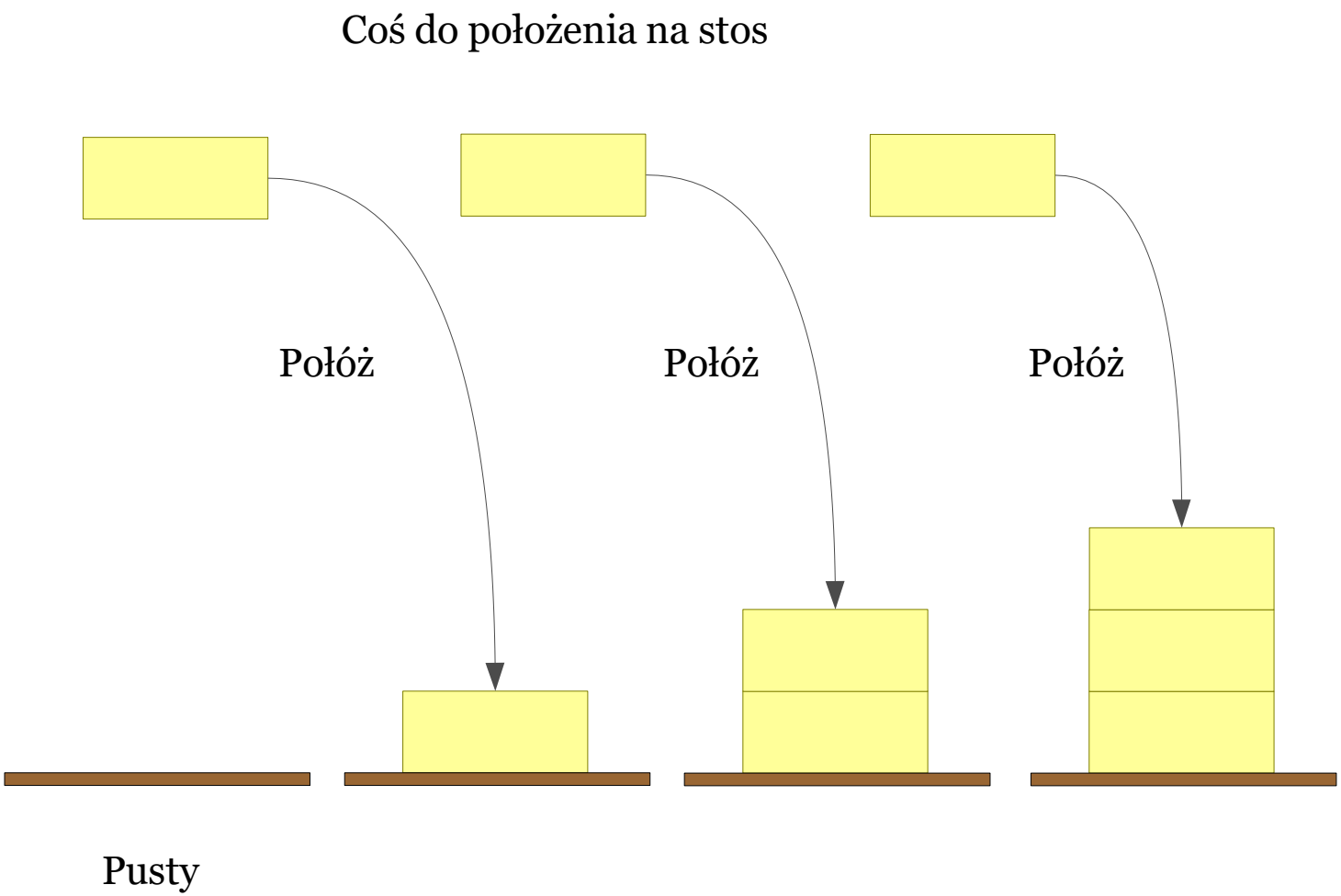
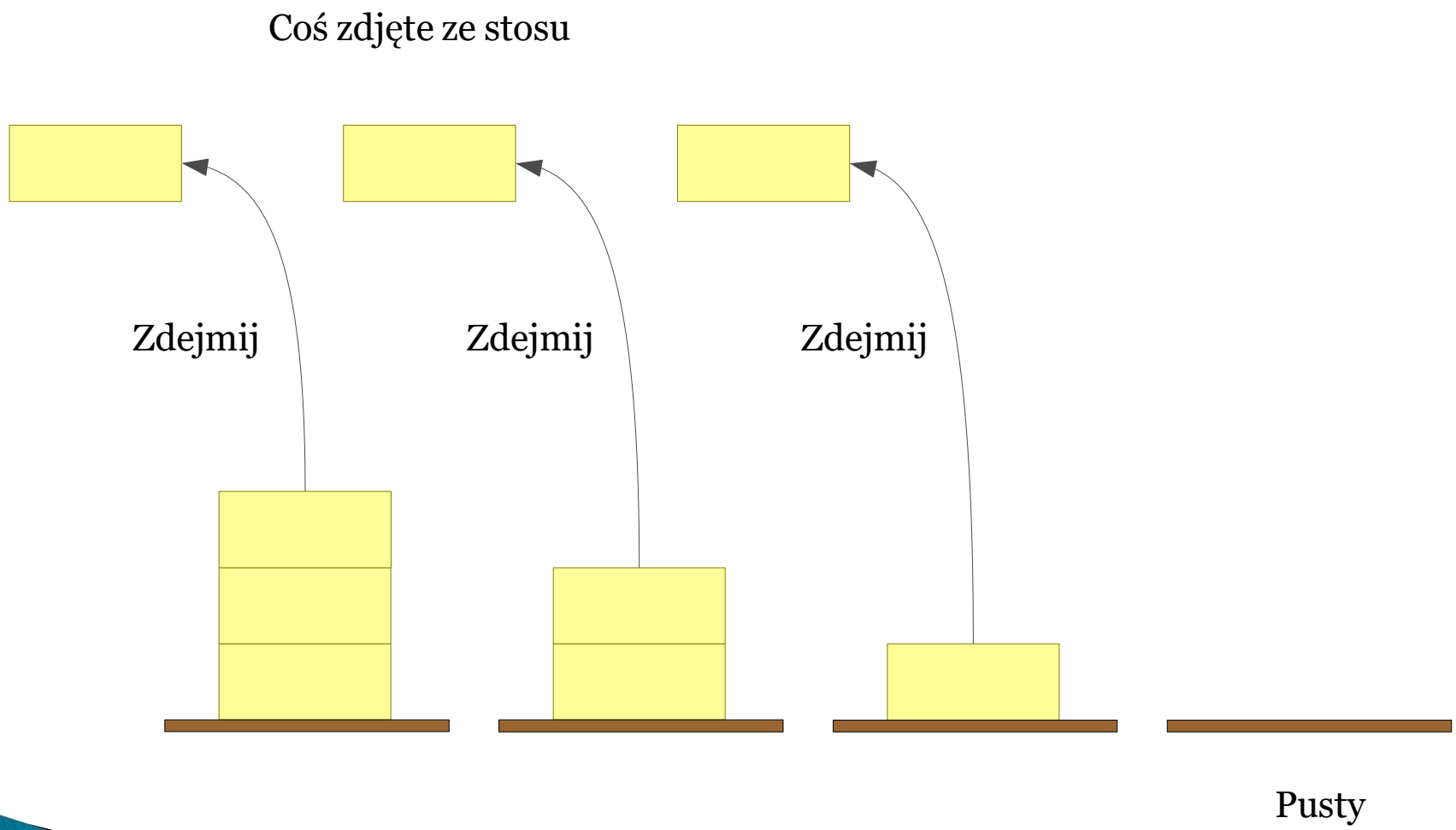
Programowanie obiektowe

Dynamiczne struktury danych, zarządzanie pamięcią.
Mechanizm wyjątków.
Klasy szablonowe.

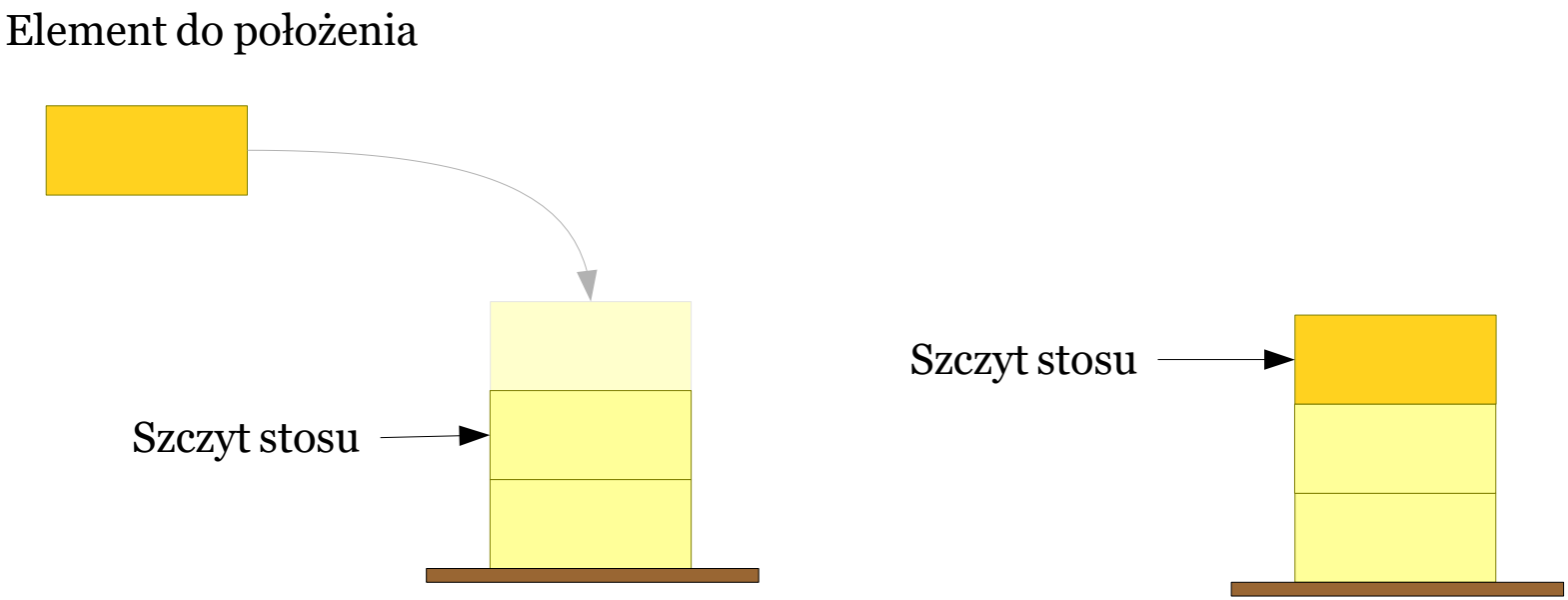
Stos – koncepcja



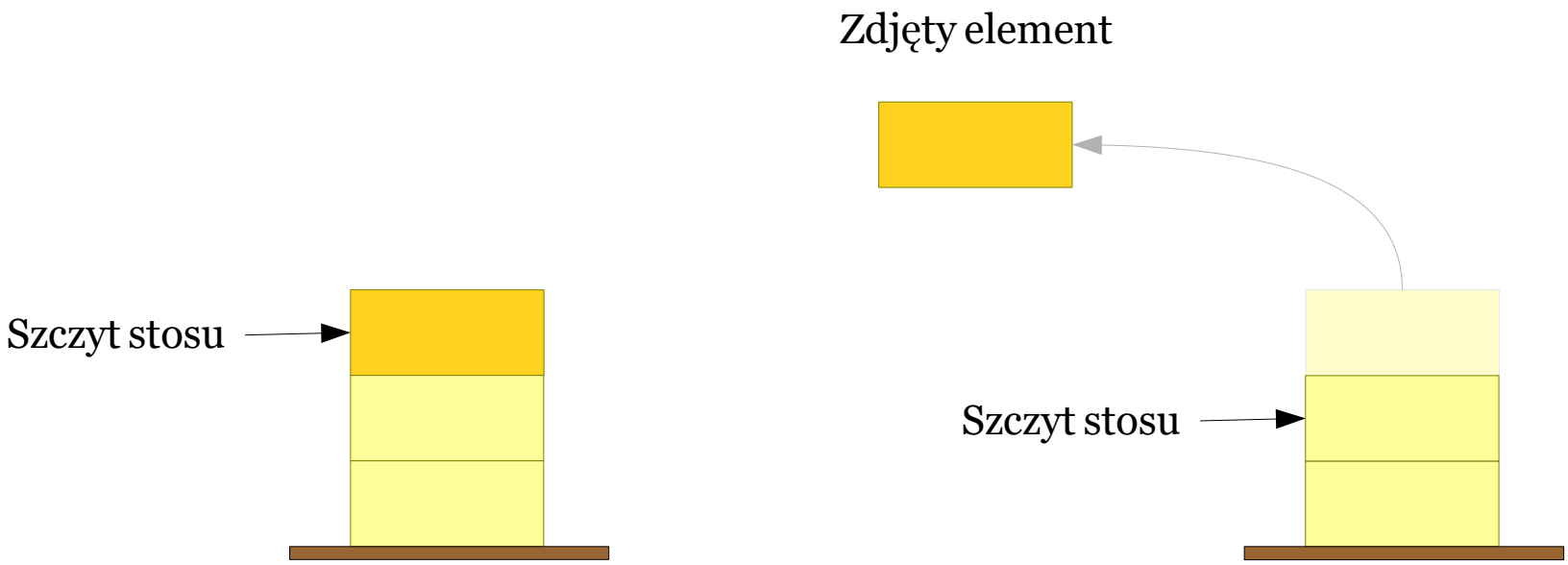
Stos – koncepcja, cd. ...



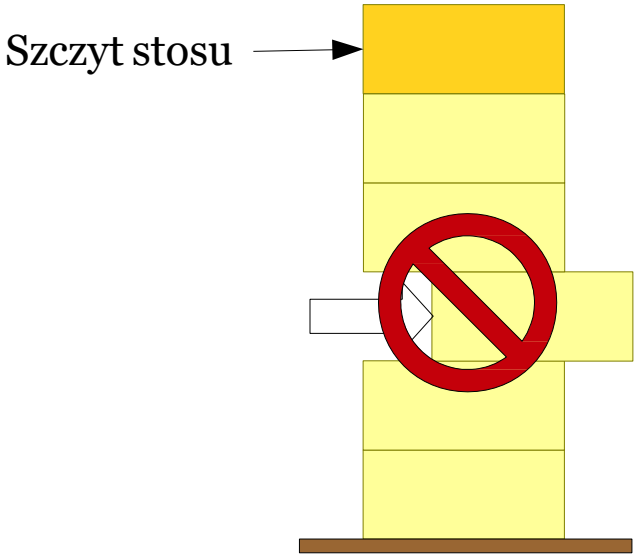
Stos, operacje: połóż (push)



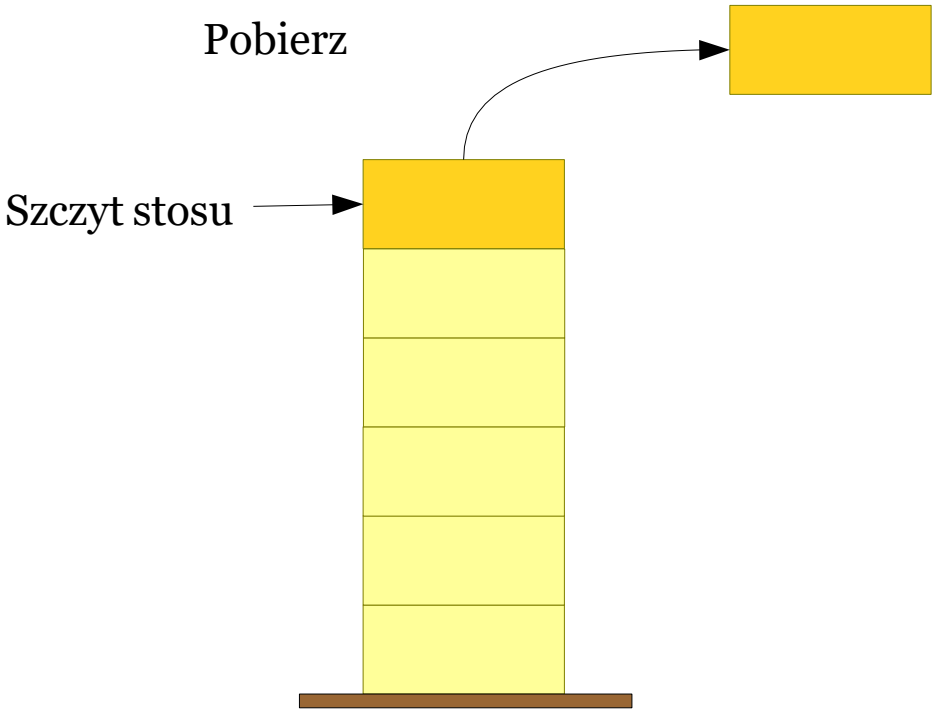
Stos, operacje: zdejmij (pop)



Stos, operacje zabronione



Stos, operacja dodatkowa: pobierz, szczyt (peek, top)

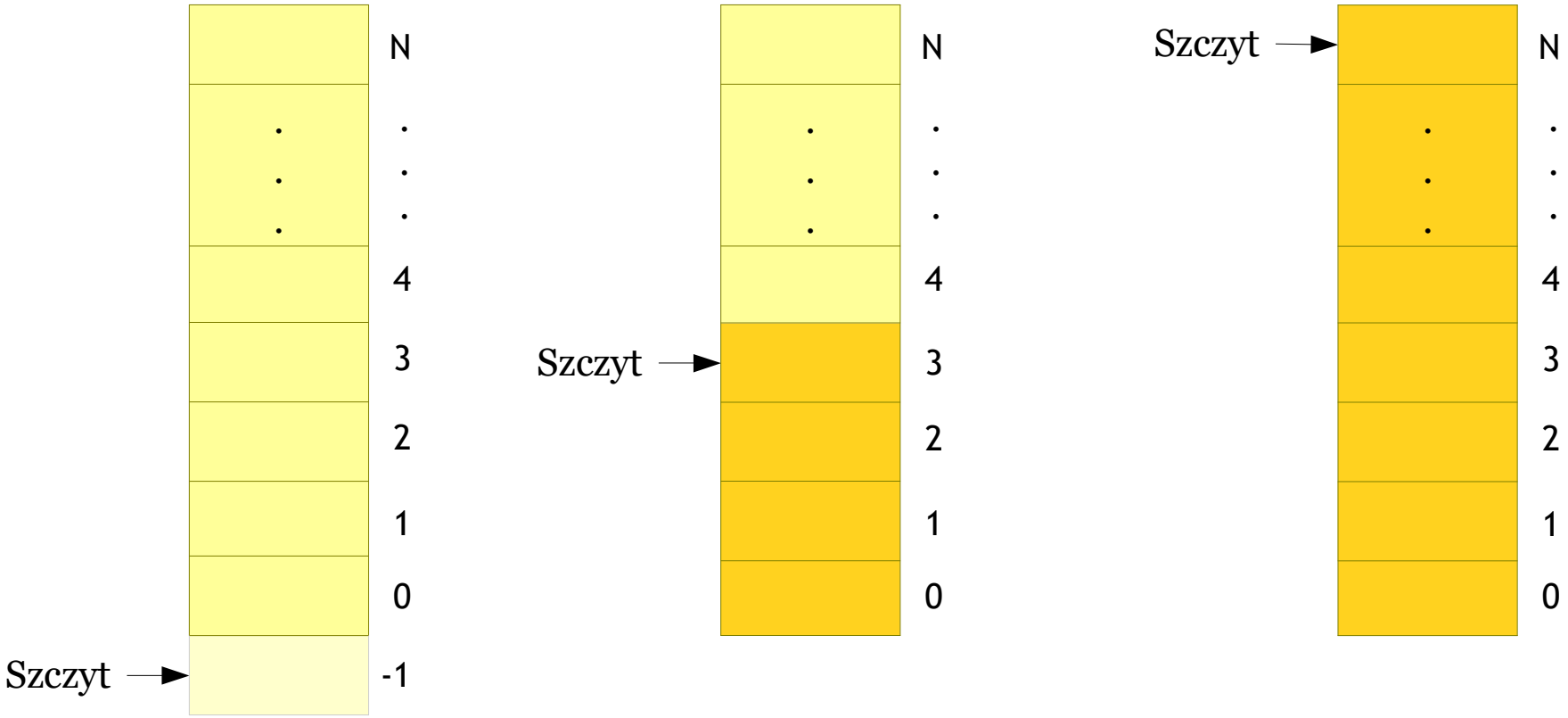


Pobierz kopię elementu ze szczytu nie zdejmując go ze stosu.
Inaczej: sprawdź co jest na szczycie stosu

Stos, pożądany interfejs

```
class Stack
{
  public:
    // Najlepiej rozrastał się w miarę potrzeb
    Stack();

    // Ale czasem trzeba albo warto ustalić pożądany rozmiar
    Stack( int size );


    // Podstawowe operacje
    bool push( Type item );
    Type pop();
    Type peek();

    // Czy stos jest pusty?
    bool empty();

    // Ile aktualnie jest elementów na stosie?
    int size();

    // Czyszczenie stosu
    void clear();
};
```

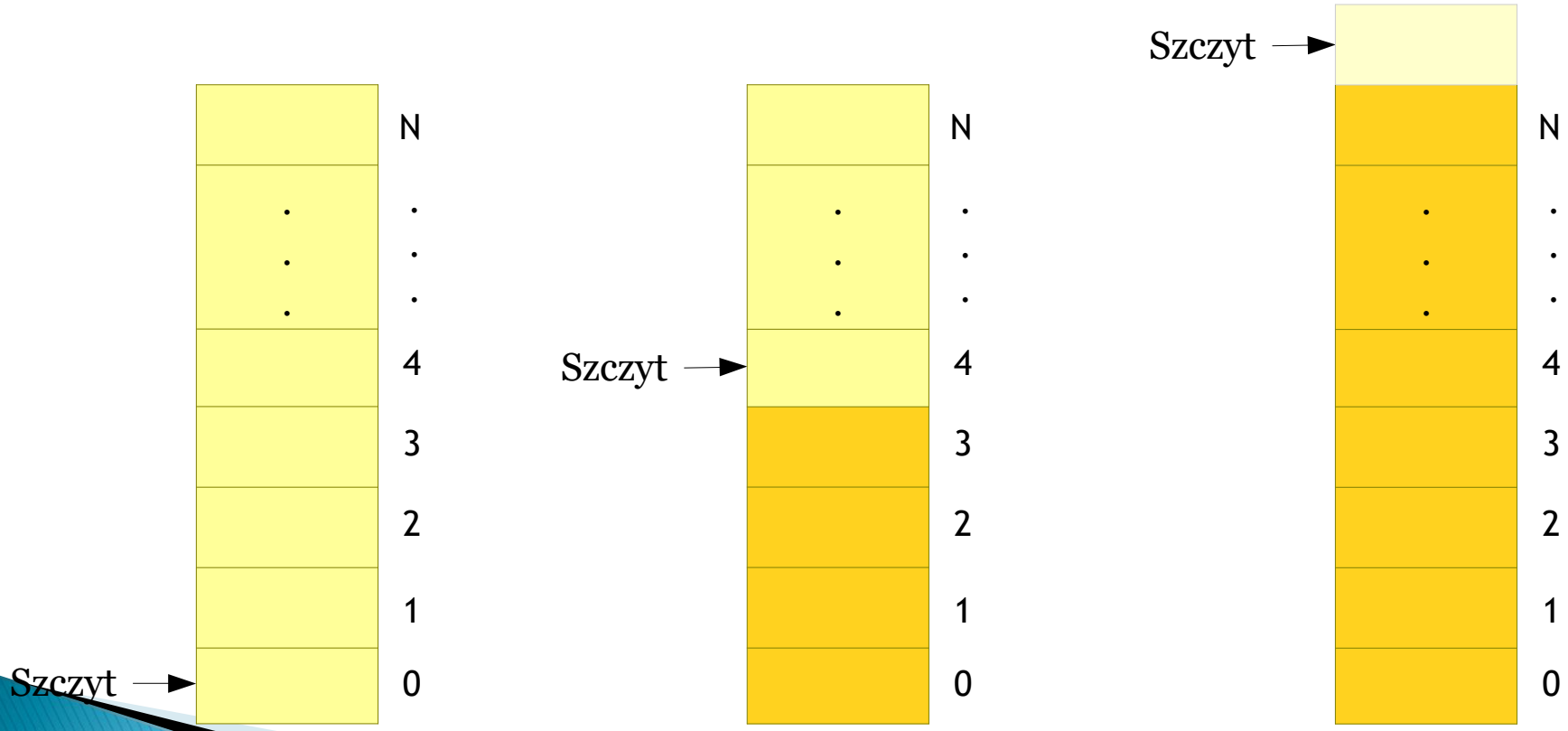
Stos: implementacja z wykorzystaniem tablicy



Rozmiar stosu jest ograniczony rozmiarem tablicy. Można jednak zaimplementować rozszerzanie rozmiaru stosu z wykorzystaniem dynamicznie tworzonych tablic.

Stos: implementacja z wykorzystaniem tablicy

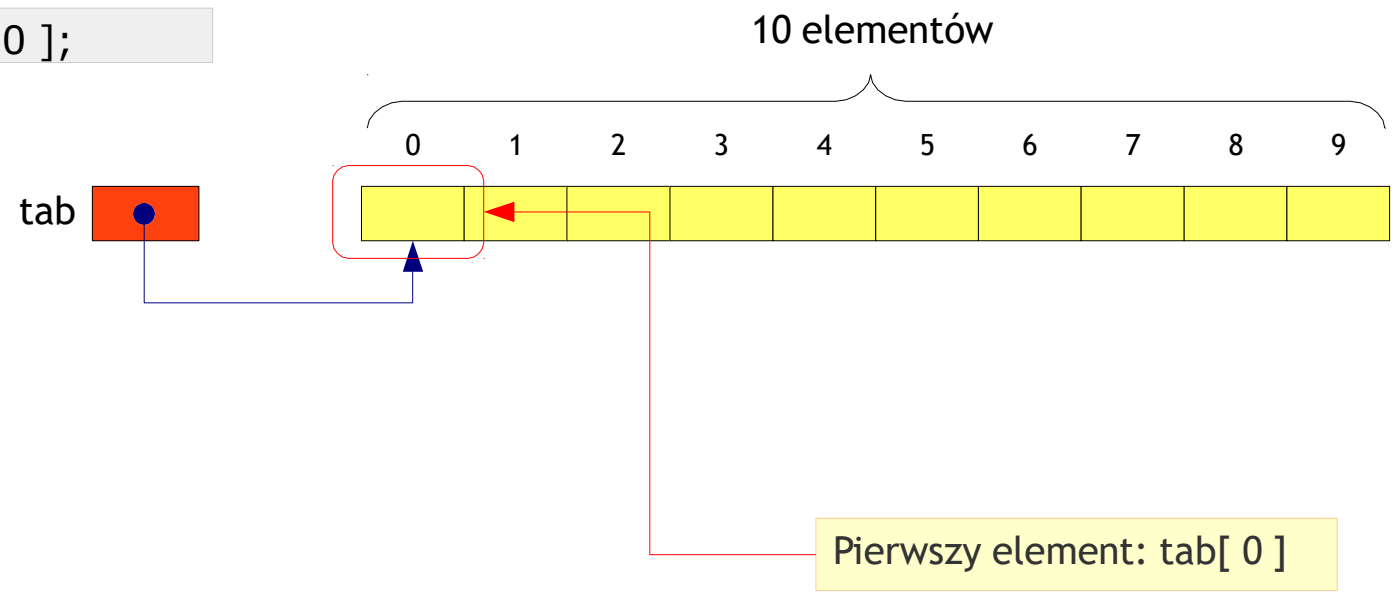
Dla wygody implementacji zakłada się czasem, że „szczyt” stosu identyfikuje pierwszy wolny element.



Nazwa tablicy jako wskaźnik na jej początek

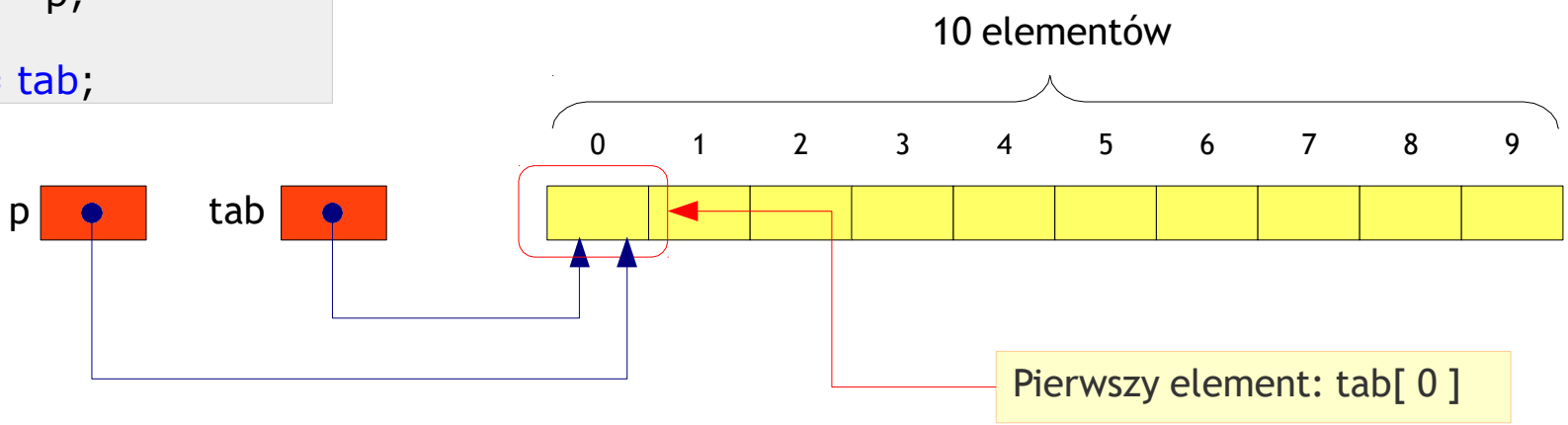
Nazwa tablicy jest interpretowana jako *ustalony wskaźnik* na jej początek (pierwszy element).

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



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

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



Przypisanie:

```
p = tab;
```

Jest równoznaczne z:

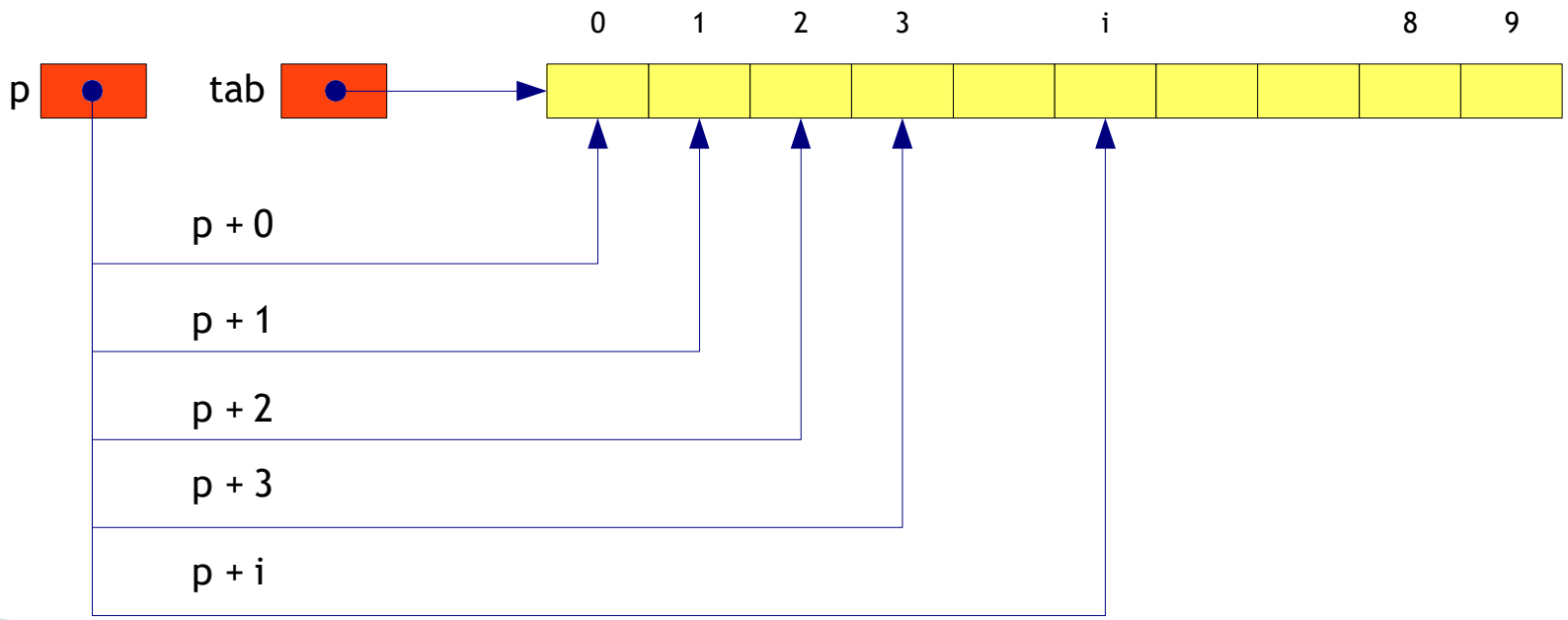
```
p = &tab[ 0 ];
```

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

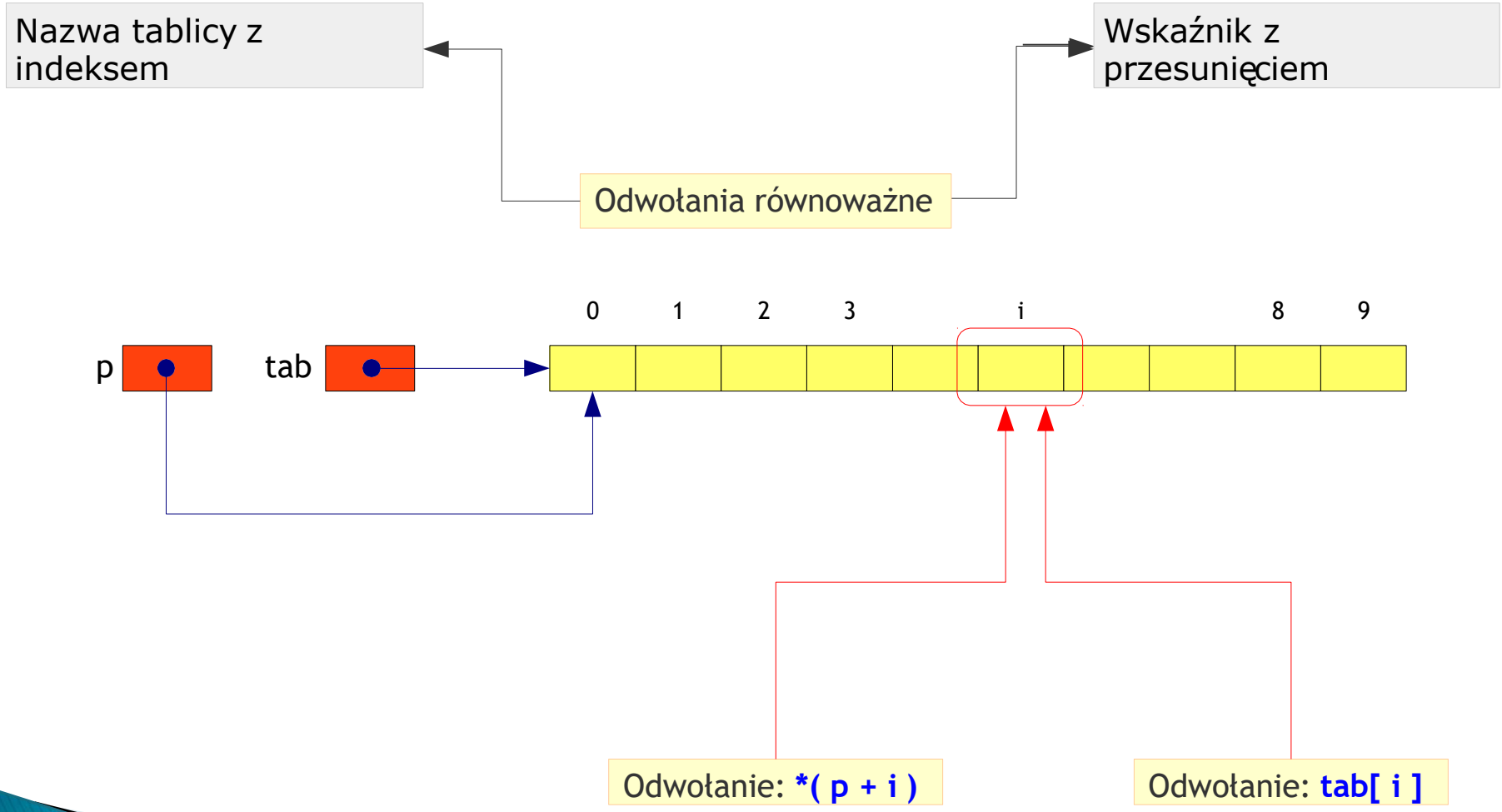
```
tab[ 0   ]  = 5;
tab[ 1   ]  = 1;
tab[ 2   ]  = 10;
. . .
tab[ i ] = 22;
```

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 )`

A także:

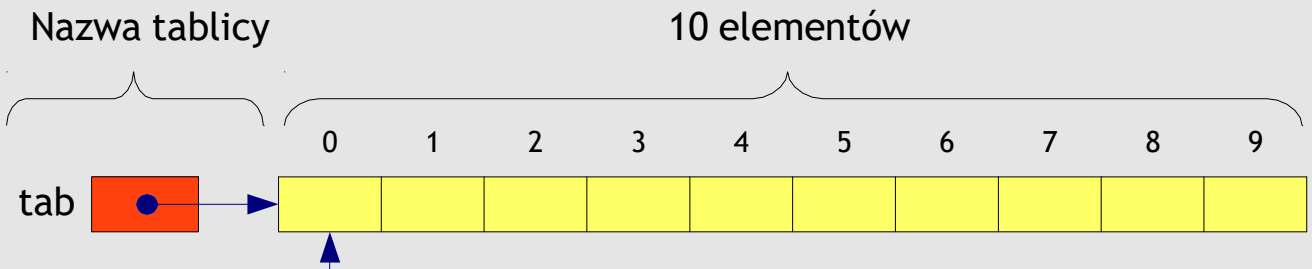
Każde odwołanie: `*( p + i )` można zapisać tak: → `p[ i ];`

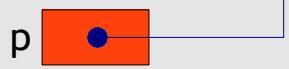
Uwaga, wskaźnik to nie tablica!

```
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

☑ OK

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.

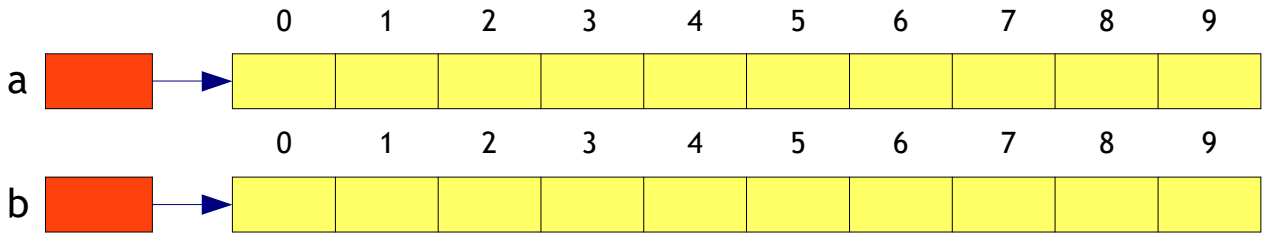
```
char napis[] = "język c";
. . .

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!
```

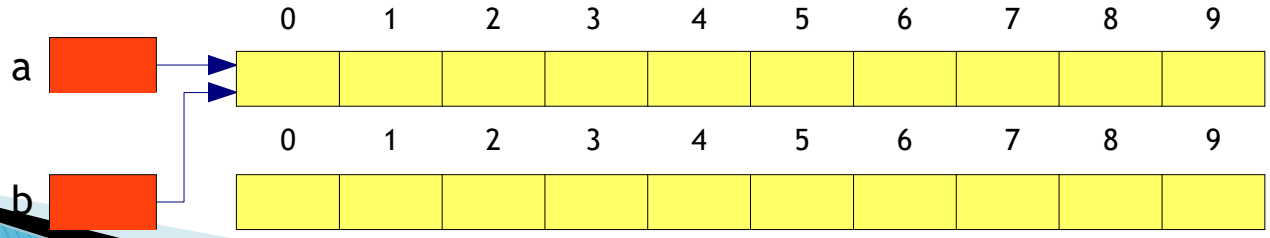
Gdyby przypisywanie było możliwe...



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 (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!

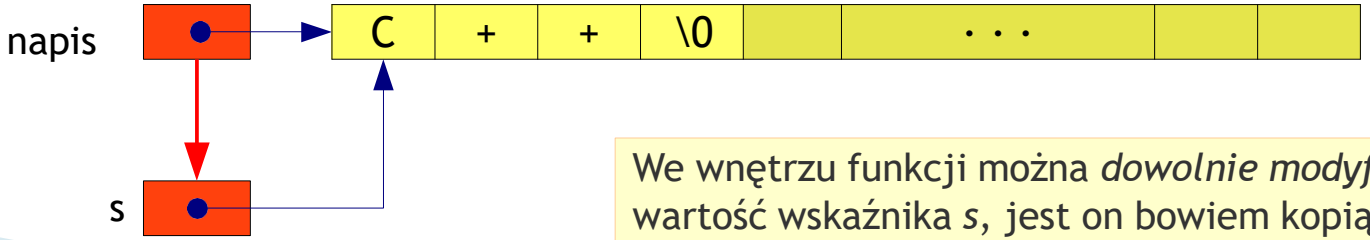
Paramterem *formalnym* funkcji jest **wskaźnik** a nie tablica!

```
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 );
```

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

wersja pierwotna

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

przechodzimy na wskaźniki

eliminujemy zmienną i

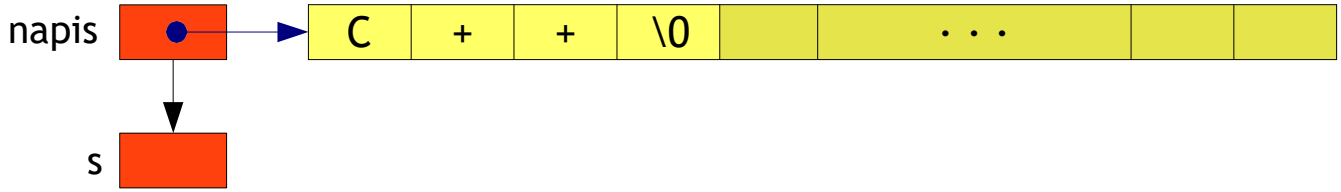
Jak to działa... ?

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

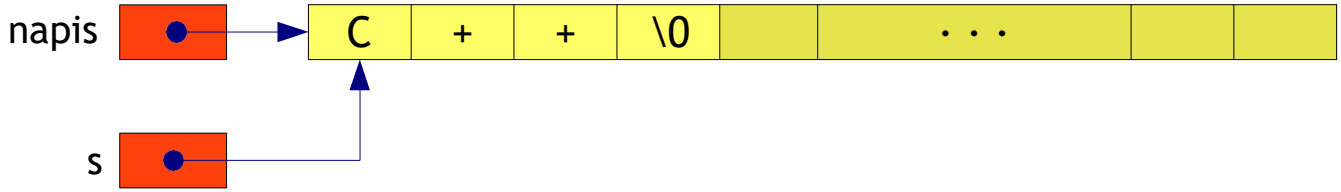


Parametr s jest kopią wskaźnika napis

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

putString( napis );
```

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

Kopiowanie parametru aktualnego napis do parametru s

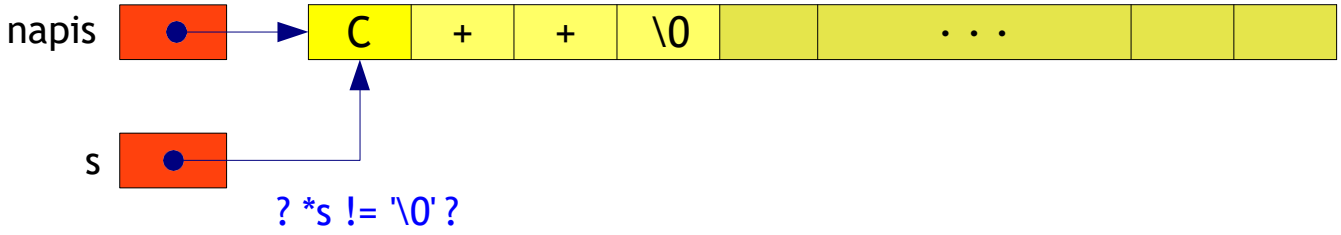


Czy obiekt wskazywany przez s jest znacznikiem końca napisu?

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

putString( napis );
```

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

Parametr s wskazuje na pierwszy element tablicy napis

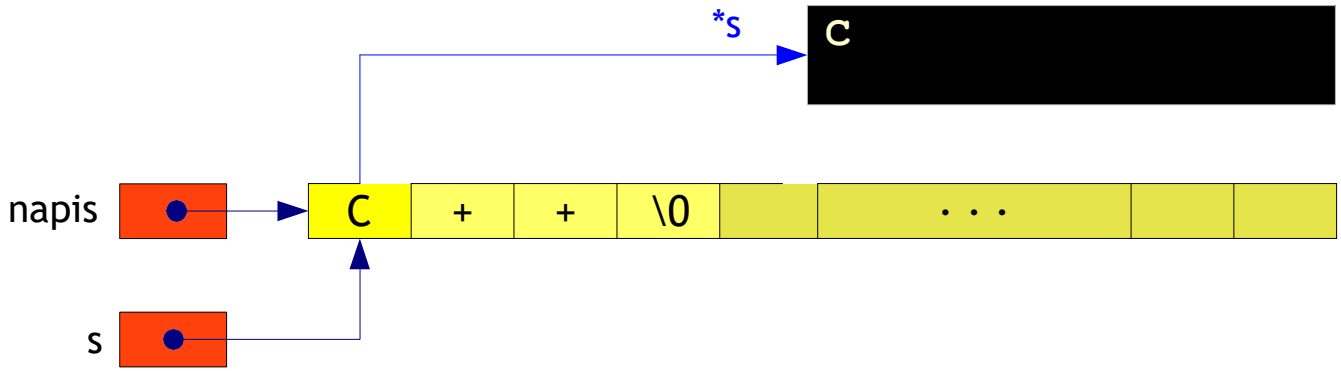


Znak wskazywany przez s wyprowadzamy jest do stdout

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

putString( napis );
```

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

Parametr s wskazuje na pierwszy element tablicy napis

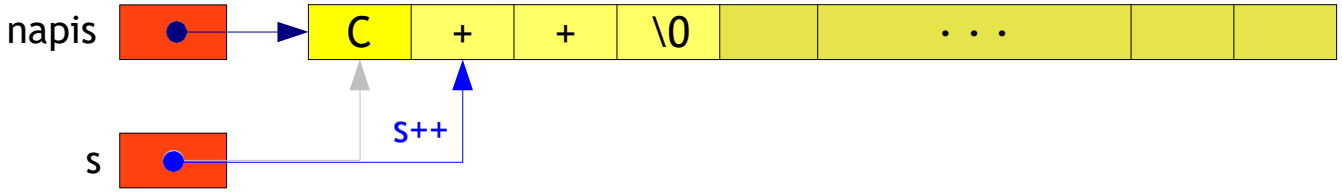


Wskaźnik s przesuwamy na następny znak

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

putString( napis );
```

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

Parametr s wskazuje na kolejny element tablicy napis

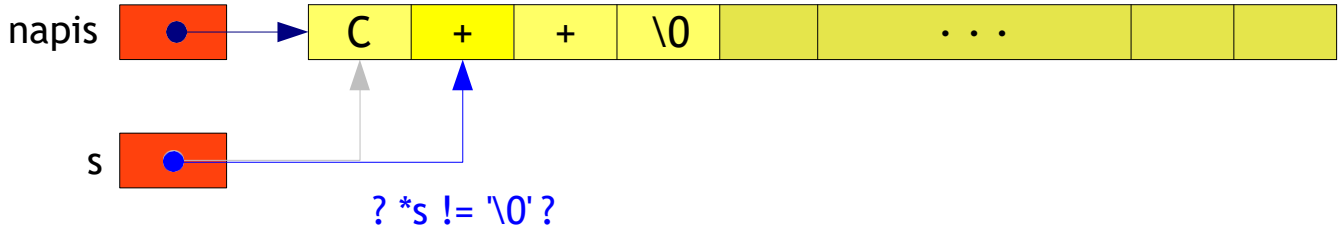


Czy obiekt wskazywany przez s jest znacznikiem końca napisu?

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

putString( napis );
```

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

Parametr s wskazuje na kolejny element tablicy napis

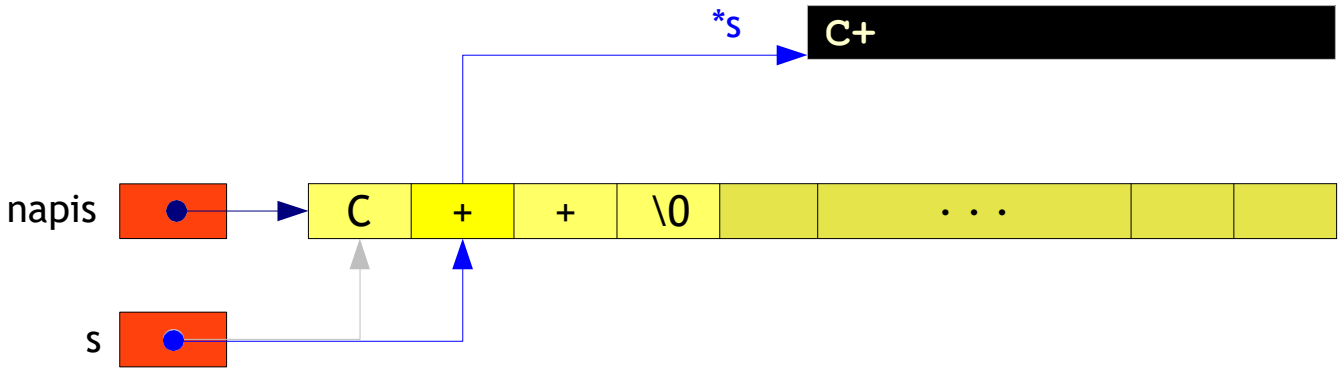


Znak wskazywany przez s wyprowadzamy jest do stdout

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

putString( napis );
```

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

Parametr s wskazuje na kolejny element tablicy napis

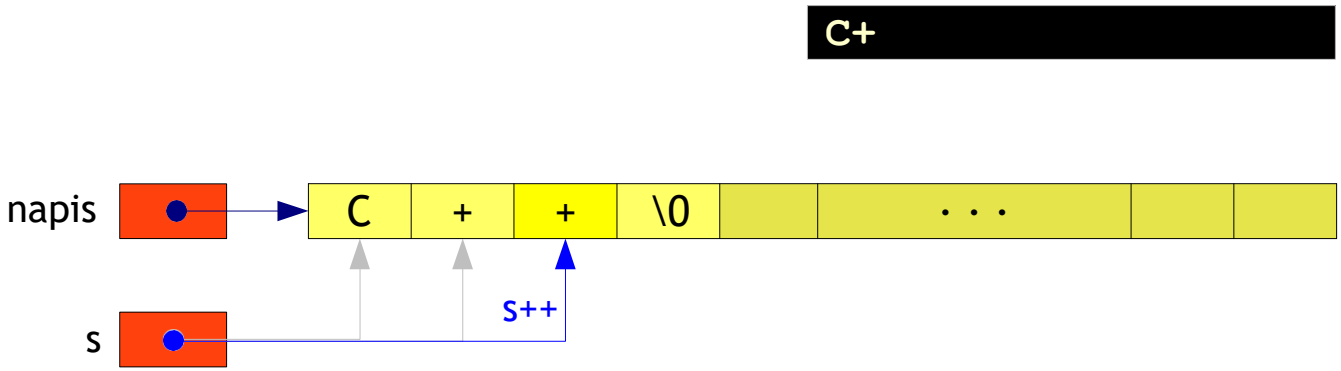


Wskaźnik s przesuwamy na następny znak

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

putString( napis );
```

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

Parametr s wskazuje na kolejny element tablicy napis

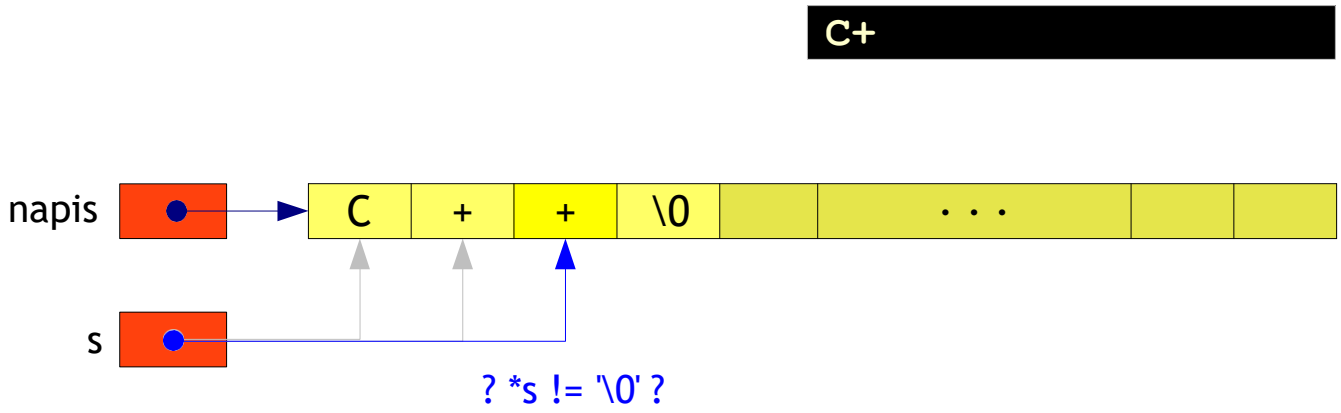


Czy obiekt wskazywany przez s jest znacznikiem końca napisu?

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

putString( napis );
```

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

Parametr s wskazuje na kolejny element tablicy napis

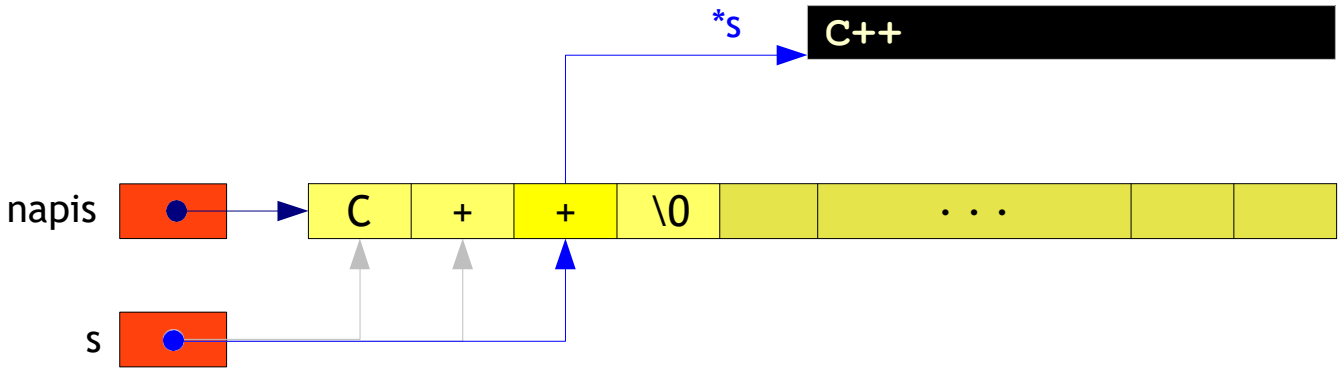


Znak wskazywany przez s wyprowadzamy jest do stdout

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

putString( napis );
```

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

Parametr s wskazuje na kolejny element tablicy napis

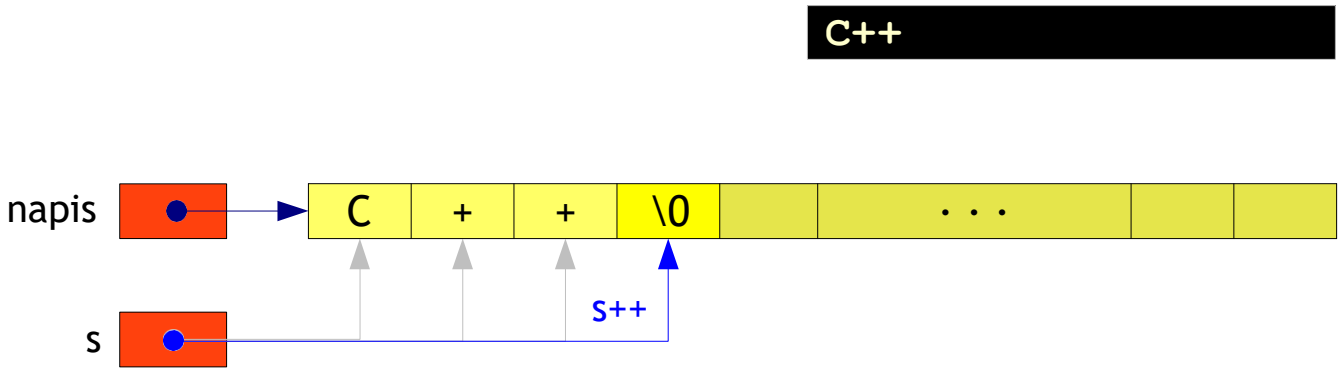


Wskaźnik s przesuwamy na następny znak

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

putString( napis );
```

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

Parametr s wskazuje na kolejny element tablicy napis

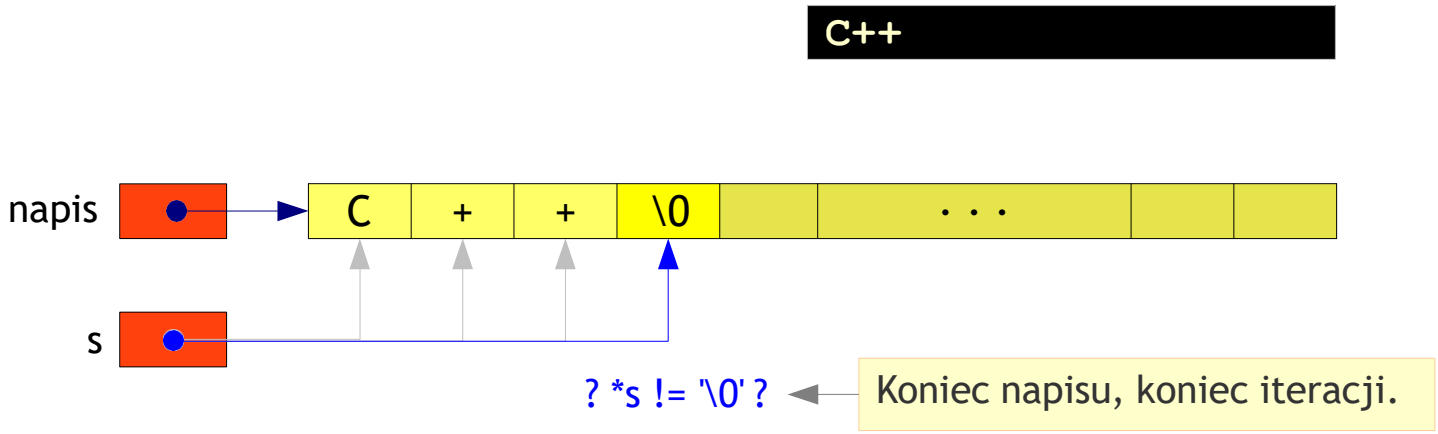


Czy obiekt wskazywany przez s jest znacznikiem końca napisu?

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

putString( napis );
```

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

Parametr s wskazuje na kolejny element tablicy napis

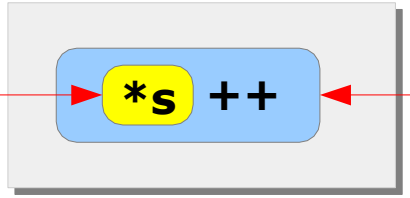


Wskaźniki w akcji — koniec metamorfozy

```
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

Stos: naiwna implementacja z wykorzystaniem tablicy o stałym rozmiarze

```
const int MAX_SIZE = 20;
class Stack
{
  public:
    Stack() : top( -1 ) {}

    bool push( int item )
    {
        if( top < MAX_SIZE - 1 )
        {
          items[ ++top ] = item;
          return true;
        }
        return false;
    }

    int pop()
    {
       if( empty() )
         return INT_MIN; // Co zrobić gdy stos jest pusty?
       return items[ top-- ];
    }

    ...
```

Zakładamy chwilowo, że rozważamy stos liczb całkowitych

Stos: naiwna implementacja z wykorzystaniem tablicy o stałym rozmiarze

```
    . . .

    int peek()
    {
       if( empty() )
         return INT_MIN; // Co zrobić gdy stos jest pusty?
       return items[ top ];
    }

    bool empty() const { return ( top <= -1 ); }

    int size() const { return top + 1; }

    void clear() { top = -1; }

  protected:
    int items[ MAX_SIZE ];
    int top;
};
```

Operacja pop – programistyczne rozterki...

```
. . .
int pop()
{
  if( empty() )
     return INT_MIN; // Nawet gdy stos jest pusty funkcja ma mieć rezultat
  return items[ top-- ];
}
. . .

Stack s;
s.push( 100 );

int number = s.pop();
```

Operacja pop – programistyczne rozterki...

```
. . .
void pop( int & item )
{
  if( !empty() )
     item = items[ top-- ];
}
. . .
```

```
Stack s;
s.push( 100 );

int number;
s.pop( number );
```

```
. . .
bool pop( int & item )
{
  if( empty() )
    return false;
  else
  {
     item = items[ top-- ];
     return true;
  }
}
. . .
```

```
Stack s;
s.push( 100 );

int number;
if( s.pop( number )
)
  . . .;
```

A może zastosować wyjątki...?

```
. . .
int pop() throw( int )
{
  if( empty() )
    throw STACK_UNDERFLOW;
   return items[ top-- ];
}
. . .
```

```
Stack s;
s.push( 100 );
try
{
  int number = s.pop();
  . . .
}
catch( ... )
{
  Reakcja na pusty stos
}
```

Mechanizm wyjątków w C++

- Wyjątek rozumiany jest w kontekście problemu, który pojawia się podczas wykonywania programu.
- Wyjątek C++ jest odpowiedzią na wyjątkową okoliczność, która pojawia się podczas działania programu, na przykład próba dzielenia przez zero.
- Wyjątki umożliwiają przeniesienie kontroli z jednej części programu do drugiej.
- Obsługa wyjątków w C++ opiera się na trzech słowach kluczowych: **try**, **catch** i **throw**.

Obsługa wyjątków

```
try
{
   // potencjalnie problematyczny fragment kodu
}
catch( ExceptionName e1 )
{
  // obsługa wyjątku e1
}
catch( ExceptionName e2 )
{
   // obsługa wyjątku e2
}
catch( ExceptionName eN )
{
  // obsługa wyjątku eN
}
```

Zgłaszanie wyjątków

```
//Zgłoszenie wyjątku:
throw <wyrażenie>
```

```
double dzielenie(int a, int b)
{
    if( b == 0 )
    {
        throw "Nastapilo dzielenie przez zero";
    }

   return (a/b);
}
```

Przykładowy program

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

double division(int a, int b) {
  if( b == 0 ) {
      throw "Nastapilo dzielenie przez zero";
  }
  return (a/b);
}

int main () {
 int x = 50;
 int y = 0;
 double z = 0;
 try {
   z = division(x, y);
   cout << z << endl;
 }
 catch (const char* msg) {
    cerr << msg << endl;
 }
 return 0;
}
```

Typy wyjątków

- C++ dostarcza listę standardowych wyjątków zdefiniowanych w <**exception**>, których możemy używać w naszych programach
- .
- Są one ułożone w hierarchii klas **rodzic-dziecko.**

- logic_error
  - invalid_argument
  - domain_error
  - length_error
  - out_of_range
  - future_error(C++11)
- bad_optional_access(C++17)
- runtime_error
  - range_error
  - overflow_error
  - underflow_error
  - regex_error(C++11)
  - system_error(C++11)
    - ios_base::failure(C++11)
    - filesystem::filesystem_error(C++17)
  - tx_exception(TM TS)
  - nonexistent_local_time(C++20)
  - ambiguous_local_time(C++20)
  - format_error(C++20)
- bad_typeid
- bad_cast
  - bad_any_cast(C++17)
- bad_weak_ptr(C++11)
- bad_function_call(C++11)
- bad_alloc
  - bad_array_new_length(C++11)
- bad_exception

Źródło:
https://en.cppreference.com/w/cpp/error/exception

Definicja własnych wyjątków

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

class MyException : public exception {
  const char * what () const throw ()
 {
     return "Wystapil Wyjatek";
  }
};

int main() {
  try
  {
       throw MyException();
   }
   catch(MyException& e)
  {
       cout << "Wyjatek zlapany" << endl;
       cout << e.what() << endl;
   }
   catch(std::exception& e)
   {
       //Inne wyjatki
    }
}
```

Najprostsza implementacja z wykorzystaniem wyjątków

```
const  int STACK_OVERFLOW    = 0;
const  int STACK_UNDERFLOW   = 1;
class  Stack
{
    . . .
    bool push( int item )   throw( int  )
    {
        if( top == MAX_SIZE - 1 )
            throw STACK_OVERFLOW;
        items[ ++top ] = item;
          return true;
    }

    int pop() throw( int )
    {
       if( empty() )
         throw STACK_UNDERFLOW;
       return items[ top-- ];
    }

    int peek() throw( int )
    {
       if( empty() )
         throw STACK_UNDERFLOW;
       return items[ top ];
    }
```

Obsługa stosu z wykorzystaniem wyjątków

```
try
{
  int number;
  s.push(  1 );
  s.push(  2 );
  s.push(  3 );

   for( ; ; )
     Number =s.pop();
}
catch( int & exceptionId )
{
  switch( exceptionId )
  {
    case STACK_OVERFLOW  : Reakcja na przepełnienie
                           break;
    case STACK_UNDERFLOW : Reakcja na pusty stos
                           break;
  }
}
```

Tablicę przechowującą elementy stosu można tworzyć dynamicznie

```
const int STACK_OVERFLOW  = 0;
const int STACK_UNDERFLOW = 1;
const int STACK_NOMEMORY  = 2;
const int MAX_SIZE = 20;
class Stack
{
  public:
  Stack( int size = MAX_SIZE ) : top( -1 ), items( 0 )
  {
      if( size > 0 ) try
     {
       items = new int[ length = size ];
     }
     catch( ... )
     {
       throw STACK_NOMEMORY;
     }
  }

  ~Stack()
  {
    clear();
  }
  . . .
```

Tablicę przechowującą elementy stosu można tworzyć dynamicznie

```
bool push( int item ) throw( int )
{
  if( items == 0 || top >= length - 1 )
    throw STACK_OVERFLOW;
  items[ ++top ] = item;
  return true;
}

int pop() throw( int )
{
   if( empty() )
     throw STACK_UNDERFLOW;
   return items[ top-- ];
}

int peek() throw( int )
{
   if( empty() )
     throw STACK_UNDERFLOW;
   return items[ top ];
}

bool empty() const
{
  return ( items == 0 || top <= -1 );
}
```

Tablicę przechowującą elementy stosu można tworzyć dynamicznie

```
    int size() const
    {
      return top + 1;
    }

    void clear()
    {
      if( items  )
        delete [] items;
      items = 0;
      length = 0;
    }

  protected:
    int * items;
    int  top;
    int length;
};
```

Mechanizm szablonów

- Co zrobić jeśli stos miałby służyć do przechowywania danych różnych typów?
- Szablony umożliwiają tworzenie generycznych klas (lub funkcji).
- Syntaktyka mechanizmu szablonów:

```
template <class Type>
template <typename Type>
```

Mechanizm szablonów – zasada działania

```
template <typename T>
T myMax(T x, T y)
{
    return (x > y)? x: y;
}

int main()
{
   cout << myMax<int>(3, 7) << endl;
   cout << myMax<char>('g', 'e') << endl;
   return 0;
}
```

Compiler internally generates and adds below code

```
int myMax(int x, int y)
{
    return (x > y)? x: y;
}
```

Compiler internally generates and adds below code.

```
char myMax(char x, char y)
{
    return (x > y)? x: y;
}
```

Źródło: https://www.geeksforgeeks.org/templates-cpp/

Stos z wykorzystaniem szablonów

```
template <class T>
class Stack
{
  public:
    Stack( int size = MAX_SIZE ) : top( -1 ), items( 0 )
    {
      if( size > 0 )
        try
        {
          items = new T[ length = size ];
        }
        catch( ... )
        {
          throw STACK_NOMEMORY;
        }
    }

    ~Stack()
    {
      clear();
    }
```

Stos z wykorzystaniem szablonów

```
bool push( T item ) throw( int )
{
  if( items == 0 || top >= length - 1 )
    throw STACK_OVERFLOW;
  items[ ++top ] = item;
  return true;
}

T pop() throw( int )
{
   if( empty() )
     throw STACK_UNDERFLOW;
   return items[ top-- ];
}

T peek() throw( int )
{
   if( empty() )
     throw STACK_UNDERFLOW;
   return items[ top ];
}
```

Stos z wykorzystaniem szablonów

```
    bool empty() const
    {
      return ( items == 0 || top <= -1 );
    }

    int size() const
    {
      return length;
    }

    void clear()
    {
      if( items  )
        delete [] items;
      items = 0;
      length = 0;
    }

  protected:
    T * items;
    int top;
    int length;
};
```

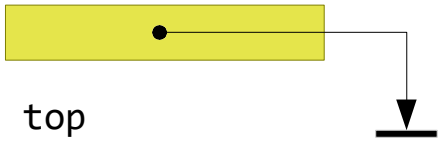
Dla tak zrealizowanego stosu można dodać operację powiększenia rozmiaru tablicy *items* gdy stos zostanie zapełniony.

Konkretyzowanie stosów

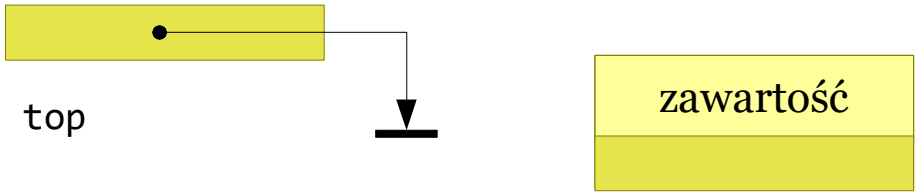
```
struct Point
{
  int x, y;
};

Stack<int> s1;
Stack<double> s2;
Stack<int> s3;
Stack<Point> s4;
Stack<char *> s5;
```

Uwaga, w języku C++ trzeba pamiętac o zdefiniowaniu konstruktora kopiującego jeżeli chcemy pamiętac na stosie obiekty alokujące indywidualne zasoby.

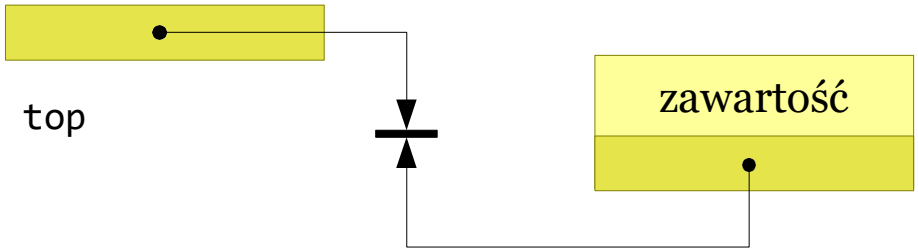
Stos jako lista – pusty stos



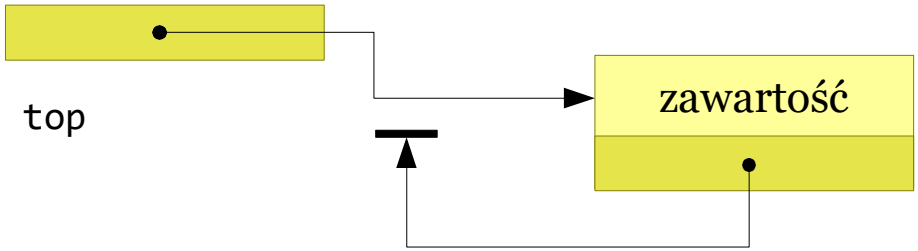
Stos jako lista – dodanie elementu, etap pierwszy



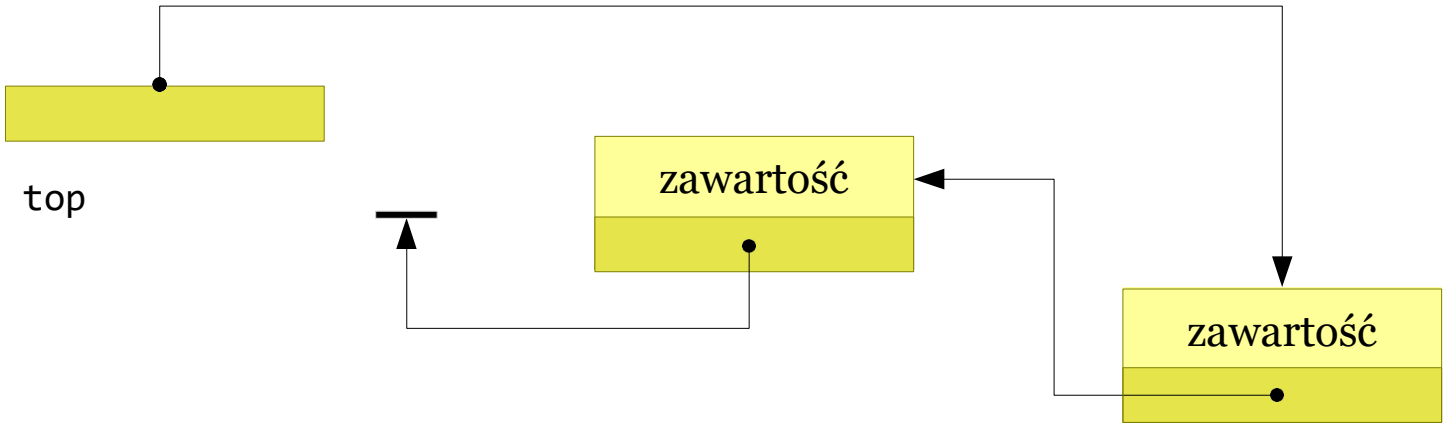
Stos jako lista – dodanie elementu, etap drugi



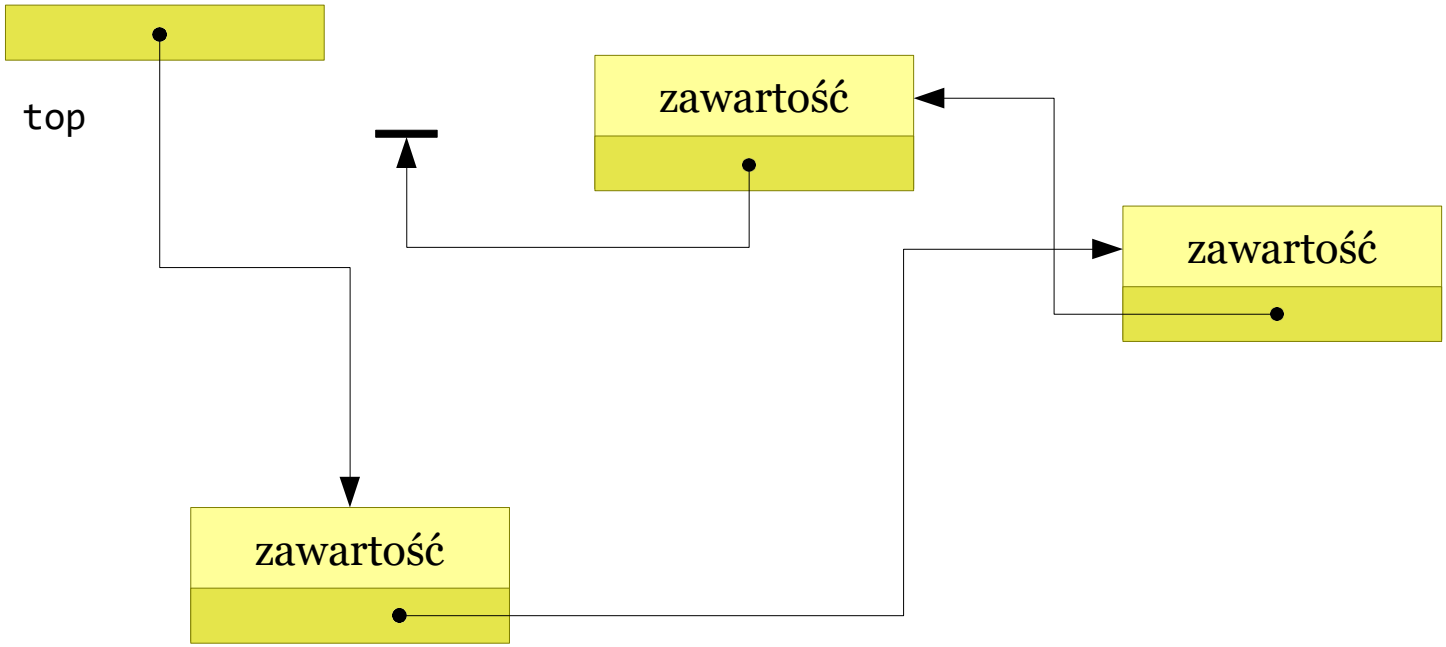
Stos jako lista – dodanie elementu, etap trzeci



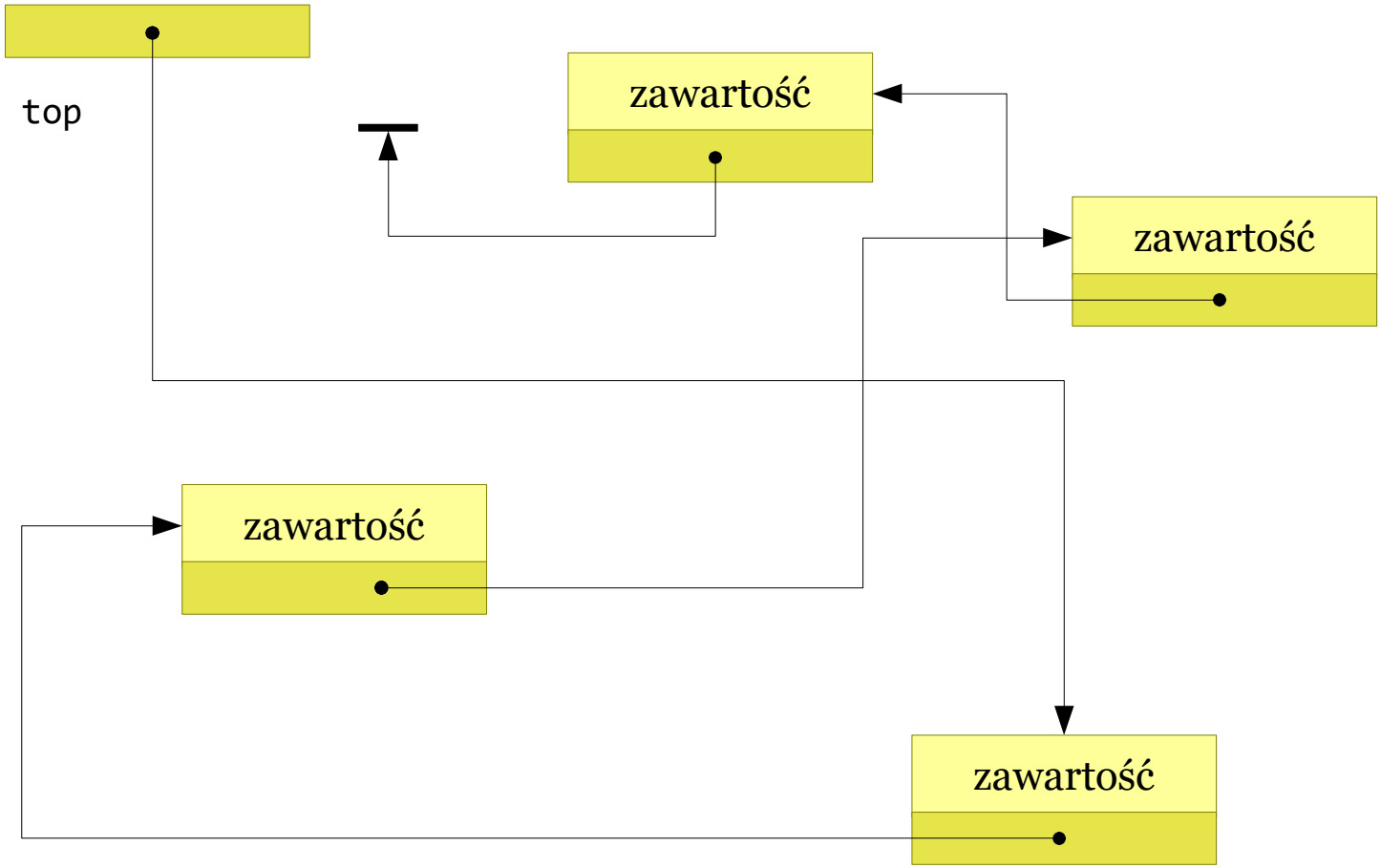
Stos jako lista – kolejny element



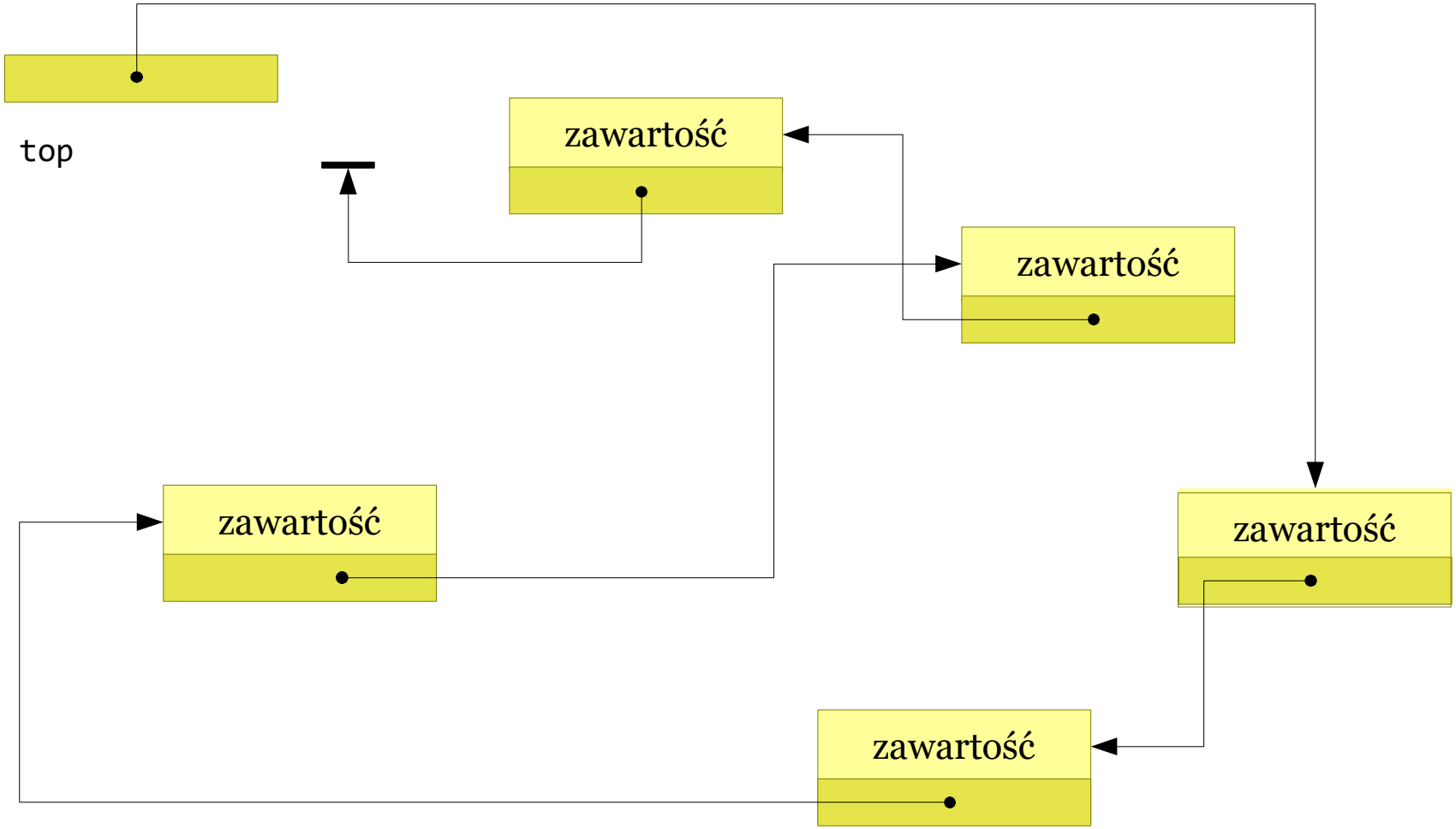
Stos jako lista – kolejny element



Stos jako lista – kolejny element

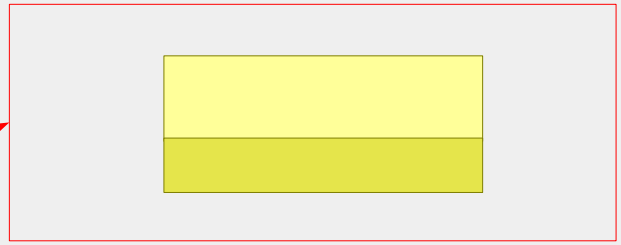


Stos jako lista – kolejny element



Należy zdefiniować klasę (strukturę) elementu listy

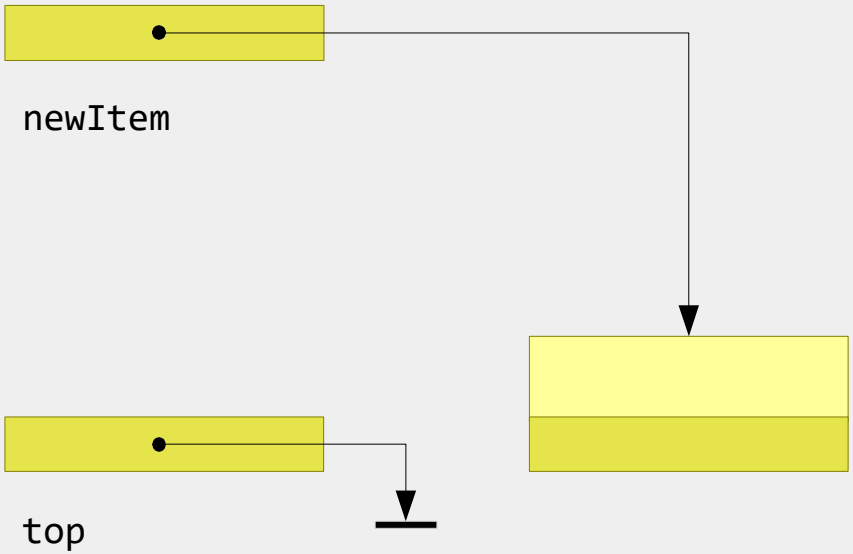
```
template <typename T>
class Stack
{
  public:
    Stack() : top( 0 ), length( 0 )
    {
    }

    ~Stack()
    {
      clear();
    }

    . . .
  protected:

    struct StackItem
    {
      T item;
      struct StackItem * next;
    };

    StackItem * top;
    int length;
};
```



Pole wskaźnikowe identyfikujące szczyt stosu

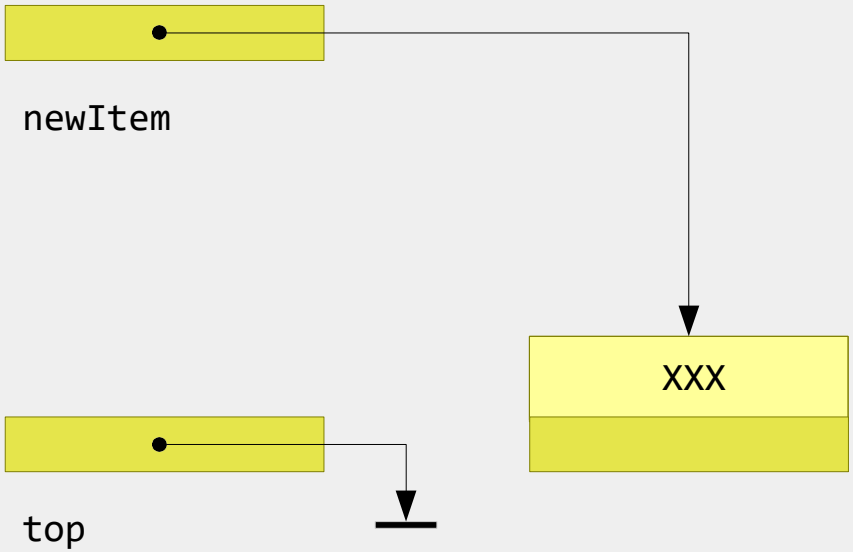
Dodawanie pierwszego elementu do stosu

```
bool push( T item ) throw( int )
{
  try
  {
    StackItem * newItem = new StackItem;

    newItem->item = item;
    newItem->next = top;
    top = newItem;
    length++;
    return true;
  }
  catch( ... )
  {
    throw STACK_NOMEMORY;
  }
}
```



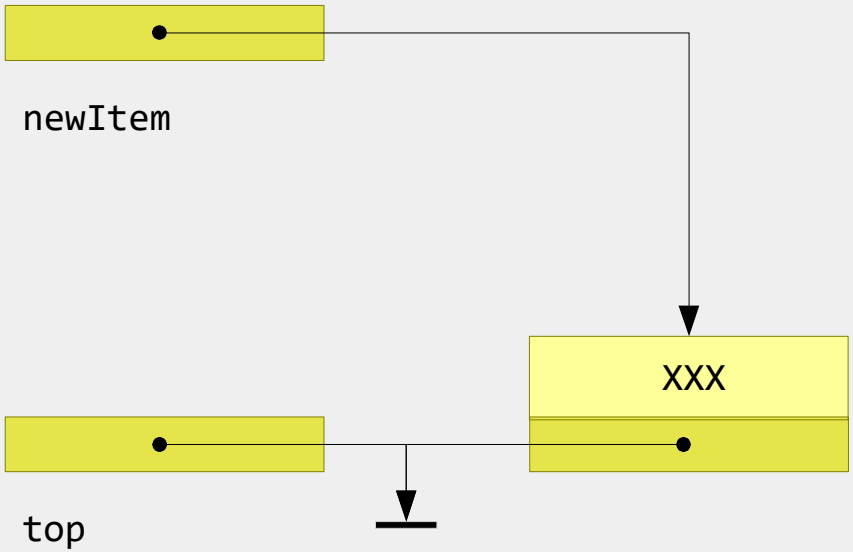
Dodawanie pierwszego elementu do stosu

```
bool push( T item ) throw( int )
{
  try
  {
    StackItem * newItem = new StackItem;

    newItem->item = item;
    newItem->next = top;
    top = newItem;
    length++;
    return true;
  }
  catch( ... )
  {
    throw STACK_NOMEMORY;
  }
}
```



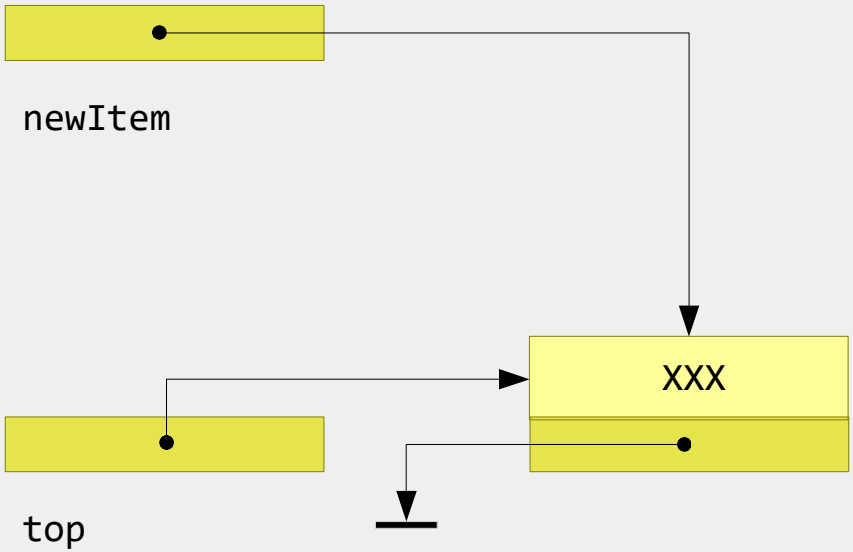
Dodawanie pierwszego elementu do stosu

```
bool push( T item ) throw( int )
{
  try
  {
    StackItem * newItem = new StackItem;

    newItem->item = item;
    newItem->next = top;
    top = newItem;
    length++;
    return true;
  }
  catch( ... )
  {
    throw STACK_NOMEMORY;
  }
}
```



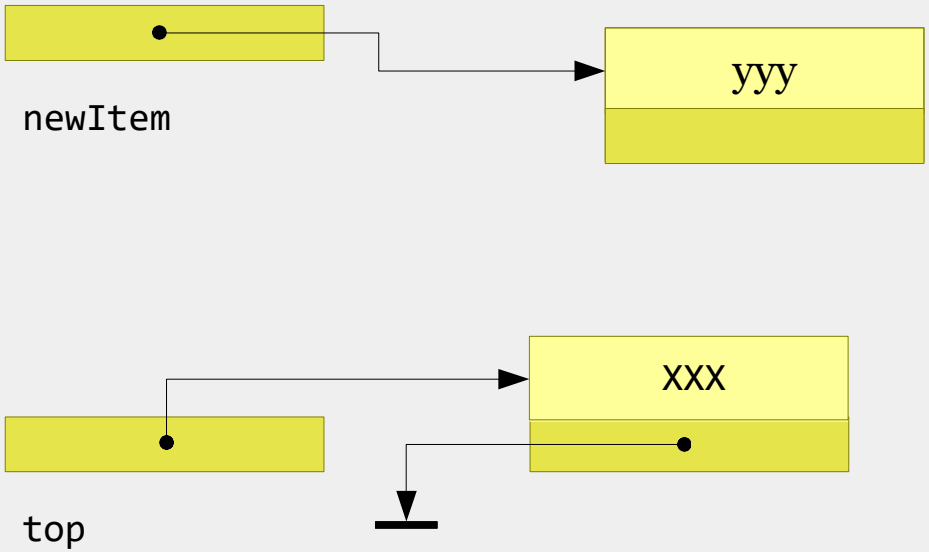
Dodawanie pierwszego elementu do stosu

```
bool push( T item ) throw( int )
{
  try
  {
    StackItem * newItem = new StackItem;

    newItem->item = item;
    newItem->next = top;
    top = newItem;
    length++;
    return true;
  }
  catch( ... )
  {
    throw STACK_NOMEMORY;
  }
}
```



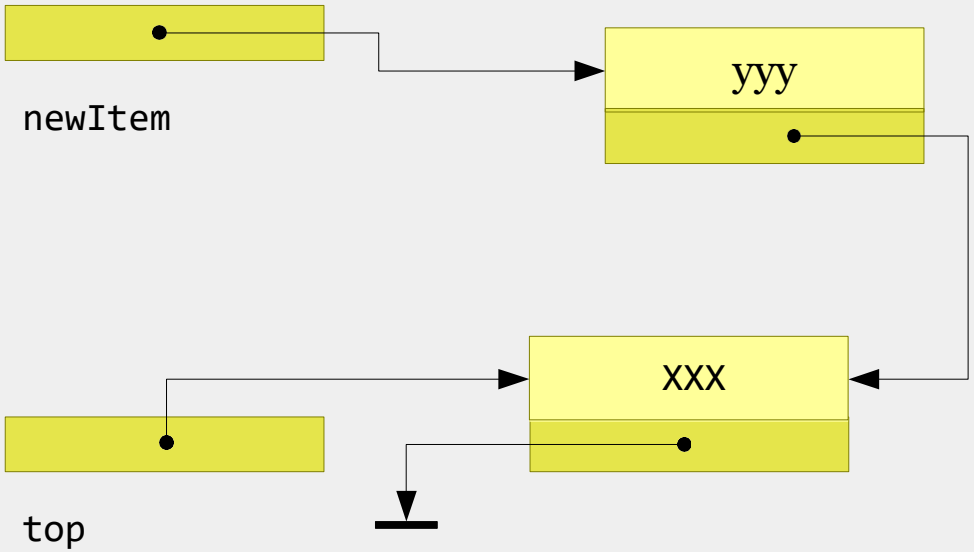
Dodawanie kolejnego elementu do stosu

```
bool push( T item ) throw( int )
{
  try
  {
    StackItem * newItem = new StackItem;

    newItem->item = item;
    newItem->next = top;
    top = newItem;
    length++;
    return true;
  }
  catch( ... )
  {
    throw STACK_NOMEMORY;
  }
}
```



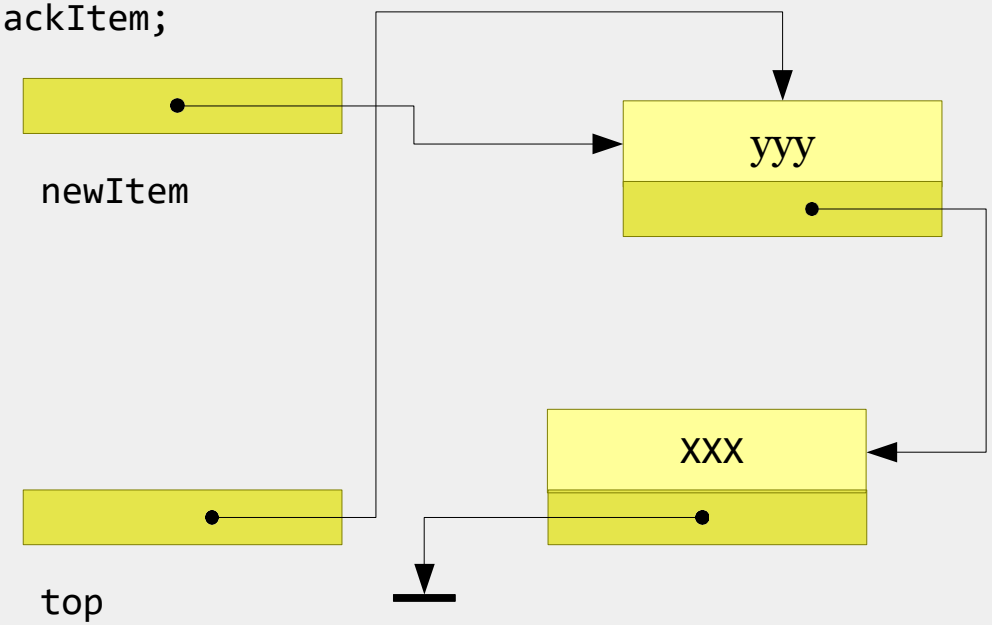
Dodawanie kolejnego elementu do stosu

```
bool push( T item ) throw( int )
{
  try
  {
    StackItem * newItem = new StackItem;

    newItem->item = item;
    newItem->next = top;
    top = newItem;
    length++;
    return true;
  }
  catch( ... )
  {
    throw STACK_NOMEMORY;
  }
}
```



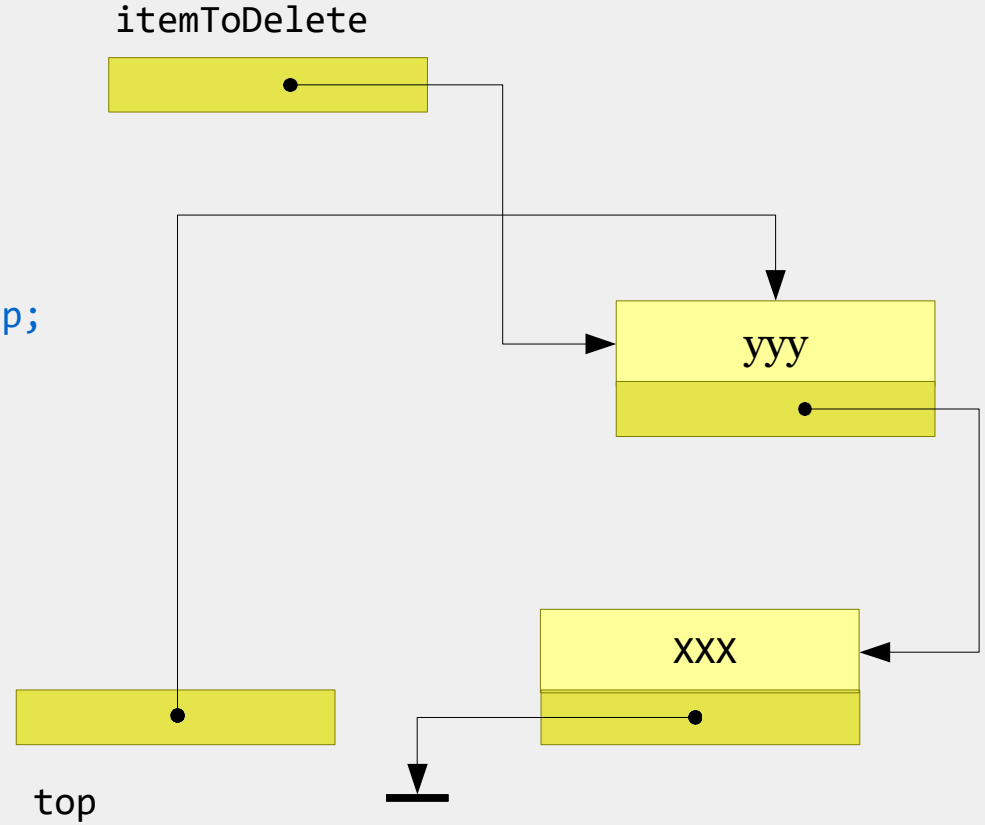
Dodawanie kolejnego elementu do stosu

```
bool push( T item ) throw( int )
{
  try
  {
    StackItem * newItem = new StackItem;

    newItem->item = item;
    newItem->next = top;
    top = newItem;
    length++;
    return true;
  }
  catch( ... )
  {
    throw STACK_NOMEMORY;
  }
}
```



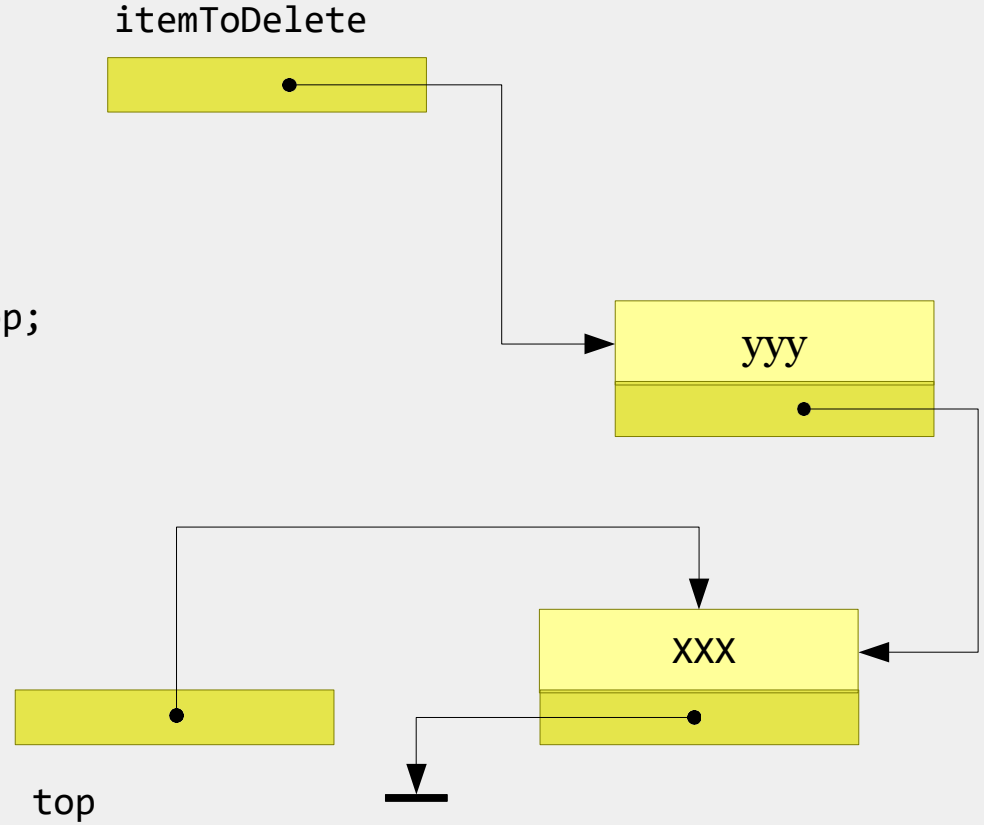
Usuwanie elementu ze stosu

```
T pop() throw( int )
{
   if( empty() )
     throw STACK_UNDERFLOW;

   T returnValue = top->item;
   StackItem * itemToDelete = top;
   top = top->next;
   delete itemToDelete;
   length--;
   return returnValue;
}
```



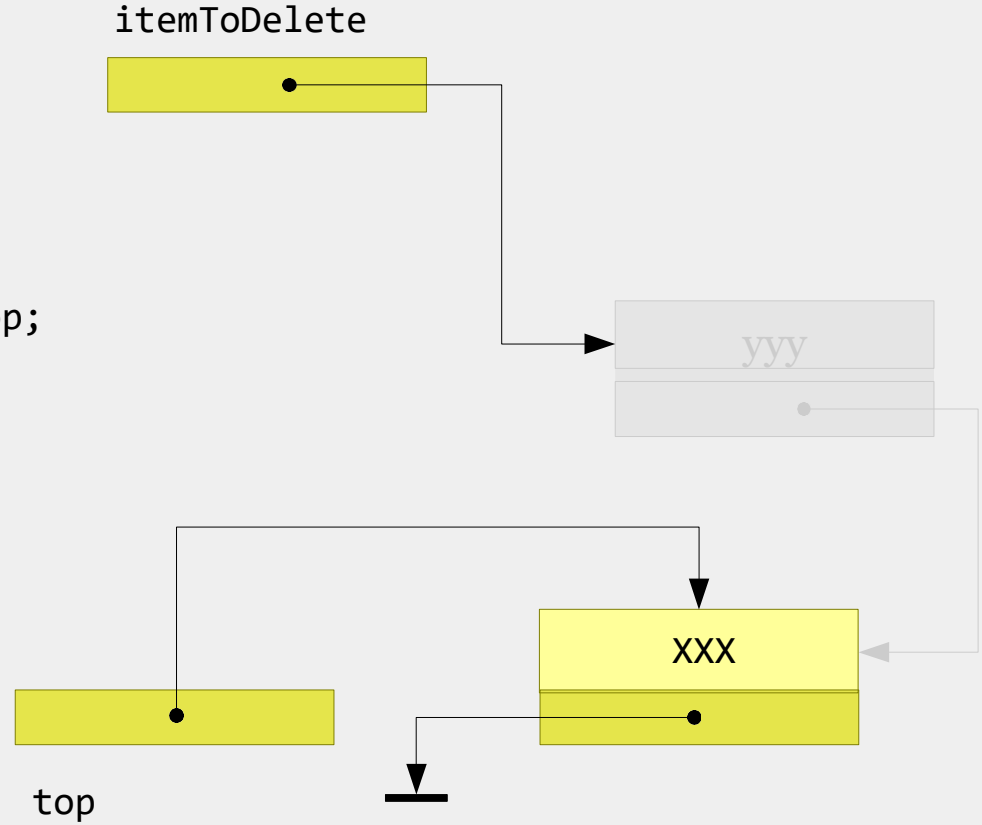
Usuwanie elementu ze stosu

```
T pop() throw( int )
{
   if( empty() )
     throw STACK_UNDERFLOW;

   T returnValue = top->item;
   StackItem * itemToDelete = top;
   top = top->next;
   delete itemToDelete;
   length--;
   return returnValue;
}
```



Usuwanie elementu ze stosu

```
T pop() throw( int )
{
   if( empty() )
     throw STACK_UNDERFLOW;

   T returnValue = top->item;
   StackItem * itemToDelete = top;
   top = top->next;
   delete itemToDelete;
   length--;
   return returnValue;
}
```



Pozostałe operacje

```
T peek() throw( int )
{
   if( empty() )
     throw STACK_UNDERFLOW;
   return top->item;
}


bool empty() const
{
  return ( top == 0 );
}

int size() const
{
  return length;
}

void clear()
{
  while( !empty() )
    pop();
}
```