Podstawy programowania 2

dr inż. Piotr Szwed
Katedra Informatyki Stosowanej
C2, pok. 403

e-mail: pszwed@agh.edu.pl

http://home.agh.edu.pl/~pszwed/

Aktualizacja: 28.02.2021

2. Referencje
Składowe klas
Biblioteka standardowa

Referencje

Referencje w C++

Referencje w języku C++ są typem, który bardzo przypomina wskaźniki. Podobnie, jak w przypadku wskaźników, wartościami zmiennych typu referencyjnego są adresy, a także referencja przechowuje informacje o typie wskazywanego obiektu.

- W odróżnieniu od wskaźników, przy korzystaniu z referencji nie stosuje się operatora *dereferencji* (*). Jest on wywoływany automatycznie.
- **Referencji nie można przestawiać, tak aby wskazywała inny obiekt.**

Referencje w C++

Za pośrednictwem referencji możemy:

- odczytać lub zmodyfikować wartość (atrybuty) obiektu zajmującego pamięć identyfikowaną przez adres
- wywołać metodę obiektu.

Składnia deklaracji:

```
type-specifier & reference
```

`type-specifier`

definiuje typ wskazywanego obiektu

`reference`

identyfikator zmiennej

Referencje w C++

Referencje mogą być bezpośrednio używane jako zmienne:

```
int x=7;
int&r1 = x; // (1)
const int&r2 = 12; // (2)
r1++; // (3)
printf(”r1=%d, r2=%d”,r1,r2);
```

- Instrukcja (1) deklaruje referencję inicjując ją adresem obiektu x.
- Instrukcja (2) alokuje pamięć dla zmiennej typu int, inicjuje ją wartością 12 oraz deklaruje referencję, która wskazuje to miejsce.
- Wszelkie operacje na referencjach są w rzeczywistości operacjami na obiektach wskazywanych przez referencje. Instrukcja (3) zwiększy wartość zmiennej x.

Referencje w C++

- Referencje są najczęściej używane jako argumenty funkcji. Podobnie jak w przypadku wskaźników, obiekty przekazywane są przez adres.

```
void min_max(int tab[],int n,int&min,int&max){
    int i;
    max=min=tab[0];
    for(i=0;i<n;i++){
        if(min>tab[i])min=tab[i];
        if(max<tab[i])max=tab[i];
    }
}
```

Referencje w C++

- Referencje mogą być używane jako wartości zwracane przez funkcje. Najczęściej są to funkcje składowe obiektu i zwracają referencję do obiektu, do którego należą.

```
class A
{
public:
    A&foo() {
        //...
        return *this;
    }
};
```

Referencje w C++

- Referencje mogą być bezpośrednio używane jako pola klas, ale wymagają inicjalizacji poprzez **listę inicjalizacyjną** konstruktora klasy

```
class A
{
    int&r;
public:
    A(int&a):r(a){}
};
```

Referencje w C++

- Podstawową różnicą pomiędzy referencjami i wskaźnikami jest to, że wartością referencji musi być adres istniejącego obiektu. Wartość referencji jest ustalana w momencie inicjalizacji i jest to statycznie sprawdzane przez kompilator.
- **Wartością referencji nie może być 0 (NULL).**

```
int &r1;    // błąd r1 nie wskazuje żadnego obiektu
int x=5;
int &r2=x;  // OK. r2 wskazuje zmienną x
```

Składowe klas

Defincje klas

W języku C++ klasy można definiować używając słów kluczowych `class`, `struct` lub `union`.

Definicja klasy najczęściej obejmuje:

- **atrybuty** (pola)
- **metody** (funkcje składowe)
- **jeden** lub **kilka konstruktorów**
- **jeden destruktor**

Deklaracja klas z użyciem słów `struct` i `union` jest zapewniona dla zgodności z językiem C. Standardowo, dostęp do ich metod i pól jest publiczny.

W przypadku użycia słowa kluczowego `class`, standardowo dostęp jest prywatny.

Deklaracja klasy

```
class File
{
    FILE*fp;
public:
    File(); // standardowy konstruktor
    File(const char*name, const char*mode); //
    ~File(); // destruktor
    int open(const char*name, const char*mode);
    int close();
    int get();
    int put(int);
};
```

Implementacja metod

```
File::File(){fp = 0;}

File::File(const char*name, const char*mode){
    open(name,mode);
}
int File::open(const char*name,const char*mode)
{
    fp = fopen(name,mode);
    return fp!=0;
}
int File::close(){
    if(fp)fclose(fp);
    fp=0;
    return 1;
}
```

Implementacja metod

```
int File::get(){
    if(fp)return getc(fp);
    else return -1;
}

int File::put(int c){
    if(fp){
        putc(c,fp);
        return 1;
    }
    return 0;
}
```

Użycie klasy

```
void f(){
    File file; // obiekt typu File

    File file2("plikwy.txt","wt");

    File *pfile = new File("plikwe.txt","rt"");
    for(;;){
        int c = pfile->get();
        if(c<0)break;
        file2.put(c);
    }
    delete pfile;
}
```

Deklaracja klasy

Składnia:

```
class identifier[base-class-specifier]
{
     member-list
}  ;
```

- Nazwa klasy staje się widoczna dla kompilatora bezpośrednio po przetworzeniu nagłówka klasy.
- Deklaracja klasy wprowadza nowy identyfikator do przestrzeni nazw. Deklaracje te są równocześnie definicjami typu w danej jednostce translacji (kompilowanym pliku źródłowym).

Pliki h i cpp

Definicje klas są zazwyczaj używane w większej liczbie jednostek translacji, stąd typową praktyką jest umieszczenie ich w plikach nagłówkowych (*.h). Implementacje metod klas umieszcza się w plikach *.cpp (odrębnych jednostkach translacji).

a.h	b.h

`class A`
`{`
`public:`
`  A();`
`};`
	`#include "a.h"`
	`class B`
	`{`
	`public:`
	`  B();`
	`  A a;`
	`};`

Definicja klasy B wymaga znajomości definicji klasy A, stąd nagłówek a.h jest włączany do nagłówka b.h.

Pliki h i cpp

Kompilator kompiluje jednostki translacji - pliki cpp wraz z włączonymi nagłówkami.

a.cpp	b.cpp	main.cpp

`#include "a.h"`
`A::A(){}`
	`#include "b.h"`
	`B::B(){}`
		`//#include "a.h"`
		`#include "b.h"`
		`int main()`
		`{`
		`  A a;`
		`  B b;`
		`  return 0;`
		`}`

Pliki h i cpp

W danej jednostce translacji może pojawić się **dokładnie jedna** definicja klasy. Problemem jest, śledzenie zależności pomiędzy plikami nagłówkowymi.

Najczęściej stosowanym zabezpieczeniem jest użycie dyrektyw warunkowej kompilacji:

a.h

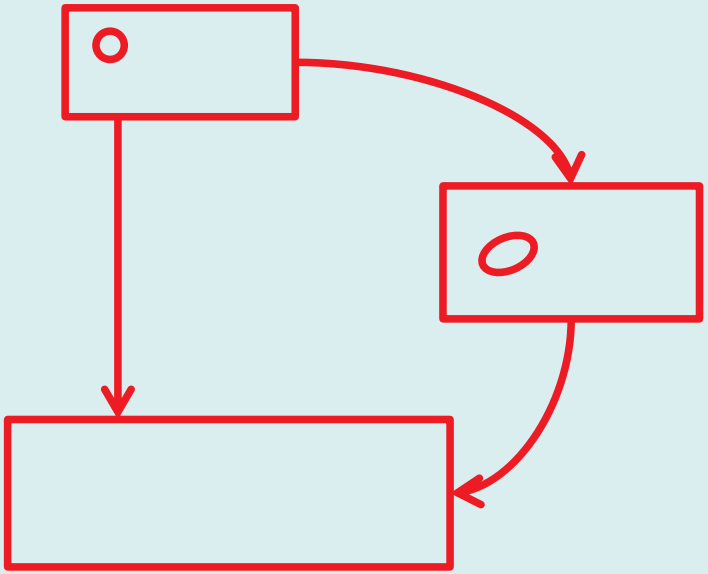
```
#if !defined _a_h_
#define _a_h_

class A
{
public:
   A();
};

#endif // _a_h_
```

Przykład

```
#include "a.h" // (1)
#include "b.h" // (2)
void main()
{
      A a;
      B b;
}
```



1. Klasa A zostanie zdefiniowana. Zdefiniowany zostanie symbol preprocesora **_a_h_**
2. Włączony zostanie nagłówek b.h. Przetwarzanie b.h pociągnie włączenie a.h. Ponieważ **_a_h_** istnieje w słowniku symboli preprocesora, powtórna definicja klasy A jest pomijana.

Elementy składowe klas

Klasy mogą mieć następujące elementy składowe:

- funkcje składowe (metody)
- przechowywane dane (pola, atrybuty)
- klasy zagnieżdżone (wewnętrzne)
- wyliczenia (enum)
- pola bitowe
- deklaracje klas zaprzyjaźnionych (friend)
- wewnętrzne deklaracje typów

Elementy klas mogą być zadeklarowane z użyciem modyfikatorów: const i static.

Metody – przykład w C 1

```
// deklaracja struktury
typedef struct {double x,y;}StrComplex;

// nadanie wartości początkowych lub przypisanie
void init(StrComplex*pc, double _x, double _y)
{
    pc->x=_x ;pc->y=_y;
}

// oblicza moduł liczby zespolonej
double module(const StrComplex*pc)
{
    return sqrt(pc->x*pc->x + pc->y*pc->y);
}
```

Metody – przykład w C 2

```
// wypisuje wartość składowych oraz moduł
void dump(const StrComplex*pc)
{
    printf("[x=%g, y=%g,module=%g]”,
    pc->x,pc->y, module(pc)) ;
}

void f(){
    StrComplex c;
    // nadajemy wartość początkową
    init(&c,2.4,3.76) ;
    // wypisujemy informacje
    dump(&c) ;
}
```

Metody - przykład w C++ 1

```
class Complex
{
public:
    double x,y;
    Complex(double _x,double _y)
            :x(_x),y(_y){}
    double module()const{
            return sqrt(x*x+y*y );
    }
    void dump()const;
    void set(double _x,double _y){
            x=_x;
            y=_y;}
};
```

Metody - przykład w C++ 2

```
void Complex::dump()const
{
    printf("[x=%g, y=%g,module=%g]”,
    this->x, this->y, this->module()) ;
}

void g() {
    Complex c(2.4, 3.76);
    c.dump();
    Complex *pc = &c;
    pc->dump();
    Complex &rc = c;
    rc.set(2.0, 3.0);
    rc.dump();
}
```

Porównanie 1

```
void init(StrComplex*pc, double _x, double _y)
{
    pc->x=_x ;pc->y=_y;
}
```

```
Complex(double _x,double _y):x(_x),y(_y){}

void set(double _x,double _y){x=_x ;y=_y ;}
```

Porównanie 2

```
double module(const StrComplex*pc)
{
    return sqrt(pc->x*pc->x + pc->y*pc->y);
}
```

```
double module()const{return sqrt(x*x+y*y ) ;}
```

Porównanie 3

```
void dump(const StrComplex*pc)
{
    printf("[x=%g, y=%g,module=%g]”,
    pc->x,pc->y, module(pc)) ;
}
```

```
void Complex::dump()const
{
    printf("[x=%g, y=%g,module=%g]”,
    this->x, this->y, this->module()) ;
}
```

Metody obiektu

1. W metodach należących do obiektu możemy bezpośrednio odwoływać się do jego danych. (Domyślnie odwołujemy się do tego obiektu, którego metoda jest wołana – `this`.)
2. W funkcjach składowych należących obiektu możemy wołać inne metody danego obiektu.
3. Wołając metody spoza obiektu wskazujemy obiekt, do którego wysyłamy komunikaty podając nazwę obiektu, wskaźnik lub referencję.
4. Funkcje, które nie modyfikują obiektu mogą być zadeklarowane jako `const`.

```
void dump(const StrComplex*pc);
void dump()const;
```

1. Funkcje składowe mogą być implementowane wewnątrz definicji klasy lub poza nią – `dump()`.

Wskaźnik this 1

- Wewnątrz niestatycznych metod obiektu można posługiwać się niemodyfikowalnym (`const`) wskaźnikiem this do obiektu danej klasy. Jest on domyślnym ukrytym argumentem każdej niestatycznej funkcji składowej.

```
CLASS * const this;
```

- Wewnątrz metod zadeklarowanych jako const (nie mających prawa modyfikować zawartości obiektu) wskaźnik this jest widoczny jako:

```
const CLASS * const this;
```

Wskaźnik this 2

```
class Light
{
    double voltage;
public:
    void on(){this->voltage = 230;}
    void off(){this->voltage = 0;}
    void brighten(){if(this->voltage<=220) this->voltage+=10;}
    void dim(){if(this->voltage>=10) this->voltage-=10;}
};
```

```
Light lt;
lt.on();
lt.dim();
lt.off();
```

```
Light*plt = new Light();
plt->on();
plt->dim();
plt->brighten();
delete plt;
```

Adres `&lt` lub `plt` są dostarczane do metod obiektu jako ich pierwszy (ukryty) argument

Wskaźnik this 3

Za pośrednictwem wskaźnika `this` można realizować dostęp do funkcji składowych i danych.

- W niektórych przypadkach pomaga to rozwiązać niejednoznaczności.
- Wskaźnika `this` używa się także często przy konieczności zwrócenia referencji do danego obiektu.

```
Complex& Complex ::set(double x,double y)
{
    this->x=x ;
    this->y=y ;
    return *this ;
}
```

Wskaźnik this 4

Może on służyć do ustalania asocjacji (powiązania) pomiędzy obiektami.

```
class Owner;

class Child
{
public:
    Owner*owner;
};
```

```
class Owner
{
    list<Child*> children;
public:
    void add (Child*child){
        child->owner=this;
        children.push_back(child);
    }
};
```

Funkcje inline 1

Funkcje inline, to funkcje, których wywołanie jest bezpośrednio zastępowane kodem funkcji. W przypadku bardzo krótkich funkcji ich użycie jest bardziej ekonomiczne, ponieważ znika dodatkowy narzut na wywołanie funkcji, powrót z wywołania oraz przesyłanie w obie strony danych poprzez stos. Wygenerowany kod może działać szybciej i być mniejszy.

```
class Int
{
    int value;
public:
    Int(int v):value(v){}
    int get()const{return value;} // inline
    void set(int v){value = v;} // inline
};
```

Funkcje inline 2

Funkcje zaimplementowane wewnątrz definicji klasy w miarę możliwości są tłumaczone jako funkcje inline. Alternatywnie, funkcje które są implementowane poza definicją klasy mogą być kompilowane jako funkcje inline po poprzedzeniu ich słowem kluczowym `inline` (traktowanym jako wskazówka dla kompilatora).

```
inline void Complex::dump()const
{
    printf("[x=%g, y=%g,module=%g]”,
    this->x, this->y, this->module()) ;
}
```

Kompilator ignoruje słowo kluczowe inline w przypadku, kiedy funkcja jest funkcją rekurencyjną lub może być wywoływana za pośrednictwem wskaźnika.

Składowe typu static

- W klasie można deklarować zarówno pola, jak i metody typu `static`. Traktuje się je jako elementy składowe klasy, a nie obiektu, stąd mogą być one dzielone przez wszystkie obiekty danej klasy, a także używane z zewnątrz.
- Metody statyczne nie mają dostępu do wskaźnika `this`, ponieważ w ich przypadku brak jest obiektu, który mógłby wskazywać. Stąd, w metodach statycznych można używać wyłącznie danych zadeklarowanych jako statyczne.

Składowe typu static

```
class A
{
public:
    A(){instanceCounter ++;}
    ~ A(){instanceCounter --;}
    static int getInstaceCounter(){
       return instanceCounter;}
    void dump()const;
private:
    static int instanceCounter;
};

void A::dump()const
{
    printf("A has %d instances", getInstaceCounter());
}
```

Możliwy dostęp do statycznego atrybutu
Brak dostępu do this

Statyczny atrybut

Składowe typu static

Pola statyczne klasy istnieją niezależnie od tego, czy istnieje jakikolwiek obiekt danej klasy. Z tego powodu pojawienie się statycznych pól w definicji klasy jest traktowane jak deklaracja `extern` , która wymaga odrębnej definicji, podczas której można także inicjować zmienne statyczne wartościami początkowymi.

```
int A::instanceCounter=0;
```

Składowe typu static

Metody statyczne mogą być wołane:

- z metody niestatycznej (poprzez bezpośrednie użycie nazwy)

```
void A::dump()const{
    printf("A has %d instances", getInstaceCounter());
}
```

- z zewnątrz za pośrednictwem obiektu

```
A a;
int n = a.getInstaceCounter();
```

- z zewnątrz poprzez podanie operatora zasięgu (*scope*)

```
printf("%d", A::getInstaceCounter());
```

Podobne reguły dotyczą zasad dostępu do statycznych atrybutów klasy.

Klasy zagnieżdżone (wewnętrzne)

- Klasy zagnieżdżone (*ang*. *nested, inner class*) są to klasy zadeklarowane wewnątrz innej klasy. Najczęściej stosuje się je jako pomocnicze struktury danych oraz tam gdzie nie chce się wprowadzać nowej nazwy do przestrzeni nazw kolidującej z istniejącymi nazwami.
- Deklaracja klasy zagnieżdżonej jest jedynie deklaracją typu

Klasy zagnieżdżone (wewnętrzne)

```
class Point {
    double x,y;
public:
    Point(double _x,  double _y);
};
```

```
class Circle
{
public:
    Circle(double _x, double _y, double r);
    class Point{
        double t[2] ;
    public:
        Point(double _x,  double _y);
    };
    Point center;
    double radius ;
};
```

Dwie klasy `Point` różniące się implementacjami:

- globalna
- wewnętrzna

Klasy zagnieżdżone (wewnętrzne)

```
// konstruktor klasy globalnej
Point::Point(double _x,  double _y){
    x=_x;
    y=_y;
}

// konstruktor klasy wewnętrznej
Circle::Point::Point(double _x,  double _y){
    t[0]=_x;
    t[1]=_y;
}

// konstruktor klasy zwewnętrznej
Circle::Circle(double _x, double _y, double r)
    :center(_x,_y),radius(r){}
```

Klasy zagnieżdżone (wewnętrzne)

- Używając klasy zagnieżdżonej w metodach klasy zewnętrznej możemy posługiwać się bezpośrednio nazwą klasy.
- Poza metodami klasy zewnętrznej musimy podawać pełną nazwę, postaci `Outer::Inner`,

```
Circle::Point pointInCircle;
```

- Dostęp do definicji klas zagnieżdżonych jest sterowany standardowymi modyfikatorami dostępu. Z zewnątrz nie możemy uzyskać dostępu do **definicji klasy** , jeżeli dostęp do niej jest ograniczony jako `private` lub `protected`.

Przykład

```
class Map
{
protected:
    class Pair{
        friend class Map;
        double x;
        double y;
    };
    Pair tab[1000];
    int cnt;
public:
    bool add(double x,double y){
        if(cnt==1000)return false;
        tab[cnt].x=x;
        tab[cnt].y=y;
        cnt++;
    }
    double get(double x){
        for(int i=0;i<cnt;i++){
            if(fabs(tab[i].x-x)<1e-5)return tab[i].y;
        }
        return -1e308;
    }
};
```

Klasa wewnętrzna

Typy wewnętrzne

Wewnątrz klas można deklarować zagnieżdżone typy za pośrednictwem konstrukcji `typedef`. Zasady dostępu do zagnieżdżonej definicji typu są analogiczne jak w przypadku klas.

```
class Tree
{
public:
    typedef Tree * PTREE;
    PTREE Left;
    PTREE Right;
    void *vData;
};

PTREE pTree;  // Error: not in class scope.
Tree::PTREE pTree;  // Ok.
```

Wyliczenia

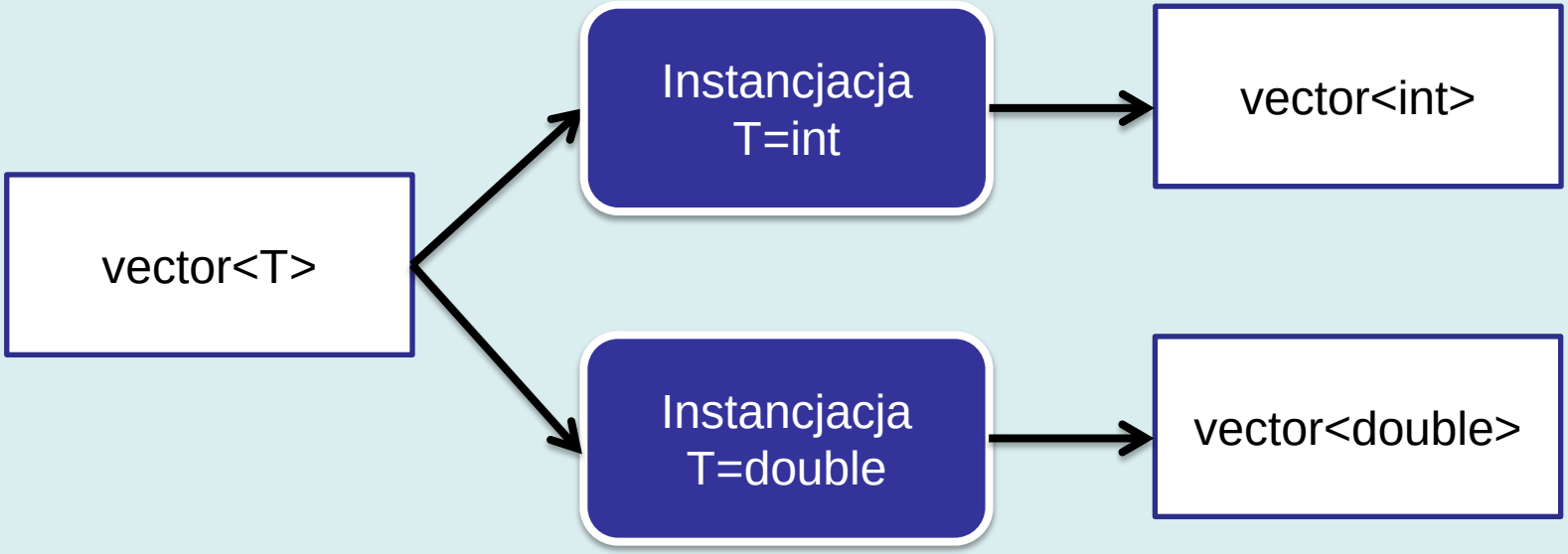
```
class HasState
{
    int state;
public:
    enum {good=0, bad, };
    HasState(){
        state = good;}
    int getState()const
    {return state;}
};
```

```
HasState hs;
int s = hs.getState();
if(s== HasState::good)     printf("good");
if(s== HasState::bad)  printf("bad");
```

Kilka podstawowych klas biblioteki standardowej

Szablon <vector>

- vector zapewnia funkcjonalność tablicy przyrastającej w miarę dodawania elementów.
- vector jest **szablonem**. Oznacza to, że w kodzie źródłowym tablica nie ma określonego typu, ale zamiast niego używany jest jakiś symbol, na przykład T.
- W momencie tak zwanej **instancjacji** automatycznie tworzony jest kod dla tablicy wskazanego typu



Przykład 1

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

int main(){
    std::vector<int> liczby;
    for(int i=0;i<20;i++)liczby.push_back(i);

    for(int i=0;i<liczby.size();i++){
        std::cout<<liczby[i]<<" ";
    }
    std::cout<<std::endl;
}
```

Przykład 2

```
#include <vector>
#include <iostream>
#include <cmath> // lub <math.h>
using namespace std;

int main(){
    vector<double> liczby;
    for(int i=0;i<20;i++)liczby.push_back(sqrt(i));

    for(int i=0;i<liczby.size();i++){
        cout<<liczby[i]<<" ";
    }
    cout<<endl;
}
```

```
0 1 1.41421 1.73205 2 2.23607 2.44949 2.64575 2.82843 3 3.16228
3.31662 3.4641 3.60555 3.74166 3.87298 4 4.12311 4.24264 4.3589
```

Przykład 3

```
#include <vector>
#include <iostream>
#include <cmath> // lub <math.h>
using namespace std;

#define SIZE 1000
int main(){
    vector<double> xs(SIZE);
    vector<double> ys(SIZE);

    for(int i=0;i<SIZE;i++){
        xs[i]=i*0.01;
        ys[i]=xs[i]*xs[i]-4*xs[i]+4;
    }

    for(int i=0;i<xs.size();i++){
        cout<<xs[i]<<" "<<ys[i]<<endl;
    }

}
```

```
0 4
0.01 3.9601
0.02 3.9204
0.03 3.8809
0.04 3.8416
0.05 3.8025
…
1.97 0.0009
1.98 0.0004
1.99 0.0001
2 0
2.01 0.0001
2.02 0.0004
2.03 0.0009
…
9.95 63.2025
9.96 63.3616
9.97 63.5209
9.98 63.6804
9.99 63.8401
```

Jak uzywać szablonu vector<T>

- Włączamy nagłówek `<vector>`
- Możemy utworzyć wektor o zadanym rozmiarze
- Dodajemy elementy na końcu wektora za pomocą metody `push_back()`
- Metoda `size()` zwraca rozmiar tablicy (liczbę zajętych elementów)
- Operator `[]` umożliwia dostęp do elementu o danym indeksie
- Iterujemy za pomocą pętli

```
for(int i=0;i<liczby.size();i++){
    cout<<liczby[i]<<endl;
}
```

Przykład - iteracja

```
int main(){
    vector<int> liczby{1,1}; // inicjuje tablicę dwóch jedynek
    while(true){
        int e = liczby[liczby.size()-1]+liczby[liczby.size()-2];
        if(e>200)break;
        liczby.push_back(e);
    }
    //iteracja range-based for
    for(int e:liczby){ // wartość – tylko odczyt
        cout<<e<<" ";
    }
    for(int&e:liczby){ // referencja – możliwa zmiana elementu
        e*=-1;
    }
    for(int e:liczby){
        cout<<e<<" ";
    }

}
```

1 1 2 3 5 8 13 21 34 55 89 144 -1 -1 -2 -3 -5 -8 -13 -21 -34 -55 -89 -144

Klasa <string>

- Klasa `std::string` reprezentującą tablicę znakową, która może dynamicznie zmieniać swoje rozmiary.
- Poza funkcjami do zarządzania pamięcią, klasa zawiera szereg metod umożliwiających dostęp do znaków lub działania na całych tekstach (konkatenacja, usuwanie, wstawianie, zastępowanie, itd.)

Zarządzanie pamięcią

`capacity()`

Zwraca rozmiar bufora.

`reserve (size_type n =0)`

Zmienia rozmiary bufora. Gwarantuje, że po jej wykonaniu funkcja capacity() będzie zwracała co najmniej n.

`size()` lub `length()`

Zwraca długość tekstu (liczbę znaków umieszczonych w buforze).

`resize(size_type n, E c = E())`

Gwarantuje, że funkcja `size()`będzie zwracała co najmniej `n`. W razie potrzeby powiększa tablicę i wypełnia znakiem `c`.

Przykład

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

int main(int argc, char* argv[]){
    string s;
    ifstream is("readme.txt");
    s.reserve(40);
    for(int i=0;;i++){
        int c=is.get();
        if(c<0)break;
        s+=(char)c;  // lub: s.append(1,c);
        if(i%100==0){
            cout<<s.capacity()<<" ";
            cout<<"("<<s.size()<<") ";
        }
    }
    cout<<endl<<s;
    return 0;
}
```

40 (1) 160 (101) 320 (201) 320 (301) 640 (401) 640 (501) 640 (601) 1280 (701) 1280 (801) 1280 (901) 1280 (1001) 1280 (1101) 1280 (1201) 2560 (1301) 2560 (1401) 2560 (1501) 2560 (1601) 2560 (1701) 2560 (1801) 2560 (1901) 2560 (2001) 2560 (2101) 2560 (2201) 2560 (2301) 2560 (2401) 2560 (2501) 8135 (2601) …tekst pliku …

Metoda c_str()

Metoda zwraca stały wskaźnik do tekstu. Może być użyta tam, gdzie wymagane jest użycie danych typu `const char*`.

```
int main(){
    string s="Tekst";
    cout<<strcmp(s.c_str(),"Tekst")<<endl;
}
```

Dostęp do znaków

Użytkownik klasy string może zrealizować dostęp do znaków z wykorzystaniem `operatora []`

```
int main(){
    string s="Ala ma kota";
    for(int i=0;i<s.size();i++){
        s[i]+=3;
    }
    cout<<s<<endl;
    for(int i=0;i<s.size();i++){
        s[i]+=253;
    }
    cout<<s<<endl;
}
```

Dod#pd#nrwd
Ala ma kota

Konkatenacja tekstów

Do konkatenacji (łączenia) tekstów służy metoda append( ) występująca w kilku przeciążonych wersjach lub operatory + i +=.

```
int main(int argc, char* argv[])
{
    string s("Ala ma");
    s.append(" kota");
    s+=" i psa";
    cout<<endl<<s;
    return 0;
}
```

Ekstrakcja tekstów

Do wyodrębniania fragmentów tekstu służy metoda `substr()`. Jej parametrami są indeks początkowy i długość tekstu.

```
int main(int argc, char* argv[])
{
    string s="abcdefgijklmn";
    cout<<s.substr(3,3)<<endl;
    return 0;
}
```

Rezultat: `def`

Usuwanie fragmentów tekstu

W klasie string zdefiniowano kilka przeciążonych wersji metody `erase()` pozwalającej na usuwanie fragmentów (lub całego) tekstu.

```
int main(int argc, char* argv[])
{
    string s="abcdefgijklmn";
    int n1= s.find('d');
    int n2 = 3; // ile znakow
    s.erase(n1,n2);
    cout<<s<<endl;
    return 0;
}
```

Rezultat:
abcgijklmn

Wstawianie fragmentów tekstu

Do wstawiania fragmentów tekstu służy (przeciążona na kilka sposobów) funkcja `insert()`. Funkcja wstawia w miejsce określone przez indeks lub iterator tekst, który może być tablicą znaków, fragmentem innego obiektu klasy string lub sekwencją identycznych znaków.

```
int main(int argc, char* argv[]){
    string s("Ala ma kota");
    s.insert(strlen("Ala ma "), "psa i ");
    cout<<s<<endl;
    s="abcdefgijklmn";
    s.insert(1,2,'x');
    cout<<s<<endl;
    return 0;
}
```

Rezultat:
Ala ma psa i kota
axxbcdefgijklmn

Znajdywanie i zastępowanie fragmentów tekstu

Funkcja `find()` pozwala na wyszukanie w tekście łańcucha znaków lub wystąpienia znaku. Funkcja `replace()` pozwala na zastąpienie wskazanego fragmentu tekstu innym.

```
int main(int argc, char* argv[])
{
    string s("Witaj $user, jestes zalogowany");
    string userTag="$user";
    int start=s.find(userTag);
    s.replace(start,userTag.size(),"Piotr Szwed");
    cout<<s<<endl;
    return 0;
}
```

```
Witaj Piotr Szwed, jestes zalogowany
```

Lista – szablon <list>

Szablon definiuje listę dwukierunkową.

- Możliwe jest:
  - dodawanie elementów na początku i końcu listy,
  - wstawianie, usuwanie, łączenie ze sobą list.
  - automatyczne sortowanie listy.
- Nie jest możliwy swobodny dostęp do elementów za pośrednictwem operatora [].
- Lista jest efektywna, jeżeli liczba elementów i ich miejsce wstawienia nie są z góry określone.
- Iteracja po liście wymaga użycia iteratorów albo pętli **range-based for**

Przykład 1

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

int main(){
    list<int> lst{11,23,123,11,4,56,-9,};
    for(int e:lst){
        cout<<e<<" ";
    }
}
```

```
11 23 123 11 4 56 -9
```

Przykład 2

```
int main(){
    list<int> lst{11,23,123,11,4,56,-9,};
    for(int i=0;i<5;i++){
        lst.push_back(i);
        lst.push_front(i);
    }
    for(int e:lst){
        cout<<e<<" ";
    }
    cout<<endl;
    lst.sort();
    for(int e:lst){
        cout<<e<<" ";
    }
}
```

```
4 3 2 1 0 11 23 123 11 4 56 -9 0 1 2 3 4
-9 0 0 1 1 2 2 3 3 4 4 4 11 11 23 56 123
```

Przykład 3

Wektor, lista mogą być parametryzowane elementami dowolnego typu…

```
int main(){
    list<string> lst{"Ala","ma","kota"};
    for(const string& e:lst){
        cout<<e<<" ";
    }
    cout<<endl;
    lst.sort();
    for(auto&e:lst){
        cout<<e<<" ";
    }
}
```

Jeżeli użyjemy **auto** kompilator sam wywnioskuje, jakiego typu jest zmienna

```
Ala ma kota
Ala kota ma
```

Przykład 4

```
class Student{
public:
    string imie;
    string nazwisko;
    int nr_legitymacji;
};

int main(){
    vector<Student> grupa;
    Student jk;
    jk.imie = "Jan";
    jk.nazwisko = "Kowalski";
    jk.nr_legitymacji=1234;
    grupa.push_back(jk);
    for(int i=0;i<grupa.size();i++){
        cout<<grupa[i].imie<<" "<<grupa[i].nazwisko<<" "
            <<grupa[i].nr_legitymacji<<endl;
    }

}
```

Jan Kowalski 1234

Przykład 5 – emplace

```
class Student{
public:
    Student(const char*im,const char*naz,int nr)
      :imie(im), nazwisko(naz),nr_legitymacji(nr){}
    string imie;
    string nazwisko;
    int nr_legitymacji;
};

int main(){
    vector<Student> grupa;
    grupa.emplace_back("Jan","Kowalski",1234);
    grupa.emplace_back("Anna","Nowak",1235);

    for(int i=0;i<grupa.size();i++){
        cout<<grupa[i].imie<<" "<<grupa[i].nazwisko<<" "
          <<grupa[i].nr_legitymacji<<endl;
    }

}
```

Funkcja `emplace_back()` – woła „w miejscu” konstruktor przekazując mu parametry z zewnątrz.
Czasem wygodniejsze…

Jan Kowalski 1234
Anna Nowak 1235

Słownik – szablon <map>

- Słownik jest zapisem funkcji odwzorowującej *klucze* w *wartości*. Jest więc zbiorem par (*klucz*, *wartość*).
  - W słowniku może wystąpić tylko pojedyncza instancja klucza.
  - Dana wartość może być przypisana wielu kluczom.
- Szablon map jest optymalizowany, aby zapewnić dużą prędkość wyszukiwania elementów słownika, dlatego jest implementowany jako drzewo.
- Klasa może być użyta jako klucz, jeżeli zapewnia operator < do porównywania elementów lub jest typem wbudowanym zapewniającym ten operator domyślnie.

Przykład

```
#include <map>

int main()
{
    map<string,int> dict;
    dict.insert(map<string,int>::value_type("open",0));
    dict.insert(pair<string,int>("o",0));
    dict.emplace("close",1);
    dict["c"]=1;
    dict["exit"]=-1;
    for(auto i=dict.begin();i!=dict.end();i++){
        cout<<i->first<<" -> "<<i->second<<endl;
    }
    //...
```

Funkcje begin(), end() i find() zwracają iteratory. Będą omówione później…

Przykład

```
//...
    for(;;){
        char command[256];;
        cin.getline(command,256);
        auto i = dict.find(command);
        if(i==dict.end()){
            cout<<command<<" ???";continue;
        }
        cout<<i->second<<endl;
        if(i->second==-1)return 0;
    }
}
```

Rezultat:

c -> 1
close -> 1
exit -> -1
o -> 0
open -> 0
... Liczby odpowiadające poleceniom

Zbiór – szablon <set>

Zbiór jest kontenerem, który przechowuje unikalne wartości elementów. Szablon definiuje trzy podstawowe metody:

- `insert()` – dodaje element
- `empty()` – testuje czy zbiór jest pusty
- `find()` – znajduje element w zbiorze

Zbiór jest implementowany jako drzewo, dlatego elementy zbioru muszą spełniać takie same wymagania dotyczące interfejsu, jak klucze dla słownika: definiować operator <.

Operator < wystarczy do wprowadzenia **liniowego porządku**
Trzy przypadki:

- $a < b$
- $b < a$
- $\neg a < b \wedge \neg \mathrm{b} < a \rightarrow a = b$

stąd
$a \leq b \Leftrightarrow a < b \vee \neg b < a$

Przykład

```
#include <set>
int main()
{
    set<int> cont;
    cout<<(cont.empty()?"empty":"!empty")<<endl;
    for(int i=10;i>=0;i--)cont.insert(i);
    for(int i=5;i<15;i++)cont.insert(i);
    cout<<(cont.empty()?"empty":"!empty")<<endl;

    const auto it = cont.find(10);
    if(it != cont.end())cout<<"has:"<<*it<<endl;

    for(auto it=cont.begin();it!=cont.end();it++)
        cout<<*it<<" ";
    cout<<endl;
}
```

```
empty
!empty
has:10
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
```

Szablon <set>

```
#include <iostream>
#include <string>
#include <set>
#include <algorithm>
#include <cstdlib>
using namespace std;

class A{
public:
    int v =  random()%100;
    bool operator<(const A&other)const{
        return v<other.v;
    }
};

int main()
{
  set<A> s;
  for(int i=0;i<20;i++)s.insert(A());;
  for(auto &a:s){
      cout<<a.v<<" ";
  }
}
```

15 21 26 27 35 36 40 49 59 62 63 72 77 83 86 90 92 93