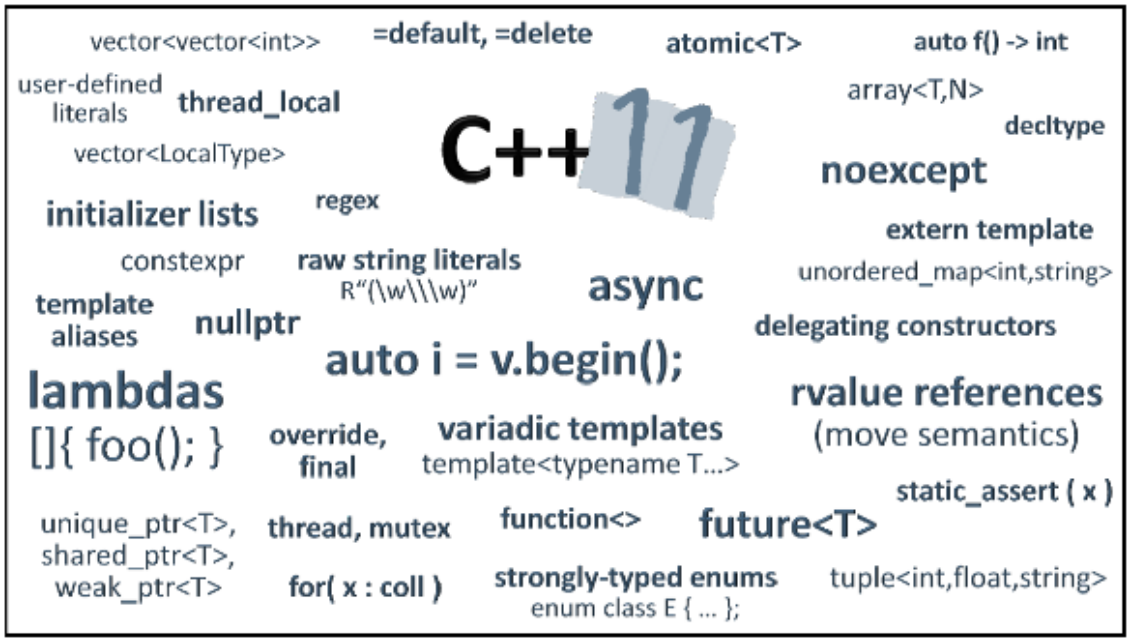
C++11

Marcin Copik

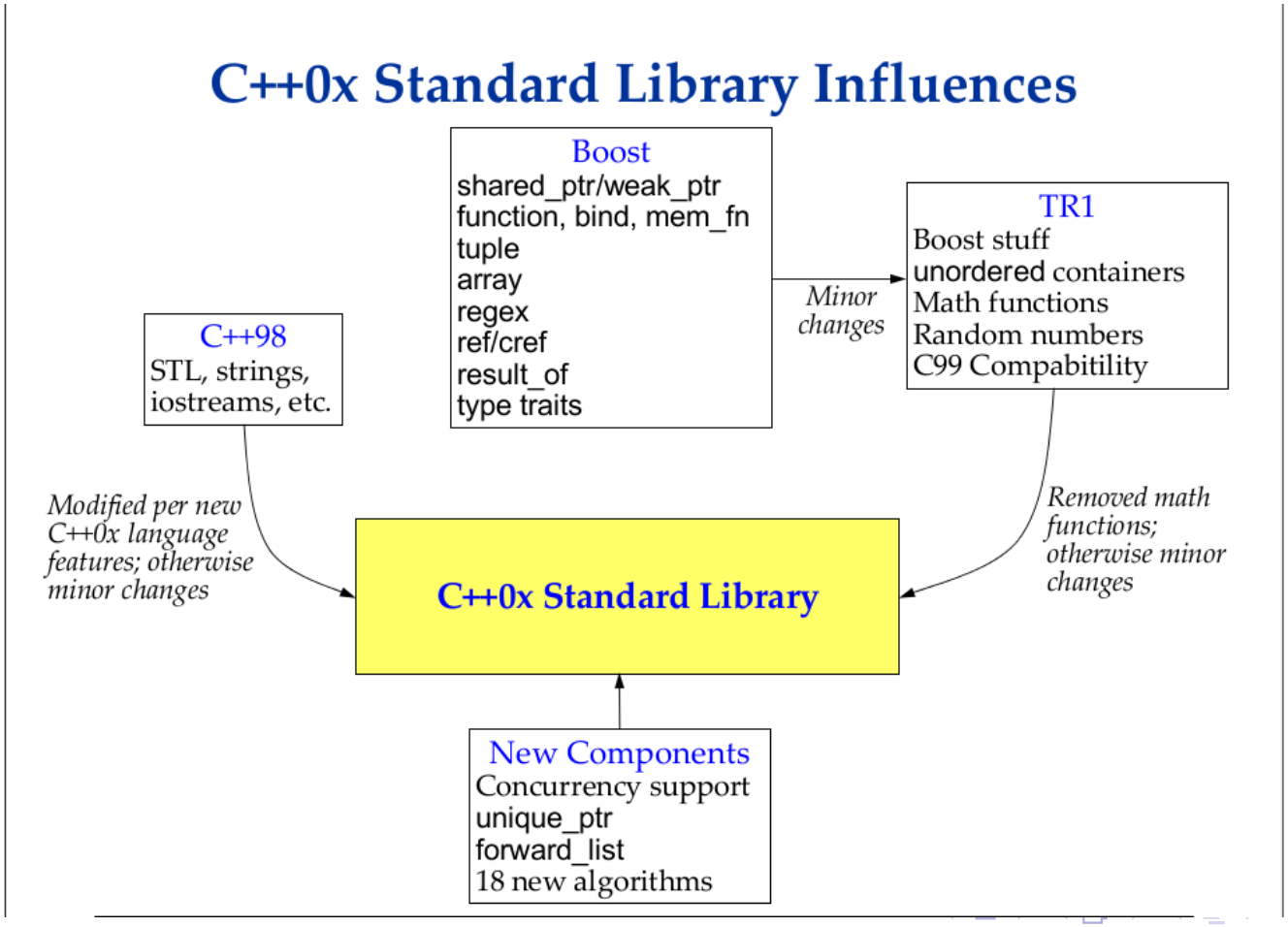
22 stycznia 2017

Spis treści

Historia


vector<vector<int>>
=default, =delete
atomic<T>
auto f() -> int
user-defined literals
thread_local
array<T,N>
decltype
vector<LocalType>
C++11
noexcept
initializer lists
regex
extern template
constexpr
raw string literals
R"(\w\\\w)"
async
unordered_map<int,string>
template aliases
nullptr
delegating constructors
auto i = v.begin();
lambdas
[]{ foo(); }
override, final
variadic templates
template<typename T...>
rvalue references
(move semantics)
static_assert (x)
unique_ptr<T>, shared_ptr<T>, weak_ptr<T>
thread, mutex
function<>
future<T>
for(x : coll)
strongly-typed enums
enum class E { ... };
tuple<int,float,string>

Historia


C++0x Standard Library Influences
Boost
shared_ptr/weak_ptr
function, bind, mem_fn
tuple
array
regex
ref/cref
result_of
type traits
Minor changes
TR1
Boost stuff
unordered containers
Math functions
Random numbers
C99 Compabitility
C++98
STL, strings, iostreams, etc.
Modified per new C++0x language features; otherwise minor changes
C++0x Standard Library
Removed math functions; otherwise minor changes
New Components
Concurrency support
unique_ptr
forward_list
18 new algorithms

L-value vs r-value

Różne ujęcia:

- może być lewą stroną przypisania
- może być pobrany jej adres
- istnieją dłużej niż jedno wyrażenie

L-value vs r-value

Różne ujęcia:

- może być lewą stroną przypisania
- może być pobrany jej adres
- istnieją dłużej niż jedno wyrażenie

L-value vs r-value

Różne ujęcia:

- może być lewą stroną przypisania
- może być pobrany jej adres
- istnieją dłużej niż jedno wyrażenie

L-value vs r-value

Różne ujęcia:

- może być lewą stroną przypisania
- może być pobrany jej adres
- istnieją dłużej niż jedno wyrażenie

Kosztowne kopiowanie

```
class X {
....
};
X f();
void g(X);
X a,b;
X c = f();
g(a+b);
```

Referencje do r-value

- zapisywane jako T &&
- przeznaczone do identyfikowania obiektów, które mogą zostać **przesunięte**
- idealne dla przechwytywania obiektów tymczasowych
- semantyka podobna do ńormalnej referencji- musi być inicjalizacja, nie może być zmieniana
- **nie** może pobrać l-value
- const T && - bez sensu

Referencje do r-value

- zapisywane jako T &&
- przeznaczone do identyfikowania obiektów, które mogą zostać **przesunięte**
- idealne dla przechwytywania obiektów tymczasowych
- semantyka podobna do ńormalnej referencji- musi być inicjalizacja, nie może być zmieniana
- **nie** może pobrać l-value
- const T && - bez sensu

Referencje do r-value

- zapisywane jako T &&
- przeznaczone do identyfikowania obiektów, które mogą zostać **przesunięte**
- idealne dla przechwytywania obiektów tymczasowych
- semantyka podobna do ńormalnej referencji- musi być inicjalizacja, nie może być zmieniana
- **nie** może pobrać l-value
- const T && - bez sensu

Referencje do r-value

- zapisywane jako T &&
- przeznaczone do identyfikowania obiektów, które mogą zostać **przesunięte**
- idealne dla przechwytywania obiektów tymczasowych
- semantyka podobna do ńormalnej referencji- musi być inicjalizacja, nie może być zmieniana
- **nie** może pobrać l-value
- const T && - bez sensu

Referencje do r-value

- zapisywane jako T &&
- przeznaczone do identyfikowania obiektów, które mogą zostać **przesunięte**
- idealne dla przechwytywania obiektów tymczasowych
- semantyka podobna do ńormalnej referencji- musi być inicjalizacja, nie może być zmieniana
- **nie** może pobrać l-value
- const T && - bez sensu

Referencje do r-value

- zapisywane jako T &&
- przeznaczone do identyfikowania obiektów, które mogą zostać **przesunięte**
- idealne dla przechwytywania obiektów tymczasowych
- semantyka podobna do ńormalnej referencji- musi być inicjalizacja, nie może być zmieniana
- **nie** może pobrać l-value
- const T && - bez sensu

Referencje do r-value

- zapisywane jako T &&
- przeznaczone do identyfikowania obiektów, które mogą zostać **przesunięte**
- idealne dla przechwytywania obiektów tymczasowych
- semantyka podobna do ńormalnej referencji- musi być inicjalizacja, nie może być zmieniana
- **nie** może pobrać l-value
- const T && - bez sensu

Kosztowne kopiowanie

```
class X {
....
};
X a;
...
g(a); //jesli przesuniemy...
std::cout << a; //...to tutaj a bedzie juz inne!
```

Zmiany w interfejsie klas

- gdy zwracamy przez wartość z metody, to bez const (np. przeciążone operatory)
- **move constructor** T(T &&)
- **move assignment** T & operator=(T&&)

Zmiany w interfejsie klas

- gdy zwracamy przez wartość z metody, to bez const (np. przeciążone operatory)
- **move constructor** T(T &&)
- **move assignment** T & operator=(T&&)

Zmiany w interfejsie klas

- gdy zwracamy przez wartość z metody, to bez const (np. przeciążone operatory)
- **move constructor** T(T &&)
- **move assignment** T & operator=(T&&)

Zmiany w interfejsie klas

- gdy zwracamy przez wartość z metody, to bez const (np. przeciążone operatory)
- **move constructor** T(T &&)
- **move assignment** T & operator=(T&&)

std::move

```
class X {
Y y;
public:
X(X && rhs): y(std::move(rhs.y)) {}
X & operator=(X && rhs){
{
y = std::move(rhs.y);
}
};
```

Zmiany w interfejsie klas

Kiedy zostaną wygenerowane automatyczne?:

- każde pole klasy musi być przesuwalne
- deklaracja operacji przesuwania **blokuje** generację operacji kopiowania
- analogiczne, deklaracja operacji kopiowania **blokuje** generację operacji przesuwania

Zmiany w interfejsie klas

Kiedy zostaną wygenerowane automatyczne?:

- każde pole klasy musi być przesuwalne
- deklaracja operacji przesuwania **blokuje** generację operacji kopiowania
- analogiczne, deklaracja operacji kopiowania **blokuje** generację operacji przesuwania

Zmiany w interfejsie klas

Kiedy zostaną wygenerowane automatyczne?:

- każde pole klasy musi być przesuwalne
- deklaracja operacji przesuwania **blokuje** generację operacji kopiowania
- analogiczne, deklaracja operacji kopiowania **blokuje** generację operacji przesuwania

Zmiany w interfejsie klas

Kiedy zostaną wygenerowane automatyczne?:

- każde pole klasy musi być przesuwalne
- deklaracja operacji przesuwania **blokuje** generację operacji kopiowania
- analogiczne, deklaracja operacji kopiowania **blokuje** generację operacji przesuwania

Inicjalizacja w C++98

Kiedy nie można było bezpośrednio dokonać inicjalizacji?:

- w konstruktorach kontenerów
- w liście inicjalizacyjnej konstruktora
- przy dynamicznej alokacji

Inicjalizacja w C++98

Kiedy nie można było bezpośrednio dokonać inicjalizacji?:

- w konstruktorach kontenerów
- w liście inicjalizacyjnej konstruktora
- przy dynamicznej alokacji

Inicjalizacja w C++98

Kiedy nie można było bezpośrednio dokonać inicjalizacji?:

- w konstruktorach kontenerów
- w liście inicjalizacyjnej konstruktora
- przy dynamicznej alokacji

Inicjalizacja w C++98

Kiedy nie można było bezpośrednio dokonać inicjalizacji?:

- w konstruktorach kontenerów
- w liście inicjalizacyjnej konstruktora
- przy dynamicznej alokacji

Inicjalizacja w C++11

```
const vector<int> x{1,2,3,4};
const double * ptr = new double[2]{0.5,0.1};
void f(const vector<int> &);
f({0,1,2,3});
```

Semantyka inicjalizacji

- *aggregates* - tablice lub niektóre klasy
- *non-aggregates* - klasy które mają zdefiniowany konstruktor LUB klasę bazową LUB funkcje wirtualne LUB dane prywatne/chronione

Semantyka inicjalizacji

- *aggregates* - tablice lub niektóre klasy
- *non-aggregates* - klasy które mają zdefiniowany konstruktor LUB klasę bazową LUB funkcje wirtualne LUB dane prywatne/chronione

Semantyka inicjalizacji

- *aggregates* - tablice lub niektóre klasy
- *non-aggregates* - klasy które mają zdefiniowany konstruktor LUB klasę bazową LUB funkcje wirtualne LUB dane prywatne/chronione

Semantyka inicjalizacji

- inicjalizacja pól po kolei
- za dużo danych? błąd
- za mało danych? typy wbudowane do zera, typy użytkownika - domyślne konstruktory

Semantyka inicjalizacji

- inicjalizacja pól po kolei
- za dużo danych? błąd
- za mało danych? typy wbudowane do zera, typy użytkownika - domyślne konstruktory

Semantyka inicjalizacji

- inicjalizacja pól po kolei
- za dużo danych? błąd
- za mało danych? typy wbudowane do zera, typy użytkownika - domyślne konstruktory

Semantyka inicjalizacji

- inicjalizacja pól po kolei
- za dużo danych? błąd
- za mało danych? typy wbudowane do zera, typy użytkownika - domyślne konstruktory

Składnia z przypisaniem

- T var = data
- działa wszędzie tam, gdzie ma sens!
- nie może wywoływać konstruktorów oznaczonych jako *explicit*

Składnia z przypisaniem

- T var = data
- działa wszędzie tam, gdzie ma sens!
- nie może wywoływać konstruktorów oznaczonych jako *explicit*

Składnia z przypisaniem

- T var = data
- działa wszędzie tam, gdzie ma sens!
- nie może wywoływać konstruktorów oznaczonych jako *explicit*

Składnia z przypisaniem

- T var = data
- działa wszędzie tam, gdzie ma sens!
- nie może wywoływać konstruktorów oznaczonych jako *explicit*

Składnia z przypisanien

```
const vector<int> func()
{
return = {1,2,3,4};
}
const double * ptr = new double[2] = {0.5,0.1};
void f(const vector<int> &);
f(={0,1,2,3});
```

Lista inicjalizacyjna

- uogólnienie nowej składni inicjalizacji
- faworyzowana konwersja do listy
- trzy ważne metody - size, begin, end
- można używać jako argument każdej funkcji

Lista inicjalizacyjna

- uogólnienie nowej składni inicjalizacji
- faworyzowana konwersja do listy
- trzy ważne metody - size, begin, end
- można używać jako argument każdej funkcji

Lista inicjalizacyjna

- uogólnienie nowej składni inicjalizacji
- faworyzowana konwersja do listy
- trzy ważne metody - size, begin, end
- można używać jako argument każdej funkcji

Lista inicjalizacyjna

- uogólnienie nowej składni inicjalizacji
- faworyzowana konwersja do listy
- trzy ważne metody - size, begin, end
- można używać jako argument każdej funkcji

Lista inicjalizacyjna

- uogólnienie nowej składni inicjalizacji
- faworyzowana konwersja do listy
- trzy ważne metody - size, begin, end
- można używać jako argument każdej funkcji

Lista inicjalizacyjna

```
class X{
public:
X(int a,int b){}
X(string a,string b){}
X(std::initializer_list<int> a){}
};
X x1(1,2);
X x2{1,2};
X x3{"1","2"}; //blad
X x4{1,2.0}; //blad
```

Dlaczego?

- szybkie, czyste, ładne funktory
- idealne jako funkcja porównująca dla sortowania czy wyszukiwania
- eliminuje konieczność tworzenia dodatkowych klas

Dlaczego?

- szybkie, czyste, ładne funktory
- idealne jako funkcja porównująca dla sortowania czy wyszukiwania
- eliminuje konieczność tworzenia dodatkowych klas

Dlaczego?

- szybkie, czyste, ładne funktory
- idealne jako funkcja porównująca dla sortowania czy wyszukiwania
- eliminuje konieczność tworzenia dodatkowych klas

Dlaczego?

- szybkie, czyste, ładne funktory
- idealne jako funkcja porównująca dla sortowania czy wyszukiwania
- eliminuje konieczność tworzenia dodatkowych klas

Przykład

```
std::vector<int> v;
...
int bound = 5;
auto it = std::find_if(v.cbegin(), v.cend(),[](in
```

Zasięg zmiennych

- lambda domyślnie posiada dostęp do zmiennych w danym zasięgu
- jeśli opuszczamy zasięg(np. zwracamy lambdę z funkcji), to gubimy dostęp do zmiennych
- zmienne mogą być złapane

Zasięg zmiennych

- lambda domyślnie posiada dostęp do zmiennych w danym zasięgu
- jeśli opuszczamy zasięg(np. zwracamy lambdę z funkcji), to gubimy dostęp do zmiennych
- zmienne mogą być złapane

Zasięg zmiennych

- lambda domyślnie posiada dostęp do zmiennych w danym zasięgu
- jeśli opuszczamy zasięg(np. zwracamy lambdę z funkcji), to gubimy dostęp do zmiennych
- zmienne mogą być złapane

Zasięg zmiennych

- lambda domyślnie posiada dostęp do zmiennych w danym zasięgu
- jeśli opuszczamy zasięg(np. zwracamy lambdę z funkcji), to gubimy dostęp do zmiennych
- zmienne mogą być złapane

Przykład

```
std::vector<int> v;
...
{
int bound = 5, exclude = 0;
auto it = std::find_if(v.cbegin(), v.cend(),[boun
&& i < bound; });
}
```

Przykład z domyślnym żłapaniem"

```
std::vector<int> v;
...
{
int bound = 5, exclude = 0;
auto it = std::find_if(v.cbegin(), v.cend(),[=](i
&& i < bound; });
auto it2 = std::find_if(v.cbegin(), v.cend(),[&](
&& i < bound; });
}
```

Źłapanie pól klasy"

```
class X{
std::vector<int> v;
int bound;
int exclude;
public:
void f();
void g();
}
void X::f() {
auto it = std::find_if(v.cbegin(), v.cend(),[this
&& i < bound; });
}
void X::g() {
```

Zwracanie wartości

- domyślnie **void**
- jeśli lambda jest postaci *return expr*, to zwracamy typ expr
- w pozostałych przypadkach - tzw. trailing return type

Zwracanie wartości

- domyślnie **void**
- jeśli lambda jest postaci *return expr*, to zwracamy typ expr
- w pozostałych przypadkach - tzw. trailing return type

Zwracanie wartości

- domyślnie **void**
- jeśli lambda jest postaci *return expr*, to zwracamy typ expr
- w pozostałych przypadkach - tzw. trailing return type

Zwracanie wartości

- domyślnie **void**
- jeśli lambda jest postaci *return expr*, to zwracamy typ expr
- w pozostałych przypadkach - tzw. trailing return type

Zwracanie wartości

```
class X{
std::vector<int> v;
int bound;
int exclude;
public:
void f();
void g();
auto h() -> double;
}
void X::f() {
auto it = std::find_if(v.cbegin(), v.cend(),[this
&& i < bound; });
}
```

Inne uwagi

- trzeba uważać na referencje!
- lambdy mogą robić praktycznie to samo co każda funkcja
- do przechowywania używamy **auto** albo szablonu **std::function**

Inne uwagi

- trzeba uważać na referencje!
- lambdy mogą robić praktycznie to samo co każda funkcja
- do przechowywania używamy **auto** albo szablonu **std::function**

Inne uwagi

- trzeba uważać na referencje!
- lambdy mogą robić praktycznie to samo co każda funkcja
- do przechowywania używamy **auto** albo szablonu **std::function**

Inne uwagi

- trzeba uważać na referencje!
- lambdy mogą robić praktycznie to samo co każda funkcja
- do przechowywania używamy **auto** albo szablonu **std::function**

std::thread

- przyjmuje dowolny obiekt, który da się wywołać
- asynchroniczne wykonanie
- przekazywanie argumentów przez referencję - ryzykowne
- kopiuj lub zapewnij żywotność zmiennych

std::thread

- przyjmuje dowolny obiekt, który da się wywołać
- asynchroniczne wykonanie
- przekazywanie argumentów przez referencję - ryzykowne
- kopiuj lub zapewnij żywotność zmiennych

std::thread

- przyjmuje dowolny obiekt, który da się wywołać
- asynchroniczne wykonanie
- przekazywanie argumentów przez referencję - ryzykowne
- kopiuj lub zapewnij żywotność zmiennych

std::thread

- przyjmuje dowolny obiekt, który da się wywołać
- asynchroniczne wykonanie
- przekazywanie argumentów przez referencję - ryzykowne
- kopiuj lub zapewnij żywotność zmiennych

std::thread

- przyjmuje dowolny obiekt, który da się wywołać
- asynchroniczne wykonanie
- przekazywanie argumentów przez referencję - ryzykowne
- kopiuj lub zapewnij żywotność zmiennych

std::thread - przykład 1

```
int temp = 2;
std::string str("napis");
void f(int & a,std::string & b);
std::thread t(f,temp,str);
std::thread t2([=]{f(temp,str)});
```

std::thread - przykład 2

```
int temp = 2;
std::string str("napis");
void f(int & a,std::string & b);
std::thread t(f,temp,std::ref(str));
std::thread t2([=,&str]{f(temp,str)});
```

Zwracanie wartości z wątku

- używamy w tym celu szablonu std::async
- możliwe zwrócenie wartości lub wyjątku
- wartość przekazywana do std::future
- kopiuj lub zapewnij żywotność zmiennych

Zwracanie wartości z wątku

- używamy w tym celu szablonu std::async
- możliwe zwrócenie wartości lub wyjątku
- wartość przekazywana do std::future
- kopiuj lub zapewnij żywotność zmiennych

Zwracanie wartości z wątku

- używamy w tym celu szablonu std::async
- możliwe zwrócenie wartości lub wyjątku
- wartość przekazywana do std::future
- kopiuj lub zapewnij żywotność zmiennych

Zwracanie wartości z wątku

- używamy w tym celu szablonu std::async
- możliwe zwrócenie wartości lub wyjątku
- wartość przekazywana do std::future
- kopiuj lub zapewnij żywotność zmiennych

Zwracanie wartości z wątku

- używamy w tym celu szablonu std::async
- możliwe zwrócenie wartości lub wyjątku
- wartość przekazywana do std::future
- kopiuj lub zapewnij żywotność zmiennych

std::async

- **std::launch::async** - funkcja będzie uruchumiona w nowym wątku
- **std::launch::async** - funkcja będzie odpalona w tym samym wątku

std::async

- **std::launch::async** - funkcja będzie uruchumiona w nowym wątku
- **std::launch::async** - funkcja będzie odpalona w tym samym wątku

std::async

- **std::launch::async** - funkcja będzie uruchumiona w nowym wątku
- **std::launch::async** - funkcja będzie odpalona w tym samym wątku

std::future - odczytanie wartości

- **get** - blokuje i oczekuje na zakończenie wątku
- *wait_for* - oczekuje określoną ilość czasu, może być zero!

std::future - odczytanie wartości

- **get** - blokuje i oczekuje na zakończenie wątku
- *wait_for* - oczekuje określoną ilość czasu, może być zero!

std::future - odczytanie wartości

- **get** - blokuje i oczekuje na zakończenie wątku
- *wait_for* - oczekuje określoną ilość czasu, może być zero!

std::thread - przykład 2

```
std::future<double> f =
std::async(std::launch::async, []{ return doSth(1
while (f.wait_for(std::chrono::seconds(0)) !=
std::future_status::ready) {
//rob cos
}
double val = f.get();
double val2 = f.wait(); // zadziala jak f.get, za
```

Dalsze zagadnienia

- mutex - wzajemne wykluczenie, umożliwia chronienie zasobów i krytycznych sekcji
- **std::mutex** - cztery możliwe wersje, włącznie z timeoutem
- **std::lock_guard** - blokuje mutex, odlobkuje w destruktorze
- **std::condition_variable** - umożliwia oczekiwanie i budzenie wątków

Dalsze zagadnienia

- mutex - wzajemne wykluczenie, umożliwia chronienie zasobów i krytycznych sekcji
- **std::mutex** - cztery możliwe wersje, włącznie z timeoutem
- **std::lock_guard** - blokuje mutex, odlobkuje w destruktorze
- **std::condition_variable** - umożliwia oczekiwanie i budzenie wątków

Dalsze zagadnienia

- mutex - wzajemne wykluczenie, umożliwia chronienie zasobów i krytycznych sekcji
- **std::mutex** - cztery możliwe wersje, włącznie z timeoutem
- **std::lock_guard** - blokuje mutex, odlobkuje w destruktorze
- **std::condition_variable** - umożliwia oczekiwanie i budzenie wątków

Dalsze zagadnienia

- mutex - wzajemne wykluczenie, umożliwia chronienie zasobów i krytycznych sekcji
- **std::mutex** - cztery możliwe wersje, włącznie z timeoutem
- **std::lock_guard** - blokuje mutex, odlobkuje w destruktorze
- **std::condition_variable** - umożliwia oczekiwanie i budzenie wątków

Dalsze zagadnienia

- mutex - wzajemne wykluczenie, umożliwia chronienie zasobów i krytycznych sekcji
- **std::mutex** - cztery możliwe wersje, włącznie z timeoutem
- **std::lock_guard** - blokuje mutex, odlobkuje w destruktorze
- **std::condition_variable** - umożliwia oczekiwanie i budzenie wątków

Czym jest inteligentny wskaźnik?

- bezpieczne przekazywanie wskaźników
- nie zostanie zniszczony, dopóki ktoś go używa
- zostanie zniszczony gdy nikt nie używa

Czym jest inteligentny wskaźnik?

- bezpieczne przekazywanie wskaźników
- nie zostanie zniszczony, dopóki ktoś go używa
- zostanie zniszczony gdy nikt nie używa

Czym jest inteligentny wskaźnik?

- bezpieczne przekazywanie wskaźników
- nie zostanie zniszczony, dopóki ktoś go używa
- zostanie zniszczony gdy nikt nie używa

Czym jest inteligentny wskaźnik?

- bezpieczne przekazywanie wskaźników
- nie zostanie zniszczony, dopóki ktoś go używa
- zostanie zniszczony gdy nikt nie używa

auto_ptr

- jednokrotne wystąpienie
- skopiowanie sprawia, że właściciel "traci"go
- w C++11 zastąpiony przez unique_ptr

auto_ptr

- jednokrotne wystąpienie
- skopiowanie sprawia, że właściciel "traci"go
- w C++11 zastąpiony przez unique_ptr

auto_ptr

- jednokrotne wystąpienie
- skopiowanie sprawia, że właściciel "traci"go
- w C++11 zastąpiony przez unique_ptr

auto_ptr

- jednokrotne wystąpienie
- skopiowanie sprawia, że właściciel "traci"go
- w C++11 zastąpiony przez unique_ptr

shared_ptr

- zlicza referencje i niszczy dane, gdy licznik osiągnie 0
- **get** umożliwia dostęp do danych
- w przeciwieństwie do auto_ptr, może przechowywać niekompletne typy
- w przeciwieństwie do auto_ptr, można rzutować na klasy bazowe

shared_ptr

- zlicza referencje i niszczy dane, gdy licznik osiągnie 0
- **get** umożliwia dostęp do danych
- w przeciwieństwie do auto_ptr, może przechowywać niekompletne typy
- w przeciwieństwie do auto_ptr, można rzutować na klasy bazowe

shared_ptr

- zlicza referencje i niszczy dane, gdy licznik osiągnie 0
- **get** umożliwia dostęp do danych
- w przeciwieństwie do auto_ptr, może przechowywać niekompletne typy
- w przeciwieństwie do auto_ptr, można rzutować na klasy bazowe

shared_ptr

- zlicza referencje i niszczy dane, gdy licznik osiągnie 0
- **get** umożliwia dostęp do danych
- w przeciwieństwie do auto_ptr, może przechowywać niekompletne typy
- w przeciwieństwie do auto_ptr, można rzutować na klasy bazowe

shared_ptr

- zlicza referencje i niszczy dane, gdy licznik osiągnie 0
- **get** umożliwia dostęp do danych
- w przeciwieństwie do auto_ptr, może przechowywać niekompletne typy
- w przeciwieństwie do auto_ptr, można rzutować na klasy bazowe

shared_ptr

```
class X;
class Y {};
class Z : public Y{};
X * create_X();
void doSth(std::shared_ptr<Y>);

std::shared_ptr<X> ptr(create_X());
std::shared_ptr<Z> ptrZ(new Z());
doSth(ptrZ);
```

weak_ptr

- służą do obserwowania innych danych
- działa tak samo jak zwykły wskaźnik, tylko wie kiedy dane są zniszczone
- metod **expired** umożliwia sprawdzenie, czy obserwowany obiekt nadal istnieje
- w ogóle nie działa jak wskaźnik!

weak_ptr

- służą do obserwowania innych danych
- działa tak samo jak zwykły wskaźnik, tylko wie kiedy dane są zniszczone
- metod **expired** umożliwia sprawdzenie, czy obserwowany obiekt nadal istnieje
- w ogóle nie działa jak wskaźnik!

weak_ptr

- służą do obserwowania innych danych
- działa tak samo jak zwykły wskaźnik, tylko wie kiedy dane są zniszczone
- metod **expired** umożliwia sprawdzenie, czy obserwowany obiekt nadal istnieje
- w ogóle nie działa jak wskaźnik!

weak_ptr

- służą do obserwowania innych danych
- działa tak samo jak zwykły wskaźnik, tylko wie kiedy dane są zniszczone
- metod **expired** umożliwia sprawdzenie, czy obserwowany obiekt nadal istnieje
- w ogóle nie działa jak wskaźnik!

weak_ptr

- służą do obserwowania innych danych
- działa tak samo jak zwykły wskaźnik, tylko wie kiedy dane są zniszczone
- metod **expired** umożliwia sprawdzenie, czy obserwowany obiekt nadal istnieje
- w ogóle nie działa jak wskaźnik!

unique_ptr

- następca auto_ptr
- umożliwa dziedziczenie i przechowywanie niekompletnych typów
- może wskazywać na tablicę, wtedy umożliwa indeksowanie, ale bez dziedziczenia

unique_ptr

- następca auto_ptr
- umożliwa dziedziczenie i przechowywanie niekompletnych typów
- może wskazywać na tablicę, wtedy umożliwa indeksowanie, ale bez dziedziczenia

unique_ptr

- następca auto_ptr
- umożliwa dziedziczenie i przechowywanie niekompletnych typów
- może wskazywać na tablicę, wtedy umożliwa indeksowanie, ale bez dziedziczenia

unique_ptr

- następca auto_ptr
- umożliwa dziedziczenie i przechowywanie niekompletnych typów
- może wskazywać na tablicę, wtedy umożliwa indeksowanie, ale bez dziedziczenia

Wyrażenia regularne

- **std::regex** przechowuje wyrażenie regularne
- **std::match_results** przechowuje wyniki działań
- **std::regex_search** i **std::regex_replace** umożliwiają wyszukiwanie lub zastępowanie wzorca

Wyrażenia regularne

- **std::regex** przechowuje wyrażenie regularne
- **std::match_results** przechowuje wyniki działań
- **std::regex_search** i **std::regex_replace** umożliwiają wyszukiwanie lub zastępowanie wzorca

Wyrażenia regularne

- **std::regex** przechowuje wyrażenie regularne
- **std::match_results** przechowuje wyniki działań
- **std::regex_search** i **std::regex_replace** umożliwiają wyszukiwanie lub zastępowanie wzorca

Wyrażenia regularne

- **std::regex** przechowuje wyrażenie regularne
- **std::match_results** przechowuje wyniki działań
- **std::regex_search** i **std::regex_replace** umożliwiają wyszukiwanie lub zastępowanie wzorca

Liczby pseudolosowe

- rand() odziedziczony po C
- niezbyt dobra jakość liczb, kłopotliwe konwersje do np. liczb zmiennoprzecinkowych
- brak wsparcia dla innych silników, różnych dystrybucji

Liczby pseudolosowe

- rand() odziedziczony po C
- niezbyt dobra jakość liczb, kłopotliwe konwersje do np. liczb zmiennoprzecinkowych
- brak wsparcia dla innych silników, różnych dystrybucji

Liczby pseudolosowe

- rand() odziedziczony po C
- niezbyt dobra jakość liczb, kłopotliwe konwersje do np. liczb zmiennoprzecinkowych
- brak wsparcia dla innych silników, różnych dystrybucji

Liczby pseudolosowe

- rand() odziedziczony po C
- niezbyt dobra jakość liczb, kłopotliwe konwersje do np. liczb zmiennoprzecinkowych
- brak wsparcia dla innych silników, różnych dystrybucji

Boost.Random -> C++11

- silnik umożliwia losowanie liczb
- szablony dystrybucji pozwalają na automatyczne konwersje

Boost.Random -> C++11

- silnik umożliwia losowanie liczb
- szablony dystrybucji pozwalają na automatyczne konwersje

Boost.Random -> C++11

- silnik umożliwia losowanie liczb
- szablony dystrybucji pozwalają na automatyczne konwersje

Przykład

```
std::random_device rd; //seed
std::mt19937 engine(rd());
std::uniform_real_distribution<> dist(0, 1);
std::cout << dist(engine) << std::endl;
```

Krotka

- uogólnienie std::pair
- **std::get<i>** pozwala na "wyciągnięcie"i-tej danej z krotki
- **std::tie** pozwala na "wyciągnięcie"większej ilości danych
- **std::make_tuple** 'dynamicznie' tworzy krotkę
- całość oparta na szablonach, więc nie jest możliwe przejście pętlą po krotce

Krotka

- uogólnienie std::pair
- **std::get<i>** pozwala na "wyciągnięcie"i-tej danej z krotki
- **std::tie** pozwala na "wyciągnięcie"większej ilości danych
- **std::make_tuple** 'dynamicznie' tworzy krotkę
- całość oparta na szablonach, więc nie jest możliwe przejście pętlą po krotce

Krotka

- uogólnienie std::pair
- **std::get<i>** pozwala na "wyciągnięcie"i-tej danej z krotki
- **std::tie** pozwala na "wyciągnięcie"większej ilości danych
- **std::make_tuple** 'dynamicznie' tworzy krotkę
- całość oparta na szablonach, więc nie jest możliwe przejście pętlą po krotce

Krotka

- uogólnienie std::pair
- **std::get<i>** pozwala na "wyciągnięcie"i-tej danej z krotki
- **std::tie** pozwala na "wyciągnięcie"większej ilości danych
- **std::make_tuple** 'dynamicznie' tworzy krotkę
- całość oparta na szablonach, więc nie jest możliwe przejście pętlą po krotce

Krotka

- uogólnienie std::pair
- **std::get<i>** pozwala na "wyciągnięcie"i-tej danej z krotki
- **std::tie** pozwala na "wyciągnięcie"większej ilości danych
- **std::make_tuple** 'dynamicznie' tworzy krotkę
- całość oparta na szablonach, więc nie jest możliwe przejście pętlą po krotce

Krotka

- uogólnienie std::pair
- **std::get<i>** pozwala na "wyciągnięcie"i-tej danej z krotki
- **std::tie** pozwala na "wyciągnięcie"większej ilości danych
- **std::make_tuple** 'dynamicznie' tworzy krotkę
- całość oparta na szablonach, więc nie jest możliwe przejście pętlą po krotce

Przykład

```
double db = 0.01;
std::tuple<int,int,double> x(5,7,db);
std::cout << std::get<0>(x) << std::endl;
int x1,y1; double z1;
std::tie(x1,y1,z1) =
std::make_tuple(-1,100,0.16);
std::cout << x1 << "␣" << y1 << "␣" << z1
<< std::endl;
```

Tablice haszujące

- **unordered_set** i **unordered_map**
- tylko iteratory przechodzące struktury 'do przodu'
- funkcje haszujące zaimplementowane dla typów wbudowanych, stringa, wskaźników inteligentnych
- implementacja własnej funkcji - specjalizacja szablonu **Hash<T>** lub przekazanie funktora

Tablice haszujące

- **unordered_set** i **unordered_map**
- tylko iteratory przechodzące struktury 'do przodu'
- funkcje haszujące zaimplementowane dla typów wbudowanych, stringa, wskaźników inteligentnych
- implementacja własnej funkcji - specjalizacja szablonu **Hash<T>** lub przekazanie funktora

Tablice haszujące

- **unordered_set** i **unordered_map**
- tylko iteratory przechodzące struktury 'do przodu'
- funkcje haszujące zaimplementowane dla typów wbudowanych, stringa, wskaźników inteligentnych
- implementacja własnej funkcji - specjalizacja szablonu **Hash<T>** lub przekazanie funktora

Tablice haszujące

- **unordered_set** i **unordered_map**
- tylko iteratory przechodzące struktury 'do przodu'
- funkcje haszujące zaimplementowane dla typów wbudowanych, stringa, wskaźników inteligentnych
- implementacja własnej funkcji - specjalizacja szablonu **Hash<T>** lub przekazanie funktora

Tablice haszujące

- **unordered_set** i **unordered_map**
- tylko iteratory przechodzące struktury 'do przodu'
- funkcje haszujące zaimplementowane dla typów wbudowanych, stringa, wskaźników inteligentnych
- implementacja własnej funkcji - specjalizacja szablonu **Hash<T>** lub przekazanie funktora

Przykład

```
class X {
...
};
struct XHasher: public
std::unary_function<X, std::size_t> {
    std::size_t operator()(const X & x) const {
...};
 };
 X x1(1),x2(2),x3(3);
std::unordered_map<X, std::string, XHasher>
structure{{x1,"a"},{x2,"b"},{x3,"cd"}};
std::cout << structure[x1] << std::endl;
```

Tablica stałego rozmiaru

- szablony mający zastąpić klasyczne tablice
- posiada większość zalet STL - iteratory
- metoda **data** zwraca wskaźnik dający bezpośredni dostęp do danych
- znamy cały czas wielkość tablicy

Tablica stałego rozmiaru

- szablony mający zastąpić klasyczne tablice
- posiada większość zalet STL - iteratory
- metoda **data** zwraca wskaźnik dający bezpośredni dostęp do danych
- znamy cały czas wielkość tablicy

Tablica stałego rozmiaru

- szablony mający zastąpić klasyczne tablice
- posiada większość zalet STL - iteratory
- metoda **data** zwraca wskaźnik dający bezpośredni dostęp do danych
- znamy cały czas wielkość tablicy

Tablica stałego rozmiaru

- szablony mający zastąpić klasyczne tablice
- posiada większość zalet STL - iteratory
- metoda **data** zwraca wskaźnik dający bezpośredni dostęp do danych
- znamy cały czas wielkość tablicy

Tablica stałego rozmiaru

- szablony mający zastąpić klasyczne tablice
- posiada większość zalet STL - iteratory
- metoda **data** zwraca wskaźnik dający bezpośredni dostęp do danych
- znamy cały czas wielkość tablicy

Przykład

```
std::array<int, 3> a1{ {1,2,3} };
std::sort(a1.begin(), a1.end());
auto it = a1.begin();
```

Bibliografia

- 'An Effective C++11/14 Sampler' Scotta Mayersa na Channel9 M$
- http://en.cppreference.com/w/cpp
- 'Effective C++11' Meyersa - w tym roku

Bibliografia

- 'An Effective C++11/14 Sampler' Scotta Mayersa na Channel9 M$
- http://en.cppreference.com/w/cpp
- 'Effective C++11' Meyersa - w tym roku

Bibliografia

- 'An Effective C++11/14 Sampler' Scotta Mayersa na Channel9 M$
- http://en.cppreference.com/w/cpp
- 'Effective C++11' Meyersa - w tym roku