Programowanie obiektowe

Mechanizm wyjątków,
Klasy szablonowe

Wyjątki – co to takiego?

Błędy i ich obsługa

W trakcie działania programu zawsze może dojść do powstania sytuacji nietypowej. Uniknąć się tego nie da. Można jednak przewidywać iż taka sytuacja wystąpi i się na nią przygotować.

Drobiazgowe kontrolowanie poprawności wykonywanych operacji jest możliwa, jednak kompleksowa obsługa błędów jest uciążliwa. Przykład obsługi w C:

```
if( ( file = fopen( name, "rt" ) ) == NULL )
   return IN_FILE_ERROR;
else
   if( ( num_of_lines = count_lines( file ) ) = 0 )
      return EMPTY_IN_FILE;
   else
      if( ( lines = malloc( num_of_lines * sizeof( char ) ) ) == NULL )
         return NO_MEM_ERROR;
      else
         if( ( buffer = malloc( MAX_LINE_LEN ) ) == NULL )
            return NO_MEM_ERROR;
         else
            while( fgets( buffer, file, MAX_LINE_LEN ) != NULL )
            {
               . . .
            }
. . .
```

Wyjątki – co to takiego?

Błąd jako sytuacja wyjątkowa

W języku C++ wprowadzono mechanizm pozwalający programiście na reagowanie na sytuacje błędne czy nietypowe.

Gdy taka sytuacja wystąpi, programista może wygenerować *wyjątek*.

- **Wyjątek** (ang. *exception*) to *obiekt* pewnej *klasy*. Wygenerowanie wyjątku polega na *przekazaniu obiektu* pisującego wyjątek z fragmentu kodu, w którym *wystąpił problem*, do fragmentu, w którym *przewidziano jego obsługę*.
- **Wygenerowanie (zgłoszenie) wyjątku** powoduje *przerwanie wykonywania* sprawiającego problemy kodu i przejście do obsługi sytuacji problematycznej. Obsługa ta może znajdować się w innym miejscu kodu.
- **Wyjątek jest obiektem**, jego *klasa* określa typ sytuacji wyjątkowej. Obiekt może w sobie posiadać pola oraz funkcje składowe, pozwalające na sprecyzowanie informacji o zaistniałej sytuacji wyjątkowej.

Wyjątki – co to takiego?

Przykład wykorzystania wyjątków

- Krok 1-szy – zdefiniowanie klasy wyjątku:

```
class Error
{
};
```

- Krok 2-gi – zgłaszanie wyjątku:

```
void allocItems()
{
   int numOfItems;

   cin >> numOfItems;

   if( numOfItems <= 0 )
      throw Error();
   else
   {
      // . . .
   }
}
```

Zgłoszenie wyjątku klasy **`Error`**

Zgłoszenie (wygenerowanie) wyjątku – instrukcja throw

```
throw wyrażenie_pewnego_typu;
```

Wyjątki – co to takiego?

Przykład wykorzystania wyjątków

- Krok 3-ci – wywołanie funkcji generującej (zgłaszającej) wyjątek, kod wywołujący funkcję generującą wyjątek musi być napisany w specjalny sposób:

```
. . .
try
{
   allocItems();
}
catch( Error )
{
   cout << "Blad";
}
. . .
```

Funkcja, która potencjalnie może wygenerować wyjątek, powinna być wywołana w obrębie bloku **`try`**.

Obsługa wyjątków (ang. *exception handler*). Tutaj kierowane jest wykonanie programu po wygenerowaniu wyjątku instrukcją **`throw`**.

Co się stanie, gdy wyjątek nie zostanie obsłużony?

```
allocItems();
// Dalszy ciąg programu
// . . .
// . . .
```

Jeżeli ta funkcja wygeneruje wyjątek, program zostanie awaryjnie zakończony, dalsza część programu się nie wykona.

Ogólny schemat zgłaszania wyjątków

Zgłaszanie wyjątków — throw — nieco dokładniej

Wyjątki są generowane instrukcją **`throw`**. Po słowie kluczowym **`throw`** występuje dowolne wyrażenie pewnego typu. Wartość tego wyrażenia jest zgłaszana jako wyjątek.

Wyjątki mogą być również typu wbudowanego:

```
if( numOfItems <= 0 )
   throw 1;
```

```
. . .
try
{
   allocItems();
}
catch( int )
{
   cout << "Wyjatek jako int";
}
. . .
```

```
if( numOfItems <= 0 )
   throw "Ujemna liczba elementow";
```

```
. . .
try
{
   allocItems();
}
catch( char const * )
{
   cout << "Wyjatek jako char const *";
}
. . .
```

Tak zdefiniowana obsługa jest „czuła” na różne typy wyjątków. W obrębie danego typu wyjątku, różne jego wartości nie są rozróżniane.

Ogólny schemat zgłaszania wyjątków

Co się dzieje po zgłoszeniu wyjątku?

Gdy wyjątek zostanie wygenerowany instrukcją **`throw`**, biblioteka czasu wykonania (ang. *RTL – run time library*):

- Pobiera wyjątek, określa jego typ.
- Przeszukuje stos wywołań funkcji w poszukiwaniu takiej, która zawiera obsługę wyjątku tego typu (czyli odpowiednią instrukcję **`catch`**).
- Jeżeli odpowiednia obsługa wyjątku zostanie znaleziona, stos jest w tym miejscu „rozwijany” (ang. *unwind*) i wyjątek jest obsługiwany.
- Jeżeli w trakcie tych poszukiwań nie został znaleziony pasujący blok obsługi wyjątku, dochodzimy do punktu wejściowego programu,czyli funkcji **`main`**.
- Jeżeli i tutaj nie ma obsługi zgłoszonego wyjątku, program jest przerywany w trybie awaryjnym.
- W trakcie przechodzenia do kolejnych bloków w stosie wywołań, usuwane są wszystkie obiekty automatyczne, czemu towarzyszy aktywowanie ich destruktorów.

Ogólny schemat zgłaszania wyjątków

Co się dzieje po zgłoszeniu wyjątku?

Propagacja wyjątku kończy się :

- **jego obsługą** w odpowiednim bloku `catch`, po czym wykonywane są kolejne instrukcje następujące po tym bloku,
- **przerwaniem wykonania programu**, gdy wyjątek nie został obsłużony.

Proces obsługi wyjątku *jest jednokierunkowy*. W jego trakcie niszczone są obiekty automatyczne, a sterowanie *nigdy nie wróci* automatyczne do miejsca zgłoszenia wyjątku.

Uwaga:

- W trakcie przechodzenia do kolejnych bloków w stosie wywołań, usuwane są wszystkie obiekty automatyczne, czemu towarzyszy aktywowanie ich destruktorów.
- Jeżeli w trakcie tego procesu, któryś z destruktorów zgłosi wyjątek i go nie obsłuży, następuje awaryjne przerwanie wykonania programu.

Ogólny schemat zgłaszania wyjątków

W jaki sposób przerywane jest wykonanie programu?

- Jeżeli wyjątek nie zostanie obsłużony, wywoływana jest biblioteczna funkcja **`terminate()`**.
- Domyślnie funkcja ta wywołuje funkcję **`abort()`** pochodzącą z standardowej biblioteki C, identyfikowanej przez plik nagłówkowy **`stdlib.h`**.
- Funkcja **`abort()`** nie zapewnia aktywowania destruktorów dla obiektów globalnych i statycznych.
- Można zmienić działanie funkcji **`terminate()`** poprzez zainstalowanie własnej funkcji kończącej, służy do tego funkcja **`set_terminate()`**.
- Własna funkcja kończąca musi być bezargumentową funkcją o rezultacie typu **`void`**.
- Rezultatem wywołania funkcji **`set_terminate()`** jest wskaźnik na poprzednio zainstalowaną funkcję.
- Można zatem zapamiętać adres oryginalnej funkcji kończącej i przywrócić ją w razie konieczności.

Ogólny schemat zgłaszania wyjątków

Jak zmienić domyślną funkcję kończącą?

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

void my_terminator()
{
   cout << "I'll be back!" << endl;
   updateErrorsLog();
   cin.get();
   exit(0);
}

// Ustaw nową funkcję kończącą, zapamiętaj adres poprzedniej
void ( *old_terminator )() = set_terminate( my_terminator );

void throwFun()
{
   throw 1;
}

int main()
{
   throwFun();
   return 0;
}
```

Konieczne jest włączenie tego pliku nagłówkowego

Własna funkcja kończąca

Ogólny schemat obsługi wyjątków

Wychwytywanie większej liczby rodzajów wyjątków

Wyjątki mogą być różnych typów. Można zatem rozróżniać rodzaje zgłaszanych sytuacji wyjątkowych.

```
class CriticalError          // Definicja klasy wyjątku krytycznego
{
};

class NormalError            // Definicja klasy wyjątku zwykłego
{
};
```

```
void allocItems()
{
   int numOfItems;
   cin >> numOfItems;

   if( numOfItems <= 0 )
      throw NormalError();
   else
   {
      // . . .
      throw CriticalError();
   }
}
```

Wygenerowanie „normalnego” wyjątku – niski priorytet

Wygenerowanie krytycznego wyjątku – wysoki priorytet

Ogólny schemat obsługi wyjątków

Wychwytywanie „wszystkich” wyjątków

Czasem chcemy wyłapać jednorazowo wszystkie klasy wyjątków:

```
int main()
{
   try
   {
      allocItems();
   }
   catch( CriticalError )
   {
      cout << "Blad krytyczny";
   }
   catch( NormalError )
   {
      cout << "Wyjatek zwykly";
   }
   return 0;
}
```

Wyłapanie krytycznego wyjątku

Wyłapanie wyjątku o niższym priorytecie

Ogólny schemat obsługi wyjątków

Wychwytywanie większej liczby rodzajów wyjątków, cd…

Wychwytywanie różnych klas wyjątków za jednym „zamachem”:

```
try
{
   allocItems();
}
catch( ... )
{
   cout << "Biore wszystkie wyjatki na siebie!!!";
}
```

Wyłapanie wszystkich wyjątków

Wychwytywanie wybranych wyjątków a reszta – obsługa „hurtowa”:

```
try
{
   allocItems();
}
catch( CriticalError )
{
    cout << "Krytyczne lapiemy osobno";
}
catch( ... )
{
   cout << "Reszta hurtowo jak leci";
}
```

Wyłapanie wszystkich wyjątków, innych niż **`CriticalError`**

Ogólny schemat obsługi wyjątków

Nazwane wyjątki — dane w wyjątkach

- Do tej pory rozróżniane były typy wyjątków. Inny typ wyjątku – inna sytuacja wyjątkowa.
- Wyjątki to obiekty. Mogą posiadać pola, można je inicjować i odczytywać.
- W polach obiektu stanowiącego wyjątek można przechowywać informację o różnych przyczynach powstania wyjątku.

Klasa wyjątku przechowującego kod błędu

```
class Error
{
   public:
      enum { CRITICAL, NORMAL };
      Error( int code ) : errCode( code ) {}
      int getCode() const { return errCode; }
   private:
      int errCode;
};
```

Ogólny schemat obsługi wyjątków

Nazwane wyjątki – dane w wyjątkach, cd...

Generowanie wyjątku jednego typu ze zróżnicowaniem kodu pamiętanego w polu klasy

```
void allocItems()
{
   int numOfItems;
   cin >> numOfItems;

   if( numOfItems <= 0 )
      throw Error( Error::NORMAL );
   else
   {
      // . . .
      throw Error( Error::CRITICAL );
   }
}
```

Wygenerowanie „normalnego” wyjątku – niski priorytet

Wygenerowanie krytycznego wyjątku – wysoki priorytet

Ogólny schemat obsługi wyjątków

Nazwane wyjątki – dane w wyjątkach, cd...

Wyłapywanie wyjątku i identyfikacja kodu błędu

```
try
{
   allocItems();
}
catch( Error & error )
{
   switch( error.getCode() )
   {
       case Error::CRITICAL    : cout << "Blad krytyczny";
                                 break;
       case Error::NORMAL     : cout << "Zwykly blad";
                                 break;
   }
}
```

Ogólny schemat obsługi wyjątków

Nazwane wyjątki — bardziej obiektowa wersja

Identyfikację błędu można poddać hermetyzacji

```
class Error
{
   public:
      enum { CRITICAL, NORMAL };
      Error( int code ) : errCode( code ) {}
      char * errorName();
   private:
      int errCode;
};

char * Error::errorName()
{
   switch( errCode )
   {
       case CRITICAL: return "Blad krytyczny";
       case NORMAL   : return "Zwykly blad";
       default       : return "Nieznany blad";
   }
}
```

Ogólny schemat obsługi wyjątków

Nazwane wyjątki — bardziej obiektowa wersja

Niech wyjątek sam powie, kim jest:

```
try
{
   allocItems();
}
catch( Error & error )
{
   cout << error.errorName();
}
```

Szablony – co to takiego?

Przeciążanie funkcji

Przeciążanie funkcji (ang. *function overloading*) — tworzenie większej liczby funkcji o *takiej samej nazwie*.

```
int add( int a, int b )                    // 1-sza wersja funkcji przeciążonej add
{
   return a + b;
}

double add( double a, double b )           // 2-ga wersja funkcji przeciążonej add
{
   return a + b;
}

cout << endl << "Dodawanie int:" << add( 1, 1 )
cout << endl << "Dodawanie double :" << add( 1.0, 1.0 );
```

- Realizacje funkcji są takie same. Różnią je tylko *typy argumentów* i *rezultatu*.
- Definicje kolejnych funkcji przeciążonych dla innych typów, np. **`Complex`**, to powielenie tego samego schematu ze zmianą nazw typów.

Funkcje szablonowe

Funkcja w postaci esencjonalnej — szablon dla funkcji add

Funkcja szablonowa (ang. *template function*) jest ogólną postacią funkcji, określającą przepis na utworzenie funkcji skonkretyzowanej.

```
template <typename T>
T add( T a, T b )
{
   return a + b;
}
```

Deklarację funkcji szablonowej poprzedza specyfikacja:

```
template <typename identyfikator>
```

lub, traktowana jako przestarzała:

```
template <class identyfikator>
```

- Identyfikator, w przypadku funkcji **`add`** to **`T`**, reprezentuje oznaczenie nieokreślonego typu.
- Typ zostanie określony przez kompilator przy wywołaniu funkcji, co spowoduje wygenerowanie jej skonkretyzowanej wersji.

Funkcje szablonowe

Wywołanie funkcji szablonowej

Na etapie wywołania funkcji szablonowej, programista określa typ na podstawie którego kompilator tworzy postać funkcji skonkretyzowanej.

```
cout << add<int>( 1, 2 ) << endl;
cout << add<double>( 1.2, 2.5 ) << endl;
```

```
3
3.7
```

Konkretyzację typu można pominąć, kompilator sam określi typy na etapie wywołania:

```
cout << add( 1, 2 ) << endl;
cout << add( 1.2, 2.5 ) << endl;
```

```
3
3.7
```

Pozwala to uniknąć niejednoznaczności, np:

```
cout << add<int>( 1.2, 2.5 );
```

```
3
```

Message
In function `int main()':
[Warning] passing `double' for converting 1 of `T add(T, T) [with T = int]'
[Warning] passing `double' for converting 2 of `T add(T, T) [with T = int]'

Funkcje szablonowe

Wywołanie funkcji szablonowej, cd...

Funkcja szablonowa może być konkretyzowana nie tylko dla typów wbudowanych:

```
Complex a( 1, 1 );
Complex b( 2, 2 );
Complex c;

c = add( a, b );
cout << c.getReal() << ' ' << c.getImag() << endl;
```

```
3 3
```

Przy rozpatrywanej postaci funkcji szablonowej **add**:

```
template <typename T>
T add( T a, T b )
{
   return a + b;
}
```

wywołanie:

```
add( 1.2, 2 );
```

będzie błędne!

Message
In function `int main()':
no matching function for call to `add(double, int)'

Funkcje szablonowe

Wywołanie funkcji szablonowej, cd...

Funkcja szablonowa może otrzymywać więcej niż jeden typ uogólniony:

```
template <typename T1, typename T2>
T1 add( T1 a, T2 b )
{
   return a + b;
}
```

Przy takiej postaci funkcji szablonowej **`add`**, poniższe wywołanie będzie poprawne:

```
add( 1.2, 2 );
```

→ `3.2`

Jednak takie wywołanie:

```
add( 2, 1.2 );
```

→ `3`

Spowoduje konwersję typu rezultatu, sygnalizowaną przez niektóre kompilatory ostrzeżeniem:

Message
In function `T1 add(T1, T2) [with T1 = int, T2 = double]':
instantiated from here
[Warning] converting to `int' from `double'

Klasy szablonowe

Dynamiczna tablica jako klasa szablonowa – motywacja

Załóżmy, że chcemy korzystać z tablicy liczb całkowitych o zmiennym rozmiarze:

```
IntArray tab;
int n;

cout << "Liczba elementow: ";
cin >> n;
tab.expand( n );

for( int i = 0; i < n ; i++ )
   tab.add( i );

for( int i = 0; i < tab.getLen() ; i++ )
   cout << tab[ i ] << ' ';

cout << endl << "Rozszerzam o trzy: " << endl;

tab.add( ++n );
tab.add( ++n );
tab.add( ++n );

for( int i = 0; i < tab.getLen() ; i++ )
   cout << tab[ i ] << ' ';
```

```
Liczba elementow: 5
0 1 2 3 4
Rozszerzam o trzy:
0 1 2 3 4 6 7 8
```

Zastosowanie szablonów pozwoli na uogólnienie tej tablicy na inne typy

Klasy szablonowe

Dynamiczna tablica jako klasa szablonowa – zastosowanie

Tablica skonkretyzowana dla typu **`int`**:

```
Array<int> tab;
int n;

cout << "Liczba elementow: ";
cin >> n;
tab.expand( n );

for( int i = 0; i < n ; i++ )
   tab.add( i );

for( int i = 0; i < tab.getLen() ; i++ )
   cout << tab[ i ] << ' ';

cout << endl << "Rozszerzam o trzy: " << endl;

tab.add( ++n );
tab.add( ++n );
tab.add( ++n );

for( int i = 0; i < tab.getLen() ; i++ )
   cout << tab[ i ] << ' ';
```

Klasy szablonowe

Dynamiczna tablica jako klasa szablonowa – zastosowanie

Tablica skonkretyzowana dla typu **`float`**:

```
Array<float> tab;

tab.expand( 5 );

tab.add( 1.0 );
tab.add( 2.0 );
tab.add( 3.0 );
tab.add( 4.0 );
tab.add( 5.0 );

for( int i = 0; i < tab.getLen() ; i++ )
   cout << tab[ i ] << ' ';
```

Klasy szablonowe

Dynamiczna tablica jako klasa szablonowa – zastosowanie

Tablica skonkretyzowana dla typu **`Complex`**:

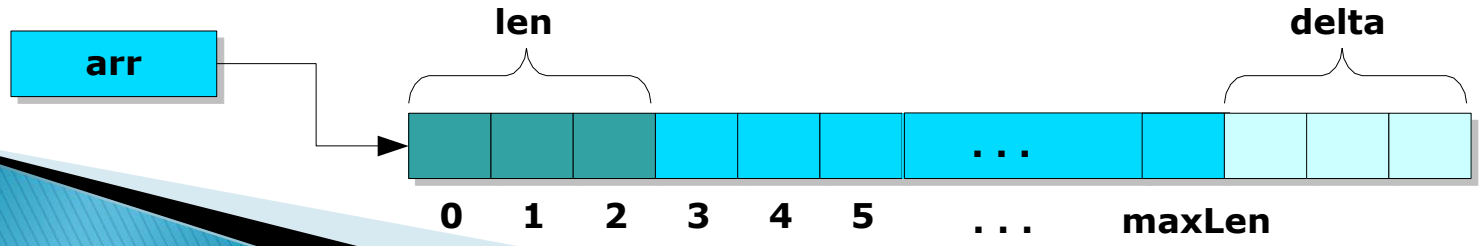
```
Array<Complex> tab;

tab.expand( 5 );
tab.add( Complex( 1, 1 ) );
tab.add( Complex( 2, 2 ) );
tab.add( Complex( 3, 3 ) );
tab.add( Complex( 4, 5 ) );
tab.add( Complex( 5, 5 ) );

for( int i = 0; i < tab.getLen() ; i++ )
   cout << tab[ i ].getReal() << ' ' << tab[ i ].getImag() << endl;

// Aternatywnie, gdy klasa Complex ma przeciążony operator << :
for( int i = 0; i < tab.getLen() ; i++ )
   cout << tab[ i ] << ' ';
```

Klasy szablonowe – anatomia prostej klasy

Dynamiczna tablica jako klasa szablonowa — jak to działa?

Tablica będzie przechowywać elementy uogólnionego typu **T**

```
template <class T>
class Array
{
   public:

      // . . .

   protected:

      T *    arr;        // Wskaźnik na bufor dla obiektów typu T
      int    len;        // Długość tablicy – liczba wstawionych elementów
      int    maxLen;     // Maksymalna liczba elementów
      int    delta;      // Lb. elem. o jakie tablica zostanie powiększona

      //. . .
};
```



Klasy szablonowe – anatomia prostej klasy

Dynamiczna tablica jako klasa szablonowa — jak to działa?

Funkcje administracyjne

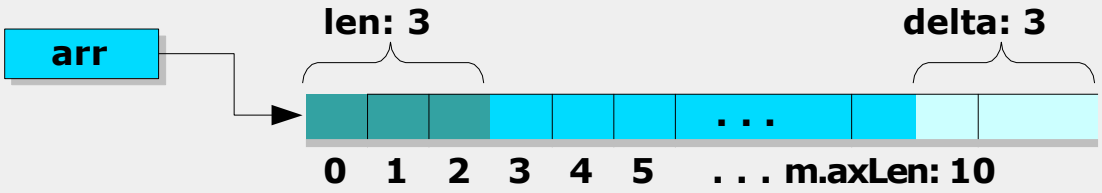
```
template <class T>
class Array
{
   public:
      int getLen() const            // Pobranie aktualnej liczby elementów
      {
          return len;
      }

      int getMaxLen() const         // Pobranie maksymalnej liczby elementów
      {
          return maxLen;
      }

      int getDelta() const          // Jaki przyrost rozmiaru gdy brak miejsca?
      {
          return delta;
      }

      T * getBegin()                // Pobranie początku tablicy elementów
      {
         return arr;
      }
};
```

Klasy szablonowe – anatomia prostej klasy

Dynamiczna tablica jako klasa szablonowa — jak to działa?

Reprezentacja wewnętrzna na przykładzie

```
Array<Complex> tab( 10, 3 );


tab.add( Complex( 1, 1 ) );
tab.add( Complex( 2, 2 ) );
tab.add( Complex( 3, 3 ) );

cout << "Dlugosc: " << tab.getLen() << endl;
cout << "Max. dlugosc: " << tab.getMaxLen() << endl;
cout << "Przyrost: " << tab.getDelta() << endl;

for( int i = 0; i < tab.getLen() ; i++ )
   cout << tab[ i ] << endl;

for( int i = 4; i < 15; i++ )
   tab.add( Complex( i, i ) );

cout << "Dlugosc: " << tab.getLen() << endl;
cout << "Max. dlugosc: " << tab.getMaxLen() << endl;
cout << "Przyrost: " << tab.getDelta() << endl;

for( int i = 0; i < tab.getLen() ; i++ )
   cout << tab[ i ] << endl;
```




```
Dlugosc: 3
Max. dlugosc: 10
Przyrost: 3
( 1 + j1 )
( 2 + j2 )
( 3 + j3 )
Dlugosc: 14
Max. dlugosc: 16
Przyrost: 3
( 1 + j1 )
( 2 + j2 )
( 3 + j3 )
( 4 + j4 )
( 5 + j5 )
( 6 + j6 )
( 7 + j7 )
( 8 + j8 )
( 9 + j9 )
( 10 + j10 )
( 11 + j11 )
( 12 + j12 )
( 13 + j13 )
( 14 + j14 )
```

Klasy szablonowe – anatomia prostej klasy

Konstruktory, destruktor i operator przypisania — deklaracje

```
template <class T>
class Array
{
   public:

      // Konstruktor ogólny (i domyślny zarazem)
      Array( int maxSize = 0, int maxDelta = 10 );

      // Konstruktor kopiujący
      Array( const Array &a );

      // Destruktor - tablica jest niszczona
      virtual ~Array();

      // Przeciazony operator przypisania tablic
      Array & operator = ( const Array &a );


   protected:

      //. . .
};
```

Klasy szablonowe – anatomia prostej klasy

Konstruktory, destruktor i operator przypisania — definicje

```
template < class T > Array< T  >::Array( int maxSize, int maxDelta )
{
   init( maxSize, maxDelta );
}

template < class T  > Array< T >::Array( const Array< T > &a )
{
   copy( a );
}

template < class T  > Array< T >::~Array()
{
   clear();
}

template < class T >
Array< T > & Array<T>::operator = ( const Array< T  > &a )
{
   if( this != &a )
   {
      clear();
      copy( a );
   }
   return ( *this );
}
```

Klasy szablonowe – anatomia prostej klasy

Inicjalizacja, czyszczenie, kopiowanie, rozszerzanie — deklaracje funkcji

```
template <class T>
class Array
{
   public:

      //. . .

      // Powiększenie maksymalnego rozmiaru tablicy o delta elementów
      void expand( int tmpDelta );

      // Wyczyszczenie tablicy,staje się pusta
      void clear();


   protected:

      void copy( const Array & a );
      void init( int startSize, int startDelta );

};
```

Klasy szablonowe – anatomia prostej klasy

Inicjalizacja — definicja funkcji

```
template < class T >
void Array< T >::init( int startSize, int startDelta )
{
   arr = 0;
   len = maxLen = delta = 0;
   if( startSize > 0 )
      arr = new T[ maxLen = startSize ];
   if( startDelta >= 0 )
      delta = startDelta;
}
```

Klasy szablonowe – anatomia prostej klasy

Kopiowanie — definicja funkcji

```
template < class T >
void Array< T >::copy( const Array< T > & a )
{
   if( a.maxLen > 0 )
   {
      arr = new T [ maxLen = a.maxLen ];
      for( int i = 0; i < a.maxLen; i++ )
         arr[ i ] = a.arr[ i ];
      len = a.len;
      delta = a.delta;
   }
   else
      clear();
}
```

Klasy szablonowe – anatomia prostej klasy

Czyszczenie — definicja funkcji

```
template < class T >
void Array< T >::clear()
{
   if( arr )
      delete [] arr;
   arr = 0;
   len = maxLen = delta = 0;
}
```

Klasy szablonowe – anatomia prostej klasy

Rozszerzanie — definicja funkcji

```
template <class T>
void Array< T >::expand( int tmpDelta )
{
   if( tmpDelta <= 0 )
      return;

   T * newArr;
   int saveMaxLen = maxLen;
   int saveLen = len;
   int saveDelta = delta;

   newArr = new T[ maxLen + tmpDelta ];

   for( int i = 0; i < maxLen; i++ )
      newArr[ i ] = arr[ i ];

   clear();

   arr = newArr;
   maxLen = saveMaxLen + tmpDelta;
   len = saveLen;
   delta = saveDelta;
}
```

Klasy szablonowe – anatomia prostej klasy

Dodawanie elementów i przeciążony operator indeksowania — deklaracje funkcji

```
template <class T>
class Array
{
   public:

      //. . .

      // Przeciazony operator indeksu
      T & operator [] ( int i );

      // Dopisanie na koniec tablicy
      void add( T item );

      // Zapisanie elementu na pozycji i tablicy —nadpisanie
      void addAt( int i, T item );

      // Zapisanie elementu na pozycji i tablicy —wstawienie
      void insertAt( int i, T item );

      // Skasowanie elementu na pozycji i. Nastepne elementy sa przesuwane.
      void deleteAt( int i );

      //. . .

};
```

Klasy szablonowe – anatomia prostej klasy

Przeciążony operator indeksowania — możliwe definicje funkcji

```
template < class T >
T & Array< T >::operator [] ( int i )
{
   return arr[ i ];
}
```

lub

```
template < class T >
T & Array< T >::operator [] ( int i )
{
   assert( i >= 0 && i < len );
   return arr[ i ];
}
```

lub

```
template < class T >
T & Array< T >::operator [] ( int i )
{
   assert( i >= 0 && i < len );
   assert( arr != 0 );
   return arr[ i ];
}
```

Klasy szablonowe – anatomia prostej klasy

Przeciążony operator indeksowania — możliwe definicje funkcji

```
template < class T >
T & Array< T >::operator [] ( int i )
{
   if( i >= 0 && i < len)
      return arr[ i ];
   else
      return arr[ 0 ];
}
```

lub

```
template < class T >
T & Array< T >::operator [] ( int i )
{
   static T dummy;
   if( arr && ( i >= 0 && i < len ) )
      return arr[ i ];
   return dummy;
}
```

Klasy szablonowe – anatomia prostej klasy

Dodawanie elementów — definicje funkcji

```
template < class T >
void Array< T >::add( T item )
{
   if( len >= maxLen)
      expand( delta );
   arr[ len++ ] = item;
}

template < class T >
void Array< T >::addAt( int i, T item )
{
   if( i < len && arr != NULL )
      arr[ i ] = item;
}
```

Klasy szablonowe – anatomia prostej klasy

Dodawanie elementów — definicje funkcji

```
template < class T >
void Array< T >::insertAt( int i, T item )
{
   if( !arr )
      return;
   if( i >= 0 && i < len )
   {
      if( len == maxLen )
         expand( delta );

      // Przepisanie elementow w dol o jeden. Zaczynam od konca.
      for( int j = len; j >= i; j-- )
         arr[ j + 1 ] = arr[ j ];

      arr[ i ] = item;
      len++;
   }
}
```

Klasy szablonowe – anatomia prostej klasy

Usuwanie elementów — definicje funkcji

```
template < class T >
void Array< T >::deleteAt( int i )
{
   if( i>= 0 && i < len && arr != NULL )
   {
      for( int j = i; j < len - 1; j++ )
         arr[ j ] = arr[ j + 1 ];
      len--;
   }
}
```

Klasy szablonowe – anatomia prostej klasy

Iterowanie po wszystkich elementach

```
template <class T>
class Array
{
   public:

      //. . .

      // Wykonaj funkcje fun dla wszystkich elementow tablicy.
      void doForEach( void ( * fun ) ( T & a ) );

      //. . .

};
```

```
template < class T >
void Array< T >::doForEach( void ( * fun ) ( T & a ) )
{
   for( int i = 0; i < getLen(); i ++ )
        fun( arr[ i ] );
}
```

Klasy szablonowe – anatomia prostej klasy

Iterowanie po wszystkich elementach — przykład wykorzystania

```
void print( Complex & z )
{
   cout << z << endl;
}

void fun()
{
   Array<Complex> tab( 10, 3 );

   for( int i = 0; i < 5; i++ )
      tab.add( Complex( i, i ) );

   tab.doForEach( print );

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

```
( 0 + j0 )
( 1 + j1 )
( 2 + j2 )
( 3 + j3 )
( 4 + j4 )
```

Ćwiczenia 1

- Korzystając z mechanizmu szablonów zaimplementuj podstawowe operacje stosowe (w ramach klasy `Stack`), tak by możliwe było przechowywanie na stosie danych typu **int**, **double** lub **char**:
  - Położenie elementu `x` na wierzchołku stosu.

  void push(`typ_danych` x)
  - Usunięcie ostatnio odłożonego elementu i zwrócenie go jako wartości funkcji.

  `typ_danych` pop()
  - Zwrócenie elementu znajdującego się na wierzchołku stosu bez jego usuwania.

  `typ_danych` topEl()
  - Sprawdzenie czy stos jest pusty.

  bool empty()

Ćwiczenia 2

- Korzystając z mechanizmu szablonów napisz klasę `Kalkulator`, której celem będzie realizacja prostych operacji arytmetycznych dla dwóch liczb całkowitych lub rzeczywistych (reprezentowanych odpowiednio jako `int` lub `double`).
- Klasa powinna posiadać następujące pola (przechowujące wspominane liczby):
  - liczba1
  - liczba2
- Ponadto klasa powinna posiadać **konstruktor ogólny**, umożliwiający inicjalizację wszystkich jej pól.
- Proszę również stworzyć metody **double dodaj(), double odejmij(), double pomnoz(), double podziel()**, które zwrócą wynik odpowiednio dodawania, odejmowania, mnożenia czy dzielenia liczb zapisanych w polach klasy.
- Mechanizmem wyjątków obsłuż błąd dzielenia przez zero.
- Przetestuj działanie stworzonej klasy w swoim programie.