej **`x`**, a więc 10. Taka też wartość zostanie wyświetlona w oknie konsoli. Sytuację tę obrazuje poniższy Rys. 4.



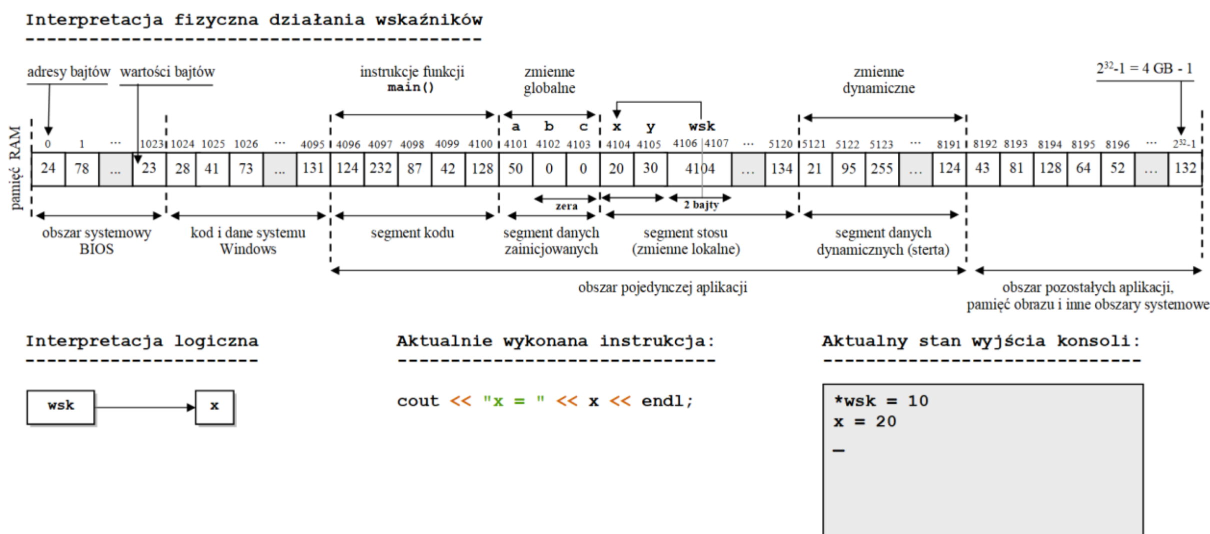
Rysunek 4. *Pamięć RAM i stan konsoli po wykonaniu instrukcji z linii nr 11 programu przykładowego.*

Operator wyłuskania służy też do zapisu wartości zmiennej wskazywanej za pomocą wskaźnika. Instrukcja **`*wsk = 20;`** w linii 12 programu przykładowego spowoduje zapis wartości 20 do zmiennej wskazywanej przez wskaźnik **`wsk`**, a więc do zmiennej **`x`**. Sytuację tę obrazuje poniższy rysunek.



Rysunek 5. *Pamięć RAM i stan konsoli po wykonaniu instrukcji z linii nr 12 programu przykładowego.*

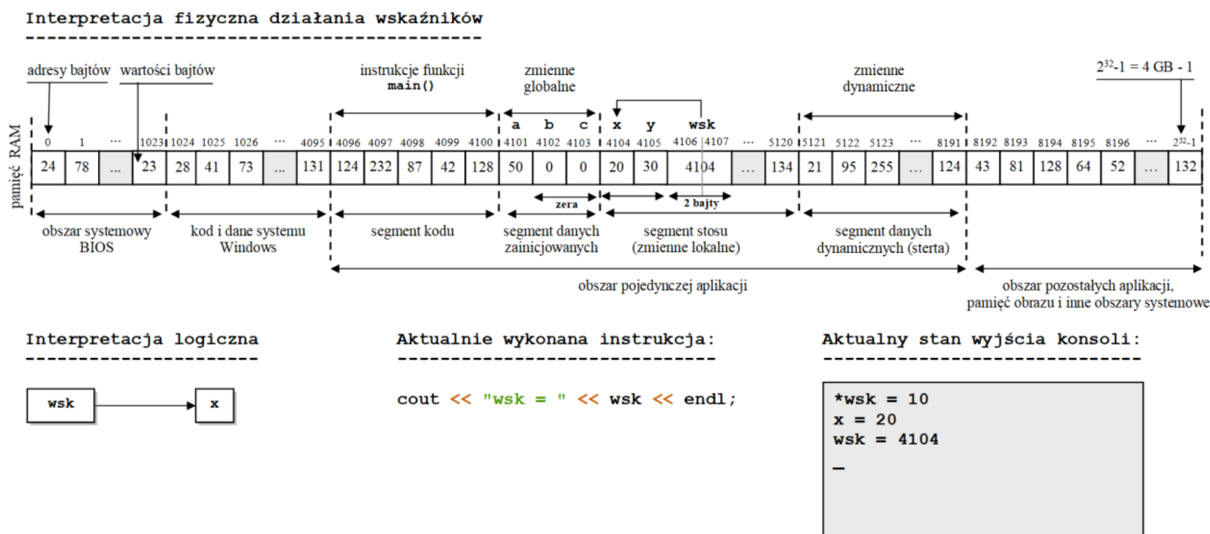
Aby przekonać się, że rzeczywiście wartość zmiennej **`x`** została zmieniona z użyciem wskaźnika **`wsk`**, w linii 13 programu przykładowego wyświetlamy aktualną wartość zmiennej **`x`**. Po wykonaniu tej instrukcji stan naszej aplikacji będzie wyglądał następująco.



Rysunek 6. *Pamięć RAM i stan konsoli po wykonaniu instrukcji z linii nr 13 programu przykładowego.*

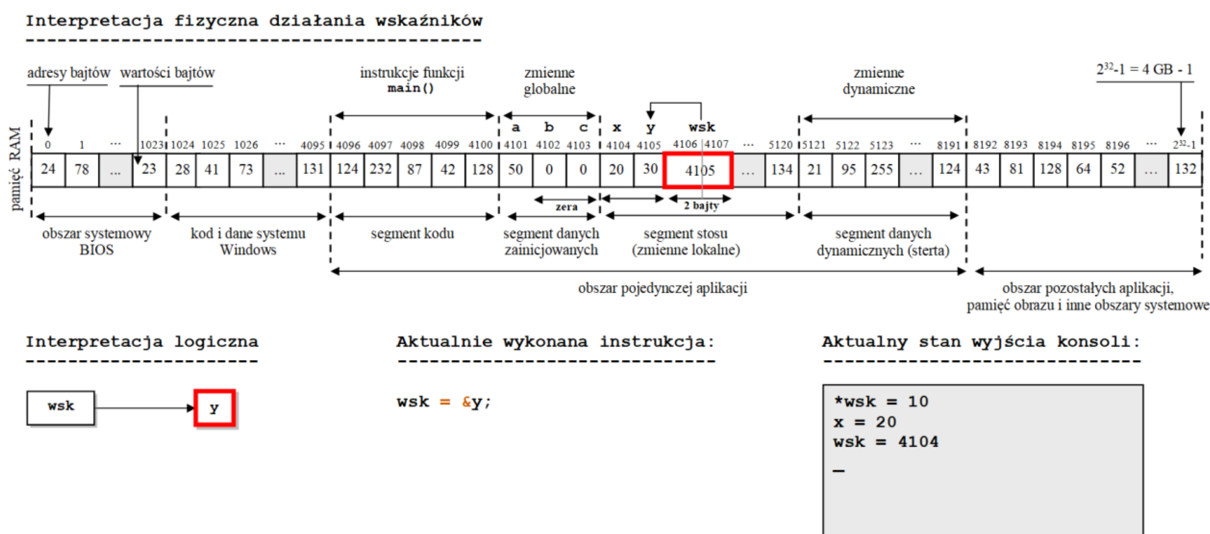
Na ekranie konsoli pojawił się teraz komunikat potwierdzający, że wartość zmiennej **`x`** rzeczywiście została zmodyfikowana za pomocą wskaźnika **`wsk`**.

Zauważmy teraz, że wartością samej zmiennej wskaźnikowej **`wsk`** jest także liczba całkowita – adres zmiennej wskazywanej – a zatem wskaźniki jako takie, mogą być traktowanie w języku C++ jak zwykłe liczby całkowite, podobnie zresztą jak inne zmienne typów wbudowanych. Oznacza to, że możemy np. dodawać zmienne wskaźnikowe do innych zmiennych (niekoniecznie wskaźnikowych), mnożyć czy porównywać wskaźniki do innych wartości całkowitych. Można też wyświetlić wartość samej zmiennej wskaźnikowej, co odbywa się w linii nr 14 naszego programu przykładowego, spójrzmy na poniższy rysunek.



Rysunek 7. *Pamięć RAM i stan konsoli po wykonaniu instrukcji z linii nr 14 programu przykładowego.*

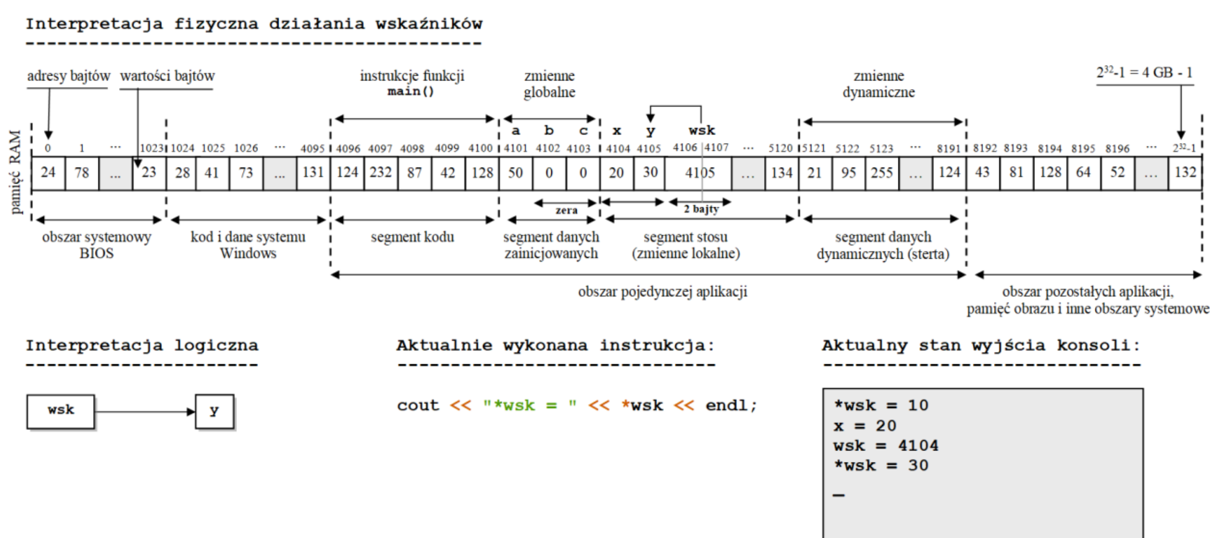
Na ekranie konsoli pojawiła się wartość zmiennej wskaźnikowej **`wsk`**, tzn. 4104, będąca zwykłą liczbą całkowitą równą adresowi w pamięci RAM zmiennej wskazywanej **`x`**, z którą obecnie związany jest wskaźnik **`wsk`**.



Rysunek 8. *Pamięć RAM i stan konsoli po wykonaniu instrukcji z linii nr 15 programu przykładowego.*

Powiedzieliśmy już wcześniej, że zmienne wskaźnikowe mogą wskazywać na inne zmienne, o ile typ zmiennych wskazywanych zgadza się z typem zadeklarowanym podczas definicji zmiennej wskaźnikowej. Do tej pory nasza przykładowa zmienna wskaźnikowa **`wsk`** wskazywała na zmienną całkowitą **`x`**, co przypomnijmy, było wynikiem wykonania instrukcji w linii 10 naszego programu przykładowego (patrz Listing 3 oraz Rys. 3). Nic nie stoi na przeszkodzie, aby zmienić wskazanie zmiennej **`wsk`**, w taki sposób, aby związać ją z inną zmienną całkowitą. Tak też się dzieje w linii nr 15 naszego programu przykładowego, zobaczmy Rys. 8.

Widzimy, że podobnie jak wcześniej używając operatora adresu '**`&`**', przypisaliśmy zmiennej wskaźnikowej **`wsk`**, adres zmiennej całkowitej **`y`**. Od tej pory wskaźnik **`wsk`** wskazuje na zmienną **`y`**, a jego wartością jest adres zmiennej **`y`** w pamięci RAM, tzn. liczba 4105. Aby potwierdzić nasze przypuszczenia wyświetlmy w oknie konsoli wartość zmiennej **`y`** przy użyciu wskaźnika **`wsk`** korzystając z operatora wyłuskania '**`*`**' – odpowiednią sytuację obrazuje Rys. 9.

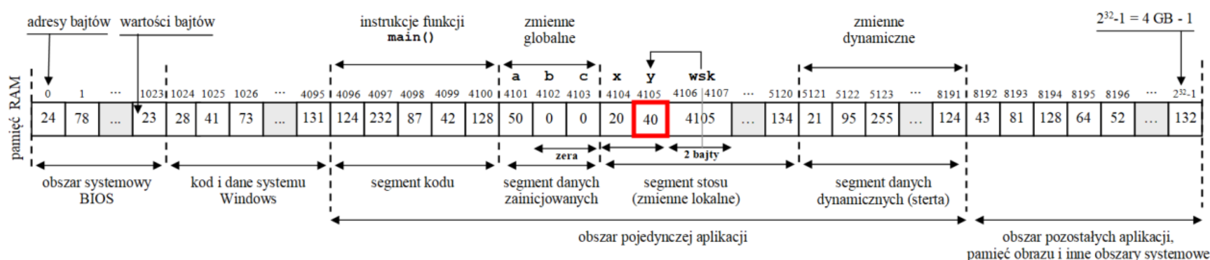


Rysunek 9. *Pamięć RAM i stan konsoli po wykonaniu instrukcji z linii nr 16 programu przykładowego.*

Jak widać komunikat w oknie konsoli potwierdza, że od tego momentu wykonania programu wskaźnik **`wsk`** rzeczywiście związany jest już ze zmienną całkowitą **`y`** (a nie, tak jak było do tej pory, ze zmienną **`x`**).

Podobnie jak wcześniej użyjmy wskaźnika **`wsk`** do zapisu nowej wartości we wskazywanej prze niego aktualnie zmiennej **`y`**. Dokonujemy tego po raz kolejny korzystając z operatora wyłuskania '**`*`**' zastosowanego na rzecz wskaźnika **`wsk`**. Aktualny stan naszej aplikacji po wykonaniu opisanej przed chwilą instrukcji (tzn. instrukcji z linii nr 17 programu przykładowego) obrazuje poniższy Rys. 10.

Interpretacja fizyczna działania wskaźników



Interpretacja logiczna

Aktualnie wykonana instrukcja:

```
*wsk = 40;
```

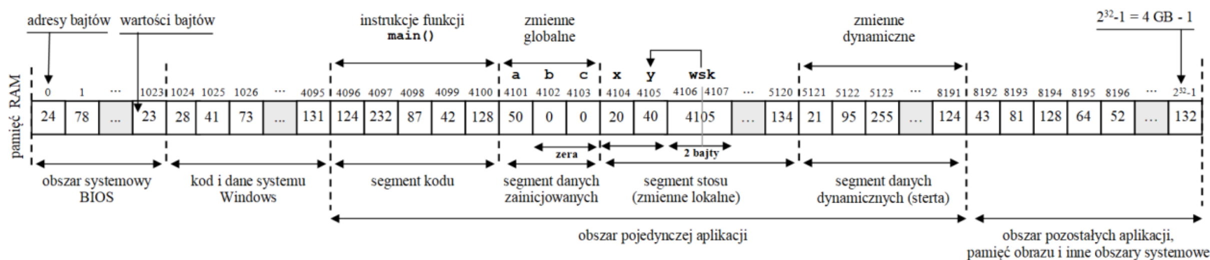
Aktualny stan wyjścia konsoli:

```
*wsk = 10
x = 20
wsk = 4104
*wsk = 30
_
```

Rysunek 10. *Pamięć RAM i stan konsoli po wykonaniu instrukcji z linii nr 17 programu przykładowego.*

Podobnie jak wcześniej, aby potwierdzić, że rzeczywiście wartość zmiennej **`y`** została zmieniona z użyciem wskaźnika **`wsk`**, w linii 18 naszego programu przykładowego wyświetlamy aktualną wartość tej zmiennej. Stan naszej aplikacji po wykonaniu opisanej przed chwilą instrukcji będzie przedstawiał się następująco.

Interpretacja fizyczna działania wskaźników



Interpretacja logiczna

Aktualnie wykonana instrukcja:

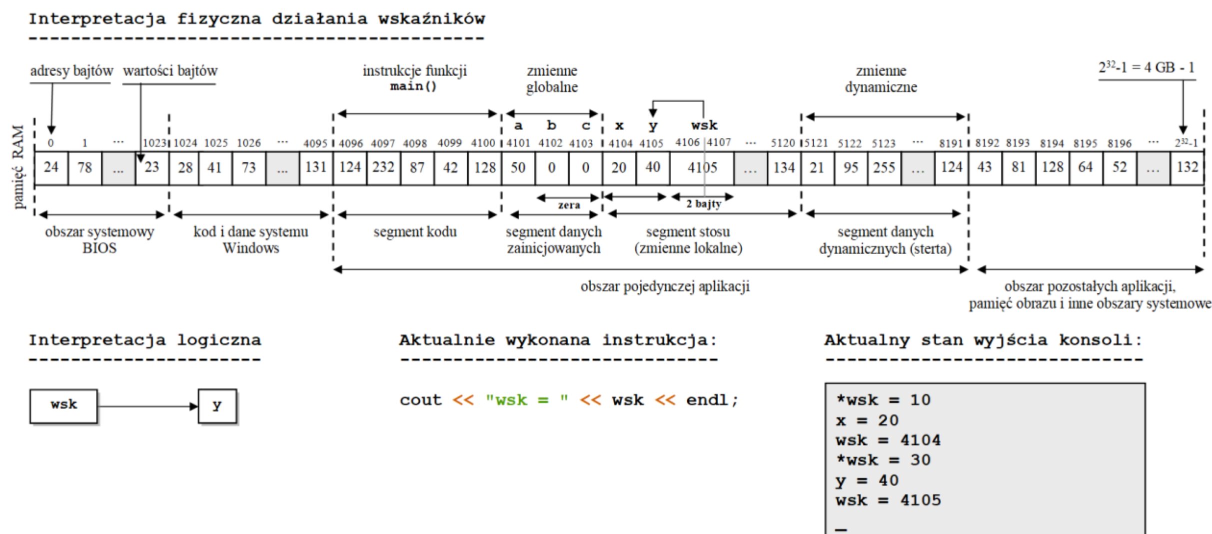
```
cout << "y = " << y << endl;
```

Aktualny stan wyjścia konsoli:

```
*wsk = 10
x = 20
wsk = 4104
*wsk = 30
y = 40
_
```

Rysunek 11. *Pamięć RAM i stan konsoli po wykonaniu instrukcji z linii nr 18 programu przykładowego.*

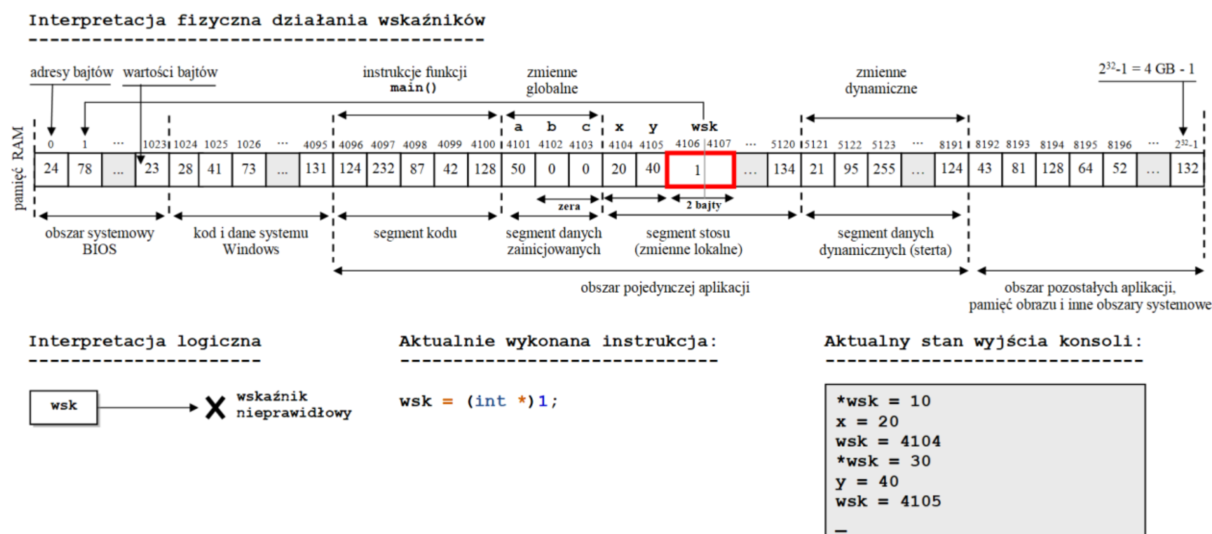
Wyświetlenie w oknie konsoli wartości zmiennej **`y`** potwierdza ostatecznie, że zmienna ta została zmodyfikowana z użyciem wskaźnika **`wsk`**. W takim razie wyświetlmy jeszcze wartość samej zmiennej wskaźnikowej **`wsk`**, aby poznać adres zmiennej **`y`** w pamięci RAM – spójrzmy na Rys. 12.



Rysunek 12. *Pamięć RAM i stan konsoli po wykonaniu instrukcji z linii nr 19 programu przykładowego.*

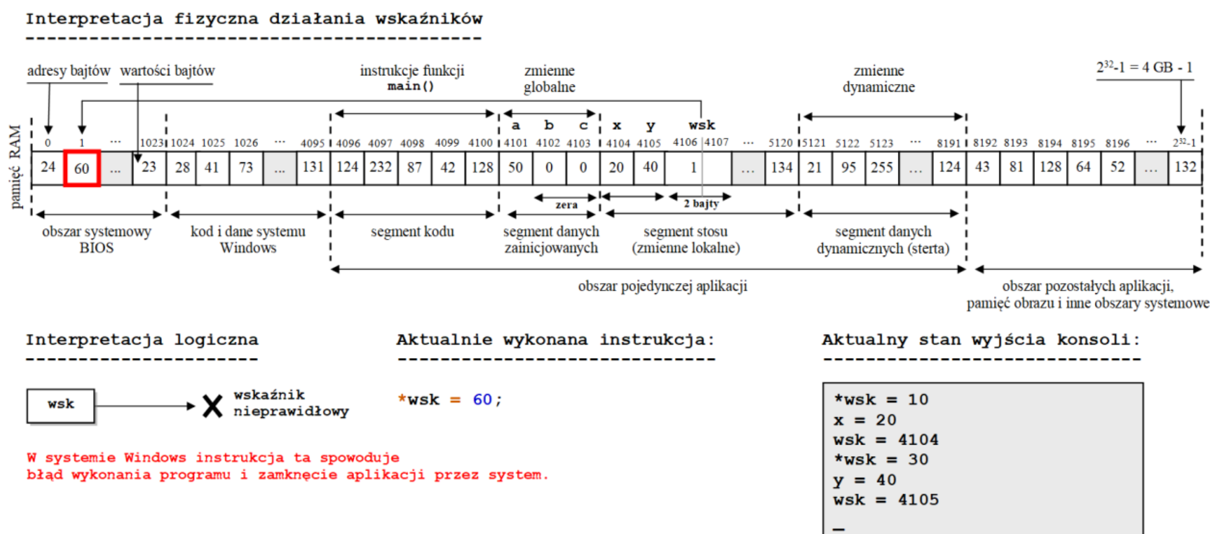
Widzimy, że zmienna **`y`** umieszczona jest w pamięci RAM pod adresem 4105, o czym mogliśmy się właśnie przekonać wyświetlając wartość wskaźnika **`wsk`**, który aktualnie jest z tą zmienną związany.

Na koniec zajmijmy się jeszcze jedną ciekawą sytuacją obrazującą naturę wskaźników w języku C++. Wszystkie pokazane do tej pory przykłady pokazują wyraźnie, że poza faktem, iż z logicznego punktu widzenia wskaźniki pozwalają na wskazywanie innych zmiennych obecnych w programie w celu odwoływania się do nich (zapisu lub odczytu ich wartości), to fizycznie rzecz biorąc, wartościami samych zmiennych wskaźnikowych są zwykłe liczby całkowite (adresy wskazywanych przez nie zmiennych). Skoro zatem są to liczby całkowite, to nic nie stoi na przeszkodzie, aby przypisać wskaźnikowi wybraną, konkretną wartość całkowitą, nie związaną z adresem jakiejkolwiek zmiennej obecnej w programie. Tak też się dzieje w linii 20 naszego programu przykładowego – spójrzmy na Rys. 13 obrazujący tę sytuację.



Rysunek 13. *Pamięć RAM i stan konsoli po wykonaniu instrukcji z linii nr 20 programu przykładowego.*

Zmiennej wskaźnikowej **wsk** została przypisana wartość 1[1]. Widzimy, że wartość ta nie odpowiada adresowi żadnej ze zmiennych obecnych w naszym programie. Mimo to wskaźnik **wsk** nadal pokazuje konkretną komórkę pamięci RAM, choć nie jest to adres żadnej ze zmiennych naszego programu. Skoro tak, to spróbujmy zapisać do wskazywanej przez wskaźnik **wsk** komórki pamięci RAM jakąś nową wartość, aby przekonać się jaki będzie tego efekt. Aby tego dokonać stosujemy jak zwykle (bez żadnych zmian) operator wyłuskania. Odpowiednia instrukcja znajduje się w linii 21 naszego programu przykładowego, zaś efekt jej działania obrazuje poniższy Rys. 14.



Rysunek 14. *Pamięć RAM i stan konsoli po wykonaniu instrukcji z linii nr 20 programu przykładowego.*

Sytuacja teraz jest następująca: używając wskaźnika **wsk** zapisaliśmy wartość 60 do komórki pamięci RAM o adresie równym 1. Pamiętamy z Rozdziału 3, że pierwszy kilobajt pamięci RAM komputera to obszar systemowy, zawierający krytyczne dane służące systemowi komputerowemu do komunikacji z urządzeniami zewnętrznymi. Każda zmiana zawartości pamięci RAM w tym obszarze grozi natychmiastowym zawieszeniem się systemu komputerowego. Co zatem stanie się faktycznie po wykonaniu instrukcji z linii 20? Okazuje się, że system Windows zablokuje możliwość takiego zapisu. Wartość 60 nie zostanie w rzeczywistości zapisana do komórki pamięci RAM o adresie 1. Zamiast tego, w reakcji na próbę takiego zapisu, system Windows natychmiast przerwie wykonanie naszego programu, wyświetli okno z informacją o błędzie wykonania programu i zakończy jego działanie. Można się o tym przekonać uruchamiając nasz program i obserwując reakcję systemu. Przykład ten pokazuje, że wskaźniki mogą pokazywać dowolny obszar pamięci RAM, niekoniecznie związany z konkretną zmienną obecną w programie. Z niskopoziomowego punktu widzenia mechanizmu wskaźników języka C++, nie ma żadnej różnicy pomiędzy wskaźnikiem związanym ze zmienną naszego programu, a takim, pokazującym inny obszar pamięci RAM niezwiązany z żadną ze zmiennych obecnych w programie. Z tego też powodu wskaźnik o wartości zerowej (zdefiniowanej w bibliotece **<cstdlib>** jako literał **NULL**), choć wskazuje prawidłowy adres w pamięci RAM – komórkę o adresie 0 – może

[1] w celu wykonania tego przypisania musieliśmy wykorzystać tzw. *operator rzutowania* **(int *)**, który zapobiega błędowi kompilacji programu, polegającemu na niezgodności typów wskaźnikowego (zmienna **wsk**) i całkowitego (stała 1). Dokładne omówienie roli operatora rzutowania zostanie przedstawione w dalszej części kursu.

być uznawany za tzw. *wskaźnik pusty*. Tzn. jeśli wartość wskaźnika wynosi 0 (**`NULL`**), to programista może przyjąć, że jest to wskaźnik pusty – nie związany z żadną zmienną naszego programu, bo i tak nie można z takiego wskaźnika skorzystać, gdyż system Windows będzie blokował wszystkie operacje odczytu i zapisu jakie chcielibyśmy wykonać z jego użyciem. A zatem wartość zerowa zmiennej wskaźnikowej może w ten sposób sygnalizować sytuację, w której z logicznego punktu widzenia, wskaźnik zawierający tę wartość nie pokazuje żadnej zmiennej.

5. Podsumowanie

W niniejszej części kursu z programowania w języku C++ poznaliśmy pojęcie wskaźników. Omówiony został mechanizm wskaźników języka C++ oraz podana została nie tylko logiczna, ale także niskopoziomowa (fizyczna) interpretacja sposobu ich działania. Nie podaliśmy na razie przykładów zastosowania wskaźników w praktycznych algorytmach realizowanych w języku C++, koncentrując się jedynie na dokładnym wyjaśnieniu sposobu ich działania (takie zastosowania pojawią się w kolejnych częściach kursu). Z doświadczeń autorów wynika bowiem, że czas poświęcony na dogłębne zrozumienie mechanizmu działania wskaźników na jego podstawowym poziomie znakomicie procentuje w przyszłości, znacząco ułatwiając zrozumienie lub/i analizę wszystkich algorytmów wykorzystujących wskaźniki, nie pozostawiając już żadnych wątpliwości co do sposobu ich funkcjonowania.

Wskaźniki i adresy

Przykład (C++)

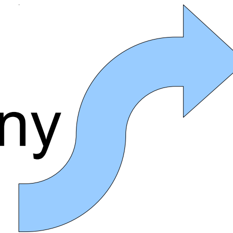
Bartłomiej Stasiak
bartlomiej.stasiak@p.lodz.pl

Wskaźniki i adresy

Centrum Mistrzostwa Informatycznego

- Przykład

```
int a;
int b;
int c;
int* ptr;
```



Pamięć

	Pamięć	
a	???	1024
b	???	1025
c	???	1026
ptr	???	1027
		1028
		1029
		1030
		1031
		1032
		1033

- Każda zmienna ma przydzielony swój obszar pamięci
- Uwaga: adresy komórek pamięci w prawej kolumnie są umowne – nie są wyrażone bezpośrednio w bajtach (gdyż każdy typ może zajmować różną ilość bajtów)
  - Jednobajtowym typem w C++ jest np. `char`
  - Zmienna typu `int` zajmuje w pamięci przynajmniej dwa, a najczęściej cztery (sąsiednie) bajty

Wskaźniki i adresy

Centrum Mistrzostwa Informatycznego

- Przykład

```
int a = 1;
int b = 2;
int c = 3;
int* ptr;
ptr = &a;
*ptr = 11;
ptr = &b;
c = *ptr + 10;
cout << a << endl;
cout << b << endl;
cout << c << endl;
```

Pamięć

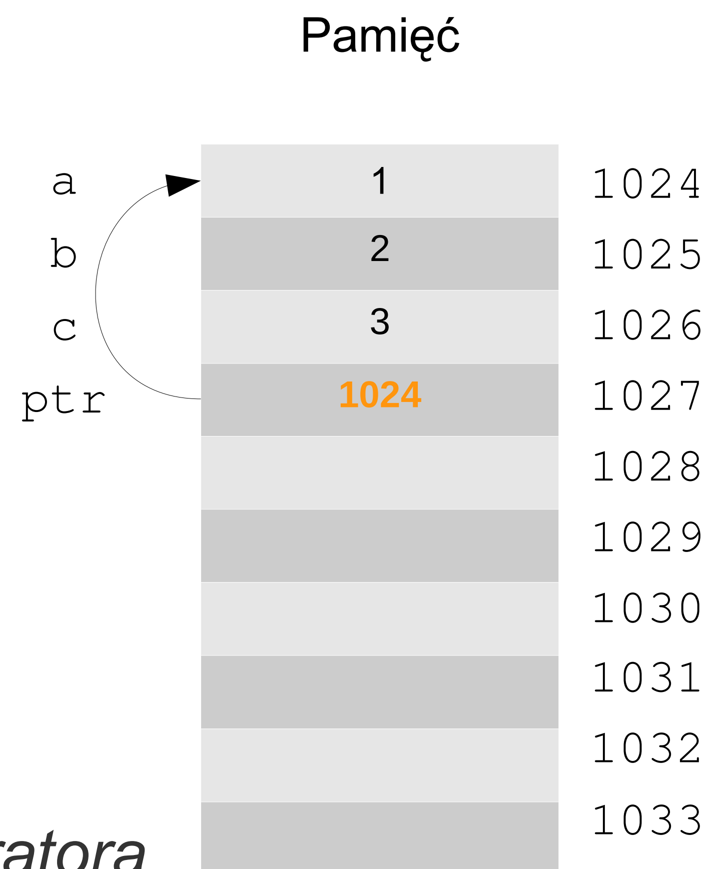
a	1	1024
b	2	1025
c	3	1026
ptr	???	1027
		1028
		1029
		1030
		1031
		1032
		1033

- Wartością zmiennej typu int jest liczba całkowita, natomiast wartością zmiennej typu wskaźnikowego do `int` (czyli: `int*`) jest *adres* zmiennej typu `int`

Wskaźniki i adresy

- Przykład

```
int a = 1;
int b = 2;
int c = 3;
int* ptr;
ptr = &a;
*ptr = 11;
ptr = &b;
c = *ptr + 10;
cout << a << endl;
cout << b << endl;
cout << c << endl;
```

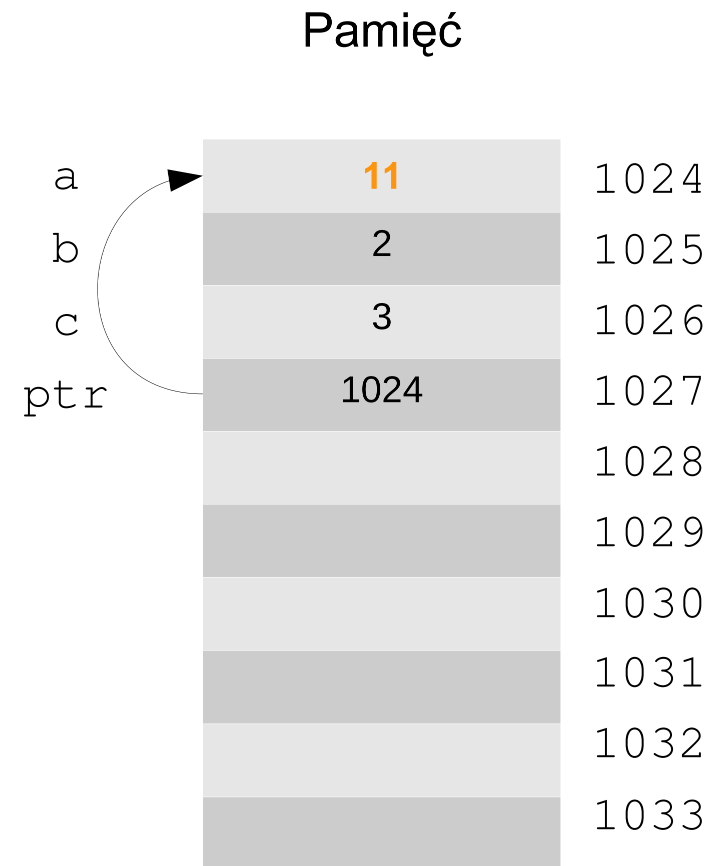


- Adres ten uzyskujemy za pomocą *operatora pobrania adresu* `&`

Wskaźniki i adresy

- Przykład

```
int a = 1;
int b = 2;
int c = 3;
int* ptr;
ptr = &a;
*ptr = 11;
ptr = &b;
c = *ptr + 10;
cout << a << endl;
cout << b << endl;
cout << c << endl;
```



- Do odczytania zawartości zmiennej wskazywanej przez wskaźnik służy *operator wyłuskania* (ang. *dereference*), oznaczany symbolem `*`

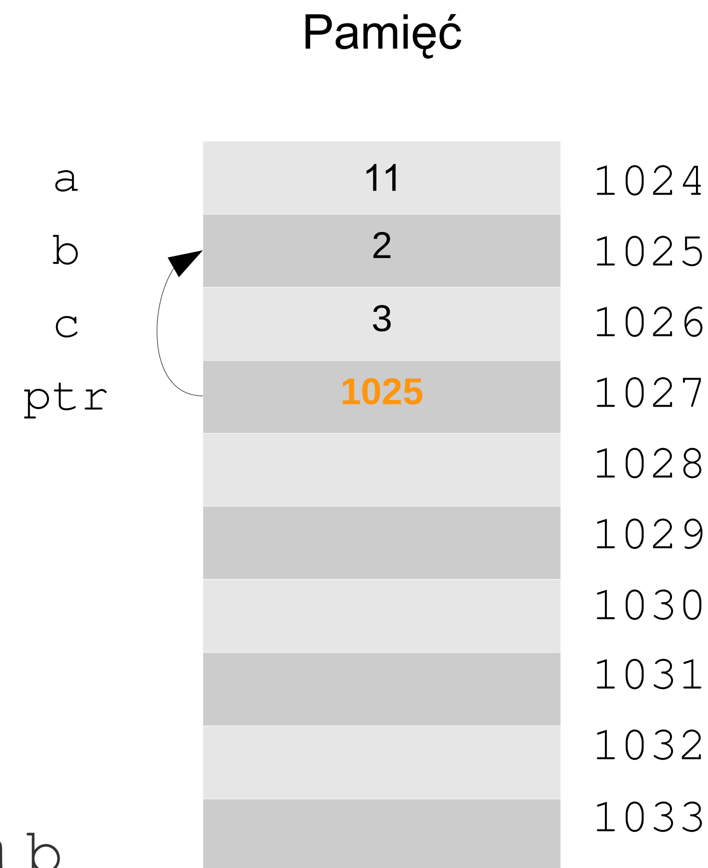
Wskaźniki i adresy

Centrum Mistrzostwa Informatycznego

6

- Przykład

```
int a = 1;
int b = 2;
int c = 3;
int* ptr;
ptr = &a;
*ptr = 11;
ptr = &b;
c = *ptr + 10;
cout << a << endl;
cout << b << endl;
cout << c << endl;
```

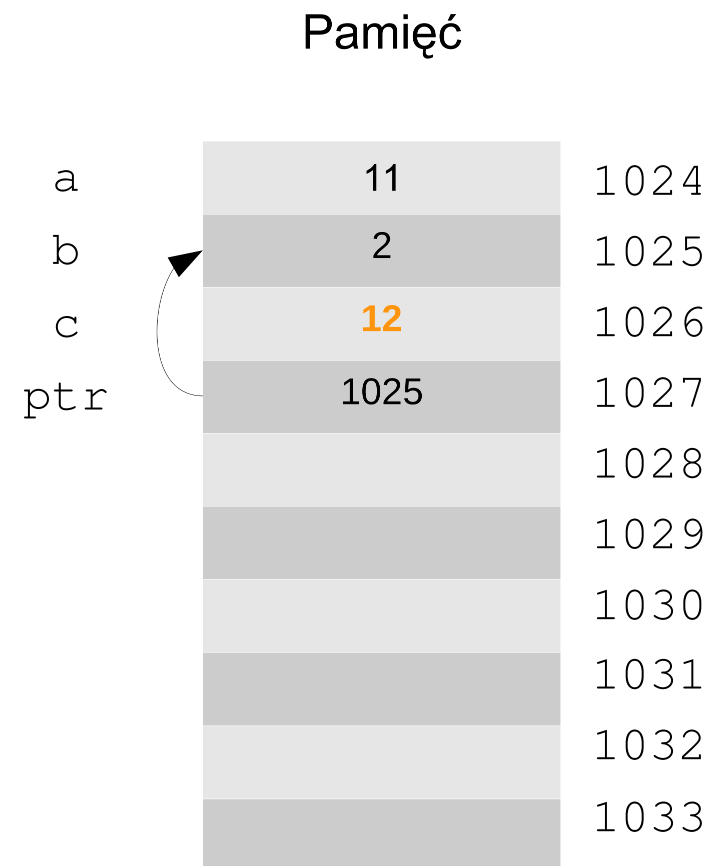


- Wskaźnik `ptr` wskazuje teraz zmienną `b` (innymi słowy: wartość zmiennej `ptr` to *adres* zmiennej `b`)

Wskaźniki i adresy

- Przykład

```
int a = 1;
int b = 2;
int c = 3;
int* ptr;
ptr = &a;
*ptr = 11;
ptr = &b;
c = *ptr + 10;
cout << a << endl;
cout << b << endl;
cout << c << endl;
```

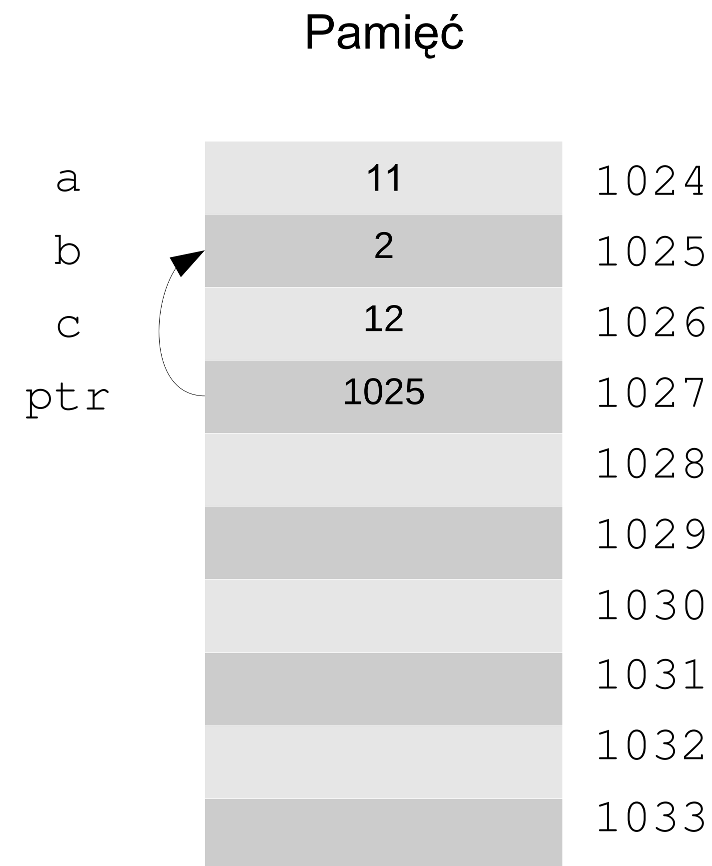


- Znów używamy operatora wyłuskania do pobrania zawartości spod wskazywanego adresu

Wskaźniki i adresy

- Przykład

```
int a = 1;
int b = 2;
int c = 3;
int* ptr;
ptr = &a;
*ptr = 11;
ptr = &b;
c = *ptr + 10;
cout << ”a = ” << a << endl;
cout << ”b = ” << b << endl;
cout << ”c = ” << c << endl;
```



- Wynik:

```
a = 11
b = 2
c = 12
```

Dziękuję za uwagę!

Centrum Mistrzostwa Informatycznego

www.cmi.edu.pl

Projekt "Centrum Mistrzostwa Informatycznego" współfinansowany jest ze środków Unii Europejskiej z Europejskiego Funduszu Rozwoju Regionalnego
w ramach Programu Operacyjnego Polska Cyfrowa na lata 2014 - 2020