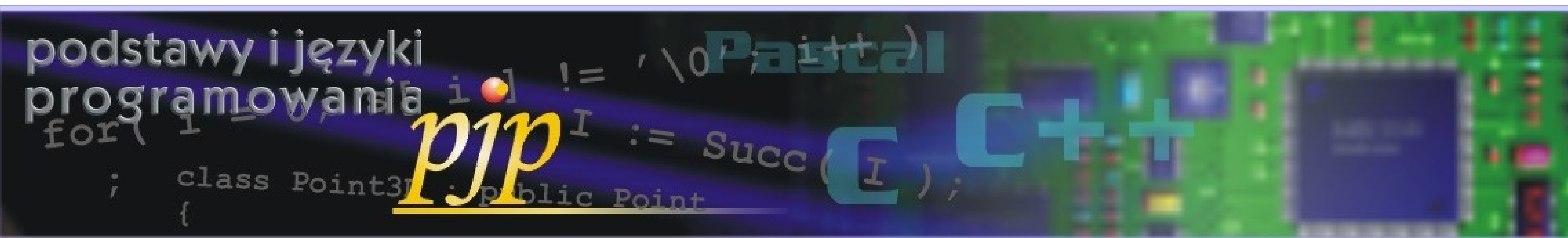
Języki programowania obiektowego

Nieobiektowe elementy języka C++

Roman Simiński

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

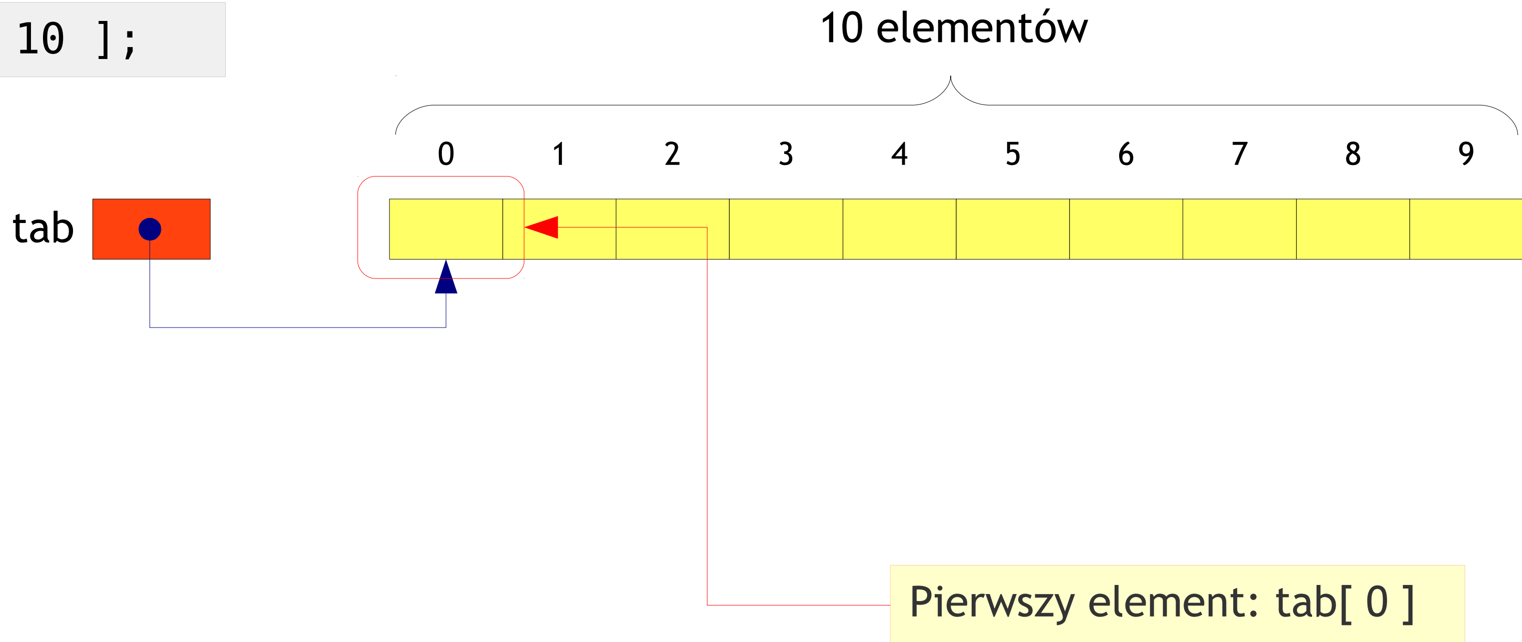
Tablice a zmienne wskaźnikowe



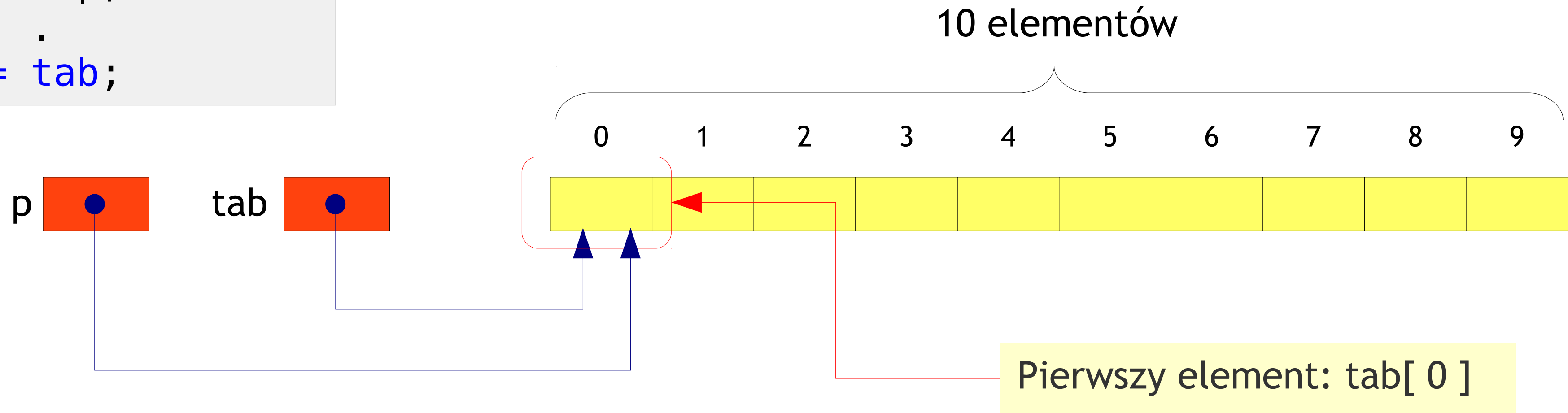
Nazwa tablicy jako wskaźnik na jej początek

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

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



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



Przypisanie:

```
p = tab;
```

Jest równoznaczne z:

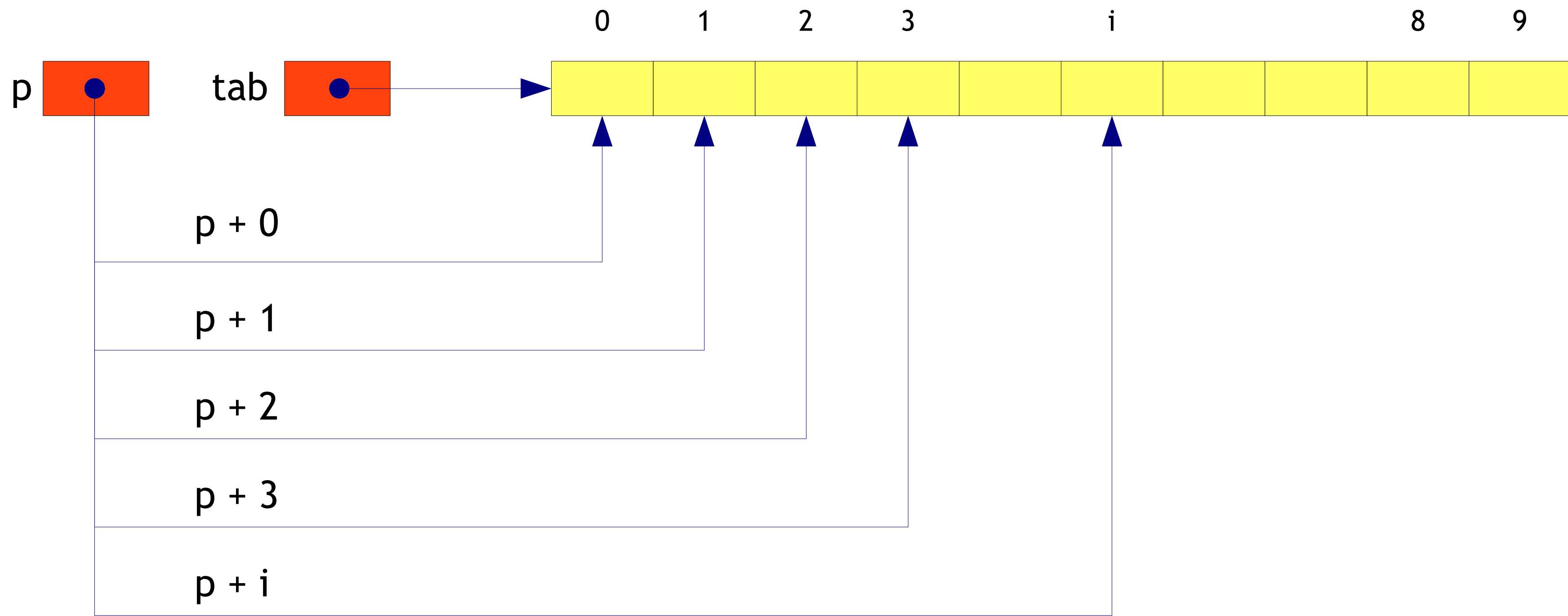
```
p = &tab[ 0 ];
```

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

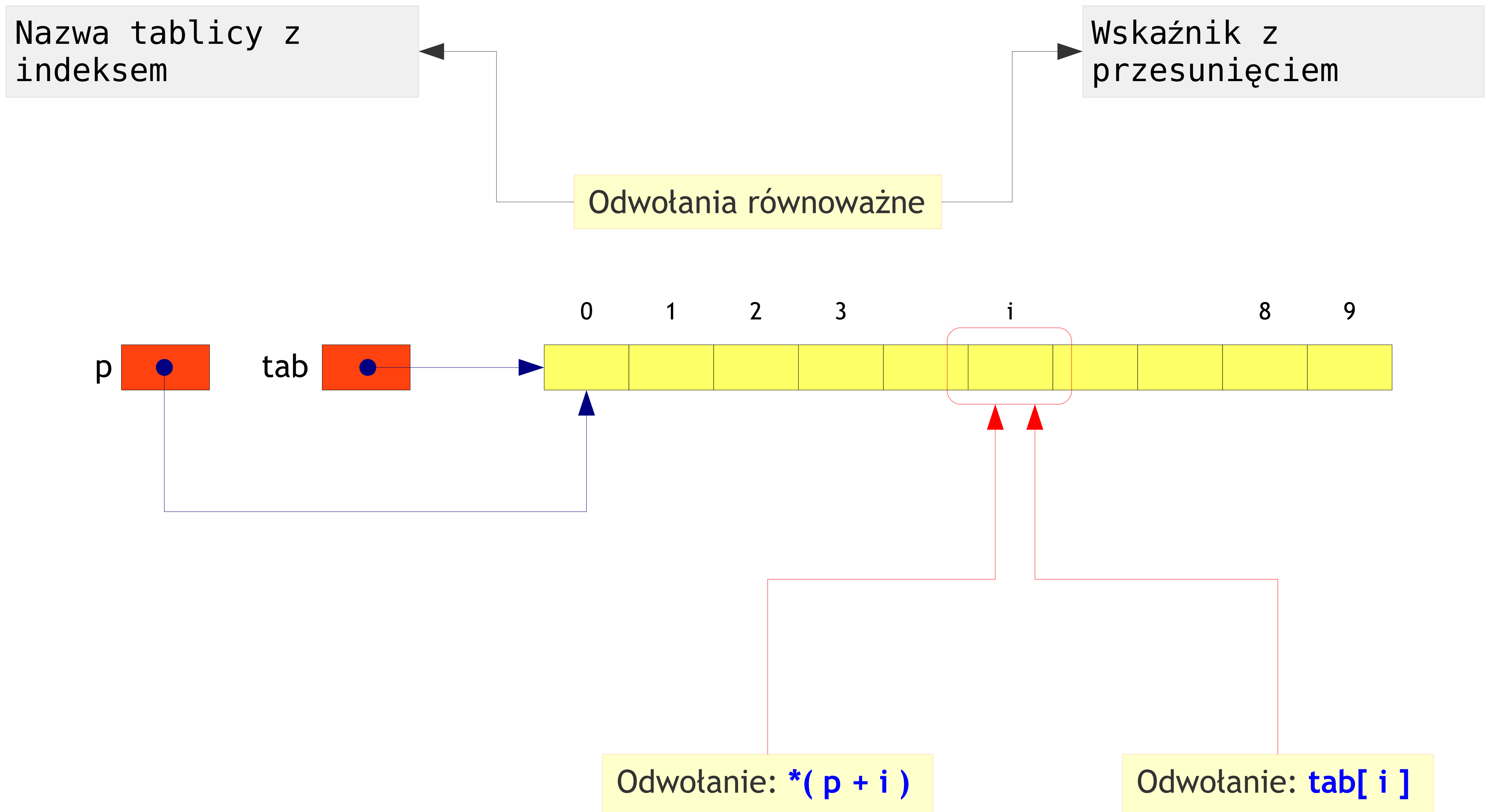
```
tab[ 0 ] = 5;
tab[ 1 ] = 1;
tab[ 2 ] = 10;
. . .
tab[ i ] = 22;
```

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 )`

A także:

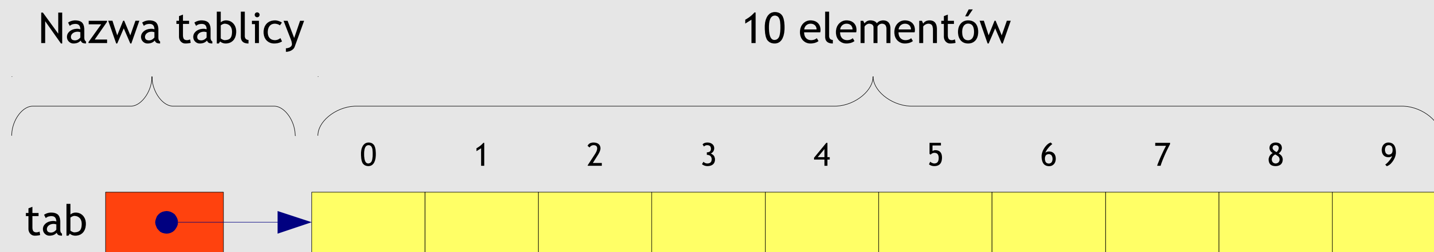
Każde odwołanie: `*( p + i )` można zapisać tak: → `p[ i ];`

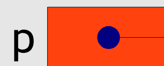
Uwaga, wskaźnik to nie tablica!

```
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;
tab++;


p = tab + 8;
p++;
```

☒ Źle (tab = p; tab++;)

☑ OK (p = tab + 8; 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 )`

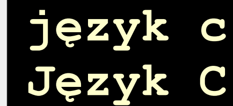
Czy zatem odwołanie:

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

Ciekawostka, cd. ...

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

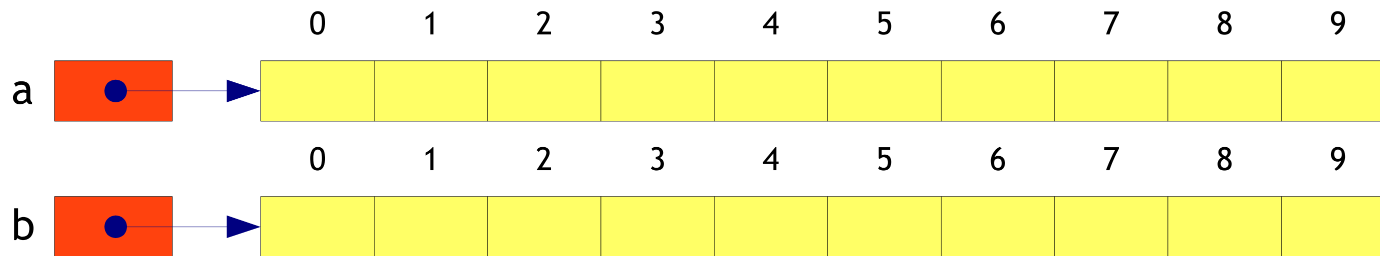
```
char napis[] = "język c";
. . .

cout << napis << endl;

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

cout << napis << endl;
```



Dlaczego nie wolno przypisywać tablic, posługując się ich nazwami?

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

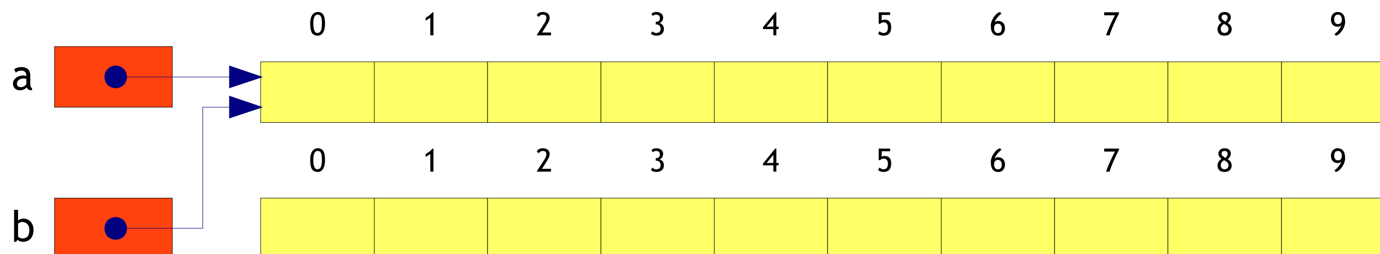
Gdyby przypisywanie było możliwe...



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

Pod lupą — tablice jako parametry funkcji

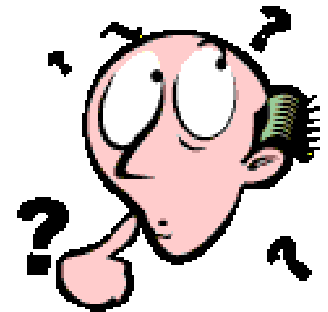
Załóżmy, że funkcja *putString* wyprowadza zawartość tablicy znaków do strumienia wyjściowego programu.

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

putString( napis );
```

Co jest parametrem aktualnym wywołania funkcji *putString*?

- Tablica o nazwie *napis*.

Czy napewno?



- No, właściwie parametrem jest *napis*, a to nazwa tablicy... .

A czym jest nazwa tablicy?

- Nazwa tablicy to ustalony wskaźnik na jej pierwszy element... .

Co z tego wynika?

Pod lupą — tablice jako parametry funkcji

Jak zdefiniujemy parametr formalny funkcji *putString*?

```
void putString( char s[] )
{
  . . .
}
```

Ale co ***naprawdę*** przekazujemy funkcji *putString*?

- Nazwę tablicy, czyli ustalony wskaźnik na jej pierwszy element... .

A zatem definicja parametru formalnego może wyglądać tak:

```
void putString( char * s )
{
  . . .
}
```

Zrozumiałe? Jeżli tak, to czas wyjaśnić do końca sprawę przekazywania tablic jako parametrów funkcji... .

Naga prawda o tablicach przekazywanych do funkcji

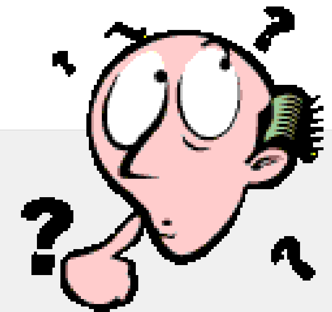
Uwaga — to nie ***same tablice*** są przekazywane jako parametry do funkcji!

Do wnętrza funkcji przekazywane są ***wskaźniki*** na tablice.

Dzięki temu wnętrze funkcji ma dostęp do elementów tablicy.

Paramterem *aktualnym* wywołania funkcji jest **wskaźnik** a nie sama tablica!

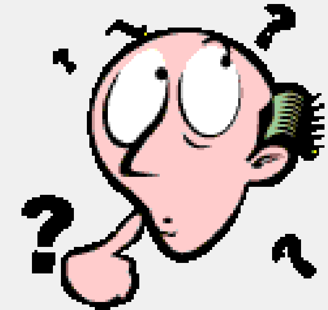
Paramterem *formalnym* funkcji jest **wskaźnik** a nie tablica!



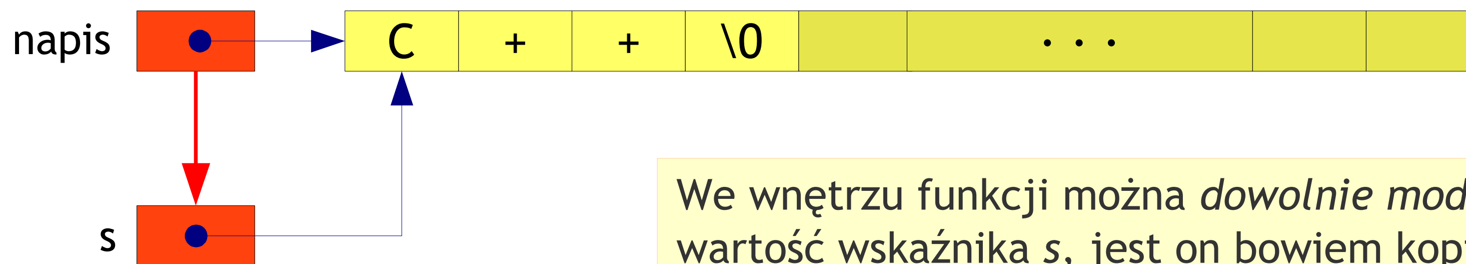
```
char napis[] = "C++";

putString( napis );


void putString( char * s )
{
   . . .
}
```

Naga prawda o tablicach przekazywanych do funkcji

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

putString( napis );


void putString( char * s )
{
  . . .
}
```



Na etapie wywołania funkcji *putString* następuje klasyczne przekazanie parametrów **przez wartość**. Parametr **aktualny** wywołania to *napis*, który jest **wskaźnikiem** na pierwszy element tablicy. **Kopiowany** jest on do paramteru formalnego *s*, który od tego momentu wskazuje na to samo co *napis*.



We wnętrzu funkcji można *dowolnie modyfikować* wartość wskaźnika *s*, jest on bowiem kopią oryginalnej lokalizacji początku tablicy.

Wskaźniki w akcji — metamorfoza funkcji putString

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

putString( napis );
```

Wersja pierwotna

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

Przechodzimy na wskaźniki
Eliminujemy zmienną i

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

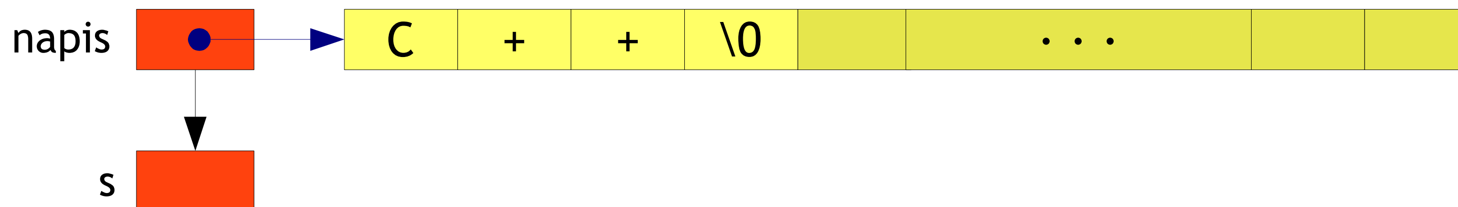
Jak to działa... ?

Wywołanie funkcji putString

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

putString( napis );
```

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

Kopiowanie parametru aktualnego napis do parametru s

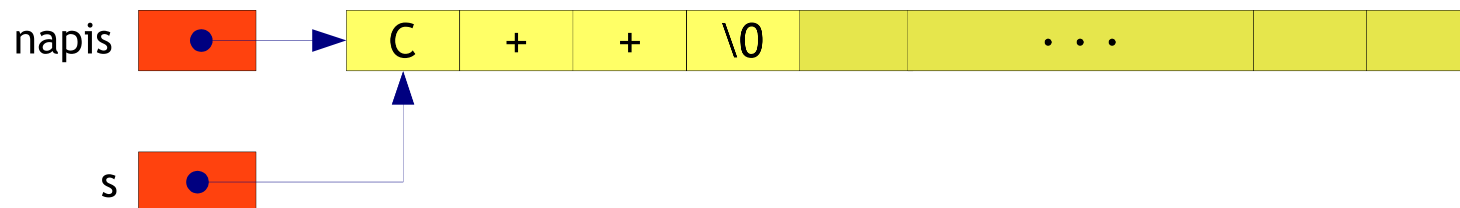


Parametr s jest kopią wskaźnika napis

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

putString( napis );
```

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

Kopiowanie parametru aktualnego napis do parametru s

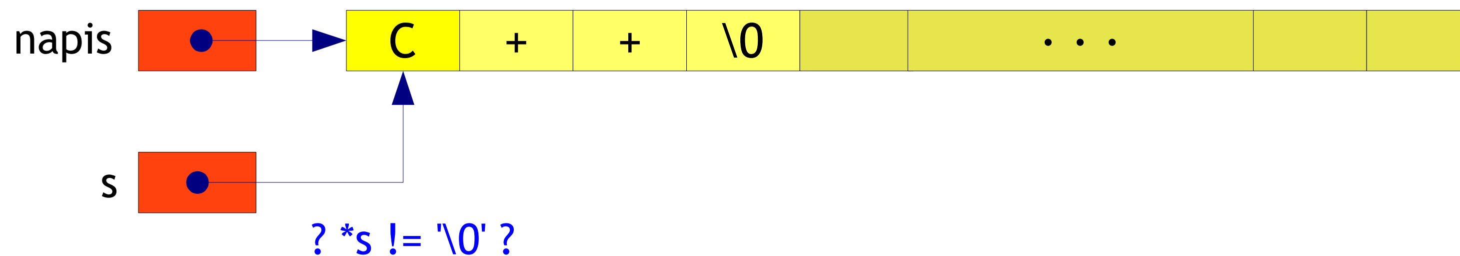


Czy obiekt wskazywany przez s jest znacznikiem końca napisu?

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

putString( napis );
```

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

Parametr s wskazuje na pierwszy element tablicy napis

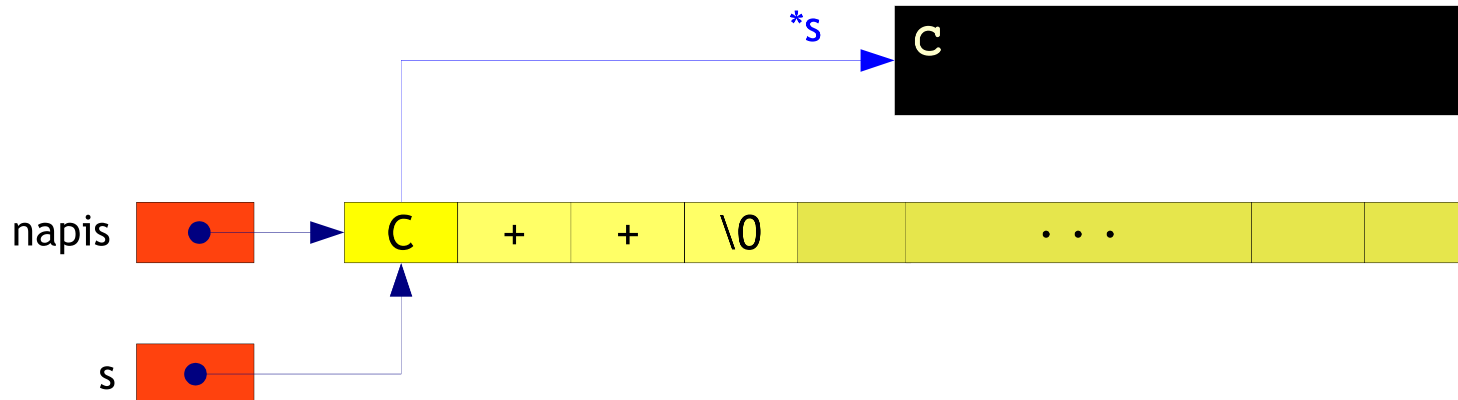


Znak wskazywany przez s wyprowadzamy jest do stdout

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

putString( napis );
```

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

Parametr s wskazuje na pierwszy element tablicy napis

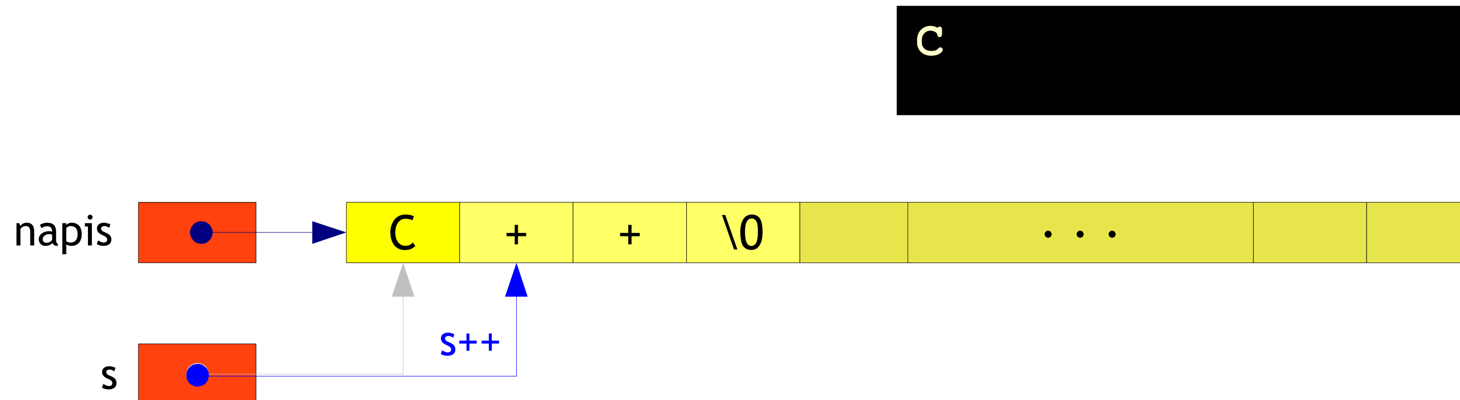


Wskaźnik s przesuwamy na następny znak

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

putString( napis );
```

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

Parametr s wskazuje na kolejny element tablicy napis

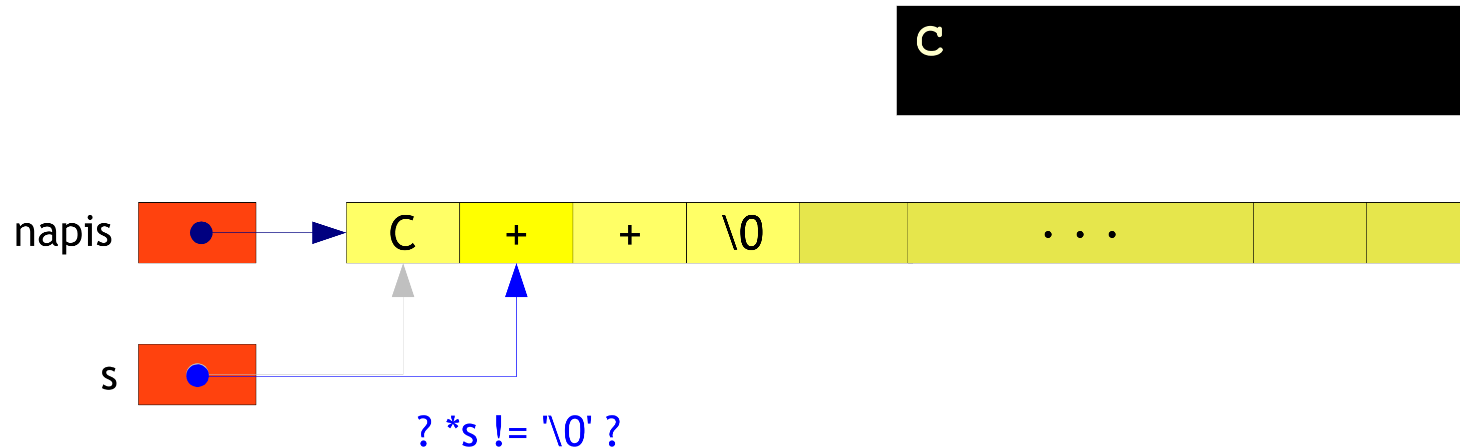


Czy obiekt wskazywany przez s jest znacznikiem końca napisu?

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

putString( napis );
```

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

Parametr s wskazuje na kolejny element tablicy napis

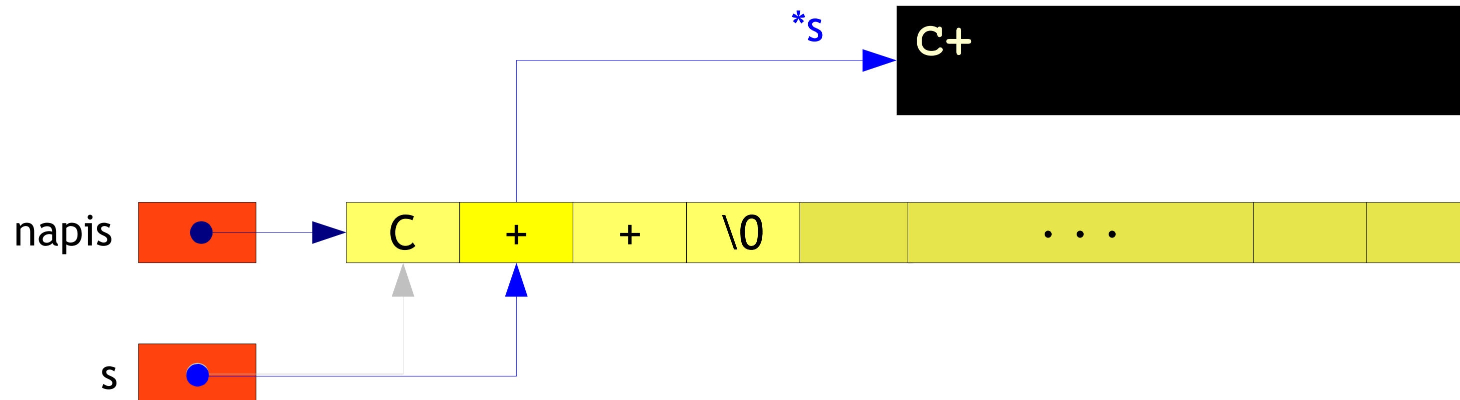


Znak wskazywany przez s wyprowadzamy jest do stdout

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

putString( napis );
```

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

Parametr s wskazuje na kolejny element tablicy napis

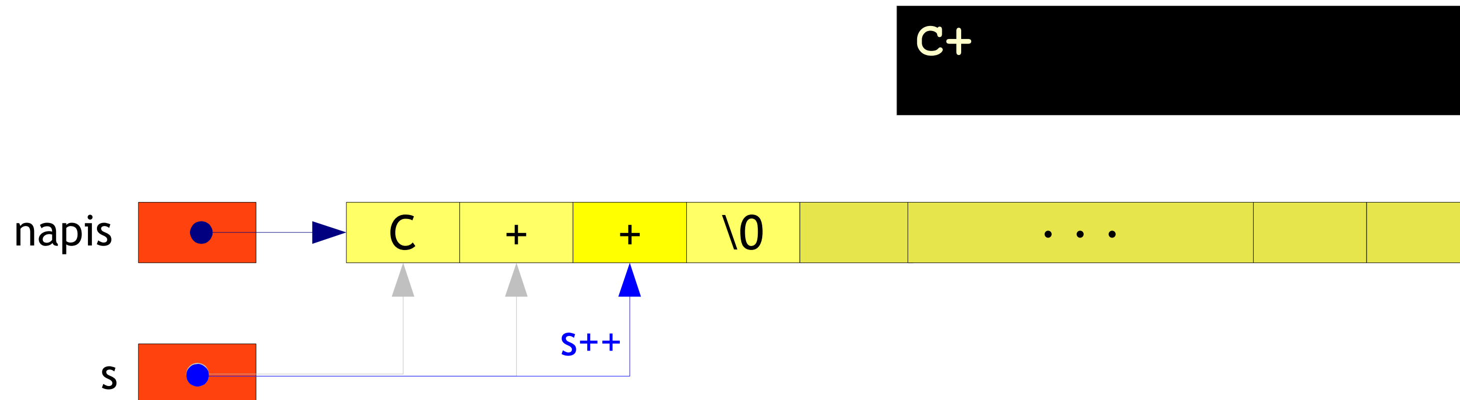


Wskaźnik s przesuwamy na następny znak

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

putString( napis );
```

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

Parametr s wskazuje na kolejny element tablicy napis

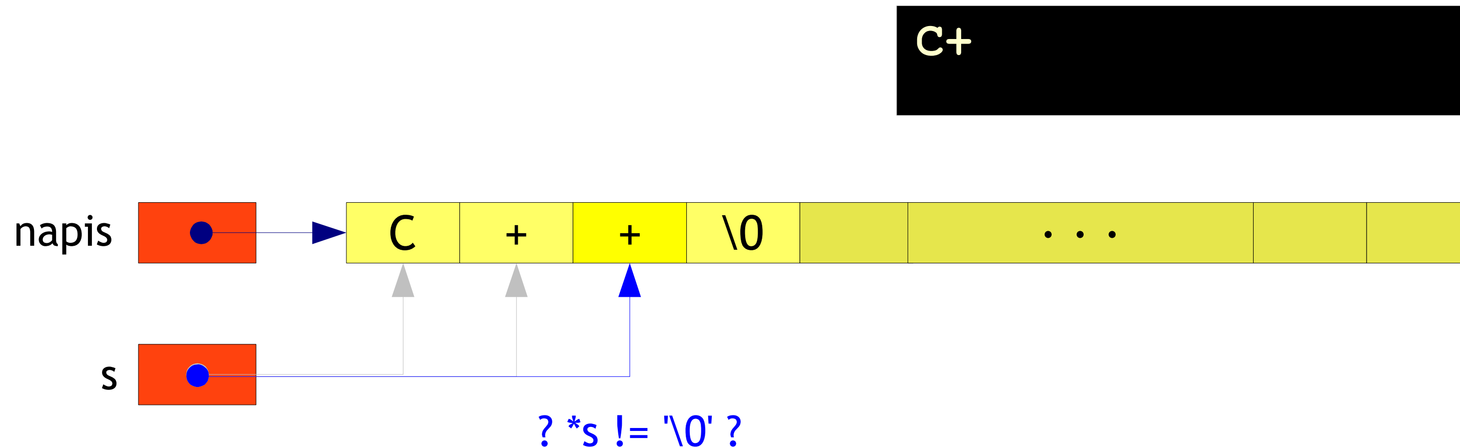


Czy obiekt wskazywany przez s jest znacznikiem końca napisu?

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

putString( napis );
```

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

Parametr s wskazuje na kolejny element tablicy napis

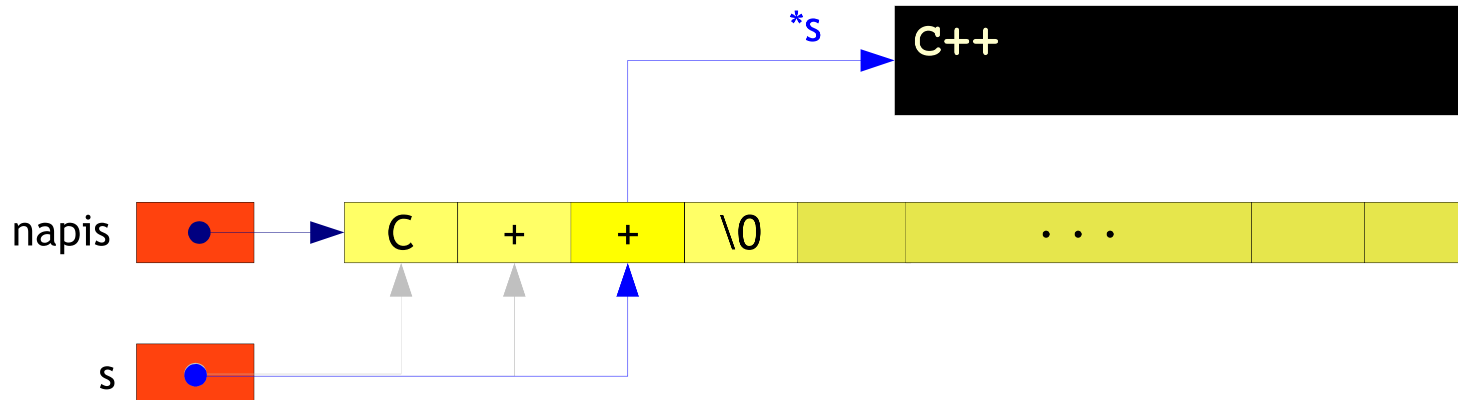


Znak wskazywany przez s wyprowadzamy jest do stdout

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

putString( napis );
```

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

Parametr s wskazuje na kolejny element tablicy napis

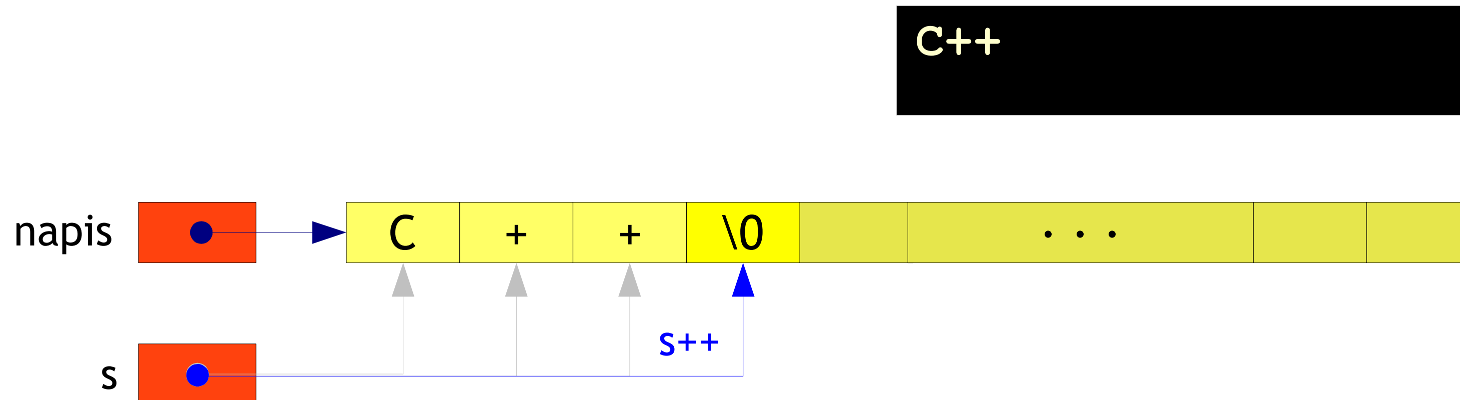


Wskaźnik s przesuwamy na następny znak

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

putString( napis );
```

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

Parametr s wskazuje na kolejny element tablicy napis

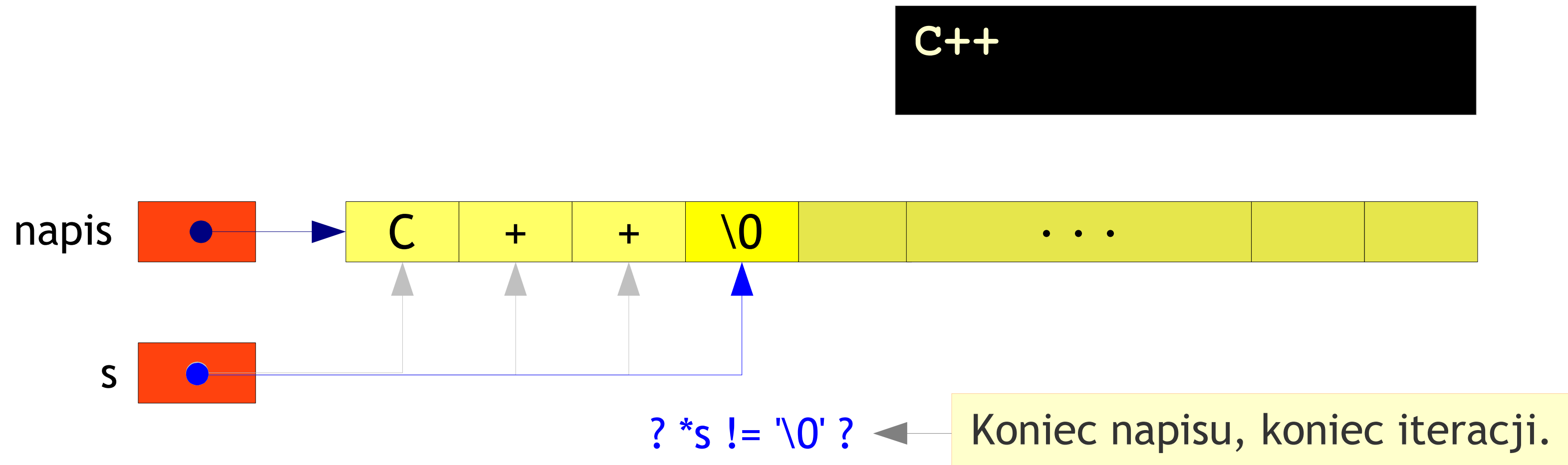


Czy obiekt wskazywany przez s jest znacznikiem końca napisu?

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

putString( napis );
```

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

Parametr s wskazuje na kolejny element tablicy napis

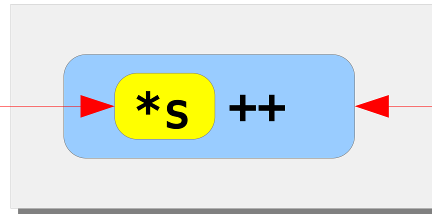


Wskaźniki w akcji — koniec metamorfozy

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

„Kompresja” iteracji for

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



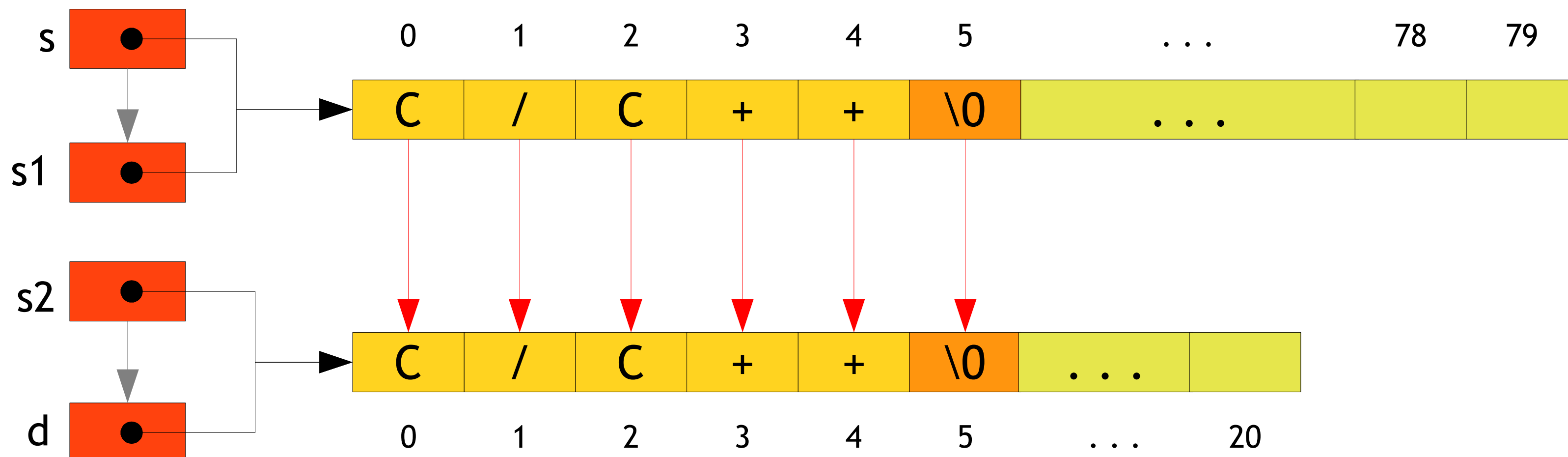
Potem zwiększ o jeden wartość wskaźnika **s** — będzie on wtedy wskazywał na następny element tablicy.

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

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

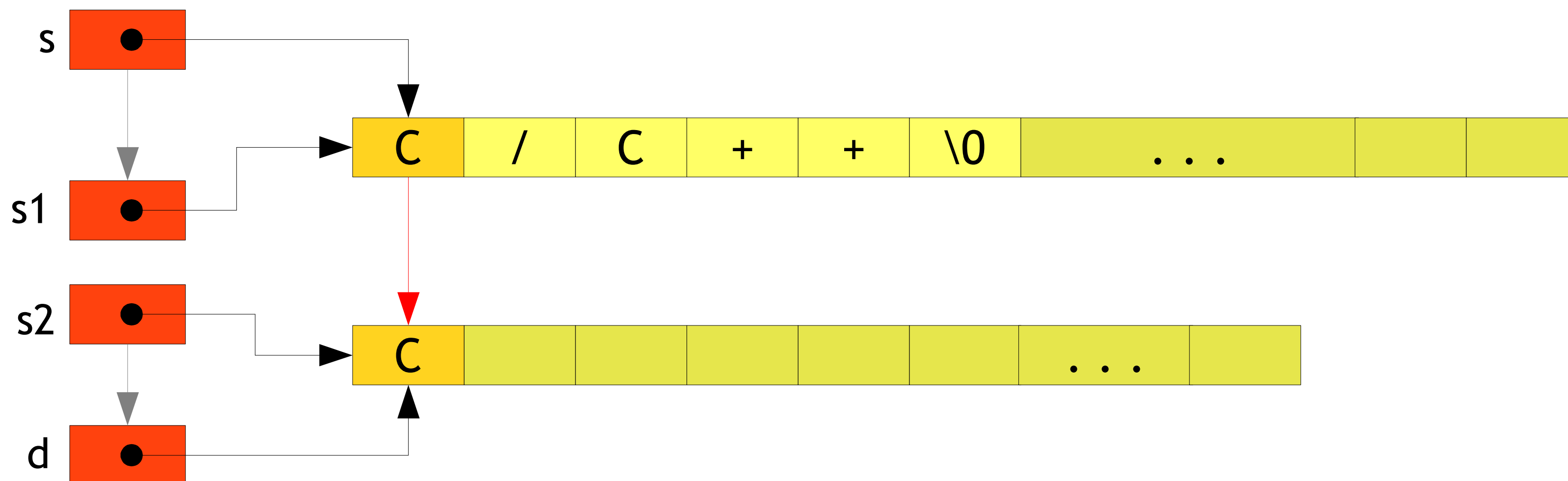
Wskaźniki pod lupą — funkcja *strcpy* w wersji klasycznej

```
void strcpy( char d[], char s[] ))
{
  int i = 0;
  while( s[ i ] != '\0' )
  {
    d[ i ] = s[ i ];
    i++;
  }
  d[ i ] = '\0';
}
. . .
char s1[ 80 ] = "C/C++";
char s2[ 20 ];
. . .
strcpy( s2, s1 );
```



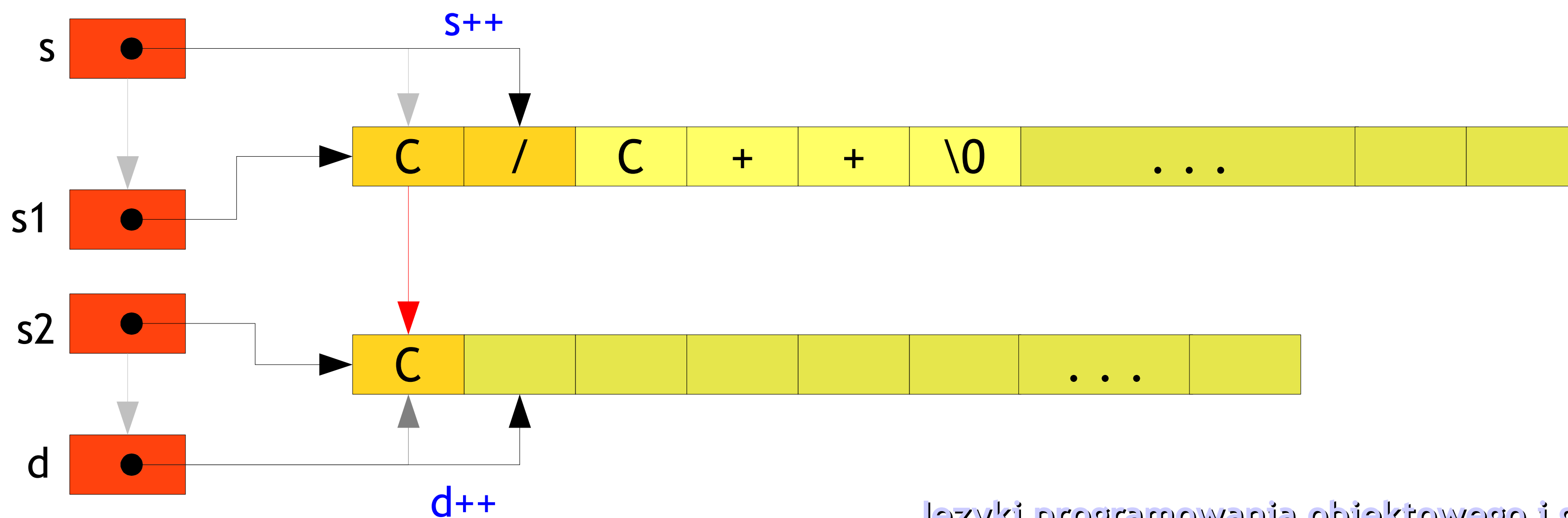
Wskaźniki pod lupą — metamorfoza funkcji *strcpy*, wersja naiwna

```
void strcpy( char * d, char * s )
{
  while( *s != '\0' )
  {
     *d = *s;
     d++;
     s++;
  }
  *d = '\0';
}
. . .
char s1[ 80 ] = "C/C++";
char s2[ 20 ];
. . .
strcpy( s2, s1 );
```



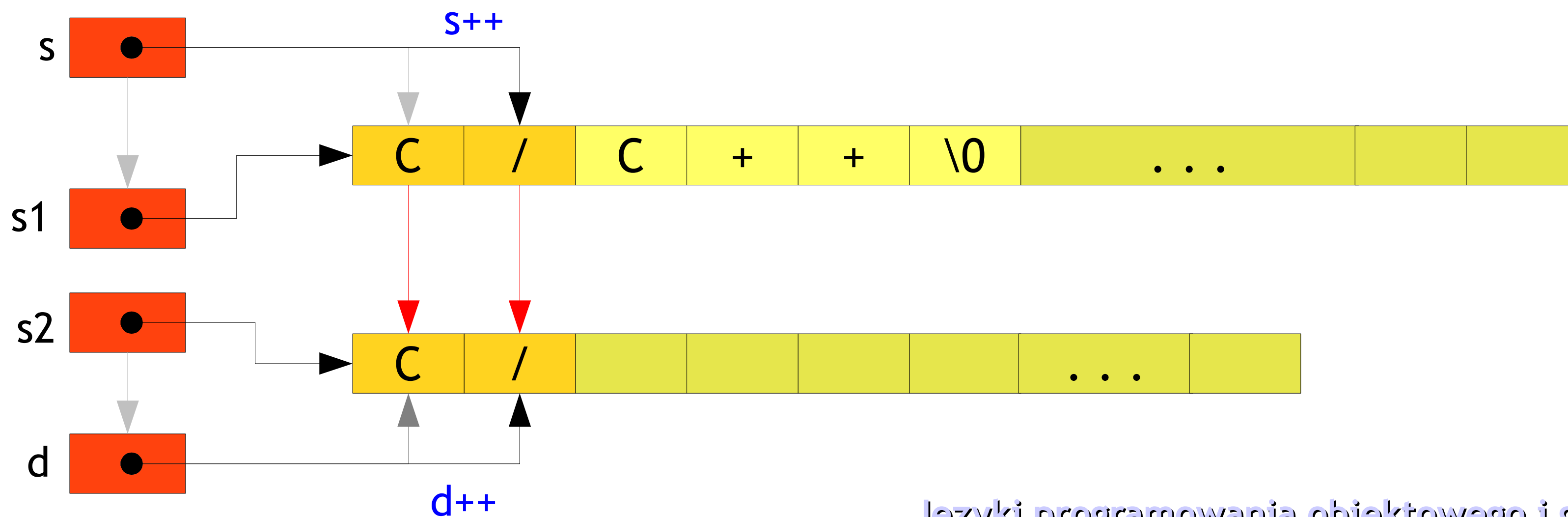
Wskaźniki pod lupą — metamorfoza funkcji *strcpy*, wersja naiwna

```
void strcpy( char * d, char * s )
{
  while( *s != '\0' )
  {
     *d = *s;
     d++;
     s++;
  }
  *d = '\0';
}
. . .
char s1[ 80 ] = "C/C++";
char s2[ 20 ];
. . .
strcpy( s2, s1 );
```



Wskaźniki pod lupą — metamorfoza funkcji *strcpy*, wersja naiwna

```
void strcpy( char * d, char * s )
{
  while( *s != '\0' )
  {
     *d = *s;
     d++;
     s++;
  }
  *d = '\0';
}
. . .
char s1[ 80 ] = "C/C++";
char s2[ 20 ];
. . .
strcpy( s2, s1 );
```

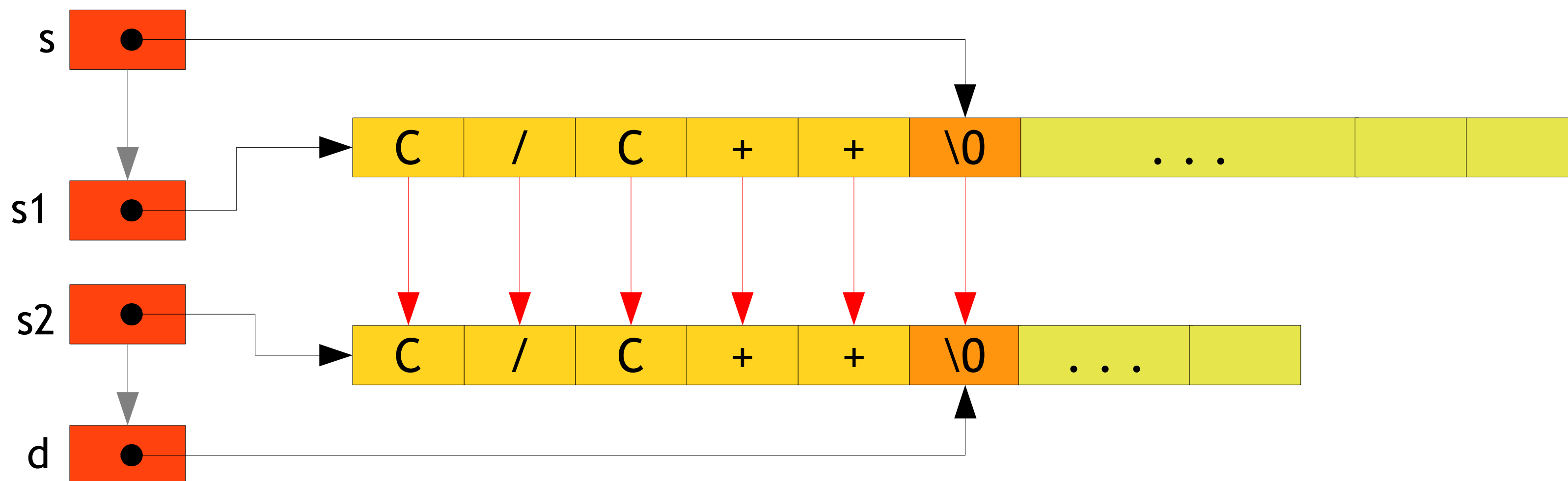


Wskaźniki pod lupą — metamorfoza funkcji *strcpy*, wersja naiwna

I tak dalej, aż do przepisania ostatniego znaku…

Wskaźniki pod lupą — metamorfoza funkcji *strcpy*, wersja naiwna

```
void strcpy( char * d, char * s )
{
  while( *s != '\0' )
  {
     *d = *s;
     d++;
     s++;
  }
  *d = '\0';
}
. . .
char s1[ 80 ] = "C/C++";
char s2[ 20 ];
. . .
strcpy( s2, s1 );
```

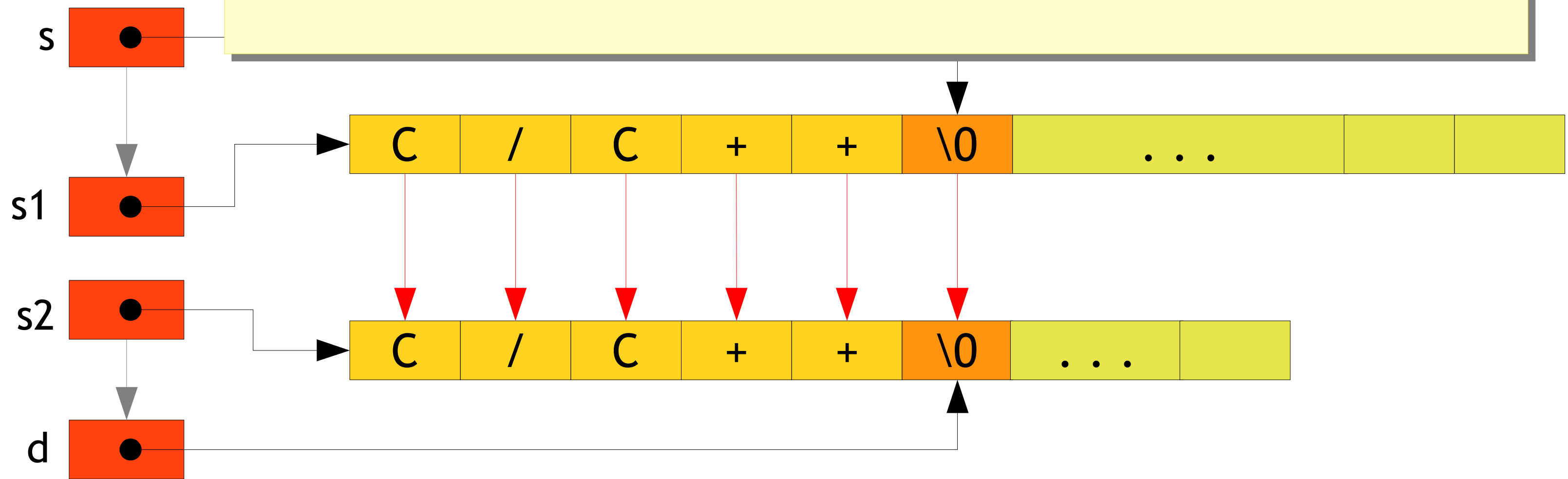


Wskaźniki pod lupą — metamorfoza funkcji *strcpy*, wersja naiwna

```
void strcpy( char * d, char * s )
{
  while( *s != '\0' )
  {
     *d = *s;
     d++;
     s++;
  }
  *d = '\0';
}
. . .
char s1[ 80 ]
char s2[ 20 ];
. . .
strcpy( s2, s1
```

Dlaczego wersja naiwna?

Wprowadzono odwołania wskaźnikowe, ale to właściwie niewiele zmienia, poza wyeliminowaniem zmiennej *i*.



Wskaźniki pod lupą — metamorfoza funkcji *strcpy*

„Kompresja” — krok pierwszy

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

```
*d = *s;
d++;
s++;
```

„Kompresja” — krok drugi

```
void strcpy( 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

```
( *d++ = *s++ ) != '\0'
```

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

Wskaźniki pod lupą — metamorfoza funkcji *strcpy*

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

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

```
( *d++ = *s++ ) != '\0'
```

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

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

Tablica d jako rezultat funkcji

Uwaga, opuszczenie `!= '\0'` może powodować powstanie *ostrzeżenia* ze strony kompilatora — może on podejrzewać, że pomyliliśmy operatory `=` i `==`.

Wskaźniki pod lupą — dlaczego strcpy udostępnia d jako rezultat?

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:

```
strcpy( s2, s1 );
strcpy( s3, s2 );
cout << s3;
```

Można zapisać krócej:

```
cout << strcpy( s3, strcpy( s2, s1 ) );
```

Rezultat funkcji operujących na napisach bardzo często wydaje się mało użyteczny, jednak jego wykorzystanie pozwala na stosowanie sztuczek i trików.

Wskaźniki pod lupą — użycie modyfikatora const, przypomnienie

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

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

Tablica źródłowa jest chroniona

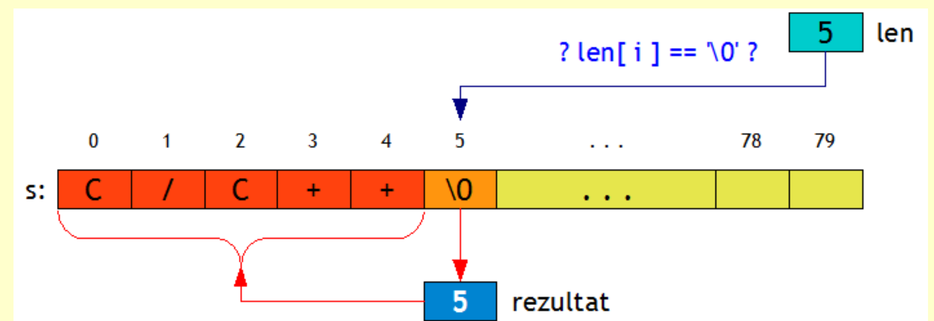
Message
In function `char* strcpy(char*, const char*)':
assignment of read-only location

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 ;
}
```

Wyznaczenie długości napisu polega na odnalezieniu znacznika końca napisu, jego indeks określa, ile jest przed nim znaków.



Realizacja w wykorzystaniem iteracji *for*:

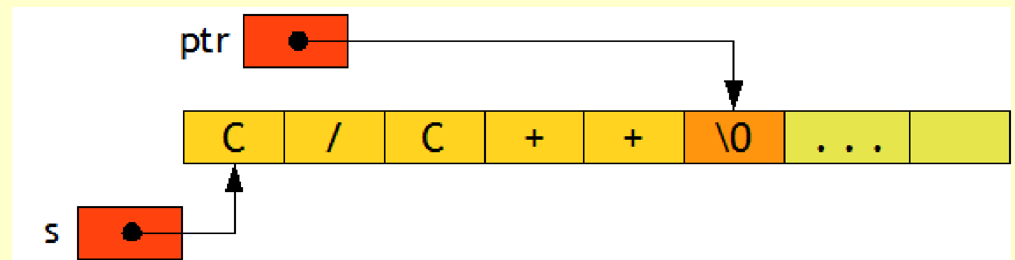
```
int strlen( char s[] )
{
  int len;

  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 ) ;
}
```

Wyznaczenie długości napisu polega na wyznaczeniu różnicy pomiędzy „adresem” znacznika końca a „adresem” pierwszego elementu tablicy.



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;
}
```

Jak to działa? To już było przy omawianiu tablic znaków.

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;
}
```

Jak to działa? Analiza na zadanie domowe..., ta funkcja to fajne zadanie na egzamin :)

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*.

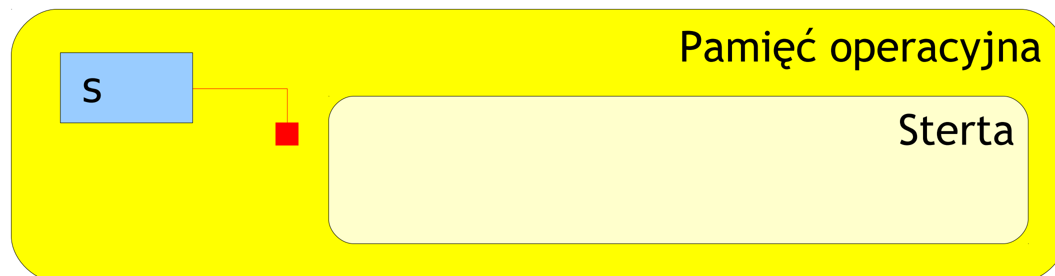
```
char * s = 0;
int n;

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

s = new char [ n ];   // A właściwie: new (nothrow) char [ n ]

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

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

Dynamiczna alokacja tablic — etap 1-szy

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

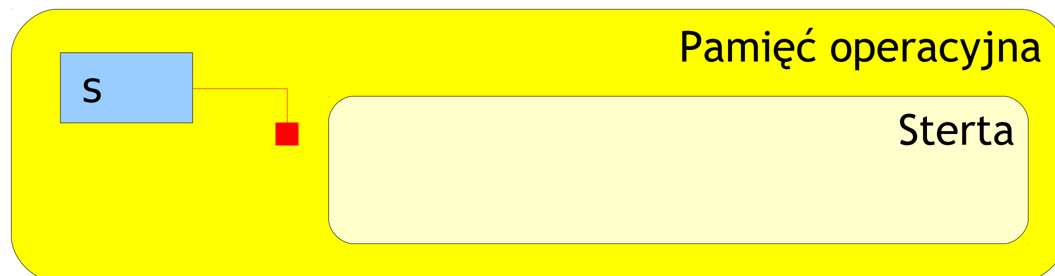
```
char * s = 0;
int n;

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

s = new char [ n ];   // A właściwie: new (nothrow) char [ n ]

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



Dynamiczna alokacja tablic — etap 2-gi

Zwykle korzystamy ze zmiennej, która pozwoli zapamiętać ilu elementowej tablicy potrzebujemy.

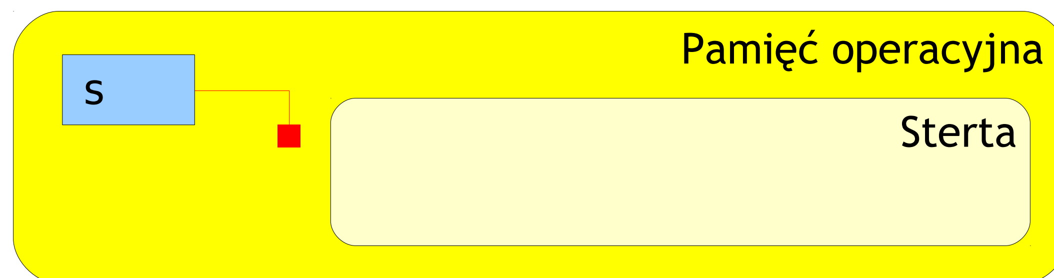
```
char * s = 0;
int n;

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

s = new char [ n ];   // A właściwie: new (nothrow) char [ n ]

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



Dynamiczna alokacja tablic — etap 3-ci

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

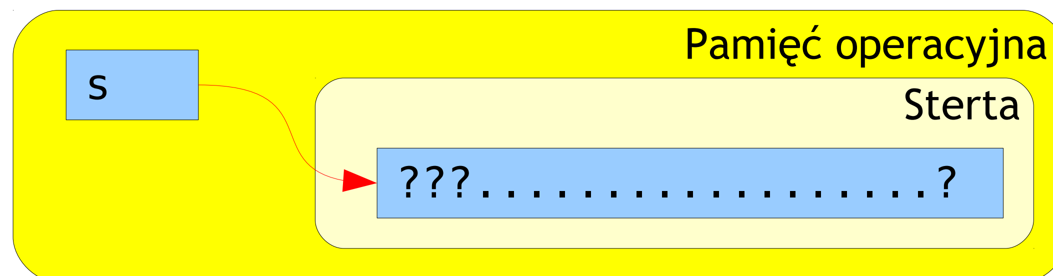
```
char * s = 0;
int n;

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

s = new char [ n ];    // A właściwie: new (nothrow) char [ n ]

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



Dynamiczna alokacja tablic — etap 4-ty

Przydział pamięci dla tablicy — operator *new* otrzymuje *typ* elementów tablicy oraz ich *liczbę*, w tym przypadku *n*.

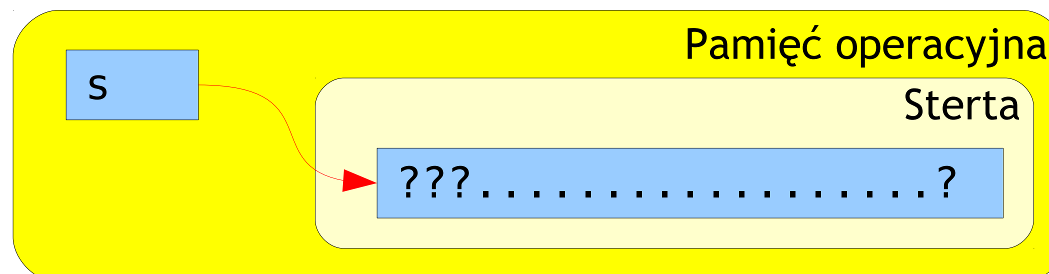
```
char * s = 0;
int n;

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

s = new char [ n ];   // A właściwie: new (nothrow) char [ n ]

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



Dynamiczna alokacja tablic — etap 5-ty

Kontrola poprawności przydziału pamięci (dla operatora w wersji nie generującej wyjątków). Przydzielony obszar pamięci ma przypadkową zawartość.

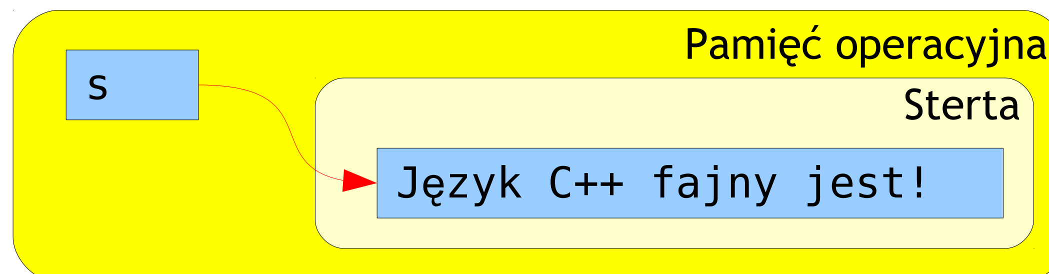
```
char * s = 0;
int n;

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

s = new char [ n ];   // A właściwie: new (nothrow) char [ n ]

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



Dynamiczna alokacja tablic — etap 6-ty

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

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

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

s = new char [ n ];    // A właściwie: new (nothrow) char [ n ]

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

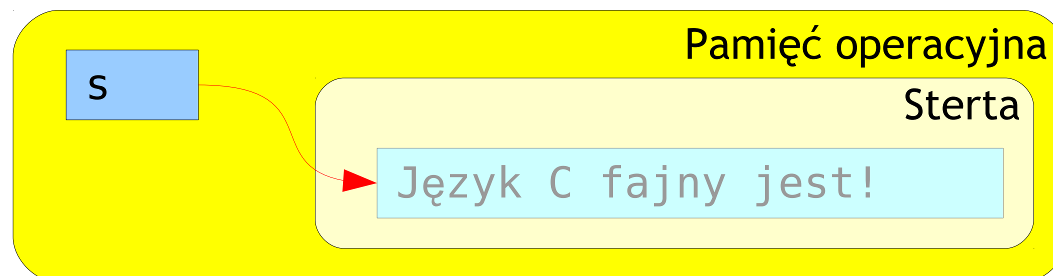


Dynamiczna alokacja tablic — etap 7-ty

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 = 0;
int n;

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

s = new char [ n ];   // A właściwie: new (nothrow) char [ n ]

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

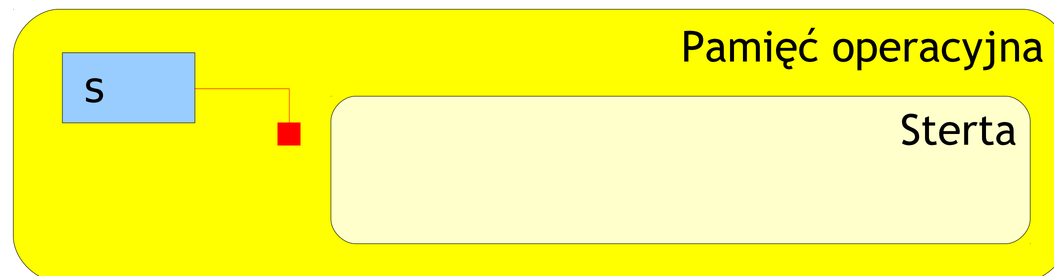
Ta wersja operatora **delete []** odpowiedzialna jest za bezpieczne usunięcie całej tablicy elementów.



Dynamiczna alokacja tablic — zerowanie wskaźnika

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 char [ n ];   // A właściwie: new (nothrow) char [ n ]

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



Dynamiczna alokacja tablic — new generujący wyjątki

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( ... ) // Wersja uproszczona, dokładniej catch( std::bad_alloc & e )
{
  cout << "Brak pamięci dla wykonania tej operacji";
}
```

Dynamiczna alokacja tablic — zerowanie wskaźnika

Zerowanie wskaźnika po zwolnieniu pamięci jest zawsze 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( ... ) // Wersja uproszczona, dokładniej catch( std::bad_alloc & e )
{
  cout << "Brak pamięci dla wykonania tej operacji";
}
```

Dynamiczna alokacja tablic — konwencja języka C

W języku C dynamiczny przydział pamięci nie jest częścią języka. Przydział ten realizują funkcje biblioteczne *malloc*, *calloc*, *realloc*, *free*.

```
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 );
}
```

Nie należy mieszać operatorów *new* i *delete* z funkcjami *malloc*, *calloc*, *realloc* i *free*. Należy trzymać się jednej konwencji zarządania pamięcią.

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.

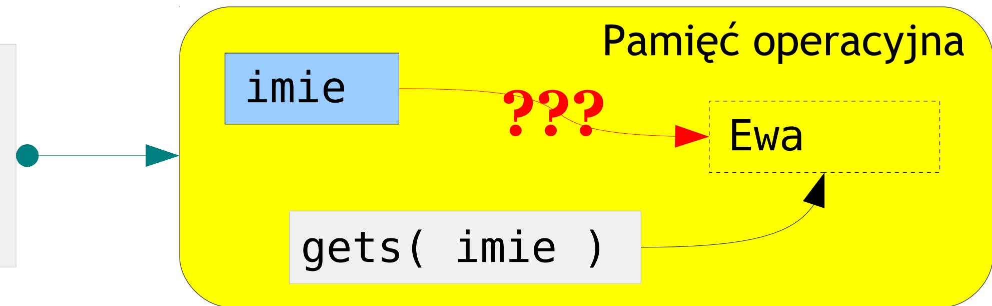
gets collects a string of characters terminated by a new line from the standard input stream stdin and puts it into s. The new line is replaced by a null character (\0) in s.

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!

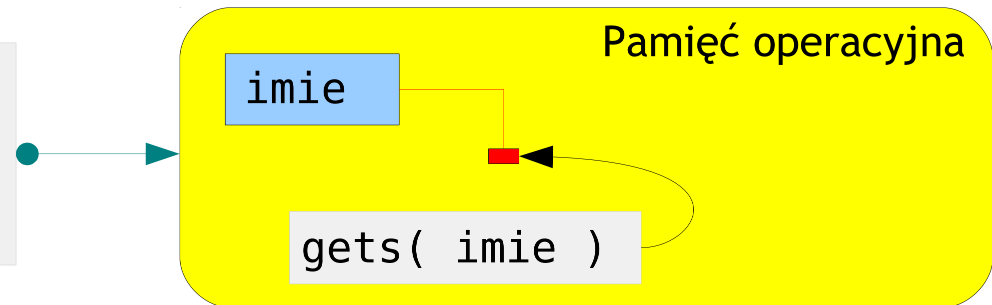
Niedokładna lektura dokumentacji może sugerować, że funkcji należy użyć tak:

```
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:

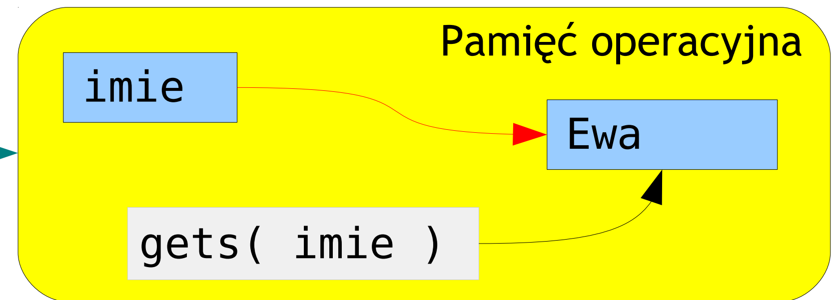
```
Null pointer assignment
```

Niektóre implementacje funkcji *gets* przerywają działanie gdy parametr jest wskaźnikiem pustym.

Ważna sprawa — ostrożnie z parametrami wskaźnikowymi!

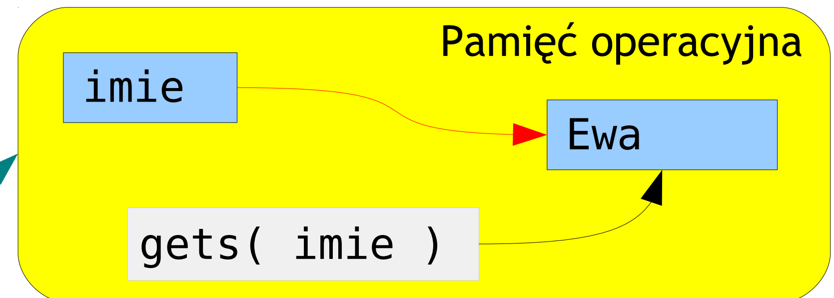
A trzeba np. tak:

```
char imie[ 80 ];

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



Lub tak:

```
char * imie = 0;

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



Można tworzyć dynamicznie tablice dowolnych typów

Tablica elementów typu *double*, liczba elementów określona przez użytkownika.

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

do
{
  cout << endl << "Podaj liczbę miesiecy okresu rozrachunkowego: ";
  cin >> liczbaMiesiecy;
  if( liczbaMiesiecy <= 0 || liczbaMiesiecy > 12 )
    cout << "Okres rozrachunkowy to od 1 do 12 miesięcy";
}
while( liczbaMiesiecy <= 0 || liczbaMiesiecy > 12 )

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

Można tworzyć dynamicznie tablice dowolnych typów

Załóżmy, że mamy wykonać operację na bitmapie w 256 kolorach.

```
typedef unsigned char byte;

byte * bitmapa = 0;
int liczbaPikseli;

// Tu ustalenie liczby pikseli rysunku bitmapowego

bitmapa = new (nothrow) byte [ liczbaPikseli ];
if( bitmapa != 0 )
{
  // Załaduj bitmapę

  for( int nrPiksela = 0; nrPiksela < liczbaPikseli; ++nrPiksela )
    // Jakaś operacja na pikselu bitmapy: bitmapa[ nrPiksela ]

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

Czy to są dynamiczne tablice?

- W C i C++ na razie nie ma *tablic dynamicznych*.
- W standardzie C99 występują tablice o *zmiennej długości* — rozmiar tablicy może być ustalany na etapie wykonania programu, jednak po ustaleniu liczby elementów tablicy, nie może być już ona zmieniana.
- Przedstawiony wcześniej mechanizm *dynamicznego tworzenia tablic* zastępuje mechanizm *tablic dynamicznych* i w zupełności wystarcza w C.
- Wystarcza również w C++, jednak istnieje tendencja do wykorzystywania klas pojemnikowych oferowanych np. przez bibliotekę STL.
- Klasy te pozwalają na wygodne, skuteczne i elestyczne manipulowanie kolekcjami obiektów. Są też bezpieczniejsze.
- Jest to jednak okupione sporym narzutem kodu — biblioteki pojemnikowe są dość złożone.