Egzamin Zaawansowany CPP

Spis rozdziałów

3.8 - Zrównoleglenie kodu przy pomocy std::async

3.9 - Do czego służy std::condition_variable?

3.10 - execution policy

Inteligentne wskaźniki

4.1 - Podstawowe cechy std::unique_ptr, std::shared_ptr, std::weak_ptr

4.2 - Do czego służą std::make_unique, std::make_shared

4.3 - Inteligentne wskaźniki a zasada RAII

Elementy C++17

5.1 - if constexpr

5.2 - constexpr

Szablony wariadyczne

6.1 - Packet expansion np. (2*args)

6.2 - fold expansion np. (args + …)

RAII

7.1 - Przykłady użycia w C++ (np. pliki, muteksy, inteligentne wskaźniki, kontenery C++, etc.)

Sanitizery

8.1 - Do czego służą ASAN, TSAN, UBSAN

Lambda

Motywacja: Sięganie w funkcjach do zmiennych nielokalnych jest wygodne, ale zdecydowanie niezalecane. w C++ pisze się duże programy, zmienne nielokalne bardzo utrudniają zapanowanie nad poprawnością działania programu.

Lambda w C++ składa się z pięciu elementów, z czego większość jest opcjonalna.

- [] - Kwadratowe nawiasy - początek wyrażenia lambda. W nich można wpisać listę przechwytywanych nazw
- () - Nawiasy okrągłe - analogicznie, jak przy zwykłej funkcji, podajemy tutaj argumenty, jakie ma przyjmować lambda. (Opcjonalne)
- Atrybuty wyrażenia lambda - np. mutable (lub constexpr, consteval) (sprawia, że zmienne przechwycone przez wartość mogą być modyfikowane wewnątrz ciała wyrażenia). (Opcjonalne)
- -> T - Typ zwracany przez wyrażenie lambda. (Opcjonalne)
- {} - Ciało wyrażenia lambda, analogicznie, jak przy zwykłej funkcji.

```
struct S { void f(int i); };

void S::f(int i) {
    [&, i]{};       // OK
    [&, &i]{};      // ERROR: i preceded by & when & is the default
    [=, this]{};    // ERROR: this when = is the default
    [=, *this]{ }; // OK: captures this by value. See below.
    [i, i]{};       // ERROR: i repeated
}
```

Przechwytywanie:

[=]	Wszystkie zmienne przez wartość, ale *this przez referencję.
[&]	Wszystkie zmienne przez referencję.
[a, &x]	a przez wartość, x przez referencję.
[=, &y]	wszystko przez wartość, ale y przez referencje.

Zapis [z = 0], oznacza inicjalizacja z = 0.
Zmienne z przestrzeni nazw, np. std::cout - nie mogą być przechwytywane

Przykład użycia:

Załóżmy, że mamy obiekt std::vector<int> v = {1, 2, 3, 4, 5};
Możemy posortować dane za pomocą:
std::sort(v.begin(), v.end(), [](int a, int b){ return a < b; });

- Funkcje lambda zachowują się, jak funkcje anonimowe, ALE NIMI NIE SĄ, SĄ FUNKTORAMI (OBIEKTAMI) KLAS AUTOMATYCZNIE GENEROWANYCH PRZEZ KOMPILATOR
  * **funktor (obiekt funkcji) -** to obiekt klasy lub struktury, który można wywołać jak funkcję.
- Można je przypisywać zmiennym(wskaźniki funkcji), przekazywać jako argumenty do i z funkcji
- Wewnątrz funkcji lambda można definiować inne lambdy!

`mutable`

- Lambdy nie mogą modyfikować swojego środowiska
  - Chyba że zmodyfikujemy je słowem `mutable`

```
#include <iostream>

int main()
{
  auto f = [z = 0] () mutable
  {
    return ++z;
  };
  std::cout << f() << f() << f() << "\n";
}
```

```
struct Porownaj
{
  bool operator()(int a, int b) const
  {
    return std::abs(a) < std::abs(b);
  }
};

int main()
{
    std::vector<int> v {1, -2, 3, 8, 0};
    std::sort(v.begin(), v.end(), Porownaj());
}
```

Anonimowy obiekt funkcyjny

```
struct Porownaj
{
  int n;
  Porownaj(int n = 1) : n{n} { }
  bool operator()(int a, int b) const
  {
        return std::pow(a, n) < std::pow(b, n);
  }
};
```

Wersja ułomna lambdy - czyli obiekty funkcyjne

- Mogą być używane, jako funkcja
- Mogą przechwytywać swój stan w składowych obiektu

Wadą tego rozwiązania jest to, że obiekty funkcyjne nie zawsze “dziedziczą” swoje środowiska automatycznie. Możliwa jest ręczna obsługa.

```
struct Porownaj
{
  int n = 0;
  bool operator()(int a, int b) const
  {
    return std::pow(a, n) < std::pow(b, n);
  }
  void set_n(int m) { n = m; }
};
```

Move Semantics(semantyka przenoszenia)

Interfejsy funkcji (C++98)

- Przez wartość - kosztowne **kopiowanie**
- Przez referencję - tylko **do obiektów mających adres** (**l-wartość**), bardzo kłopotliwe przy zwracaniu wartości funkcji.
- Przez stałą referencję - uniwersalne, ale przy przekazaniu obiektu bezadresowego (**r-wartości**) po cichu wykonywana jest kopia, co kosztuje.
- Wskaźnik - jak referencja, ale nagie wskaźniki są ble. tak jak my xD

Przykład interfejsów funkcji (C++98)

Załóżmy, że mamy: **int x = 1;**

void f(int n) f(10); f(x); f(10 + x);
void f(int &n) f(10); f(x); f(10 + x);
void f(const int &n) f(10); f(x); f(10 + x);
* W przypadku f(10) i f(10 + x) wystąpi kopiowanie.

Jak działa stała referencja?

- Jeżeli argument ma adres, to jest on przekazany do funkcji (czyli działa jak zwykła referencja lub wskaźnik).
- Jeżeli argument nie ma adresu, to najpierw w pamięci konstruowana jest kopia jego wartości, a następnie funkcja dostaje adres.

L-wartość, r-wartość i x-wartość

- lvalue - to obiekt posiadający zdefiniowaną nazwę, który można umieścić po lewej stronie operatora przypisania (ale też po prawej) i pobrać jego adres operatorem & (miejsce w RAM);
- glvalue („generalized(uogólniona)” lwartość) to wyrażenie, którego ewaluacja określa tożsamość obiektu lub funkcji;
- rvalue - to obiekt nie posiadający zdefiniowanej nazwy – obiekt tymczasowy, który może być umieszczony tylko i wyłącznie po prawej stronie operatora przypisania (nie można pobrać jego adresu operatorem &). (zmienne tymczasowe), np. przechowywane w rejestrach, cache’u, stałe wkompilowane w kod . Referencję do r-wartości definiuje się za pomocą kombinacji && (podwójny ampersand)
- xvalue - nazwa xvalue od eXpire (wygaśnięcia); wartość typu gl-value: wszystkie wyrażenia, których wynik jest typu rvalue referencja; xvalue - obiekt między adresowalnym i nie adresowalnym (xvalue jest zwracane przez std::move). Ogólna idea z wyrażeniami xvalue jest taka, że obiekt, który reprezentują, zostanie wkrótce zniszczony (stąd część „eXpiring”), a zatem przeniesienie ich zasobów jest w porządku.

Przykład l-wartości i r-wartości

- int x = 9; x jest l-wartością,
- x + 9; r-wartość
- sin(x); wartością funkcji jest r-wartość
- std::cout l-wartość
- sin l-wartość
- std::vector<int> v l-wartość
- 5 r-wartość

Referencja do r-wartości

Przede wszystkim - po co kopiować coś, co zaraz zginie?

- int x = 9;
- int &b = x; l-referencja do obiektu,
- int &&c = 9; r-referencja do stałej,
- int &d = 9; nie da się, 9 nie ma adresu
- int &&e = x; nie da się, x ma adres.

Jak działa referencja do r-wartości?

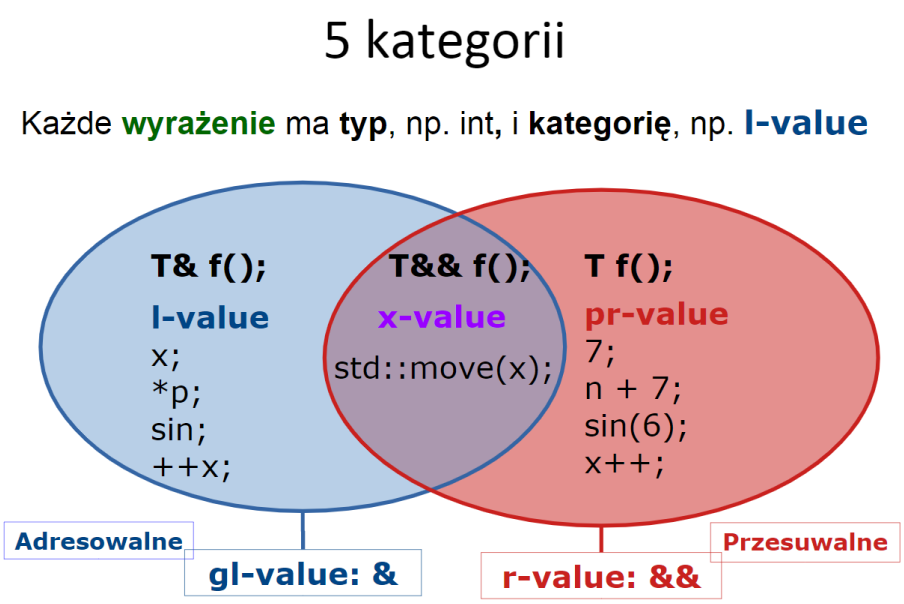
- int &&c = 9; r-referencja do stałej.

- Kompilator generuje kopię wartości (w tym przypadku 9), i umieszcza w **c** jej adres.
- Przypomina to obsługę stałych referencji (**const &**)

Po co to wszystko?

Mając do dyspozycji referencje do l-wartości (&) i r-wartości (&&) można na poziomie języka, czyli automatycznie odróżnić przekazywanie do i z funkcji parametrów mających adres i nie mających adresu, co z kolei **umożliwia automatyczną optymalizację** tych drugich poprzez zastosowanie przeniesienia zamiast głębokiej kopii.

r-value, x-value i l-value



Reguła trzech, reguła pięciu

Reguła trzech - Jeśli klasa wymaga destruktora zdefiniowanego przez użytkownika, konstruktora kopiowania zdefiniowanego przez użytkownika lub operatora przypisania kopii zdefiniowanego przez użytkownika, prawie na pewno wymaga wszystkich trzech. Te specjalne funkcje członkowskie będą wywoływane, jeśli są dostępne. Jeśli nie są zdefiniowane przez użytkownika, są one niejawnie zdefiniowane przez kompilator.

Jeżeli klasa zarządza zasobem, to zdefiniuj w niej:

1. Zdefiniowany destruktor,
2. Konstruktor kopiowania,
3. Kopiujący operator przypisania.

Source: https://en.cppreference.com/w/cpp/language/rule_of_three

```
My_Array first;           // initialization by default constructor
My_Array second(first);   // initialization by copy constructor
My_Array third = first;   // Also initialization by copy constructor
second = third;           // assignment by copy assignment operator
```

Reguła pięciu - każda klasa dla której move semantics jest pożądane, musi zdefiniować pięć rzeczy(destruktor, konstruktor kopiujący, konstruktor przenoszący, operator “=” ,kopiujący, operator ”=” przenoszący), ze względu na obecność zdefiniowania przez użytkownika(lub **=default** lub **=delete)** destruktora, konstruktora kopiującego, albo operatora kopiującego, aby zapobiec niejawnej definicji konstruktora przenoszenia i operatora przenoszenia. (weszło wraz z C++11 ze względu na semantykę move)

```
a::a(const a& rhs) {
    std::cout << "D\n";
    pole = new char[strlen(rhs.pole) + 1];
    memcpy(pole, rhs.pole, strlen(rhs.pole) + 1);
}

a::a(a&& rhs) {
    std::cout << "T\n";
    pole = rhs.pole;
    rhs.pole = nullptr;
}
```

Jeżeli klasa zarządza zasobem, to zdefiniuj w niej:

1.Destruktor,
2. Konstruktor kopiujący,
3. Konstruktor przenoszący,
4. Kopiujący operator przypisania,
5. Przenoszący operator przypisania.

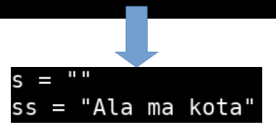
Link =default/delete: What does "default" mean after a class' function declaration? - Stack Overflow

Reguła zera - Jeśli klasa nie zarządza zasobami, to nie powinna mieć zdefiniowanej żadnej z wielkiej piątki. Kompilator zrobi to automatycznie!

std::move

std::move(obj) – umożliwia sprawne **przenoszenie zasobów** z “obj”. Dany “obj” można traktować jako obiekt tymczasowy/bezadresowy (xvalue) w chwili wywołania, natomiast “obj” **po użyciu** zostaje obiektem **nieokreślonym**.

```
int main()
{
   std::string s = "Ala ma kota";
   std::string ss = std::move(s);
   std::cout << "s = \"" << s << "\"\n";
   std::cout << "ss = \"" << ss << "\"\n";
}
```



std::move - **std::move** jest szablonem funkcji, której w nowoczesnym języku C++ używa się bardzo często. Jest sposobem na to, żeby powiedzieć kompilatorowi: **Traktuj argument funkcji std::move jak r-wartość.**

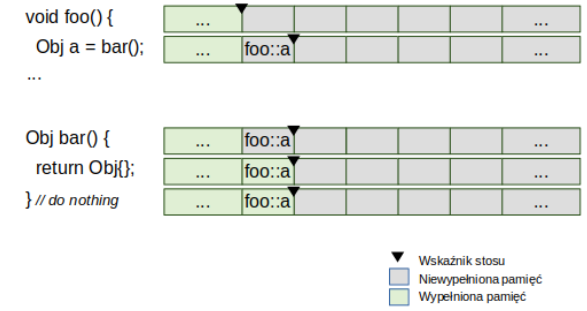
Copy elision, NRVO, RVO

Copy elision - jest to celowe pomijanie wywołania konstruktora kopiującego i/lub przenoszącego przez kompilator w celu optymalizacji.

NRVO ("Named Return Value Optimization[optymalizacja nazwanej wartości zwracanej]") - Jeżeli funkcja zwraca klasę przez wartość i wyrażeniem instrukcji “return” jest **nazwa nieulotnego obiektu** z automatycznym czasem przechowywania (który nie jest parametrem funkcji), wtedy kopiowanie/przenoszenie, które byłyby wykonywane przez (nie-optymalizujący) kompilator, **można pominąć**.

RVO ("Return Value Optimization[optymalizacja wartości zwracanej]") - Jeżeli funkcja zwraca **nienazwany, tymczasowy obiekt**, który mógłby zostać przeniesiony, lub skopiowany przez naiwny kompilator, kopiowanie/przenoszenie **może zostać pominięte**.

```
Obj bar()
{
    Obj odd_obj{1};
    Obj even_obj{2};
    return (std::time(nullptr) % 2 == 1) ? odd_obj : even_obj;
}
```

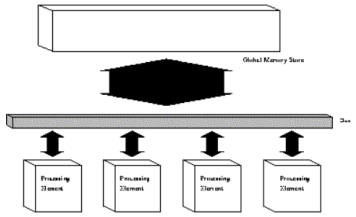


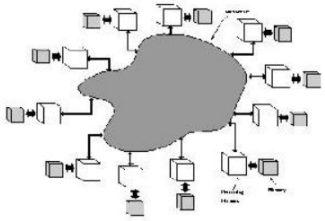
Stackoverflow: What are copy elision and return value optimization? - Stack Overflow
https://cpp-polska.pl/post/zarzadzanie-zasobami-w-c-3-rvo-nrvo-i-obowiazkowe-rvo-w-c17

Dlaczego move semantics jest ważne dla std::vector, a nie np. dla std::complex?

std::vector - to kontener, który przechowuje dane. Wyróżnia się tym, że co jakiś czas, najczęściej (bodajże) 2*n rozszerza się - oznacza to tyle, że trzeba dane skopiować i zaalokować ponownie(wtedy przyda się przenoszenie, bo po co kopiować tyle danych?).
Dokładnie to samo w momencie, kiedy wywołujemy metodę “emplace_back”(po co dodatkowo obciążać nasz program i kopiować dane, które i tak są tymczasowe?), przy korzystaniu z v.emplace() też wykorzystywane jest move semantics
Ładnie wyjaśnione tutaj: std::vector<T,Allocator>::emplace_back - cppreference.com

Programowanie równoległe





Modele pamięci: współdzielonej i rozproszonej

współdzielona – pamięć z której może korzystać wiele programów, służy do umożliwienia komunikacji między nimi lub uniknięcia redundantnych (nadmiarowych) kopii (zwykle pamięć nie nazywa się współdzieloną w kontekście pamięci, z której korzysta wiele wątków jednego programu)
rozproszony – no rozproszony, gdzie wspólna pamięć fizyczna jest emulowana

Które z modeli pamięci współdzielonej i rozproszonej są obsługiwane domyślne w technologiach OpenMP, MPI, std::thread, TBB i pthreads

OpenMP	umożliwiający tworzenie programów komputerowych dla systemów wieloprocesorowych z pamięcią dzieloną.
MPI	Główny model MPI-1 nie wspiera koncepcji współdzielonej pamięci, MPI-2 wspiera (w sposób ograniczony) rozproszony system pamięci dzielonej.
std::thread	współdzielona
TBB	technologia od intela pozwala tylko na współdzieloną
pthreads	współdzielona

Wyścig i zakleszczenie

Wyścig to wykonywanie się tego samego zadania na kilku wątkach i operowanie na tej samej pamięci – sytuacja w której wynik działania kilku procesów lub wątków zależy od kolejności wykonywania się operacji.
Zakleszczenie (deadlock) - gdy jeden z wątków czeka na barierze zarządzanej przez inny wątek

OpenMP (na czym polega zrównoleglenie kodu w tej technologii, kilka zalet, wad i ograniczeń)

- *omp for* lub *omp do* – używane do rozdzielania iteracji pętli pomiędzy wątki, zwane także konstrukcjami pętli.
- *sections* – przypisanie następujących po sobie, ale niezależnych bloków kodu do różnych wątków
- *single* – wyszczególnienie bloku kodu, który będzie przetwarzany przez tylko jeden wątek, z barierą na końcu[2]
- *master* – podobne do single, przy czym kod jest przetwarzany tylko przez wątek główny, bez bariery na końcu[2]

```
int main(int argc, char* argv[])
{
  #pragma omp parallel
  printf("Hello, world!\n");
  return 0;
}
```

Zalety:

- wymaga małej modyfikacji kodu w porównaniu do MPI (prostota)
- dyrektywy mogą po prostu zostać zignorowane jeśli kompilator nie może skorzystać z dyrektywy openmp``

- charakteryzuje się przenośnością, skalowalnością, elastycznością i prostotą użycia

Wady:

- nie można wykorzystywać w przypadku pamięci rozproszonej (wyjątek openmp od Intela)
- wymaga kompilatora, który obsłuży /openmp
- skalowalność jest zależna od przepustowości pamięci
- brakuje niezawodnej obsługi błędów

MPI (na czym polega zrównoleglenie kodu w tej technologii, różnice między OpenMP), jak wygląda komunikacja, jak się kompiluje i uruchamia programy (polecam prezkę Kozy)

MPI łączy się czasem z OpenMP. MPI pozwala na wiele procesów na tej samej lub różnych maszynach. Procesy synchronizują się przez tzw. komunikaty MPI (np. mpi_send, recv). Przykład w C. Trzeba zainstalować bibliotekę, compile: mpic++ your_code_file.c, run: mpirun -np <no. of Processors> ./a.out

Zalety:

- dobra efektywność w systemach wieloprocesorowych
- dobra dokumentacja
- bogata biblioteka funkcji
- status public domain

Wady:

- brak wielowątkowości, następują zakleszczenia (ze względu na pamięć rozproszoną)
- statyczna struktura procesów w trakcie realizacji programu (dotyczy to implementacji opartych na MPI-1). Wersja MPI-2 (wspierana np. przez LAM 7.0.4) umożliwia dynamiczne zarządzanie strukturą procesów biorących udział w obliczeniach – MPI_Spawn()
- statyczna konfiguracja jednostek przetwarzających
- stosunkowo złożone w korzystaniu

```c
int main(int argc, char **argv)
{
    char buf[256];
    int my_rank, num_procs;

    /* Initialize the infrastructure necessary for communication */
    MPI_Init(&argc, &argv);

    /* Identify this process */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    /* Find out how many total processes are active */
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

    /* Until this point, all programs have been doing exactly the same.
       Here, we check the rank to distinguish the roles of the programs */
    if (my_rank == 0) {
        int other_rank;
        printf("We have %i processes.\n", num_procs);

        /* Send messages to all other processes */
        for (other_rank = 1; other_rank < num_procs; other_rank++)
        {
            sprintf(buf, "Hello %i!", other_rank);
            MPI_Send(buf, sizeof(buf), MPI_CHAR, other_rank,
                     0, MPI_COMM_WORLD);
        }

        /* Receive messages from all other process */
        for (other_rank = 1; other_rank < num_procs; other_rank++)
        {
            MPI_Recv(buf, sizeof(buf), MPI_CHAR, other_rank,
                     0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("%s\n", buf);
        }

    } else {
```

Co reprezentują klasy std::thread, std::mutex, std::lock_guard, std::unique_lock

std::thread wątek

std::mutex	(mutual exclusion) tylko jeden wątek może w danej chwili znajdować się wewnątrz sekcji krytycznej kodu zarządzanego mutexem,
std::lock_guard	zamyka mutex w parze konstruktor / destruktor (po wyjściu z bloku w którym został utworzony wywołuje się destruktor),
std::unique_lock	działa jak lock_guard, ale można na nim używać .lock() i .unlock()

promise / future

Służą do przesyłania **jednego obiektu z jednego wątku do drugiego**.
Obiekt `std::promise` jest ustawiany przez wątek, który generuje wynik. Należy połączyć obiekt “promise” z “future”:
` `std::future fut = promiseOBJ.get_future() ;`
Obiekt `std::future` może być użyty do pobrania wartości `fut.get()` , sprawdzenia, czy wartość jest dostępna, lub do zatrzymania wykonania, dopóki wartość nie będzie dostępna.

Zrównoleglenie kodu przy pomocy std::async

std::async - to wrapper dla “std::thread” i “std::promise”. Zwraca “std::future”.

```
std::future ftr = std::async( std::launch::async, f)          // tworzy nowy wątek i natychmiast uruchamia funkcję
std::future ftr2 = std::async( std::launch::deferred, f)      // nie tworzy nowego wątku i uruchamia funkcję przy pierwszym wywołaniu “wait” lub “get” na “future”
ftr.get();
ftr2.get();
```

Do czego służy std::condition_variable?

Zmienna warunkowa(ang. condition variable) - to obiekt, którego można użyć do zablokowania wątku/wątków, dopóki inny wątek tej zmiennej nie odblokuje.

Do zablokowania wątku/watków dzięki “std::condition_variable” potrzebujemy “std::mutex” i np. “std::unique_lock”.

```
bool check = true;
std::mutex m;
std::condition_variable cv;
std::unique_lock locker(m);
cv.wait(lock, [](){return check; });
…….
locker.unlock();
```

execution policy

Używane jako unikalny typ rozróżniania przeciążania algorytmów równoległych i wskazań. Pilnuje, aby wykonanie kodu było równoległe, wszelkie tego typu użycia wykonywane są w tym samym wątku są nieokreślenie sekwencjonowane (używane) względem siebie.

Inteligentne wskaźniki

Podstawowe cechy std::unique_ptr, std::shared_ptr, std::weak_ptr

Wspólną cechą smart pointerów jest to, że pamięć na którą wskazują ma zostać zwolniona w momencie zniszczenia smart pointera (std::unique_ptr<> / std::shared_ptr / std::weak_ptr)
std::shared_ptr – może posiadać wielu właścicieli (wskaźnik nie zostanie usunięty, dopóki wszyscy właściciele nie wyjdą z bloku lub nie usunął wskaźnika u siebie [liczba indeksów spadnie do 0]). **Shared_ptr simplifies reference counting by doing all the increments and decrements automatically based on assignments and scoping rules.**

Do czego służą std::make_unique, std::make_shared

- Służą do unikania new/new[]/delete/delete[],(unikania alokacji pamięci na stercie?)
- Zapobiegają wyciekaniu pamięci,

Krótszy zapis:
`std::unique_ptr < std::string > text( new std::string );` vs: `auto text = std::make_unique < std::string >();`
Zwiększa bezpieczeństwo wątków np.:

```
int Abc() { … }
void saveText( std::unique_ptr < std::string > text, int textDepth ){ ... }

saveText( new std::string( "Ala ma świnkę morską" ), Abc() );
```

Wywołując funkcję oraz wiedząc, że nie jest zdefiniowana kolejność ewaluacji argumentów funkcji może się zdarzyć sytuacja, że wpierw alokujemy pamięć dla obiektu, następnie funkcja Abc() wyrzuci wyjątek – wtedy nastąpi wyciek pamięci. W podobnej sytuacji stosując “std::make_unique”, destruktor obiektu “std::unique_ptr” posprząta zaalokowaną pamięć.

Inteligentne wskaźniki a zasada RAII

Inteligentne wskaźniki działają zgodnie z zasadą RAII, ponieważ w odpowiednim momencie wywołują destruktor i usuwają obiekt…

Elementy C++17

if constexpr

Dzięki C++17 mamy możliwość oceny wyrażeń warunkowych w czasie kompilacji. Kompilator jest wtedy w stanie całkowicie wyeliminować fałszywą gałąź.
Z pewnego punktu widzenia kompilatory robiły to już, jeśli masz instrukcję if z warunkiem, który był stałą czasową kompilacji: kompilatory i optymalizatory były w stanie dowiedzieć się, kiedy nasz kod zawierał gloryfikowany if (prawda) i zoptymalizowałby oddział else.

Jednak ze starym if, druga gałąź wciąż musiała się skompilować. Jeśli constexpr już tak nie jest, kompilator nie będzie już tego próbował. Oczywiście nadal musi to być poprawna składnia C++, ponieważ parser musi przynajmniej dowiedzieć się, gdzie kończy się blok warunkowy.

if constexpr warunkowo usuwa z szablonów gałęzie kodu

constexpr

Podobnie jak const, można go zastosować do zmiennych: błąd kompilatora jest zgłaszany, gdy dowolny kod próbuje zmodyfikować wartość. W przeciwieństwie do “const”, “constexpr” można również zastosować do funkcji i konstruktorów klas. “constexpr” wskazuje, że wartość lub wartość zwracana jest stała i, jeśli to możliwe, jest obliczana w czasie kompilacji.
Wartość całkowita “constexpr” może być używana wszędzie tam, gdzie wymagana jest stała liczba całkowita, na przykład w argumentach szablonu i deklaracjach tablic. A gdy wartość jest obliczana w czasie kompilacji zamiast w czasie wykonywania, pomaga to Twojemu programowi działać szybciej i zużywa mniej pamięci.

constexpr jest tylko wskazówką dla kompilatora – Kompilator nie musi jej respektować (zwłaszcza w trybie Debug) – Kompilator może włączyć optymalizację „constexpr” nawet bez tego słowa kluczowego (zwłaszcza w trybie Release) • constexpr daje gwarancję, że optymalizacja jest możliwa

Szablony wariadyczne

Mogą być używane do definiowania szablonów klas i funkcji. Pozwalają tworzyć struktury danych o nieokreślonej liczbie typów składowych i funkcje o nieokreślonej liczbie i typach argumentów. Mogą podlegać specjalizacji.

Packet expansion np. (2*args…)

- args… - rozwija się do listy argumentów określonych przecinkami,
- expr(args)... - rozwija się do listy expr(e1), expr(e2), …, expr(en)
- … - zawsze występuje po rozwijanym wyrażeniu.

Można zastosować do m.in. listy inicjalizacyjnej i wyrażenia z operatorem przecinkowym.
Rozwiązanie ma wady: typy parametrów muszą być takie same, nie można użyć pustej listy argumentów.

```
template <typename... Args>
void f(Args... args)
{
   auto list = {(args + 10)...};
   for (auto n : list)
     std::cout << n << " ";
   std::cout << "\n";
}

int main()
{
  f(1, 2, 3);
  f(2.5, 3.14);
}
```

A nawet tak:

```
auto list = {10 + args...};
```

```
11 12 13
12.5 13.14
```

fold expansion np. (args + …)

Zwyczajny pack expansion powoduje rozwinięcie argumentu szablonu do listy z elementami oddzielonymi przecinkami. **Fold expression** pozwala uogólnić to rozwinięcie na niemal dowolny operator dwuargumentowy.

Składnia wyrażeń fold:

Niech e=e1,e2,…,en będzie wyrażeniem, które zawiera nierozpakowany parameter pack i ⊗ jest operatorem fold, wówczas wyrażenie fold ma postać:

Unary left fold(jednoargumentowy lewy fałd) : (…⊗e), który jest rozwijany do postaci (((e1⊗e2)…)⊗en)

Unary right fold(jednoargumentowy prawy fałd) : (e⊗…), który jest rozwijany do postaci (e1⊗(…(en−1⊗en)))

Jeśli dodamy argument nie będący paczką parametrów do operatora ...,
dostaniemy dwuargumentową wersję wyrażenia fold.
W zależności od tego po której stronie operatora ... dodamy dodatkowy argument otrzymamy:

Suma kwadratów

```
template<typename... Args >
auto sum_sqr(Args... args)
{
   return ((args * args) + ... + 0);
}
```

Nawiasy są tu konieczne

```
int main()
{
   float t = std::sqrt(3.0f);
   double d = std::sqrt(2.0);
   std::cout << sum_sqr(1, 2u, 4ll) << "\n";
   std::cout << sum_sqr(-1, d, t) << "\n";
}
```

```
21
6
```

Binary left fold: (a⊗…⊗e), który jest rozwijany do postaci (((a⊗e1)…)⊗en)

Binary right fold: (e⊗…⊗a), który jest rozwijany do postaci (e1⊗(…(en⊗a)))

Operatorem ⊗ może być jeden z poniższych operatorów C++:

```
+ - * / % ^ & | ~ = < > << >>
+= -= *= /= %= ^= &= |= <<= >>=
== != <= >= && || , .* ->*
```

Więcej: https://infotraining.bitbucket.io/cpp-17/fold-expressions.html

RAII(”nabywanie zasobu jest inicjalizacją”)

a. zasada RAII [Resource Acquisition Is Initialization]

// wzorzec, więc nie trzeba korzystać, ale jednak dobrze to robić; Zasoby należy zdobywać w konstruktorach i zwalniać je w odpowiadających im destruktorach(automatycznie)!
Zapewnia sprzątanie zasobów;
stack unwinding - mechanizm rozwijania stosu. Jeżeli któraś funkcja wyrzuci wyjątek, to przy powrocie do miejsca, w którym jest on łapany(catch), wywoływane są destruktory wszystkich obiektów „zdejmowanych” ze stosu.

Aby jej zadośćuczynić, w C++ z zasady nie otwiera się pliku wywołaniem funkcji (fopen), lecz konstruuje się obiekt odpowiedniej klasy (np. std::ifstream lub std::ofstream). Analogicznie zamiast tworzyć tablicę operatorem **new**, lepiej jest zdefiniować obiekt klasy “std::vector<T>”.
W miarę możliwości bezpośrednie zdobycie zasobu powinno być ostatnią czynnością konstruktora. Jeśli nie jest, kod następujący po zdobyciu zasobu powinien być zabezpieczony przed zgłoszeniem wyjątku.

* **RAII** jest wzorcem projektowym, którego celem jest powiązanie czasu zajmowania zasobu z czasem życia obiektu – od jego konstrukcji po destrukcję.

b. wyjątek w konstruktorze

// czy to OK? Jeśli wyjątek został zgłoszony podczas konstrukcji obiektu, to obiekt ten nie powstanie, jego destruktor nie zostanie wywołany, a wszystkie do tej pory utworzone składowe zostaną usunięte. Mogą to być już utworzone składowe obiektowe: dla nich destruktory zostaną wywołane. Kłopot, jeśli są w klasie składowe wskaźnikowe, a same obiekty, na które one wskazują, zostały w konstruktorze zaalokowane na stercie lub odnoszą się do zasobów systemowych, jak np. plików. Tego typu obiekty są zwykle usuwane (zwalniane) w destruktorze. W ten sposób, w razie wystąpienia sytuacji wyjątkowej, nieudany obiekt zostanie co prawda usunięty, ale zasoby (pamięć, otwarte pliki) nie zostaną zwolnione. Można temu zaradzić „opakowując” tego rodzaju składowe wskaźnikowe tak, aby uczynić z nich obiekty, dla których w razie niepowodzenia wywołany zostanie destruktor zwalniający zasoby.

c. wyjątek w destruktorze

// czy to OK?
FAKT: Jeśli może nam wyskoczyć wyjątek w destruktorze to musi być również jego obsługa (cała konstrukcja try, catch).

Nie można zgłaszać wyjątku obsługiwanego gdzieś indziej, poza destruktorem, ponieważ destruktor został wywołany przy zwijaniu stosu, przerwane jest jego działanie, przechodzi do obsługi wyjątku, która ponownie wywołuje zwijanie stosu. Mamy w takim układzie podwójne zwijanie stosu!!!
Jeśli w destruktorze może być wyjątek to musi być cała konstrukcja try catch by destruktor po zgłoszeniu wyjątku go wchłonął (catch), by nie doszło do podwójnego zwijania stosu.

```
try { test(); }
   catch( std::runtime_error & e ) { std::cout << "Blad wykonania programu: " << e.what() << '\n';  }
   catch( std::exception& e ) { std::cout << "Blad ogólny programu: " << e.what() << '\n'; }
   catch( ... ) { std::cout << "To co exception nie wyłapał " << '\n'; }
```

Najlepiej nie przechwytywać tego, czego nie można obsłużyć. Przy tworzeniu niestandardowych najlepiej dziedziczyć po exception. Gdy po wyłapaniu wyjątku program nie będzie w stanie zagwarantować poprawnego działania, najlepiej przy obsłużeniu wyjątku go zamknąć.

Przykłady użycia w C++ (np. pliki, muteksy, inteligentne wskaźniki, kontenery C++, etc.)

Sanitizery:

Nowoczesny sposób diagnostyki programów w C++
Potężne narzędzie służące do wychwytywania błędów w programie,
Zwykle wiążą się z odczuwalnym zmniejszeniem prędkości badanych programów, a także ze znacznym spowolnieniem kompilacji
G++ dodaje do kompilowanego programu dodatkowe instrukcje mające wychwycić różnego typu błędy,
Istnieje kilka sanitizerów; nie mogą one działać jednocześnie:

AddressSanitizer (ASan) - to technologia kompilatora i środowiska uruchomieniowego, która uwidacznia wiele trudnych do znalezienia usterek bez wyników fałszywie dodatnich:

- Niezgodność alloc/dealloc i niezgodność typów new/delete
- Zbyt duże alokacje na stercie
- calloc overflow i alloca overflow
- Podwójne free na pamięci oraz użycie po operacji free
- Przepełnienie zmiennej globalnej
- Przepełnienie buforu sterty
- memcpy i nakładanie na siebie parametrów strncat
- Przepełnienie i niedopełnienie buforu stosu
- Użycie stosu po returnie i użycie poza scope'm
- Użycie pamięci po jej zatruciu

Użyj addressSanitizer, aby skrócić czas spędzony na:

- Podstawowa poprawność
- Przenośność międzyplatformowa
- Zabezpieczenia
- Testowanie obciążenia
- Integrowanie nowego kodu

ThreadSanitizer (TSan) - jest detektorem wyścigu wątków. Wyścig występuje gdy kilka wątków odwołuje się do tej samej pamięci równocześnie bez jakiejkolwiek synchronizacji lub zabezpieczenia przed zapisem/odczytem.

UndefinedBehaviorSanitizer (UBSan) - jest szybkim detektorem undefined behavior. UBSan modyfikuje program podczas kompilacji aby wyłapać możliwy undefined behavior w naszym programie, np.:

- index tablicy poza jego granicą/wielkością
- przesunięcia bitowe które są poza granicami dla jego typów
- dereferencja lub null pointers
- signed integer overflow
- konwersja do/z/pomiędzy floating-point typami które mogą spowodować overflow