Rozwijanie w linii, konstruktory, destruktory, szablony

Bogdan Kreczmer

bogdan.kreczmer@pwr.edu.pl

Katedra Cybernetyki i Robotyki
Wydziału Elektroniki
Politechnika Wrocławska

Kurs: Programowanie obiektowe

Niniejszy dokument zawiera materiały do wykładu dotyczącego programowania obiektowego. Jest on udostępniony pod warunkiem wykorzystania wyłącznie do własnych prywatnych potrzeb i może on być kopiowany

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

Definiowanie metody na zewnątrz definicji klasy

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

    void Sprzezenie( );
}; // ..........................................................


LZespolona  LZespolona::Sprzezenie( )
{
    im = –im;
}


int main( )
{
    LZespolona    lz1;

    lz1.Sprzezenie( );
}
```

Definiowanie metody na zewnątrz definicji klasy

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

    void Sprzezenie( );
}; // ..........................................................

LZespolona  LZespolona::Sprzezenie( )
{
    im = –im;
}

int main( )
{
    LZespolona    lz1;

    lz1.Sprzezenie( );
}
```

Definiowanie metody na zewnątrz definicji klasy

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

      void  Sprzezenie( );
 }; // ..........................................................


LZespolona  LZespolona::Sprzezenie( )
{
      im = –im;
}


int  main( )
{
      LZespolona     lz1;

      lz1.Sprzezenie( );
}
```

Definiowanie metody na zewnątrz definicji klasy

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

      void  Sprzezenie( );
 }; // ..........................................................


LZespolona  LZespolona::Sprzezenie( )
{
      im = –im;
}


int  main( )
{
      LZespolona    lz1;

      lz1.Sprzezenie( );
}
```

Definiowanie metody na zewnątrz definicji klasy

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

      void  Sprzezenie( );
 }; // ..................................................................

LZespolona  LZespolona::Sprzezenie( )
{
      im = –im;
}

int  main( )
{
      LZespolona     lz1;

      lz1.Sprzezenie( );
}
```

Definiowanie metod wewnątrz definicji klasy

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

      void  Sprzezenie( )
       {
         im = –im;
       }
}; // ..........................................................
```

```
int  main( )
{
      LZespolona    lz1;

      lz1.Sprzezenie( );
}
```

Definiowanie metod wewnątrz definicji klasy

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

    void Sprzezenie( ) {  im = –im;  }
}; // ..........................................................
```

```
int  main( )
{
    LZespolona    lz1;

    lz1.Sprzezenie( );
}
```

Definiowanie metod wewnątrz definicji klasy

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

    void Sprzezenie( ) {  im = -im;  }
}; // ..........................................................
```

```
int main( )
{
    LZespolona    lz1;

    lz1.Sprzezenie( );
}
```

Definiowanie metod wewnątrz definicji klasy

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

    void Sprzezenie( ) {  im = -im;  }
}; // ..........................................................
```

```
int main( )
{
    LZespolona  lz1;

    lz1.Sprzezenie( );
}
```

Efekt rozwinięcia metody

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

    void Sprzezenie( ) {  im = –im;  }
}; // ..........................................................
```

```
int main( )
{
    LZespolona   lz1;

    {
      LZespolona  &This = lz1;

      This.im = -This.im;
    }
}
```

Efekt rozwinięcia metody

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

    void Przemnoz(double Mnoznik ) {  im *= Mnoznik;  }
}; // ..........................................................


int main( )
{
    LZespolona  lz1;

    lz1.Przemnoz(5.1);
}
```

Efekt rozwinięcia metody

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

    void Przemnoz(double Mnoznik ) {  im *= Mnoznik;  }
}; // ..................................................................


int main( )
{
    LZespolona   lz1;

    {
      LZespolona  &This = lz1;
      double          Mnoznik = 5.1;

      This.im *= Mnoznik;
    }
}
```

Definiowanie metod wewnątrz definicji klasy

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

      void  Sprzezenie( ) {  im = -im;  }

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

```
int  main( )
{
      LZespolona    lz1;

      lz1.Sprzezenie( );
}
```

Definiowanie metod wewnątrz definicji klasy

```
struct LZespolona  { //  ..........................................
    void  Sprzezenie( ) {  im = –im;  }
    LZespolona  operator + ( LZespolona  Z2 )
     {
        Z2.re += re;    Z2.im += im;
     }  return   Z2;

    float  re;
    float  im;
}; //  ..........................................................


int  main( )
{
    LZespolona   lz1;

    lz1.Sprzezenie( );
}
```

Definiowanie metod wewnątrz definicji klasy

```
struct LZespolona  { // ..........................................
    void Sprzezenie( ) {  im = –im;  }
    LZespolona operator + ( LZespolona  Z2 )
     {
        Z2.re += re;    Z2.im += im;
     }  return   Z2;

    float  re;
    float  im;
}; // ..........................................................
```

```
int  main( )
{
    LZespolona   lz1;

    lz1.Sprzezenie( );
}
```

W ciele definicji klasy kolejność definicji pól i metod nie ma znaczenia.

Definiowanie metod wewnątrz definicji klasy

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

      void  Sprzezenie( )  {  im = -im;  }

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


int  main( )
{
      LZespolona    lz1;

      lz1.Sprzezenie( );
}
```

Definiowanie metod wewnątrz definicji klasy

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

      void  Sprzezenie( )  {  im = -im;  }

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

```
int  main( )
{
      LZespolona    lz1;

      lz1.Sprzezenie( );
}
```

Kod metody definiowanej w ciele klasy domyślnie rozwijany jest przez kompilator w miejscu wywołania metody.

Definiowanie metod wewnątrz definicji klasy

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

      void  Sprzezenie( )  {  im = -im;  }

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

```
int  main( )
{
      LZespolona    lz1;

      lz1.Sprzezenie( );
}
```

Domyślne zachowanie kompilatora może zostać jednak zmienione poprzez użycie odpowiednich opcji, np. dla g++ jest to `-fno-default-inline`.

Definiowanie metod wewnątrz definicji klasy

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

    inline void Sprzezenie( ) { im = -im; }

    inline LZespolona operator + ( LZespolona Z2 )
     {
       Z2.re += re;    Z2.im += im;
       return Z2;
     }
}; // ....................................................
```

```
int main( )
{
    LZespolona lz1;

    lz1.Sprzezenie( );
}
```

Chcąc uniezależnić się od ustawień kompilatora można użyć słowa kluczowe `inline`.

Definiowanie metod wewnątrz definicji klasy

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

      inline  void  Sprzezenie( )  {  im = -im;  }

      inline  LZespolona  operator + ( LZespolona  Z2 )
       {
          Z2.re += re;    Z2.im += im;
          return    Z2;
       }
}; // ..........................................................
```

```
int  main( )
{
      LZespolona    lz1;

      lz1.Sprzezenie( );
}
```

Użycie `inline` nie gwarantuje rozwinięcia kodu. Jest ono traktowane jedynie jako zalecenie. Kompilator sam podejmuje decyzję, czy kod można rozwinąć.

Definiowanie metod *inline* na zewnątrz definicji klasy

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

    inline void Sprzezenie( ) { im = -im; }

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

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

int main( )
{
    LZespolona lz1;

    lz1.Sprzezenie( );
}
```

Metody typu *inline* mogą być definiowane również poza ciałem klasy. Jednak wtedy koniecznie należy użyć modyfikatora `inline`.

Plan prezentacji

Funkcje typu *inline*

```
inline double Poteguj( double Wykladnik, unsigned int Potega )
{
    double Wynik = 1;
    for (; Potega; --Potega) Wynik *= Wykladnik;
    return Wynik;
}

int main( )
{
    double Wynik = Poteguj(4,3);
    cout << Wynik << endl;
}
```

Funkcje typu *inline*

```
inline double Poteguj( double Wykladnik, unsigned int Potega )
{
    return Potega == 0 ? 1 : Wykladnik*Poteguj(Wykladnik,Potega – 1);
}
```

```
int main( )
{
    double Wynik = Poteguj(4,3);
    cout << Wynik << endl;
}
```

Funkcje typu *inline*

```
inline double Poteguj( double Wykladnik, unsigned int Potega )
{
    return Potega == 0 ? 1 : Wykladnik*Poteguj(Wykladnik,Potega – 1);
}
```

```
int main( )
{
    double Wynik = Poteguj(4,3);
    cout << Wynik << endl;
}
```

Sposób rozwinięcia kodu zależy od *inteligencji* kompilatora. Dla przedstawionego wywołania może ono mieć postać `4*Poteguj(4,2)`, albo też może ono sprowadzić się do zapisu 64.

Funkcje typu *inline*

modul.cpp

```
inline double Poteguj( double Wykladnik, unsigned int Potega )
{
    return Potega == 0 ? 1 : Wykladnik*Poteguj(Wykladnik,Potega – 1);
}
```

program.cpp

```
int main( )
{
    double Wynik = Poteguj(4,3);
    cout << Wynik << endl;
}
```

Funkcje typu *inline*

modul.cpp

```
inline double Poteguj( double Wykladnik, unsigned int Potega )
{
    return Potega == 0 ? 1 : Wykladnik*Poteguj(Wykladnik,Potega – 1);
}
```

program.cpp

```
inline double Poteguj( double Wykladnik, unsigned int Potega );

int main( )
{
    double Wynik = Poteguj(4,3);
    cout << Wynik << endl;
}
```

Funkcje typu *inline*

modul.cpp

```
inline double Poteguj( double Wykladnik, unsigned int Potega )
{
    return Potega == 0 ? 1 : Wykladnik*Poteguj(Wykladnik,Potega – 1);
}
```

program.cpp

```
inline double Poteguj( double Wykladnik, unsigned int Potega );

int main( )
{
    double Wynik = Poteguj(4,3);
    cout << Wynik << endl;
}
```

Odwołanie do funkcji rozwijanej w linii może być zrealizowane tylko i wyłącznie w tej samej jednostce translacyjnej, w której jest ona zdefiniowana.

Funkcje typu *inline*

modul.cpp

```
inline double Poteguj( double Wykladnik, unsigned int Potega )
{
    return Potega == 0 ? 1 : Wykladnik*Poteguj(Wykladnik,Potega – 1);
}
```

program.cpp

```
inline double Poteguj( double Wykladnik, unsigned int Potega );

int main( )
{
    double Wynik = Poteguj(4,3);
    cout << Wynik << endl;
}
```

Konsolidacja tego kodu zakończy się niepowodzeniem. Linker zgłosi brak kodu funkcji `Poteguj`.

Funkcje typu *inline*

modul.hh

```
inline double Poteguj( double Wykladnik, unsigned int Potega )
{
    return Potega == 0 ? 1 : Wykladnik*Poteguj(Wykladnik,Potega – 1);
}
```

program.cpp

```
#include "modul.hh"

int main( )
{
    double Wynik = Poteguj(4,3);
    cout << Wynik << endl;
}
```

Funkcje typu *inline*

modul.hh

```
inline double Poteguj( double Wykladnik, unsigned int Potega )
{
    return Potega == 0 ? 1 : Wykladnik*Poteguj(Wykladnik,Potega – 1);
}
```

program.cpp

```
#include "modul.hh"

int main( )
{
    double Wynik = Poteguj(4,3);
    cout << Wynik << endl;
}
```

Funkcje rozwijane w linii, które są *eksportowane* do innych modułów muszą być definiowane w plikach nagłówkowych.

Plan prezentacji

Uwarunkowania rozwinięć kodu

- Specyfikator **inline** nie jest dla kompilatora poleceniem rozwinięcia funkcji lub metody w linii wywołania. Pełni on jedynie rolę zalecenia, które powinno być w miarę możliwości uwzględnione.
- Kompilator przy podejmowaniu decyzji o rozwinięciu kodu funkcji może kierować się heurystycznymi ocenami uwzględniającymi, np. rozmiar kodu rozwijanej funkcji i/lub ilość dokonanych rozwinięć w aktualnie kompilowanej funkcji.

Uwarunkowania rozwinięć kodu

- Specyfikator **inline** nie jest dla kompilatora poleceniem rozwinięcia funkcji lub metody w linii wywołania. Pełni on jedynie rolę zalecenia, które powinno być w miarę możliwości uwzględnione.
- Kompilator przy podejmowaniu decyzji o rozwinięciu kodu funkcji może kierować się heurystycznymi ocenami uwzględniającymi, np. rozmiar kodu rozwijanej funkcji i/lub ilość dokonanych rozwinięć w aktualnie kompilowanej funkcji.

Konsekwencje rozwinięcia kodu

- Rozwinięcie funkcji nie zawsze musi oznaczać zwiększenie rozmiaru kodu (aczkolwiek zazwyczaj tak jest).
- Stosowanie częstych rozwinięć funkcji nie zawsze musi oznaczać przyśpieszenie wykonywania kodu (aczkolwiek zazwyczaj tak jest).
- Specyfikator **inline** nie wpływa na znaczenie funkcji; funkcja rozwijana w miejscu wywołania nadal ma unikatowy adres.

Konsekwencje rozwinięcia kodu

- Rozwinięcie funkcji nie zawsze musi oznaczać zwiększenie rozmiaru kodu (aczkolwiek zazwyczaj tak jest).
- Stosowanie częstych rozwinięć funkcji nie zawsze musi oznaczać przyśpieszenie wykonywania kodu (aczkolwiek zazwyczaj tak jest).
- Specyfikator **inline** nie wpływa na znaczenie funkcji; funkcja rozwijana w miejscu wywołania nadal ma unikatowy adres.

Konsekwencje rozwinięcia kodu

- Rozwinięcie funkcji nie zawsze musi oznaczać zwiększenie rozmiaru kodu (aczkolwiek zazwyczaj tak jest).
- Stosowanie częstych rozwinięć funkcji nie zawsze musi oznaczać przyśpieszenie wykonywania kodu (aczkolwiek zazwyczaj tak jest).
- Specyfikator **inline** nie wpływa na znaczenie funkcji; funkcja rozwijana w miejscu wywołania nadal ma unikatowy adres.

Plan prezentacji

Inicjalizacja obiektów

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

struct Wektor { float x, y; };


int main()
{
  Wektor W;

  cout << W.x << ", " << W.y << endl;
}
```

Kto zgadnie co ten program wyświetli? >8-|

Inicjalizacja obiektów

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

struct Wektor { float x, y; };


int main()
{
  Wektor W;

  cout << W.x << ", " << W.y << endl;
}
```

Jedno jest pewne. Tego nikt nie zgadnie. ;-)

Inicjalizacja obiektów

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

struct Wektor { // .....
    float x, y;

    Wektor() { x = 0; y = 0; }    // ..... Ta metoda inicjalizuje obiekt.
}; // .....


int main()
{
    Wektor W;

    cout << W.x << ", " << W.y << endl;
}
```

Teraz sposób inicjalizacji jest już jednoznacznie określony.

Inicjalizacja obiektów

```
. . .
struct Wektor { // ..................................................
    float x, y;

    Wektor() { x = 0; y = 0; }
    Wektor( float x_nowa, float y_nowa) { x = x_nowa; y = y_nowa; }
}; // ..................................................

int main()
{
    Wektor W(4,5);      // ..... Inicjalizujemy obiekt tak jak chcemy.

    cout << W.x << ", " << W.y << endl;
}
```

Konstruktory pozwalają na wybór sposobu inicjalizacji obiektu.

Inicjalizacja obiektów

```
struct Wektor { //.....
    float x, y;

    Wektor( float x_nowa, float y_nowa) { x = x_nowa; y = y_nowa; }
}; //.....


int main()
{
    Wektor V;      //.....Taka definicja jest teraz zabroniona.
    Wektor W(4,5);
      ...
}
```

Definicja konstruktora unieważnia niejawny konstruktor bezparametryczny.

Inicjalizacja obiektów

```
. . .
struct Wektor { // ..................................................
        float x, y;

        Wektor( float x_nowa, float y_nowa) { x = x_nowa; y = y_nowa; }
      private :
        Wektor() { x = y = 0; }
}; // ..................................................................

int main()
{
      Wektor ~~V;~~    // ..... Ponownie definicja ta staje się niedopuszczalna.
      Wektor W(4,5);
        . . .
}
```

Sekcje *private* i *protected* pozwalają zabronić korzystania z wybranych konstruktorów na poziomie publicznym.

Inicjalizacja obiektów

```
struct Wektor {
      float x, y;
      Wektor Dodaj(const Wektor & Arg) const ;
   private :
      Wektor() { x = y = 0; }  // ..... W ten spsób niemożliwe jest definiowanie obiektu poza daną klasą.
};

Wektor Wektor ::Dodaj( const Wektor & Arg) const
{
   Wektor   Suma;      // ..... Tutaj można korzystać z prywatnego konstruktora.
   Suma.x = x + Arg.x;   Suma.y = y + Arg.y;
   return   Suma;
}
```

Sekcje *private* i *protected* nie ograniczają dostępu do konstruktorów oraz innych metod i pól na poziomie metod danej klasy.

Inicjalizacja tablic

```
. . .
struct Wektor {   //.....................................................
        float    x, y;

        Wektor() { x = y = 0; }
        Wektor( float  x_nowa, float  y_nowa ) { x = x_nowa;  y = y_nowa; }
};   //.....................................................

. . .
Wektor    Tab1[3];
```

Przy definiowaniu tablic domyślnie używany jest konstruktor bezparametryczny.

Inicjalizacja tablic

```
. . .
struct Wektor {   //.....................................................
        float    x, y;

        Wektor() { x = y = 0; }
        Wektor( float  x_nowa, float  y_nowa ) { x = x_nowa;  y = y_nowa; }
};   //.....................................................

. . .
Wektor    Tab1[3];
Wektor    Tab2[3] = { Wektor(1,1), Wektor(2,1), Wektor(4,2) };
```

Przy definiowaniu tablic domyślnie używany jest konstruktor bezparametryczny.

Inicjalizacja tablic

```
. . .
struct Wektor {   //.....................................................
        float    x, y;

        Wektor() { x = y = 0; }
        Wektor( float  x_nowa, float  y_nowa ) { x = x_nowa;  y = y_nowa; }
};   //.....................................................

. . .
Wektor    Tab1[3];
Wektor    Tab2[3] = { Wektor(1,1), Wektor(2,1), Wektor(4,2) };
Wektor    Tab3[3] = { Wektor(2,2) };
```

Przy definiowaniu tablic domyślnie używany jest konstruktor bezparametryczny.

Inicjalizacja tablic

```
. . .
struct Wektor {   //.....................................................
        float    x, y;

        Wektor() { x = y = 0; }
        Wektor( float  x_nowa, float  y_nowa ) { x = x_nowa;  y = y_nowa; }
};   //.....................................................

. . .
Wektor    Tab1[3];
Wektor    Tab2[3] = { Wektor(1,1), Wektor(2,1), Wektor(4,2) };
Wektor    Tab3[3] = { Wektor(2,2) };
Wektor    Tab4[3] = { Wektor(2,2), Wektor(), Wektor(5,5) };
```

Przy definiowaniu tablic domyślnie używany jest konstruktor bezparametryczny.

Wykorzystanie konstruktorów

```
struct Wektor {    //.....
        float     x, y;

        Wektor( )  { x = y = 0; }
        Wektor( float  x_nowa, float  y_nowa )  { x = x_nowa; y = y_nowa; }
};    //.....

Wektor Dodaj1( Wektor  W1, Wektor  W2 )
  {   W1.x += W2.x;   W1.y += W2.y;   return  W1; }

Wektor Dodaj2( Wektor  W1, Wektor  W2 )
  {
    Wektor  W(W1.x+W2.x, W1.y+W2.y);
    return  W;
  }

Wektor Dodaj3( Wektor  W1, Wektor  W2 )
  {   return  Wektor(W1.x+W2.x, W1.y+W2.y); }
```

Konstruktory pozwalają także na bardziej zwięzły zapis i bardziej efektywną implementację funkcji i metod.

Wykorzystanie konstruktorów

```
struct Wektor {    //.....
        float     x, y;

        Wektor( )  { x = y = 0; }
        Wektor( float  x_nowa, float  y_nowa )  { x = x_nowa; y = y_nowa; }
};    //.....

Wektor Dodaj1( Wektor  W1, Wektor  W2 )
  {   W1.x += W2.x;   W1.y += W2.y;   return  W1; }

Wektor Dodaj2( Wektor  W1, Wektor  W2 )
  {
    Wektor  W(W1.x+W2.x, W1.y+W2.y);
    return  W;
  }

Wektor Dodaj3( Wektor  W1, Wektor  W2 )
  {   return  Wektor(W1.x+W2.x, W1.y+W2.y); }
```

Konstruktory pozwalają także na bardziej zwięzły zapis i bardziej efektywną implementację funkcji i metod.

Wykorzystanie konstruktorów

```
struct Wektor {    //.....
        float     x, y;

        Wektor( )  { x = y = 0; }
        Wektor( float  x_nowa, float  y_nowa )  { x = x_nowa; y = y_nowa; }
};    //.....

Wektor Dodaj1( Wektor  W1, Wektor  W2 )
  {   W1.x += W2.x;   W1.y += W2.y;   return  W1; }

Wektor Dodaj2( Wektor  W1, Wektor  W2 )
  {
    Wektor  W(W1.x+W2.x, W1.y+W2.y);
    return  W;
  }

Wektor Dodaj3( Wektor  W1, Wektor  W2 )
  {   return  Wektor(W1.x+W2.x, W1.y+W2.y); }
```

Konstruktory pozwalają także na bardziej zwięzły zapis i bardziej efektywną implementację funkcji i metod.

Wykorzystanie konstruktorów

```
struct Wektor {    //.....
        float    x, y;

        Wektor( )  { x = y = 0; }
        Wektor( float  x_nowa, float  y_nowa )  { x = x_nowa; y = y_nowa; }
};    //.....

Wektor Dodaj1( Wektor  W1, Wektor  W2 )
  {   W1.x += W2.x;   W1.y += W2.y;   return  W1; }

Wektor Dodaj2( Wektor  W1, Wektor  W2 )
  {
    Wektor  W(W1.x+W2.x, W1.y+W2.y);
    return  W;
  }

Wektor Dodaj3( Wektor  W1, Wektor  W2 )
  {   return  Wektor(W1.x+W2.x, W1.y+W2.y); }
```

Konstruktory pozwalają także na bardziej zwięzły zapis i bardziej efektywną implementację funkcji i metod.

Konstruktory i destruktory

```
struct PustaKlasa { //.....

        PustaKlasa( ) { cout << "++PustaKlasa" << endl; }
        ~PustaKlasa( ) { cout << "--PustaKlasa" << endl; }
}; //.....


int  main( )
{
      cout <<  "Witajcie !!! :-)" <<  endl;
      PustaKlasa  Ob;
      cout <<  "Zegnajcie !!! :-(" <<  endl;
}
```

Wynik działania

```
Witajcie !!! :-)
++PustaKlasa
Zegnajcie !!! :-(
--PustaKlasa
```

W przypadku zmiennych lokalnych (automatycznych) konstruktor wywoływany jest w chwili tworzenia obiektu, tj. w miejscu jego definicji. Destruktor natomiast jest wywoływany w miejscu końca zakresu ważności definicji obiektu.

Plan prezentacji

Dostęp do pól klasy Wektor

```
struct Wektor { // .....................................................
    float    x, y, z;


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

```
int  main( )
{
    ...
    Liczba = V.x;
}
```

Dostęp do pól klasy Wektor

```
struct Wektor { // ..........
    float    x, y, z;


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

```
int  main( )
{
    ...
    Liczba = V[0];
}
```

Odczyt pola poprzez operator indeksujący

```
struct Wektor { // .....................................................
    float    x, y, z;
    float    operator[ ] (int Ind) const { return  Ind == 0 ? x : Ind == 1 ? y : z; }

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

```
int  main( )
{
    ...
    Liczba = V[0];
}
```

Odczyt pola poprzez operator indeksujący

```
struct Wektor { //.....
   float    x, y, z;
   float   operator[ ] (int Ind) const { return  Ind == 0 ? x : Ind == 1 ? y : z; }

}; //.....

float  IloczynSkal( const Wektor& V1, const Wektor& V2 )
 {
   float   Wynik = 0;
   for  (int Ind = 0; Ind < 3; ++Ind)  Wynik += V1[Ind]*V2[Ind];
   return  Wynik;
 }

int  main( )
{
   ...
   Liczba = IloczynSkal(W1,W2);
}
```

Odczyt pola poprzez operator indeksujący

```
struct Wektor { // ..........................................................
    float    x, y, z;
    float    operator[ ] (int Ind) const { return  Ind == 0 ? x : Ind == 1 ? y : z; }

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

```
int  main( )
{
    ...
    W[0] = Liczba;
}
```

Modyfikacja pól poprzez operator indeksujący

```
struct Wektor { //.....
    float    x, y, z;
    float   operator[ ] (int Ind) const { return  Ind == 0 ? x : Ind == 1 ? y : z; }
    float& operator[ ] (int Ind)       { return  Ind == 0 ? x : Ind == 1 ? y : z; }
}; //.....
```

```
int  main( )
{
    ...
    W[0] = Liczba;
}
```

Modyfikacja pól poprzez operator indeksujący

```
struct Wektor { //.....
   float    x, y, z;
   float   operator[ ] (int Ind) const { return  Ind == 0 ? x : Ind == 1 ? y : z; }
   float& operator[ ] (int Ind)       { return  Ind == 0 ? x : Ind == 1 ? y : z; }
}; //.....

void  Odbicie( Wektor& V )
 {
   for  (int Ind = 0; Ind < 3; ++Ind)  V[Ind] = -V[Ind];
 }


int  main( )
{
   ...
   Odbicie( W );
}
```

Modelowanie wektora z wykorzystaniem tablicy

```
struct Wektor { //.....
   float   _Wsp[3];
   float   operator[ ] (int Ind) const { return  _Wsp[Ind]; }
   float& operator[ ] (int Ind)        { return  _Wsp[Ind]; }
}; //.....
```

```
int  main( )
{
   ...
   Liczba = IloczynSkal(W1,W2);
}
```

Modelowanie wektora z wykorzystaniem tablicy

```
struct Wektor { //.....
   float  _Wsp[3];
   float  operator[ ] (int Ind) const { return  _Wsp[Ind]; }
   float& operator[ ] (int Ind)       { return  _Wsp[Ind]; }
}; //.....

float IloczynSkal( const Wektor& V1, const Wektor& V2 )
 {
   float  Wynik = 0;
   for  (int Ind = 0; Ind < 3; ++Ind)  Wynik += V1[Ind]*V2[Ind];
   return  Wynik;
 }

int  main( )
{
   ...
   Liczba = IloczynSkal(W1,W2);
}
```

Modelowanie wektora z wykorzystaniem tablicy

```
struct Wektor { //.....
   float  _Wsp[3];
   float  operator[ ] (int Ind) const { return _Wsp[Ind]; }
   float& operator[ ] (int Ind)       { return _Wsp[Ind]; }
}; //.....

float operator & ( const Wektor& V1, const Wektor& V2 )
 {
   float  Wynik = 0;
   for (int Ind = 0; Ind < 3; ++Ind) Wynik += V1[Ind]*V2[Ind];
   return Wynik;
 }

int main( )
{
   ...
   Liczba = W1 & W2;
}
```

Modelowanie wektora z wykorzystaniem tablicy

```
struct Wektor { //.....
   float  _Wsp[3];
   float  operator[ ] (int Ind) const { return  _Wsp[Ind]; }
   float& operator[ ] (int Ind)       { return  _Wsp[Ind]; }
   float operator  & (const Wektor& V2) const
}; //.....

float  Wektor::operator  & ( const Wektor& V2 ) const
 {
   float  Wynik = 0;
   for  (int Ind = 0; Ind < 3; ++Ind)  Wynik += _Wsp[Ind]*V2[Ind];
   return  Wynik;
 }
```

Modelowanie wektora z wykorzystaniem tablicy

```
struct Wektor { //.....
   float  _Wsp[3];
   float  operator[ ] (int Ind) const { return _Wsp[Ind]; }
   float& operator[ ] (int Ind)       { return _Wsp[Ind]; }
   float operator & (const Wektor& V2) const
}; //.....

float Wektor::operator & ( const Wektor& V2 ) const
 {
   float  Wynik = 0;
   int    Ind = -1;
   for (float WspWektora : _Wsp) Wynik += WspWektora*V2[++Ind];
   return Wynik;
 }
```

Plan prezentacji

Jak należy to interpretować

Zwykła pętla **for**

```
double   Tab[5];
int      Idx;

for ( Idx = 0; Idx < 5; ++Idx) {

   cout << Tab[Idx] << endl;

}
```

Pętla **for** z wykorzystaniem zakresu tablicy

Jak należy to interpretować

Zwykła pętla **for**

```
double   Tab[5];
int      Idx;

for ( Idx = 0; Idx < 5; ++Idx) {

   cout << Tab[Idx] << endl;

}
```

Pętla **for** z wykorzystaniem zakresu tablicy

```
double   Tab[5];

for ( double   ElemTab : Tab) {

   cout << ElemTab << endl;

}
```

Jak należy to interpretować

Zwykła pętla **for**

```
double Tab[5];

for (int Idx = 0; Idx < 5; ++Idx) {
  cout << Tab[Idx] << endl;
}
```

Pętla **for** z wykorzystaniem zakresu tablicy

```
double Tab[5];

for ( double ElemTab : Tab) {
  cout << ElemTab << endl;
}
```

Jak należy to interpretować

Zwykła pętla **for**

```
double   Tab[5];

for (int Idx = 0; Idx < 5; ++Idx) {
   double   ElemTab = Tab[Idx];

   cout << ElemTab << endl;
}
```

Pętla **for** z wykorzystaniem zakresu tablicy

```
double   Tab[5];

for ( double   ElemTab : Tab) {

   cout << ElemTab << endl;

}
```

Jak należy to interpretować

Zwykła pętla **for**

```
double   Tab[5];

for (int Idx = 0; Idx < 5; ++Idx) {

   Tab[Idx] = 2*Tab[Idx]+1;

}
```

Pętla **for** z wykorzystaniem zakresu tablicy

```
double   Tab[5];

for ( double  &ElemTab : Tab) {

   ElemTab = 2*ElemTab + 1;

}
```

Jak należy to interpretować

Zwykła pętla **for**

```
double   Tab[5];

for (int Idx = 0; Idx < 5; ++Idx) {
   double  & ElemTab = Tab[Idx];

   ElemTab = 2*ElemTab + 1;
}
```

Pętla **for** z wykorzystaniem zakresu tablicy

```
double   Tab[5];

for ( double  &ElemTab : Tab) {

   ElemTab = 2*ElemTab + 1;

}
```

Jak należy to interpretować

Zwykła pętla **for**

```
Wektor Tab[5];

for (int Idx = 0; Idx < 5; ++Idx) {
  Wektor Wek = Tab[Idx];

  cout << Wek[0]
       << Wek[1] << endl;
}
```

Pętla **for** z wykorzystaniem zakresu tablicy

```
Wektor Tab[5];

for (Wektor Wek : Tab) {

  cout << Wek[0]
       << Wek[1] << endl;

}
```

Jak należy to interpretować

Zwykła pętla **for**

```
Wektor  Tab[5];

for (int Idx = 0; Idx < 5; ++Idx) {
   Wektor  &Wek = Tab[Idx];

  cout << Wek[0]
       << Wek[1] << endl;
}
```

Pętla **for** z wykorzystaniem zakresu tablicy

```
Wektor  Tab[5];

for (Wektor  &Wek : Tab) {

  cout << Wek[0]
       << Wek[1] << endl;

}
```

Jak należy to interpretować

Zwykła pętla **for**

```
Wektor Tab[5];

for (int Idx = 0; Idx < 5; ++Idx) {
  const Wektor &Wek = Tab[Idx];

  cout << Wek[0]
       << Wek[1] << endl;
}
```

Pętla **for** z wykorzystaniem zakresu tablicy

```
Wektor Tab[5];

for (const Wektor &Wek : Tab) {

  cout << Wek[0]
       << Wek[1] << endl;

}
```

Plan prezentacji

Dostęp do pól klasy Wektor

```
struct Wektor { //.....
    float    _Wsp[3];
    float    operator[ ] (int Ind) const { return  _Wsp[Ind]; }
    float&   operator[ ] (int Ind)       { return  _Wsp[Ind]; }
}; //.....
```

```
int  main( )
{
    Wektor   W;
    float    Liczba;
    Liczba = W[0];  W[1] = 3;;
}
```

Dostęp do pól klasy Wektor

```
struct Wektor { //.....
   float  _Wsp[3];
   float  operator( ) (int Ind) const { return _Wsp[Ind]; }
   float& operator( ) (int Ind)       { return _Wsp[Ind]; }
}; //.....
```

```
int main( )
{
   Wektor W;
   float  Liczba;
   Liczba = W(0);  W(1) = 3;
}
```

Dostęp do pól klasy Wektor

```
struct Wektor { //.....
   float    _Wsp[3];
   float   operator( ) (int Ind) const { return  _Wsp[Ind]; }
   float& operator( ) (int Ind)        { return  _Wsp[Ind]; }
}; //.....

void  Odbicie( Wektor& V )
 {
   for  (int Ind = 0; Ind < 3; ++Ind)  V(Ind) = -V(Ind);
 }

int  main( )
{
   Wektor  W;
   float      Liczba;
   Liczba = W(0);   W(1) = 3;
}
```

Dostęp do pól klasy Wektor

```
struct Wektor { //.....
    float    _Wsp[3];
    float    operator( ) (int Ind) const { return  _Wsp[Ind]; }
    float&   operator( ) (int Ind)       { return  _Wsp[Ind]; }
    void  Odbicie( );
}; //.....

void  Wektor::Odbicie( )
 {
   for  (float  &WspWek : _Wsp)  WspWek = -WspWek;
 }
```

Dostęp do pól klasy Macierz – operator funkcyjny

```
struct Macierz { //.....
    float  _Tab[3][3];


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

```
int main( )
{
   Macierz  M;
   float    Liczba;
   Liczba = M._Tab[0][0];   M._Tab[0][2] = 3;
}
```

Dostęp do pól klasy Macierz – operator funkcyjny

```
struct Macierz { //.....
    float    _Tab[3][3];
    float    operator( ) (int Wie, int Kol) const { return  _Tab[Wie][Kol]; }
    float&   operator( ) (int Wie, int Kol)       { return  _Tab[Wie][Kol]; }
}; //.....
```

```
int  main( )
{
   Macierz   M;
   float     Liczba;
   Liczba = M._Tab[0][0];   M._Tab[0][2] = 3;
}
```

Dostęp do pól klasy Macierz – operator funkcyjny

```
struct Macierz { //.....
    float    _Tab[3][3];
    float    operator( ) (int Wie, int Kol) const { return  _Tab[Wie][Kol]; }
    float&   operator( ) (int Wie, int Kol)       { return  _Tab[Wie][Kol]; }
}; //.....
```

```
int  main( )
{
   Macierz   M;
   float     Liczba;
   Liczba = M(0,0);   M(0,2) = 3;
}
```

Dostęp do pól klasy Macierz – operator indeksujący

```
struct Macierz { //.....
    float _Tab[3][3];


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

```
int main( )
{
   Macierz M;
   float Liczba;
   Liczba = M[0][0];   M[0][2] = 3;
}
```

Dostęp do pól klasy Macierz – operator indeksujący

```
struct Macierz { //.....
    float _Tab[3][3];
    const float* operator[ ] (int Wie) const { return _Tab[Wie]; }
          float* operator[ ] (int Wie)       { return _Tab[Wie]; }
}; //.....
```

```
int main( )
{
   Macierz M;
   float Liczba;
   Liczba = M[0][0];   M[0][2] = 3;
}
```

Dostęp do pól klasy Macierz – operator indeksujący

```
struct Macierz { //.....
    float _Tab[3][3];


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

```
int main( )
{
   Macierz M;
   float Liczba;
   Liczba = M[0][0];   M[0][2] = 3;
}
```

Dostęp do pól klasy Macierz – operator indeksujący

```
struct Macierz { //.....
    Wektor _Wiersz[3];


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

```
int main( )
{
   Macierz M;
   float Liczba;
   Liczba = M[0][0];   M[0][2] = 3;
}
```

Dostęp do pól klasy Macierz – operator indeksujący

```
struct Macierz { //.....
    Wektor  _Wiersz[3];
    const Wektor&  operator[ ] (int Wie) const { return _Wiersz[Wie]; }
          Wektor&  operator[ ] (int Wie)       { return _Wiersz[Wie]; }
}; //.....
```

```
int main( )
{
   Macierz  M;
   float    Liczba;
   Liczba = M[0][0];   M[0][2] = 3;
}
```

Macierz – reprezentacja kolumn

```
struct Macierz { //.....
    Wektor _Kolumna[3];




}; //.....


int main( )
{
   Macierz M;
   Wektor Kol_i;
   float Liczba;


}
```

Macierz – reprezentacja kolumn

```
struct Macierz { //.....
    Wektor _Kolumna[3];

  const Wektor& operator[ ] (int Kol) const { return _Kolumna[Kol]; }
        Wektor& operator[ ] (int Kol)       { return _Kolumna[Kol]; }


}; //.....


int main( )
{
   Macierz M;
   Wektor Kol_i;
   float Liczba;
   Kol_i = M[0];  M[2] = Kol_i;

}
```

Macierz – reprezentacja kolumn

```
struct Macierz { //.....
    Wektor _Kolumna[3];

  const Wektor& operator[ ] (int Kol) const { return _Kolumna[Kol]; }
        Wektor& operator[ ] (int Kol)       { return _Kolumna[Kol]; }

  const float operator( ) (int Wi, int Ko) const {return _Kolumna[Ko][Wi];}
        float& operator( ) (int Wi, int Ko)       {return _Kolumna[Ko][Wi];}
}; //.....
```

```
int main( )
{
   Macierz M;
   Wektor Kol_i;
   float Liczba;
   Kol_i = M[0];   M[2] = Kol_i;
   Liczba = M(0,2);   M(1,1) = Liczba;
}
```

Plan prezentacji

Dlaczego szablony?

W językach takich jak *C* i *Pascal* mamy do czynienia z separacją kodu i typu parametrów. Wartości z jakimi wywoływane są funkcje i procedury mogą parametryzować ich działanie. Jednak ich typy zostają ustalone raz na zawsze w momencie ich definicji.

Problem:

Należy zaimplementować algorytm sortowania dla obiektów różnych typów.

Dlaczego szablony?

W językach takich jak *C* i *Pascal* mamy do czynienia z separacją kodu i typu parametrów. Wartości z jakimi wywoływane są funkcje i procedury mogą parametryzować ich działanie. Jednak ich typy zostają ustalone raz na zawsze w momencie ich definicji.

Problem: *Należy zaimplementować algorytm sortowania dla obiektów różnych typów.*

Dlaczego szablony?

Problem:

Należy zaimplementować algorytm sortowania dla obiektów różnych typów.

Możliwe rozwiązania:

- Implementacja algorytmu dla wszystkich typów, dla których przewidziane jest jego zastosowanie.
- Implementacja algorytmu dla typu podstawowego takiego jak ***void****$*$ lub ***Object***.
- Zdefiniowanie makr i wykorzystanie specjalnych preprocesorów (np. *cpp* dla *C/C++*).

Dlaczego szablony?

Problem: *Należy zaimplementować algorytm sortowania dla obiektów różnych typów.*

Możliwe rozwiązania:

- Implementacja algorytmu dla wszystkich typów, dla których przewidziane jest jego zastosowanie.
- Implementacja algorytmu dla typu podstawowego takiego jak ***void****∗ lub ***Object***.
- Zdefiniowanie makr i wykorzystanie specjalnych preprocesorów (np. *cpp* dla *C/C++*).

Dlaczego szablony?

Problem: *Należy zaimplementować algorytm sortowania dla obiektów różnych typów.*

Możliwe rozwiązania:

- Implementacja algorytmu dla wszystkich typów, dla których przewidziane jest jego zastosowanie.
- Implementacja algorytmu dla typu podstawowego takiego jak ***void****$*$ lub ***Object***.
- Zdefiniowanie makr i wykorzystanie specjalnych preprocesorów (np. *cpp* dla *C/C++*).

Dlaczego szablony?

Problem: *Należy zaimplementować algorytm sortowania dla obiektów różnych typów.*

Możliwe rozwiązania:

- Implementacja algorytmu dla wszystkich typów, dla których przewidziane jest jego zastosowanie.
- Implementacja algorytmu dla typu podstawowego takiego jak ***void****$*$ lub ***Object***.
- Zdefiniowanie makr i wykorzystanie specjalnych preprocesorów (np. *cpp* dla *C/C++*).

Najlepszym rozwiązaniem dla postawionego problemu jest koncepcja szablonów.

Dlaczego szablony?

Problem: *Należy zaimplementować algorytm sortowania dla obiektów różnych typów.*

Możliwe rozwiązania:

- Implementacja algorytmu dla wszystkich typów, dla których przewidziane jest jego zastosowanie.
- Implementacja algorytmu dla typu podstawowego takiego jak ***void****$*$ lub ***Object***.
- Zdefiniowanie makr i wykorzystanie specjalnych preprocesorów (np. *cpp* dla *C/C++*).

Najlepszym rozwiązaniem dla postawionego problemu jest koncepcja szablonów.

Podstawowe cechy

- Szablony pozwalają na definiowanie funkcji, których typy parametrów są także parametrami tych funkcji.
- Możliwe jest definiowanie klas, które parametryzowane mogą być typami pól występujących w tych klasach i/lub też typami parametrów metod.
- Programista definiuje tylko raz dany szablon. Kompilator dokonuje dedukcji typów parametrów danego szablonu i konkretyzuje go tworząc kod dla użytych typów w wywołaniu funkcji lub definicji obiektu danej klasy.
- Programista może też jawnie określić “wartości” parametrów szablonu.

Wady i zalety

Zalety:

- ⋆ Szablony dają możliwość tworzenia uniwersalnych algorytmów i uniwersalnych struktur danych.
- ⋆ W odróżnieniu od makr możliwe jest zachowanie przejrzystości kodu.
- ⋆ W odróżnieniu od wykorzystywania typów bazowych pozwalają zachować ścisłą kontrolę typów w trakcie kompilacji.

Wady:

- ◦ Brak możliwości tworzenia oddzielnych jednostek kompilacji (modułów) w postaci "czystych" szablonów.

Szablony funkcji – przykład dla typów wbudowanych

```
template <class Typ>
Typ Max( Typ w1, Typ w2 )
{
    return w1 < w2 ? w2 : w1;
}

enum Symbole { a=1, b, c };

int main( )
{
    cout << Max(1,2) << endl;
    cout << Max(1.1, 2.2) << endl;
    cout << Max('A','B') << endl;
    cout << Max( a , b ) << endl;
}
```

Szablony funkcji – przykład dla typów wbudowanych

```
template <class Typ>
Typ  Max( Typ  w1, Typ  w2 )
{
    return   w1 < w2 ? w2 : w1;
}

enum   Symbole { a=1, b, c };

int  main( )
{
    cout <<  Max(1,2) <<  endl;
    cout <<  Max(1.1, 2.2) <<  endl;
    cout <<  Max('A','B') <<  endl;
    cout <<  Max( a , b ) <<  endl;
}
```

Szablony funkcji – przykład dla typów wbudowanych

```
template <class Typ>
Typ  Max( Typ  w1, Typ  w2 )
{
    return  w1 < w2 ? w2 : w1;
}
```

W tym przykładzie kompilator generuje kod funkcji *Max* dla czterech przypadków. Słowo kluczowe **class** może zostać zastąpione przez **typename**.

```
enum  Symbole { a=1, b, c };

int  main( )
{
    cout <<  Max(1,2) <<  endl;
    cout <<  Max(1.1, 2.2) <<  endl;
    cout <<  Max('A','B') <<  endl;
    cout <<  Max( a , b ) <<  endl;
}
```

Szablony funkcji – własna klasa

```
template <class Typ>
Typ  Max( Typ w1, Typ w2 )
{
    return  w1 < w2 ? w2 : w1;
}
```

Czy szablon można stosować również dla własnych klas?

```
int  main( )
{
    Wektor  W1(1,1), W2(4,5), W3;

    W3 = Max(W1,W2);
}
```

Szablony funkcji – własna klasa

```
struct Wektor {
    float x, y;
};



template <class Typ>
Typ Max( Typ w1, Typ w2 )
{
    return w1 < w2 ? w2 : w1;
}

int main( )
{
    Wektor W1(1,1), W2(4,5), W3;

    W3 = Max(W1,W2);
}
```

Szablony funkcji – własna klasa

```
struct Wektor {
    float x, y;
};
```

```
template <class Typ>
Typ Max( Typ w1, Typ w2 )
{
    return w1 < w2 ? w2 : w1;
}

int main( )
{
    Wektor W1(1,1), W2(4,5), W3;
    W3 = Max(W1,W2);
}
```

W takiej postaci na pewno nie będzie dobrze. Dlaczego?

Szablony funkcji – własna klasa

```
struct Wektor {
    float x, y;
};
```

```
template <class Typ>
Typ Max( Typ w1, Typ w2 )
{
    return w1 < w2 ? w2 : w1;
}

int main( )
{
    Wektor W1(1,1), W2(4,5), W3;
    W3 = Max(W1,W2);
}
```

Problemem jest operacja porównania dwóch wektorów. Dlaczego?

Szablony funkcji – własna klasa

```
struct Wektor {
    float x, y;
    bool operator < ( const Wektor& W ) const
            { return x*x+y*y < W.x*W.x+W.y*W.y; }
};

template <class Typ>
Typ Max( Typ w1, Typ w2 )
{
    return w1 < w2 ? w2 : w1;
}

int main( )
{
    Wektor W1(1,1), W2(4,5), W3;
    W3 = Max(W1,W2);
}
```

Aby było dobrze, należy zdefiniować przeciążenie operatora porównania.

Plan prezentacji

Szablony klas

- Szablony klas pozwalają na przedstawienie ogólnych pojęć i wzajemnych ich związków.
- Pozwalają na abstrahowanie od typu poszczególnych atrybutów związanych z danym pojęciem.
- Pozwalają programiście skoncentrować na ogólnych zależnościach i mechanizmach.
- Szablony pozwalają na generowanie i optymalizowania już na etapie kompilacji poprzez użycie specyficznych konstrukcji programistycznych.
- Umożliwiają realizację idei programowania uogólnionego.

Ogólna postać szablonu

template < *lista-parametrow-rozdzielonych-przecinkami* >
class ***Klasa*** {

. . .

};

Ogólna postać szablonu

template < *lista-parametrow-rozdzielonych-przecinkami* >
class ***Klasa*** {

. . .

};

Dopuszczalne parametry:

- typ wbudowany lub zdefiniowany przez użytkownika,
- stała w czasie kompilacji (liczba, wskaźnik, znaki itp.),
- inny szablon.

Ogólna postać szablonu

template < *lista-parametrow-rozdzielonych-przecinkami* >
class ***Klasa*** {

...

};

Dopuszczalne parametry:

- typ wbudowany lub zdefiniowany przez użytkownika,
- stała w czasie kompilacji (liczba, wskaźnik, znaki itp.),
- inny szablon.

Ogólna postać szablonu

template < *lista-parametrow-rozdzielonych-przecinkami* >
class ***Klasa*** {

. . .

};

Dopuszczalne parametry:

- typ wbudowany lub zdefiniowany przez użytkownika,
- stała w czasie kompilacji (liczba, wskaźnik, znaki itp.),
- inny szablon.

Stos

```
class Stos {
      int                 _Tab[ROZ_STOSU];
      int unsigned  int   _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(int& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  int& El)
        { return   _Ilosc >= ROZ_STOSU ? false : _Tab[_Ilosc++] = El, true; }
};

int  main( )
{
   Stos    St;
}
```

Przykład szablonu stosu

```
template < typename  TYP >
class Stos {
      TYP             _Tab[ROZ_STOSU];
      unsigned int    _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(TYP& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  TYP& El)
        { return   _Ilosc >= ROZ_STOSU ? false : _Tab[_Ilosc++] = El, true; }
};
```

Słowo kluczowe *typename* sygnalizuje, że parametr szablonu jest typem.

Przykład szablonu stosu

```
template < class  TYP >
class Stos {
      TYP             _Tab[ROZ_STOSU];
      unsigned int    _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(TYP& El)
        { return  !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  TYP& El)
        { return  _Ilosc >= ROZ_STOSU ? false : _Tab[_Ilosc++] = El, true; }
};
```

Przykład szablonu stosu

```
template < typename  TYP >
class Stos {
      TYP             _Tab[ROZ_STOSU];
      unsigned  int   _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(TYP& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  TYP& El)
        { return   _Ilosc >= ROZ_STOSU ? false : _Tab[_Ilosc++] = El, true; }
};
```

Przykład szablonu stosu

```
template < typename  TYP >
class Stos {
      TYP              _Tab[ROZ_STOSU];
      unsigned  int    _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(TYP& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  TYP& El)
        { return   _Ilosc >= ROZ_STOSU ? false : _Tab[_Ilosc++] = El, true; }
};

int  main( )
{
   Stos<float>   St;
}
```

Przykład szablonu stosu

```
template < typename  TYP >
class Stos {
      TYP              _Tab[ROZ_STOSU];
      unsigned  int    _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(TYP& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  TYP& El)
        { return   _Ilosc >= ROZ_STOSU ? false : _Tab[_Ilosc++] = El, true; }
};

int  main( )
{
   Stos<double[100]>   St;
}
```

Przykład szablonu stosu

```
template < typename  TYP >
class Stos {
      TYP              _Tab[ROZ_STOSU];
      unsigned  int    _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(TYP& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  TYP& El)
        { return   _Ilosc >= ROZ_STOSU ? false : _Tab[_Ilosc++] = El, true; }
};

int  main( )
{
   Stos<std::string>   St;
}
```

Przykład szablonu stosu

```
template < typename  TYP >
class Stos {
      TYP             _Tab[ROZ_STOSU];
      unsigned  int   _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(TYP& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  TYP& El)
        { return   _Ilosc >= ROZ_STOSU ? false : _Tab[_Ilosc++] = El, true; }
};

int  main( )
{
   Stos<std::istream>   St;
}
```

Przykład szablonu stosu

```
template < typename  TYP >
class Stos {
      TYP              _Tab[ROZ_STOSU];
      unsigned  int    _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(TYP& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  TYP& El)
        { return   _Ilosc >= ROZ_STOSU ? false : _Tab[_Ilosc++] = El, true; }
};

int  main( )
{
   Stos<std::istream>   St;
}
```

Przykład szablonu stosu

```
template < typename  TYP >
class Stos {
      TYP              _Tab[ROZ_STOSU];
      unsigned  int    _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(TYP& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  TYP& El)
        { return   _Ilosc >= ROZ_STOSU ? false : _Tab[_Ilosc++] = El, true; }
};

int  main( )
{
   Stos<std::istream>   St;
}
```

Przykład szablonu stosu

```
template < typename  TYP >
class Stos {
      TYP              _Tab[ROZ_STOSU];
      unsigned  int    _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(TYP& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  TYP& El)
        { return   _Ilosc >= ROZ_STOSU ? false : _Tab[_Ilosc++] = El, true; }
};

int  main( )
{
   Stos<std::istream*>   St;
}
```

Przykład szablonu stosu

```
template < typename  TYP >
class Stos {
      TYP              _Tab[ROZ_STOSU];
      unsigned  int    _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(TYP& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  TYP& El)
        { return   _Ilosc >= ROZ_STOSU ? false : _Tab[_Ilosc++] = El, true; }
};

int  main( )
{
   Stos<std::istream&>   St;
}
```

Przykład szablonu stosu

```
template < typename  TYP >
class Stos {
      TYP             _Tab[ROZ_STOSU];
      unsigned  int   _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(TYP& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  TYP& El)
        { return   _Ilosc >= ROZ_STOSU ? false : _Tab[_Ilosc++] = El, true; }
};

int  main( )
{
   Stos<std::istream&>   St;
}
```

Przykład szablonu stosu

```
template < typename  TYP >
class Stos {
      TYP              _Tab[ROZ_STOSU];
      unsigned  int    _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(TYP& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  TYP& El)
        { return   _Ilosc >= ROZ_STOSU ? false : _Tab[_Ilosc++] = El, true; }
};

int  main( )
{
   Stos< Stos< Stos< char[20]  > > >   St;
}
```

Przykład szablonu stosu

```
template < typename  TYP >
class Stos {
      TYP              _Tab[ROZ_STOSU];
      unsigned  int    _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(TYP& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  TYP& El)
        { return   _Ilosc >= ROZ_STOSU ? false : _Tab[_Ilosc++] = El, true; }
};
```

Przykład szablonu stosu

```
template < typename  TYP, unsigned int Rozmiar >
class Stos {
      TYP    _Tab[Rozmiar];
      unsigned  int  _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(TYP& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  TYP& El)
        { return   _Ilosc >= Rozmiar ? false : _Tab[_Ilosc++] = El, true; }
};
```

Przykład szablonu stosu

```
template < typename  TYP, unsigned int Rozmiar >
class Stos {
      TYP    _Tab[Rozmiar];
      unsigned  int  _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(TYP& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  TYP& El)
        { return   _Ilosc >= Rozmiar ? false : _Tab[_Ilosc++] = El, true; }
};

int  main( )
{
   Stos<float, 100>   St;
}
```

Przykład szablonu stosu

```
template < typename  TYP, unsigned int Rozmiar= 100 >
class Stos {
      TYP    _Tab[Rozmiar];
      unsigned  int  _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(TYP& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  TYP& El)
        { return   _Ilosc >= Rozmiar ? false : _Tab[_Ilosc++] = El, true; }
};

int  main( )
{
   Stos<float, 100>   St;
}
```

Przykład szablonu stosu

```
template < typename  TYP, unsigned int Rozmiar= 100 >
class Stos {
      TYP    _Tab[Rozmiar];
      unsigned  int  _Ilosc;
   public :
      Stos( ) { _Ilosc = 0; }

      bool  Pobierz(TYP& El)
        { return   !_Ilosc ? false : El = _Tab[--_Ilosc], true; }

      bool  Poloz(const  TYP& El)
        { return   _Ilosc >= Rozmiar ? false : _Tab[_Ilosc++] = El, true; }
};

int  main( )
{
   Stos<float>   St;
}
```

Plan prezentacji

Klasa wektor – plik nagłówkowy: wektor.hh

```
#ifndef WEKTOR_HH
#define WEKTOR_HH
#include <iostream>

#define ROZMIAR 3
#define TYP double

class Wektor {
        TYP _Wsp[ ROZMIAR ];
      public :
        Wektor( );
        TYP operator [ ] (int Ind) const { return _Wsp[Ind]; }
        TYP& operator [ ] (int Ind) { return _Wsp[Ind]; }
};

std::ostream & operator << (std::ostream & StrmWy, const Wektor & Wek);
std::istream & operator >> (std::istream & StrmWe, Wektor & Wek);
#endif
```

Klasa wektor – plik modułu: wektor.cpp

```
#include "wektor.hh"
using namespace std;

Wektor::Wektor( )
{
   for (int Ind = 0; Ind < ROZMIAR; ++Ind) _Tab[Ind] = 0;
}

ostream & operator << (ostream & StrmWy, const Wektor & Wek)
{
      ...
}

istream & operator >> (istream & StrmWe, Wektor & Wek)
{
      ...
}
```

Klasa wektor – plik nagłówkowy: wektor.hh

```
#ifndef WEKTOR_HH
#define WEKTOR_HH
#include <iostream>


template <typename Typ, int Rozmiar>
class Wektor {
      private :
        Typ _Wsp[ Rozmiar ];
      public :
        Wektor();
        Typ operator [ ] (unsigned int Ind) const { return _Wsp[Ind]; }
        Typ& operator [ ] (unsigned int Ind)      { return _Wsp[Ind]; }
};



#endif
```

Klasa wektor – plik nagłówkowy: wektor.hh

```
...
template <typename Typ, int Rozmiar>
class Wektor {
      private :
        Typ _Wsp[ Rozmiar ];
      public :
        Wektor();
        Typ operator [ ] (unsigned int Ind) const { return _Wsp[Ind]; }
        Typ& operator [ ] (unsigned int Ind)      { return _Wsp[Ind]; }
};

template <typename Typ, int Rozmiar>
std::ostream & operator << (std::ostream & StrmWy, Wektor<Typ,Rozmiar>& Wek)
{
   ...
}

template <typename Typ, int Rozmiar>
std::istream & operator >> (std::istream & StrmWe, Wektor<Typ,Rozmiar>& Wek)
{
   ...
}
```

Klasa wektor – plik nagłówkowy: wektor.hh

```
...
template <typename Typ, int Rozmiar>
class Wektor {
        ...
        public :
           Wektor();
           ...
};

template <typename Typ, int Rozmiar>
Wektor<Typ,Rozmiar>::Wektor()
{
    for (int Ind = 0; Ind < Rozmiar; ++Ind) _Wsp[Ind] = 0;
}

 ...
template <typename Typ, int Rozmiar>
std::istream & operator >> (std::istream & StrmWe, Wektor<Typ,Rozmiar>& Wek)
{
    ...
}
```

Klasa wektor – z pętlą zakresu

```
...
template <typename Typ, int Rozmiar>
class Wektor {
        ...
        public :
           Wektor();
           ...
};

template <typename Typ, int Rozmiar>
Wektor<Typ,Rozmiar>::Wektor()
{
    for (int &Wsp_i : _Wsp) Wsp_i = 0;
}

 ...
template <typename Typ, int Rozmiar>
std::istream & operator >> (std::istream & StrmWe, Wektor<Typ,Rozmiar>& Wek)
{
    ...
}
```

Wykorzystanie *pętli zakresu* jest bardziej bezpiecznym, jak też bardziej efektywnym rozwiązaniem.

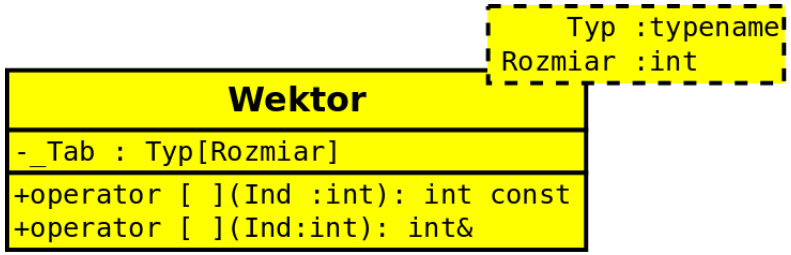
Klasa wektor – plik nagłówkowy: wektor.hh

```
...
inline
bool Wczytaj_OkreslonyZnak(std::istream &StrmWe, const char Wzorzec)
{
    ...
}

template <typename Typ>
bool Wczytaj_Liczbe_OkreslonyZnak(std::istream & StrmWe, Typ & Liczba, const char Wzorzec)
{
    ...
}


template <typename Typ, int Rozmiar>
std::istream & operator >> (std::istream & StrmWe, Wektor<Typ,Rozmiar>& Wek)
{
  ...
}
...
```

Plan prezentacji

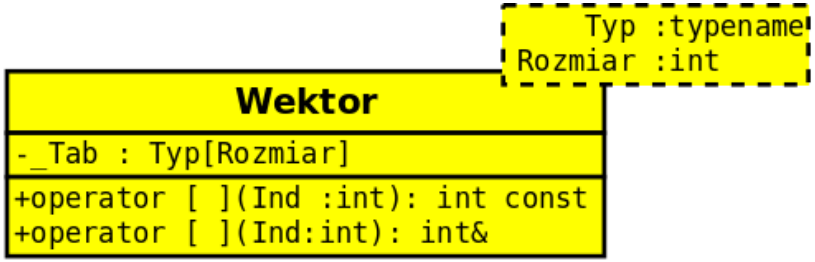
Szablon klasy Wektor w UML

```
template <typename Typ, int Rozmiar>
class Wektor {
      private :
        Typ _Wsp[ Rozmiar ];
      public :
        Wektor();
        Typ operator [ ] (unsigned int Ind) const { return _Wsp[Ind]; }
        Typ& operator [ ] (unsigned int Ind)      { return _Wsp[Ind]; }
};
```



Klasa będąca instancją szablonu

```
template <typename Typ, int Rozmiar>
class Wektor {
      private :
         Typ _Wsp[ Rozmiar ];
      public :
         Wektor();
         Typ operator [ ] (unsigned int Ind) const { return _Wsp[Ind]; }
         Typ& operator [ ] (unsigned int Ind)      { return _Wsp[Ind]; }
};
```




```
Wektor<double, 3>    W;
```

Koniec prezentacji
Dziękuję za uwagę