

Metodyka i Techniki Programowania II

Katedra Telekomunikacji, EiT

dr inż. Jarosław Bułat (c)

kwant@agh.edu.pl

Plan prezentacji

- » Programowanie uogólnione (generyczne)
- » Biblioteka standardowa STL
- » Vector, ...
- » Wersje standardu C++
- » Przyszłość języka

Programowanie uogólnione

programowanie generyczne, szablony

Programowanie uogólnione

```
int add(int x1, int x2){  
    return x1+x2;  
}
```

- » C++ jest stało-typowany
- » Zmienne muszą mieć typ

```
int main(){  
    cout << add(2, 3) << endl;  
    cout << add(0.1, 0.3) << endl;  
}
```

Programowanie uogólnione

```
int add(int x1, int x2){  
    return x1+x2;  
}
```

```
int main(){  
    cout << add(2, 3) << endl;  
    cout << add(0.1,0.3) << endl;  
}
```

- » C++ jest stało-typowany
- » Zmienne muszą mieć typ
- » Rezultat:
 - 5 ← ok
 - 0 ← błąd !!!
- » Dlaczego błąd???

Programowanie uogólnione

```
int add(int x1, int x2){  
    return x1+x2;  
}
```

```
int main(){  
    cout << add(2, 3) << endl;  
    cout << add(0.1, 0.3) << endl;  
}
```

- » C++ jest stało-typowany
- » Zmienne muszą mieć typ
- » Rezultat:
 - 5 ← ok
 - 0 ← błąd !!!
- » Dlaczego błąd???
- konwersja **double**->**int**
(zaokrąglenie w dół -
odcięcie części ułamkowej)
- przypomnienie: literał **0.1** w
kodzie źródłowym C++ jest
typu **double** a nie float

Programowanie uogólnione

```
int add(int x1, int x2){  
    return x1+x2;  
}
```

```
double add(double x1, double x2){  
    return x1+x2;  
}
```

```
int main(){  
    cout << add(2, 3) << endl;  
    cout << add(0.1, 0.3) << endl;  
}
```

- » C++ jest stało-typowany
- » Zmienne muszą mieć typ
- » Rezultat:
 - 5 ← ok
 - 0.4 ← ok

Programowanie uogólnione

```
int add(int x1, int x2){  
    return x1+x2;  
}
```

```
double add(double x1, double x2){  
    return x1+x2;  
}
```

```
int main(){  
    cout << add(2, 3) << endl;  
    cout << add(0.1,0.3) << endl;  
}
```

- » C++ jest stało-typowany
- » Zmienne muszą mieć typ
- » Rezultat:
 - 5 ← ok
 - 0.4 ← ok
- » Użyłem przeciążenia

Programowanie uogólnione

```
int add(int x1, int x2){  
    return x1+x2;  
}
```

```
double add(double x1, double x2){  
    return x1+x2;  
}
```

```
int main(){  
    cout << add(2, 3) << endl;  
    cout << add(0.1, 0.3) << endl;  
}
```

- » C++ jest stało-typowany
- » Zmienne muszą mieć typ
- » Rezultat:
 - 5 ← ok
 - 0.4 ← ok
- » Użyłem przeciążenia
- » Na etapie kompilacji, wybierana jest inna implementacja dla:
 - int

Programowanie uogólnione

```
int add(int x1, int x2){  
    return x1+x2;  
}
```

```
double add(double x1, double x2){  
    return x1+x2;  
}
```

```
int main(){  
    cout << add(2, 3) << endl;  
    cout << add(0.1, 0.3) << endl;  
}
```

- » C++ jest stało-typowany
- » Zmienne muszą mieć typ
- » Rezultat:
 - 5 ← ok
 - 0.4 ← ok
- » Użyłem przeciążenia
- » Na etapie kompilacji, wybierana jest inna implementacja dla:
 - int
 - double

Programowanie uogólnione

```
int add(int x1, int x2){  
    return x1+x2;  
}  
  
double add(double x1, double x2){  
    return x1+x2;  
}  
  
int main(){  
    cout << add(2, 3) << endl;  
    cout << add(0.1, 0.3) << endl;  
}
```

- » C++ jest stało-typowany
- » Zmienne muszą mieć typ
- » Rezultat:
 - 5 ← ok
 - 0.4 ← ok
- » Użyłem przeciążenia
- » Na etapie kompilacji, wybierana jest inna implementacja dla:
 - int
 - double
- » A co z innymi typami?
- » Operator+ jest zdefiniowany również dla innych typów

Programowanie uogólnione

- » Projektowanie kodu niezależnie od typów danych na których algorytm będzie operował - dlatego “uogólnione”
- » C++ jest statycznie typowany (zmienna musi mieć typ)
- » Możliwość pisania generycznego wymaga więc specjalnej funkcjonalności: **szablon (ang. template)**
- » Szablon:
 - kod używający abstrakcyjny typ danych
 - właściwy typ danych (np. int) jest wstawiany podczas kompilacji w zależności od sposobu wykorzystania, **następuje wtedy konkretyzacja szablonu (ang. template instantiation)**
- » Przykład: algorytm sortowania

Programowanie uogólnione

- » Programowanie generyczne to **następny poziom abstrakcji**
- » **Zalety:**
 - pozwala skupić się na algorytmie
 - zmniejsza objętość programu (nie trzeba implementować tego samego algorytmu dla różnych typów danych)
- » **Wady:** w C++ szablony działają trochę jak preprocesor
 - w przypadku błędu kompilator często wskazuje błąd w bibliotece a nie w implementowanym kodzie
 - komunikaty błędów są bardzo nieczytelne
 - kod szablonu musi być pliku nagłówkowym - jest konkretyzowany podczas użycia (trudno zrobić bibliotekę)

Dodawanie szablonem

zamiast przeciążenia `add(...)`

Szablony

oryginalny kod →

```
int add(int x1, int x2){  
    return x1+x2;  
}
```

```
double add(double x1, double x2){  
    return x1+x2;  
}
```

```
int main(){  
    cout << add(2, 3) << endl;  
    cout << add(0.1,0.3) << endl;  
}
```

Szablony

```
template <class T>
T add(T x1, T x2){
    return x1+x2;
}
```

```
int main(){
    cout << add(2, 3) << endl;
    cout << add(0.1, 0.3) << endl;
}
```

```
int add(int x1, int x2){
    return x1+x2;
}
```

```
double add(double x1, double x2){
    return x1+x2;
}
```

```
int main(){
    cout << add(2, 3) << endl;
    cout << add(0.1,0.3) << endl;
}
```


Szablony

```
template <class T>
T add(T x1, T x2){
    return x1+x2;
}
```

» Rezultat:

- 5 ← ok
- 0.4 ← ok

```
int main(){
    cout << add(2, 3) << endl;
    cout << add(0.1, 0.3) << endl;
}
```

Szablony

```
template <class T>
T add(T x1, T x2){
    return x1+x2;
}
```

```
int main(){
    cout << add(2, 3) << endl;
    cout << add(0.1, 0.3) << endl;
}
```

- » Rezultat:
 - 5 ← ok
 - 0.4 ← ok
- » Można stosować wymiennie:
 - <class T>
 - <typename T>
- » **T** - Type (konwencja)
- » **T** - generyczny typ, który zostanie **skonkretyzowany** podczas kompilacji

Szablony

```
template <class T>
T add(T x1, T x2){
    cout << typeid(x1).name();
    return x1+x2;
}
```

```
int main(){
    add(2, 3);
    add(0.1, 0.3);
    add(0.1f, 0.3f);
}
```

- » Operator `typeid(...)` pozwala sprawdzić jakiego typu jest zmienna
- » Wywołuję funkcję `add` 3 razy z argumentami:
 - `int`
 - `double`
 - `float`

Szablony

```
template <class T>
T add(T x1, T x2){
    cout << typeid(x1).name();
    return x1+x2;
}
```

```
int main(){
    add(2, 3);
    add(0.1, 0.3);
    add(0.1f, 0.3f);
}
```

- » Operator typeid(...) pozwala sprawdzić jakiego typu jest zmienna
- » Wywołuję funkcję add 3 razy z argumentami:

- int
- double
- float

» **Rezultat:**

- i
- d
- f

Szablony

```
template <class T>
T add(T x1, T x2){
    return x1+x2;
}
```

```
int main(){
    cout << add(2, 3) << endl;
    cout << add(0.1, 0.3) << endl;
}
```

- » Można napisać kod generyczny
- » Algorytm **add(...)** jest napisany “bez typów”, czyli dla wszystkich typów argumentów
- » Kompilator na podstawie szablonu utworzy dwie definicje funkcji add()

Szablony

```
int add(int x1, int x2){  
    return x1+x2;  
}
```

```
int main(){  
    cout << add(2, 3) << endl;  
    cout << add(0.1, 0.3) << endl;  
}
```

- » Można napisać kod generyczny
- » Algorytm **add(...)** jest napisany “bez typów”, czyli dla wszystkich typów argumentów
- » Kompilator na podstawie szablonu utworzy dwie definicje funkcji add()

Szablony

```
int add(int x1, int x2){  
    return x1+x2;  
}
```

```
double add(double x1, double x2){  
    return x1+x2;  
}
```

```
int main(){  
    cout << add(2, 3) << endl;  
    cout << add(0.1, 0.3) << endl;  
}
```

- » Można napisać kod generyczny
- » Algorytm **add(...)** jest napisany “bez typów”, czyli dla wszystkich typów argumentów
- » Kompilator na podstawie szablonu utworzy dwie definicje funkcji add()

Szablony - wnioski

```
int add(int x1, int x2){  
    return x1+x2;  
}
```

```
double add(double x1, double x2){  
    return x1+x2;  
}
```

```
int main(){  
    cout << add(2, 3) << endl;  
    cout << add(0.1, 0.3) << endl;  
}
```

- » Zaawansowany preprocesor
- » Zmniejsza ilość kodu (**kompilator sam sobie generuje**)
- » **Kod generowany podczas** użycia funkcji add() - **kompilacji** programu używającego add()
- » Kod zadziała dlatego każdego typu posiadającego **Operator+**

Zasięg, mieszanie typów

Szablony

```
template <class T>
T add(T x1, T x2){
    return x1+x2;
}
```

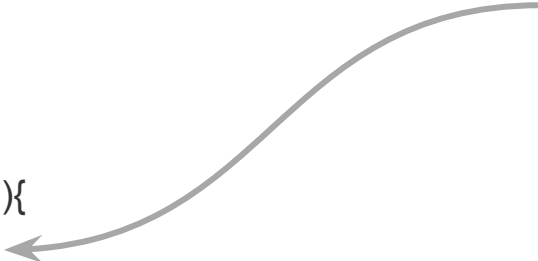
```
int main(){
    add(2, 3);
    add(0.1, 0.3);
    add(0.1f, 0.3f);
}
```

» Zasięg T jest ograniczony w tym przypadku do funkcji add

Szablony - zasięg

```
template <class T>
T add(T x1, T x2){
    return x1+x2;
}
```

```
int main(){
    T x;
    add(2, 3);
    add(0.1, 0.3);
    add(0.1f, 0.3f);
}
```



- » Zasięg T jest ograniczony w tym przypadku do funkcji add
- » Próba deklaracji zmiennej x skończy się błędem:

error: 'T' was not declared in this scope

Szablony - wiele

```
template <class T, class W>
bool add(T x1, W x2){
    if (x1 < x2){
        return true;
    } else {
        return false;
    }
}
```

» Można użyć dowolnej ilości
różnego typu zmiennych

Szablon **klasy**

Szablony - klasy

```
template <class T>
```

```
class Vector {
```

```
    T x1_, x2_;
```

```
public:
```

```
    Vector(T x1, T x2);
```

```
};
```

```
int main(){
```

```
    Vector <int> v(1, 2);
```

```
}
```

- » W klasie **Vector**, komponenty mogą być lub mogą używać typu T

Szablony - klasy

```
template <class T>
class Vector {
    T x1_, x2_;
public:
    Vector(T x1, T x2);
};

int main(){
    Vector<int> v(1, 2);
}
```

- » W klasie Vector, komponenty mogą być lub mogą używać typu T
- » Podczas tworzenia obiektu danej klasy należy ustalić (skonkretyzować) typ T
- » Dlaczego kompilator sam tego nie wymyśli???

Szablony - klasy

```
template <class T>
class Vector {
    T x1_, x2_;
public:
    Vector(T x1, T x2);
};

int main(){
    Vector v(1, 2);
}
```

- » W klasie Vector, komponenty mogą być lub mogą używać typu T
- » Podczas tworzenia obiektu danej klasy należy ustalić (skonkretyzować) typ T
- » Dlaczego kompilator sam tego nie wymyśli???
- dla obiektu v kompilator potrafi ustalić typ

Szablony - klasy

```
template <class T>
class Vector {
    T x1_, x2_;
public:
    Vector(T x1, T x2);
};
```

```
int main(){
    Vector v(1, 2);
    Vector x;
}
```

- » W klasie Vector, komponenty mogą być lub mogą używać typu T
- » Podczas tworzenia obiektu danej klasy należy ustalić (skonkretyzować) typ T
- » Dlaczego kompilator sam tego nie wymyśli???
- dla obiektu **v** kompilator potrafi ustalić typ
- dla obiektu **x** kompilator nie zna intencji programisty

Szablony - klasy

```
template <class T>
```

```
class Vector {
```

```
    T x1_, x2_;
```

```
public:
```

```
    Vector(T x1, T x2);
```

```
};
```

```
template <class T>
```

```
Vector<T>::Vector(T x1, T x2) :
```

```
    x1_(x1), x2_(x2){
```

```
}
```

- » Definicja metody (konstruktora)
- » Zwrócić uwagę na sposób definiowania zasięgu

Szablony - klasy

```
template <class T>
```

```
class Vector {
```

```
    T x1_, x2_;
```

```
public:
```

```
    Vector(T x1, T x2);
```

```
    T add(T x1, T x2);  
};
```

```
template <class T>
```

```
Vector<T>::Vector(T x1, T x2) :
```

```
    x1_(x1), x2_(x2){  
}
```

```
template <class T>
```

```
T Vector<T>::add(T x1, T x2){  
    return x1+x2;  
}
```

- » Definicja metody (konstruktora)
- » Zwrócić uwagę na sposób definiowania zasięgu
- » Definicja metody add(...) zwracającej wartość typu T

Zmiana typu szablonu “w locie”
jest niemożliwa !!!!

Szablony - zmiana typu

```
template <class T>
class Print {
public:
    void stdout(T x);
};
```

```
template <class T>
void Print<T>::stdout(T x){
    cout << x << endl;
}
```

```
int main(){
    Print<int> p;
    p.stdout(3);
}
```

» Klasa **Print** z jedną **metodą drukującą** na stdout otrzymany argument

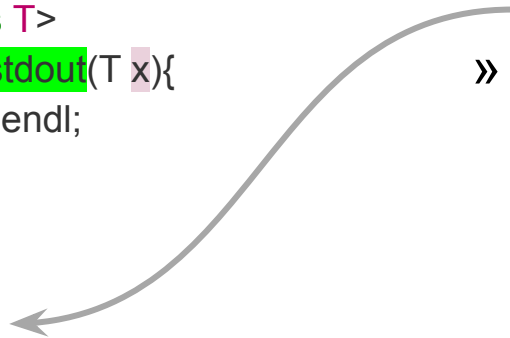
Szablony - zmiana typu

```
template <class T>
class Print {
public:
    void stdout(T x);
};
```

```
template <class T>
void Print<T>::stdout(T x){
    cout << x << endl;
}
```

```
int main(){
    Print<int> p;
    p.stdout(3);
}
```

- » Klasa **Print** z jedną **metodą drukującą** na stdout otrzymany argument
- » Podczas deklaracji obiekt został skonkretyzowany na typ **int**
- » Działa, wynik: **“3”**



Szablony - zmiana typu

```
template <class T>
class Print {
public:
    void stdout(T x);
};
```

```
template <class T>
void Print<T>::stdout(T x){
    cout << x << endl;
}
```

```
int main(){
    Print<int> p;
    p.stdout(3);
    p.stdout(3.14);
}
```

- » Klasa Print z jedną metodą drukującą na stdout otrzymany argument
- » Podczas deklaracji obiekt został skonkretyzowany na typ **int**
- » Działa, wynik: **“3”**
- » Próba zmiany typu na **double**:
 - ostrzeżenie kompilatora
 - wynik: **“3”**, zamiast “3.14”
- » Cała klasa (w tym składowe) została skonkretyzowana na typ **int** i tego już się nie da zmienić

Szablony - zmiana typu

```
template <class T>
class Print {
public:
    void stdout(T x);
};

template <class T>
void Print<T>::stdout(T x){
    cout << x << endl;
}

int main(){
    Print<int> p;
    p.stdout(3);
    Print<double> pd;
    pd.stdout(3.14);
}
```

- » Można powołać dwa obiekty, który skonkretyzują się na różne typy:
 - <int>p
 - <double>pd
- » Wynik (oczekiwany):
 - 3
 - 3.14
- » p.stdout() zostało skompilowane jako typ **int**
- » pd.stdout() zostało skompilowane jako typ **double**

Specjalizacja

typu

Szablony - specjalizacja

```
template <class T>
```

```
T add(T x1, T x2){
```

```
    return x1+x2;
```

```
}
```

```
int main(){
```

```
    cout << add(2, 3) << endl;
```

```
    cout << add(0.1, 0.3) << endl;
```

```
    cout << add(0.1f, 0.3f) << endl;
```

```
}
```

» Przykład **add(...)**

» Czy to muszą być liczby?!?

Szablony - specjalizacja

```
template <class T>
T add(T x1, T x2){
    return x1+x2;
}

int main(){
    cout << add(2, 3) << endl;
    cout << add(0.1, 0.3) << endl;
    cout << add(0.1f, 0.3f) << endl;

    string x("abc"), y("def");
    cout << add(x, y) << endl;
}
```

- » Przykład **add(...)**
- » Operator+ jest przeciążony na typie string i powoduje “połączenie” (ang. concatenation) napisów.
- » Rezultat:
 - 5
 - 0.4
 - 0.4
 - abcdef

Szablony - specjalizacja

```
struct vector{  
    double a, b;  
};  
  
template <class T>  
T add(T x1, T x2){  
    return x1+x2;  
}  
  
int main(){  
    vector x1={1,2};  
    vector x2={3,4};  
  
    vector x3 = add(x1, x2);  
}
```

- » “Nieobsługiwany” typ zmiennych
- » Zdefiniowałem własny typ
- » Próbuję go użyć na jako argument funkcji add(...) ale kończy się to katastrofą

Szablony - specjalizacja

```
struct vector{  
    double a, b;  
};  
  
template <class T>  
T add(T x1, T x2){  
    return x1+x2;  
}  
  
int main(){  
    vector x1={1,2};  
    vector x2={3,4};  
  
    vector x3 = add(x1, x2);  
}
```

- » “Nieobsługiwany” typ zmiennych
- » Zdefiniowałem **własny typ**
- » Próbuję go użyć na jako **argument** funkcji add(...) ale kończy się to katastrofą
- » Nie ma **operator+** dla **struct vector**
- » **Rozwiązania:**
 - przeciążyć operator
 - specjalizować szablon

Szablony - specjalizacja

```
struct vector{  
    double a, b;  
};
```

```
template <class T>  
T add(T x1, T x2){  
    return x1+x2;  
}
```

```
template<>  
vector add<>(vector x1, vector x2){  
    vector y;  
    y.a = x1.a + x2.a;  
    y.b = x1.b + x2.b;  
    return y;  
}
```

» Specjalizacja:

- dostarczenie ogólnego szablonu
- dostarczenie osobnej implementacji dla konkretnego typu

Szablony - specjalizacja

```
struct vector{  
    double a, b;  
};
```

```
template <class T>  
T add(T x1, T x2){  
    return x1+x2;  
}
```

```
template<>  
vector add<>(vector x1, vector x2){  
    vector y;  
    y.a = x1.a + x2.a;  
    y.b = x1.b + x2.b;  
    return y;  
}
```

- » Specjalizacja:
 - dostarczenie ogólnego szablonu
 - dostarczenie osobnej implementacji dla konkretnego typu
- » Zwrócić uwagę na **puste** klamery szablonu!!!
- » Zamiast przeciążać **operator+** dla typu vector, wykonałem **specjalną** implementację
- » Specjalizować można zarówno funkcje jak i klasy

Błędy podczas kompilacji

....

Szablony - błędy kompilacji

```
struct vector{  
    double a, b;  
};
```

```
template <class T>  
T add(T x1, T x2){  
    return x1+x2;  
}
```

```
int main(){  
    vector x1={1,2};  
    vector x2={3,4};  
  
    cout << add(x1, x2) << endl;  
}
```

» “g++ ex.cc” produkuje komunikat błędu który ma:

280 linii i 23000 znaków !!!

Szablony - błędy kompilacji

```
struct vector{  
    double a, b;  
};
```

```
template <class T>  
T add(T x1, T x2){  
    return x1+x2;  
}
```

```
int main(){  
    vector x1={1,2};  
    vector x2={3,4};  
  
    cout << add(x1, x2) << endl;  
}
```

» “g++ ex.cc” produkuje komunikat błędu który ma:

280 linii i 23000 znaków !!!



Szablony - błędy kompilacji

```
struct vector{  
    double a, b;  
};
```

```
template <class T>  
T add(T x1, T x2){  
    return x1+x2;  
}
```

```
int main(){  
    vector x1={1,2};  
    vector x2={3,4};  
  
    cout << add(x1, x2) << endl;  
}
```

» “g++ ex.cc” produkuje komunikat błędu który ma:

280 linii i 23000 znaków !!!

» Dlaczego?

» W skomplikowanej bibliotece zmieniłeś typ zmiennej w tysiącach miejsc na nieobsługiwany

Szablony - błędy kompilacji

```
struct vector{  
    double a, b;  
};  
  
template <class T>  
T add(T x1, T x2){  
    return x1+x2;  
}  
  
int main(){  
    vector x1={1,2};  
    vector x2={3,4};  
  
    cout << add(x1, x2) << endl;  
}
```

- » “g++ ex.cc” produkuje komunikat błędu który ma:
280 linii i 23000 znaków !!!
- » Dlaczego?
- » W skomplikowanej bibliotece zmieniłeś typ zmiennej w tysiącach miejsc na nieobsługiwany
- » Na: stackexchange.com znalazłem **challenge** na kod który generuje najdłuższy błąd kompilatora ;-)

Programowanie uogólnione

- » Zmienna **auto** w C++11 może zastąpić niektóre szablony
- » Rozwinięciem szablonów są **koncepty**
 - wyeliminowanie części wad szablonów
 - jeszcze bardziej generyczne podejście
 - poprawienie diagnostyki błędów (kompilator)
 - C++17
- » W C++ jest wiele innych przykładów programowania uogólnionego, nazywanego jeszcze bardziej ogólnie: **meta programowaniem**
- » Meta programowanie to na przykład modyfikacja (uzupełnienie) kodu programu w trakcie kompilacji lub nawet działania

A to wszystko był wstęp do STL

Standard Template Library

STL - co to jest?

- » Zbiór **kontenerów** (kolekcji), algorytmów, iteratorów
- » Kontener to rodzaj struktury danych służący do przechowywania danych (tego samego typu)
- » **Lepsze od tablicy**, mają wiele udogodnień:
 - możliwość dodawania/usuwania
 - możliwość zmiany rozmiaru
 - dostęp na wiele sposobów
 - ułatwienia: podawanie długości, wyszukiwanie, sortowanie, etc...
- » STL wchodzi w zakres biblioteki standardowej (dostępna w każdej* implementacji gcc)

STL - kontenery

- » **list** lista
- » **vector** tablica (można dodawać na końcu)
- » **deque** tablica podwójnie kończona (można dodawać elementy z obu końców)
- » **bitset** tablica bitowa
- » **set** uporządkowany zbiór unikalnych wartości
- » **multiset** uporządkowany zbiór wartości
- » **map** mapa poszukiwań
- » **multimap** wielokrotna mapa poszukiwań

STL - właściwości?

- » Jest biblioteką **generyczną**, prawie każdy komponent jest szablonem (może operować na dowolnym typie danych)
- » Najważniejsze są kontenery, są profilowane pod różne zastosowania
- » **Vector** trzyma obiekty w ciągłej przestrzeni pamięci
 - kompatybilne z C, szybki, swobodny dostęp
 - wstawienie elementu kosztowne
- » **List** jest listą łączoną, nie gwarantuje ciągłej przestrzeni pamięci
 - wstawienie/usunięcie elementu jest szybkie
 - nie da się indeksować jak tablicę

STL - Vector

» Wymaga dołączenia nagłówka

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

int main(){
    vector<int> v(10);

    for (size_t i=0; i<10; ++i){
        v[i] = i;
        cout << v[i] << endl;
    }
}
```

STL - Vector

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

```
int main(){
    vector<int> v(10);

    for (size_t i=0; i<10; ++i){
        v[i] = i;
        cout << v[i] << endl;
    }
}
```

- » Wymaga dołączenia nagłówka
- » Znajduje się w przestrzeni nazw `std`, więc warto ją dołączyć żeby nie pisać `std::vector`

STL - Vector

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

```
int main(){
    vector<int> v(10);

    for (size_t i=0; i<10; ++i){
        v[i] = i;
        cout << v[i] << endl;
    }
}
```

- » Wymaga dołączenia nagłówka
- » Znajduje się w przestrzeni nazw std, więc warto ją dołączyć żeby nie pisać std::vector
- » Jest to szablon

STL - Vector

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

int main(){
    vector<int> v(10);

    for (size_t i=0; i<10; ++i){
        v[i] = i;
        cout << v[i] << endl;
    }
}
```

- » Wymaga dołączenia nagłówka
- » Znajduje się w przestrzeni nazw std, więc warto ją dołączyć żeby nie pisać std::vector
- » Jest to szablon
- » **v** będzie obiektem klasy **vector** na typie **int**

STL - Vector

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

int main(){
    vector<int> v(10);

    for (size_t i=0; i<10; ++i){
        v[i] = i;
        cout << v[i] << endl;
    }
}
```

- » Wymaga dołączenia nagłówka
- » Znajduje się w przestrzeni nazw `std`, więc warto ją dołączyć żeby nie pisać `std::vector`
- » Jest to szablon
- » **v** będzie obiektem klasy **vector** na typie `int`
- » Kontener **v** będzie zawierał **10** elementów typu **int**

STL - Vector

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

int main(){
    vector<int> v(10);

    for (size_t i=0; i<10; ++i){
        v[i] = i;
        cout << v[i] << endl;
    }
}
```

- » Wymaga dołączenia nagłówka
- » Znajduje się w przestrzeni nazw `std`, więc warto ją dołączyć żeby nie pisać `std::vector`
- » Jest to szablon
- » **v** będzie obiektem klasy **vector** na typie **int**
- » Kontener **v** będzie zawierał 10 elementów typu **int**
- » Po wyjściu z zasięgu zostanie usunięty jak zmienna automatyczna

STL - Vector

» Pusty wektor

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

int main(){
    vector<int> v;

    for (size_t i=0; i<10; ++i){
        v.push_back(i);
    }

    for (size_t i=0; i<10; ++i){
        cout << v[i] << endl;
    }
}
```


STL - Vector

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

```
int main(){
    vector<int> v;

    for (size_t i=0; i<10; ++i){
        v.push_back(i);
    }

    for (size_t i=0; i<10; ++i){
        cout << v[i] << endl;
    }
}
```

- » Pusty wektor
- » Metodą **push_back(...)** dodaje na końcu nowe elementy

STL - Vector

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

int main(){
    vector<int> v;

    for (size_t i=0; i<10; ++i){
        v.push_back(i);
    }

    for (size_t i=0; i<10; ++i){
        cout << v[i] << endl;
    }
}
```

- » Pusty wektor
- » Metodą **push_back(...)** dodaję **na końcu** nowe elementy
- » **Rozmiar wektora się zmienia!!!**
- » **Nareszcie dynamiczne tablice!!!**

STL - Vector

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

int main(){
    vector<int> v;

    for (size_t i=0; i<10; ++i){
        v.push_back(i);
    }

    for (size_t i=0; i<10; ++i){
        cout << v[i] << endl;
    }
}
```

- » Pusty wektor
- » Metodą **push_back(...)** dodaję **na końcu** nowe elementy
- » **Rozmiar wektora się zmienia!!!**
- » **Nareszcie dynamiczne tablice!!!**
- » Nadal mogę korzystać z wektora jak z tablicy - przeciążony operator[]

STL - Vector - udogodnienia

```
int main(){  
    vector<int> v;  
  
    for (size_t i=0; i<10; ++i){  
        v.push_back(i);  
        cout << v.size() << endl;  
    }  
}
```

» Metoda **size()** podaje liczbę elementów w wektorze

STL - Vector - udogodnienia

```
int main(){
    vector<int> v;

    for (size_t i=0; i<10; ++i){
        v.push_back(i);
        cout << v.size() << endl;
    }

    for (size_t i=0; i<v.size(); ++i){
        cout << v.at(i) << endl;
    }
}
```

- » Metoda **size()** podaje liczbę elementów w wektorze
- » Rezultat: 1, 2, 3, ..., 10
- » Metoda **size()** umożliwia wygodne iterowanie po całej zawartości bez przechowywania aktualnej liczby elementów!!!

STL - Vector - udogodnienia

```
int main(){  
    vector<int> v;  
  
    for (size_t i=0; i<10; ++i){  
        v.push_back(i);  
        cout << v.size() << endl;  
    }  
  
    for (size_t i=0; i<v.size(); ++i){  
        cout << v.at(i) << endl;  
    }  
}
```

- » Metoda **size()** podaje liczbę elementów w wektorze
- » Rezultat: 1, 2, 3, ..., 10
- » Metoda **size()** umożliwia wygodne iterowanie po całej zawartości bez przechowywania aktualnej liczby elementów!!!
- » Indeksowanie elementów metodą **at(...)** sprawdza granice

STL - Vector - udogodnienia

```
int main(){
    vector<int> v;

    for (size_t i=0; i<10; ++i){
        v.push_back(i);
        cout << v.size() << endl;
    }

    for (size_t i=0; i<v.size(); ++i){
        cout << v.at(i) << endl;
    }
}
```

- » Metoda **size()** podaje liczbę elementów w wektorze
- » Rezultat: 1, 2, 3, ..., 10
- » Metoda **size()** umożliwia wygodne iterowanie po całej zawartości bez przechowywania aktualnej liczby elementów!!!
- » Indeksowanie elementów metodą **at(...)** sprawdza granice
- » **Semantycznie błędny kod**
v.at(v.size()) nie spowoduje błędu adresowania

STL - Vector - przykłady

.front()	dostęp do pierwszego elementu
.data()	zwraca wskaźnik do danych - tablica C
.empty()	==true jeżeli pusta tablica
.reserve()	prealokuje miejsce na dane (szybkość!)
.capacity()	zwraca ilość prealokowanego miejsca
.shrink_to_fit()	redukuje zużycie pamięci
.insert()	wstawia w dowolne miejsce nowy element
.erase()	usuwa dowolny element z wektora
.clear()	usuwa wszystkie elementy
.push_back()	dodaje na końcu element
.pop_back()	usuwa element z końca

...

Ile wersji C++ mamy?

C++ != C++

g++ 5.4.0 20160609

- » flaga **-std=c++98** pozwala wybrać wersję języka (dialektu)
 - gnu++98 GNU dialect of -std=c++98 **default**
 - gnu++03 GNU dialect of -std=c++03
 - gnu++11 GNU dialect of -std=c++11
 - gnu++14 GNU dialect of -std=c++14
 - gnu++1z GNU dialect of -std=c++1z (**experimental**)
 - c++03 2003 ISO C++ + poprawki z 2003 **-ansi**
 - c++11 2011 ISO C++ standard
 - c++14 2014 ISO C++ standard + poprawki
 - c++1z 2017 ISO C++ standard
- » dialekt GNU to ISO + rozszerzenia standardu
- » Intel, clang, VisualC++ (MS) implementują ISO + własne rozszerzenia

Standaryzacja C++

- » **1998** ISO/IEC 14882:1998[20] C++98
- » **2003** ISO/IEC 14882:2003[21] C++03
- » **2011** ISO/IEC 14882:2011[22] C++11
- » **2014** ISO/IEC 14882:2014[23] C++14
- » **2017** ISO/IEC 14882:2017[8] C++17
- » **2020** w trakcie prac C++20
- » Współczesne kompilatory implementują w całości C++11, nowsze wersje są zazwyczaj częściowo wspierane

Standaryzacja C++

- » **1998** ISO/IEC 14882:1998[20] C++98
- » **2003** ISO/IEC 14882:2003[21] C++03
- » **2011** ISO/IEC 14882:2011[22] C++11
- » **2014** ISO/IEC 14882:2014[23] C++14
- » **2017** ISO/IEC 14882:2017[8] C++17
- » **2020** w trakcie prac C++20
- » Współczesne kompilatory implementują w całości C++11, nowsze wersje są zazwyczaj częściowo wspierane
- » 1985 - pierwsza wersja języka, upubliczniona ale niepełna
- » C++98 - pierwsza wersja standardu ISO

Standaryzacja C++

- » **1998** ISO/IEC 14882:1998[20] C++98
- » **2003** ISO/IEC 14882:2003[21] C++03
- » **2011** ISO/IEC 14882:2011[22] C++11
- » **2014** ISO/IEC 14882:2014[23] C++14
- » **2017** ISO/IEC 14882:2017[8] C++17
- » **2020** w trakcie prac C++20
- » Współczesne kompilatory implementują w całości C++11, nowsze wersje są zazwyczaj częściowo wspierane
- » 1985 - pierwsza wersja języka, upubliczniona ale niepełna
- » C++98 - pierwsza wersja standardu ISO
- » Główne wydania - duże zmiany

Standaryzacja C++

- » **1998** ISO/IEC 14882:1998[20] C++98
- » **2003** ISO/IEC 14882:2003[21] C++03
- » **2011** ISO/IEC 14882:2011[22] C++11
- » **2014** ISO/IEC 14882:2014[23] C++14
- » **2017** ISO/IEC 14882:2017[8] C++17
- » **2020** w trakcie prac C++20
- » Współczesne kompilatory implementują w całości C++11, nowsze wersje są zazwyczaj częściowo wspierane
- » 1985 - pierwsza wersja języka, upubliczniona ale niepełna
- » C++98 - pierwsza wersja standardu ISO
- » Główne wydania - duże zmiany
- » Wydania pomocnicze - bugfixy

Najważniejsze zmiany C++03

- » Usprawniona inicjalizacja obiektów
- » poprawiono 93 błędy w języku
- » poprawiono 125 błędów w bibliotekach standardowych

błędy == defekty

Najważniejsze zmiany C++11

- » Ulepszone obszary: programowanie generyczne, obsługa wielowątkowości, jednolite inicjowanie, wydajność.
- » Najważniejsze rozszerzenia języka:
 - wyrażenia lambda
 - typ auto: **auto x=0;**
 - jednolita inicjalizacja obiektów **int *a=new int[3] {1,2,3};**
 - nullptr: void **f(int); f(char *); f(0)** - niejednoznaczne
 - częściowy **garbage-collector**
 - sprytnie wskaźniki **std::smart_ptr**
 - referencja do r-wartości
 - uogólnione wyrażenia stałe
 - pętla for oparta na zakresie **for(auto &x : my_tab){...}**
 - i wiele innych: <https://en.wikipedia.org/wiki/C%2B%2B11>

Najważniejsze zmiany C++14

- » Bugfixy + kilka “pomniejszych” rozszerzeń i doprecyzowań:
 - return o typie auto: **auto f(int i){ return 2*i; }**
 - rozszerzenia w obszarze wyrażeń lambda
 - doprecyzowanie constexpr
 - separator cyfr: 1'000'000 zamiast 1000000
 - binarne literały **0b**
 - literały definiowane przez użytkownika (np. 60s - sekund)
 - poprawki i optymalizacje dynamicznego zarządzania pamięcią
 - słowo kluczowe **deprecated**
 - 4 nowe funkcje związane z generowaniem liczb losowych
 - i wiele innych: <https://en.wikipedia.org/wiki/C%2B%2B14>

Najważniejsze zmiany C++17

- » Duże wydanie:
 - nowe literały UTF-8
 - if/switch z inicjalizatorami zmiennych
 - zmienne inline
 - zagnieżdżenie przestrzeni nazw
 - uaktualnienia w wyrażeniach lambda
 - nowe funkcje języka związane z meta-programowaniem (constexpr if, fold, auto jak parametr szablonu, ...)
 - rozszerzenia w bibliotece standardowej:
 - std::string_view, std::byte, std::optional, std::any...
 - i wiele innych: <https://en.wikipedia.org/wiki/C%2B%2B17>

Najważniejsze zmiany C++20

» Propozycje:

- Concepts (rozszerzenie wzorców)
- modyfikacji/rozszerzenia w wyrażeniach lambda
- trójstronny operator porównania \leq
- współprogramy (ang. *coroutines*)
- pamięć transakcyjna - alternatywny sposób synchronizacji pomiędzy wątkami
- mechanizm refleksji (modyfikacja programu w run-time)
- metaklasy
- rozszerzenia w obrębie programowania sieciowego
- i wiele innych: <https://en.wikipedia.org/wiki/C%2B%2B20>

ToDo

czy to wszystko?

Dziękuję