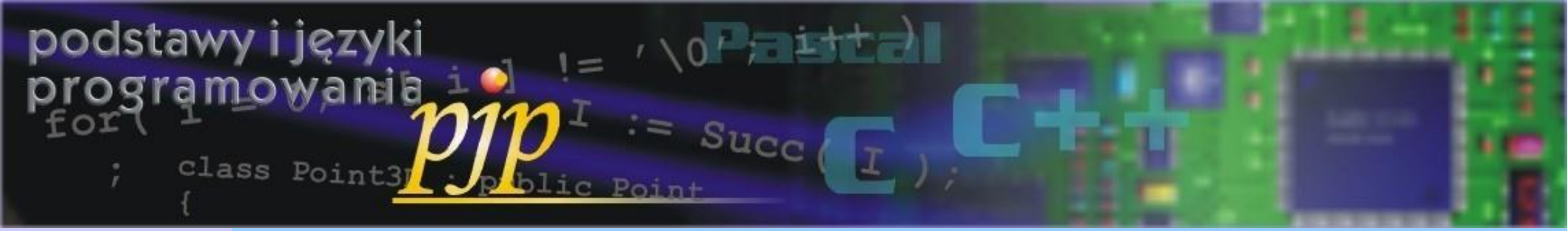


Języki programowania

Część czwarta

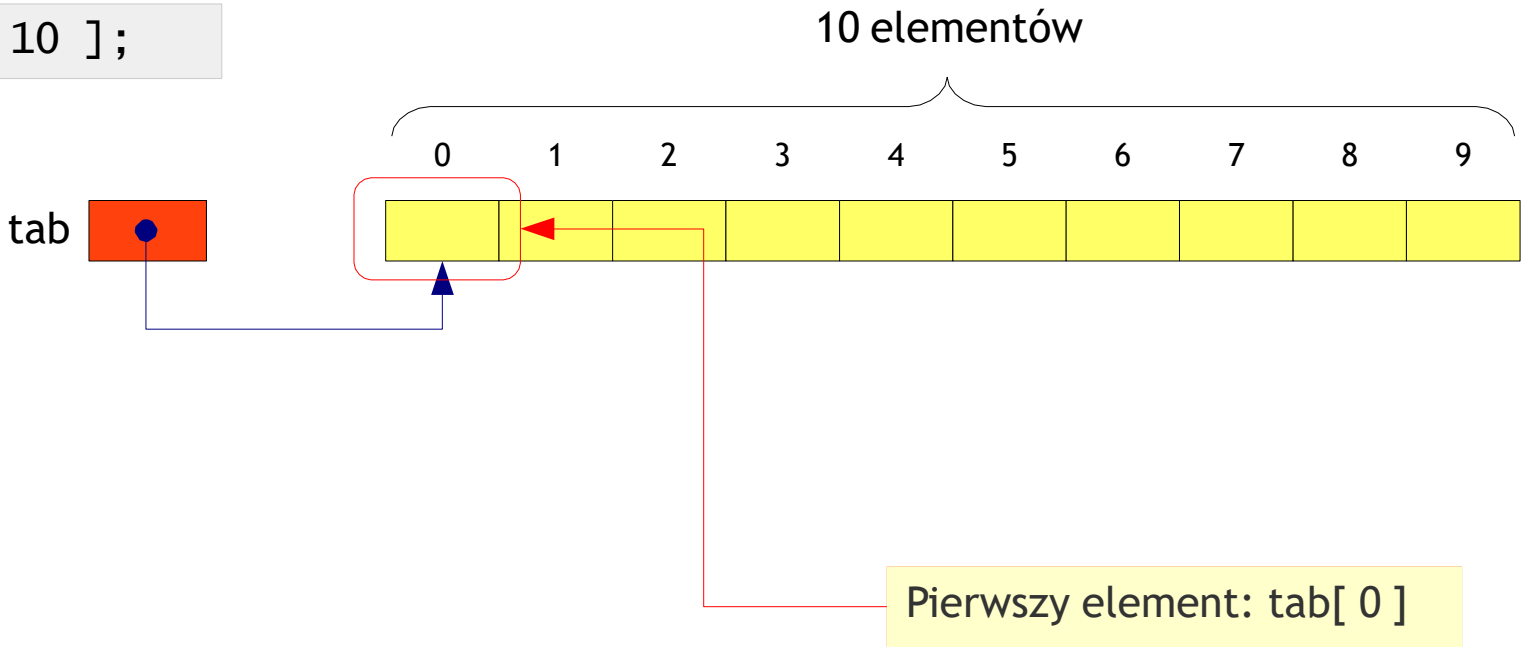
Tablice a zmienne wskaźnikowe

Autorzy
Tomasz Xięski
Roman Simiński

Nazwa tablicy jako wskaźnik na jej początek

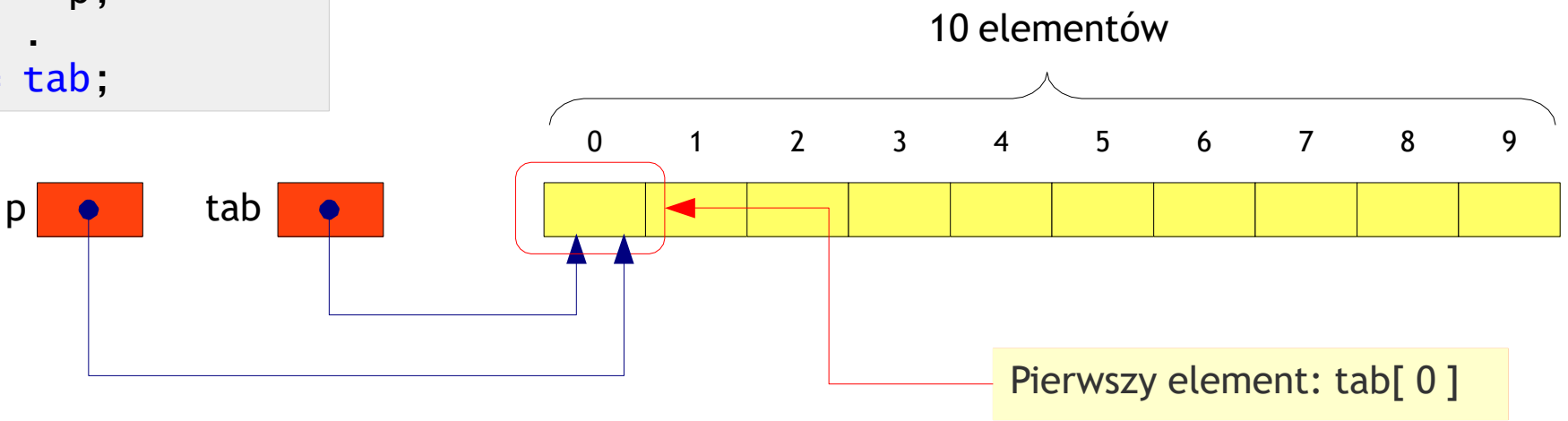
Nazwa tablicy jest interpretowana jako *ustalony wskaźnik* na jej początek (pierwszy element).

```
int tab[ 10 ];
```



Nazwa tablicy jako wskaźnik na jej początek, cd. ...

```
int tab[ 10 ];
int * p;
. . .
p = tab;
```



Przypisanie:

```
p = tab;
```

Jest równoznaczne z:

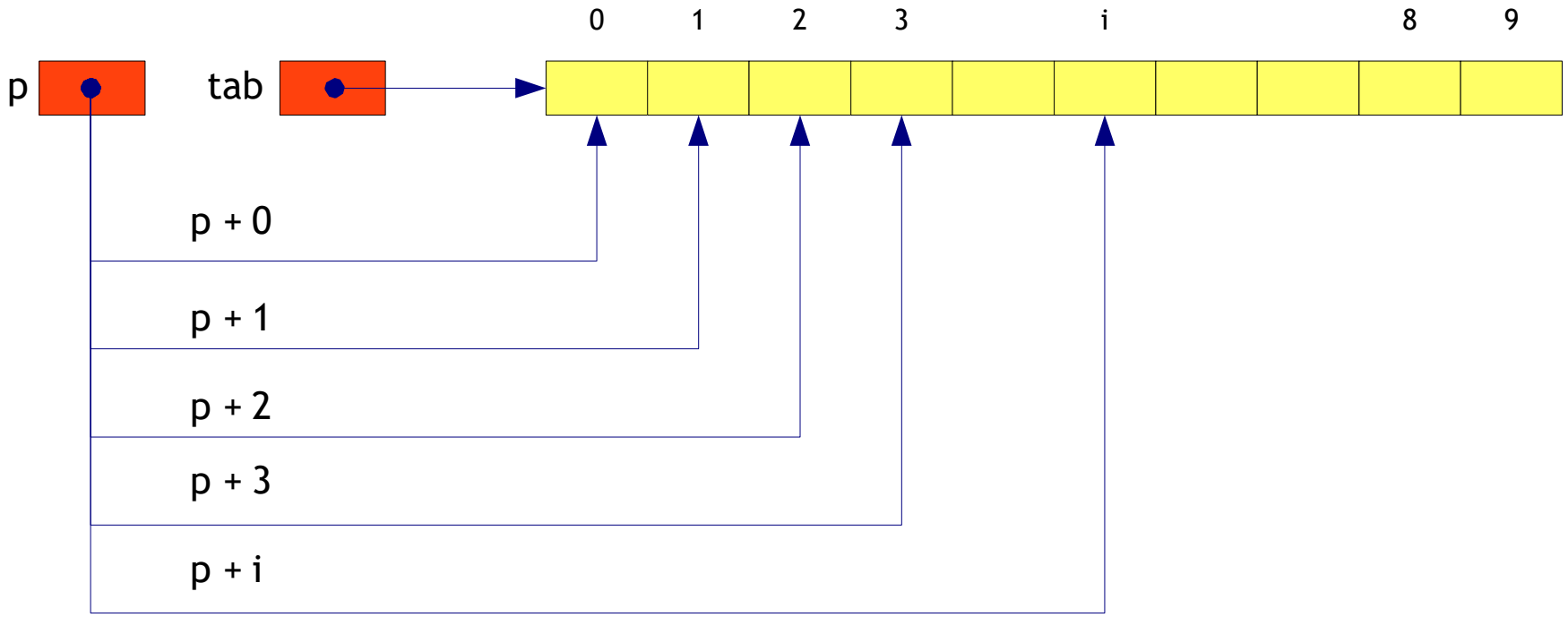
```
p = &tab[ 0 ];
```

Nazwa tablicy jako wskaźnik na jej początek, cd. ...

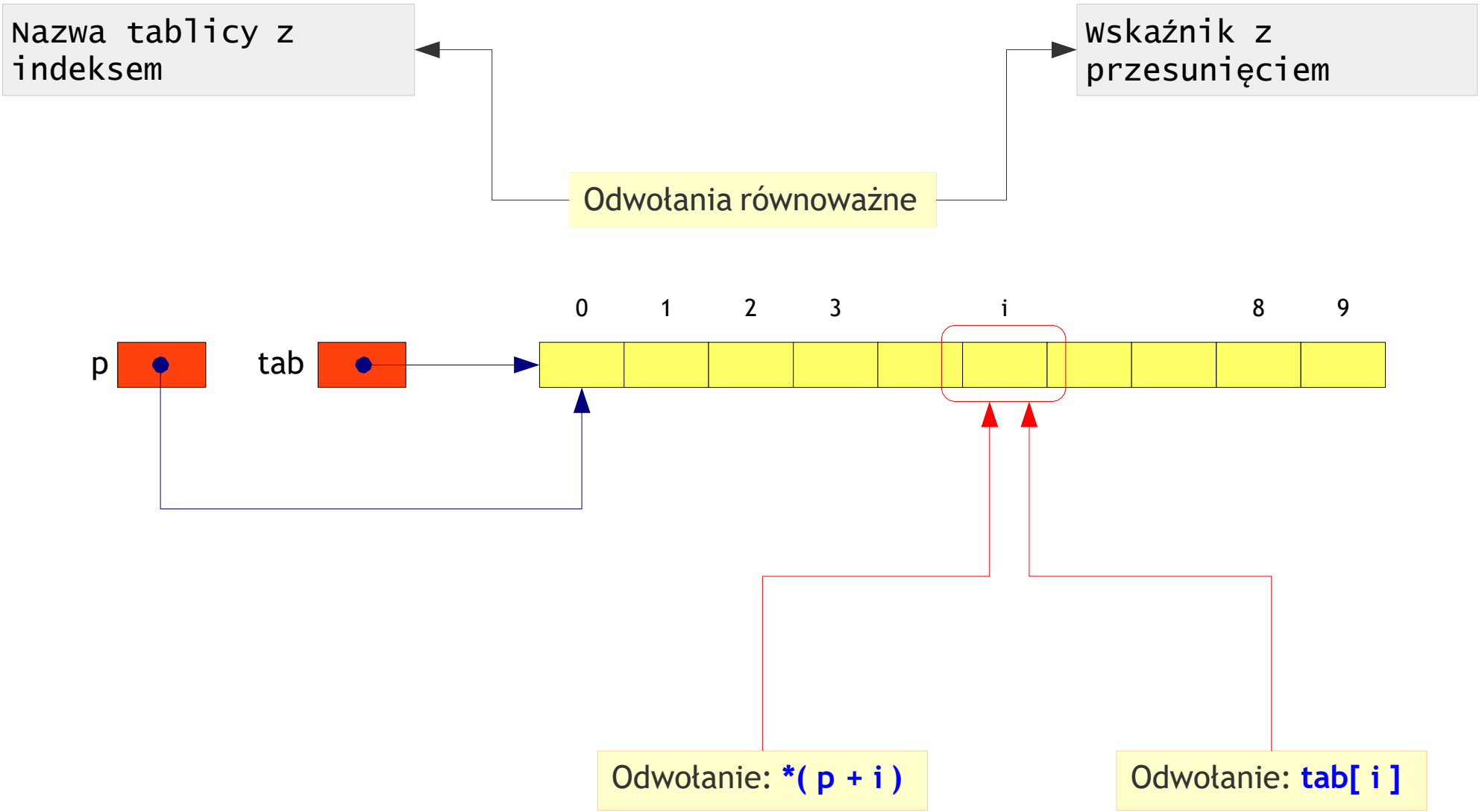
```
tab[ 0 ] = 5;
tab[ 1 ] = 1;
tab[ 2 ] = 10;
. . .
tab[ i ] = 22;
```

Odwołania równoważne

```
*p = 5
*( p + 1 ) = 1
*( p + 2 ) = 10
. . .
*( p + i ) = 22
```



Nazwa tablicy jako wskaźnik na jej początek, cd. ...



Nazwa tablicy jako wskaźnik na jej początek, cd. ...

Wyrażenie *p* + *i* jest *wyrażeniem wskaźnikowym*, wskazuje ono na obiekt oddalony o *i* obiektów danego typu od *p*.

Wartość dodawana do wskaźnika jest *skalowana* rozmiarem typu *obiektu wskazywanego*.

Każde odwołanie: `tab[ i ];` można zapisać tak: → `*( tab + i )`

Oraz

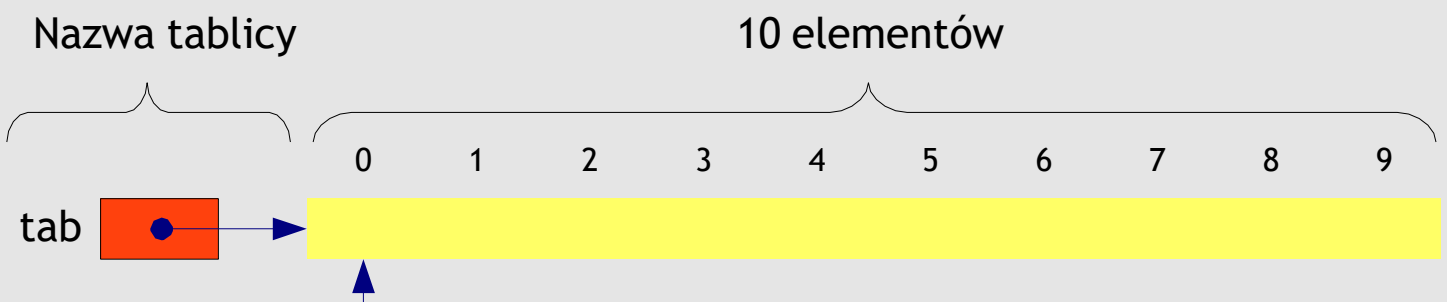
Każde odwołanie: `*( p + i )` można zapisać tak: → `p[ i ];`

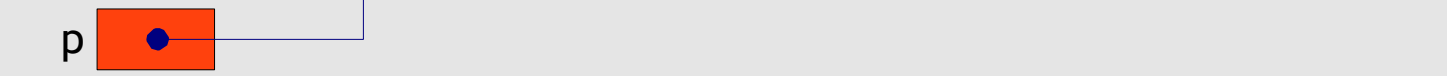
Uwaga, wskaźnik to nie tablica!

```
int tab[ 10 ];
int * p = tab;
```

Czy to jest to samo? Nie!

int tab[10] → obszar danych + wskaźnika na jego początek





int * p = tab → wskaźnik zakotwiczony o początek tablicy

Nazwa tablicy to *ustalony* wskaźnik na jej początek

Nazwa tablicy jest *ustalonym* (*niemodyfikowalnym*) wskaźnikiem na pierwszy jej element. Nazw tablic *nie wolno modyfikować*! Zwykłe wskaźniki *można*.

```
int tab[ 10 ];
int * p = tab;


tab = p;        ☒ Źle
tab++;


p = tab + 8;    ☑ OK
p++;
```

Ciekawostka

Wiemy, że odwołanie:

`tab[ i ]` można zapisać tak: → `*( tab + i )`

Oraz, że odwołanie

`*( tab + i )` można zapisać tak: → `tab[ i ]`

Wiemy również, że dodawanie jest przemienne, zatem każde odwołanie:

`*( tab + i )` można zapisać tak: → `*( i + tab )`

Czy zatem odwołanie:

`*( i + tab )` można zapisać tak: → `i[ tab ]` ?

Ciekawostka, cd. ...

Tak, można, dla kompilatora nie ma to większego znaczenia.

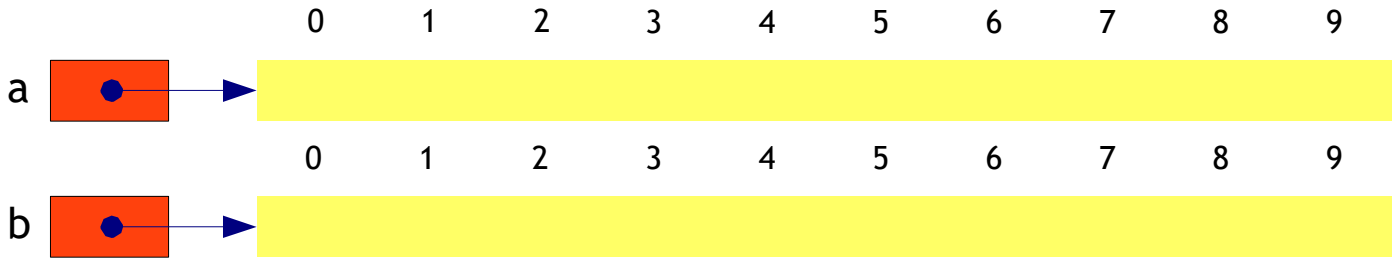
```
char napis[] = "język c";
. . .

cout << napis << endl;

0[ napis ] = 'J'; // Zamiast napis[ 0 ]
6[ napis ] = 'C'; // Zamiast napis[ 6 ]

cout << napis << endl;
```

```
język c
Język C
```

Dlaczego nie wolno przypisywać tablic, posługując się ich nazwami?

```
int a[ 10 ];
int b[ 10 ];
b = a;   // Nie wolno przypisywać do siebie tablic!
```

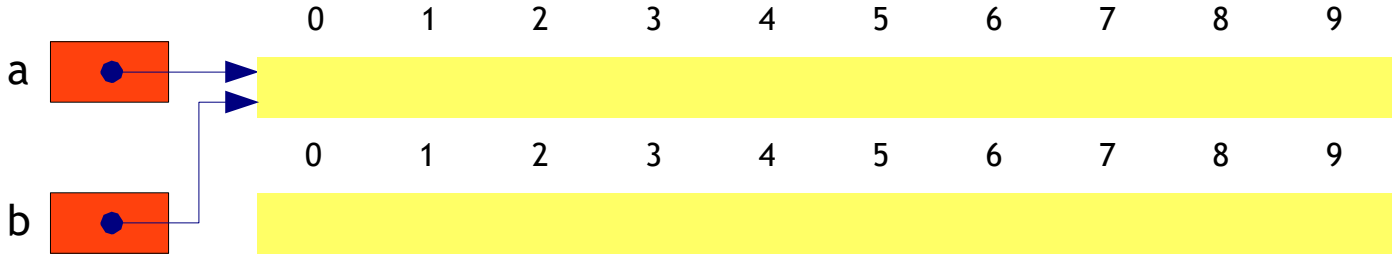
Gdyby przypisywanie było możliwe...



To po wykonaniu tej linii:

```
b = a;
```

gubimy obszar danych tablicy *b*!



Arytmetyka na wskaźnikach — podsumowanie

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 związanych z ta samą tablicą,
- przypisanie wskaźnikowi *wartości zero* (lub wskazania puste *NULL*) lub *porównanie ze wskazaniem pustym*.

Wskaźniki w akcji — metamorfoza funkcji put_string

```
char napis[] = "Język C i C++";

put_string( napis );
```

Wersja pierwotna

```
void put_string( char s[] )
{
  int i;
  for( i = 0; s[ i ] != '\0'; i++ )
    putchar( s[ i ] );
}
```

Wskaźniki w akcji — metamorfoza funkcji put_string

```
char napis[ 80 ] = "C++";

put_string( napis );
```

```
void put_string2( char * s )
{
  for( ; *s != '\0'; s++ )
    putchar( *s );
}
```

Eliminujemy zmienną i

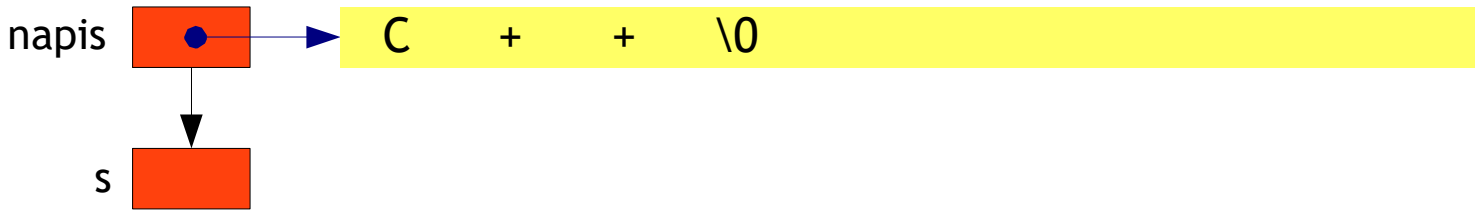
Jak to działa... ?

Wywołanie funkcji put_string

```
char napis[ 80 ] = "C++";

put_string( napis );
```

```
void put_string2( char * s )
{
  for( ; *s != '\0'; s++ )
    putchar( *s );
}
```

Kopiowanie parametru aktualnego napis do parametru s

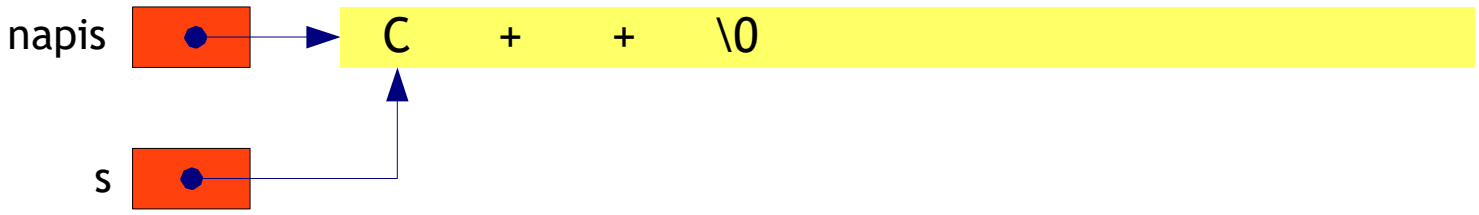


Parametr s jest kopią wskaźnika napis

```
char napis[ 80 ] = "C++";

put_string( napis );
```

```
void put_string2( char * s )
{
  for( ; *s != '\0'; s++ )
    putchar( *s );
}
```

Kopiowanie parametru aktualnego napis do parametru s

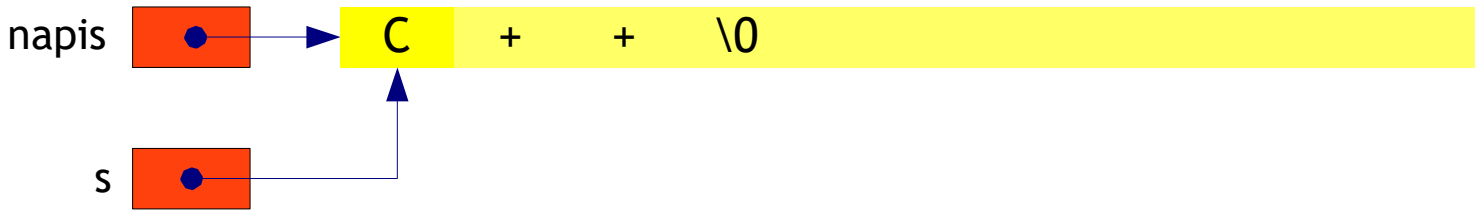


Czy obiekt wskazywany przez s jest znacznikiem końca napisu?

```
char napis[ 80 ] = "C++";

put_string( napis );
```

```
void put_string2( char * s )
{
  for( ; *s != '\0'; s++ )
    putchar( *s );
}
```

Parametr s wskazuje na pierwszy element tablicy napis

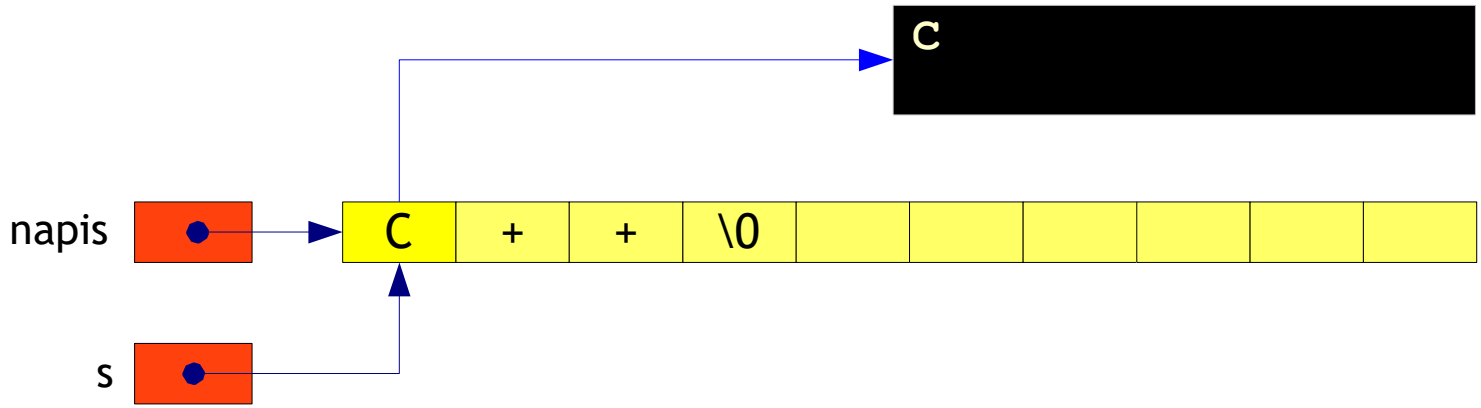


Znak wskazywany przez s wyprowadzamy jest do stdout

```
char napis[ 80 ] = "C++";

put_string( napis );
```

```
void put_string2( char * s )
{
  for( ; *s != '\0'; s++ )
    putchar( *s );
}
```

Parametr s wskazuje na pierwszy element tablicy napis

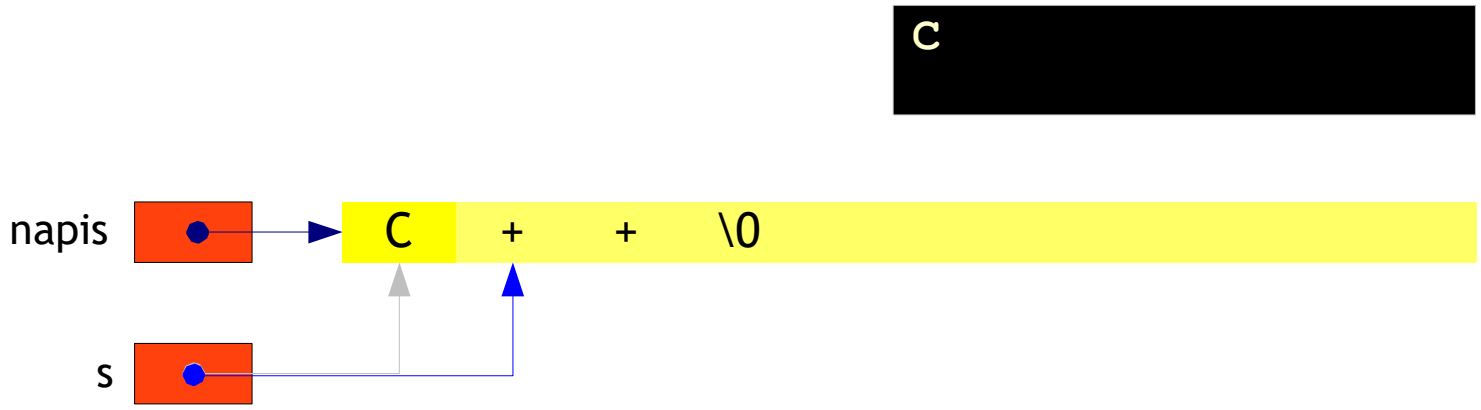


Wskaźnik s przesuwamy na następny znak

```
char napis[ 80 ] = "C++";

put_string( napis );
```

```
void put_string2( char * s )
{
  for( ; *s != '\0'; s++ )
    putchar( *s );
}
```

Parametr s wskazuje na kolejny element tablicy napis

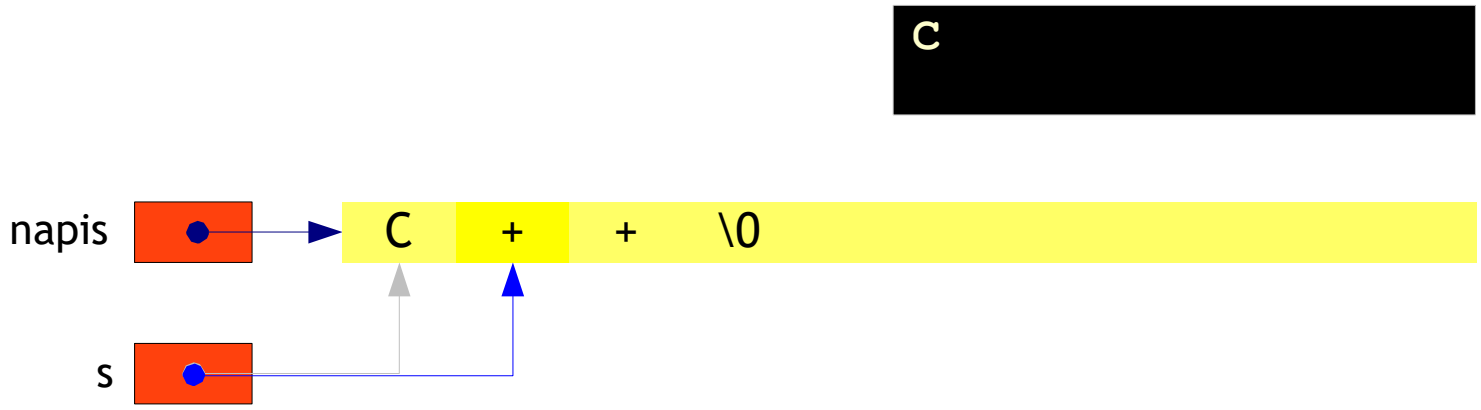


Czy obiekt wskazywany przez s jest znacznikiem końca napisu?

```
char napis[ 80 ] = "C++";

put_string( napis );
```

```
void put_string2( char * s )
{
  for( ; *s != '\0'; s++ )
    putchar( *s );
}
```

Parametr s wskazuje na kolejny element tablicy napis

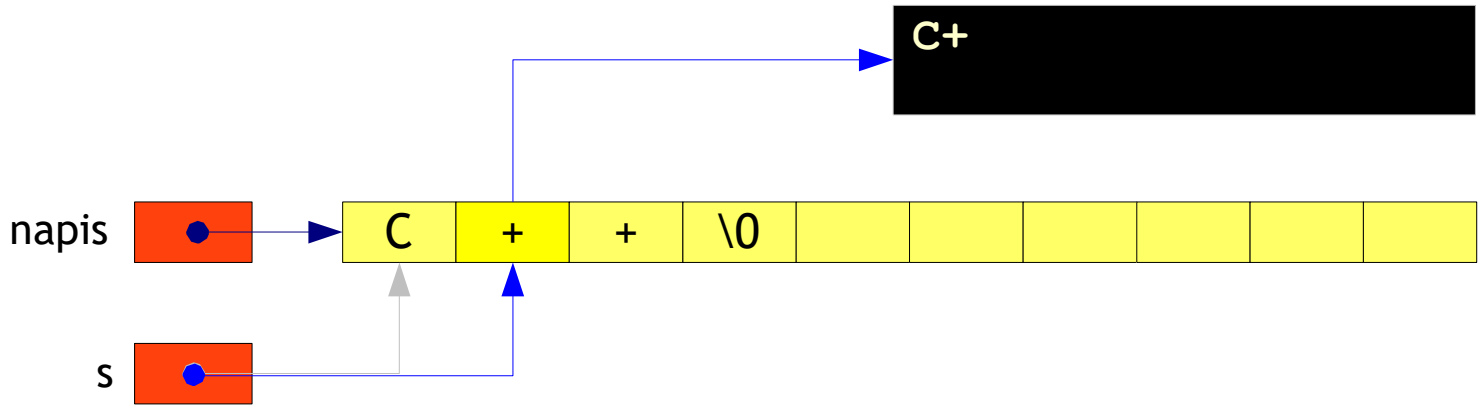


Znak wskazywany przez s wyprowadzamy jest do stdout

```
char napis[ 80 ] = "C++";

put_string( napis );
```

```
void put_string2( char * s )
{
  for( ; *s != '\0'; s++ )
    putchar( *s );
}
```

Parametr s wskazuje na kolejny element tablicy napis

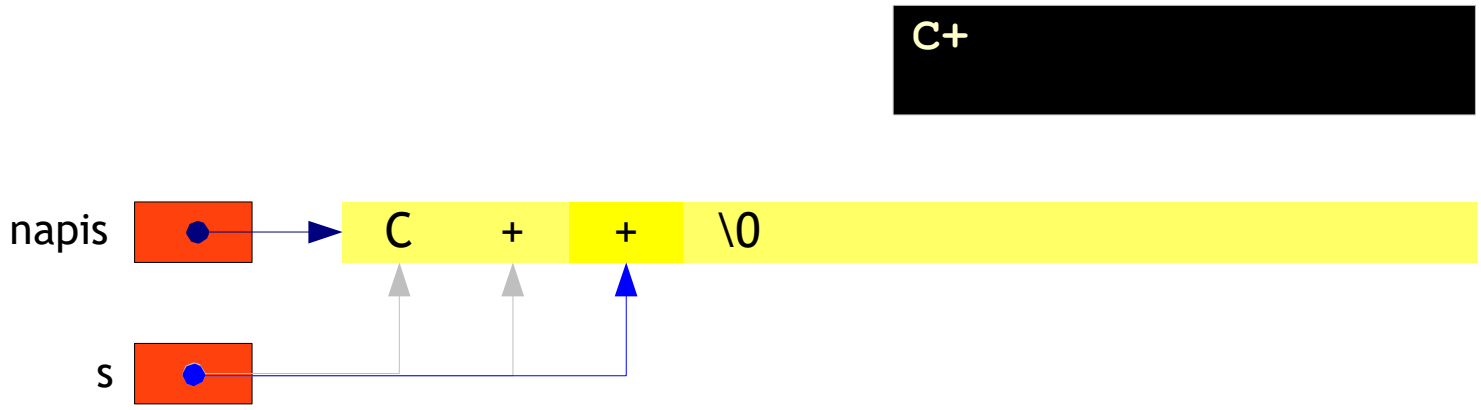


Wskaźnik s przesuwamy na następny znak

```
char napis[ 80 ] = "C++";

put_string( napis );
```

```
void put_string2( char * s )
{
  for( ; *s != '\0'; s++ )
    putchar( *s );
}
```

Parametr s wskazuje na kolejny element tablicy napis

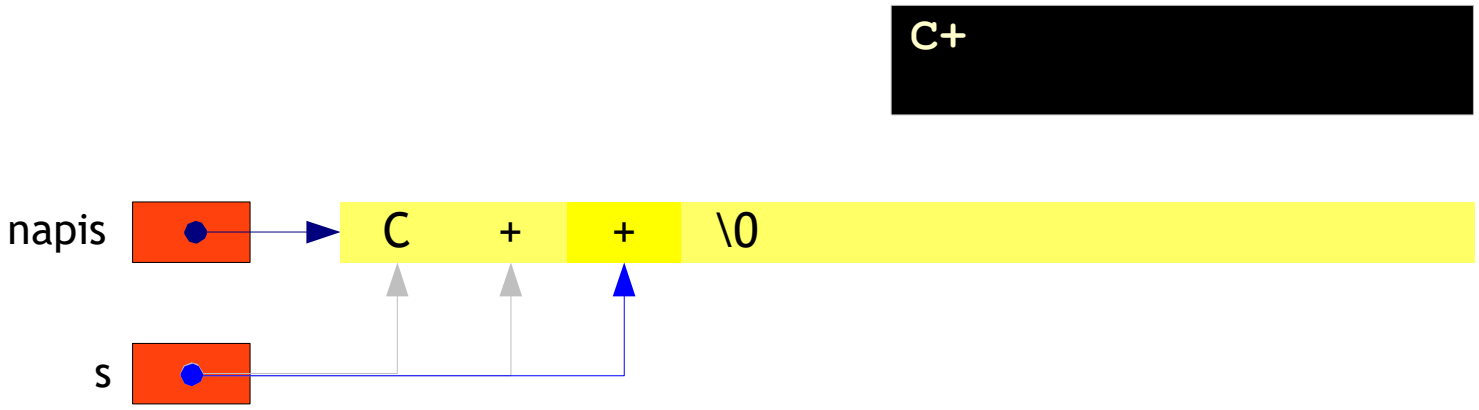


Czy obiekt wskazywany przez s jest znacznikiem końca napisu?

```
char napis[ 80 ] = "C++";

put_string( napis );
```

```
void put_string2( char * s )
{
  for( ; *s != '\0'; s++ )
    putchar( *s );
}
```

Parametr s wskazuje na kolejny element tablicy napis

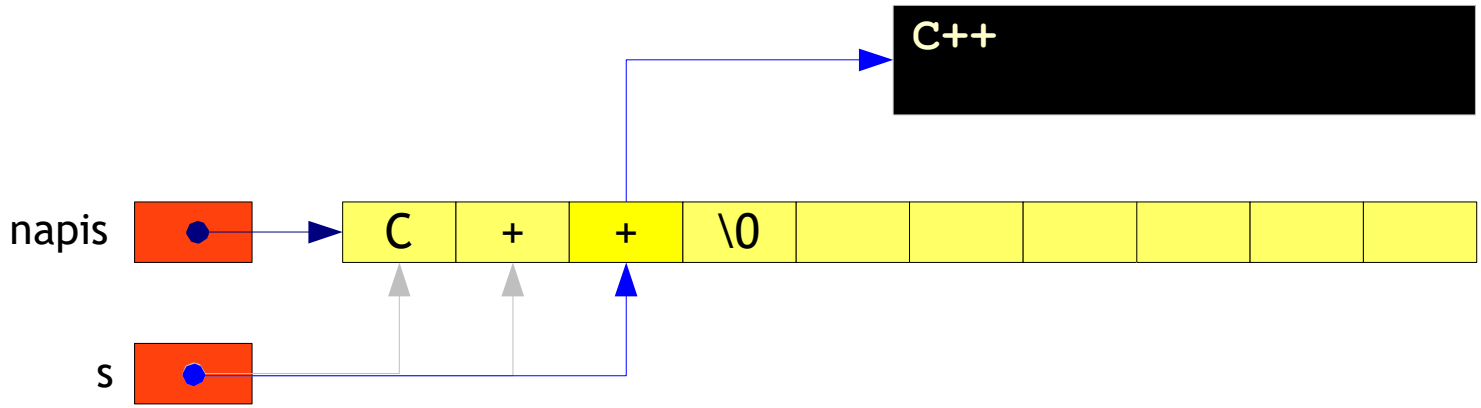


Znak wskazywany przez s wyprowadzamy jest do stdout

```
char napis[ 80 ] = "C++";

put_string( napis );
```

```
void put_string2( char * s )
{
  for( ; *s != '\0'; s++ )
    putchar( *s );
}
```

Parametr s wskazuje na kolejny element tablicy napis

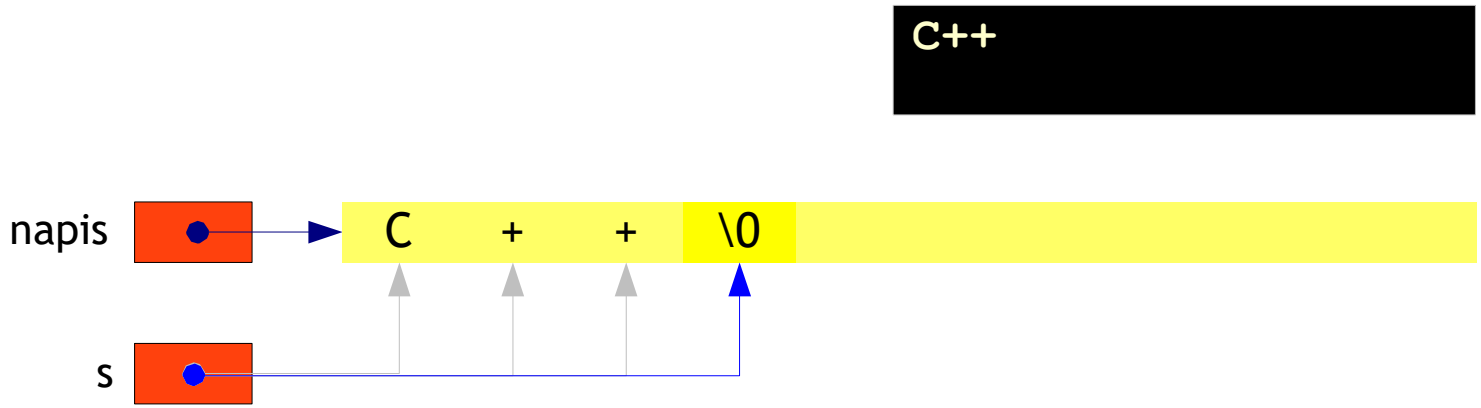


Wskaźnik s przesuwamy na następny znak

```
char napis[ 80 ] = "C++";

put_string( napis );
```

```
void put_string2( char * s )
{
  for( ; *s != '\0'; s++ )
    putchar( *s );
}
```

Parametr s wskazuje na kolejny element tablicy napis

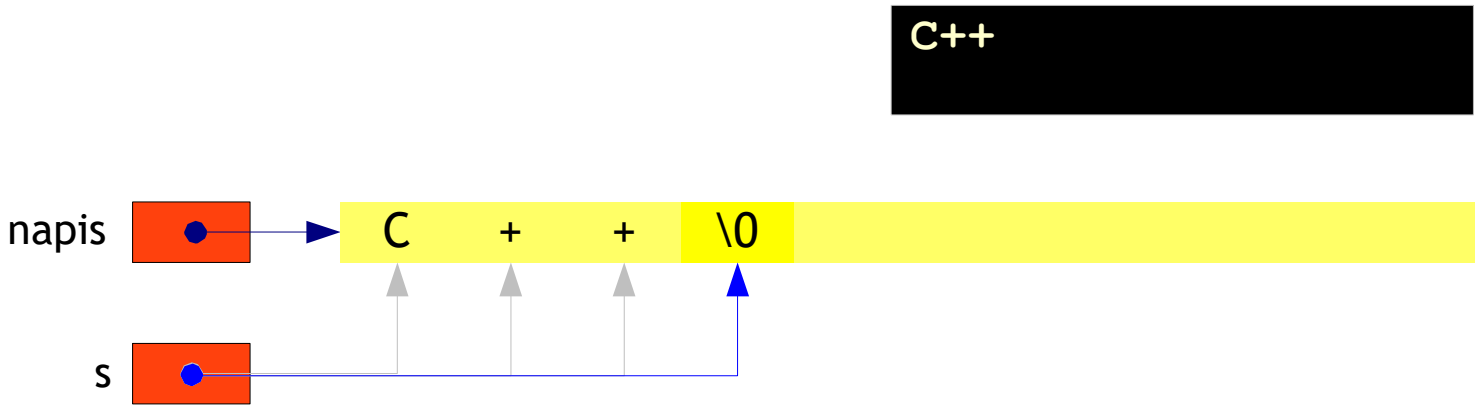


Czy obiekt wskazywany przez s jest znacznikiem końca napisu?

```
char napis[ 80 ] = "C++";

put_string( napis );
```

```
void put_string2( char * s )
{
  for( ; *s != '\0'; s++ )
    putchar( *s );
}
```

Parametr s wskazuje na kolejny element tablicy napis

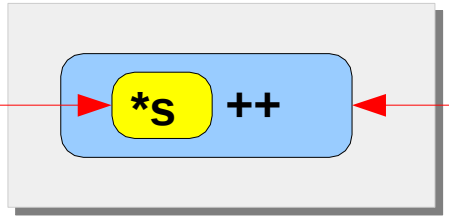


Wskaźniki w akcji — metamorfoza funkcji *put_string*, wersja 3 i 4

```
void put_string3( char * s )
{
  for( ; *s != '\0' ; putchar( *s++ ) )
    ;
}
```

„Kompresja” iteracji for

Najpierw pobierz znak wskazywany przez **s**, użyj go.



Potem zwiększ o jeden wartość wskaźnika **s** — będzie on wtedy wskazywał na następny element tablicy.

```
void put_string4( char * s )
{
  while( *s )
    putchar( *s++ );
}
```

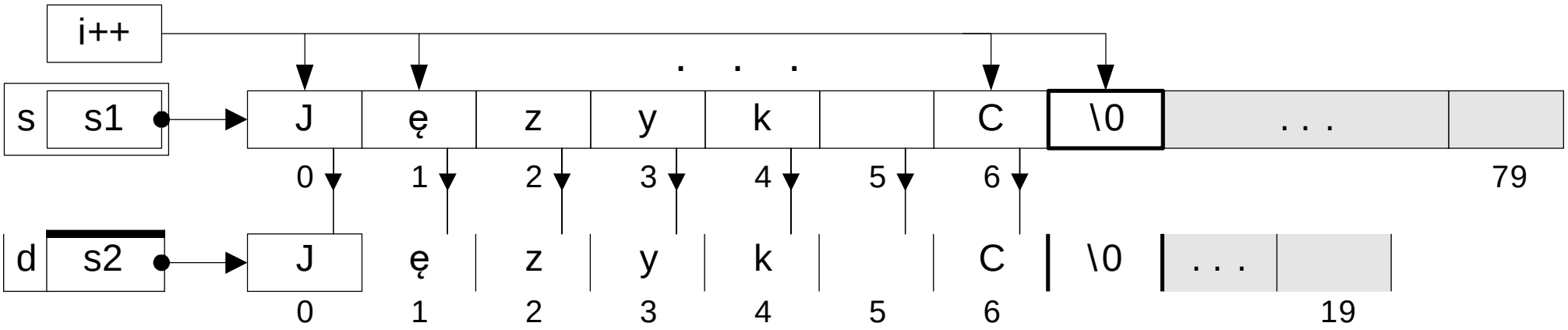
Iteracja while nie jest taka zła...
Znak '\0' to bajt o wartości 0

Wskaźniki pod lupą — metamorfoza funkcji *strcpy*

```
char s1[ 80 ] = "Język C";
char s2[ 20 ];
. . .
strcpy( s2, s1 );
```

Wersja początkowa

Przypomnienie jak to działa




```
void strcpy( char d[], char s[] )
{
  int i;
  for( i = 0; s[ i ] != '\0'; i++ )
    d[ i ] = s[ i ];
  d[ i ] = '\0';
}
```

Wskaźniki pod lupą — metamorfoza funkcji *strcpy*

```
void strcpy1( char * d, char * s )
{
  while( *s != '\0' )
  {
     *d = *s;
     d++;
     s++;
  }
  *d = '\0';
}
```

Odwołania wskaźnikowe
To właściwie nie wiele zmienia,
poza wyeliminowaniem
zmiennej i

```
void strcpy2( char * d, char * s )
{
  while( *s != '\0' )
     *d++ = *s++;
  *d = '\0';
}
```

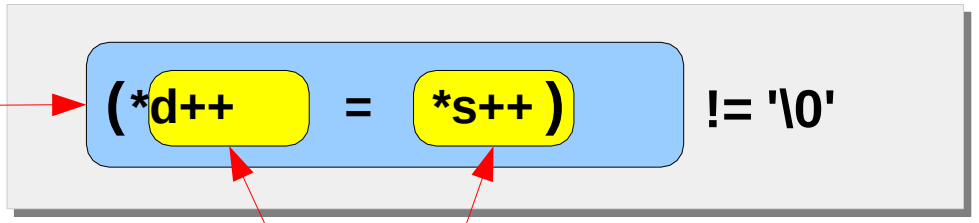
„Kompresja” — krok pierwszy

Wskaźniki pod lupą — metamorfoza funkcji *strcpy*

„Kompresja” – krok drugi

```
void strcpy3( char * d, char * s )
{
  while( ( *d++ = *s++ ) != '\0' )
    ;
}
```

Wartością tego wyrażenia jest znak (bajt) przepisany z obszaru wskazywanego przez **s** do obszaru wskazywanego przez **d**.



Operator = jest lewostronnie łączny

Pobierz znak wskazywany, wykorzystaj go, zwiększ wskaźnik tak, by pokazywał na następny element tablicy.

Wskaźniki pod lupą — metamorfoza funkcji *strcpy*

```
void strcpy4( char * d, char * s )
{
  while( *d++ = *s++ )
    ;
}
```

„Kompresja” — krok trzeci
Znak '\0' to bajt o wartości 0

Często spotykaną praktyką w funkcjach bibliotecznych jest udostępnianie wskaźnika do tablicy (jednej z tablic) będącej parametrem:

```
char * strcpy5( char * d, char * s )
{
  while( *d++ = *s++ )
    ;
  return d;
}
```

Tablica d jako rezultat funkcji

Wskaźniki pod lupą — metamorfoza funkcji *strcpy*

Pozwala to na skrócenie kodu, załóżmy następujące definicje tablic *s1*, *s2*, *s3*:

```
char s1[ 80 ] = "C i C++";
char s2[ 80 ];
char s3[ 80 ];
```

Następujący fragment kodu:

```
strcpy5( s2, s1 );
strcpy5( s3, s2 );
puts( s3 );
```

Można zapisać krócej:

```
puts( strcpy5( s3, strcpy5( s2, s1 ) ) );
```

Wskaźniki pod lupą — metamorfoza funkcji *strcpy*

W dotychczasowych realizacjach funkcji *strcpyX*, funkcja może modyfikować zawartość tablicy źródłowej:

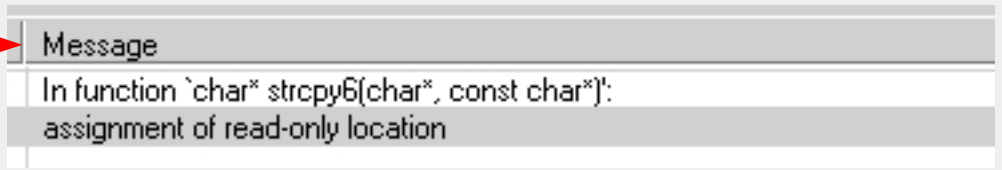
```
char * strcpy5( char * d, char * s )
{
  *s = 'A';
  . . .
}
```

Modyfikacja tablicy źródłowej dozwolona, choć merytorycznie niepoprawna

Aby temu zaradzić, można zadeklarować parametr reprezentujący tablicę źródłową w specyficzny sposób:

```
char * strcpy6( char * d, const char * s )
{
  *s = 'A';
  . . .
}
```

Tablica źródłowa jest chroniona



Wskaźniki pod lupą — zastosowanie modyfikatora *const*

Aby funkcja nie mogła zmodyfikować parametru przekazanego za pośrednictwem wskaźnika, należy w deklaracji użyć słowa *const*. Deklaracja:

*const char * s*;

oznacza, że *s* jest wskaźnikiem na stały (niemodyfikowalny) obiekt typu *char*.

Można wyróżnić następujące kombinacje definicji wskaźnika z/bez *const*:

```
const int * const p; // Ustalony wskaźnika na niemodyfikowalny obiekt
      int * const p; // Ustalony wskaźnika na modyfikowalny obiekt
const int *       p; // Zwykły wskaźnika na niemodyfikowalny obiekt
      int *       p; // Zwykły wskaźnik na zwykły obiekt
```

Wskaźniki pod lupą — zastosowanie modyfikatora *const*

Wersja najbardziej restrykcyjna pod lupą:

```
const int * const p;
```

To się nie uda, ustalony wskaźnik należy zainicjować!

Posługiwanie się ustalonym wskaźnikiem do stałego obiektu:

```
int i = 10;
const int * const p = &i;
. . .

j = *p + 10;
. . .

*p = 20;
. . .

p = &j;
```

To jest OK, odwołanie nie modyfikujące obiektu

Niedozwolone, odwołanie modyfikujące obiekt

Niedozwolone, odwołanie modyfikujące wskaźnik

Wyznaczanie długości napisu — funkcja strlen klasycznie

Realizacja w wykorzystaniem iteracji *while*:

```
int strlen( char s[] )
{
  int len = 0;

  while( s[ len ] != '\0' )
    len++;

  return len;
}
```

Realizacja w wykorzystaniem iteracji *for*:

```
int strlen( char s[] )
{
  int len;

  for( len = 0; s[ len ] != '\0'; len++ )
    ;

  return len;
}
```

Wyznaczanie długości napisu — funkcja strlen wskaźnikowo

Realizacja w wykorzystaniem iteracji *while*:

```
int strlen( char * s )
{
  char * ptr = s;

  while( *ptr != '\0' )
    ptr++;

  return ( int )( ptr - s );
}
```

Realizacja w wykorzystaniem iteracji *for*:

```
int strlen( char * s )
{
  char * ptr;

  for( ptr = s;  *ptr != '\0'; ptr++ )
    ;

  return ( int )( ptr - s );
}
```

Odwracanie kolejności znaków w napisie — strrev klasycznie

```
char * strrev( char s[] )
{
  int begin, end;

  // Szukanie konca napisu
  for( end = 0; s[ end ] != '\0'; end++ )
    ;

  // Zamiana znakow miejscami
  for( begin = 0, end--; begin < end; begin++, end-- )
  {
    char c = s[ begin ];
    s[ begin ] = s[ end ];
    s[ end ] = c;
  }
  return s;
}
```

Odwracanie kolejności znaków w napisie — strrev wskaźnikowo

```
char * strrev( char * s )
{
  char * begin, * end;

  // Szukanie znacznika konca
  for( end = s; *end ; end++ )
    ;

  // Zamiana znakow miejscami
  for( begin = s, end--; begin < end; begin++, end-- )
  {
    char c = *begin;
    *begin = *end;
    *end = c;
  }
  return s;
}
```

Dynamiczna alokacja tablic — konwencja języka C

Na tablicach alokowanych dynamicznie na stercie, można wykonywać *takie same operacje*, jak na *tablicach statycznych*. Należy tylko *uważnie przydzielać i zwalniać pamięć*.

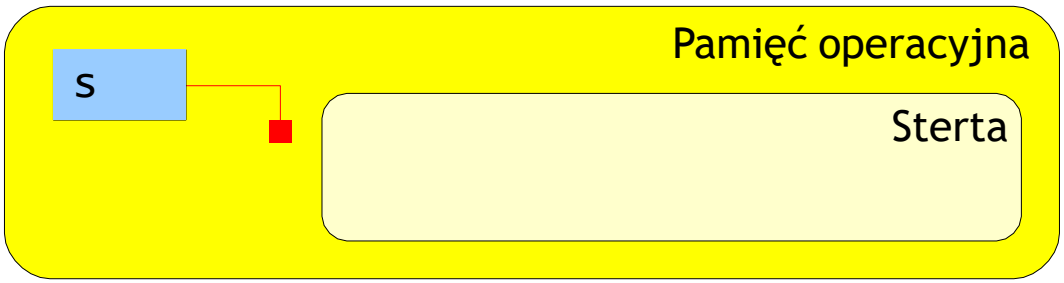
```
char * s = NULL;
int n;

/* Tu ustalenie liczby potrzebnych elementów i zapamiętanie w zmiennej n */

s = malloc( n * sizeof( char ) );

if( s != NULL )
{
  strcpy( s, "Język C " );
  strcat( s, "fajny jest!" );
  puts( s );
  . . .
  free( s );
}
```

Zobaczmy, jak wyglądają kolejne etapy definiowania i wykorzystania takiej tablicy...

Dynamiczna alokacja tablic — konwencja języka C, etap 1-szy

Definicja wskaźnika — typ obiektu wskazywanego taki, jak typ elementów tablicy jakich potrzebujemy. Zerowanie wskaźnika to dobra praktyka.

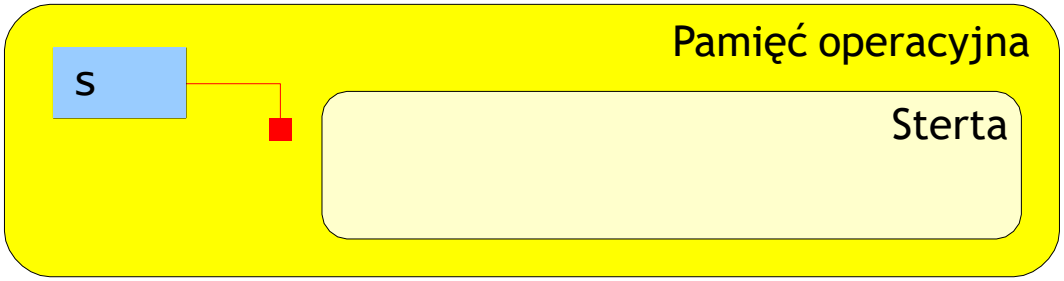
```
char * s = NULL;
int n;

/* Tu ustalenie liczby potrzebnych elementów i zapamiętanie w zmiennej n */

s = malloc( n * sizeof( char ) );

if( s != NULL )
{
  strcpy( s, "Język C " );
  strcat( s, "fajny jest!" );
  puts( s );
  . . .
  free( s );
}
```



Dynamiczna alokacja tablic — konwencja języka C, etap 2-gi

Zwykle korzystamy ze zmiennej, która pozwoli zapamiętać ilu elementową tablicę potrzebujemy.

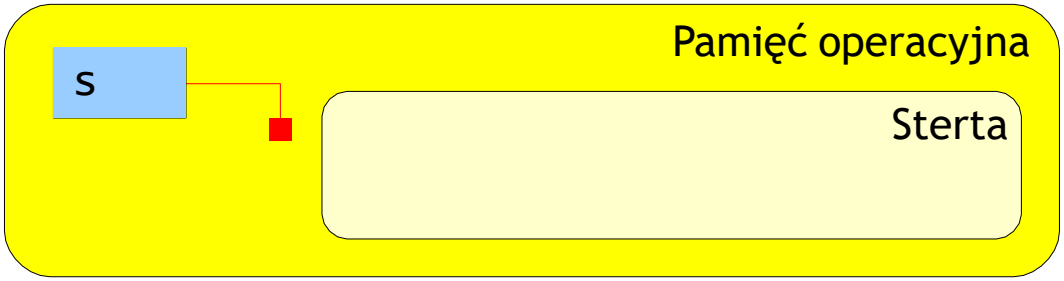
```
char * s = NULL;
int n;

/* Tu ustalenie liczby potrzebnych elementów i zapamiętanie w zmiennej n */

s = malloc( n * sizeof( char ) );

if( s != NULL )
{
  strcpy( s, "Język C " );
  strcat( s, "fajny jest!" );
  puts( s );
  . . .
  free( s );
}
```



Dynamiczna alokacja tablic — konwencja języka C, etap 3-ci

Przed utworzeniem tablicy musimy ustalić konkretną liczbę elementów tablicy. Jak ustalimy tę liczbę zależy od konkretnego zastosowania.

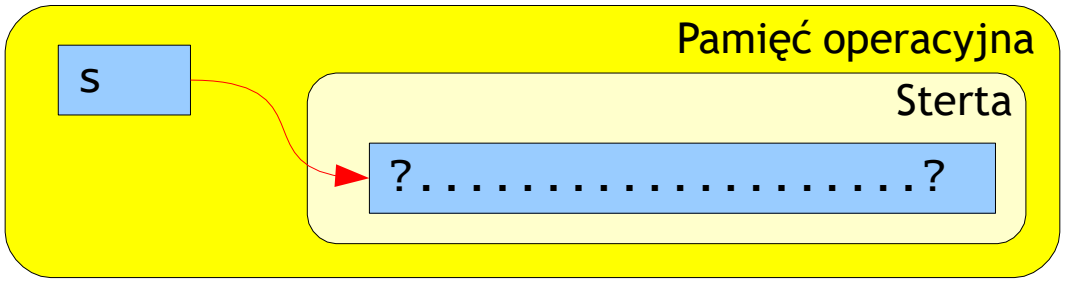
```
char * s = NULL;
int n;

/* Tu ustalenie liczby potrzebnych elementów i zapamiętanie w zmiennej n */

s = malloc( n * sizeof( char ) );

if( s != NULL )
{
  strcpy( s, "Język C " );
  strcat( s, "fajny jest!" );
  puts( s );
  . . .
  free( s );
}
```



Dynamiczna alokacja tablic — konwencja języka C, etap 4-ty

Przydział pamięci dla tablicy — funkcja *malloc* otrzymuje liczbę bajtów potrzebnych do przechowania ustalonej liczby elementów tablicy.

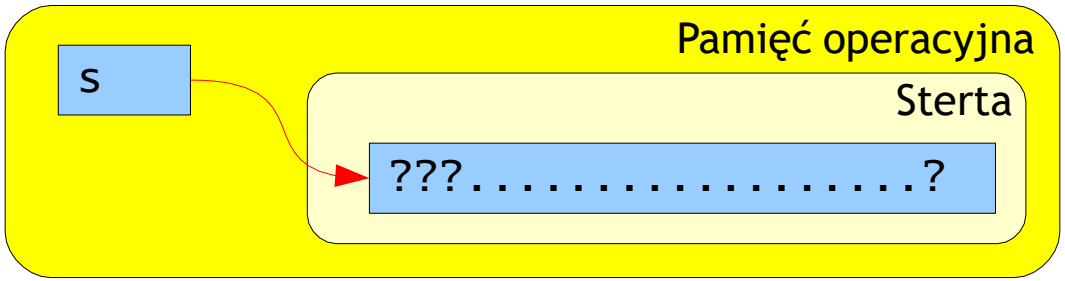
```
char * s = NULL;
int n;

/* Tu ustalenie liczby potrzebnych elementów i zapamiętanie w zmiennej n */

s = malloc( n * sizeof( char ) );

if( s != NULL )
{
  strcpy( s, "Język C " );
  strcat( s, "fajny jest!" );
  puts( s );
  . . .
  free( s );
}
```



Dynamiczna alokacja tablic — konwencja języka C, etap 5-ty

Kontrola poprawności przydziału pamięci. Uwaga — to koniecznie niezbędny etap! Przydzielony obszar pamięci ma przypadkową zawartość.

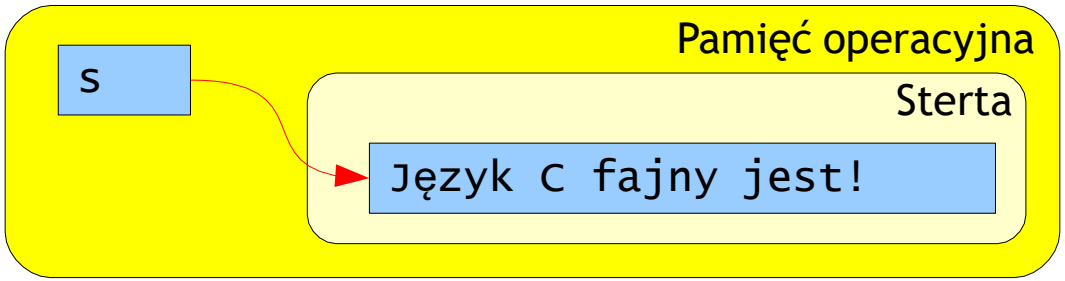
```
char * s = NULL;
int n;

/* Tu ustalenie liczby potrzebnych elementów i zapamiętanie w zmiennej n */

s = malloc( n * sizeof( char ) );

if( s != NULL )
{
  strcpy( s, "Język C " );
  strcat( s, "fajny jest!" );
  puts( s );
  . . .
  free( s );
}
```



Dynamiczna alokacja tablic — konwencja języka C, etap 6-ty

Tak utworzona tablicę można używać tak samo, jak każdą inną tablicę w języku C. Wszystkie funkcje do manipulowania np. napisami działają bez problemu.

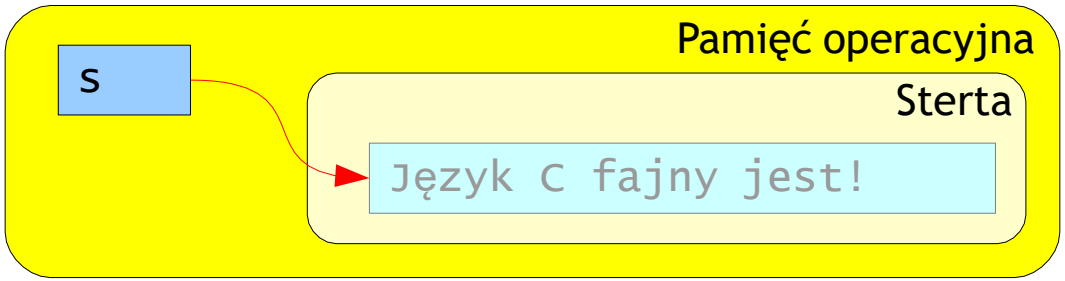
```
char * s = NULL;
int n;

/* Tu ustalenie liczby potrzebnych elementów i zapamiętanie w zmiennej n */

s = malloc( n * sizeof( char ) );

if( s != NULL )
{
  strcpy( s, "Język C " );
  strcat( s, "fajny jest!" );
  puts( s );
  . . .
  free( s );
}
```



Dynamiczna alokacja tablic — konwencja języka C, etap 6-ty

Gdy tablica nie jest już potrzebna, zwalniamy przydzieloną pamięć i oddajemy do puli wolnych bloków. Uwaga, wskaźnik pokazuje dalej na zwolniony obszar pamięci!

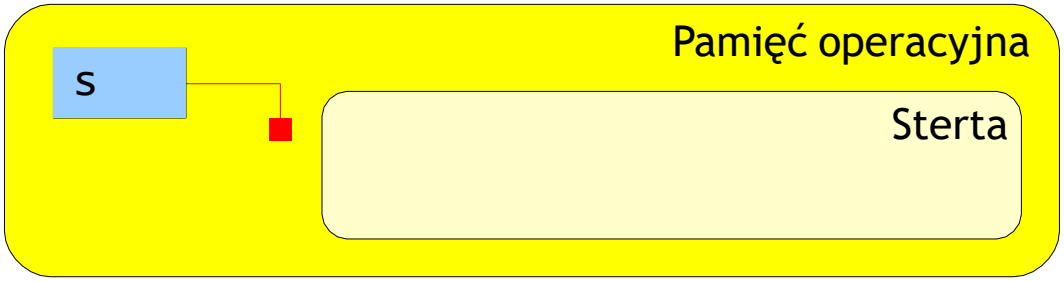
```
char * s = NULL;
int n;

/* Tu ustalenie liczby potrzebnych elementów i zapamiętanie w zmiennej n */

s = malloc( n * sizeof( char ) );

if( s != NULL )
{
  strcpy( s, "Język C " );
  strcat( s, "fajny jest!" );
  puts( s );
  . . .
  free( s );
}
```



Dynamiczna alokacja tablic — konwencja języka C, etap 6-ty

Zerowanie wskaźnika po zwolnieniu pamięci jest dobrą praktyką.

```
char * s = NULL;
int n;

/* Tu ustalenie liczby potrzebnych elementów i zapamiętanie w zmiennej n */

s = malloc( n * sizeof( char ) );

if( s != NULL )
{
  strcpy( s, "Język C " );
  strcat( s, "fajny jest!" );
  puts( s );
  . . .
  free( s );
  s = NULL;
}
```



Dynamiczna alokacja tablic — konwencja języka C++

W języku C++ wykorzystujemy operatory *new* i *delete*. „Stara” wersja, zakładająca, że operator *new* oddaje wskaźnik zerowy w przypadku braku wolnej pamięci:

```
char * s = 0;
int n;

// Tu ustalenie liczby potrzebnych elementów i zapamiętanie w zmiennej n

s = new char [ n ];

if( s != 0 )
{
  strcpy( s, "Język C " );
  strcat( s, "fajny jest!" );
  puts( s );
  . . .
  delete [] s;
}
```

Dynamiczna alokacja tablic — konwencja języka C++

„Nowa” wersja, zakładająca, że operator *new* generuje wyjątek w przypadku braku wolnej pamięci:

```
char * s = 0;
int n;

// Tu ustalenie liczby potrzebnych elementów i zapamiętanie w zmiennej n

try
{
   s = new char [ n ];

   strcpy( s, "Język C " );
   strcat( s, "fajny jest!" );
   puts( s );
   . . .
   delete [] s;
}
catch( ... )
{
   cout << "Brak pamięci dla wykonania tej operacji";
}
```

Dynamiczna alokacja tablic — konwencja języka C++

Wykorzystanie operatora *new (nothrow)* nie generującego wyjątków, obsługa jak w „starej” wersji:

```
char * s = 0;
int n;

// Tu ustalenie liczby potrzebnych elementów i zapamiętanie w zmiennej n

s = new (nothrow) char [ n ];

if( s != 0 )
{
  strcpy( s, "Język C " );
  strcat( s, "fajny jest!" );
  puts( s );
  . . .
  delete [] s;
}
```

Dynamiczna alokacja tablic — konwencja języka C++

Zerowanie wskaźnika po zwolnieniu pamięci jest dobrą praktyką.

```
char * s = 0;
int n;

// Tu ustalenie liczby potrzebnych elementów i zapamiętanie w zmiennej n

s = new (nothrow) char [ n ];

if( s != 0 )
{
  strcpy( s, "Język C " );
  strcat( s, "fajny jest!" );
  puts( s );
  . . .
  delete [] s;
  s = 0;
}
```

Dynamiczna alokacja tablic — konwencja języka C++

Zerowanie wskaźnika po zwolnieniu pamięci jest dobrą praktyką.

```
char * s = 0;
int n;

// Tu ustalenie liczby potrzebnych elementów i zapamiętanie w zmiennej n

try
{
  s = new char [ n ];

  strcpy( s, "Język C " );
  strcat( s, "fajny jest!" );
  puts( s );
  . . .
  delete [] s;
  s = 0;
}
catch( ... )
{
  cout << "Brak pamięci dla wykonania tej operacji";
}
```

Ważna sprawa — ostrożnie z parametrami wskaźnikowymi!

W funkcjach bibliotecznych języka C i C++ stałą praktyką jest deklarowanie parametrów tablicowych z wykorzystaniem wskaźników, np:

```
int strlen( char * s );
```

zamiast

```
int strlen( char s[] );
```

Wymaga to dokładnego przeczytania dokumentacji, bowiem programiści często się mylą. Rozważmy następujący przykład (fragment systemu pomocy firmy Borland):

Prototype

```
char *gets(char *s);
```

Description

Gets a string from stdin.

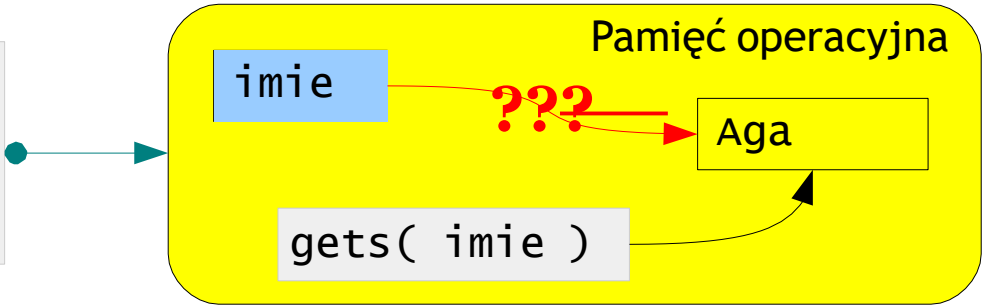
gets collects a string of characters terminated by a new line from the standard input stream stdin and puts it into s. The new line is replaced by a null character (\0) in s.

gets allows input strings to contain certain whitespace characters (spaces, tabs). gets returns when it encounters a new line; everything up to the new line is copied into s.

Ważna sprawa — ostrożnie z parametrami wskaźnikowymi!

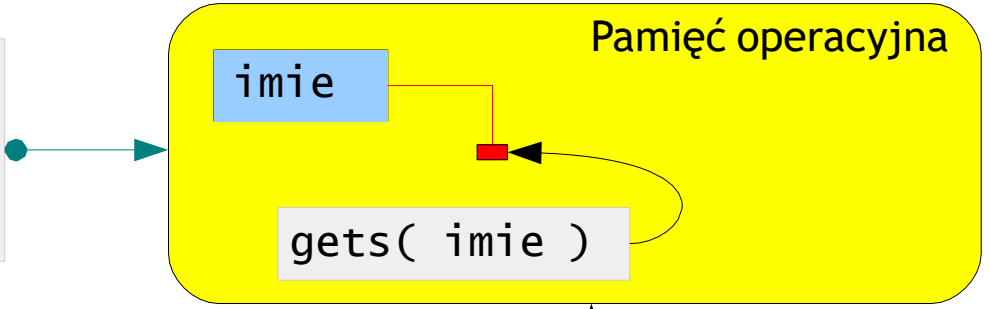
Niedokładna lektura dokumentacji może sugerować, że funkcji należy użyć tak:

```
char * imie;

printf( "Podaj imie: " );
gets( imie );
```



Gdyby wskaźnik był wyzerowany, kompilator czasem pomoże:

```
char * imie = NULL;

printf( "Podaj imie: " );
gets( imie );
```



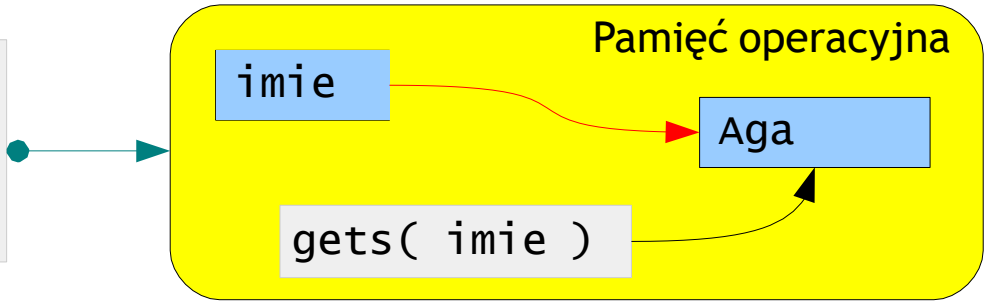
Stare kompilatory firmy Borland:

```
Null pointer assignment
```

Ważna sprawa — ostrożnie z parametrami wskaźnikowymi!

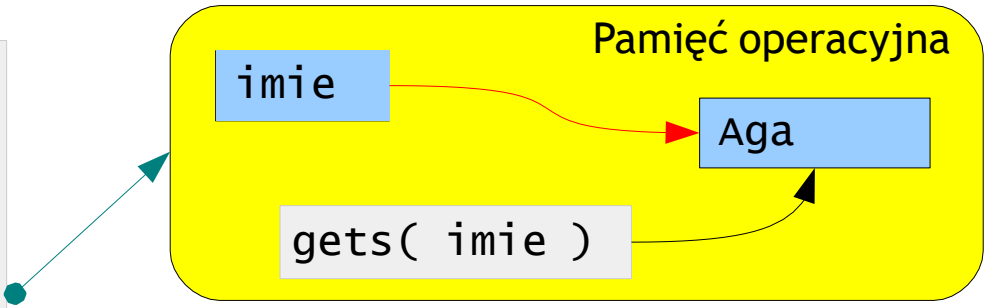
A trzeba np. tak:

```
char imie[ 80 ];

printf( "Podaj imie: " );
gets( imie );
```



Lub tak:

```
char * imie = NULL;

imie = new (nothrow) char[ 80 ];
if( imie != 0 )
{
  printf( "Podaj imie: " );
  gets( imie );
  . . .
  delete [] imie;
}
```



Można tworzyć dynamicznie tablice dowolnych typów

```
double * dochody = 0;
int liczbaMiesiecy;

// Tu ustalenie liczby miesiecy okresu rozrachunkowego

dochody = new (nothrow) double [ liczbaMiesiecy ];
if( dochody != 0 )
{
  for( int miesiac = 0; miesiac < liczbaMiesiecy; miesiac++ )
    dochody[ miesiac ] = 0;
  . . .
  delete [] dochody;
}
```

Można tworzyć dynamicznie tablice dowolnych typów

```
typedef unsigned char byte;

byte * bitmapa = 0;
int rozmiarRysunku;

// Tu ustalenie liczby bajtów rysunku bitmapowego

bitmapa = new (nothrow) byte [ rozmiarRysunku ];
if( bitmapa != 0 )
{
  // Zaladuj bitmape

  // Zrob z nia co trzeba

  // Gdy juz niepotrzebna
  delete [] bitmapa;
}
```