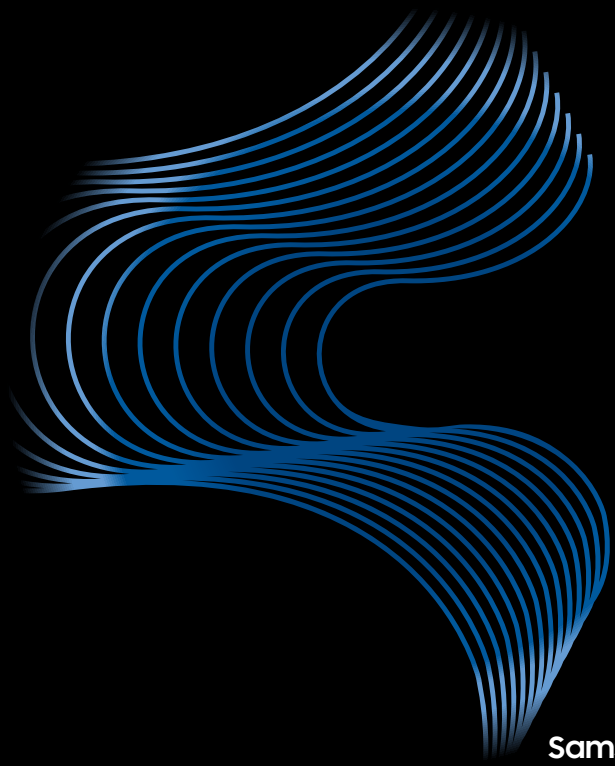
Samsung R&D Institute Poland

C++ Tech Guide

Poszerzaj i utrwalaj znajomość zagadnień technicznych, korzystając z praktycznych wskazówek naszych ekspertów.



Samsung Tech Guides

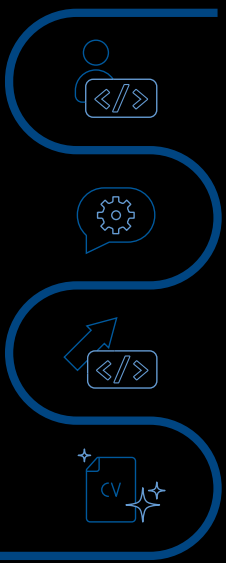
Drogi kandydacie!
Droga kandydatko!

Zapraszamy Cię do zapoznania się z C++ Technical Guide - technicznym przewodnikiem poświęconym jednemu z najbardziej popularnych języków programowania!

Znajdziesz w nim tematy, które warto powtórzyć przed spotkaniem technicznym w rekrutacji do Samsung R&D!

Dodatkowo nasi Inżynierowie przygotowali przykładowe zadania testowe, które pomogą Ci sprawdzić swoją wiedzę.

C++ Technical Guide - Dla kogo?



dla kandydatów aplikujących na stanowiska, w których wymagana jest znajomość C++,

dla kandydatów zaproszonych już na spotkanie techniczne do Samsung R&D,

dla początkujących Inżynierów chcących poszerzyć/utrwalić swoją wiedzę z C++,

dla osób zainteresowanych w przyszłości rekrutacją do Samsung R&D.

Powodzenia!

C++ - podstawy

C++ to język programowania ogólnego przeznaczenia. Nie sposób opisać wszystkich elementów języka w tak krótkim opracowaniu. Zakładamy, że znasz podstawy C++11. Jeżeli jesteś zaznajomiony z nowościami wprowadzanymi przez kolejne wersje standardu - tym lepiej. Ten dokument ma wskazać Ci najważniejsze elementy, które przydadzą się w codziennej pracy.

Typy danych, konwersja typów, specyfikator typu auto, modyfikatory const i volatile

Upewnij się, że znasz podstawowe typy danych w C++ i ich właściwości:

- Czym się różnią? Jakiego rodzaju dane reprezentują? Z jakiego zakresu?
- Ile bajtów pamięci zajmują? Czy jest tak niezależnie od architektury komputera?
- Jaką wartość mają zmienne po zdefiniowaniu?

W przypadku typów całkowitoliczbowych rozważ używanie biblioteki `stdint.h`, która daje programiście większą kontrolę nad wykorzystywanymi typami.

Język C++ pozwala na niejawną konwersję typów. To może być źródłem wielu kłopotów, których można uniknąć stosując konwersję jawną. Można używać rzutowania w stylu języka C, ale C++ 11 pozwala na nawet lepszą kontrolę konwersji poprzez stosowanie operacji `static_cast`, `dynamic_cast`, `reinterpret_cast` i `const_cast`.
Naucz się posługiwać nimi.

C++ 11 wprowadził też specyfikator typu `auto`. Upewnij się, że wiesz, jak działa i kiedy warto (albo w ogóle można) go stosować.

Przyda się też sprawne posługiwanie modyfikatorem `const`, a przy niskopoziomowej pracy ze sprzętem również modyfikatorem `volatile`. Sprawdź, co powoduje ich użycie.

Wskaźniki, tablice, alokowanie pamięci, inteligentne wskaźniki

Wskaźniki to specjalne typy danych - zmienne wskaźnikowe przechowują adres pamięci obiektu. Biegłe posługiwanie się wskaźnikami to podstawa.

- Czy umiesz wyłuskać wartość zmiennej, na którą wskazuje wskaźnik (operator `*`)?
- Czy umiesz pobrać adres zmiennej, żeby móc przypisać go do wskaźnika (operator `&`)?
- Czy umiesz posługiwać się wskaźnikami do wskaźników (`T**`)?

Wskaźniki potrzebne są między innymi do dynamicznego alokowania pamięci przy pomocy operatorów `new` i `new[]`. Można nimi zarezerwować pamięć odpowiednio na jedną zmienną lub na tablicę. Rozważ, jakie wartości będą przyjmować zmienne dynamiczne zaraz po utworzeniu. Pamiętaj, że zaalokowaną dynamicznie pamięć zawsze należy zwolnić. Służą do tego operatory `delete` i `delete[]`.

Tablice przechowują elementy danego typu tak, że możliwy jest sekwencyjny dostęp do tych elementów przy pomocy operatora indeksowego `[]` lub poprzez notację wskaźnikową. Pamiętaj, że nazwa tablicy jest wskaźnikiem do pierwszego elementu. To często się przydaje.

- Czy umiesz wyznaczyć rozmiar tablicy korzystając z operatora `sizeof`?

„Problem" dynamicznej alokacji i zarządzania pamięcią rozwiązują inteligentne wskaźniki: `std::unique_ptr` i `std::shared_ptr`, które po spełnieniu pewnych warunków (np. dla `std::unique_ptr` - wyjście poza zakres) zwalniają zarządzaną pamięć.

Zmienne automatyczne, dynamiczne, statyczne, globalne

Zmienne mają określony czas życia i miejsce w pamięci. Z tego względu możemy je podzielić na zmienne automatyczne, dynamiczne, statyczne i globalne. Upewnij się, że rozumiesz te pojęcia.

- Kiedy zmienne każdej z tych kategorii przestają istnieć?
- Gdzie są przechowywane?

Ze zmiennymi automatycznymi wiąże się pojęcie stosu (ang. stack), a dynamicznymi - sterty (ang. heap). Wynika to z najczęściej spotykanej implementacji sposobu przechowywania zmiennych programu. Sprawdź, czym różni się stos od sterty i jaka jest ich charakterystyka.

Referencje

Referencje to aliasy (alternatywne nazwy) istniejących zmiennych. Wykonując operacje na referencji do zmiennej, tak naprawdę modyfikujemy tę zmienną.

- Jak zainicjalizować referencję?
- Czy po inicjalizacji można przypisać referencję do innej zmiennej?
- Jakie są różnice i podobieństwa między referencjami i wskaźnikami?

Przekazy--wanie przez wartość, referencję, wskaźnik

W C++ zmienne można przekazywać do funkcji na kilka sposobów: przez wartość, przez referencję i przez wskaźnik. Przekazanie przez wartość oznacza, że funkcja będzie używać kopii zmiennej. Nie zmienimy w ten sposób wartości zmiennej będącej argumentem wywołania funkcji, ponadto samo przekazanie zmiennej przez wartość może być kosztowne w przypadku dużych obiektów. Przekazywanie przez referencję i wskaźnik rozwiązuje ten problem. W codziennej pracy musisz biegle posługiwać się tymi mechanizmami.

Klasy

C++ pozwala na definiowanie własnych, rozbudowanych typów, które nazywane są klasami. Instancje klas to obiekty. Klasy składają się z pól (zmiennych) i metod (funkcji) pozwalających na modyfikowanie stanu obiektów.

Klasy pozwalają na stosowanie abstrakcji, czyli techniki polegającej na upraszczaniu i przedstawieniu jakiegoś problemu lub rzeczywistości tylko przy pomocy istotnych cech. Pozwala to na tworzenie programów w bardziej intuicyjny sposób, bardziej odpornych na błędy, prostszych do utrzymania. Poniżej przedstawiamy podstawy klas w C++, które musisz znać.

Konstruktory, konstruktory kopiujące, konstruktory przenoszące

Klasy mogą zawierać specjalne metody, które służą do tworzenia instancji klas, czyli obiektów.

- Ile może być konstruktorów?
- Czym mogą (albo muszą) się różnić?
- Jak tworzyć obiekty przy pomocy konstruktorów?

Istnieją dwa specjalne typy konstruktorów: kopiujące i przenoszące (te drugie zostały wprowadzone w C++11). Jeżeli nie zostaną jawnie zadeklarowane, to kompilator sam zadeklaruje konstruktor kopiujący, a przenoszący w przypadku spełnienia kilku warunków. Jeżeli klasa zawiera pola dynamiczne, to należałoby zadeklarować i zdefiniować własne konstruktory kopiujące i przenoszące.

W obu przypadkach warto zastanowić się:

- Jak działają domyślne konstruktory wygenerowane przez kompilator?
- Kiedy takie działanie będzie błędne?
- Jak je wywołać/kiedy będą wywoływane?

W przypadku konstruktorów przenoszących warto przy okazji zapoznać się z pojęciami wartości lewo i prawostronnych (ang. lvalue i rvale), czyli typami wyrażeń.

To pewne uproszczenie, bo w C++ typów wyrazeń jest więcej, ale tyle wystarczy do zrozumienia idei przenoszenia.

Destruktory

Skoro obiekty można tworzyć, to powinno też dać się je zniszczyć. Służy do tego destruktor, czyli specjalna metoda wołana jawnie lub niejawnie, gdy istnienie obiektu dobiega końca. Jeżeli obiekty w trakcie swojego istnienia zarządzają jakimiś zasobami (np. otwierają pliki albo alokują pamięć), to destruktor służy do zwalniania tych zasobów.

- Kiedy destruktor wywoływany jest niejawnie?
- Kiedy i jak należy jawnie wywołać destruktor?
- Czy tablicę obiektów niszczy się tak samo, jak pojedyncze obiekty?

Składowe statyczne

Składowe statyczne, czyli poprzedzone słowem kluczowym static, należą do klasy, ale nie są częścią obiektów tej klasy. Jest tylko jeden egzemplarz takiej składowej, obiekty klasy nie posiadają swoich egzemplarzy. Brzmi zawile, ale sprowadza się do tego, że nie trzeba nawet tworzyć obiektu klasy, żeby posługiwać się polami i metodami statycznymi. W ten sposób można np. zrealizować licznik obiektów danej klasy i go odczytywać. Zwróć uwagę, że słowo kluczowe static ma też inne znaczenia w zależności od kontekstu, np. w przypadku zmiennych statycznych w funkcjach.

Składowe publiczne i prywatne

Programista może decydować, co ma być interfejsem klasy (czyli ma być dostępne z zewnątrz), a co wewnętrzną implementacją, do której dostęp mają tylko metody klasy i funkcje zaprzyjaźnione. Służą do tego specyfikatory dostępu `public`, `private` i `protected`. Ten ostatni ma znaczenie przy dziedziczeniu, o którym mowa później. Poprzez zezwalanie/blokowanie dostępu do pól i metod możemy realizować ideę hermetyzacji (kapsułkowania), jednej z podstaw programowania obiektowego. Dzięki temu zwiększamy bezpieczeństwo kodu i zmniejszamy ryzyko popełniania błędów przez programistów wykorzystujących nasze klasy. Może lepiej nie dawać bezpośredniego dostępu do pola, którego bezmyślna modyfikacja może skończyć się katastrofą?

Zamiast tego możemy zabronić dostępu do tego pola, ale za to udostępnić metodę, która będzie pilnować wartości przekazywanych przez użytkownika. Jeżeli konieczne jest stworzenie funkcji poza klasą, która powinna mieć dostęp do prywatnych składowych klasy, to możemy ją zaprzyjaźnić z klasą. Dobrym przykładem są operatory strumieniowe, o których dalej.

- Jeżeli nie podamy jawnie żadnego modyfikatora dostępu, to jaki jest domyślny dostęp do pól i metod klasy?
- Jak zaprzyjaźnić funkcję z klasą?

Przeciążanie operatorów

C++ umożliwia przeciążanie operatorów, dzięki czemu możemy posługiwać się obiektami w wyrażeniach tak jak typami wbudowanymi. Można się obyć bez tego, wykorzystując metody klasy. Niektórzy nazywają to składniowym lukrem (ang. syntax sugar). Skoro jednak jest i można się z nim spotkać w istniejącym kodzie, wypada umieć się nim posługiwać. Szczególnie istotny jest operator przypisania, który, podobnie jak konstruktor kopiujący, potrzebny jest dla klas posiadających pola dynamiczne.

Często przydaje się też umiejętność tworzenia operatorów strumieniowych, które są specyficzne - ich pierwszym argumentem jest strumień, a nie definiowana klasa. Kodu strumienia nie zmodyfikujemy, a więc nie utworzymy dla strumienia operatora strumieniowego przyjmującego jako drugi argument obiekt naszej klasy. To musi być zewnętrzna funkcja, dla wygody zaprzyjaźniona z naszą klasą. Nieintuicyjny może być też operator funkcyjny, który pozwala tworzyć obiekty funkcyjne, używane w algorymach biblioteki standardowej. Nietypową składnię ma definicja operatora inkrementacji przyrostkowej, z kolei operator indeksowy nie musi nawet przyjmować jako argument typu całkowitoliczbowego.

- Które operatory można przeciążać (a może prościej: których nie można)?
- Jak deklaruje się i definiuje operatory?
- Operator przypisania - kiedy jest potrzebny? Kiedy jest wywoływany?
- Które operatory tworzone są automatycznie przez kompilator, jeżeli ich nie zdefiniujemy?

Dziedziczenie

W praktyce często okazuje się, że wiele klas w programie ma wspólne cechy. Typowym przykładem są figury geometryczne, które chcielibyśmy rysować na ekranie. Elipsy i prostokąty będą mieć składowe takie jak współrzędne położenia na ekranie i metody do rysowania albo wyznaczania pola powierzchni. W związku z tym, zamiast tworzyć w każdej klasie identyczne pola i metody, które czasami robią trochę co innego, można byłoby zdefiniować bardziej ogólną klasę posiadającą wspólne składowe. Do tego właśnie służy dziedziczenie, które pozwala na tworzenie hierarchii klas. To kolejna ważna cecha języka C ++, która jest podstawą programowania obiektowego. Tworzymy klasy bazowe, a następnie klasy pochodne, które dziedziczą po bazowych.

Zrozumienie dziedziczenia jest istotne w codziennej pracy. To dosyć złożony temat. Co wypada wiedzieć:

- Jak dziedziczyć po klasie?
- Które składowe są dziedziczone? Czym różni się specyfikator dostępu `private` od `protected` w klasie bazowej?
- W jakiej kolejności wywoływane są konstruktory i destruktory w klasach pochodnych?
- Jak w klasie pochodnej przedefiniować działanie metody z klasy bazowej (słowo kluczowe `virtual`)?
- Co to jest wielodziedziczenie? Co, jeżeli po jednej klasie dziedziczą dwie inne, a po nich wielodziedziczy jeszcze inna (problem diamentu i dziedziczenie wirtualne)?
- Jak wywoływać przesłonięte metody z klasy bazowej?
- Co to jest klasa abstrakcyjna? Jak ją zdefiniować?

Polimorfizm dynamiczny

Jednym z powodów stosowania dziedziczenia jest potrzeba stworzenia wspólnego interfejsu dla różnych klas. Chcemy posługiwać się w jednolity sposób klasami, które są do siebie podobne. Tworzymy więc bazową klasę abstrakcyjną, która dostarcza interfejs, a następnie tworzymy klasy, które dziedziczą po niej. Teraz możemy posługiwać się klasami pochodnymi przy pomocy wskaźników lub referencji typu bazowego.

Wywołując metody wirtualne poprzez taki wskaźnik/referencję zadziała mechanizm, który zdecyduje, jakiego typu jest tak naprawdę wskazywany obiekt i wywoła metodę tej konkretnej klasy. Nazywamy to polimorfizmem dynamicznym lub czasu działania. W ten sposób możemy na przykład tworzyć tablice wskaźników typu bazowego, a do tych wskaźników przypisywać adresy obiektów typów pochodnych. Opanuj ten mechanizm.

Napisy i łańcuchy znaków

Język C, z którego wywodzi się C++, nie ma specjalnego typu reprezentującego napisy. Zamiast tego, posługujemy się tablicami znaków zakończonych znakiem `'\0'`. Wszystkie funkcje przetwarzające napisy opierają się na tej konwencji. Biblioteka standardowa języka C++ udostępnia klasę `std::string`, która znacznie upraszcza używanie napisów. Można patrzeć na tę klasę jak na kontener znaków udostępniający wiele pomocniczych metod. Upewnij się, że:

- Umiesz posługiwać się łańcuchami znaków w stylu C (np. umiesz policzyć, ile znaków ma napis).
- Znasz metody klasy `std::string`.
- Umiesz utworzyć obiekt typu std::string z tablicy znaków i na odwrót.

Strumienie wejścia i wyjścia, operacje na plikach

W programowaniu częstym zadaniem jest przetwarzanie plików. W języku C++ służą do tego klasy nazywane strumieniami. Strumienie plikowe znajdują się w nagłówku `fstream`. Kilka istotnych kwestii, na które warto zwrócić uwagę:

- Otwieranie plików w różnych trybach, zamykanie plików.
- Czym różni się plik tekstowy od pliku binarnego?
- Czytanie pliku - znak po znaku, całymi liniami, przy pomocy operatorów strumieniowych.
- Przesuwanie wskaźnika pozycji w pliku.

Szablony

Szablony to wygodny sposób tworzenia ogólnych (generycznych) klas i funkcji, które można parametryzować. Wyobraźmy sobie, że chcemy stworzyć klasę przechowującą elementy danego typu. Nazwijmy ją `Kontener`. W języku C musielibyśmy zadeklarować, jakiego typu elementy będziemy przechowywać. Jeżeli chcielibyśmy teraz stworzyć identyczną klasę przechowującą elementy innego typu, to musielibyśmy skopiować kod i podmienić wystąpienia jednego typu na drugi. To bardzo mozolna i podatna na błędy praca. Dzięki szablonom w C++ możemy napisać taką klasę raz, a kompilator wygeneruje odpowiednie wersje w zależności od parametrów użytych w kodzie (np. `Kontener<int> x, Kontener<float> y` itd.).

- Jak tworzyć szablony?
- Jak tworzyć zmienne typu szablonowego z konkretnym parametrem?
- Czy szablon można parametryzować tylko jednym argumentem, czy wieloma?

Wyjątki

Każdy dobrze napisany program musi obsługiwać błędy. Przypuśćmy, że przetwarzamy pliki. Co, jeżeli wskazany plik nie istnieje na dysku lub z jakiegoś powodu nie można go otworzyć? Nie możemy w ciemno zakładać, że operacja otwarcia pliku do czytania/pisania powiodła się i kontynuować przetwarzanie, musimy sprawdzić, czy nie wystąpił żaden błąd. W języku C prowadzi to na ogół do kodu z mnóstwem instrukcji warunkowych. C++ wprowadził mechanizm wyjątków, który pozwala na bardziej elegancką obsługę błędów.

Wystarczy objąć „podejrzany" kod blokiem `try-catch` i jeżeli kod wewnątrz **`try`** zgłosi wyjątek (potocznie mówimy, że rzuci wyjątek, ponieważ wyjątki zgłasza się przy pomocy słowa kluczowego `throw`), wykonanie reszty kodu w `try` zostanie przerwane i zostaniemy przekierowani do `catch`. Tam możemy obsłużyć wyjątek i zdecydować, czy chcemy sytuację naprawić, czy lepiej jednak zakończyć działanie programu. Sekcji `catch` może być wiele — na każdy typ wyjątku, którego się spodziewamy.

- Jak zgłosić wyjątek?
- Jak obsłużyć wyjątek?
- Jak obsłużyć dowolny wyjątek, jeżeli nie mamy pewności, jakiego typu oczekiwać?
- Co, jeżeli blok kodu może zgłaszać wyjątki będące klasami z pewnej hierarchii? Czy kolejność bloków `catch` ma jakieś znaczenie?

Kontenery, iteratory, algorytmy

Biblioteka standardowa języka C++ oferuje szeroki wybór kontenerów, czyli klas przechowujących elementy oraz udostępnia narzędzia do wygodnej pracy z nimi. W język wbudowane są tablice, ale posługiwanie się nimi potrafi być niewygodne (aczkolwiek często konieczne). Tablica ma określony rozmiar i trzeba uważać, żeby nie odwoływać się do elementów spoza zakresu.

W przypadku tablic dynamicznych można realokować pamięć, żeby powiększyć tablicę. Kontenery ukrywają te wszystkie operacje i udostępniają wygodny interfejs do operacji na elementach (hermetyzacja). Są klasami szablonowymi, a więc mogą przechowywać dowolne elementy. Podstawowym kontenerem jest `std::vector`, który pozwala między innymi na dodawanie elementów na koniec, usuwanie elementów z końca czy wyłuskiwanie dowolnych elementów. Można też sprawdzić przy pomocy metody `size()`, ile elementów już jest w wektorze.

Są jeszcze inne kontenery sekwencyjne:

- `std::array` (tablica o stałym rozmiarze, ale wzbogacona o metody),
- `std::deque` (kolejka dwukierunkowa),
- `std::forward_list` (lista jednokierunkowa),
- `std::list` (lista dwukierunkowa).

Kontenery, iteratory, algorytmy

Zapoznaj się z nimi, poznaj różnice, zwróć szczególnie uwagę na złożoność obliczeniową operacji takich jak dodawanie, pobieranie i usuwanie elementów. Oprócz powyższych dostępne są też adaptacje kontenerów, które udostępniają inny interfejs dla powyższych klas:

- `std::stack` (stos, czyli bufor typu LIFO),
- `std::queue` (kolejka, czyli bufor typu FIFO),
- `std::priorty_queue` (kolejka priorytetowa).

Dostępne są też uporządkowane kontenery asocjacyjne, których przeszukiwanie ma złożoność logarytmiczną:

- `std::set` (kolekcja unikalnych, posortowanych kluczy),
- `std::map` (kolekcja par klucz-wartość posortowanych po unikalnych kluczach),
- `std::multiset` (kolekcja posortowanych kluczy),
- `std::multimap` (kolekcja par klucz-wartość posortowanych po kluczach),

oraz nieuporządkowane kontenery asocjacyjne (dodaj `unordered_` przed nazwy z powyższej listy), które w najlepszym przypadku mogą mieć stałą złożoność obliczeniową wyszukiwania elementów. W zależności od zadania przydatny może być inny rodzaj kontenera. To może mieć kluczowy wpływ na wydajność rozwiązania.

Kontenery, iteratory, algorytmy

Kontenery udostępniają metody i operatory pozwalające na dostęp i modyfikację elementów, przy czym ze względu na specyfikę każdego kontenera, interfejs tych operacji może być kompletnie różny. Dla wygody oraz aby ujednolicić posługiwanie się kontenerami, wprowadzono iteratory. Są to klasy zaprojektowane tak, aby posługiwanie się nimi przypominało arytmetykę wskaźnikową na tablicach. W zależności od kontenera możliwe jest uzyskanie iteratorów jednokierunkowych, dwukierunkowych lub o dostępie swobodnym.

Naucz się:

- Co oznaczają wyżej wymienione typy iteratorów.
- Uzyskiwać iteratory do kontenerów.
- Co oznacza iterator za koniec kontenera (wyłuskiwany przy pomocy metody `end()`)?
- Przechodzić po kontenerze przy pomocy iteratorów.
- Odczytywać/modyfikować wartości elementów przy pomocy iteratorów.

Iteratory używane są w wielu funkcjach z biblioteki standardowej, w szczególności tych z nagłówka `algorithm`. Znajdziemy tam mnóstwo algorytmów służących do wyszukiwania, sortowania, zliczania i manipulowania elementami kontenerów. Przejrzyj dokumentację tej biblioteki i zapoznaj się z tymi funkcjami.

Wątki

Niektóre programy wymagają stosowania współbieżności - wykonywania zadań równolegle. Jeżeli dysponujemy urządzeniem z wieloma procesorami lub procesorem z wieloma rdzeniami, możliwe jest rzeczywiste wykonywanie zadań równolegle; w przeciwnym wypadku zadania wykonywane są naprzemiennie - system operacyjny przełącza je co jakiś czas tak, aby sprawiać wrażenie równoległości.

Zagadnienie współbieżności jest dosyć skomplikowane i wymaga dobrego zrozumienia działania komputera i systemu operacyjnego. Trzeba mieć chociażby świadomość, że wątki wykorzystują wspólną przestrzeń adresową, a więc mogą konkurować ze sobą o dostęp do danych i należy je wtedy synchronizować. Biblioteka standardowa C++11 zawiera wygodne narzędzia do współbieżności. Wątki uruchamiane są przez utworzenie obiektów typu `std::thread` przekazując jako argument konstruktora zadanie w postaci funkcji lub obiektu funkcyjnego.

Upewnij się, że opanowałeś poniższe tematy:

- Tworzenie wątków, przekazywanie argumentów, zwracanie wyników przez argumenty (np. referencje), czekanie na zakończenie wątku.
- Wspólne używanie danych/synchronizacja wątków przy pomocy muteksów (`std::mutex`, `funkcja std::lock()`, `std::unique_lock`).
- Czekanie na zdarzenia zewnętrzne (`std::condition_variable`).
- Komunikacja między zadaniami przy pomocy `std::future` i `std::promise`, `std::packaged_task` oraz `std::async()`.

Wyrażenie lambda

Wyrażenia lambda pozwalają na definiowanie i używanie anonimowych obiektów funkcyjnych, co jest szczególnie przydatne podczas wywoływania algorytmów. Zamiast tworzyć funkcję lub obiekt funkcyjny, możemy w miejscu wywołania algorytmu napisać wyrażenie lambda, które ma w uproszczeniu taką postać:

```
[]()->typ{}
```

Gdzie kolejne elementy oznaczają:

- `[]` - lista zmiennych lokalnych (może być pusta), które mogą być używane w treści właściwej wyrażenia lambda,
- `()` - lista argumentów, opcjonalna,
- `>` - typ - typ zwracany, opcjonalny,
- `{}` - treść właściwa wyrażenia.

Naucz się posługiwać wyrażeniami lambda.

Przestrzenie nazw

Być może Twoją uwagę zwróciło, że w tym poradniku nazwy klas i funkcji z biblioteki standardowej poprzedzone są nazwą `std` i dwoma dwukropkami. Jest to kwalifikator przestrzeni nazw mówiący, skąd pochodzą klasy i funkcje. Przestrzeń nazw to blok zaczynający się od słowa kluczowego `namespace`, po którym podaje się nazwę przestrzeni. Wszystkie elementy wewnątrz tego bloku należą do tej przestrzeni. Jest to technika pozwalająca unikać konfliktu nazw, co jest istotne w dużych programach składających się z wielu bibliotek. Jeżeli jakaś nazwa znajduje się w przestrzeni nazw, to można ją wyłuskać tak, jak opisano wyżej. Można też posłużyć się dyrektywą `using`, aby wprowadzić nazwy ze wskazanej przestrzeni do przestrzeni globalnej. Pewnie nie raz Twój program zaczynał się od `using namespace std`.
Naucz się posługiwać przestrzeniami nazw.

- Jak wyłuskać z przestrzeni konkretne elementy?
- Co, jeżeli zdefiniujemy kilka przestrzeni o tej samej nazwie?
- Czy można zagnieżdżać przestrzenie nazw?
- Co, jeżeli w dwóch przestrzeniach nazw jest klasa lub funkcja o tej samej nazwie, a obie przestrzenie dodaliśmy przy pomocy dyrektywy `using`?

Testowanie kodu

Dobrym zwyczajem jest testowanie kodu. Jeżeli stworzymy nietrywialny program, który będzie dalej rozwijany, to lepiej poświęcić trochę czasu na napisanie testów, które będą uruchamiane przy każdym budowaniu. Manualne sprawdzenie, czy program działa jest średnim pomysłem i zajmuje dużo czasu. Najprostszym sposobem jest zastosowanie makra `assert` z nagłówka `cassert`.

Makro sprawdza, czy jego argument ma wartość zero. Jeżeli tak, wypisuje na standardowe wyjście błędów informacje i kończy działanie programu. To bardzo proste narzędzie. W praktyce lepiej stosować bardziej zaawansowane biblioteki do testowania.

Styl pisania kodu

Wbrew pozorom styl pisania kodu jest ważny. Ułatwi pracę Tobie, ale przede wszystkim uprości życie wszystkim, którzy będą pracować z Twoim kodem. Upewnij się, że stosujesz jednolity, spójny styl pisania kodu: stosujesz wcięcia, otwierasz i zamykasz bloki kodu w ten sam sposób, stosujesz spację (lub nie) po instrukcjach sterujących, a przed nawiasem.
Przyjmij jeden sposób nazywania zmiennych, funkcji i klas tak, aby od razu było widać, co jest czym.

Narzędzia: g++, gdb, cmake, git

W codziennej pracy potrzebne jest sprawne posługiwanie się narzędziami. na ogół do pracy używana jest jedna z dystrybucji GNU/LInux, zatem dobrze jest znać:

- Podstawowe komendy systemu GNU/LInux
- Kompilator *g++*
- Debugger *gdb*

Projekty, które mają więcej niż jeden plik (czyli praktycznie wszystkie) ciężko kompilować ręcznie. Dlatego dobrze jest znać jakieś narzędzie do automatycznego budowania programów:

- *Makefile*
- *CMake*
- *Meson*

Nie obędzie się też bez systemu kontroli wersji - tu na pewno przyda się znajomość *gita*.

1.
Napisz następujący program:

- W funkcji main zadeklaruj 10-elementową tablicę liczb całkowitych.
- Napisz funkcję, do której przekażesz tę tablicę i do każdego jej elementu dodasz 1. Wypisz tablicę przed i po wywołaniu funkcji.

2.
Zadeklaruj 64-bitową zmienną całkowitoliczbową i zdefiniuj jej dowolną wartość:

- Wypisz na ekran wszystkie bity tej liczby.
- Zmień 41. bit na przeciwny.

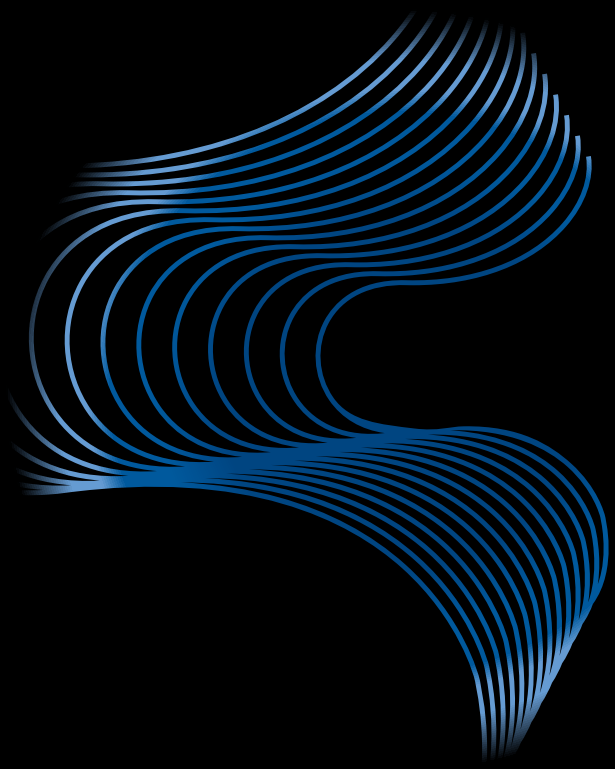
3.
Napisz własną klasę string:

- Nie używaj elementów z biblioteki standardowej.
- Klasa powinna mieć możliwość utworzenia obiektu na podstawie przekazanej stałej znakowej.
- Klasa powinna mieć metodę zwracającą długość napisu.
- Klasa powinna mieć operator indeksowy zwracający/ zmieniający podaną literę.
- Klasa powinna mieć metodę/ operator pozwalający na dodanie kolejnych znaków do napisu.
- Powinny zostać zdefiniowane operatory strumieniowe, które pozwolą wypisywać obiekty na ekran/ do pliku.

4.
Zadeklaruj klasę abstrakcyjną:

- Napisz dwie klasy pochodne, które będą przeciążać metodę wirtualną.
- Zaprezentuj polimorficzne wywoływanie tej metody.

Samsung R&D Institute Poland



Samsung Tech Guides