Język Programowania C++

Notatki

„Jak coś jest przyjemne, to w C++ jest brzydka składnia do tego.”

Maciej Mikołajczak
Kraków, 2024/2025

Spis treści

1. Kompilowanie

2024-10-07

Proces kompilacji: najpierw plik jest konwertowany do *basic source character set* (kilka podstawowych znaków, liczby, litery, znaki przestankowe), nieznane znaki do \u (universal character name), następnie trigrafy są zamieniane na swoje normalne odpowiedniki (tylko do C++17, od C++17 nie są wspierane), np. ??< na {. Następnie usuwane są komentarze (zamieniane na spacje), preprocesor uwzględnia dyrektywy (includy, define'y) i rekurencyjnie wykonuje się przetworzenie dołączonych plików. Ostatecznie wszystkie dyrektywy znikają. Na koniec znaki zostają sprowadzone do *execution character set* (w szczególności występujące po sobie stringi są konkatenowane), jednostki translacyjne są analizowane pod kątem poprawności, następuje generowanie kodu assemblerowego, a potem maszynowego, na koniec linkowanie.

Jednostka translacji – pojedynczy plik .cpp, kompilowany osobno od reszty.

ODR – one definition rule, czyli nie mogą wystąpić różne definicje w różnych jednostkach translacji (wtedy linker nie wie co połączyć). Wyjątkiem są trochę funkcje inline – zawartość funkcji jest wsadzana w środek kodu, nie następuje definicja.

Pliki .h nie podlegają kompilacji, tylko deklarują rzeczy, ich typy etc.

```
//A.h
void foo();

//A.cpp
#include <iostream>
#include "A.h"

void foo() {
    std::cout<<"GOSCIU!";
}

//main.cpp
#include <iostream>
#include "A.h"

int main() {
    foo();
    return 0;
}
```

Podczas kompilowania plik A.cpp jest łączony z A.h i nagłówkami biblioteki iostream, powstaje A.o, analogicznie main.o. To jest kompilacja właściwa. Potem następuje linkowanie, łączymy pliki .o i binarne pliki bibliotek, dostając plik wykonywalny. Po kompilacji mamy wskaźniki na funkcje, ale brakuje ich adresów. Linker wstawia w te wskaźniki odpowiednie wartości. Kompilator sprawdza poprawność typów i inne takie rzeczy, linker tylko łączy.

Rodzaje linkowania (linkage):

- brak łączenia – nazwa może być używana jedynie w zakresie w którym została wprowadzona
- łączenie wewnętrzne – nazwa istnieje tylko w zakresie jednej jednostki translacji, słowo kluczowe static, wtedy zmienna jest lokalna w swojej jednostce translacji, w różnych mogą istnieć różne rzeczy o tej samej nazwie
- łączenie zewnętrzne – słowo kluczowe extern, oznacza, że deklaracja znajduje się w innej jednostce translacji, linker będzie jej szukał na zewnątrz.

Flagi g++ do wykonywania różnych części kompilacji:

- -E: wynik preprocesora, wynik jest przygotowany do kompilacji
- -s: kompilacja do assemblera, dalej plik tekstowy, ale z instrukcjami preprocesora

- -c: kompilacja do kodu maszynowego, czytanie za pomocą objdump z flagą -t, występują jeszcze symbole undefined (*UND*) – będą dostępne dopiero po zlinkowaniu

2. Podstawy

2024-10-14

Identyfikator to dowolnie długi ciąg znaków (Unicode). Używa liter, cyfr, znaków podkreślenia, nie może zaczynać się od cyfry. W identyfikatorach duże i małe litery są rozróżniane. Niektóre nazwy są zarezerwowane (słowa kluczowe i nazwy predefiniowane, takie jak `__TIME__`, ogólnie nazwy zaczynające się od dwóch podkreślników lub podkreślnika i dużej litery dają undefined behaviour – to mogą być np. makra, takie nazwy są zarezerwowane dla biblioteki standardowej).

Literały to ciągi znaków oznaczające wartości, np. `10`, `"Hello"`, `true`.

Nazwa to identyfikator, który nie jest słowem kluczowym (z wyjątkiem `this` – to jest nazwą), nazwa funkcji operatorowej, np. `operator+`, nazwa klasy poprzedzona `~` (nazwa destruktora), konkretyzacja szablonu.

Nazwa kwalifikowana to nazwa poprzedzona operatorem zakresu `::` lub ciągiem nazw rozdzielonych operatorem zakresu, np. `std::vector` (określa namespace).

Wyrażenie to konstrukcja syntaktyczna zbudowana z identyfikatorów oraz literałów powiązanych operatorami.

Typ jest cechą obiektów, referencji, funkcji oraz wyrażeń, ustalaną w czasie kompilacji (typ statyczny – typy dynamiczne są związane z polimorfizmem). Typ określa zbiór wartości dla danych, operacje dozwolone na tych danych oraz reprezentację w pamięci.

Od C++11 istnieje typ wskaźnika pustego: typ `std::nullptr_t`, który przyjmuje jedynie wartość `nullptr`. Przydaje się on np. gdy mamy kilka przeciążonych funkcji przyjmujących wskaźnik – przekazanie `NULL` (lub 0) da błąd, bo wszystkie typy będą pasować. Można zrobić osobny overload przyjmujący `std::nullptr_t` i obsłużyć w nim pusty wskaźnik.

Dla każdego typu, który nie jest referencyjny i funkcyjny, istnieją wersje cv-kwalifikowane: takie z modyfikatorami `const` (ograniczenie modyfikacji) i/lub `volatile` (obszar pamięci, który może w każdej chwili ulec zmianie, ważne np. przy operacjach wielowątkowych) oraz `mutable` (pola możliwe do zmiany w constowym obiekcie klasy).

Typ statyczny obiektu może się różnić od dynamicznego przy polimorfizmie – kompilator traktuje zmienną `*A` jak wskaźnik na `A`, mimo że w runtimie może tam być zmienna o typie dziedziczącym po `A`.

Obiekt to obszar pamięci o określonym rozmiarze (`sizeof` – daje wielkość w czasie kompilacji), rodzaju alokacji, czasie życia, typie, wartości i opcjonalnie nazwie. Obiektami nie są funkcje, referencje, typy, namespace'y, enumeratory (wartości, które może przyjmować jakiś enum) ani szablony. Większość pozostałych rzeczy to obiekty. Obiekty mogą zawierać podobiekty.

Rodzaje alokacji pamięci (storage duration):

- automatic (automatyczny) – zmienna jest alokowana i dealokowana na początku i końcu bloku, w którym występuje. Są to wszystkie nieglobalne zmienne bez modyfikatorów.
- static (statyczny) – zmienna powstaje na początku programu i jest dealokowana na końcu, istnieje tylko jedna w całym programie. Są to zmienne globalne i te z modyfikatorami `static` i `extern`.
- thread (lokalny dla wątku) – zmienna powstaje i kończy się razem z wątkiem, w każdym wątku istnieje osobna instancja tej zmiennej. Są to zmienne o modyfikatorze `thread_local` (mogą mieć dodatkowo `static` lub `extern` by dostosować linkowanie).
- dynamic (dynamiczny) – alokacja i dealokacja dynamicznie, czyli wtedy, gdy programista to zrobi.

Deklaracja wprowadza nazwę do jednostki translacyjnej, a definicja dodatkowo uściśla w pełni tę nazwę, w szczególności daje jej wartość. Zwykle deklaracja jest definicją, ale nie, gdy np. deklarujemy funkcję bez ciała, nazwę klasy, deklarujemy zmienną z `extern` i jej nie inicjalizujemy, deklarujemy ze `static` pole w definicji klasy, deklarujemy parametr funkcji w jej deklaracji bez ciała, deklarujemy alias (za pomocą `using` lub `typedef`).

Każda deklaracja składa się ze specyfikatorów, a następnie rozdzielonej przecinkami listy deklaratorów (opcjonalnie z inicjalizacją), np. `const int *a, b = 5`, gdzie `*a` i `b` to deklaratory. Specyfikatory to np. `typename`, `friend`, specyfikatory typu i typu alokacji, cv-kwalifikatory, `typedef`.

Zakres widoczności deklaracji danej nazwy to obszar kodu, w którym ta nazwa może być użyta w sposób niekwalifikowany oraz odnosi się do zadeklarowanego bytu. Zazwyczaj zaczyna się w punkcie deklaracji – wyjątkiem są np. deklaracje składowych klas (dostępne gdziekolwiek w definicji klasy). Punkt deklaracji danej nazwy zaczyna się na końcu deklaratora (przed inicjalizatorem). Ilustruje to kod:

```
int x = 12;
{ int x = x; } // x ma nieokreśloną wartość (!)
{ int x = ::x; } // x ma wartość 12
```

Wyjątkiem są enumeratory, których punkt deklaracji jest po identyfikatorze:

```
const int x = 12;
{
    enum {
        x = x + 1, // punkt deklaracji za przecinkiem
        y = x + 1 // nowy x jest już w zakresie
    };
}
```

Nazwy mogą się przesłaniać. Nazwa może zostać przykryta przez deklarację tej samej nazwy w zagnieżdżonym obszarze lub w klasie pochodnej. Nazwa klasy lub wyliczenia może zostać przykryta przez nazwę zmiennej, pola, funkcji lub enumeratora zadeklarowanego w tym samym obszarze deklaracyjnym. Nazwa wprowadzona za pomocą dyrektywy using może być przykryta przez deklarację tej samej nazwy w danej przestrzeni nazw.

3. Wyszukiwanie nazw

2024-10-21

Wyszukiwanie nazwy (name lookup) to procedura, w której nazwa napotkana kodzie jest łączona z deklaracją (lub wieloma), która wprowadziła tą nazwę. Dla funkcji wyszukiwanie nazwy może zwrócić kilka deklaracji (do tego występuję proces ADL – argument dependent lookup, który daje inne wyniki niż standardowe), które następnie są przekazywane do procedury znajdowania przeciążenia (overload resolution), która wybiera tę najlepszą. Dla nazw niefunkcyjnych wyszukiwanie nazwy musi być jednoznaczne. Co ważne, kontrola dostępu (jeśli występuje) jest sprawdzana na końcu.

Nazwa niekwalifikowana to nazwa, przed którą nie ma operatora zakresu. Dla nazwy niekwalifikowanej, procedura przeszukuje zakresy w określonym porządku dopóki nie znajdzie przynajmniej jednej deklaracji danej nazwy.

W pewnych kontekstach wyszukiwanie nazwy pomija pewne deklaracje – wyszukiwanie nazwy po lewej stronie operatora `::` pomija funkcje, zmienne i deklaracje enumeratorów, nazwa na lewo od wszystkich operatorów `::` musi być deklaracją typową.

ADL – jak funkcja ma argumenty i one są w jakimś namespace'ie, to funkcja też jest szukana od tego namespace'a.

4. Typy

2024-10-28

Typy dzielimy na podstawowe (arytmetyczne, `bool`, `nulptr_t` i inne) oraz złożone, czyli pochodzące od typów podstawowych.

Typy złożone to typy referencyjne (l- i r-wartości), wskaźnikowe, tablicowe, funkcyjne, wyliczeniowe i klasowe (`class`, `struct`, `union`), do tego typy niefunkcyjne i niereferencyjne mają wersje cv-kwalifikowane.

Typy obiektowe to wszystkie typy niefunkcyjne, niereferencyjne i inne niż `void`, czyli te, których zmienne są obiektami.

Typy skalarne to typy arytmetyczne, wskaźnikowe, wyliczeniowe. To te, na których można robić obliczenia.

Typy trywialne to te literałowe, oraz POD (plain old types) kompatybilne z typami z C.

Gdy do wykonania danej operacji potrzebny jest rozmiar lub układ w pamięci typu, wymagana jest jego pełna definicja. Dzieje się to np. gdy definiujmy lub wywołujemy funkcję z parametrem tego typu, tworzymy obiekt za pomocą `new` lub tablicę, chcemy się dostać do składowej obiektu danego typu.

W innych sytuacjach (np. stworzenie wskaźnika) wystarczy typ niekompletny – typ `void`, niezdefiniowany (ale zadeklarowany) typ klasowy, typ tablicowy bez rozmiaru, typ tablicy elementów niekompletnego typu i podobne.

Kategoria wartości wyrażenia jest charakteryzowana przez dwie własności:

- czy ma tożsamość, czyli czy oznacza konkretny byt i można to sprawdzić (np. sprawdzając adres)
- czy można z niego przenieść zasoby (czy chcemy, aby operatory mogły je zniszczyć podczas swojego działania)

Daje nam to kombinacje:

- ma tożsamość i nie należy przenosić zasobów: l-wartość
- ma tożsamość i można przenieść zasoby: x-wartość
- nie ma tożsamości i można przenieść zasoby: pr-wartość

Do tego definiujemy pojęcia:

- gl-wartość to l-wartość lub x-wartość
- r-wartosć to pr-wartość lub x-wartość

Operator przypisania wymaga, aby wartość była gl-wartością, dla innych się nie skompiluje. Nie każdej gl-wartości można przypisać wartość – istnieje modyfikator `const` (w szczególności działają tak literały stringów).

Technika RAII (resource allocation is initialisation) – przejęcie zasobu jest połączone z konstrukcją, a zwolnienie z destrukcją. W ten sposób nie ma problemu w przypadku rzucenia wyjątku, bo destruktor i tak się wykona i zasób zostanie zwolniony. Tak działa `lock_guad` w C++.

W szczególności l-wartościami są wyrażenia typu: `std::cout << "hello"` (wywołanie funkcji zwracającej referencję na l-wartość), `a = b, a += b` dla wbudowanych operatorów przypisania, preinkrementacja i predekrementacja, wzięcie wartości wskaźnika (lub tablicy, jeśli ona nie jest r-wartością, czyli nie można przenieść z niej zasobów), ternary operator na l-wartościach i literały napisowe.

Od l-wartości można brać adres operatorem `&` i można ich użyć przy tworzeniu referencji l-wartości.

Nazwa pr-wartość to skrót od pure right value, są nią np. literały inne niż napisowe, wywołania funkcji zwracających przez wartość, wywołania operatorów typu `a+b, a==b` oraz `-a`, tak samo operator postin(de)krementacji, operator adresu, konwersje na typy niereferencyjne.

Pr-wartość nie może być polimorficzna (jej typ dynamiczny jest identyczny ze statycznym), jeśli ma typ nieklasowy nie może być cv-kwalifikowana, nie może mieć niekompletnego typu (poza `void`).

Nazwa x-wartość jest skrótem od expiring value, są nią np. wywołania funkcji zwracających przez referencję r-wartości (np. `std::move`), niestatyczne i niereferencyjne pola obiektów będących r-wartością, elementy tablic będących r-wartością, konwersje na referencje r-wartości (np. `static_cast<char&&>`).

Gl-wartość ma tożsamość, może być przekonwertowana niejawnie na pr-wartość (niejawne rzutowanie l-wartości na r-wartość, tablicy na wskaźnik, funkcji na wskaźnik), może być polimorficzna i może mieć niekompletny typ (o ile kontekst na to pozwala).

R-wartość nie może być użyta po lewej stronie przypisania, nie można brać z niej adresu, może być użyta do inicjalizacji referencji stałych l-wartości i referencji r-wartości (życie obiektu zostaje przedłużone o zakres tej referencji), jeśli funkcja ma dwa przeciążenia, jedno z referencją stałej l-wartości, drugie z referencją r-wartości, to r-wartość wiąże się z referencją r-wartości.

Referencje mogą powstawać:

- referencje l-wartości: z l-wartości
- stałe referencje l-wartości: ze stałych l-wartości lub r-wartości
- referencje r-wartości: z r-wartości