Wprowadzenie do Sekcji Algorytmicznej, zawodów w programowaniu i WZAiPów

Paweł Tarasiuk, Instytut Informatyki PŁ
Sekcja Algorytmiczna Koła Informatycznego Niepokoju „KINo”

Łódź 2016

Spis treści

1 Wstęp **4**

1.1 Motywacja do powstania tego skryptu 4
1.2 Inne materiały 6
1.2.1 Tutoriale 6
1.2.2 Podręczniki języka programowania 6
1.2.3 Podręczna dokumentacja języka 7
1.2.4 Podstawy teoretyczne 7
1.2.5 Techniki programowania 8
1.2.6 Zbiory zadań 8
1.2.7 Tło kulturowe 8
1.3 Weryfikacja osiągnięć 9
1.3.1 Ogólnodostępne systemy typu online-judge 10
1.3.2 Zawody w programowaniu 10

2 Jak zacząć **11**

2.1 Czy i dlaczego `C++`? 11
2.1.1 Wymagania co do języka programowania 11
2.1.2 Najczęściej dostępne języki 11
2.1.3 Java 12
2.1.4 Języki skryptowe 13
2.1.5 Wersje `C++` 13
2.2 Podstawowe narzędzia 14
2.2.1 Kompilator 14
2.3 Wskazówki dla skazanych na Microsoft Windows 18
2.4 Narzędzia pomocnicze 18
2.4.1 Lokalne testowanie programu 18
2.4.2 `gdb` 18
2.4.3 `Valgrind` 18
2.4.4 `Code::Blocks` 18

3 Typy danych **19**
3.1 Typy proste . . . 19
3.2 Wskaźniki . . . 19
3.3 Tablice . . . 19
3.4 Aliasy typów . . . 19
3.5 Struktury lub klasy . . . 19
3.6 Szablony . . . 19
3.7 Rzutowanie typów . . . 19
3.8 Podsumowanie operatorów . . . 19

4 Podstawowe elementy języka **20**
4.1 Wejście/wyjście . . . 20
4.2 Instrukcje warunkowe . . . 20
4.3 Pętle . . . 20
4.4 Funkcje . . . 20
4.5 Rekurencja . . . 20

5 Wybrane elementy biblioteki standardowej `C++` **21**
5.1 Kontenery i iteratory . . . 21
5.2 Przydatne funkcje . . . 21
5.3 Przykłady . . . 21

6 Zaawansowane elementy `C++` **22**
6.1 Dziedziczenie . . . 22
6.2 Przeciążanie operatorów . . . 22
6.3 Spostrzeżenia na temat implementacji wejścia/wyjścia różnych typów 22

Rozdział 1

Wstęp

1.1 Motywacja do powstania tego skryptu

Inspiracją do powstania tego skryptu były doświadczenia związane ze zdolnymi studentami, którzy potrzebowali w krótkim czasie uzyskać pewne podstawowe umiejętności programowania w języku `C++`. Studenci ci najczęściej chcieli jak najszybciej zacząć brać aktywny udział w spotkaniach Sekcji Algorytmicznej Koła Informatycznego Niepokoju „KINo", w zajęciach przedmiotów z serii „Wybrane Zagadnienia Algorytmiki i Programowania" (WZAiP) lub w zawodach w programowaniu zespołowym. Na szczególne wyróżnienie jako inspiracja do nauki technik programowania o których traktuje ten skrypt zasługują coroczne Mistrzostwa Wielkopolski w Programowaniu Zespołowym – które są stacjonarne, otwarte i przeprowadzane są na tak dużą skalę, że udział w nich może wziąć praktycznie każdy zainteresowany takim doświadczeniem.

Styl programowania, o jakim traktuje ten skrypt jest związany z tradycjami zawodów w stylu ACM-ICPC (ang. *Association for Computing Machinery – International Collegiate Programming Contest*). Dotyczy to wspomnianych Mistrzostw Wielkopolski w Programowaniu Zespołowym oraz Algorytmicznych Mistrzostw Polski w Programowaniu Zespołowym – które z kolei są najbardziej prestiżowym wydarzeniem tego typu w Polsce i stanowią lokalne eliminacje do mistrzostw świata ACM-ICPC. Zadania z takich mistrzostw stanowią pewne abstrakcyjne problemy, które należy rozwiązać poprzez dostarczenie kodu źródłowego programu, który dla dowolnych danych wyznacza prawidłową odpowiedź, przy zachowaniu ograniczeń czasowych oraz pamięciowych. Z pewnych praktycznych (ale także tradycyjnych) przyczyn preferowany w tym zastosowaniu jest język `C++` (2.1). W istocie wykorzystywany jest jednak tylko pewien podzbiór języka – kody źródłowe rozwiązań mają charakter jednorazowy, unikalny dla danego rozwiązania i tradycyjnie zapisywane są w całości w pojedynczym pliku (2.1.1). Takie wymagania mogą się wydać eg-

zotyczne nawet studentom informatyki, którzy zaliczyli przedmioty przecinające się zakresem materiału z niniejszym skryptem („Algorytmy i Struktury Danych", „Podstawy Programowania" i „Programowanie Obiektowe") – dlatego zawarty tu materiał zapewne będzie zawierał wiele nowych informacji także dla większości z nich.

Działalność dydaktyczna Sekcji Algorytmicznej często zawierała w sobie wyjaśnianie pewnych podstaw podczas samych spotkań lub w ramach merytorycznych paneli dyskusyjnych. Proponowane dotychczas źródła (1.2) nie były jednak dostosowane do potrzeb uczestników idealnie – rozwijało to w nich bardzo cenną zdolność do samodzielnego wyszukiwania źródeł i oceny ich przydatności, ale niestety także spowalniało sam proces kształtowania podstawowych umiejętności programistycznych. Studenci Sekcji Algorytmicznej wymieniają się często zestawem napisanych bardzo kolokwialnym językiem „czytanek" proponowanych nowym uczestnikom. Niniejszy skrypt ma docelowo mieć jednak nad dawnymi materiałami pewne zalety – celem jest uzupełnienie luk i sporządzenie materiału bardziej reprezentatywnego (w przeciwieństwie do czytanek, które można określić jako hermetyczne i nie mające racji bytu poza kontekstem relacji towarzyskich Sekcji Algorytmicznej). Niniejszy skrypt ma zatem być przydatny do celów promocji WZAiPów i Sekcji kierowanej do nowych, nie wdrożonych jeszcze w naszą barwną społeczność studentów.

W przypadku Wybranych Zagadnień Algorytmiki i Programowania problem tradycyjnie nie był aż tak istotny, gdyż do udziału w przedmiocie zapraszani byli tylko studenci otrzymujący odpowiednio dobry wynik podczas obowiązkowego testu kompetencyjnego (wyjątki od tej reguły wymagały znacznego zaangażowania i wkładu pracy związanego z programowaniem) – zapewniało to, że studenci biorący udział w przedmiocie znaczną część podstaw opanowali w ramach pracy samodzielnej. Drobne braki w wiedzy bez problemu uzupełniane były podczas zajęć lub indywidualnych konsultacji. Często pozwalało to studentom, którzy uzyskali w ramach testu kompetencyjnego wynik na granicy przyjęcia na przedmiot dorównać reszcie grupy i korzystać z uczestnictwa w przedmiocie w pełni.

Dodatkowe materiały często były potrzebne studentom matematyki, którzy chcieli brać udział w wymienionych aktywnościach o charakterze programistycznym. Znaczna część materiału WZAiPów (algebra, kombinatoryka i probabilistyka, geometria obliczeniowa, teoria grafów, optymalizacja) jest łatwiejsza do zrozumienia dla studentów kierunku matematyka – często sprawia to, że po pewnym czasie osiągnięcia studentów matematyki przewyższają średnią grupy z danego roku. Opanowanie kwestii technicznych często sprawia jednak studentom matematyki dodatkową trudność i wymaga od nich dodatkowego nakładu pracy w początkowych tygodniach lub miesiącach WZAiPów.

1.2 Inne materiały

1.2.1 Tutoriale

Próby sporządzenia zwięzłego wprowadzenia do języka `C++` można spotkać w wielu miejscach, ale ich jakość pozostawia wiele do życzenia. Bardzo ambitną, lecz wiecznie nieukończoną próbą spisania takiego wprowadzenia jest *Wikibooks. C++ Programming. 2012. URL:* https://en.wikibooks.org/wiki/C++_Programming [23]. Wprowadzeniem jeszcze bardziej zwięzłym, a jednocześnie kompletnym jest © *cplusplus.com. C++ Language Tutorial. 2000–2016. URL:* http://www.cplusplus.com/doc/tutorial/ [20]. Źródła te mają charakter ogólny i nieznacznie wykraczają poza styl ACM-ICPC, ale są godne polecenia ze względu na czytelną strukturę oraz niewielką długość.

1.2.2 Podręczniki języka programowania

Wskazanie obszernych źródeł książkowych do nauki języka programowania jest zadaniem jeszcze trudniejszym. Jedną z możliwości jest rozpoczęcie nauki programowania nie od języka `C++`, lecz od klasycznego `C` – jest to dobra droga do uzyskania szerszego spojrzenia na temat oraz pod wieloma względami optymalna kolejność poznawania tych języków. W tym celu można śmiało korzystać z kanonu literatury jakim jest *Brian W. Kernighan.* The C Programming Language. *Prentice Hall Professional Technical Reference, 2nd edition, 1988* [11].

W przypadku samego języka `C++` analogicznym „kanonem" literatury jest *Bjarne Stroustrup.* The C++ Programming Language. *Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000* [18]. Pozycję tę w istocie polecam raczej teoretykom języka, niż osobom zainteresowanym zaledwie umiejętnością posługiwania się pewnym jego podzbiorem. Z całą pewnością pozycja ta nie jest kierowana do czytelników, którzy wcześniej nie potrafili programować w żadnym języku imperatywnym.

Pozycją dalece bardziej przystępną jest *Stephen Prata.* C++ Primer Plus (Fifth Edition). *5th edition, 2004* [15]. Jest to bardzo dobry kompromis między pedantyczną starannością a przystępnym sposobem, w jaki opisany został cały język `C++`.

Przykładem książki szczególnie wartościowej dla studentów Politechniki ze względu na charakter zawartych w niej przykładów, a jednocześnie – starannie wprowadzającej we wszystkie elementy języka `C++` jest *Larry Nyhoff.* Programming in C++ for Engineering and Science. *CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2012* [14].

1.2.3 Podręczna dokumentacja języka

Referencje języka to materiały mające na celu nie tyle uczenie się nieznanych elementów języka, co szybki dostęp do formalnego opisu znanych elementów (nie oczekujemy wszakże, aby programista zawsze wszystko umiał na pamięć – programista powinien być twórczy, a brak zdolności odtwórczych nie musi być wielką przeszkodą).

Znaczną formalną elegancją cechuje się referencja *cppreference.com. C++ reference. 2014–2016. URL: http://en.cppreference.com/w/* [4]. Wiąże się to jednak z pewnymi praktycznymi wadami – referencja ta jest napisana bardzo technicznym, suchym językiem i o ile ułatwia przypominanie sobie zasady działania dowolnego elementu języka, to praktycznie uniemożliwia zrozumienie jej od podstaw. Można przypuszczać, że taka forma jest niewygodna dla programistów nie mających jeszcze wielkiego doświadczenia.

Dokumentacją sugerowaną dla początkujących programistów jest zatem © *cplusplus.com. Standard C++ Library reference. 2000–2016. URL: http://www.cplusplus.com/reference/* [21]. Zapisana tam referencja języka `C++` opatrzona jest zrozumiałym komentarzem i czytelnymi przykładami, które skutecznie pobudzają wyobraźnię na temat tego, jak użyć każdego z omawianych elementów języka.

1.2.4 Podstawy teoretyczne

W istocie przydatne podstawy teoretyczne mogłyby objąć lwią część programów studiów na kierunkach matematyka i informatyka. Ta część skryptu będzie zatem traktowała tylko o pozycjach najważniejszych, bądź wyróżniających się z innego powodu.

Temat potrzebnych podstaw matematycznych można w znacznej części sprowadzić do zagadnień opisanych w *Ronald L. Graham, Donald E. Knuth, and Oren Patashnik.* Concrete Mathematics: A Foundation for Computer Science. *Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1994* [8].

Rozmowy na temat teorii grafów w języku polskim warto prowadzić posługując się terminologią ze skryptu *Tomasz Filipczak.* Grafy i ich zastosowania. *2012* [7]. Tłumacze najbardziej znanych pozycji z dziedziny teorii grafów nie wypracowali języka, który byłby do końca konsekwentny – zaproponowany skrypt, poza wartościowym meritum, zawiera bardzo rozsądne propozycje ujednolicenia polskich tłumaczeń wielu pojęć.

Wreszcie, jeśli chodzi o samą algorytmikę, warto przynajmniej raz w życiu przeczytać *Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson.* Introduction to Algorithms. *McGraw-Hill Higher Education, 2nd edition, 2001* [1] i wracać do tej pozycji przy każdej potrzebie. Obszerność tej pozycji

nie umniejsza temu, że jest ona obowiązkowa.

Przykładem bardziej przystępnej, pomocniczej pozycji traktującej o podstawach algorytmiki jest *Robert Sedgewick and Christopher J. Van Wyk.* Algorithms in C++: Fundamentals, Data Structures, Sorting, Searching. *Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1998* [16].

1.2.5 Techniki programowania

Pozycją traktującą o tym, jak programować dobrze jest przede wszystkim *Donald E. Knuth.* The Art of Computer Programming Volumes 1-3 Boxed Set. *Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1998* [12] wraz z wciąć powstającymi dalszymi tomami.

Książką dotyczącą programowania na zawodach typu ACM-ICPC jest *Piotr Stańczyk.* Algorytmika praktyczna. *Wydawnictwo Naukowe PWN, Warszawa, 2009* [17]. Pewne sugestie na temat intensywnego zastosowania preprocesora zapisane w tej książce polecam jednak traktować jako ciekawostkę – przepisywanie znacznej części języka w celu zaoszczędzenia kilku wpisywanych znaków, które jednocześnie sprawia, że kod staje się niezrozumiały dla każdego poza znającym konkretne makra autorem zdaje się być nieoptymalne.

1.2.6 Zbiory zadań

Klasycznym źródłem zbiorów zadań z omówieniami są związane z Olimpiadami Informatycznymi „niebieskie książeczki", czyli *Maciej M. Sysło, Krzysztof Diks, Przemysława Kanarek, and Jakub Radoszewski.* {I. . . XXII} Olimpiada Informatyczna . *Komitet Główny Olimpiady Informatycznej, Warszawa, 1993–2016* [19].

Pozycjami – o ile to możliwe – jeszcze bardziej interesującymi są polskie zbiory zadań, które w znacznej części traktują o zadaniach z zawodów akademickich, czyli *Krzysztof Diks.* W poszukiwaniu wyzwań: wybór zadań z konkursów programistycznych Uniwersytetu Warszawskiego. *Wydział Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego, 2012* [5] i *Krzysztof Diks, Tomasz Idziaszek, Jakub Łącki, Jakub Radoszewski, and Szymon Acedański.* W poszukiwaniu wyzwań 2: zadania z Akademickich Mistrzostw Polski w Programowaniu Zespołowym 2011-2014. *Wydział Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego, 2015* [6].

1.2.7 Tło kulturowe

Znajomość materiałów takich jak *J.D.* UserFriendly. *1997–2016. URL:* http://www.userfriendly.org/ [10], *Zach Weinersmith.* Saturday Morning Breakfast Cereal. *2002–2016. URL:* http://www.smbc-comics.com/ [22],

Abstruse Goose Corporation. Abstruse Goose. *2008–2016. URL:* http://abstrusegoose.com/ [2] i *Randall Munroe.* xkcd – A webcomic of romance, sarcasm, math, and language. *2005–2016. URL:* http://xkcd.com/ [13] to absolutna podstawa i nie wymaga dalszego komentarza.

W jednym ze źródeł wspomnianych powyżej można znaleźć odpowiedź na pytanie, jak po upływie krótkiego czasu od rozpoczęcia nauki poznać język `C++` [3] – zaproponowane rozwiązanie jest znacznie skuteczniejsze niż jakakolwiek propozycja bibliograficzna, nie wyłączając niniejszego skryptu.

Ciekawym połączeniem meritum i humoru jest *Roedy Green.* How To Write Unmaintainable Code. *2000. URL:* https://www.thc.org/root/phun/unmaintain.html [9] – pozycja, która poza zapewnieniem dożywotniej pensji na stanowisku programisty w istocie porusza wiele merytorycznych aspektów pisania czytelnego kodu oraz składni języków programowania obiektowego, w tym `C++`.

1.3 Weryfikacja osiągnięć

Poznawanie tajników algorytmiki i programowania można rozpatrywać jako sztukę dla sztuki. Jednak efekty nauki są bardziej mierzalne, jeśli znajdują one odzwierciedlenie w rozwiązywanych problemach. Rozwiązywanie zadań w ramach systemów typu online-judge jest ponadto źródłem pewnej satysfakcji, gdyż w większości z nich dostępne są publiczne rankingi, w ramach których można obserwować swoje postępy na tle pozostałych uczestników serwisu. Źródłem znacznie większej satysfakcji jest jednak rozwiązywanie zadań w ramach zawodów w programowaniu. Ta część skryptu traktuje o tym, gdzie można takie zadania napotkać.

1.3.1 Ogólnodostępne systemy typu online-judge

SPOJ.com

Polski SPOJ

MAIN

Adjule

Codeforces

CodeChef

UVa Online Judge

HackerRank

CodinGame

CodeFights

1.3.2 Zawody w programowaniu

AMPPZ

MWPZ

PTwPZ

Deadline24

Marathon24

Potyczki Algorytmiczne

Google Code Jam

Facebook Hacker Cup

Rundy na Codeforces

CodeChef: wyzwania i Snackdown

Konkursy na CodinGame

Maratony na CodeFights

Rozdział 2

Jak zacząć

2.1 Czy i dlaczego C++?

2.1.1 Wymagania co do języka programowania

Zadania z algorytmiki naśladujące swoim stylem mistrzostwa świata w programowaniu ACM-ICPC polegają najczęściej na tym, aby napisać program, który wczyta pewne dane testowe ze standardowego wejścia, wypisze odpowiadającą im odpowiedź na standardowe wyjście, a ponadto zrobi to w odpowiednio krótkim czasie i zmieści się w ograniczeniach związanych z pamięcią. Dodatkowe założenie dotyczy tego, że kod programu należy zawrzeć w jednym pliku oraz że program nie powinien korzystać z żadnych bibliotek spoza samego standardu języka. Czas zawodów jest zwykle ograniczony, więc warto, aby jak najwięcej rzeczy dało się napisać szybko. W tej części skryptu zajmiemy się intuicyjną analizą tego, jakie języki programowania mogą się do tego nadawać. Dodatkowym ograniczeniem pozostaje dostępność poszczególnych języków na zawodach w programowaniu, związana między innymi z ich popularnością.

2.1.2 Najczęściej dostępne języki

Tradycyjnie na zawodach programistycznych dostępne były języki `Pascal`, `C` oraz `C++`. Są to języki kompilowane, co oznacza, że kod programu przekształcany jest do postaci interpretowanej bezpośrednio przez procesor, co oznacza brak dodatkowych narzutów i potencjalnie pełne wykorzystanie możliwości komputera do szybkich obliczeń. Ponadto języki te są niezależne od platformy – dla każdego z nich istnieje wiele kompilatorów, które pozwalają na uruchamiania programów napisanych w tych językach na różnych urządzeniach i systemach operacyjnych.

Język `C++` stopniowo zdobył jednak popularność tak zdecydowaną, że wybór `C` lub `Pascala` bywa postrzegany jako ezoteryczny i omawiany przez organizatorów

jako ciekawostka (zdarza się to na Mistrzostwach Wielkopolski w Programowaniu Zespołowym). Przyczyn tego faktu najłatwiej doszukać się w tym, że biblioteka standardowa języka `C++` zawiera wiele gotowych elementów, pozwalających zapisać rozwiązanie w zwięzły, abstrakcyjny sposób, z wykorzystaniem pewnych klasycznych i uniwersalnych algorytmów i struktur danych. Dotyczy to przede wszystkim struktur danych związanych z szablonami (ta część standardu języka bywa określana jako STL – ang. *Standard Template Library*) oraz funkcji z nagłówka `algorithm`. Nie bez znaczenia pozostają jednak dostępne w `C++` mechanizmy programowania obiektowego. Pozwala to przygotować sobie gotowe rozwiązania specyficznych problemów w taki sposób, aby cechowały się one zwięzłą implementacją, czytelnym API i łatwością zastosowania. Jakkolwiek języki z dłuższą historią, jakimi są `Pascal` i C pozwalają napisać wszystko to co jest obecne w `C++` tak, aby działało równie dobrze, to w przypadku wielu zadań algorytmicznych byłoby to istotnie bardziej pracochłonne.

2.1.3 Java

Na niektórych zawodach (dotyczy to na przykład Potyczek Algorytmicznych) dopuszczalnym językiem programowania jest także `Java`. Wybór tego języka wiąże się z dostępem do bardzo szerokiej biblioteki języka, gdzie faktycznie dostępny od ręki jest bardzo bogaty wybór gotowych implementacji algorytmów i struktur danych. Korzystanie z tych elementów jest o tyle proste, że API jest bardzo dobrze przemyślane i udokumentowane. Optymalne posługiwanie się językiem o tak obszernej specyfikacji wymaga niestety dużego nakładu pracy związanego z poznawaniem języka. `Java` nie jest jednak językiem stosownym do zadań w stylu ACM-ICPC. Nawet samo pisanie całego programu w jednym pliku nie jest zgodne z ogólnie przyjętymi technikami programowania w języku `Java`. Ponadto, nawet jeśli tworzenie kodu miałoby się sprowadzić do umiejętnego połączenia gotowych elementów w całość, to trudno jednak powiedzieć aby `Java` była językiem, w którym programy tworzy się najszybciej – jest tak chociażby dlatego, że składnia `C++` jest bardziej zwięzła nawet na poziomie długości nazw słów kluczowych elementów biblioteki standardowej. Także obiektowość `Javy`, jakkolwiek dobrze dostosowana do tworzenia w niej dużych projektów, nie ułatwia pisania zwięzłych programów – jedną z kluczowych pod tym względem różnic w stosunku do `C++` jest brak możliwości przeciążania operatorów. Ostateczna wada `Javy` wiąże się jednak z czasami wykonania programów interpretowanych w ramach maszyny wirtualnej – jest to wada na tyle poważna, że w niektórych edycjach Potyczek Algorytmicznych limit czasu dla `Javy` był dwu- lub trzykrotnie większy, niż w przypadku języków kompilowanych. W obliczu tych spostrzeżeń nie sposób proponować `Javę` jako język, którego poznanie byłoby przydatne akurat w kontekście zawodów w stylu ACM-ICPC.

2.1.4 Języki skryptowe

Oddzielne rozważania można poświęcić zawodom w stylu Google Code Jam lub eliminacji do Deadline24/Marathon24. Są to zawody zdalne gdzie programy w czasie zawodów nie są kompilowane ani uruchamiane przez organizatorów, lecz to uczestnik ma pewien niedługi czas od pobrania zestawu danych do przesłania odpowiedzi. Wiąże się z tym limit czasu wynoszący po kilka minut, zamiast kilku sekund typowych dla ACM-ICPC. Oznacza to, że kosztem dłuższego czasu wykonania programów można wybrać język, w którym napisanie samego rozwiązania może być przebiegać szybciej (także ze względu na zwięzłość składni) niż w przypadku `C++`. Takie zalety ma wiele języków skryptowych – wielu programistów jako przykład języka mającego taką zaletę wskazałoby `Pythona`. Można przypuszczać, że w zależności do preferencji programisty nie gorzej sprawdziłyby się `Perl` i `Ruby`. Zależnie od zadania nawet wtedy zdarza się jednak, że `C++` jest wyborem optymalnym. Dlatego wybór pierwszego języka, który pozwoli na sprawdzanie i rozwijanie swoich zdolności algorytmicznych w ramach możliwie wielu zawodów zdaje się być jednoznaczny – i jest to właśnie `C++`. Poznanie innych języków może czasem pomóc, ale jest to sprawa drugorzędna którą można uzupełnić dopiero na dalszym etapie nauki.

2.1.5 Wersje `C++`

Osobne rozważania można poświęcić stosowanej wersji języka `C++`. Porównując języki pod względem szybkości wykonania pewnych instrukcji (ze względu mechanizmy pozwalające unikać kopiowania danych w pamięci) oraz zwięzłości kodu (więcej gotowych elementów ułatwiających abstrakcyjne programowanie obiektowe) można by dojść do wniosku, że nowoczesne wersje standardu języka (począwszy od `C++11`, wraz z bieżącym `C++14`) są jednoznacznie lepsze. Problem dostępności tych wersji języka w systemach testujących rozwiązania na różnych zawodach jest aktywnie rozwiązywany, lecz wciąż można go napotkać. `C++11` jest jednak językiem bardzo szerokim, którego poprawne używanie wymaga posiadania znacznie większej wiedzy, niż w przypadku standardów języka do `C++03` włącznie. Korzyści związane z `C++11` istnieją, ale nie są aż tak istotne, aby poświęcać samemu językowi programowania aż tyle uwagi od samego początku nauki programowania – jest to coś, co można uzupełni później. Kolejność zdobywania wiedzy zakładająca zaczęcie od `C++03` pozwala lepiej zrozumieć także `C++11`, gdzie występuje wiele skróconych zapisów, które można by stosować bez świadomości, co kryje się pod spodem. Taki brak zrozumienia praktycznie uniemożliwiłby szacowanie złożoności obliczeniowej w oparciu o kod źródłowy programu. Dlatego zalecanym w tym skrypcie językiem programowania jest `C++03`, zdefiniowany w standardzie `ISO/IEC 14882:2003`. Wspominane będą także rozszerzenia specyficzna dla kom-

pilatora z kolekcji GNU (ang. *GNU Compiler Collection*).

Warto jednak zauważyć, że rozumowanie bardzo podobne do tego, które zdecydowało o wskazaniu `C++03` zamiast nowszych wersji języka mogłoby równie dobrze dotyczyć wyboru `C` zamiast `C++` (i kilka lat temu często dokonywano takiego właśnie wyboru). Rozpoczęcie nauki od `C` jest ponadczasowo dobrym pomysłem dla programisty szukającego bardzo głębokiego spojrzenia na programowanie w wielu różnych zastosowaniach. W przypadku samych zadań algorytmicznych byłoby to jednak przesadne, gdyż rozwiązanie wielu z nich wymagałoby samodzielnego pisania w `C` tych samych klasycznych elementów, które w `C++` stanowią element biblioteki standardowej.

2.2 Podstawowe narzędzia

2.2.1 Kompilator

Dla uproszczenia przyjmiemy, że zadanie polegające na tym, aby na podstawie pliku z kodem źródłowym (w języku `C++`) stworzyć plik wykonywalny stanowi kompilację. W istocie istotnym elementem tworzenia plików wykonywalnych jest także linkowanie (budowanie powiązań z bibliotekami, których funkcje zostały wywołane w ramach skompilowanego obiektu) – jednak ze względu na brak wykorzystania jakichkolwiek nagłówków i bibliotek spoza biblioteki standardowej języka `C++`, przebieg tego etapu nie podlega dodatkowej parametryzacji – w istocie myślenie o nim nie jest przy naszych założeniach istotne.

Przykładowym zestawem narzędzi pozwalający na kompilację programów (miedzy innymi w języku `C++`), który jest stosowany podczas wielu zawodów i dostępny w wielu serwisach typu online-judge jest zestaw narzędzi GNU (rozwijany na licencjach GNU GPL, w ramach działania fundacji GNU). Do tego zestawu narzędzi zalicza się kolekcja kompilatorów GNU (GCC - ang. *GNU Compiler Collection*), wraz z kompilatorem języka `C++` dostępnym poprzez polecenie `g++`. W wielu dystrybucjach GNU/Linuksa aby z niego korzystać trzeba po prostu zainstalować pakiet o nazwie `g++`.

Kod źródłowy w języku `C++` jest po prostu plikiem tekstowym. Tradycyjnie przyjmuje się, że nazwa pliku powinna posiadać jeden z sufiksów: `.cc`, `.cpp`, `.cp`, `.cxx` lub `.c++`. Przyjmijmy, że plik `hello.cc` znajdujący się w bieżącym katalogu ma następującą treść:

Listing 2.1: Najprostszy program – `hello.cc`.

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5
```

```
6 int main() {
7   cout << "Hello world!\n";
8   return 0;
9 }
```

Najprostsze wywołanie kompilatora miałoby postać:

```
$ g++ hello.cc
```

Wykonanie takiego polecenia spowodowałoby powstanie pliku wykonywalnego o standardowej nazwie `a.out`. W tej chwili można na przykład odczytać informacje o tym pliku a także uruchomić go.

```
$ file a.out
a.out: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=53d645b85db56fb7f1f4e2ec78c6dc9b9b260b80, not stripped
$ ./a.out
Hello world!
```

Aby zamiast zawsze kompilować program do standardowej nazwy `a.out` (lub, w przypadku systemów Microsoft Windows – `a.exe`) podać własną nazwę, należy zastosować parametr `-o` (wielkość liter ma znaczenie) podając po nim oczekiwaną nazwę pliku binarnego. Na przykład:

```
$ g++ hello.cc -o hello
```

Skompiluje program do pliku binarnego o nazwie `hello`.
Inne opcje kompilatora warte uwagi w kontekście niniejszego skryptu to:

- `-std=` – wersja standardu języka. Adekwatnie do wyjaśnień z sekcji 2.1, interesującymi nas wariantami są przede wszystkim `-std=c++03` lub, jeśli wiemy co robimy i mamy pewność że nowszy standard jest obsługiwany tam, gdzie wysyłamy rozwiązania: `-std=c++14`.

- `-O` – poziom/rodzaj optymalizacji. Na przykład `-O0` oznacza brak optymalizacji, co sprawia że kompilacja trwa bardzo krótko, lecz wynikowy program może działać wolno. Z kolei `-O3` sprawia, że nawet programy napisane niechlujnie (nadmiar rekurencji, redundantne kopiowanie pamięci) mogą działać szybko – lecz w przypadku `-O3` nawet drobne niedoskonałości kodu mogą prowadzić do poważnych błędów w działaniu programu. Na zawodach programistycznych najczęściej stosowane jest `-O2` i taką opcję warto stosować także podczas lokalnych testów.

  Oddzielnym przypadkiem jest kompilowanie programu nie po to, aby badać jego szybkość działania, lecz aby szukać błędów występujących podczas jego przebiegu (czyli stosować *debugger*, na przykład `gdb` opisany w sekcji 2.4.2). Wtedy właściwą opcją jest `-Og`.

- `-g` – opcja, którą należy włączyć jeżeli kompilujemy program na potrzeby *debuggera*. W przypadku systemów GNU/Linux i debuggera `gdb` stosowny zestaw pokrewnych opcji to `-gdwarf-4 -g3 -ggdb3 -Og` – wtedy w plik wykonywalny zostanie wbudowanych dużo dodatkowych informacji, które mogą ułatwić szukanie przyczyn błędów występujących podczas wykonywania programu.

- `-static` – powoduje, że wszystkie wykorzystywane biblioteki zostają wbudowane w plik wykonywalny programu. W rezultacie taki plik jest większy (zajmuje kilka dodatkowych megabajtów), ale dzięki temu pomiar czasu działania programu staje się bardziej deterministyczny,gdyż wywoływanie funkcji bibliotecznych nigdy nie wymaga od systemu operacyjnego czytania oddzielnych plików z bibliotekami współdzielonymi.

  Opcji `-static` nie należy jednak stosować podczas szukania błędów podczas wykonania programu. Nie chcemy, aby informacje o potencjalnych zagrożeniach wynikających z badanego kodu zostały zgubione między znaczną liczbą ostrzeżeń dotyczących kodu biblioteki standardowej (jakkolwiek została ona dobrze przetestowana i raczej nie zawiera błędów, to jej kod może generować ostrzeżenia).

- `-m32` – jeśli używamy komputera o architekturze x86-64 (najpopularniejszej 64-bitowej) a chcemy zbudować program pod 32-bitową architekturę x86, to możemy to osiągnąć dzięki tej opcji. Wymaga to zainstalowania dostosowanej do tego zadania wersji kompilatora – w systemach Ubuntu i Debian jest ona zawarta w pakiecie `g++-multilib`. Ma to sens, jeśli system testujący jest 32-bitowy i chcemy odzwierciedlić te same warunki u siebie – różnice między architekturą 64- i 32-bitową dotyczą zarówno czasów wykonania (przeważnie na korzyść architektury 64-bitowej) jak i zużycia pamięci (na korzyść architektury 32-bitowej, ale zazwyczaj nieznacznie – jest tak dlatego, że rodzaj architektury dotyczy dokładnie tego, ile bitów pamięci zajmuje pojedyncza zmienna typu wskaźnikowego).

- `-march=`, `-mtune=` – opcje dotyczące dostosowania programu pod konkretny rodzaj procesorów. Jeżeli chcemy osiągnąć jak najlepszy czas na bieżącym komputerze, wystarczającym pomysłem zapewne jest użycie opcji `-march=native`. Aby uzyskać program dostosowany pod wiele różnych procesorów, gotowy do rozprzestrzenienia na inne komputery (lub: dokonać pomiaru czasu działania programu, który będzie nieco mniej zaburzony przez specyficzne parametry bieżącego sprzętu) można użyć opcji `-march=x86-64 -mtune=generic` (lub, w przypadku kompilacji dla architektury 32-bitowej, `-march=i686 -mtune=generic`).

- `-W` – opcje, które decydują o tym jakie ostrzeżenia na temat domniemanej poprawności kodu programu zostaną wypisane. Ostrzeżenia są dobre – często ich lektura pozwala wyeliminować błędy, których wykrycie na podstawie samej obserwacji działania programu byłoby bardzo trudne. Dlatego włączenie szerokiej gamy ostrzeżeń o niezgodności ze standardem języka, zapisach potencjalnie niebezpiecznych i zapisach powszechnie uznanych za nieczytelne jest wskazane – można to osiągnąć za pomocą opcji `-W -Wall -Wextra -Wpedantic`.

  Część ostrzeżeń charakteryzuje się jednak tym, że w przypadku zadań w stylu ACM-ICPC są one redundantne. Za pomocą odpowiednich opcji można je jednak pojedynczo powyłączać. Na przykład `-Wno-sign-compare` pozwala uniknąć ostrzeżenia o porównaniu zmiennej całkowitoliczbowej ze znakiem ze zmienną bez znaku (dotyczy to także porównywania zmiennych typu `int` z wartościami zwracanymi przez metodę `.size()` standardowych kontenerów). Natomiast `-Wno-char-subscripts` pozwala swobodnie używać typu `char` jako indeksu tablicy (zabieg ten częściej stosuje się celowo, niż przypadkiem).

- `-D` – opcje powodujące, że zdefiniowane zostają dodatkowe dyrektywy preprocesora. Opcja `-DARBITRARY_NAME` sprawia, że program jest kompilowany tak, jakby przed jego kodem dodatkowo pojawiła się linia treści `#define ARBITRARY_NAME`. Z kolei opcje typu `-DANOTHER_NAME=VALUE` działają tak, jakby na początku kodu dodano `#define ANOTHER_NAME VALUE`.

Podsumowując powyższe, kompilacja programu z myślą o szukaniu błędów w trakcie jego wykonania może przebiegać następująco:

```
$ g++ -std=c++03 -gdwarf-4 -g3 -ggdb3 -march=x86-64 -mtune=generic -pipe -W \
-Wall -Wextra -Wpedantic -Wno-sign-compare -Wno-char-subscripts -Og \
-DLOCAL_TESTS hello.cc -o hello
```

Natomiast do kompilacji wykonywanej z myślą o pomiarze czasu stosownym poleceniem byłoby:

```
$ g++ -std=c++03 -march=x86-64 -mtune=generic -pipe -W -Wall -Wextra \
-Wpedantic -Wno-sign-compare -Wno-char-subscripts -O2 -static hello.cc -o hello
```

Opcja `-DLOCAL_TESTS` może zostać zastąpiona dowolną inną nazwą, która nie wywoła konfliktów ze standardem języka `C++`. Zastosowanie takiej opcji pozwoli na pisanie kodu, który się będzie zachowywał podczas lokalnych testów inaczej, niż po wysłaniu do systemu testującego. Istnieją przypadki, w których takie zachowanie jest pożądane (w istocie zaproponowany zostanie szablon kodu, który będzie zakładał wykorzystanie tego mechanizmu w każdym pisanym programie).

W dalszej części skryptu omówione zostanie opakowanie wszystkich tych opcji w gotowy skrypt – pozwala to uniknąć patrzenia na te opcje na ekranie przy każdej kompilacji programu, oraz o żadnej z nich nie zapominać.

2.3 Wskazówki dla skazanych na Microsoft Windows

2.4 Narzędzia pomocnicze

2.4.1 Lokalne testowanie programu

2.4.2 `gdb`

2.4.3 `Valgrind`

2.4.4 `Code::Blocks`

Rozdział 3

Typy danych

3.1 Typy proste

3.2 Wskaźniki

3.3 Tablice

3.4 Aliasy typów

3.5 Struktury lub klasy

3.6 Szablony

3.7 Rzutowanie typów

3.8 Podsumowanie operatorów

Rozdział 4

Podstawowe elementy języka

4.1 Wejście/wyjście

4.2 Instrukcje warunkowe

4.3 Pętle

4.4 Funkcje

4.5 Rekurencja

Rozdział 5

Wybrane elementy biblioteki standardowej C++

5.1 Kontenery i iteratory

5.2 Przydatne funkcje

5.3 Przykłady

Rozdział 6

Zaawansowane elementy C++

6.1 Dziedziczenie

6.2 Przeciążanie operatorów

6.3 Spostrzeżenia na temat implementacji wejścia/wyjścia różnych typów

Bibliografia

[1] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[2] Abstruse Goose Corporation. *Abstruse Goose*. 2008–2016. URL: http://abstrusegoose.com/.

[3] Abstruse Goose Corporation. *How to Teach Yourself Programming*. 2010. URL: http://abstrusegoose.com/249.

[4] cppreference.com. C++ reference. 2014–2016. URL: http://en.cppreference.com/w/.

[5] Krzysztof Diks. *W poszukiwaniu wyzwań: wybór zadań z konkursów programistycznych Uniwersytetu Warszawskiego*. Wydział Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego, 2012.

[6] Krzysztof Diks, Tomasz Idziaszek, Jakub Łącki, Jakub Radoszewski, and Szymon Acedański. *W poszukiwaniu wyzwań 2: zadania z Akademickich Mistrzostw Polski w Programowaniu Zespołowym 2011-2014*. Wydział Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego, 2015.

[7] Tomasz Filipczak. *Grafy i ich zastosowania*. 2012.

[8] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1994.

[9] Roedy Green. *How To Write Unmaintainable Code*. 2000. URL: https://www.thc.org/root/phun/unmaintain.html.

[10] J.D. *UserFriendly*. 1997–2016. URL: http://www.userfriendly.org/.

[11] Brian W. Kernighan. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[12] Donald E. Knuth. *The Art of Computer Programming Volumes 1-3 Boxed Set*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1998.

[13] Randall Munroe. *xkcd – A webcomic of romance, sarcasm, math, and language*. 2005–2016. URL: http://xkcd.com/.

[14] Larry Nyhoff. *Programming in C++ for Engineering and Science*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2012.

[15] Stephen Prata. *C++ Primer Plus (Fifth Edition)*. 5th edition, 2004.

[16] Robert Sedgewick and Christopher J. Van Wyk. *Algorithms in C++: Fundamentals, Data Structures, Sorting, Searching*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1998.

[17] Piotr Stańczyk. *Algorytmika praktyczna*. Wydawnictwo Naukowe PWN, Warszawa, 2009.

[18] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000.

[19] Maciej M. Sysło, Krzysztof Diks, Przemysława Kanarek, and Jakub Radoszewski. *{I...XXII} Olimpiada Informatyczna* . Komitet Główny Olimpiady Informatycznej, Warszawa, 1993–2016.

[20] © cplusplus.com. C++ Language Tutorial. 2000–2016. URL: http://www.cplusplus.com/doc/tutorial/.

[21] © cplusplus.com. Standard C++ Library reference. 2000–2016. URL: http://www.cplusplus.com/reference/.

[22] Zach Weinersmith. *Saturday Morning Breakfast Cereal*. 2002–2016. URL: http://www.smbc-comics.com/.

[23] Wikibooks. C++ Programming. 2012. URL: https://en.wikibooks.org/wiki/C++_Programming.