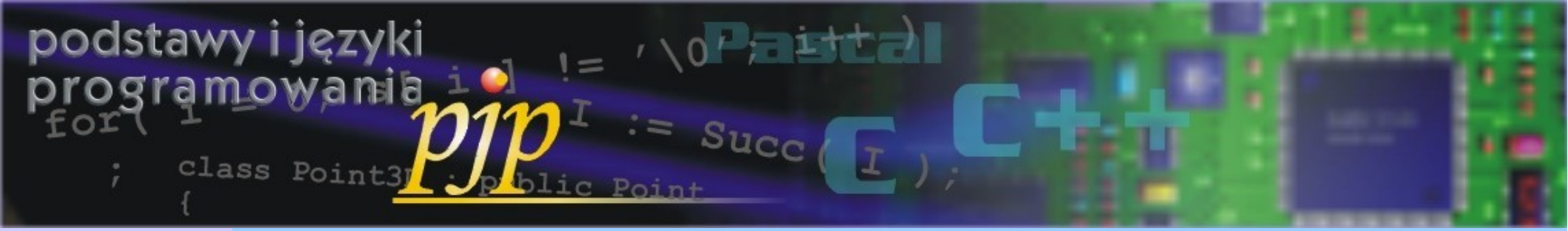


Podstawy programowania

Część siódma

Zmienne wskaźnikowe — wprowadzenie

Autor
Roman Simiński

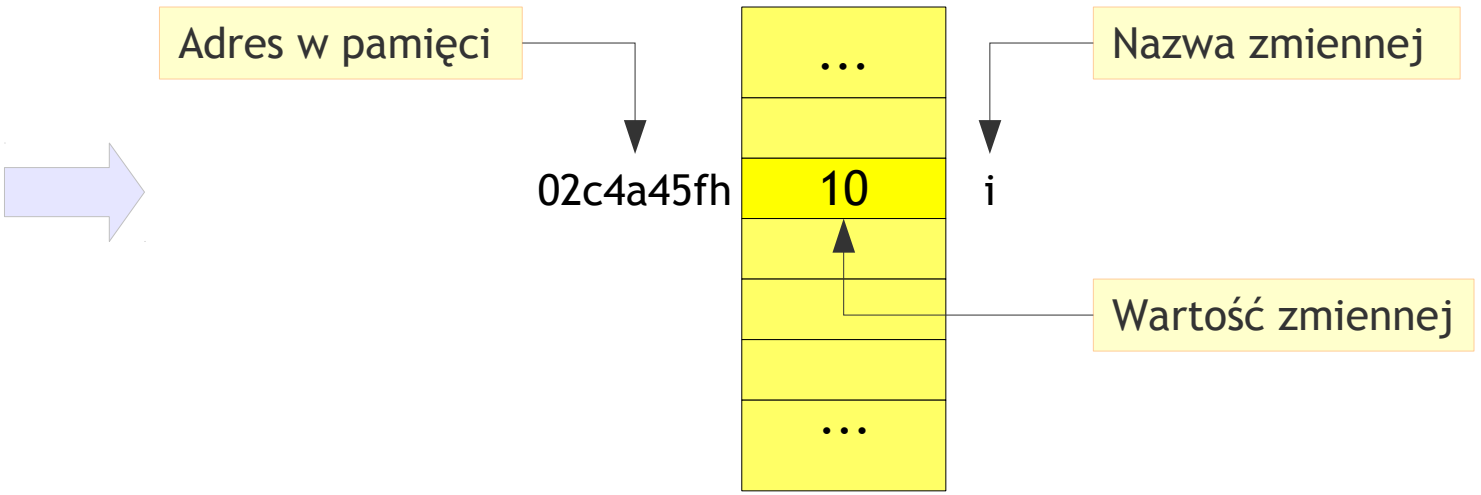
Kontakt
roman.siminski@us.edu.pl
www.programowanie.siminskionline.pl

Co to jest zmienna — przypomnienie

Zmienna jest obiektem w programie, rezydującym w pamięci operacyjnej, przeznaczonym do *przechowywania wartości* pewnego typu.

- Każda zmienna ma swoją *nazwę*, oraz *typ wartości*.
- Zmienne są przechowywane w pamięci operacyjnej, liczba zajętych bajtów zależy od *typu zmiennej*.
- Nazwa zmiennej *identyfikuje zmienną* w programie zwalniając programistę od zastanawiania się, *pod jakim adresem* w pamięci zmienna jest zlokalizowana.

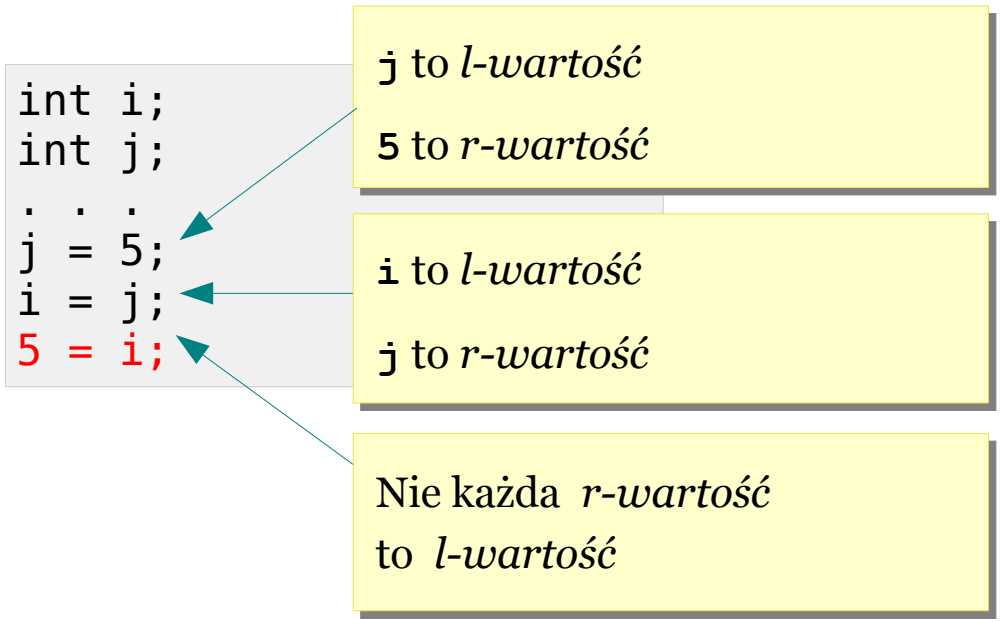
```
int i;

i = 10;
```



Dziwne pojęcia — *l*-wartość i *r*-wartość

Obiekt jest pewnym *nazwanym obszarem pamięci*. Pod pojęciem ***l-wartości*** rozumiemy *wyrażenie lokalizujące* ten obiekt w pamięci

- *Zmienna* może występować po *lewej stronie* operatora przypisania, mówi się, że jest wtedy *l-wartością*.
- Wszystko, co może występować *po prawej stronie* operatora przypisania jest *r-wartością*.



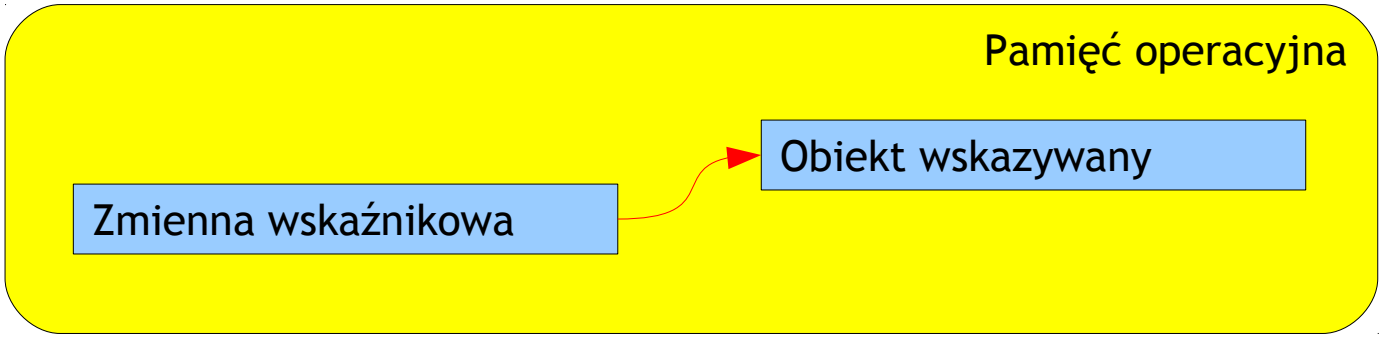
Zmienne wskaźnikowe — motywacja do nauki

- W języku C intensywnie wykorzystuje się *l*-wartości oparte na zmiennych wskaźnikowych oraz na wyrażeniach te zmienne zawierających.
- Dokładne opanowanie zasad posługiwania się wskaźnikami jest niezbędne do efektywnego i sprawnego programowania w C i C++.
- Tej umiejętności nie można pominąć, przeskoczyć lub zostawić na później.
- Nie oszukujmy się — ten, kto nie opanuje zasad posługiwania się wskaźnikami nigdy nie będzie prawdziwym, profesjonalnym programistą, wykorzystującym język C lub C++.

Koncepcja wskaźników oraz metody ich wykorzystania *są proste*. Wymagają one jednak *uwagi*, *zrozumienia* i *myślenia*.

Po co są zmienne wskaźnikowe?

- ***Zmienna wskaźnikowa*** przeznaczona jest do *lokalizowania* (inaczej *wskazywania*) obiektów w pamięci operacyjnej.
- Jedyną rolą zmiennej wskaźnikowej jest umożliwienie odwoływania się do obiektów wskazywanych.



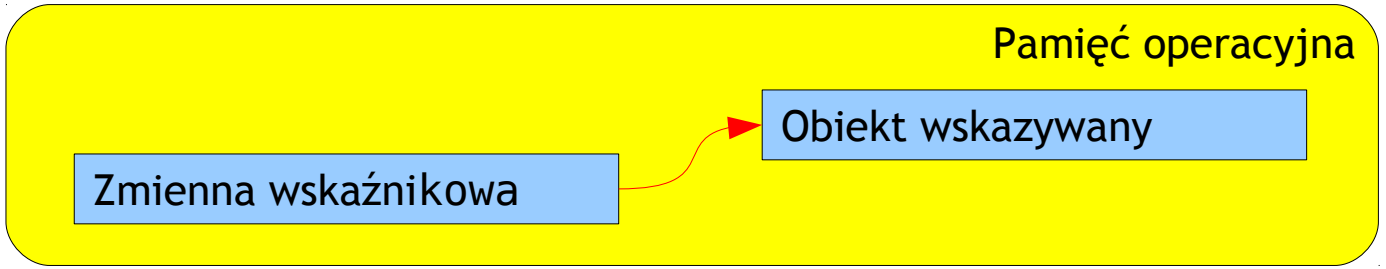
- Zmienna wskaźnikowa może lokalizować w pamięci operacyjnej:
  - *inne zmienne*,
  - *nienazwane bloki pamięci*,
  - bloki zawierające *kod programu*, np. *funkcje*.

Czym jest zmienna wskaźnikowa?

- Zmienna wskaźnikowa rezyduje w pamięci operacyjnej.
- Sama zmienna wskaźnikowa może być również „wskazywana” przez inną zmienną wskaźnikową.

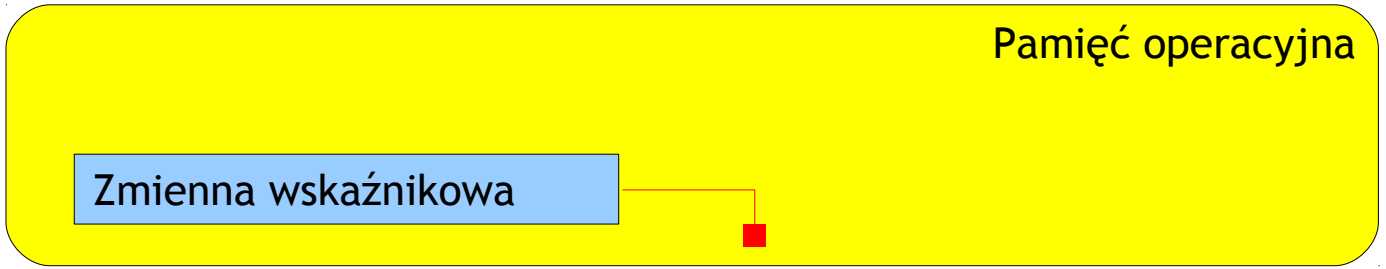
Trzy stany zmiennej wskaźnikowej

- Zmienna wskaźnikowa wskazuje na konkretny obiekt w pamięci:



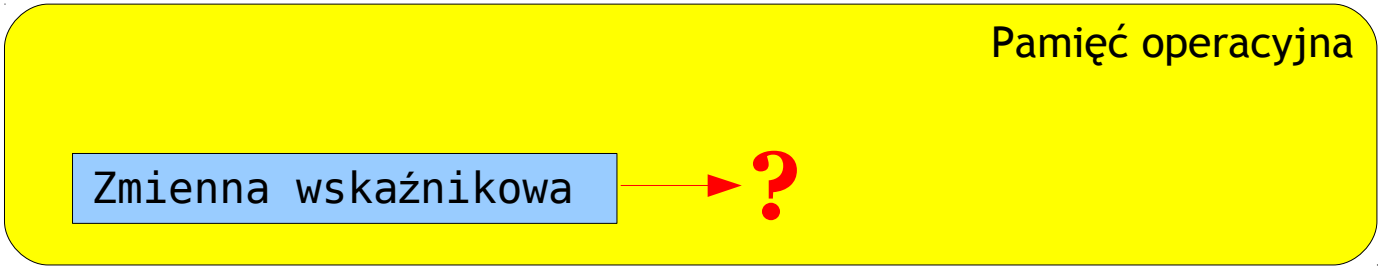
☑ OK

- Zmienna wskaźnikowa nie wskazuje na żaden obiekt:



☑ OK

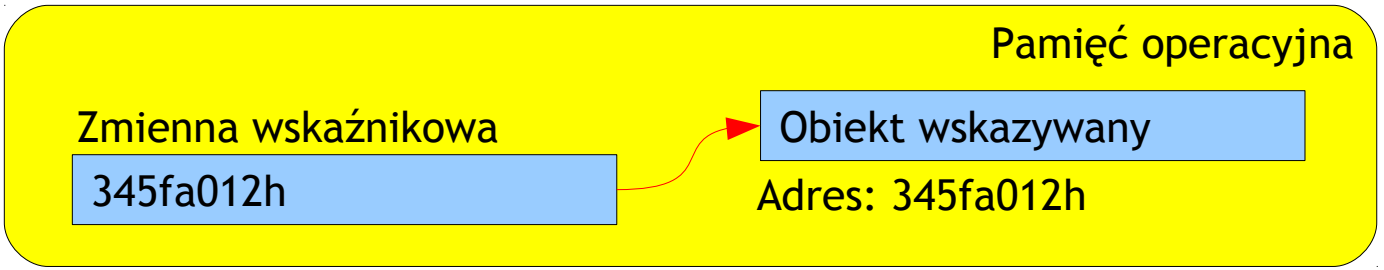
- Zmienna wskaźnikowa wskazuje na nie wiadomo co:



☒ Kiepsko

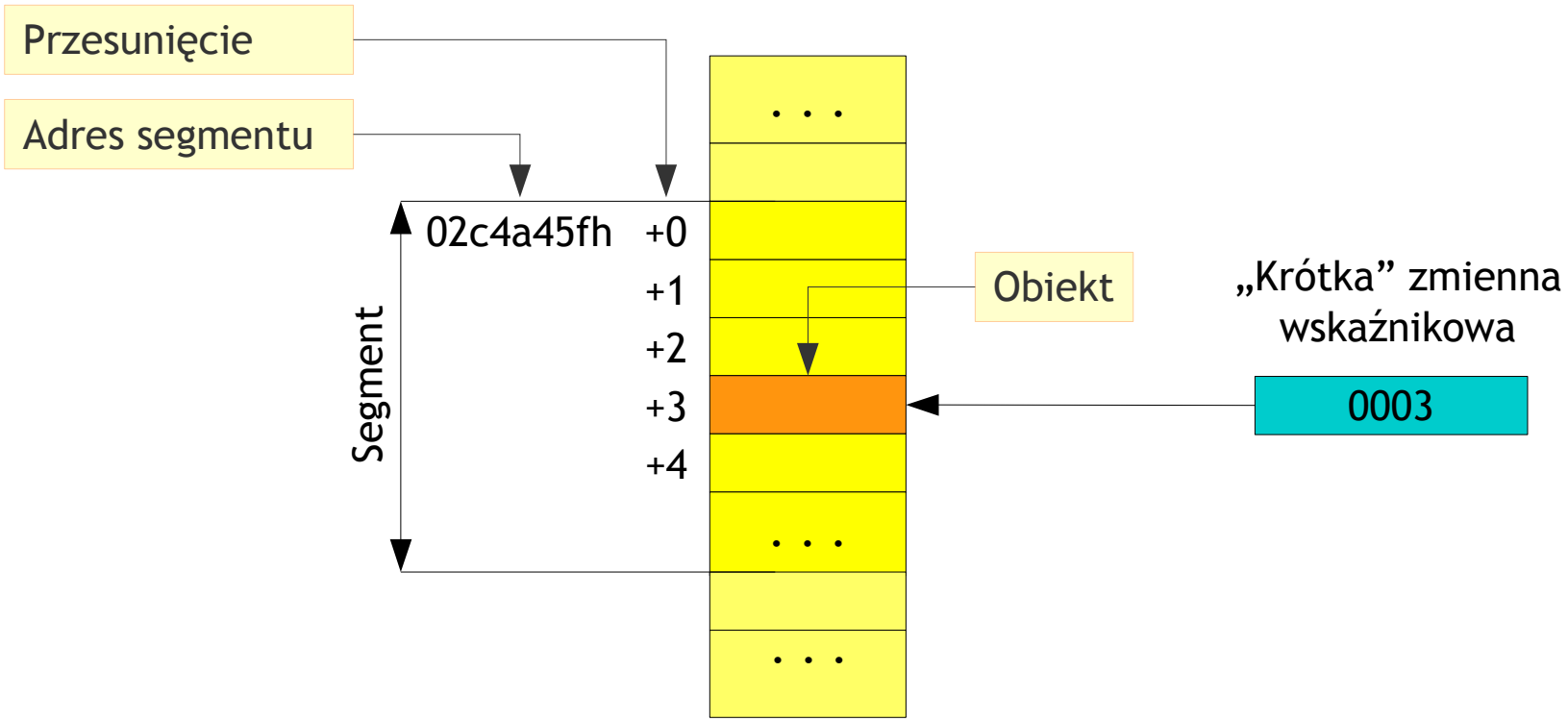
Co zawiera zmienna wskaźnikowa?

- Zwykle przyjmuje się, że zmienna wskaźnikowa zawiera w sobie *adres obiektu wskazywanego*.
- Jednak zmienna wskaźnikowa nie musi w sobie zawierać adresu bezpośredniego (fizycznego).
- Zawartość zmiennej wskaźnikowej może zawierać inną informację, pozwalającą na precyzyjne i jednoznaczne zidentyfikowanie położenia obiektu w pamięci.



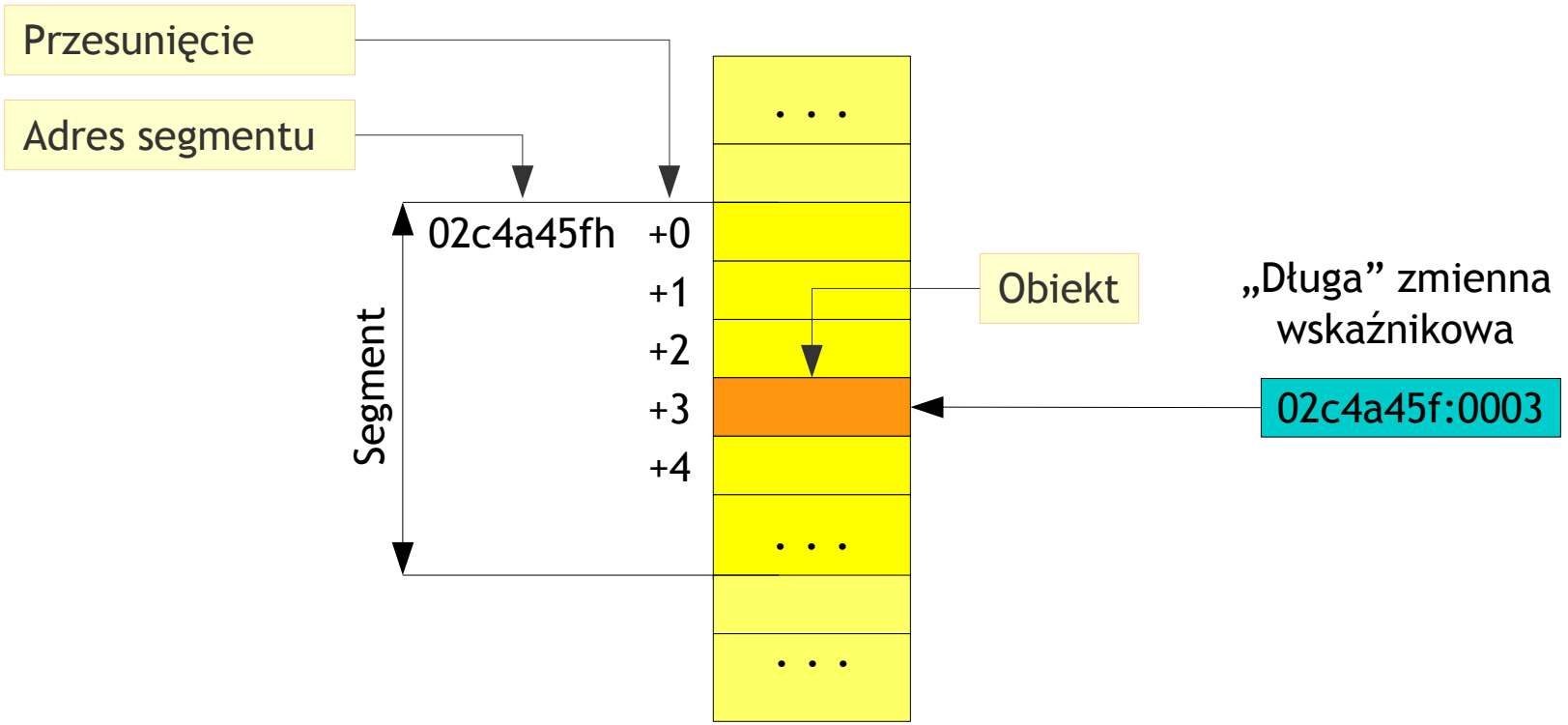
Przykład implementacji zmiennej wskaźnikowej — Intel 8086

- Zmienna wskaźnikowa zawiera *przesunięcie* (ang. *offset*) obiektu względem początku segmentu gdy wskaźniki są „krótkie” (ang. *near*) — odwołania wewnątrz segmentu.



Przykład implementacji zmiennej wskaźnikowej — Intel 8086

- Zmienna wskaźnikowa zawiera *adres segmentu* i *przesunięcie* obiektu gdy wskaźniki są „długie” (ang. *far*) — odwołania międzysegmentowe.



Deklaracja zmiennej wskaźnikowej

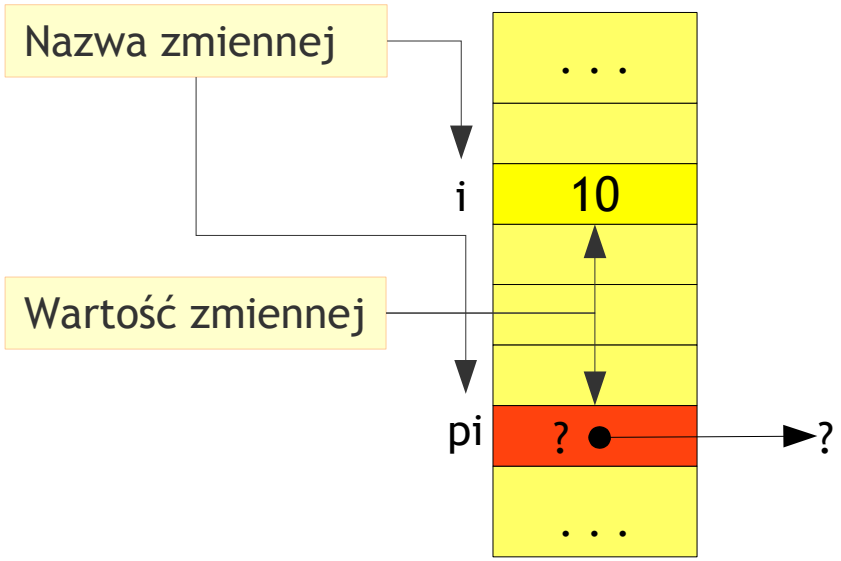
Deklarowana zmienna będzie wskaźnikiem, kompilator wie, ile dla niej zarezerwować pamięci.

To oznacza, że deklarowana zmienna wskaźnikowa będzie przeznaczona do lokalizowania w pamięci obiektów typu **`int`**.

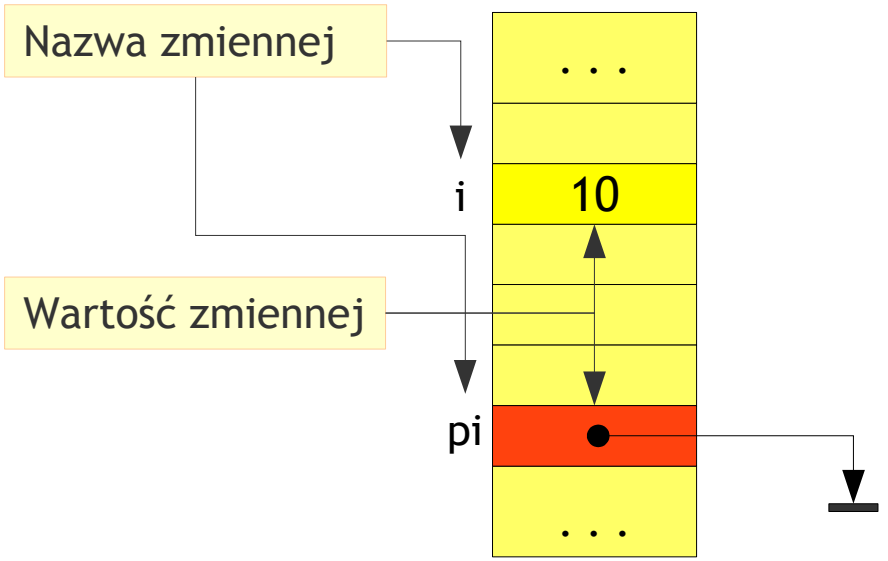
```
int * pi;
```

Nazwa deklarowanej zmiennej wskaźnikowej zbudowana wg. zwykłych reguł, często zawiera **p** lub **ptr** od *pointer*.

```
int i = 10;
int * pi;
```




```
int i = 10;
int * pi = 0;
```



Rola wskaźnika pustego NULL

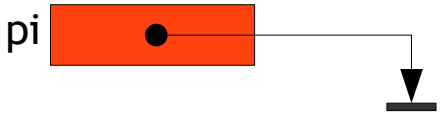
Tak zdefiniowana zmienna wskaźnikowa:

```
int * pi;
```

ma wartość początkową zależną od kontekstu deklaracji. Jeżeli ta zmienna jest klasy *auto*, to jej wartość jest *przypadkowa* — zmienna „wskazuje” zatem na bliżej nieznany obiekt w pamięci.

W pliku nagłówkowym *stddef.h* zdefiniowana stałą *NULL*, reprezentującą wskaźnik pusty, niezależny od platformy i implementacji. Tak zdefiniowana zmienna:

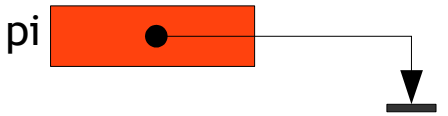
```
int * pi = NULL;
```



jest wskaźnikiem pustym, a więc nie wskazuje żadnego obiektu w pamięci.

W języku C++ preferuje się wykorzystanie wartości 0 zamiast stałej NULL.

```
int * pi = 0;
```



Wartość NULL kontra 0

Stała NULL jest definiowana jako wartość 0 lub 0L. Można zatem zamiast wartością NULL, posługiwać się wartością 0.

W języku C praktykuje stosowanie wartości NULL a nie wartości 0.

W języku C++ praktykuje stosowanie wartości 0 zamiast NULL.

Niezależnie od przyjętej wartości wskaźnika pustego, jawnie inicjowanie zmiennych wskaźnikowych oraz posługiwanie się wartością pustą dla wskaźników niezakotwiczonych jest dobrą praktyką programistyczną w języku C i C++.

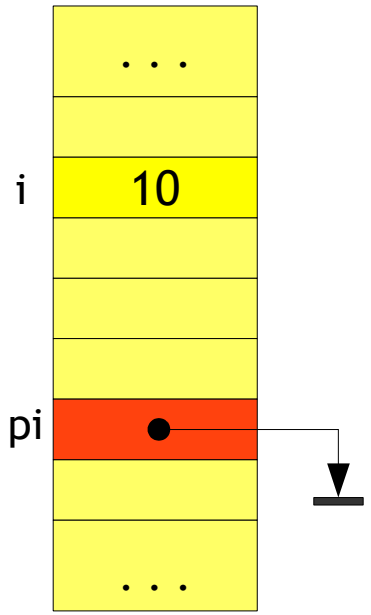
To, czy zmienna wskaźnikowa jest wskaźnikiem pustym można sprawdzić:

```
if( pi != NULL )
{
  // Tu jakieś operacje na obiekcie
  // wskazywanym przez pi
  . . .
}
```

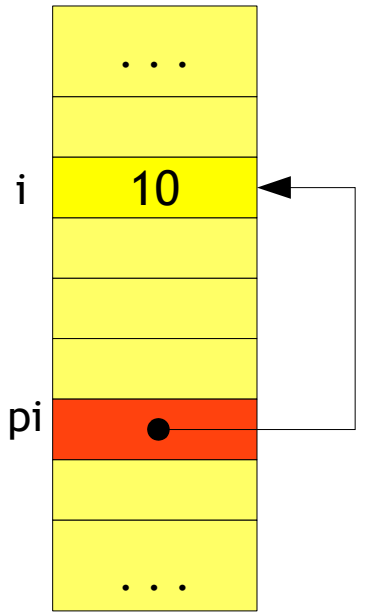
```
if( pi == NULL )
{
  // Nie odwołujemy się do obiektu
  // wskazywanego przez pi – nie ma go!
  . . .
}
```

Przypisywanie wartości zmiennym wskaźnikowym

```
int i = 10;
int * pi = 0;
```




```
int i = 10;
int * pi = 0;
. . .
pi = &i;
```



Od momentu tego przypisania, *pi* wskazuje zmienną *i*, umożliwiając realizację dowolnych operacji na tej zmiennej.

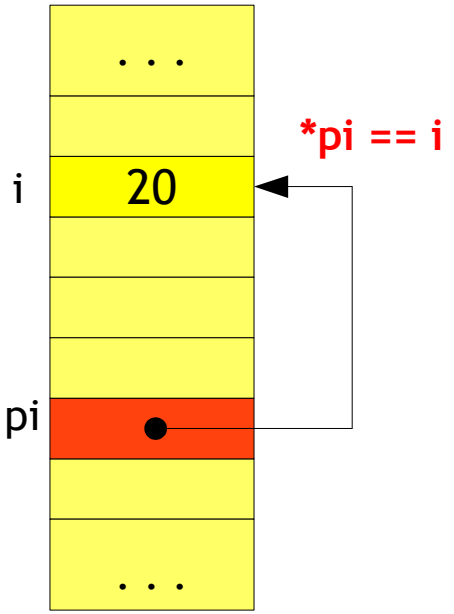
```
pi = &i;
```

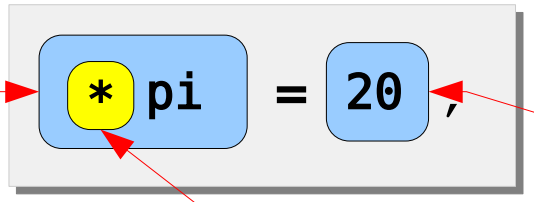
Wyrażenie wskaźnikowe lokalizujące zmienną ***i*** w pamięci.

Jednoargumentowy operator **&** buduje wyrażenie wskaźnikowe lokalizujące zmienną w pamięci operacyjnej. Argument musi być *l*-wartością, nie odnoszącą się do obiektu register ani pola bitowego.

Odwoływanie się do obiektu wskazywanego

```
int i = 10;
int * pi = 0;
. . .
pi = &i;
*pi = 20;
```





Ten zapis oznacza obiekt wskazywany przez **pi**. Zapis ***pi** może wystąpić wszędzie tam, gdzie może wystąpić **i**.

Jednoargumentowy operator *adresowania pośredniego* *, daje w wyniku obiekt wskazywany przez argument **pi**.

Dowolne wyrażenie typu zgodnego z typem obiektu wskazywanego.

Odwoływanie się do obiektu wskazywanego, uwagi

Po przypisaniu:

```
pi = &i;
```

te fragmenty kodu są równoważne:

```
cin >> *pi;

if( *pi == 0 )
  cout << "Bledna wartosc";
else
{
  y = *pi * x;
  cout << "Wynik: " << y;
}
```

```
cin >> i;

if( i == 0 )
  cout << "Bledna wartosc";
else
{
  y = i * x;
  cout << "Wynik: " << y;
}
```

Jeżeli wskaźnik *pi* wskazuje na zmienną *i*, to **pi* może wystąpić wszędzie tam, gdzie może wystąpić *i*. Zmienna *pi* jest *linkiem* (odnośnikiem) do zmiennej *i*, a wyrażenie **pi* jest *aliasem* (alternatywną nazwą) zmiennej *i*.

Jednoargumentowe operatory & i * — podsumowanie

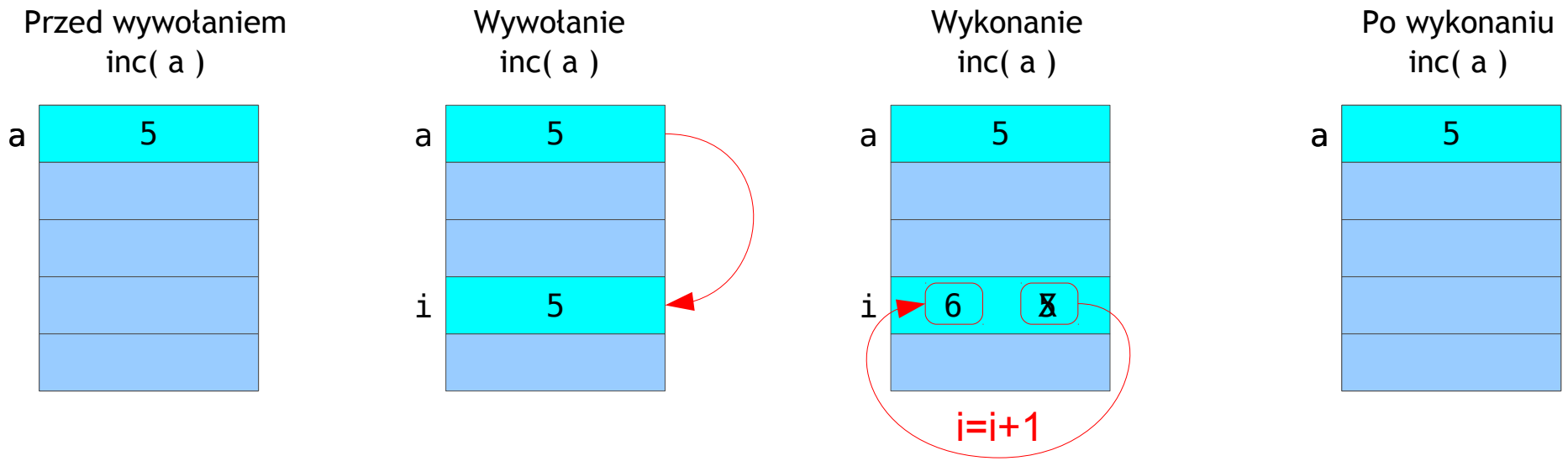
- Operatory **&** i ***** występują jako jedno i dwuargumentowe. W wersji dwuargumentowej oznaczają odpowiednio *bitową koniunkcję* i *iloczyn arytmetyczny*.
- W wersji jednoargumentowej oznaczają operacje wskaźnikowe.
- Wyrażenie **&coś_tam** oznacza „*gdzie jest coś_tam*” — operator **&** to zatem *lokalizator* lub *pobieracz adresu*.
- Wyrażenie ***wskaźnik** oznacza „*obiekt lokalizowany przez wskaźnik*” — operator ***** to zatem *ekstraktor* (*wydobywacz*) *obiektu wskazywanego*.
- Wydobywanie obiektu wskazywanego nazywa się *dereferencją wskaźnika*.

Typowe zastosowania zmiennych wskaźnikowych

- Realizacja przekazywania parametrów przez *zmienną*.
- Wykorzystanie pamięci zarządzanej dynamicznie.
- Manipulowanie tablicami (osobny wykład).
- Budowa rekurencyjnych struktur danych (osobny wykład).

Przypomnienie: przekazywanie parametrów przez wartość

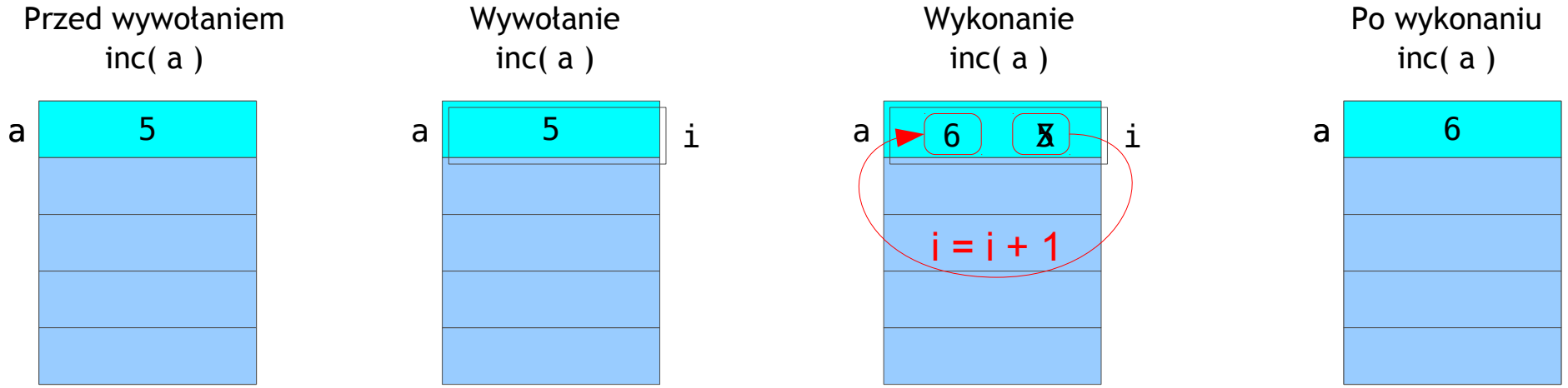
```
void inc( int i )
{
  i = i + 1;
}

. . .

int a = 5;

inc( a );

cout << a;
```



Przypomnienie: przekazywanie parametrów przez referencję (tylko C++)

```
void inc( int & i )
{
  i = i + 1;
}

. . .

int a = 5;

inc( a );

cout << "a = " << a;
```

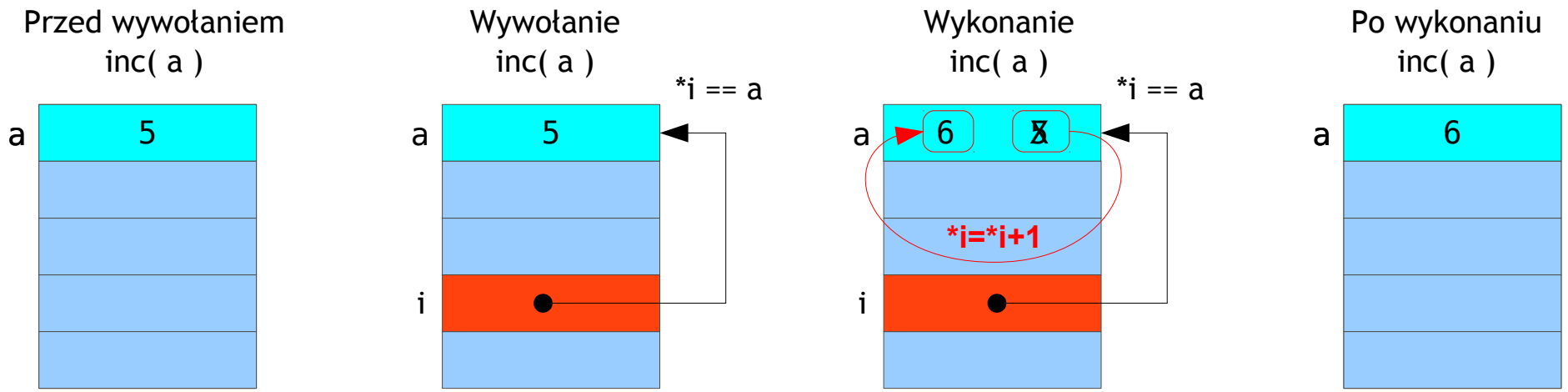
Parametr formalny *i* jest ***referencją*** do parametru aktualnego wywołania funkcji.



Wskaźniki a przekazywanie parametrów prawie jak przez referencję

```
void inc( int * i )
{
  *i = *i + 1;
}

. . .

int a = 5;

inc( &a );

cout << a;
```

Parametr formalny *i* jest wskaźnikiem.
Parametr aktualny wywołania również.

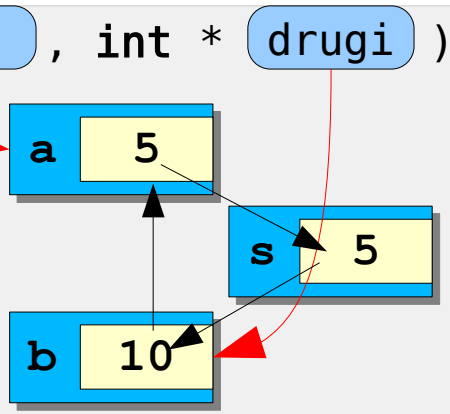


Wskaźniki a przekazywanie parametrów

W języku C wykorzystuje się parametry będące wskaźnikami do realizacji przekazywania parametrów działającego podobnie do przekazywania przez *referencję*.

Przykład przekazywania parametrów za pośrednictwem wskaźnika:

```
void zamien( int * pierwszy , int * drugi )
{
  int s; // Schowek

  s = *pierwszy;
  *pierwszy = *drugi;
  *drugi = s;
}
```




```
int a = 5, b = 10;
. . .
cout << "\na=" << a << " " << "b=" << b;
zamien( &a, &b );
cout << "\na=" << a << " " << "b=" << b;
```

```
a=5 b=10
a=10 b=5
```

Wskaźniki a przekazywanie parametrów — modyfikacja wewnątrz funkcji

W języku C/C++ często wykorzystuje się parametry wskaźnikowe po to, żeby przekazywanie parametrów odbywało się *szybciej* i nie zabierało *dodatkowej pamięci*, jednocześnie *nie oczekuje się*, że wnętrze funkcji będzie *modyfikować* parametr przekazywany za pośrednictwem wskaźnika.

- Dzieje się tak szczególnie wtedy, gdy przekazywany parametr jest duży.
- Przekazanie dużego parametru za pośrednictwem wskaźnika jest rzeczywiście szybsze i nie powoduje konieczności utworzenia kopii (oszczędzamy *pamięć*) w parametrze formalnym i skopiowania zawartości parametru aktualnego do parametru formalnego (oszczędzamy *czas*).
- W C++ w tym samym celu wykorzystuje się parametry referencyjne.
- Załóżmy na chwilę, że dana typu *double* jest duża i opłaca się ją przekazywać do wnętrz funkcji via wskaźnik, nie chcąc jednocześnie modyfikować jej zawartości we wnętrzu tej funkcji… .

Wskaźniki a przekazywanie parametrów — modyfikacja wewnątrz funkcji

- Załóżmy również, że korzystamy z napisanych przez *kogoś innego* funkcji, które *nie posiadają dokumentacji*, znamy tylko ich *prototypy*.

```
int main()
{
  double cena = 100;

  doliczVat23IWypisz( &cena );
  ksiegujKwoteNetto( &cena );
  . . .
}
```

- Prototypy:

```
void doliczVat23IWypisz( double * cenaNetto  );
void ksiegujKwoteNetto( double * cenaNetto  );
. . .
```

Czy obawy są uzasadnione?

Wskaźniki a przekazywanie parametrów — modyfikacja wewnątrz funkcji

Tak!

- Prototyp *wprost* nie mówi niczego o realizacji funkcji!

```
void doliczVat23IWypisz( double * cenaNetto  );
```

- Realizacja może być taka (źle):

```
void doliczVat23IWypisz( double * cenaNetto  )
{
  *cenaNetto *= 1.23;

  cout << *cenaNetto;
}
```

- Realizacja może też być taka (dobrze):

```
void doliczVat23IWypisz( double * cenaNetto  )
{
  cout << *cenaNetto * 1.23;
}
```

Wskaźniki a przekazywanie parametrów — modyfikator *const*

- Aby ustrzec się przed *niezamierzoną* modyfikacją parametru przekazywanego przez wskaźnik, można użyć słowa kluczowego *const*.
- Umieszczenie *const* ***przed*** typem obiektu wskazywanego jest *obietnicą* tego, że będzie on obiektem niemodyfikowalnym (*read-only*).
- Funkcja nieskutecznie usiłuje zmodyfikować obiekt wskazywany przez *cenaNetto*:

```
void doliczVat23IWypisz( const double * cenaNetto  )
{
  *cenaNetto *= 1.23;

  cout << *cenaNetto;
}
```

```
error: assignment of read-only location
```

- Funkcja nie modyfikuje obiektu wskazywanego przez *cenaNetto*:

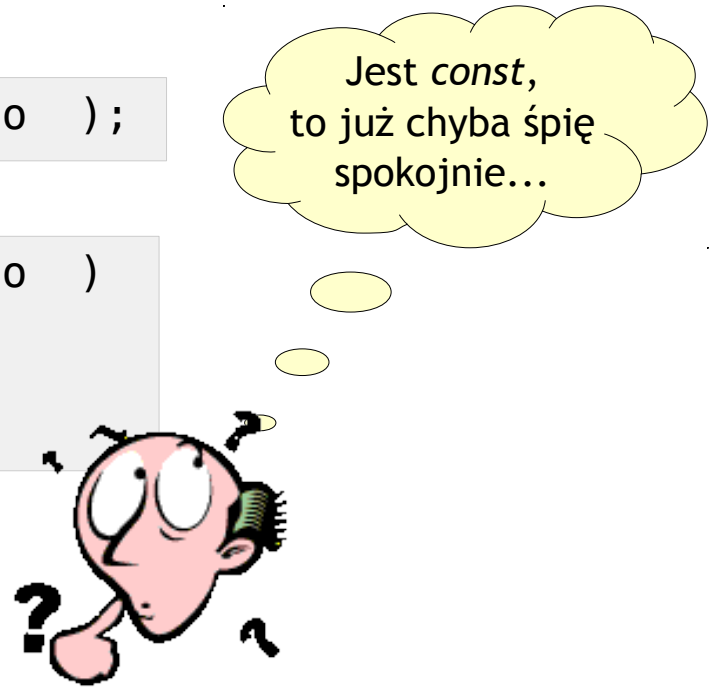
```
void doliczVat23IWypisz( const double * cenaNetto  )
{
  cout << *cenaNetto * 1.23;
}
```

Wskaźniki a przekazywanie parametrów — modyfikator *const*

- Wystąpienie słowa kluczowego *const* jest obietnicą, że wnętrze funkcji nie będzie modyfikować *obiektu wskazywanego*.
- Ta informacja umieszczona w prototypie pozwala oczekiwać, że parametr nie zostanie zmodyfikowany w sposób *niezamierzony*.

```
void doliczVat23IWypisz( const double * cenaNetto  );
```

```
void doliczVat23IWypisz( const double * cenaNetto  )
{
  cout << *cenaNetto * 1.23;
}
```



Czy rzeczywiście można spać spokojnie?

Wskaźniki a przekazywanie parametrów — modyfikator *const*

Nie!

- Wystąpienie słowa kluczowego *const* jest *obietnicą*, że wnętrze funkcji nie będzie modyfikować *obiektu wskazywanego* w sposób *niezamierzony*.
- Ale można w sposób *zamierzony* tej obietnicy *nie dotrzymać*!

```
void doliczVat23IWypisz( const double * cenaNetto  )
{
  *( ( double * ) cenaNetto ) *= 1.23;

  cout << *cenaNetto;
}
```

Rzutowanie wskaźnika typu
*const double **
na
*double **
„zdejmujące” atrybut *read-only*.

- Taka sytuacja to zamierzona złośliwość, całe szczęście nie spotyka się jej zbyt często.
- Ale skoro to nie jest niemożliwe... .

Parametry referencyjne w C++ też mogą być *const*

- Wystąpienie słowa kluczowego *const* jest obietnicą, że wnętrze funkcji nie będzie modyfikować *obiektu referencyjnego*.
- Ta informacja umieszczona w prototypie pozwala oczekiwać, że parametr nie zostanie zmodyfikowany w sposób *niezamierzony*.

```
void doliczVat23IWypisz( const double & cenaNetto  );
```

```
void doliczVat23IWypisz( const double & cenaNetto  )
{
  cout << cenaNetto * 1.23;
}
```

Czy rzeczywiście można spać spokojnie?

Parametry referencyjne w C++ też mogą być *const*

Nie!

- Wystąpienie słowa kluczowego *const* jest *obietnicą*, że wnętrze funkcji nie będzie modyfikować *obiektu referencyjnego* w sposób *niezamierzony*.
- Ale można w sposób *zamierzony* tej obietnicy *nie dotrzymać*!

```
void doliczVat23IWypisz( const double & cenaNetto  )
{
  ( double & )cenaNetto *= 1.23;

  cout << cenaNetto;
}
```

Rzutowanie referencji typu
const double &
na
double &
„zdejmujące” atrybut *read-only*.

- Zatem referencje w C++ też pozwalają nieźle zamieszać!

Wariacje na temat wskaźników i słowa kluczowego *const*

- Można modyfikować wartość wskaźnika *p*, można modyfikować obiekt wskazywany **p*:

```
int i = 10;
int * p; // Zwykły wskaźnik na zwykły obiekt
. . .
p = &i;     // Modyfikowalny wskaźnik
*p = 20;    // Modyfikowalny obiekt
cout << *p; // Wolno odczytywać wartość wskaźnika i obiektu
```

- Można modyfikować wartość wskaźnika *p*, *nie można* modyfikować obiektu wskazywanego **p*, który staje się obiektem tylko do odczytu:

```
int i = 10;
const int * p; // Zwykły wskaźnika na niemodyfikowalny obiekt
. . .
p = &i;     // Modyfikowalny wskaźnik
*p = 20;    // Niemodyfikowalny obiekt
cout << *p; // Wolno odczytywać wartość wskaźnika i obiektu
```

Wariacje na temat wskaźników i słowa kluczowego *const*

- *Nie można* modyfikować wartość wskaźnika *p*, *można* modyfikować obiekt wskazywany **p*, wskaźnik *p* jest zakotwiczony „na zawsze”:

```
int i = 10;
int * const p = &i; // Ustalony wskaźnika na modyfikowalny obiekt
. . .               // Inicjalizacja takiego wskaźnika jest obowiązkowa
p = &i;     // Niemodyfikowalny wskaźnik
*p = 20;    // Modyfikowalny obiekt
cout << *p; // Wolno odczytywać wartość wskaźnika i obiektu
```

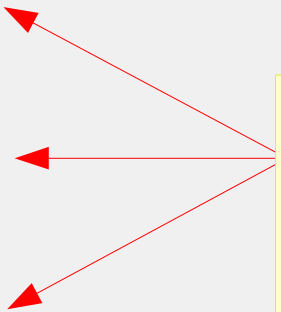
- *Nie można* modyfikować wartość wskaźnika *p*, *nie można* modyfikować obiektu wskazywanego **p*, wskaźnik *p* jest zakotwiczony „na zawsze” o obiekt *read-only*:

```
int i = 10;
const int * const p = &i; // Ustalony wskaźnika na niemodyfikowalny obiekt
. . .                     // Inicjalizacja takiego wskaźnika jest obowiązkowa
p = &i;     // Niemodyfikowalny wskaźnik
*p = 20;    // Modyfikowalny obiekt
cout << *p; // Wolno odczytywać wartość wskaźnika i obiektu
```

Wskaźniki typu void *

- Typ **`void *`** oznacza wskazanie niezwiązane z żadnym typem.
- Wskaźnik takiego typu może wskazywać daną dowolnego typu.

```
float  f = 2.5;
int    i = 5;
char   c = 'A';

void * ptr;


ptr = &f;
. . .

ptr = &i;
. . .

ptr = &c;
. . .
```



Wskaźnik *ptr* może pokazywać na obiekty różnych typów.

Uwaga! Po przypisaniu do wskaźnika typu *void* * tracimy informację o typie obiektu wskazywanego. Dlatego operacja **ptr* nie ma sensu — kompilator nie wie, czym jest obiekt wskazywany, ile zajmuje bajtów w pamięci operacyjnej. Wiadomo tylko, gdzie taki obiekt jest.

Wskaźniki typu void *, cd. ...

- Nie można wprost odwoływać się do obiektu wskazywanego przez wskaźnika *void* * — inaczej mówiąc, ***nie można*** dokonać *dereferencji* takiego wskaźnika.
- Aby odwołać się do obiektu wskazywanego, należy poinformować kompilator jaki jest jego typ, dokonując konwersji (tzw. *rzutowania*) typu wskaźnika.

```
void * ptr;

ptr = &f;
cout << endl << *( ( float * ) ptr );
```

Rzutowanie wskaźnika *ptr* – wskazanie na obiekt typu *float*.

```
void * ptr;

ptr = &f;
cout << endl << * ( float * ) ptr;
```

Wskaźniki typu void *, cd. ...

- Wskaźnik **`void *`** można rzutować na różne typy:

```
float f = 2.5;
int   i = 5;
char  c = 'A';

void * ptr;


ptr = &f;
cout << endl << *( ( float * )ptr );

ptr = &i;
cout << endl << *( ( int * )ptr );

ptr = &c;
cout << endl << *( ( char * )ptr );
```

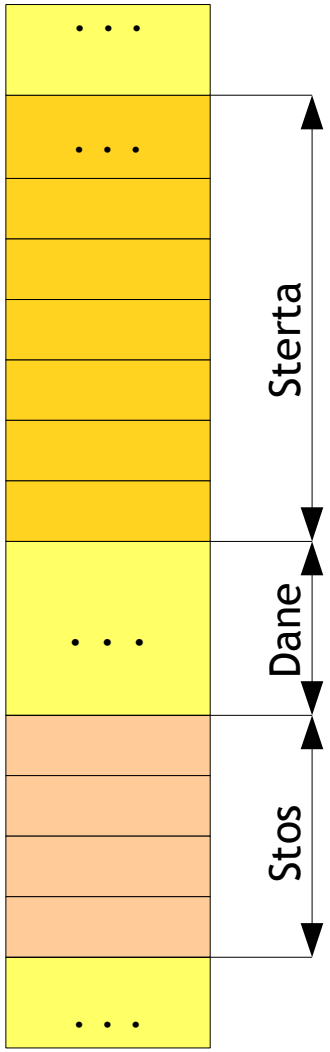
Dynamiczny przydział pamięci — pojęcie sterty, sterta a stos

Sterta (ang. *heap*) to wydzielony obszar *pamięci wolnej*:

- przeznaczony do przechowywania *danych dynamicznych*,
- *kontrolowany ręcznie* przez programistę,
- *ograniczony* pod względem rozmiaru,
- przydzielany pasującymi *fragmentami*.

Stos (ang. *stack*) to wydzielony obszar *pamięci roboczej*:

- przeznaczony do przechowywania *danych automatycznych*,
- *nie jest bezpośrednio kontrolowany* przez programistę,
- *ograniczony* pod względem rozmiaru,
- przydzielany wg. zasady LIFO (ang. *last in, first out*).



Dynamiczny przydział pamięci

Dynamiczny przydział pamięci polega na zarezerwowaniu fragmentu pamięci w obszarze *pamięci wolnej* (*sterty*), dla obiektu pamięciowego zwanego dynamicznym.

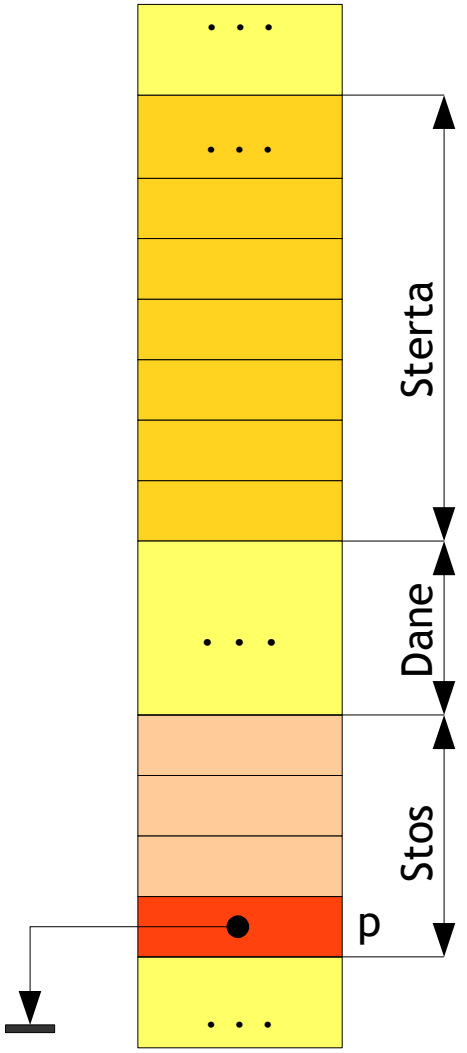
Typowy scenariusz wykorzystania dynamicznego przydziału pamięci:

- Określenie wielkości potrzebnego obszaru pamięci.
- Przydział pamięci i zapamiętanie wskazania tego obszaru w zmiennej wskaźnikowej.
- Sprawdzenie czy przydział pamięci się powiódł, jeżeli tak to:
  - Wykorzystanie przydzielonego bloku pamięci.
  - Zwolnienie przydzielonego bloku pamięci, gdy nie jest już potrzebny.

Dynamiczny przydział pamięci w języku C++ — etap I

Definicja zmiennej wskaźnikowej *p*, zainicjowanej wskaźnikiem pustym.

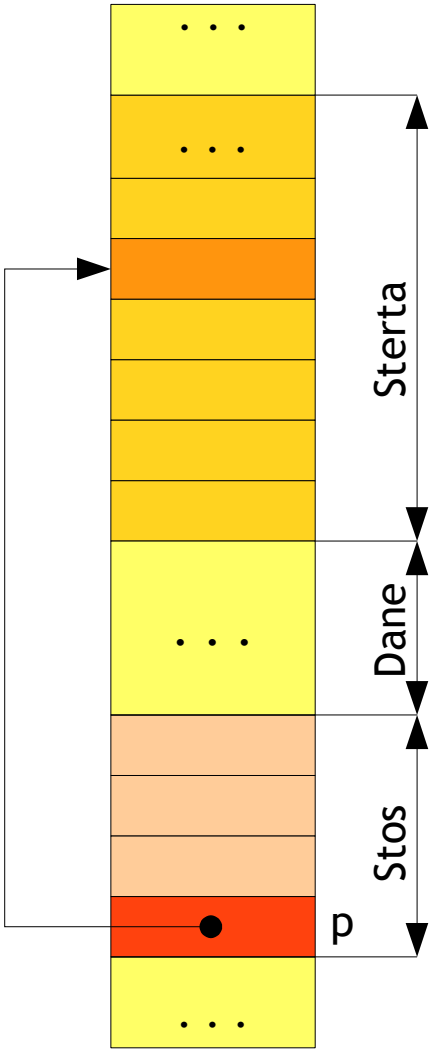
```
int main()
{
  int * p = 0;

  p = new int;

  if( p != 0 )
  {
    *p = 10;
    . . .
    cout << ++(*p);
    . . .
    delete p;
  }
  . . .
}
```



Dynamiczny przydział pamięci w języku C++ — etap II

Operator *new* przydziela na stercie blok pamięci dla danej typu *int*. Rezultatem funkcji jest wskaźnik do przydzielonego obszaru lub 0 jeżeli polecenie nie może być zrealizowane (czasem jest inaczej, o tym za chwilę).

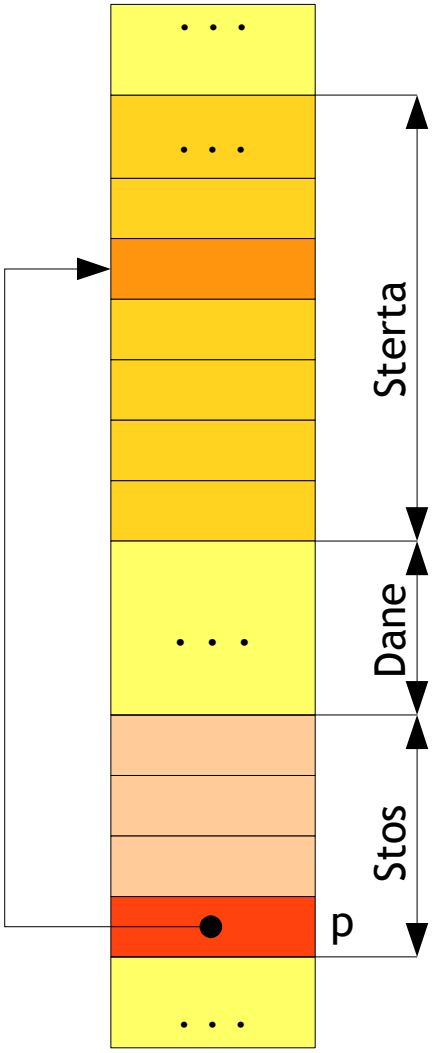
```
int main()
{
  int * p = 0;

  p = new int;

  if( p != 0 )
  {
    *p = 10;
    . . .
    cout << ++(*p);
    . . .
    delete p;
  }
  . . .
}
```



Dynamiczny przydział pamięci w języku C++ — etap III

Zawsze należy sprawdzić poprawność przydziału pamięci. Odwołanie do obiektu lokalizowanego przez wskaźnik pusty jest błędem.

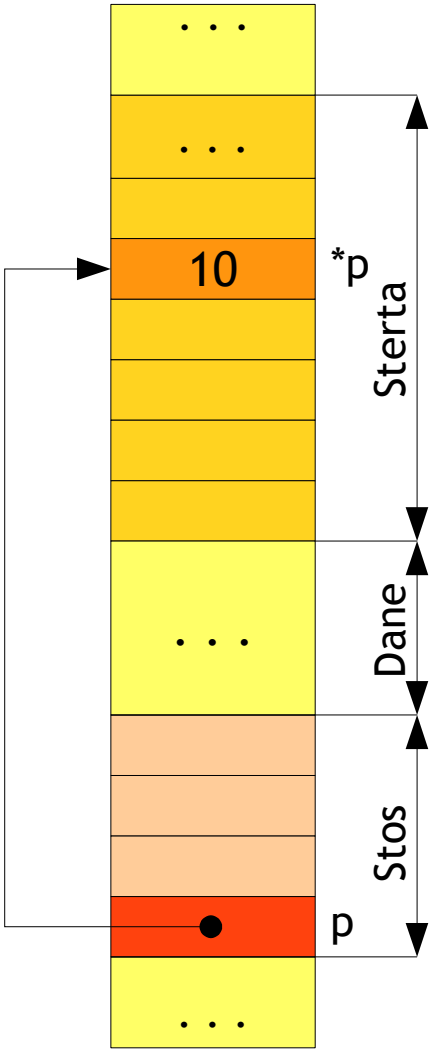
```
int main()
{
  int * p = 0;

  p = new int;

  if( p != 0 )
  {
    *p = 10;
    . . .
    cout << ++(*p);
    . . .
    delete p;
  }
  . . .
}
```



Dynamiczny przydział pamięci w języku C++ — etap IV

Wykorzystanie przydzielonego bloku pamięci. Ponieważ zmienna wskaźnikowa *p* jest skojarzona z typem *int*, przydzielony obszar traktowany jest jak dana typu *int*.

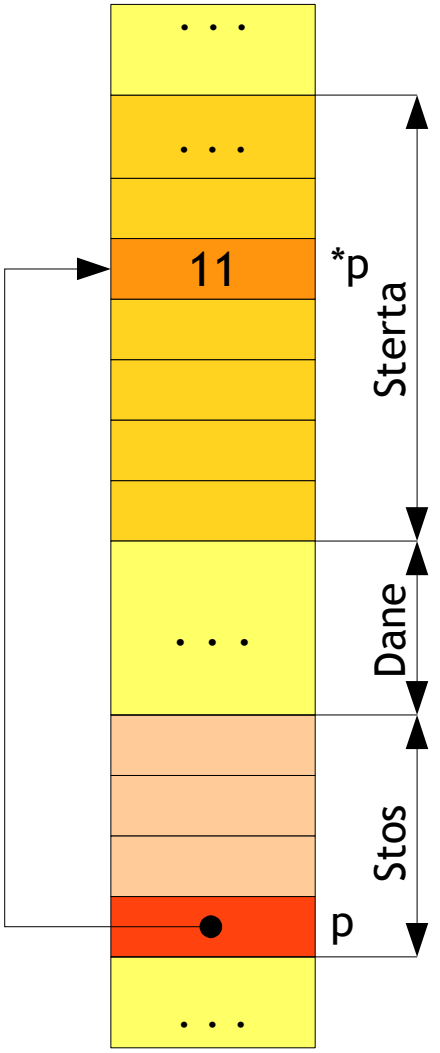
```
int main()
{
  int * p = 0;

  p = new int;

  if( p != 0 )
  {
    *p = 10;
    . . .
    cout << ++(*p);
    . . .
    delete p;
  }
  . . .
}
```



Dynamiczny przydział pamięci w języku C — etap IV, cd. ...

Z obiektem wskazywanym przez zmienną *p* można robić wszystko to, co dozwolone dla danej typu *int*. Wyrażenie `++(*p)` zwiększa obiekt wskazywany przez zmienną *p*.

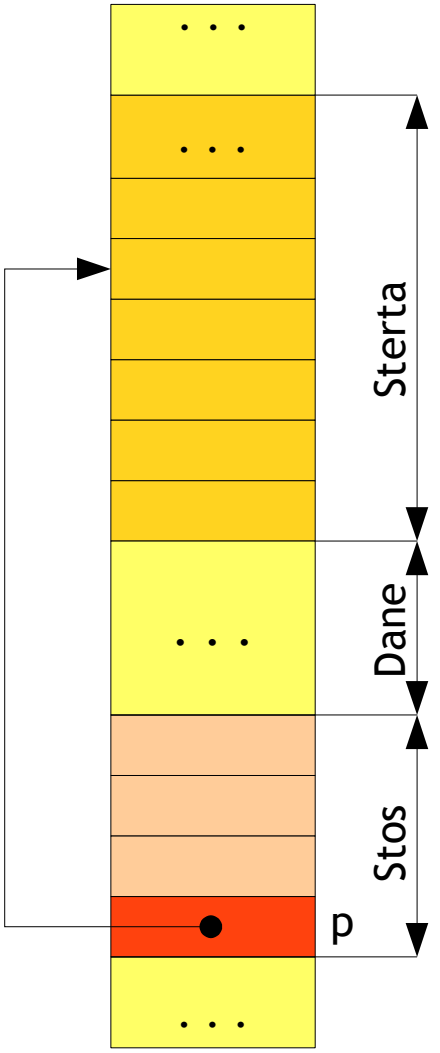
```
int main()
{
  int * p = 0;

  p = new int;

  if( p != 0 )
  {
    *p = 10;
    . . .
    cout << ++(*p);
    . . .
    delete p;
  }
  . . .
}
```



Dynamiczny przydział pamięci w języku C++ — etap V

Wywołanie operatora *delete* powoduje zwolnienie bloku pamięci wskazywanego przez *p*, blok ten zwracany jest do puli bloków wolnych. Uwaga — po wywołaniu *delete* wskaźnik *p* dalej pokazuje na zwolniony blok pamięci!

```
int main()
{
  int * p = 0;

  p = new int;

  if( p != 0 )
  {
    *p = 10;
    . . .
    cout << ++(*p);
    . . .
    delete p;
  }
  . . .
}
```



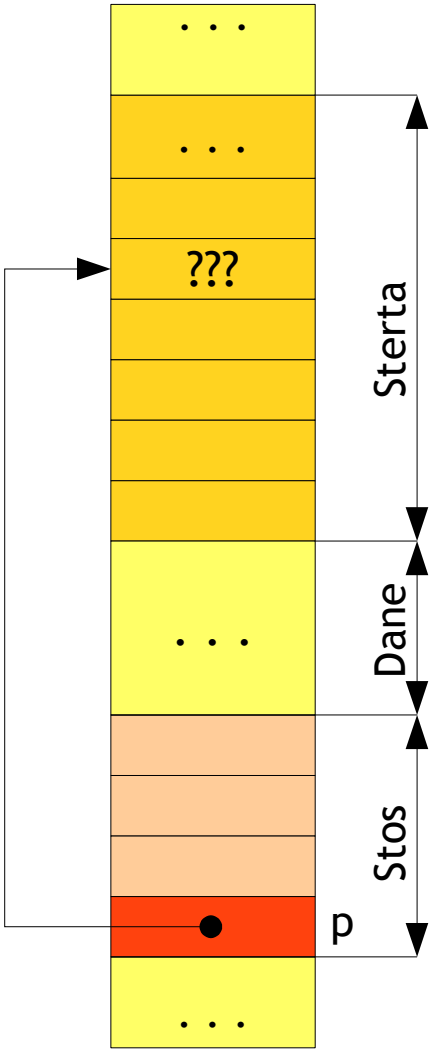
Dynamiczny przydział pamięci w języku C++ — uwagi

Mimo, że po wywołaniu *free* wskaźnik *p* dalej pokazuje na zwolniony obszar, próba odwołania się do niego *jest błędem*.

Ten obszar być może został właśnie przydzielony ponownie.

```
int main()
{
  int * p = 0;

  p = new int;

  if( p != 0 )
  {
    *p = 10;
    . . .
    cout << ++(*p);
    . . .
    delete p;
    *p = 0;
  }
  *p = 100;
  . . .
}
```

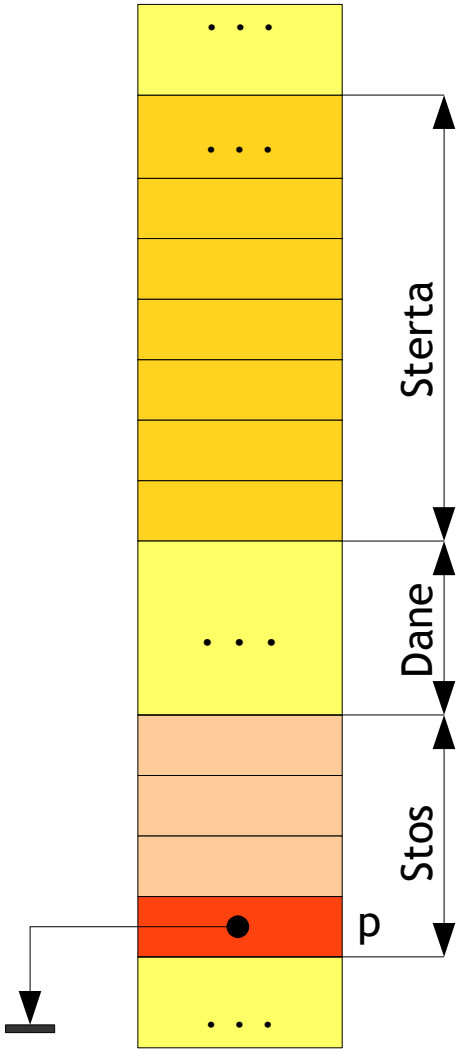
Błąd. Odwołanie do zwolnionego lub nieprzydzielonego bloku



Dynamiczny przydział pamięci w języku C++ — uwagi, cd. ...

Dobrą praktyką jest zerowanie zmiennych wskaźnikowych na etapie ich deklaracji, po zwolnieniu pamięci oraz testowanie czy wskaźnik nie jest pusty przed odwołaniem do obiektu wskazywanego.

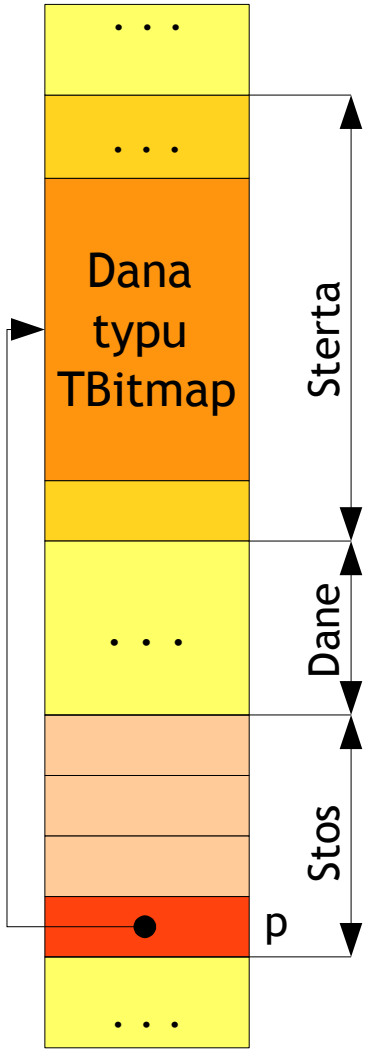
```
int main()
{
  int * p = 0;

  p = new int;

  if( p != 0 )
  {
    *p = 10;
    . . .
    cout << ++(*p);
    . . .
    delete p;
    p = 0;
  }
  . . .
}
```



Dynamiczny przydział pamięci w języku C++ — dla int nie ma sensu...

Dynamiczny przydział pamięci dla pojedynczych danych typu *int*, *char* czy *double* najczęściej nie ma sensu. Ale ma sens dla obiektów zajmujących dużo pamięci operacyjnej oraz dla złożonych struktur danych.

```
int main()
{
  TBitmap * p = 0;

  p = new TBitmap;

  if( p != 0 )
  {
    Operacje na bitmapie
    wskazywanej przez p;
    . . .
    delete p;
    p = 0;
  }
  . . .
}
```



Dynamiczny przydział pamięci w języku C++, zaszłości

Od standardu C++ z 2003 operator new działa inaczej. Aby zachować omówiony styl przydziału pamięci, należy użyć jego specjalnej wersji: *nothrow*.

Stare dzieje w C ++

```
int main()
{
  int * p = 0;

  p = new int;

  if( p != 0 )
  {
    *p = 10;
    . . .
    cout << ++(*p);
    . . .
    delete p;
    p = 0;
  }
  . . .
}
```

Aktualnie w C ++

```
int main()
{
  int * p = 0;

  p = new (nothrow) int;

  if( p != 0 )
  {
    *p = 10;
    . . .
    cout << ++(*p);
    . . .
    delete p;
    p = 0;
  }
  . . .
}
```

Aktualnie, gdy operator *new* nie potrafi przydzielić pamięci to generuje *wyjątek*, zamiast oddawania rezultatu w postaci wskaźnika pustego.

Dynamiczny przydział pamięci w języku C++, wyjątki

Jeżeli aktualnie użyjemy operatora w wersji **new**, wygenerowany zostanie wyjątek klasy **bad_alloc**.

Aktualnie w C ++

```
int main()
{
  try
  {
    int * p = new int;

    *p = 10;
    . . .
    cout << ++(*p);
    . . .
    delete p;
    p = 0;
  }
  catch( ... )
  {
    // Zrob cos gdy brak pamieci
  }
}
```

Mechanizm obsługi *wyjątków* oraz zasady stosowania *try-catch* zostaną omówione osobno.

Zarządzanie pamięcią dynamiczną to rzecz podwójnie nieprosta

Zarządzanie pamięcią rozgrywane na poziomie kodu programu wymaga uwagi od programisty. To pierwsza rzecz.

Druga jest po stronie systemu operacyjnego. Uczestniczy on w przydziale pamięci dla procesów, oferując *pamięć wirtualną*. W rzeczywistości bloki pamięci naszego programu mogą czasem znajdować się na dysku... . Proces zarządzania pamięcią wirtualną bywa czasem bardzo złożony.

Interesujące artykuły na temat zarządzania pamięcią:

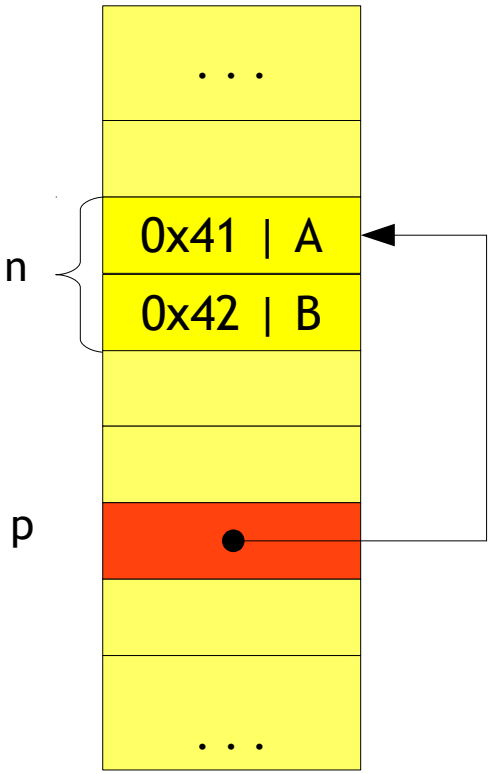
- `http://www.cprogramming.com/tutorial/virtual_memory_and_heaps.html`
- `http://www.ibm.com/developerworks/linux/library/l-memory/`
- `http://www.cantrip.org/wave12.html`
- `http://linuxdevcenter.com/pub/a/linux/2003/05/08/cpp_mm-1.html`

Wyrażenia wskaźnikowe

Wskaźniki *lokalizują* obiekty w *pamięci operacyjnej*. Można budować *wyrażenia* zawierające wskaźniki, *wyrażenia* te *lokalizują* również pewne obiekty w pamięci operacyjnej.

W językach C/C++ obowiązuje specjalna ***arytmetyka na wskaźnikach***.

```
// ASCII: 0x41 - A 0x42 - B
short int n = 0x4241;
char * p;

p = ( char * )&n;

cout << endl << *p;

++p;

cout << endl << *p;
```

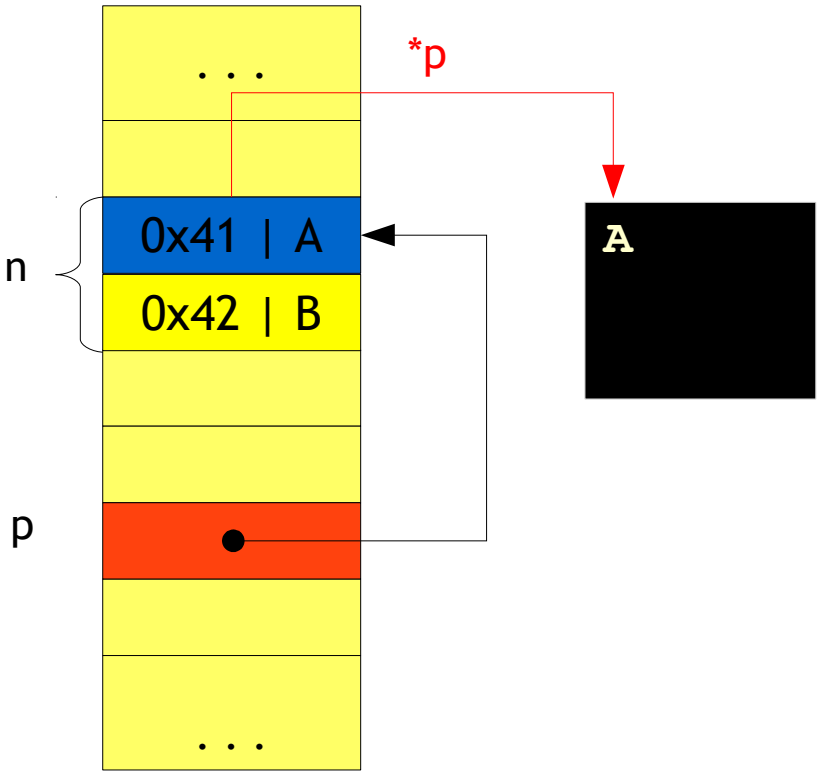


Wyrażenia wskaźnikowe

Wskaźnik *p* lokalizuje młodszy bajt zmiennej *m*. Reszta tej liczby nie jest dla wskaźnika *p* „widoczna” ponieważ służy on do lokalizowania znaków (bajtów).

Wyprowadzenie obiektu wskazywanego przez *p* do *cout* spowoduje potraktowanie młodszego bajtu zmiennej *m* jako znaku i wyprowadzenie go do strumienia wyjściowego.

```
// ASCII: 0x41 - A 0x42 - B
short int n = 0x4241;
char * p;

p = ( char * )&n;

cout << endl << *p;

++p;

cout << endl << *p;
```

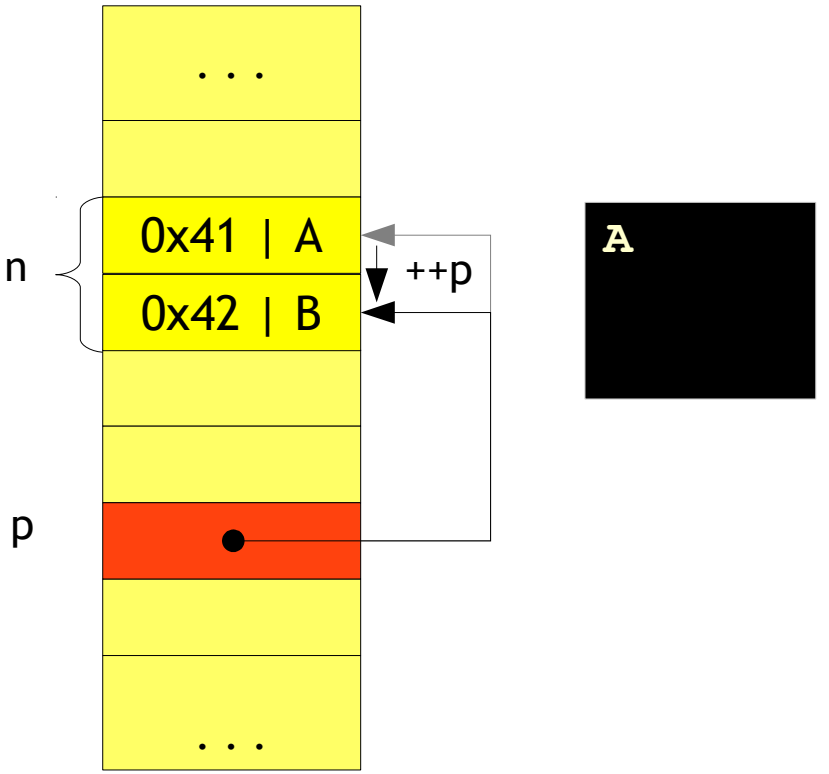


Wyrażenia wskaźnikowe

Do zmiennej wskaźnikowej wolno *dodać* (*odjąć*) liczbę całkowitą. Takie wyrażenie lokalizuje w pamięci operacyjnej obiekt przesunięty w stosunku do wskaźnika bazowego. Wyrażenie $p = p + 1$ przesuwa wskaźnik do następnego obiektu w pamięci, wyrażenie $p = p - 1$ do poprzedniego obiektu, zgodnie z typem wskaźnika.

Wyrażenia te można oczywiście zapisać: $++p$ oraz $--p$.

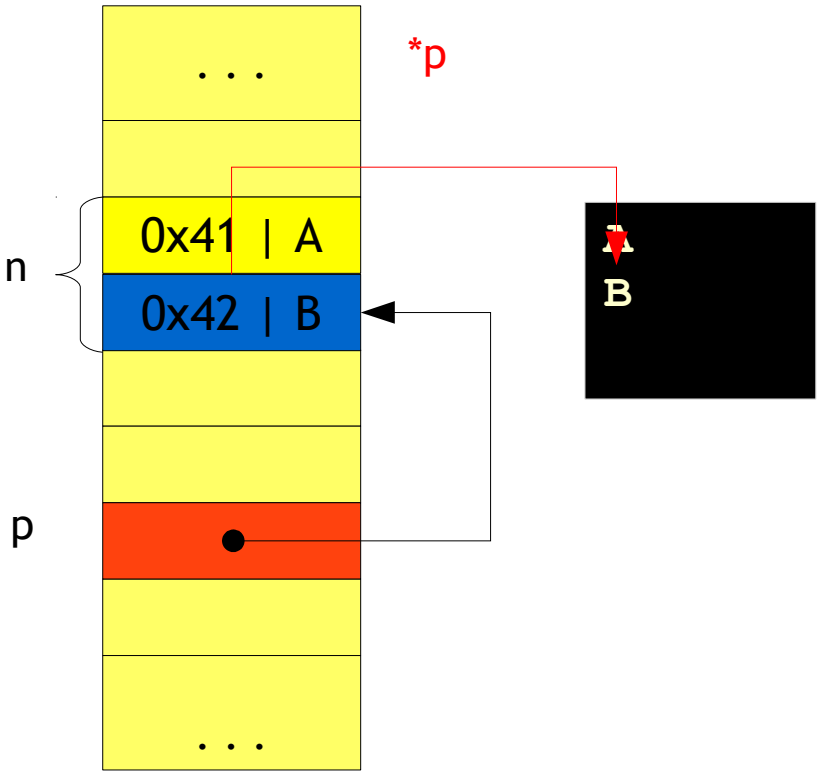
```
// ASCII: 0x41 - A 0x42 - B
short int n = 0x4241;
char * p;

p = ( char * )&n;

cout << endl << *p;

++p;

cout << endl << *p;
```



Wyrażenia wskaźnikowe

Liczby dodawane lub odejmowana od wskaźnika są *skalowane* przez rozmiar *typu wskaźnika*. Oznacza to, że dla *char * p* operacja *++p* spowoduje przesunięcie wskaźnika do następnego znaku w pamięci operacyjnej, a dla *int * p* operacja *++p* spowoduje przesunięcie wskaźnika do następnej liczby całkowitej, itp.

```
// ASCII: 0x41 - A 0x42 - B
short int n = 0x4241;
char * p;

p = ( char * )&n;

cout << endl << *p;

++p;

cout << endl << *p;
```



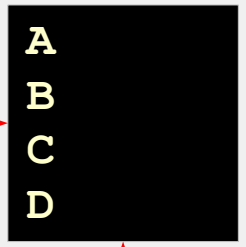
Arytmetyka na wskaźnikach — zasady

Dozwolone operacje wskaźnikowe to:

- przypisywanie wskaźników do obiektów tego *samego typu*,
- przypisywanie wskaźników do obiektów *innego typu* po *konwersji*,
- dodawanie lub odejmowanie *wskaźnika i liczby całkowitej*,
- *odejmowanie* lub *porównanie* dwóch wskaźników,
- przypisanie wskaźnikowi *wartości zero* (lub wskazania puste *NULL*) lub *porównanie ze wskazaniem pustym*.

Wyrażenia wskaźnikowe — kolejny przykład

```
// ASCII: 0x41 - A 0x42 - B 0x43 - C 0x44 - D
int m = 0x44434241; // Zakładamy, że int jest 32 bitowy
char * p;

p = ( char * )&m;
cout << endl << *p ;
++p;
cout << endl << *p ;
++p;
cout << endl << *p ;
++p;
cout << endl << *p ;
```



lub:

```
// ASCII: 0x41 - A 0x42 - B 0x43 - C 0x44 - D
int m = 0x44434241; // Zakładamy, że int jest 32 bitowy
char * p;

int i;

for( i = 0, p = ( char * )&m; i < sizeof( int ); i++ )
  cout << endl << *( p++ );
```

Dla dociekliwych — funkcja ze zmienną liczbą parametrów

```
int addInts( int count, ... )
{
  int total = 0;
  va_list argList;

  va_start( argList, count );
  for( ; count; count-- )
    total += va_arg( argList, int );
  va_end( argList );
  return total;
}

. . .

cout << endl << "Suma: " << addInts( 2, 1, 2 );
cout << endl << "Suma: " << addInts( 3, 4, -1, 6 );
cout << endl << "Suma: " << addInts( 0 );
cout << endl << "Suma: " << addInts( 5, 1, 2, 3, 4, 5 );
```

```
Suma: 3
Suma: 9
Suma: 0
Suma: 15
```

Dla dociekliwych — funkcja ze zmienną liczbą parametrów

Aby obsłużyć zmienną liste parametrów nie trzeba koniecznie używać makr z pliku *stdarg.h*, wystarczy wiedzieć jak są przekazywane parametry i rozumieć wskaźniki... .

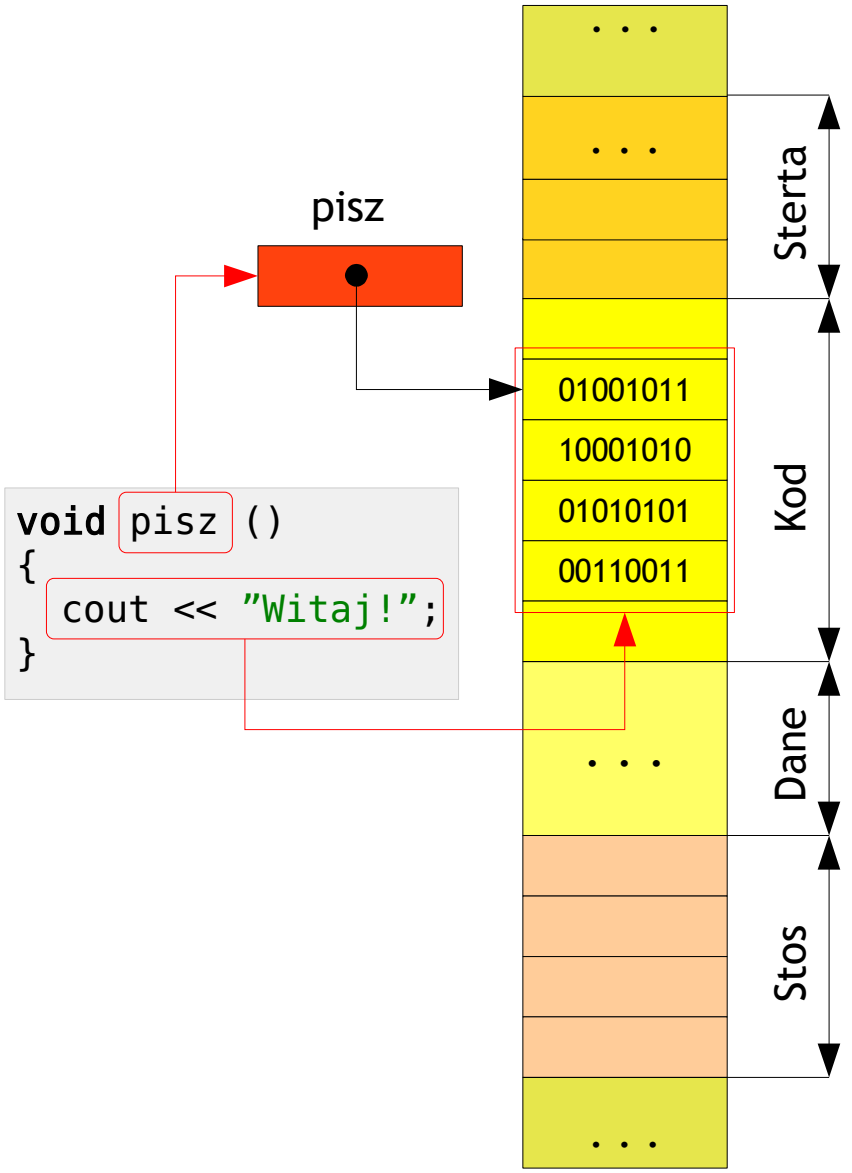
Poniżej przykład do indywidualnego przemyślenia.

```
int addIntsOwn( int count, ... )
{
  int total = 0;
  char * argList;

  argList = (( char * )&count ) + sizeof( count );
  for( ; count; count-- )
    total += *( ( int * )( ( argList += sizeof( int ) ) - sizeof( int ) ) );
  return total;
}

. . .

cout << endl << "Suma: " << addInts( 2, 1, 2 );
cout << endl << "Suma: " << addInts( 3, 4, -1, 6 );
cout << endl << "Suma: " << addInts( 0 );
cout << endl << "Suma: " << addInts( 5, 1, 2, 3, 4, 5 );
```

```
Suma: 3
Suma: 9
Suma: 0
Suma: 15
```

Wskaźniki do funkcji — koncepcja

- W trakcie uruchamiania programu, jego kod maszynowy odczytywany z pliku wykonywalnego, jest ładowany do pamięci operacyjnej.
- Każda funkcja w programie posiada określony adres, począwszy od tego adresu rozpoczyna się ciało funkcji w postaci kodu maszynowego.
- Nazwa funkcji w językach C/C++ jest właśnie adresem funkcji w pamięci operacyjnej.
- Skoro funkcje posiadają swoje adresy, to za możliwe jest operowanie na adresach funkcji z wykorzystaniem zmiennych wskaźnikowych.



Wskaźniki do funkcji — jak deklarować

- Wskaźniki do funkcji są deklarowane w specyficzny sposób.
- W deklaracji wskaźnika do funkcji należy precyzyjnie określić informacje o funkcji, jaką będzie mógł dany wskaźnik lokalizować.
- Te informacje obejmują:
  - typ rezultatu funkcji,
  - liczbę i typy kolejnych parametrów,
- Nazwy parametrów są nieistotne.

Wskaźniki do funkcji — jak deklarować

- Zakładamy, że wskazywana funkcja ma następującą definicję:

```
void pisz()
{
  cout << "Witaj!";
}
```

- Zmienną wskaźnikowa, która może lokalizować taką funkcję, deklarujemy :

```
void ( *funPtr )();
```

co oznacza, że *funPtr* jest wskaźnikiem na bezparametrowe funkcje, które nie mają rezultatu (rezultat typu **void**).

- Nawiasy są *niezbędne*, bez nich następująca deklaracja:

```
void *funPtr();
```

oznaczała by, że *funPtr* to nazwa bezparametrowej funkcji, której rezultatem jest wskaźnik typu **void ***.

Wskaźniki do funkcji — „kotwiczenie” wskaźnika

- Wskaźnik do funkcji może być zerowany na etapie deklarowania:

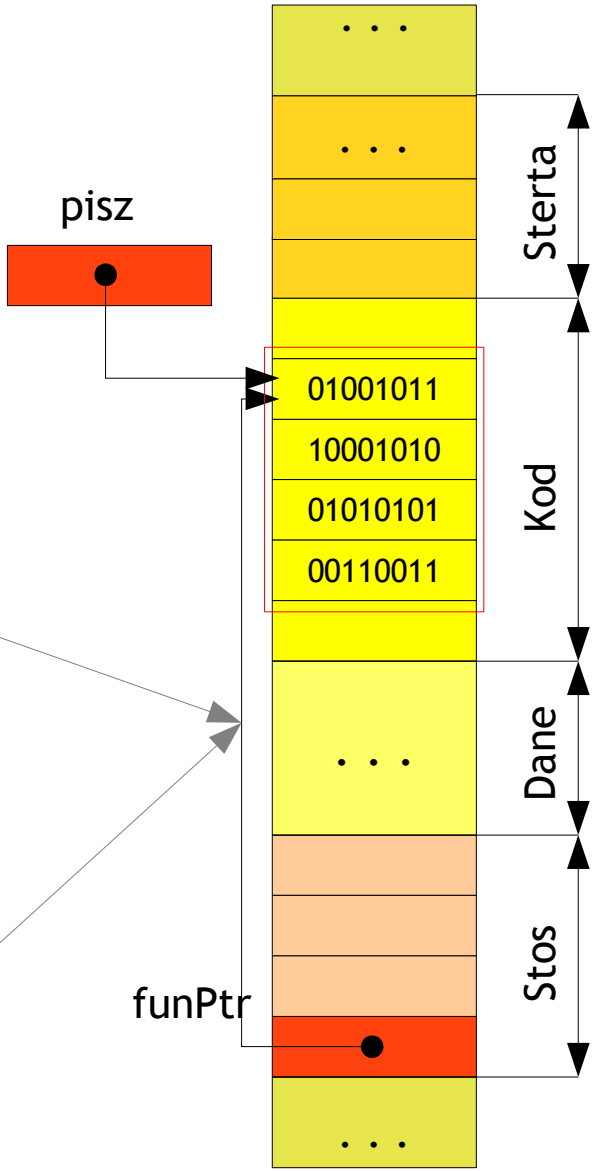
```
void ( *funPtr )() = 0;
```

- Wskaźnik do funkcji może być inicjowany na etapie deklarowania:

```
void ( *funPtr )() = &pisz; // Wersja nr 1

void ( *funPtr )() = pisz;  // Wersja nr 2
```

- Po deklaracji, do wskaźnika można przypisywać wskazanie na funkcje pisząc:

```
funPtr = &pisz;    // Wersja nr 2

funPtr = pisz;     // Wersja nr 2
```



Wskaźniki do funkcji — wywołanie funkcji via wskaźnik

- Po „zakotwiczeniu” wskaźnika o funkcję:

```
funPtr = pisz;
```

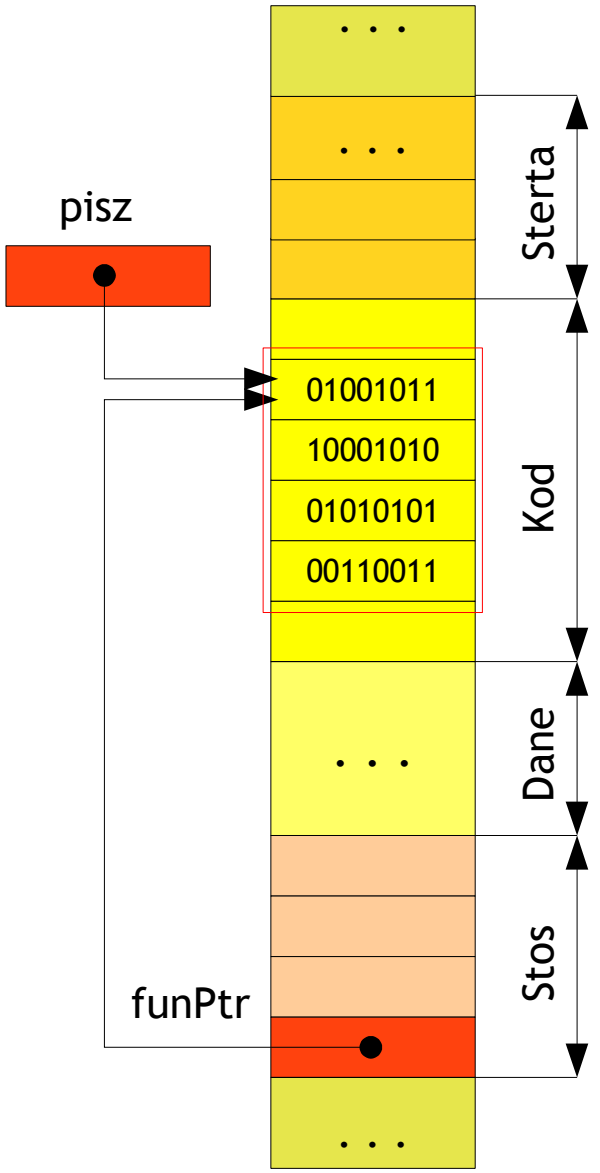
można wywołać jej kod pisząc:

```
( *funPtr )(); // Wersja nr 1
```

lub:

```
funPtr();      // Wersja nr 2
```

- W sensie *semantycznym* wskaźnik na funkcję i nazwa funkcji są tożsame, zatem wersja 1 powyżej jest niepotrzebnie skomplikowana, większość programistów wykorzysta wersję nr 2.



Wskaźniki do funkcji — wskaźniki bywają niezbyt wierne...

- Wskaźnik zadeklarowany w ten sposób:

```
void ( *funPtr )() = 0;
```

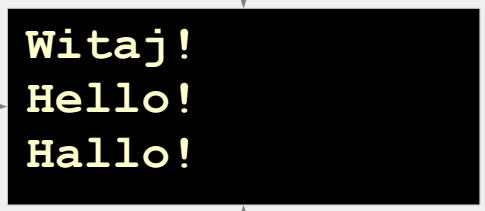
tak na prawdę, może pokazywać na dowolną bezparametrową funkcję, która nie ma rezultatu (rezultat typu **void**).

```
void pisz()
{
  cout << "\nWitaj!";
}
```

```
void write()
{
  cout << "\nHello!";
}
```

```
void schreiben()
{
  cout << "\nHallo!";
}
```

```
funPtr = pisz;
funPtr();

funPtr = write;
funPtr();

funPtr = schreiben;
funPtr();
```



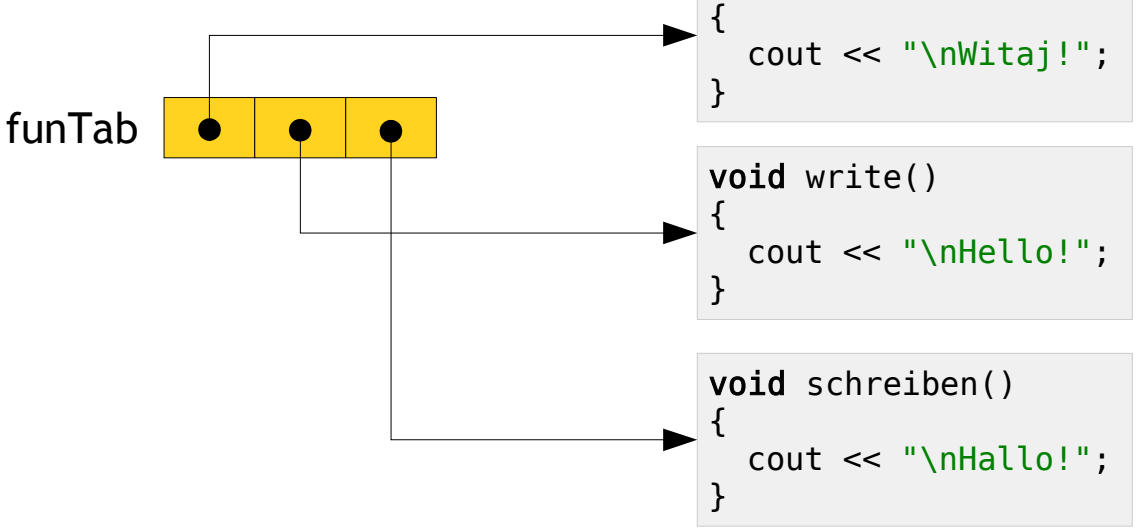
Wskaźniki do funkcji — tablicujemy kod?

- Tablica wskaźników na bezparametrowe funkcje nieudostępniające rezultatu:

```
const int N = 3;
void ( * funTab[ N ] )();
```

- „Zakotwiczenie” kolejnych elementów tablicy o funkcje:

```
funTab[ 0 ] = pisz;
funTab[ 1 ] = write;
funTab[ 2 ] = schreiben;
```



- Wywołanie funkcji, lokalizowanych przez kolejne elementy tablicy:

```
for( int i = 0; i < N; ++i )
  funTab[ i ]();
```

Wskaźniki do funkcji — inicjalizacja tablicy wskaźników funkcyjnych

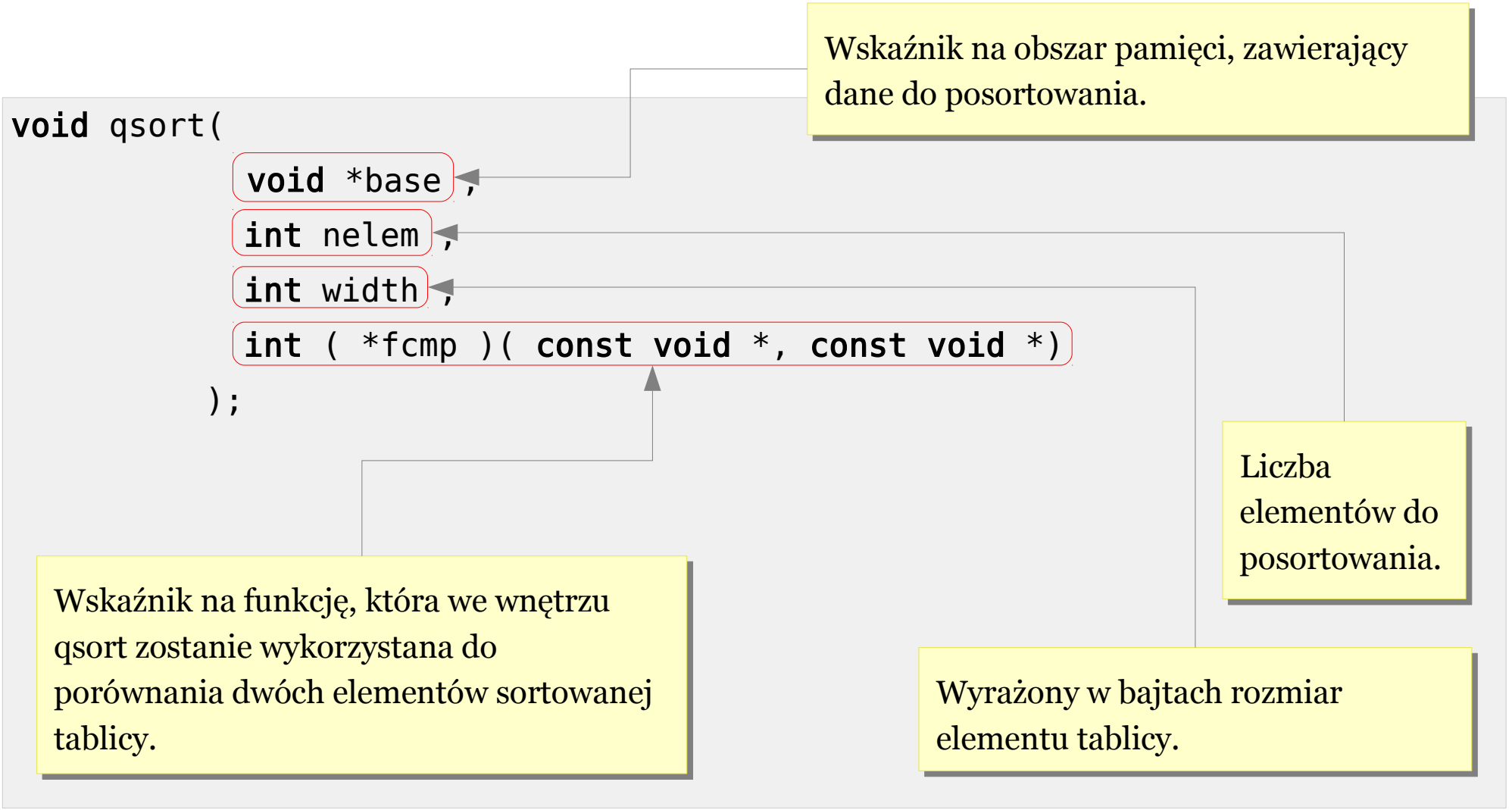
```
. . .
void pisz ()
{
  cout << "\nWitaj!";
}

void write()
{
  cout << "\nHello!";
}

void schreiben()
{
  cout << "\nHallo!";
}

int main()
{
  const int N = 3;
  void ( * funTab[ N ] )() = { pisz, write, schreiben };

  for( int i = 0; i < N; ++i )
    funTab[ i ]();

  . . .
}
```

Wskaźniki do funkcji — a po co to wszystko?

- Jest wiele bardzo ciekawych zastosowań wskaźników do funkcji. Ich przykłady będą sukcesywnie omawiane.
- Jednym z nich określanie funkcji, która ma być wywołana we wnętrzu innej funkcji.
- Przykład — sortowanie tablic z wykorzystaniem bibliotecznej funkcji *qsort* (wymaga włączenia *stdlib.h* lub *cstdlib*).
- Funkcja *qsort* pozwala na sortuje metodą *quick sort* dowolną tablicę.
- Funkcja *qsort* to *kwintesencja* wykorzystania wskaźników, również do funkcji.

Wskaźniki do funkcji — *qsort*

- Prototyp funkcji *qsort* (może różnić się w zależności od kompilatora):



Wskaźniki do funkcji — *qsort* w akcji

```
#include <cstdlib>
#include <iostream>
using namespace std;

?

int main()
{
  const int N = 5;
  int tab[ N ] = { 5, 3, 4, 1, 2 };

  qsort( tab, N, sizeof( tab[ 0 ] ), compInt );

  for( int i = 0; i < N; ++i )
    cout << endl << tab[ i ];

  . . .
}
```

Wskaźniki do funkcji — *qsort*, rola funkcji porównującej

- Funkcja *qsort* musi porównywać ze sobą pary elementów. Jednak funkcja ta przecież nie wie, jakie są elementy sortowanej tablicy.
- Programista musi zdefiniować odpowiednią funkcję porównującą i przekazać wskaźnik do tej funkcji do wnętrza funkcji *qsort*.
- Funkcja porównująca powinna mieć następującą postać:

```
int jakasNazwa( const void * a, const void * b )
{
  . . .
}
```

- Parametry *a* i *b* to wskaźniki na elementy do porównania. Rezultatem funkcji powinna być:
  - wartość ujemna gdy $a < b$,
  - wartość zero gdy $a == b$,
  - wartość dodatnia gdy $a > b$.

Wskaźniki do funkcji — *qsort* w akcji, funkcja porównująca

```
#include <cstdlib>
#include <iostream>
using namespace std;

int compInt( const void * a, const void * b )
{
  return ( *( int * )a ) - ( *( int *)b ) );
}

int main()
{
  const int N = 5;
  int tab[ N ] = { 5, 3, 4, 1, 2 };

  qsort( tab, N, sizeof( tab[ 0 ] ), compInt );

  for( int i = 0; i < N; ++i )
    cout << endl << tab[ i ];

  . . .
}
```

Suplement I: dynamiczny przydział pamięci w języku C

Przydział pamięci realizują funkcje:

- `void * malloc( size_t size )`
- `void * calloc( size_t nitems, size_t size )`
- `void * realloc( void * ptr, size_t size )`

Obszary pamięci przydzielone tymi funkcjami należało zwolnić funkcją:

- `void free( void * ptr )`

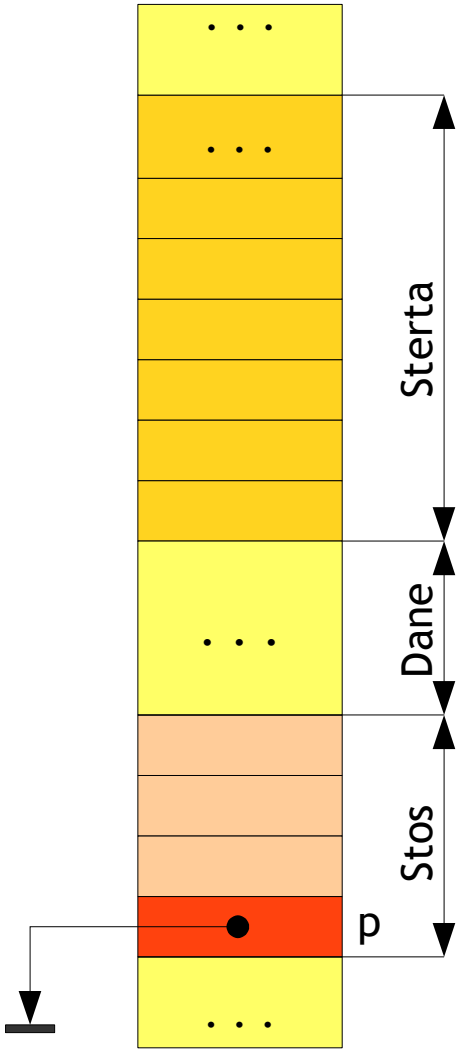
Funkcje zarządzające przydziałem/zwalnianiem bloków pamięci operują na wskaźnikach `void *`. Przydzielane bloki są amorficzne — są to „kawałki” pamięci o rozmiarze liczonym w bajtach.

Wykorzystanie powyższych funkcji wymaga włączenia pliku nagłówkowego dyrektywą *#include <stdlib.h>* lub *#include <cstdlib>* w C++.

Suplement I: dynamiczny przydział pamięci w języku C — etap I

Definicja zmiennej wskaźnikowej *p*, zainicjowanej wskaźnikiem pustym.

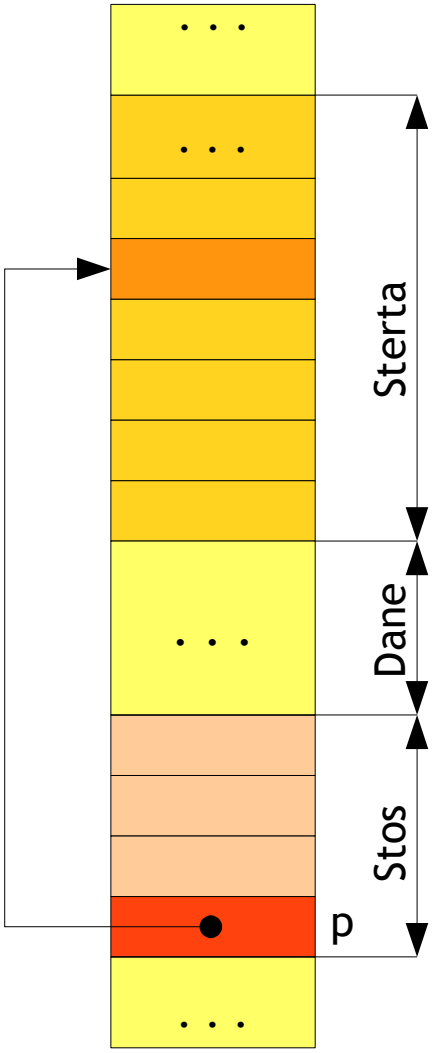
```
int main()
{
  int * p = NULL;

  p = malloc( sizeof( int ) );

  if( p != NULL )
  {
    *p = 10;
    . . .
    cout << ++(*p);
    . . .
    free( p );
  }
  . . .
}
```



Suplement I: dynamiczny przydział pamięci w języku C — etap II

Funkcja *malloc* przydzielan na stercie blok pamięci o rozmiarze *sizeof*(*int*). Rezultatem funkcji jest wskaźnik do przydzielonego obszaru lub NULL jeżeli polecenie nie może być zrealizowane.

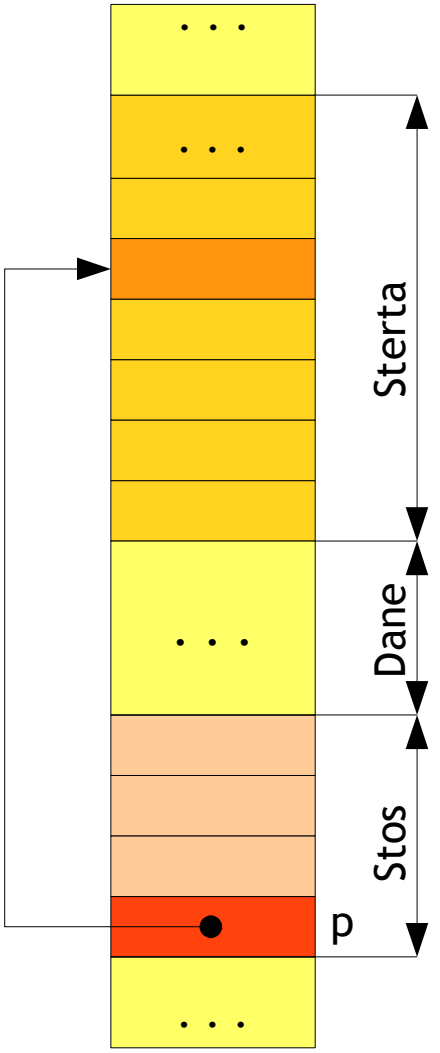
```
int main()
{
  int * p = NULL;

  p = malloc( sizeof( int ) );

  if( p != NULL )
  {
    *p = 10;
    . . .
    cout << ++(*p);
    . . .
    free( p );
  }
  . . .
}
```



Suplement I: dynamiczny przydział pamięci w języku C — etap III

Zawsze należy sprawdzić poprawność przydziału pamięci.
Odwołanie do wskaźnika pustego jest błędem.

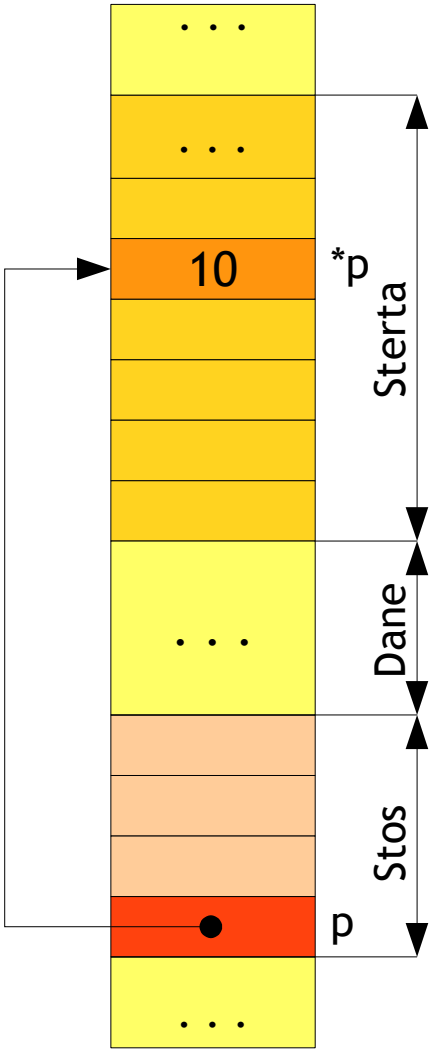
```
int main()
{
  int * p = NULL;

  p = malloc( sizeof( int ) );

  if( p != NULL )
  {
    *p = 10;
    . . .
    cout << ++(*p);
    . . .
    free( p );
  }
  . . .
}
```



Suplement I: dynamiczny przydział pamięci w języku C — etap IV

Wykorzystanie przydzielonego bloku pamięci. Ponieważ zmienna wskaźnikowa *p* jest skojarzona z typem *int*, przydzielony obszar traktowany jest jak dana typu *int*.

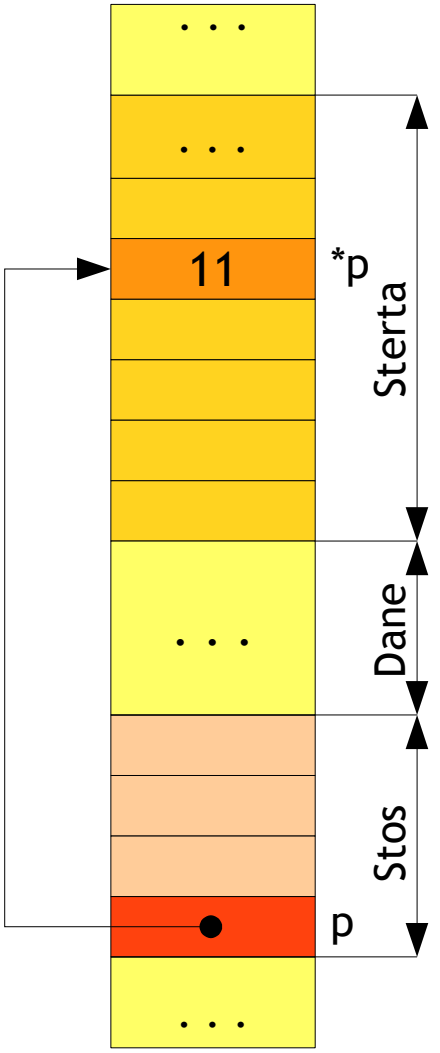
```
int main()
{
  int * p = NULL;

  p = malloc( sizeof( int ) );

  if( p != NULL )
  {
    *p = 10;
    . . .
    cout << ++(*p);
    . . .
    free( p );
  }
  . . .
}
```



Suplement I: dynamiczny przydział pamięci w języku C — etap IV, cd. ...

Z obiektem wskazywanym przez zmienną *p* można robić wszystko to, co dozwolone dla danej typu *int*. Wyrażenie **`++(*p)`** zwiększa obiekt wskazywany przez zmienną *p*.

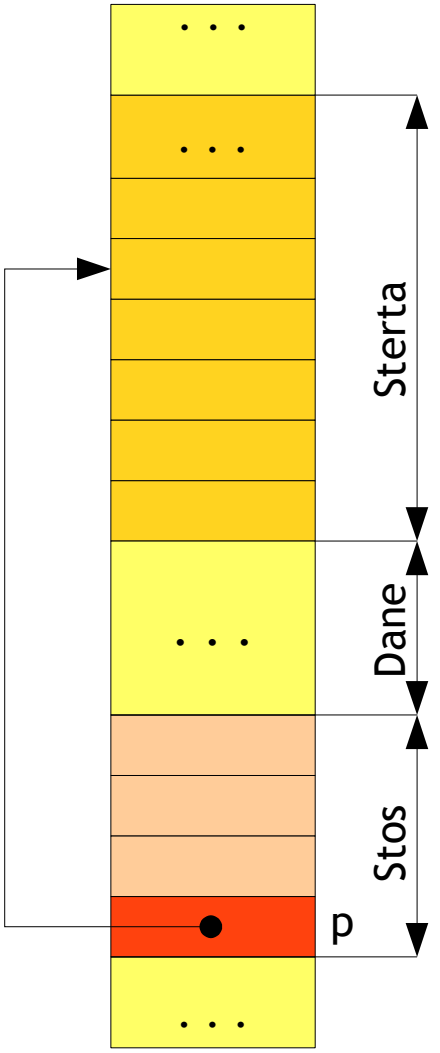
```
int main()
{
  int * p = NULL;

  p = malloc( sizeof( int ) );

  if( p != NULL )
  {
    *p = 10;
    . . .
    cout << ++(*p);
    . . .
    free( p );
  }
  . . .
}
```



Suplement I: dynamiczny przydział pamięci w języku C — etap V

Wywołanie funkcji *free* powoduje zwolnienie bloku pamięci wskazywanego przez *p*, blok ten zwracany jest do puli bloków wolnych. Uwaga — po wywołaniu *free* wskaźnik *p* dalej pokazuje na zwolniony blok pamięci!

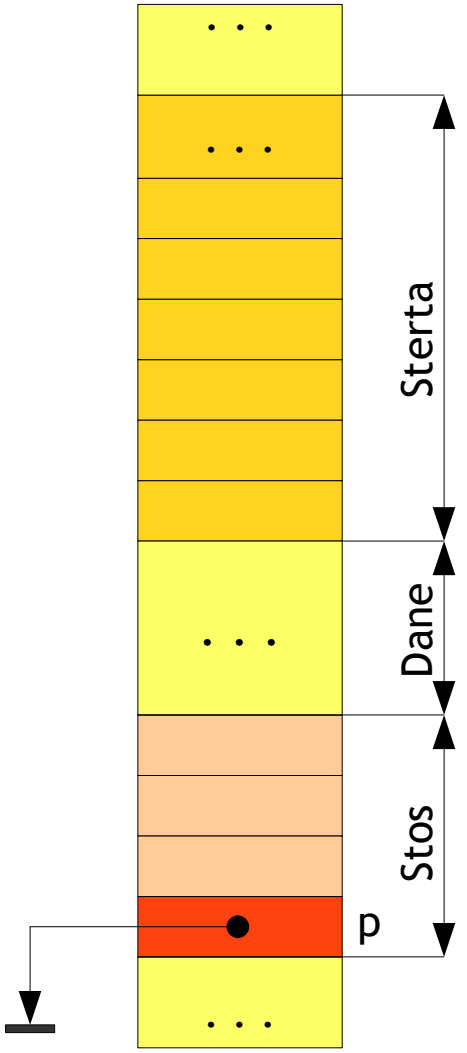
```
int main()
{
  int * p = NULL;

  p = malloc( sizeof( int ) );

  if( p != NULL )
  {
    *p = 10;
    . . .
    cout << ++(*p);
    . . .
    free( p );
  }
  . . .
}
```



Suplement I: dynamiczny przydział pamięci w języku C — uwagi

W C do zerowania wskaźnika wykorzystuje się zwyczajowo stała symboliczną *NULL*.

```
int main()
{
  int * p = NULL;

  p = malloc( sizeof( int ) );

  if( p != NULL )
  {
    *p = 10;
    . . .
    cout << ++(*p);
    . . .
    free( p );
    p = NULL;
  }
  . . .
}
```



Suplement II: opis funkcji z biblioteki języka C

- `void * malloc( size_t size );`

  Rezultatem funkcji *malloc* jest wskaźnik do obszaru pamięci przeznaczonego dla obiektu o rozmiarze *size*. Rezultatem jest NULL, jeżeli polecenie nie może być zrealizowane. Obszar *nie jest* inicjowany.

- `void * calloc( size_t nitems, size_t size );`

  Rezultatem funkcji *calloc* jest wskaźnik do obszaru pamięci przeznaczonego dla *nitems* obiektów o rozmiarze *size*. Rezultatem jest NULL, jeżeli polecenie nie może być zrealizowane. Obszar *jest* inicjowany zerami.

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

  Funkcja dokonuje próby zmiany rozmiaru bloku wskazywanego przez *ptr*, który był poprzednio przydzielony wywołaniem funkcji *calloc* lub *malloc*. Zawartość wskazywanego obszaru pozostaje niezmieniona.

  Jeżeli nowy rozmiar jest większy od poprzednio przydzielonego, dodatkowe bajty mają nieokreśloną wartość. Jeżeli nowy rozmiar jest mniejszy, bajty z różnicowego obszaru są zwalniane.

  Jeżeli *ptr* == NULL to funkcja działa jak *malloc*. Rezultatem funkcji jest wskaźnik na obszar pamięci o nowym rozmiarze (może być ulokowany w pamięci w innej lokalizacji niż poprzednio). Rezultatem jest NULL w przypadku błędu lub próby przydziału bloku o zerowym rozmiarze.

- `void free( void * ptr );`

  Zwalnia obszar pamięci wskazywany przez *ptr*. Parametr musi być wskaźnikiem do obszaru pamięci przydzielonego uprzednio przez *malloc*, *calloc* lub *realloc*.