Pierwszy program w C++

```
#include <iostream>

class World
{
public:
    World ()  { std::cout << "Hello!\n"; }
    ~World () { std::cout << "Good bye!\n"; }
};

World TheWorld;

void main () {}
```

Uwagi:

- Każdy program w C++ musi zawierać definicję funkcji **`main`**.
- Program przechowuje informacje w obiektach. Każdy obiekt musi mieć swój typ, zwany klasą.
- Definicja obiektu jest instrukcją, powodującą jego utworzenie.
- Podczas tworzenia obiektu wywoływany jest jego konstruktor, a podczas niszczenia – destruktor.
- Obiekty globalne tworzone są przed rozpoczęciem wykonywania funkcji **`main`** i niszczone po jej zakończeniu, w odwrotnej kolejności.
- Jeżeli funkcja nie zwraca wartości, typ jej wartości definiujemy jako **`void`**.
- Wiersze, rozpoczynające się znakiem #, zawierają dyrektywy preprocesora.

Zakres globalny

```
#include <iostream>

class World
{
public:
    World ()  { std::cout << "Hello!\n"; }
    ~World () { std::cout << "Good bye!\n"; }
};

World TheWorld;

void main () {}
```


Zakres lokalny
Przekazywanie parametrów do konstruktora

```
#include <iostream>

class World
{
public:
    World (int i)             // konstruktor pobiera parametr
    {
        std::cout << "Hello from " << i << ".\n";
    }

    ~World ()
    {
        std::cout << "Good bye.\n";
    }
};

World TheWorld (1);           // obiekt globalny

void main()
{
    World myWorld (2);        // obiekt lokalny
    std::cout << "Hello from main!\n"; // instrukcja w main()
}
```

Uwagi:

- Zapis **`World (int i)`** oznacza, że konstruktor pobiera jeden argument, o nazwie formalnej **`i`**, reprezentujący liczbą całkowitą (ang. **int**eger).

Obiekty mogą zapamiętywać swój stan

```
#include <iostream>

class World
{
public:                       // część publiczna ("interfejs")
  World (int id)
     : _identifier (id)       // preambuła ("lista wartości początkowych")
  {
    std::cout << "Hello from " << _identifier
              << ".\n";       // typowy ciąg odwołań do obiektu std::cout
  }

  ~World ()
  {
    std::cout << "Good bye from "
              << _identifier
              << ".\n";
  }
private:                      // część prywatna ("implementacja")
  const int _identifier;      // obiekt stały ("tylko do odczytu")
};

World TheWorld (1);

int main ()
{
    World myWorld (2);
    for (int i = 3; i < 6; ++i) // pętla "for"
    {
        World aWorld (i);
    }
    World oneMoreWorld (6);
}
```

Pętla `for` a pętla `while`

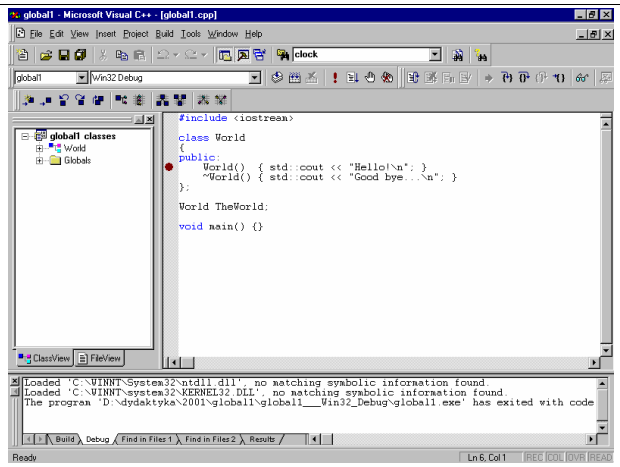
```
...
for (instrukcja_początkowa;
     warunek_kontynuacji;
     krok_końcowy)
{
      instrukcja_1;
      instrukcja_2;
      ...
}
...
```

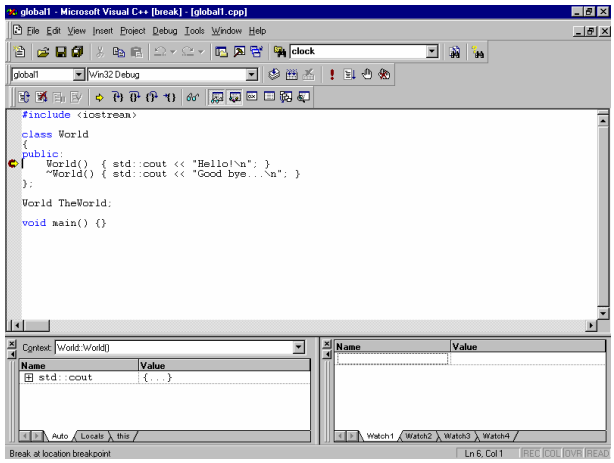
Jest równoważna pętli **`while`:**

```
...
{   // Klamra! Nowy zakres lokalny!!! Tylko w najnowszych kompilatorach C++!!!!!
  instrukcja_początkowa; // Zazwyczaj: definicja zmiennej sterującej
  while(warunek_kontynuacji)
  {
      instrukcja_1;
      instrukcja_2;
      ...
      krok_końcowy;  // krok końcowy wykonywany jest na końcu...
  }
}
...
```

Uwagi:

- Preambuła pętli **`for`** składa się z trzech części oddzielonych średnikami; można pominąć dowolną z nich.
- Pusty warunek kontynuacji pętli **`for`** oznacza, iż się go w ogóle nie sprawdza, a pętlę trzeba przerwać w inny sposób.
- Typowy zapis pętli nieskończonej:
  `for( ; ; ) {...}` // wersja 1
  `while(1) {...}` // wersja 2
- Instrukcje **`for`**, **`while`**, **`do`**, **`if`** i **`else`** działają na pojedynczą instrukcję (zakończoną średnikiem) lub blok instrukcji (w nawiasach klamrowych). Unikniemy wielu kłopotów, konsekwentnie używając klamer.





Obiekty jako składowe innych obiektów

```
#include <iostream>

class Matter
{
public:
  Matter (int id)
    : _identifier(id)
  {
    std::cout << "  Matter for " << _identifier << " created\n";
  }
  ~Matter ()
  {
    std::cout << "  Matter in " << _identifier << " annihilated\n";
  }
private:
  const int _identifier;
};

class World
{
public:
  World (int id)
    : _identifier (id), _matter (_identifier)  // inicjowanie składowych
  {
    std::cout << "Hello from world " << _identifier << ".\n";
  }

  ~World ()
  {
    std::cout << "Good bye from world " << _identifier << ".\n";
  }
private:
  const int     _identifier;
  const Matter  _matter;  // Obiekt składowy (typu Matter)
};

World TheUniverse (1);

int main ()
{
  World myWorld (2);
}
```


Preambuła konstruktora

```
World (int id)
        : _identifier (id), _matter (_identifier)
{
  ...
}
```

W preambule konstruktora podajemy parametry, wykorzystywane podczas konstruowania obiektów składowych, np.

Kolejność inicjowania składowych

O kolejności inicjowania składowych decyduje kolejność ich definiowania w definicji klasy

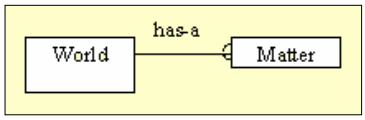
```
private:
   const int     _identifier;  // inicjowana jako pierwsza
   const Matter  _matter;      // inicjowana jako druga
```

~~a nie w preambule~~

```
: _matter (_identifier), _identifier (id)
```

Realacja „A zawiera B”

Jeżeli obiekt A posiada składową klasy B to mówimy, że „A zawiera B”:



Dziedziczenie

```
#include <iostream>

class CelestialBody
{
public:
   CelestialBody (double mass)
      : _mass (mass)
   {
      std::cout << "Creating celestial body of mass " <<  _mass << "\n";
   }

   ~CelestialBody ()
   {
      std::cout << "Destroying celestial body of mass " <<   _mass << "\n";
   }

private:
   const double _mass;
};

class Star: public CelestialBody       // Star jest typu CelestialBody
{
public:
   Star (double mass, double brightness)
      : CelestialBody (mass), _brightness (brightness)
   {
      std::cout << "Creating a star of brightness " <<  _brightness << "\n";
   }

   ~Star ()
   {
      std::cout << "Destroying a star of brightness " <<  _brightness << "\n";
   }

private:
   const double _brightness;
};

int main ()
{
   std::cout << "    Entering main.\n";
   Star aStar ( 1234.5, 0.1 );
   std::cout << "    Exiting main.\n";
}
```


Funkcje składowe („metody")

```
#include <iostream>

class  InputNum
{

// I N T E R F E J S
public:
   InputNum ()
   {
      std::cout << "Enter number ";
      std::cin >> _num;
   }

   int GetValue () const {  return _num; }      // metoda publiczna typu "const"

// I M P L E M E N T A C J A
private:
   int _num;
};

int main()
{
   InputNum num;
   std::cout << "The value is " << num.GetValue() << "\n";
   return 0;                                  // funkcja main() zakończyla się "normalnie"
}
```

Uwagi

- W definicji klasy z reguły umieszcza się definicje funkcji składowych, czyli metod, które mogą być publiczne (*public*), chronione (*protected*) lub prywatne (*private*).
- Metody wywołuje się na rzecz ("*w kontekście*") obiektu. W pewnym sensie obiekt jest jednym z argumentów metody (specyfikowanym "przed kropką").
- "Kropka" (.) jest operatorem udostępniającym składowe i metody obiektu.
- Zbiór metod i składowych publicznych tworzy *interfejs* klasy.
- Zbiór składowych i metod prywatnych tworzy *implementację* klasy.
- Interfejs jest ustalony "raz na zawsze" (ale może ulec rozszerzeniu), implementacja może ulec zmianie w każdej chwili.
- dobrze zaprojektowany i dobrze zaimplementowany obiekt potrafi wypełnić dobrze zdefiniowany *kontrakt*
- Metody zadeklarowane jako stałe (*const*) służą wyłącznie do *odczytu* stanu obiektu.

Upraszczanie zapisu

```
using std::cout;
using std::cin;

// lub nawet tak:

using namespace std;
cout << "teraz prosciej!\n";
```

- Operator :: jest selektorem przestrzeni nazw

Lokalny zakres funkcji składowej

```
#include <iostream>
using std::cout;
using std::cin;

class  InputNum
{
public:
   InputNum ()
   {
      cout << "Enter number ";
      cin >> _num;
   }

   int GetValue () const {  return _num; }

   void AddInput ()
   {
      InputNum aNum;  // get a number from user
      _num = _num + aNum.GetValue ();
   }

private:
   int _num;
};

int main()
{
   InputNum num;
   cout << "The value is " << num.GetValue() << "\n";
   num.AddInput();
   cout << "Now the value is " << num.GetValue() << "\n";
   return 0;
}
```


Literały napisowe a tablice znaków

```
#include <iostream>
using std::cout;
using std::cin;

class  InputNum
{
public:
     InputNum (const char msg []) // msg jest tablicą znaków
     {
          cout << msg;
          cin >> _num;
     }

     int GetValue () const {  return _num; }

     void AddInput (const char msg [])
     {
          InputNum aNum (msg);
          _num = GetValue () + aNum.GetValue ();
     }

private:
     int _num;
};

const char SumString [] = "The sum is   ";

int main()
{
     InputNum num ("Enter number ");
     num.AddInput ("Another one  ");
     num.AddInput ("One more     ");
     cout << SumString << num.GetValue () << "\n";
     return 0;
}
```

- Tablice definiujemy, posługując się nawiasami kwadratowymi.
- Literały znakowe (np. `"Suma"`) są tablicami znaków, zakończonych niewidocznym znakiem `'\0'`. Ilość znaków w napisie `"Suma"` wynosi 5.
- Tak naprawdę funkcje nie otrzymują całej tablicy, przekazanej im jako argument, lecz tylko adres jej pierwszego elementu. Tablica nie zna swojego rozmiaru!
- Jeżeli definiując tablicę od razu podajemy jej elementy, nie musimy definiować jej rozmiaru – kompilator sam ją wyliczy.

Typy wbudowane

W języku C++ rozróżniamy **typy wbudowane** (tj. określone w definicji języka) i **typy zdefiniowane przez użytkownika** (klasy, struktury i wyliczenia).

Typy całkowite:

- **`bool`** – typ logiczny. Dwie wartości: **`true`** (=1) i **`false`** (=0)
- **`char`** – typ znaków.
- **`int`** – podstawowy typ do zapisu liczb całkowitych. Zazwyczaj zajmuje 4 bajty.
- **`short`** – krótsza wersja typu **`int`** (zazwyczaj zajmuje 2 bajty).
- **`long`** – dłuższa wersja typu **`int`** (jednak zazwyczaj mu tożsama).

Nazwy typów całkowitych (wyjątek: **`bool`**) można poprzedzić słowem kluczowym **`unsigned`** powodując, że będą reprezentować liczby nieujemne.

Typy zmiennopozycyjne:

- **`double`** – podstawowy typ liczb zmiennopozycyjnych, reprezentujących liczby rzeczywiste. Zazwyczaj zajmuje 8 bajtów.
- **`float`** – krótsza wersja typu **`double`**. Zazwyczaj zajmuje 4 bajty.
- **`long double`** – dłuższa wersja typu **`double`**. Zazwyczaj jednak jest mu tożsama.

Przykłady:

```
bool b = a < 0, bb = true;
char c = 'a';
int i = 45, szesnastkowo = 0x7fff, osemkowo = 011;
short int si = -34;
long int li = 2345L, dluga = 5641;
unsigned int ui = 34U, uu = 1u;
double x = 1.0, y = 3.4e-12, z = .987;
float f = -34e-4f;
long double q = 2.345L;
```

Typy całkowite i zmiennopozycyjne tworzą **typy arytmetyczne**.
Podczas wykonywania operacji arytmetycznych:

- **`true`** jest zamieniane na **`1`**, **`false`** na **`0`**
- typy „krótsze" od **`int`** są zamieniane na **`int`** (promocja całkowita), przy czym znaki typu **`char`** mają wartość odpowiadającą ich kodom ASCII, np. '0' → 48.
- jeżeli argumenty są różnych typów, to mniej dokładny usiłuje się zamienić na bardziej dokładny; (**`int`** → **`unsigned`** → **`long`** → **`unsigned long`** → **`float`** → **`double`** → **`long double`**).

Reprezentacja liczb zmiennopozycyjnych

$$x = z*m*2^{c}$$

$$z = \pm 1$$

$$1 \le m < 2$$

$$c = w - 2^{(k-1)}$$

$$-2^{(k-1)} \le c < 2^{(k-1)}$$

8 bajtów: `k=11,n=52`
4 bajty: `k=8, n=23`

$$1.0 + 1/2^{53} = 1.0 + 1.11*10^{-16} = 1.0$$

epsilon = min(x): x+1.0 != 1.0
epsilon = 2.22045e-016

Reprezentacja liczb całkowitych

$$d_{B-1}\ d_{B-2}\ \ldots d_2\ d_1\ d_0$$

$$d_i = 0 \text{ lub } 1$$

B : liczba bitów

a) reprezentacja prosta (bez znaku)

$$n = \sum_{i=0}^{B-1} d_i 2^i$$

b) reprezentacja uzupełnieniowa (ze znakiem)

$$n = \sum_{i=0}^{B-2} d_i 2^i - d_{B-1} 2^{B-1}$$

Jeśli B = 8, to 128*2 = 0

Reprezentacje bitowe typów wbudowanych

Literał	Typowa reprezentacja bitowa
`0`	00000000000000000000000000000000
`0.0`	0**00000000000**0000000000000000000000000000000000000000000000000000
`0.0f`	0**00000000**00000000000000000000000
`'0'`	00110000
`'\0'`	00000000
`"0"`	00110000**00000000**
`1`	00000000000000000000000000000001
`'1'`	00110001
`"1"`	00110001**00000000**
`'a'`	01100001
`"a1"`	01100001**00110001**00000000
`1.0f`	0**01111111**00000000000000000000000
`1.0`	0**01111111111**0000000000000000000000000000000000000000000000000000
`0.5`	0**01111111110**0000000000000000000000000000000000000000000000000000
`2.0`	0**10000000000**0000000000000000000000000000000000000000000000000000
`0.3`	0**01111111101**0011001100110011001100110011001100110011001100110011
`-1`	11111111111111111111111111111111
`-0.3`	1**01111111101**0011001100110011001100110011001100110011001100110011
`1.#inf`	0**11111111111**0000000000000000000000000000000000000000000000000000
`-1.#inf`	1**11111111111**0000000000000000000000000000000000000000000000000000
`NaN`	1**11111111111**1000000000000000000000000000000000000000000000000000

Wniosek:
literały `0`, `0.0`, `'0'`, `'\0'` i `"0"` reprezentują zupełnie różne obiekty!

- **`NaN`** = „Not a Number", np. **`sqrt(-1.0)`**
- **`1.#inf`** = „plus nieskończoność", np. **`1.0/0.0`**
- **`-1.#inf`** = „minus nieskończoność", np. **`–1.0/0.0`**

```
int i = 'a'; // i = 97 (97 to kod ASCII litery 'a')
```

Popularna pętla:

```
for (char c = 'a';  c <= 'z'; c++)
{
        rob_cos(c);
}
```

Informacje o właściwościach typów (wbudowanych) można uzyskać z klasy **numeric_limits**, sparametryzowanej interesującym nas typem:

```
#include <limits>
#include <iostream>

int main()
{
   using namespace std;

   cout << "najwieksza liczba typu double =                "
        << numeric_limits<double>::max() << "\n";
   cout << "najwieksza liczba typu long double =           "
        << numeric_limits<long double>::max() << "\n";
   cout << "najmniejsza liczba typu double =               "
        << numeric_limits<double>::min() << "\n";
   cout << "najmniejsza zdenormalizowana liczba typu double = "
        << numeric_limits<double>::denorm_min() << "\n";
   cout << "epsilon dla typu float =                       "
        << numeric_limits<float>::min() << "\n";
   cout << "Ilosc znakow w reprezentacji typu long int =   "
        <<  numeric_limits<long int>::digits << endl;
   cout << "Ilosc znakow w reprezentacji typu unsigned long int = " <<
         numeric_limits<unsigned long int>::digits << endl;
   cout << "czy typ double jest dokladny =                 "
        << numeric_limits<double>::is_exact << "\n";
   cout << "Reprezentacja \"NaN\" dla typu float =      " <<
        numeric_limits<float>::quiet_NaN() << endl;
   cout << "Czy dostepne sa informacje o typie char =  " <<
        numeric_limits<char>::is_specialized << endl;

   return 0;
}
```

```
WYNIKI:
najwieksza liczba typu double =                  1.79769e+308
najwieksza liczba typu long double =             1.79769e+308
najmniejsza liczba typu double =                 2.22507e-308
najmniejsza zdenormalizowana liczba typu double = 4.94066e-324
epsilon dla typu float =                         1.17549e-038
Ilosc znakow w reprezentacji typu long int =     31
Ilosc znakow w reprezentacji typu unsigned long int = 32
czy typ double jest dokladny =                   0
Reprezentacja "NaN" dla typu float =          -1.#IND
Czy dostepne sa informacje o typie char =        1
```


Typedef

Instrukcja **typedef** wprowadza alternatywną nazwę istniejącego typu. Składnia tej instrukcji przypomina składnię deklaracji zmiennych.

Przykłady:

```
unsigned int x;
typedef unsigned int UINT;
UINT y = 90;

int tab[2][4];
typedef int TAB_2_4[2][4];
TAB_2_4 tab2;
tab2[1][3] = 0;

double sin(double);
typedef double (*PFUN)(double);
PFUN f = sin;
std::cout << f(3.141) << '\n'; // == sin(3.141)
```

Zwyczajowo nazwy typów, wprowadzanych instrukcją **typedef**, zapisuje się DUŻYMI LITERAMI

Wyliczenia

Wyliczenie przechowuje zbiór wartości całkowitych, podanych przez programistę, lub zastępuje "const int".

```
enum Kolor {bialy, zielony, czerwony, czarny};
enum {E_min = -100, E_max = 1000};

//tryby otwarcia pliku z klasy std::ios
enum _Openmode {in = 0x01, out = 0x02, ate = 0x04,  app = 0x08, trunc = 0x10,
                binary = 0x20};
ofstream F("mój_plik", std::ios::app +std::ios::binary);

bool ladny_kolor(Kolor k);
if (ladny_kolor(samochod.Kolor())
   Zakochaj_się_w_kierowcy(s1);

for (int i = E_min; i < E_max; i++)
      rob_cos(i);
```


Operatory

Operatory charakteryzują się:
- Priorytetem
- Łącznością
- Wartością (niekiedy)

Wybrane operatory

Operator	Opis	Wartość	Operator	Opis	Wartość
::	Zasięg		<<	przesun. bitów (L)	
.	Wybór składowej		>>	przesun. bitów (P)	
->	Wybór składowej		==	czy równe	false/
[]	Indeksowanie tab.		!=	czy różne	true
++	x++	x	&	bitowe AND	
--	x--	x	^	bitowe XOR	
()	wyw. funkcji		|	bitowe OR	
sizeof	rozmiar				
++	++x	x+1	&&	logiczne "i"	
--	--x	x-1	||	logiczne "lub"	
~	negacja bitowa		? :	warunek	
!	negacja log.		=	p	
-	-x		*=	r	
+	+x		/=	z	
&	adres		%=	y	
*	wyłuskanie		+=	p	
new	przydział pamięci	adres	-=	i	
delete	zwoln. pamięci		<<=	s	
*	mnożenie		>>=	a	
/	dzielenie		&=	n	
%	reszta z dzielenia		|=	i	
+	dodawanie		^=	a	
-	odejmowanie		throw	wyjątek	

Niebezpieczne wyrażenia i instrukcje:

```
v[i] = i++;
f(i, i++);
f(i++) + f(i--);
f( (i, i++) );
v[i,j];
if (a = 0);
```


Instrukcje sterujące w C++

Instrukcja if

```
if (warunek)
     instrukcja_1;
else
     instrukcja_2;
```

Zapis **if (x)** oznacza **if (x != 0)**

Instrukcję

```
if (a < b) max = b;
else max = a;
```

Można zapisać jako

```
max = a < b ? b : a;
```

Instrukcja switch

```
int k;
...
switch(k)
{
case 0:
     f1();
     break;
case 1:
     {
        int j = k*k;
        f2(j);
        break;
     }
default:
     std::cerr << "nieoczekiwana wartość k!\n";
     exit(2);
}
```

Uwagi:
- W instrukcji **switch** nie wolno zapominać o instrukcjach **break**!
- Jako selektora instrukcji **switch** często używa się zmiennej wyliczeniowej.
- Instrukcji **goto** nie używamy!

Instrukcje *break* i *continue* w pętlach

```
// wczytujemy z pliku liczby naturalne i wyznaczamy sumę odwrotności
// tych z nich, które są liczbami pierwszymi

std::ifstream F("liczby.dat");
double suma = 0.0;

for( ; ; )
{
    int n;
    F >> n;
    if (!F)
        break;
    if (!isPrime(n))
        continue;
    suma += 1.0/n;
}
std::cout << "suma = " << suma << "\n";
```

- Instrukcja **break** przerywa działanie (wewnętrznej) pętli.
- Instrukcja **continue** przerywa bieżący krok (wewnętrznej) pętli.
- Pętle przerywa się też w inny sposób, np.:
  - instrukcją **return** (zakończenie działania funkcji),
  - instrukcją **exit** (zakończenie całego programu),
  - instrukcją **throw** (zgłoszenie wyjątku)

Referencje

```
int i = 5;
int j = 10;
int &iref = i;  // iref jest referencją do i
iref = j;       // równoważne instrukcji i = j
```

- Referencje to "inne nazwy" istniejących obiektów.
- Referencje wprowadzają relację "A ma dostęp do B".
- Referencje muszą być zainicjowane i nie mogą ulec zmianie.
- Referencje są standardowym mechanizmem przekazywania obiektów jako argumentów funkcji.; natomiast typy wbudowane standardowo przekazuje się **przez wartość**.

Przykład:

```
class Dual
{
    void ByValue (int j)
    {
        ++j;
        cout << j << endl;
    }

    void ByRef (int & j)
    {
        ++j;
        cout << j << endl;
    }
};

void main ()
{
    Dual dual;
    int i = 1;
    dual.ByValue (i);
    cout << "After calling ByValue, i = " << i << endl;
    dual.ByRef (i);
    cout << "After calling ByRef, i = " << i << endl;
}
```


Stos liczb całkowitych i iterator stosu

Interfejs ("stack2.h")

```
#if !defined STACK2_H
#define STACK2_H

const int maxStack = 16;

class IStack
{
    friend class StackIterator; // udostępniono składowe prywatne
public:
    IStack (): _top (0) {}
    void Push (int i);
    int  Pop ();
private:
    int _arr [maxStack];
    int _top;
};

class StackIterator
{
public:
    StackIterator (IStack const & stack);
    bool AtEnd () const;
    void Advance ();
    int GetNum () const;
private:
    IStack const & _stack;   // referencja do stosu
    int    _iCur;    // bieżący indeks do stosu
};

#endif
```


Implementacja stosu ("stack.cpp")

```
#include "stack2.h"
#include <cassert>
#include <iostream>
using std::cout;
using std::endl;

void IStack::Push (int i)
{
    // Czy nie przepełniamy stosu?
    assert (_top < maxStack);
    _arr [_top] = i;
    ++_top;
}

int IStack::Pop ()
{
    // Nie wolno pobierać liczb z pustego stosu
    assert (_top > 0);
    --_top;
    return _arr [ _top ];
}
```

implementacja iteratora stosu i funkcja main ("stack.cpp" - c.d.)

```
StackIterator::StackIterator (IStack const & stack)
    : _iCur (0), _stack (stack) // inicjacja referencji
{}

bool StackIterator::AtEnd () const
{
    return _iCur == _stack._top; // friend: ma dostep do _top
}

void StackIterator::Advance ()
{
    assert (!AtEnd());
    ++_iCur;
}

int StackIterator::GetNum () const
{
    return _stack._arr [_iCur]; // friend: ma dostep do _arr
}

void main ()
{
    IStack TheStack;
    TheStack.Push (1);
    TheStack.Push (2);
    TheStack.Push (3);

    for (StackIterator seq (TheStack);
         !seq.AtEnd();
         seq.Advance() )
    {
        cout << "    " << seq.GetNum () << endl;
    }
}

/* WYNIKI
    1
    2
    3
*/
```


Zagnieżdżanie klas

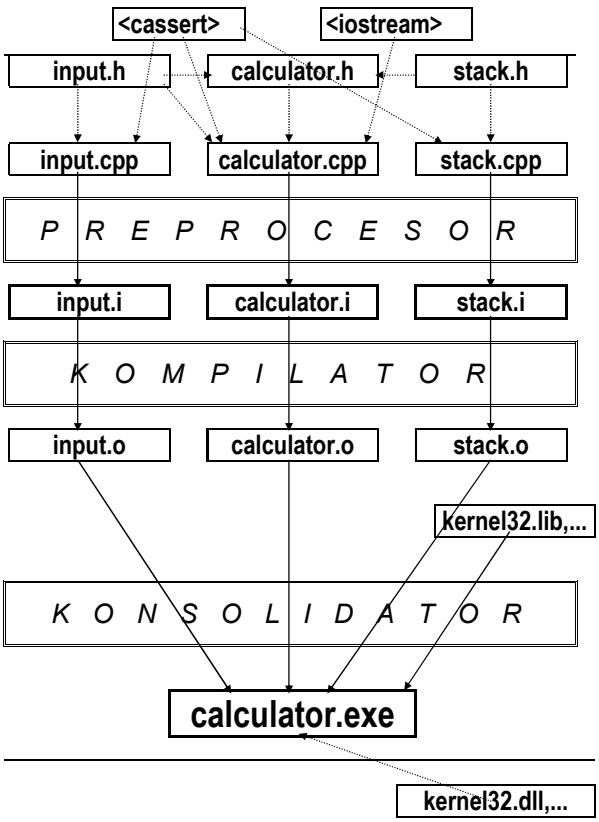
```
const int maxStack = 16; // pojemność stosu

class IStack
{
public:
  // publiczna klasa zagnieżdżona w klasie IStack
    class iterator
    {
    public:
        iterator (IStack const & stack);
        bool AtEnd () const;
        void Advance ();
        int GetNum () const;
    private:
        IStack const & _stack;   // referencja do stosu
        int   _iCur;    // bieżący indeks do stosu
    };

  // klasie IStack::iterator udostępniamy składowe prywatne klasy IStack
    friend class IStack::iterator;

    IStack (): _top (0) {}
    void Push (int i);
    int  Pop ();

private:
    int _arr [maxStack];
    int _top;
};

// ...

// implementacja metod klasy IStack - bez zmian!
```


Implementacja iteratora

```
IStack::iterator::iterator (IStack const & stack)
    : _iCur (0), _stack (stack)
{}

bool IStack::iterator::AtEnd () const
{
    return _iCur == _stack._top;   // friend: ma dostep do _top
}

void IStack::iterator::Advance ()
{
    assert (!AtEnd());
    ++_iCur;
}

int IStack::iterator::GetNum () const
{
    return _stack._arr [_iCur]; // friend: ma dostep do _arr
}

void main ()
{
    IStack TheStack;
    TheStack.Push (1);
    TheStack.Push (2);
    TheStack.Push (3);

    for (IStack::iterator iter (TheStack);
         !iter.AtEnd();
         iter.Advance() )
    {
        cout << "    " << iter.GetNum () << endl;
    }
}
```

- **Iterator to obiekt, służący do przeglądania zawartości innych obiektów określonej klasy**

Kompilacja programu wieloplikowego



Preprocesor

```
#ifndef PREPROC_H__
# define PREPROC_H__
# include <climits>
# include <iostream>

# define SIZE 10
  //lepiej:  const int size = 10;

# define MAX(x,y) ((x)>(y)? (x) : (y))
  //lepiej:
  inline int     Max (int x, int y)      { return x>y ? x : y; }
  inline double  Max (double x, double y) { return x>y ? x : y; }

# ifdef WIN32
    const int N = 100;
# elif defined (__OSF1__)
    const int N = 1000000;
# else
#  error unknown platform!
# endif

# if INT_MAX == 0x7fff
  const int INT_BITS = 2;
# endif

# ifdef NDEBUG
#  define assert(p)
# else
#  define assert(p) \
  if (p) {  std::cout << "Assertion failed: " << #p  \
        << ", file " << __FILE__ << ", line " <<  __LINE__ << "\n"; \
     exit(1); \
  }
# endif
#endif

int main( )
{
   int a[SIZE][N];
   assert (MAX(1,2) == 2);
   return 0;
}
```


Typy parametryczne (szablony)

- **Typy parametryczne to typy, których argumenty podajemy w nawiasach ostrokątnych.**

```
#if !defined STACK_H
#define STACK_H

#include <cassert>

const int maxStack = 16;

template <class T>
class Stack
{
public:
   Stack (): _top (0) {}
   void Push (T const& i );
   T  Pop ();
private:
   T _arr [maxStack];
   int _top;
};

template <class T>
void Stack<T>::Push ( const T& i )
{
  assert ( _top < maxStack );
   _arr [ _top ] = i;
   ++_top;
}

template <class T>
T Stack<T>::Pop ()
{
   assert ( _top > 0 );
   --_top;
   return _arr [ _top ];
}
#endif
```


Test szablonu (main.cpp)

```
#include "stack.h"
#include <iostream>

struct Para
{
        int x;
        double z;
        Para(int x0, double z0)
                : x(x0), z(z0)
        { }
        Para() { }
        Para(const Para& p)
                : x(p.x), z(p.z)
        { }
};


int main()
{
        Stack<Para> stos;

        for (int i = 1; i < 5; i++)
                stos.Push(Para(i, 1.0/i));

        for(int j = 1; j < 5; j++)
                std::cout << stos.Pop().z << " ";

        std::cout << "\n";

        return 0;
}
```


Liczby zespolone

```
#include <iostream>
#include <complex>
#include <string>
#include <cmath>

using namespace std;

int main()
{
        const complex<double> i = complex<double>(0,1.0);
        const double pi = 2.0*acos(0.0);
        string hello = "to jest program demonstrujacy ";
        hello += "mozliwosci szablonow biblioteki standardowej\n";

        cout << hello + "exp(i*pi) = " << exp(pi*i) << "\n";

        return 0;
}

/* WYNIK:
to jest program demonstrujacy mozliwosci szablonow biblioteki
standardowej
exp(i*pi) = (-1,1.22461e-016)
*/
```

Podstawowe metody klasy complex<class T>:

- Operatory arytmetyczne (`+`, `-`, `*`. `/`), przypisania (`+=`, `-=`, `*=`, `/=`, `=`), relacyjne (`==`, `!=`).
- Automatyczna konwersja z typu podstawowego (T), np. `complex<double> = 2.0;`.
- Składowe `imag()` i `real()`.

Szablon *vector<T>*

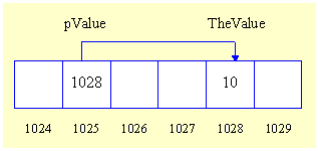
```
#include <iostream>
#include <complex>
#include <vector>

using namespace std;

int main()
{
        typedef vector<int> kolumna;
        typedef vector<kolumna> macierz;

        cout << "podaj rozmiar macierzy: ";
        int n;
        cin >> n;
        macierz m(n);
        int i;
        for (i = 0; i < n; i++)
                m[i].resize(n);

        for (i = 0; i < n; i++)
                for (int j = 0; j < n; j++)
                        m.at(i).at(j) = j*i;

        double suma = 0.0;
        for (i = 0; i < n; i++)
                for (int j = 0; j < n; j++)
                        suma += m[i][j];

        cout << "suma = " << suma << "\n";
        return 0;
}
```

Podstawowe metody klasy vector<T>:

- `at(int)`, `operator[](int i)`, `size()`, `resize(int)`, `reserve(int)`, `capacity()`, `empty()`, `push_back(T)`, `pop_back()`, `erase(...)`, `clear()`, `swap(vector)`.
- Uwaga: `at(int i)` sprawdza poprawność indeksu, a `operator[]` – nie sprawdza!

Wskaźniki

```
int *pValue;         // definicja zmiennej wskaźnikowej
int TheValue = 10;   // definicja "zwykłej zmiennej"
pValue = &TheValue;  // przypisanie adresu (operator &)
```




```
int i = *pValue;    // i = 10
*pValue = 20;       // TheValue = 20
pValue = &i;        // *pValue = 10
*pp2 = &pValue      // pp2 jest wskaźnikiem do wskaźnika
```

- **Wskaźniki umożliwiają uzyskanie pośredniego dostępu do obiektów.**
- **W przeciwieństwie do zmiennych referencyjnych, zmienne wskaźnikowe mogą zmieniać swoją wartość (wskazywać w różnym czasie na różne obiekty).**
- **Operator & (ampersand) służy do pobierania adresu dowolnego obiektu.**
- **Operator * (gwiazdka) służy do wyłuskania wartości obiektu. wskazywanego przez wskaźnik.**
- **Można definiować wskaźniki do wskaźników.**
- **Wskaźniki powinno się inicjować w punkcie definicji, np.:**

```
int *pInt = &i; // definicja + inicjacja wskaźnika pInt
```


Wskaźniki a referencje

```
int TheValue = 10;
int& aliasValue = TheValue; // aliasValue jest referencją do TheValue
int i = aliasValue;  // odczytujemy wartość TheValue poprzez referencję
aliasValue = 20;     // modyfikujemy TheValue poprzez referencję
```

- **Referencje są wygodniejsze i bezpieczniejsze w użyciu**

Operator wyłuskania składowej ("–>")

```
IStack TheStack;
IStack* pStack = &TheStack;
pStack->Push (7);   // umieść na stosie liczbę 7 poprzez wskaźnik pStack
                    // równoważne instrukcji (*pStack).Push (7)
```

to samo przy pomocy referncji:

```
IStack TheStack;
IStack& aliasStack = TheStack;
aliasStack.Push (7);
```

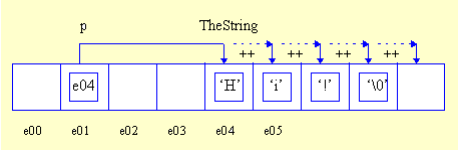
- **Operator `->` służy do wyłuskiwania składowych obiektów, do których odwołujemy się poprzez wskaźnik.**
- **Zapis `p->x` jest równoważny (niewygodnemu) zapisowi `(*p).x`.**
- **Operator -> w wyrażeniach często występuje sekwencyjnie (np. `p->Next()->Value()`.**

Wskaźniki a tablice

```
int main ()
{
    char aString [] = "dlugi napis";
    int len = StrLen (aString); // StrLen zwraca dlugosc napisu
    std::cout << "Dlugosc napisu " << aString
              << " wynosi " << len << std::endl;
}
```

Tablicowa implementacja funkcji StrLen

```
int StrLen (char const str [] )
{
    int i;
    for (i = 0; str [i] != '\0'; ++i)
        continue;
    return i;
}
```



Wskaźnikowa implementacja funkcji StrLen

```
int StrLen (char const * pStr)
{
    char const * p = pStr;
    while (*p++);       // ← niezwykle urocza pętla while
    return p - pStr - 1;
}
```

- **Wyrażenie `*p++` jest równoważne wyrażeniu `*(p++)`.**

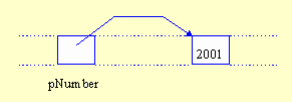
Nie zastępuj wskaźnikami operatora [ ]

Dynamiczny przydział pamięci

```
int * pNumber = new int;
```

lepiej:

```
int * pNumber = new int (2001);
```



Konstrukcja obiektów dynamicznych

```
IStack* pStack = new IStack; // konstruktor bezargumentowy
Star* pStar = new Star (1234.5, 10.2); // k-r dwuargumentowy
```

Dynamiczne tablice

```
int* pNumbers = new int [n];
for (int i = 0; i < n; ++i)
    pNumbers [i] = i * i;
IStack* aStack = new IStack [4];
```

Niszczenie obiektów dynamicznych

```
delete pNumber;      // zwolnienie pamięci
delete pStar;        // "Destroying a star of brightness ..."
delete [] pNumbers;  // destrukcja tablicy
```

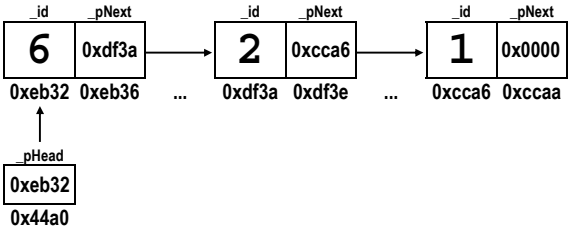
- operator `new` przydziela obszar pamięci na stercie i zwraca jej adres.
- operator `new ... [n]` przydziela pamięć na n-elementową tablicę.
- operatory `delete` i `delete[]` służą do zwalniania pobranej pamięci.
- operatory `new`/`delete` wywołują odpowiednie konstruktory/destruktory.
- Za zarządzanie pamięcią dynamiczną odpowiada programista.
- Nie można jawnie skonstruować elementów tablicy, tworzonej dynamicznie

Lista powiązana

Klasa Link

```
class Link  // ogniwo listy
{
public:
    Link (Link* pNext, int id)
        : _pNext (pNext), _id (id) {}
    Link *  Next () const { return _pNext; }
    int     Id () const { return _id; }
private:
    int     _id;
    Link *  _pNext;
};
```

- `Link` jest przykładem rekurencyjnej struktury danych.
- Można definiować wskaźnik lub referencję do obiektu klasy, której (pełnej) definicji jeszcze nie podano.

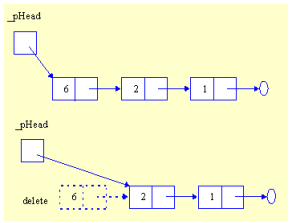


Klasa List

```
class List
{
public:
    List (): _pHead (0) {}
    ~List ();
    void Add ( int id );
    Link const * GetHead () const { return _pHead; }
private:
    Link* _pHead;
};
```


Destruktor – wariant iteracyjny

```
List::~List ()
{
  // usuwamy z listy wszystkie elementy ("ogniwa")
    while ( _pHead != 0 )
    {
        // w pLink zapamiętujemy położenie pierwszego ogniwa
        Link* pLink = _pHead;
        // odłączamy obiekt, wskazywany przez pLink
        _pHead = _pHead->Next();
        // usuwamy ze sterty niepotrzebne ogniwo
        delete pLink;
    }
}
```

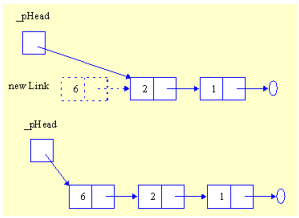


Destruktor – wariant rekurencyjny

```
List::~List ()
{
    delete _pHead; // _pHead jest wskaźnikiem do obiektu klasy Link
}

Link::~Link ()
{
    delete _pNext; // _pNext jest wskaźnikiem do obiektu klasy Link
}
```

- Rekurencja prowadzi do prostszej implementacji, ale może doprowadzić do spowolnienia programu i nadmiernego obciążenia pamięci operacyjnej.

Dodawanie nowego elementu do listy




```
void List::Add ( int id )
{
    // dodajemy nowe ogniwo na początku listy
    Link * pLink = new Link (_pHead, id);
    // uaktualniamy wartość wskaźnika do pierwszego elementu listy
    _pHead = pLink;
}
```

Przeglądanie listy

```
for (Link const * pLink = list.GetHead();
     pLink != 0;
     pLink = pLink->Next ())
{
    if (pLink->Id() == id)
    {
        // na liście znaleziono liczbę o wartości id
        // rób_coś;
        break;
    }
}
```

- Uwaga: funkcja `GetHead()` zwraca wskaźnik do stałej:

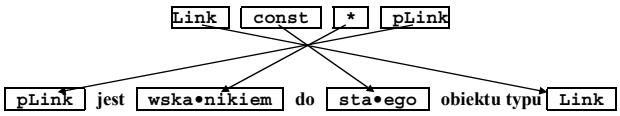
  `Link const * Link::GetHead () const {return _pHead;}`

  dzięki czemu klient tej metody (tu: pętla for) nie może zmienić stanu listy!

Wskaźniki stałe i do stałej

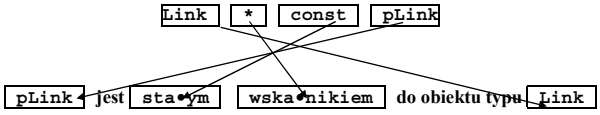
- **Wskaźnik do stałej służy do odczytu stanu wskazywanego przez siebie obiektu**

```
Link const *pLink;
lub:
const Link *pLink;
```



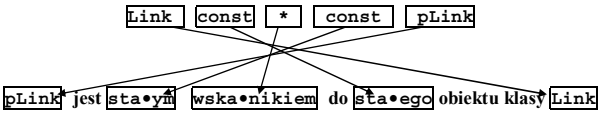
- Wskaźnik stały zawsze wskazuje na ten sam obiekt (jak referencja)

```
Link *const pLink;
```

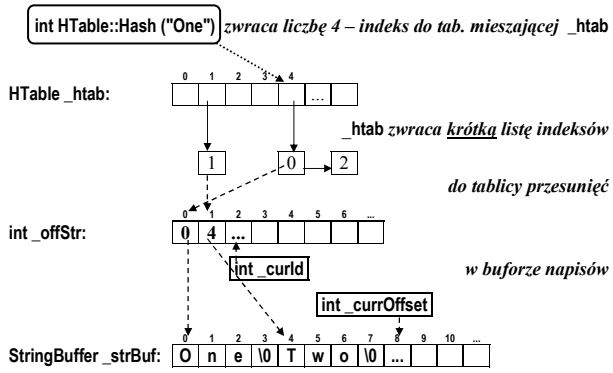


- Stały wskaźnik do stałej przypomina referencję do stałej

```
Link const *const pLink;
```



Tablica napisów



Klasa StringTable

```
const int idNotFound = -1;

const int maxStrings = 100;

// Klasa StringTable odwzorowuje napisy w kolejne liczby całkowite,
// a kolejne liczby całkowite w napisy

class StringTable
{
public:
      StringTable ();
      int ForceAdd (char const * str);
      int Find (char const * str) const;
      char const * GetString (int id) const;
private:
      HTable          _htab;     // napis -> krótka lista liczb całkowitych
      // _offStr: liczba całkowita --> przesunięcie ("offset")
      int             _offStr [maxStrings];
      int             _curId;
      StringBuffer    _strBuf;  // przesunięcie (tzw. "offset") --> napis
};
```


Klasa StringTable

```
StringTable::StringTable ()
    : _curId (0)
{}


// Bezwarunkowo dodaj do tablicy napisów napis str i zwróć jego położenie
int StringTable::ForceAdd (char const * str)
{
  int len = strlen (str);

  // czy w buforze jest jeszcze miejsce na kolejny napis?
  if (_curId == maxStrings || !_strBuf.WillFit (len))
      return idNotFound;

  // zapisujemy info o miejscu, w którym przechowywany jest napis str
  _offStr [_curId] = _strBuf.GetOffset ();
  _strBuf.Add (str);

  // dodajemy do tablicy mieszającej info o odwzorowaniu str w _curId
  _htab.Add (str, _curId);
  ++_curId;
  return _curId - 1;  // zwracamy indeks do tablicy przesunięć
}
```



```
// Czy i gdzie w tablicy napisów znajduje się napis str
int StringTable::Find (char const * str) const
{
      // Z tablicy mieszającej pobieramy krótką listę liczb całkowitych
      List const & list = _htab.Find (str);

      // Przeglądamy kolejne elementy listy
      for ( Link const * pLink = list.GetHead ();
            pLink != 0;
            pLink = pLink->Next () )
      {
            int id = pLink->Id ();
            int offStr = _offStr [id];
            if (_strBuf.IsEqual (offStr, str))
                  return id;
      }
      return idNotFound;
}


// f-cja GetString odwzorowuje liczbę całkowitą id w napis.
char const * StringTable::GetString (int id) const
{
      assert (id >= 0);
      assert (id < _curId);
      int offStr = _offStr [id];
      return _strBuf.GetString (offStr);
}
```

Klasa StringBuffer (bufor napisów)

```
const int maxBufSize=500;

class StringBuffer
{
public:
   StringBuffer () : _curOffset (0)
   { }

   // czy w buforze jest jeszcze miejsce na napis o długości len znaków
   bool WillFit (int len) const
   {
      return _curOffset + len + 1 < maxBufSize;
   }

   // dodaj do bufora napis str
   void Add (char const * str)
   {
      // alternatywny zapis: strcpy (_buf + _curOffset, str);
      strcpy (&_buf [_curOffset], str);
         _curOffset += strlen (str) + 1;
   }

   // zwróć pozycję pierwszego wolnego znaku w buforze
   int GetOffset () const
   {
      return _curOffset;
   }

   bool IsEqual (int offStr, char const * str) const
   {
      // char const * strStored = _buf + offStr;
      char const * strStored = &_strBuf [offStr];
      // jeżeli napisy są takie same, funkcja strcmp zwraca 0
      return strcmp (str, strStored) == 0;
   }

   char const * GetString (int offStr) const
   {
      return &_buf [offStr]; //alternatywnie: return _buf + offStr;
   }

private:
   char  _buf [maxBufSize];
   int   _curOffset;
};
```


Dygresja: Algorytmy tablicowe

Algorytmy tablicowe zazwyczaj służą do przyspieszenia działania programu kosztem użycia dodatkowej pamięci operacyjnej.

Problem: jak sprawdzić, czy znak c reprezentuje literę?

- **Wersja krótka, ale nieefektywna:**

```
inline bool IsDigitSlow (char c)
{
   return c >= '0' && c <= '9';
}
```

- **Wersja dłuższa, ale bardziej efektywna**

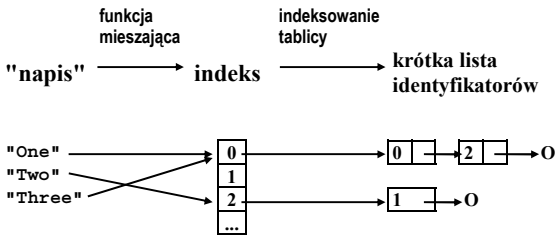
```
class CharTable
{
public:
   CharTable ();
   bool IsDigit (unsigned char c) { return _tab [c]; }
private:
   // def. stałej UCHAR_MAX znajduje się w pliku <climits>
   bool _tab [UCHAR_MAX + 1];
};

CharTable::CharTable ()
{
   for (int i = 0; i <= UCHAR_MAX; ++i)
   {
      // tu stać nas na nieefektywny algorytm
      _tab [i] = i >= '0' && i <= '9';
      // if (i >= '0' && i <= '9')
      //  _tab [i] = true;
      // else
      //  _tab [i] = false;
   }
}

CharTable TheCharTable;
```

- **Teraz możemy sprawdzić, czy zmienna c reprezentuje literę:**

```
TheCharTable.IsDigit (c)
```


Tablica mieszająca

Załóżmy, że
"One" ma identyfikator 0
"Two" ma identyfikator 1
"Three" ma identyfikator 2



Funkcja mieszająca przyporządkowuje napisom "One" i "Three" ten sam indeks 0. Aby poradzić sobie z powstałą w ten sposób kolizją, zerowy element tablicy mieszającej zawiera krótką listę, w której zapisano identyfikatory obu napisów (czyli dwie liczby całkowite, 0 i 2)

Tablica mieszająca – implementacja

```
const int sizeHTable = 127;

class HTable
{
public:
   // funkcja Find zwraca krótką listę "kandydatów"
   List const & Find (char const * str) const;
   // umożliwia dodanie kolejnego odwzorowania napisu str w liczbę id
   void Add (char const * str, int id);
private:
   // funkcja mieszająca
   int hash (char const * str) const;

   List _aList [sizeHTable]; // tablica (krótkich) list
};

// Funkcja HTable::Find znajduje w tablicy mieszającej listę,
// która może (ale nie musi) zawierać identyfikator napisu str.

List const & HTable::Find (char const * str) const
{
      int i = hash (str);
      return _aList [i];
}

void HTable::Add (char const * str, int id)
{
      int i = hash (str);
      _aList [i].Add (id);
}
```

Funkcja mieszająca

```
int HTable::hash (char const * str) const
{
      // argumentem tej funkcji nie może być napis pusty!
      assert (str != 0 && str [0] != 0);

      unsigned int h = str [0];
      for (int i = 1; str [i] != 0; ++i)
            h = (h << 4) + str [i];
      return h % sizeHTable; // wyznacz resztę z dzielenia
}
```

Działanie funkcji mieszającej dla napisu "One"			
znak	kod ASCII (szesnastkowo)	h (szesnastkowo)	h (dziesiętnie)
'O'	0x4F	0x4F	72
'n'	0x6E	0x55E = 0x4F0 + 0x6E	1374
'e'	0x65	0x5645 = 0x55E0 + 0x65	22085
"One" zostanie odwzorowany w liczbę 22085 %127 == 114			

Test tablicy napisów

```
int main ()
{
   StringTable strTable;
   strTable.ForceAdd ("One");
   strTable.ForceAdd ("Two");
   strTable.ForceAdd ("Three");

   int id = strTable.Find ("One");
   cout << "One at " << id << endl;
   id = strTable.Find ("Two");
   cout << "Two at " << id << endl;
   id = strTable.Find ("Three");
   cout << "Three at " << id << endl;
   id = strTable.Find ("Minus one");
   cout << "Minus one at " << id << endl;

   cout << "String 0 is " << strTable.GetString (0)
        << endl;
   cout << "String 1 is " << strTable.GetString (1)
        << endl;
   cout << "String 2 is " << strTable.GetString (2)
        <<endl;
}
```


Argumenty linii komend

```
#include <iostream>

int main(int argc, char* argv[], char* envp[] )
{
   std::cout << "Nazwa programu: " << argv[0] << "\n";

   std::cout << "\nArgumenty linii komend: ";
   if (argc == 1) cout << "nie podano\n";
   else           cout << argc - 1 << "\n";

   for (int i = 1; i < argc; i++)
      cout << "arg[" << i << "]: \""
           << argv[i] << "\"\n";

   std::cout << "\nZmienne srodowiskowe:\n";
   for (int j = 0; envp[j] != 0; j++)
      cout << envp[j] << "\n";

   std::cout << "\nZmienna srodowiskowa PATH:\n";

   return 0;
}
```

```
/* WYNIK:
Nazwa programu:
D:\DYDAKT~1\2001\MAIN_A~1\DEBUG\MAIN_A~1.EXE

Argumenty linii komend: 2
arg[1]: "-o"
arg[2]: "ola"

Zmienne srodowiskowe:
COMSPEC=C:\WINDOWS\COMMAND.COM
SBPCI=C:\SBPCI
winbootdir=C:\WINDOWS
BLASTER=A220 I7 D1 H7 P330 T6
windir=C:\WINDOWS
SHELL=C:\WINDOWS\COMMAND.COM
PATH=c:\djgpp\bin;c:\windows;c:\windows\command;

Zmienna środowiskowa PATH:
path = c:\djgpp\bin;c:\windows;c:\windows\command;
*/
```


W wywołaniu funkcji

```
int main(int argc, char* argv[], char* envp[] )
```

- `argc + 1` przekazuje ilość argumentów linii komend
- `argv i envp` to tablice napisów
- `argv[0]` zawiera nazwę programu
- `argv[n]` to kolejne argumenty linii komend
- `argv[argc] == (char*) 0;` (nadmiarowość informacji)
- `envp []` Kolejne elementy zawierają wartości kolejnych zmiennych środowiskowe (jako napisy); Ostatnim elementem `envp` jest adres zerowy
- `int main(...)` Funkcja `main()` zwraca kod wykonania programu (0 oznacza normalne zakończenie programu)

Funkcję `main` można wywołać z 0, 1, 2 lub 3 standardowymi argumentami, np.

```
int main ()
int main (int argc, char* argv[])
int main (int argc, char* argv[], char* envp[] )
```


Funkcje rekurencyjne

Problem: napisać program, wyznaczający ilość k-elementowych kombinacji zbioru n-elemntowego.

Plik testowy ("test.cpp")

```
#include <iostream>
#include <cstdlib>
#include "kombin.h"

int main(int argc, char* argv[])
{
   if (argc != 3)
   {
      std::cerr << "Program " << argv[0]
                << "\nwyznacza ilosc kombinacji k-elementowych "
                << "w zbiorze n-elementowym\n";
      std::cerr << "\n*** UWAGA! Program uruchomiono z nieprawidlowa"
                << " iloscia argumentow! ***\n";
      std::cerr << "Prawidlowy sposob uzycia:\n" << argv[0] << " n k\n\n";
      return 1;
   }
   int n = atoi(argv[1]);
   int k = atoi(argv[2]);

   if (n < 0 || n > MaxN || n < k ||  k < 0)
   {
      std::cerr << "Bledne dane wejsciowe!\n";
      return 2;
   }

   std::cout << n << " po " << k << " = " << Newton(n,k) << "\n";
   return 0;
}
```

Interfejs ("newton.h")

```
#ifndef NEWTON_H_
#define NEWTON_H_

const int MaxN = 50;

int Newton(int n, int k);

#endif
```

Rozwiązanie powolne – bezpośrednia rekurencja

```
#include <cassert>
#include "kombin.h"

// ilość kombinacji k-elementowych w zbiorze n-elementowym
// zastosowano nieefektywny algorytm rekurencyjny

int Newton(int n, int k)
{
    // warunek zakończenia rekurencji
    if (k == 0 || k == n)
       return 1;

    // sprawdzamy, czy nie robimy jakiegoś głupstwa
    assert (k >= 1 && n >= k && n <= MaxN);

    // rekurencyjne wywołanie funkcji kombin
    int wynik = Newton(n-1,k) + Newton(n-1,k-1);

    // sprzwdzamy, czy w powyższym dodawaniu nie przekroczono zakresu
    if (wynik < 0)
    {
       throw 1; // THROW!! Błąd wewnętrzny nr 1 – przekroczenie zakresu?
    }
    return wynik;
}
```

- **Funkcja rekurencyjna to funkcja, która wywołuje sama siebie (bezpośrednio lub pośrednio).**
- **Implementując funkcję rekurencyjną należy zadbać, by wywołania rekurencyjne nie mogły następować w nieskończoność.**
- **Funkcje rekurencyjne potrafią być bardzo czasochłonne i nieefektywne – np. w powyższej implementacji obliczenia dla `n` = 20 i `k` = 10 powodują wygenerowanie 369 511 wywołań funkcji `Kombin`, a dla `n` = 30 i `k` = 15 ilość ta wynosi 310 235 039!**
- **W języku C++ użycie operatora `throw` stanowi podstawową metodę sygnalizowania błędów, wykrytych podczas wykonywania programu – ale o tym później! Na razie proszę zakładać, że `throw` służy do awaryjnego zakończenia działania programu.**

Rozwiązanie szybkie : algorytm hybrydowy, rekurencyjno-tablicowy

```
#include <cassert>
#include "newton.h"

// Globalną tablicę ukrywamy w nienazwanej przestrzeni nazw
namespace
{
   // deklaracja i inicjacja zerami 2-wymiarowej tablicy t
   // wartość 0 oznacza w tym algorytmie "wartość nieznaną"
   // Stałą MaxN zadeklarowano w "newton.h"
   int t[MaxN][MaxN/2] = {0};
};

// funkcja oblicza ilość kombinacji k-elementowych w zbiorze n-elementowym
int Newton(int n, int k)
{
    // redukcja wartości k – por. definicję tablicy t
    if (n-k < k) k = n-k;
    // czy zakończyć rekurencję?
    if (k == 0) return 1;

    assert (   k >=2      &&  n >= k
            && n <= MaxN  &&  k <= MaxN/2 );

    if (t[n-1][k-1] == 0)
       t[n-1][k-1] = Kombin(n-1, k) + Kombin(n-1, k-1);
    int wynik = t[n-1][k-1];

    if (wynik < 0)
       throw 1;      // THROW sygnalizuje błąd przekroczenia zakresu

    return wynik;
}
```

- **Zastosowanie tablicy do przechowania obliczonej już wartości funkcji powoduje radykalne przyspieszenie działania programu kosztem użycia 5 kB pamięci operacyjnej (50*25*4B).**
- **Globalną tablicę `t` ukryto w nienazwanej przestrzeni nazw, co ogranicza jej widoczność wyłącznie do pliku, w którym ją zdefiniowano.**