Od funkcji do metody, od struktury do klasy

Bogdan Kreczmer

bogdan.kreczmer@pwr.wroc.pl

Zakład Podstaw Cybernetyki i Robotyki
Instytut Informatyki, Automatyki i Robotyki
Politechnika Wrocławska

Kurs: Programowanie obiektowe

Niniejsza prezentacja została wykonana przy użyciu systemu składu LaTeX *oraz stylu* `beamer`, *którego autorem jest Till Tantau.*

Strona domowa projektu `Beamer`:
`http://latex-beamer.sourceforge.net`

Plan prezentacji

Plan prezentacji

Znaczenie modyfikatora const

Posługując się typem wskaźnikowym możliwe jest zdefiniowanie:

- zmiennej wskaźnikowej na modyfikowalny obszar pamięci.

Modyfikator **const** w połączeniu z typem wskaźnikowym pozwala na zdefiniowanie:

- zmiennej wskaźnikowej na *niemodyfikowalny* obszar pamięci,
- wskaźnika *stałego* na *modyfikowalny* obszar pamięci,
- wskaźnika *stałego* na *niemodyfikowalny* obszar pamięci.

Znaczenie modyfikatora const

Posługując się typem wskaźnikowym możliwe jest zdefiniowanie:

- zmiennej wskaźnikowej na modyfikowalny obszar pamięci.

Modyfikator **const** w połączeniu z typem wskaźnikowym pozwala na zdefiniowanie:

- zmiennej wskaźnikowej na *niemodyfikowalny* obszar pamięci,
- wskaźnika *stałego* na *modyfikowalny* obszar pamięci,
- wskaźnika *stałego* na *niemodyfikowalny* obszar pamięci.

Znaczenie modyfikatora const

Posługując się typem wskaźnikowym możliwe jest zdefiniowanie:

- zmiennej wskaźnikowej na modyfikowalny obszar pamięci.

Modyfikator **const** w połączeniu z typem wskaźnikowym pozwala na zdefiniowanie:

- zmiennej wskaźnikowej na *niemodyfikowalny* obszar pamięci,
- wskaźnika *stałego* na *modyfikowalny* obszar pamięci,
- wskaźnika *stałego* na *niemodyfikowalny* obszar pamięci.

Znaczenie modyfikatora const

Posługując się typem wskaźnikowym możliwe jest zdefiniowanie:

- zmiennej wskaźnikowej na modyfikowalny obszar pamięci.

Modyfikator **const** w połączeniu z typem wskaźnikowym pozwala na zdefiniowanie:

- zmiennej wskaźnikowej na *niemodyfikowalny* obszar pamięci,
- wskaźnika *stałego* na *modyfikowalny* obszar pamięci,
- wskaźnika *stałego* na *niemodyfikowalny* obszar pamięci.

Znaczenie modyfikatora const

Posługując się typem wskaźnikowym możliwe jest zdefiniowanie:

- zmiennej wskaźnikowej na modyfikowalny obszar pamięci.

Modyfikator **const** w połączeniu z typem wskaźnikowym pozwala na zdefiniowanie:

- zmiennej wskaźnikowej na *niemodyfikowalny* obszar pamięci,
- wskaźnika *stałego* na *modyfikowalny* obszar pamięci,
- wskaźnika *stałego* na *niemodyfikowalny* obszar pamięci.

Znaczenie modyfikatora const

Posługując się typem wskaźnikowym możliwe jest zdefiniowanie:

- zmiennej wskaźnikowej na modyfikowalny obszar pamięci.

Modyfikator **const** w połączeniu z typem wskaźnikowym pozwala na zdefiniowanie:

- zmiennej wskaźnikowej na *niemodyfikowalny* obszar pamięci,
- wskaźnika *stałego* na *modyfikowalny* obszar pamięci,
- wskaźnika *stałego* na *niemodyfikowalny* obszar pamięci.

Przykłady definicji zmiennych wskaźnikowych

Posługując się typem wskaźnikowym możliwe jest zdefiniowanie:

- zmiennej wskaźnikowej na modyfikowalny obszar pamięci.

Modyfikator **const** w połączeniu z typem wskaźnikowym pozwala na zdefiniowanie:

- zmiennej wskaźnikowej na *niemodyfikowalny* obszar pamięci,
- wskaźnika *stałego* na *modyfikowalny* obszar pamięci,
- wskaźnika *stałego* na *niemodyfikowalny* obszar pamięci.

Przykłady definicji zmiennych wskaźnikowych

Posługując się typem wskaźnikowym możliwe jest zdefiniowanie:

$\longrightarrow$ **char*** wNapis;

Modyfikator **const** w połączeniu z typem wskaźnikowym pozwala na zdefiniowanie:

- zmiennej wskaźnikowej na *niemodyfikowalny* obszar pamięci,
- wskaźnika *stałego* na *modyfikowalny* obszar pamięci,
- wskaźnika *stałego* na *niemodyfikowalny* obszar pamięci.

Przykłady definicji zmiennych wskaźnikowych

Posługując się typem wskaźnikowym możliwe jest zdefiniowanie:

$\longrightarrow$ **char**$*$ wNapis;

Modyfikator **const** w połączeniu z typem wskaźnikowym pozwala na zdefiniowanie:

- zmiennej wskaźnikowej na *niemodyfikowalny* obszar pamięci,
- wskaźnika *stałego* na *modyfikowalny* obszar pamięci,
- wskaźnika *stałego* na *niemodyfikowalny* obszar pamięci.

Przykłady definicji zmiennych wskaźnikowych

Posługując się typem wskaźnikowym możliwe jest zdefiniowanie:

$\longrightarrow$ **char**$*$ wNapis;

Modyfikator **const** w połączeniu z typem wskaźnikowym pozwala na zdefiniowanie:

$\longrightarrow$ **const char**$*$ wNapis;

- wskaźnika *stałego* na *modyfikowalny* obszar pamięci,
- wskaźnika *stałego* na *niemodyfikowalny* obszar pamięci.

Przykłady definicji zmiennych wskaźnikowych

Posługując się typem wskaźnikowym możliwe jest zdefiniowanie:

$\longrightarrow$ **char**$*$ wNapis;

Modyfikator **const** w połączeniu z typem wskaźnikowym pozwala na zdefiniowanie:

$\longrightarrow$ **const char**$*$ wNapis;

- wskaźnika *stałego* na *modyfikowalny* obszar pamięci,
- wskaźnika *stałego* na *niemodyfikowalny* obszar pamięci.

Przykłady definicji zmiennych wskaźnikowych

Posługując się typem wskaźnikowym możliwe jest zdefiniowanie:

$\longrightarrow$ **char**$*$ wNapis;

Modyfikator **const** w połączeniu z typem wskaźnikowym pozwala na zdefiniowanie:

$\longrightarrow$ **const char**$*$ wNapis;

$\longrightarrow$ **char**$*$ **const** wNapis;

- wskaźnika *stałego* na *niemodyfikowalny* obszar pamięci.

Przykłady definicji zmiennych wskaźnikowych

Posługując się typem wskaźnikowym możliwe jest zdefiniowanie:

$\longrightarrow$ **char**$*$ wNapis;

Modyfikator **const** w połączeniu z typem wskaźnikowym pozwala na zdefiniowanie:

$\longrightarrow$ **const char**$*$ wNapis;

$\longrightarrow$ **char**$*$ **const** wNapis;

- wskaźnika *stałego* na *niemodyfikowalny* obszar pamięci.

Przykłady definicji zmiennych wskaźnikowych

Posługując się typem wskaźnikowym możliwe jest zdefiniowanie:

→ **char**∗ wNapis;

Modyfikator **const** w połączeniu z typem wskaźnikowym pozwala na zdefiniowanie:

→ **const char**∗ wNapis;

→ **char**∗ **const** wNapis;

→ **const char**∗ **const** wNapis;

Definicje zmiennych wskaźnikowych z **const**

```
char*             wNapis;
const char*       wNapis;
char* const       wNapis;
const char* const wNapis;
```

Wskaźnik na tekst, który **może** być modyfikowany.

Wskaźnik na modyfikowalny obszar pamięci

```
char  ObszarPam[ ] = "łódka";
char* wNapis = ObszarPam;
```

Wskaźnik na modyfikowalny obszar pamięci

```
char  ObszarPam[ ] = "łódka";
char* wNapis = ObszarPam;
wNapis[0] = 'w';
```

Wskaźnik na modyfikowalny obszar pamięci

```
char  ObszarPam[ ] = "łódka";
char* wNapis = ObszarPam;
wNapis[0] = 'w';
```

Wskaźnik na modyfikowalny obszar pamięci

```
char  ObszarPam[ ] = "łódka";
char* wNapis = ObszarPam;
wNapis[0] = 'w';
char* ObszarPam2[ ] = "udka";
wNapis = ObszarPam2;
```

Wskaźnik na modyfikowalny obszar pamięci

```
char  ObszarPam[ ] = "łódka";
char* wNapis = ObszarPam;
wNapis[0] = 'w';
char* ObszarPam2[ ] = "udka";
wNapis = ObszarPam2;
```

Podsumowanie:

- obszar pamięci dostępny poprzez wskaźnik może być modyfikowany (dotyczy to operacji z wykorzystaniem tej zmiennej wskaźnikowej),
- zawartość zmiennej wskaźnikowej może ulegać zmianie,
- zmienna wskaźnikowa nie musi być inicjalizowana w momencie jej definicji.

Wskaźnik na modyfikowalny obszar pamięci

```
char  ObszarPam[ ] = "łódka";
char* wNapis = ObszarPam;
wNapis[0] = 'w';
char* ObszarPam2[ ] = "udka";
wNapis = ObszarPam2;
```

Podsumowanie:

- obszar pamięci dostępny poprzez wskaźnik może być modyfikowany (dotyczy to operacji z wykorzystaniem tej zmiennej wskaźnikowej),
- zawartość zmiennej wskaźnikowej może ulegać zmianie,
- zmienna wskaźnikowa nie musi być inicjalizowana w momencie jej definicji.

Wskaźnik na modyfikowalny obszar pamięci

```
char  ObszarPam[ ] = "łódka";
char* wNapis = ObszarPam;
wNapis[0] = 'w';
char* ObszarPam2[ ] = "udka";
wNapis = ObszarPam2;
```

Podsumowanie:

- obszar pamięci dostępny poprzez wskaźnik może być modyfikowany (dotyczy to operacji z wykorzystaniem tej zmiennej wskaźnikowej),
- zawartość zmiennej wskaźnikowej może ulegać zmianie,
- zmienna wskaźnikowa nie musi być inicjalizowana w momencie jej definicji.

Wskaźnik na modyfikowalny obszar pamięci

```
char  ObszarPam[ ] = "łódka";
char* wNapis = ObszarPam;
wNapis[0] = 'w';
char* ObszarPam2[ ] = "udka";
wNapis = ObszarPam2;
```

Podsumowanie:

- obszar pamięci dostępny poprzez wskaźnik może być modyfikowany (dotyczy to operacji z wykorzystaniem tej zmiennej wskaźnikowej),
- zawartość zmiennej wskaźnikowej może ulegać zmianie,
- zmienna wskaźnikowa nie musi być inicjalizowana w momencie jej definicji.

Wskaźnik na modyfikowalny obszar pamięci

```
char  ObszarPam[ ] = "łódka";
char* wNapis;
wNapis = ObszarPam;
wNapis[0] = 'w';
    ...
```

Podsumowanie:

- obszar pamięci dostępny poprzez wskaźnik może być modyfikowany (dotyczy to operacji z wykorzystaniem tej zmiennej wskaźnikowej),
- zawartość zmiennej wskaźnikowej może ulegać zmianie,
- zmienna wskaźnikowa nie musi być inicjalizowana w momencie jej definicji.

Definicje zmiennych wskaźnikowych z **const**

char*	wNapis;
const char*	wNapis;
char* const	wNapis;
const char* const	wNapis;

Wskaźnik na tekst, który **nie może** być modyfikalny.

Wskaźnik na niemodyfikowalny obszar pamięci

```
char ObszarPam[ ] = "łódka";
const char* wNapis = ObszarPam;
```

Wskaźnik na niemodyfikowalny obszar pamięci

```
char ObszarPam[ ] = "łódka";
const char* wNapis = ObszarPam;
wNapis[0] = 'w';
```

Wskaźnik na niemodyfikowalny obszar pamięci

```
char ObszarPam[ ] = "łódka";
const char* wNapis = ObszarPam;
wNapis[0] = 'w';
```

```
napis.cpp: In function ‘int main()’:
napis.cpp:22: error: assignment of read-only location ‘* wNapis’
```

Wskaźnik na niemodyfikowalny obszar pamięci

```
char ObszarPam[ ] = "łódka";
const char* wNapis = ObszarPam;
wNapis[0] = 'w';
wNapis = "budka";
```

Wskaźnik na niemodyfikowalny obszar pamięci

```
char ObszarPam[ ] = "łódka";
const char* wNapis = ObszarPam;
wNapis[0] = 'w';
wNapis = "budka";
```

Podsumowanie:

- obszar pamięci dostępny poprzez wskaźnik nie może być modyfikowany (dotyczy to tylko operacji z wykorzystaniem tej zmiennej wskaźnikowej),
- zawartość zmiennej wskaźnikowej może ulegać zmianie,
- zmienna wskaźnikowa nie musi być inicjalizowana w momencie jej definicji.

Wskaźnik na niemodyfikowalny obszar pamięci

```
char  ObszarPam[ ] = "łódka";
const char*  wNapis = ObszarPam;
wNapis[0] = 'w';
wNapis = "budka";
```

Podsumowanie:

- obszar pamięci dostępny poprzez wskaźnik nie może być modyfikowany (dotyczy to tylko operacji z wykorzystaniem tej zmiennej wskaźnikowej),
- zawartość zmiennej wskaźnikowej może ulegać zmianie,
- zmienna wskaźnikowa nie musi być inicjalizowana w momencie jej definicji.

Wskaźnik na niemodyfikowalny obszar pamięci

char ObszarPam[ ] = "łódka";

const char* wNapis = ObszarPam;

wNapis[0] = 'w';

wNapis = "budka";

Podsumowanie:

- obszar pamięci dostępny poprzez wskaźnik nie może być modyfikowany (dotyczy to tylko operacji z wykorzystaniem tej zmiennej wskaźnikowej),
- zawartość zmiennej wskaźnikowej może ulegać zmianie,
- zmienna wskaźnikowa nie musi być inicjalizowana w momencie jej definicji.

Wskaźnik na niemodyfikowalny obszar pamięci

```
char  ObszarPam[ ] = "łódka";
const char*  wNapis = ObszarPam;
wNapis[0] = 'w';
wNapis = "budka";
```

Podsumowanie:

- obszar pamięci dostępny poprzez wskaźnik nie może być modyfikowany (dotyczy to tylko operacji z wykorzystaniem tej zmiennej wskaźnikowej),
- zawartość zmiennej wskaźnikowej może ulegać zmianie,
- zmienna wskaźnikowa nie musi być inicjalizowana w momencie jej definicji.

Wskaźnik na niemodyfikowalny obszar pamięci

```
char ObszarPam[ ] = "łódka";
const char* wNapis;
wNapis = ObszarPam;
wNapis[0] = 'w';
wNapis = "budka";
```

Podsumowanie:

- obszar pamięci dostępny poprzez wskaźnik nie może być modyfikowany (dotyczy to tylko operacji z wykorzystaniem tej zmiennej wskaźnikowej),
- zawartość zmiennej wskaźnikowej może ulegać zmianie,
- zmienna wskaźnikowa nie musi być inicjalizowana w momencie jej definicji.

Dlaczego czasem coś nie działa

const char* wNapis = "łódka";

Dlaczego czasem coś nie działa

```
char* wNapis = "łódka";
```

Dlaczego czasem coś nie działa

```
char* wNapis = "łódka";
```

Takie podstawienie może być zródłem poważnych błędów

Dlaczego czasem coś nie działa

```
char* wNapis = "łódka";
wNapis[0] = 'w';
```

```
napis.cpp:6: warning: deprecated conversion from string constant to ‘char*’
```

Dlaczego czasem coś nie działa

```
char* wNapis = "łódka";
wNapis[0] = 'w';
```

Jeżeli jednak zignorujemy ostrzeżenie, to ...

Dlaczego czasem coś nie działa

```
char* wNapis = "łódka";
wNapis[0] = 'w';
```

```
Segmentation fault (core dumped)
```

Dlaczego czasem coś nie działa

```
char* wNapis = "łódka";
wNapis[0] = 'w';
```

`Segmentation fault (core dumped)`

Wystąpienie tego typu błędu zależy od systemu i użytego kompilatora. Nie zmienia to jednak faktu, że operacja jest niepoprawna.

Definicje zmiennych wskaźnikowych z **const**

```
char*             wNapis;
const char*       wNapis;
char* const       wNapis;
const char* const wNapis;
```

Wskaźnik **staly** na tekst, który **może** być modyfikowany.

Wskaźnik stały na modyfikowalny obszar pamięci

```
char ObszarPam[ ] = "łódka";
char* const wNapis = ObszarPam;
```

Wskaźnik stały na modyfikowalny obszar pamięci

```
char ObszarPam[ ] = "łódka";
char* const wNapis = ObszarPam;
wNapis[0] = 'w';
```

Wskaźnik stały na modyfikowalny obszar pamięci

```
char ObszarPam[ ] = "łódka";
char* const wNapis = ObszarPam;
wNapis[0] = 'w';
```

Wskaźnik stały na modyfikowalny obszar pamięci

```
char ObszarPam[ ] = "łódka";
char* const wNapis = ObszarPam;
wNapis[0] = 'w';
wNapis = "budka";
```

Wskaźnik stały na modyfikowalny obszar pamięci

```
char ObszarPam[ ] = "łódka";
char* const wNapis = ObszarPam;
wNapis[0] = 'w';
wNapis = "budka";
```

Wskaźnik stały na modyfikowalny obszar pamięci

```
char ObszarPam[ ] = "łódka";
char* const wNapis = ObszarPam;
wNapis[0] = 'w';
wNapis = "budka";
```

Podsumowanie:

- obszar pamięci dostępny poprzez wskaźnik może być modyfikowany,
- zawartość zmiennej wskaźnikowej **nie** może ulegać zmianie,
- zmienna wskaźnikowa **musi** być inicjalizowana w momencie jej definicji.

Wskaźnik stały na modyfikowalny obszar pamięci

```
char ObszarPam[ ] = "łódka";
char* const wNapis = ObszarPam;
wNapis[0] = 'w';
wNapis = "budka";
```

Podsumowanie:

- obszar pamięci dostępny poprzez wskaźnik może być modyfikowany,
- zawartość zmiennej wskaźnikowej **nie** może ulegać zmianie,
- zmienna wskaźnikowa **musi** być inicjalizowana w momencie jej definicji.

Wskaźnik stały na modyfikowalny obszar pamięci

```
char ObszarPam[ ] = "łódka";
char* const wNapis = ObszarPam;
wNapis[0] = 'w';
wNapis = "budka";
```

Podsumowanie:

- obszar pamięci dostępny poprzez wskaźnik może być modyfikowany,
- zawartość zmiennej wskaźnikowej **nie** może ulegać zmianie,
- zmienna wskaźnikowa **musi** być inicjalizowana w momencie jej definicji.

Wskaźnik stały na modyfikowalny obszar pamięci

```
char ObszarPam[ ] = "łódka";
char* const wNapis = ObszarPam;
wNapis[0] = 'w';
wNapis = "budka";
```

Podsumowanie:

- obszar pamięci dostępny poprzez wskaźnik może być modyfikowany,
- zawartość zmiennej wskaźnikowej **nie** może ulegać zmianie,
- zmienna wskaźnikowa **musi** być inicjalizowana w momencie jej definicji.

Wskaźnik stały na modyfikowalny obszar pamięci

```
char ObszarPam[ ] = "łódka";
char* const wNapis;
wNapis = ObszarPam;
wNapis[0] = 'w';
wNapis = "budka";
```

Podsumowanie:

- obszar pamięci dostępny poprzez wskaźnik może być modyfikowany,
- zawartość zmiennej wskaźnikowej **nie** może ulegać zmianie,
- zmienna wskaźnikowa **musi** być inicjalizowana w momencie jej definicji.

Definicje zmiennych wskaźnikowych z **const**

char*	wNapis;
const char*	wNapis;
char* const	wNapis;
const char* const	wNapis;

Wskaźnik **stały** na tekst, który **nie może** być modyfikalny.

Podsumowanie

char*	wNapis
const char*	wNapis
char* const	wNapis
const char* const	wNapis

Wskaźnik na tekst, który **może** być modyfikany.

Podsumowanie

char*	wNapis
const char*	wNapis
char* const	wNapis
const char* const	wNapis

Wskaźnik na tekst, który **nie może** być modyfikany.

Podsumowanie

char*	wNapis
const char*	wNapis
char* const	wNapis
const char* const	wNapis

Wskaźnik **stały** na tekst, który **może** być modyfikany.

Podsumowanie

char*	wNapis
const char*	wNapis
char* const	wNapis
const char* const	wNapis

Wskaźnik **stały** na tekst, który **nie może** być modyfikany.

Konwencja zapisu

char*	wNapis
const char*	wNapis
char* const	wNapis
const char* const	wNapis

Konwencja zapisu definicji zmiennej wskaźnikowej nie jest jednoznaczna.

Konwencja zapisu

char*	wNapis
char const*	wNapis
char* **const**	wNapis
char const* **const**	wNapis

Konwencja zapisu definicji zmiennej wskaźnikowej nie jest jednoznaczna.

Użycie modyfikatora **const**

Modyfikator **const** odgrywa znacznie większą rolę w odniesieniu do zmiennych wskaźnikowych.
Jego podstawowym miejscem użycia jest lista parametrów funkcji, np.

```
char *strcpy(char *dest, const char *src);
```

(patrz: `man strcpy`)

Użycie modyfikatora **const** pozwala wskazać, co może się zmienić wskutek działania funkcji, a co zmienić się nie może.
Pozwala również uniknąć przypadkowych błędów w konstrukcji samej funkcji.

Użycie modyfikatora **const**

Modyfikator **const** odgrywa znacznie większą rolę w odniesieniu do zmiennych wskaźnikowych.
Jego podstawowym miejscem użycia jest lista parametrów funkcji, np.

```
char *strcpy(char *dest, const char *src);
```

(patrz: man strcpy)

Użycie modyfikatora **const** pozwala wskazać, co może się zmienić wskutek działania funkcji, a co zmienić się nie może.
Pozwala również uniknąć przypadkowych błędów w konstrukcji samej funkcji.

Przykład wykorzystania **const** w liście parametrów

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

void WyswietlPomijajacSpacje( const char *Napis )
{
   for (; *Napis && (*Napis == ' '); ++Napis);
   cout << Napis << endl;
}

int main( )
{
   WyswietlPomijajacSpacje("     Ile jeszcze do końca?");
}
```

Przykład wykorzystania **const** w liście parametrów

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

void WyswietlPomijajacSpacje( const char *Napis )
{
   for (; *Napis && (*Napis == ' '); ++Napis);
   cout << Napis << endl;
}

int main( )
{
   WyswietlPomijajacSpacje("      Ile jeszcze do końca?");
}
```

Czy wszystko jest tu dobrze?

Przykład wykorzystania **const** w liście parametrów

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

void WyswietlPomijajacSpacje( char *Napis )
{
   for (; *Napis && (*Napis == ' '); ++Napis);
   cout << Napis << endl;
}

int main( )
{
   WyswietlPomijajacSpacje("     Ile jeszcze do końca?");
}
```

Przy takiej deklaracji faktycznie program pod względem składni byłby poprawny. Jednakże jego konstrukcja nie byłaby poprawna.

Przykład wykorzystania **const** w liście parametrów

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

void WyswietlPomijajacSpacje( const char *Napis )
{
   for (; *Napis && (*Napis == ' '); ++Napis);
   cout << Napis << endl;
}

int main( )
{
   WyswietlPomijajacSpacje("      Ile jeszcze do końca?");
}
```

Właściwa deklaracja parametrów zapewnia lepszą kontrolę operacji wewnątrz funkcji. Pozwala to wykryć błąd o znaczeniu semantycznym.

Przykład wykorzystania **const** w liście parametrów

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

void WyswietlPomijajacSpacje( const char *Napis )
{
   for (; *Napis && (*Napis == ' '); ++Napis);
   cout << Napis << endl;
}

int main( )
{
   WyswietlPomijajacSpacje("     Ile jeszcze do końca?");
}
```

Właściwa deklaracja parametrów zapewnia lepszą kontrolę operacji wewnątrz funkcji. Pozwala to wykryć błąd o znaczeniu semantycznym.

Przykład wykorzystania **const** w liście parametrów

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

void WyswietlPomijajacSpacje( const char *Napis )
{
   for (; *Napis && (*Napis == ' '); ++Napis);
   cout << Napis << endl;
}

int main( )
{
   WyswietlPomijajacSpacje("      Ile jeszcze do końca?");
}
```

Komunikat kompilatora:

```
jkowalsk@noxon: prog> g++ wyswietl.cpp
wyswietl.cpp: In function 'void WyswietlPomijajacSpacje(const char*)':
wyswietl.cpp:6: error: assignment of read-only location
```

Przykład wykorzystania **const** w liście parametrów

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

void WyswietlPomijajacSpacje( const char *Napis )
{
   for (; *Napis && (*Napis == ' '); ++Napis);
   cout << Napis << endl;
}

int main( )
{
   WyswietlPomijajacSpacje("      Ile jeszcze do końca?");
}
```

Teraz jest już wszystko dobrze.

Plan prezentacji

Przekazywanie parametrów przez referencję

```
void Poteguj(int & Wart)
{
   Wart *= Wart;
}

int main( )
{
   int Zm=2;

   Poteguj(Zm);
   cout << Zm << endl;
}
```

Używając referencji możemy poprzez parametr przekazać do funkcji zmienną, której zmiany wartości będą widoczne na „zewnątrz”.

Przekazywanie parametrów przez referencję

```
void Poteguj(int & Wart)
{
   Wart *= Wart;
}

int main( )
{
   int Zm=2;

   Poteguj(Zm);
   cout << Zm << endl;
}
```

```
void Poteguj(int * const wWart)
{
   *wWart *= *wWart;
}

int main( )
{
   int Zm=2;

   Poteguj(&Zm);
   cout << Zm << endl;
}
```

Analog tej konstrukcji możemy napisać wykorzystując przekazywanie parametru poprzez wskaźnik.

Przekazywanie parametrów przez referencję

```
int Poteguj(int & Wart)
{
   return Wart * Wart;
}

int main( )
{
   int Zm=2;

   Zm = Poteguj(Zm);
   cout << Zm << endl;
}
```

```
int Poteguj(int * const wWart)
{
   return *wWart * *wWart;
}

int main( )
{
   int Zm=2;

   Zm = Poteguj(&Zm);
   cout << Zm << endl;
}
```

Zapiszmy funkcję *Poteguj* faktycznie jako funkcje, tzn. procedurę obliczeniową zwracającą wyliczoną wartość.

Przekazywanie parametrów przez referencję

```
int Poteguj(int & Wart)
{
   return Wart * Wart;
}

int main( )
{
   int Zm=2;

   Zm = Poteguj(2);
   cout << Zm << endl;
}
```

```
int Poteguj(int * const wWart)
{
   return *wWart * *wWart;
}

int main( )
{
   int Zm=2;

   Zm = Poteguj(&Zm);
   cout << Zm << endl;
}
```

Korzystając z funkcji możemy bezpośrednio w liście parametrów posłużyć się stałą liczbową, zaś wynik przypisać zmiennej.

Przekazywanie parametrów przez referencję

```
int Poteguj(int & Wart)
{
   return Wart * Wart;
}

int main( )
{
   int Zm=2;
   Zm = Poteguj(2);
   cout << Zm << endl;
}
```

```
int Poteguj(int * const wWart)
{
   return *wWart * *wWart;
}

int main( )
{
   int Zm=2;
   Zm = Poteguj(&Zm);
   cout << Zm << endl;
}
```

W tym konkretnym przypadku nie jest to jednak możliwe, gdyż w liście parametrów występuje referencja do zmiennej.

Przekazywanie parametrów przez referencję

```
int Poteguj(int & Wart)
{
   return Wart * Wart;
}

int main( )
{
   int Zm=2;
   Zm = Poteguj(2);
   cout << Zm << endl;
}
```

```
int Poteguj(int * const wWart)
{
   return *wWart * *wWart;
}

int main( )
{
   int Zm=2;
   Zm = Poteguj(&2);
   cout << Zm << endl;
}
```

A czy analogiczna konstrukcja byłaby poprawna przy przekazaniu parametru przez wskaźnik?

Przekazywanie parametrów przez referencję

```
int Poteguj(int & Wart)
{
   return Wart * Wart;
}

int main( )
{
   int Zm=2;
   Zm = Poteguj(2);
   cout << Zm << endl;
}
```

```
int Poteguj(int * const wWart)
{
   return *wWart * *wWart;
}

int main( )
{
   int Zm=2;
   Zm = Poteguj(&2);
   cout << Zm << endl;
}
```

Absolutnie NIE!!! Gdyż wskaźnik w tym przykładzie, podobnie jak referencja w konstrukcji obok, jest wskaźnikiem na obszar modyfikowalny.

Przekazywanie parametrów przez referencję

```
int Poteguj(int & Wart)
{
   return Wart * Wart;
}

int main( )
{
   int Zm=2;
   Zm = Poteguj(2);
   cout << Zm << endl;
}
```

```
int Poteguj(int * const wWart)
{
   return *wWart * *wWart;
}

int main( )
{
   int Zm=2;
   Zm = Poteguj(&2);
   cout << Zm << endl;
}
```

Czy naprawdę nic się nie da tu zrobić?

Przekazywanie parametrów przez referencję

```
int Poteguj(const int & Wart)
{
   return Wart * Wart;
}

int main( )
{
   int Zm=2;

   Zm = Poteguj(2);
   cout << Zm << endl;
}
```

```
int Poteguj(int * const wWart)
{
   return *wWart * *wWart;
}

int main( )
{
   int Zm=2;

   Zm = Poteguj(&2);
   cout << Zm << endl;
}
```

W przypadku parametru przekazywanego przez referencję wystarczy, że parametr ten zadeklarujemy jako parametr niemodyfikowalny.

Przekazywanie parametrów przez referencję

```
int Poteguj(const int & Wart)
{
   return Wart * Wart;
}

int main( )
{
   int Zm=2;

   Zm = Poteguj(2);
   cout << Zm << endl;
}
```

```
int Poteguj(const int * const wWart)
{
   return *wWart * *wWart;
}

int main( )
{
   int Zm=2;

   Zm = Poteguj(&2);    ???
   cout << Zm << endl;
}
```

Można by sądzić (naiwnie), że analogiczna konstrukcja będzie również poprawna w przypadku parametrów przekazywanych przez wskaźnik.

Przekazywanie parametrów przez referencję

```
int Poteguj(const int & Wart)
{
   return Wart * Wart;
}

int main( )
{
   int Zm=2;

   Zm = Poteguj(2);
   cout << Zm << endl;
}
```

```
int Poteguj(const int * const wWart)
{
   return *wWart * *wWart;
}

int main( )
{
   int Zm=2;

   Zm = Poteguj(&2);
   cout << Zm << endl;
}
```

Tak jednak nie jest. Nie można posługiwać się jawnymi adresami do stałych skalarnych, gdyż nie muszą się one w ogóle znajdować w obszarze danych programu.

Plan prezentacji

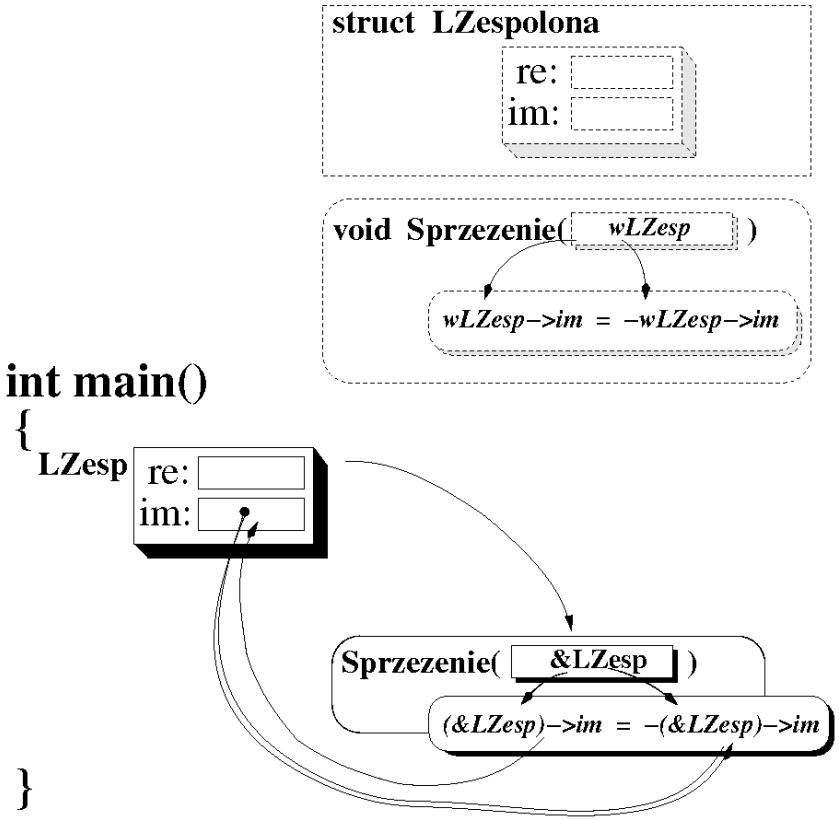
Od funkcji do metody

Jeszcze funkcja

```
struct LZespolona {
                    float  re;
                    float  im;
                  };


void  sprzezenie( LZespolona *wLZesp )
{
      wLZesp->im = - wLZesp->im;
}

int  main( )
{
      LZespolona    LZesp;

      LZesp.re = 5;   LZesp.im = -6;
      sprzezenie(&LZesp);
}
```

Model struktury

Struktura i funkcja



Od funkcji do metody

Jeszcze funkcja

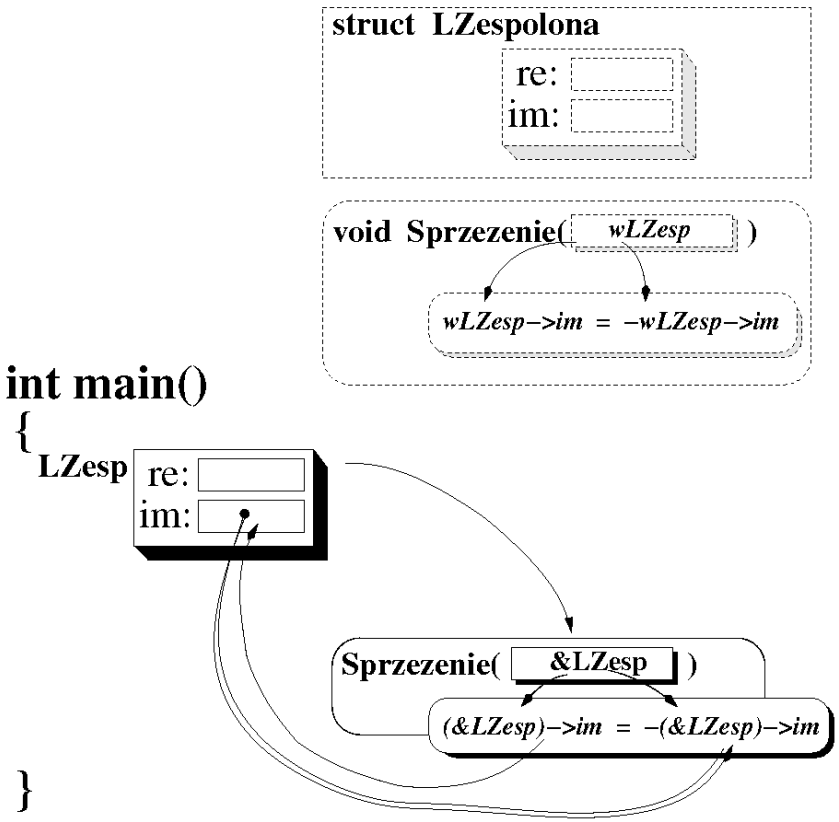
```
struct LZespolona {
                    float  re;
                    float  im;
                  };


void  sprzezenie( LZespolona *wLZesp )
{
      wLZesp->im = - wLZesp->im;
}

int  main( )
{
      LZespolona     LZesp;

      LZesp.re = 5;   LZesp.im = -6;
      sprzezenie(&LZesp);
}
```

Już metoda

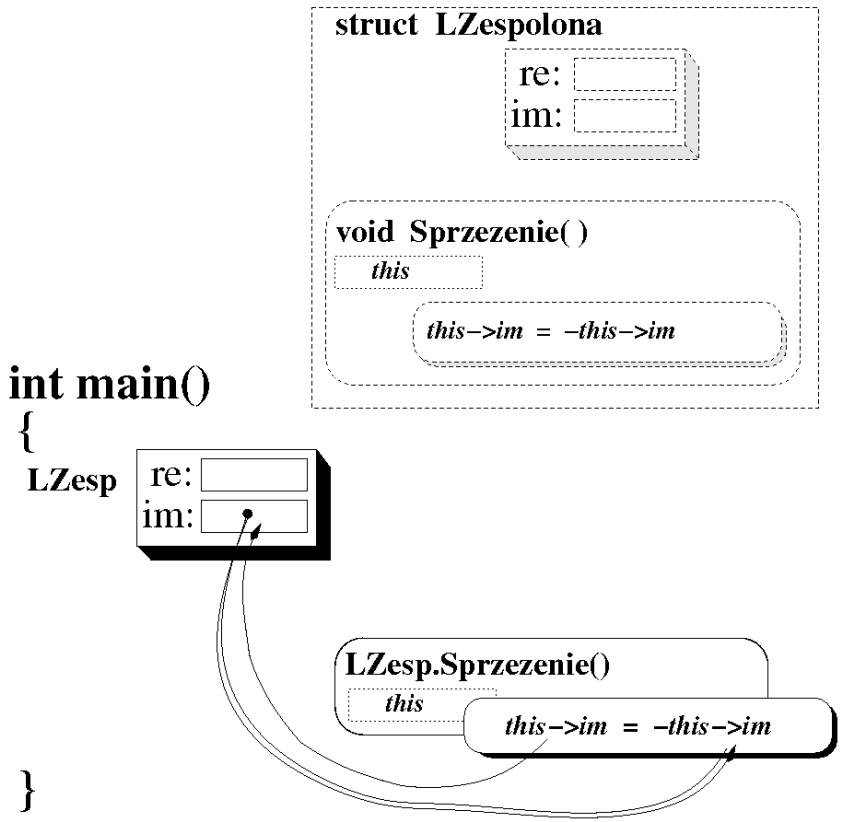
```
struct LZespolona {
                    float  re;
                    float  im;

                    void  Sprzezenie( );
                  };

void  LZespolona::Sprzezenie( )
{
      im = - im;
}

int  main( )
{
      LZespolona     LZesp;

      LZesp.re = 5;   LZesp.im = -6;
      LZesp.Sprzezenie( );
}
```

Model obiektu

Struktura i funkcja



Struktura i metoda



Funkcja działająca na strukturze

```
struct LZespolona  { // .....................................
    float  re;
    float  im;
}; // ..........................................................

LZespolona  dodaj( LZespolona  Z1,  LZespolona  Z2 )
{
    Z2.re += Z1.re;    Z2.im += Z1.im;
    return  Z2;
}

int  main( )
{
    LZespolona  lz1,  lz2,  lz3;

    lz3 = dodaj( lz1, lz2 );
}
```

Funkcja działająca na strukturze

```
struct LZespolona  { //  ..........................................
      float  re;
      float  im;
 }; //  ..........................................................


LZespolona  dodaj( LZespolona  Z1,  LZespolona  Z2  )
{
     Z2.re += Z1.re;    Z2.im += Z1.im;
}    return    Z2;


int  main( )
{
     LZespolona    lz1,  lz2,  lz3;

     lz3 = dodaj( lz1, lz2 );
}
```

Funkcja działająca na strukturze

```
struct LZespolona  { // .....................................
      float  re;
      float  im;
 }; // ..........................................................


LZespolona  dodaj( LZespolona  Z1,  LZespolona  Z2  )
{
      Z2.re += Z1.re;    Z2.im += Z1.im;
}     return    Z2;


int  main( )
{
      LZespolona    lz1,  lz2,  lz3;

      lz3 = dodaj( lz1, lz2 );
}
```

Od funkcji do metody

```
struct LZespolona  { //  ..........................................
    float  re;
    float  im;
}; //  ..........................................................

LZespolona  dodaj( LZespolona  Z1,  LZespolona  Z2  )
{
    Z2.re += Z1.re;    Z2.im += Z1.im;
    return  Z2;
}

int  main( )
{
    LZespolona  lz1,  lz2,  lz3;
    lz3 = dodaj( lz1, lz2 )
}
```

Od funkcji do metody

```
struct LZespolona  { //  ..........................................
    float  re;
    float  im;

    LZespolona  dodaj(  LZespolona  Z2  );
};  //  ..........................................................

LZespolona  dodaj( LZespolona  Z1,  LZespolona  Z2  )
{
    Z2.re += Z1.re;    Z2.im += Z1.im;
    return  Z2;
}


int  main( )
{
    LZespolona  lz1,  lz2,  lz3;

    lz3 = dodaj( lz1, lz2 )
}
```

Od funkcji do metody

```
struct LZespolona  { //  ..........................................
    float  re;
    float  im;

    LZespolona  dodaj(  LZespolona  Z2  );
};  //  ..........................................................

LZespolona  dodaj( LZespolona  Z1,  LZespolona  Z2  )
{
    Z2.re += Z1.re;    Z2.im += Z1.im;
    return  Z2;
}


int  main( )
{
    LZespolona  lz1,  lz2,  lz3;

    lz3 = dodaj( lz1, lz2 )
}
```

Od funkcji do metody

```
struct LZespolona  { //  ..........................................
    float re;
    float im;

    LZespolona dodaj( LZespolona Z2 );
}; //  ..........................................................

LZespolona LZespolona::dodaj( LZespolona Z2 )
{
    Z2.re += re;    Z2.im += im;
    return Z2;
}


int main( )
{
    LZespolona  lz1, lz2, lz3;

    lz3 = dodaj( lz1, lz2 )
}
```

Od funkcji do metody

```
struct LZespolona  { //  ..........................................
      float  re;
      float  im;

      LZespolona  dodaj(  LZespolona  Z2  );
  }; //  ..........................................................

LZespolona  LZespolona::dodaj( LZespolona  Z2  )
{
      Z2.re += re;    Z2.im += im;
}     return    Z2;


int  main( )
{
      LZespolona    lz1,  lz2,  lz3;

      lz3 = dodaj( lz1, lz2 )
}
```

Od funkcji do metody

```
struct LZespolona  { //  ..........................................
      float  re;
      float  im;

      LZespolona  dodaj(  LZespolona  Z2  );
  }; //  ..........................................................

LZespolona  LZespolona::dodaj( LZespolona  Z2  )
{
      Z2.re += re;    Z2.im += im;
}     return    Z2;


int  main( )
{
      LZespolona    lz1,  lz2,  lz3;

      lz3 = lz1.dodaj( lz2 )
}
```

Od funkcji do metody

```
struct LZespolona  { //  ..........................................
      float  re;
      float  im;

      LZespolona  dodaj(  LZespolona  Z2  );
  }; //  ..........................................................

LZespolona  LZespolona::dodaj( LZespolona  Z2  )
{
      Z2.re += this->re;    Z2.im += this->im;
}     return    Z2;

int  main( )
{
      LZespolona    lz1,  lz2,  lz3;

      lz3 = lz1.dodaj( lz2 )
}
```

Od metody do funkcji w C

```
struct LZespolona  { // ..........................................
      float  re;
      float  im;
 }; // ..........................................................
 typedef  struct LZespolona  LZespolona;

LZespolona  LZespolona::dodaj( LZespolona  Z2  )
{
     Z2.re += this->re;    Z2.im += this->im;
}    return   Z2;

int  main( )
{
     LZespolona   lz1,  lz2,  lz3;
     lz3 = lz1.dodaj( lz2 );
     return  0;
}
```

Od metody do funkcji w C

```
struct LZespolona  { // ..........................................
      float  re;
      float  im;
 }; // ..........................................................
 typedef  struct LZespolona  LZespolona;

LZespolona  dodaj( LZespolona  *this,  LZespolona  Z2  )
{
     Z2.re += this->re;    Z2.im += this->im;
     return    Z2;
}


int  main( )
{
     LZespolona    lz1,  lz2,  lz3;
     lz3 = lz1.dodaj( lz2 );
     return  0;
}
```

Od metody do funkcji w C

```
struct LZespolona  { // ..........................................
      float  re;
      float  im;
 }; // ..........................................................
 typedef  struct LZespolona  LZespolona;

LZespolona  dodaj( LZespolona  *this,  LZespolona  Z2  )
{
     Z2.re += this->re;    Z2.im += this->im;
     return    Z2;
}

int  main( )
{
     LZespolona    lz1,  lz2,  lz3;
     lz3 = lz1.dodaj( lz2 );
     return  0;
}
```

Od metody do funkcji w C

```
struct LZespolona  { // ..........................................
      float  re;
      float  im;
 }; // ..........................................................
 typedef  struct LZespolona  LZespolona;

LZespolona  dodaj( LZespolona  *this,  LZespolona  Z2  )
{
     Z2.re += this->re;    Z2.im += this->im;
     return    Z2;
}

int  main( )
{
     LZespolona    lz1,  lz2,  lz3;
     lz3 = dodaj( &lz1, lz2 );
     return  0;
}
```

Ten sam zapis w C++

```
struct LZespolona  { //  ..........................................
    float  re;
    float  im;

    LZespolona  dodaj(  LZespolona  Z2  );
}; //  ..........................................................

LZespolona  LZespolona::dodaj( LZespolona  Z2  )
{
    Z2.re += this->re;    Z2.im += this->im;
    return    Z2;
}


int  main( )
{
    LZespolona   lz1,  lz2,  lz3;

    lz3 = dodaj( &lz1, lz2 );
}
```

Ten sam zapis w C++

```
struct LZespolona  { //  ..................................
    float  re;
    float  im;

    LZespolona  dodaj(  LZespolona  Z2  );
}; //  ..................................................

LZespolona  LZespolona::dodaj( LZespolona  Z2  )
{
    Z2.re += this->re;    Z2.im += this->im;
    return  Z2;
}


int  main( )
{
    LZespolona  lz1,  lz2,  lz3;

    lz3 = lz1.dodaj( lz2 );
}
```

Ten sam zapis w C++

```
struct LZespolona  { // ..................................
    float re;
    float im;

    LZespolona dodaj( LZespolona Z2 );
}; // ..................................................

LZespolona LZespolona::dodaj( LZespolona Z2 )
{
    Z2.re += re;    Z2.im += im;
    return Z2;
}

int main( )
{
    LZespolona lz1, lz2, lz3;

    lz3 = lz1.dodaj( lz2 );
}
```

Modifikator **const**

```
struct LZespolona  { // .....................................
      float  re;
      float  im;
 }; // ..........................................................
 typedef  struct LZespolona  LZespolona;

LZespolona  dodaj( LZespolona  *this,  LZespolona  Z2  )
{
     Z2.re += this->re;    Z2.im += this->im;
}    return    Z2;


int  main( )
{
     LZespolona    lz1,  lz2,  lz3;
     lz3 = dodaj( &lz1, lz2 );
     return  0;
}
```

Modifikator **const**

```
struct LZespolona  { // .....................................
      float  re;
      float  im;
 }; // ..........................................................
 typedef  struct LZespolona  LZespolona;

LZespolona  dodaj( LZespolona  *this,  LZespolona  Z2  )
{
     Z2.re += this->re;    Z2.im += this->im;
}    return    Z2;

int  main( )
{
     LZespolona    lz1,  lz2,  lz3;
     lz3 = dodaj( &lz1, lz2 );
     return  0;
}
```

Modifikator **const**

```
struct LZespolona  { // .....................................
      float  re;
      float  im;
 }; // ..........................................................
 typedef  struct LZespolona  LZespolona;

LZespolona  dodaj( const  LZespolona  *this,  LZespolona  Z2  )
{
     Z2.re += this->re;    Z2.im += this->im;
     return    Z2;
}

int  main( )
{
     LZespolona    lz1,  lz2,  lz3;
     lz3 = dodaj( &lz1, lz2 );
     return  0;
}
```

Modifikator **const** w metodach

```
struct LZespolona  { //  ..........................................
    float  re;
    float  im;

    LZespolona  dodaj(  LZespolona  Z2  ) const ;
}; //  ..........................................................

LZespolona  LZespolona::dodaj( LZespolona  Z2  ) const
{
    Z2.re += this->re;    Z2.im += this->im;
    return    Z2;
}


int  main( )
{
    LZespolona    lz1,  lz2,  lz3;

    lz3 = lz1.dodaj( lz2 );
}
```

Modifikator **const** w metodach

```
struct LZespolona  { //  ..........................................
    float  re;
    float  im;

    LZespolona  dodaj(  LZespolona  Z2  ) const ;
}; //  ..........................................................

LZespolona  LZespolona::dodaj( LZespolona  Z2  ) const
{
    Z2.re += re;    Z2.im += im;
}   return    Z2;


int  main( )
{
    LZespolona    lz1,  lz2,  lz3;

    lz3 = lz1.dodaj( lz2 );
}
```

Plan prezentacji

Od funkcji do operatora

```
struct LZespolona  { // .....................................
    float re;
    float im;
}; // ..........................................................

LZespolona dodaj( LZespolona Z1, LZespolona Z2 )
{
    Z2.re += Z1.re;    Z2.im += Z1.im;
    return Z2;
}

int main( )
{
    LZespolona lz1, lz2, lz3;

    lz3 = dodaj( lz1, lz2 );
}
```

Od funkcji do operatora

```
struct LZespolona  { //  ..........................................
      float  re;
      float  im;
 }; //  ..........................................................

LZespolona  dodaj( LZespolona  Z1,  LZespolona  Z2  )
{
     Z2.re += Z1.re;    Z2.im += Z1.im;
}    return    Z2;

int  main( )
{
     LZespolona    lz1,  lz2,  lz3;

     lz3 = dodaj( lz1, lz2 );
}
```

Od funkcji do operatora

```
struct LZespolona  { //  ..........................................
    float  re;
    float  im;
 }; //  ..........................................................

LZespolona  operator + ( LZespolona  Z1,  LZespolona  Z2  )
{
    Z2.re += Z1.re;    Z2.im += Z1.im;
}   return   Z2;

int  main( )
{
    LZespolona   lz1,  lz2,  lz3;

    lz3 = dodaj( lz1, lz2 );
}
```

Od funkcji do operatora

```
struct LZespolona  { // .....................................
      float  re;
      float  im;
 }; // ..........................................................


LZespolona  operator + ( LZespolona  Z1,  LZespolona  Z2  )
{
     Z2.re += Z1.re;    Z2.im += Z1.im;
}    return    Z2;


int  main( )
{
     LZespolona    lz1,  lz2,  lz3;

     lz3 = dodaj( lz1, lz2 );
}
```

Od funkcji do operatora

```
struct LZespolona  { // .....................................
      float  re;
      float  im;
 }; // ..........................................................


LZespolona  operator + ( LZespolona  Z1,  LZespolona  Z2  )
{
     Z2.re += Z1.re;    Z2.im += Z1.im;
}    return    Z2;


int  main( )
{
     LZespolona    lz1,  lz2,  lz3;

     lz3 = lz1 + lz2;
}
```

Od funkcji do operatora

```
struct LZespolona  { // .....................................
      float  re;
      float  im;
 }; // ..........................................................


LZespolona  operator + ( LZespolona  Z1,  LZespolona  Z2  )
{
     Z2.re += Z1.re;    Z2.im += Z1.im;
     return    Z2;
}


int  main( )
{
     LZespolona    lz1,  lz2,  lz3;

     lz3 = lz1 + lz2 + lz1;
}
```

Od funkcji do operatora

```
struct LZespolona  { // .....................................
      float  re;
      float  im;
 }; // ..........................................................


LZespolona  operator + ( LZespolona  Z1,  LZespolona  Z2  )
{
     Z2.re += Z1.re;    Z2.im += Z1.im;
}    return    Z2;


int  main( )
{
     LZespolona    lz1,  lz2,  lz3;

     lz3 = lz1 + lz2 + lz1+ (lz1 = l2 + lz1);
}
```

Od funkcji do operatora

```
struct LZespolona  { //  ..........................................
    float  re;
    float  im;
}; //  ..........................................................
```

```
LZespolona  operator + ( LZespolona  Z1,  LZespolona  Z2 )
{
    Z2.re += Z1.re;    Z2.im += Z1.im;
    return  Z2;
}
```

```
int  main( )
{
    LZespolona  lz1,  lz2,  lz3;

    lz3 = lz1 + lz2;
}
```

Od funkcji do operatora

```
struct LZespolona  { //  ..........................................
    float re;
    float im;

    LZespolona operator + ( LZespolona Z2 );
}; //  ..........................................................

LZespolona operator + ( LZespolona Z1, LZespolona Z2 )
{
    Z2.re += Z1.re;    Z2.im += Z1.im;
    return Z2;
}


int main( )
{
    LZespolona  lz1, lz2, lz3;

    lz3 = lz1 + lz2;
}
```

Od funkcji do operatora

```
struct LZespolona  { //  ..........................................
    float re;
    float im;

    LZespolona operator + ( LZespolona Z2 );
}; //  ..........................................................

LZespolona operator + ( LZespolona Z1, LZespolona Z2 )
{
    Z2.re += Z1.re;    Z2.im += Z1.im;
}   return Z2;


int main( )
{
    LZespolona  lz1, lz2, lz3;

    lz3 = lz1 + lz2;
}
```

Od funkcji do operatora

```
struct LZespolona  { //  ..........................................
    float  re;
    float  im;

    LZespolona  operator  + ( LZespolona  Z2 );
}; //  ..........................................................

LZespolona  LZespolona::operator + ( LZespolona  Z2 )
{
    Z2.re += re;    Z2.im += im;
}   return  Z2;


int  main( )
{
    LZespolona   lz1,  lz2,  lz3;

    lz3 = lz1 + lz2;
}
```

Od funkcji do operatora

```
struct LZespolona  { // .....................................
      float  re;
      float  im;

      LZespolona  operator  + ( LZespolona  Z2 );
 }; // ..........................................................

LZespolona  LZespolona::operator + ( LZespolona  Z2 )
{
      Z2.re += re;    Z2.im += im;
}     return    Z2;


int  main( )
{
      LZespolona    lz1,  lz2,  lz3;

      lz3 = lz1 + lz2;
}
```

Od funkcji do operatora

```
struct LZespolona  { //  ..........................................
    float  re;
    float  im;

    LZespolona  operator  + ( LZespolona  Z2 );
}; //  ..........................................................

LZespolona  LZespolona::operator + ( LZespolona  Z2 )
{
    re += Z2.re;    im += Z2.im;
    return    *this;
}


int  main( )
{
    LZespolona   lz1,  lz2,  lz3;

    lz3 = lz1 + lz2;
}
```

A czy tak będzie dobrze?

Od funkcji do operatora

```
struct LZespolona  { //  ..........................................
      float  re;
      float  im;

      LZespolona  operator  + ( LZespolona  Z2  );
 }; //  ..........................................................

LZespolona  LZespolona::operator + ( LZespolona  Z2  )
{
      re += Z2.re;    im += Z2.im;
}     return    *this;


int  main( )
{
      LZespolona    lz1,  lz2,  lz3;

      lz3 = lz1 + lz2;
}
```

Nie jest to poprawna implementacja, gdyż przy okazji obliczenia sumy zmieniana jest zawartość obiektu i jest ona widoczna "na zewnątrz".

Operatory dla różnych argumentów

```
struct LZespolona  { // .....................................
    float re;
    float im;

    LZespolona operator + ( LZespolona Z2 );
}; // ..........................................................

LZespolona LZespolona::operator * ( float liczba )
{
    LZespolona Z;
    Z.re = re * liczba;    Z.im = im * liczba;
    return Z;
}

int main( )
{
    LZespolona lz1, lz2, lz3;

    lz3 = lz1 * 3;
}
```

Operatory dla różnych argumentów

```
struct LZespolona  { // .....................................
    float re;
    float im;

    LZespolona operator * ( float liczba );
}; // ..........................................................

LZespolona LZespolona::operator * ( float liczba )
{
    LZespolona Z;
    Z.re = re * liczba;    Z.im = im * liczba;
    return Z;
}

int main( )
{
    LZespolona lz1, lz2, lz3;

    lz3 = lz1 * 3;
}
```

Operatory dla różnych argumentów

```
struct LZespolona  { //  ..........................................
      float  re;
      float  im;

      LZespolona  operator  * ( float   liczba );
 }; //  ..........................................................


LZespolona  LZespolona::operator * ( float   liczba )
{
      LZespolona   Z;
      Z.re = re * liczba;     Z.im = im * liczba;
      return     Z;
}


int  main( )
{
      LZespolona    lz1,  lz2,  lz3;

      lz3 = lz1 * 3;
}
```

Operatory dla różnych argumentów

```
struct LZespolona  { // ..........................................
    float re;
    float im;

    LZespolona operator * ( float liczba );
}; // ..........................................................

LZespolona LZespolona::operator * ( float liczba )
{
    LZespolona Z;
    Z.re = re * liczba;    Z.im = im * liczba;
    return Z;
}

int main( )
{
    LZespolona lz1, lz2, lz3;

    lz3 = lz1 * 3;
}
```

Operatory dla różnych argumentów

```
struct LZespolona  { //  ..........................................
      float  re;
      float  im;

      LZespolona  operator  * ( float   liczba );
 }; //  ..........................................................

LZespolona  LZespolona::operator * ( float   liczba )
{
      LZespolona   Z;
      Z.re = re * liczba;     Z.im = im * liczba;
      return     Z;
}

int  main( )
{
      LZespolona    lz1,  lz2,  lz3;

      lz3 = 3 * lz1;
}
```

Czy ta operacja `lz3 = 3 * lz1` jest poprawna?

Operatory dla różnych argumentów

```
struct LZespolona  { // .....................................
      float  re;
      float  im;

 }; // ..........................................................

LZespolona  operator * ( LZespolona  Z,  float   liczba )
{
      Z.re = re * liczba;    Z.im = im * liczba;
      return    Z;
}

int  main( )
{
      LZespolona    lz1,  lz2,  lz3;

      lz3 = 3 * lz1;
}
```

Czy ta operacja `lz3 = 3 * lz1` jest poprawna?

Operatory dla różnych argumentów

```
struct LZespolona  { // .....................................
      float  re;
      float  im;

 }; // ..........................................................

LZespolona  operator * ( LZespolona  Z,  float  liczba )
{
      Z.re = re * liczba;    Z.im = im * liczba;
      return    Z;
}

int  main( )
{
      LZespolona    lz1,  lz2,  lz3;

      lz3 = operator * ( 3, lz1 );
}
```

Czy ta operacja `lz3 = 3 * lz1` jest poprawna?

Operatory dla różnych argumentów

```
struct LZespolona  { // .....................................
      float  re;
      float  im;

 }; // ..........................................................

LZespolona  operator * ( LZespolona  Z,  float  liczba )
{
      Z.re = re * liczba;    Z.im = im * liczba;
      return    Z;
}

int  main( )
{
      LZespolona    lz1,  lz2,  lz3;
      lz3 = operator * ( 3, lz1 );
}
```

Operacja nie jest poprawna, gdyż brak jest zgodności typów poszczególnych argumentów.

Przemienność operatorów

```
struct LZespolona  { // .....................................
      float  re;
      float  im;

 }; // ..........................................................

LZespolona  operator * ( float  liczba,  LZespolona  Z )
{
      Z.re = re * liczba;    Z.im = im * liczba;
      return    Z;
}

int  main( )
{
      LZespolona    lz1,  lz2,  lz3;

      lz3 = operator * ( 3, lz1 );
}
```

Przemienność operatorów

```
struct LZespolona  { // .....................................
      float  re;
      float  im;

 }; // ..........................................................

LZespolona  operator * ( float  liczba,  LZespolona  Z )
{
      Z.re = re * liczba;    Z.im = im * liczba;
      return    Z;
}

int  main( )
{
      LZespolona    lz1,  lz2,  lz3;

      lz3 = 3 * lz1;
}
```

Przemienność operatorów

```
struct LZespolona  { // .....................................
      float  re;
      float  im;

 }; // ..........................................................

LZespolona  operator * ( float   liczba,  LZespolona  Z )
{
      Z.re = re * liczba;     Z.im = im * liczba;
      return     Z;
}

int  main( )
{
      LZespolona    lz1,  lz2,  lz3;
      lz3 = 3 * lz1 * 5;
}
```

A czy teraz wszystko będzie dobrze?

Przemienność operatorów

```
struct LZespolona  { // .....................................
      float  re;
      float  im;

 }; // ..........................................................

LZespolona  operator * ( float   liczba,  LZespolona  Z )
{
      Z.re = re * liczba;     Z.im = im * liczba;
      return     Z;
}

int  main( )
{
      LZespolona    lz1,  lz2,  lz3;
      lz3 = 3 * lz1 * 5;
}
```

Aby wszystko było dobrze należy zapewnić przemienność operacji mnożenia.

Przemienność operatorów

```
struct LZespolona  { // .....................................
    float re;
    float im;

    LZespolona operator * ( float liczba );
}; // ..........................................................

LZespolona operator * ( float liczba, LZespolona Z )
{

    Z.re = re * liczba;    Z.im = im * liczba;
    return    Z;
}

int main( )
{
    LZespolona   lz1, lz2, lz3;

    lz3 = 3 * lz1 * 5;
}
```

Przemienność operatorów

```
struct LZespolona  { // .....................................
    float re;
    float im;

    LZespolona operator * ( float liczba );
}; // ..........................................................

LZespolona LZespolona::operator * ( float liczba )
{ ... }

LZespolona operator * ( float liczba, LZespolona Z )
{ ... }


int main( )
{
    LZespolona  lz1, lz2, lz3;
    lz3 = 3 * lz1 * 5;
}
```

Teraz jest już dobrze.

Przemienność operatorów

```
struct LZespolona  { // .....................................
      float  re;
      float  im;

 }; // ..........................................................

LZespolona  operator * ( LZespolona  Z,  float  liczba )
{ ... }

LZespolona  operator * ( float  liczba,  LZespolona  Z )
{ ... }


int  main( )
{
      LZespolona  lz1,  lz2,  lz3;
      lz3 = 3 * lz1 * 5;
}
```

Teraz jest również dobrze.

Plan prezentacji

Przeciążanie na zewnątrz klasy

Kiedy należy definiować przeciążenie operatora zewnętrznie względem danej klasy?

Przeciążanie na zewnątrz klasy

Kiedy należy definiować przeciążenie operatora zewnętrznie względem danej klasy?

- Gdy pierwszym argumentem jest jednym z wbudowanych typów, np. `int`, `char`, `char*`, itd.
- Gdy pierwszym argumentem jest klasa, której definicja nie może być zmodyfikowana, np. `std::ostream`, `std::istream` itd.
- Gdy przeciążenie ma mieć charakter lokalny, np. ma być *widoczne* tylko w danym module. To znaczy, gdy nie chcemy, aby przeciążenie to było trwałą własnością klasy.

Przeciążanie na zewnątrz klasy

Kiedy należy definiować przeciążenie operatora zewnętrznie względem danej klasy?

- Gdy pierwszym argumentem jest jednym z wbudowanych typów, np. `int`, `char`, `char*`, itd.
- Gdy pierwszym argumentem jest klasa, której definicja nie może być zmodyfikowana, np. `std::ostream`, `std::istream` itd.
- Gdy przeciążenie ma mieć charakter lokalny, np. ma być *widoczne* tylko w danym module. To znaczy, gdy nie chcemy, aby przeciążenie to było trwałą własnością klasy.

Przeciążanie na zewnątrz klasy

Kiedy należy definiować przeciążenie operatora zewnętrznie względem danej klasy?

- Gdy pierwszym argumentem jest jednym z wbudowanych typów, np. `int`, `char`, `char*`, itd.
- Gdy pierwszym argumentem jest klasa, której definicja nie może być zmodyfikowana, np. `std::ostream`, `std::istream` itd.
- Gdy przeciążenie ma mieć charakter lokalny, np. ma być *widoczne* tylko w danym module. To znaczy, gdy nie chcemy, aby przeciążenie to było trwałą własnością klasy.

Przeciążanie na zewnątrz klasy

Kiedy należy definiować przeciążenie operatora zewnętrznie względem danej klasy?

- Gdy pierwszym argumentem jest jednym z wbudowanych typów, np. `int`, `char`, `char*`, itd.
- Gdy pierwszym argumentem jest klasa, której definicja nie może być zmodyfikowana, np. `std::ostream`, `std::istream` itd.
- Gdy przeciążenie ma mieć charakter lokalny, np. ma być *widoczne* tylko w danym module. To znaczy, gdy nie chcemy, aby przeciążenie to było trwałą własnością klasy.

Przeciążanie wewnątrz klasy

Kiedy należy definiować przeciążenie operatora jako *metodę* klasy?

Przeciążanie wewnątrz klasy

Kiedy należy definiować przeciążenie operatora jako *metodę* klasy?

- Gdy przeciążenie ma należeć do trwałych własności klasy.

  Zdefiniowanie przeciążenia w klasie poprawia czytelność dokumentacji kodu, jak też samego kodu.

Przeciążanie wewnątrz klasy

Kiedy należy definiować przeciążenie operatora jako *metodę* klasy?

- Gdy przeciążenie ma należeć do trwałych własności klasy.

  Zdefiniowanie przeciążenia w klasie poprawia czytelność dokumentacji kodu, jak też samego kodu.

Plan prezentacji

Operacje wyświetlania dla własnej struktury

```
struct   LiczbaZespolona {
        float  re, im;
 };

ostream & operator << ( ostream & StrmWy, LiczbaZespolona Lz )
{
     return   StrmWy <<  Lz.re <<  showpos <<  Lz.im <<  noshowpos <<  'i';
}

int  main( )
{
     LiczbaZespolona LZesp;
     LZesp.re = 2;     LZesp.im = 5;

     cout <<  "Liczba zespolona: " << LZesp << endl;
}
```

Dla części urojonej wymuszone zostaje uwidocznienie znaku liczby.

Operacje wyświetlania dla własnej struktury

```
struct   LiczbaZespolona {
        float  re, im;
 };

ostream & operator << ( ostream & StrmWy, LiczbaZespolona Lz )
{
     return   StrmWy <<  Lz.re <<  showpos <<  Lz.im <<  noshowpos <<  'i';
}

int  main( )
{
     LiczbaZespolona LZesp;
     LZesp.re = 2;     LZesp.im = 5;

     cout <<  "Liczba zespolona: " << LZesp << endl;
}
```

Wyświetlona wartość:
2+5i

Dla części urojonej wymuszone zostaje uwidocznienie znaku liczby.

Operacje zapisu do pliku

```
struct   LiczbaZespolona {
        float  re, im;
 };

ostream & operator << ( ostream & StrmWy, LiczbaZespolona Lz )
{
     return   StrmWy <<  Lz.re <<  showpos <<  Lz.im <<  noshowpos <<  'i';
}

int  main( )
{
     LiczbaZespolona   LZesp;
     ofstream          PlikWy;

     PlikWy.open("nowy_test.txt");
     if (!PlikWy.is_open()) return 1;
     LZesp.re = 2;    LZesp.im = 5;

     PlikWy <<  "Liczba zespolona: " << LZesp << endl;
     if (PlikWy.fail())  cerr << "Blad operacji zapisu\n";
     PlikWy.close();
}
```

W analogiczny sposób możemy daną liczbę zapisać do pliku.
Pamiętać jedynie należy aby dołączyć plik nagłówkowy `fstream`.

Operacje wyświetlania dla własnej struktury

```
struct   LiczbaZespolona {
        float  re, im;
 };

ostream & operator << ( ostream & StrmWy, LiczbaZespolona Lz )
{
     return   StrmWy <<  Lz.re <<  showpos <<  Lz.im <<  noshowpos <<  'i';
}

int  main( )
{
     LiczbaZespolona LZesp;
     LZesp.re = 2;     LZesp.im = 5;

     cout <<  "Liczba zespolona: " << LZesp << endl;
}
```

Wyświetlona wartość:
2+5i

Czy w tym przypadku jedynym możliwym sposobem przekazania parametru jest przekazanie go przez wartość?

Operacje wyświetlania dla własnej struktury

```
struct   LiczbaZespolona {
        float  re, im;
 };

ostream & operator << ( ostream & StrmWy, const LiczbaZespolona &Lz )
{
     return   StrmWy <<  Lz.re <<  showpos <<  Lz.im <<  noshowpos <<  'i';
}

int  main( )
{
     LiczbaZespolona LZesp;
     LZesp.re = 2;     LZesp.im = 5;

     cout <<  "Liczba zespolona: " << LZesp << endl;
}
```

Wyświetlona wartość:
2+5i

Można go również przekazać przez referencję do obiektu stałego. W tym konkretnym przypadku jest to bardziej właściwe.

Plan prezentacji

Równoważność definicji `struct` i `class`

```
struct LZespolona  { //  ..........................................
    double  re;
    double  im;

    double  Modul2( ) const  { return  re*re + im*im;  }
    double  Modul( )  const  { return  sqrt(Modul2( ));  }

    LZespolona  operator + (LZespolona  Z2) const ;
}; //  ..........................................................
```

Równoważność definicji `struct` i `class`

```
struct LZespolona  { //  ..........................................
    double re;
    double im;

    double Modul2( ) const  { return re*re + im*im; }
    double Modul( ) const  { return sqrt(Modul2( )); }

    LZespolona operator + (LZespolona Z2) const ;
}; //  ..........................................................
```

```
class LZespolona  { //  ..........................................
  public:
    double re;
    double im;

    double Modul2( ) const  { return re*re + im*im; }
    double Modul( ) const  { return sqrt(Modul2( )); }

    LZespolona operator + (LZespolona Z2) const ;
}; //  ..........................................................
```

Równoważność definicji `struct` i `class`

struct ***LZespolona*** { // .....

...

}; // .....

---

class ***LZespolona*** { // .....

public:

...

}; // .....

Równoważność definicji `struct` i `class`

struct ***LZespolona*** { // .....

...

}; // .....

---

class ***LZespolona*** { // .....

public:

...

}; // .....

Plan prezentacji

Podstawowe własności

W języku *C++* istnieją trzy rodzaje definicji struktur złożonych rozpoczynających się od słów kluczowych: **class** , **struct** , **union** . Dwie pierwsze definiują klasę. Jednak nie każda z nich może zawierać wszystkie dostępne własności klasy. Poprzez **union** definiowana jest struktura specjalna (analogicznie jak w języku *C*).

```
class NazwaKlasy {
  // Domyślnie prywatna
        ...
  protected:
        ...
  private:
        ...
  public:
        ...
};
```

```
struct NazwaKlasy {
  // Domyślnie publiczna
        ...
  private:
        ...
  protected:
        ...
  public:
        ...
};
```

```
union NazwaKlasy {
  // Domyślnie publiczna
        ...
  private:
        ...
  protected:
        ...
  public:
        ...
};
```

Tryby dostępu

Sekcje **private**, **protected**, **public**; definiują tryb dostępu do poszczególnych elementów sekcji w niej zawartych, tzn. pól i metod (w tym również konstruktora i destruktora). Ich porządek występowania w definicji nie jest ustalony. Mogą występować wielokrotnie i wzajemnie się przeplatać.

```
class NazwaKlasy {
  // Domyślnie prywatna
        ...
  protected:
        ...
  private:
        ...
  public:
        ...
  private:
        ...
};
```

private – dostęp i bezpośrednie odwołanie się do pól i metod tej sekcji możliwy jest tylko z poziomu metod klasy, w której deklarowane i definiowane są te pola i metody.

protected – dostęp i bezpośrednie odwołanie się do pól i metod tej sekcji możliwy jest z poziomu metod klasy, w której deklarowane i definiowane są te pola i metody. Odwołanie się i bezpośredni dostęp możliwy jest również w klasie pochodnej.

public – nie ma żadnego ograniczenia na dostęp do pól i metod z tej sekcji.

Definicja klasy z wykorzystaniem słowa kluczowego *class*

class – definiuje klasę, która może posiadać wszystkie możliwe atrybuty klasy dostępne w języku *C++*, tzn. może posiadać metody, konstruktory, destruktory, metody wirtualne, pola i metody statyczne itd. Pozwala na dziedziczenie innych struktur i sama może być dziedziczona. Oprócz tego klasę definiowaną poprzez **class** można parametryzować tworząc w ten sposób szablony.

Domyślną sekcją w definicji klasy jest zawsze sekcja **private**.

```
class NazwaKlasy {
  //Domyślnie prywatna
        . . .
  protected:
        . . .
  private:
        . . .
  public:
        . . .
};
```

Przykład definicji:

```
class LZepolona {
      float re, im;
  public:
      float Re( ) const { return re; }
      float Im( ) const { return im; }
      void Zmien(float r, float i);
};
```

Definicja klasy z wykorzystaniem słowa kluczowego *struct*

struct – definiuje klasę, która może posiadać prawie wszystkie możliwe atrybuty klasy dostępne w języku *C++*, tzn. może posiadać metody, konstruktory, destruktory, metody wirtualne, pola i metody statyczne itd. Pozwala na dziedziczenie innych struktur i sama może być dziedziczona. Jedyne ograniczenie polega na tym, że definiowaną w ten sposób klasę nie można parametryzować, tzn. za jej pomocą nie można tworzyć szablonów.

Domyślną sekcją w definicji klasy jest zawsze sekcja **public**.

```
struct NazwaKlasy {
  //Domyślnie publiczna
        ...
  protected:
        ...
  private:
        ...
  public:
        ...
};
```

Przykład definicji:

```
struct LZepolona {
      float Re( ) const { return re; }
      float Im( ) const { return im; }
      void Zmien(float r, float i);
  private:
      float re, im;
};
```

Definicja klasy z wykorzystaniem słowa kluczowego *union*

union – definiuje strukturę, w której każde pole ma ten sam adres. Unia może mieć pola i metody. Nie może mieć natomiast zarówno pól statycznych, jak też metod statycznych. Nie może również posiadać konstruktorów oraz destruktora. Struktura zdefiniowana przez unię może być dziedziczona, ale nie może dziedziczyć innych struktur. Unia nie może też posiadać metod wirtualnych.

Domyślną sekcją w definicji klasy jest zawsze sekcja **public**.

```
union NazwaKlasy {
  //Domyślnie publiczna
        ...
  protected:
        ...
  private:
        ...
  public:
        ...
};
```

Przykład definicji:

```
union Wartosc {
      int   Absolutna;
      float Wzgledna;
};
```

Dostęp do elementów klasy poza klasą

```
class Macierz2x2 { //  .....
      double _Tab[2][2];
    public :
      double Wez( int i, int j ) const { return _Tab[i][j]; }
      void Zmien( double elem, int i, int j ) { _Tab[ i ][ j ] = elem; }
}; .....

float Wyznacznik( const Macierz2x2 &M )
{
    return M._Tab[0][0]*M._Tab[1][1] – M._Tab[0][1]*M._Tab[1][0];
}
```

Czy jest to poprawne (*i dlaczego nie* ;-) ?

Dostęp do elementów klasy poza klasą

```
class Macierz2x2 { //  .....
      double _Tab[2][2];
    public :
      double Wez( int i, int j ) const { return _Tab[i][j]; }
      void Zmien( double elem, int i, int j ) { _Tab[ i ][ j ] = elem; }
}; .....

float Wyznacznik( const Macierz2x2 &M )
{
    return M._Tab[0][0]*M._Tab[1][1] – M._Tab[0][1]*M._Tab[1][0];
}
```

Poza definicjami metod danej klasy nie można odwoływać się wprost do elementów tej klasy, które znajdują się w sekcji prywatnej. Z tego powodu odwołanie się do pola `_Tab` obiektu `M` w funkcji `Wyznacznik` jest niedopuszczalne.

Dostęp do elementów klasy poza klasą

```
class Macierz2x2 { //  .....
    protected :
        double _Tab[2][2];
    public :
        double Wez( int i, int j ) const { return _Tab[i][j]; }
        void Zmien( double elem, int i, int j ) { _Tab[ i ][ j ] = elem; }
};  .....

float Wyznacznik( const Macierz2x2 &M )
{
    return M._Tab[0][0]*M._Tab[1][1] – M._Tab[0][1]*M._Tab[1][0];
}
```

Jeżeli pole zadeklarowane jest w sekcji **protected**, to bezpośredni dostęp do tego pola poza metodami klasy jest również niemożliwy. Różnice między sekcją **private**, a **protected** pojawiają się dopiero przy dziedziczeniu. Dla pojedynczej klasy, która nie jest dziedziczona, sekcje te mają takie samo znaczenie.

Dostęp do elementów klasy poza klasą

```
class Macierz2x2 { //  .....
      double _Tab[2][2];
    public :
      double Wez( int i, int j ) const { return _Tab[i][j]; }
      void Zmien( double elem, int i, int j ) { _Tab[ i ][ j ] = elem; }
}; .....

float Wyznacznik( const Macierz2x2 &M )
{
    return M.Wez(0,0)*M.Wez(1,1) – M.Wez(0,1)*M.Wez(1,0);
}
```

W sekcji publicznej zdefiniowany jest interfejs do pól klasy ***Macierz2x2***. Pozwala to zachować kontrolę nad sposobem dostępu do tych pól oraz sposobem ich udostępniania. Pozwala to również udostępnić tylko te pola, które chcemy faktycznie udostępnić.

Dostęp do elementów klasy poza klasą

```
class Macierz2x2 { //  .....
      friend   float Wyznacznik( const  Macierz2x2  &M );
       double  _Tab[2][2];
     public :
       double  Wez( int  i, int  j ) const  { return   _Tab[i][j]; }
       void  Zmien( double  elem, int  i, int  j )  { _Tab[ i ][ j ] = elem; }
};  .....

float  Wyznacznik( const Macierz2x2  &M )
{
     return   M._Tab[0][0]*M._Tab[1][1] – M._Tab[0][1]*M._Tab[1][0];
}
```

Pomimo ograniczenia dostępu do pola `_Tab` można uczynić wyjątek dla wybranych funkcji. Pozwala na to deklaracja funkcji zaprzyjaźnionych z daną klasą. Musi ona znaleźć się w obrębie definicji klasy. Miejsce wystąpienia nie jest istotne. Zwyczajowo jednak jest to początek definicji klasy.

Dostęp do elementów klasy poza klasą

```
class Macierz2x2 { //.....
    float _Tab[2][2];
  public :
    float Wez(int i, int j) const
                { return _Tab[ i ][ j ]; }
    void Zmien(float elem, int i, int j);
}; //  ..................................
```

```
class Macierz3x3 { //.....
    float _Elem[3][3];

  public :
    float Wez(int i, int j) const
                { return _Elem[i][j]; }
    void Zmien(float elem, int i, int j);
    void Wstaw(Macierz2x2 M);
}; //  ..................................
```

```
void Macierz3x3::Wstaw( Macierz2x2 M2 )
{
    _Elem[0][0] = M2._Tab[0][0];  _Elem[0][1] = M2._Tab[0][1];
    _Elem[1][0] = M2._Tab[1][0];  _Elem[1][1] = M2._Tab[1][1];
}
```

Dostęp do elementów klasy poza klasą

```
class Macierz2x2 { //.....
    float _Tab[2][2];
  public:
    float Wez(int i, int j) const
                { return _Tab[ i ][ j ]; }
    void Zmien(float elem, int i, int j);
}; //  ..................................
```

```
class Macierz3x3 { //.....
    float _Elem[3][3];

  public:
    float Wez(int i, int j) const
                { return _Elem[i][j]; }
    void Zmien(float elem, int i, int j);
    void Wstaw(Macierz2x2 M);
}; //  ..................................
```

```
void Macierz3x3::Wstaw( Macierz2x2 M2 )
{
    _Elem[0][0] = M2._Tab[0][0];  _Elem[0][1] = M2._Tab[0][1];
    _Elem[1][0] = M2._Tab[1][0];  _Elem[1][1] = M2._Tab[1][1];
}
```

Ze względu na podobieństwo klas tu jest wszystko poprawnie. Czyż nie?

Dostęp do elementów klasy poza klasą

```
class Macierz2x2 { //.....
    float _Tab[2][2];
  public :
    float Wez(int i, int j) const
                { return _Tab[ i ][ j ]; }
    void Zmien(float elem, int i, int j);
}; //  .....
```

```
class Macierz3x3 { //.....
    float _Elem[3][3];

  public :
    float Wez(int i, int j) const
                { return _Elem[i][j]; }
    void Zmien(float elem, int i, int j);
    void Wstaw(Macierz2x2 M);
}; //  .....
```

```
void Macierz3x3::Wstaw( Macierz2x2 M2 )
{
    _Elem[0][0] = M2._Tab[0][0];   _Elem[0][1] = M2._Tab[0][1];
    _Elem[1][0] = M2._Tab[1][0];   _Elem[1][1] = M2._Tab[1][1];
}
```

Podobieństwo tych klas ma charakter subiektywny. Dla kompilatora to są tak samo różne klasy jak każde inne. Dostęp do pól prywatnych danej klasy nie jest możliwy na poziomie zarówno funkcji jak też metody innej klasy.

Dostęp do elementów klasy poza klasą

```
class Macierz2x2 { // ....................
friend                    void    Macierz3x3::Wstaw(Macierz2x2);

    float _Tab[2][2];
  public :
    float Wez(int i, int j) const
                  { return _Tab[ i ][ j ]; }

    void Zmien(float elem, int i, int j);
}; // ....................................
```

```
class Macierz3x3 { // ....................

    float _Elem[3][3];


  public :
    float Wez(int i, int j) const
                  { return _Elem[i][j]; }

    void Zmien(float elem, int i, int j);

    void Wstaw(Macierz2x2 M);
}; // ....................................
```

```
void Macierz3x3::Wstaw( Macierz2x2 M2 )
{
        _Elem[0][0] = M2._Tab[0][0];   _Elem[0][1] = M2._Tab[0][1];
        _Elem[1][0] = M2._Tab[1][0];   _Elem[1][1] = M2._Tab[1][1];
}
```

Dostęp do pól prywatnych staje się możliwy, gdy "zaprzyjaźnimy" daną metodę z klasą, do której pól prywatnych metody tej klasy mają uzyskać dostęp.

Dostęp do elementów klasy poza klasą

```
class Macierz2x2 { //.....
friend void Macierz3x3::Wstaw(Macierz2x2);

    float _Tab[2][2];
  public :
    float Wez(int i, int j) const
                          { return _Tab[i][j]; }

    void Zmien(float elem, int i, int j);
}; //  ..........................................
```

```
class Macierz3x3 { //.....................

    float _Elem[3][3];


  public :
    float Wez(int i, int j) const
                          { return _Elem[i][j]; }

    void Zmien(float elem, int i, int j);
    void Wstaw(Macierz2x2 M);
    void WstawTrans(Macierz2x2 M);
}; //  ..................................
```

Zaprzyjaźnienie jednej metody nie rozwiązuje problemu dostępu do pól prywatnych na poziomie innej metody tej samej klasy.

```
void Macierz3x3::Wstaw( Macierz2x2 M2 )
{
    _Elem[0][0] = M2._Tab[0][0];  _Elem[0][1] = M2._Tab[0][1];
    _Elem[1][0] = M2._Tab[1][0];  _Elem[1][1] = M2._Tab[1][1];
}

void Macierz3x3::WstawTrans( Macierz2x2 M2 )
{
    _Elem[0][0] = M2._Tab[0][0];  _Elem[0][1] = M2._Tab[1][0];
    _Elem[1][0] = M2._Tab[0][1];  _Elem[1][1] = M2._Tab[1][1];
}
```

Dostęp do elementów klasy poza klasą

```
class Macierz2x2 { //.....
friend class Macierz3x3;

    float _Tab[2][2];     public :
    float Wez(int i, int j) const
                          { return _Tab[i][j]; }

    void Zmien(float elem, int i, int j);
}; //  .....
```

```
class Macierz3x3 { //.....

    float _Elem[3][3];


  public :
    float Wez(int i, int j) const
                          { return _Elem[i][j]; }

    void Zmien(float elem, int i, int j);
    void Wstaw(Macierz2x2 M);
    void WstawTrans(Macierz2x2 M);
}; //  .....
```

Zaprzyjaźnienie całej klasy pozwala odwoływać się do pól prywatnych na poziomie metod klasy zaprzyjaźnionej.

```
void Macierz3x3::Wstaw( Macierz2x2 M2 )
{
    _Elem[0][0] = M2._Tab[0][0];  _Elem[0][1] = M2._Tab[0][1];
    _Elem[1][0] = M2._Tab[1][0];  _Elem[1][1] = M2._Tab[1][1];
}

void Macierz3x3::WstawTrans( Macierz2x2 M2 )
{
    _Elem[0][0] = M2._Tab[0][0];  _Elem[0][1] = M2._Tab[1][0];
    _Elem[1][0] = M2._Tab[0][1];  _Elem[1][1] = M2._Tab[1][1];
}
```

Dostęp do elementów klasy poza klasą

```
class Macierz2x2 { // ....................
    float _Tab[2][2];
  public :
    float Wez(int i, int j) const
                        { return _Tab[i][j]; }
    void Zmien(float elem, int i, int j);
}; // ....................................
```

```
class Macierz3x3 { // ....................
    float _Elem[3][3];

  public :
    float Wez(int i, int j) const
                        { return _Elem[i][j]; }
    void Zmien(float elem, int i, int j);
    void Wstaw(Macierz2x2 M);
}; // ....................................
```

```
void Macierz3x3::Wstaw( Macierz2x2 M2 )
{
    _Elem[0][0] = M2.Wez(0,0);  _Elem[0][1] = M2.Wez(0,1);
    _Elem[1][0] = M2.Wez(1,0);  _Elem[1][1] = M2.Wez(1,1);
}
```

Zaprzyjaźnienie z klasą jest ostatecznością. O ile tylko jest to możliwe należy korzystać z metod wchodzących w skład interfejsu danej klasy.

Pojęcie prywatności

```
class Macierz2x2 { //.....
      float _Tab[2][2];
    public :
      float Wez( int i, int j ) const { return _Tab[i][j]; }
      void Zmien( float elem, int i, int j );
      void Wstaw( Macierz2x2 M );
}; //.....

void Macierz2x2::Wstaw( Macierz2x2 M )
{
    _Tab[0][0] = M._Tab[0][0];  _Tab[0][1] = M._Tab[0][1];
    _Tab[1][0] = M._Tab[1][0];  _Tab[1][1] = M._Tab[1][1];
}
```

Czy można na poziomie metody danej klasy odwoływać się do pól prywatnych obiektu tej samej klasy?

Pojęcie prywatności

```
class Macierz2x2 { // ..........................................
      float _Tab[2][2];
    public :
      float Wez( int i, int j ) const { return _Tab[i][j]; }
      void Zmien( float elem, int i, int j );
      void Wstaw( Macierz2x2 M );
}; // ..........................................................

void Macierz2x2::Wstaw( Macierz2x2 M )
{
    _Tab[0][0] = M._Tab[0][0];  _Tab[0][1] = M._Tab[0][1];
    _Tab[1][0] = M._Tab[1][0];  _Tab[1][1] = M._Tab[1][1];
}
```

Prywatność nie jest cechą obiektu. Jest to cecha klasy.

Koniec prezentacji
Dziękuję za uwagę